VI Annexe : VHDL

VHDL est l'abréviation de « Very high speed integrated circuits Hardware Description Langage ». L'ambition des concepteurs du langage est de fournir un outil de description homogène des circuits, qui permette de créer des modèles de simulation et de « compiler » le silicium à partir d'un programme unique. Initialement réservé au monde des circuits numériques, VHDL est en passe d'être étendu aux circuits analogiques.

Deux des intérêts majeurs du langage sont :

- Des *niveaux de description* très divers: VHDL permet de représenter le fonctionnement d'une application tant du point de vue système que du point de vue circuit, en descendant jusqu'aux opérateurs les plus élémentaires. A chaque niveau, la description peut être structurelle (portrait des interconnexions entre des sous-fonctions) ou comportementale (langage évolué).
- Son aspect « *non propriétaire* »: le développement des circuits logiques a conduit chaque fabriquant à développer son propre langage de description. VHDL est en passe de devenir le langage commun à de nombreux systèmes de CAO, indépendants ou liés à des producteurs de circuits, des (relativement) simples outils d'aide à la programmation des PALs aux ASICs, en passant par les FPGAs.

La description qui suit est loin d'être exhaustive, héritier d'ADA, VHDL est un « gros » langage. Nous en présentons un sous-ensemble qui, nous l'espérons, doit permettre à un néophyte d'aborder ses premières réalisations avec un bagage minimum, limité à des *constructions synthétisables*, et, en principe, portables sur n'importe quel compilateur.

VI.1. Principes généraux

VI.1.1 Description descendante : le « top down design »

Une application un tant soit peu complexe est découpée en sous-ensembles qui échangent des informations suivant un protocole bien défini. Chaque sous-ensemble est, à son tour, subdivisé, et ainsi de suite jusqu'aux opérateurs élémentaires.

Un système est construit comme une hiérarchie d'objets, les détails de réalisation se précisant au fur et à mesure que l'on descend dans cette hiérarchie. A un niveau donné de la hiérarchie, les détails de fonctionnement interne des niveaux inférieurs sont *invisibles*, C'est le principe même de la programmation structurée. Plusieurs réalisations d'une même fonction pourront être envisagées, sans qu'il soit nécessaire de remettre en cause la conception des niveaux supérieurs ; plusieurs personnes pourront collaborer à un même projet, sans que chacun ait à connaître tous les détails de l'ensemble.

La conception descendante consiste à définir le système en partant du sommet de la hiérarchie, en allant du général au particulier. VHDL permet, par exemple, de tester la validité de la conception d'ensemble, avant que les détails des sous-fonctions ne soient complètement définis. A titre d'exemple, l'architecture générale d'un processeur peut être évaluée sans que le mode de réalisation de ses registres internes ne soit connu, le fonctionnement des registres en question sera alors décrit au niveau comportemental.

VI.1.2 Simulation et/ou synthèse

VHDL a été, initialement, conçu comme un langage de simulation, il est fortement marqué par cet héritage très informatique, ce qui est parfois un peu déroutant pour l'électronicien, proche du matériel, qui n'est pas toujours un spécialiste des langages de programmation. Citons quelques exemples :

- Contrairement à C ou PASCAL, VHDL est un langage qui comprend le « parallélisme », c'est à dire que des blocs d'instructions peuvent être exécutés simultanément, par opposition à séquentiellement comme dans un langage procédural traditionnel. Autant ce parallélisme est fondamental pour comprendre le fonctionnement d'un simulateur logique, et peut être déroutant pour un programmeur habitué au déroulement séquentiel des instructions qu'il écrit, autant il est évident que le fonctionnement d'un circuit ne dépend pas de l'ordre dans lequel ont été établies les connexions. L'utilisateur de VHDL gagnera beaucoup en ne se laissant pas enfermer dans l'aspect langage de programmation, en se souvenant qu'il est en train de créer un vrai circuit. Les parties séquentielles du langage, car il y en a, doivent, dans ce contexte, être comprises soit comme une facilité offerte dans

l'écriture de certaines fonctions, soit comme *le* moyen de décrire des opérateurs fondamentalement séquentiels : les opérateurs synchrones.

- La modélisation correcte d'un système suppose de prendre en compte, au niveau du simulateur, les imperfections du monde réel. VHDL offre donc la possibilité de spécifier des retards, de préciser ce qui se passe lors d'un conflit de bus etc. Pour simuler toutes ces vicissitudes, le langage offre toute une gamme d'outils : signaux qui prennent une valeur inconnue, messages d'erreurs quand un « circuit » détecte une violation de *set-up time*, changements d'états retardés pour simuler les temps de propagation. Toutes les constructions associées de ce type ne sont évidemment *pas* synthétisables ! La difficulté principale est que, suivant les compilateurs, la frontière entre ce qui est synthétisable et ce qui ne l'est pas n'est pas toujours la même, même pour des compilateurs qui respectent la norme IEEE-1076. Avant d'utiliser un outil de synthèse, le concepteur de circuit a tout à gagner à lire très attentivement la présentation du sous-ensemble de VHDL accepté par cet outil.
- Trois classes de données existent en VHDL : les constantes, les variables et les signaux. La nature des signaux ne présente aucune ambiguïté, ce sont des objets qui véhiculent une information logique tant du point de vue simulation que dans la réalité. Les signaux qui ont échappé aux simplifications logiques, apportées par l'optimiseur toujours présent, sont des vraies équipotentielles du schéma final. Les variables sont destinées, comme dans tout langage, à stocker temporairement des valeurs, dans l'optique d'une utilisation future, sans chercher à représenter la réalité. Certains compilateurs considèrent que les variables n'ont aucune existence réelle, au niveau du circuit, qu'elles ne sont que des outils de description fonctionnelle. D'autres transforment, éventuellement (cela dépend de l'optimiseur), les variables en cellules mémoires...

Tous les exemples qui sont donnés ici, et dans les chapitres précédents, ont été compilés sur un logiciel destiné à la synthèse de circuits programmables[1]. Créé par et pour des concepteurs de circuits, ce compilateur ne comporte aucune construction spécifique de la simulation, mais autorise cependant des édifices relativement élaborées, notamment dans l'utilisation des variables et des boucles[2]. Nous avons parfois eu quelques surprises désagréables lors du portage de ces exemples sur d'autres systèmes, syntaxiquement acceptés, certains programmes étaient refusés par l'outil de synthèse, ou généraient une quantité surprenante de bascules.

En conclusion citons l'un des « grands » de la CAO électronique[3]:

« Des mythes communs existent :

[1]WARP, Cypress Semiconductors

[2]Les variables ne « sortent » pas du programme, seuls les signaux se retrouvent dans le circuit final.

[3]Mentor Graphics, Methods for using Autologic in top down design, 1994.

La conception descendante est une démarche presse-bouton, la compétence de l'expert est rarement nécessaire.

Faux. VHDL, les outils de synthèse et d'optimisation ne peuvent pas transformer un mauvais concepteur en un bon. Ce sont de simples outils supplémentaires qui peuvent aider un ingénieur à réaliser plus rapidement et plus efficacement un matériel quand ils sont utilisés correctement.

