Langage VHDL et conception de circuits

Patrice NOUEL

TABLE DES MATIERES

(Dernières mise à jour : Juillet 2003)

1 Champs d'application du langage VHDL

1.1 Introduction

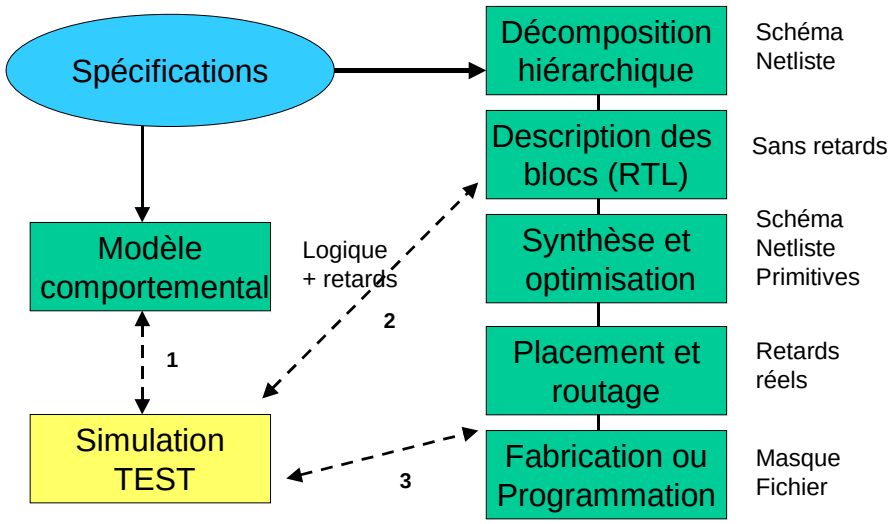
En analysant l'évolution de la production industrielle d'ASICS (Application Specific Integrated Circuit) ou de FPGA (Field Programmable Gate Array), on constate que ceux-ci, bénéficiant des progrès technologiques, sont de plus en plus complexes. On sait intégrer à l'heure actuelle sur silicium des millions de portes pouvant fonctionner à des fréquences supérieures à 600 MHz . On parle beaucoup de SOC (System On a Chip) . En effet, plus de 80% des ASICS futurs comporteront un ou plusieurs microprocesseurs.

Par ailleurs, si on considère qu'un ingénieur confirmé valide 100 portes par jour, il lui faudrait 500 ans pour un projet de 12 millions de portes et ceci pour un coût de 75 millions de dollars [3] . Ceci parait totalement absurde et si c'est pourtant réalisable cela est uniquement dû à l'évolution des méthodes de CAO (flot de conception en spirale, équipes travaillent en parallèle) intégrant en particulier la réutilisation efficace d'un savoir antérieur.

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.

1.2 Flot de conception – Importance du VHDL

Dans ce document, nous n'abordons pas le problème des SOC , nous utiliserons un flot de conception classique de type cascade. Le langage de description choisi est le VHDL (cela pourrait sans problème être le Verilog) et le niveau de complexité abordé est celui de la mise au point d'un circuit de quelques milliers de portes.



On observe sur cette figure le rôle fédérateur 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 validées par une même famille de fichiers de test eux-même 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.

1.3 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 prés 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. 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.

1.4 Historique

Le VHDL (**V**ery High Speed Integrated Circuit, **H**ardware **D**escription language) est le fruit du besoin de normalisation des langages de description de matériel (Première norme IEEE 1076-87 en décembre 1987).

Auparavant, chaque fournisseur de CAO proposait son propre langage de modélisation (GHDL chez GENRAD, BLM ou M chez Mentor Graphics, Verilog chez Cadence etc...) mais aussi un autre langage pour la synthèse et encore un autre pour le test. Au début des années 80, le ministère de la défense des Etats Unis (D.O.D) confiait le soin à Intermetrics, IBM et Texas Instruments de mettre au point ce langage. L'objectif était bien sûre de s'assurer une

certaine indépendance vis à vis des fournisseurs de matériels et de logiciels et ainsi une assurance de maintenabilité des équipements.

En 1994 la version IEEE 1076-93 suivie du standard IEEE 1164 fut établie. Celui-ci rajoute la notion de forces sur les signaux et est souvent appelé MVL9 (multivalued logic, nine values).

Il y aura aussi la norme IEEE 1076.3 (Numeric Standart pour la synthèse).

En 1995, afin de normaliser les méthodes de modélisation des retards des circuits ASIC ou FPGA, de grands industriels se sont associés dans la ”VITAL initiative” (VHDL Initiative Toward ASIC Libraries) fournissant des bibliothèques normalisées VITAL. Ceci est concrétisé par la norme IEEE 1076.4

Très récemment, le VHDL a connu une nouvelle extension avec la première norme IEEE-1076.1-1999 du langage de modélisation mixte et multi-technologique VHDL-AMS

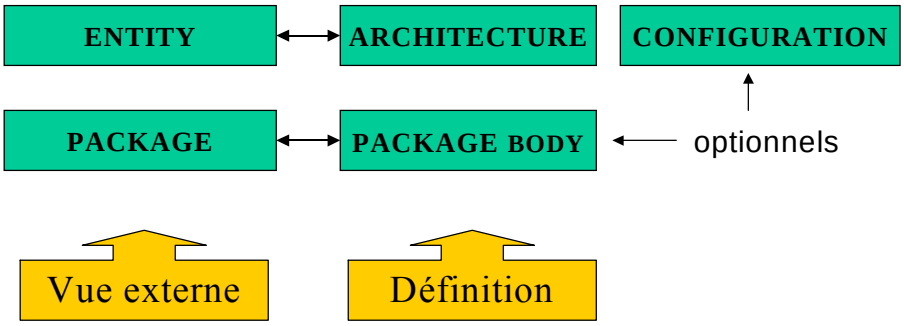
2 VHDL : Un langage à instructions concurrentes

2.1 Unités de conception

2.1.1 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 respecter 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...

2.1.2 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.

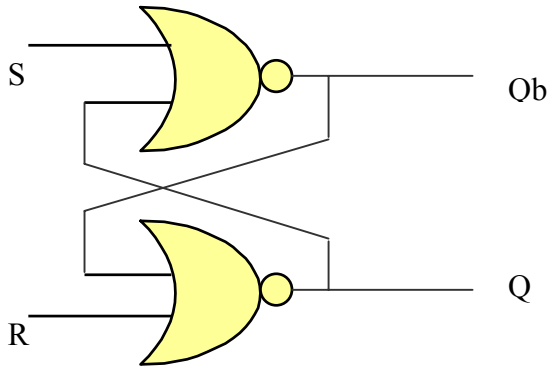
```
ARCHITECTURE simple OF comparateur IS
-- zone de déclaration vide
BEGIN
      egal <= '1' WHEN a = b ELSE '0';
END simple ;
```

2.2 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 caractérisent :

- 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
```

2.3 Les signaux

2.3.1 Signal avec un seul pilote

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

heure	état
0	´0´
10	´1´
25	´0´

:

Le pilote est en fait une mémoire associée au signal.

2.3.2 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.

2.4 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 »*

2.5 Simulation événementielle

La simulation d'un modèle porte sur les signaux utilisés pour le décrire. Elle est de type événementielle. 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.

2.5.1 Premier exemple: Génération d´horloge

```
SIGNAL h : bit;
BEGIN
horloge : PROCESS
```

```
BEGIN
                        h <= '0', '1' AFTER 75 ns;
                        WAIT FOR 100 ns;
END PROCESS;
```

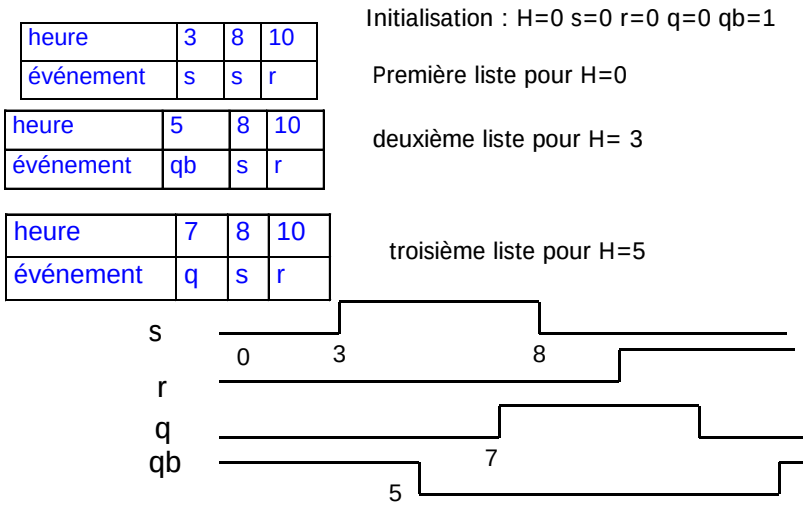
Ce processus produit un signal répétitif de période 100 ns avec 75ns pour le niveau bas et 25 ns pour le niveau haut.

2.5.2 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;
```



2.6 Affectation séquentielle des signaux

2.6.1 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".

2.6.2 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 différé 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

2.6.3 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 1 et bb vaut 2

 P2: PROCESS
BEGIN
                              WAIT FOR 10 ns;
                              aaa <= 1 ; -- 1 dans pilote de aa
                              bbb <= aaa + 8; -- 11 dans pilote de b
                              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.

2.7 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.

2.7.1 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;
```

2.7.2 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;
```

2.7.3 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;
```

2.8 Le délai Delta

On appelle ainsi un retard nul pour l'utilisateur mais correspondant à une unité d'itération du simulateur. La forme la plus simple du délai delta est la suivante:

```
s <= e;  -- équivalent à: s <= e AFTER 0 ns;
```

Au moment de la simulation, il n'y a que des processus communiquant entre eux par des événements sur des signaux. On peut alors voir trois types de temps.

- Le **cycle machine** caractéristique du processeur exécutant la simulation. A part des problèmes éventuels de patience à avoir, il ne rentre pas dans la discussion actuelle.
- Le **timestep** qui représente la résolution du simulateur (le plus petit temps simulable hormis 0). Il est fixé en standard à 1 femto seconde. C'est la base de temps utile pour la simulation.
- Le **délai delta** sans intérêt à priori pour l'utilisateur (car compté comme retard nul), mais essentiel si l'on veut comprendre les problèmes de simultanéités. Il représente une itération c'est à dire le temps d'exécution de toutes les instructions entre deux WAIT.

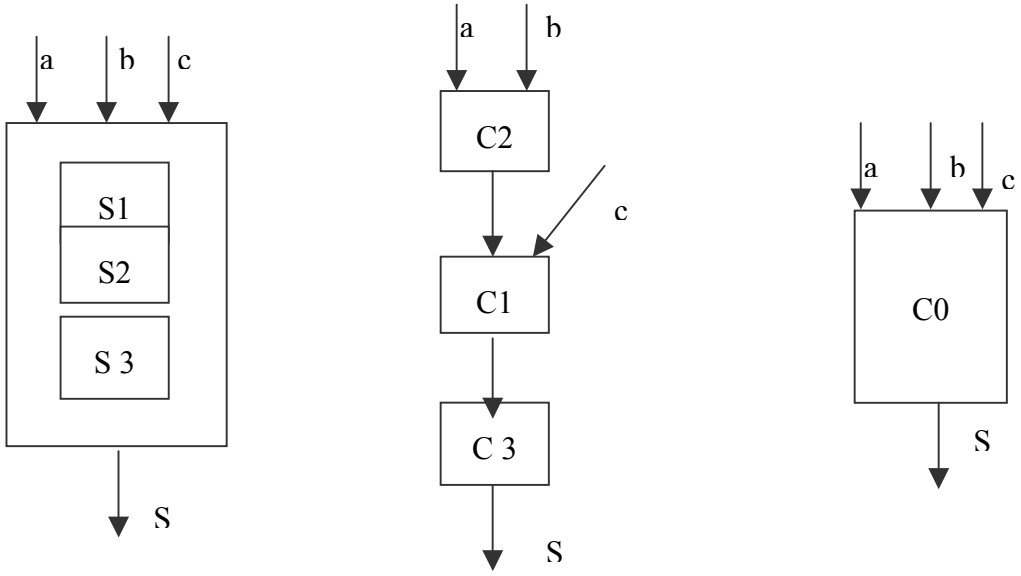
L'exemple simple de descriptions équivalentes pour l'utilisateur illustre les différences dans les délais delta.

```
ENTITY delta IS
  PORT ( a,b,c : IN BIT;
    s : OUT BIT);
END;

ARCHITECTURE un_process OF delta IS
 BEGIN

  un : PROCESS ( a, b, c)
VARIABLE i1, i2 : BIT; -- variables internes
BEGIN
-- l'ordre d'écriture est nécessaire
                    i1 := a AND b ; -- S1
                    i2 := c OR i1 ; -- S2
                    s <= NOT i2 ;   -- S3
END PROCESS;
 END un_process;

ARCHITECTURE concurrent OF delta IS
SIGNAL i1, i2 : BIT; -- signaux internes
 BEGIN
                    -- n'importe quel ordre pour les instructions
i2 <= c OR i1 ; -- C1
```

```
i1 <= a AND b ; -- C2
s <= NOT i2 ;   -- C3
END;

ARCHITECTURE plus_simple OF delta IS
BEGIN
-- une seule instruction concurrente
s <= NOT( c OR (a AND b)); -- C0
END;
```

La première et la dernière architecture contiennent un seul PROCESS = Une itération. Dans ces deux cas, la sortie sera affectée à heure de simulation + 1 δ.

La deuxième contient trois processus équivalents et concurrents relies par des signaux. Chaque processus nécessité une itération. Dans le pire cas, la sortie sera affectée à heure de simulation + 3 δ, ou alors ce sera à l'heure de simulation + 2 δ.



2.9 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*, page 4))

- Déclarer les listes de signaux internes nécessaires pour le câblage: SIGNAL... (exemple ligne 7de *structure1*, page 4)
- 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*)

2.9.1 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:

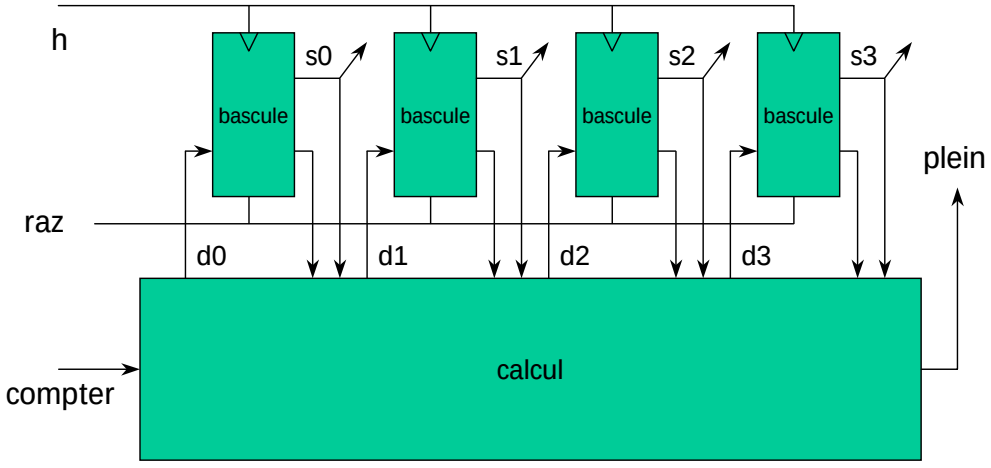
```
ENTITY compteur4 IS
PORT ( h, raz, compter : IN BIT;
                         sortie : OUT BIT_VECTOR( 3 DOWNTO 0);
                         plein : OUT BIT);
END;
```

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

2.9.1.1 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.

Schéma de compteur 4 bits



```
ARCHITECTURE structure1 OF compteur4 IS
-- déclaration des signaux nécessaires aux 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;
```

2.9.1.2 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 écrit 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;
```

2.9.1.3 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;
```

2.9.2 Configuration

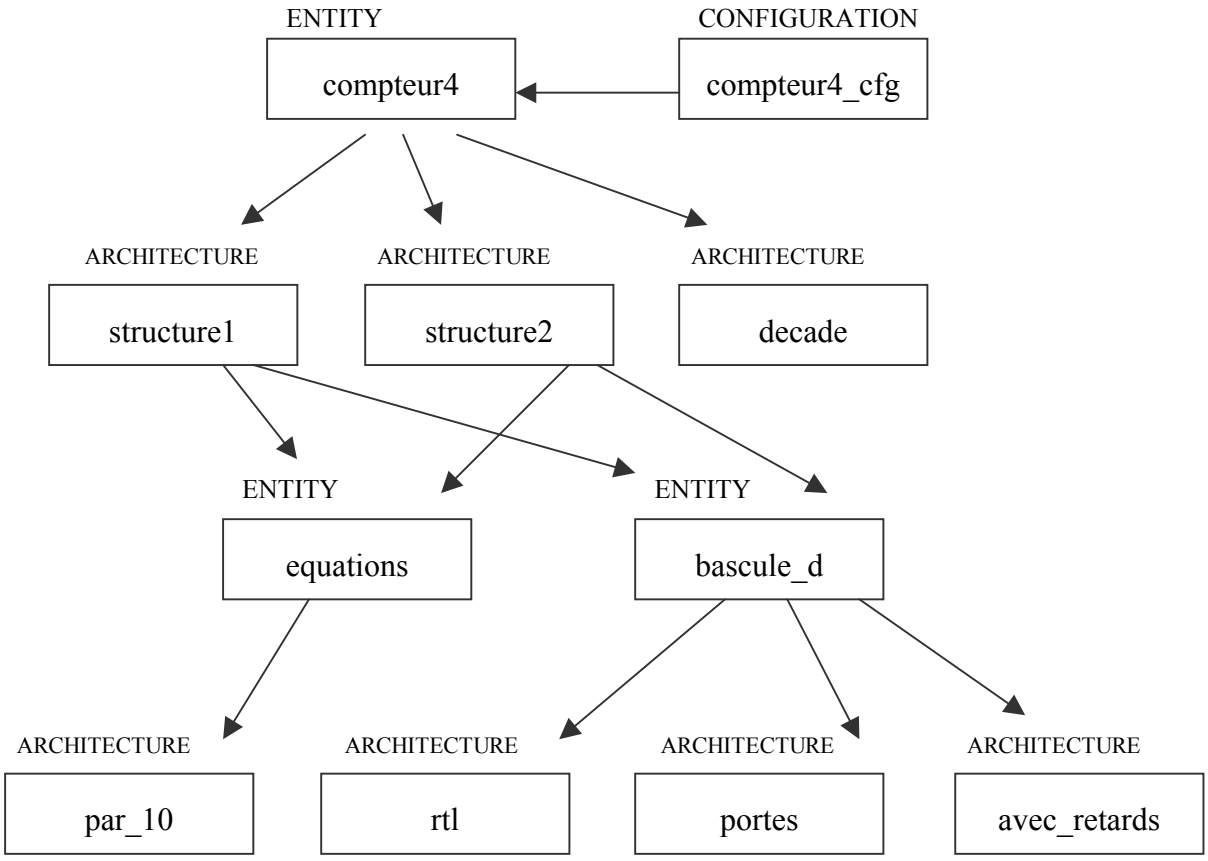
2.9.2.1 Objet

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).

2.9.2.2 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;
```

2.9.3 Compilation

Les unités de conception sont bien compilables séparément mais il faut respecter la 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.

De façon générale, au niveau de la compilation, le fichier n'est qu'un support de texte. Il peut comporter toutes les unités de conception mais **dans l'ordre de compilation**! Inversement il peut être souhaitable de s'astreindre à n'avoir qu'une unité par fichier. Compte tenu de leur contenu, les fichiers relatifs au projet compteur4 devront être compilés dans l'ordre suivant :

1. Fichier **utils.vhd** PACKAGEFichier **compteur4_e.vhd**: ENTITY compteur4
2. Fichier **bascule_d_e.vhd**: ENTITY bascule_d
3. Fichier **equations10.vhd**: ENTITY equations, ARCHITECTURE par_10
4. Fichier **bascule_d_a1.vhd**: ARCHITECTURE rtl
5. Fichier **bascule_d_a2.vhd** ARCHITECTURE portes
6. Fichier **bascule_d_a3.vhd**: ARCHITECTURE avec_retards
7. Fichier **compteur4_a1.vhd**: ARCHITECTURE structure1
8. Fichier **compteur4_a2.vhd**: ARCHITECTURE structure2
9. Fichier **compteur4_a3.vhd**: ARCHITECTURE decade
10. Fichier **compteur4_cfg.vhd**: CONFIGURATION

On remarquera que dans le cas de la configuration séparée, celle-ci représente le niveau supérieur du compteur.

2.9.4 Simplifications éventuelles

2.9.4.1 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

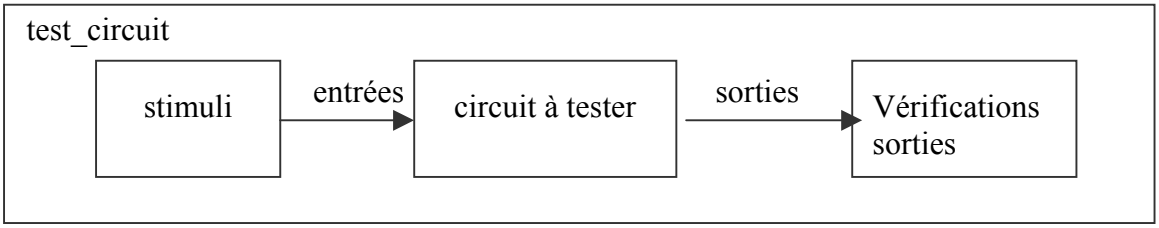
COMPONENT bascule_d et COMPONENT equations sont faites.

2.9.4.2 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.

2.9.5 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.



2.9.5.1 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, compter , 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;
```

3 Les spécificités du langage

3.1 Eléments lexicaux

3.1.1 Identificateurs et Mots réservés

Ils commencent par un caractère alphabétique et continuent par un caractère alphanumérique. Le caractère sous-lignée est autorisé, le nombre de caractères est quelconque; un caractère peut être indifféremment majuscule ou minuscule.

Ex: Mon_type VHDL1076 ... sont correct

74LS00 n'est pas correct car commençant par un chiffre

Un certain nombre d'identificateurs sont utilisés comme mots clefs du langage, ce sont les **mots «réservés»**. Ils sont américains, cela peut être un avantage lors de l'utilisation d'identificateurs français, mais attention à «transport» «report» etc...

ABS ACCESS AFTER ALIAS ALL AND ARCHITECTURE ARRAY ASSERT ATTRIBUTE BEGIN BLOCK BODY BUFFER BUS CASE COMPONENT CONFIGURATION CONSTANT DISCONNECT DOWNTO ELSE ELSIF END ENTITY EXIT FILE FOR FUNCTION GENERATE GENERIC GUARDED IF IN INOUT IS LABEL LIBRARY LINKAGE LOOP MAP MOD NAND NEW NEXT NOR NOT NULL OF ON OPEN OR OTHERS OUT PACKAGE PORT PROCEDURE PROCESS RANGE RECORD REGISTER REM REPORT RETURN SELECT SEVERITY SIGNAL SUBTYPE THEN TO TRANSPORT TYPE UNITS UNTIL USE VARIABLE WAIT WHEN WHILE WITH XOR

3.1.2 Littéraux

- Caractères: ‘0‘, ‘X‘, ‘a‘, ‘%‘
- Chaînes: ”1110101”, ”XX”, ”bonjour”, ”$^&@!”
- Chaînes de bits: B”0010_1101”, X”2D”, O”055”
- Décimaux : 27, -5, 4E3, 76_562, 4.25
- Basés : 2#1001#, 8#65_07#, 16#C5#E+2

3.1.3 Agrégats

Un agrégat permet de représenter une valeur pour un type composite. Il consiste en une séquence d’associations séparées par des virgules ou entre parenthèses. Pour un type tableau à une dimension, les associations correspondent aux éléments du tableau; pour un tableau à n dimensions les associations sont des tableaux de dimension n-1. pour un type enregistrement, les associations sont des éléments de l’enregistrement.

Chaque association comporte une partie formelle *optionnelle* (index pour un tableau, nom de l’élément pour un enregistrement ou encore le mot réservé OTHERS) suivie par le symbole d’association => et une partie actuelle à la suite. Les associations sans partie formelles sont dites **positionnelles**, les autres sont dites **par dénomination**. On peut utiliser les deux méthodes mais alors les associations positionnelles précédent toujours

les association par dénomination qui elles-mêmes précédent les éléments éventuels OTHERS.

Exemples:

(1, 2, 3, 4, 5) agrégat convenant pour un vecteur ou un enregistrement de 5 entiers.

(jour => 10, mois => juin, annnee => 1998) agrégat convenant à un type enregistrement ayant 2 champs entiers et un champ énuméré.

('1', '0', '0', '1', OTHERS => '0') agrégat convenant à un vecteur de bits. Seule les 4 premiers sont fixés, tous les autres (quel que soit leur nombre) valent '0'.

(('X', '0', 'X'), ('0', '0', '0'), ('x', '0', '1')) agrégat convenant à un tableau bidimensionnel de trois bits. Cela pourrait être aussi bien un tableau unidimensionnel de trois éléments eux-mêmes tableau de trois caractères.

3.1.4 Opérateurs prédéfinis

3.1.4.1 Logiques

Ils s'appliquent aux types BOOLEAN ou BIT ou STD_ULOGIC...

AND, OR, NAND, NOR, XOR, NOT

3.1.4.2 Relationnels

=, /=, <, <=, >, >= Le résultat est booléen

3.1.4.3 Arithmétiques

+, -, *, /, **, MOD, REM

La définition exacte de REM (reste de la division de 2 entiers A et B) est donnée par l´expression: A = (A/B)*B + (A REM B)

La définition exacte de MOD est donnée par l´expression: A = B*N + (A MOD B), l´expression entre parenthèses est inférieurs en valeur absolue à B et prend le signe de B.

3.1.4.4 Concaténation d´éléments de tableaux: &

"bon" & "jour" produira "bonjour" ;

"101"&"10" produira "10110"

3.2 Types, Sous-types

Le VHDL est un langage fortement typé. Chaque objet doit être déclaré et appartenir à un type connu (ensemble de valeurs possibles).

Pour illustrer ce paragraphe, nous prendrons en priorité les types prédéfinis de la bibliothèque standard 1076 ou 1164.

3.2.1 Scalaires

Ils sont ordonnés (=, /=, <, <=, >, >=) et peuvent être restreints « RANGE *limite_basse* TO *limite_haute*» ou « RANGE *limite_haute* DOWNTO l*imite_basse*»

3.2.1.1 Enumérés

Ensemble de littéraux.

