

Synthèse des CPLD et FPGA en VHDL : aide mémoire

Le VHDL (**V**ery high speed - or scale - integrated circuits **H**ardware **D**escription **L**anguage) est utilisé pour la modélisation et la synthèse de circuits numériques.

La modélisation permet de simuler le comportement d'un circuit numérique, tandis que la synthèse permet de le programmer (si nécessaire).

Nous ne nous intéresserons donc ici qu'à la synthèse.

Cet aide mémoire n'a pas la prétention d'être exhaustif, le but étant simplement de rappeler le minimum que doit connaître un électronicien pour lire le VHDL (de synthèse). Aucun exemple concret n'est donné dans la partie principale, on se reportera pour cela à la bibliographie et à l'annexe.

Depuis sa création, le VHDL a fait l'objet de deux normalisations, une en 87 et l'autre en 93. Quelques différences existent entre les deux normes.

Il faut également avoir à l'esprit que l'on trouvera également des différences, parfois importantes, entre les divers outils de synthèse proposés par les constructeurs. L'opération de multiplication par exemple n'est définie que comme une multiplication par 2 par de nombreux compilateurs. Il est conseillé de se référer à la documentation constructeur.

Ce langage ne fait théoriquement pas de distinction entre majuscules et minuscules. Certains outils de synthèse font cependant une distinction pour quelques cas particuliers (description de l'état haute impédance 'Z' – et non 'z' - avec le logiciel Max+plus II par exemple). Pour une meilleure lisibilité les mots réservés ont été écrits ici en minuscules et en gras, les noms donnés par l'utilisateur en majuscules.

En VHDL les commentaires sont précédés par « -- » et s'arrêtent au retour à la ligne. Pour une meilleure lisibilité, ils sont écrits ici en italique.

Remarque : cet aide mémoire est la version simplifiée d'une description plus complète.

1. Notions de base

La structure typique d'une description VHDL est donnée ci-après :

-- déclaration des ressources externes.

```
library NOM_DE_LA_BIBLIOTHEQUE ;  
use ELEMENT_DE_LA_BIBLIOTHEQUE ;
```

les ressources externes

-- description de l'entité vue comme une « boîte noire » avec des entrées et des sorties, caractérisée par des paramètres

```
entity NOM_DE_L'ENTITE is  
  generic (PARAMETRES : « type »:=VALEURS_FACULTATIVES ) ;  
  port ( NOM_DES_ENTREES_SORTIES : « direction » « type » ) ;  
end NOM_DE_L'ENTITE ;
```

l'entité

-- description de l'architecture à l'intérieure de l'entité ; à une entité peut
 -- correspondre plusieurs architectures

l'architecture

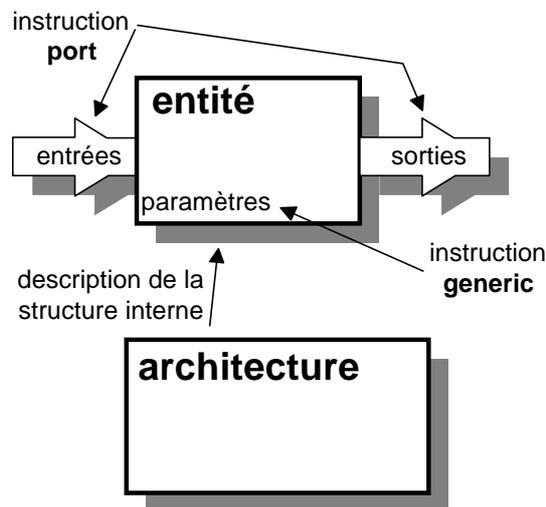
```

architecture NOM_DE_L'ARCHITECTURE of NOM_DE_L'ENTITE is
-- déclaration si nécessaire des objets internes
  NOM_DES_OBJETS_INTERNES : « type » ;
-- déclaration si nécessaire des types
  type NOM_DES_TYPER is (VALEURS_POSSIBLES) ;
-- déclaration si nécessaire des composants
  component NOM_DU_COMPOSANT port (ENTREES :in bit ; SORTIES :out bit) ;

-- description du fonctionnement
  begin
    DESCRIPTION ;
  end NOM_DE_L'ARCHITECTURE ;
  
```

les ressources internes de l'architecture

Comme on peut le voir, la structure comprend une entité, c'est à dire une "boîte noire" décrivant les entrées sorties, et une architecture associée à l'entité décrivant le fonctionnement.



