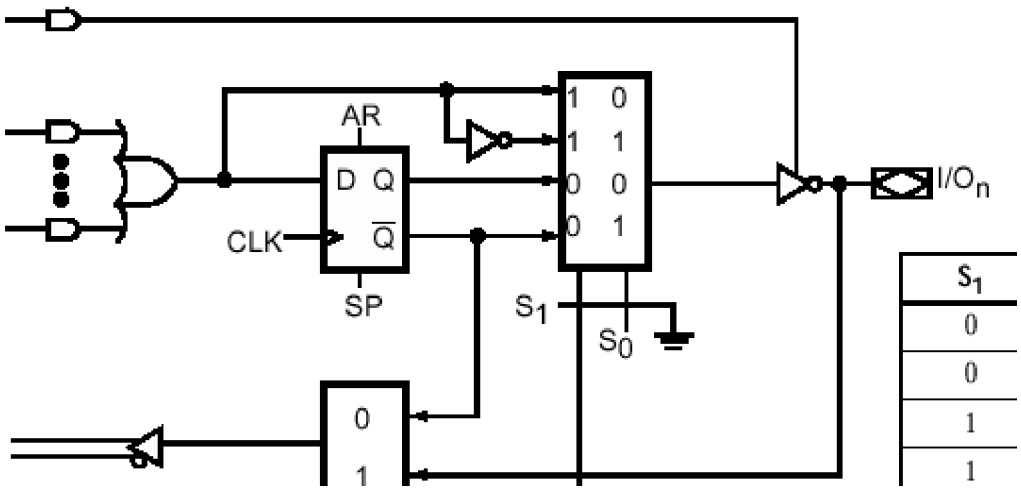
VHDL : résumé de syntaxe



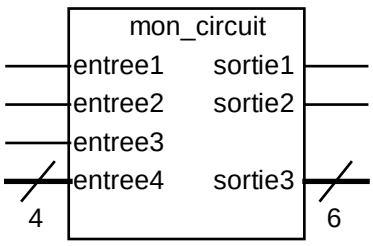

```
ARCHITECTURE vecteur OF decade IS
   signal countTemp: unsigned(3 downto 0) ;
begin
   count <= chiffre(to_integer(countTemp)) ; --
conversion
   dix <= en when countTemp = 9 else '0' ;
   incre : PROCESS
      BEGIN
      WAIT UNTIL rising_edge(clk) ;
         if raz = '1' then
            countTemp <= X"0" ;
         elsif en = '1' then
            countTemp <= (countTemp + 1) mod 10 ;
         end if ;
      END process incre ;
END vecteur ;
```

VHDL : Résumé partiel de la syntaxe

Bibliographie

1. J. Weber and M. Meaudre, *Circuits numériques et synthèse logique, un outil : VHDL*, MASSON, Paris, 1995 (épuisé).
2. J. Weber and M. Meaudre, *VHDL du langage au circuit, du circuit au langage*, MASSON, Paris, 1997.
3. *IEEE Standard VHDL Language Reference Manual, Std 1076-1993*, IEEE – 1993.
4. R. Airiau, J.M. Bergé, V. Olive, J. Rouillard, *VHDL du langage à la modélisation*, Presses Polytechniques et Universitaires Romandes – 1990.

Entity




```
entity mon_circuit is port (
  entree1 : in bit;
  entree2, entree3  : in bit;
  entree4 : in bit_vector(0 to 3);
  sortie1, sortie2 : out bit;
  sortie3 : out bit_vector(0 to 5));
end mon_circuit;
```

Le nom est connu à l'intérieur de toutes les architectures qui font référence à l'entité correspondante. Il est constitué par une chaîne de caractères alphanumériques qui commence par une lettre et qui peut contenir le caractère souligné (_) ; VHDL ne distingue pas les majuscules des minuscules, AmI et aMi représentent donc le même objet. Le mode précise le sens de transfert des informations, s'il n'est pas précisé le mode in est supposé.

Les modes **in** et **out** sont simples à interpréter : les informations transitent par les ports correspondants comme sur des voies à sens unique. Ils correspondent, dans le cas d'un circuit, aux broches d'entrées et de sorties des informations ; un circuit ne peut évidemment pas modifier de lui même les valeurs de ses signaux d'entrées. Il ne peut pas non plus, et cela est moins évident, " relire " la valeur de ses signaux de sortie.

Le mode **buffer** décrit un port de sortie dont la valeur peut être " relue " à l'intérieur de l'unité décrite. Il correspond à un signal de sortie associé à un rétrocouplage vers l'intérieur d'un circuit, par exemple.

Le mode **inout** décrit un signal d'interface réellement bidirectionnel : les informations peuvent transiter dans les deux sens. Il faut bien sûr, dans ce cas, prévoir la gestion de conflits potentiels, la valeur du signal présent sur le port pouvant avoir deux sources différentes. Les deux modes précédents ne doivent pas être confondus, le premier correspond à un signal dont la source est unique, interne à l'unité décrite, le second correspond à un signal dont les sources peuvent être multiples, internes et externes : un bus. Dans la figure 1 des flèches illustrent, pour chaque mode, les sens de transfert des informations entre l'intérieur d'une unité et l'extérieur.