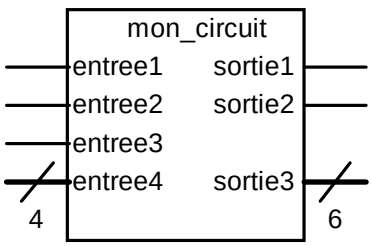
Les outils d'optimisation libèrent l'utilisateur de la nécessité de comprendre les détails physiques de sa réalisation.

Faux. Il est toujours nécessaire de comprendre les détails physiques de la façon dont est implémentée une réalisation. L'ingénieur doit *regarder par dessus l'épaule* de l'outil pour s'assurer que le résultat est conforme à ses exigences et à sa philosophie, et que le résultat est obtenu en un temps raisonnable. »

VI.1.3 L'extérieur de la boîte noire : une « ENTITÉ »

Nous avons mentionné que, dans une construction hiérarchique, les niveaux supérieurs n'ont pas à connaître les détails des niveaux inférieurs. Une fonction logique sera vue, dans cette optique, comme un assemblage de « boîtes noires », dont, syntaxiquement parlant, seules les modes d'accès sont nécessaires à l'utilisateur[4].

La construction qui décrit l'extérieur d'une fonction est l'entité (*entity*). La déclaration correspondante lui donne un nom et précise la liste des signaux d'entrée et de sortie :




```
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;
```

Figure VI-1

[4]Ce point est à mettre en parallèle avec les prototypes d'un langage comme le C. Dans un programme qui utilise la fonction sinus, le remplacement de celle-ci par une fonction exponentielle ne posera aucun problème de syntaxe, les modes d'accès sont les mêmes. Cela ne veut évidemment pas dire que les deux programmes fourniront les mêmes résultats, ce dernier point est un problème de sémantique.

Dans l'exemple qui précède, les noms des objets, qui dépendent du choix de l'utilisateur, sont écrits en italique, les autres mots sont des mots-clés du langage.

Les choix possibles pour le sens de transfert sont : in, out, inout et buffer (une sortie qui peut être « lue » par l'intérieur du circuit).

Les choix possibles pour les types de données échangées sont les mêmes que pour les signaux (voir ci-dessous).

VI.1.4 Le fonctionnement interne : une « ARCHITECTURE »

L'architecture décrit le fonctionnement interne d'un circuit auquel est attaché une entité. Ce fonctionnement peut être décrit de différentes façons :

Description structurelle - le circuit est vu comme un assemblage de composants de niveau inférieur, c'est une description « schématique ». Souvent ce mode de description est utilisé au niveau le plus élevé de la hiérarchie, chaque composant étant lui-même défini par un programme VHDL (entité et architecture).

Description comportementale - le comportement matériel du circuit est décrit par un algorithme, indépendamment de la façon dont il est réalisé au niveau structurel.

Description par un flot de données - le fonctionnement du circuit est décrit par un flot de données qui vont des entrées vers les sorties, en subissant, étape par étape, des transformations élémentaires successives. Ce mode de description permet de reproduire l'architecture logique, en couches successives, des opérateurs combinatoires.

Flot de données et représentation comportementale sont très voisines, dans les deux cas le concepteur peut faire appel à des instructions de haut niveau. La première méthode utilise un grand nombre de signaux internes qui conduisent au résultat par des transformations de proche en proche, la seconde utilise des blocs de programme (les processus explicites), qui manipulent de nombreux signaux avec des algorithmes séquentiels[5].

La syntaxe générale d'une architecture comporte une partie de déclaration et un corps de programme :

```
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;
```

[5]On relira avec profit les trois descriptions d'une bascule D-Latch, qui sont données au paragraphe III.3.

```
        instanciation (i.e. importation
        dans un schéma) de composants.
end exemple ;
```

On se reportera aux exemples du chapitre III pour des illustrations simples.

VI.1.5 Des algorithmes séquentiels décrivent un câblage parallèle : les « PROCESSUS »

« Un processus est une instruction *concurrente* (N.D.T deux instructions concurrentes sont simultanées) qui définit un comportement qui doit avoir lieu quand ce processus devient actif. Le comportement est spécifié par une suite d'instructions *séquentielles* exécutées dans le processus. »[6]

Que cela signifie-t-il ?

Trois choses :

1. Les différentes parties d'une réalisation interagissent simultanément, peu importe l'ordre dans lequel un câbleur soude ses composants, le résultat sera le même. Le langage doit donc comporter une contrainte de « parallélisme » entre ses instructions. Cela implique des différences notables avec un langage procédural comme le C.
   En VHDL :
   ```
   a <= b ;
   c <= a + d ;
   ```
   et
   ```
   c <= a + d ;
   a <= b ;
   ```
   représentent la même chose, ce qui est notablement différent de ce qui se passerait en C pour :
   ```
   a = b ;
   c = a + d ;
   ```
   et
   ```
   c = a + d ;
   a = b ;
   ```
   Les affectations de signaux, à *l'extérieur* d'un processus explicite, sont traitées comme des processus tellement élémentaires qu'il est inutile de les déclarer comme tels. Ces affectations sont traitées en parallèle, de la même façon que plusieurs processus indépendants.

[6]WARP, VHDL Reference, Cypress Semiconductors.

2. L'algorithmique fait grand usage d'instructions séquentielles pour décrire le monde. VHDL offre cette facilité à *l'intérieur* d'un processus explicitement déclaré. Dans le corps d'un processus il sera possible d'utiliser des variables, des boucles, des conditions, dont le sens est le même que dans les langages séquentiels. Même les affectations entre signaux sont des instructions séquentielles quand elles apparaissent à l'intérieur d'un processus. Seul sera visible de l'extérieur le résultat final obtenu à la fin du processus.

3. Les opérateurs séquentiels, surtout synchrones, mais pas exclusivement eux, comportent « naturellement » la notion de mémoire, qui est le fondement de l'algorithmique traditionnelle. Les processus sont *la* représentation privilégiée de ces opérateurs[7]. Mais attention, *la réciproque n'est pas vraie*, il est parfaitement possible de décrire un opérateur purement combinatoire par un processus, le programmeur utilise alors de cet objet la seule facilité d'écriture de l'algorithme[8].

Outre les simples affectations de signaux, qui sont en elles mêmes des processus implicites à part entière, la description d'un processus obéit à la syntaxe suivante :

Processus : syntaxe générale

```
[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 :

---

[7]Ce n'est pas la seule, les descriptions structurelles et flot de données, plus proches du câblage du circuit, permettent de décrire tous les opérateurs séquentiels avec des opérateurs combinatoires élémentaires. Pour les circuits qui comportent des bascules comme éléments primitifs, connus de l'outil de synthèse, les deux seules façons d'utiliser ces bascules sont les process et leur instanciation comme composants dans une description structurelle.

[8]Voir à titre d'exemple la description que nous avons donnée du ou-exclusif comme contrôleur de parité, § III.2..

L'instruction `wait`

Cette instruction indique au processus que son déroulement doit être suspendu dans l'attente d'un événement sur un signal (un signal change de valeur), et tant qu'une condition n'est pas réalisée.

Sa syntaxe générale est[9] :

```
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*. La tendance, pour les évolutions futures du langage, semble être à la suppression des listes de sensibilités, pour n'utiliser que les instructions d'attente.