```
TYPE boolean IS ( false, true);
TYPE bit IS (‘0‘, ‘1‘); -- type énuméré de 2 caractères
TYPE type_etat IS (init, debut, fin);
TYPE couleur IS ( bleu, blanc, rouge); -- cocorico
```

3.2.1.2 Entiers

```
TYPE integer IS RANGE -2147483648 TO 2147483647;
SUBTYPE natural IS integer RANGE 0 TO integer´high;
TYPE pour_mon_compteur IS RANGE 0 TO 99;
```

3.2.1.3 Flottants

```
TYPE real IS RANGE -1.0E38 TO 1.0E38;
```

3.2.1.4 Physique

```
TYPE time IS RANGE -2147483647 TO 2147483647
UNITS
                        fs;
                        ps = 1000 fs;
                        ns = 1000 ps;
                        us = 1000 ns;
                        ms = 1000 us;
                        sec = 1000 ms;
                        min = 60 sec;
                        hr = 60 min;
END UNITS;
```

3.2.2 Composites (Tableaux et enregistrement)

3.2.2.1 Tableaux

Les tableaux (ARRAY) peuvent être à une ou plusieurs dimensions, contraints ou non contraints.

```
TYPE bit_vector IS ARRAY ( natural RANGE <> OF bit); -- non contraint
TYPE string IS ARRAY ( positive RANGE <> OF character);
TYPE matrice IS ARRAY ( 1 TO 4, 0 TO 7) OF bit; -- matrice de bits à 2 dimensions 4X8
```

3.2.2.2 Enregistrements

Les enregistrements (RECORD) permettent de définir des collections de valeurs elles-mêmes typées comme dans l´exemple suivant:

```
TYPE date IS RECORD
                    jour  : natural RANGE 1 TO 31;
                    mois  : string;
                    annee : integer RANGE 0 TO 4000;
END RECORD;
```

La déclaration CONSTANT date_de_naissance : date := (29, JUIN, 1963); est parfaitement correcte.

On peut accéder à un élément de l´enregistrement en désignant son champ par un point comme dans... date_de_naissance.jour ou date_de_naissance.mois etc... De manière similaire si un SIGNAL ou une VARIABLE est de type RECORD, il est possible de faire des affectations partielles d´un élément. Pointeur

Le type pointeur (ACCESS) n´est d´aucune utilité lors de description au niveau synthèse. C´est donc un type peu employé en électronique. Par contre, il est essentiel en modélisation et lors du travail sur les fichiers.

```
TYPE line IS ACCESS string;
```

définie une ligne comme un pointeur sur une chaîne de caractères.

3.2.3 fichier

Les fichiers (FILE) sont très utilisés pour stocker des données telles que Mémoires ROM, Données de test.

TYPE text IS FILE OF string;

3.2.4 Sous-Types

Un sous-type reste compatible avec son type d'origine contrairement à un type nouvelement créé.

SUBTYPE narural IS integer RANGE 0 TO integer'high : *natural* reste compatible avec *integer*

TYPE nouvel_entier IS RANGE 0 TO 99: *nouvel_entier* est un type différent

3.2.5 Conversion de type

La conversion de type, <type>(<expression>), est prévue en VHDL mais avec la restriction que les types doivent être en relation...

```
TYPE donnee_longue  IS RANGE 0 TO 1000;
TYPE donnee_courte IS RANGE 0 TO 15;

SIGNAL data : donnee_longue;
SIGNAL donnee :donnee_courte;

data <= donnee_longue(donnee * 5); -- correct
```

On a 2 types différents à droite et à gauche de l´affectation mais le compilateur est capable de faire la transformation (data et donnee sont des entiers).

Pour des types qui ne sont pas en relation, il est necessaire de disposer de fonctions de conversion. On en trouve un grand nombre dans les paquetages standards.

3.3 Structures de contrôle

Ces structures sont des instructions séquentielles et sont donc obligatoirement placée dans des PROCESS ou dans des sous-programmes PROCEDURE ou FUNCTION. Pour des instructions conditionnelles concurrentes, on se reportera au chapitre 1.

3.3.1 IF...THEN...ELSIF...ELSE...END IF

La structure IF permet d'exécuter une ou plusieurs instructions séquentielles selon une condition booléenne VRAIE ou FAUSSE.

ELSIF est une contraction entre ELSE et IF et permet de faire l'économie d'écriture d'un END IF

```
IF a = '1' THEN
        s := '1';
ELSIF b = '1' THEN
        s := '1';
ELSE
        s := '0';
END IF;
```

Dans ce petit exemple, s ne prend la valeur '0' que si a = '0' ET que b = '0'. En effet, l'évaluation se fait selon l'ordre d'écriture. Est-ce que a = '1' ? puis si a= '0' est ce que b = '1' puis si a = '0' Et b = '0' alors s := '0'. On aurait pu aussi bien ecrire:

```
IF a = '1' OR  b = '1' THEN
        s := '1';
ELSE
        s := '0';
END IF;
```

3.3.2 CASE ... IS...WHEN...

La structure CASE permet d'exécuter une ou plusieurs instructions séquentielles selon le résultat d'une expression. Celle-ci doit être un entier, un type énuméré, ou un tableau de caractère à une dimension. Toutes les valeurs de l'expression doivent apparaître dans les différents choix. Chaque valeur de choix est différentes des autres.

```
CASE entree IS -- entree est de type tableau de caractere
        WHEN "11" => s:= '1';
        WHEN "01" => s:= '1';
        WHEN "10" => s:= '1';
        WHEN OTHERS => s :='0';
END CASE;
```

Cet exemple est le même que précédemment, il a l'avantage d'offrir un style table de vérité. "00" à la place de OTHERS est correct s'il est sure que le tableau n'a que 4 éléments. Dans le

cas contraire OTHERS permet de faire une affectation unique pour tous les cas en suspend (par exemple ”00”, ”UU”, ”X0”...).

3.3.3 LOOP...

La structure LOOP permet d'exécuter une ou plusieurs instructions séquentielles selon un scénario itératif défini par un FOR ou un WHILE.

On peut intervenir sur le déroulement des itérations par les instructions EXIT et NEXT.

Les variables de boucle n'ont pas besoin d'être déclarées.

```
boucle1: -- étiquette optionnelle
FOR i IN 0 TO 10 LOOP
        b := 2**i; -- calcul des puissances de 2
        WAIT FOR 10 ns; -- toutes les 10 ns
END LOOP;
```

ou encore

```
boucle2:
WHILE b < 1025 LOOP
        b := 2**i; -- calcul des puissances de 2
        WAIT FOR 10 ns; -- toutes les 10 ns
END LOOP;
```

ou encore:

```
boucle3:
LOOP
        b := 2**i; -- calcul des puissances de 2
        WAIT FOR 10 ns; -- toutes les 10 ns
        EXIT boucle3 WHEN b > 1000;
END LOOP;
```

3.3.4 ASSERT

L'instruction ASSERT est la seule qui permet de générer des messages à l'écran. Elle n'a donc pas de sens en synthèse (circuit) sinon pendant la phase de mise au point, par contre elle correspond exactement à la démarche du test par assertion. On vérifie des propriétés du circuit par un ensemble d'affirmation (vraies ou fausses).

```
ASSERT FALSE REPORT ” Ceci n'est qu'un petit bonjour a l'écran”;
```

On a détourné l'instruction pour n'avoir que l'affichage. Autre exemple pris dans un contexte séquentiel:

```
WAIT UNTIL h = ’1’;
ASSERT s = ’1’ REPORT ” la sortie vaut ’0’ et ce n'est pas normal” SEVERITY WARNING;
```

Exemple typique de test d'un signal .

3.4 Sous-programmes

Les sous-programmes permettent de regrouper une suite instructions séquentielles pour décrire une seule fois un calcul particulier devant (ou non) se répéter par exemple: conversion de type, partie de processus, fonction de vérification de timing, fonction de résolution, générateur de stimuli etc...

Ce sont des éléments de **structuration** pour un circuit.

3.4.1 Procédures et fonctions

Les procédures peuvent agir par effet de bord (modification de l´environnement) alors que les fonctions retournent un résultat typé.

```
PROCEDURE  horloge (  SIGNAL h : OUT Bit; th, tb :  TIME)  IS
 BEGIN
        LOOP
                h <= '0', '1'  AFTER tb;
                WAIT  FOR tb + th ;
        END LOOP;
END;
```

Cette procédure définit un signal d´horloge qui est ensuite instanciée de façon concurrente sur un signal C

```
CLK: horloge(C, 10 ns, 20 ns) ;
```

Les fonctions retournent une valeur typée comme dans l´exemple suivant de fonction de conversion d'un bit_vector en entier qui n'existe pas dans la bibliothèque standard 87

```
FUNCTION Convert (b : BIT_VECTOR) RETURN NATURAL IS
        VARIABLE temp  :   BIT_VECTOR ( b'LENGTH - 1 DOWNTO 0) := b;
        VARIABLE valeur : NATURAL := 0;
        BEGIN
                FOR i IN temp'RIGHT TO temp'LEFT
                LOOP
                        IF temp(i) = '1' THEN
                                valeur :=  valeur + (2**i);
                        END IF;
                END   LOOP;
                RETURN  valeur;
        END;
```

L´utilisation dans une affectation séquentielle ou concurrente de signal est possible.

```
N <= Convert(B) after 1 ns;
```

Le passage des paramètres se fait soit par position (ordre prédéfini) soit par dénomination) ordre quelconque. (voir agrégats)

La fonction de conversion d'un entier en bit_vector serait :

```
FUNCTION Convert (n,l: NATURAL) RETURN BIT_VECTOR  IS
                VARIABLE temp :   BIT_VECTOR( l-1 DOWNTO 0);
                VARIABLE valeur : NATURAL := n;
```

```
BEGIN
 FOR i IN temp'RIGHT TO temp'LEFT
        LOOP  temp(i) := BIT'VAL( valeur MOD 2);
             valeur :=  valeur / 2;
        END   LOOP;
RETURN temp;
END;
```

3.4.2 Surcharge

La surcharge pour deux sous-programmes consiste à tolérer le même nom et des profils différents (nombre ou type de paramètres qui différent). Par exemple, les deux fonctions du Paquetage UTILS n´ont pas le même nombre de paramètres:

```
FUNCTION Convert (n,l: NATURAL) RETURN BIT_VECTOR  IS
FUNCTION Convert (b : BIT_VECTOR) RETURN NATURAL IS
```

La surcharge des opérateurs est possible comme dans exemple suivant qui définie l´addition de 2 bit_vectors.

```
FUNCTION  ”+” ( a, b : bit_vector) RETURN bit_vector;
```

3.5 Blocs

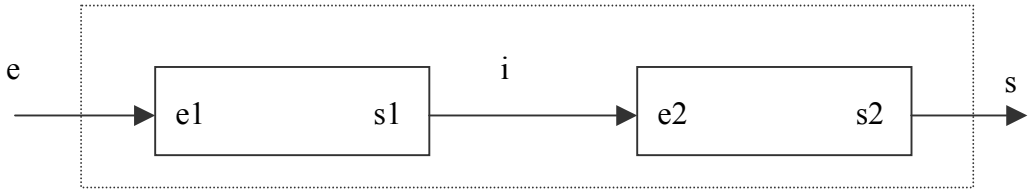
Le bloc (BLOCK) est élément de base de la structuration de la description. Il permet de définir une hiérarchie. mais il n´est pas exportable et reste interne à une architecture. L´entité apparaît ainsi comme un bloc exportable. Plusieurs blocs peuvent être imbriqués.

Dans un bloc, on ne trouve que des instructions concurrentes. Il est possible d’y définir des frontières (PORT) , d’y mettre en commun des conditions logiques (GUARD) , de le concevoir comme paramétrable (GENERIC).

3.5.1 Blocs avec Ports

```
ENTITY es IS
PORT(e IN bit ; -- une entrée
             s : OUT bit); -- une sortie
END es;

ARCHITECTURE deux_blocs OF es IS
SIGNAL i : bit; -- signal intermédiaire
BEGIN
  B1:BLOCK PORT (e1 : IN bit; s1 ; OUT bit);
        PORT MAP ( e1 => e, s1 => i);
  BEGIN
                s1 <= e1; -- notations locales
  END BLOCK B1;

  B2:BLOCK PORT (e2 : IN bit; s2 ; OUT bit);
        PORT MAP ( e2 => i, s2 => s);
  BEGIN
```

```
        s2 <= e2; -- notations locales
  END BLOCK B2;
END deux_blocs;
```



3.5.2 Bloc avec condition de garde

s <= GUARDED e ; signifie s reçoit e si la condition de garde est vraie

3.5.2.1 Exemple1: latch implicite

```
SIGNAL d, q, en : bit;
BEGIN
latch: BLOCK ( en = ´1´)
BEGIN
        q <= GUARDED d; -- instruction concurrente
END BLOCK;
```

Le processus équivalent serait :

```
latch : PROCESS(en,d)
BEGIN
        IF en = ´1´ THEN
                q <= d;
        END IF;
END PROCESS;
```

3.5.2.2 Exemple2 : registre avec raz

```
registre : BLOCK (h´event AND h = ´1´)
BEGIN
        q <= GUARDED ´0´ WHEN raz = ´1´ ELSE d;
END BLOCK;
```

3.6 Généricité

La généricité est un moyen de transmettre une information à un bloc ou à un couple entité-architecture. Cette information est statique pour le bloc. Vue de extérieur, c´est un paramètre. En fait, il s´agit d´une constante qui sera fixée au moment de l´ instanciation du bloc ou du composant.

3.6.1 Exemple:Compteur générique

```
ENTITY compteur_gen IS
```

```
GENERIC (CONSTANT Nb_Bits : natural := 4;
       Modulo : natural :=16;
       Tp : time := 2 ns);
PORT ( h : IN Bit;
            sortie :  OUT bit_vector (0 to Nb_bits-1);
     retenue :  OUT Bit);
BEGIN
ASSERT ----- controle des parametres
            Modulo <= 2**Nb_bits
            REPORT "erreur sur les parametres"
     SEVERITY warning;
END compteur_gen;
```

Les valeurs par défaut des constantes ne sont pas obligatoires, celles-ci devant être fixées au moment de l´instanciation d´un composant associé à cette entité. par exemple...

```
          C1: compt
               GENERIC MAP (5,17,2 ns)   -- compteur modulo 17
               PORT MAP  (h,sortie,r);
```

3.7 Attributs

Un attribut est une caractéristique associée à un type ou un objet qui pourra être évalué soit au moment de la compilation soit dynamiquement au cours de la simulation. Chaque attribut est référencé par son nom consistant en un préfixe, une apostrophe et l´attribut lui-même. Le préfixe doit être un type, sous-type, tableau, ou bloc.

3.7.1 Définition d´attributs

Exemple montrant comment créer son propre attribut.

3.7.1.1 Déclaration

Elle définie l´attribut

```
SIGNAL bus_adresse : bit_vector(15 DOWNTO 0);
ATTRIBUTE moitie : integer; -- declaration
```

3.7.1.2 Spécification

Elle associe l´attribut à un ou plusieurs membres d´ une «classe d´entité». Les classes d´entité sont ENTITY, ARCHITECTURE, CONFIGURATION, PROCEDURE, FUNCTION, PACKAGE, TYPE, SUBTYPE, CONSTANT, SIGNAL, VARIABLE, COMPONENT, LABEL.

```
ATTRIBUTE moitie OF bus_adresse : SIGNAL IS bus_adresse´LENGTH/2;
```

------- bus_adresse ´moitie -- retourne 8

3.7.2 Attributs prédéfinis

Ils font partie du langage et on se reportera aux exemples présentés dans la documentation en annexe.

3.7.2.1 Attributs de type et de sous-types

3.7.2.1.1 Type de base

T'Base retourne le type de base de son préfixe. Pour un type donné, le type de base est le type lui-même. Pour un sous-type le type de base est le type à l'origine du sous-type. Par exemple, le type de base d'un *natural* est *integer.* Le type de base d'un *sous-type de natural* est toujours *integer.*

T'Base ne s'utilise que comme préfixe d'un autre attribut.

3.7.2.1.2 Bornes des types scalaires

Les attributs suivants s'appliquent aux types ou sous-types scalaires, et retournent une de leurs bornes limites. Un type scalaire est soit entier, réel, énuméré ou type physique. Pour illustrer ces attributs, nous considérerons les exemples suivants:

```
TYPE T1 IS 1 TO 10;
TYPE T2 IS 10 TO 1;
SUBTYPE S1 IS T1 RANGE 3 TO 5;
SUBTYPE S2 IS T1 RANGE 4 DOWNTO 3;
SUBTYPE S3 IS T2 RANGE 1 DOWNTO 2; -- un subtype NULL
```

T'Left retourne 'la limite la plus gauche du type préfixe. T1'Left vaut 1, T2'Left vaut 10, S1'Left vaut 3, S2'Left vaut 4, S3'Left vaut 1.

T'Right retourne la borne droite de définition. T1'Right retourne 10, T2'Right 1, S1'Right 5, S2'Right 3, S3'Right 2.

T'High retourne la plus grande des deux limites du type ou sous-type préfixe. T1'High et T2'High donnent 10, S1'High 5, S2'High 4, S3'High 2.

T'low retourne la plus petite des deux limites du type ou sous-type préfixe. T1'Low et T2'Low retournent 1, S1'low et S2'Low retournent 3 et S3'low retourne 1.

On remarque que pour un type défini en ordre ascendant T'Low≡T'Left et T'High≡T'Right.

3.7.2.1.3 Conversion Valeur-Position

Valable pour des types entier, énuméré ou physique, ces attributs permettent la conversion entre variable de position d'un type (compte a partir de T'Left jusqu'a la position courante) et la valeur dans le type.

T'Pos(X) retourne la variable de position pour la valeur X. Le résultat de cette fonction est un entier initialisé à 0 (T'Pos(T'Left)). Ainsi, T1'Pos(4)=3 puisque la position de 1 est 0, de 2 est 1...De même, T2'Pos(2)=8, S1'Pos(4)=1. Mais S3'Pos(1) n'a pas de valeur.

T'Val(X) est la fonction inverse. X représente une position et la fonction retourne la valeur selon la définition du type ou sous-type. T'Val(T'Pos(X))□X tant que X est une valeur dans la définition. T1'Val(0) =1, T2'Val(T1'Pos(S2'Left) vaut 7. Character´VAL(49 + i) retourne le caractère '0'si i = 0, '1'si i = 1 etc.... ce qui permet de convertir facilement un entier en caractère.

3.7.2.1.4 Déplacement de position

Ces quatre attributs permettent de d'incrémenter ou de décrémenter une position. C'est une valeur dans le type ou le sous-type qui est retournée.

T'Succ(X) retourne le successeur de X dans le type de base. On a l'équivalence T'Succ(X) ≡T'Base'Val(T'Pos(X)+1). cela suppose que X_T'base'High. Une erreur sera générée si on essaye d'évaluer T'Succ(T'base'High).

T'Pred(X) retourne le prédécesseur de X dans le type de base. On a T'Pred(X) ≡T'base'Val(T'Pos(x)-1) avec X_T'Base'Low. Une erreur sera générée si on essaye d'évaluer T'Pred(T'base'Low).

T'LeftOf(X) retourne la valeur gauche de X. C'est donc la même chose que le prédécesseur pour un type défini en mode descendant et le même chose que le successeur pour un type défini de façon ascendante. Il y aura erreur lors de l'évaluation de T'LeftOf(T'base'Left).

T'RightOf(X) retourne la valeur droite de X. C'est donc la même chose que le prédécesseur pour un type défini en mode ascendant et le même chose que le successeur pour un type défini de façon descendante. Il y aura erreur lors de l'évaluation de T'RigtOf(T'base'Right).

3.7.2.2 Attributs de tableau

Pour ces attributs, le préfixe doit être un objet (signal, variable, constante) de type tableau, un sous-type tableau contraint,.un type pointeur ou un alias y faisant référence.

Dans tous les cas l'attribut a un paramètre optionnel N, dimension du tableau, par défaut 1. ce paramètre permet de faire porter l'attribut sur une dimension particulière du tableau.

Comme exemple, nous définirons:

```
TYPE a1 IS ARRAY (1 TO 10, 10 DOWNTO 1) OF INTEGER;
SUBTYPE word IS BIT_VECTOR (31 DOWNTO 0);
```

3.7.2.2.1 Bornes d'index de tableau

A'Left[(N)] retourne la borne gauche du Nème index de A. a1'Left=a1'Left(1) retournera 1, a1'Left(2) vaut 10, word'Left vaut 31.

A'RIGHT[(N)] retourne la borne droite du Nème index de A. a1'Right vaut 10,a1'Right(2) vaut 1 et word'right vaut 0.

A'High[(N)] retourne la borne supérieure du Nème index de A. a1'High et a1'High(2) valent 10, word'High vaut 31.

A'Low[(N)] retourne la borne inférieure du Nème index de A. **a**1'Low et a1'Low2) valent 1, word'Low vaut 0.

3.7.2.2.2 Intervalle de variation d'index de tableau

Ces attributs sont très utilisés lors de l'écriture de boucles ou pour spécifier à partir d'un type tableau d'autres types ou sous-types.

A'Range[(N)] retourne l'intervalle de variation de la dimension N de A. L'ordre prédéfini est inchangé. a1'Range retourne 1 TO 10, a1'Range(2) retourne 10 DOWNTO 1 et word'Range retourne 31 DOWNTO 0.

La boucle: FOR i IN s´RANGE LOOP correspond à 16 itérations (de 15 à 0) si s est déclaré ainsi: SIGNAL s : bit_vector(15 DOWNTO 0);

A'Reverse_Range[(N)] retourne l'intervalle de variation de la dimension N de A. L'ordre prédéfini est inversé. a1'Reverse_Range retourne 10 DOWNTO 0 a1'Reverse_Range(2) retourne 1 TO 10 et word'Reverse_Range retourne 0 TO 31.

A'Length[(N)] retourne le nombre d'éléments du Neme index de A. Pour un tableau NULL, le résultat sera 0. a1'Length et a1'Length(2) valent 10, word'Length vaut 32.

3.7.2.3 Attributs de signal

Pour cette catégorie d'attribut, le préfixe doit être un signal. Le type du résultat peut être signal ou fonction.

3.7.2.3.1 Attribut signal

Ces quatre attributs retournent un signal. Les deux premiers attributs sont sensibles à des événements, les deux autres à des transactions portant sur le signal préfixe. Les trois premiers possèdent un paramètre optionnel de type Time qui ne peut être négatif. Par défaut, ce paramètre vaut 0. ces attributs ne peuvent pas être utilises à l'intérieur de fonctions ou de procédures.

S'Delayed[(T)] est un signal du même type que S dont les valeurs sont retardées de T unités de temps. Cette relation peut être décrite par le processus équivalent suivant:

```
P: Process (S)
Begin
                        R <= Transport S AFTER T;
End Process;
```

Si on suppose que le type de S est bien le même que celui de R et que la valeur initiale de R et de S sont identiques, alors pour tout T >= 0 ns, S'Delayed(T)= R toujours.

Remarque: S'delayed(0 ns) _ S au moment précis où S change c'est à dire lorsqu'il y a un événement sur S.

S'Stable[(T)] est un signal de type booléen qui est vrai si S n'a pas changé de valeur dans l'intervalle de temps écoulé T. S'Stable est faux si S'Event est vrai.

On utilise couramment cet attribut pour réaliser des vérifications de propriétés temporelles telles que temps de pré-conditionnement ou temps de maintien.