Le type des données transférées doit être précisé. Comme ces données sont échangées avec le monde extérieur, les définitions de leurs types doivent être faites à l'extérieur de l'entité ; il est donc logique que la zone de déclaration locale se trouve après celle qui définit les ports, cela n'aurait aucun sens de vouloir l'utiliser pour définir le type associé à un port, ce type serait inconnu de l'extérieur.

Architecture

```
architecture exemple of mon_circuit is
      partie déclarative optionnelle :    types, constantes,
                        signaux locaux, composants.
```

```
begin
corps de l'architecture.
suite d'instructions parallèles :
      affectations de signaux;
      processus explicites;
      blocs;
      instanciation (i.e. importation
      dans un schéma) de composants.
end exemple ;
```

Process

```
[étiquette : ] process [ (liste de sensibilité) ]
      partie déclarative optionnelle : variables notamment
begin
      corps du processus.
      instructions séquentielles
end process [ étiquette ] ;
```

Les éléments mis entre crochets sont optionnels, ils peuvent être omis sans qu'il y ait d'erreur de syntaxe.

La liste de sensibilité est la liste des signaux qui déclenchent, par le changement de valeur de l'un quelconque d'entre eux, l'activité du processus. Cette liste peut être remplacée par une instruction « wait » dans le corps du processus :

```
wait [on liste_de_signaux ] [until condition ] ;
```

La liste des signaux dont l'instruction attend le changement de valeur joue exactement le même rôle que la liste de sensibilité du processus, mais *l'instruction `wait` ne peut pas être utilisée en même temps qu'une liste de sensibilité.*

Description d'un opérateur séquentiel

Pour modéliser un comportement purement synchrone on peut indifféremment utiliser la liste de sensibilité ou une instruction wait :

```
architecture fsm_liste of jk_simple is
signal etat : bit;
begin
q <= etat;
process(clock) -- un seul signal d'activation
begin
if(clock = '1'and clock'event) then
case etat is
      when '0' =>
            IF (j = '1' ) then
                  etat <= '1';
            end if;
      when '1' =>
            if (k = '1' ) then
                  etat <= '0';
            end if;
end case;
end if;
end process;
end fsm_liste;
```

Ou, de façon strictement équivalente, en utilisant une instruction « wait » :

```
architecture fsm_wait of jk_simple is
signal etat : bit;
begin
q <= etat;
process  -- pas de liste de sensibilité
begin
wait until (clock = '1') ;
```

```
case etat is
      when '0' =>
            IF (j = '1' ) then
                  etat <= '1';
            end if;
      when '1' =>
            if (k = '1' ) then
                  etat <= '0';
            end if;
end case;
end process;
end fsm_wait;
```

Cas des commandes **synchrones et asynchrones** :

```
architecture fsm of jk_raz is
signal etat : bit;
begin
q <= etat;
process(clock,raz) -- deux signaux d'activation
begin
if(raz = '1') then -- raz asynchrone
      etat <= '0';
elsif(clock = '1'and clock'event) then
case etat is
      when '0' =>
            IF (j = '1' ) then
                  etat <= '1';
            end if;
      when '1' =>
            if (k = '1' ) then
                  etat <= '0';
            end if;
end case;
end if;
end process;
end fsm;
```

Description par un processus d'un opérateur combinatoire ou asynchrone

```
entity comb_seq is
port (
      e1, e2 : in  bit ;
      s_et, s_latch, s_edge : out bit
      ) ;
end comb_seq ;

architecture exproc of comb_seq is
begin

et : process(e1,e2) -- équivalent à s_et <= e1 and e2 ;
begin
if( e1 = '1' ) then
      s_et <= e2 ;
else
      s_et <= '0' ;
end if ;
end process ;

latch : process(e1,e2) -- bascule D Latch, e1 est la commande.
begin
if( e1 = '1' ) then
      s_latch <= e2 ;
end if; -- si e1 = '0' la valeur de s_latch est non spécifiée.
end process ;

edge : process(e1) -- bascule D Edge, e1 est l'horloge.
```

```
begin
if( e1'event and e1 = '1' ) then -- e1 agit par un front.
      s_edge <= e2 ;
end if ;
end process ;
end exproc ;
```

Classes et types

Les classes : signaux, variables et constantes

Signaux

Syntaxe de déclaration :

```
signal nom1 , nom2 : type ;
```

Affectation d'une valeur :

```
nom <= valeur_compatible_avec_le_type ;
```

Variables

Les variables sont des objets qui servent à stocker un résultat intermédiaire pour faciliter la construction d'un *algorithme séquentiel*.

