

Introduction

au

VHDL

1 INTRODUCTION

Cette introduction à VHDL n'a pas la prétention de couvrir tous les aspects de ce langage. En fait, elle suppose une connaissance préalable du langage ADA. Nous nous attarderons d'avantage sur une méthode de travail avec le VHDL que sur la grammaire de ce langage. Il existe pour cette dernière une quantité de livres dont certains sont cités dans la bibliographie en fin de manuel. De plus, pour une référence sûre, il est également possible de commander la norme VHDL 1993 (IEEE 1164).

Ce cours d'introduction se compose de trois modules :

- 1) une présentation générale du langage qui permet d'écrire très rapidement des applications simples.
- 2) une partie plus poussée sur le langage orientée vers l'écriture de programmes plus complexes.
- 3) une série d'exemples utilisables, de procédures utiles et de trucs et astuces.

1.1 BREF HISTORIQUE

Le VHDL - VHSIC (Very High Speed Integrated Circuit) **H**ardware **D**escription **L**anguage , à l'instar de l'ADA, a été demandé par le DOD (Département de la défense américain) pour décrire les circuits complexes, de manière à établir un langage commun avec ses fournisseurs. C'est un langage, standard IEEE 1076 depuis 1987, qui aurait du assurer la portabilité du code pour différents outils de travail (simulation, synthèse pour tous les circuits et tous les fabricants). Malheureusement, ce n'est pour l'instant pas le cas, bien que plusieurs vendeurs aient tendance à se rapprocher du VHDL utilisé par Synopsis. Une mise à jour du langage VHDL s'est faite en 1993 (IEEE 1164) et en 1996, la norme 1076.3 a permis de standardiser la synthèse VHDL.

1.2 UTILITÉ DU VHDL

Le VHDL est un langage de spécification, de simulation et également de conception.

Contrairement à d'autres langages (CUPL, ABEL) qui se trouvaient être en premier lieu des langages de conception, VHDL est d'abord un langage de spécification. La normalisation a d'abord eu lieu pour la spécification et la simulation (1987) et ensuite pour la synthèse (1993). Cette notion est relativement importante pour comprendre le fonctionnement du langage et son évolution. Grâce à la normalisation, on peut être certain qu'un système décrit en VHDL standard est lisible quel que soit le fabricant de circuits. Par contre, cela demande un effort important aux fabricants de circuits pour créer des compilateurs VHDL adaptés à et autant que possible optimisés pour leurs propres circuits.

1.2.1 *Spécification*

Etabli en premier lieu pour de la spécification, c'est dans ce domaine que la norme est actuellement la mieux établie. Il est tout-à-fait possible de décrire un circuit en un VHDL standard pour qu'il soit lisible de tous. Certains fabricants (de circuits ou de CAO) adaptent ce langage pour donner à l'utilisateur quelques facilités supplémentaires, au détriment de la portabilité du code. Heureusement, il y a une nette tendance de la part des fabricants à revoir leurs positions et à uniformiser le VHDL. Le rapprochement se fait, comme précité, autour du VHDL de Synopsis. Il est donc probable que l'on s'approche d'un vrai standard VHDL et non plus d'un standard théorique. Il y aura toujours des ajouts de la part des fabricants, mais il ne s'agira plus d'une modification du langage (aussi légère soit elle), mais de macros offertes à l'utilisateur pour optimiser le code VHDL en fonction du circuit cible (en vue de la synthèse).

Cette possibilité de décrire des circuits dans un langage universel est aussi très pratique pour éviter les problèmes de langue. De longues explications dans une langue peuvent ainsi être complétées par du code VHDL pour en faciliter la compréhension.

1.2.2 *Simulation*

Le VHDL est également un langage de simulation. Pour ce faire, la notion de temps, sous différentes formes, y a été introduite. Des modules, destinés uniquement à la simulation, peuvent ainsi être créés et utilisés pour valider un fonctionnement logique ou temporel du code VHDL.

La possibilité de simuler avec des programmes VHDL devrait considérablement faciliter l'écriture de tests avant la programmation du circuit et éviter ainsi de nombreux essais sur un prototype qui sont beaucoup plus coûteux et dont les erreurs sont plus difficiles à trouver.

Bien que la simulation offre de grandes facilités de test, il est toujours nécessaire de concevoir les circuits en vue des tests de fabrication, c'est-à-dire en permettant l'accès à certains signaux internes.

1.2.3 *Conception*

Le VHDL permet la conception de circuits avec une grande quantité de portes. Les circuits actuels comprennent, pour les FPGA par exemple, entre 500 et 100'000 portes et ce nombre augmente très rapidement. L'avantage d'un langage tel que celui-ci par rapport aux langages précédents de conception matérielle est comparable à l'avantage d'un langage informatique de haut niveau (Pascal, ADA, C) vis-à-vis de l'assembleur. Ce qui veut aussi dire que malgré l'évolution fulgurante de la taille des circuits, la longueur du code VHDL n'a pas suivi la même courbe. Cependant, ce langage étant conçu en premier lieu pour de la spécification, certaines variantes du langage ne sont pour l'instant pas utilisables pour la conception.

Il faut noter que lorsqu'il s'agit de concevoir quelque chose en VHDL, il ne faut pas le faire tête baissée. Le VHDL, bien que facilement accessible dans ses bases, peut devenir extrêmement compliqué s'il s'agit d'optimiser le code pour une architecture de circuit. C'est pour cette raison que de plus en plus de fabricants offrent des macros, gratuites pour les fonctions sans grandes difficultés et payantes pour les autres. Donc avant de concevoir une ALU, un processeur RISC, une interface PCI ou d'autres éléments de cette complexité, il peut être judicieux de choisir un circuit cible en fonction des besoins et d'acheter la macro offerte par le constructeur. Il est bien évident qu'il faudra évaluer les besoins (performance du code nécessaire, quantité de pièces à produire) et le coût d'une telle macro.

2 COMPARAISON AVEC ADA

La structure du VHDL a été tirée du langage ADA, ce qui implique que de nombreuses notions acquises avec l'ADA seront utilisées ici. Pour éviter de faire double emploi avec des cours sur l'ADA, seules les différences entre VHDL et ADA seront traitées ci-dessous. Cela permettra d'écrire très rapidement des programmes simples en VHDL.

A ne pas oublier :

ADA est un langage pour écrire des logiciels, VHDL est destiné à décrire, simuler et synthétiser des circuits. Il faut toujours faire attention à ce que l'on veut faire lorsque l'on programme en VHDL, sinon de mauvaises surprises peuvent apparaître.

2.1 STRUCTURE DES PROGRAMMES

La structure est celle de l'ADA. Par contre, VHDL a des particularités spécifiques à la modélisation de circuits. De nouveaux types sont définis, ainsi que de nouveaux mots réservés. Ceux-ci sont directement liés à l'aspect physique de la conception de circuits. De plus, la notion de temps intervient dans le VHDL de la même manière que dans la réalité.