On pourra se reporter au premier exemple de l'annexe 2 pour voir une description minimale en VHDL. A une entité peuvent correspondre plusieurs architectures. De cette manière, il devient possible de modifier la structure interne d'une description sans modifier la vue externe, cette propriété du VHDL devenant très intéressante dès lors que plusieurs description sont imbriquées les unes dans les autres.

2.Syntaxe

On notera au passage la présence de points virgules à la fin de chaque ligne de déclaration et après chaque instruction.

Lorsqu'une liste d'entrées, de sorties ou de paramètres est énumérée, la séparation se fait par des virgules.

Comme dans la plupart des langages informatique, il est fortement conseillé de se servir des tabulations pour améliorer la lisibilité. Les outils de synthèse proposent en général l'utilisation de couleurs différentes.

Lorsque plusieurs opérations sont décrites sur une même ligne, les parenthèses permettent également une lecture plus facile.

Passons maintenant en revue les différents éléments utilisés.

2.1.les instructions port et generic

L'instruction **port** permet de décrire les entrées et sorties.

Facultative l'instruction **generic** permet de spécifier certains paramètres de l'entité, comme par exemple la largeur d'un bus. Le paramètre peut être initialisée directement dans l'entité à une certaine valeur, puis modifié ultérieurement lors de son utilisation (voir plus loin les notions avancées). De cette manière la description peut rester très générale.

2.2.Les directions

Elles permettent de préciser les sens des connexions entrantes ou sortantes de l'entité. Le sens de circulation de l'information est alors clairement défini, permettant une bonne lisibilité.

in : entrée

out : sortie. Une sortie ne peut être relue à l'intérieur de l'architecture associée à l'entité.

inout : entrée-sortie. La connexion peut alors être relue et réutilisée à l'intérieur de l'architecture associée, au détriment de la lisibilité du programme (il devient impossible de dire si une information externe peut entrer ou s'il s'agit d'un simple réutilisation interne).

buffer : sortie pouvant être réutilisée à l'intérieur de l'architecture (contrairement à « **out** »). Cependant, si à l'intérieure de l'architecture, cette sortie est connectée à la sortie de composants internes (voir plus loin), la sortie des composants internes doit aussi être déclarée comme buffer. Cette limitation nuit fortement à l'utilisation de bibliothèques de composants génériques.

2.3.Les objets

Les objets sont des ressources internes à une unité de conception, qui n'apparaissent pas comme entrée ou sortie de l'entité. Il existe trois sortes d'objets :

signal : le signal représente une connexion interne, déclaré dans une entité, une architecture ou un paquetage (voir plus loin). Le signal permet de pallier les limitations des directions **inout** et **buffer**. Il est généralement affecté à une sortie après utilisation en interne.

variable : la variable est généralement utilisée comme index dans la génération de boucle et n'a habituellement pas de correspondance avec un signal physique. Elle ne peut être déclarée que dans un process (voir plus loin) ou un sous-programme (fonction ou une procédure -voir plus loin-). Hormis cette limitation, rien n'interdit cependant d'utiliser une variable à la place d'un signal (en modélisation une variable utilise moins de ressources qu'un signal). Théoriquement une variable change de valeur dès l'affectation, tandis qu'un signal ne change qu'en sortie d'un process. Les outils de synthèse ne respectent cependant pas systématiquement cette règle.

constant : permet une meilleure lisibilité du programme, ainsi qu'une meilleure convivialité. Un composant définit avec une constante de N bits pourra facilement passer de 8 à 64 bits (grâce à l'instruction **generic** par exemple).

2.4.Notation des littéraux

Ces sont les valeurs particulières qu'on donnera aux grandeurs utilisées :

- bits et caractères : entre apostrophes ; exemple '0'.

- chaîne de bits et chaîne de caractères : entre guillemets ; exemple "01110".

- nombre décimaux : exemple pour la valeur mille ; 1000, 1_000, 1E3, 1.00E3.

- nombre hexadécimaux : entre guillemets précédés de x ; exemple x"1AFF" .Certains outils de synthèse ne reconnaissent pas cette notation, et on trouvera plus facilement la notation 16#1AFF# pour le nombre 1AFF (de même 2#.# pour le binaire et .8#...# pour l'octal).

