

UTILISATION DE VHDL

l'outil de synthèse
comportementale.



IUT de Vélizy

2001

TABLE DES MATIERES.

Cours

HISTORIQUE.....	3
PRESENTATION DE VHDL.....	4
EXEMPLE.....	4
LES OPERATEURS SEQUENTIELS.....	5
<i>L'Instruction IF.....</i>	<i>5</i>
<i>Syntaxe :.....</i>	<i>5</i>
<i>Exemple :.....</i>	<i>5</i>
<i>L'Instruction CASE.....</i>	<i>6</i>
<i>Syntaxe :.....</i>	<i>6</i>
<i>Exemple :.....</i>	<i>6</i>
<i>L'Instruction FOR.....</i>	<i>7</i>
<i>Syntaxe :.....</i>	<i>7</i>
<i>Exemple :.....</i>	<i>7</i>
<i>L'Instruction WHILE.....</i>	<i>7</i>
<i>Syntaxe :.....</i>	<i>7</i>
<i>Exemple :.....</i>	<i>7</i>
<i>L'Instruction PROCESS.....</i>	<i>8</i>
Syntaxe.....	8
TABLEAU DES OPERATEURS.....	8
COMBINATOIRE OU SEQUENTIEL.....	9
<i>Exemple de multiplexeur 4 vers 1.....</i>	<i>9</i>
Avec des Instructions combinatoires.....	9
Instructions Séquentielles.....	10
STRUCTURE DES COMPOSANTS.....	10
SYNCHRONISATION SUR L'HORLOGE.....	11
<i>Les signaux asynchrones.....</i>	<i>12</i>
<i>Les signaux synchrones.....</i>	<i>13</i>
LES CHAINES DE DEVELOPPEMENT VHDL.....	13
LES BIBLIOTHEQUES.....	14
EXEMPLE DE COMPTEUR.....	14
<i>Bibliothèques standards.....</i>	<i>14</i>
LES TYPES ET LES SOUS-TYPES.....	17
LES FONCTIONS MATHEMATIQUES.....	18
SYNTHESE DES COMPOSANTS.....	19
LE SYNCHRONISME.....	19
LES TEMPS DE PROPAGATION.....	20
LE BROCHAGE DU COMPOSANT.....	20
Exemple	
LE COMPTEUR BINAIRE UNIVERSEL :	21

L'outil VHDL.

Historique.

VHDL est un langage très utilisé dans le monde industriel. Né dans le début des années 80 à la suite d'un projet militaire américain, il est devenu un outil indispensable pour tous les concepteurs.

Le principe de ce langage est radicalement différent de celui de ses prédécesseurs, en effet, il ne se base plus sur une description du contenu d'une fonction mais sur la définition de son comportement. En clair, on ne définit plus un système électronique en terme de portes logiques séquentielles ou combinatoires, mais en terme de signaux entrants et de signaux sortants dans une fonction dont la composition n'est pas définie en terme matériel.

Commençons donc par un petit historique de VHDL. Dans les années 80, l'évolution trop rapide des technologies électroniques impose trop de remise en cause des procédures de conceptions, ce qui rend l'armée américaine tributaire des fabricants de composants. Le département de la défense Américain initie alors un projet qui a pour but, outre de développer des composants rapides et résistants, un langage de conception des circuits numériques. Lancé sous le nom de projet VHSIC (Very High Speed Integrated Circuit), la partie langage devient VDHL (VHSIC Hardware Descriptive Language). On peut grâce à ce langage créer des fonctions électroniques avec une totale abstraction matérielle.

En 1987, la norme IEEE 1076 B fige VHDL dans sa version définitive et l'impose comme nouvelle norme internationale.

Depuis, VHDL a continué son évolution et la description analogique vient d'y être intégrée.

Mais VHDL ne se résume pas à une simple définition comportementale des circuits, il permet bien des choses comme par exemple, le mariage du matériel avec le logiciel, c'est à dire que VHDL ne fait pas de différence entre un logiciel (par exemple une fonction de test) et une fonction électronique. Il peut de plus intégrer les stimuli, les retards et les fonctions spéciales, tels que les vecteurs de test des ASIC au sein même du silicium ou alors les réserver à la phase de test en atelier.

Il faut noter que VHDL est un langage qui est basé sur ADA (un langage assez ancien, très répandu dans les universités et qui fut réalisé par des français...

Présentation de VHDL.

En VHDL, les fonctions sont présentées sous la forme d'une paire de fonctions, d'une part une entité (**ENTITY**) qui décrit la boîte dans laquelle on va implanter la fonction, et d'autre part, l'architecture (**ARCHITECTURE**) qui est la description des fonctionnalités de la fonction.