2.2 LES SIGNAUX

Un nouveau mot réservé fait son apparition avec VHDL, il s'agit du mot **signal**. Il permet de désigner les signaux qui seront utilisés dans le circuit intégré. Certains types particuliers, souvent des types énumérés, sont utilisés pour un signal. Par exemple le type BIT ('0', '1'), qui est pré-défini en VHDL, ou des types plus complets comme le BIT_ETENDU ('0', '1', 'X', 'Z'). [BIT_ETENDU n'est pas un type prédéfini en VHDL]

2.3 *TYPES PHYSIQUES*

De nouveaux types, les types physiques (distance, capacité, résistance, ...), permettent de définir des valeurs concrètes et de calculer à partir de celles-ci. Certains de ces types seront donnés dans les annexes, ainsi que des exemples de calculs. Pour illustrer cela, un exemple est donné ci-dessous pour représenter les temps. Ce type est prédéfini en VHDL.

```

TYPE time IS RANGE -9_223_372_036_854_775_808 TO
                    9_223_372_036_854_775_807      -- 64 bits pour le codage

UNITS fs;          -- femtoseconde, unite de base
        ps=1000 fs;  -- picoseconde
        ns=1000 ps;  -- nanoseconde
        us=1000 ns;  -- microseconde
        ms=1000 us;  -- milliseconde
        sec=1000 ms; -- seconde
        min=60 sec;  -- minute
        hr=60 min;   -- heure

END UNITS;

```

Ce type est pré-défini à cause de la notion de temps qui a été ajoutée au VHDL. L'unité de base est la femtoseconde et toutes les autres unités sont définies à partir de celle-ci. De même les autres types physiques se définissent à partir d'une unité et on peut référencer toutes les autres à la première. Il sera donc possible de représenter, par exemple, la durée de la charge d'un condensateur qui se trouverait à l'entrée du circuit intégré, ce qui rendra les simulations de plus en plus réalistes.

2.4 *NOTION DE TEMPS*

Un ajout important du VHDL par rapport à l'ADA est la notion de temps. Si celle-ci apparaît quand même dans l'ADA, mais de manière liée à l'horloge, elle s'exprime en unités de temps sous VHDL qui pourraient tout aussi bien être convertis en pas de simulation. Cette notion permet de tenir compte des temps de retard des portes logiques ou de créer des programmes pour la simulation des circuits.

Il faut également savoir qu'à chaque assignation de signal, cette notion de temps est utilisée. Pour mieux comprendre cela, quelques exemples sont donnés ci-dessous avec leurs particularités.

```

somme <= a XOR b;

```

```
somme <= a XOR b AFTER 0ns;
```

Les deux assignements ci-dessus ont exactement la même signification. On appelle temps delta le temps nul, qu'il soit représenté ou non par le code 'after 0ns'. En effet, il est impossible pour un système d'évaluer immédiatement une sortie, par conséquent on place sur la droite de l'équation (A\$B) la valeur future qu'aura la partie gauche de l'équation (Somme), avec le temps de retard imposé par la logique nécessaire. Il est tout-à-fait possible, pour une simulation, que l'on ne désire pas tenir compte des temps de transfert des portes, on écrira alors les équations sous l'une des deux formes ci-dessus, mais nous aurons alors une simulation purement fonctionnelle. La figure 2 représente le temps de réponse de la sortie et ce qui sera affiché lors d'une simulation si l'on choisi un temps de 0ns.

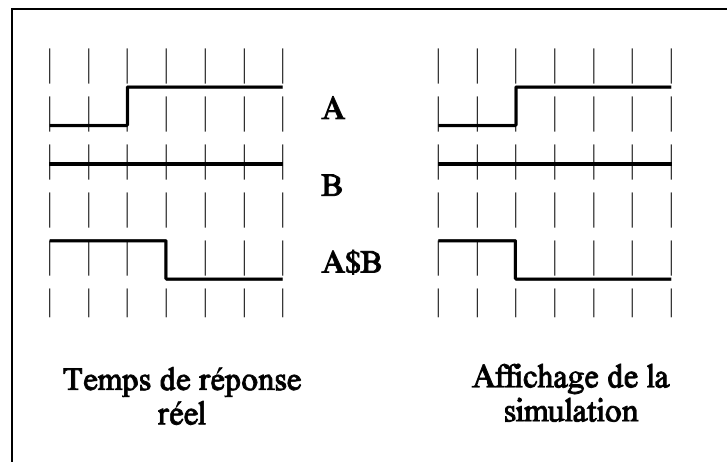


Image 2

```
somme <= a XOR b AFTER 10ns;
```

Cette équation indique que la valeur future de Somme vaut A\$B, cette modification interviendra 10ns après un changement sur l'une des deux variables.

2.4.1 Deux types de délai : inertiel et transport

Ci-dessus, nous avons vu la possibilité d'avoir un délai entre l'action sur l'entrée et l'action sur la sortie et cela sans nous occuper de quelle manière ce délai était utilisé. Il existe deux types de délais en VHDL, l'un agit soit comme un retard, soit comme un filtre (le délai inertiel) et l'autre comme un retard pur (le délai transport).

2.4.1.1 *Le délai inertiel*

Le délai inertiel est le délai défini par défaut dans VHDL. Il peut s'écrire de deux manières:

```
somme <= INERTIAL a XOR b AFTER 10ns;
somme <= a XOR b AFTER 10ns;
```

Il agit comme un filtre, c'est-à-dire que toutes les impulsions de durée inférieure à celle indiquée sont supprimées. Celles qui sont égales ou supérieures sont retardées de cette valeur. Les deux exemples ci-dessous représentent mieux cela.

Exemple 1 :

```
somme <= INERTIAL a XOR b AFTER 10ns;
```

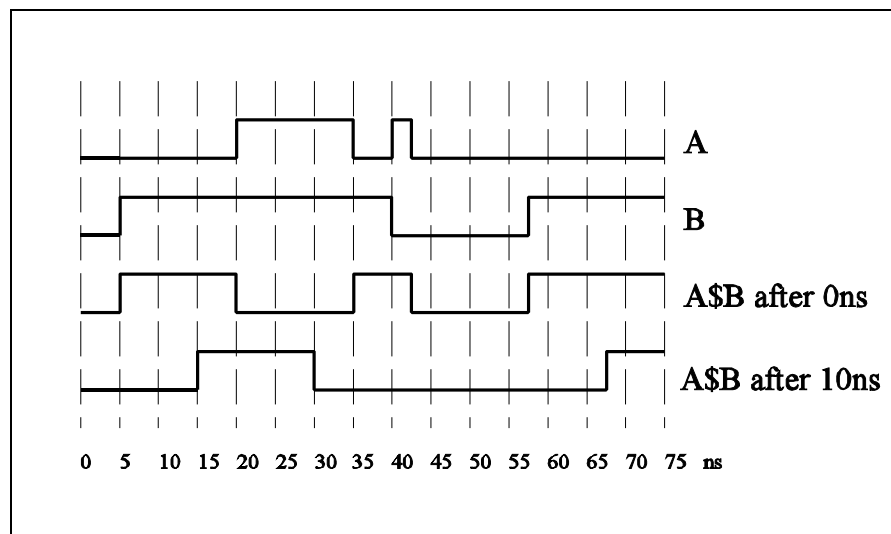


Image 3

Il est possible de voir dans cet exemple que l'impulsion qui se trouve à 35ns et qui ne dure que 7.5ns est éliminée car elle est inférieure aux 10ns spécifiées. Le reste du signal est reporté de 10ns.

