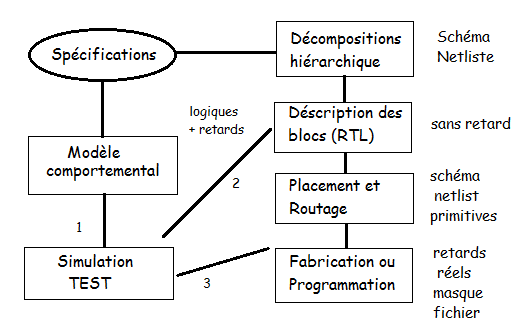
**Chapitre IV Le langage VHDL**

***1. Introduction***

Le **VHDL** (pour **Very High Speed Integrated Circuit Hardware Description Language**, *Langage de description du matériel pour circuit intégré à très haute vitesse* en anglais) est un langage de description du matériel destiné à décrire le comportement et/ou l'architecture d’un système électronique numérique. L'intérêt d'une telle description réside dans son caractère exécutable : une spécification décrite en VHDL peut être vérifiée par simulation, avant que la conception détaillée ne soit terminée. En outre, les outils CAO (Conception Assistée par Ordinateur) de synthèse logique qui permettent de passer directement d'une description fonctionnelle en VHDL à un schéma en porte logique ont révolutionné les méthodes de conception des circuits numériques, ASIC ou FPGA.

Le concepteur va travailler avec des IP (Intellectual Property) s’il est intégrateur de système, mais il peut être lui-même développeur d’IP et la méthode qui devra le guider est appelée Design Reuse. Ces expressions désignent des composants génériques incluant des méthodes d’interfaçages rigoureuses et suffisamment normalisées pour pouvoir être rapidement inclus dans un design quelconque.

***2. Flot de conception (Design Flow)***



Le flot de conception illustre schématiquement les différentes étapes suivies afin de réaliser un circuit intégré, standards ou programmable. Ci-dessus est montré un flot de conception destiné pour les circuits programmables, notamment, les FPGAs.

On observe sur cette figure le rôle important du VHDL car il intervient au moins sur 3 niveaux, celui de la description comportementale traduisant les spécifications, celui du code RTL (Register Transfert Level) et enfin au niveau technologique post-routage censé représenter le circuit « vrai ». Ces trois types de description seront validés par une même famille de fichiers de test eux-mêmes écrits en VHDL.

Le langage est utilisé pleinement dans ses deux déclinaisons : généraliste quand il s’agit de décrire des vecteurs de test ou des comportements abstraits, VHDL synthétisable en vue de générer automatiquement un circuit.

## *3. Qu’est-ce que le VHDL?*

Le VHDL est un langage **portable** qui va trouver place dans le cycle de conception du niveau spécification au niveau porte mais aussi lors de la génération des vecteurs de test. La description VHDL est inséparable de la simulation de type événementielle.

On peut apprécier le coté **généraliste** du langage et développer un niveau d’abstraction souhaité. Ce sera le cas pour des routines de **test** ou des **descriptions** **comportementales**. On fera alors de la **modélisation**.

Il est aussi possible de rester le plus près possible du niveau portes et utiliser un VHDL de **synthèse** qui apparaît comme un sous-ensemble du VHDL. Comme langage source d’un synthétiseur, le VHDL permet d’étendre très largement les bibliothèques en conception d’ASIC ou encore de favoriser la conception descendante (Top-Down Design). Les descriptions sont alors fortement **documentées**.

On peut aussi être obligé d’utiliser des descriptions VHDL puisque des outils de description de plus haut niveau sont capables de générer du VHDL privilégiant la forme **netlist**; le langage est alors la couche d’interface indispensable (car portable) dans le flot de conception.

*« Le VHDL de synthèse est un sous-ensemble du VHDL généraliste »*

***Remarque***

Il ne faut pas opposer la description textuelle VHDL aux autres outils de description plus traditionnels tels que la schématique. En effet, les systèmes de CAO savent de plus en plus traduire un schéma en VHDL ou inversement créer un schéma à partir du VHDL. La mixité est aussi une excellente pratique puisqu’en définitive pour qu’une description soit bonne c’est à dire lisible il faut savoir ce que l’on cache et ce que l’on montre et comment.

