IUT de l'Indre
Département Génie Electrique et Informatique Industrielle

Module ENSL1 : Initiation au langage VHDL

Cours

Eric PERONNIN

Chateauroux, le 4 octobre 2012

Table des matières

1 Eléments de base du langage VHDL **7**
1 Introduction . . . 7
2 Identificateurs . . . 7
3 Constantes, variables, signaux et ports . . . 7
3.1 Les constantes . . . 7
3.2 Les ports . . . 8
3.3 Les signaux . . . 8
3.4 Les variables . . . 8
4 Types de données . . . 8
4.1 Les types énumérés . . . 8
4.1.1 Type *BIT* . . . 8
4.1.2 Type *BIT_VECTOR* . . . 9
4.1.3 Types *STD_LOGIC* et *STD_LOGIC_VECTOR* . . . 9
4.1.4 Type *BOOLEAN* . . . 10
4.1.5 Types *CHARACTER* et *STRING* . . . 10
4.1.6 Type *SEVERITY_LEVEL* . . . 11
4.2 Types numériques non énumérés . . . 11
4.2.1 Types *INTEGER* et *REAL* . . . 11
4.2.2 Types *UNSIGNED* et *SIGNED* . . . 11
4.2.3 Sous-types *NATURAL* et *POSITIVE* . . . 11
4.3 Types définis par l'utilisateur . . . 11
4.3.1 Types énumérés définis par l'utilisateur . . . 11
4.3.2 Types non énumérés définis par l'utilisateur . . . 12
4.3.3 Enregistrement : *RECORD* . . . 12
4.3.4 Types vectoriels . . . 12
4.4 Conversions de types : transtypages . . . 13
4.5 Attributs . . . 13
4.5.1 Attributs associés aux types . . . 14
4.5.2 Attributs associés à un signal . . . 15
5 Opérateurs . . . 15
5.1 Opérateurs d'affectation . . . 15
5.1.1 Affectation à un signal . . . 15
5.1.2 Affectation à une variable . . . 15
5.2 Opérateurs logiques . . . 16
5.3 Opérateurs relationnels . . . 16
5.4 Opérateurs arithmétiques . . . 17

5.5 Opérateur de concaténation 17

2 Structure d'un programme VHDL **19**

3 Interfaçage avec l'extérieur : Entité **21**

1 Déclaration des entrées et des sorties 21

2 Exemples de déclarations d'entités 21

2.1 Interfaçage d'une fonction ET à 2 entrées 21

2.2 Compteur décimal 4 bits avec remise à 0 asynchrone 21

4 Section Architecture **23**

1 Rôle de la section Architecture 23

2 Traitements de flots de données 23

2.1 Constantes 23

2.2 Signaux 24

2.2.1 Définition 24

2.2.2 Déclaration 24

2.3 Exemples de programmes VHDL utilisant seulement un traitement de flots de données 24

2.3.1 Description fonctionnelle de la fonction ET2 24

2.3.2 Description en langage VHDL d'un multiplexer 4 vers 1 avec 2 entrées de sélection 24

2.3.3 Démultiplexeur 3 vers 8 (versions 1, 2 et 3) 25

3 Description comportementale 27

3.1 Utilité de la structure process 28

3.2 Détails sur l'exécution d'un processus 28

3.2.1 Activation d'un processus 28

3.2.2 Affectation de signaux dans un processus : « Scheduling » 28

3.2.3 Exemples de processus simples. Exécutions commentées 28

3.3 Variables 29

3.3.1 Dualité signal - variable 29

3.3.2 Déclaration d'une variable 29

3.3.3 Utilisation et comportement des variables 30

3.4 Structures conditionnelles 30

3.4.1 Instruction if 30

3.4.2 Exemples de programmes complets utilisant l'instruction if 30

3.4.3 Instruction case 31

3.4.4 Exemple de programme utilisant l'instruction case : démultiplexeur 3 vers 8 (version 5) 31

3.5 Structures répétitives 32

3.5.1 Instruction for 32

3.5.2 Instruction while 32

3.6 Cas d'un système rythmé par un signal d'horloge : compteur BCD 33

3.6.1 Objectif 33

3.6.2 Brève analyse 33

3.6.3 Codage 33

3.6.4 Variante avec une remise à 0 synchrone 34

4 Architecture de test 34

4.1 Objectif de l'architecture de test 34

4.2 Test bench (banc de test) 35

4.3 Création et Ecriture du test bench 35

4.4 Contrôle et report d'erreurs 38

Chapitre 1

Eléments de base du langage VHDL

1 Introduction

Lors de sa définition, les concepteurs du VHDL se sont appuyés partiellement sur un langage connu pour ses qualités dans le domaine des applications "temps réel" : ADA. Celui-ci, présentant de nombreuses similitudes avec le Pascal (au moins pour le vocabulaire de base), il n'est pas étonnant de retrouver un certain nombre de termes de ce dernier langage (cf. types, définition des tableaux, déclaration des variables, mots clés Begin et end).

2 Identificateurs

Les noms (variables, signaux, constantes et autres) choisis par le programmeur, appelés aussi identificateurs, doivent respecter les règles suivantes :

- il doivent commencer par une lettre ou un caractère _ (underscore),
- il peuvent comporter des signes _ à condition de ne jamais en inclure deux de suite,
- il peuvent aussi comporter des chiffres,
- Deux signes - (moins) placés successivement déclarent le début d'une zone de commentaires valable uniquement sur leur ligne d'apparition.

Même s'il n'y a pas de généralités, le manuel Xilinx préconise d'utiliser les règles de notations suivantes lors de l'écriture d'un programme en VHDL :

- les mots clés du VHDL doivent être écrits en minuscules,
- les identificateurs définis par l'utilisateur s'écrivent en majuscules avec des caractères _ pour séparer les mots.

3 Constantes, variables, signaux et ports

Avant d'introduire les types de données connus du VHDL, il importe de savoir quels sont les éléments auxquels seront assignés un type.

Le langage VHDL exploite quatre objets différents.

3.1 Les constantes

Elles permettent d'apporter une plus grande lisibilité du code lors des opérations d'affectation ou de comparaison.

3.2 Les ports

Les Ports sont les signaux d'entrées et de sorties de la fonction décrite : ils représentent en quelques sortes les signaux des broches du composant VHDL en cours de description.

3.3 Les signaux

En VHDL, le signal est une information qui existe physiquement dans le circuit. En quelques sortes, les ports font la liaison avec l'extérieur et les signaux sont internes à la fonction décrite. Leur utilisation est identique à celle des ports ; par contre, leur déclaration a lieu au moment de la description comportementale de la fonction.

Note : les signaux internes qui ne disparaissent pas lors de la synthèse (à cause de simplifications essentiellement) peuvent être observés lors du fonctionnement du FPGA à l'aide d'outils de debug en circuit utilisant le port JTAG de certains FPGA.

3.4 Les variables

Par opposition aux signaux, les variables ne sont pas des informations synthétisables. Elles apportent une aide précieuse dans les processus de description et permettent de décrire la nature d'évolution des signaux dans des structures de programmation séquentielles, conditionnelles ou répétitives.

Note : L'utilisation des ports, signaux et variables sera envisagée par la suite.

4 Types de données

Comme la plupart des langages structurés, le VHDL impose de typer les données à manipuler. Ce caractère incontournable justifie la présence de fonctions spécifiques permettant de réaliser des conversions de type (transtypage).

On trouve trois formes de types de données :

- les types énumérés,
- les types numériques non énumérés,
- les types utilisateurs.

Remarque importante : la suite du document présente de nombreux exemples de déclarations de variables ou signaux suivis immédiatement d'opérations d'affectations. Il est nécessaire de préciser que dans un programme complet, ces opérations (déclaration et affectation) ne peuvent être effectuées à la suite l'une de l'autre : les déclarations doivent être faites à une endroit particulier du programme ; les affectations sont quant à elles possibles à certains moments bien précis.

4.1 Les types énumérés

Les variables et signaux de types énumérés prennent un nombre fini et restreint de valeurs parfaitement connues.

4.1.1 Type *BIT*

Une grandeur de type *BIT* peut prendre deux valeurs :

- '0'
- '1'

Ces deux valeurs ne sont pourtant pas suffisantes pour satisfaire les besoins de la synthèse logique : par exemple, l'état haute impédance n'est pas envisagé ! Pour cette raison, ce type sera d'une piètre utilité en matière de synthèse.

La définition de ce type dans le paquetage VHDL standard est la suivante :

```
type BIT is ('0', '1');
```

Exemple de déclarations et d'affectations :

```
variable V : BIT;              -- Déclaration d'une variable de type BIT
V:='1';                        -- Affecte '1' à la variable V
signal S : BIT;                -- Déclare un signal de type BIT
S <= '0';                      -- Affecte '0' au signal S
S <= BIT'('1');                -- Affecte '1' au signal S en précisant
                               -- le type pour lever toute ambiguïté
```

Note : une constante de type *BIT* (plus généralement un scalaire) doit être spécifié entre deux apostrophes.

Remarque : ces premiers exemples montrent que l'opérateur d'affectation pour une variable (opérateur :=) est différent de celui utilisé pour un signal (opérateur <=) .

4.1.2 Type *BIT_ VECTOR*

Le type *BIT_ VECTOR* permet de manipuler des grandeurs résultant de l'association d'éléments de type *BIT* (i.e. des vecteurs de bits ou mot). Le VHDL permet de gérer des tableaux multi-dimensionnels et donc en particulier des vecteurs.

Note : les tableaux ne sont pas abordés dans ce cours.

La définition de ce type dans le paquetage VHDL standard est la suivante :

```
type BIT_VECTOR is array (NATURAL RANGE <>) of BIT;
```

Exemples de déclarations et d'affectations :

```
variable VECTEUR1 : BIT_VECTOR(0 to 3);     -- Déclare un mot de 4 bits
variable VECTEUR2 : BIT_VECTOR(3 downto 0); -- Déclare un mot de 4 bits
VECTEUR1 := "0001";                         -- Affecte le mot binaire 1000 à VECTEUR1
VECTEUR2 := "1000";                         -- Affecte la même valeur à VECTEUR2
VECTEUR2(3) := '0';                         -- Met à 0 le BIT de poids fort de VECTEUR2
VECTEUR1(0) := '1';                         -- Met à 1 le BIT de poids faible de VECTEUR1
```

Notes :

- Une constante de type *BIT_ VECTOR* (plus généralement vectorielle) s'écrit entre deux doubles apostrophes (guillemets informatiques).
- La variable VECTEUR1 utilise le mot clé to dans sa définition et par conséquent son BIT de poids fort (le 3) se trouve à droite. Ainsi, la première affectation met à 1 le BIT de poids fort (VECTEUR1 := "0001" ;).
- La variable VECTEUR2 utilise le mot clé downto dans sa définition et par conséquent son BIT de poids fort se trouve à gauche (écriture conventionnelle).