Exemple 2:

```
retard <= INERTIAL signal AFTER 10ns;
```

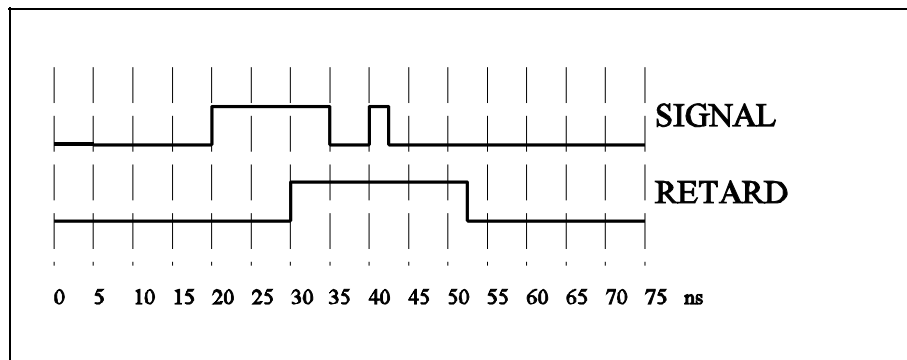



Image 4

A nouveau, cet exemple montre que l'impulsion à 35ns et ne durant que 5ns n'est pas prise en compte et laisse le signal de sortie à la valeur 1 avec toujours le même retard de 10ns. Il est impératif de lire ces diagrammes temporels de gauche à droite, sinon on commettra l'erreur de vouloir supprimer l'impulsion à '1' qui se trouve à 40ns et qui dure 2.5ns.

2.4.1.2 Le délai transport

```
somme <= TRANSPORT a XOR b AFTER 10ns;
```

Avec ce type de délai, toutes les impulsions sont transmises, mais avec le retard indiqué. Il faut toujours prendre garde à placer le mot-clé *transport* si l'on veut un retard pur sans s'occuper de filtrer des impulsions de durée inférieure à celle indiquée.

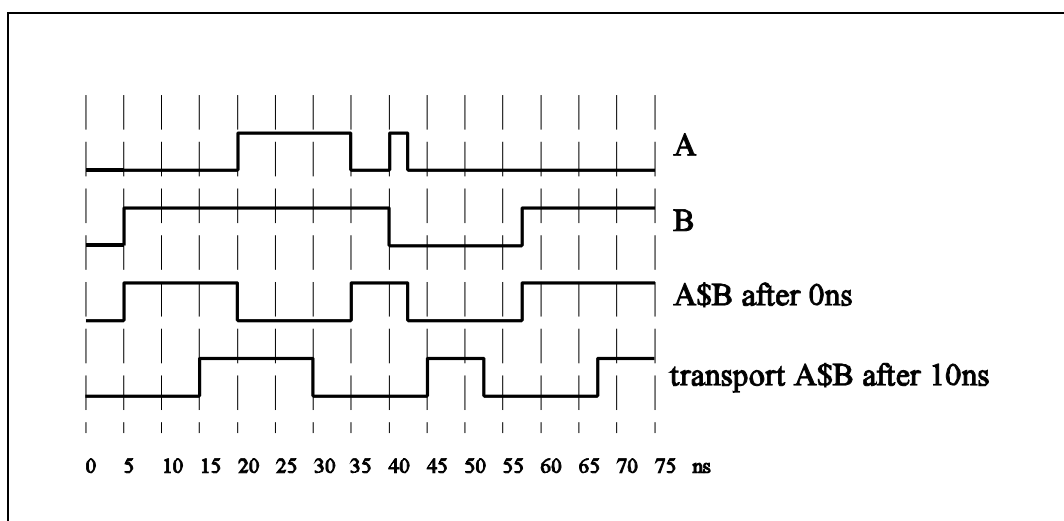


figure 4

La même impulsion à 35ns durant 7.5ns qui avait été supprimée en délai inertiel est conservée ci-dessus. Seul le retard de 10ns subsiste.

2.4.2 Utilisation du temps pour exprimer un retard

Une dernière question subsiste : pourquoi avoir défini cette notion de temps ?

Deux raisons principales à cela :

Premièrement tout circuit réel a des temps de retard et il faut bien les exprimer d'une manière ou d'une autre.

Deuxièmement, la notion de temps est nécessaire pour la simulation.

Il semble évident qu'un circuit logique génère un retard entre son entrée et sa sortie, cependant, jusqu'à présent aucun compilateur ne tenait compte de ce facteur. Etant donné que VHDL a aussi été conçu pour la simulation, il est désormais possible d'en tenir compte à la conception déjà. Le temps indiqué pour un retard sera alors utilisé par la simulation pour rendre celle-ci plus réaliste. Il est bien entendu que ces temps ne sont utilisés que pour la simulation. La synthèse n'en tiendra pas compte. Il n'est, par conséquent, pas possible de générer un signal d'horloge en VHDL.

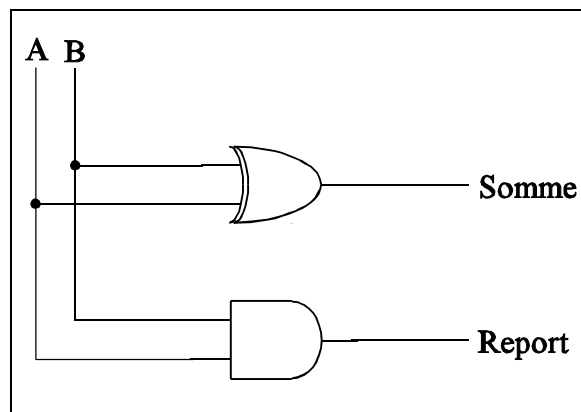


Figure 6

Sur la figure 6, qui représente un additionneur 1 bit, il y a deux portes logiques. Pour décrire un tel système, il est possible d'évaluer le retard de chacune des portes pour l'introduire dans le code VHDL de la manière suivante :

```
somme <= a XOR b AFTER 5ns;  
report <= a AND b AFTER 4ns;
```

Ce code indiquera au simulateur les temps de propagation liés aux différentes fonctions du

circuit. Par contre, il ne sera pas tenu compte de ces temps pour créer le fichier qui sera utilisé pour la programmation du circuit. Le circuit aura les retards réels générés par ses portes internes.