L'entité, c'est le nom de la fonction, (une boîte vide, sans entrée ni sortie). C'est dans l'entité que se définissent les ports (**PORT**) d'entrée (**IN**) et de sortie (**OUT**), bidirectionnel (**INOUT**) ou "tout en un" (**BUFFER**) et les constantes (**GENERIC**).

En VHDL, comme en langage C ou en ADA, les lignes doivent se terminer par un point virgule. On utilise aussi des fonctions **END** (et aussi des fonctions **BEGIN**) pour marquer respectivement la fin ou le début d'une fonction.

Exemple.

ENTITY	et	IS	On fabrique une boîte appelée <i>et</i>
PORT (On définit les ports de la boîte.
		a,b: IN BIT ;	On crée 2 entrées <i>a</i> et <i>b</i> qui sont des bits
		s: OUT BIT ;	On crée 1 sortie <i>s</i> qui est un bit.
END	et ;		Fin de la description de la boîte et.
ARCHITECTURE simpliste	OF et IS		On commence à décrire le fonctionnement de la boîte <i>et</i> .
BEGIN			On marque le début de la description
		s <= a AND b ;	On écrit comment <i>s</i> se comporte vis à vis de <i>b</i> et de <i>a</i> .
END simpliste ;			On marque la fin de la description.

Cet exemple vous montre comment décrire une porte ET. Cette porte, je l'ai décrite comme A **et** B (opération logique directe), j'aurais pu la décrire de la façon suivante :

```
s <= '1' WHEN (a = '1' AND b = '1') ELSE '0';
```

L'exemple précédent décrit une opération combinatoire, il faut noter que si VHDL sait décrire correctement des fonctions combinatoires, les circuits programmables sont généralement prévus pour des utilisations séquentielles. VHDL dispose donc de fonctions spécialement conçues pour le séquençage des commandes.

Les opérateurs séquentiels.

L'Instruction **IF**.

Syntaxe :

L'instruction IF permet de choisir entre 2 types d'actions selon le résultat d'un test dont le résultat est soit vrai, soit faux.

```
IF (<variable(s)> <opérateur(s)> <valeur(s)>) THEN
    <instruction 1> ;
    ...
    <instruction n> ;
ELSE
    <instruction n+1> ;
    ...
    <instruction n+m> ;
END IF ;
```

On peut aussi faire abstraction du ELSE, ou encore n'avoir qu'une seule action. Il existe enfin un prolongement de la fonction IF, la fonction ELSIF, combinaison de ELSE et de IF.

Exemple :

```
IF (a=1) THEN etat <= 10;
ELSIF (b='1') THEN
    IF (c='1' AND d='0') THEN
        etat <= 0;
    ELSE
        etat <= etat +1;
    END IF ;
END IF ;
```

Attention, Si une condition est vraie, elle est exécutée puis on sort de l'instruction IF, même si d'autres conditions sont elles aussi vraies

De même on n'est pas contraint de traiter tous les états d'une variable. On n'est donc pas forcé d'avoir un ELSE pour chaque IF.

L'Instruction **CASE**.

Syntaxe :

L'instruction CASE permet de sélectionner parmi plusieurs alternatives en fonction non plus du résultat booléen d'un test, mais à partir des valeurs d'une variable.

```
CASE <variable> IS
  WHEN <valeur 1> => <suite d'instruction 1> ;
  ...
  WHEN <valeur n> => <suite d'instruction n> ;
END CASE;
```

Pour que l'instruction CASE s'effectue correctement, il faut que toutes les valeurs que peut prendre la variable soient définies. Comme c'est souvent difficile, on a recours à l'instruction **WHEN OTHER** (dans tous les autres cas).

Exemple :

```
CASE etat IS
  WHEN 9 => etat <= 0 ;
  WHEN OTHER => etat <= etat +1 ;
END CASE;
```

On peut aussi combiner plusieurs valeurs dans un WHEN, pour cela, on utilise le symbole du OU logique | (Ex: WHEN <valeur 1>|<valeur 2> => <action 1> ;), on peut aussi lui dire d'exécuter la même action pour toutes les valeurs comprises entre 2 bornes bien définies (Ex: WHEN <valeur 1> TO <valeur 2> => <action 1> ;).

L'Instruction **FOR**.

Syntaxe :

L'instruction FOR permet de créer des boucles s'incrémentant (par pas de 1 exclusivement) selon l'évolution d'une variable interne. Cette variable (qui ne doit être composée que d'une seule lettre) n'est pas à définir, il suffit donc d'écrire :

```
FOR <variable> IN <début> TO <fin> LOOP  
    <instruction 1>  
    ....  
    <instruction n>  
END LOOP
```

Exemple :

```
FOR n IN 0 TO 9 LOOP  
    etat <= n ;  
END LOOP ;
```

L'Instruction **WHILE**.