2.5. Les types

2.5.1. types principaux en synthèse

La notion de type est très importante en VHDL, chaque entrée, sortie, signal, variable ou constante, est associé à un type. Sans artifice particulier, il n'est pas question d'effectuer une opération entre deux grandeurs de type différent.

integer : nombre entier pouvant aller de -2^{31} à $2^{31}-1$ (suivant les compilateurs), peut être limité entre deux extrêmes par la déclaration :

« integer range MINI to MAXI ; »

La déclaration d'un entier sans préciser de valeurs limites équivaut donc à créer un bus de 32 bits à l'intérieur du composant cible.

bit : deux valeurs possibles : 0 ou 1 ; l'utilisation de la bibliothèque **ieee1164** permet de d'ajouter une extension avec le type **std_logic**, lequel peut prendre en plus l'état haute impédance : **Z**. (Attention : certains outils de synthèse exigent une majuscule pour le Z)

Les autres états du type **std_logic** définis dans la bibliothèque (**U** -non initialisé-, **X** -inconnu-, **W** -inconnu forçage faible- **H** -NL1 forçage faible-, **L** -NL0 forçage faible- et -quelconque-) sont utilisés en modélisation.

bit_vector : un vecteur de bits est un bus déclaré par exemple pour une largeur de N bits par **bit_vector(0 to N)** ou bien **bit_vector(N downto 0)**. Le bit de poids fort est celui déclaré à gauche, c'est à dire le bit 0 dans la première écriture et le bit N dans la seconde, ce qui explique que cette dernière est souvent préférée.

L'utilisation de la bibliothèque **ieee**, grâce à son paquetage (voir plus loin) **ieee.std_logic_1164**, permet d'utiliser le type **std_logic_vector**, avec la possibilité d'un état haute impédance.

On trouvera également dans le paquetage **ieee.std_logic_arith** de la même bibliothèque la possibilité d'utiliser les type **signed** (représentation de la grandeur en complément à 2) et **unsigned** (représentation en binaire naturel) qui permettent de réaliser des opérations arithmétiques signées ou non.

boolean : ce type est utilisé pour les conditions (dans une boucle par exemple) et comprend deux valeurs, **true** (vrai) ou **false** (faux).

2.5.2. type tableau

Le type **std_logic_vector** définit un bus de N bits qui peut être vu comme un tableau d'une colonne et N lignes ; on parle alors de vecteur.

Il est également possible de définir des tableaux comprenant des grandeurs d'un type autre que **bit** et même des tableaux de plusieurs colonnes.

Dans la pratique, les outils de synthèse ne reconnaissent souvent que les tableaux de 2 dimensions, que l'on utilise pour la synthèse de mémoire.

Voici un exemple de déclaration d'un type tableau pouvant représenter le contenu d'une mémoire M mots de N bits :

```
type MEMOIRE is array (0 to M, 0 to N) of std_logic ;
```

Il est possible d'accéder à un élément particulier du tableau, soit pour lui affecter une valeur, soit pour affecter sa valeur à une autre grandeur (de même type). Imaginons que nous voulions affecter à la grandeur **s** de type **std_logic** le deuxième bit du premier mot, nous pourrions écrire :

```
s <= MEMOIRE(2,0) ;
```

Le tableau peut aussi être vu comme un "vecteur de vecteurs" :

```
type MEMOIRE is array (0 to M) of std_logic_vector (0 to N) ;
```

Cette notation est généralement préférée car elle permet un accès direct aux mots, alors que la précédente donnait un accès aux bits.

Imaginons dans l'exemple précédent que la mémoire contiennent 2 mots de 4 bits. Pour affecter la valeur hexadécimale A au premier mot, on peut écrire :

```
MEMOIRE(0) <= "1001" ;
```

ou bien encore :

```
MEMOIRE(0) <= x"A" ;
```

2.5.3. initialisation

A chaque type (qu'il soit créé par l'utilisateur ou définit dans une bibliothèque normalisée) correspond une description des valeurs possibles. Le compilateur initialise automatiquement les grandeurs à la première valeur définie (par exemple E1 pour l'exemple du signal ETAT_DE_LA_MACHINE du paragraphe sur les types énumérés, 0 pour une grandeur de type bit etc...). Si cela pose un problème, il est possible d'initialiser les grandeurs à une autre valeur lors de la déclaration par une affectation, par exemple :

```
signal ETAT_ACTUEL : ETAT_DE_LA_MACHINE := E2;
```



Initialiser une grandeur demande cependant de gérer le circuit à la mise sous tension, ce qui n'est généralement possible que par des éléments extérieurs au composant (circuit RC par exemple). Aussi l'instruction précédente, hormis associée à l'instruction **generic**, reste sans effet avec les outils de synthèse.