Syntaxe de déclaration :

```
variable nom1 , nom2 : type [:= expression];
nom := valeur_compatible_avec_le_type ;
```

Constantes

```
'0', '1', "01101001", 1995, "azerty"
constant nom1 : type [ := valeur_constante ] ;
X"3A007", O"237015"  pour hexadécimal et octal.
```

Les valeurs entières peuvent être écrites dans une autre base que la base 10 :

```
16#ABCDEF0123#, 2#001011101# ou 2#0_0101_1101#,
                        pour plus de lisibilité.
```

Des types adaptés à l'électronique numérique

Les entiers

VHDL manipule des valeurs entières qui correspondent à des mots de 32 bits, soit comprises entre

```
-2147483648 et +2147483647.
```

Déclarations :

```
signal nom : integer ;
variable nom : integer ;
constant nom : integer ;
signal etat : integer range 0 to 1023 ;
subtype etat_10 is integer range 0 to 1023 ;
signal etat1 , etat2 : etat_10 ;
```

Les types énumérés

```
type drinkState is (zero,five,ten,fifteen,twenty,twentyfive,owedime);
signal drinkStatus: drinkState;
```

Les bits

```
signal | variable nom : bit ;
```

Les booléens

Autre type énuméré, le type booléen peut prendre deux valeurs : "`true`" et "`false`".

Les tableaux

```
SUBTYPE Natural IS Integer RANGE 0 to Integer'high;
   TYPE bit_vector IS ARRAY (Natural RANGE <>) OF BIT;
```

Dans l'exemple qui précède, le nombre d'éléments n'est pas précisé dans le type, ce sera fait à l'utilisation. Par exemple :

```
signal etat : bit_vector (0 to 4) ;
```

ou :

```
type cinq_bit is array (0 to 4) of bit;
signal etat : cinq_bit ;
```

Les enregistrements

```
type clock_time is record
      hour : integer range 0 to 12 ;
      minute , seconde : integer range 0 to 59 ;
end record ;

variable time_of_day : clock_time ;
```

Utilisation de l'objet précédent :

```
time_of_day.hour := 3 ;
time_of_day.minute := 45 ;
chrono := time_of_day.seconde ;
```

Les attributs

`hor'event and hor = '1'` renvoie la valeur booléenne `true` si le signal `hor`, de type bit, vaut 1 après un changement de valeur, ce qui revient à tester la présence d'une transition montante de ce signal.

Attributs prédéfinis dans le langage

Quelques exemples :

attribut	agit sur	valeur retournée
‘left	type scalaire	élément de gauche
‘left(n)	type tableau	borne de gauche de l’indice de la dimension n, n=1 par défaut
‘right	type scalaire	élément de droite
‘right(n)	type tableau	borne de droite de l’indice de la dimension n, n=1 par défaut
‘high	type scalaire	élément le plus grand
‘high(n)	type tableau	borne maximum de l’indice de la dimension n, n=1 par défaut
‘low	type scalaire	élément le plus petit
‘low(n)	type tableau	borne minimum de l’indice de la dimension n, n=1 par défaut
‘event	signal	valeur booléenne "TRUE" si la valeur du signal vient de changer

Attributs spécifiques à un système

```
attribute synthesis_off of som4 : signal is true ;
```

permet, avec l’outil « WARP », d’empêcher l’élimination du signal som4 par l’optimiseur.

```
attribute pin_numbers of T_edge:entity is "s:20 ";
```

permet, avec le même outil, de préciser que le port s, de l’entité T_edge, doit être placé sur la broche N° 20 du circuit.

Les opérateurs élémentaires

Classe	Opérateurs	Types d’opérandes	Résultat
Op. logiques	and or nand nor xor	bits ou booléens	bit ou booléen
Op. relationnels	= /= < <= > >=	tous types	booléen
Op. additifs	+ –	numériques	numérique
	&	tableaux (concaténation)	tableau
Signe	+ –	numériques	numérique
Opérateurs multiplicatifs	* /	numériques (restrictions)	numérique
	mod rem	entiers (restrictions)	entier
Op. divers	not	bit ou booléen	bit ou booléen
	abs	numérique	numérique
	**	numériques (restrictions)	numérique

Remarques :

- Les opérateurs multiplicatifs et l’opérateur d’exponentiation (**) sont soumis à des restrictions, notamment en synthèse où seules les opérations qui se résument à des décalages sont généralement acceptées.
- Certaines librairies (numeric_bit et numeric_std de la librairie IEEE) surdéfinissent (au sens des langages objets) les opérateurs d’addition et de soustraction pour les étendre au type bit_vector, par exemple.
- On notera que tous les opérateurs logiques ont la même priorité, il est donc plus que conseillé de parenthéser toutes les expressions qui contiennent des opérateurs différents de cette classe.

Opérandes

Les opérandes d’une expression peuvent être des objets nommés, une expression entre parenthèse, un appel de fonction, une constante littérale ou un agrégat. Dans certains cas, plutôt rares, on peut être amené à préciser ou, de façon très restrictive, modifier le sous-type d’un opérande.

Les noms

Les noms des objets simples, comme des constantes ou des signaux scalaires, sont formés de lettres, de chiffres et du caractère souligné (‘_’). Le premier caractère doit être une lettre.

Un membre d’un enregistrement est désigné par un nom sélectionné : il s’agit d’un nom composé d’un préfixe et d’un suffixe séparés par un point. Par exemple, étant donnée la déclaration :

```
signal afficheur : date ;-- type défini précédemment
```

Le champ jour du signal afficheur est repéré par le nom composé :

```
afficheur.jour
```

Les noms sélectionnés interviennent également pour accéder aux objets déclarés dans des librairies ; schématiquement le préfixe représente le chemin d'accès et le suffixe le nom simple de l'objet, dans une construction similaire à celle qui est utilisée pour retrouver un fichier informatique dans un système de fichiers.

