Synthèse des CPLD et FPGA en VHDL : aide mémoire

Le VHDL (**V**ery high speed - or scale - integrated circuits **H**ardware **D**escription **L**anguage) est utilisé pour la modélisation et la synthèse de circuits numériques.
La modélisation permet de simuler le comportement d'un circuit numérique, tandis que la synthèse permet de le programmer (si nécessaire).
Lors du développement d'un CPLD ou d'un FPGA, la simulation du comportement de ce dernier est effectuée à l'aide d'un modèle. Celui-ci est déterminé directement par le logiciel programmation (fourni par un constructeur de CPLD) utilisé, à partir de la description VHDL entrée (simulation fonctionnelle) et en fonction du composant cible et de ses temps de propagation (simulation temporelle). Il reste seulement à définir les chronogrammes des signaux d'entrée du composant pour vérifier que le fonctionnement est bien celui souhaité. On constate que la synthèse d'un composant programmable ne nécessite pas de connaissances particulières en modélisation.
Nous ne nous intéresserons donc ici qu'à la synthèse. Il ne faudra pas chercher dans ce qui suit des instructions tel que l'affectation après un retard (s<=a after 5 ns par exemple, ce qui n'est pas synthétisable) ou une explication complète du type « std_logic ».
Cet aide mémoire n'a pas la prétention d'être exhaustif, le but étant simplement de rappeler le minimum que doit connaître un électronicien pour lire le VHDL (de synthèse). Aucun exemple concret n'est donné dans la partie principale, on se reportera pour cela à la bibliographie et à l'annexe.
Depuis sa création, le VHDL a fait l'objet de deux normalisations, une en 87 et l'autre en 93. Quelques différences existent entre les deux normes.
Il faut également avoir à l'esprit que l'on trouvera également des différences, parfois importantes, entre les divers outils de synthèse proposés par les constructeurs. L'opération de multiplication par exemple n'est définie que comme une multiplication par 2 par de nombreux compilateurs. Il est conseillé de se référencer à la documentation constructeur.

Ce langage ne fait théoriquement pas de distinction entre majuscules et minuscules. Certains outils de synthèse font cependant une distinction pour quelques cas particuliers (description de l'état haute impédance 'Z' – et non 'z' - avec le logiciel Max+plus II par exemple). Pour une meilleure lisibilité les mots réservés ont été écrits ici en minuscules et en gras, les noms donnés par l'utilisateur en majuscules.
En VHDL les commentaires sont précédés par « **--** » et s'arrêtent au retour à la ligne. Pour une meilleure lisibilité, ils sont écrits ici en italique.

1.Notions de base

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

```
------------------------------------------------------------------------------------------------------------
-- déclaration des ressources externes.
        library NOM_DE_LA_BIBLIOTHEQUE ;
        use ELEMENT_DE_LA_BIBLIOTHEQUE ;

-- description de l'entité vue comme une « boîte noire » avec des entrées et des
-- sorties, caractérisée par des paramètres
        entity NOM_DE_L'ENTITE is
                generic (PARAMETRES : « type »:=VALEURS_FACULTATIVES ) ;
                port ( NOM_DES_ENTREES_SORTIES : « direction » « type ») ;
        end NOM_DE_L'ENTITE ;
```

les ressources externes

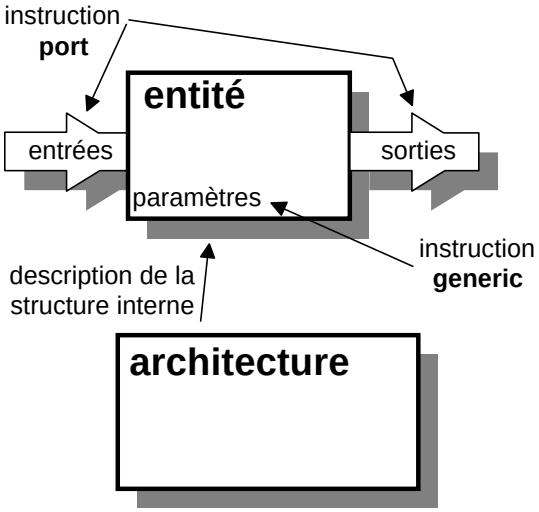
l'entité

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

        architecture NOM_DE_L'ARCHITECTURE of NOM_DE_L'ENTITE is

-- déclaration si nécessaire des objets internes
        NOM_DES_OBJETS_INTERNES :  « type »;
-- déclaration si nécessaire des types
        type NOM_DES_TYPES is (VALEURS_POSSIBLES) ;
-- déclaration si nécessaire des composants
        component NOM_DU_COMPOSANT port (ENTREES :in bit ;  SORTIES :out bit) ;

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

l'architecture

les ressources internes de l'architecture

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



On pourra se reporter au premier exemple de l'annexe 2 pour voir une description minimale en VHDL.
A une entité peuvent correspondre plusieurs architectures. De cette manière, il devient possible de modifier la structure interne d'une description sans modifier la vue externe, cette propriété du VHDL devenant très intéressante dès lors que plusieurs description sont imbriquées les unes dans les autres (voir les notions avancées).
Les entités et architectures sont des unités de conception, dites respectivement primaires et secondaire. Nous verrons qu'il existe d'autres unités de conception (paquetage, corps de paquetage et configuration).
Un couple entité-architecture donne une description complète d'un élément. Ce couple est appelé un modèle.

2.Syntaxe

On notera au passage la présence de points virgules à la fin de chaque ligne de déclaration et après chaque instruction.
Lorsqu'une liste d'entrées, de sorties ou de paramètres est énumérée, la séparation se fait par des virgules.