4.1.3 Types *STD_ LOGIC* et *STD_ LOGIC_ VECTOR*

Les types *BIT* et *BIT_ VECTOR* ne pouvant prendre que deux valeurs ('0' et '1'), la norme IEEE 1164 les substitue à deux autres types énumérés (*STD_ ULOGIC* et *STD_ ULOGIC_ VECTOR*), énumérés eux aussi, mais pouvant prendre 9 valeurs différentes :

- 'U' : non défini (non initialisé)
- 'X' : inconnu forcé
- 'O' : zéro forcé
- '1' : un forcé
- 'Z' : haute impédance
- 'W' : inconnu faible (Weak)

- ’L’ : zéro faible
- ’H’ : un faible
- ’-’ : indifférent (don’t care)

La spécification d’une valeur *STD_ULOGIC* s’effectue toujours entre deux ’ (Exemple : A<=’1’). Le type *STD_LOGIC_VECTOR* étant la réunion de plusieurs objets *STD_ULOGIC*, sa spécification utilise un formalisme différent et emploie pour cela deux " (Exemple : A<="001001").

De la même façon, les types *STD_LOGIC* et *STD_LOGIC_VECTOR* ont été définis. Ils correspondent en fait à deux sous-types dont la définition complète dépend du logiciel de simulation et de synthèse utilisé. Comme ce sont des sous-types, ils ne prennent en compte qu’une partie de l’ensemble des valeurs possibles offertes par les types *STD_ULOGIC* et *STD_ULOGIC_VECTOR* (de nombreux simulateurs ne savent pas gérer les valeurs ’H’, ’W’ ou ’L’ du type *STD_ULOGIC*, ils utilisent plutôt le type *STD_LOGIC* défini sans ces mêmes valeurs).

Remarque importante : puisqu’ils correspondent aux dernières avancées de la norme VHDL, les types *STD_LOGIC* et *STD_LOGIC_VECTOR* sont à préférer aux types *BIT* et *BIT_VECTOR*. Cette remarque est d’autant plus vraie que ***seuls ces types sont bien adaptés à la simulation et à la synthèse.***

L’utilisation de ces types est par ailleurs identiques à celle de leurs homologues que sont *BIT* et *BIT_VECTOR*.

La définition de ces types dans le paquetage VHDL IEEE 1164 est la suivante :

```
type STD_ULOGIC is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
type STD_ULOGIC_VECTOR is (NATURAL RANGE <>) of STD_ULOGIC;
```

4.1.4 Type *BOOLEAN*

Une grandeur de type *BOOLEAN* ne peut prendre que deux valeurs différentes :

- TRUE (vrai)
- FALSE (faux)

Même si ce type est simplement binaire, il ne faut pas pour autant le confondre avec le type *BIT* vu précédemment. D’une part, ce type est le type de référence des instructions conditionnelles (voir suite) ; d’autre part, les mots FALSE et TRUE apportent une lisibilité au code que ne fournit pas le type *BIT* lors d’opérations d’affectations ou de comparaisons (FALSE est plus clair que ‘0’ ; idem pour TRUE vis à vis de ‘1’).

La définition de ce type dans le paquetage VHDL standard est la suivante :

```
type BOOLEAN is (FALSE, TRUE);
```

Exemples de déclarations et d’affectations :

```
variable FINI : BOOLEAN := TRUE;   -- Déclare une variable de type BOOLEAN et lui affecte TRUE
                                   -- comme valeur initiale
FINI := FALSE ;                    -- Affecte FALSE à la variable FINI
constant VRAI : BOOLEAN := TRUE ;  -- Déclare une constante de type BOOLEAN
```

4.1.5 Types *CHARACTER* et *STRING*

Les variables de type *CHARACTER* prennent 95 valeurs différentes correspondant à une partie du jeux de caractères ASCII (tous les caractères sans les codes de contrôle ; voir documentation du compilateur VHDL pour de plus amples informations).

Le type *STRING* résulte de l’association de plusieurs éléments de type *CHARACTER*.

La définition de ces types dans le paquetage VHDL standard est la suivante :

```
type CHARACTER is ('A', 'B', 'C',. . . , 'a', 'b',. . . , '0', '1', '2',. . . );
type STRING is (POSITIVE RANGE <>) of CHARACTER;
```

4.1.6 Type *SEVERITY_LEVEL*

Ce type est employé avec une fonction spécifique aux programmes de tests et permet de caractériser le niveau de gravité programmable lors d'incidents de simulation. Les variables de ce type peuvent prendre les valeurs suivantes : *NOTE, WARNING, ERROR, FAILURE*.

4.2 Types numériques non énumérés

4.2.1 Types *INTEGER* et *REAL*

VHDL propose aussi des types non énumérés. Ces types sont définis par l'intervalle des valeurs que peuvent prendre leurs instances.

- type *INTEGER* : valeurs signées codées sur 32 bits.
- type *REAL* : l'étendue des valeurs dépend du compilateur VHDL et la norme VHDL impose des valeurs aux puissances de 10 au moins comprises entre -38 et +38.

Un type Time existe aussi.

Note : peu utiliser pour la synthèse, ces types ne justifient pas davantage d'information dans ce manuel.

4.2.2 Types *UNSIGNED* et *SIGNED*

Le paquetage NUMERIC_STD apporte de nombreuses possibilités de conversions entre types ainsi que deux types permettant une manipulation aisée de données entières signées ou non.

Les types *UNSIGNED* et *SIGNED* utilisent pour base des vecteurs de bits. L'éventail des valeurs que peut couvrir une variable ou un signal de type *UNSIGNED* ou *SIGNED* dépendra du nombre de bits qui lui sera attribué lors de la déclaration.

Exemples de déclarations et d'affectations :

```
variable NON_SIGNE : UNSIGNED(0 to 3);        -- Déclare une variable non signée
                                              --codé sur 4 bits poids fort à droite
signal NON_SIGNE : UNSIGNED(7 downto 0);      -- Déclare un signal non signé sur
                                              -- 8 bits poids fort à gauche
```

4.2.3 Sous-types *NATURAL* et *POSITIVE*

Dans la paquetage VHDL standard, il existe deux sous-types définis comme suit :

```
subtype NATURAL is INTEGER RANGE 0 to INTEGER'high;
subtype POSITIVE is INTEGER RANGE 1 to INTEGER'high;
```

Le sous-type *NATURAL* correspond en fait à une restriction du type *INTEGER* dans on conserve toutes les valeurs supérieures ou égales à 0. De la même façon, le sous-type *POSITIVE* est une restriction du type *INTEGER* dont seules les valeurs supérieures à 0 sont conservées.

Note : Les éléments *RANGE* et *INTEGER'high* seront explicités un peu plus loin.

4.3 Types définis par l'utilisateur

4.3.1 Types énumérés définis par l'utilisateur

Il est possible de créer de nouveaux types de données. Une telle opération permettra de rendre le code source encore plus lisible. Par exemple, une machine à états pourra être l'occasion de créer un type utilisateur énuméré listant les différents états de la machine :

```
type TYPE_ETAT is (INITIALISATION, ARRET_URGENCE, ATTENTE, ETAT1, ETAT2, ETAT3);
```

4.3.2 Types non énumérés définis par l'utilisateur

Pour la création de type non énuméré, le VHDL dispose du mot clé RANGE faisant office de directive et permettant de spécifier l'intervalle des valeurs valides pour le type créé (pour la simulation ou la synthèse). En plus de fournir des informations de contrôle au simulateur, cette directive permet aussi au simulateur et au synthétiseur d'affecter le nombre de bits adéquat pour le codage de l'instance du type créé.

Exemple de déclaration :

```
type TYPE_BYTE is INTEGER RANGE 0 to 255 ;
```

Dans cet exemple, le type créé s'appelle TYPE_BYTE. Il apparaît comme un sous-type du type *INTEGER* dont l'intervalle de valeurs, fixé par la directive *RANGE*, est 0 à 255.

Note : de fait, ce nouveau type occupera 8 bits lors de la synthèse.

Remarque : en l'absence de directive RANGE, le nombre de bits alloués est fixé à 32.

4.3.3 Enregistrement : *RECORD*

Lorsque des informations de toute nature doivent être réunies, VHDL permet la création d'un type spécifique qui autorisera la manipulation de toutes ces données sous un même identificateur, appelé enregistrement. On utilise alors le mot clé record.

Exemple de déclaration :

```
type TYPE_TABLE_VERITE is record
  ENTREE1 : STD_LOGIC_VECTOR(7 downto 0) ;
  ENTREE2 : STD_LOGIC_VECTOR(7 downto 0) ;
  SORTIE : STD_LOGIC_VECTOR(7 downto 0) ;
  CARRY_OUT : STD_LOGIC ;
end record ;
```

Cet exemple crée un enregistrement dont l'instance peut stocker un vecteur de table de vérité.

Exemple d'utilisation :

```
variable VECTEUR_TABLE : TYPE_TABLE_VERITE ;     -- déclare une instance
signal CARRY : STD_LOGIC ;                       -- déclare un signal
CARRY <= VECTEUR_TABLE.CARRY_OUT;                -- extrait juste le champ CARRY_OUT
                                                 -- de l'enregistrement.
```

Comme le montre cet exemple, chaque champ de l'enregistrement peut être manipulé comme une simple variable (ou signal) en utilisant l'opérateur . (point). Dans l'exemple précédent, VECTEUR_TABLE.ENTREE1 est un vecteur de 8 *STD_LOGIC*.

Note : les types *record* sont très utiles pour créer des tableaux assurant le stockage de tables de vérité ou de vecteurs de test.

4.3.4 Types vectoriels

Les types utilisateurs qui viennent d'être décrits sont des scalaires. On peut aussi créer des types vectoriels dont on fixe le nombre d'éléments dans la déclaration du type ou lors de la déclaration de l'instance. Ces types reposent sur le mot clé *array*.