L'accès aux éléments d'un tableau se fait par un nom composé qui comporte comme préfixe le nom du tableau et comme suffixe une expression, ou une liste d'expressions, entre parenthèses. Il peut se faire élément par élément :

```
sort <= entree(sel) ;
```

ou, dans le cas des vecteurs uniquement, par tranche :

```
mem1(5 to 12) <= mem2(0 to 7) ;
```

Le préfixe du nom d'un tableau ou d'un enregistrement réfère tous les éléments de l'objet structuré :

```
constant bastille : date := (14,jul,1789) ;
constant bast1 : date := bastille ;
```

La dernière forme de noms composés s'applique aux attributs. La référence à un attribut se fait par le nom de l'objet, en préfixe, suivi du nom de l'attribut en suffixe, séparé du précédent par une apostrophe :

```
if hor'event and hor = '1' then -- etc.

function ouex ( a : in bit_vector ) return bit is
variable parite : bit ;
begin
   parite := '0' ;
   for i in a'low to a'high loop -- etc.
```

Alias

Il est parfois pratique de désigner par un autre nom une entité nommée qui existe déjà sous un premier nom. Cette entité peut être un objet, un sous programme, un type etc.

Il est important de noter qu'une déclaration d'alias ne crée rien d'autre qu'un synonyme. Elle ne crée en aucun cas un nouvel objet, la classe de l'objet visé n'est pas modifiée.

La syntaxe générale (simplifiée) de création d'un alias est :

```
alias alias_designator [: subtype_indication ] is name ;
```

Exemples :

```
signal instr : bit_vector(0 to 15) ;
alias opcode : bit_vector(0 to 9) is instr (0 to 9) ;
alias source : bit_vector(2 downto 0) is instr (10 to 12) ;
```

Les constantes littérales

Les constantes littérales désignent des valeurs numériques, des caractères ou des chaînes de caractères, ou des chaînes de valeurs binaires :

Des nombres :

```
14      0     1E4    123_456  -- des entiers en base 10
14.0    0.0   1.0e4  123.456  -- des flottants en base 10
2#1010#      16#A#   8#12#    -- l'entier 10
16#F.FF#E+2  2#1.1111_1111_111#E11 -- le flottant 4095.0
```

Des caractères et des chaînes :

```
'a'   'B'   '5'              -- des caractères
"ceci est une chaine"  "B"   -- des chaînes de
                             -- caractères
```

Des chaînes binaires :

```
B"1111_1111"     -- équivaut à "11111111"
X"FF"            -- la même chaîne
```

```
O"377"          -- équivaut à "011111111"
```

La différence entre chaînes binaires et chaînes de caractères équivalentes réside dans les facilités d'écritures apportées par la base et les caractères soulignés autorisés dans les premières pour faciliter la lecture. Il est important de noter que le nombre de caractères binaires générés dépend de la base, les deux derniers exemples ne sont donc pas équivalents, même si leurs équivalents numériques le sont.

Les agrégats

Les agrégats constituent une notation efficace pour combiner plusieurs valeurs de façon à constituer une valeur de type structuré (enregistrement ou tableau). Compte tenu de leur utilisation fréquente, nous allons un peu détailler la syntaxe de leur construction.

```
aggregate ::=
      ( element_association { , element_associaton } )

element_association ::=
      [ choices => ] expression

choices ::=
      choice { | choice }

choice ::=
        simple_expression
      | discrete_range
      | element_simple_name
      | others
```

Un agrégat est construit comme une liste d'associations élémentaires, qui fait correspondre une valeur à un ou plusieurs membres d'un type structuré. Le type doit évidemment être déductible du contexte de l'expression. Chaque association élémentaire peut se faire par position - on ne précise pas explicitement le membre destinataire, les valeurs sont énumérées dans l'ordre utilisé dans la définition du type - ou en spécifiant le ou les membres destinataires explicitement. Dans ce dernier cas, il est possible de créer des associations « collectives », qui sont pratiques pour initialiser plusieurs éléments d'un tableau à une même valeur, par exemple.

Exemples :

```
      -- associations par position
type table_verite_2 is array(bit,bit) of bit ;
constant ouex_tv  : table_verite_2 := (('0','1'),('1','0')) ;
constant ouex_tv1 : table_verite_2 := ("01","10") ;
constant bastille : date := (14,jul,1789) ;

      -- associations explicites
constant bast2 : date := (mois => jul,annee => 1789,jour => 14) ;
constant ouex_tv2 : table_verite_2 := ('0'=>('0'=>'0','1'=>'1'),
                                       '1'=>('0'=>'1','1'=>'0'));
      -- tableau à deux dimensions ::= vecteur(vecteur).

      -- associations collectives
constant rom_presque_vide : mem8 := (0 to 10 => 3, others => 255);
constant rom_vide : mem8 := (others => 16#FF#) ;
donnee <= temp when direction = '0' else (others => 'Z') ;
```

Si on panache les différents modes d'association, les associations par position doivent être mises en premier, celles par référence en dernier. Le mot clé `others`, s'il est utilisé, ne peut que définir la dernière association de la liste.

