Langage VHDL et conception de circuits

1ère partie: Le langage VHDL

2ème partie: Synthèse VHDL

Patrice NOUEL

http://vhdl33.free.fr

Révision : Mai 2009

TABLE DES MATIERES

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 [1] . 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.

2 Méthodologie

2.1 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 (Figure 1) . 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.

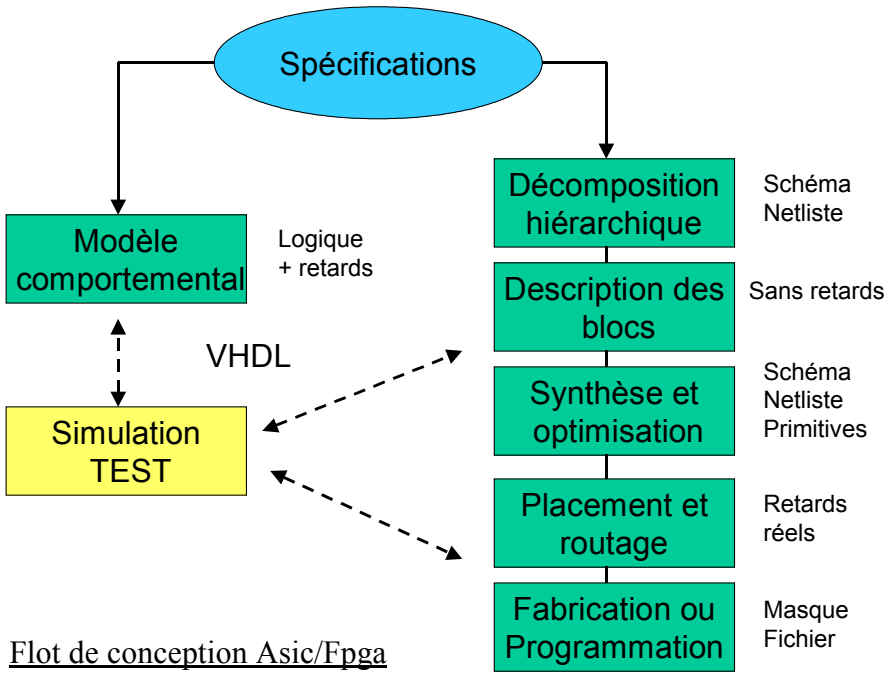


Figure 1 : Flot de conception type « waterfall »

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.

2.2 La synthèse

2.2.1 Fonction

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.

2.3 Saisie du code

2.3.1 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.

2.3.2 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

2.3.3 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.

2.3.4 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 et prenant raz (ici synchrone, voir la version asynchrone au paragraphe 4.3) 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;
```

2.4 La cible technologique

2.4.1 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

1. Peut être cher pour de grandes séries
2. Consommation globale accrue par les circuits de configuration
3. 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.

2.4.2 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.

2.4.3 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.

2.4.4 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

2.5 Le placement routage

Le défaut du travail de synthèse classique se situe au niveau de l'évaluation réelle de la performance temporelle. En effet, le synthétiseur se contente de sommer des retards nominaux alors que l'implantation physique dans le silicium des primitives apporte des retards supplémentaires dus à la longueur des connexions. En technologie CMOS la vitesse peut parfaitement être divisée par 2 ou 3 si le routage n'est pas fait soigneusement.

Dans tous les cas on a besoin de la connaissance du fondeur de silicium pour cette évaluation et on utilisera toujours le kit de développement matériel propriétaire. Ceci permettra en particulier d'obtenir pour le projet finalisé un modèle VHDL post-routage avec retards.

2.6 La simulation (Modelsim)

Le simulateur est l'outil permettant de valider les différentes étapes de la conception descendante. On traduira par une modélisation VHDL tout l'environnement du circuit. Le Test Bench est ainsi une véritable maquette de test du circuit. Celui-ci, au cours de sa conception garde sa spécification d'entité et voit son architecture s'affiner du comportemental jusqu'au structurel du niveau porte.

De façon tout à fait standard, un test bench est une entité fermée :

```
ENTITY mon_test_bench is END ;
```

Dans l'architecture associée, on instancie le composant à tester , ses générateurs de stimuli, les fonctions d'observation nécessaires.