- Type vectoriel avec taille libre

Exemple :

```
type VECTEUR_BOOLEAN is array (NATURAL RANGE <>) of BOOLEAN;
```

Cet exemple crée un type de vecteur composé d'éléments *BOOLEAN* dont le nombre d'éléments doit être fixé lors de l'instanciation (instanciation = déclaration d'une variable répondant au type) à l'aide des mots clés *to* et *downto*.

Note : dans l'exemple, "*NATURAL RANGE* <>" signifie que l'indice peut prendre toutes les valeurs couvertes par le type *NATURAL* (i.e. 0 à la valeur maximale d'un *INTEGER*).

Instanciation :

```
variable VECTEUR1 : VECTEUR_BOOLEAN (15 downto 0);
```

VECTEUR1 est alors un tableau composé de 16 éléments *BOOLEAN* dont les indices couvrent l'intervalle [0 ; 15].

- Type vectoriel avec taille fixée

On peut aussi directement imposer la taille du tableau lors de la déclaration du type.

Exemple :

```
type BYTE is array (7 downto 0) of BIT ;     -- Créer un type pour stocker un octet
variable A : BYTE ;                          -- Déclare A de type BYTE ;
```

- Type matriciel

Pour d'autres besoins (calcul matriciel, stockage d'une table de vérité ...), on peut définir des tableaux multi-dimensionnels.

Exemple :

```
type TYPE_MATRICE is array(1 to 20, 1 to 30) of INTEGER ; -- Déclaration du type
variable A : TYPE_MATRICE ;                               -- Instanciation
A(6,15) <= 12 ;                                           -- Affectation
```

Cet exemple crée un type pour manipuler des matrices d'entiers contenant 20 lignes et 30 colonnes ; crée une instance, A, de ce type et lui affecte 12 à la ligne 6, colonne 15.

Remarque : le paragraphe montre comment utiliser un tableau de vecteurs pour réaliser la synthèse d'un système combinatoire à partie de sa table de vérité.

4.4 Conversions de types : transtypages

Le contrôle rigoureux des types oblige à de nombreuses conversions pour passer d'un type à un autre. Dans le paquetage standard, l'addition n'est pas définie pour le type *STD_LOGIC_VECTOR*. Il existe alors deux possibilités pour parvenir malgré tout à ses fins :

- surcharger l'opérateur + pour que le compilateur sache comment additionner une variable de type *STD_LOGIC_VECTOR* avec une constante de type *INTEGER* : solution intéressante mais faisant appel à des notions de programmation avancées en VHDL ;
- opérer un certain nombre de conversions afin de conserver la cohérence des types,
- trouver la bonne bibliothèque proposant l'opération (par une des méthodes suscitée dans les deux points précédents).

Ainsi, on pourra procéder par une double conversion :

B <= STD_LOGIC_VECTOR (UNSIGNED (B) + "3") ;

UNSIGNED réalise la conversion de B en *UNSIGNED* et l'addition avec "3", de type *UNSIGNED*, est possible. Le résultat obtenu étant de type *UNSIGNED*, il faut le convertir en type *STD_LOGIC_VECTOR* avec la fonction du même nom (écrite en minuscules).

Plus simplement, on pourra faire appel à une bibliothèque (un paquetage) rendant l'opération possible.

4.5 Attributs

Un attribut est une caractéristique associée à un type ou à un signal. L'ensemble des attributs prédéfinis peut être élargi par des attributs définis par le programmeur.

Utilisation : on spécifie le nom du type ou du signal suivi d'une apostrophe puis du nom de l'attribut affecté d'éventuelles grandeurs d'entrées.

4.5.1 Attributs associés aux types

Pour illustrer cette classe d'attributs, les types énumérés suivants seront utilisés :

```
type BUS_ARBITRER is ( WAITING, ACK1, ACK2, ACK3);
type TAB_ENTIER is array (10 to 2) of INTEGER;
```

Remarque : on définit la position d'une valeur valide pour un type par la valeur 0 pour la grandeur définie à gauche. La seconde valeur possède la position 1 . . .

- Cas des types et sous-types énumérés

Attribut	Définition	Exemple	Résultat
pos(x)	Renvoie la position de x dans la liste des valeurs énumérés pour le type	BUS_ARBITRER'pos(ACK1)	1
val(x)	Renvoie la valeur du type à sa position x	BUS_ARBITRER'val(1)	ACK1
pred(x)	Renvoie la valeur de l'élément placé avant x	BUS_ARBITRER'pred(ACK1)	WAITING
succ(x)	Renvoie la valeur de l'élément placé après x	BUS_ARBITRER'succ(ACK2)	ACK2

- Cas de tous les types et sous-types scalaires

Attribut	Définition	Exemple	Résultat
left	Renvoie la valeur minimale ou limite gauche dans la déclaration du type	BUS_ARBITRER'left	WAITING
right	Idem left pour la définition à droite	BUS_ARBITRER'right	ACK3
high	Renvoie la valeur la plus grande dans la définition du type (type numérique)	BUS_ARBITRER'high	$2^{31}-1$
low	Idem pour la valeur la plus petite	BUS_ARBITRER'low	-2^{31}

Il existe des attributs supplémentaires permettant de retrouver les caractéristiques d'un type vectoriel (ou d'un tableau de dimension x d'une façon générale). Pour les tableaux, les attributs left, right, high et low existent aussi mais renvoie un indice (position dans le tableau).

Attribut	Définition	Exemple	Résultat
left(x)	Renvoie la valeur la plus à gauche du premier indice pour la dimension x	TAB_ENTIER'left(1)	10
right(x)	Idem left pour la définition à droite	TAB_ENTIER'right(1)	2
high(x)	Renvoie la valeur la plus grande de l'intervalle des valeurs du tableau, pour la dimension x	TAB_ENTIER'high(1)	10
low(x)	Idem pour la valeur la plus petite	TAB_ENTIER'low(1)	2
RANGE(x)	Renvoie l'intervalle de variation de gauche à droite des indices du tableau pour sa dimension x	TAB_ENTIER'RANGE	10 downto 2
reverse_RANGE(x)	Idem de droite à gauche	TAB_ENTIER'reverse_RANGE	2 to 10

Note : ce dernier tableau utilise pour les exemples le tableau TAB_ENTIER de dimension 1 seulement ce qui explique que la valeur donnée à x est toujours 1.

4.5.2 Attributs associés à un signal

Les attributs associés à un signal permettent de mieux connaître la façon dont évolue ce signal (exemple : détection de front montant ou descendant, changement d'état ...). Afin de comprendre le fonctionnement de ces attributs, il est nécessaire de préciser ce qu'est une Transaction.

Définition : une transaction sur un signal est une opération modifiant ou affirmant la valeur de ce signal pendant le cycle de simulation en cours.

Attribut	Définition	Exemple
event	Fonction booléenne qui retourne TRUE si le signal change d'état pendant le cycle de simulation en cours	CLK'event
active	Fonction booléenne retournant TRUE si le signal est actif	RESET'active
quiet(T)	Signal booléen VRAI si, pendant la durée T, le signal n'a fait l'objet d'aucune transaction (T=0 s'il n'est pas précisé)	SORTIE'quiet(10 ns)
stable(T)	Signal booléen VRAI si, pendant la durée T, le signal n'a pas fait l'objet de changement d'état (T=0 s'il nest pas précisé)	ENTREE'stable(10 us)
transaction	Signal de type BIT qui change d'état à chaque transaction du signal testé	ENTREE'transaction

Remarque : la bibliothèque VHDL standard Std_Numeric_1164 définit deux fonctions permettant de détecter des fronts montants (fonction rising_edge) et descendants (fonction falling_edge) de signaux (très utilisés pour la synthèse de systèmes synchrones). Ces deux fonctions doivent leur existences à celle des attributs qui autorisent la détection de changement d'état de signaux.

Soit CLK un signal de type *STD_LOGIC* reflétant un signal d'entrée appelé "horloge".

- Si l'équation booléenne CLK'event and CLK = '1' renvoie la valeur TRUE alors le signal CLK vient de présenter un front montant.
- Si CLK'event and CLK = '0' renvoie TRUE alors de signal CLK vient de présenter un front descendant.

5 Opérateurs

5.1 Opérateurs d'affectation

Le VHDL utilise deux opérateurs d'affectations.

5.1.1 Affectation à un signal

Pour réaliser une affectation à un signal, on utilise l'opérateur <=

Exemples :

```
signal A : STD_LOGIC_VECTOR (3 downto 0);     -- Déclaration du signal
signal B, C : STD_LOGIC_VECTOR (3 downto 0);  -- Déclaration multiple de signaux
B <= "0100";                                  -- Affectation d'une constante
C <= "1000";                                  -- Idem
A <= B and C;                                 -- Affectation à partir d'une expression
```

5.1.2 Affectation à une variable

L'affectation à une variable exploite l'opérateur :=

Exemples :

```
variable A : INTEGER;       -- Déclaration de variable
variable B, C : INTEGER;    -- Déclaration multiple de variables
B := 12;                    -- Affectation d'une constante
C := 8;                     -- Idem
A := B + C;                 -- Affectation à partir d'une expression
```

5.2 Opérateurs logiques

Ils s'appliquent à des signaux ou des variables de types booléens, bits et dérivés : *BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, BOOLEAN.*

En voici les principaux : *not*, *and*, *or*, *nand*, *nor*, *xor*, *nxor* (en VHDL 93 seulement).

Les règles de priorité sont celles de la logique combinatoire traditionnelle.

Notes :

- Dans le cas d'opérandes de types *STD_LOGIC_VECTOR* ou *BIT_VECTOR*, l'opération logique s'effectue BIT à BIT (cela implique des opérandes de tailles identiques pour les opérateurs binaires).
- Le type du résultat est le même que celui de (ou des) opérande(s).