Syntaxe :

L'instruction WHILE est le pendant de l'instruction FOR, toutefois, on devra privilégier les boucles FOR aux boucles WHILE vis à vis de la synthèse de VHDL. Les boucles WHILE peuvent être réalisées avec des variables sous forme de chaînes de caractères. Toutefois ces dernières doivent être prédéfinies.

```
WHILE <variable> <opérateur> <constante> LOOP  
    <instruction 1> ;  
    .....  
    <instruction n> ;  
END LOOP ;
```

Exemple :

```
WHILE etat < 10 LOOP  
    etat <= etat + 1 ;  
    IF etat = 10 THEN etat <= 0 ;  
    END IF ;  
END LOOP ;
```

L'Instruction PROCESS.

L'instruction **PROCESS** n'est pas, comme les instructions précédentes, fonctionnelle, elle n'assure aucun service identifiable. Toutefois elle est indispensable à l'utilisation d'instructions séquentielles.

Prenons l'exemple d'une bascule D à déclenchement sur front, la présence d'un front actif sur l'entrée d'horloge conditionne la lecture de D et sa recopie sur Q. Donc c'est suite à un front, donc suite à un changement d'état de l'horloge que les entrées sont prises en compte.

L'instruction PROCESS joue un peu ce rôle de déclencheur puisqu'elle permet de définir quels sont les signaux qui déclenche l'exécution de la fonction. Ces signaux sont listés entre parenthèse à la suite de la commande PROCESS. On nomme ces signaux les réceptivités du PROCESS. L'absence de réceptivité veut dire que tous les signaux sont dans la liste de réceptivité.

Syntaxe.

```
PROCESS (<var1>,<var2>,...)
BEGIN
    <instruction 1>;
    ...
    <instruction n>;
END PROCESS;
```

Tableau des Opérateurs.

Classes	Opérateurs	Types	Resultats
Logiques	AND OR NAND NOR XOR	booléen	booléen
relationnels	= /= < <= > >=	tous	booléen
Additifs	+ -	numérique	numérique
	& (concaténation)	tableau	tableau
Signe	+ -	numérique	numérique
Multiplicatifs	* /	numérique ®	numérique
	mod (modulo) rem (reste)	entier ®	entier
Divers	Not	booléen	booléen
	abs (valeur absolue)	numérique	numérique
	** (puissance)	numérique ®	numérique
Rotatifs	sll srl sla sra rol ror	booléen	booléen
Affectants	:= <=	tous	tous

Attention les marques ® signifient que des restrictions interviennent sur ces opérateurs

Combinatoire ou séquentiel.

Bien que les composants ne soient pas forcément adaptés aux fonctions combinatoires, on essaye autant que possible d'utiliser des opérateurs non séquentiels pour réaliser une fonction purement combinatoire. On essaiera donc de ne pas utiliser de fonction IF ou CASE pour une fonction qui n'utilise pas, électroniquement, de bascules.

Exemple de multiplexeur 4 vers 1.

```
ENTITY mux41 IS
  PORT (
    entrée: IN BIT_VECTOR (3 DOWNTO 0);
    sel: IN BIT_VECTOR (1 DOWNTO 0);
    s: OUT BIT);
END mux41 ;
```

Un BIT_VECTOR est un bus composé de n fils (n BIT), on doit définir sa taille. Dans notre exemple, sa taille est 4 (de 3 à 0 soit 4 fils). On décrit en général les signaux en descendant (DOWNTO) pour respecter le sens commun d'écriture (les poids fort à gauche et les poids faibles à droite).

Avec des Instructions combinatoires

```
ARCHITECTURE combinatoire1 OF mux41 IS
BEGIN
  s <= entree(0) WHEN (sel="00") ELSE
        entree(1) WHEN (sel="01") ELSE
        entree(2) WHEN (sel="10") ELSE
        entree(3) WHEN (sel="11");
END combinatoire1 ;
```

```
ARCHITECTURE combinatoire2 OF mux41 IS
BEGIN
  WITH sel SELECT
    s <= entree(0) WHEN "00",
        entree(1) WHEN "01",
        entree(2) WHEN "10",
        entree(3) WHEN "11";
END combinatoire2;
```

Ces 2 fonctions sont totalement équivalentes, elles permettent de décrire le comportement d'un multiplexeur de façon combinatoire. Le compilateur va donc créer pour s une équation identique pour les 2 architectures.