Comme dans la plupart des langages informatique, il est fortement conseillé de se servir des tabulations pour améliorer la lisibilité. Les outils de synthèse proposent en général l'utilisation de couleurs différentes.
Lorsque plusieurs opérations sont décrites sur une même ligne, les parenthèses permettent également une lecture plus facile.

Passons maintenant en revue les différents éléments utilisés.

2.1.les instructions port et generic

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

2.2.Les directions

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

in : entrée
out : sortie. Une sortie ne peut être relue à l'intérieur de l'architecture associée à l'entité.
inout : entrée-sortie. La connexion peut alors être relue et réutilisée à l'intérieur de l'architecture associée, au détriment de la lisibilité du programme (il devient impossible de dire si une information externe peut entrer ou s'il s'agit d'un simple réutilisation interne).
buffer : sortie pouvant être réutilisée à l'intérieur de l'architecture (contrairement à « **out** »). Cependant, si à l'intérieure de l'architecture, cette sortie est connectée à la sortie de composants internes (voir plus loin), la sortie des composants internes doit aussi être déclarée comme buffer. Cette limitation nuit fortement à l'utilisation de bibliothèques de composants génériques.

2.3.Les objets

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

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

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

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

2.4.Notation des littéraux

Ces sont les valeurs particulières qu'on donnera aux grandeurs utilisées :
- bits et caractères : entre apostrophes ; exemple **'**0**'**.
- chaîne de bits et chaîne de caractères : entre guillemets ; exemple **"**01110**"**.
- nombre décimaux : exemple pour la valeur mille ; 1000, 1_000, 1E3, 1.00E3.
- nombre hexadécimaux : entre guillemets précédés de x ; exemple **x"**1AFF**"** .Certains outils de synthèse ne reconnaissent pas cette notation, et on trouvera plus facilement la notation 16#1AFF# pour le nombre 1AFF (de même 2#..# pour le binaire et .8#…# pour l'octal).

2.5.Les types

2.5.1.types principaux en synthèse

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

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

« integer range MINI to MAXI ;»

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

bit : deux valeurs possibles : 0 ou 1 ; l'utilisation de la bibliothèque **ieee1164** permet de d'ajouter une extension avec le type **std_logic**, lequel peut prendre en plus l'état haute impédance : **Z.** (Attention : certains outils de synthèse exigent une majuscule pour le Z)
Les autres états du type **std_logic** définis dans la bibliothèque (**U** -non initialisé-, **X** -inconnu-, **W** -inconnu forçage faible- **H** -NL1 forçage faible-, **L** -NL0 forçage faible- et **-** -quelconque-) sont utilisés en modélisation.

bit_vector : un vecteur de bits est un bus déclaré par exemple pour une largeur de N bits par **bit_vector**(0 **to** N) ou bien **bit_vector**(N **downto** 0). Le bit de poids fort est celui déclaré à gauche, c'est à dire le bit 0 dans la première écriture et le bit N dans la seconde, ce qui explique que c°ette dernière est souvent préférée.
L'utilisation de la bibliothèque **ieee** , grâce à son paquetage (voir plus loin) **ieee.std_logic_1164**; permet d'utiliser le type **std_logic_vector**. avec la possibilité d'un état haute impédance.
On trouvera également dans le paquetage **ieee.std_logic_arith** de la même bibliothèque la possibilité d'utiliser les type **signed** (représentation de la grandeur en complément à 2) et **unsigned** (représentation en binaire naturel) qui permettrons de réalisée des opérations arithmétiques signées ou non.

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

2.5.2.types énumération et sous-types

Il est également possible de déclarer un type nouveau à l'intérieur d'une architecture. On parle alors de type énuméré. On peut par exemple décrire ainsi les différents les états du fonctionnement d'une machine (d'état) :

```
type ETATS_DE_LA_MACHINE is (E1, E2, ...EN) ;
```

Il devient alors possible d'utiliser un signal (par exemple) de ce type en le déclarant par :

```
signal ETAT_ACTUEL : ETAT_DE_LA_MACHINE ;
```

Dans la même idée le VHDL permet de définir des sous type à partir d'un type déjà existant. On pourra par exemple créer un type octet à partir du type **std_logic_vector** :

```
subtype OCTET is std_logic_vector (7 downto 0) ;
```

2.5.3.type tableau

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

Il est également possible de définir des tableaux comprenant des grandeurs d'un type autre que **bit** et même des tableaux de plusieurs colonnes.
Dans la pratique, les outils de synthèse ne reconnaissent souvent que les tableaux de 2 dimensions, que l'on utilise pour la synthèse de mémoire.
Voici un exemple de déclaration d'un type tableau pouvant représenter le contenu d'une mémoire M mots de N bits :

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

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

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

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

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

Cette notation est généralement préférée car elle permet un accès direct aux mots, alors que la précédente donnait un accès aux bits.
Imaginons dans l'exemple précédent que la mémoire contiennent 2 mots de 4 bits. Pour affecter la valeur hexadécimale A au premier mot, on peut écrire :

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

ou bien encore :

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

2.5.4.Type enregistrement

Le VHDL permet la description de tableau comprenant des grandeurs de types différents. Dans une synthèse cela se résume souvent à la juxtaposition de vecteurs de bits de tailles différentes. Prenons l'exemple de la description d'un plan mémoire définit par 2^P pages de 2^M mots de N bits et créons un type MEM incluant tous ces éléments :

```
type MEM is record
        page : std_bit_vector (P downto 0) ;
        adr : std_bit_vector (M downto 0) ;
        mot : std_bit_vector (N downto 0) ;
end record ;
```

En définissant un signal S de type MEM, on peut ensuite accéder à chaque élément du signal, la page par exemple et lui affecter la valeur hexadécimale 0 :

```
S.page<= x"0" ;
```

2.5.5.initialisation

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

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

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

2.5.6.Les agrégats

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

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

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

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

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

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

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

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

2.5.7.Les alias

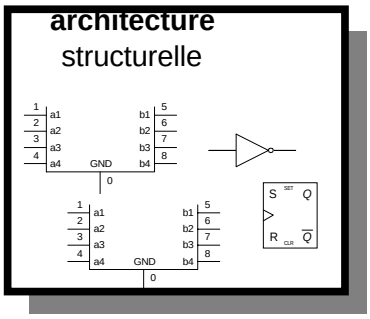
Ils permettent de désigner sous un autre nom une partie d'une grandeur, par exemple appeler par signe le bit de poids fort d'un mot de 16 bits (codé en complément à 2).