## *4. Unités de conception*

### *4.1 Définition*

Une unité de conception est une partie de programme qui peut être compilée séparément. Cet aspect modulaire est la base de la structuration de la description. Le support du programme est un fichier texte qui peut contenir une ou plusieurs unités. Par défaut le résultat de compilation sera placé dans une bibliothèque ayant comme nom logique **WORK**. Pour la compilation de plusieurs unités, un certain ordre devra être respecté à cause de leur dépendance éventuelle.



* Les unités de conception **primaires** correspondent à la vue externe des objets.
* La spécification d’entité (**ENTITY**) (très proche du symbole en schématique) définie les **signaux** d’entrées-sorties, leur **type** ainsi que leur **mode** (lecture seule, écriture seule, lecture-écriture) ainsi que les procédures éventuellement associées (par exemple: vérifications de relations temporelles). La généricité permet de paramétrer ces définitions.
* La spécification de paquetage (**PACKAGE**) permet de regrouper des déclarations de types et/ou de sous-programmes et en fait de construire des bibliothèques. Elle offre ainsi la possibilité d’exporter un de ces objets.
* Les unités de conception **secondaires** correspondent aux algorithmes des modèles et des sous-programmes.
* L’architecture (**ARCHITECTURE**) est relative à une entité. Elle contient les fonctionnalités et éventuellement les relations temporelles du modèle (dans une description en vue de la synthèse, il ne doit pas y avoir de retards, ceux-ci sont liés à la technologie).
* Le corps de paquetage (**PACKAGE** **BODY**), pas toujours nécessaire, contient l’écriture proprement dite des fonctions ou des procédures déclarées au niveau paquetage.
* La configuration (**CONFIGURATION**) est une unité de conception **primaire** qui permet de créer un couple **entité-architecture** (plusieurs architectures pouvant être associées à une entité). Dans les cas simple, elle peut être remplacée par une ligne comportant la clause **USE**...

### *Exemple simple*

**ENTITY** comparateur **IS**

**PORT(**

**SIGNAL** a **: IN bit\_vector(7 DOWNTO 0);**

**SIGNAL** b **: IN bit\_vector(7 DOWNTO 0);**

**SIGNAL** egal **: OUT bit);**

**END ;**

* Le mot clef **SIGNAL** peut être omis car pris par défaut
* Le mode **IN** protège le signal en écriture.
* Le mode **OUT** protège le signal en lecture.

**ARCHITECTURE** simple **OF** comparateur **IS**

-- zone de déclaration vide

**BEGIN**

egal **<= .1. WHEN** a **=** b **ELSE .**0**.;**

**END** simple **;**

## *4.2 Instructions concurrentes*

Entre le BEGIN et le END de l’ARCHITECTURE on est dans un contexte d’instructions concurrentes ou plutôt d’objets concurrents. Ce qui les caractérise :

* L’ordre des instructions concurrentes est indifférent.
* Pour le compilateur, chaque objet concurrent est en fait un processus
* Les objets concurrents (partie d’un circuit) sont reliés entre eux par des signaux.

**Exemple : Circuit Set-Reset**



On peut le décrire simplement par 2 instructions concurrentes (2 processus implicites).

ARCHITECTURE memoire\_rs OF circuit\_plus\_important IS

SIGNAL s, r, q, qb : BIT := .1.;

BEGIN

q <= qb NOR r ; -- premier processus

qb <= q NOR s; --deuxième processus

-- autres processus en parallèle

END

## *4.3 Les signaux*

### *4.3.1 Signal avec un seul pilote (Driver)*

Le signal est un objet de première importance car c’est lui qui permet de connecter entre eux les circuits implantés sous forme de processus. Il joue un rôle équivalent à celui du nœud ou NET en schématique.

Un signal est caractérisé par

* Un type déclaré
* Un état en référence à ce type
* Un ou plusieurs pilotes (driver) qui lui sont associé

Le pilote du signal contient une liste de couples état-heure, celle-ci étant comptée relativement à l´heure actuelle du simulateur.

Au moment de son exécution, l´instruction VHDL suivante

**s <= ´0´, ´1´ AFTER 10 ns,´0´ AFTER 25 ns ;**

Placera dans la pilote de s



Le pilote est en fait une mémoire associée au signal.

### *4.3.2 Evènements*

On appelle événement tout changement d’état d´un signal. (Alors qu´une transaction est une opération effectuée sur le signal mais qui peut ne pas se traduire par un événement). Tant qu’un événement potentiel se trouve dans le pilote mais n´est pas affecté au signal, il est dit **non** **mûr**.