```
ARCHITECTURE tb OF mon_test_bench IS
        COMPONENT dut
                PORT(
        END COMPONENT
        COMPONENT generateur
                PORT (…..
        END COMPONENT
        ……
```

Pour la simulation post-routage, on dispose d'un modèle structurel du circuit. Ce modèle est généré automatiquement par l'outil de placement-routage et utilise pour les fonctions de vérifications temporelles la bibliothèque VITAL. Ainsi pourront être signalées toutes violations de temps de pré conditionnement et de maintien des bascules. Les retards sont, quant à eux, stockés dans des fichiers au format **sdf** (Standard delay file) que l'on pourra facilement associer à la simulation. Il est donc possible d'effectuer une simulation post-routage sans retard ou avec retard.

3 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.

3.1 Multiplexeurs, démultiplexeurs, décodeurs

3.1.1 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 ;
```

3.1.2 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;
```

3.1.3 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;
```

3.2 Circuits arithmétiques

3.2.1 Représentations des nombres en virgule fixe

3.2.1.1 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.

3.2.1.2 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 :

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

3.2.2 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.
Dautres 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.

3.2.3 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

3.2.4 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.

3.2.5 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 .
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 5.3).

3.2.6 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 Conception synchrone

4.1 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**.

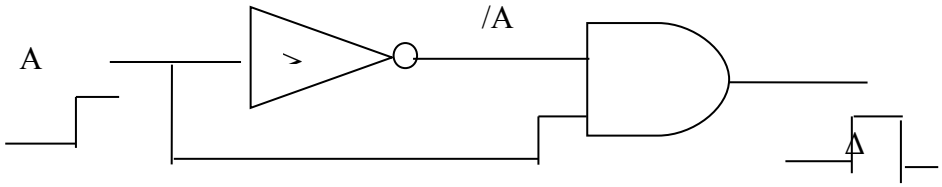


Figure 2

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**.

4.2 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
.

4.3 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.

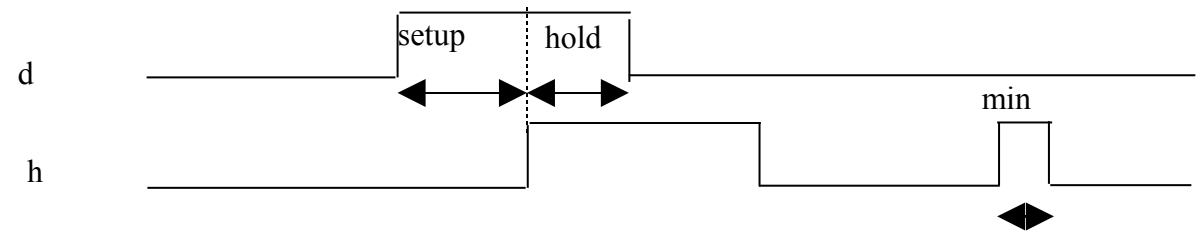


Figure 3

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.

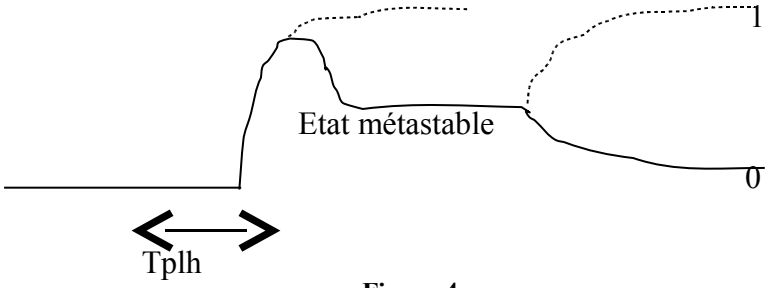


Figure 4

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.

4.4 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.