```
Verif_Tsetup: PROCESS
BEGIN
WAIT UNTIL h'Event AND h = '1';
ASSERT donnee'STABLE(Tsetup)
```

```
                        REPORT " violation de temps de preconditionnement";
END PROCESS

Verif_Thold: PROCESS
BEGIN
WAIT UNTIL h'DELAYED(Thold)'EVENT AND
                                h'DELAYED(Thold) = '1';
ASSERT donnee'STABLE(Tsetup)
                        REPORT " violation de temps de maintien";
END PROCESS
```

S'Quiet[(T)] est un signal de type booléen qui est vrai si S n'a pas connu de transaction dans l'intervalle de temps écoulé T.

S'Transaction est un signal de type Bit qui change de valeur à chaque transaction sur S. cet attribut permet ainsi de transformer une transaction en événement. On peut ainsi sensibiliser indirectement un Process par une transaction.

3.7.2.3.2 Attribut fonction

Puisque les attributs de signal retournant un signal ne peuvent pas être placés dans un sous-programme, il existe pour cela des attributs fonction.

S'event est un booléen, vrai quand S'stable(0 ns) est faux, c'est à dire au moment précis où S change de valeur.

S'Active est un booléen, vrai quand S'Quiet(0 ns) est faux, c'est à dire au moment précis où il y a une transaction sur S.

S'Last_Event retourne le laps de temps écoulé depuis le dernier changement de valeur de S.

S'Last_Active retourne le laps de temps écoulé depuis la dernière transaction sur S.

S'Last_Value retourne la valeur qu'avait S juste avant le dernier événement le concernant. (S'Last_Value=S tant que S n'a pas changé une seule fois).

Exemples de fonctions relatives types standards :

```
FUNCTION front_montant  (SIGNAL s : bit) RETURN BOOLEAN IS
  BEGIN
    RETURN (s'EVENT AND s = '1');
  END;

 FUNCTION front_descendant (SIGNAL s :BIT) RETURN BOOLEAN IS
  BEGIN
    RETURN (s'EVENT AND s = '0');
  END;
```

3.7.3 Attributs de bloc

Le préfixe pour ce type d'attribut doit être le nom d'une architecture ou l'étiquette identifiant un bloc

B'Behavior est un booléen vrai si et seulement si B ne contient pas d'instanciation de composant(COMPONENT).

B'Structure est un booléen vrai pour une structure pure. B ne doit contenir aucune affectation de signal concurrente ou dans un processus.

Remarque: Il est possible d'avoir simultanément B'Behavior et B'Structure à FALSE mais il est impossible de les avoir simultanément à TRUE.

3.8 Fonctions de résolution

Lorsqu´un signal est multi-sources, il est nécessaire de lui affecter une fonction de résolution pour ses pilotes. Ce cas est fréquent en électronique, en particulier pour tous les problèmes de logique "câblée" ou de logique trois-états à la base de la notion de bus. La fonction de résolution reçoit un vecteur en entrée (les sources possibles) et fournit une valeur d'arbitrage en retour. La fonction de résolution est appellée automatiquement par le simulateur.

3.8.1 Exemple1: NOR cablé

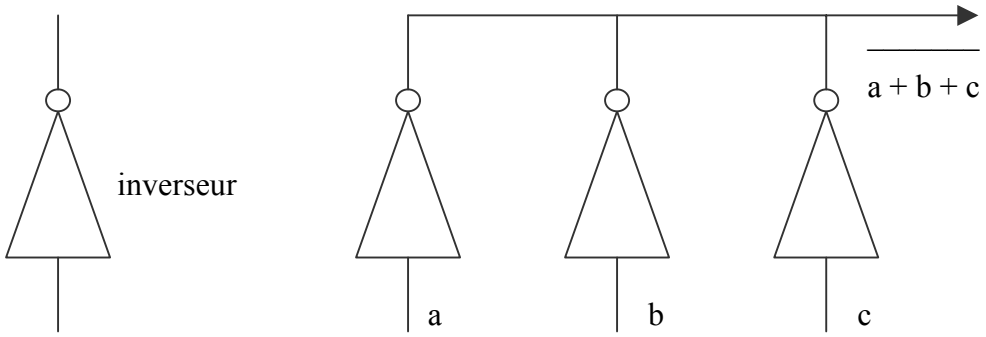
Un NOR câblé fournira ´0´ chaque fois qu´une de ses sources est à ´1´. La fonction de résolution est écrite avec en entrée un tableau sans dimension de bits (les sources) et en sortie le bit résultat

```
FUNCTION resolution_nor ( b : BIT_VECTOR) RETURN BIT IS
-- Fonction NOR cablée
        VARIABLE niveau : BIT := '1' ;-- valeur par défaut
BEGIN
        FOR i IN b'RANGE LOOP
                IF b(i) = '1' THEN
                        niveau := '0';
                END IF;
        END LOOP;
        RETURN niveau;
END;
```

La constitution du NOR câblé se fera par l´intermédiaire de cette fonction comme dans l´exemple ci-dessous:

```
SIGNAL a, b ,c : BIT; -- trois sources
SIGNAL s : resolution_nor BIT; -- s est déclaré comme signal résolu
BEGIN
        s <= a;
        s <= b;
        s <= c;
END;
```

Si maintenant a, b et c valent ´0´ alors s vaut ´1´



3.8.2 Signal gardé

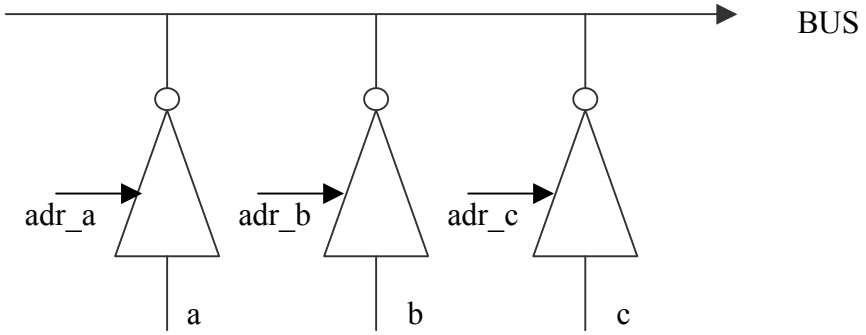
Pour traiter efficacement le problème des bus (avec état haute impédance Z) il existe des signaux gardés définis comme BUS ou REGISTER.

Une affectation **gardée** est celle qui sera faite dans un bloc avec condition de garde (GUARDED). Si la condition de garde est TRUE , alors il ya affectation normale du signal. Dans le cas contraire (FALSE), alors le signal est déconnecté.

Le délai de **déconnexion** est défini par défaut après 0 ns. L'instruction DISCONNECT permet de fixer un délai autre. Le REGISTER mémorise la dernière valeur lorsque tous les pilotes sont déconnectés alors que le bus réévalue la sortie.

3.8.3 Exemple2: Bus trois-états

Un exemple simple est la connexion à un bus s de deux portes d'adresse *adr_a* et *adr_b* ,avec comme entrée respective *a* et *b.*




```
SIGNAL a, b, adr_a,adr_b : std_ulogic    --package std_logic_1164
SIGNAL s : std_logic; --('U',´X´,´0´,´1´,´Z,'W', 'L', 'H', '-')
-- std_logic est un std_ulogic avec fonction de résolution

trois_etats_a : BLOCK (adr_a = ´1´)
BEGIN
        s <= GUARDED a ;
```

```
END BLOCK;

trois_etats_b : BLOCK (adr_b = ´1´)
BEGIN
        s <= GUARDED b ;
END BLOCK;
```

On aurait pu aussi écrire un seul processus

```
la_meme_chose : PROCESS
BEGIN
WAIT ON a, b, adr_a, adr_b;
        IF adr_a  = ´1´  THEN
                s <=  a ;
        ELSIF adr_b = ´1´ THEN
                s <=  b ;
        ELSE
                s <= NULL; -- affectation qui déconnecte un bus
-- ou bien          s <= 'Z'; qui est l'état du bus déconnecté
        END IF;
END PROCESS;
```

3.9 Les Bibliothèques

3.9.1 Paquetages

Le paquetage (PACKAGE) est une unité de conception permettant de regrouper des définitions de types, de constantes ou des déclarations de sous- programmes.

Le corps de paquetage (PACKAGE BODY), optionnel sert à instancier des constantes ou à décrire les sous-programmes.

Les bibliothèques de ressources (LIBRARY) regroupent les paquetages par famille. La clause USE <Library.package.nom> rend visible nom.

3.9.1.1 Exemple

```
PACKAGE utils  IS
  PROCEDURE  horloge (  SIGNAL h: OUT Bit; th, tb :  TIME);
  FUNCTION convert (n,l: NATURAL) RETURN BIT_VECTOR;
  FUNCTION convert (b: BIT_VECTOR) RETURN NATURAL;
  PROCEDURE verif_precond (SIGNAL h, d : IN BIT; Tsetup : IN TIME);
  FUNCTION resolution_nor ( b : BIT_VECTOR) RETURN BIT;
  FUNCTION front_montant  (SIGNAL s : bit) RETURN BOOLEAN;
  FUNCTION front_descendant (SIGNAL s :BIT) RETURN BOOLEAN;
END utils;
```

3.9.2 LIBRARY WORK

C'est la bibliothèque de travail par défaut. Y seront rangés les objets compilés relatifs au projet courant: ENTITY, ARCHITECTURE, PACKAGE, CONFIGURATION

3.9.3 LIBRARY STD

Bibliothèque standard 1987

3.9.3.1 Package standard

```
-- This is Package STANDARD as defined in the VHDL 1992 Language Reference Manual.
--
--   NOTE: VCOM and VSIM will not work properly if these declarations
--         are modified.

-- Version information: @(#)standard.vhd

package standard is
                        type boolean is (false,true);
                        type bit is ('0', '1');
                        type character is (
                                nul, soh, stx, etx, eot, enq, ack, bel,
                                bs,  ht,  lf,  vt,  ff,  cr,  so,  si,
                                dle, dc1, dc2, dc3, dc4, nak, syn, etb,
                                can, em,  sub, esc, fsp, gsp, rsp, usp,
                                ' ', '!', '"', '#', '$', '%', '&', ''',
                                '(', ')', '*', '+', ',', '-', '.', '/',
                                '0', '1', '2', '3', '4', '5', '6', '7',
                                '8', '9', ':', ';', '<', '=', '>', '?',
                                '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
                                'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
                                'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
                                'X', 'Y', 'Z', '[', '\', ']', '^', '_',
                                '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
                                'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
                                'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
                                'x', 'y', 'z', '{', '|', '}', '~', del,

                        type severity_level is (note, warning, error, failure);
                        type integer is range -2147483648 to 2147483647;
                        type real is range -1.0E308 to 1.0E308;
                        type time is range -2147483647 to 2147483647
                                units
                                        fs;
                                        ps = 1000 fs;
                                        ns = 1000 ps;
                                        us = 1000 ns;
                                        ms = 1000 us;
                                        sec = 1000 ms;
                                        min = 60 sec;
```

```
                                hr = 60 min;
                        end units;
                subtype delay_length is time range 0 fs to time'high;
                impure function now return delay_length;
                subtype natural is integer range 0 to integer'high;
                subtype positive is integer range 1 to integer'high;
                type string is array (positive range <>) of character;
                type bit_vector is array (natural range <>) of bit;
                type file_open_kind is (
                        read_mode,
                        write_mode,
                        append_mode);
                type file_open_status is (
                        open_ok,
                        status_error,
                        name_error,
                        mode_error);
                attribute foreign : string;
end standard;
```

3.9.3.2 Package textio

```
------------------------------------------------------------------------
-- Package TEXTIO as defined in Chapter 14 of the IEEE Standard VHDL
--   Language Reference Manual (IEEE Std. 1076-1987), as modified
--   by the Issues Screening and Analysis Committee (ISAC), a subcommittee
--   of the VHDL Analysis and Standardization Group (VASG) on
--   10 November, 1988.  See "The Sense of the VASG", October, 1989.
------------------------------------------------------------------------
-- Version information: %W% %G%
------------------------------------------------------------------------

package TEXTIO is
   type LINE is access string;
   type TEXT is file of string;
   type SIDE is (right, left);
   subtype WIDTH is natural;

                        -- changed for vhdl92 syntax:
   file input : TEXT open read_mode is "STD_INPUT";
   file output : TEXT open write_mode is "STD_OUTPUT";

                        -- changed for vhdl92 syntax (and now a built-in):
   procedure READLINE(file f: TEXT; L: out LINE);

   procedure READ(L:inout LINE; VALUE: out bit; GOOD : out BOOLEAN);
   procedure READ(L:inout LINE; VALUE: out bit);

   procedure READ(L:inout LINE; VALUE: out bit_vector; GOOD : out BOOLEAN);
```

```
    procedure READ(L:inout LINE; VALUE: out bit_vector);

    procedure READ(L:inout LINE; VALUE: out BOOLEAN; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out BOOLEAN);

    procedure READ(L:inout LINE; VALUE: out character; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out character);

    procedure READ(L:inout LINE; VALUE: out integer; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out integer);

    procedure READ(L:inout LINE; VALUE: out real; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out real);

    procedure READ(L:inout LINE; VALUE: out string; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out string);

    procedure READ(L:inout LINE; VALUE: out time; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out time);

                                -- changed for vhdl92 syntax (and now a built-in):
    procedure WRITELINE(file f : TEXT; L : inout LINE);

    procedure WRITE(L : inout LINE; VALUE : in bit;
                                    JUSTIFIED: in SIDE := right;
                                    FIELD: in WIDTH := 0);

    procedure WRITE(L : inout LINE; VALUE : in bit_vector;
                                    JUSTIFIED: in SIDE := right;
                                    FIELD: in WIDTH := 0);

    procedure WRITE(L : inout LINE; VALUE : in BOOLEAN;
                                    JUSTIFIED: in SIDE := right;
                                    FIELD: in WIDTH := 0);

    procedure WRITE(L : inout LINE; VALUE : in character;
                                    JUSTIFIED: in SIDE := right;
                                    FIELD: in WIDTH := 0);

    procedure WRITE(L : inout LINE; VALUE : in integer;
                                    JUSTIFIED: in SIDE := right;
                                    FIELD: in WIDTH := 0);

    procedure WRITE(L : inout LINE; VALUE : in real;
                                    JUSTIFIED: in SIDE := right;
                                    FIELD: in WIDTH := 0;
                                    DIGITS: in NATURAL := 0);

    procedure WRITE(L : inout LINE; VALUE : in string;
                                    JUSTIFIED: in SIDE := right;
```

```
                                FIELD: in WIDTH := 0);

    procedure WRITE(L : inout LINE; VALUE : in time;
                                JUSTIFIED: in SIDE := right;
                                FIELD: in WIDTH := 0;
                                UNIT: in TIME := ns);

                            -- is implicit built-in:
                            -- function ENDFILE(file F : TEXT) return boolean;

    -- function ENDLINE(variable L : in LINE) return BOOLEAN;
    --
    -- Function ENDLINE as declared cannot be legal VHDL, and
    --   the entire function was deleted from the definition
    --   by the Issues Screening and Analysis Committee (ISAC),
    --   a subcommittee of the VHDL Analysis and Standardization
    --   Group (VASG) on 10 November, 1988.  See "The Sense of
    --   the VASG", October, 1989, VHDL Issue Number 0032.
end;
```

3.9.4 LIBRARY IEEE

Bibliothèque standard 1993 encore appelée MVL9 qui contient les types évolués à 9 états et un ensemble de fonctions de conversion, de résolution et d'opérations en surcharge. Parmi les nombreux packages disponibles, celui qui semble le plus stable et le plus universel est numeric_std. L'utiliser garantie une plus grande portabilité.

3.9.4.1 Package std_logic_1164

```
-- ---------------------------------------------------------------------
--
-- Title     :  std_logic_1164 multi-value logic system
-- Library   :  This package shall be compiled into a library
--           :  symbolically named IEEE.
--           :
-- Developers:  IEEE model standards group (par 1164)
-- Purpose   :  This packages defines a standard for designers
--           :  to use in describing the interconnection data types
--           :  used in vhdl modeling.
--           :
-- Limitation:  The logic system defined in this package may
--           :  be insufficient for modeling switched transistors,
--           :  since such a requirement is out of the scope of this
--           :  effort. Furthermore, mathematics, primitives,
--           :  timing standards, etc. are considered orthogonal
--           :  issues as it relates to this package and are therefore
--           :  beyond the scope of this effort.
--           :
-- Note      :  No declarations or definitions shall be included in,
--           :  or excluded from this package. The "package declaration"
```

```
--            : defines the types, subtypes and declarations of
--            : std_logic_1164. The std_logic_1164 package body shall be
--            : considered the formal definition of the semantics of
--            : this package. Tool developers may choose to implement
--            : the package body in the most efficient manner available
--            : to them.
--            :
-- ---------------------------------------------------------------------
--   modification history :
-- ---------------------------------------------------------------------
--  version | mod. date:|
--   v4.200 | 01/02/92  |
-- ---------------------------------------------------------------------

PACKAGE std_logic_1164 IS
   ---------------------------------------------------------------------
   -- logic state system  (unresolved)
   ---------------------------------------------------------------------
   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
                      );
   ---------------------------------------------------------------------
   -- unconstrained array of std_ulogic for use with the resolution function
   ---------------------------------------------------------------------
   TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_ulogic;
   ---------------------------------------------------------------------
   -- resolution function
   ---------------------------------------------------------------------
   FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;

   ---------------------------------------------------------------------
   -- *** industry standard logic type ***
   ---------------------------------------------------------------------
   SUBTYPE std_logic IS resolved std_ulogic;
   ---------------------------------------------------------------------
   -- unconstrained array of std_logic for use in declaring signal arrays
   ---------------------------------------------------------------------
   TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;
   ---------------------------------------------------------------------
   -- common subtypes
   ---------------------------------------------------------------------
   SUBTYPE X01     IS resolved std_ulogic RANGE 'X' TO '1'; -- ('X','0','1')
   SUBTYPE X01Z    IS resolved std_ulogic RANGE 'X' TO 'Z'; -- ('X','0','1','Z')
```

```
SUBTYPE UX01    IS resolved std_ulogic RANGE 'U' TO '1'; -- ('U','X','0','1')
SUBTYPE UX01Z   IS resolved std_ulogic RANGE 'U' TO 'Z'; -- ('U','X','0','1','Z')
--------------------------------------------------------------------
-- overloaded logical operators
--------------------------------------------------------------------
FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
function "xnor" ( l : std_ulogic; r : std_ulogic ) return ux01;
FUNCTION "not"  ( l : std_ulogic                 ) RETURN UX01;
--------------------------------------------------------------------
-- vectorized overloaded logical operators
--------------------------------------------------------------------
FUNCTION "and"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "and"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
FUNCTION "nand" ( l, r : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "nand" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
FUNCTION "or"   ( l, r : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "or"   ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
FUNCTION "nor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "nor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
FUNCTION "xor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "xor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

function "xnor" ( l, r : std_logic_vector  ) return std_logic_vector;
function "xnor" ( l, r : std_ulogic_vector ) return std_ulogic_vector;
FUNCTION "not"  ( l : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "not"  ( l : std_ulogic_vector ) RETURN std_ulogic_vector;
--------------------------------------------------------------------
-- conversion functions
--------------------------------------------------------------------
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;
FUNCTION To_StdULogicVector ( s : std_logic_vector  ) RETURN std_ulogic_vector;
   --------------------------------------------------------------------
-- strength strippers and type convertors
--------------------------------------------------------------------
FUNCTION To_X01  ( s : std_logic_vector  ) RETURN  std_logic_vector;
FUNCTION To_X01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
FUNCTION To_X01  ( s : std_ulogic        ) RETURN  X01;
FUNCTION To_X01  ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
FUNCTION To_X01  ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
FUNCTION To_X01  ( b : BIT               ) RETURN  X01;
```

```
  FUNCTION To_X01Z ( s : std_logic_vector  ) RETURN  std_logic_vector;
  FUNCTION To_X01Z ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
  FUNCTION To_X01Z ( s : std_ulogic        ) RETURN  X01Z;
  FUNCTION To_X01Z ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
  FUNCTION To_X01Z ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
  FUNCTION To_X01Z ( b : BIT               ) RETURN  X01Z;
  FUNCTION To_UX01  ( s : std_logic_vector  ) RETURN  std_logic_vector;
  FUNCTION To_UX01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
  FUNCTION To_UX01  ( s : std_ulogic        ) RETURN  UX01;
  FUNCTION To_UX01  ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
  FUNCTION To_UX01  ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
  FUNCTION To_UX01  ( b : BIT               ) RETURN  UX01;
  ------------------------------------------------------------------
  -- edge detection
  ------------------------------------------------------------------
  FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN;
  FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN;
  ------------------------------------------------------------------
  -- object contains an unknown
  ------------------------------------------------------------------
  FUNCTION Is_X ( s : std_ulogic_vector ) RETURN  BOOLEAN;
  FUNCTION Is_X ( s : std_logic_vector  ) RETURN  BOOLEAN;
  FUNCTION Is_X ( s : std_ulogic        ) RETURN  BOOLEAN;
END std_logic_1164;
```

3.9.4.2 Package numeric_std

```
-- ------------------------------------------------------------------
--
-- Copyright 1995 by IEEE. All rights reserved.
--
-- This source file is considered by the IEEE to be an essential part of the use
-- of the standard 1076.3 and as such may be distributed without change, except
-- as permitted by the standard. This source file may not be sold or distributed
-- for profit. This package may be modified to include additional data required
-- by tools, but must in no way change the external interfaces or simulation
-- behaviour of the description. It is permissible to add comments and/or
-- attributes to the package declarations, but not to change or delete any
-- original lines of the approved package declaration. The package body may be
-- changed only in accordance with the terms of clauses 7.1 and 7.2 of the
-- standard.
--
-- Title      : Standard VHDL Synthesis Package (1076.3, NUMERIC_STD)
--
-- Library    : This package shall be compiled into a library symbolically
--            : named IEEE.
--
-- Developers : IEEE DASC Synthesis Working Group, PAR 1076.3
--
-- Purpose    : This package defines numeric types and arithmetic functions
```

```
--          : for use with synthesis tools. Two numeric types are defined:
--          : -- > UNSIGNED: represents UNSIGNED number in vector form
--          : -- > SIGNED: represents a SIGNED number in vector form
--          : The base element type is type STD_LOGIC.
--          : The leftmost bit is treated as the most significant bit.
--          : Signed vectors are represented in two's complement form.
--          : This package contains overloaded arithmetic operators on
--          : the SIGNED and UNSIGNED types. The package also contains
--          : useful type conversions functions.
--          :
--          : If any argument to a function is a null array, a null array is
--          : returned (exceptions, if any, are noted individually).
--
-- Limitation :
--
-- Note       : No declarations or definitions shall be included in,
--          : or excluded from this package. The "package declaration"
--          : defines the types, subtypes and declarations of
--          : NUMERIC_STD. The NUMERIC_STD package body shall be
--          : considered the formal definition of the semantics of
--          : this package. Tool developers may choose to implement
--          : the package body in the most efficient manner available
--          : to them.
--
-- -------------------------------------------------------------------
--   modification history :
-- -------------------------------------------------------------------
--   Version:  2.4
--   Date   :  12 April 1995
-- -------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package NUMERIC_STD is
  constant CopyRightNotice: STRING
     := "Copyright 1995 IEEE. All rights reserved.";

  attribute builtin_subprogram: string;

  --
============================================================================
=
  -- Numeric array type definitions
  --
============================================================================
=

  type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
  type SIGNED is array (NATURAL range <>) of STD_LOGIC;
```

```
  --
================================================================================
=
  -- Arithmetic Operators:
  --
================================================================================

  -- Id: A.1
  function "abs" (ARG: SIGNED) return SIGNED;
   attribute builtin_subprogram of
      "ABS"[SIGNED return SIGNED]: function is "numstd_abs_ss";
  -- Result subtype: SIGNED(ARG'LENGTH-1 downto 0).
  -- Result: Returns the absolute value of a SIGNED vector ARG.

  -- Id: A.2
  function "-" (ARG: SIGNED) return SIGNED;
   attribute builtin_subprogram of
      "-"[SIGNED return SIGNED]: function is "numstd_unary_minus_ss";
  -- Result subtype: SIGNED(ARG'LENGTH-1 downto 0).
  -- Result: Returns the value of the unary minus operation on a
  --         SIGNED vector ARG.

  --
================================================================================
=

  -- Id: A.3
  function "+" (L, R: UNSIGNED) return UNSIGNED;
   attribute builtin_subprogram of

      "+"[UNSIGNED, UNSIGNED return UNSIGNED]: function is "numstd_plus_uuu";
  -- Result subtype: UNSIGNED(MAX(L'LENGTH, R'LENGTH)-1 downto 0).
  -- Result: Adds two UNSIGNED vectors that may be of different lengths.

  -- Id: A.4
  function "+" (L, R: SIGNED) return SIGNED;
   attribute builtin_subprogram of
      "+"[SIGNED, SIGNED return SIGNED]: function is "numstd_plus_sss";
  -- Result subtype: SIGNED(MAX(L'LENGTH, R'LENGTH)-1 downto 0).
  -- Result: Adds two SIGNED vectors that may be of different lengths.

  -- Id: A.5
  function "+" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
   attribute builtin_subprogram of
      "+"[UNSIGNED, NATURAL return UNSIGNED]: function is "numstd_plus_unu";
  -- Result subtype: UNSIGNED(L'LENGTH-1 downto 0).
  -- Result: Adds an UNSIGNED vector, L, with a non-negative INTEGER, R.

  -- Id: A.6
  function "+" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
```

```
  attribute builtin_subprogram of
     "+"[NATURAL, UNSIGNED return UNSIGNED]: function is "numstd_plus_nuu";
  -- Result subtype: UNSIGNED(R'LENGTH-1 downto 0).
  -- Result: Adds a non-negative INTEGER, L, with an UNSIGNED vector, R.