$$s = e_0 \cdot \overline{sel_0} \cdot \overline{sel_1} + e_1 \cdot sel_0 \cdot \overline{sel_1} + e_2 \cdot \overline{sel_0} \cdot sel_1 + e_3 \cdot sel_0 \cdot sel_1$$

Instructions Séquentielles

```
ARCHITECTURE sequentielle1 OF mux41 IS
BEGIN
  PROCESS (sel, entree)
  BEGIN
    CASE sel IS
      WHEN "00" => s <= entree(0);
      WHEN "01" => s <= entree(1);
      WHEN "10" => s <= entree(2);
      WHEN OTHERS => s <= entree(3);
    END CASE;
  END PROCESS;
END sequentielle;
```

```
ARCHITECTURE sequentielle2 OF mux41 IS
BEGIN
  PROCESS (sel, entree)
  BEGIN
    IF      (sel="00") THEN s <= entree(0);
    ELSIF  (sel="01") THEN s <= entree(1);
    ELSIF  (sel="10") THEN s <= entree(2);
    ELSIF  (sel="11") THEN s <= entree(3);
    END IF;
  END PROCESS;
END sequentielle2;
```

L'utilisation de fonctions séquentielles pour résoudre des problèmes combinatoires est assez dangereuse, primo, une mauvaise définition des réceptivité du PROCESS peut créer un dysfonctionnement. Secundo, l'effort de synthèse augmente et enfin tertio, la taille du code source augmente aussi.

Structure des composants.

Les CPLD (Complex Programmable Logic Devices) que nous utilisons sont des composants programmables qui se composent de 256 fonctions (macrocells). Ces fonctions peuvent être soit combinatoires, soit séquentielles, or pour ces dernières, il faut impérativement respecter un temps de pré positionnement (setup time) précédent le front de l'horloge. Malheureusement, l'addition des temps de propagation des portes combinatoires peut amener un retard tel qu'il y a risque de non respect de cette règle, donc passage à des états métastables.

Comme il est difficile de connaître (surtout avec VHDL) le temps de propagation lié à une fonction, le plus sûr moyen de respecter les temps de setup, consiste à utiliser des fonctions séquentielles (donc resynchronisés par une horloge) le plus régulièrement possible.

Synchronisation sur l'horloge.

```

ENTITY    d_flip_flop    IS
    PORT (    clk,d: IN BIT ;
              q:      OUT BIT) ;

END        d_flip_flop ;

ARCHITECTURE sequentielle1 OF d_flip_flop IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL (clk = '1' AND clk'event) ;
            q <= d ;
    END PROCESS;

END sequentielle1;

```

La commande **WAIT UNTIL** permet d'attendre qu'un événement définit se produise pour lancer l'exécution de la fonction. Dans notre cas, on attendra que l'horloge soit à "1" et qu'il y ait eut un front sur l'horloge (clk'event)

On peut aussi utiliser la commande **RISING_EDGE(clk)** (front montant) pour détecter un front montant (Attention, cette commande n'est pas reconnue par tous les logiciels synthétisant du VHDL).

```

ARCHITECTURE sequentielle2 OF d_flip_flop IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF (RISING_EDGE(clk))THEN ;
            q <= d ;
        END PROCESS;
    END sequentielle2;

```

Par contre, on ne peut pas utiliser les réceptivités de la commande **PROCESS** pour détecter des événements d'horloge, le principe de fonctionnement de la commande **PROCESS** étant de "lancer" l'exécution du programme, pas de détecter les évolutions d'un signal. Ainsi le programme suivant

```

...
PROCESS (clk)
BEGIN
    IF (clk='1') THEN ...
...

```

ne permet pas de détecter les fronts, c'est donc un fonctionnement en **LATCH** (verrou) des bascules.

Le problème s'accroît encore lorsqu'on utilise des fonctions qui combinent des portes séquentielles et des portes combinatoires. Le mélange de ces fonctions impose de définir pour chaque signal si ce dernier est synchrone (l'horloge prend le pas sur lui) ou asynchrone (le signal prend le pas sur l'horloge).

Les signaux asynchrones.

Les signaux asynchrones sont assez dangereux, ils peuvent présenter (si la fonction qui les génère est très complexe), un risque de non respect des temps de pré positionnement. Il faut donc être circonspect quand à leur usage.

```
ENTITY    d_edge_with_clear IS
  PORT (   clk, d, reset :    IN BIT ;
          q              :    OUT BIT) ;
END       d_edge_with_clear ;

ARCHITECTURE asynchrone OF d_edge_with_clear IS
BEGIN
  PROCESS (clk , reset)
  BEGIN
    IF reset = '1' THEN q<='0' ;
    ELSIF (clk'event AND clk = '1') THEN q<=d ;
    END IF ;
  END PROCESS ;
END asynchrone ;
```

Tous les signaux asynchrones, ainsi que l'horloge doivent faire partie de la liste des réceptivités du PROCESS. De plus, le test de la fonction reset avant le test de l'état de l'horloge, indique que le signal RESET est asynchrone (il ne dépend pas de l'horloge).