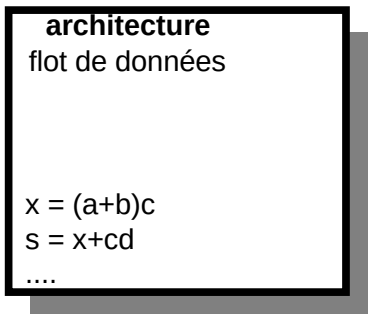
```
signal mot :std_logic_vector(15 downto 0) ;
alias signe :std_logic is mot(15) ;
```

2.6.La description

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

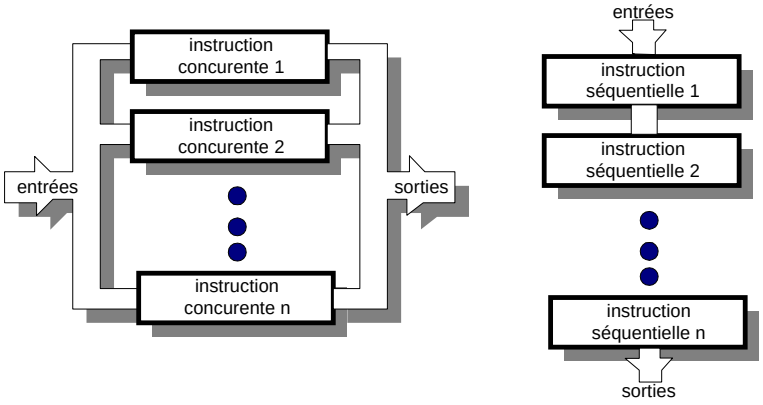
- comportementale (behavioral), sans référence à des structures ou des équations.
- structurelle, c'est à dire réalisée à partir de composants prédéfinis.
- de type flot de données, c'est à dire réalisée à partir d'équations booléennes.





Elle peut se faire en utilisant des instructions :

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



2.6.1.Les instructions concurrentes

Affectation simple
il s'agit d'une interconnexion entre deux équipotentielles.

```
NOM_D'UNE_GRANDEUR<=VALEUR_OU_NOM_D'UNE_GRANDEUR ;
```

L'affectation conditionnelle
L'interconnexion est cette fois soumise à conditions.

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

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

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

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

Remarque : noter la présence d'une virgule (et non un point virgule) à la fin des lignes intermédiaires.
La boucle for generate
Une boucle est répétée plusieurs fois en fonction d'un indice.

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

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

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

Le processus

Avec les processus, une liste de sensibilité définit les signaux autorisant les actions.
Deux syntaxes sont couramment utilisées. Dans la première les signaux à surveiller sont énumérés dans une liste de sensibilité juste après le mot **process** :

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

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

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

Avec la seconde syntaxe on utilise l'instruction **wait** pour énoncer la liste de sensibilité :

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

le déclenchement sur un front montant d'un élément de la liste de sensibilité (CLK par exemple) se fait en replaçant l'instruction wait par la forme suivante :

```
wait until (CLK='event and CLK='1') ;
```

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

La variation d'un des signaux de la liste de sensibilité provoque le déroulement des instructions du process. Aussi, pour synthétiser une structure déclenchant sur front montant par exemple, il suffirait d'écrire **if** CLK**='**1**'** ou bien **wait until** CLK**='**1**'**. Pour une meilleure lisibilité du programme on préfère cependant les écritures présentées précédemment, qui restent également très pratique si la liste de sensibilité présente plusieurs grandeurs.

Les changements de valeur des grandeurs modifiées par le déroulement d'un process ne prennent effet qu'à la fin de celui-ci. Aussi, si un signal ou une variable, modifié par un process est réutilisé, en lecture cette fois, dans le même process, c'est la valeur avant modification qui sera prise en compte.

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

L'instruction **process** peut être précédée d'une étiquette afin de faciliter la lecture du programme :

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

```
        end process ;
```

2.6.2.Les instructions séquentielles

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

L'affectation séquentielle

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

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

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

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

*Le test « **case..when..end case** »*

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

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

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

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

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

La boucle « while..loop..end loop »

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

L'attente « wait until »

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

2.6.3.Principaux opérateurs

Les opérateurs sont définis initialement pour un certain type, par exemple l'addition pour le type entier. Cependant, grâce aux surcharges définies dans certains paquetage (voir plus loin), il sera possible d'étendre le champ d'application d'un opérateur, par exemple l'addition avec le type **std_logic_vector.**

Opérateurs logiques : **and, nand, or, nor, xor, not, xnor** (**xnor** n'existe qu'en VHDL93), ils renvoient un type boléen

Opérateurs de décalage (en VHDL93 uniquement) : ils sont définis en VHDL93 uniquement et ne sont pas supportés par tous les outils de développement (même récents)
- **sll** (décalage à gauche avec remplissage par 0 à droite),
- **slr** (idem à gauche), **sla** (décalage à droite avec remplissage à droite par le LSB),
- **sra** (idem à gauche et MSB),
- **rol** (rotation à gauche, le MSB devenant LSB),
- **ror** (idem dans l'autre sens).