Exemples d'utilisation :

```
variable V1, V2, V3 : STD_LOGIC;                  -- Déclaration de 3 variables
V1 := '0';                                        -- Mise à 0 de V1
V2 := '1';                                        -- Mise à 1 de V2
V3 := (V1 or not V2) and V2;                      -- Affectation utilisant une expression
                                                  -- à base d'opérateurs logiques
signal A, B, C : STD_LOGIC_VECTOR (3 downto 0);   -- Déclaration de 3 vecteurs de 4
                                                  -- éléments STD_LOGIC
B <= "0101";                                      -- Affectation d'un mot
C <= "1100";                                      -- Affectation d'un mot
A <= B and C;                                     -- Et logique BIT à BIT entre B et C
```

Résultats des exemples :

```
V3 = '0'
A = "0100"
```

5.3 Opérateurs relationnels

Ils sont valables avec des signaux et des variables scalaires et vectoriels.

Voici la liste de ces opérateurs : =, /= (différent), <, <=, >, <, >=

Généralement ces différents opérateurs doivent être employés avec des opérandes de mêmes types. Cependant, il est possible de trouver une définition permettant de comparer des grandeurs de types différents entre elles : cela suppose une surcharge des opérateurs relationnels (i.e. l'écriture d'une fonction qui réalise la comparaison pour un opérateur donné avec deux opérandes de types donnés).

Le résultat d'une comparaison a un type : c'est le type BOOLEAN. Ainsi, on parle de comparaison vrai (TRUE) ou fausse (FALSE).

Note : la différence entre l'opérateur relationnel <= et l'opérateur d'affectation <= est effectuée par le compilateur en fonction de son contexte d'utilisation.

Exemples :

```
variable FIN : BOOLEAN;     -- Déclaration d'une variable booléenne
signal A, B : STD_LOGIC;    -- Déclaration de deux signaux de
                            -- type binaire étendu
A <= '1';                   -- Affectation d'une constante
B <= '0';                   -- Idem
FIN := ( A /= B);           -- Test si A est différent de B
                            -- et affecte le résultat à FIN
```

Résultat de l'exemple :

```
FIN = TRUE car A est différent de B
```

5.4 Opérateurs arithmétiques

Ce sont +, - et *abs* pour les types *BIT* et dérivés et, *, /, *mod*, *rem* (reste de division entière), sur les types *INTEGER*, *REAL* et leurs dérivés.

Note : l'utilisation des opérateurs arithmétiques avec les types *STD_LOGIC* ou *STD_LOGIC_VECTOR* est possible à condition d'inclure les paquetages IEEE qui réalisent la surcharge de ces opérateurs (en début de programme ajouter les lignes : use ieee.logic_arith.all ; use ieee.logic_UNSIGNED.all ;).

Exemple d'utilisation des opérateurs arithmétiques :

```
variable VA, VB, VC : INTEGER; -- Déclare 3 variables entières
VA := 3;                       -- Affecte une constante
VB := 4;                       -- Idem
VC := VA * VB + VA - VC;       -- Affecte VC par un calcul
```

Résultat :

```
VC = 11
```

5.5 Opérateur de concaténation

L'opérateur de concaténation & permet de manipuler des tableaux de tout type et de différentes tailles pour en réaliser l'association sous un autre label de dimension plus importante.

Exemples d'utilisation de l'opérateur de concaténation :

```
signal A, B : STD_LOGIC_VECTOR (3 downto 0); -- Déclare deux mots
                                             -- de 4 bits
signal C : STD_LOGIC_VECTOR (7 downto 0);    -- Déclare un mot de 8 bits
signal D : STD_LOGIC_VECTOR (2 downto 0);
A <= "1100";                                 -- Affecte une constante
B <= "0011";                                 -- Idem
D <= "101" ;                                 -- Idem
C <= A & B;                                  -- Affecte la concaténation de A et B
A <= '1' & D;                                -- Affecte la concaténation
                                             -- d'une constante et D à A
```

Résultats des exemples :

```
C = "11000011" A = "1101"
```

Chapitre 2

Structure d'un programme VHDL

Un programme VHDL écrit pour caractériser un bloc hiérarchique par un système combinatoire simple ou un modèle comportemental aura la structure générale suivante :

- Déclaration des bibliothèques utilisées
- Interfaçage de la fonction décrite avec l'extérieur (Entité : mot clé entity)
- Description de la fonction à l'aide d'équations combinatoires ou/et de processus (Architecture : mot clé Architecture + éventuellement process).

La page suivante propose un modèle de programme reflétant une structure plus détaillée.

Remarques :

- Dans la structure qui suit, les identificateurs en majuscules sont choisis par le programmeur.
- Lors de l'écriture de code en VHDL, on utilise les mêmes règles d'indentation qu'en C.
- Une ligne commençant par deux caractères – successifs est une ligne de commentaires.
- Une ligne présentant une séquence de points signifie que des lignes analogues à la précédentes peuvent apparaître successivement (cf. déclaration de plusieurs variables . . .). Les indices ne, ns, v, s sont des indices.

```
-- Déclaration d'une bibliothèque (l'opération complète peut
-- être répétée pour d'autres bibliothèques)
library Nom_de_bibliothèque;
use Nom_de_bibliothèque.Nom_du_paquetage.Fonction_utilisée;
-- Description de la vue externe de la fonction à décrire
entity NOM_DE_LA_FONCTION_DECRITE is port (
  NOM_ENTREE_1    : in TYPE_ENTREE_1;
   ...
  NOM_ENTREE_ne   : in TYPE_ENTREE_ne;
  NOM_SORTIE_1    : out TYPE_SORTIE_1;
   ...
  NOM_SORTIE_ns   : out TYPE_SORTIE_ns);
end NOM_DE_LA_FONCTION_DECRITE;

-- Description interne
architecture COMPORTEMENT of NOM_DE_LA_FONCTION_DECRITE is
  -- Déclaration des signaux internes
  signal SIGNAL_INTERNE_1 : TYPE_SIGNAL_INTERNE_1;
   ...
  signal SIGNAL_INTERNE_s : TYPE_SIGNAL_INTERNE_s;
  -- Déclaration des constantes
  constant CONSTANTE_1 : TYPE_CONSTANTE_1 := Valeur;
   ...
  constant CONSTANTE_c : TYPE_CONSTANTE_c := Valeur;
   ...
begin
  -- Ecriture des équations du domaine concurrent
   ...
  -- Ecriture des processus
  NOM_PROCESS_1 : process (Liste_de_sensibilité_1)
    -- Déclaration des variables
    variable NOM_VARIABLE_1 : TYPE_VARIABLE_1 := Valeur_Initiale;
     ...
    variable NOM_VARIABLE_v : TYPE_VARIABLE_v := Valeur_Initiale;
    -- Déclaration de constantes (idem début section Architecture)
     ...
  begin
    -- Instructions du domaine séquentielle : opérations d'affectation,
    -- structures conditionnelles, structures répétitives.
  end process;
   ...
  -- Idem pour d'autres processus
end COMPORTEMENT;
```

Chapitre 3

Interfaçage avec l'extérieur : Entité

1 Déclaration des entrées et des sorties

L'entité (mot clé entity) permet de préciser les signaux vus de l'extérieur intervenant dans la fonction créée : c'est la vue fonctionnelle externe et c'est celle qui sera représentée sous la forme d'un symbôle dans un projet hiérarchique ayant un schéma pour sommet. Pour chaque signal, il est nécessaire de préciser le type de données qu'il véhicule ainsi que le sens de circulation de ces données.

2 Exemples de déclarations d'entités

2.1 Interfaçage d'une fonction ET à 2 entrées

L'extrait de code ci-dessous est l'entité de la porte ET *porte_et2* présente sur le schéma de la figure 3.1.

```
entity porte_et2 is port (
  entree_a : in std_logic;
  entree_b : in std_logic;
  sortie   : out std_logic);
end porte_et2;
```

Remarques :

- Quartus impose que le fichier VHDL et l'entité aient le même nom. Ici, *porte_et2* sera donc stockée dans un fichier VHDL dénommé *porte_et2.vhd*
- Avant de préciser le type d'une grandeur, on précise sa direction : *in*, *out* ou *inout*

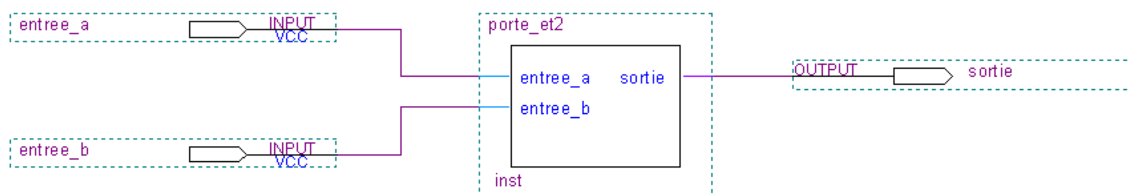

FIGURE 3.1 – Sommet de la hierarchie de la porte ET2

2.2 Compteur décimal 4 bits avec remise à 0 asynchrone

C'est typiquement un compteur BCD tel que celui auquel il est fait référence dans la section **??** de ce document.

On a donc deux entrées scalaires et une sortie vectorielle :

- Entrées :

  - clk_1hz : le signal d'horloge de type STD_LOGIC
  - reset : le signal de remise à 0 asynchrone (c'est à dire pris en compte dès son apparition et donc prioritaire sur le signal d'horloge) de type STD_LOGIC
- Sortie :
  - bcd[3.0] : le vecteur de sortie (valeur codée en binaire comprise entre 0 et 9 dans le cas de ce compteur) de type STD_LOGIC_VECTOR

Vu de l'extérieur, l'entité peut être représentée par le symbôle de la figure 3.2.

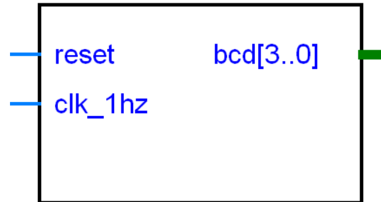

FIGURE 3.2 – Vue externe du compteur BCD

Le code VHDL correspondant est donné ci-dessous :

```
entity compteur_bcd is port (
  reset    : in   std_logic;
  clk_1hz  : in   std_logic;
  bcd      : inout std_logic_vector(3 downto 0));
end compteur_bcd;
```

Remarque : bcd est déclaré en *inout* car il sera vu plus tard dans le code VHDL à la fois comme une entrée et une sortie (en effet, quand on écrit bcd <- bcd + 1, on peut aisément voir que bcd apparaît à la fois en entrée et en sortie de l'équation).