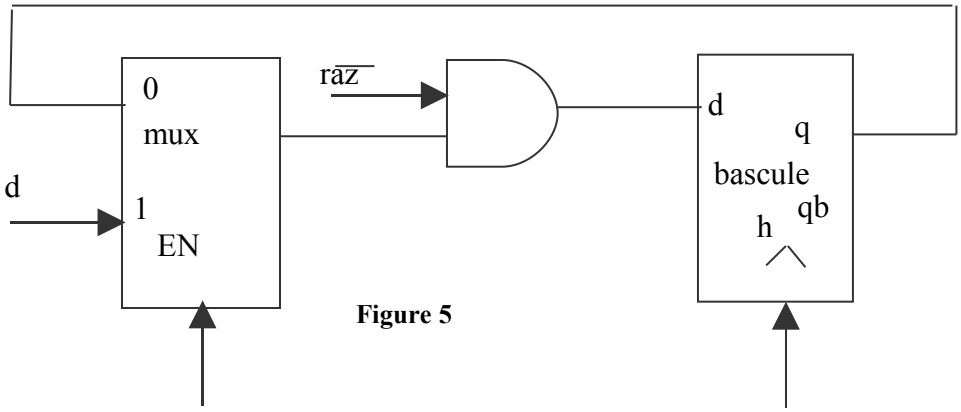


Figure 5

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

4.5 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;-
```

4.6 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.

4.6.1 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 unsigned (Nb_bits-1 DOWNTO 0);
      retenue :  OUT std_ulogic);
BEGIN – contrôle des paramètres
 ASSERT Modulo <= 2**Nb_bits
                  REPORT "erreur sur les parametres"
                  SEVERITY warning;
END compteur_gen;

ARCHITECTURE synchrone OF compteur_gen IS
BEGIN
 un_seul: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;
```

```
      retenue <= '0' ;
    ELSE
      c := 0;
      retenue <= '1';
    END IF;
  END IF;
  sortie <= to_unsigned(c, Nb_bits);
 END PROCESS;
END synchrone;
```

4.6.2 Compteur générique asynchrone

```
ARCHITECTURE asynchrone OF compteur_gen IS
BEGIN
 un_seul:PROCESS
  VARIABLE c : natural range 0 TO Modulo -1;
 BEGIN
  WAIT ON raz, h;
  IF raz = '1' THEN
   c :=0;     -- asynchrone
  ELSIF  RISING_EDGE(h) THEN
   IF  compter = '1' THEN
    IF c < Modulo -1 THEN
     c := c + 1;   --synchrone
     retenue <= '0' ;
    ELSE
     c := 0;
     retenue <= '1';
    END IF;
   END IF;
  END IF ;
  sortie <= to_unsigned(c, Nb_bits);
 END PROCESS;
END asynchrone;
```