  -- Id: A.7
  function "+" (L: INTEGER; R: SIGNED) return SIGNED;
   attribute builtin_subprogram of
     "+"[INTEGER, SIGNED return SIGNED]: function is "numstd_plus_iss";
  -- Result subtype: SIGNED(R'LENGTH-1 downto 0).
  -- Result: Adds an INTEGER, L(may be positive or negative), to a SIGNED
  --         vector, R.

  -- Id: A.8
  function "+" (L: SIGNED; R: INTEGER) return SIGNED;
   attribute builtin_subprogram of
     "+"[SIGNED, INTEGER return SIGNED]: function is "numstd_plus_sis";
  -- Result subtype: SIGNED(L'LENGTH-1 downto 0).
  -- Result: Adds a SIGNED vector, L, to an INTEGER, R.

  --
==============================================================================
=

  -- Id: A.9
  function "-" (L, R: UNSIGNED) return UNSIGNED;
   attribute builtin_subprogram of
     "-"[UNSIGNED, UNSIGNED return UNSIGNED]: function is "numstd_minus_uuu";
  -- Result subtype: UNSIGNED(MAX(L'LENGTH, R'LENGTH)-1 downto 0).
  -- Result: Subtracts two UNSIGNED vectors that may be of different lengths.

  -- Id: A.10
  function "-" (L, R: SIGNED) return SIGNED;
   attribute builtin_subprogram of
     "-"[SIGNED, SIGNED return SIGNED]: function is "numstd_minus_sss";
  -- Result subtype: SIGNED(MAX(L'LENGTH, R'LENGTH)-1 downto 0).
  -- Result: Subtracts a SIGNED vector, R, from another SIGNED vector, L,
  --         that may possibly be of different lengths.

  -- Id: A.11
  function "-" (L: UNSIGNED;R: NATURAL) return UNSIGNED;
   attribute builtin_subprogram of
     "-"[UNSIGNED, NATURAL return UNSIGNED]: function is "numstd_minus_unu";
  -- Result subtype: UNSIGNED(L'LENGTH-1 downto 0).
  -- Result: Subtracts a non-negative INTEGER, R, from an UNSIGNED vector, L.

  -- Id: A.12
  function "-" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
   attribute builtin_subprogram of
     "-"[NATURAL, UNSIGNED return UNSIGNED]: function is "numstd_minus_nuu";
```

```
  -- Result subtype: UNSIGNED(R'LENGTH-1 downto 0).
  -- Result: Subtracts an UNSIGNED vector, R, from a non-negative INTEGER, L.

  -- Id: A.13
  function "-" (L: SIGNED; R: INTEGER) return SIGNED;
   attribute builtin_subprogram of
      "-"[SIGNED, INTEGER return SIGNED]: function is "numstd_minus_sis";
  -- Result subtype: SIGNED(L'LENGTH-1 downto 0).
  -- Result: Subtracts an INTEGER, R, from a SIGNED vector, L.

  -- Id: A.14
  function "-" (L: INTEGER; R: SIGNED) return SIGNED;
   attribute builtin_subprogram of
      "-"[INTEGER, SIGNED return SIGNED]: function is "numstd_minus_iss";
  -- Result subtype: SIGNED(R'LENGTH-1 downto 0).
  -- Result: Subtracts a SIGNED vector, R, from an INTEGER, L.

  --
==============================================================================
=

  -- Id: A.15
  function "*" (L, R: UNSIGNED) return UNSIGNED;
   attribute builtin_subprogram of
      "*"[UNSIGNED, UNSIGNED return UNSIGNED]: function is "numstd_mult_uuu";
  -- Result subtype: UNSIGNED((L'LENGTH+R'LENGTH-1) downto 0).
  -- Result: Performs the multiplication operation on two UNSIGNED vectors
  --         that may possibly be of different lengths.

  -- Id: A.16
  function "*" (L, R: SIGNED) return SIGNED;
   attribute builtin_subprogram of
      "*"[SIGNED, SIGNED return SIGNED]: function is "numstd_mult_sss";
  -- Result subtype: SIGNED((L'LENGTH+R'LENGTH-1) downto 0)
  -- Result: Multiplies two SIGNED vectors that may possibly be of
  --         different lengths.

  -- Id: A.17
  function "*" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
   attribute builtin_subprogram of
      "*"[UNSIGNED, NATURAL return UNSIGNED]: function is "numstd_mult_unu";
  -- Result subtype: UNSIGNED((L'LENGTH+L'LENGTH-1) downto 0).
  -- Result: Multiplies an UNSIGNED vector, L, with a non-negative
  --         INTEGER, R. R is converted to an UNSIGNED vector of
  --         SIZE L'LENGTH before multiplication.

  -- Id: A.18
  function "*" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
   attribute builtin_subprogram of
      "*"[NATURAL, UNSIGNED return UNSIGNED]: function is "numstd_mult_nuu";
```

```
  -- Result subtype: UNSIGNED((R'LENGTH+R'LENGTH-1) downto 0).
  -- Result: Multiplies an UNSIGNED vector, R, with a non-negative
  --         INTEGER, L. L is converted to an UNSIGNED vector of
  --         SIZE R'LENGTH before multiplication.

  -- Id: A.19
  function "*" (L: SIGNED; R: INTEGER) return SIGNED;
   attribute builtin_subprogram of
      "*"[SIGNED, INTEGER return SIGNED]: function is "numstd_mult_sis";
  -- Result subtype: SIGNED((L'LENGTH+L'LENGTH-1) downto 0)
  -- Result: Multiplies a SIGNED vector, L, with an INTEGER, R. R is
  --         converted to a SIGNED vector of SIZE L'LENGTH before
  --         multiplication.

  -- Id: A.20
  function "*" (L: INTEGER; R: SIGNED) return SIGNED;
   attribute builtin_subprogram of
      "*"[INTEGER, SIGNED return SIGNED]: function is "numstd_mult_iss";
  -- Result subtype: SIGNED((R'LENGTH+R'LENGTH-1) downto 0)
  -- Result: Multiplies a SIGNED vector, R, with an INTEGER, L. L is
  --         converted to a SIGNED vector of SIZE R'LENGTH before
  --         multiplication.

  --
==============================================================================
=
  --
  -- NOTE: If second argument is zero for "/" operator, a severity level
  --       of ERROR is issued.

  -- Id: A.21
  function "/" (L, R: UNSIGNED) return UNSIGNED;
   attribute builtin_subprogram of

      "/"[UNSIGNED, UNSIGNED return UNSIGNED]: function is "numstd_div_uuu";
  -- Result subtype: UNSIGNED(L'LENGTH-1 downto 0)
  -- Result: Divides an UNSIGNED vector, L, by another UNSIGNED vector, R.

  -- Id: A.22
  function "/" (L, R: SIGNED) return SIGNED;
   attribute builtin_subprogram of
      "/"[SIGNED, SIGNED return SIGNED]: function is "numstd_div_sss";
  -- Result subtype: SIGNED(L'LENGTH-1 downto 0)
  -- Result: Divides an SIGNED vector, L, by another SIGNED vector, R.

  -- Id: A.23
  function "/" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
   attribute builtin_subprogram of
      "/"[UNSIGNED, NATURAL return UNSIGNED]: function is "numstd_div_unu";
  -- Result subtype: UNSIGNED(L'LENGTH-1 downto 0)
```

```
  -- Result: Divides an UNSIGNED vector, L, by a non-negative INTEGER, R.
  --         If NO_OF_BITS(R) > L'LENGTH, result is truncated to L'LENGTH.

  -- Id: A.24
  function "/" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
   attribute builtin_subprogram of
      "/"[NATURAL, UNSIGNED return UNSIGNED]: function is "numstd_div_nuu";
  -- Result subtype: UNSIGNED(R'LENGTH-1 downto 0)
  -- Result: Divides a non-negative INTEGER, L, by an UNSIGNED vector, R.
  --         If NO_OF_BITS(L) > R'LENGTH, result is truncated to R'LENGTH.

  -- Id: A.25
  function "/" (L: SIGNED; R: INTEGER) return SIGNED;
   attribute builtin_subprogram of
      "/"[SIGNED, INTEGER return SIGNED]: function is "numstd_div_sis";
  -- Result subtype: SIGNED(L'LENGTH-1 downto 0)
  -- Result: Divides a SIGNED vector, L, by an INTEGER, R.
  --         If NO_OF_BITS(R) > L'LENGTH, result is truncated to L'LENGTH.

  -- Id: A.26
  function "/" (L: INTEGER; R: SIGNED) return SIGNED;
   attribute builtin_subprogram of
      "/"[INTEGER, SIGNED return SIGNED]: function is "numstd_div_iss";
  -- Result subtype: SIGNED(R'LENGTH-1 downto 0)
  -- Result: Divides an INTEGER, L, by a SIGNED vector, R.
  --         If NO_OF_BITS(L) > R'LENGTH, result is truncated to R'LENGTH.

  --
================================================================================
=
  --
  -- NOTE: If second argument is zero for "rem" operator, a severity level
  --       of ERROR is issued.

  -- Id: A.27
  function "rem" (L, R: UNSIGNED) return UNSIGNED;
   attribute builtin_subprogram of
      "REM"[UNSIGNED, UNSIGNED return UNSIGNED]: function is "numstd_rem_uuu";
  -- Result subtype: UNSIGNED(R'LENGTH-1 downto 0)
  -- Result: Computes "L rem R" where L and R are UNSIGNED vectors.

  -- Id: A.28
  function "rem" (L, R: SIGNED) return SIGNED;
   attribute builtin_subprogram of
      "REM"[SIGNED, SIGNED return SIGNED]: function is "numstd_rem_sss";
  -- Result subtype: SIGNED(R'LENGTH-1 downto 0)
  -- Result: Computes "L rem R" where L and R are SIGNED vectors.

  -- Id: A.29
  function "rem" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
```

```
    attribute builtin_subprogram of
        "REM"[UNSIGNED, NATURAL return UNSIGNED]: function is "numstd_rem_unu";
  -- Result subtype: UNSIGNED(L'LENGTH-1 downto 0)
  -- Result: Computes "L rem R" where L is an UNSIGNED vector and R is a
  --         non-negative INTEGER.
  --         If NO_OF_BITS(R) > L'LENGTH, result is truncated to L'LENGTH.

  -- Id: A.30
  function "rem" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
    attribute builtin_subprogram of
        "REM"[NATURAL, UNSIGNED return UNSIGNED]: function is "numstd_rem_nuu";
  -- Result subtype: UNSIGNED(R'LENGTH-1 downto 0)
  -- Result: Computes "L rem R" where R is an UNSIGNED vector and L is a
  --         non-negative INTEGER.
  --         If NO_OF_BITS(L) > R'LENGTH, result is truncated to R'LENGTH.

  -- Id: A.31
  function "rem" (L: SIGNED; R: INTEGER) return SIGNED;
    attribute builtin_subprogram of
        "REM"[SIGNED, INTEGER return SIGNED]: function is "numstd_rem_sis";
  -- Result subtype: SIGNED(L'LENGTH-1 downto 0)
  -- Result: Computes "L rem R" where L is SIGNED vector and R is an INTEGER.
  --         If NO_OF_BITS(R) > L'LENGTH, result is truncated to L'LENGTH.

  -- Id: A.32
  function "rem" (L: INTEGER; R: SIGNED) return SIGNED;
    attribute builtin_subprogram of
        "REM"[INTEGER, SIGNED return SIGNED]: function is "numstd_rem_iss";
  -- Result subtype: SIGNED(R'LENGTH-1 downto 0)
  -- Result: Computes "L rem R" where R is SIGNED vector and L is an INTEGER.
  --         If NO_OF_BITS(L) > R'LENGTH, result is truncated to R'LENGTH.

  --
================================================================================
=
  --
  -- NOTE: If second argument is zero for "mod" operator, a severity level
  --       of ERROR is issued.

  -- Id: A.33
  function "mod" (L, R: UNSIGNED) return UNSIGNED;
    attribute builtin_subprogram of
        "MOD"[UNSIGNED, UNSIGNED return UNSIGNED]: function is "numstd_mod_uuu";
  -- Result subtype: UNSIGNED(R'LENGTH-1 downto 0)
  -- Result: Computes "L mod R" where L and R are UNSIGNED vectors.

  -- Id: A.34
  function "mod" (L, R: SIGNED) return SIGNED;
    attribute builtin_subprogram of
        "MOD"[SIGNED, SIGNED return SIGNED]: function is "numstd_mod_sss";
```

```
  -- Result subtype: SIGNED(R'LENGTH-1 downto 0)
  -- Result: Computes "L mod R" where L and R are SIGNED vectors.

  -- Id: A.35
  function "mod" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
   attribute builtin_subprogram of
      "MOD"[UNSIGNED, NATURAL return UNSIGNED]: function is "numstd_mod_unu";
  -- Result subtype: UNSIGNED(L'LENGTH-1 downto 0)
  -- Result: Computes "L mod R" where L is an UNSIGNED vector and R
  --         is a non-negative INTEGER.
  --         If NO_OF_BITS(R) > L'LENGTH, result is truncated to L'LENGTH.

  -- Id: A.36
  function "mod" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
   attribute builtin_subprogram of
      "MOD"[NATURAL, UNSIGNED return UNSIGNED]: function is "numstd_mod_nuu";
  -- Result subtype: UNSIGNED(R'LENGTH-1 downto 0)
  -- Result: Computes "L mod R" where R is an UNSIGNED vector and L
  --         is a non-negative INTEGER.
  --         If NO_OF_BITS(L) > R'LENGTH, result is truncated to R'LENGTH.

  -- Id: A.37
  function "mod" (L: SIGNED; R: INTEGER) return SIGNED;
   attribute builtin_subprogram of
      "MOD"[SIGNED, INTEGER return SIGNED]: function is "numstd_mod_sis";
  -- Result subtype: SIGNED(L'LENGTH-1 downto 0)
  -- Result: Computes "L mod R" where L is a SIGNED vector and
  --         R is an INTEGER.
  --         If NO_OF_BITS(R) > L'LENGTH, result is truncated to L'LENGTH.

  -- Id: A.38
  function "mod" (L: INTEGER; R: SIGNED) return SIGNED;
   attribute builtin_subprogram of
      "MOD"[INTEGER, SIGNED return SIGNED]: function is "numstd_mod_iss";
  -- Result subtype: SIGNED(R'LENGTH-1 downto 0)
  -- Result: Computes "L mod R" where L is an INTEGER and
  --         R is a SIGNED vector.
  --         If NO_OF_BITS(L) > R'LENGTH, result is truncated to R'LENGTH.

  --
================================================================================
=
  -- Comparison Operators
  --
================================================================================
=

  -- Id: C.1
  function ">" (L, R: UNSIGNED) return BOOLEAN;
   attribute builtin_subprogram of
```

```
      ">"[UNSIGNED, UNSIGNED return BOOLEAN]: function is "numstd_gt_uu";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L > R" where L and R are UNSIGNED vectors possibly
  --         of different lengths.

  -- Id: C.2
  function ">" (L, R: SIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      ">"[SIGNED, SIGNED return BOOLEAN]: function is "numstd_gt_ss";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L > R" where L and R are SIGNED vectors possibly
  --         of different lengths.

  -- Id: C.3
  function ">" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      ">"[NATURAL, UNSIGNED return BOOLEAN]: function is "numstd_gt_nu";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L > R" where L is a non-negative INTEGER and
  --         R is an UNSIGNED vector.

  -- Id: C.4
  function ">" (L: INTEGER; R: SIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      ">"[INTEGER, SIGNED return BOOLEAN]: function is "numstd_gt_is";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L > R" where L is a INTEGER and
  --         R is a SIGNED vector.

  -- Id: C.5
  function ">" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
   attribute builtin_subprogram of
      ">"[UNSIGNED, NATURAL return BOOLEAN]: function is "numstd_gt_un";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L > R" where L is an UNSIGNED vector and
  --         R is a non-negative INTEGER.

  -- Id: C.6
  function ">" (L: SIGNED; R: INTEGER) return BOOLEAN;
   attribute builtin_subprogram of
      ">"[SIGNED, INTEGER return BOOLEAN]: function is "numstd_gt_si";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L > R" where L is a SIGNED vector and
  --         R is a INTEGER.

  --
================================================================================
=

  -- Id: C.7
```

```
  function "<" (L, R: UNSIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      "<"[UNSIGNED, UNSIGNED return BOOLEAN]: function is "numstd_lt_uu";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L < R" where L and R are UNSIGNED vectors possibly
  --         of different lengths.

  -- Id: C.8
  function "<" (L, R: SIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      "<"[SIGNED, SIGNED return BOOLEAN]: function is "numstd_lt_ss";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L < R" where L and R are SIGNED vectors possibly
  --         of different lengths.

  -- Id: C.9
  function "<" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      "<"[NATURAL, UNSIGNED return BOOLEAN]: function is "numstd_lt_nu";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L < R" where L is a non-negative INTEGER and
  --         R is an UNSIGNED vector.

  -- Id: C.10
  function "<" (L: INTEGER; R: SIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      "<"[INTEGER, SIGNED return BOOLEAN]: function is "numstd_lt_is";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L < R" where L is an INTEGER and
  --         R is a SIGNED vector.

  -- Id: C.11
  function "<" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
   attribute builtin_subprogram of
      "<"[UNSIGNED, NATURAL return BOOLEAN]: function is "numstd_lt_un";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L < R" where L is an UNSIGNED vector and
  --         R is a non-negative INTEGER.

  -- Id: C.12
  function "<" (L: SIGNED; R: INTEGER) return BOOLEAN;
   attribute builtin_subprogram of
      "<"[SIGNED, INTEGER return BOOLEAN]: function is "numstd_lt_si";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L < R" where L is a SIGNED vector and
  --         R is an INTEGER.

  --
================================================================================
=
```

```
-- Id: C.13
function "<=" (L, R: UNSIGNED) return BOOLEAN;
 attribute builtin_subprogram of
    "<="[UNSIGNED, UNSIGNED return BOOLEAN]: function is "numstd_lte_uu";
-- Result subtype: BOOLEAN
-- Result: Computes "L <= R" where L and R are UNSIGNED vectors possibly
--         of different lengths.

-- Id: C.14
function "<=" (L, R: SIGNED) return BOOLEAN;
 attribute builtin_subprogram of
    "<="[SIGNED, SIGNED return BOOLEAN]: function is "numstd_lte_ss";
-- Result subtype: BOOLEAN
-- Result: Computes "L <= R" where L and R are SIGNED vectors possibly
--         of different lengths.

-- Id: C.15
function "<=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
 attribute builtin_subprogram of
    "<="[NATURAL, UNSIGNED return BOOLEAN]: function is "numstd_lte_nu";
-- Result subtype: BOOLEAN
-- Result: Computes "L <= R" where L is a non-negative INTEGER and
--         R is an UNSIGNED vector.

-- Id: C.16
function "<=" (L: INTEGER; R: SIGNED) return BOOLEAN;
 attribute builtin_subprogram of
    "<="[INTEGER, SIGNED return BOOLEAN]: function is "numstd_lte_is";
-- Result subtype: BOOLEAN
-- Result: Computes "L <= R" where L is an INTEGER and
--         R is a SIGNED vector.

-- Id: C.17
function "<=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
 attribute builtin_subprogram of
    "<="[UNSIGNED, NATURAL return BOOLEAN]: function is "numstd_lte_un";
-- Result subtype: BOOLEAN
-- Result: Computes "L <= R" where L is an UNSIGNED vector and
--         R is a non-negative INTEGER.

-- Id: C.18
function "<=" (L: SIGNED; R: INTEGER) return BOOLEAN;
 attribute builtin_subprogram of
    "<="[SIGNED, INTEGER return BOOLEAN]: function is "numstd_lte_si";
-- Result subtype: BOOLEAN
-- Result: Computes "L <= R" where L is a SIGNED vector and
--         R is an INTEGER.
```

```
  --
============================================================================
=

  -- Id: C.19
  function ">=" (L, R: UNSIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      ">="[UNSIGNED, UNSIGNED return BOOLEAN]: function is "numstd_gte_uu";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L >= R" where L and R are UNSIGNED vectors possibly
  --         of different lengths.

  -- Id: C.20
  function ">=" (L, R: SIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      ">="[SIGNED, SIGNED return BOOLEAN]: function is "numstd_gte_ss";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L >= R" where L and R are SIGNED vectors possibly
  --         of different lengths.

  -- Id: C.21
  function ">=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      ">="[NATURAL, UNSIGNED return BOOLEAN]: function is "numstd_gte_nu";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L >= R" where L is a non-negative INTEGER and
  --         R is an UNSIGNED vector.

  -- Id: C.22
  function ">=" (L: INTEGER; R: SIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      ">="[INTEGER, SIGNED return BOOLEAN]: function is "numstd_gte_is";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L >= R" where L is an INTEGER and
  --         R is a SIGNED vector.

  -- Id: C.23
  function ">=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
   attribute builtin_subprogram of
      ">="[UNSIGNED, NATURAL return BOOLEAN]: function is "numstd_gte_un";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L >= R" where L is an UNSIGNED vector and
  --         R is a non-negative INTEGER.

  -- Id: C.24
  function ">=" (L: SIGNED; R: INTEGER) return BOOLEAN;
   attribute builtin_subprogram of
      ">="[SIGNED, INTEGER return BOOLEAN]: function is "numstd_gte_si";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L >= R" where L is a SIGNED vector and
```

```
  --        R is an INTEGER.

  --
==============================================================================
=

  -- Id: C.25
  function "=" (L, R: UNSIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      "="[UNSIGNED, UNSIGNED return BOOLEAN]: function is "numstd_eq_uu";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L = R" where L and R are UNSIGNED vectors possibly
  --        of different lengths.

  -- Id: C.26
  function "=" (L, R: SIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      "="[SIGNED, SIGNED return BOOLEAN]: function is "numstd_eq_ss";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L = R" where L and R are SIGNED vectors possibly
  --        of different lengths.

  -- Id: C.27
  function "=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      "="[NATURAL, UNSIGNED return BOOLEAN]: function is "numstd_eq_nu";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L = R" where L is a non-negative INTEGER and
  --        R is an UNSIGNED vector.

  -- Id: C.28
  function "=" (L: INTEGER; R: SIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      "="[INTEGER, SIGNED return BOOLEAN]: function is "numstd_eq_is";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L = R" where L is an INTEGER and
  --        R is a SIGNED vector.

  -- Id: C.29
  function "=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
   attribute builtin_subprogram of
      "="[UNSIGNED, NATURAL return BOOLEAN]: function is "numstd_eq_un";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L = R" where L is an UNSIGNED vector and
  --        R is a non-negative INTEGER.

  -- Id: C.30
  function "=" (L: SIGNED; R: INTEGER) return BOOLEAN;
   attribute builtin_subprogram of
      "="[SIGNED, INTEGER return BOOLEAN]: function is "numstd_eq_si";
```

```
  -- Result subtype: BOOLEAN
  -- Result: Computes "L = R" where L is a SIGNED vector and
  --         R is an INTEGER.

  --
================================================================================
=

  -- Id: C.31
  function "/=" (L, R: UNSIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      "/="[UNSIGNED, UNSIGNED return BOOLEAN]: function is "numstd_neq_uu";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L /= R" where L and R are UNSIGNED vectors possibly
  --         of different lengths.

  -- Id: C.32
  function "/=" (L, R: SIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      "/="[SIGNED, SIGNED return BOOLEAN]: function is "numstd_neq_ss";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L /= R" where L and R are SIGNED vectors possibly
  --         of different lengths.

  -- Id: C.33
  function "/=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      "/="[NATURAL, UNSIGNED return BOOLEAN]: function is "numstd_neq_nu";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L /= R" where L is a non-negative INTEGER and
  --         R is an UNSIGNED vector.

  -- Id: C.34
  function "/=" (L: INTEGER; R: SIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      "/="[INTEGER, SIGNED return BOOLEAN]: function is "numstd_neq_is";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L /= R" where L is an INTEGER and
  --         R is a SIGNED vector.

  -- Id: C.35
  function "/=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
   attribute builtin_subprogram of
      "/="[UNSIGNED, NATURAL return BOOLEAN]: function is "numstd_neq_un";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L /= R" where L is an UNSIGNED vector and
  --         R is a non-negative INTEGER.

  -- Id: C.36
  function "/=" (L: SIGNED; R: INTEGER) return BOOLEAN;
```

```
    attribute builtin_subprogram of
        "/="[SIGNED, INTEGER return BOOLEAN]: function is "numstd_neq_si";
  -- Result subtype: BOOLEAN
  -- Result: Computes "L /= R" where L is a SIGNED vector and
  --         R is an INTEGER.

  --
============================================================================
=
  -- Shift and Rotate Functions
  --
============================================================================
=

  -- Id: S.1
  function SHIFT_LEFT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
    attribute builtin_subprogram of
        SHIFT_LEFT[UNSIGNED, NATURAL return UNSIGNED]: function is "array_sll_1164";
  -- Result subtype: UNSIGNED(ARG'LENGTH-1 downto 0)
  -- Result: Performs a shift-left on an UNSIGNED vector COUNT times.
  --         The vacated positions are filled with '0'.
  --         The COUNT leftmost elements are lost.

  -- Id: S.2
  function SHIFT_RIGHT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
    attribute builtin_subprogram of
        SHIFT_RIGHT[UNSIGNED, NATURAL return UNSIGNED]: function is "array_srl_1164";
  -- Result subtype: UNSIGNED(ARG'LENGTH-1 downto 0)
  -- Result: Performs a shift-right on an UNSIGNED vector COUNT times.
  --         The vacated positions are filled with '0'.
  --         The COUNT rightmost elements are lost.