Chapitre 4

Section Architecture

1 Rôle de la section Architecture

L'entité étant créée et fournissant l'interfaçage avec les autres schémas, programmes, etc., il reste à expliciter le ou les comportements de l'élément réalisé. Pour cela, on dispose de structures de programmation permettant de traiter des flots de données (DataFlow) par des instructions dites concurrentes ou d'opérer à une description comportementale d'un phénomène en utilisant plusieurs processus (mot clé process) dans la section architecture.

2 Traitements de flots de données

Il s'agit d'effectuer des traitements dit concurrents (parallèles) totalement asynchrones. Les équations combinatoires relatives au traitement du flot de données sont données directement par le programmeur.

Dans le domaine concurrent, toutes les lignes de codes ont une action immédiate et l'ordre d'écriture des équations n'a pas d'incidence sur les résultats : les équations sont permanentes et appartiennent d'une certaine façon à des travaux parallèles. Concrètement, cela signifie que ces équations sont calculées dans des portions distinctes les unes des autres du composant cible. C'est d'ailleurs ce qui fait la puissance des FPGAs dans le domaines du traitement du signal puisqu'il est en effet possible d'effectuer à chaque instant plusieurs milliers de calculs en parallèles. L'affectation d'une valeur à un signal utilise l'opérateur $<=$

Notes :

• on parle souvent de domaine concurrent pour désigner la partie d'un programme ne traitant qu'un flot de données ;

• il est possible d'ajouter aux équations des critères temporels utiles dans le cadre d'une simulation temporelle (timing simulation).

2.1 Constantes

Pour une meilleure lisibilité du code VHDL, le programmeur peut déclarer des constantes basées sur un type prédéfini et précédemment déclaré. La déclaration d'une constante doit être faite avant le mot clé begin d'une section *process* (voir suite) ou une section *architecture*. La déclaration d'un constante utilise le mot clé *constant*.

Syntaxe :

```
constant IDENTIFICATEUR : TYPE_DE_LA_CONSTANTE := Valeur;
```

Exemples :

```
constant MA_CONSTANTE_1 : STD_LOGIC_VECTOR (7 downto 0) := "00001001";
constant MA_CONSTANTE_2 : INTEGER := 12;
```

2.2 Signaux

2.2.1 Définition

Afin de clarifier l'écriture de certaines équations, il peut être intéressant d'en effectuer une décomposition basée sur l'introduction de nouveaux signaux. Ces signaux, qualifiés de signaux internes, représentent alors des équipotentielles réelles que le compilateur pourra faire disparaître ultérieurement si nécessaire.

2.2.2 Déclaration

La déclaration d'un signal s'effectue entre les mots clés *architecture* et de son *begin* associé (voir structure détaillée d'un programme VHDL au paragraphe 2). Elle utilise le mot clé *signal*.

Syntaxe :

```
signal IDENTIFICATEUR : TYPE_DU_SIGNAL := Valeur d'initialisation;
```

Exemples :

```
signal MON_SIGNAL_1 : STD_LOGIC_VECTOR (7 downto 0) := "00001001";
signal MON_SIGNAL_2 : INTEGER := 12;
```

Note : la valeur d'initialisation n'est utile que lors d'une simulation. La synthèse ne l'exploite pas.

2.3 Exemples de programmes VHDL utilisant seulement un traitement de flots de données

2.3.1 Description fonctionnelle de la fonction ET2

Cet exemple propose une description fonctionnelle en VHDL du bloc *porte_et2*.

```
architecture comportement of porte_et2 is
begin
  sortie <= entree_a and entree_b;
end comportement;
```

On retrouve dans cette description les opérateurs <= et *and* vus dans le chapitre des éléments de base du langage VHDL.

A noter qu'une ligne de code, ou qu'une section de code, se cloture par un point virgule avec pour corrolaire qu'une longue équation peut donc s'écrire sur plusieurs lignes de texte ; seul le point virgule cloturant la ligne de code à proprement parlé comme le montre l'exemple ci-dessous.

```
    sortie <= entree_a
                  and entree_b;
```

2.3.2 Description en langage VHDL d'un multiplexer 4 vers 1 avec 2 entrées de sélection

Le multiplexeur dont la description concurrente est donnée ici possède le fonctionnement suivant :

sel_0 = sel_1 = 0 -> sortie = entree_1

sel_0 = 1, sel_1 = 0 -> sortie = entree_2

sel_0 = 0, sel_1 = 1 -> sortie = entree_3

sel_0 = sel_1 = 1 -> sortie = entree_4

Le symbôle externe de ce multiplexeur est donné à la figure 4.1.

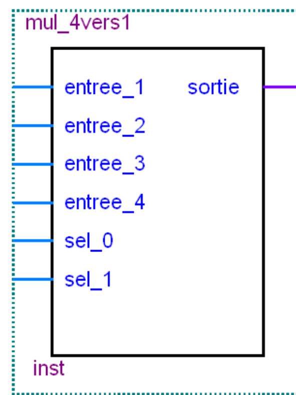

FIGURE 4.1 – Symbôle du multiplexeur 4 vers 1 à 2 entrées

Le code VHD associé est le suivant :

```
library ieee; use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mul_4vers1 is port (
  entree_1  : in std_logic;
  entree_2  : in std_logic;
  entree_3  : in std_logic;
  entree_4  : in std_logic;
  sel_0     : in std_logic;
  sel_1     : in std_logic;
  sortie    : out std_logic);
end entity;
architecture comportement_mul_4vers1 of mul_4vers1 is
begin
  sortie <= (entree_1 and not(sel_0) and not(sel_1))
            or (entree_2 and sel_0 and not(sel_1))
            or (entree_3 and not(sel_0) and sel_1)
            or (entree_4 and sel_0 and sel_1);
end comportement_mul_4vers1;
```

2.3.3 Démultiplexeur 3 vers 8 (versions 1, 2 et 3)

Le bloc démultiplexeur étudié possède une entrée vectorielle sur 3 bits et une sortie scalaire sous la forme d'un bus de 8 bits. Sa table de vérité, exprimée avec des données binaires, est rappelée dans le tableau ci-dessous :

entree	sortie
"000"	"00000001"
"001"	"00000010"
"010"	"00000100"
"011"	"00001000"
"100"	"00010000"
"101"	"00100000"
"110"	"01000000"
"111"	"10000000"

La programmation de cette table de vérité assez particulière n'a pas de solution unique. Dans cette partie, trois solutions sont fournies.

La première version, simple mais lourde consiste à élaborer l'équation des différentes sorties et à procéder à une programmation dans le domaine concurrent :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
--------------------------------------------------------------------------------
-- Description externe du démultiplexeur demul_3vers8
-- Objet : Démultiplexeur 3 vers 8
-- Signaux d'entrées :
-- entree(2..0) : Bus de 3 STD_LOGIC
-- Signaux de sorties :
-- sortie(7..0) : Vecteur de 7 STD_LOGIC
--------------------------------------------------------------------------------
entity demul_3vers8 is port (
  entree : in std_logic_vector (2 downto 0);
  sortie : out std_logic_vector (7 downto 0));
end demul_3vers8;
--------------------------------------------------------------------------------
-- Description dans le domaine concurrent par déduction des
-- équations des sorties d'après la table de vérité (sans commentaire)
--------------------------------------------------------------------------------
architecture comportement_demul_3vers8 of demul_3vers8 is
begin
  sortie(0) <= not(entree(2)) and not(entree(1)) and not(entree(0));
  sortie(1) <= not(entree(2)) and not(entree(1)) and entree(0);
  sortie(2) <= not(entree(2)) and entree(1) and not(entree(0));
  sortie(3) <= not(entree(2)) and entree(1) and entree(0);
  sortie(4) <= entree(2) and not(entree(1)) and not(entree(0));
  sortie(5) <= entree(2) and not(entree(1)) and entree(0);
  sortie(6) <= entree(2) and entree(1) and not(entree(0));
  sortie(7) <= entree(2) and entree(1) and entree(0);
end comportement_demul_3vers8;
```

Remarques :

- L'analyse de ce programme montre que la représentation vectorielle des entrées et des sorties n'apporte rien avec la technique de programmation choisie ici.
- La section architecture étant la seule à changer, tout ce qui la précède ne sera plus rappelé dans les prochaines solutions.

Résultats de simulation obtenus avec la première version du programme :

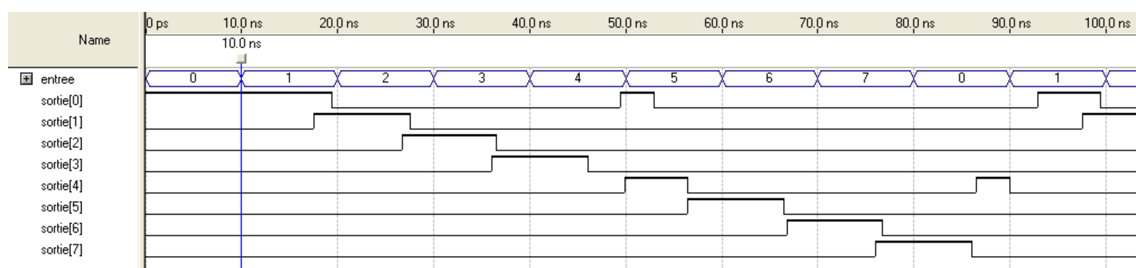

FIGURE 4.2 – Simulation temporelle du démultiplexeur 3 vers 8

Note : on retrouve ce que la simulation de la porte ET à 2 entrées avait fait apparaître, à savoir que la réponse dans un composant quelqu'il soit n'est pas instantannée. Ici, le temps de retard varie en fonction de la sortie et peut atteindre pratiquement 10 ns dans certains cas (la sortie 4).

Seconde version.

Elle propose un codage reposant directement sur la table de vérité pour une meilleure lisibilité d'une part et plus de facilité dans l'écriture (on saute l'étape consistant à déduire les équations de la table de vérité ainsi que leur simplification éventuelle.

Le code correspondant est le suivant :