Un signal asynchrone prend le pas sur l'horloge, on doit donc faire en sorte que le signal asynchrone le plus prioritaire soit le premier à être testé...

Certains synthétiseurs sont quasiment incapable de traiter un signal asynchrone. Donc on doit essayer au maximum de s'affranchir de ce genre de fonction. Il est donc préférable d'utiliser surtout des architectures synchrones.

Les signaux synchrones.

Les signaux synchrones sont intégralement pilotés par l'horloge. Seul un état actif de cette dernière permet de prendre en compte l'état des entrées. Il est donc très peu probable qu'un signal, suite à son passage au travers de fonctions combinatoires, arrive à ne pas respecter les règles de pré positionnement.

```
ENTITY    d_edge_with_clear IS
  PORT (   clk, d, reset :   IN BIT ;
          q               :   OUT BIT) ;
END       d_edge_with_clear ;

ARCHITECTURE synchrone OF d_edge_with_clear IS
BEGIN
  PROCESS (clk)
  BEGIN
    IF (clk'event AND clk = '1') THEN
      IF (reset = '1') THEN q<='0' ;
      ELSE q<=d ;
      END IF;
    END IF;
  END PROCESS ;
END synchrone ;
```

Dans une architecture synchrone, il n'y a que le signal d'horloge dans la liste des réceptivités du PROCESS. De plus seul la détection d'un état actif de l'horloge déclenche le test des fonctions de reset.

Les chaînes de développement VHDL.

Une chaîne de développement VHDL inclus différents éléments, un éditeur de code (souvent un éditeur de texte), un compilateur (vérifications syntaxiques), un synthétiseur (décomposition en fonctions logiques compatible avec la cible), un "FITTER" (outil de programmation des composants). A cela s'ajoute en général des simulateurs.

On définit deux types de simulation, d'une part la simulation fonctionnelle (le code répond-il aux attentes du cahier des charges au niveau de la fonctionnalité), d'autre part la simulation temporelle (la réalisation respectera t-elle les temps de réaction, propagation et maintient imposés par le composant).

Le rôle du synthétiseur est de transformer du code VHDL simple, "généraliste", en un code dédié au produit que l'on souhaite réaliser. Pour établir cette liaison, il utilise des définitions liées au produit cible et placées dans des bibliothèques de fonctions.

Les bibliothèques.

Les bibliothèques permettent de stocker des éléments, mais aussi de définir des fonctions générales. Généralement, on utilise certaines bibliothèques pour le développement des fonctions en VHDL. Mais on peut aussi se créer sa propre bibliothèque, dans laquelle, on stocke ses composants, avec la possibilité de rappeler plus tard la fonction. En langage C++, ces bibliothèques seraient l'équivalent des classes de fonctions.

On notera que certaines bibliothèques sont très importantes pour le développement en VHDL, il s'agit, par exemple, des bibliothèques IEEE et WORK, et en particulier des fonctions `std_logic_1164` (définition de la norme IEEE 1164) et de `std_logic_signed` (calculs sur des signaux booléens).

Exemple de compteur.

Dans l'exemple qui suit, nous allons réaliser un compteur modulo 10, avec une entrée horloge active sur front montant, une entrée RAZ active à l'état bas, une entrée de chargement parallèle active à l'état bas, une entrée de validation de comptage, active à l'état haut et une sortie de retenue active à l'état haut.

Bibliothèques standards.

```
LIBRARY IEEE, WORK ;
USE IEEE.std_logic_1164.ALL ;
USE IEEE.std_logic_signed.ALL ;
```

```
ENTITY compteur_10 IS
```

```
    PORT      (      clk, en, clr_b, load_b : IN STD_LOGIC ;
                   data : IN STD_LOGIC_VECTOR (n DOWNTO 0) ;
                   q : BUFFER STD_LOGIC_VECTOR (n DOWNTO 0) ;
                   rco : OUT STD_LOGIC      ) ;
```

```
END compteur_10 ;
```

On décrit les signaux entrant et sortant comme des `STD_LOGIC`, c'est à dire des signaux électroniques composés non plus de 3 états ('0' pour le zéro logique, '1' pour le un logique ou 'X' pour l'état logique indéterminé), mais d'un ensemble de 9 états. Les `STD_LOGIC_VECTOR` sont des bus de `STD_LOGIC`.