Description d'un opérateur séquentiel

La représentation des horloges : pour représenter les opérateurs synchrones de façon comportementale il *faut* introduire l'horloge dans la liste de sensibilité, *ou* insérer dans le code du processus une instruction « `wait` » explicite. Rappelons qu'il est interdit d'utiliser à la fois une liste de sensibilité et une instruction `wait`. Quand on modélise un opérateur qui comporte à la fois des commandes synchrones et des commandes asynchrones, il *faut*, avec certains compilateurs, mettre ces commandes dans la liste de sensibilité.

Exemple :

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

[9]On peut spécifier un temps d'attente maximum (wait ... `for` *`temps`*), mais cette clause n'est pas synthétisable.

```
            end if;
         end case;
      end if;
   end process;
   end fsm;
```

Dans l'exemple précédent, la priorité de la mise à zéro asynchrone, sur le fonctionnement synchrone normal de la bascule JK, apparaît par l'ordre des instructions de la structure `if...elsif`. Le processus est utilisé là à la fois pour modéliser un opérateur essentiellement séquentiel, la bascule, et pour faciliter la description de l'effet de ses commandes par un algorithme 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 : std_logic := '0' ; -- Librairie IEEE
begin
q <= etat;
process(clock) -- un seul signal d'activation
begin
   if rising_edge(clock) then  -- Librairie IEEE
      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 : std_logic := '0' ; -- Librairie IEEE
begin
q <= etat;
process  -- pas de liste de sensibilité
begin
wait until rising_edge(clock) ;
   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;
```

Description par un processus d'un opérateur combinatoire ou asynchrone

Un processus permet de décrire un opérateur purement combinatoire ou un opérateur séquentiel asynchrone, en utilisant une démarche algorithmique.

Dans ces deux cas la liste de sensibilité, ou l'instruction wait équivalente, est obligatoire ; le caractère combinatoire ou séquentiel de l'opérateur réalisé va dépendre du code interne au processus. On considère un signal qui fait l'objet d'une affectation dans le corps d'un processus :

- Si au bout de l'exécution du processus, pour *toutes* les combinaisons possibles des valeurs de la liste de sensibilité la valeur de ce signal, objet d'une affectation, est connue, l'opérateur correspondant est combinatoire.
- Si certaines des combinaisons précédentes de la liste de sensibilité conduisent à une indétermination concernant la valeur du signal examiné, objet d'une affectation, ce signal est associé à une cellule mémoire.
-

Précisons ce point par un exemple :

```
entity comb_seq is
port (
      e1, e2 : in std_logic ;
      s_et, s_latch, s_edge : out std_logic
      ) ;
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 inconnue.
end process ;

edge : process(e1) -- bascule D Edge, e1 est l'horloge.
begin
   if rising_edge(e1) then -- e1 agit par un front.
      s_edge <= e2 ;
   end if ;
end process ;
end exproc ;
```

Dans l'exemple qui précède, le premier processus est combinatoire, le signal s_et a une valeur connue à la fin du processus, quelles que soient les valeurs des entrées e1 et e2. Dans le deuxième processus, l'instruction « if » ne nous renseigne pas sur la valeur du signal s_latch quand e1 = '0'. Cette méconnaissance est interprétée, par le compilateur VHDL, comme un maintien de la valeur précédente, d'où la génération d'une cellule mémoire dont la commande de mémorisation, e1, est active sur un niveau. Le troisième processus conduit également, et pour le même type de raison, à la synthèse d'une cellule mémoire pour le signal s_edge. Mais la commande de mémorisation est, cette fois, active sur un front, explicitement mentionné dans la condition de l'instruction « if » : `e1'event`. La façon dont est traitée la commande de mémorisation e1 dépend donc de l'écriture du test : niveau ou front[10].

VI.2. Eléments du langage

VI.2.1 Les données appartiennent à une classe et ont un type

VHDL, héritier d'ADA, est un langage *fortement typé*. Toutes les données ont un type qui doit être déclaré avant l'utilisation[11] et aucune conversion de type automatique (une souplesse et un piège immense du C, par exemple) n'est effectuée. Pour passer du type entier au type bit_vector, par exemple, il faut faire appel à une fonction de conversion.

[10]Il peut y avoir des petites différences d'interprétation, suivant les compilateurs, entre les deux types de bascules, si on omet l'attribut 'event.
[11]Sauf, et c'est bien pratique, les variables entières des boucles « FOR ».

Une donnée appartient à une classe qui définit, avec son type, son comportement. Des données de deux classes différentes, mais de même type, peuvent échanger des informations directement : on peut affecter la valeur d'une variable à un signal, par exemple (nous verrons ci-dessous que variables et signaux sont deux classes différentes).

La portée des noms est, en général, locale. Un nom déclaré à l'intérieur d'une architecture, par exemple, n'est connu que dans celle-ci. Des objets globaux sont possibles, on peut notamment définir des constantes, comme `zero` ou `one`, extérieures aux unités de programmes que constituent les couples entité-architecture. A l'intérieur d'une architecture les objets déclarés dans un bloc (délimité par les mots-clés `begin` et `end`) sont visibles des blocs plus internes uniquement.

Les objets déclarés dans une entité sont connus de toutes les architectures qui s'y rapportent.

Les classes : signaux, variables et constantes

Signaux

Les signaux représentent les données physiques échangées entre des blocs logiques d'un circuit. Chacun d'entre eux sera matérialisé dans le schéma final par une *équipotentielle* et, éventuellement, une *cellule mémoire* qui conserve la valeur de l'équipotentielle entre deux commandes de changement. Les « ports » d'entrée et de sortie, attachés à une entité, par exemple, sont une variété de signaux qui permettent l'échange d'informations entre différentes fonctions. Leur utilisation est similaire à celle des arguments d'une procédure en PASCAL, le sens de transfert de l'information doit être précisé.

Syntaxe de déclaration (se place dans la partie déclarative d'une architecture[12]) :

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

Affectation d'une valeur (se place dans le corps d'une architecture ou d'un processus) :

```
nom <= valeur_compatible_avec_le_type ;
```

La valeur affectée peut être le résultat d'une expression, simple ou conditionnelle (when), ou la valeur renvoyée par l'appel d'une fonction.

A l'extérieur d'un processus toutes les affectations de signaux sont concurrentes, *c'est donc une erreur* (sémantique, pas syntaxique) *d'affecter plus d'une fois une valeur à un signal*. L'affectation d'une valeur à un signal traduit, en fait, la connexion de la sortie d'un opérateur à l'équipotentielle correspondante. Il

[12]ou d'un paquetage, voir plus loin.

s'agit là d'une opération permanente, une soudure sur une carte, par exemple, qu'il est hors de question de modifier ailleurs dans le programme. Si un signal est l'objet d'affectations multiples, ce qui revient à mettre en parallèle plusieurs sorties d'opérateurs (trois états ou collecteurs ouverts, par exemple), il faut adjoindre à ce signal, pour les besoins de la simulation, une fonction de résolution qui permet de résoudre le conflit[13].

Variables

Les variables sont des objets qui servent à stocker un résultat intermédiaire pour faciliter la construction d'un *algorithme séquentiel*. Elles ne peuvent être utilisées *que dans les processus, les procédures ou les fonctions*, et dans les boucles « generate » qui servent à créer des schémas répétitifs.

Syntaxe de déclaration (se place dans la partie déclarative d'un processus, d'une procédure ou d'une fonction) :

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

L'expression facultative qui apparaît dans la déclaration précédente permet de donner à une variable une valeur initiale choisie par l'utilisateur. A défaut de cette expression *le compilateur, qui initialise toujours les variables*[14], utilise une valeur par défaut qui dépend du type déclaré.

Affectation d'une valeur :