```
-- voir version 1 du programme de décodeur pour les
-- déclarations de bibliothèques et d'entité ...
-- Description dans le domaine concurrent en utilisant le mode
-- d'affectation conditionnelle : structure ... when ... else ...
-- La table de vérité est clairement retranscrite (sans commentaire)
architecture comportement_demul_3vers8 of demul_3vers8 is
begin
  sortie <= "00000001" when entree = "000" else
            "00000010" when entree = "001" else
            "00000100" when entree = "010" else
            "00001000" when entree = "011" else
            "00010000" when entree = "100" else
            "00100000" when entree = "101" else
            "01000000" when entree = "110" else
            "10000000" when entree = "111" else
            "00000000";
end comportement_demul_3vers8;
```

La troisième version propose une structure du domaine concurrent encore mieux adaptée à la description de tables de vérité. Il s'agit de la structure with ... select ...

Le programme suivant en montre un exemple d'utilisation :

```
-- voir version 1 du programme de décodeur pour les déclarations
-- de bibliothèques et d'entité...
-- Description dans le domaine concurrent en utilisant le mode
-- d'affectation conditionnelle : structure ... when ... else ...
-- La table de vérité est clairement retranscrite (sans commentaire)
architecture comportement_demul_3vers8 of demul_3vers8 is
begin
  with entree select
    sortie <= "00000001" when "000",
              "00000010" when "001",
              "00000100" when "010",
              "00001000" when "011",
              "00010000" when "100",
              "00100000" when "101",
              "01000000" when "110",
              "10000000" when "111";
end comportement_demul_3vers8;
```

Note 1 : cette forme impose de prévoir toutes les possibilités de la table (le cas échéant, le mot clé when others permet de prévoir une solution aux cas non envisagés de façon explicite).

Note 2 : lorsque le vecteur de sortie est le même pour une succession de valeur du vecteur d'entrée exprimé sous la forme d'une variable entière, il est possible de simplifier l'opération :

Soient X la variable d'entrée de type entier d'un système et S son vecteur de sortie.

S = a pour les valeurs de X comprises entre 10 et 15 pourra s'écrire (avec S = 0 sinon) :

```
with X select S <= a when 10 to 15, 0 when others;
```

De la même façon, si S = a pour X = 10 et X=12, puis S = 0 dans les autres cas, on écrira :

```
with X select S <= a when 10 | 12, 0 when others;
```

où le signe | s'apparente à un OU logique.

3 Description comportementale

3.1 Utilité de la structure process

La description comportementale repose sur l'utilisation d'une structure, le processus, à l'intérieur duquel on utilise des structures répétitives, conditionnelles ou séquentielles. La prise en compte des lignes de code d'un processus se fait séquentiellement et l'exploitation de l'ensemble permet au compilateur de synthétiser les équations à intégrer dans le circuit.

3.2 Détails sur l'exécution d'un processus

3.2.1 Activation d'un processus

On peut considérer un processus VHDL comme une section de code dont l'exécution peut être lancée lors du changement d'état d'une certaine classe de signaux du modèle créé. Ces signaux susceptibles d'activer le processus doivent être répertoriés dans une liste dite liste de sensibilité du processus.

3.2.2 Affectation de signaux dans un processus : « Scheduling »

Afin de pouvoir traiter des problèmes synchrones facilement, les signaux sont perçus différemment dans le domaine séquentiel (à l'intérieur d'un processus) que dans le domaine concurrent. Au début de l'exécution d'un processus, les états des signaux sont mémorisés dans des variables tampons. Lors de l'exécution séquentielle du code du processus, l'affectation d'un signal ne modifie que la variable tampon qui lui est associée. A la fin du processus, le signal reçoit le contenu de la variable tampon. Les anglo-saxons appellent ça le Scheduling.

3.2.3 Exemples de processus simples. Exécutions commentées

Exemple (code partiel) :

```
...
architecture comportement_exemple of exemple is
  signal A,B,C : std_logic;
begin
  process(A)
  begin
    B <= A;
    C <= B;
  end process;
end comportement_exemple;
```

Explications :

Dans cette exemple, trois signaux A, B, C de type STD_LOGIC sont traités par un processus. Ce processus possède une liste de sensibilité se limitant au signal A, c'est à dire qu'un changement d'état de A entraînera son exécution.

A t = 0-, on suppose que A = '1', B = '1', C = '0'. A t = 0, A change de valeur de sorte qu'il vaut '0' ce qui active le processus le traitement s'opère comme suit :

- les signaux A, B et C sont mémorisés dans des variables tampons notées ici : a, b et c.
- B <= A signifie que b (le tampon de B) reçoit A, i.e. b = '0'
- C <= B signifie que c (le tampon de C) reçoit B, i.e. c = '1'.
- Finalement les tampons a, b, c, sont transférés dans les signaux A, B, C. A t = 0+, les signaux ont les valeurs suivantes : A = '0', B = '0', C = '1'.

Remarque : oublier le mécanisme des mémoires tampons conduit à trouver C = '0' à la fin du processus au lieu de C = '1'.

Reprenons le code de l'exemple précédant et ajoutons B à la liste de sensibilité de l'unique processus.

Exemple (code partiel) :

```
architecture comportement_exemple of exemple is
  signal A,B,C : std_logic;
begin
  process(A,B)
  begin
    B <= A;
    C <= B;
  end process;
end comportement_exemple;
```

Explications :

A t = 0-, on suppose que A = ’1’, B = ’1’, C = ’0’.

A t = 0, A change de valeur de sorte qu’il vaut ’0’ ce qui active le process dont le traitement s’opère comme suit :

- les signaux A, B et C sont mémorisés dans des variables tampons notées ici : a, b et c.
- B <= A signifie que b (le tampon de B) reçoit A, i.e. b = ’0’.
- C <= B signifie que c (le tampon de C) reçoit B, i.e. c = ’1’.
- Finalement les tampons a, b, c, sont transférés dans les signaux A, B, C. A t = 0+, les signaux ont les valeurs suivantes : A = ’0’, B = ’0’, C = ’1’. Note : le cycle d’exécution du processus qui vient d’être décrit s’appelle un cycle delta. B a donc changé d’état par rapport à ce qu’il valait en t = 0-. Comme il appartient à la liste de sensibilité du processus celui-ci est relancé : on assiste à un second cycle delta.
- les signaux A, B et C sont mémorisés dans des variables tampons notées ici : a, b et c.
- B <= A signifie que b (le tampon de B) reçoit A, i.e. b = ’0’.
- C <= B signifie que c (le tampon de C) reçoit B, i.e. c = ’0’.
- Finalement les tampons a, b, c, sont transférés dans les signaux A, B, C. A t = 0++, les signaux ont les valeurs suivantes : A = ’0’, B = ’0’, C = ’0’. L’état des différentes variables de la liste de sensibilité étant stabilisé, le processus n’est pas relancé une troisième fois.

Note : on découvre intuitivement qu’il faut être très vigilant dans l’écriture des processus et le choix des éléments de la liste de sensibilité sous peine d’aboutir à quelque chose de non synthétisable car conduisant à l’exécution d’une infinité de cycle delta. On utilisera par contre cette propriété pour la création de signaux d’horloge dans un banc de test.

Remarque importante : le premier exemple de processus montrait un processus utilisant le signal B comme une entrée sans pour autant l’avoir spécifié dans la liste de sensibilité.

3.3 Variables

3.3.1 Dualité signal - variable

De même qu’il était possible de définir des signaux dans le domaine concurrent, le programmeur peut recourir à des variables dans le domaine séquentiel.

3.3.2 Déclaration d’une variable

La déclaration d’une variable se place entre l’entête du process et le début de ce process identifié par le mot clé begin (voir structure détaillée d’un programme VHDL au paragraphe 4). Elle utilise le mot clé variable.

Syntaxe :

```
variable IDENTIFICATEUR : TYPE_DE_LA_VARIABLE := Valeur d'initialisation;
```

Exemple :

```
variable VARIABLE_1 : std_logic_vector(7 downto 0) := "00001001";
```

3.3.3 Utilisation et comportement des variables

L'affectation d'une valeur à une variable utilise l'opérateur := Contrairement à l'affectation à un signal qui utilise une variable temporaire au sein d'un processus, l'affectation à une variable revêt un caractère immédiat. On peut affecter la valeur d'un signal à une variable ou le contraire.

```
...
signal COMPTEUR : unsigned(3 downto 0) := "1001";
process(COMPTEUR)
  variable ENTIER : integer;
begin
  ENTIER := to_integer(COMPTEUR);
  ENTIER := ENTIER + 3;
  COMPTEUR <= to_unsigned(ENTIER, 4);
end process;
...
```

3.4 Structures conditionnelles

3.4.1 Instruction if

Syntaxes contractées :

```
if condition then instruction_1; else instruction2; end if;
if condition then instruction_1; end if;
```

Syntaxes étendues :

```
if condition then
  instruction;
  ...
  ...
[
else
  instruction;
  ...
  ...
]
end if;
```

Syntaxes (cas des conditions succédant au else) :

```
if condition_1 then instruction_1;
elsif condition_2 then instruction_2; end if;
end if;
```

Pour plus de précisions, le guide de référence complétera l'étendue de la syntaxe de cette structure.

Note : les zones entre crochets sont facultatives.

3.4.2 Exemples de programmes complets utilisant l'instruction if

Démultiplexeur 3 vers 8 (version 4) Ce démultiplexeur a été décrit dans le paragraphe 2.3.3. Deux programmes ont déjà été proposés dans le domaine concurrent pour sa réalisation. Dans le domaine séquentielle, l'instruction if permet d'obtenir une solution intéressante.

Version 3 du décodeur 3 vers 8 :

```
-- voir version 1 du programme de décodeur pour les déclarations de bibliothèques et d'entité.
-- Description dans le domaine séquentiel en utilisant l'instruction if
-- La table de vérité est clairement retranscrite dans le cas du décodeur pris en
-- exemple mais la méthode n'est pas généralisable pour des tables de vérité plus
-- complexes.
...
architecture comportement_demul_3vers8 of demul_3vers8 is
begin
  process(entree)
  begin
    if (entree = b"000") then sortie(0) <= '1'; else sortie(0) <= '0'; end if;
    if (entree = b"001") then sortie(1) <= '1'; else sortie(1) <= '0'; end if;
    if (entree = b"010") then sortie(2) <= '1'; else sortie(2) <= '0'; end if;
    if (entree = b"011") then sortie(3) <= '1'; else sortie(3) <= '0'; end if;
    if (entree = b"100") then sortie(4) <= '1'; else sortie(4) <= '0'; end if;
    if (entree = b"101") then sortie(5) <= '1'; else sortie(5) <= '0'; end if;
    if (entree = b"110") then sortie(6) <= '1'; else sortie(6) <= '0'; end if;
    if (entree = b"111") then sortie(7) <= '1'; else sortie(7) <= '0'; end if;
  end process;
end comportement_demul_3vers8;
```

Note : De même que pour la version 3, seul le code relatif à la section architecture est présenté ici. On pourra se reporter au paragraphe 2.3.3 pour se remémorer ce qui précède cette partie.

3.4.3 Instruction case

L'instruction case permet d'effectuer un traitement adapté à chaque valeur de l'objet qu'elle prend en paramètre. Elle est particulièrement intéressante pour la description de systèmes combinatoires dont les tables de vérité sont explicitement définies (elle est l'équivalent de la structure with ... select ... du domaine concurrent dans le domaine séquentiel).

Elle se révèle surtout précieuse pour la description de machines d'états.

Syntaxe : voir guide de référence VHDL en ligne.

3.4.4 Exemple de programme utilisant l'instruction case : démultiplexeur 3 vers 8 (version 5)