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

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

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

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

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

2.6.4.Les attributs

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

```
NOM_DE_LA_GRANDEUR ' NOM_DE_L'ATTRIBUT ;
```

Pour illustrer les principaux attributs utilisés pour les tableaux, prenons comme exemple le type tableau :

```
type T is array (4 to 8, 7 downto 2) of std_logic;
```

Nous aurons dans ce cas :

- left(n**)** qui restitue un entier égal à la valeur de la *borne* gauche de l'index n dans l'intervalle de variation de la *déclaration* du tableau, ce qui donne pour notre exemple 4 ou 7 suivant que n a pour valeur 1 ou 2. Le paramètre n est optionnel et prend la valeur 1 par défaut.

- right(n**)**, même chose avec la borne de droite, soit dans notre exemple 8 ou 2 suivant n.

- low(n**)**, même chose avec la borne la plus petite, soit 4 ou 2 suivant n.

- hign(n**)**, même chose avec la borne la plus grande soit 8 ou 7.

- range(n**)**, qui restitue l'intervalle de variation des indices, soit 4 **to** 8 si n=1 et 7 **downto** 2 si n=2. Cet attribut est très utile pour générer l'index d'une boucle.

- reverse_range(n**)**, même chose en inversant le sens de variation, soit 8 **downto** 4 et 2 **to** 7.

- length(n**)**, qui restitue un entier égal à la longueur de l'intervalle d'index n, soit 5 si n=1 et 6 si n=2.

Il faut bien noter que ces attributs ne donne pas accès aux valeurs du tableau (utiliser alors la syntaxe vue dans le paragraphe sur les tableaux), mais aux bornes de la déclaration du tableau.

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

3.Notions avancées

Ce que nous allons voir maintenant permet de donner une plus grande convivialité au VHDL par l'utilisation de ressources externes à la description. Celles-ci seront disponibles pour plusieurs projets, voire plusieurs utilisateurs. Elles permettront d'autre part une modification facile d'un projet (largeur de bus par exemple).

3.1.Les composants

Comme nous venons de le voir, une description en VHDL est un couple entité/architecture. Cette association est appelée un modèle. Celui-ci peut éventuellement être utilisé par d'autres couples entité/architecture, tout comme un schéma structurel utilise une ou plusieurs fois un composant. Par analogie on parle alors de composants.
Après avoir été décrit par une entité et une architecture, le composant doit être déclaré dans la zone déclarative de l'architecture dans laquelle il est utilisé. La syntaxe est la suivante

```
component NOM_DE_L'ENTITE_DECRIVANT_LE_COMPOSANT
        generic(PARAMETRES :« type » :=VALEURS_FACULTATIVES) ;
        port (ENTREES :in bit ; SORTIES :out bit) ;

end component ;
```

Le composant doit avoir un nom au sein de l'architecture où il va être utilisé (ici U1) On précise ensuite les liaisons existantes entre les ports du composant (ports formels) et les connexions de l'architecture (ports effectifs).
On parle alors d'instance (ou d'instanciation) du composant. Elle doit être faite entre les mots **begin** et **end** de l'architecture :

```
U1 : NOM_DE_L'ENTITE_DECRIVANT_LE_COMPOSANT
                generic map (VALEUR_DES_PARMETRES)
                port map ( ENTREES SORTIES A UTILISER) ;
```

Les entrées sorties à utiliser avec le composant, ainsi que les paramètres doivent évidemment être décrits dans le même ordre que dans la déclaration. On peut aussi utiliser une suite d'affectation (séparée par des virgules) entre les ports formels et les ports effectifs du composant si on ne souhaite pas respecter cet ordre ; l'écriture est alors plus longue, mais aussi plus expressive.
Dans la déclaration comme dans l'instance, l'instruction **generic** (si elle existe) doit précéder la description des ports. On notera dans l'instance l'absence de point virgule après cette instruction.

3.2.Les sous-programmes

Comme dans tous les langages informatiques, lorsqu'on utilise plusieurs fois les mêmes fonctionnalités, on préfère écrire une fois pour toutes un sous programme et l'appeler autant de fois que nécessaire.
On distingue les fonctions et les procédures

3.2.1.Les fonctions

Les fonctions sont des sous-programmes à qui on fournit un ou plusieurs paramètres en entrée et qui retournent un (et un seul) argument en sortie. La syntaxe de la description est la suivante :

```
function NOM_DE_LA_FONCTION (PARAMETRES : « type ») return « type du paramètre retourné » is
        déclaration des variables si nécessaire ;
begin
        INSTRUCTIONS_SEQUENTIELLES ;
        return NOM_OU_VALEUR_DU_PARAMETRE_DE_RETOUR ;
end ;
```

L'utilisation se fait ensuite par la syntaxe suivante :

```
NOM <= NOM_DE_LA_FONCTION (PARAMETRES_D'UTILISATION) ;
```

A la grandeur "NOM" (qui doit être de même type que le paramètre de retour) sera alors affecté la valeur de retour de la fonction. Les paramètres d'utilisation (propres au programme principal) doivent évidemment être énumérés dans le bon ordre pour correspondre à ceux de la fonction (ou alors utiliser l'affectation).

Les fonctions permettent entres autres la surcharge d'opérateur, c'est à dire l'utilisation d'un opérateur avec des types pour lequel il n'a pas été prévu initialement :

- l'addition n'est par exemple initialement définie que pour des entiers (ou équivalents) ; aussi si le compilateur rencontre le signe « + » entre deux std_logicl_vector il générera une erreur ; pour éviter cela, il est possible de définir une fonction permettant la surcharge de l'opérateur « + » pour des types std_logic_vector.