## *4.4 Les processus*

Les processus constituent les éléments calculatoires de base du simulateur. Au niveau descriptif, ils peuvent être explicites (**PROCESS**) ou implicites (instructions concurrentes). Du point de vue de l’exécution, on peut affirmer que « tout est processus ».

Un processus vit toujours depuis le chargement du code exécutable dans le simulateur. Il ne peut prendre fin qu’avec la simulation mais peut par contre être endormi pour une durée plus ou moins longue. L’instruction **WAIT**, obligatoire au moins une fois, synchronise le processus. Elle possède trois formes pouvant être combinées :

**WAIT ON *événement*;**

**WAIT FOR *durée*;**

**WAIT UNTIL *condition*;**

**WAIT;**

Le processus s’exécute au moins une fois jusqu’à rencontrer le WAIT, puis la condition de réveil une fois validée, l’exécution continue de façon séquentielle et cyclique (après la fin, le début). Donc à retenir :

***« Tout est processus »***

***« Un processus est synchronisé par WAIT »***

***« Un processus est cyclique »***

## *4.5 Simulation événementielle*

La simulation d’un modèle porte sur les signaux utilisés pour le décrire. Elle est de type événementiel. Le principe est le suivant:

Au départ, l’heure de simulation est fixée à 0. Des stimuli sont associés aux signaux d’entrées. La simulation fonctionne selon le principe d’une roue temporelle.

* Les événements non mûrs (stimuli au départ) sont classés selon les heures futures croissantes.
* L’heure de simulation courante est avancée au premier événement non mûr de la liste et celui-ci est supprimé de la liste. La valeur en attente dans le pilote du signal affecte celui-ci.
* Cet événement réveille un processus qui s’exécute immédiatement et affecte des pilotes d’autres signaux qui vont contribuer à établir une nouvelle liste.

### *4.5.1 Premier exemple: Génération d´horloge*

SIGNAL h : bit;

BEGIN

horloge : PROCESS

BEGIN

h <= .0., .1. AFTER 75 ns;

WAIT FOR 100 ns;

END PROCESS;

Ce processus produit un signal répétitif de période 100 ns avec 75ns pour le niveau bas et 25 ns pour le niveau haut.

### *4.5.2 Deuxième exemple: Mémoire Set-Reset*

Ce deuxième exemple est purement didactique et non synthétisable. Il modélise une mémoire Set-Reset avec différenciations des retards à la montée et à la descente.

ENTITY memoire\_rs IS

PORT ( s, r : IN BIT;

q, qb : OUT BIT);

END;

ARCHITECTURE processus OF memoire\_rs IS

CONSTANT Tplh : TIME := 2 ns;

CONSTANT Tphl : TIME := 1 ns;

SIGNAL qi : BIT := .0.;

SIGNAL qbi : BIT := .1.;

BEGIN

n1 : PROCESS

VARIABLE qtmp : BIT;

BEGIN

WAIT ON s, qi ;

qtmp := s NOR qi; -- la primitive NOR

IF qtmp /= qbi THEN -- Test du changement éventuel

IF qtmp = .0. THEN

qbi <= qtmp AFTER Tphl;

ELSE

qbi <= qtmp AFTER Tplh;

END IF;

END IF;

END PROCESS;

n2 : PROCESS

VARIABLE qtmp : BIT;

BEGIN

WAIT ON r, qbi ;

qtmp := r NOR qbi; -- la primitive NOR

IF qtmp /= qi THEN -- Test du changement eventuel

IF qtmp = .0. THEN

qi <= qtmp AFTER Tphl;

ELSE

qi <= qtmp AFTER Tplh;

END IF;

END IF;

END PROCESS;

q <= qi;

qb <= qbi;

END;



## *4.6 Affectation séquentielle des signaux*

### *4.6.1 Contexte*

Dans un **processus** déclaré ou dans un **sous-programme**, il ne peut y avoir que des ***instructions séquentielles*** c’est à dire exécutées dans l’ordre d’écriture l’une après l’autre. On retrouve ici la programmation classique et le débutant ne manquera pas de dire ça ressemble à du PASCAL..

### *4.6.2 Signal et Variable*

Une **variable** ne peut exister que dans un contexte séquentiel, elle est affectée **immédiatement**. Elle n’est pas visible à l’extérieur d’un processus

x := 1 + 2; -- x prend la valeur 3

Le signal est le seul objet qui peut être affecté soit de façon concurrente, soit de façon séquentielle selon le contexte. Il est l’agent de communication entre processus.