4.6.3 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 UNSIGNED (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 : UNSIGNED(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 s < To_unsigned(M,N)
      REPORT "sortie hors intervalle";
    ASSERT (s - s0) < "00010"
      REPORT "on doit etre en fin de comptage";
    s0 := s;
  END PROCESS;
END  par_assertion;
```

5 Structuration d'un circuit synchrone

5.1 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

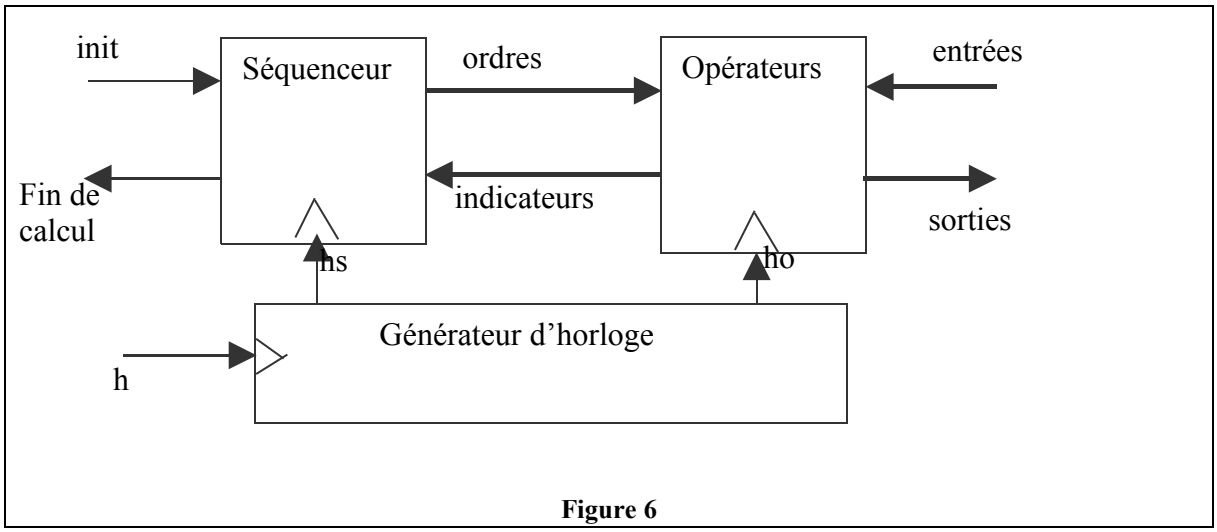
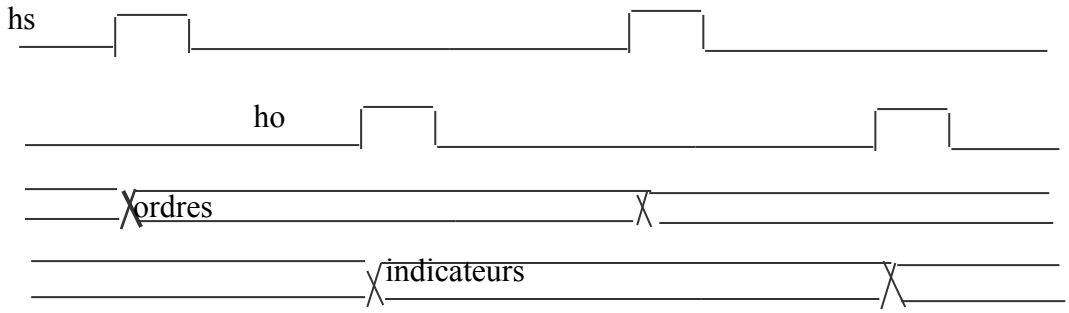


Figure 6

5.2 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.



5.3 Machines d'états finis

5.3.1 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.

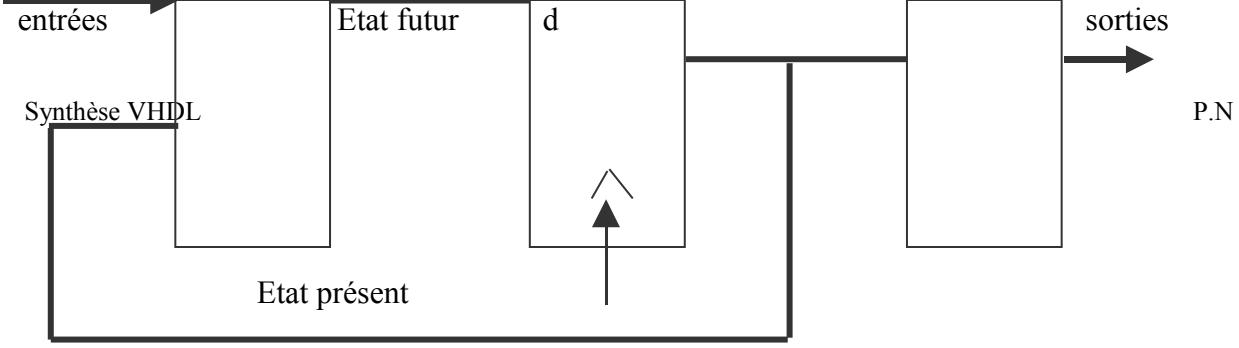


Figure 7 : machine de Moore synchrone

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).

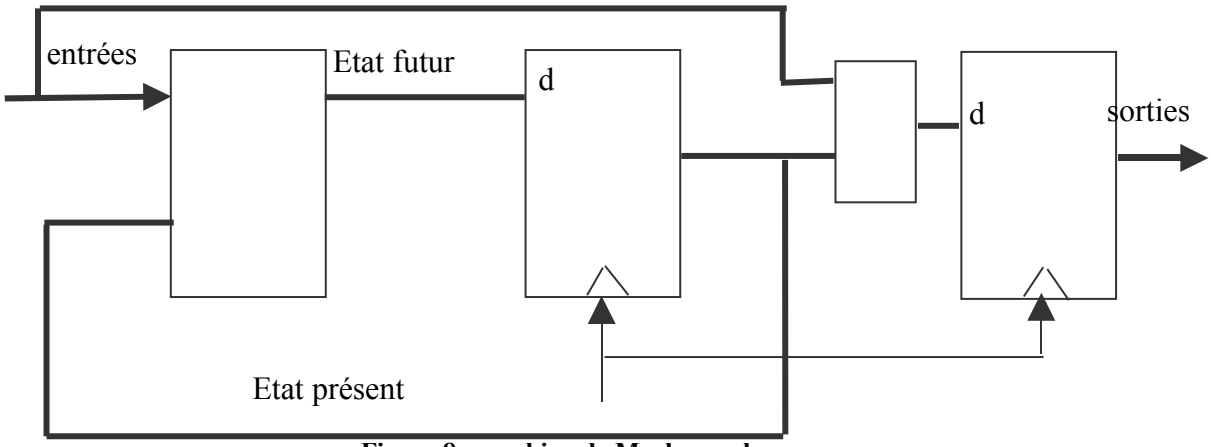


Figure 8 : machine de Mealy synchrone

.

5.3.2 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.

5.3.2.1 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;
```