```
-- Description dans le domaine séquentiel en utilisant l'instruction case
-- C'est l'instruction permettant la description la plus élégante
-- d'un phénomène combinatoire dont la table de vérité peut être explicitée.
...
architecture comportement_demul_3vers8 of demul_3vers8 is
begin
  process(entree)
  begin
    case entree is
      when "000" => sortie <= "00000001"; -- Le signe => se lit "alors"
      when "001" => sortie <= "00000010"; -- Quand entree = "001" alors
                                          -- sortie reçoit "00000010"
      when "010" => sortie <= "00000100";
      when "011" => sortie <= "00001000";
      when "100" => sortie <= "00010000";
      when "101" => sortie <= "00100000";
      when "110" => sortie <= "01000000";
      when "111" => sortie <= "10000000"; -- Affectation pour tous les cas
                                          -- qui n'ont pas encore été prévus.
      when others => sortie <= "00000000";
    end case;
  end process;
end comportement_demul_3vers8;
```

Remarque : lorsque plusieurs cas nécessitent le même traitement, il est possible d'utiliser le mot clé when suivi de l'ensemble des valeurs de même comportement séparées par des symbôles |.

Exemple :

```
when "110" | "111" => ......
```

3.5 Structures répétitives

3.5.1 Instruction for

Elle permet une description plus rapide de phénomènes itératifs en contrôlant l'évolution d'une variable.

Syntaxe :

```
for Nom_Variable in Valeur1 to Valeur2 loop
...
end loop;
```

Note : lorsque Valeur1 > Valeur2, le mot clé *to* doit être remplacé par *downto*.

Exemple : utilisation de la structure for ... loop pour l'implémentation du démultiplexeur 3 vers 8

```
...
architecture comportement_demul_3vers8 of demul_3vers8 is
begin
  process(entree)
    variable N : unsigned (2 downto 0);
  begin
    N := unsigned(entree);
    sortie <= "00000000";
    for I in 0 to 7 loop
      if (N=I) then sortie(I) <= '1'; end if;
    end loop;
  end process;
end comportement_demul_3vers8;
```

Remarques :

- on introduit une variable intermédiaire de type *unsigned* dans cet exemple pour pouvoir effectuer la comparaison N=I dont la définition n'existe pas avec le type *std_logic_vector*.
- on retrouve de nombreux glitchs dans la simulation temporelle (cf. figure 4.3).

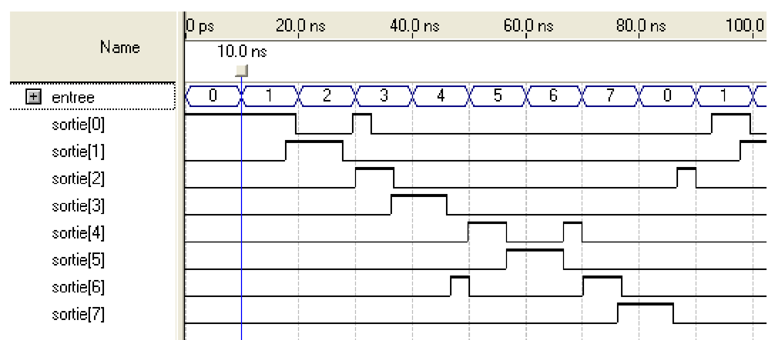

FIGURE 4.3 – Simulation temporelle faisant apparaître de nombreux glitchs

3.5.2 Instruction while

L'exemple précédent peut être écrit avec la structure while ... loop ...

```
...
architecture comportement_demul_3vers8 of demul_3vers8 is
begin
  process(entree)
    variable N : integer range 0 to 8;
  begin
    N := to_integer(unsigned(entree));
    sortie <= "00000000";
    while I < 8 loop
      if (N=I) then sortie(I) <= '1'; end if;
    end loop;
  end process;
end comportement_demul_3vers8;
```

L'indice I utilisé dans la structure while ... loop est initialisé automatiquement à 0 au début de la structure. Son incrémentation est elle aussi automatique et se produit en arrivant au mot clé end loop.

Remarque : l'utilisation de cette structure est déconseillée. On lui préfère la structure for ... loop dont l'écriture donne une description plus explicite.

3.6 Cas d'un système rythmé par un signal d'horloge : compteur BCD

Un des grands champs d'application des processus concerne les systèmes rythmés par un signal d'horloge. La suite de cette sous-section propose l'étude détaillée d'un compteur BCD (comptant de 0 à 9).

3.6.1 Objectif

Il est de décrire en VHDL le fonctionnement d'un compteur binaire 4 bits synchrone avec remise à 0 asynchrone. Vu de l'extérieur, ce compteur se présente sous la forme d'un bloc comportant deux entrées, une entrée d' horloge de comptage et une entrée de reset, et une sortie vectorielle sur 4 bits.

Remarques :

- Le comptage s'effectue sur front montant d'horloge, ici *clk*.
- L'entrée de remise à 0, ici *reset*, est active à l'état haut.

3.6.2 Brève analyse

De ce cahier des charges succinct, on peut trouver les bases du programme permettant de décrire le compteur. La fonction à décrire étant séquentielle, l'emploi de un ou plusieurs processus est indispensable (la simplicité du problème permet de se limiter à un seul processus).

Il faut en outre définir ce processus :

- Il doit répondre aux événements des deux entrées que sont *clk* et *reset* (cf. liste de sensibilité).
- Si *reset* = '0' (i.e. pas de reset), on cherche un front montant de *clk* pour incrémenter la valeur du compteur.
- Si *reset* = '1' on remet le compteur à 0.
- Le signal du compteur étant pris comme entrée dans le processus, il faut l'ajouter à la liste de sensibilité du processus. De plus, il doit être déclaré sous la forme buffer au niveau de l'entité plutôt que out.

3.6.3 Codage

Cette analyse rapide permet d'aboutir au programme suivant :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity compteur_bcd is port(
  reset    : in  std_logic;
  clk      : in  std_logic;
  bcd      : inout std_logic_vector(3 downto 0)
);
end compteur_bcd;
architecture comportement_compteur_bcd of compteur_bcd is
begin
  process(clk, reset, bcd)
  begin
   -- le reset est prioritaire (il est asynchrone)
    if reset = '1' then bcd <= "0000";
    elsif clk'event and clk = '1'
    then
      -- l'addition n'est pas supportée pour les std_logic_vector, il faut
      -- passer par différentes étapes de conversions pour incrémenter
      - le compteur
      bcd <= bcd+1;
      -- si bcd valait 9 en entrant dans le processus, il faut le remettre
      -- à 0 (remarque: l'incrémentation précédente n'a pas d'effet sur la
      -- valeur lue de bcd. C'est la valeur future de bcd qui a été affectée
      -- par cette incrémentation)
      if bcd = 9 then bcd <= "0000"; end if;
    end if;
  end process;
end comportement_compteur_bcd;
```

3.6.4 Variante avec une remise à 0 synchrone

Dans certains cas, on peut avoir besoin de réaliser une remise à 0 du compteur sur demande et qui soit synchrone d'horloge, c'est à dire effective seulement au prochain front montant de l'horloge (les deux modes de reset peuvent même être présents simultanément).

Le code est le suivant :

```
...
...
architecture comportement_compteur_bcd of compteur_bcd is
begin
  process(clk, reset, bcd)
  begin
    if clk'event and clk = '1'
    then
      bcd <= bcd + 1;
      if bcd = 9 then bcd <= "0000"; end if;
      -- le reset est traité en dernier
      if reset = '1' then bcd <= "0000"; end if;
    end if;
  end process;
end comportement_compteur_bcd;
```

4 Architecture de test

4.1 Objectif de l'architecture de test

Lors des étapes de test et de mise au point, Quartus permet de créer des stimuli sur la base d'une interface interactive (Vector Weveform Editor). Ce mode de description trouve rapidement ses limites lorsque la complexité des projets augmente et justifie l'utilisation d'un langage de description pour les stimuli plus

importants. Lorsque le simulateur utilisé est un simulateur VHDL (i.e. qu'il est capable d'exploiter un code source VHDL pour la simulation), il est alors possible de créer une architecture spécifique pour appliquer les stimuli sur l'ensemble du projet ou seulement une partie du projet. On parle alors logiquement d'architecture de test, et de test bench pour le programme qui l'intègre.

Note : le simulateur intégré à Quartus II n'étant pas capable de simulation au niveau source VHDL (il se limite à une simulation RTL), il est donc impossible d'exploiter des bancs de test écrit en VHDL à l'intérieur de Quartus. Pour réaliser de tels tests, on utilise un simulateur externe (EDA Simulator Extern Tool) : Modelsim-Altera.

4.2 Test bench (banc de test)

Le test bench est un programme VHDL qui vient se placer à un niveau hiérarchique supérieur au projet ou module à tester. Il contient une instance du module à tester (mot clé *component*) auquel le reste des signaux du programme de test sont reliés (mot clé *port*). La connaissance détaillée de la syntaxe des mots clés *component* et *port* n'est pas forcément utile. En effet, un fichier de test vectoriel créé avec l'éditeur de Quartus pouvant être exporté en fichier de test VHDL, il suffira de se contenter de créer un fichier de test restreint avec l'éditeur de Quartus et de l'exporter pour le modifier et l'agrémenter en fonction de besoins plus spécifiques avec un simple éditeur de texte.