L’affectation du signal est différée à cause de son pilote. Lors de l’affectation séquentielle, le ou les couples *valeur\_future: heure\_de\_simulation* sont placés dans le pilote. La valeur sera (ou dans quelques cas conflictuels, ne sera pas) effectivement passée au signal au moment de la suspension du Process par une instruction WAIT. (Remarque: on peut être amené à écrire WAIT FOR 0 ns; pour forcer cette affectation à être immédiate).

***« C’est le pilote du signal qui est affecté et non le signal lui-même »***

s <= 1 + 2 AFTER 10 ns; -- le pilote de s reçoit le couple de valeurs 3,10

### *4.6.3 Illustration 1: différence dans les affectations*

L’exemple suivant illustre ce point fondamental.

ENTITY varsig IS

END;

ARCHITECTURE exercise OF varsig IS

SIGNAL aa, aaa : INTEGER := 3;

SIGNAL bb, bbb : INTEGER := 2;

BEGIN

P1: PROCESS

VARIABLE a: INTEGER :=7;

VARIABLE b: INTEGER :=6;

BEGIN

WAIT FOR 10 ns;

a := 1; --- a est égal à 1

b := a + 8 ; --- b est égal à 9

a := b - 2 ; --- a est égal à 7

aa <= a; -- 7 dans pilote de aa

bb <= b ; -- 9 dans pilote de bb

END PROCESS;

-- à l.heure H = 10 ns , aa prend la valeur 7 , bb prend la valeur 9

-- entre 0 et 10 ns aa vaut 3 et bb vaut 2

P2: PROCESS

BEGIN

WAIT FOR 10 ns;

aaa <= 1 ; -- 1 dans pilote de aaa

bbb <= aaa + 8; -- 11 dans pilote de bbb

aaa <= bbb - 2; -- 0 dans pilote de aaa

END PROCESS;

-- à l.heure H = 10 ns , aaa prend la valeur 0, bbb prend la valeur 11

-- entre 0 et 10 ns aaa vaut 3 et bbb vaut 2

END;

**Remarque**: De deux affectations successives d’un même signal, seule la deuxième compte. Le **pilote** du signal constitue une **mémoire** associée au signal.

***Illustration 2: Mémorisation implicite du signal***

Soit dans un PROCESS la séquence suivante (sur des type bit)

WAIT UNTIL h = .1.;

x <= e; -- e est mis dans le pilote de x

y <= x; -- la valeur actuelle de x est mis dans le pilote de y

On a réalisé un registre à décalage à 2 étages.

Alors que la séquence:

WAIT UNTIL h = .1.;

x := e; -- x prend la valeur e

y <= x; -- la valeur actuelle de x est mis dans le pilote de y montre que x n’est qu’un intermédiaire de calcul. En fait, ***y*** va recevoir ***e***. C’est une bascule de type D.

## *4.7 Processus équivalents aux instructions concurrentes*

Puisque «tout est processus», il est toujours possible de décrire un circuit par un processus explicite même si sa description en instruction concurrente est plus simple. On prendra comme exemple les trois instructions concurrentes les plus simples correspondant à des fonctions combinatoires.

### *4.7.1 Affectation simple*

**s <= a AND b AFTER 10 ns;**

Le processus équivalent demande cinq lignes au lieu d’une :

P1: PROCESS

BEGIN

**WAIT ON a, b ;**

s <= a AND b AFTER 10 ns;

END PROCESS;

### *4.7.2 Affectation conditionnelle*

**Neuf <= .1. WHEN etat = .1001 . ELSE ´0´ ;**

Le processus équivalent remplace le WHEN par une instruction IF :

P2: PROCESS

BEGIN

**WAIT ON etat ;**

IF etat = .1001. THEN

neuf <= ´1´ ;

ELSE

neuf <= ´0´ ;

END PROCESS;

### *4.7.3 Affectation avec sélection*

WITH ad SELECT

S <= e0 WHEN 0,

e1 WHEN 1,

e2 WHEN 2,

e3 WHEN OTHERS;

Le processus équivalent remplace le WITH par une instruction CASE :

P3: PROCESS;

BEGIN

**WAIT ON e0, e1, e2, e3 ;**

CASE ad IS

WHEN 0 => s <= e0 ;

WHEN 1 => s <= e1 ;

WHEN 2 => s <= e2 ;

WHEN 3 => s <= e3 ;

END CASE;

END PROCESS;

## *4.8 Description Structurelle*

C´est une description de type **hiérarchique** par liste de connexions tout à fait comparable à la réalisation d´un circuit imprimé comprenant des supports, des circuits et des équipotentielles.