5.3.2.2 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.

5.4 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.numeric_std.ALL;
USE ieee.std_logic_1164.ALL;

ENTITY multiplieur_ad IS
  GENERIC (
        nbr_bits : natural := 4);
  PORT (
        operande_1 : IN unsigned(nbr_bits-1 DOWNTO 0);
        operande_2 : IN unsigned(nbr_bits-1 DOWNTO 0);
        h          : IN std_ulogic;
        init       : IN std_ulogic;
        resultat   : OUT unsigned (2*nbr_bits-1 DOWNTO 0);
        fin_calcul : OUT std_ulogic );
END multiplieur_ad;
```

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

```
INIT : n2 = operande_2 ; n1 = operande_1 ; 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
```

5.4.1 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)

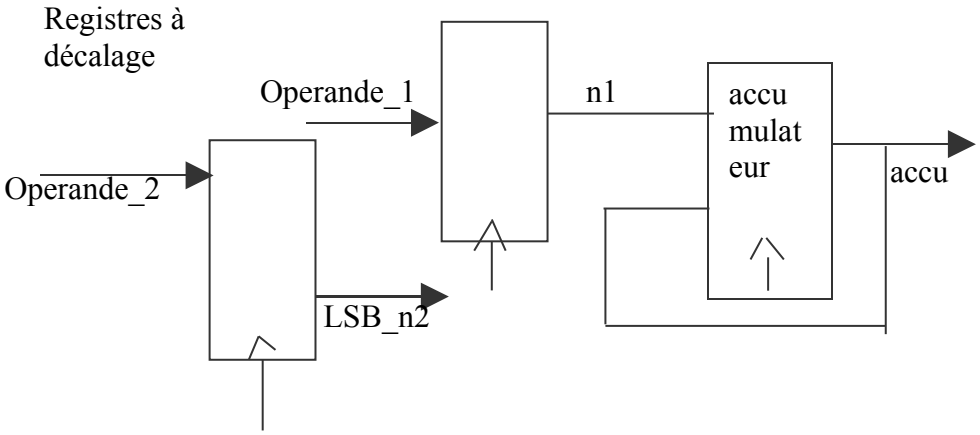


Figure 9

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.

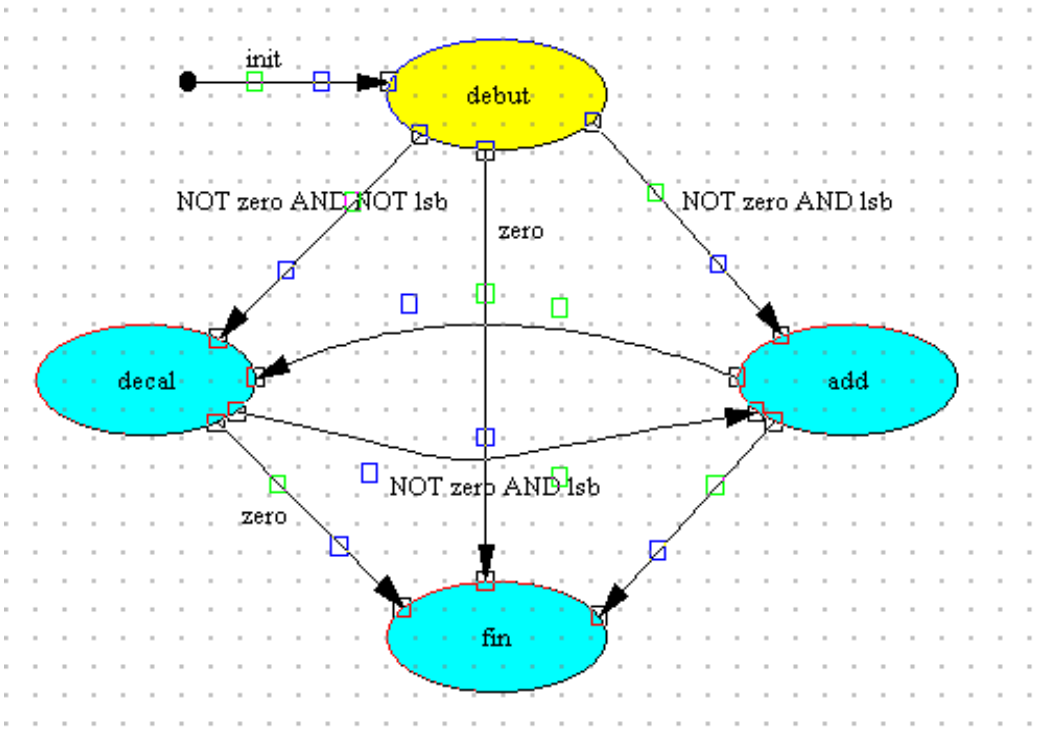
```
ARCHITECTURE rtl OF multiplieur_ad IS
  SIGNAL initialiser : boolean;
  SIGNAL accumuler : boolean;
  SIGNAL decaler : boolean;
  SIGNAL zero : boolean;                -- registre n2 vide
  SIGNAL lsb_n2 : boolean;               -- bit de poids faible n2
-- suite des déclarations
BEGIN