4.3 Création et Ecriture du test bench

Le code ci-dessous propose un exemple de test bench écrit en VHDL et dont le squelette de base est fourni par la fonction d'exportation d'un fichier Waveform de Quartus en fichier VHDL à l'extension .vht :

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- Entité du module de test
ENTITY demul_3vers8_vhd_vec_tst IS
END demul_3vers8_vhd_vec_tst;

-- Architecture de test
ARCHITECTURE demul_3vers8_arch OF demul_3vers8_vhd_vec_tst IS
  SIGNAL entree : STD_LOGIC_VECTOR(2 DOWNTO 0);
  SIGNAL sortie : STD_LOGIC_VECTOR(7 DOWNTO 0);
  constant duree_entree : time := 2000 ns;

  COMPONENT demul_3vers8  PORT (
      entree : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
      sortie : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
  END COMPONENT;

BEGIN
  i1 : demul_3vers8
  PORT MAP ( -- list connections between master ports and signals
    entree => entree,
    sortie => sortie
  );

  -- Portion de code ajoutée pour générer les signaux d'entrées
  process
  begin
    entree(0) <= '0';
    wait for 100 ns;
    entree(0) <= '1';
    wait for 100 ns;
  end process;

  process
  begin
    entree(1) <= '0';
    wait for 200 ns;
    entree(1) <= '1';
    wait for 200 ns;
  end process;
        process
  begin
    entree(2) <= '0';
    wait for 400 ns;
    entree(2) <= '1';
    if now >= duree_entree then wait; else wait for 400 ns; end if;
  end process;
END demul_3vers8_arch;
```

Instruction à suivre pour créer ce fichier :

1. Sélectionner l'entité à tester dans le project navigator et la placer au sommet de la hierarchie (menu contextuel -> set as top level entity).
2. Ajouter un nouveau fichier de type Vector Waveform File (File -> New ...).
3. N'y ajouter aucun signal et l'exporter immédiatement avec File -> Export ... et taper le nom du fichier qui en résultera (conseil de nom : test_ suivi du nom de l'entité à tester).
4. Fermer le fichier Vector Waveform sans le sauver et ouvrir le fichier de test bench (extension .vht).

5. Ajouter vos lignes de code à partir de l'avant dernière ligne du fichier fourni par Quartus pour réaliser votre test.

Remarques sur l'écriture du code de test :

- Afin d'assurer le démarrage de la simulation, le programme doit posséder au moins un process sans liste de sensibilité.
- On utilise un ou plusieurs process pour générer un ou plusieurs signaux.
- Ces process sont généralement spécifiés sans liste de sensibilité (cela signifie qu'ils seront immédiatement exécutés sans condition).
- Pour gérer le temps dans un process, on utilise le mot clé *wait* dans diverses formes :
  - Il permet de réaliser une pause pendant une durée déterminée.
    - Syntaxe : *wait for DUREE*
  - L'exécution du process peut être reprise lors du changement d'état d'un élément d'une liste de sensibilité.
    - Syntaxe : *wait on LISTE_DE_SENSIBILITE* ;
  - De même, le système peut attendre qu'une condition soit vraie avant de continuer l'exécution du process.
    - Syntaxe : *wait until CONDITION* ; ou *wait until CONDITION for DUREE ;* – reprise si la condition est vraie pendant un laps de temps défini par DUREE.
  - *Note :* on peut bloquer définitivement un processus en insérant une instruction *wait ;* utilisée sans argument.

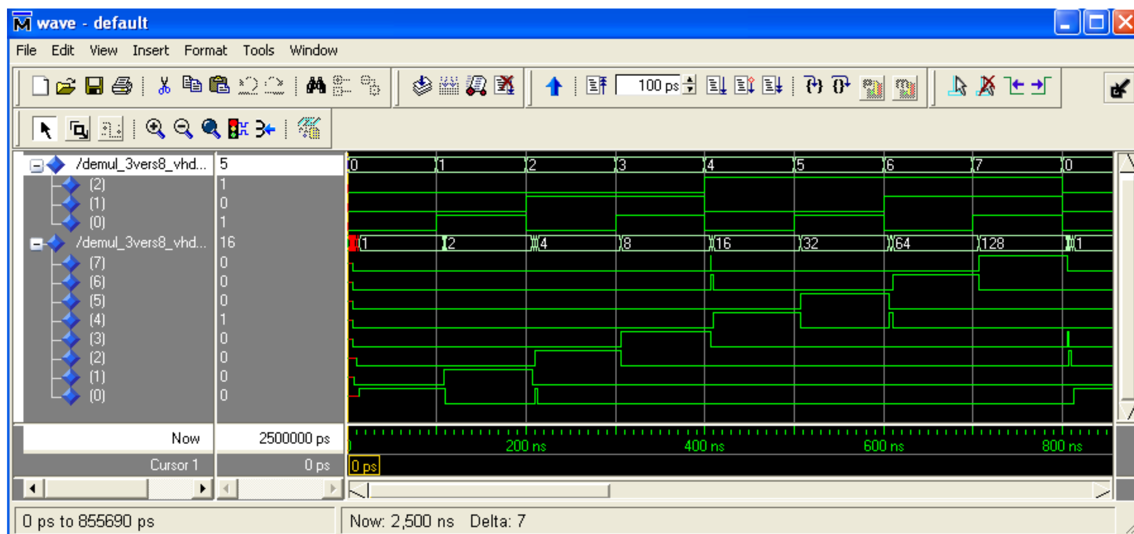

FIGURE 4.4 – Simulation temporelle avec Modelsim-Altera du démultiplexeur 3 vers 8

On peut aussi écrire des processus un peu plus sophistiqués en exploitant davantage les possibilités du langage VHDL comme le montre l'exemple ci-dessous :

```
process
  variable i : integer range 0 to 9;
  variable j : integer range 0 to 4;
  variable uentree : unsigned(2 downto 0);
begin
  for i in 0 to 9 loop
    uentree <= to_unsigned(i, 3);      -- Conversion d'un entier en unsigned

    -- La boucle sur j permet de réaliser la conversion d'un
    -- unsigned vers un std_logic_vector, tous deux basés sur des
    -- éléments de type std_logic.
    for j in 0 to 2 loop
      entree(j) <= uentree(j);
    end loop;

    wait for 100 ns;  -- pause de 100 ns
  end loop;
end process;
```

4.4 Contrôle et report d'erreurs

Le résultat de certaines simulations complexes ne peut pas être facilement exploités à partir des chronogrammes et autres tables de changement d'états.

D'une façon générale, dès que le projet devient un peu complexe, le test bench doit reposer sur l'utilisation de séquences de test dont les résultats sont connus et peuvent être contrôlés au moment de la simulation (table de vecteurs de test par exemple). Les résultats attendus étant accessibles dans le programme de test, le concepteur du test peut inclure des lignes de code assurant que la simulation suit son cours sans erreur de façon à bloquer l'exécution dans le cas contraire. A cet effet, le VHDL possède une structure (assert ... report ... severity ...) dont la syntaxe est : assert CONDITION report MESSAGE severity NIVEAU ; Si la CONDITION est vraie, le process se poursuit sans problème. Si elle ne l'est pas, le process est interrompu et le MESSAGE est affiché dans une boite de dialogue. Une information complémentaire dans cette boite de dialogue renseigne sur le NIVEAU de gravité de l'assertion et l'exécution reprend ensuite son cours si le niveau est NOTE ou WARNING.

Annexes

Interface des bibliothèques IEEE principales

Bibliothèque numeric_std.vhd

Elle définit les types numériques *SIGNED* et *UNSIGNED* qui sont des vecteurs de *STD_LOGIC*. Les opérateurs courants sont définis entre *SIGNED* et *UNSIGNED* ainsi qu'entre *SIGNED* et *INTEGER* puis *UNSIGNED* et *INTEGER*.

Quatre fonctions de conversions intéressantes sont définies pour assurer le passage vers des entiers (naturels pour le type *NATURAL* ou relatifs pour le type *INTEGER*) :

```
function TO_INTEGER (ARG: UNSIGNED) return NATURAL;
function TO_INTEGER (ARG: SIGNED) return INTEGER;
function TO_UNSIGNED (ARG, SIZE: NATURAL) return UNSIGNED;
function TO_SIGNED (ARG: INTEGER; SIZE: NATURAL) return SIGNED;
```

Bibliothèque std_logic_1164.vhd

C'est la bibliothèque IEEE indispensable pour caractériser correctement les signaux réels. Elle définit les types scalaires *STD_ULOGIC* et *STD_LOGIC* ainsi que leurs versions vectoriels *STD_ULOGIC_VECTOR* et *STD_LOGIC_VECTOR*.

Quelques informations complémentaires sur les signaux

Stub Series Terminated Logic (SSTL) devices are a family of electronic devices for driving transmission lines. They are specifically designed for driving the DDR (double-data-rate) SDRAM modules used in computer memory. However, they are also used in other applications, notably, some PCI Express PHYs and other high-speed devices.

Three voltage levels for SSTL are defined :

* SSTL_3, 3.3 V, defined in EIA/JESD8-8 1996 * SSTL_2, 2.5 V, defined in EIA/JESD8-9B 2002 * SSTL_18, 1.8 V, defined in EIA/JESD8-15

All SSTL voltage specs reference a voltage that is exactly VDDQ/2. For example, the VREF for an SSTL_18 signal is exactly 0.9 Volts.

Terminations can be :

* Class I (one series resistor at the source and one parallel resistor at the load) * Class II (one series resistor at the source and two parallel resistors, one at each end). * Class III (asymmetrically parallel terminated) * Class IIII (asymmetrically doubly parallel terminated)

HSTL

High-speed transceiver logic or HSTL is a technology-independent standard for signalling between integrated circuits. The nominal signalling range is 0 V to 1.5 V, though variations are allowed, and signals may be single-ended or differential. It is designed for operation beyond 180 MHz.

The following classes are defined by standard EIA/JESD8-6 from EIA/JEDEC :

* Class I (unterminated, or symmetrically parallel terminated) * Class II (series terminated) * Class III (asymmetrically parallel terminated) * Class IV (asymmetrically double parallel terminated)

Note : Symmetric parallel termination means that the termination resistor at the load is connected to half the output buffer's supply voltage. Double parallel termination means that parallel termination resistors are fitted at both ends of the transmission line.