1 0	états logiques 1 et 0 (forts)	X	état logique indéterminé (fort)
H L	états logiques 1 et 0 (résistifs)	W	état logique indéterminé (résistif)
U	état non initialisé	Z	état haute impédance
-	état indifférent (on s'en fout).		

```

ARCHITECTURE synchrone OF compteur_10 IS
BEGIN
  PROCESS
  BEGIN
    WAIT UNTIL clk = '1' ;
    IF clr_b = '0' THEN q <= "0000" ;
    ELSIF load_b = '0' THEN q <= data ;
    ELSIF en = '1' THEN
      CASE q IS
        WHEN "1001" =>
          q <= "0000" ;
          rco <= '1' ;
        WHEN OTHERS =>
          q <= q + 1 ; -- ← c'est ici qu'on utilise
          rco <= '0' ; -- la librairie STD_LOGIC_SIGNED
      END CASE ;
    END IF
  END PROCESS ;
END synchrone ;

```

Bien entendu, dès le démarrage de la session, une librairie est chargée d'office, c'est la librairie WORK (souvent le répertoire de travail du compilateur et du simulateur). On peut toutefois réorienter cette librairie en modifiant les options de compilation.

Pour pouvoir sauvegarder un composant dans la bibliothèque WORK, il faut définir une boîte (**PACKAGE**) dans laquelle on va placer l'empreinte du composant (**COMPONENT**). Cette boîte est "unique", on ne peut pas la réutiliser. Une fois un composant placé dedans, on ne peut rien y ajouter. De plus, le nom du composant et tous ses ports doivent rester les mêmes.

On doit aussi ajouter les bibliothèques et les fonctions utilisées pour la définition des ports du composant. Par exemple, notre compteur peut être mis en mémoire en utilisant les commandes suivantes :

```

LIBRARY IEEE, WORK ;
USE IEEE.std_logic_1164.ALL ;

```

```

PACKAGE my_cmpt IS
  COMPONENT compteur_10
    PORT ( clk, en, clr_b, load_b : IN STD_LOGIC ;
          data : IN STD_LOGIC_VECTOR (n DOWNTO 0) ;
          q : OUT STD_LOGIC_VECTOR (n DOWNTO 0) ;
          rco : OUT STD_LOGIC ) ;
  END COMPONENT ;
END my_cmpt ;

```

Il faudra enfin rajouter la ligne USE WORK.my_cmpt.ALL; dans la liste des bibliothèques qui servent à définir l'entité du composant.

On peut dès lors utiliser le composant que l'on vient de créer dans un autre composant. Par exemple, si l'on souhaite réaliser un compteur modulo 100 en code DCB, on peut donc le décrire en réutilisant le compteur modulo 10.

```
LIBRARY IEEE, WORK;
USE IEEE.std_logic_1164.ALL;
USE WORK.my_cmpt.ALL;

ENTITY cmpt100 IS
    PORT (
        h, reset, charge, valid : IN STD_LOGIC;
        retenue : OUT STD_LOGIC;
        entree : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        sortie : BUFFER STD_LOGIC_VECTOR (7 DOWNTO 0));
END cmpt100;
```

On utilise des bus de 8 bits pour le compteur pour concaténer les 2 bus de 4 bits qui servent à compter de 0 à 9 en DCB.

```
ARCHITECTURE descriptive OF cmpt100 IS
    SIGNAL ripple : STD_LOGIC;
BEGIN
    cmpt1 : compteur_10
        PORT MAP (h, reset, charge, valid,
                ripple,
                entree (3 DOWNTO 0),
                sortie (3 DOWNTO 0));

    cmpt2 : compteur_10
        PORT MAP (
            h, reset, charge, ripple,
            retenue,
            entree (7 DOWNTO 4),
            sortie (7 DOWNTO 4));
END descriptive;
```

On doit, pour cascader 2 composants synchrones utiliser une retenue (rco) créée par le compteur des poids faibles (cmpt1) pour alimenter l'entrée de validation (en) du compteur des poids forts (cmpt2). Or tout signal utilisé doit être décrit, soit, s'il est sortant ou entrant, dans l'entité, soit, s'il est interne, comme un SIGNAL, c'est à dire une "variable interne"...

Dans l'architecture que je viens de décrire, le compteur modulo 100 est réalisé par association de 2 compteurs modulo 10. On utilise donc la commande USE WORK.my_cmpt.ALL pour créer un lien entre le mot compteur_10 et l'entité compteur_10, qui est rangée dans la mémoire.

Enfin on doit associer des signaux locaux (h, reset, charge, valid, retenue, entree, sortie et ripple) avec les signaux définis lors de la création de compteur_10 (clk, raz_b, load_b, en, rco, data, q). Pour cela on utilise la commande PORT MAP qui permet, dans le nouveau composant, d'énumérer dans le même ordre que celui de compteur_10, la liste des signaux décrit dans le PACKAGE.