  operateurs: PROCESS
    VARIABLE n1 : unsigned(2*nbr_bits-1 DOWNTO 0);
    VARIABLE n2 : unsigned(nbr_bits-1 DOWNTO 0);
    VARIABLE accu : unsigned(2*nbr_bits-1 DOWNTO 0);
  BEGIN
    WAIT UNTIL falling_edge(h);
    IF initialiser THEN
          n1 := resize(operande_1,n1'length);
```

```
            n2 := operande_2;
            accu :=(OTHERS =>'0');
        END IF;
        IF accumuler THEN
            accu := accu + n1;
        END IF;
        IF decaler THEN
            n2 := shift_right(n2,1) ;
            n1 := shift_left(n1,1) ;
        END IF;
        resultat <= accu;
        zero <= (n2 = 0);
        lsb_n2 <= n2(0)= '1';
    END PROCESS operateurs;
-- suite des process
```

5.4.2 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 rising_edge(h);
    IF init = '1' THEN              -- synchrone
      etat <= debut;
    ELSE
      CASE etat IS
        WHEN debut => IF zero THEN
                        etat <= fin;
                      ELSIF lsb_n2 THEN
                        etat <= add;
                      ELSE
                        etat <= decal;
                      END IF;
        WHEN add => etat <= decal;
        WHEN decal => IF zero THEN
                        etat <= fin;
                      ELSIF lsb_n2 THEN
                        etat <= add;
                      END IF;
        WHEN OTHERS => NULL;
      END CASE;
    END IF;
  END PROCESS sequenceur;
```

5.4.3 Sorties du séquenceur

```
  initialiser <= (etat = debut);
  accumuler <= (etat = add);
  decaler <= (etat = decal);
  fin_calcul <= '1' WHEN etat=fin ELSE '0';

END deux_process;
```