```
nom := valeur_compatible_avec_le_type ;
```

La valeur affectée peut être le résultat d'une expression ou la valeur renvoyée par l'appel d'une fonction.

Les variables de VHDL jouent le rôle des variables automatiques des langages procéduraux, comme C ou Pascal, elles ont une portée limitée au module de programme dans lequel elles ont été déclarées, et sont détruites à la sortie de ce module.

La différence entre variables et signaux est que les premières n'ont pas d'équivalent physique dans le schéma, contrairement aux seconds. Certains outils de synthèse ne respectent malheureusement pas cette distinction. On notera qu'il est possible d'affecter la valeur d'une variable à un signal, et inversement, pourvu que les types soient compatibles.

Constantes

Les constantes sont des objets dont la valeur est fixée une fois pour toute.

---

[13]Dans les applications de synthèse les portes à sorties non standard sont généralement introduites dans une description structurelle.

[14]Le programmeur ne doit, notamment, pas s'attendre à retrouver les variables d'un processus dans l'état où il les avait laissées lors d'une activation précédente de ce processus.

Exemples de valeurs constantes simples :

```
'0', '1', "01101001", 1995, "azerty"
```

On peut créer des constantes nommées :

```
constant nom1 : type [ := valeur_constante ] ;
```

On notera que les vecteurs de bits (bit_vector) sont traités comme des chaînes, on peut préciser une base différente de la base 2 pour ces constantes :

```
X"3A007", O"237015"  pour hexadécimal et octal.
```

De même, 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.
```

En général les nombres flottants ne sont pas acceptés par les outils de synthèse.

Des types adaptés à l'électronique numérique

VHDL connaît un nombre limité de types de base, qui reflètent le fonctionnement des systèmes numériques (pour l'instant, VHDL est en passe de devenir un langage de description des circuits analogiques), et offre à l'utilisateur de construire à partir de ces types génériques :

- des sous-types (sous-ensembles du type de base), obtenus en précisant un domaine de variation limité de l'objet considéré,
- des types composés, obtenus par la réunion de plusieurs types de base identiques (tableaux) ou de types différents (enregistrements).

En plus des types prédéfinis et de leurs dérivés, l'utilisateur a la possibilité de créer ses propres types sous forme de types énumérés.

Les entiers

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

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

Attention, sur les PC qui sont des machines dont les entiers continuent à hésiter entre 16 et 32 bits, l'utilisateur peut rencontrer de désagréables surprises.

Les nombres négatifs ne sont pas toujours acceptés dans la description des signaux physiques.

Déclaration :

```
signal nom : integer ;
```

ou

```
variable nom : integer ;
```

ou encore :

```
constant nom : integer ;
```

que l'on résume classiquement par :

```
signal | variable |constant nom : integer ;
```

Le symbole | signifiant « ou ».

On peut spécifier une plage de valeurs inférieure à celle obtenue par défaut, par exemple :

```
signal etat : integer range 0 to 1023 ;
```

permet de créer un compteur 10 bits.

La même construction permet de créer un sous-type :

```
subtype etat_10 is integer range 0 to 1023 ;
signal etat1 , etat2 : etat_10 ;
```

Attention ! La restriction d'étendue de variation est utilisée pour générer le nombre de chiffres binaires nécessaires à la représentation de l'objet, l'arithmétique sous-jacente n'est (pour l'instant) pas traitée par les compilateurs. Cela veut dire que

```
signal chiffre : integer range 0 to 9 ;
```

permet de créer un objet codé sur quatre bits, mais

```
chiffre <= chiffre + 1 ;
```

ne crée *pas* un compteur décimal.

Pour ce faire il faut écrire explicitement :

```
if chiffre < 9 then
   chiffre <= chiffre + 1 ;
else
   chiffre <= 0 ;
end if ;
```

La déclaration, au niveau le plus élevé d'une hiérarchie, de ports d'entrée ou de sortie comme nombres entiers pose un problème de contrôle par l'utilisateur de l'assignation des broches physiques du circuit final aux chiffres binaires générés. Cette assignation sera faite automatiquement par l'outil de développement. Si ce non contrôle est gênant, il est possible de transformer un nombre entier en tableau de bits, via les fonctions de conversion de la librairie associée à un compilateur.

Les types énumérés

L'utilisateur peut créer ses propres types, par simple énumération de constantes symboliques qui fixent toutes les valeurs possibles du type. Par exemple :

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

Les bits

Il s'agit là, évidemment, du type de base le plus utilisé en électronique numérique. Un objet de type bit peut prendre deux valeurs : '0' et '1', il s'agit, en fait, d'un type énuméré prédéfini.

Déclaration :

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

Pour les besoins de la simulation et des connexions en bus, la librairie IEEE propose un type bit plus étoffé (`std_logic`), pouvant prendre, entre autres, les valeurs '0', '1', 'X' (X pour inconnu) et 'Z' (pour haute impédance). Ce type est traduit, en synthèse, par desvaleurs binaires ordinaires plus l'état « déconnecté » :

```
Library IEEE ;
use IEEE.std_logic_1164.all ;

entity basc_tri_state is
  port( clk, oe : in std_logic ;
      sort : inout std_logic );
       -- type std_logic défini dans la librairie
end basc_tri_state ;
```

Ces extensions sont portables tant en simulation qu'en synthèse.

Les booléens

Autre type énuméré, le type booléen peut prendre deux valeurs : "`true`" et "`false`". Il intervient essentiellement comme résultat d'expressions de comparaisons, dans des `IF`, par exemple, ou dans les valeurs renvoyées par des fonctions.

Les tableaux

A partir de chaque type de base on peut créer des tableaux, collection d'objets du même type. L'un des plus utilisés est le type bit_vector, défini dans la librairie standard par :

```
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) ;
```

définit un tableau de cinq éléments binaires nommé état.
On aurait également pu définir directement un sous-type :

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

Le nombre de dimensions d'un tableau n'est pas limité, les indices peuvent être définis dans le sens croissant (2 to 6) ou décroissant (6 downto 2) avec des bornes quelconques (mais cohérentes avec le sens choisi).

On notera qu'il *faut* passer par une définition de type, ce qui n'est pas le cas en C ou en PASCAL.