De même une fonction peut être surchargée pour être utilisée avec des paramètres d'entrée différents.
Les surcharges classiques sont souvent déclarées dans des paquetages prédéfinis (voir plus loin).
Les fonctions prédéfinies dans les paquetages permettent également les très utiles (voir annexe) conversions de type ; citons :

- **conv_integer (std_logic_vector)** qui restitue un entier égal à la valeur binaire du std_logic_vector entre parenthèse. Cette fonction est issue du paquetage ieee.std_logic_arith. Elle considérera le **std_logic_vector** à convertir comme codé en binaire naturel si le paquetage **ieee.std_logic_unsigned** à été déclaré (l'entier retourné sera alors positif).
  Elle considérera le **std_logic_vector** à convertir comme codé en complément à 2 si le paquetage **ieee.std_logic_signed** à été déclaré (l'entier retourné sera alors positif ou négatif suivant le poids fort du **std_logic_vector**).

- **conv_std_logic_vector (** entier **,** nombre de bits **)** qui convertit l'entier entre parenthèse en un std_logic_vector dont le nombre de bits est précisé.

Les fonctions disponibles dans un outil de synthèse dépendent des paquetages fournis avec cet outils. Les exemples précédents sont donnés pour le logiciel Max+plus II de chez Altera.

3.2.2.Les procédures

Equivalentes aux fonctions, les procédures en diffèrent par le fait qu'elles peuvent retourner plusieurs paramètres. Il faut donc préciser les directions dans la description :

```
procedure NOM_DE_LA_PROCEDURE (PARAMETRES : « direction » « type ») is
        déclaration des variables si nécessaire
begin
        INSTRUCTIONS_SEQUENTIELLES ;
end ;
```

On peut remarquer dans la définition des paramètres entre parenthèse , la présence d'une information direction, contrairement à ce qui se passe pour une fonction.
L'appel de la procédure se fait par :

```
NOM_DE_LA_PROCEDURE (PARAMETRE_D'UTILISATION) ;
```

L'énumération des paramètres d'utilisation doit se faire dans le même ordre que dans la description de la procédure (ou alors utiliser l'affectation). La procédure va donc modifier dans le programme principal les paramètres de sortie.

3.3.Les bibliothèques

Ce sont des répertoires -au sens informatique du terme- où se trouvent des fichiers contenant des unités de conception, c'est à dire :

- des entités et architectures associée,

- des spécifications de paquetage, ainsi que la description du corps associé,
- des configurations.

Ces ressources n'auront donc pas besoin d'être décrite dans le programme principal. Il suffira d'appeler la bibliothèque avec la syntaxe :

```
library NOM_DE_LA_BIBLIOTHEQUE ;
```

Il existe des bibliothèques prédéfinies (**ieee** ou **std** par exemple), mais l'utilisateur peut créer sa propre bibliothèque. Dans ce dernier cas, il est nécessaire de préciser à l'outil de synthèse où se trouvent les bibliothèques utilisées.

Les entités et architectures permettront la description de composants. Voyons maintenant ce que sont les paquetages et configurations.

3.3.1.Les paquetages

Un paquetage permettra d'utiliser sans avoir a les redécrire essentiellement des objets (signaux, constantes), des types et sous-types, des sous-programmes et des composants.
Le paquetage est composé d'une unité de conception primaire qui en donne une vue externe (comme une entité), dont la syntaxe est :

```
package NOM_DU_PAQUETAGE is
        définition des types, sous-types, constantes ;
        déclaration des fonctions, procédures, composants, signaux ;
end NOM_DU_PAQUETAGE ;
```

Lorsque la déclaration comprend des sous-programmes, une unité de conception secondaire (comme une architecture) doit être associée, avec la syntaxe suivante :

```
package body NOM_DU_PAQUETAGE is
        description des fonctions et procédures ;
end ;
```

L'ensemble est placé, dans un fichier VHDL au sein d'une bibliothèque.

On peut noter qu'un composant est seulement déclaré par un paquetage, mais pas décrit par le corps du paquetage, une unité de conception ne pouvant contenir une autre unité de conception. La description se trouve dans un fichier VHDL d'une bibliothèque.

L'appel du paquetage se fait après l'appel de la bibliothèque qui le contient. On indique alors le nom du fichier contenant le paquetage par l'instruction qui suit :

```
use NOM_DE_LA_BIBLIOTHEQQUE . NOM_DU_PAQUETAGE.all ;
```

Dans la pratique, il n'y a généralement qu'un paquetage par fichier, et les deux portant le même nom. Le mot réservé point **all** permet théoriquement d'appeler tous les paquetages du fichier (il serait possible d'en appeler un seul, mais cela présente peu d'intérêt).
Il existe des paquetages prédéfinis fournis avec les outils de synthèse :

- le paquetage **standard** qui contient la déclaration de tous les types prédéfinis de base que nous avons vu. Son utilisation est implicite, il n'est pas nécessaire de l'appeler.
- le paquetage **std_logic_1164** contenu dans la bibliothèque **ieee** qui contient entre autre la définition des types std_logic et std_logic_vector.
- le paquetage **std_logic_arith** (très semblable au paquetage **numeric_std**) de la bibliothèque **ieee** qui permet des conversions de type (entier vers **std_logic** et inversement).
- les paquetages **std_logic_signed** et **std_logic_unsigned** de **ieee** permettant des opérations arithmétiques signées ou non sur des types std_logic.

On rappelle qu'un **std_logic_vector** sera considéré comme codé en binaire naturel si le paquetage **ieee.std_logic_unsigned** à été déclaré et comme codé en complément à 2 si le paquetage **ieee.std_logic_signed** a été déclaré .

3.3.2.Les configurations

Une configuration est une unité primaire de conception qui permet de décrire la liaison entre un composant et le modèle (entité-architecture) auquel il est attaché.
Les outils de synthèse ne supportant généralement pas les configurations, nous n'en parlerons pas plus.