2.4.3 *Utilisation du temps pour un module de simulation*

VHDL permet de créer des modules qui ne servent qu'à simuler un système. Dans ces modules apparaîtront une suite de modifications des signaux d'entrée du système qui éviteront au concepteur de les entrer manuellement. Un exemple très simple est donné ci-après, mais ce point sera traité plus tard dans ce cours. Cet exemple génère un fichier d'entrée pour le schéma de la figure 6.

```
a <= '0' AFTER 0ns, '1' AFTER 100ns, '0' AFTER 200ns, '1' AFTER
300ns;
b <= '0' AFTER 0ns, '1' AFTER 200ns;
```

2.5 *ÉVÉNEMENTS ET TRANSACTIONS*

Dans un programme concurrent, c'est la "modification" d'un signal qui déclenche l'exécution du code. Deux types de modifications existent et sont définis sous les noms de *transaction* et *événement*. Une définition de ces deux termes suffira au lecteur pour reconnaître ce qui se cache sous chacun d'eux.

Événement : un événement se passe lorsqu'il y a un changement de la valeur d'un signal.

Transaction : une transaction a lieu lorsqu'une valeur est assignée à un signal après un temps donné. Si la valeur du signal est différente de celle qu'il avait précédemment (l'instant d'avant), alors elle génère un événement.

2.6 NOTION D'EXÉCUTION CONCURRENTTE ET SÉQUENTIELLE

Dans un langage informatique, toutes les instructions se déroulent de manière séquentielle. Par contre sur un circuit intégré, toutes les portes fonctionnent simultanément et tous les signaux sont transmis de manière concurrente. C'est une erreur très fréquente dans un langage de modélisation de circuits comme le VHDL qu'il y ait des confusions entre les deux notions.

Pour éviter celles-ci par la suite, une définition de quelques termes est donnée ici.

Programme concurrent : programme dans lequel les instructions s'exécutent de manière simultanée. Il n'y a aucun ordre d'exécution de ces événements. VHDL est un programme à exécution concurrente.

Programme séquentiel : programme se déroulant de manière séquentielle. Toutes les opérations ont lieu les unes après les autres de manière ordonnée (comme dans un programme informatique par exemple, ADA, C, C++, Pascal, ...).

Système combinatoire : système numérique dans lequel les valeurs des sorties sont déterminées par les valeurs "actuelles" (aux temps de propagation près) des entrées. Lorsqu'un événement a lieu à une entrée, il est immédiatement (aux temps de propagation près) répercuté sur la sortie.

Système séquentiel : système numérique dans lequel les valeurs des sorties sont déterminées par des informations précédemment mémorisées dans des bascules, et par les valeurs "actuelles" (aux temps de propagation près) des entrées. Les informations à mémoriser sont enregistrées dans des bascules en synchronisme avec un signal d'horloge. Un événement à une entrée ne se répercute pas forcément de façon "immédiate" (aux temps de propagation près) sur les sorties. Il n'influencera les informations mémorisées que lors du prochain "coup" d'horloge.

Pour l'instant, il ne sera pas parlé de la manière de coder en VHDL des systèmes combinatoires et séquentiels. Ceci sera traité par la suite dans ce cours. Par contre quelques notions seront données pour mieux discerner la manière d'écrire du code pour des éléments concurrents par rapport à un programme informatique, qui se veut séquentiel. Prenons des exemples, l'un concurrent et l'autre séquentiel.

2.6.1 Programme à exécution concurrente :

```
1- ARCHITECTURE concurrent OF exemple IS
2-   BEGIN
3-       sortie_a <= entree_a AND entree_b;
4-       sortie_b <= sortie_c OR entree_a;
5-       sortie_c <= sortie_d AND entree_b;
6-       sortie_d <= entree_a OR entree_c;
7-   END concurrent;
```

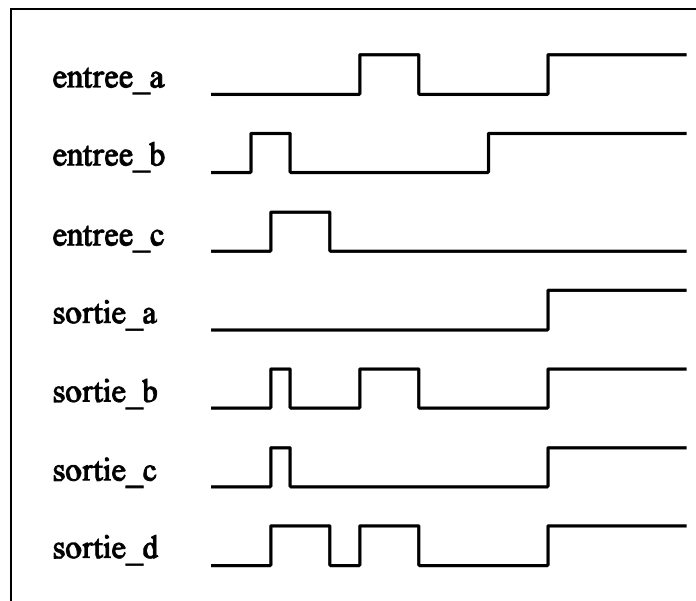


Image 7

Sur cette figure n° 7, le résultat des opérations est donné de manière brute sans voir de quelle manière elles ont eu lieu. Une autre figure ci-dessous représente les mêmes opérations, mais cette fois-ci en indiquant tous les changements qui ont lieu, ce qui permettra de voir que l'ordre des opérations dans un programme concurrent n'a aucune importance, ce qui en complique généralement la compréhension.

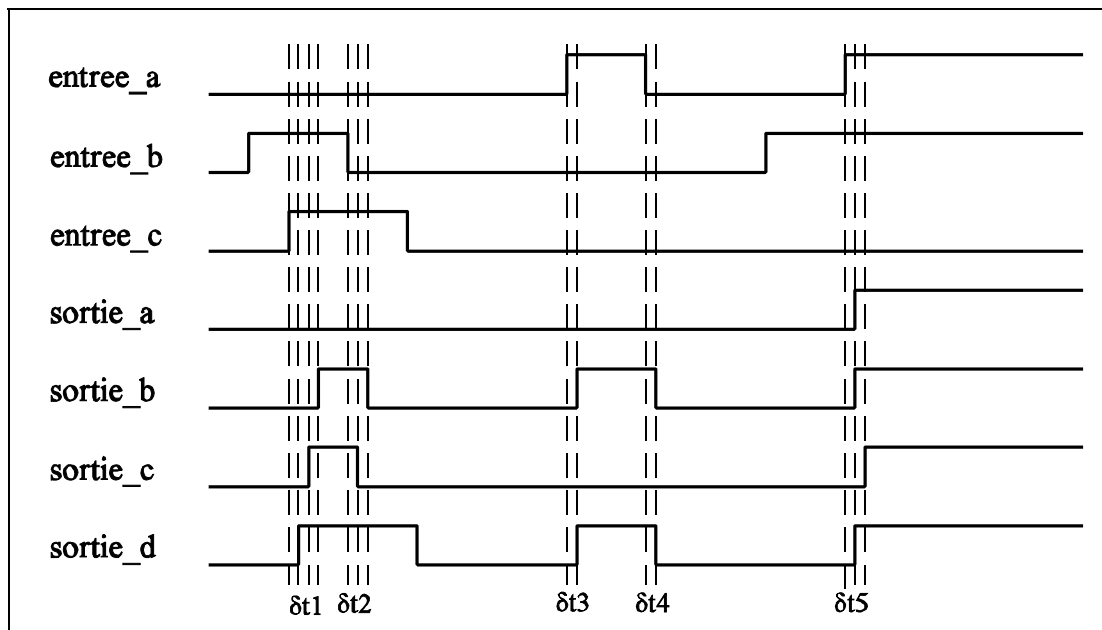


Image 8

Les temps δt_1 , δt_2 , δt_3 , δt_4 et δt_5 sont des temps infiniment courts. En portant une étude sur chacun de ces temps, il est possible de voir dans quel ordre les opérations se sont déroulées.