  -- Id: S.3
  function SHIFT_LEFT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;
    attribute builtin_subprogram of
        SHIFT_LEFT[SIGNED, NATURAL return SIGNED]: function is "array_sll_1164";
  -- Result subtype: SIGNED(ARG'LENGTH-1 downto 0)
  -- Result: Performs a shift-left on a SIGNED vector COUNT times.
  --         The vacated positions are filled with '0'.
  --         The COUNT leftmost elements are lost.

  -- Id: S.4
  function SHIFT_RIGHT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;
    attribute builtin_subprogram of
        SHIFT_RIGHT[SIGNED, NATURAL return SIGNED]: function is "array_sra";
  -- Result subtype: SIGNED(ARG'LENGTH-1 downto 0)
  -- Result: Performs a shift-right on a SIGNED vector COUNT times.
  --         The vacated positions are filled with the leftmost
  --         element, ARG'LEFT. The COUNT rightmost elements are lost.
```

```
  --
============================================================================
=

  -- Id: S.5
  function ROTATE_LEFT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
   attribute builtin_subprogram of
      ROTATE_LEFT[UNSIGNED, NATURAL return UNSIGNED]: function is "array_rol";
  -- Result subtype: UNSIGNED(ARG'LENGTH-1 downto 0)
  -- Result: Performs a rotate-left of an UNSIGNED vector COUNT times.

  -- Id: S.6
  function ROTATE_RIGHT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
   attribute builtin_subprogram of
      ROTATE_RIGHT[UNSIGNED, NATURAL return UNSIGNED]: function is "array_ror";
  -- Result subtype: UNSIGNED(ARG'LENGTH-1 downto 0)
  -- Result: Performs a rotate-right of an UNSIGNED vector COUNT times.

  -- Id: S.7
  function ROTATE_LEFT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;
   attribute builtin_subprogram of
      ROTATE_LEFT[SIGNED, NATURAL return SIGNED]: function is "array_rol";
  -- Result subtype: SIGNED(ARG'LENGTH-1 downto 0)
  -- Result: Performs a logical rotate-left of a SIGNED
  --         vector COUNT times.

  -- Id: S.8
  function ROTATE_RIGHT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;
   attribute builtin_subprogram of
      ROTATE_RIGHT[SIGNED, NATURAL return SIGNED]: function is "array_ror";
  -- Result subtype: SIGNED(ARG'LENGTH-1 downto 0)
  -- Result: Performs a logical rotate-right of a SIGNED
  --         vector COUNT times.

  --
============================================================================
=

  --
============================================================================
=

  ----------------------------------------------------------------------------
  --   Note : Function S.9 is not compatible with VHDL 1076-1987. Comment
  --   out the function (declaration and body) for VHDL 1076-1987 compatibility.
  ----------------------------------------------------------------------------
  -- Id: S.9
  function "sll" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;
   attribute builtin_subprogram of
      "SLL"[UNSIGNED, INTEGER return UNSIGNED]: function is "array_sll_1164";
```

```
-- Result subtype: UNSIGNED(ARG'LENGTH-1 downto 0)
-- Result: SHIFT_LEFT(ARG, COUNT)

--------------------------------------------------------------------------------
-- Note : Function S.10 is not compatible with VHDL 1076-1987. Comment
--   out the function (declaration and body) for VHDL 1076-1987 compatibility.
--------------------------------------------------------------------------------
-- Id: S.10
function "sll" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;
 attribute builtin_subprogram of
    "SLL"[SIGNED, INTEGER return SIGNED]: function is "array_sll_1164";
-- Result subtype: SIGNED(ARG'LENGTH-1 downto 0)
-- Result: SHIFT_LEFT(ARG, COUNT)

--------------------------------------------------------------------------------
--   Note : Function S.11 is not compatible with VHDL 1076-1987. Comment
--   out the function (declaration and body) for VHDL 1076-1987 compatibility.
--------------------------------------------------------------------------------
-- Id: S.11
function "srl" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;
 attribute builtin_subprogram of
    "SRL"[UNSIGNED, INTEGER return UNSIGNED]: function is "array_srl_1164";
-- Result subtype: UNSIGNED(ARG'LENGTH-1 downto 0)
-- Result: SHIFT_RIGHT(ARG, COUNT)

--------------------------------------------------------------------------------
--   Note : Function S.12 is not compatible with VHDL 1076-1987. Comment
--   out the function (declaration and body) for VHDL 1076-1987 compatibility.
--------------------------------------------------------------------------------
-- Id: S.12
function "srl" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;
 attribute builtin_subprogram of
    "SRL"[SIGNED, INTEGER return SIGNED]: function is "array_srl_1164";
-- Result subtype: SIGNED(ARG'LENGTH-1 downto 0)
-- Result: SIGNED(SHIFT_RIGHT(UNSIGNED(ARG), COUNT))

--------------------------------------------------------------------------------
--   Note : Function S.13 is not compatible with VHDL 1076-1987. Comment
-- out the function (declaration and body) for VHDL 1076-1987 compatibility.
--------------------------------------------------------------------------------
-- Id: S.13
function "rol" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;
 attribute builtin_subprogram of
    "ROL"[UNSIGNED, INTEGER return UNSIGNED]: function is "array_rol";
-- Result subtype: UNSIGNED(ARG'LENGTH-1 downto 0)
-- Result: ROTATE_LEFT(ARG, COUNT)

--------------------------------------------------------------------------------
--   Note : Function S.14 is not compatible with VHDL 1076-1987. Comment
--   out the function (declaration and body) for VHDL 1076-1987 compatibility.
```

```
  ------------------------------------------------------------------------
  -- Id: S.14
  function "rol" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;
   attribute builtin_subprogram of
      "ROL"[SIGNED, INTEGER return SIGNED]: function is "array_rol";
  -- Result subtype: SIGNED(ARG'LENGTH-1 downto 0)
  -- Result: ROTATE_LEFT(ARG, COUNT)

  ------------------------------------------------------------------------
  -- Note : Function S.15 is not compatible with VHDL 1076-1987. Comment
  --   out the function (declaration and body) for VHDL 1076-1987 compatibility.
  ------------------------------------------------------------------------
  -- Id: S.15
  function "ror" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;
   attribute builtin_subprogram of
      "ROR"[UNSIGNED, INTEGER return UNSIGNED]: function is "array_ror";
  -- Result subtype: UNSIGNED(ARG'LENGTH-1 downto 0)
  -- Result: ROTATE_RIGHT(ARG, COUNT)

  ------------------------------------------------------------------------
  --   Note : Function S.16 is not compatible with VHDL 1076-1987. Comment
  --   out the function (declaration and body) for VHDL 1076-1987 compatibility.
  ------------------------------------------------------------------------
  -- Id: S.16
  function "ror" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;
   attribute builtin_subprogram of
      "ROR"[SIGNED, INTEGER return SIGNED]: function is "array_ror";
  -- Result subtype: SIGNED(ARG'LENGTH-1 downto 0)
  -- Result: ROTATE_RIGHT(ARG, COUNT)

  --
================================================================================
=
  --   RESIZE Functions
  --
================================================================================
=

  -- Id: R.1
  function RESIZE (ARG: SIGNED; NEW_SIZE: NATURAL) return SIGNED;
   attribute builtin_subprogram of
      RESIZE[SIGNED, NATURAL return SIGNED]: function is "numstd_resize_sns";
  -- Result subtype: SIGNED(NEW_SIZE-1 downto 0)
  -- Result: Resizes the SIGNED vector ARG to the specified size.
  --         To create a larger vector, the new [leftmost] bit positions
  --         are filled with the sign bit (ARG'LEFT). When truncating,
  --         the sign bit is retained along with the rightmost part.

  -- Id: R.2
  function RESIZE (ARG: UNSIGNED; NEW_SIZE: NATURAL) return UNSIGNED;
```

```
    attribute builtin_subprogram of
        RESIZE[UNSIGNED, NATURAL return UNSIGNED]: function is "numstd_resize_unu";
  -- Result subtype: UNSIGNED(NEW_SIZE-1 downto 0)
  -- Result: Resizes the SIGNED vector ARG to the specified size.
  --         To create a larger vector, the new [leftmost] bit positions
  --         are filled with '0'. When truncating, the leftmost bits
  --         are dropped.

  --
============================================================================
=
  -- Conversion Functions
  --
============================================================================
=

  -- Id: D.1
  function TO_INTEGER (ARG: UNSIGNED) return NATURAL;
    attribute builtin_subprogram of
        TO_INTEGER[UNSIGNED return NATURAL]: function is "numstd_conv_integer_un2";
  -- Result subtype: NATURAL. Value cannot be negative since parameter is an
  --             UNSIGNED vector.
  -- Result: Converts the UNSIGNED vector to an INTEGER.

  -- Id: D.2
  function TO_INTEGER (ARG: SIGNED) return INTEGER;
    attribute builtin_subprogram of
        TO_INTEGER[SIGNED return INTEGER]: function is "numstd_conv_integer_si";
  -- Result subtype: INTEGER
  -- Result: Converts a SIGNED vector to an INTEGER.

  -- Id: D.3
  function TO_UNSIGNED (ARG, SIZE: NATURAL) return UNSIGNED;
    attribute builtin_subprogram of
        TO_UNSIGNED [NATURAL, NATURAL return UNSIGNED]: function is
"numstd_conv_unsigned_nu";
  -- Result subtype: UNSIGNED(SIZE-1 downto 0)
  -- Result: Converts a non-negative INTEGER to an UNSIGNED vector with
  --         the specified SIZE.

  -- Id: D.4
  function TO_SIGNED (ARG: INTEGER; SIZE: NATURAL) return SIGNED;
    attribute builtin_subprogram of
        TO_SIGNED [INTEGER, NATURAL return SIGNED]: function is "numstd_conv_signed_is";
  -- Result subtype: SIGNED(SIZE-1 downto 0)
  -- Result: Converts an INTEGER to a SIGNED vector of the specified SIZE.

  --
============================================================================
=
```

```
  -- Logical Operators
  --
================================================================================
=

  -- Id: L.1
  function "not" (L: UNSIGNED) return UNSIGNED;
   attribute builtin_subprogram of
      "NOT"[UNSIGNED return UNSIGNED]: function is "array_not_1164";
  -- Result subtype: UNSIGNED(L'LENGTH-1 downto 0)
  -- Result: Termwise inversion

  -- Id: L.2
  function "and" (L, R: UNSIGNED) return UNSIGNED;
   attribute builtin_subprogram of
      "AND"[UNSIGNED, UNSIGNED return UNSIGNED]: function is "array_and_1164";
  -- Result subtype: UNSIGNED(L'LENGTH-1 downto 0)
  -- Result: Vector AND operation

  -- Id: L.3
  function "or" (L, R: UNSIGNED) return UNSIGNED;
   attribute builtin_subprogram of
      "OR"[UNSIGNED, UNSIGNED return UNSIGNED]: function is "array_or_1164";
  -- Result subtype: UNSIGNED(L'LENGTH-1 downto 0)
  -- Result: Vector OR operation

  -- Id: L.4
  function "nand" (L, R: UNSIGNED) return UNSIGNED;
   attribute builtin_subprogram of
      "NAND"[UNSIGNED, UNSIGNED return UNSIGNED]: function is "array_nand_1164";
  -- Result subtype: UNSIGNED(L'LENGTH-1 downto 0)
  -- Result: Vector NAND operation

  -- Id: L.5
  function "nor" (L, R: UNSIGNED) return UNSIGNED;
   attribute builtin_subprogram of
      "NOR"[UNSIGNED, UNSIGNED return UNSIGNED]: function is "array_nor_1164";
  -- Result subtype: UNSIGNED(L'LENGTH-1 downto 0)
  -- Result: Vector NOR operation

  -- Id: L.6
  function "xor" (L, R: UNSIGNED) return UNSIGNED;
   attribute builtin_subprogram of
      "XOR"[UNSIGNED, UNSIGNED return UNSIGNED]: function is "array_xor_1164";
  -- Result subtype: UNSIGNED(L'LENGTH-1 downto 0)
  -- Result: Vector XOR operation

  -- ----------------------------------------------------------------------
  -- Note : Function L.7 is not compatible with VHDL 1076-1987. Comment
  -- out the function (declaration and body) for VHDL 1076-1987 compatibility.
```

```
-- ------------------------------------------------------------------------
-- Id: L.7
function "xnor" (L, R: UNSIGNED) return UNSIGNED;
 attribute builtin_subprogram of
    "XNOR"[UNSIGNED, UNSIGNED return UNSIGNED]: function is "array_xnor_1164";
-- Result subtype: UNSIGNED(L'LENGTH-1 downto 0)
-- Result: Vector XNOR operation

-- Id: L.8
function "not" (L: SIGNED) return SIGNED;
 attribute builtin_subprogram of
    "NOT"[SIGNED return SIGNED]: function is "array_not_1164";
-- Result subtype: SIGNED(L'LENGTH-1 downto 0)
-- Result: Termwise inversion

-- Id: L.9
function "and" (L, R: SIGNED) return SIGNED;
 attribute builtin_subprogram of
    "AND"[SIGNED, SIGNED return SIGNED]: function is "array_and_1164";
-- Result subtype: SIGNED(L'LENGTH-1 downto 0)
-- Result: Vector AND operation

-- Id: L.10
function "or" (L, R: SIGNED) return SIGNED;
 attribute builtin_subprogram of
    "OR"[SIGNED, SIGNED return SIGNED]: function is "array_or_1164";
-- Result subtype: SIGNED(L'LENGTH-1 downto 0)
-- Result: Vector OR operation

-- Id: L.11
function "nand" (L, R: SIGNED) return SIGNED;
 attribute builtin_subprogram of
    "NAND"[SIGNED, SIGNED return SIGNED]: function is "array_nand_1164";
-- Result subtype: SIGNED(L'LENGTH-1 downto 0)
-- Result: Vector NAND operation

-- Id: L.12
function "nor" (L, R: SIGNED) return SIGNED;
 attribute builtin_subprogram of
    "NOR"[SIGNED, SIGNED return SIGNED]: function is "array_nor_1164";
-- Result subtype: SIGNED(L'LENGTH-1 downto 0)
-- Result: Vector NOR operation

-- Id: L.13
function "xor" (L, R: SIGNED) return SIGNED;
 attribute builtin_subprogram of
    "XOR"[SIGNED, SIGNED return SIGNED]: function is "array_xor_1164";
-- Result subtype: SIGNED(L'LENGTH-1 downto 0)
-- Result: Vector XOR operation
```

```
  -- ------------------------------------------------------------------------
  -- Note : Function L.14 is not compatible with VHDL 1076-1987. Comment
  -- out the function (declaration and body) for VHDL 1076-1987 compatibility.
  -- ------------------------------------------------------------------------
  -- Id: L.14
  function "xnor" (L, R: SIGNED) return SIGNED;
   attribute builtin_subprogram of
      "XNOR"[SIGNED, SIGNED return SIGNED]: function is "array_xnor_1164";
  -- Result subtype: SIGNED(L'LENGTH-1 downto 0)
  -- Result: Vector XNOR operation

  --
============================================================================
=
  -- Match Functions
  --
============================================================================
=

  -- Id: M.1
  function STD_MATCH (L, R: STD_ULOGIC) return BOOLEAN;
   attribute builtin_subprogram of
      STD_MATCH[STD_ULOGIC, STD_ULOGIC return BOOLEAN]: function
                                 is "numstd_match_xx";
  -- Result subtype: BOOLEAN
  -- Result: terms compared per STD_LOGIC_1164 intent

  -- Id: M.2
  function STD_MATCH (L, R: UNSIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      STD_MATCH[UNSIGNED, UNSIGNED return BOOLEAN]: function
                                 is "numstd_array_match_1164";
  -- Result subtype: BOOLEAN
  -- Result: terms compared per STD_LOGIC_1164 intent

  -- Id: M.3
  function STD_MATCH (L, R: SIGNED) return BOOLEAN;
   attribute builtin_subprogram of
      STD_MATCH[SIGNED, SIGNED return BOOLEAN]: function
                                 is "numstd_array_match_1164";
  -- Result subtype: BOOLEAN
  -- Result: terms compared per STD_LOGIC_1164 intent

  -- Id: M.4
  function STD_MATCH (L, R: STD_LOGIC_VECTOR) return BOOLEAN;
   attribute builtin_subprogram of
      STD_MATCH[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return BOOLEAN]: function
                                 is "numstd_array_match_1164";
  -- Result subtype: BOOLEAN
  -- Result: terms compared per STD_LOGIC_1164 intent
```

```
  -- Id: M.5
  function STD_MATCH (L, R: STD_ULOGIC_VECTOR) return BOOLEAN;
   attribute builtin_subprogram of
      STD_MATCH[STD_ULOGIC_VECTOR, STD_ULOGIC_VECTOR return BOOLEAN]: function
                                  is "numstd_array_match_1164";
  -- Result subtype: BOOLEAN
  -- Result: terms compared per STD_LOGIC_1164 intent

  --
================================================================================
=
  -- Translation Functions
  --
================================================================================
=

  -- Id: T.1
  function TO_01 (S: UNSIGNED; XMAP: STD_LOGIC := '0') return UNSIGNED;
   attribute builtin_subprogram of
      TO_01[UNSIGNED, STD_LOGIC return UNSIGNED]: function is "numstd_to01_uu";
  -- Result subtype: UNSIGNED(S'RANGE)
  -- Result: Termwise, 'H' is translated to '1', and 'L' is translated
  --         to '0'. If a value other than '0'|'1'|'H'|'L' is found,
  --         the array is set to (others => XMAP), and a warning is
  --         issued.

  -- Id: T.2
  function TO_01 (S: SIGNED; XMAP: STD_LOGIC := '0') return SIGNED;
   attribute builtin_subprogram of
      TO_01[SIGNED, STD_LOGIC return SIGNED]: function is "numstd_to01_ss";
  -- Result subtype: SIGNED(S'RANGE)
  -- Result: Termwise, 'H' is translated to '1', and 'L' is translated
  --         to '0'. If a value other than '0'|'1'|'H'|'L' is found,
  --         the array is set to (others => XMAP), and a warning is
  --         issued.

 end NUMERIC_STD;
```

3.10 Conclusions

3.10.1 Conseils de méthode

Les exemples simples des chapitres précédents et futur suggèrent les principes suivants en ce qui concerne la description de circuits:

- Dans la mesure du possible, dans une description, on séparera les parties clairement séquentielles et les parties clairement combinatoires.

- Pour chaque bloc combinatoire, les instructions concurrentes sont en général plus directement adaptées.
- Pour chaque bloc séquentiel, les processus explicites sont en général plus simples à écrire ou à lire. Attention aux mémorisations implicites du signal. Utiliser au maximum les variables !
- Une bonne description est une description lisible. En application des principes ci-dessus, elle apparaîtra comme un ensemble d'instructions concurrentes et de processus concurrents, la liaison entre ces différents blocs se faisant de manière dynamique (par les signaux) au moment de la simulation ou alors lors de la synthèse du circuit global.

« Une bonne description est une description lisible »

3.10.2 Instructions séquentielles

Elles se situent obligatoirement dans un PROCESS ou un Sous-programme (PROCEDURE ou FUNCTION). L´ordre d´exécution est l´ordre d´écriture

- Contrôle IF, CASE, WAIT, Boucle LOOP, NEXT, EXIT
- Affectation de variable, de signal (le signal accepte les deux modes)
- ASSERT... REPORT
- RETURN pour une fonction
- NULL
- appel de sous-programme

3.10.3 Instructions concurrentes

Elles se situent dans n´importe quel ordre entre le BEGIN et le END d´une ARCHITECTURE ou d´un BLOCK.

- Affectation concurrente de signaux (le signal accepte les deux modes)
- PROCESS concurrents entre eux
- Appel de procédures
- BLOCK
- Instanciation de COMPONENT
- Instruction GENERATE

Pour chaque instruction concurrente, il existe un processus équivalent

4 Synthèse des circuits

4.1 Le synthétiseur

Le synthétiseur est un compilateur particulier capable, à partir du langage de descriptions VHDL ou Verilog, de générer une description structurelle du circuit. A travers le **style d'écriture** de la description synthétisable (niveau RTL Registre Transfer Level), le synthétiseur va reconnaître un certain nombre de **primitives** qu'il va implanter et connecter entre elles. Ce sera au minimum des portes NAND, NOR, NOT ,des bascule D, des buffers d'entrées-sorties, mais cela peut être aussi un compteur, un multiplexeur , une RAM, un registre , un additionneur, un multiplieur ou tout autre bloc particulier.

Le VHDL décrit la fonctionnalité souhaitée et ceci indépendamment de la technologie. Le nombre de primitives trouvées par le synthétiseur donne une évaluation de surface (pour un ASIC) ou de remplissage (pour un FPGA).

La technologie choisie apporte toutes les données temporelles relatives aux primitives, notamment les temps de traversée de chaque couche logique sont connus. Une première évaluation de vitesse d'ensemble du circuit peut ainsi être réalisée par le synthétiseur (en considérant des retards fixes par porte, ce qui constitue une approximation grossière).

Outre la description VHDL, le concepteur a la possibilité de fournir au synthétiseur des **contraintes** temporelles (réalistes) ou de caractéristiques électriques des entrées-sorties. Le synthétiseur, par un certain nombre d'itérations, essaiera d'**optimiser** le produit surface-vitesse .

Le synthétiseur idéal est un outil capable d'avoir jusqu'à la vision physique du circuit projeté. De tels outils sont en train de sortir sur le marché. De façon plus classique, l'optimisation du circuit final sera obtenu après avoir fait remonter les informations temporelles d'après routage au niveau du synthétiseur et après avoir procédé à plusieurs itérations.

4.2 La cible technologique

4.2.1 Asic et PLD

Les ASIC (Application Specific Integreted Circuit) sont des circuits intégrés numériques ou mixtes originaux. En ce qui concerne l'aspect numérique, on dispose en général d'une bibliothèque de fonctions pré-caractérisées c'est à dire optimisées par le fondeur et quant au reste du Design, le VHDL ou VERILOG ciblera des primitives NAND, NOR, Multiplexeur, Bascules D, et Mémoires RAM ou ROM.

Avantages de l'ASIC

- Originalité
- Intégration mixte analogique numérique
- Choix de la technologie
- Consommation

Inconvénients

- Coût de développement très élevés (ne peut être amorti en général que pour de très grand nombre de circuits)

- Délais de fabrication et de test
- Dépendance vis à vis du fondeur.

Les PLD (Programmable Logic Device) sont des circuits disponibles sur catalogue mais exclusivement numériques.

Avantages des PLD

- Personnalisation et mise en œuvre simple
- Outils de développement peu coûteux (souvent gratuits)
- Pas de retour chez le fabriquant
- Idéal pour le prototypage rapide

Inconvénients des PLD

- Peut être cher pour de grandes séries
- Consommation globale accrue par les circuits de configuration
- Les primitives du fabricant pouvant être complexes, la performance globale est dépendante pour beaucoup de l'outil utilisé.

Afin d'annuler en partie les inconvénients des PLD, les fabricants proposent des circuits au Top de la technologie 5 ce qui rend encore plus cher l'ASIC équivalent). Ainsi actuellement on trouve des circuits en technologie 0,13µm tout cuivre basse tension. La consommation est ainsi réduite et la vitesse augmentée d'autant.

4.2.2 Architectures

- CPLD (Complex Programmable Logic Device) : Ce sont des assemblages de macro-cellules programmables « simples » réparties autour d'une matrice d'interconnexion. Les temps de propagation de chaque cellules sont en principe prévisibles.
- FPGA (Field Programmable Gate Array) sont formés d'une mer de petits modules logiques de petite taille, noyés dans un canevas de routage. Du fait de la granularité plus fine des FPGA, les temps de propagation sont le résultat d'additions de chemins et sont plus difficiles à maîtriser que celui des CPLD.

4.2.3 Technologies

Les circuits sont des pré-diffusés, c'est à dire qu'une grande quantité de fonctions potentielles préexistent sur la puce de silicium. La programmation est l'opération qui consiste à créer une application en personnalisant chaque opération élémentaire.

- Eeprom : Le circuit se programme normalement et conserve sa configuration même en absence de tension.
- Sram : La configuration doit être téléchargée à la mise sous tension du circuit. S'il y a coupure d'alimentation, la configuration est perdue.
- Antifusible : La configuration consiste à faire sauter des fusibles pour créer des connexions. L'opération est irréversible mais en contre-partie offre l'avantage d'une grande robustesse et de sécurité au niveau du piratage possible du circuit.

- Flash : Le circuit se programme normalement et conserve sa configuration même en absence de tension.

4.2.4 Fabricants

Actel	Antifusible FPGA, Flash FPGA
Altera	Eeprom CPLD, Sram FPGA
Atmel	Flash CPLD, Sram FPGA
Cypress	Sram CPLD, Flash CPLD
Lattice	Eeprom CPLD, Sram CPLD, Sram FPGA
QuickLogic	Antifusible FPGA
Xilinx	Sram CPLD, Sram FPGA

4.3 Saisie du code RTL

4.3.1 Texte et graphique

Un simple éditeur de texte suffit bien évidemment pour saisir le code. Tous les synthétiseurs incluent leur propre éditeur plus ou moins élaboré. Certains outils généralistes comme ***emacs (GNU)*** offrent un mode VHDL personnalisable très sophistiqué permettant de gagner du temps lors de la saisie du texte.

Le graphique n'est pourtant pas exclu des outils de saisie. Il est toujours primordial de pouvoir dessiner graphiquement un diagramme d'état. De nouveaux outils sont apparus capable de générer automatiquement du VHDL à partir de graphique. Nous ne citerons que ***HDL_Designer*** de Mentor Graphics qui permet de créer des schémas blocs, des tables de vérité, des diagrammes d'état, des organigrammes avec intégration complète de la syntaxe VHDL et génération automatique du texte. Celui-ci reste en fin de compte la véritable source du projet.

4.3.2 Style

Le VHDL offre de nombreuses possibilités de style d'écriture pour une même fonctionnalité. Il est donc impératif de faire dans chaque cas le meilleur choix. La meilleure solution sera toujours **la plus lisible** c'est à dire simple, claire, documentée. Pour cela, outre les autres problèmes de conception, il n'est pas mauvais de se fixer quelques règles de conduite comme :

- Tous les identificateurs seront en minuscule, les mots clefs du langage en majuscule et les constantes commenceront par une majuscule et seront ensuite en minuscule.
- Les identificateurs auront un sens fort : adresse_ram plutôt que ra
- Les horloges s'appelleront toujours h ou clock
- Les signaux actifs à l'état bas seront terminés par _b ou _n

- Privilégier les DOWNTO aux TO pour la définitions des vecteurs
- Donner au fichier le même nom que l'entité qu'il contient.
- Privilégier au niveau de l'entité les types std_ulogic , unsigned ou signed

4.3.3 Principes

Le VHDL est un langage à instruction concurrentes, il faut savoir en profiter lorsqu'on décrit un circuit. La règle simple est de séparer les parties franchement combinatoires des parties comportant une horloge.

- Les parties combinatoires seront décrites par des instructions concurrentes
- Les parties séquentielles seront décrites par des processus explicites.

4.3.4 Limitation du langage

Lors des descriptions VHDL en vue de synthèse, c'est le style d'écriture et lui seul qui va guider le synthétiseur dans ses choix d'implantation au niveau circuit. Il est donc nécessaire de produire des instructions ayant une équivalence non ambiguë au niveau porte.

Le synthétiseur est un compilateur un peu particulier susceptible au fil des ans d'améliorer sa capacité à implanter des fonctions de plus en plus abstraites. Cependant, il reste des règles de bon sens comme « les retards des opérateurs est d'ordre technologique » ou bien « un fichier n'est pas un circuit » etc. En conséquence, les limitations du langage du niveau RTL les plus courantes sont :

- Un seul WAIT par PROCESS
- Les retards sont ignorés (pas de sens)
- Les initialisations de signaux ou de variables sont ignorées
- Pas d'équation logique sur une horloge (conseillé)
- Pas de fichier ni de pointeur
- Restriction sur les boucles (LOOP)

- Restriction sur les attributs de détection de fronts (EVENT, STABLE)
- Pas de type REAL
- Pas d attributs BEHAVIOR, STRUCTURE,LAST_EVENT, LAST_ACTIVE,TRANSACTION
- Pas de mode REGISTER et BUS

Exemple-1:

```
WAIT UNTIL h'EVENT AND h= '1';
  x := 2;
WAIT UNTIL h'EVENT AND h= '1';
```

Ces trois lignes sont incorrectes. Il faut gérer soi-même le comptage des fronts d'horloge:

```
WAIT UNTIL h'EVENT AND h= '1';
IF c = 1 THEN
    x := 2;
```

Exemple-2:

On ne sait pas faire le"ET" entre un front et un niveau. Il s'ensuit que la ligne suivante:

```
WAIT UNTIL h'EVENT AND h= '1' AND raz = '0';
```

doit être remplacée par

```
WAIT UNTIL h'EVENT AND h= '1' ;
IF raz = '0' THEN
```

Le synthétiseur fera le choix d'un circuit fonctionnant sur le front montant de h (bascule D) et prenant raz comme niveau d'entrée de validation.

Exemple-3:

```
SIGNAL compteur : INTEGER;
```

produira certainement un compteur 32 bits (l'entier par défaut). Si ce n'est pas cela qui est voulu, il faut le préciser . Par exemple pour 7 bits,

```
SIGNAL compteur : INTEGER RANGE 0 TO 99;
```

4.4 Circuits combinatoires

Une fonction combinatoire est une fonction dont la valeur est définie pour toute combinaison des entrées. Elle correspond toujours à une table de vérité. La fonction peut être complète (toutes les valeurs de sortie sont imposées à priori) ou incomplète (certaines sorties peuvent être définies comme indifféremment 1 ou 0 au grès de la simplification). Ce dernier état sera noté "–" en VHDL.

Une fonction combinatoire est donc donnée :

- Par une table de vérité
- Par une équation simplifiée ou non

Une fonction combinatoire peut être décrite de façon séparée

- Par une instruction concurrente (méthode à privilégier)
- Par un processus avec toutes les entrées en liste de sensibilité
- Par un tableau de constantes
- Par une fonction qui sera ensuite assignée de façon concurrente ou séquentielle à un signal.

Il y a aussi un cas très fréquent correspondant à une équation combinatoire noyée dans une description comportant des éléments de mémorisation. Dans ce dernier cas seule, la vigilance est conseillée.

4.4.1 Multiplexeurs, démultiplexeurs, décodeurs

4.4.1.1 Multiplexeur 4 bits

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY multiplexeur4 is
      PORT( entree : IN std_logic_VECTOR (3 DOWNTO 0);
            adresse : IN std_logic_VECTOR (1 DOWNTO 0);
                  s : OUT std_logic);
 END;
                  ARCHITECTURE concurrente OF multiplexeur4 IS
BEGIN
      s <= entree(0) WHEN adresse = "00" ELSE
           entree(1) WHEN adresse = "01" ELSE
           entree(2) WHEN adresse = "10" ELSE
           entree(3) ;
END;
--------------------------------------------------------------------------
          ARCHITECTURE selection OF multiplexeur4 IS
          BEGIN
              WITH adresse SELECT
 s  <=   entree(0) WHEN  "00",
          entree(1) WHEN "01",
          entree(2) WHEN "10",
           entree(3) WHEN OTHERS;
          END;
          --------------------------------------------------------------------------
          ARCHITECTURE processus_explicite OF multiplexeur4 IS
          BEGIN
               PROCESS(entree, adresse)
              BEGIN
CASE adresse IS
       WHEN "00" => s<= entree(0);
       WHEN "01"  => s<= entree(1);
        WHEN  "10" => s <= entree(2);
        WHEN OTHERS => s <= entree(3);
END CASE;
             END PROCESS;
          END;
         --------------------------------------------------------------------------
          LIBRARY ieee;
         USE ieee.numeric_std.ALL;

        ARCHITECTURE rapide OF multiplexeur4 IS
       BEGIN
s <= entree(to_integer(unsigned(adresse)));
       END ;
```

4.4.1.2 Décodeur binaire

Décodeur binaire du commerce type 74ls 138 ou équivalent.

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

ENTITY ls138 IS
                    PORT (          g1, g2a, g2b, a, b, c : IN STD_ULOGIC;
                            y0, y1, y2, y3, y4, y5, y6,y7 : OUT STD_ULOGIC);
END ls138;


ARCHITECTURE avec_conversion OF ls138 IS
BEGIN
    PROCESS
        VARIABLE y : unsigned(0 TO 7);
        VARIABLE adresse : INTEGER RANGE 0 TO 7;
        VARIABLE AConvertir : unsigned(3 DOWNTO 0);
    BEGIN
        WAIT ON g1, g2a, g2b, a, b, c;
        AConvertir := '0' & c & b & a ;
        adresse := to_integer(AConvertir);
        y := "11111111";
        IF g1 = '1' AND g2a = '0' AND g2b = '0' THEN
            y(adresse) := '0';
        END IF;
        y0 <= y(0);
        y1 <= y(1);
        y2 <= y(2);
        y3 <= y(3);
        y4 <= y(4);
        y5 <= y(5);
        y6 <= y(6);
        y7 <= y(7);
    END PROCESS;
END;

 ARCHITECTURE equations OF ls138 IS

                    SIGNAL valide : STD_ULOGIC;

BEGIN
                    valide <= g1 AND (NOT g2a) AND (NOT g2b);
                    y7 <= NOT(c AND b AND a AND valide) ;
                    y6 <= NOT(c AND b AND (NOT a) AND valide);
                    y5 <= NOT(c AND (NOT b) AND a AND valide);
                    y4 <= NOT(c AND (NOT b) AND (NOT a) AND valide);
                    y3 <= NOT((NOT c) AND b AND a AND valide);
                    y2 <= NOT((NOT c) AND b AND (NOT a) AND valide);
```

```
                    y1 <= NOT((NOT c) AND (NOT b) AND a AND valide);
                    y0 <= NOT((NOT c) AND (NOT b) AND (NOT a) AND valide);
END equations;
```

Décodeur BCD-7segments
Les afficheurs sont des anodes communes. Pour émettre de la lumière, ils doivent être traversés par un courant et donc recevoir un niveau haut (’1’) sur leur anode et un niveau bas (’0’) sur leur cathode.

On considère 16 affichages possibles de 0 a 9, de A à F
La fonction se trouve dans le fichier primitives.vhd

```
function hexa_7seg (e : unsigned)
  RETURN unsigned IS

  TYPE tableau IS ARRAY (0 TO 15) OF unsigned(6 DOWNTO 0);
  CONSTANT segments : tableau := ("0000001",
                    "1001111",
                    "0010010",
                    "0010010",
                    "1001100",
                    "0100100",
                    "0100000",
                    "0001111",
                    "0000000",
                    "0001100",
                    "0001000",
                    "1100000",
                    "0110001",
                    "1000010",
                    "0110000",
                    "0111000");
  VARIABLE entree : unsigned(3 DOWNTO 0);

 BEGIN
  entree := e;
  RETURN segments(to_integer(entree));
END;
```

4.4.2 Circuits arithmétiques

4.4.2.1 Représentations des nombres en virgule fixe

4.4.2.1.1 Nombres non signés

En base 2, avec n bits, on dispose d’un maximum de 2^n combinaisons. Un nombre non signé peut être exprimé par $b_{n-1}2^{n-1}+b_{n-2}2^{n-2}+\ldots+b_0+b_{-1}2^{-1}+b_{-2}2^{-2}+\ldots+b_{-m}2^{-m}$ avec n bits de partie entière et m bits de partie fractionnaire. Cette représentation est appelée virgule fixe. La virgule est quelque chose de totalement fictif d’un point de vue circuit et un tel nombre peut toujours être vu comme un entier à condition de le multiplier par 2^m. La suite 0101 sur 4 bits représente aussi bien 5 que 2.5 ou 1 .25.

4.4.2.1.2 Nombres signés : complément à 2

A l'origine, on cherche une représentation pour des nombres signés la plus simple possible au niveau des circuits devant les traiter. En l'occurrence l'**addition**. Quel est le nombre que je dois rajouter à A pour avoir 0 ? Je déciderais qu'un tel nombre est une représentation de –A. Par exemple sur 4 bits,

$A = (a_3\ a_2\ a1\ a_0)$

$\bar{A} = (/a_3\ /a_2\ /a1\ /a_0)$

$A + \bar{A} = (1\ 1\ 1\ 1)$

$A + \bar{A} + 1 = (1\ 0\ 0\ 0\ 0)$

La somme sur n + 1 bits vaut 2^n mais 0 sur n bits. Privilégiant ce derniers cas, on admet alors que $\bar{A} + 1$ est une représentation possible de –A. Il en découle alors que sur n bits, un entier signé sera représenté de -2^{n-1} à $+2^{n-1}-1$. On peut aussi plus généralement exprimer un nombre signé par la relation :

$$-b_{n-1}2^{n-1} + b_{n-2}{}^{n-2} + \ldots + b_1 2^1 + b_0$$

4.4.2.2 Additionneurs , soustracteurs, décalages, comparateurs

Les opérateurs de base arithmétiques ou logiques +, -, NOT, * , AND, >, > , = , shift_left, shift_right sont des fonctions courantes disponibles dans des paquetages standards tels que IEEE.NUMERIC_STD. Le synthétiseur les accepte tel quel avec des types integer, signed ou unsigned.

D'autres fonctions particulières pourraient enrichir le catalogue de primitives

Il n'y a aucun problème pour décrire une quelconque opération mais attention à l'éventuel dimensionnement des mots traités qui pourrait s'avérer incorrect surtout lorsqu'on utilise des conversions de type.

Exemple: S <= a + b ;

S doit comporter le même nombre de bits que a ou b.

4.4.2.3 Multiplication ou division par une puissance de 2

D'après ce qui précède, nous déduisons une première règle en terme de circuit :

Une division ou une multiplication par une puissance de 2 **ne coûte rien** sinon en nombres de bits significatifs.

Diviser ou multiplier un nombre par une puissance de 2 revient à un simple câblage.

S <= '0' & e(7 DOWNTO 1); -- S est égal à e/2

4.4.2.4 Multiplication par une constante

Cela sera le plus souvent très facilement réalisé par un nombre limité d'additions ou encore mieux, par un nombre minimal d'additions et de soustractions.

Supposons la multiplication 7 * A en binaire. Elle sera décomposée en A + 2*A + 4 *A. Comme les multiplications par 2 et 4 ne coûtent rien, la multiplication par 7 ne nécessite que 2 additionneurs.

On remarque cependant que 7*A peut s'écrire 8*A –A , on peut donc réaliser cette même multiplication par un seul soustracteur.

De façon générale, l'optimisation du circuit se ferra en cherchant la représentation ternaire (1, 0, -1) des bits qui minimise le nombre d'opérations à effectuer (et maximise le nombre de bits à zéro). Mais attention, une telle représentation d'un nombre n'est pas unique. 6 = 4 + 2 = 8 – 2.

4.4.2.5 Multiplication d'entiers

A priori, la multiplication est une opération purement combinatoire. Elle peut d'ailleurs être implantée de cette façon. Disons simplement que cela conduit à un circuit très volumineux et qui a besoin d'être optimisé en surface et en nombre de couches traversées. De telles optimisations existent assez nombreuses. Citons simplement le multiplieur disponible en synthèse avec Leonardo, c'est le multiplieur de Baugh-Wooley .

```
S <= a * b ; -- multiplieur combinatoire
```

L'autre méthode beaucoup plus économique mais plus lente car demandant n itérations pour n bits est la méthode par additions et décalages. On peut en donner deux formes algorithmique, une dite sans restauration et l'autre plus anticipative dite avec restauration (voir paragraphe consacré à la structuration)

4.4.2.6 Division d'entiers non signés

Le diviseur le plus simple procède à l'inverse de la multiplication par addition et décalage et produit un bit de quotient par itération. On peut en donner une forme algorithmique dite « sans restauration ». Le dividende n bits est mis dans un registre 2n + 1 bit qui va construire le reste et le quotient. Le diviseur est place dans un registre n + 1 bits.

```
INIT : P = 0, A = dividende, B = diviseur
REPETER n fois
    SI P est négatif ALORS
        décaler les registres (P,A) d'une position à gauche
        Ajouter le contenu de B à P
    SINON
        décaler les registres (P,A) d'une position à gauche
        soustraire de P le contenu de B
    FIN_SI
    SI P est négatif
        mettre le bit de poids faible de A à 0
    SINON
        Le mettre à 1
    FIN_SI
FIN_REPETER
```

On constate très facilement que cet algorithme implique pour un circuit de division :

- Un additionneur/soustracteur 2n + 1 bits
- Un registre à décalage 2n + 1 bits avec chargement parallèle
- Un registre parallèle n + 1 bits

4.5 Conception synchrone

4.5.1 Maîtriser les retards

La difficulté de la conception des circuits numériques provient du fait que les méthodes connues et éprouvées (simplification des fonctions combinatoires entre autres) ne font que mettre en équations (aspect fonctionnel du problème posé) mais ignorent totalement les temps de propagation des signaux (dépendance avec la technologie). Ainsi , une fonction valant toujours 0 au niveau de son équation (de type A **.** /A) peut valoir 1 de façon transitoire une fois implantée sous forme technologique. Ceci est appelé **aléa de continuité**.

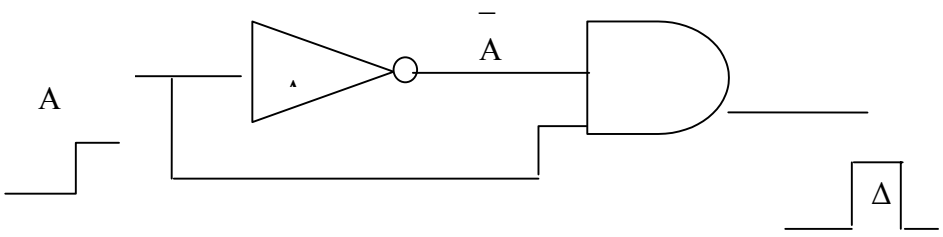


Figure 1

Lorsqu'on implante un circuit combinatoire complexe (grand nombre d'entrées et de sorties), ce phénomène apparaît inévitablement sur les sorties pour certaines séquences d'entrées. Il suffit pour cela que entre la sortie considérée et une entrée particulière, existent plusieurs **chemins** possibles avec des temps de propagation différents (généralisation de l'exemple proposé ici). Ces impulsions parasites non prévues par la théorie peuvent être la source de non fonctionnement du circuit si une parade n'est pas mise en œuvre.

Un premier moyen de supprimer l'aléa de continuité est d'introduire des redondances au niveau de chaque fonction combinatoire (on rajoute le consensus par rapport à la variable qui pourrait générer l'aléa. Dans l'exemple servant d'illustration, on implanterait A **.** /A **.** //A). Ce n'est pas toujours simple à faire et cela introduit un surcoût important de surface occupée (3 portes au lieu de 2 dont un ET à 3 entrées au lieu de 2 dans le même exemple).

La deuxième méthode pour supprimer les aléas est de les laisser se produire puis de les filtrer temporellement en échantillonnant le signal de sortie à des moments privilégiés (en dehors des instants d'apparition des aléas) . On va utiliser systématiquement un signal extérieur qui donnera les instants d'échantillonnage, c'est ce qui est appelé la **conception synchrone**.

4.5.2 Définition du synchronisme

Un système est synchrone si :

⇒ Tous les éléments de mémorisation sont sensibles à un front et non sensibles à un niveau (bascules)

⇒ L'horloge d'entrée de chaque composant sensible à un front (bascule) est construite à partir d'un même front de l'horloge primaire

En VHDL, le synchronisme et le signal d'horloge sont traduits par une des écritures suivantes :

WAIT UNTIL rising_edge (h);

WAIT UNTIL h'event AND h = '1' AND h'LAST_VALUE='0'; -- ce qui est la même chose. Ou bien , dans un processus sensibilisé par h (WAIT ON h),

IF rising_edge(h) THEN

Le synthétiseur va implanter un élément sensible à un front (une bascule, un registre) et

chaque signal affecté à la suite d'une de ces instructions correspondra à une sortie de bascule ou de registre

4.5.3 Bascule (ou registre) D

C'est l'élément de base de toute synchronisation puisqu'elle fonctionne sur front et que sa sortie ne fait que recopier l'entrée. $Q_{n+1} = D$. Elle peut être réalisée selon la technologie par 6 portes NAND ou directement optimisée par des portes de transfert en technologie CMOS. La seule nuance que l'on peut y trouver concerne sa remise à zéro (ou à un) qui peut être synchrone ou asynchrone.

En synthèse VHDL, une bascule D est insérée chaque fois qu'un signal est affecté sous le contrôle d'une horloge, ainsi :

```
PROCESS :
BEGIN
        WAIT UNTIL rising_edge(h); -- définit un signal h horloge
        IF raz = '1' THEN -- raz prioritaire
                q <= '0'; -- définit une raz synchrone pour q sortie d'une bascule
        ELSE
                q <=  d ; -- la sortie synchronise l'entrée
        END IF ;
END PROCESS ;
```

Si q, d sont de type vecteur de n bits, alors sera implanté un **registre** (n bascules en parallèle) avec raz synchrone. La bascule avec remise à zéro asynchrone peut aussi être reconnue par le synthétiseur. Il suffit d'adopter le style d'écriture suivant :

```
PROCESS :
BEGIN
  WAIT ON h, raz ;
  IF raz = '1' THEN -- raz prioritaire et asynchrone
          q <= '0'; -- définit une raz synchrone pour q sortie d'une bascule
  ELSIF rising_edge(h) THEN -- définit un signal h horloge
          q <=  d ; -- la sortie synchronise l'entrée
  END IF ;
END PROCESS ;
```

Un certain nombre de contraintes temporelles sont associées à ces bascules ; ce sont principalement les temps de préconditionnement (Tsetup), de maintien (Thold) et la largeur minimale d'horloge.

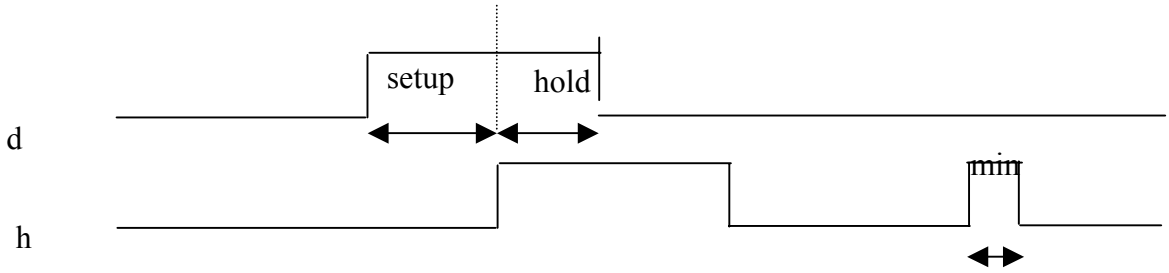


Figure 2

La réponse correcte de la bascule en fonctionnalité comme en temps de propagation (Tphl, Tplh) est garantie si on respecte ces caractéristiques, à savoir :

Les données d doivent être stable avant le font d'horloge (préconditionnement) mais aussi après (maintien). Bien heureusement, dans un système synchrone ces contraintes seront facilement assurées. Lorsque les entrées d sont des signaux internes à un système, on maîtrise le moment de leur transition. A l'inverse il y a des cas où ces contraintes ne peuvent pas être toujours assurées, c'est si les entrées d sont de type externes dont les transitions peuvent survenir à un instant quelconque. Il existe alors une probabilité pour que la transition se trouve dans la fenêtre interdite Tsetup + Thold autour du front de h. Que risque t-il de se passer alors ?

Le circuit peut « hésiter » c'est à dire rallonger son temps de propagation en restant un certain temps en état métastable puis rejoindre un état logique 0 ou 1 de façon non prévisible.

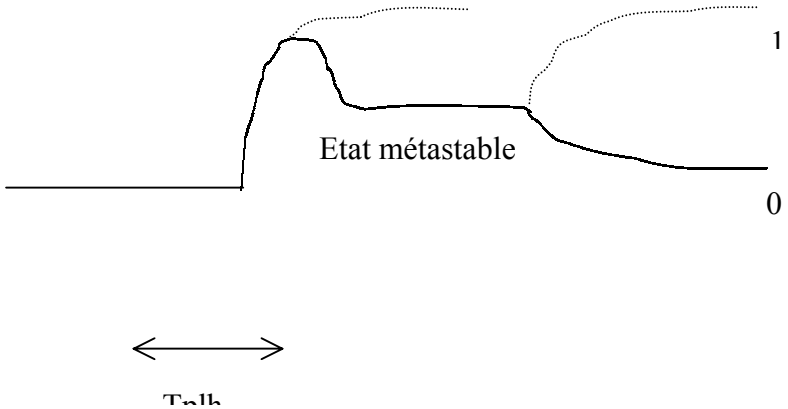


Figure 3

Ce phénomène ne peut avoir aucune conséquence. Un signal en train de passer à 1 sera vu comme zéro pour ce front d'horloge puis comme 1 au front suivant. S'il avait été vu comme 1, on aurait juste anticipé la prise de décision d'une période.

Le cas contraire est celui d'un défaut de synchronisation. La bascule censée synchroniser rentre en état métastable et deux circuits connectés sur la sortie de cette même bascule divergent d'appréciation. L'un voit dans l'état métastable un 1 alors que l'autre voit un 0. Une erreur peut ainsi intervenir dans le système. Cette erreur est liée à un probabilité (celle de l'apparition de l'état métastable), on la définie en lui associant un MTBF (Mean Time

Between Failures) taux moyen de bon fonctionnement. Celui-ci se calcule par l'équation suivante :

$MTBF = e^{K2*t} / (F1*F2*K1)$

K1 représente la fenêtre de préconditionnement

K2 est un coefficient représentatif de la vitesse avec laquelle le circuit sort de l'état métastable. Il est proportionnel au produit gain-bande passante des inverseurs constituants la mémoire. Une peite augmentation de K2 améliore énormément le MTBF

Avec F1=1MHz, F2= 10MHz et K1 = 0 .1ns , on obtient $MTBF = 10^{-3} * e^{k2*T}$

Par exemple , pour un circuit Xilinx XC4005-3 CLB, K2 = 7.9 /ns ce qui donne un MTBF égal à 1heure pour un échantillonnage en sortie retardé de 2 ns par rapport à l'horloge mais un MTBF de presque 1 année pour un retard de 3 ns.

4.5.4 Bascule (ou registre) E

La bascule E (comme ENABLE) est construite à partir d'une bascule D mais en lui rajoutant une condition d'activation ce qui correspond le cas le plus courant d'utilisation.

Elle correspond à une écriture VHDL telle que :

```
PROCESS :
        BEGIN
                WAIT UNTIL rising_edge(h); -- définit un signal h horloge
                IF raz = '1' THEN -- raz prioritaire
                        q <= '0'; -- définit une raz synchrone pour q sortie d'une bascule
                ELSIF en = '1' THEN – condition de validation
                        q <=  d ; -- la sortie synchronise l'entrée
                END IF ;
END PROCESS ;
```

Autrement dit , si en = '0', la sortie ne change pas $Q_{n+1}=Q_n$, il y a mémorisation. Dans le cas contraire, on retrouve une bascule D.

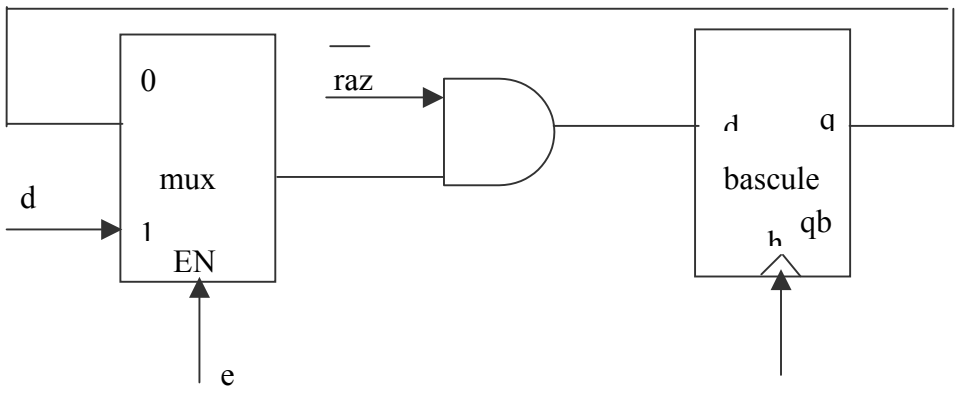


Figure 4

C'est l'élément le plus couramment utilisé dans un système synchrone.

4.5.5 Registre à décalages

Il s'agit d'une association synchrone de bascules D en cascade .

Description d'un registre a décalage gauche-droite, 8 bits avec chargement parallèle (synchrone) et remise a zéro (asynchrone). L'horloge est active sur front montant.