Précisions de types

VHDL ne tolère à priori pas les mélanges de types. Deux cas particuliers, de natures fort différentes, il est vrai, conduisent parfois (rarement, de fait) un programmeur à préciser le type d'une construction :

– Le résultat de l'évaluation d'une expression ou d'un agrégat peut être d'un type ambigu. On peut alors préciser ce type par une expression qualifiée :

```
type x01z is ('x','0','1','z') ;
constant zero  : bit  := '0' ;-- type bit
constant zero1 : x01z := x01z'('0') ;-- expression qualifiée
```

Rarement nécessaire, cette précision peut être utile, par exemple, quand des fonctions ou des procédures existent sous plusieurs formes qui dépendent des types des opérandes (surcharge).

– Entre deux types « proches » (*closely related*) il est possible de forcer une conversion de type. L'exemple le plus classique est celui des types numériques entre lesquels les conversions sont permises :

```
constant pi    : real := 3.14159 ;
constant intpi : integer := integer(10000.0*pi)
```

Le seul autre cas de conversions autorisées par le langage concerne des tableaux qui ont le même nombre de dimensions, et dont les indices et les éléments sont de mêmes types.

Instructions concurrentes

Affectation simple

L'affectation simple traduit une simple interconnexion entre deux équipotentielles. L'opérateur d'affectation de signaux (<=) a été vu précédemment :

```
nom_de_signal <= expression_du_bon_type ;
```

Affectation conditionnelle

```
cible <=    source_1 when condition_booléenne_1 else
      source_2 when condition_booléenne_2 else
      ...
      source_n ;
```

Affectation sélective

```
with expression select
cible <= source_1 when valeur_11 | valeur_12 ... ,
    source_2 when valeur_21 | valeur_22 ... ,
    ...
    source_n when others ;
```

Instanciation de composant

Déclaration :

```
component nom_composant -- même nom que l'entité
      port ( liste_ports ) ; -- même liste que dans
         -- l'entité
end component ;
```

Cette déclaration est à mettre dans la partie déclarative de l'architecture du circuit utilisateur, ou dans un paquetage qui sera rendu visible par une clause « use ».

« Instanciation » :

```
Etiquette : nom port map ( liste_d'association ) ;
```

Exemple :

```
architecture exemple of xyz is
component et
port ( a , b  : in bit ;
      a_et_b : out bit ) ;
end component ;
signal s_a, s_b, s_a_et_b, s1, s2, s_1_et_2 : bit;

begin
....
-- utilisation : association par position
et1 : et port map ( s_a , s_b , s_a_et_b ) ;
-- ou : association par référence
et2 : et port map
      (a_et_b => s_1_et_2,a => s1, b => s2) ;
```

```
....
end exemple ;
```

En raison de sa simplicité, l'association par position est la plus fréquemment employée.

Generate

Une instruction generate permet de dupliquer un bloc d'instructions concurrentes un certain nombre de fois, ou de créer un tel bloc si une condition est vérifiée.

Syntaxe :

```
-- structure répétitive :
etiquette : for variable in debut to fin generate
       instructions concurrentes
end generate [etiquette] ;
```

ou :

```
-- structure conditionnelle :
etiquette : if condition generate
instructions concurrentes
end generate [etiquette] ;
```

Exemple :

```
ENTITY cnt16 IS
      PORT (ck,raz,en : IN BIT;
            s : OUT BIT_VECTOR (0 TO 3) );
END cnt16;

ARCHITECTURE struct OF cnt16 IS
      SIGNAL etat : BIT_VECTOR(0 TO 3);
      SIGNAL inter: BIT_VECTOR(0 TO 3);
COMPONENT T_edge -- supposé présent dans la librairie work
      port ( T,hor,zero : in bit;
            s : out bit);
END COMPONENT;

BEGIN
      s <= etat ;
gen_for : for i in 0 to 3 generate
   gen_if1 : if i = 0 generate
      inter(0) <= en ;
   end generate gen_if1 ;
   gen_if2 : if i > 0 generate
      inter(i) <= etat(i - 1) and inter(i - 1) ;
   end generate gen_if2 ;
   comp1_3 : T_edge port map (inter(i),ck,raz,etat(i));
end generate gen_for ;
END struct;
```

Instructions séquentielles

Les instructions séquentielles sont *internes* aux processus, aux procédures et aux fonctions.

L'instruction « if ... then ... else ... end if »

Syntaxe :

```
if expression_logique then
instructions séquentielles
[ elsif expression_logique then ]
instructions séquentielles
[ else ]
instructions séquentielles
end if ;
```

L'instruction « case ... when ... end case »

Syntaxe :

```
case expression is
when choix | choix | ... choix => instruction sequentielle ;
when choix | choix | ... choix => instruction sequentielle ;
....
when others => instruction sequentielle ;
end case ;
```

Les boucles « for »

```
[ etiquette : ] for parametre in minimum to maximum loop
séquence d'instructions
end loop [ etiquette ] ;
```

Ou :

```
[ etiquette : ] for parametre in maximum downto minimum loop
séquence d'instructions
end loop [ etiquette ] ;
```

Les boucles « while »

```
[ etiquette : ] while condition loop
séquence d'instructions
end loop [ etiquette ] ;
```

« Next » et « exit »

```
next [ etiquette ] [ when condition ] ;
```

Permet de passer à l'itération suivante d'une boucle.

```
exit [ etiquette ] [ when condition ] ;
```

Permet de provoquer une sortie de boucle.