Une fois défini, un objet composé peut être manipulé collectivement par son nom :

```
signal etat1 : bit_vector (0 to 4) ;
variable etat2 : bit_vector (0 to 4) ;
...
etat 1 <= etat2 ;  -- parfaitement correct
```

Le compilateur contrôle que les dimensions des deux objets sont les mêmes. On remarquera, à partir de l'exemple précédent, que les classes des deux objets peuvent être différentes.

On peut, bien sûr, ne manipuler qu'une partie des éléments d'un tableau :

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

```
signal sous_etat : bit_vector (0 to 1) ;
signal flag : bit ;
...
sous_etat <= etat ( 1 to 2 ) ;
flag      <= etat ( 3 ) ;
```

Il est possible de fusionner deux tableaux (concaténation) pour affecter les valeurs correspondantes à un tableau plus grand :

```
signal etat : bit_vector (0 to 4) ;
signal sous_etat2 : bit_vector (0 to 1) ;
signal sous_etat3 : bit_vector (0 to 2) ;
...
etat <= sous_etat2 & sous_etat3; -- concaténation.
```

Les enregistrements

Les enregistrements (record) définissent des collections d'objets de types, ou de sous types, différents. Ils correspondent aux structures du C ou aux enregistrements de PASCAL.

Définition d'un type :

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

Déclaration d'un objet de ce type :

```
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 ;
```

L'ensemble d'un enregistrement peut être manipulé par son nom.

VI.2.2 Les attributs précisent les propriétés des objets

Déterminer, de façon dynamique, la taille d'un tableau, le domaine de définition d'un objet scalaire, l'élément suivant d'un type énuméré, détecter la transition montante d'un signal, piloter l'optimiseur d'un outil de synthèse, attribuer des numéros de broches à des signaux d'entrées-sorties ... etc.

Les attributs permettent tout cela.

Un attribut est une propriété, qui porte un nom, associée à une entité, une architecture, un type ou un signal. Cette propriété, une fois définie, peut être utilisée dans des expressions.

L'utilisation d'un attribut se fait au moyen d'un nom composé : le préfixe est le nom de l'objet auquel est rattaché l'attribut, le suffixe est le nom de l'attribut. Préfixe et suffixe sont séparés par une apostrophe « ' ».

```
nom_objet'nom_de_l_attribut
```

Par exemple :

`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.

Certains attributs sont prédéfinis par le langage, d'autres sont attachés à un outil de développement ; l'utilisateur, enfin, peut définir, et utiliser ses propres attributs.

Attributs prédéfinis dans le langage

Les attributs prédéfinis permettent de déterminer les contraintes qui pèsent sur des objets ou des types : domaine de variation d'un type scalaire, bornes des indices d'un tableau, éléments voisins d'un objet de type énuméré, etc.

Ils permettent également de préciser les caractéristiques dynamiques de signaux, comme la présence d'un front, évoquée précédemment.

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
‘length(n)	type tableau	nombre d’éléments de la dimension n, n=1 par défaut
‘pos(v)	type scalaire	position de l’élément v dans le type
‘val(p)	type scalaire	valeur de l’élément de position p dans le type
‘succ(v)	type scalaire	valeur qui suit (position + 1) l’élément de valeur v dans le type
‘pred(v)	type scalaire	valeur qui précède (pos. – 1) l’élément de valeur v dans le type
‘leftof(v)	type scalaire	valeur de l’élément juste à gauche de l’élément de valeur v
‘rightof(v)	type scalaire	valeur de l’élément juste à droite de l’élément de valeur v
‘event	signal	valeur booléenne "TRUE" si la valeur du signal vient de changer
‘base	tous types	renvoie le type de base d’un type dérivé
‘range(n)	type tableau	renvoie la plage de variation de l’indice de la dimension n, défaut n=1, dans une boucle : "for i in bus’range loop..."
‘reverse_ range(n)	type tableau	renvoie la plage de variation, retournée (to ↔ downto), de l’indice de la dimension n, défaut n=1

Le tableau ci-dessus précise le nom de quelques uns des attributs les plus utilisés, les catégories d’objets qu’ils permettent de qualifier et la valeur renvoyée par l’attribut.

Attributs spécifiques à un système

Chaque système de développement fournit des attributs qui aident à piloter l’outil de synthèse, ou le simulateur, associé au compilateur VHDL.

Ces attributs, qui ne sont évidemment pas standard, portent souvent sur le pilotage de l’optimiseur, permettent de passer au routeur des informations concernant le brochage souhaité, ... etc.

Par exemple :

```
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.

Attributs définis par l'utilisateur

Syntaxe :

déclaration

```
attribute att_nom : type ;
```

spécification

```
attribute nom_att of nom_objet:nom_classe is expression ;
```

utilisation

```
nom_objet'att_nom
```

VI.2.3 Les opérateurs élémentaires

Les opérateurs connus du langage sont répartis en six classes, en fonction de leurs priorités. Dans chaque classe les priorités sont identiques ; les parenthèses permettent de modifier l'ordre d'évaluation des expressions, modifiant ainsi les priorités, et sont obligatoires lors de l'utilisation d'opérateurs non associatifs comme « `nand` ». Le tableau ci-dessous fournit la liste des opérateurs classés par priorités croissantes, de haut en bas :

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 tableaux (concaténation)	numérique tableau
Signe	+ –	numériques	numérique
Opérateurs multiplicatifs	* / mod rem	numériques (restrictions) entiers (restrictions)	numérique entier
Op. divers	not abs **	bit ou booléen numérique numériques (restrictions)	bit ou booléen numérique numérique

Ce tableau appelle quelques 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 standard (int_math et bv_math) surdéfinissent (au sens des langages objets) les opérateurs d'addition et de soustraction pour les étendre au type bit_vector.
- 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.
- La priorité intermédiaire des opérateurs unaires de signe interdit l'écriture d'expressions comme
  « a * –b », qu'il faut écrire « a * (–b) ».

VI.2.4 Instructions concurrentes

Les instructions concurrentes interviennent à l'intérieur d'une architecture, dans la description du fonctionnement d'un circuit. En raison du parallélisme du langage, ces instructions peuvent être écrites dans un ordre quelconque. Les principales instruction concurrentes sont :

- les affectations concurrentes de signaux,
- les « processus » (décrits précédemment),
- les instanciations de composants
- les instructions « generate »
- les définitions de blocs.

Affectations concurrentes de signaux

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

L'affectation conditionnelle permet de déterminer la valeur de la cible en fonction des résultats de tests logiques :

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

On notera un danger de confusion entre l'opérateur d'affectation et l'un des opérateurs de comparaison, l'instruction suivante est syntaxiquement juste, mais fournit vraisemblablement un résultat fort différent de celui escompté par son auteur :