4.Remarques

Bien qu'il soit aisé de décrire n'importe quelle structure en VHDL, il ne faut pas perdre de vue l'objectif final, à savoir la programmation d'un composant, qui doit se faire en essayant d'optimiser celui-ci en terme d'intégration et de vitesse.
Ainsi, par exemple un bit mémoire dans un CPLD classique utilisera une cellule élémentaire (c'est à dire une dizaine de portes) alors qu'il ne faut qu'un transistor dans une mémoire flash (certains CPLD et tous les FPGA prévoient une zone optimisée pour la mémoire).
De même, pour la réalisation d'un compteur de N bits, l'opération d'addition d'un entier utilise à l'intérieur du composant deux bus de N bus ; une simple incrémentation sera moins gourmande.

D'une manière générale, il sera nécessaire d'avoir une vue d'ensemble du projet que l'on souhaite synthétiser, avant de le décomposer en sous-ensembles relativement simples, de manière à ce qu'ils s'intègrent facilement dans le circuit cible.

Bibliographie

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

Annexe 1 : instructions les plus utilisées

nom	description	remarques
Directions	**in** **out** **inout** **buffer**	- - ne peut être relu en interne - -
Types	**integer** **std_logic** **std_logic_vector**	- - permet l'état haute impédances 'Z' - permet l'état haute impédances 'Z'
Objets	**signal** **variable** **constant**	- - à l'intérieur d'un process uniquement - -
Notation des littéraux	bits et caractères : '0' chaînes : ''001101'' décimaux : 100, 1E2 hexadécimaux : X''1A''	
Instructions concurrentes	**<=** **when else** **whith select when when others** **for to generate** **process**	- affectation - - - -
Instructions séquentielles	**<=** **if then elsif else end if** **case is when end case** **for to loop end loop** **while loop end loop** **wait until**	- **:=** pour les variables - - - - -
Opérateurs logiques	**and, nand, or, nor, not, xor** **xnor** (vhdl93 uniquement)	les opérateurs de décalage ne sont généralement pas reconnus par les outils de synthèse
Opérateurs arithmétiques	**+,** **-,** ***,** **/**	nécessite l'utilisation de fonction de surcharge ou de fonction de conversion de type pour être utilisé avec des std_logic_vector La multiplication et la division ne sont effectuées que par puissance de 2 par les outils de synthèse
Opérateurs relationnels	**=**, **/=**, **<**, **<=**, **>**, **>=**	attention, le symbole « inférieur ou égal » est le même que celui de l'affectation
Opérateurs divers	**&**	concaténation
Attributs	**'event** **'range** **'reverse_range**	
Fonctions de conversion	**conv_integer (std_logic_vector)** **conv_std_logic_vector**(entier,nb_bits)	
Paquetages	**std_logic_1164** **std_logic_arith** **std_logic_signed** **std_logic_unsigned**	- pour le type std_logic et std_logic_vector - pour les fonctions de conversion - pour des opérations signées - pour des opérations non signées

Annexe 2 : exemples de programmes

Voici quelques exemples de descriptions en VHDL Toutes ces descriptions ont été synthétisées avec l'outil MAX+PLUS II version 9.5 de chez Altera. La compilation et la simulation se sont déroulées avec succès.
Le premier programme illustre la notion d'entité et d'architecture. Aucune ressource externe n'est déclarée, le paquetage **standard** de la bibliothèque **std** étant d'utilisation implicite. Les entrées et sorties étant de type bit (pas d'état haute impédance), il n'est pas nécessaire d'utiliser le paquetage **std_logic_1164** de la bibliothèque **ieee**.

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

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

-----------------------------------------------------------------------------------------------------------------------
```

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

```
Architecture ARCH_ET of ET is
Begin
S <=    '1'  when E1='1' and E2='1' else
                '0';
end ARCH_ET;
```

```
architecture ARCH_ET of ET is
begin
        with E1='1' and E2='1' select
                        S<=     '1' when true,
                                '0' when others;
end ARCH_ET;
```

```
architecture ARCH_ET of ET is
begin
        process (E1, E2)
        begin
                if E1='1' and E2='1' then S<='1' ;
                else    S<='0' ;
                end if ;
        end process ;
end ARCH_ET;
```

```
architecture ARCH_ET of ET is
begin
        process (E1, E2)
        begin
                case E1='1' and E2='1' is
                when true       =>      S<='1' ;
                when others =>          S<='0' ;
                end case ;
        end process ;
end ARCH_ET;
```

L'exemple suivant qui décrit un compteur, met en évidence l'utilisation
- d'une variable qui améliore la lisibilité de l'entité
- du process qui permet de décrire des systèmes synchrones
- des bibliothèques et paquetage (même si l'état haute impédance n'est pas utilisé ici)
- de l'instruction generic

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

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

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


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


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

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

Le programme suivant décrit le même compteur, mais avec cette fois une entrée de validation OE des sorties (qui seront en haute impédance si elles ne sont pas validées, le comptage continuant toujours), une remise à zéro synchrone RAZS et une remise à zéro asynchrone RAZAS. Noter que c'est cette dernière qui apparaît dans la liste de sensibilité du process.
L'introduction du modulo M met en évidence l'utilisation d'une fonction (déclaré dans le paquetage std_logic_arith) qui permet la conversion de type d'un entier M vers un std_logic_vector. Cette fonction est utile pour affecter une valeur entière à X, mais n'est pas nécessaire pour les comparaisons.
Citons une petite particularité de cette description : avec le compilateur altera, l'état haute impédance 'Z' doit obligatoirement être écrit en majuscule.

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

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

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

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

entity COMPTEUR2 is

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

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

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

                process (H, RAZ_AS)

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


-------------------------------------------------------------------------------------------------------------------------
```