2.5.4. Les agrégats

L'agrégat est un moyen d'indiquer la valeur d'un type composite. Prenons l'exemple du signal Q suivant :

```
signal Q : std_logic_vector (5 downto 0);
```

Si on souhaite donner, par exemple, la valeur "101111", il est possible d'utiliser une affectation simple :

```
Q <= "101111";
```

Dans certains cas, par exemple lorsque le nombre de bits est important, ou pour ne modifier que certains bits, l'utilisation d'un agrégat simplifie l'écriture. Plusieurs formes sont possibles :

```
Q <= ('1', '0', '1', '1', '1', '1');
Q <= (5 => '1', 3 => '1', 4 => '0', 0 => '1', 2 => '1', 1 => '1');
Q <= (5 => '1', 3 downto 0 => '1', others => '0');
```

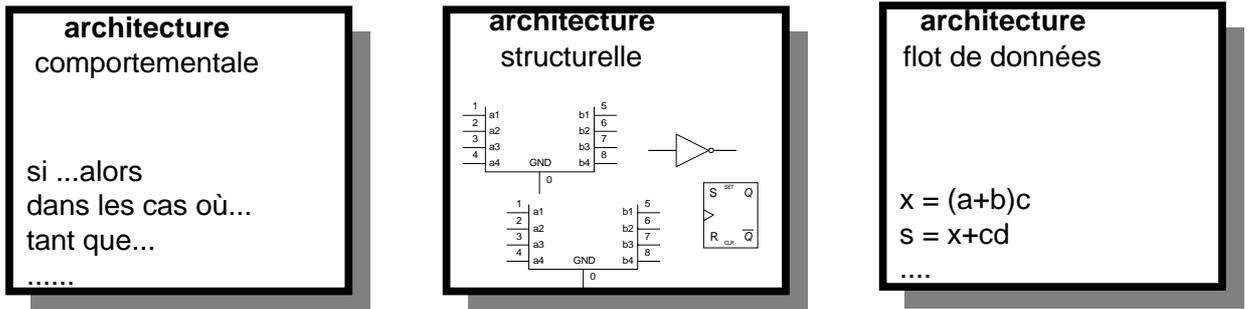
Pour mettre tout les bits du bus au même état (en haute impédance par exemple), on peut écrire :

```
Q <= ( others => 'Z' );
```

2.6. La description

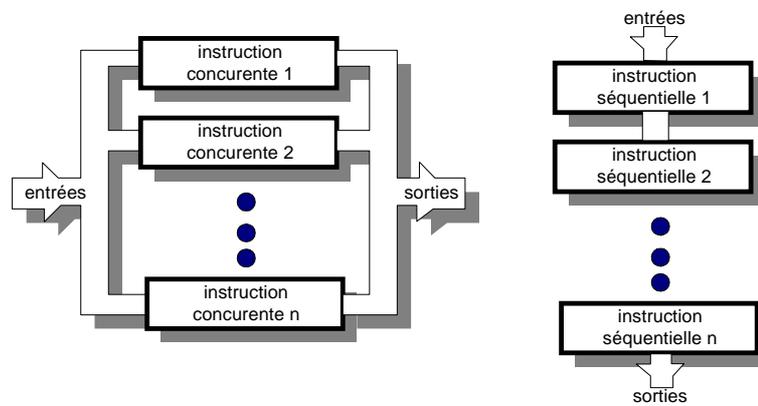
Elle permet de décrire à l'intérieur de l'architecture le comportement de l'entité associée; Elle peut être :

- comportementale (behavioral), sans référence à des structures ou des équations.
- structurelle, c'est à dire réalisée à partir de composants prédéfinis.
- de type flot de données, c'est à dire réalisée à partir d'équations booléennes.