```
-- Résultat bizarre :
cible <= source_1 when condition else
cible <= source_2; -- <= est ici une comparaison !
       -- dont le résultat est affecté à
       -- cible si la condition est fausse.
```

Affectation sélective

En fonction des valeurs possibles d'*une* expression, il est possible de choisir la valeur à affecter à un signal :

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

Un exemple typique d'affectation sélective est la description d'un multiplexeur.

Instanciation de composant

Le mécanisme qui consiste à utiliser un sous-ensemble (une paire entité-architecture), décrit en VHDL, comme composant dans un ensemble plus vaste est connu sous le nom d'*instanciation*. Trois opérations sont nécessaires :

- Le couple entité-architecture du sous-ensemble doit être créé et annexé à une librairie de l'utilisateur, par défaut la librairie « work ».
- Le sous-ensemble précédent doit être déclaré comme composant dans l'ensemble qui l'utilise, cette déclaration reprend les éléments principaux de l'entité du sous-ensemble.
- Chaque exemplaire du composant que l'on souhaite inclure dans le schéma en cours d'élaboration doit être connecté aux équipotentielles de ce schéma, c'est le mécanisme de l'instanciation.

Syntaxe de la déclaration[15] :

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

[15]Simplifiée, nous omettons volontairement ici la possibilité de créer des composants « génériques », c'est à dire dont certains paramètres peuvent être fixés au moment de l'instanciation, une largeur de bus, par exemple.

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 d'un composant :

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

La liste d'association établit la correspondance entre les équipotentielles du schéma et les ports d'entrée et de sortie du composant. Cette association peut se faire par position, les noms des signaux à connecter doivent apparaître dans l'ordre des ports auxquels ils doivent correspondre, ou explicitement au moyen de l'opérateur d'association « => » :

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

begin
....
-- utilisation :
et1 : et port map ( s_a , s_b , s_a_et_b ) ;
-- ou :
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 dans les cas très simples. L'association explicite est préférable dès que le nombre de ports dépasse deux ou trois.

Generate

Les instructions « generate » permettent de créer de façon compacte des structures régulières, comme les registres ou les multiplexeurs. Elles sont particulièrement efficaces dans des descriptions structurelles.

Une instruction generate permet de dupliquer une 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] ;
```

Donnons à titre d'exemple le code d'un compteur modulo 16, construit au moyen de bascules T, disposant d'une remise à zéro (raz) et d'une autorisation de comptage (en) actives à '1' :

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL ;

ENTITY cnt16 IS
      PORT (ck,raz,en : IN STD_LOGIC;
         s : OUT STD_LOGIC_VECTOR (0 TO 3) );
END cnt16;

ARCHITECTURE struct OF cnt16 IS
  SIGNAL etat : STD_LOGIC_VECTOR(0 TO 3) := (OTHERS => '0');
  SIGNAL inter: STD_LOGIC_VECTOR(0 TO 3) := (OTHERS => '0');
COMPONENT T_edge -- supposé présent dans la librairie work
      port ( T,hor,zero : in std_logic;
         s : out std_logic);
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;
```

Block

Une architecture peut être subdivisée en blocs, de façon à constituer une hiérarchie interne dans la description d'un composant complexe.

Syntaxe :

```
etiquette : block [( expression_de_garde )]
-- zone de déclarations de signaux, composants, etc...
begin
-- instructions concurrentes
end block [etiquette] ;
```

Dans des applications de synthèse, l'intérêt principal des blocs est de permettre de contrôler la portée et la visibilité des noms des objets utilisés (signaux notamment) : un nom déclaré dans un bloc est local à celui-ci. Dans des applications de simulation les blocs permettent en outre de contrôler les instructions qu'ils contiennent par une expression « de garde », de type booléen[16].

VI.2.5 Instructions séquentielles

Les instructions séquentielles sont *internes* aux processus, aux procédures et aux fonctions (pour les deux dernières constructions voir paragraphes suivants). Elles permettent d'appliquer à la description d'une partie d'un circuit une démarche algorithmique, même s'il s'agit d'une fonction purement combinatoire. Les principales instructions séquentielles sont :

- L'affectation séquentielle d'un signal, qui utilise l'opérateur « <= », a une syntaxe qui est identique à celle de l'affectation concurrente simple. Seule la place, dans ou hors d'un module de programme séquentiel, distingue les deux types d'affectation ; cette différence, qui peut sembler mineure, cache des comportements différents : alors que les affectations concurrentes peuvent être écrites dans un ordre quelconque, pour leurs correspondantes séquentielles, rarement utilisées hors d'une structure de contrôle, l'ordre d'écriture n'est pas indifférent.
- L'affectation d'une variable, qui utilise l'opérateur « := », est *toujours* une instruction séquentielle.
- Les tests « if » et « case ».
- Les instructions de contrôle des boucles « loop », « for » et « while ».

[16]Les expressions de garde ne sont pas gérées par tous les compilateurs.

Les instructions de test

Les instructions de tests permettent de sélectionner une ou des instructions à exécuter, en fonction des valeurs prises par une ou des expressions. On notera que, dans un processus, si toutes les branches possibles des tests ne sont pas explicitées, une cellule mémoire est générée pour chaque affectation de signal.

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

L'instruction `if` permet de sélectionner une ou des instructions à exécuter, en fonction des valeurs prises par *une ou des* conditions.

Syntaxe :

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

Son interprétation est la même que dans les langages de programmation classiques comme C ou Pascal.

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

L'instruction `case` permet de sélectionner une ou des instructions à exécuter, en fonction des valeurs prises par *une* expression.

Syntaxe :

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

« | choix », pour « ou ... », et « when others » sont syntaxiquement facultatifs. Les choix représentent différentes valeurs possibles de l'expression testée ; on notera que *toutes* les valeurs possibles doivent être traitées, soit explicitement, soit par l'alternative « others ». Chacune de ces valeurs ne peut apparaître que dans une seule alternative.

Cette instruction est à rapprocher du « switch » de C, ou de « case of » de Pascal.

Les boucles

Les boucles permettent de répéter une séquence d'instructions.

Syntaxe générale

```
[ etiquette : ] [ schéma itératif ] loop
séquence d'instructions
end loop [ etiquette ] ;
```

Trois catégories de boucles existent en VHDL, suivant le schéma d'itération choisi :

- Les boucles simples, sans schéma d'itération, dont on ne peut sortir que par une instruction « exit ».
- Les boucles « for », dont le schéma d'itération précise le nombre d'exécution.
- Les boucles « while », dont le schéma d'itération précise la condition de maintien dans la boucle.

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.

VI.3. Programmation modulaire

Small is beautyfull, un gros programme ne peut être écrit, compris, testable et testé que s'il est subdivisé en petits modules que l'on met au point indépendamment les uns des autres et rassemblés ensuite. VHDL offre, bien évidemment, cette possibilité.

Chaque module peut être utilisé dans plusieurs applications différentes, moyennant un ajustage de certains paramètres, sans avoir à en réécrire le code. Les outils de base de cette construction modulaire sont les sous programmes, procédures ou fonctions, les *paquetages* et librairies, et les paramètres génériques.

VI.3.1 Procédures et fonctions

Les sous programmes sont le moyen par lequel le programmeur peut se constituer une bibliothèque *d'algorithmes séquentiels* qu'il pourra inclure dans une description. Les deux catégories de sous programmes, procédures et fonctions, diffèrent par les mécanismes d'échanges d'informations entre le programme appelant et le sous programme.

Les fonctions

Une fonction retourne au programme appelant une valeur unique, elle a donc un type. Elle peut recevoir des arguments, exclusivement des signaux ou des constantes, dont les valeurs lui sont transmises lors de l'appel. Une fonction ne peut en aucun cas modifier les valeurs de ses arguments d'appel.

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 ] ;
```

Le corps d'une fonction ne peut pas contenir d'instruction `wait`, les variables locales, déclarées dans la fonction, cessent d'exister dès que la fonction se termine.