- δt_1 : - changement de l'entrée c
 - changement de la sortie d ↗ ligne 6 du code VHDL
 - changement de la sortie c ↗ ligne 5 du code VHDL
 - changement de la sortie b ↗ ligne 4 du code VHDL
- δt_2 : - changement de l'entrée b
 - changement de la sortie c ↗ ligne 5 du code VHDL
 - changement de la sortie b ↗ ligne 4 du code VHDL
- δt_3 : - changement de l'entrée a
 - changement des sorties b & d ↗ lignes 4 & 6 du code VHDL
- δt_4 : - changement de l'entrée a
 - changement des sorties b & d ↗ lignes 4 & 6 du code VHDL
- δt_5 : - changement de l'entrée a
 - changement des sorties a, b & d ↗ lignes 3, 4 & 6 du code VHDL
 - changement de la sortie c ↗ ligne 5 du code VHDL

En réalité, il y a aussi un temps delta entre deux changements simultanés de signaux, mais il n'ajoute rien à la compréhension de l'ordre de l'exécution d'un code concurrent.

Il est donc possible de voir ci-dessus que l'ordre des instructions dans le programme n'a aucune importance, c'est les modifications de la valeur des signaux qui dicteront l'ordre d'exécution des instructions.

2.6.2 Programme à exécution séquentielle

Pour éviter l'utilisation de tel ou tel langage, du pseudo-code sera utilisé pour l'exemple.

```

1- Constante max_compteur = 10;

2- Début du programme
3-  compteur = 0;                                "Initialise le
   compteur"

4-  "Boucle permettant d'afficher toutes les valeurs du compteur"
5-  boucle tant que compteur < max_compteur
6-    début de la boucle
7-      compteur = compteur + 1;
8-      afficher compteur;                        "Procédure d'affichage
externe"
9-    fin de la boucle;

10- fin du programme;

```

Le déroulement séquentiel d'un programme est très aisé à comprendre puisque le lecteur sait exactement quelle est l'instruction exécutée par le programme. Les instructions se déroulent dans un ordre très précis et il n'y a aucun calcul à faire pour rechercher l'instruction exécutée. C'est de cette manière que se déroule l'exécution des divers programmes à utilisation informatique. Cependant il est très probable qu'apparaissent des programmes de type informatique qui auront une exécution dite concurrente. En effet les prochaines technologies permettront probablement de reprogrammer les processeurs (VISC - Variable Instruction Set Computer) à partir d'un logiciel. Cela permettra d'optimiser ceux-ci, par contre le code devra gérer une quantité d'informations actuellement inexistantes [voir Electronique n°65- Décembre 1996].

Certaines parties du code VHDL s'écrivent et s'exécutent de manière séquentielle, comme par exemple ce qui se trouve dans le corps d'un processus. Les signaux réagiront différemment selon qu'il s'agisse de signaux inertiels ou transports.

Soit deux transactions, appelons-les "première" et "seconde" selon leur ordre d'apparition dans le listing d'un processus.

	TRANSPORT	INERTIAL
La seconde transaction a lieu avant la première	Ecrit par-dessus la première transaction	Ecrit par-dessus la première transaction
La seconde transaction a lieu après la première	Ajoute la seconde transaction au signal	Ecrit par-dessus la première transaction si les valeurs sont différentes, sinon les deux sont maintenues

VHDL Analysis and Modeling of Digital Systems - Zainalabedin Navabi

4 graphiques représenteront les effets du codage en VHDL.

1a) Inertiel avec la seconde transaction qui a lieu avant la première (fig.9)

```
BEGIN
  PROCESS
  BEGIN
    sortie <='1' AFTER 10ns; -- premiere transaction
    sortie <='0' AFTER 5ns;  -- seconde transaction
    WAIT;
  END PROCESS;
END;
```

1b) Transport avec la seconde transaction qui a lieu avant la première (fig.9)

```
BEGIN
  PROCESS
  BEGIN
    sortie <= TRANSPORT '1' AFTER 10ns; -- premiere transaction
    sortie <= TRANSPORT '0' AFTER 5ns;  -- seconde transaction
    WAIT;
  END PROCESS;
END;
```

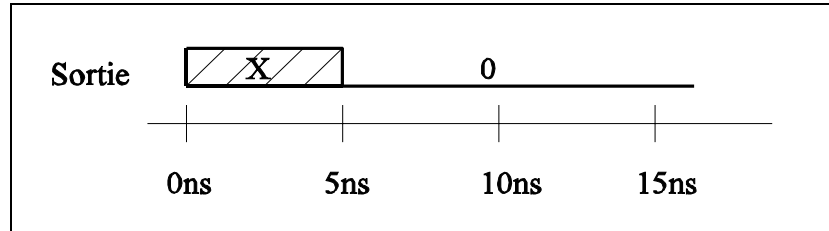


Image 9

2) Inertiel avec la seconde transaction qui a lieu après la première, valeurs semblables (fig. 10)

```
BEGIN
  PROCESS
  BEGIN
    sortie <='0' AFTER 10ns; -- premiere transaction
    sortie <='0' AFTER 15ns; -- seconde transaction
    WAIT;
  END PROCESS;
END;
```

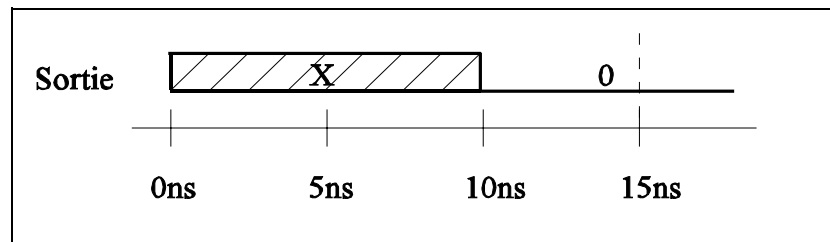



Image 10

3) Inertiel avec la seconde transaction qui a lieu après la première, valeurs différentes (fig. 11)

```
BEGIN
  PROCESS
  BEGIN
    sortie <= '1' AFTER 10ns; -- premiere transaction
    sortie <= '0' AFTER 15ns; -- seconde transaction
    WAIT;
  END PROCESS;
END;
```

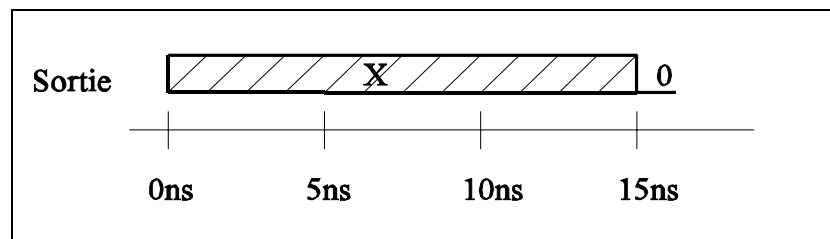


Image 11

L'état '1' qui aurait dû exister entre t=10ns et t=15ns a été filtré (supprimé).