```
ENTITY registre_decalage IS
        GENERIC (Nb_bits : natural := 8;
                Tps : Time := 15 ns; --Temps de propagation horloge->sortie : 15 ns
                Tpas : Time := 18 ns);-- Temps de propagation raz->sortie : 18 ns
        PORT ( h, raz, edg, edd : IN STD_ULOGIC;
                        sel : IN STD_ULOGIC_VECTOR(1 DOWNTO 0);
                d_entree : IN STD_ULOGIC_VECTOR(Nb_bits -1 DOWNTO 0);
                    d_sortie : OUT STD_ULOGIC_VECTOR(Nb_bits -1 DOWNTO 0));
END;
------------------- 4 fonctions selon les valeurs de selection (sel)
--              | sel    | d_sortie
--              | "00"  | inchangee
--              | "01"  | decalage a droite avec bit edd a gauche
--              | "10"  | decalage a gauche avec bit edg a droite
--              | "11"  | Chargement de d_entree

ARCHITECTURE un_process OF registre_decalage IS
BEGIN
sr : PROCESS
VARIABLE stmp : STD_ULOGIC_VECTOR(Nb_bits -1 DOWNTO 0) ;
BEGIN
        WAIT ON raz, h;
        IF raz = '1' THEN
                stmp := ( OTHERS => '0');
                d_sortie <= stmp after tpas;
        ELSIF h'last_value = '0' AND h = '1' THEN
                CASE sel IS
                        WHEN "11" => stmp := d_entree ;
                        WHEN "10" => stmp := stmp(Nb_bits - 2 DOWNTO 0) & edg;
                        WHEN "01" => stmp := edd & stmp(Nb_bits -1 DOWNTO 1);
                        WHEN OTHERS => NULL;
                END CASE;
                d_sortie <= stmp after Tps;
        END IF;
END PROCESS;
END;
```

4.5.6 Les compteurs

Les compteurs sont reconnus en tant que tels par le synthétiseur. L'incrémentation d'une variable dans une boucle suffit pour désigner cette fonction.

4.5.6.1 Compteur générique synchrone

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

ENTITY compteur_gen IS
  GENERIC (Nb_Bits : natural := 4; --les valeurs par defaut
         Modulo : natural :=16);  -- sont optionelles
  PORT ( h, compter, raz : IN std_ulogic;
        sortie :  OUT std_logic_vector(Nb_bits-1 DOWNTO 0);

        retenue :  OUT std_ulogic);
BEGIN
  ASSERT ----- controle des parametres
                          Modulo <= 2**Nb_bits
                      REPORT "erreur sur les parametres"
                      SEVERITY warning;
END compteur_gen;

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

ARCHITECTURE synchrone OF compteur_gen IS

   SIGNAL s  : unsigned(Nb_bits-1 DOWNTO 0);
BEGIN

  P1 :PROCESS
    VARIABLE c : natural range 0 TO Modulo -1;
  BEGIN
-- description entierement synchrone
    WAIT UNTIL RISING_EDGE(h);
    IF raz = '1' THEN
      c :=0;
    ELSIF compter = '1' THEN
      IF c < Modulo -1 THEN
        c := c + 1;
      ELSE
        c := 0;
      END IF;
    END IF;
    s <= to_unsigned(c, Nb_bits);
  END PROCESS;

  sortie <= std_logic_vector(s);       -- cast
```

```
  retenue <= '1' WHEN s =to_unsigned(Modulo-1,Nb_bits) ELSE '0';

END synchrone;
```

4.5.6.2 Compteur générique asynchrone

```
ARCHITECTURE asynchrone OF compteur_gen IS
  SIGNAL s  : unsigned(Nb_bits-1 DOWNTO 0);
BEGIN

 P1 :PROCESS
  VARIABLE c : natural range 0 TO Modulo -1;
 BEGIN
  WAIT ON raz, h;
  IF raz = '1' THEN
    c :=0;
  ELSIF  RISING_EDGE(h) THEN
    IF  compter = '1' THEN
      IF c < Modulo -1 THEN
        c := c + 1;
      ELSE
        c := 0;
      END IF;
    END IF;
  END IF ;
  s <= to_unsigned(c, Nb_bits) ;
 END PROCESS;

  sortie <= std_logic_vector(s);       -- cast

 retenue <= '1' WHEN s = to_unsigned( Modulo-1, Nb_bits) ELSE '0';

END asynchrone;
```

4.5.6.3 Test du compteur générique

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

ENTITY test_compteur_gen IS END;

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

USE WORK.utils.all; --pour acces a la fonction horloge
```

```
ARCHITECTURE par_assertion OF test_compteur_gen IS

 COMPONENT compt
  GENERIC (Nb_Bits : natural ; --les valeurs par defaut
        Modulo : natural);  -- sont optionelles
  PORT ( h, compter, raz : IN STD_ULOGIC;
       sortie :  OUT std_logic_vector(Nb_bits-1 DOWNTO 0);
       retenue :  OUT STD_ULOGIC);
 END COMPONENT;

--   FOR C1 : compt USE ENTITY work.compteur_gen(asynchrone);
 FOR C1 : compt USE ENTITY work.compteur_gen(synchrone);

 CONSTANT Temps : TIME := 10 ns;
 CONSTANT M : natural := 17;
 CONSTANT N : natural := 5;
 SIGNAL h,ra,c, re : STD_ULOGIC;
 SIGNAL s : std_logic_vector(N-1 downto 0);

BEGIN
 H1: horloge(h,Temps,Temps); -- frequence 50 Meg

 C1: compt
  GENERIC MAP ( N,M)   -- compteur modulo 17
  PORT MAP  (h, c, ra, s, re);

 ra <= '1', '0' AFTER (Temps + Temps/2);
 c <= '1';

 verif:PROCESS
  VARIABLE s0 : unsigned(N-1 downto 0);
-- puisque le compteur fonctionne sur front montant, on le teste
-- sur front descendant
 BEGIN
  WAIT UNTIL FALLING_EDGE(h);
  ASSERT unsigned(s) < To_unsigned(M,N)
   REPORT "sortie hors intervalle";
  ASSERT (unsigned(s) - s0) < "00010"
   REPORT "on doit etre en fin de comptage";
  s0 := unsigned(s);
 END PROCESS;

END  par_assertion;
```

4.6 Structuration d'un circuit synchrone

4.6.1 Principe de la structuration par flot de données

Pour des circuits dont la complexité ne nécessite pas une architecture basée sur un microprocesseur, on va pouvoir simplement structurer c'est à dire décomposer en des parties plus facilement optimisables le circuit à réaliser.

En général, le circuit doit implanter une certaine relation entre des données d'entrée et des données de sorties, cette relation étant souvent spécifiée par un algorithme. On regarde tout d'abord quelles sont les opérations élémentaires de cet algorithme, quelles sont les variables internes à prévoir. On a alors une idée assez précise des opérateurs à implanter ainsi que des registres à prévoir pour les variables.

On peut alors organiser le pilotage des opérations, leur séquencement. Ce sera la conception d'une machine d'états finis spécifique du problème traité. Cela donne la structure en deux blocs de la figure 6

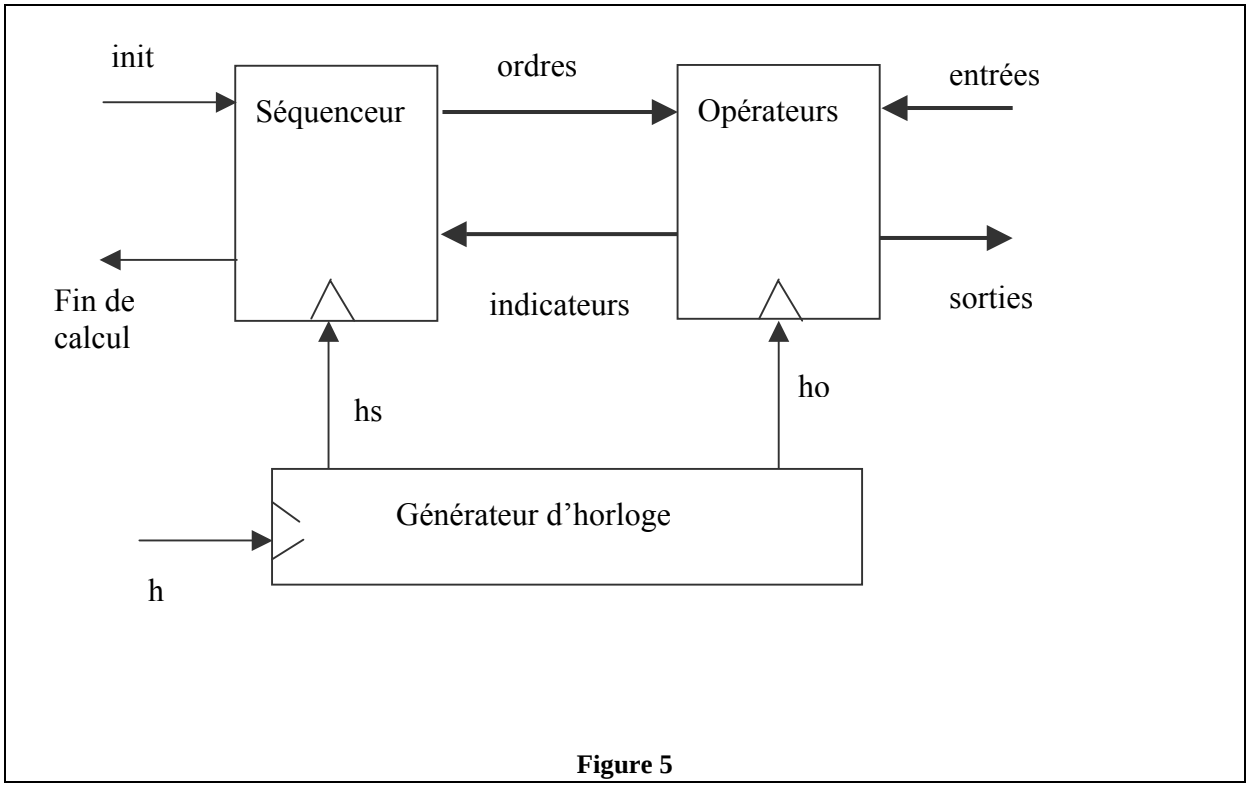
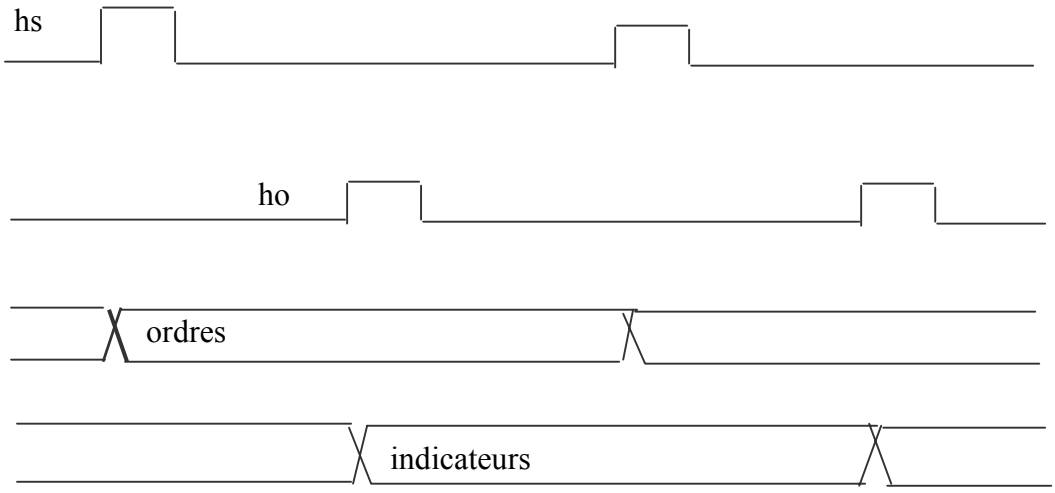


Figure 5

4.6.2 Séquencement

Le séquenceur et le bloc opération sont par principes synchrones. Une étude du séquencement de l'ensemble montre qu'il faut disposer d'une horloge biphasée ce qui peut être obtenue simplement en prenant hs = h et ho = inverse de h.



4.6.3 Machines d'états finis

4.6.3.1 Moore ou Mealy ?

Pour les circuits séquentiels simples que sont les compteurs ou plus généralement les machines d'état où l'on raisonne en *état présent* → *état futur*, avec des conditions de transitions, l'état sera matérialisé par n bascules d avec horloge commune . L'état futur est calculé par le décodeur d'entrée en fonction de l'état présent et des conditions d'entrée.

Dans une machine de type MOORE, les sorties ne dépendent que de l'état interne.

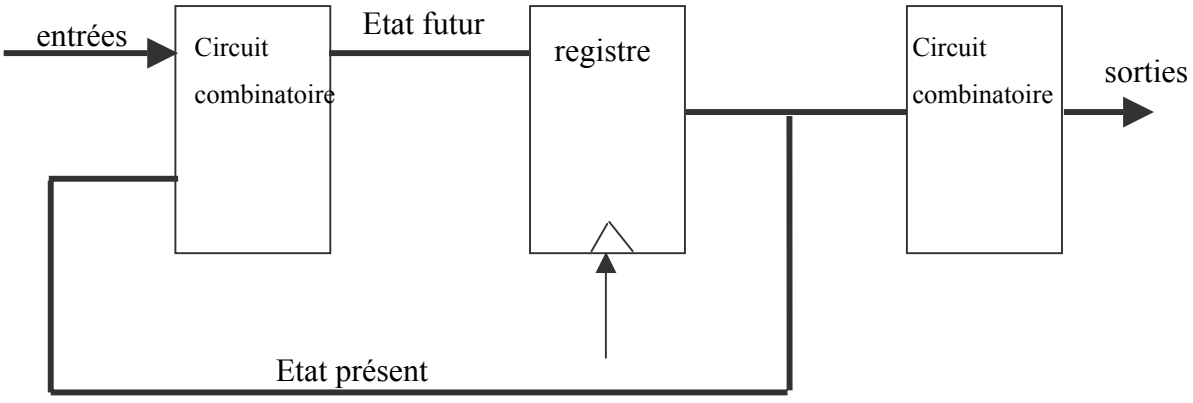


Figure 6 : machine de Moore synchrone

Dans une machine de type MEALY, les sorties sont fonctions de l'état courant et des entrées. Ceci implique un aspect partiellement asynchrone, on doit donc resynchroniser ces sorties par

un registre si l'on veut être totalement synchrone. Mais les sorties sont alors retardées d'une période d'horloge (ou moins pour leur part asynchrone).

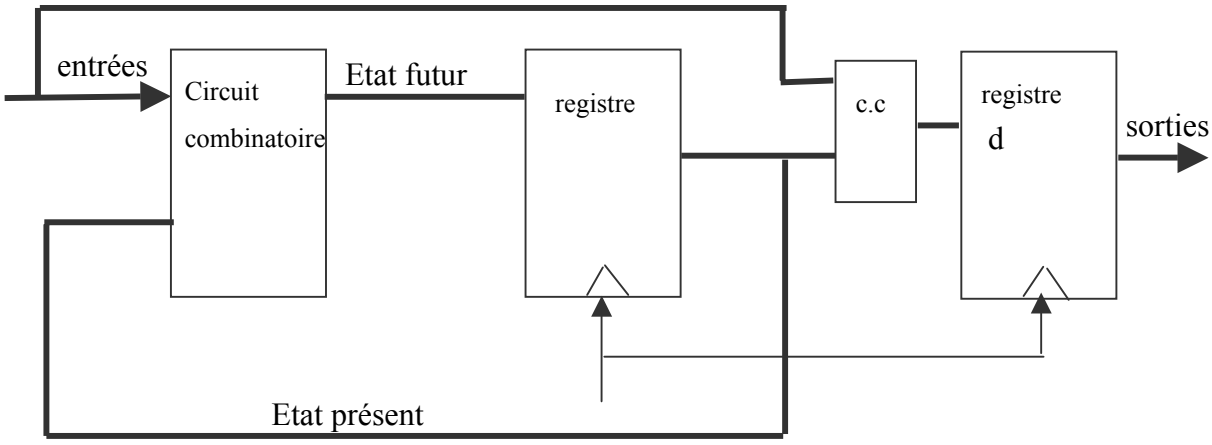


Figure 7 : machine de Mealy synchrone

.

4.6.3.2 Implantation VHDL

L´état peut être un signal de type entier, booléen, bit_vector ou énuméré. Dans ce dernier cas, en synthèse, une option permet de fixer la méthode de codage. Ce peut être binaire pur(0,1,2...), binaire réfléchi (0,1,3,2,6...), code à décalage(...001,...010,...100), optimisé ou encore codage aléatoire.

Le graphe de transitions se décrit par une instruction CASE portant sur les différents états. La partie mémorisation associée peut être intégrée dans le même process ou encore totalement dissociée.

Pour les équations de sortie d'une machine de Moore, chaque sortie ne dépend que de l´état présent. On l´exprime très simplement par une instruction concurrente (ou un process sensible au signal etat_present). Il en est de même pour une machine de Mealy, la seule différence étant la présence des entrées dans l'équation de sortie.

4.6.3.2.1 Description abstraite

- Définition du type et des signaux d'entrée.

```
TYPE etat IS (debut, etat1,etat2, fin); --4 etats par exemple
SIGNAL  raz, h : BIT;
SIGNAL etat_present, etat_futur : etat;
SIGNAL c1, c2, c3 : BOOLEAN;
BEGIN
```

- Mémorisation de l'état

```
initialisation: PROCESS
BEGIN
                        WAIT ON raz, h;
                        IF raz = ´1´ THEN -- asynchrone
                                etat_present <=  a ;
                        ELSIF h´EVENT AND h = ´1´ THEN
                                etat_present <= etat_futur;
                        END IF;
END PROCESS
```

- Description du graphe de transition

```
graphe: PROCESS
BEGIN
                        WAIT ON etat_present,c1,c2,c3 ;
                        CASE  etat_present IS
                                WHEN  debut =>
                                        IF c1 THEN etat_futur <= etat1 ;
                                        ELSE etat_futur <= etat2;
                                        END IF;
                                WHEN etat1 =>
..... etc .....
                        END CASE;
END PROCESS;
```

4.6.3.2.2 Description en un seul process :

Au lieu de séparer combinatoire et séquentiel, on écrit tout dans un seul process. Cette méthode sera utilisée dans l'exemple qui suit.

4.6.4 Multiplieur par additions et décalages

Soit à réaliser un circuit multiplieur n-bits par n-bits non signés basé sur des opérations d'addition et de décalage. Un tel algorithme nécessitera n itérations (au rythme d'une entrée d'horloge) mais la surface du multiplieur sera réduite.

Le circuit à réaliser a l'aspect suivant :

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

ENTITY multiplieur IS
  GENERIC (
    Nbr_bits : natural := 4);
  PORT                  (
    SIGNAL op1          : IN std_logic_vector (Nbr_bits-1 DOWNTO 0) ;   --premier nombre
    SIGNAL op2          : IN std_logic_vector (Nbr_bits-1 DOWNTO 0);    -- deuxieme nombre
    SIGNAL init         : IN std_ulogic;                -- nouveau calcul
    SIGNAL  h           : IN STD_ULOGIC;        -- horloge
    SIGNAL result       : OUT std_logic_vector(2*Nbr_bits-1 downto 0);  -- resultat
    SIGNAL fin_calcul           : OUT STD_ULOGIC);  -- indique un résultat disponible
END  multiplieur;
```

Le flot de données est spécifié par l'algorithme suivant :

```
INIT : n2 = op2 ; n1 = op1 ; accu = 0
Tant_que (n2 # 0) faire
        Si (LSB_n2 = 1) alors
                accu = accu +  n1
        Fin_si
        Décaler n1 à gauche, n2 à droite ( avec 0 en MSB)
Fin_tant_que
```

4.6.4.1 Opérateurs :

Une analyse de l'algorithme permet de déduire immédiatement qu'il nécessite :

- Un accumulateur 2n-bits pour la variable résultat (avec remise à zéro)
- Un registre à décalage gauche n-bits pour la variable n1 (avec chargement parallèle)
- Un registre à décalage droite n-bits pour la variable n2 (avec chargement parallèle)

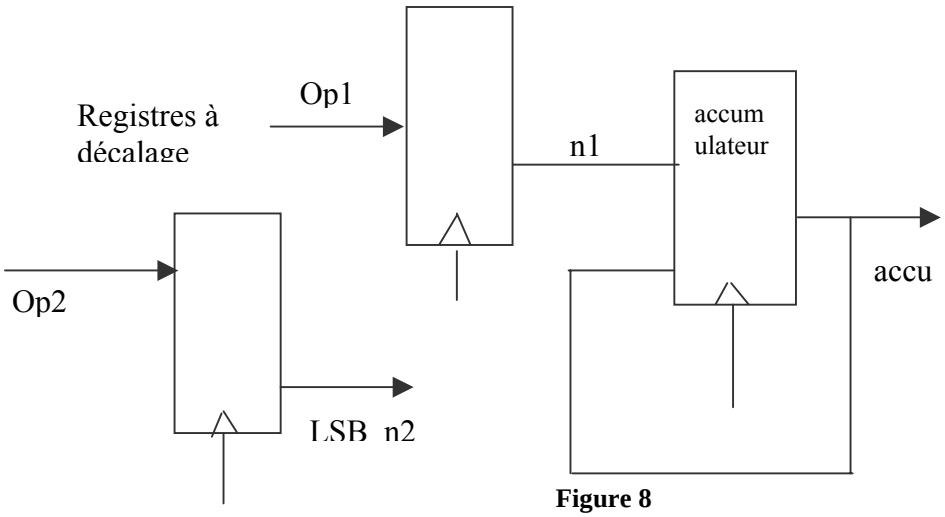


Figure 8

Ceci conduit à créer 3 signaux de commande et 2 signaux indicateurs pour ces opérateurs