Une description est structurelle si elle comporte un ou plusieurs composants (mot clé: **COMPONENT**). Ceux-ci jouent le rôle de support pour le câblage. Ils doivent donc avoir le même aspect extérieur que les modèles qu´ils vont supporter. Il est, bien sûre, possible de concevoir des structures imbriquées donc à plusieurs niveaux de hiérarchie.

Pour de telles descriptions, la **marche à suivre** est la suivante:

* Déclarer autant de composants que nécessaire: COMPONENT...(exemple lignes 10 et suivantes de *structure1*))
* Déclarer les listes de signaux internes nécessaires pour le câblage: SIGNAL... (exemple ligne 7de *structure1*)
* Déclarer quel modèle (couple entité-architecture) doit être utilisé sur chaque composant. C´est la configuration: clause USE...(exemple lignes 22,23 de *structure1*)
* Instancier chaque composant en indiquant sa liste de connexions: PORT MAP... (exemple lignes 26 et suivantes de *structure1*)

### *4.8.1 Exemple d’école*

Soit à décrire un simple compteur 4 bits synchrone avec autorisation de comptage et sortie report selon la spécification d’entité suivante:

ENTITY compteur4 IS

PORT ( h, raz, compter : IN BIT;

sortie : OUT BIT\_VECTOR( 3 DOWNTO 0);

plein : OUT BIT);

END;

Le modulo n’est pas fixé à priori et pourra raisonnablement être compris entre 9 et 16.

#### 4.8.1.1 Première description de type structurelle

La première idée de découpe structurelle est de considérer le cas général d´une machine de Moore et de séparer l´état de la machine (4 bascules D) des équations logique s de type état\_présent : état\_futur ici représentées par le bloc «calcul» qui fixe le modulo.



ARCHITECTURE **structure1** OF **compteur4** IS

-- déclaration des signaux nécessaires au câblage

SIGNAL d, s , sb : BIT\_VECTOR( 3 DOWNTO 0);

SIGNAL ra1 : BIT;

-- déclaration des composants nécessaires

COMPONENT **bascule**

PORT ( h, d ,raz, ra1 : IN BIT;

s, sb : OUT BIT);

END COMPONENT;

COMPONENT **calcul**

PORT ( s, sb : IN BIT\_VECTOR( 3 DOWNTO 0);

compter : IN BIT;

d : OUT BIT\_VECTOR( 3 DOWNTO 0);

plein : OUT BIT);

END COMPONENT;

-- configurations locales (si pas de CONFIGURATION séparée)

-- FOR ALL : bascule USE ENTITY WORK.bascule\_d(rtl);

-- FOR ALL : calcul USE ENTITY WORK.equations(par\_10);

BEGIN

ra1 <= ’0’;

**ba : bascule**

PORT MAP ( h, d(3), raz, ra1, s(3), sb(3));

-- instanciation par position

**bb : bascule**

PORT MAP ( h => h, d => d(2), raz => raz, ra1 => ra1,

s => s(2), sb => sb(2));

-- instanciation par dénomination

**bc : bascule**

PORT MAP ( h, d(1), sb => sb(1), s => s(1), ra1 => ra1, raz => raz);

-- instanciation par position et dénomination

**bd : bascule**

PORT MAP ( sb => sb(0), s => s(0), h => h, ra1 => ra1, d => d(0), raz => raz);

-- instanciation par dénomination

**combi : calcul**

PORT MAP ( s, sb, compter, d, plein);

sortie <= s;

END **structure1**;

Le bloc «calcul» est lui-même défini par une entité et une architecture associée (dans notre exemple, l´implantation d´une décade).