Les fonctions

Déclaration :

```
function nom [ ( liste de paramètres formels ) ]
return nom_de_type ;
```

Corps de la fonction :

```
function nom [ ( liste de paramètres formels ) ]
return nom_de_type is
[ déclarations ]
begin
instructions séquentielles
end [ nom ] ;
```

Exemple

Déclaration :

```
FUNCTION inc_bv   (a : BIT_VECTOR)   RETURN BIT_VECTOR ;
```

Corps de la fonction :

```
FUNCTION inc_bv   (a : BIT_VECTOR)  RETURN BIT_VECTOR IS
VARIABLE s  : BIT_VECTOR (a'RANGE);
VARIABLE carry    : BIT;
BEGIN
carry := '1';

FOR i IN a'LOW TO a'HIGH LOOP -- les attributs LOW et
                              -- HIGH déterminent les
                              -- dimensions du vecteur.
    s(i)    := a(i) XOR carry;
    carry   := a(i) AND carry;
END LOOP;

RETURN      (s);
END inc_bv;
```

Utilisation dans un compteur :

```
ARCHITECTURE behavior OF counter IS
BEGIN
PROCESS
BEGIN
WAIT UNTIL (clk = '1');
IF reset = '1' THEN
      count <= "0000";
ELSIF load = '1' THEN
      count <= dataIn;
ELSE
      count <= inc_bv(count); -- increment du bit vector
END IF;
END process;
END behavior;
```

Les procédures

Déclaration :

```
procedure nom [ ( liste de paramètres formels ) ];
```

Corps de la procédure :

```
procedure nom [ ( liste de paramètres formels ) ] is
[ déclarations ]
begin
```

```
instructions séquentielles
end [ nom ] ;
```

Dans la liste des paramètres formels, la nature des arguments doit être précisée :

```
procedure exemple ( signal a, b : in bit ;
                    signal s : out bit ) ;
```

Utilisation :

```
nom ( liste de paramètres réels ) ;
```

Soit :

```
exemple (entree1, entree2, sortie) ;
```

Ou :

```
exemple (s => sortie, a => entree1, b => entree2);
```

Les paquetages

```
library ieee ; -- rend la librairie IEEE visible
use ieee.std_logic_1164.all; -- rend le paquetage
      -- std_logic_1164, de la librairie ieee,
      -- visible dans sa totalité.
```

Exemple : un diviseur par 50

```
-- div50.vhd
library ieee ;
use ieee.numeric_bit.all ;

entity div50 is
port (      hor : in bit;
            s : out bit ); -- sortie
end div50 ;

architecture arith_bv of div50 is
   signal etat : unsigned(5 downto 0) ; -- vecteur vu comme un nombre
begin
s <= etat(5);
process
begin
  wait until hor ='1' ;
    if etat = 24 then -- opérateur "=" surchargé.
      etat <= to_unsigned(39,6); -- fonction de
                  -- conversion d'un entier en
                  -- bit_vector de 6 bits.
    else
  etat <= etat + 1;
            -- opérateur "+" surchargé.
    end if ;
end process ;
end arith_bv ;
```

Les paquetages créés par l'utilisateur

La syntaxe de la déclaration d'un paquetage est la suivante :

```
package identificateur is
déclarations de types, de fonctions,
de composants, d'attributs,
clause use, ...  etc
end [identificateur] ;
```

S'il existe, le corps du paquetage doit porter le même nom que celui qui figure dans la déclaration :

```
package body identificateur is
      corps des sous programmes déclarés.
```

```
end [identificateur] ;
```

Exemple :

```
package T_edge_pkg is
COMPONENT T_edge -- une bascule T avec mise à 0.
port ( T,hor,raz : in bit;
       s : out bit);
END COMPONENT;
end T_edge_pkg ;
```

Le compteur proprement dit :

```
ENTITY cnt4 IS
PORT (ck,razero,en : IN BIT;
      s : OUT BIT_VECTOR (0 TO 1)
      );
END cnt4;

use work.T_edge_pkg.all ;
      -- rend le contenu du package
      -- précédent visible.

ARCHITECTURE struct OF cnt4  IS
SIGNAL etat : BIT_VECTOR(0 TO 1);
signal inter:bit;

BEGIN
s <= etat ;
inter <= etat(0) and en ;
g0 : T_edge port map (en,ck,razero,etat(0));
g1 : T_edge port map (inter,ck,razero,etat(1));
END struct;
```

Les paquetages de la librairie IEEE

La librairie IEEE joue un rôle fédérateur et remplace tous les dialectes locaux. En cours de généralisation, y compris en synthèse, elle définit un type de base à neuf états, `std_ulogic` (présenté précédemment comme exemple de type énuméré) et des sous-types dérivés simples et structurés (vecteurs). Des fonctions et opérateurs surchargés permettent d'effectuer des conversions et de manipuler les vecteurs comme des nombres entiers.