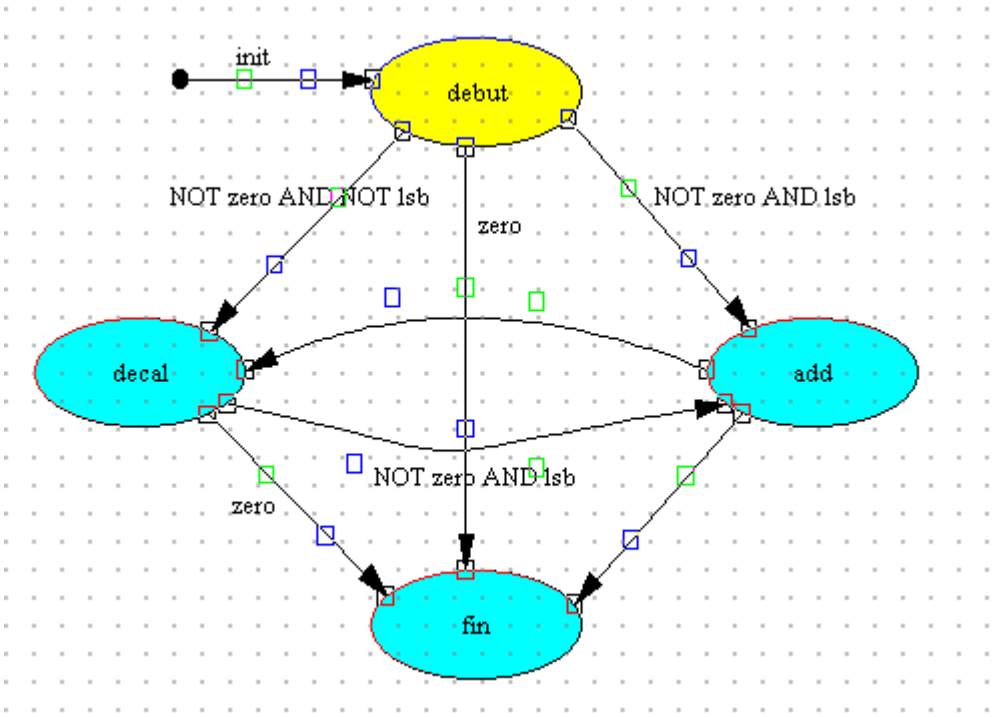
- Signal *accumuler* commande l'ouverture de l'accumulateur
- Signal *initialiser* commande la remise à zéro de l'accumulateur
- Signal *décaler* commun aux deux registres à décalage
- Signal *lsb_n2* est le bit de poids faible de n2
- Signal *zero* décodage de la valeur 0 sur l'ensemble des bits de n2

Il est alors très facile de produire le code RTL pour cette partie.

```
ARCHITECTURE rtl OF multiplieur IS
        SIGNAL n1 : unsigned(2*Nbr_bits-1 DOWNTO 0);
        SIGNAL n2 : unsigned(Nbr_bits-1 DOWNTO 0);
        SIGNAL accu : unsigned(2*Nbr_bits-1 DOWNTO 0);
        SIGNAL additionner, decaler, initialiser, zero, lsb : BOOLEAN;
-- suite des déclarations
BEGIN


  operateurs: PROCESS
        BEGIN                           ------ sur front montant
        WAIT UNTIL rising_edge(h);
                IF initialiser THEN
                        n2 <= unsigned(op2);
                        n1 <=  resize(unsigned(op1),n1'length);
                        accu <= (OTHERS => '0');
                ELSIF additionner THEN
                        accu <= accu + n1;
                ELSIF decaler THEN
                        n2 <= '0' & n2(Nbr_bits-1  downto 1);
                        n1  <= n1(2*Nbr_bits-2 downto 0) & '0';
                END IF;
        END PROCESS operateurs;

        result <= std_logic_vector(accu);
        zero <= (n2 = 0);
        lsb <= (n2(0) = '1');
        -- suite des process
```

4.6.4.2 Séquenceur

Le séquenceur a donc comme entrées (conditions de transition) les signaux *zero , lsb et init* (reset global) et comme sorties les ordres *initialiser, décaler et additionner*. On le traite comme une machine de Moore.




```
-- autres déclarations
 TYPE type_etat IS (debut,add,decal,fin);
  SIGNAL etat : type_etat;
BEGIN
 -- autre process (opérateurs)

    sequenceur:PROCESS
          BEGIN
          WAIT UNTIL falling_edge(h);      -- synchrone sur front montant
                IF (init = '1') THEN
                      etat <= debut;
                ELSE
                  CASE etat IS
                      WHEN debut =>IF zero THEN etat <= fin;
                                   ELSIF lsb THEN etat <= add;
                                   ELSE etat <= dec;
                                   END IF;
                      WHEN add => etat <= dec;
                      WHEN dec => IF zero THEN etat <= fin;
                                  ELSIF lsb THEN etat <= add;
                                  END IF;
                      WHEN fin => etat <= fin;
                  END CASE;
                END IF;
          END PROCESS sequenceur ;
```

4.6.4.3 Sorties du séquenceur

```
initialiser <= (etat = debut);
additionner <= (etat = add);
```

```
decaler <= (etat = dec);
 fin_calcul <= '1' WHEN etat = fin ELSE '0';

END rtl;
```

4.7 Evaluation des performances temporelles d'un système synchrone

Les retards associés à l'aspect physique des circuits numériques constituent une limitation temporelle pour les fonctions réalisées. Si l'on veut agir en vue d'optimiser les performances (surface – vitesse d'exécution) d'un circuit, il est donc essentiel de bien comprendre où se situent ses limitations.

Dans le cas le plus général d'un chemin de données limité par deux registres synchrones, pour les circuits combinatoires placés entre ces registres, il existe un grand nombre de chemins possibles entre une entrée D du deuxième registre et une sortie Q du premier registre. Toutes les bascules constituant les registres ayant le même signal d'horloge, tous les chemins devront vérifier la relation simple :

Période d'horloge > temps de propagation bascule + retards combinatoires + Tsetup bascule

ainsi que l'illustre la Figure 9

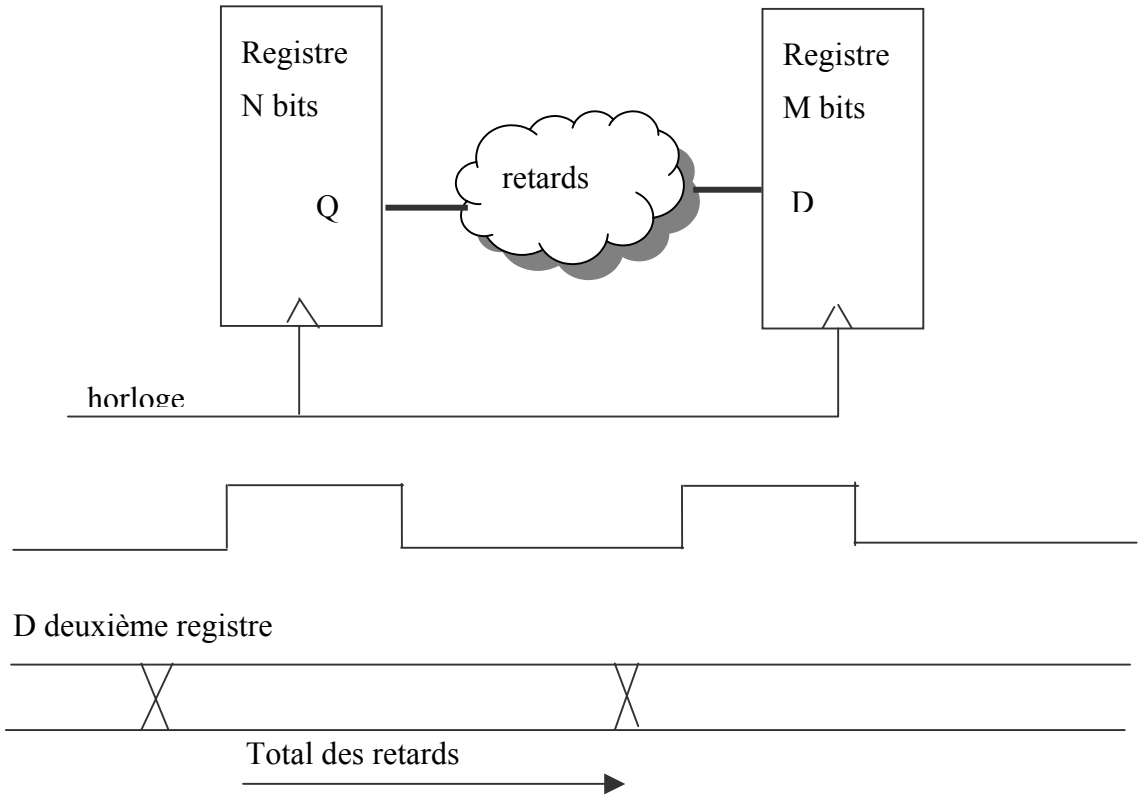


Figure 9 : Contrainte sur la période d'horloge

On constate à l'évidence que c'est le chemin le plus long qui limite la performance en vitesse et connaître ce chemin, c'est savoir où agir pour améliorer la performance. Une deuxième relation traduit cette réalité de chemin à retard maximum en fréquence maximum :

Fréquence maximum = 1/ (retard maximum + Tsetup)

Dans cette deuxième relation, le temps de propagation de la première bascule a été comptabilisé.

Le synthétiseur (VHDL ou Verilog) est fort utile et efficace pour faire une analyse exhaustive de tous les chemins possibles entre registres. Il fournira toutes les informations détaillées sur le chemin limitant et la fréquence maximum théorique associée. Si les circuits élémentaires entrant dans la synthèse sont correctement modélisés, tout un ensemble de paramètres électriques seront pris en compte dans cette évaluation.

Origine des retards :

- Retard intrinsèque : constant pour une porte donnée
- Retard dû à la pente (slope delay) : retard additionnel
- Retard de transition : en fonction de la charge, le temps de montée mesuré de 10% à 90% du signal va varier et introduire éventuellement un retard supplémentaire ;
- Retard dû aux interconnexions : retard provenant des résistances et capacités des interconnexions. Cela correspond à l'intervalle de temps entre la sortie qui à commutée et l'entrée correspondante qui commute.

4.8 Procédés de communication asynchrone

Pour les circuits synchrones, un problème très important et qui peut s'avérer assez délicat est celui de leur interfaçage lorsqu'il est asynchrone. Deux circuits synchrones fonctionnent à des vitesses différentes et doivent échanger des données. Le théorème de Nyquist énonce une certaine évidence en disant que le débit de données pris en compte par le récepteur doit être au moins aussi important que celui des données fournies par l'émetteur.

On ne veut perdre aucune donnée dans l'échange. Le protocole d'échange le plus simple est appelé « handshake » ou « poignée de main ». Il nécessite deux signaux de contrôle *donnée_prête* provenant de l'émetteur et *donnée_reçue* provenant du récepteur. Le premier signal constitue une requête et le deuxième l'acquittement. Le protocole peut être illustré par un chronogramme (Figure 11)

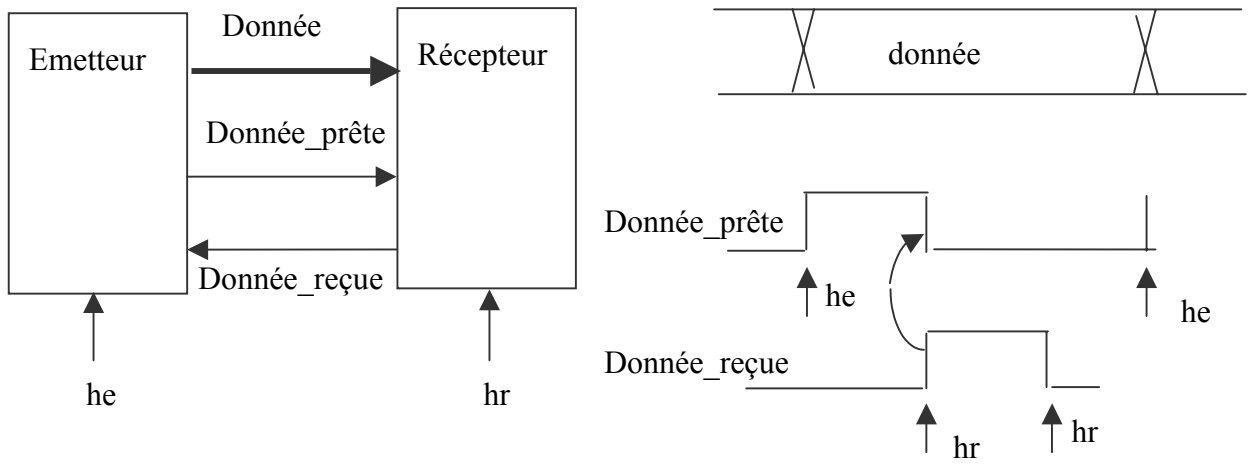


Figure 10 : handshake

Le circuit émetteur synchrone de l'horloge *he* positionne une requête *donnée_prêt*e en même temps qu 'une nouvelle *donnée*. Le circuit récepteur, synchrone de l'horloge *hr* (qui peut être plus lente ou plus rapide que *he*), va fournir un acquittement *donnée_reçue* au moment de son acquisition de *donnée*. Cet acquittement doit remettre à zéro de façon asynchrone et donc immédiate le signal *donnée_prête*. Le circuit émetteur peut alors émettre la donnée suivante. On constate assez vite que par ce procédé, le débit des données va être réglé par la vitesse du circuit le plus lent et qu'il n'y aura pas de perte de donnée à condition, bien évidemment que l'émetteur soit capable de ralentir l'envoie des données en fonction des acquittements successifs.

4.8.1 Description VHDL

Le fonctionnement du handshake peut être décrit par un Process coté émetteur :

```
Emetteur :PROCESS(he, donnée_prête)
BEGIN     -- desciption partiellement asynchrone
          IF donnee_reçue = '1' THEN
                Donnée_prête <= '0';
          ELSIF rising_edge(he) THEN
                IF donnee_valide = '1' THEN
                Donnee_prête <= '1';
                END IF ;
          END IF;
END PROCESS ;
```

Avec

```
Donnée_valide <= NOT(donnée_reçue) AND nouvelle_donnée ;
```

Puis de l'autre coté :

```
Recepteur : PROCESS
BEGIN     --tout synchrone
          WAIT UNTIL rising_edge(hr);
          IF donnee_prête = '1' AND je_lis THEN
                Var_e := donnée ;
          Donnee_reçue <= '1' ;
          ELSE
                Donnee_reçue <= '0' ;
          END IF ;
END PROCESS ;
```

Ce système simple implante une bascule D avec raz asynchrone coté émetteur et une simple bascule D coté récepteur.

Tel qu'il est décrit, on en voit les limites. En particulier, le signal donnée_prête ne doit pas passer à '1' trop vite après être passé à zéro (moins d'une période de hr).

Même si on peut l'améliorer (en le compliquant), il assure cependant correctement en grande partie la fonctionnalité recherchée.

4.9 Les Bus

Les bus s'implantent facilement à condition d'utiliser un type avec fonction de résolution pour les signaux devant y être connectés.. Le type **std_logic** pour un seul bit et le type **std_logic_vector** pour un paquet de fils sont définis ainsi dans la bibliothèque std_logic_1164 :

```
TYPE std_logic IS resolved std_ulogic;
TYPE std_logic_vector IS ARRAY (natural RANGE <> ) OF std_logic;
```

Les types **unsigned** et **signed** de la bibliothèque numeric_std ont exactement la même définition que std_logic_vector et ont le même usage.

Le mot clef « bus » au niveau d'un signal doit être banni en synthèse.

4.9.1 Bus unidirectionnel (buffer trois-états)

Voici un exemple simple d'écriture en vue de synthèse.

```
ENTITY bus_trois_etat IS
PORT (periph_a, periph_b : IN std_logic_vector (7 DOWNTO 0);
        cs_a, cs_b : IN  std_logic;
        sortie_commune : OUT std_logic_vector );
END ;
ARCHITECTURE synthetisable OF bus_trois_etat IS
BEGIN
        sortie_commune <= periph_a WHEN cs_a = '1'
                          ELSE  "ZZZZZZZZ";

        sortie_commune <= periph_b WHEN cs_b = '1'
                          ELSE  "ZZZZZZZZ";
END;
```

4.9.2 BUS bidirectionnel ;

Pour l'implanter il faut disposer d'un signal avec un mode **inout** qui pourra être et lu et écrit.

```
ENTITY bus_bidirectionnel IS
PORT(  liaison : INOUT std_logic ;  -- un seul fil
        Direction : IN std_logic;
        …
        );
END;

ARCHITECTURE synthetisable OF  bus_bidirectionnel IS
        SIGNAL  signal_interne, entree_interne : std_logic ;
BEGIN
        Liaison <= signal_interne WHEN direction = '1' ELSE 'Z'
```

```
                Entree_interne <= liaison ;
                …
END ;
```

5 La modélisation

5.1 Introduction

Un modèle est une description valide pour la **simulation**. En modélisation tout le langage VHDL est donc utilisable. Une description synthétisable reste donc aussi un modèle.

Modélisation et synthèse sont tout à fait complémentaires. En effet, lors de la conception d'un circuit, il faut être capable de le tester dans un environnement aussi complet que possible. On a donc besoin pour cela de modèles comportementaux comportant des aspects fonctionnel et temporel. Tel est la cas pour les générateurs de stimuli, les comparateurs de sorties, les modèles de mémoires RAM ou ROM ou tout autre circuit décrit au niveau comportemental.

5.2 Modélisation des retards

5.2.1 Les moyens de pris en compte

- Affectation de signal: s <= a AFTER Tp;
- Instruction FOR: WAIT FOR delay1; -- affecte tous les signaux du Processus
- Fonction NOW: retourne l'heure actuelle de simulation. Limitée au contexte séquentiel
- Attributs définis fonctions du temps: S'delayed[(T)], S'Stable [(T)], S'Quiet [(T)].
- Bibliothèques VITAL

5.2.2 Transport ou inertiel ?

Deux types de retards sont à considérer : Retard *inertiel* (par défaut) ou retard *Transport*. Toute sortie affectée d'un retard *inertiel* filtre toute impulsion de largeur inférieure à ce retard tandis qu'un retard de type *transport* est un retard pur pour le signal auquel il est appliqué.

5.3 Modèle de RAM

```
-- Fichier          :   cy7c150.vhd
-- Date de creation  :   Sep 98
-- Contenu          :   RAM CYPRESS avec e/s separees
-- Synthetisable ?   :   Non
--------------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE ieee.numeric_std.ALL;


ENTITY cy7c150 IS
                              PORT ( a : IN unsigned( 9 DOWNTO 0);
                                   d : IN unsigned( 3 DOWNTO 0);
                                   o : OUT unsigned( 3 DOWNTO 0);
                                   rs_l, cs_l, oe_l, we_l : IN std_logic);
END cy7c150;
-- modele sans-retard d'apres catalogue CYPRESS
```

```
ARCHITECTURE sans_retard OF cy7c150 IS
                        TYPE matrice IS ARRAY (0 TO 1023 ) OF unsigned( 3 DOWNTO 0);
                        SIGNAL craz , csort, cecri : BOOLEAN;
        SIGNAL contenu : matrice;
BEGIN
                        craz <= ( rs_l = '0' AND cs_l = '0');
                        csort <= ( cs_l = '0' AND rs_l = '1' AND oe_l = '0' AND we_l = '1');
                        cecri <= ( cs_l = '0' AND we_l = '0');

ecriture:PROCESS
                        BEGIN
                                WAIT ON craz, cecri;
                                IF craz THEN
                                   FOR i IN matrice'RANGE LOOP
                                        contenu(i) <= "0000";
                                   END LOOP;
                                ELSIF cecri THEN
                                        contenu(to_integer('0'& a)) <= d ;
                                ELSE
                                         o <= unsigned(contenu(to_integer('0'& a)));
                                END IF;
                        END PROCESS;
-- attention , il faut rajouter '0' en bit de signe de a afin que l'entier correspondant soit positif
                END;
```

5.4 Lecture de Fichier

On doit fournir à un circuit de traitement d'images des pixels extraits d'un fichier image noir et blanc au format PGM. Le format de ce fichier est défini ainsi :

```
P2
#CREATOR XV
512 512
255
155 155 154 154 150…
```

P2 est l'indicateur du format. La ligne suivante est une ligne de commentaire, 512 est le nombre de points par ligne, 512 est le nombre de lignes d'une image et 255 est la valeur maximale du pixel dans l'échelle de gis (codé sur 8 bits). Viennent ensuite la succession des 262144 pixels. C'est un exemple de création de stimuli complexes par lecture de données dans un fichier. On se sert du package Textio et en particulier des procédures READLINE ET READ permettant de parcourir séquentiellement un fichier ASCII.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

LIBRARY std;
USE std.textio.ALL;

ENTITY camerapgm IS
```

```
 GENERIC (nom_fichier : string := "lena.pgm";
        en_tete          : IN  string := "P2";
        largeur          : IN  natural := 512;
        hauteur          : IN  natural := 512 ;
        couleur_max      : IN  natural := 255);
 PORT (
   pixel   : OUT unsigned( 7  DOWNTO 0 ) ;  -- pixels
    h      : IN std_logic);       -- horloge pour appeller la sortie

END camerapgm ;

ARCHITECTURE sans_tableau OF camerapgm IS


BEGIN  -- sans_tableau

 lecture  : PROCESS
   FILE fichier : text IS IN nom_fichier;
   VARIABLE index : natural := 0;
   VARIABLE ligne : line;
   VARIABLE donnee : natural;
   VARIABLE entete : string (1 TO 2);  -- on attend P3 ou P2
   VARIABLE nx : natural := 15;             -- nombre de caracteres par ligne
   VARIABLE ny : natural;              --  nombre de lignes;
   VARIABLE max : natural;             -- intensite max
   VARIABLE bon : boolean;
   CONSTANT nbr_pixels  : natural := largeur * hauteur ;  --  512 * 512 = 262144

 BEGIN  -- PROCESS charger

   ASSERT false REPORT "debut de lecture du fichier " SEVERITY note;
   readline(fichier, ligne);           -- ouverture du fichier image
   read(ligne,entete,bon);
   ASSERT bon REPORT "l'en-tete n'est pas trouvee" SEVERITY error;
   ASSERT entete = en_tete REPORT "en-tete incorrecte" SEVERITY error ;
   readline(fichier, ligne);        -- 1 ligne de commentaire
   readline(fichier, ligne);
   read(ligne,nx,bon);
   ASSERT bon REPORT "erreur de format sur nx" SEVERITY error;
   ASSERT nx = largeur REPORT "Dimension de  ligne incorrect" SEVERITY error;
   read(ligne,ny,bon);
   ASSERT bon REPORT "erreur de format sur ny" SEVERITY error;
   ASSERT ny = hauteur REPORT "Nombre de lignes  incorrect" SEVERITY error;
   readline(fichier, ligne);
   read(ligne,max,bon);
   ASSERT bon REPORT "erreur de format sur max" SEVERITY error;
   ASSERT max = couleur_max REPORT "niveau max de couleur incorrect" SEVERITY error;
   index := 0;
    l1: WHILE NOT ENDFILE(fichier) LOOP
```

```
    readline(fichier, ligne);
    read(ligne,donnee, bon);
    WHILE bon  LOOP
      WAIT UNTIL rising_edge(h);        -- pour synchroniser
      index := index + 1;
      pixel <= to_unsigned(donnee,8);
      IF index = nbr_pixels /4   THEN
        ASSERT false  REPORT "Patience ... plus que 75%" SEVERITY note;
      END IF;
      IF  index = nbr_pixels /2  THEN
        ASSERT false  REPORT "Patience ... plus que 50%" SEVERITY note;
      END IF;
      IF  index = nbr_pixels *3 /4  THEN
        ASSERT false  REPORT "Patience ... plus que 25%" SEVERITY note;
      END IF;
      read(ligne,donnee, bon);
    END LOOP;
  END LOOP;
  ASSERT false REPORT "Ouf !!, c'est fini" SEVERITY note;
  ASSERT index = nbr_pixels REPORT "Nombre de pixels incorrect" SEVERITY error;
  WAIT;
END PROCESS;


END sans_tableau;
```

5.5 Les bibliothèques VITAL

VITAL (VHDL Initiative Towards ASIC Libraries) est un organisme regroupant des industriels qui s'est donné pour objectif d'accélérer le développement de bibliothèques de modèles de cellules pour la simulation dans un environnement VHDL (depuis 1992).

La spécification du standard de modélisation fixe les types , les méthodes, le style d'écriture efficace pour implémenter les problèmes de retards liés à la technologie. L'efficacité est jauchée selon la hiérarchie suivante :

- Précision des relations temporelles
- Maintenabilité du modèle
- Performance de la simulation

La plupart des outils de CAO utilisent les bibliothèques VITAL au niveau technologique.

5.5.1 Niveaux de modélisation et conformité

Une cellule est représentée par un couple entity/architecture VHDL. La spécification définie trois niveaux de modélisation correspondant à des règles précises.

- Vital Level 0
- Vital Level 1
- Vital Level 1 Memory

5.5.2 Les packages standards

5.5.2.1 VITAL_Timing

Il définie les types de données et les sous-programmes de modélisation des relations temporelles . Sélection des retards, pilotage des sorties, violation de timing (ex: Setup, Hold) et messages associés.

5.5.2.2 Vital_Primitives

Il définie un jeu de primitives combinatoires d'usage courant et des tables de vérité

5.5.2.3 Vital_Memory

Spécifique pour modélisation des mémoires

6 Bibliographie

[1] *VHDL: Du langage à la modélisation*. R.Airiau, J.M.Bergé, V.Olive et J.Rouillard – CENT

[2] *VHDL Designer's Reference*, J.M. Bergé, A.Fonkoua, S.Maginot et J.Rouillard. Kluwer Academic Publishers.

[3] Reuse Methodology Manual For System On Chip Designs, Michael Keating and Pierre Bricaud – Kluwer Academic Publishers 1999

[4] Conception des Asics, P.Naish, P.Bishop – Masson 1990

[5] Architecture des ordinateurs , une approche quantitative, John L.Hennessy et David A.Patterson

[6] VHDL Du langage au circuit, du circuit au langage par J.Weber et M.Meaudre - Masson

[7] http://www.eda.org

[8] Newsgoup : comp.lang.vhdl

[9] Documentations Mentor Graphics