ENTITY **equations** IS

PORT ( s, sb : IN BIT\_VECTOR( 3 DOWNTO 0);

compter : IN BIT;

d : OUT BIT\_VECTOR( 3 DOWNTO 0);

plein : OUT BIT);

END;

--Implantation directe des équations d´une décade

-- A = b\*c\*d + a \* /d

-- B = /b \* c\*d + b\* ( /c+/d)

-- C = /a\*/c\*d + c\*/d

-- D = /d

---------------------------------------------------

ARCHITECTURE **par\_10** OF equations IS

SIGNAL pas\_compter : BIT;

BEGIN

pas\_compter <= NOT compter;

d(3) <= (compter AND s(2) AND s(1) AND s(0))

OR (s(3) AND (sb(0) OR pas\_compter)) ;

d(2) <= (compter AND sb(2) AND s(1) AND s(0))

OR ( s(2) AND (pas\_compter OR sb(1) OR sb(0)));

d(1) <= (compter AND sb(3) AND sb(1) AND s(0))

OR ( s(1) AND (pas\_compter OR sb(0)));

d(0) <= compter XOR s(0);

plein <= s(3) AND s(0);

END ;

De même, la bascule D possède son propre modèle (ou ses propres modèles)

-- Fichier : bascule\_d\_e.vhd

ENTITY **bascule\_d** IS

PORT ( h, d ,raz, ra1 : IN BIT;

s, sb : OUT BIT);

USE WORK.utils.all;

CONSTANT Tsetup : Time := 5 ns;

BEGIN

verif\_precond (h,d, Tsetup);

END **bascule\_d** ;

-- Fichier : bascule\_d\_a3.vhd

-- Non syhthétisable

-- bascule d avec différenciation des retards Tplh et Tphl

ARCHITECTURE **avec\_retards** OF **bascule\_d** IS

CONSTANT Tplh : TIME := 2 ns; -- concerne s ou sb

CONSTANT Tphl : TIME := 3 ns; -- concerne s ou sb

CONSTANT Tplh\_asyn : TIME := 1 ns; -- raz concerne sb

CONSTANT Tphl\_asyn : TIME := 2 ns; -- concerne s

BEGIN

p1: PROCESS (ra1, raz, h)

BEGIN

IF ra1 = .1. AND raz = .0. THEN

s <= .1. after Tplh\_asyn;

sb <= .0. after Tphl\_asyn;

ELSIF ( ra1 = .0. AND raz = .1.) THEN

s <= .0. after Tphl\_asyn;

sb <= .1. after Tplh\_asyn;

ELSIF ( ra1 = .1. AND raz = .1.) THEN

s <= .1. after Tplh\_asyn;

sb <= .1. after Tplh\_asyn;

ELSE IF (h = .1. AND h.EVENT) THEN -- front montant

IF (d = .0.) THEN

s <= .0. after Tphl;

sb <= .1. after Tplh;

ELSE s <= .1. after Tplh;

sb <= .0. after Tphl;

END IF;

END IF;

END IF;

END PROCESS p1;

END **avec\_retards**;

#### 4.8.1.2 Deuxième description de type structurelle

L´instruction **GENERATE** permet de réaliser des instanciations récurrentes, des instanciations conditionnelles.

Dans l´exemple précédent, les quatre instanciations de bascules peuvent ainsi être écrites plus simplement en considérant une variable de boucle i variant de 0 à 3. Cette variable rentre dans la définition des signaux nécessaires au câblage mais aussi à la génération automatique des étiquettes d´instanciation des composants.

ARCHITECTURE **structure2** OF **compteur4** IS

-- déclaration des signaux nécessaires au câblage

SIGNAL d, s, sb : BIT\_VECTOR( 3 DOWNTO 0);

-- déclaration des composants nécessaires

COMPONENT **bascule**

PORT ( h, d, raz : IN BIT;

s, sb : OUT BIT);

END COMPONENT;

COMPONENT **calcul**

PORT ( s, sb : IN BIT\_VECTOR( 3 DOWNTO 0);

compter : IN BIT;

d : OUT BIT\_VECTOR( 3 DOWNTO 0);

plein : OUT BIT);

END COMPONENT;

-- configuration (si pas d.unite CONFIGURATION)

-- FOR ALL : bascule USE ENTITY WORK.bascule\_d(simple);

-- FOR combi : calcul USE ENTITY WORK.equations(par\_10);

BEGIN

-- instanciation des composants

implant : FOR i IN 0 TO 3 GENERATE

**b : bascule**

PORT MAP (h, d(i), raz, s(i), sb(i));

END GENERATE;

**combi : calcul**

PORT MAP ( s, sb, compter, d, plein);

sortie <= s;

END structure2;

#### 4.8.1.3 Troisième description de type comportementale

En fait un compteur est une fonction de bas niveau qui se décrit très facilement sans décomposition, on préférera en général cette dernière description (synthétisable) de type comportementale et non hiérarchique.