Utilisation :

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

Lors de son utilisation, le nom d'une fonction peut apparaître partout, dans une expression, où une valeur du type correspondant peut être utilisée.

Exemple

Les librairies d'un compilateur VHDL contiennent un grand nombre de fonctions, dont le programme source est souvent fourni. L'exemple qui suit incrémente de 1 un vecteur d'éléments std_logic. On peut l'utiliser, par exemple, pour créer un compteur binaire.

Déclaration :

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL ;

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

Corps de la fonction :

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL ;

FUNCTION inc_bv (a : STD_LOGIC_VECTOR)
         RETURN STD_LOGIC_VECTOR IS
VARIABLE s : STD_LOGIC_VECTOR (a'RANGE);
VARIABLE carry : STD_LOGIC ;
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 rising_edge(clk);
   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

Une procédure, comme une fonction, peut recevoir du programme appelant des arguments : constantes, variables ou signaux. Mais ces arguments peuvent être déclarés de modes « in », « inout » ou « out » (sauf les constantes qui sont toujours de mode « in »), ce qui autorise une procédure à renvoyer un nombre quelconque de valeurs au programme appelant.

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 ) ;
```

Le corps d'une procédure peut contenir une instruction `wait`, les variables locales, déclarées dans la procédure, cessent d'exister dès que la procédure se termine.

Utilisation :

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

Une procédure peut être appelée par une instruction concurrente ou par une instruction séquentielle, mais si l'un de ses arguments est une variable, elle ne peut être appelée que par une instruction séquentielle. La correspondance entre paramètres réels (dans l'appel) et paramètres formels (dans la description de la procédure) peut se faire par position, ou par associations de noms :

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

Ou :

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

VI.3.2 Les paquetages et les librairies

Un `paquetage` permet de rassembler des déclarations et des sous programmes, utilisés fréquemment dans une application, dans un module qui peut être compilé à part, et rendu visible par l'application au moyen de la clause `use`.

Un `paquetage` est constitué de deux parties : la déclaration, et le corps (*body*).

- La déclaration contient les informations publiques dont une application a besoin pour utiliser correctement les objets décrits par le paquetage[17] : essentiellement des déclarations, des définitions de types, des définitions de constantes ...etc.
- Le corps, qui n'existe pas obligatoirement, contient le code des fonctions ou procédures définies par le paquetage, s'il en existe.

L'utilisation d'un paquetage se fait au moyen de la clause `use` :

```
use work.mes_fonctions.all; -- rend le paquetage
      -- mes_fonctions, de la librairie work,
      -- visible dans sa totalité.
```

Le mot clé `work` indique l'ensemble des librairies accessibles, par défaut, au programmeur. Ce mot cache, notamment, des chemins d'accès à des répertoires de travail. Ces chemins sont gérés par le système de développement, et l'utilisateur n'a pas besoin d'en connaître les détails. Le nom composé qui suit la clause use doit être compris comme une suite de filtres : « utiliser tous les éléments du module int_math de la librairie work ».

Les paquetages prédéfinis

Un compilateur VHDL est toujours assorti de librairies, décrites par des paquetages, qui offrent à l'utilisateur des outils variés :

- Définitions de types, et fonctions de conversions entre types : VHDL est un langage objet, fortement typé. Aucune conversion de type implicite n'est

[17]On peut rapprocher la partie visible d'un package des fichiers « *.h » du langage C, ces fichiers contiennent, entre autres, les prototypes des objets, variables ou fonctions, utilisés dans un programme. La clause « use » de VHDL est un peu l'équivalent, dans cette comparaison, de la directive #include <xxx.h> du C.

autorisée dans les expressions, mais une librairie peut offrir des fonctions de conversion explicites, et redéfinir les opérateurs élémentaires pour qu'ils acceptent des opérandes de types variés. Un bus, par exemple, peut être vu, dans le langage, comme un vecteur (tableau à une dimension) de bits, et il est possible d'étendre les opérateurs arithmétiques et logiques élémentaires pour qu'ils agissent sur un bus, vu comme la représentation binaire d'un nombre entier (package `numeric_std` de la librairie IEEE).

- Les blocs structurels des circuits programmables, notamment les cellules d'entrées–sorties, peuvent être déclarés comme des composants que l'on peut inclure dans une description. Une porte trois états, par exemple, sera vue, dans une architecture, comme un composant dont l'un des ports véhicule des signaux de type particulier : aux deux états logiques vient se rajouter un état haute impédance. L'emploi d'un tel opérateur dans un schéma nécessite, outre la description du composant, une fonction de conversion entre signaux logiques et signaux « trois états ».
- Un simulateur doit pouvoir résoudre, ou indiquer, les conflits éventuels. Les signaux utilisés en simulation ne sont pas, pour cette raison, de type binaire : on leur attache un type énuméré plus riche qui rajoute aux simples valeurs '0' et '1' la valeur 'inconnue', des nuances de force entre les sorties standard et les sorties collecteur ouvert, etc. (librairie IEEE)
- La bibliothèque standard offre également des procédures d'usage général comme les moyens d'accès aux fichiers, les possibilités de dialogue avec l'utilisateur, messages d'erreurs, par exemple.

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 paquetages créés par l'utilisateur

L'utilisateur peut créer ses propres paquetages. Cette possibilité permet d'assurer la cohérence des déclarations dans une application complexe, évite d'avoir à répéter un grand nombre de fois ces mêmes déclarations et donne la possibilité de créer une librairie de fonctions et procédures adaptée aux besoins des utilisateurs.

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] ;
```

Dans l'exemple qui suit, on réalise un compteur au moyen de deux bascules, dans une description structurelle. La déclaration du composant bascule est mise dans un paquetage :

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL ;

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

Le compteur proprement dit :

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL ;

ENTITY cnt2 IS
   PORT (ck,razero,en : IN STD_LOGIC;
                    s : OUT STD_LOGIC_VECTOR (0 TO 1)
      );
END cnt2;
```

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

ARCHITECTURE struct OF cnt2  IS
   SIGNAL etat : STD_LOGIC_VECTOR(0 TO 1) := "00";
   signal inter: std_logic := '0' ;

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 librairies

Une librairie est une collection de modules VHDL qui ont déjà été compilés. Ces modules peuvent être des paquetages, des entités ou des architectures.

Une librairie par défaut, work, est systématiquement associée à l'environnement de travail de l'utilisateur. Ce dernier peut ouvrir ses propres librairies par la clause `library` :

```
library nom_de_la_librairie ;
```

La façon dont on associe un nom de librairie à un, ou des, chemins, dans le système de fichiers de l'ordinateur, dépend de l'outil de développement utilisé.

VI.3.3 Les paramètres génériques

Lorsque l'on crée le couple entité-architecture d'un opérateur, que l'on souhaite utiliser comme composant dans une construction plus large, il est parfois pratique de pouvoir laisser certains paramètres modifiables par le programme qui utilise le composant. De tels paramètres, dont la valeur réelle peut n'être fixée que lors de l'instanciation du composant, sont appelés 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 ] ) ;
```

La même déclaration doit apparaître dans la déclaration de composant, mais au moment de l'instanciation la taille peut être modifiée par une instruction « generic map », de construction identique à l'instruction « port map », précédemment rencontrée :

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

Dans l'exemple ci-dessous, on réalise un compteur, sur 4 bits par défaut, qui est ensuite instancié comme un compteur 8 bits. Bien évidemment, le code du compteur ne doit faire aucune référence explicite à la valeur par défaut.