Elle peut se faire en utilisant des instructions :

- concurrentes ; l'ordre d'écriture n'a alors pas d'importance, les opérations étant réalisées simultanément ;
- séquentielle ; un ou plusieurs événements prédéfinis déclenchent l'exécution des instructions dans l'ordre où elles sont écrites.



2.6.1. Les instructions concurrentes

Affectation simple

il s'agit d'une interconnexion entre deux équipotentiels.

```
NOM_D'UNE_GRADEUR <= VALEUR_OU_NOM_D'UNE_GRADEUR ;
```

L'affectation conditionnelle

L'interconnexion est cette fois soumise à conditions.

```
NOM_D'UNE_GRADEUR <= Q1 when CONDITION1 else
    Q2 when CONDITION2 else
    ..
    Qn ;
```

Rq : noter l'absence de ponctuation à la fin des les lignes intermédiaires.

L'affectation sélective

Suivant la valeur d'une expression, l'interconnexion sera différente.

```
with EXPRESSION select
    NOM_D'UNE_GRADEUR <= Q1 when valeur1,
    Q2 when valeur2,
    ..
    Qn when others ;
```

Remarque : noter la présence d'une virgule (et non un point virgule) à la fin des lignes intermédiaires.

La boucle for generate

Une boucle est répétée plusieurs fois en fonction d'un indice.

```
for i in MIN to MAX generate
    INSTRUCTIONS ;
end generate ;
```

La suite d'instruction peut comprendre des grandeurs indexées par i qui seront écrite par exemple D(i) et Q(i)



Ce type d'instruction peut induire une certaine confusion : il faut bien avoir à l'esprit que le VHDL décrit une structure ou le comportement d'une structure implantée dans le circuit cible, mais n'exécute pas les instructions au sens où on l'entend avec la programmation en assembleur ou en C d'un microprocesseur

Le processus

Avec les processus, une liste de sensibilité définit les signaux autorisant les actions.

Les signaux à surveiller sont énumérés dans une liste de sensibilité juste après le mot **process** :

```
process (LISTE_DE_SENSIBILITE)
    NOM_DES_OBJETS_INTERNES : « type » ; --zone facultative
begin
    INSTRUCTIONS_SEQUENTIELLES ;
end process ;
```

le déclenchement sur un front montant d'un élément de la liste de sensibilité (CLK par exemple) se fait par les instructions suivantes :

```
if (CLK 'event and CLK = '1') then
    INSTRUCTIONS ;
end if ;
```



Le déclenchement d'un process est provoqué par la variation d'une des grandeurs de sa liste de sensibilité. Les valeurs prises en compte lors du déroulement, sont celles présente avant le front actif, comme dans le cas d'une bascule (flip flop) classique, avec le respect du temps de répositionnement (set up time) et du temps de maintien (hold time) au niveau du circuit cible.

Dans les utilisations classiques du process, il est important de noter que la liste de sensibilité contient l'horloge et les entrées asynchrones. Les entrées synchrones sont en effet présent en compte sur les fronts d'horloge.

2.6.2. Les instructions séquentielles

Elles s'utilisent uniquement à l'intérieur d'un « **process** » et sont examinées dans l'ordre d'écriture.

L'affectation séquentielle

même syntaxe « **<=** » et même signification que pour une instruction concurrente ; pour les variables, on utilise « **:=** »

Le test « **if..then..elsif..else..end if** »

```
if CONDITION1 then
    INSTRUCTION1 ;
elsif CONDITION2 then
    INSTRUCTION2 ;
elsif CONDITION3 then
    INSTRUCTION3 ;
..
else INSTRUCTIONX ;
end if ;
```



Les conditions sont examinées les unes après les autres dans l'ordre d'écriture ; lorsque l'une d'elle est vérifiée, on sort de la boucle et les conditions suivantes sont pas examinées.

Le test « **case..when..end case** »

```
case EXPRESSION is
  when ETAT1 => INSTRUCTION1 ;
  when ETAT2 => INSTRUCTION2 ;
  ..
  when others => INSTRUCTIONn ;
end case ;
```

Ces instructions sont particulièrement adaptées à la description des machines d'états.