A l'heure actuelle la librairie IEEE comporte trois paquetages dont nous examinerons plus en détail certains aspects au paragraphe II-7 :

- `std_logic_1164` définit les types, les fonctions de conversion, les opérateurs logiques et les fonctions de recherches de fronts `rising_edge()` et `falling_edge()`.
- `numeric_bit` définit les opérateurs arithmétiques agissant sur des `bit_vector` interprétés comme des nombres entiers.
- `numeric_std` définit les opérateurs arithmétiques agissant sur des `std_logic_vector` interprétés comme des nombres entiers.

Le paquetage `ieee.std_logic_1164` définit le type `std_ulogic` qui est le type de base de la librairie IEEE :

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

Le sous-type `std_logic`, qui est le plus utilisé, est associé à la fonction de résolution `resolved` :

```
function resolved ( s : std_ulogic_vector )
                  return std_ulogic;
```

```
subtype std_logic is resolved std_ulogic;
```

Cette fonction de résolution utilise une table de gestion des conflits qui reproduit les forces respectives des valeurs du type :

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

Le paquetage définit également des vecteurs :

```
type std_logic_vector is array ( natural range <> )
                         of std_logic;

type std_ulogic_vector is array ( natural range <> )
                          of std_ulogic;
```

Et les sous-types résolus `X01`, `X01Z`, `UX01` et `UX01Z`.

Des fonctions de conversions permettent de passer du type binaire aux types IEEE et réciproquement, ou d'un type IEEE à l'autre :

```
function To_bit ( s : std_ulogic; xmap : bit := '0')
                  return bit;
function To_bitvector ( s : std_logic_vector ;
                        xmap : bit := '0') return bit_vector;
function To_bitvector ( s : std_ulogic_vector;
                        xmap : bit := '0') return bit_vector;
function To_StdULogic ( b : bit ) return std_ulogic;
function To_StdLogicVector ( b : bit_vector )
                             return std_logic_vector;
function To_StdLogicVector ( s : std_ulogic_vector )
                             return std_logic_vector;
function To_StdULogicVector ( b : bit_vector )
                              return std_ulogic_vector;
```

Par défaut les fonctions comme `To_bit` remplacent, au moyen du paramètre `xmap`, toutes les valeurs autres que `'1'` et `'H'` par `'0'`.

La détection d'un front d'horloge se fait au moyen des fonctions :

```
function rising_edge (signal s : std_ulogic) return boolean;
function falling_edge(signal s : std_ulogic) return boolean;
```

Tous les opérateurs logiques sont surchargés pour agir sur les types IEEE comme sur les types binaires. Ces opérateurs retournent les valeurs *fortes* `'0'`, `'1'`, `'X'`, ou `'U'`.

Nombres et vecteurs

Un nombre entier peut être assimilé à un vecteur, dont les éléments sont les coefficients binaires de son développement polynomial en base deux. Restent à définir sur ces objets les opérateurs arithmétiques, ce que permet la surcharge d'opérateurs.

Les paquetages `numeric_std` et `numeric_bit` correspondent à l'utilisation, sous forme de nombres, de vecteurs dont les éléments sont des types `std_logic` et `bit`, respectivement. Comme les types définis dans ces paquetages portent les mêmes noms, ils ne peuvent pas être rendus visibles simultanément dans un même module de programme ; il faut choisir un contexte ou l'autre.

Les deux paquetages ont pratiquement la même structure, et définissent les types `signed` et `unsigned` :

```
-- ieee.numeric_bit :
type UNSIGNED is array (NATURAL range <> ) of BIT;
type SIGNED is array (NATURAL range <> ) of BIT;

-- ieee.numeric_std :
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
type SIGNED is array (NATURAL range <>) of STD_LOGIC;
```

Les vecteurs doivent être rangés dans l'ordre descendant (`downto`) de l'indice, de sorte que le coefficient de poids fort soit toujours écrit à gauche, et que le coefficient de poids faible, d'indice `0`, soit à droite, ce qui est l'ordre naturel. La représentation interne des nombres signés correspond au code complément à deux, dans laquelle le chiffre de poids fort est le bit de signe (`'1'` pour un nombre négatif, `'0'` pour un nombre positif ou nul).

Les opérations prédéfinies dans ces paquetages, agissant sur les types `signed` et `unsigned`, sont :

- Les opérations arithmétiques.
- Les comparaisons.
- Les opérations logiques pour `numeric_std`, elles sont natives pour les vecteurs de bits.
- Des fonctions de conversion entre nombres et vecteurs :

```
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;
```

- Une fonction de recherche d'identité (`std_match`) qui utilise l'état *don't care* du type `std_logic` comme joker.
- Les recherches de fronts (`rising_edge` et `falling_edge`), agissant sur le type `bit`, dans `numeric_bit`.