ARCHITECTURE **decade** OF **compteur4** IS

USE WORK.utils.ALL;

SIGNAL tmp : natural RANGE 0 TO 9 := 0;

BEGIN

P1: PROCESS

BEGIN

WAIT UNTIL front\_montant(h) ;

IF raz = .1. THEN

tmp <= 0;

ELSIF compter = .1. THEN

IF tmp < 9 THEN

tmp <= tmp + 1;

ELSE

tmp <= 0;

END IF;

END IF;

END PROCESS;

sortie <= convert(tmp,4);

plein <= .1. WHEN tmp = 9 ELSE .0.;

END;

### *4.8.2 Configuration*

#### 4.8.2.1 Objet

Lorsqu.il y a hiérarchie, il est toujours nécessaire de préciser pour chaque composant instancié, quel est le modèle (couple entité-architecture) choisi. Cela se fait par l’emploi d.une clause USE qui peut être soit locale dans la zone de déclaration de l’architecture structurelle, soit explicitement dans une unité CONFIGURATION explicite. Le second cas offre l’avantage d’une compilation séparée et est parfaitement adapté aux gros Projets tandis que la première méthode est plus adaptée aux cas simples sans problème particulier de configuration (par exemple un seul modèle disponible).

#### 4.8.2.2 Application

La figure ci-dessous montre les différentes possibilités de configuration de compteur4



Pour cet exemple on pourrait considérer:

CONFIGURATION **compteur4\_cfg** OF **compteur4** IS

FOR **structure1**

FOR ALL : bascule USE ENTITY WORK.bascule\_d(rtl);

END FOR;

FOR ALL : calcul USE ENTITY WORK.equations(par\_10);

END FOR;

END FOR;

END compteur4\_cfg;

ou bien...

CONFIGURATION **compteur4\_cfg2** OF **compteur4** IS

FOR **structure2**

FOR implant (0 TO 3)

FOR ALL : bascule

USE ENTITY WORK.bascule\_d(avec\_retards);

END FOR;

END FOR;

FOR ALL : calcul USE ENTITY WORK.equations(par\_10);

END FOR;

END FOR;

END compteur4\_cfg2;

### *4.8.3 Compilation*

Les unités de conception sont bien compilables séparément mais il faut respecter les dépendances des fichiers. On peut ainsi compiler sans problème toutes les entités, puis toutes les architectures en remontant la hiérarchie du circuit, puis la configuration.

### *4.8.4 Simplifications éventuelles*

#### 4.89.4.1 Omission de la configuration

Si le COMPONENT porte le même nom que l’entité qu.il va permettre d’instancier, alors la configuration est implicite.

Dans l’exemple précédent les clauses de configuration ne sont plus nécessaires si les déclarations

COMPONENT bascule\_d et COMPONENT equations sont faites.

#### 4.8.4.2 Omission de la déclaration de COMPONENT

La méthode la plus expéditive (mais qui ne sera pas toujours portable) pour une description structurelle consiste à instancier directement une ENTITY sans déclaration de COMPONENT ni CONFIGUTATION. Cette méthode est illustrée dans le paragraphe suivant.

### *4.8.5 Description du test*

Le véritable intérêt du langage VHDL est de pouvoir à la fois décrire un circuit (en vue de synthèse) mais aussi bien d’être capable de construire la procédure de test du circuit. On réalise pour cela une véritable maquette comprenant le circuit à tester, les générateurs de stimuli à appliquer, puis éventuellement, les évaluations de sorties à effectuer.



#### 4.8.5.1 Illustration

La maquette de test (en anglais ***testbench***) est représentée par une entité dénuée d’entrées-sorties. Ainsi:

**ENTITY test\_compteur4 IS END;**

Représente la maquette de test du circuit *compteur4*. L’architecture associée à cette entité vide implante les différents éléments du test.

ARCHITECTURE tb OF test\_compteur4 IS

-- déclaration des signaux pour les connexions

SIGNAL h, raz, compteur , plein: bit;

SIGNAL sortie: bit\_vector(3 DOWNTO 0);

BEGIN

-- instanciation avec syntaxe simplifiée

C1: ENTITY WORK.compteur4(decade) PORT MAP(h, raz, compter, sortie, plein);

G1: ENTITY WORK.generateur(stimuli) PORT MAP (h, raz, compter):

END;