La boucle « **for..in..to..loop..end loop** »

```
for N in X to Y loop
  INSTRUCTION ;
end loop ;
```



Ce type d'instruction peut induire une certaine confusion : il faut bien avoir à l'esprit que le VHDL décrit une structure ou le comportement d'une structure implantée dans le circuit cible, mais n'exécute pas les instructions au sens où on l'entend avec la programmation en assembleur ou en C d'un microprocesseur

La boucle « **while..loop..end loop** »

```
while CONDITION loop
  INSTRUCTION ;
end loop ;
```

L'attente « **wait until** »

```
wait until EVENEMENT ;
INSTRUCTION ;
```

2.6.3.Principaux opérateurs

Les opérateurs sont définis initialement pour un certain type, par exemple l'addition pour le type entier. Cependant, grâce aux surcharges définies dans certains paquetage (voir plus loin), il sera possible d'étendre le champ d'application d'un opérateur, par exemple l'addition avec le type **std_logic_vector**. Opérateurs logiques : **and, nand, or, nor, xor, not, xnor** (**xnor** n'existe qu'en VHDL93), ils renvoient un type booléen

Opérateurs de décalage (en VHDL93 uniquement) : ils sont définis en VHDL93 uniquement et ne sont pas supportés par tous les outils de développement (même récents)

- **sll** (décalage à gauche avec remplissage par 0 à droite),
- **slr** (idem à gauche), **sra** (décalage à droite avec remplissage à droite par le LSB),
- **sra** (idem à gauche et MSB),
- **rol** (rotation à gauche, le MSB devenant LSB),
- **rор** (idem dans l'autre sens).

Les opérateurs de décalage ne sont pas supportés par tous les outils

Opérateurs relationnels : **=, /=** (non égal), **<, <=** (inférieur ou égal), **>, >=** (supérieur ou égal).

Opérateurs arithmétiques : **+, -, *, /** (addition, soustraction, multiplication, division, de nombreux compilateur ne définissant ces deux dernières opérations que pour des puissances de 2).

Opérateurs divers : **&** dit de concaténation, qui permet de juxtaposer deux littéraux (transformation de deux octets en un mot de 16 bits par exemple).

Les opérateurs ne sont définis que pour certains types ; l'addition par exemple ne peut normalement s'effectuer entre deux types `std_logic_vector`. L'utilisation de paquetages particuliers ou de fonction de conversion (voir plus loin) permettent cependant cette pratique.

2.6.4. Les attributs

Ils permettent d'accéder à une caractéristique particulière d'un signal, d'un tableau ou d'un type. Le VHDL laisse la possibilité à l'utilisateur de créer ses propres attributs. Nous ne nous intéresserons cependant qu'aux attributs prédéfinis. La syntaxe d'utilisation est :

```
NOM_DE_LA_Grandeur ' NOM_DE_L'Attribut ;
```

Pour les signaux citons l'attribut **event** qui restitue un type booléen vrai si un événement vient d'arriver sur le signal (voir le paragraphe sur les processus).

Bibliographie

Fonctions logiques élémentaires et langages comportementaux par T. Blottin sur <http://www.electron.cndp.fr>
Éléments d'analyse et de synthèse en électronique numérique par P. Cohen sur <http://www.iufm.toulouse.fr>
Syntaxe du VHDL par A. Moufflet sur <http://www.chez.com/amouf/syntaxe.htm>
Logique programmable par L. Dutrieux et D. Demigny chez Eyrolle
Langage de description VHDL par T. Blotin sur <http://www.electron.cndp.fr>
Utilisation du logiciel Max+plus II par J. Weiss sur <http://www.supelec-rennes.fr/ren/rd/etscm/base/altera/>
Cours et tp VHDL par J. Maillefert sur <http://perso.wanadoo.fr/joelle.maillefert/>
Cours VHDL sur <http://www.home.ch/~spaw4366/publica.htm>
Site constructeurs de CPLD et FPGA (altera, xilinx, atmel, vantis etc...) <http://www.altera.com> , <http://www.xilinx.com> ...