Des opérations prédéfinies

Les opérandes et les résultats des opérateurs arithmétiques et relationnels appellent quelques commentaires. Ces opérateurs acceptent comme opérandes deux vecteurs ou un vecteur et un nombre. Dans le cas de deux vecteurs dont l'un est signé, l'autre pas, il est à la charge du programmeur de prévoir les conversions de types nécessaires.

Les opérateurs arithmétiques

L'addition et la soustraction sont faites sans aucun test de débordement, ni génération de retenue finale. La dimension du résultat est celle du plus grand des opérandes, quand l'opération porte sur deux vecteurs, ou celle du vecteur passé en argument dans le cas d'une opération entre un vecteur et un nombre. Le résultat est donc calculé implicitement modulo 2^n, où n est la dimension du vecteur retourné. Ce modulo implicite allège, par exemple, la description d'un compteur binaire, évitant au programmeur de prévoir explicitement l'incrémentation modulo la taille du compteur.

La multiplication retourne un résultat dont la dimension est calculée pour pouvoir contenir le plus grand résultat possible : somme des dimensions des opérandes moins un, dans le cas de deux vecteurs, double de la dimension du vecteur passé en paramètre moins un dans le cas de la multiplication d'un vecteur par un nombre.

Pour la division entre deux vecteurs, le quotient a la dimension du dividende, le reste celle du diviseur. Quand les opérations portent sur un nombre et un vecteur, la dimension du résultat ne peut pas dépasser celle du vecteur, que celui-ci soit dividende ou diviseur.

Les opérateurs relationnels

Quand on compare des vecteurs interprétés comme étant des nombres, les résultats peuvent être différents de ceux que l'on obtiendrait en comparant des vecteurs sans signification. Le tableau ci-dessous donne quelques exemples de résultats en fonction des types des opérandes :

Expression	Types des opérandes		
	bit_vector	unsigned	signed
"001" = "00001"	FALSE	TRUE	TRUE
"001" > "00001"	TRUE	FALSE	FALSE
"100" < "01000"	FALSE	TRUE	TRUE
"010" < "10000"	TRUE	TRUE	FALSE
"100" < "00100"	FALSE	FALSE	TRUE

Ces résultats se comprennent aisément si on garde à l'esprit que la comparaison de vecteurs ordinaires, sans signification numérique, se fait de gauche à droite sans notion de poids attaché aux éléments binaires.

Compteur décimal

Comme illustration de l'utilisation de la librairie IEEE, donnons le code source d'une version possible de la décade, instanciée comme composant dans un compteur décimal :

```
library ieee ;
use ieee.numeric_bit.all ;

ARCHITECTURE vecteur OF decade IS
   signal countTemp: unsigned(3 downto 0) ;
begin
   count <= chiffre(to_integer(countTemp)) ; -- conversion
   dix <= en when countTemp = 9 else '0' ;
   incre : PROCESS
      BEGIN
      WAIT UNTIL rising_edge(clk) ;
         if raz = '1' then
            countTemp <= X"0" ;
-- ou :     countTemp <= to_unsigned(0) ;
         elsif en = '1' then
            countTemp <= (countTemp + 1) mod 10 ;
         end if ;
      END process incre ;
END vecteur ;
```

L'intérêt de ce programme réside dans l'aspect évident des choses, le signal `countTemp`, un vecteur d'éléments binaires, est manipulé dans des opérations arithmétiques exactement comme s'il s'agissait d'un nombre. Seules certaines opérations de conversions rappellent les différences de nature entre les types `unsigned` et `integer`.

Les paramètres génériques

Un paramètre générique se déclare au début de l'entité, et peut avoir une valeur par défaut :

```
generic (nom : type [ := valeur_par_defaut ] ) ;
```

Au moment de l'instanciation la taille peut être modifiée par une instruction « generic map » :

```
Etiquette : nom generic map ( valeurs )
       port map ( liste_d'association ) ;
```

Exemple :

```
-- compteur tristate
-- cnt_tri.vhd

library ieee ;
use ieee.std_logic_1164.all, ieee.numeric_std.all ;
-- type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;

entity compteur_tri is
   generic(taille : integer := 10) ;
   port( hor,raz,en,oe : in std_ulogic ;
         compte : out unsigned(taille-1 downto 0) ) ;
```

```
end compteur_tri ;

architecture ieee_lib of compteur_tri is
   signal etat : unsigned(taille-1 downto 0) ;
begin
   compte <= etat when oe = '1' else (others => 'Z') ;
   compteur : process
   begin
      wait until rising_edge(hor) ;
      if raz = '1' then
         etat <= (others => '0') ;
      elsif en = '1' then
         etat <= etat + 1 ;
      end if ;
   end process compteur ;
end ieee_lib ;
```

Ce compteur est instancié comme un compteur 16 bits :

```
entity compt16 is
port( ck,raz,en,oe : in std_ulogic ;
         val : out unsigned(15 downto 0) ) ;
end compt16 ;

architecture large of compt16 is
component compteur
generic(taille : integer) ;
   port( hor,raz,en,oe : in std_ulogic ;
         compte : out unsigned(taille-1 downto 0) ) ;
end component ;
begin
u1 : compteur
      generic map (taille => 16)
      port map (hor => ck , raz => raz, en => en, oe => oe
                compte => val) ;
end large ;
```