4) Transport avec la seconde transaction qui a lieu après la première (fig. 12)

```
BEGIN
  PROCESS
  BEGIN
    sortie <= TRANSPORT '1' AFTER 10ns; -- p r e m i e r e
transaction
    sortie <= TRANSPORT '0' AFTER 15ns; -- seconde transaction
    WAIT;
  END PROCESS;
END;
```

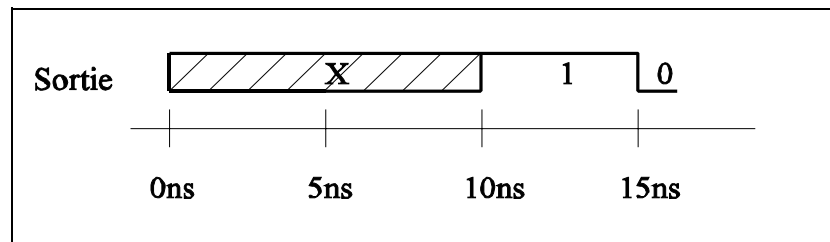


Image 12

2.7 DIFFÉRENCE ENTRE L'UTILISATION D'UNE VARIABLE ET D'UN SIGNAL

Maintenant, il reste à savoir à quel moment utiliser un signal, plutôt qu'une variable. Pour exprimer cette différence, nous allons reprendre une remarque de Navabi [1 - p.40].

- Les **signaux** ont une représentation matérielle. De plus, une composante de temps leur est associée. Les signaux peuvent autant être utilisés dans des corps séquentiels ou concurrents, par contre leur déclaration ne se fait que dans une partie concurrente du code VHDL. L'assignement se fait par les deux symboles <= (" $<$ " et " $=$ ").

Exemples:

```
1) et_prov <= a0 AND a1 AFTER 10ns, '1' AFTER 60ns;
```

Par cette instruction (utilisée pour un programme de simulation), nous indiquons que le signal ET_PROV vaut 'A0&A1' après 10ns et que 50ns plus tard (càd 60ns après le début), il vaut '1'.

```
2) sortie_et <= a AND b AFTER 40ns;
```

Cette instruction (utilisée pour un programme de conception de circuit) nous indique que SORTIE_ET vaut le ET logique de A et B et qu'il ne change qu'après 40ns. Ces 40ns ne seront pas utilisées par un synthétiseur. Par contre, cela permet d'évaluer par simulation des temps de retard sur un circuit que l'on conçoit.

- Les **variables** n'ont au contraire pas de notion de temps qui leur soit associée. Elles sont souvent utilisées pour définir des valeurs intermédiaires dans un code de type comportemental. Dès le moment où l'on veut récupérer la valeur d'une de ces variables pour la représenter physiquement, il faut la convertir en signal. Elles ne peuvent être déclarées et utilisées que dans les parties séquentielles d'un code VHDL (processus, fonctions,

procédures).

L'assignement se fait par les deux symboles := (“:=” et “=”).

Exemple:

```
increment := increment + 3;
```

2.8 DIFFÉRENCE ENTRE LE BIT ET LE BOOLEAN

Le type *bit* est un type associé à un signal. Ce type est donc directement lié au circuit lui-même.

Le type *boolean*, au contraire, n'a pas de représentation physique. Il est utilisé pour les opérations logiques du programme VHDL.

Pourquoi expliquer la différence entre ces deux types ? A cause de l'utilisation qui en est faite. Certaines personnes font la confusion entre ces deux types au début de leur apprentissage du VHDL. En effet, ces deux types peuvent travailler avec les mêmes opérateurs et leur résultats semblent équivalents. Pourtant quelques nuances existent, et pour les cerner il faut observer les résultats obtenus avec les différents opérateurs.

Les opérateurs logiques *and*, *or*, *nand*, *nor* et *xor* donnent un résultat du même type que les opérands. Il existe des portes logiques de ce nom avec lesquelles on utilisera le type *bit* pour représenter les entrées et les sorties et il s'agit aussi d'opérateurs booléens comme dans d'autres langages informatique. Ce sont donc bien deux types distincts ne s'utilisant pas pour les mêmes choses qui peuvent travailler avec ces opérateurs.

Les opérateurs relationnels *=*, */=*, *>*, *<*, *>=*, *<=* génèrent automatiquement un résultat booléen. Ce qui est tout-à-fait normal, puisqu'il est impossible de définir une valeur pour une comparaison. Il n'est possible que de dire si le résultat de la comparaison est vrai ou faux, ce qui correspond au type booléen. Si on veut faire une comparaison entre deux bits, il faudra selon le résultat assigner la valeur 0 ou 1 à la sortie désirée.

L'opérateur de concaténation *&* s'utilisera avec un type *bit*. Cela permet de regrouper plusieurs bits en un vecteur de bits. C'est très pratique pour définir des bus.

3 DÉBUTS EN VHDL

3.1 MÉTHODE DE TRAVAIL

Lorsque l'on commence un développement, il est utile de se demander de quelle manière celui-ci sera réalisé. Faut-il attaquer les problèmes les uns après les autres ou en parallèle ? Quel type de description faut-il utiliser pour telle ou telle partie du circuit à concevoir ? Et il y en a d'autres encore. Ce chapitre tente d'y répondre par une méthode de travail. Celle-ci minimisera les erreurs de développement en VHDL et facilitera la compréhension du code VHDL écrit par quelqu'un d'autre (pour autant que ce quelqu'un ait suivi cette méthode).

L'organigramme (fig. 13) de la page 21 donne trois informations :

- 1) Quelles sont les diverses étapes d'un projet.
- 2) Dans quel ordre va-t-on les traiter.
- 3) Quel type de structure VHDL faut-il employer pour les différentes parties du développement.

Pour faciliter la compréhension de l'organigramme, tout ce qui s'écrit en VHDL se trouve dans des cases ombrées.

Un développement se décompose donc de la manière suivante :

- une première étape consiste à préparer la documentation. Celle-ci, écrite en VHDL, permet aussi de valider les spécifications avant de poursuivre le développement.
- l'étape suivante consiste à développer conjointement deux parties du système. Il s'agit de la description des bus et registres (généralement par une description du type *flot de données*) et de la description du pseudo-code ou des machines d'état (en général à l'aide d'une description *comportementale*). La simulation valide ces descriptions.
- un assemblage du tout est alors réalisé par une description *structurelle* et l'ensemble est aussi simulé.
- Les dernières étapes sont pour le placement-routage. La simulation de celui-ci et la fabrication du produit.