Annexe 1 : instructions les plus utilisées

| nom | description | remarques |
|----------------------------|--|--|
| Directions | in out inout buffer | - - ne peut être relu en interne - - |
| Types | integer std_logic std_logic_vector | - - permet l'état haute impédances 'Z' - permet l'état haute impédances 'Z' |
| Objets | signal variable constant | - - à l'intérieur d'un process uniquement - - |
| Notation des littéraux | bits et caractères : '0' chaînes : "001101" décimaux : 100, 1E2 hexadécimaux : X"1A" | |
| Instructions concurrentes | <= when else which select when when others for to generate process | - affectation - - - - |
| Instructions séquentielles | <= if then elsif else end if case is when end case for to loop end loop while loop end loop wait until | - := pour les variables - - - - |
| Opérateurs logiques | and, nand, or, nor, not, xor xnor (vhdl93 uniquement) | les opérateurs de décalage ne sont généralement pas reconnus par les outils de synthèse |
| Opérateurs arithmétiques | + , - , * , / | nécessite l'utilisation de fonction de surcharge ou de fonction de conversion de type pour être utilisé avec des std_logic_vector La multiplication et la division ne sont effectuées que par puissance de 2 par les outils de synthèse |
| Opérateurs relationnels | =, /=, <, <=, >, >= | attention, le symbole « inférieur ou égal » est le même que celui de l'affectation |
| Opérateurs divers | & | concaténation |
| Attributs | 'event | |

Annexe 2 : exemples de programmes

Voici quelques exemples de descriptions en VHDL. Toutes ces descriptions ont été synthétisées avec l'outil MAX+PLUS II version 9.5 de chez Altera. La compilation et la simulation se sont déroulées avec succès.

Le premier programme illustre la notion d'entité et d'architecture. Aucune ressource externe n'est déclarée, le paquetage **standard** de la bibliothèque **std** étant d'utilisation implicite. Les entrées et sorties étant de type bit (pas d'état haute impédance), il n'est pas nécessaire d'utiliser le paquetage **std_logic_1164** de la bibliothèque **ieee**.

```
entity ET is
    port (E1, E2      : in bit;
          S          : out bit );
end ET;

architecture ARCH_ET of ET is
begin
    S <= E1 and E2;
end ARCH_ET;
```

L'architecture associée à l'entité peut être écrite en utilisant des syntaxes très différentes, même si pour un cas aussi simple ce choix n'est pas très judicieux. Le tableau ci-dessous donne quelques exemples :

| | |
|--|---|
| <pre>Architecture ARCH_ET of ET is Begin S <= '1' when E1='1' and E2='1' else '0'; end ARCH_ET;</pre> | <pre>architecture ARCH_ET of ET is begin with E1='1' and E2='1' select S<= '1' when true, '0' when others; end ARCH_ET;</pre> |
| <pre>architecture ARCH_ET of ET is begin process (E1, E2) begin if E1='1' and E2='1' then S<='1' ; else S<='0' ; end if ; end process ; end ARCH_ET;</pre> | <pre>architecture ARCH_ET of ET is begin process (E1, E2) begin case E1='1' and E2='1' is when true => S<='1' ; when others => S<='0' ; end case ; end process ; end ARCH_ET;</pre> |

L'exemple suivant qui décrit un compteur, met en évidence l'utilisation

- d'une variable qui améliore la lisibilité de l'entité
- du process qui permet de décrire des systèmes synchrones
- des bibliothèques et paquetage (même si l'état haute impédance n'est pas utilisé ici)
- de l'instruction generic

```
-----
-- la bibliothèque ieee contient les paquetages dont la déclaration suit
library ieee;

--ce paquetage permet l'utilisation des types STD_LOGIC et STD_LOGIC_VECTOR
use ieee.std_logic_1164.all;

-- permet d'utiliser l'addition non signée avec le type STD_LOGIC_VECTOR
use ieee.std_logic_unsigned.all;

entity COMPTEUR is
-- N définit le nombre de bits du compteur. Initialisé à 4, il peut être modifier ailleurs
    generic (N: integer := 4);
    port (H : in std_logic; --signal d'horloge
          S : out std_logic_vector (N-1 downto 0) ); --sortie N bits
end compteur;

architecture ARCH_COMPTEUR of COMPTEUR is
-- déclaration d'un signal pour utilisation interne
    signal X : std_logic_vector(N-1 downto 0);
begin
    process (H)
    begin
        if (H'event and H = '1') then
            X <= X + 1 ;
        end if;
    end process;

-- la valeur de la variable interne est affectée à la sortie
    S <= X;
end ARCH_COMPTEUR ;
```

Le programme suivant décrit le même compteur, mais avec cette fois une entrée de validation OE des sorties (qui seront en haute impédance si elles ne sont pas validées, le comptage continuant toujours), une remise à zéro synchrone RAZS et une remise à zéro asynchrone RAZAS. Noter que c'est cette dernière qui apparaît dans la liste de sensibilité du process.