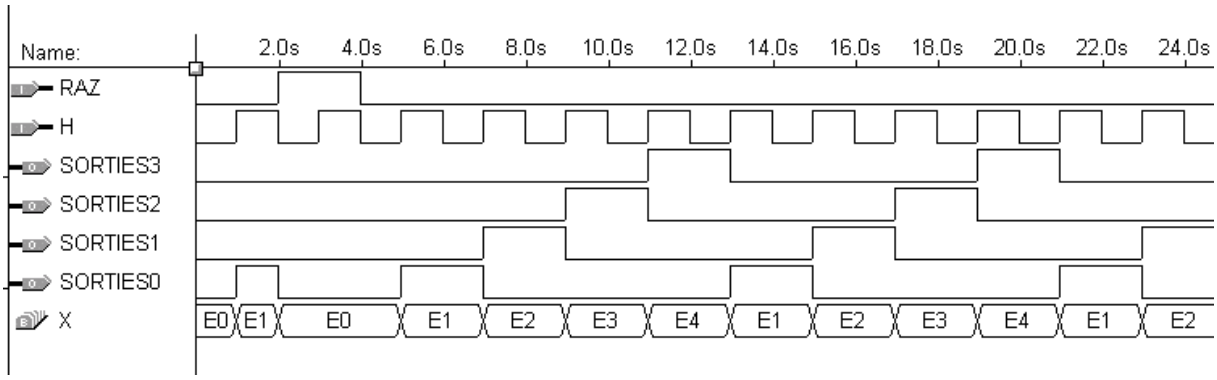
L'exemple suivant décrit une mémoire RAM dont la largeur de bus de données peut être paramètrée. Le paramétrage de la largeur du bus d'adresse est plus complexe, le compilateur ne reconnaissant pas la fonction puissance.
Cette RAM étant réalisée sur un CPLD quelconque, chaque bit sera mémorisé par une bascule, ce qui est peu optimal du point de vue de la densité d'intégration. On a en contre partie la possibilité de lire la RAM pendant un cycle d'écriture, d'où les deux bus de données et les deux bus d'adresse.
La lecture peut se faire à n'importe quel moment en plaçant DATA_READ au niveau logique 1. L'écriture se fait sur le front montant de H.

Cet exemple permet de mettre en évidence l'utilisation :

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

```
--------------------------------------------------------------------------------------------------------------------------------

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


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


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

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

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


--------------------------------------------------------------------------------------------------------------------------------
```

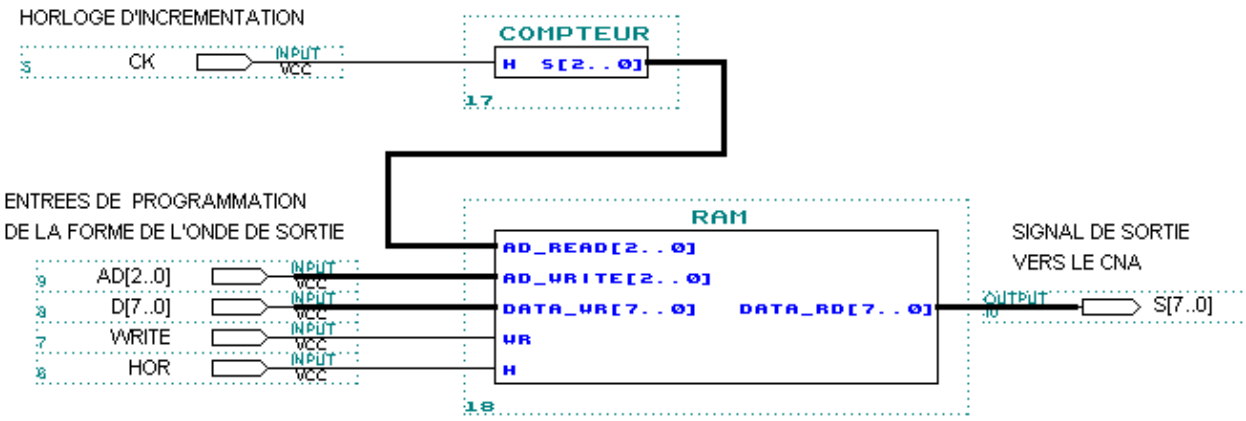
Voyons maintenant un exemple de description d'une machine d'état. Nous avons pris pour cela une commande de moteur pas à pas 4 phases unipolaire : à chaque front montant d'horloge, une des commandes de phase passe au niveau logique 1, les autres à 0. Une remise à zéro asynchrone remet toutes les commandes de phase au niveau logique zéro.



Cet exemple permet de plus une illustration des instructions "case" et "with select" ainsi que la déclaration d'un "type".

```
------------------------------------------------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity CDE_MOT is
        port(   H, RAZ                  : in      std_logic;
                SORTIES                 : out std_logic_vector (3 downto 0));
end CDE_MOT;

architecture ARC_CDE OF CDE_MOT is
        type TYPE_ETAT is (E0, E1, E2, E3, E4);
        signal X:TYPE_ETAT;
-- d'après la déclaration du TYPE_ETAT, X sera initialisé à la valeur E0
begin
        process (H,RAZ)
        begin
                if RAZ = '1' then X<=E0;
                elsif H'event and H = '1' then
                        case X is
                                when E0         =>      X <= E1;
                                when E1         =>      X <= E2;
                                when E2         =>      X <= E3;
                                when E3         =>      X <= E4;
                                when others     =>      X <= E1;
                        end case;
                end if;
        end process;

        with  X select
                SORTIES                 <=      "0000"  when    E0,
                                                "0001"  when    E1,
                                                "0010"  when    E2,
                                                "0100"  when    E3,
                                                "1000"  when    E4;

end ARC_CDE;

------------------------------------------------------------------------------------------------------------------------
```

Pour illustrer l'utilisation de bibliothèques et de composants, nous allons maintenant réaliser un synthétiseur numérique simple (voir simpliste, mais le but n'est pas ici de faire de la synthèse numérique directe) à partir du premier compteur et de la RAM, décrits précédemment. La sortie du compteur incrémentera le bus d'adresse (en lecture) de la RAM ; le bus de donnée (en lecture) de la RAM attaquera un CNA extérieur au projet.
Les bus d'adresses et de données en écriture de la RAM seront utilisés pour programmer la forme d'onde désirée en sortie.