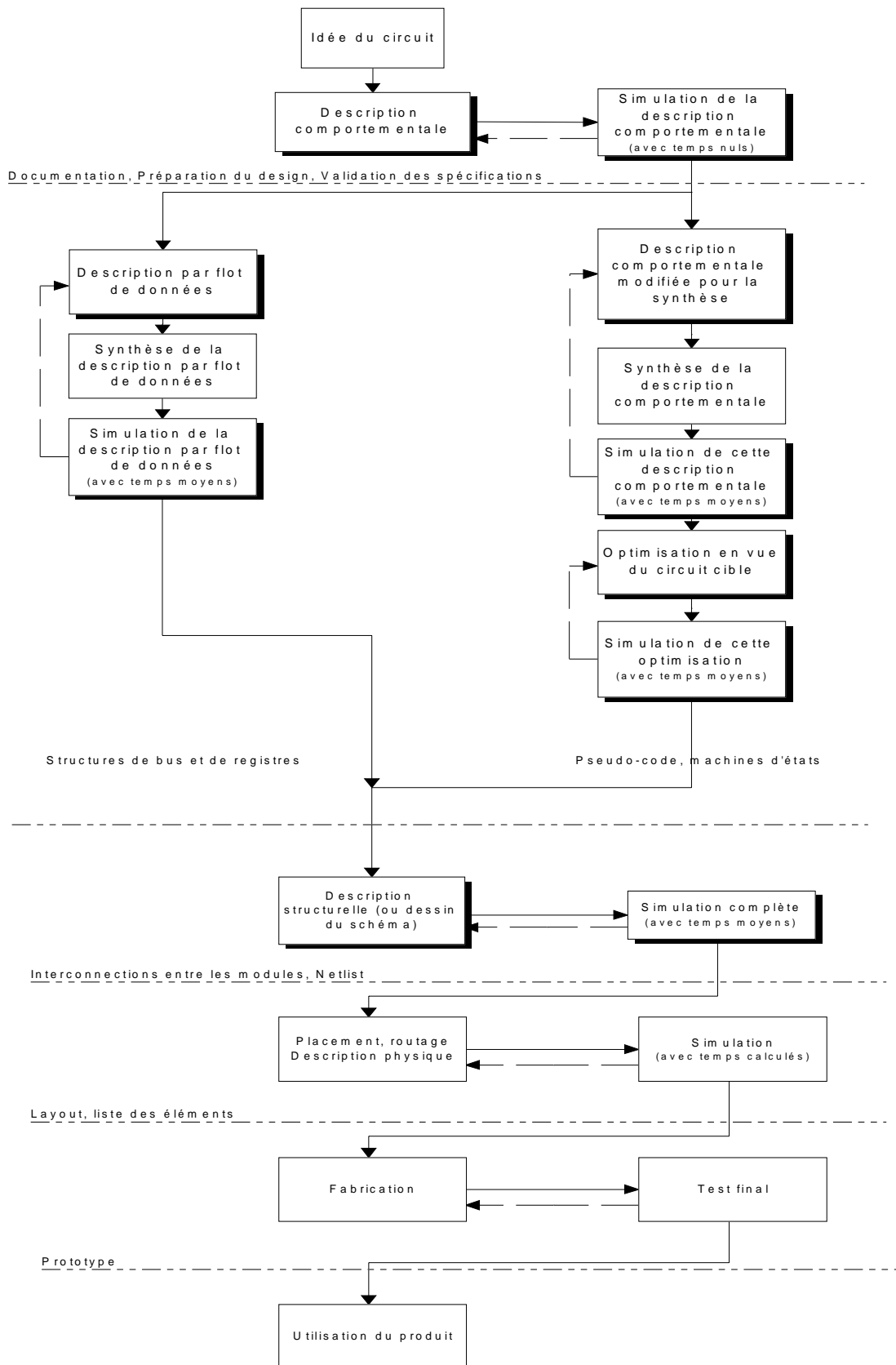


Image 13

3.2 *SPÉCIFICATION ET SYNTHÈSE*

Deux étapes particulières apparaissent dans la figure 13, celle de la spécification et de la synthèse. En VHDL, il convient de bien distinguer ces deux étapes, car le code écrit pour l'une ou l'autre n'est pour l'instant pas complètement identique. Avant de commencer l'apprentissage du VHDL, il est donc nécessaire de les définir.

La spécification est la partie d'un développement qui consiste à valider par la simulation ce qui a été demandé par le mandataire. Elle permet de corriger un cahier des charges incorrect ou de compléter celui-ci. La spécification en VHDL permet de créer une sorte de maquette qui a, vu de l'extérieur, le comportement du système désiré.

La synthèse permet de générer automatiquement à partir du code VHDL un schéma de câblage permettant la programmation du circuit cible. La description du code pour une synthèse est, en théorie, indépendante de l'architecture du circuit. En pratique, le style utilisé aura une influence sur le résultat de la synthèse, influence liée au type de circuit et au synthétiseur utilisé. Le code devra être optimisé pour un type circuit. Si le besoin d'optimisation n'est pas très important, cette partie peut se dérouler très rapidement, alors qu'au contraire, si le besoin d'optimisation est grand, les subtilités pour coder de manière adéquate en VHDL obligeront le concepteur à y consacrer beaucoup de temps. D'ailleurs, ce n'est souvent plus en VHDL que l'on optimisera, car ce langage se révèle, pour l'instant en tout cas, peu adapté à cet usage. Il faudra travailler à un niveau plus proche du circuit ou acheter des macros pré-existantes.

3.3 *L'INTERFACE VHDL*

Avant d'écrire un programme en VHDL, il faut définir l'interface entre l'intérieur du circuit et le monde extérieur (représentation graphique sur la figure 14, 28). Il ne suffit pas d'avoir un programme qui décrit un certain comportement. Ce comportement réagit selon des entrées et agit sur des sorties (ou des entrées-sorties) et tous ces ports seront connectés à une carte externe au circuit. Toutes ces "connexions" correspondent à cette interface.