On parle en général d'un travail hiérarchique (c'est à dire pyramidal) où l'on commence par décrire des fonctions élémentaires pour les assembler ensuite dans des niveaux de plus en plus élevés.

Les Types et les sous-types.

Sous VHDL on peut créer des types (**TYPE**) énumérés, c'est à dire en énumérant l'ensemble des valeurs que peut prendre une variable. On peut, par exemple, créer un type jour qui serait composé de 7 valeurs (lundi, mardi, mercredi, jeudi, vendredi, samedi et dimanche). Ce type doit être défini dans un PACKAGE.

```
PACKAGE my_type IS
    TYPE jour IS (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche);
    TYPE mois IS (jan, fev, mar, avr, mai, jui, jul, aou, sep, oct, nov, dec);
END my_type;
```

Dès lors, l'utilisation du type "jour" peut se faire de façon directe. On peut par exemple dans une instruction CASE ne pas utiliser l'instruction WHEN OTHERS, puisqu'on peut lister l'ensemble des cas applicables.

```
LIBRARY IEEE,WORK;
USE IEEE.std_logic_1164.ALL;
USE WORK.my_type.ALL;

ENTITY calendrier IS
    PORT (
        clk : IN std_logic;
        n_day : BUFFER jour :=lundi);
END calendrier;
```

```
ARCHITECTURE calcul OF calendrier IS
BEGIN
  PROCESS (clk)
  BEGIN
    IF (rising_edge(clk)) THEN
      CASE n_day IS
        WHEN lundi => n_day <= mardi;
        WHEN mardi => n_day <= mercredi;
        WHEN mercredi => n_day <= jeudi;
        WHEN jeudi => n_day <= vendredi;
        WHEN vendredi => n_day <= samedi;
        WHEN samedi => n_day <= dimanche;
        WHEN dimanche => n_day <= lundi;
      END CASE;
    END IF;
  END PROCESS;
END calcul;
```

On peut aussi pour des raisons de convenance utiliser des sous-types, c'est à dire des surnoms que l'on donne aux signaux. Comme pour les types, les sous-types doivent être définis dans un package. Par exemple, on peut créer des sous-types bus4 de la façon suivante :

```
PACKAGE my_subtype IS
  SUBTYPE bus4 IS std_logic_vector (3 DOWNT0 0);
  SUBTYPE bus3 IS std_logic_vector (2 DOWNT0 0);
  SUBTYPE bus2 IS std_logic_vector (1 DOWNT0 0);
END my_subtype;
```

Les fonctions mathématiques.

On peut ne pas limiter l'utilisation de VHDL à des fonctions logiques, les fonctions "analogiques" (ou plutôt mathématiques) sont elles aussi utilisables. Mais par contre, elles ne sont pas synthétisables (ou alors difficilement). Rien n'empêche toutefois de développer des petits programmes en VHDL utilisant des fonctions mathématiques évoluées.

On doit alors utiliser des bibliothèques contenant des fonctions mathématiques, comme par exemple IEEE.math_real.ALL;. On peut par exemple décrire une fonction mathématique :

```
LIBRARY IEEE;
USE IEEE.math_real.ALL;
```

```

entity Sinus is
  port (
    CLK: in bit;
    Freq: integer;
    Y: out real);
end Sinus;

architecture Generateur of Sinus is
  signal X : real :=0.0;
  signal DELTA : real;
begin
  DELTA <= MATH_PI*real(Freq)/1000.0;
  process (clk)
  begin
    if clk='1' and clk'event then
      X <= "MOD"( X + delta , 2.0*MATH_PI);
      Y <= SIN(X);
    end if;
  end process;
end Generateur;

```

On utilise les signaux DELTA et X pour créer respectivement, l'incrément de phase (DELTA) et la valeur de l'angle (X). On utilise enfin la fonction MOD pour limiter l'amplitude de X à l'intervalle $0 - 2\pi$ (MOD veut dire modulo).

Ainsi, à chaque front actif de l'horloge on ajoute à X (qui a été initialisé à 0 au début) la valeur de DELTA $(\frac{\pi \times F}{1000})$, où F est la fréquence du signal), puis on effectue $Y=\sin(X)$.

Synthèse des composants.

L'utilisation de VHDL peut avoir pour but la création de composants électroniques, il faut alors respecter quelques règles simples de création.

Le synchronisme.

Tous (ou presque) les composants sont synchrones, cela veut dire qu'ils n'ont qu'une et une seule horloge, on ne peut donc pas en principe détecter les fronts d'un signal autre que l'horloge. On doit donc se contenter de détecter l'état des signaux.