L'introduction du modulo M met en évidence l'utilisation d'une fonction (déclaré dans le paquetage std_logic_arith) qui permet la conversion de type d'un entier M vers un std_logic_vector. Cette fonction est utile pour affecter une valeur entière à X, mais n'est pas nécessaire pour les comparaisons.

Citons une petite particularité de cette description : avec le compilateur altera, l'état haute impédance 'Z' doit obligatoirement être écrit en majuscule.

```

-----
-- la bibliothèque ieee contient les paquetages dont la déclaration suit
library ieee;

--ce paquetage permet l'utilisation des types STD_LOGIC et STD_LOGIC_VECTOR
use ieee.std_logic_1164.all;

-- ce paquetage permet d'utiliser les conversions de types
use ieee.std_logic_arith.all;

-- permet d'utiliser l'addition avec le type STD_LOGIC_VECTOR
use ieee.std_logic_unsigned.all;

entity COMPTEUR2 is

-- N définit le nombre de bits du compteur. Initialisé à 4, il peut être modifié ailleurs
-- M définit le modulo du compteur. Initialisé à 5, il peut être modifié ailleurs
    generic (N: integer := 4;
            M: integer := 5 );

    port (H, RAZ_S, RAZ_AS, OE      : in    std_logic;
          S                          : out std_logic_vector (N-1 downto 0) );
-- sortie de N bits
end COMPTEUR2;

architecture ARCH_COMPTEUR2 of COMPTEUR2 is
    signal X      : STD_LOGIC_VECTOR (N-1 downto 0);
    begin

        process (H, RAZ_AS)

            begin
-- la fonction suivante convertit l'entier 0 en un std_vector_logic de N bits
                if RAZ_AS = '1' then X <= conv_std_logic_vector (0,N);
                elsif (H 'event and H = '1') then
                    if (RAZ_S='1'or X>=M ) then X <= conv_std_logic_vector (0,N);
                    else X <= X + 1 ;
                    end if;
                end if;
            end process;
            S <= X when OE='1' else (others =>'Z');
end ARCH_COMPTEUR2 ;
-----

```

L'exemple suivant décrit une mémoire RAM dont la largeur de bus de données peut être paramétrée. Le paramétrage de la largeur du bus d'adresse est plus complexe, le compilateur ne reconnaissant pas la fonction puissance.

Cette RAM étant réalisée sur un CPLD quelconque, chaque bit sera mémorisé par une bascule, ce qui est peu optimal du point de vue de la densité d'intégration. On a en contre partie la possibilité de lire la RAM pendant un cycle d'écriture, d'où les deux bus de données et les deux bus d'adresse.

La lecture peut se faire à n'importe quel moment en plaçant DATA_READ au niveau logique 1. L'écriture se fait sur le front montant de H.

Cet exemple permet de mettre en évidence l'utilisation :

- de tableaux
- d'une étiquette devant un "process" et d'une nouvelle syntaxe pour décrire la liste de sensibilité
- de nouvelles fonctions de conversion

```

-----
library ieee;
use ieee.std_logic_1164.all;
use ieee_std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity RAM is
  generic (N      : integer :=4);
  PORT( DATA_RD      : out std_logic_vector(N-1 DOWNTO 0);
        AD_READ, AD_WRITE : in  std_logic_vector(2 DOWNTO 0);
        DATA_WR      : in  std_logic_vector(N-1 DOWNTO 0);
        WR, H         : in  std_logic);
end RAM;

architecture ARCH_RAM of RAM is
  --définition d'un nouveau type
  type TYPE_MEM is array (0 to 7) of std_logic_vector(N-1 downto 0);
  signal MEM : TYPE_MEM;

begin
  -- Lecture ; la fonction conv_integer convertit l'adresse en un entier
  DATA_RD <= MEM(conv_integer(AD_READ(2 downto 0)));

  -- écriture synchrone (étiquette afin de repérer le "process" facilement)
  ECRITURE : process
  begin
    wait until H'event and H='1';
    if (WR = '1') then
      MEM(conv_integer(AD_WRITE(2 downto 0))) <= DATA_WR;
    end if;
  end process;
end ARCH_RAM;
-----

```