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.ALL ;

entity compteur is
generic (taille : integer := 4 ) ;
   port ( hor : in std_logic ;
       sortie : out std_logic_vector(taille – 1 downto 0));
end compteur ;

architecture simple of compteur is
   signal etat : unsigned(taille – 1 downto 0) ;
begin
sortie <= std_logic_vector(etat) ;
process
begin
   wait until rising_edge(hor) ;
   etat <= etat + 1 ;
end process ;
end simple ;
```

Ce compteur est instancié comme un compteur 8 bits :

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.ALL ;

entity compt8 is
   port (ck : in std_logic ;
        val : out std_logic_vector(7 downto 0) ) ;
end compt8 ;

architecture large of compt8 is
component compteur
   generic (taille : integer ) ;
   port ( hor : in std_logic ;
       sortie : out std_logic_vector(taille – 1 downto 0));
end component ;
begin
u1 : compteur
   generic map (8)
   port map (ck , val) ;
end large ;
```

Les paramètres génériques prennent toute leur efficacité quand leur emploi est associé à la création de librairies de composants, décrits par des paquetages. Il est alors possible de créer des fonctions complexes au moyen de programmes construits de façon hiérarchisée, chaque niveau de la hiérarchie pouvant être mis au point et testé indépendamment de l'ensemble.

VI.4. En guise de conclusion

VHDL est un langage qui peut déconcerter, au premier abord, le concepteur de systèmes numériques, plus habitué aux raisonnements traditionnels sur des schémas que familier des langages de description abstraite. Il est vrai que le langage est complexe, et peut présenter certains pièges, la description des horloges en est un exemple. Nous espérons avoir aidé le lecteur à gagner un peu de temps dans sa découverte, et lui avoir mis en évidence quelques uns des chausse-trappes classiques.

Ayant fait l'effort de « rentrer dedans », l'utilisateur découvre que ce type d'approche est d'une très grande souplesse, et d'une efficacité redoutable. Des problèmes de synthèse qui pouvaient prendre des heures de calcul, dans une démarche traditionnelle, sont traités en quelques lignes de programme.

Renouvelons ici la mise en garde du début de ce chapitre : n'oubliez jamais que vous êtes en train de créer un circuit, et que le meilleur des compilateurs ne peut que traduire la complexité sous-jacente de vos équations, il n'augmentera pas la capacité de calcul des circuits que vous utilisez. Le simple programme de description d'un additionneur 4 bits, comme le 74_283 :

```
ENTITY addit IS
PORT   (a,b: IN INTEGER RANGE 0 TO 15;
        cin: IN INTEGER RANGE 0 TO 1;
        som: OUT INTEGER RANGE 0 TO 31);
END addit;

ARCHITECTURE behavior OF addit IS

BEGIN
   som <= a+b+cin;
END behavior;
```

génère plus d'une centaine de termes, quand ses équations sont ramenées brutalement à une somme de produits logiques. Charge reste à l'utilisateur de piloter l'optimiseur de façon un peu moins sommaire que de demander la réduction de la somme à une expression canonique en deux couches logiques.

Exercices

Du code VHDL au schéma.

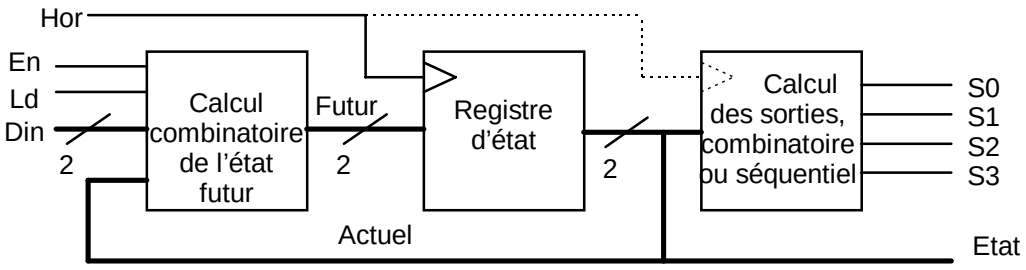
On considère le programme VHDL suivant qui décrit le fonctionnement d'une bascule :

```
entity basc is
port ( T,hor,raz : in bit;
      s : out bit);
end basc;

architecture primitive of basc is
signal etat : bit;
begin
s <= etat ;
process
begin
wait until (hor = '1') ;
if(raz = '1') then
      etat <= '0';
elsif(T = '1') then
      etat <= not etat;
end if;
end process;
end primitive;
```

1. A quoi reconnait-on qu'il s'agit d'un circuit séquentiel synchrone ?
2. La commande « raz » est-elle synchrone ou asynchrone ?
3. Etablir le diagramme de transition de cette bascule.
4. Déduire du diagramme précédent les équations logiques et le schéma d'une réalisation avec une bascule D, avec une bascule J-K.

compteur modulo 4 à sorties décodées.



Le code d'une première version du compteur est le suivant :

```
entity compteur is
  port ( hor, en, ld: in bit; din: in bit_vector(1 downto 0) ;
  etat:out bit_vector(1 downto 0);s:out bit_vector(0 to 3) );
end compteur ;
use work.int_math.all;
architecture comporte of compteur is
signal actuel : bit_vector(1 downto 0) ;
begin
etat <= actuel ;        -- instruction 1
with actuel select            -- instruction 2
   s <= "1000" when "00", "0100" when "01",
        "0010" when "10", "0001" when "11"; -- fin instruction 2
process                       -- instruction 3
begin
      wait until (hor = '1');
      if(ld = '1') then       actuel <= din ;
      elsif (en = '1') then   actuel <= actuel + 1 ;
      end if ;
end process ;                 -- fin instruction 3
end comporte ;
```

- Peut-on permuter les instructions (ou blocs d'instructions) 1, 2 et 3 ?
- **Dans** le processus peut-on permuter les instructions, même si on veille à conserver une syntaxe correcte ?
- Réécrire l'instruction 2 en utilisant une affectation conditionnelle au lieu d'un sélecteur parallèle.
- Réécrire le programme en créant trois processus qui respectent le découpage donné dans le synoptique précédent.
- Réécrire le programme précédent en calculant les sorties s(i) dans un processus synchrone, mais en veillant à ce que leurs valeurs restent identiques à celles décrites précédemment (le piège réside dans un éventuel décalage d'une période d'horloge).