6 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 10

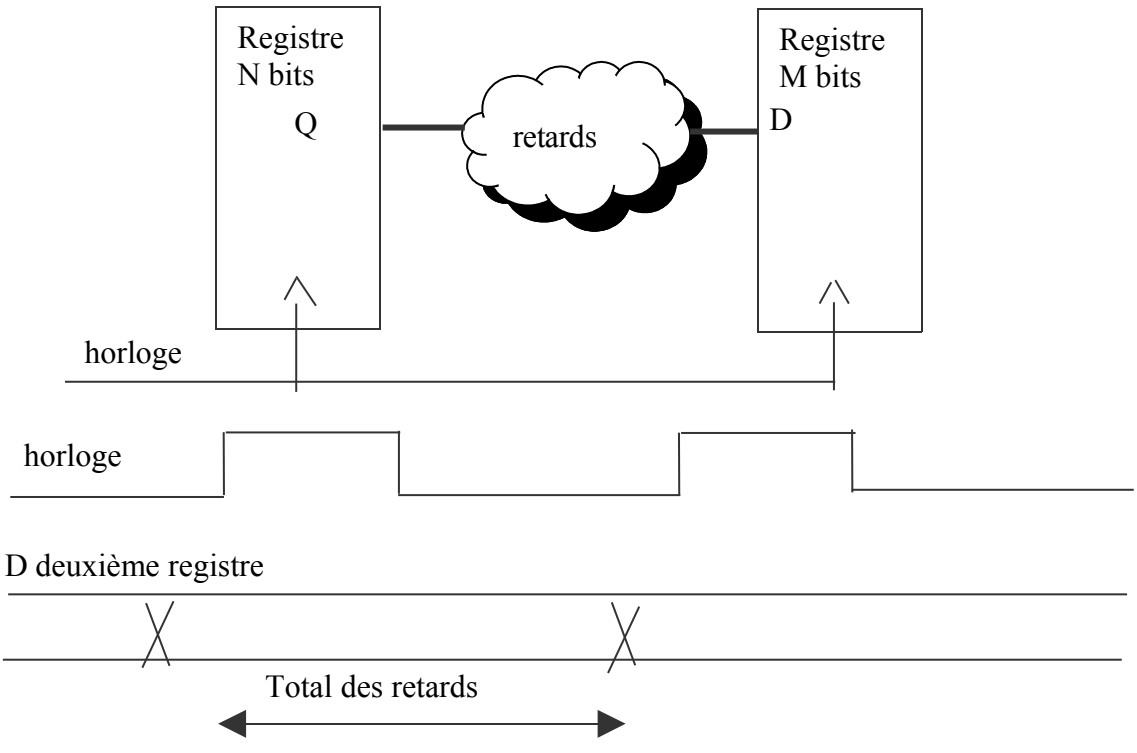


Figure 10 : Contrainte sur la période d'horloge

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.

7 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)

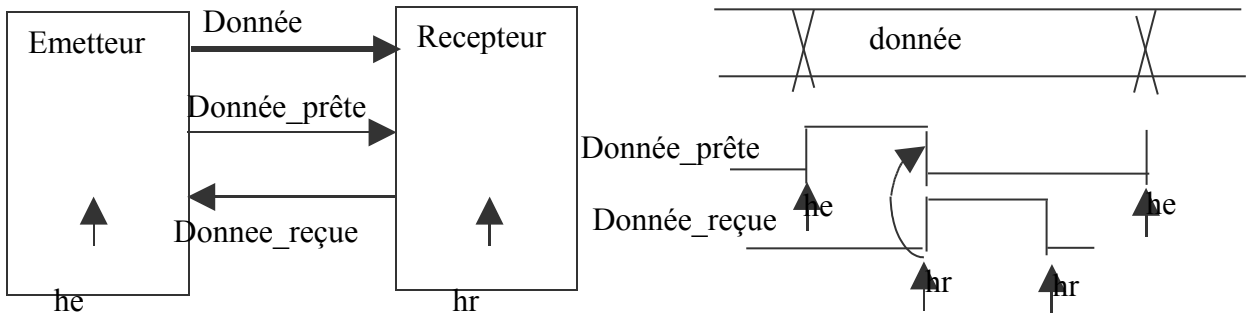


Figure 11 : handshake

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.

7.1.1 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.

8 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.

8.1 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;
```

8.2 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 ;
```

9 Bibliographie

[1] Reuse Methodology Manual For System On Chip Designs, Michael Keating and Pierre Bricaud – Kluwer Academic Publishers 1999
[2] Conception des Asics, P.Naish, P.Bishop – Masson 1990
[3] Le langage VHDL – P.Nouel , ENSEIRB
[4] Architecture des ordinateurs , une approche quantitative, John L.Hennessy et David A.Patterson
[5] http://www.eda.org