Une première solution possible consiste à placer dans le même fichier les trois descriptions (compteur, RAM et synthétiseur) des modèles entité-architecture. D'un point de vue pratique, l'outil de synthèse utilisé n'accepte qu'un seul modèle contenant une instruction "generic", rendant cette option impossible ici.

Une autre solution consiste à placer les fichiers correspondant aux descriptions dans le même répertoire, dit répertoire de travail. L'outil de synthèse va alors de lui-même chercher les descriptions correspondantes.

Enfin troisième solution, les fichiers correspondant à la description du compteur et de la RAM sont placés dans un répertoire c:\BIBLI_PERSO. La description du synthétiseur proprement dit se trouve dans un répertoire quelconque et déclare l'utilisation de ressources externe par l'instruction **library**. L'endroit où se situe la bibliothèque appelée doit de plus être déclaré à l'outil de synthèse (pour l'outil Altera la déclaration se fait dans "VHDL Netlist Reader Setting" du menu "Interfaces" du compilateur et dans "Users Librairies" du menu "Option" de l'éditeur).

```
library bibli_perso;
library ieee;
use ieee.std_logic_1164.all;


entity SYNTHE_NUM is
                port (HOR, WRITE, CK: in       std_logic;
                S                : out std_logic_vector (7 downto 0);-- sorties
                D                : in std_logic_vector (7 downto 0); -- données d'entrée de la RAM
                AD               : in std_logic_vector (2 downto 0)); --adresses de la RAM
end SYNTHE_NUM;



architecture ARCH_SYNTHE of SYNTHE_NUM is
```

```
        component RAM
                generic (N      : integer);
                port (DATA_RD                   : out std_logic_vector(N-1 downto 0);
                AD_READ, AD_WRITE               : in std_logic_vector(2 downto  0);
                DATA_WR                         : in std_logic_vector(N-1 downto 0);
                WR, H                           : in std_logic);
        end component;

        component COMPTEUR
                generic         (N: integer);
                port(   H               : in    std_logic;
                        S               : out std_logic_vector (N-1 downto 0) );
        end component;

        signal X        : std_logic_vector (2 downto 0);

begin

        U1: RAM         generic map (N=>8)
                        port map (DATA_RD=>S, AD_READ=>X, AD_WRITE=>AD,
                                DATA_WR=>D, WR=>WRITE, H=>HOR);

        U2: COMPTEUR    generic map (N=>3)
                        port map (CK, X);

end ARCH_SYNTHE ;
```

Il est également possible de placer la déclaration des composants (qui sera faite une fois pour toute, et pour tous les composants de la bibliothèque) dans un paquetage qui sera sauvegardé et compilé dans le répertoire BIBLI_PERSO.

```
--------------------------------------------------------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

package PAK_PERSO is

        component RAM
                generic (N      : integer);
                port (DATA_RD                   : out std_logic_vector(N-1 downto 0);
                AD_READ, AD_WRITE               : in std_logic_vector(2 downto  0);
                DATA_WR                         : in std_logic_vector(N-1 downto 0);
                WR, H                           : in std_logic);
        end component;

        component COMPTEUR
                generic         (N: integer);
                port(   H               : in    std_logic;
                        S               : out std_logic_vector (N-1 downto 0) );
        end component;

end PAK_PERSO;
--------------------------------------------------------------------------------------------------------------------------------
```

Le paquetage ne comprend qu'une unité de conception primaire (pas de "corps de paquetage"), celle-ci ne déclarant ni fonction, ni procédure. Il faut noter que les composants ne sont pas décrit (ils sont juste déclarés), la bibliothèque devant contenir les fichiers de description.
Le programme principal se résume alors au fichier suivant, où la déclaration des composants a été remplacée par la déclaration du paquetage. On peut alors utiliser tous les composants de la bibliothèque (même si ici il n'y en a que deux) :

```
--------------------------------------------------------------------------------------------------------------------------------
library bibli_perso;
library ieee;
use ieee.std_logic_1164.all;
use bibli_perso.PAK_PERSO.all;

entity SYNTHE_NUM is
        port (HOR, WRITE, CK: in       std_logic;
                S                : out std_logic_vector (7 downto 0);-- sorties
                D                : in std_logic_vector (7 downto 0); -- données d'entrée de la RAM
                AD               : in std_logic_vector (2 downto 0)); --adresses de la RAM
end SYNTHE_NUM;

architecture ARCH_SYNTHE of SYNTHE_NUM is

        signal X        : std_logic_vector (2 downto 0);

begin

        U1: RAM         generic map (N=>8)
                        port map (DATA_RD=>S, AD_READ=>X, AD_WRITE=>AD,
                                DATA_WR=>D, WR=>WRITE, H=>HOR);

        U2: COMPTEUR            generic map (3)
                                port map (CK, X);
end ARCH_SYNTHE ;
--------------------------------------------------------------------------------------------------------------------------------
```

Remarque : une description VHDL utilisant des composants n'est pas toujours très facile à lire et on sera tenté de faire un schéma pour comprendre cette description. Aussi, il est parfois avantageux d'utiliser l'outils de description graphique que proposent la plupart des logiciels de synthèse, et de relier graphiquement entre elles les différentes description VHDL correspondant à chaque composant. Le schéma utilisé pour la présentation du synthétiseur est extrait de l'éditeur graphique du logiciel Max+plus II.