On ne peut pas non plus avoir de temporisation ou de délais, comme par exemple avec l'instruction **WAIT FOR** (la syntaxe de cette instruction est WAIT FOR x ns où x est une valeur numérique et où ns peut être indépendamment remplacée par ps, us, ms ou s). En effet on ne peut pas synthétiser un retard, d'une part à cause de l'absence de définition de la référence temporelle (quelle sera la fréquence exacte de l'entrée d'horloge puisqu'elle n'est pas connue du synthétiseur),

et d'autre part, par l'absence de liaison entre ce retard et l'horloge (c'est en général un signal autre que l'horloge qui est soumis au retard, donc un signal pas forcément synchrone qui doit subir un retard non lié à l'horloge. En d'autres termes, il faut réaliser une fonction de retard sans exploiter les temporisations de VHDL puisqu'elles ne sont pas synthétisables.

Les temps de propagation.

Un composant électronique induit un temps de propagation entre la modification de l'état de ses entrées et l'apparition d'un état de sortie. Le cas particulier des composants programmables en est la plus flagrante expression. Un composant programmable est une suite de fonction. On a pu voir lors de la justification du synchronisme des fonctions, qu'il existe des risques à l'utilisation de fonctions combinatoires (non respect des temps de pré positionnement ou de maintien), mais le simple fait de programmer un composant amène à cascader des fonctions donc à incrémenter les retards.

On définit donc des éléments comme par exemple, le chemin critique, qui permettent de donner les limites d'utilisation de la fonction. La fréquence maximum d'utilisation du composant est alors obligatoirement inférieure à $\frac{1}{2 \cdot t_c}$, où t_c est la durée du chemin critique.

Le brochage du composant.

Les composants programmables ont ceci de particulier, que des changements même minime du code ont des effets dévastateur sur le résultat de la compilation, en particulier sur le brochage du composant. Pour éliminer ces aléas, qui peuvent être assez gênant si le composant doit être réimplanté sur une carte préexistante, on peut définir des attributs pour que "le compilateur" place ces signaux sur les broches indiqués.

Par exemple, **Attribute PIN_NUMBERS** of clk : SIGNAL is "19";

Mais on peut aussi donner d'autres attributs, pas forcément compréhensible par le compilateur (ce qui a peu d'intérêt généralement), à destination d'autres fonctions, qui vont suivre la compilation.

Exemples :

Le compteur binaire universel :

Le compteur universel est un compteur totalement synchrone dont la taille est fonction d'un entier définissant le nombre de bits de sortie.

Ce compteur sera placé dans le package : cmpt_package.

```
library IEEE,WORK;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_signed.ALL;

PACKAGE cmpt_package IS
    COMPONENT compteur
        GENERIC ( n : INTEGER);
        PORT(      clk, raz, load, en : in std_logic;
                 rco : out std_logic;
                 Pin : in std_logic_vector ((n-1) downto 0);
                 Qout : out std_logic_vector ((n-1) downto 0));
    END COMPONENT;
END cmpt_package;

library IEEE,WORK;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_signed.ALL;
USE WORK.cmpt_package.ALL;

ENTITY compteur IS
    GENERIC( n : INTEGER :=3);
    PORT(      clk, raz, load, en : in std_logic;
             rco : out std_logic;
             Pin : in std_logic_vector ((n-1) downto 0);
             Qout : out std_logic_vector ((n-1) downto 0));
End compteur;
```

```
ARCHITECTURE test OF compteur IS
SIGNAL etat : std_logic_vector ((n-1) DOWNTO 0);
BEGIN
    PROCESS (clk)
    BEGIN
        IF (rising_edge(clk)) THEN
            IF (raz = '0') THEN
                etat <= (OTHERS=>'0');
            ELSE
                IF (load = '0') THEN
                    etat <= Pin;
                ELSE
                    IF (en = '1') THEN
                        etat <= etat + 1;
                    END IF;
                END IF;
            END IF;
        END IF;
    END PROCESS;

    Qout <= etat;

    PROCESS (etat, en)
    BEGIN
        IF (etat = (2**n)-1 AND en = '1') THEN
            RCo <= '1';
        ELSE
            RCo <= '0';
        END IF;
    END PROCESS;
END test;
```

L'entier n définit le nombre de bits du compteur. Le comptage s'effectue en binaire naturel (sans limitation de comptage). On peut modifier ce code pour obtenir un système avec une valeur limite de comptage. Il faut définir cette limite (soit sous la forme d'un entier, soit sous la forme d'un bus) puis, il faut tester si la valeur de l'état interne égale cette valeur pour prévoir une remise à zero en lieu et place de l'incrémentation...