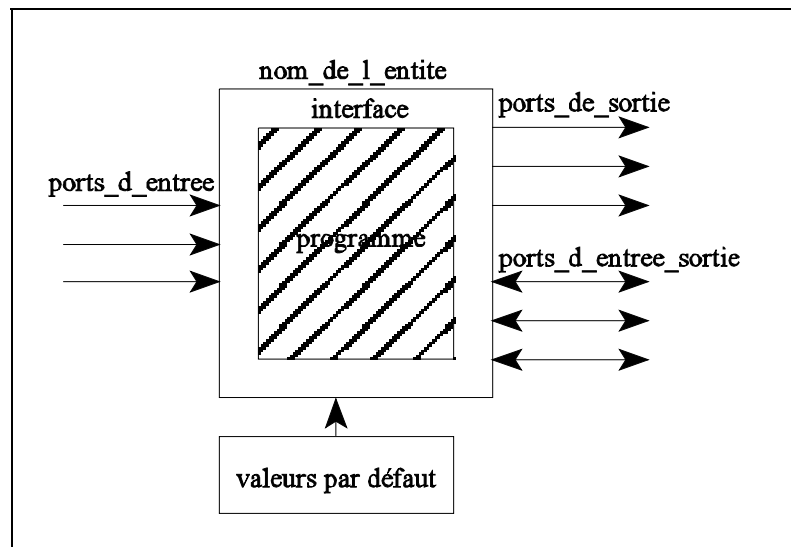


Image 14

La figure 14, 28 représente les besoins d'un concepteur. Il a un certain nombre d'entrées, de sorties et d'entrées-sorties, de plus quelques valeurs par défaut lui sont connues (comme par exemple le délai typique d'une porte logique simple). A ce moment de la conception, il ne connaît pas encore ce qu'il va programmer. Il possède juste ces informations et le nom du module qu'il doit créer (*nom_de_l_entite*). La manière de coder cette figure en VHDL est la suivante :

```

-----
-- declaration et utilisation des bibliotheques necessaires -
-----

USE work.bibliotheques_necessaires.ALL;

-----
-- declaration de l'entite et de ses ports d'entree-sortie -
-----

ENTITY nom_de_l_entite IS
    GENERIC (parametres_generiques: type := Valeur_par_defaut);
    PORT (ports_d_entree : IN type_ports_d_entree;
          ports_de_sortie : OUT type_ports_de_sortie;
          ports_d_entree_sortie : INOUT type_ports_entree_sortie);
END nom_de_l_entite;

-----
-- architecture du programme, structure interne -
-----

ARCHITECTURE type_de_description OF nom_de_l_entite IS
BEGIN

```

```
-- programme interne, encore inconnu pour le concepteur
END type_de_description;
```

On déclare en premier lieu les différentes bibliothèques qui seront utilisées. Le code se décompose en deux parties principales, l'entité et l'architecture.

Entité : L'entité est la vue externe de l'élément créé. On y déclare les ports d'entrée et de sortie et on y donne un nom à l'élément (*nom_de_l_entite* pour l'exemple ci-dessus). Il est aussi possible de définir des paramètres génériques, c'est-à-dire des paramètres qui ont une valeur utilisée par défaut si l'utilisateur de cette entité n'en donne pas une autre. Par exemple, si une architecture appelle l'entité ci-dessus sans spécifier de temps de délai, alors la valeur *Valeur_par_defaut* sera utilisée.

Architecture : L'architecture est l'évaluation du contenu de l'élément créé. Il est tout-à-fait possible d'avoir plusieurs architectures différentes pour une même entité. Lors de la définition de l'architecture, on y indique le type de description utilisée (*type_de_description*) et l'entité utilisée (*nom_de_l_entite*). Aucun de ces deux mots n'est réservé.

Un commentaire commence à l'aide des deux caractères -- et se termine par la fin de ligne.

3.4 LE VHDL POUR LE COMBINATOIRE

L'étude du VHDL pour des circuits concrets ne se fera pas selon l'approche classique qui consiste à donner une liste des instructions pour les utiliser ensuite, mais elle s'appuiera sur les besoins, les outils étant alors fournis. Cela double parfois les informations, mais permet de ne pas avoir besoin de comprendre tout le VHDL avant de démarrer un petit projet. Une première étape traitera du développement de programmes VHDL pour de la logique combinatoire et une deuxième s'occupera de la logique séquentielle.

Regardons d'abord quelques modules classiques en combinatoire. La description de chacun de ces modules sera faite sous la forme la plus appropriée.

Système	Type de description	Page
composé seulement de portes logiques	flot de données / structurelle	25
composé d'éléments en bibliothèque	structurelle	30
table de vérité	flot de données	28

Systeme	Type de description	Page
bascule asynchrone (RS par exemple)	flot de données	26
multiplexeur	flot de données	27
éléments en cascade	structurelle	30
comparateur	comportementale	31

3.4.1 Description par flot de données

Pour décrire des éléments simples, des tables de vérité ou autres multiplexeurs, il est très pratique d'utiliser la description dite *par flot de données*. Ce type de description se déroule de manière concurrente. Tous les signaux sont assignés en même temps. Il suffit de placer à la droite d'une équation ce que l'on veut assigner au signal qui se trouve à la gauche de l'équation.

3.4.1.1 L'additionneur

Un exemple, un demi-additionneur 1 bit, aidera à la compréhension de ce type de description.

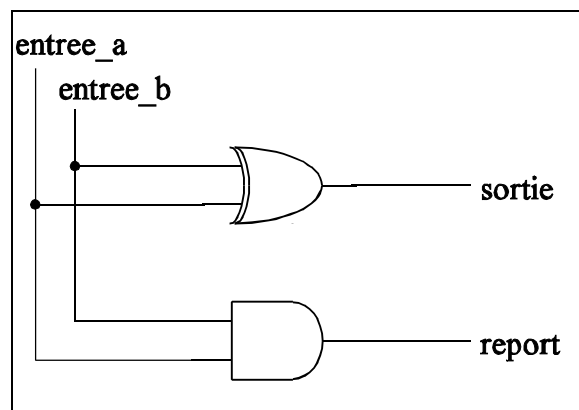


Image 15

Code VHDL :

```

ENTITY demi_add_1bit IS
  PORT (entree_a : IN bit;
         entree_b : IN bit;
         sortie  : OUT bit;
         report  : OUT bit);
END demi_add_1bit;

ARCHITECTURE flot_de_donnees OF demi_add_1bit IS
BEGIN
  Report <= entree_a AND entree_b;
  Sortie <= entree_a XOR entree_b;
END flot_de_donnees;

```

L'entité et l'architecture se déclarent de manière habituelle. Pour l'assignation des signaux, deux opérateurs ont été utilisés AND et XOR (la liste exhaustive se trouve en annexe).

3.4.1.2 *La bascule RS asynchrone*

La bascule RS asynchrone est aussi un exemple très simple pour une description combinatoire par flot de données. Il complétera l'exemple de l'additionneur.

```

ENTITY bascule_RS IS
  GENERIC (delai: TIME := 5 ns); -- delai par default, modifiable
  PORT (set, reset :IN bit;
        sortie      :INOUT bit);
END bascule_RS;

ARCHITECTURE flot_de_donnees OF bascule_RS IS
BEGIN
  sortie <= set OR (sortie AND NOT reset) AFTER delai;
END flot_de_donnees;

```

Certains compilateurs demanderont cette architecture :

```

ARCHITECTURE flot_de_donnees OF bascule_RS IS
  SIGNAL sortie_int : bit;
BEGIN
  sortie_int <= sortie AND NOT reset;
  sortie <= set OR sortie_int AFTER delai;
END flot_de_donnees;

```

3.4.1.3 Le multiplexeur

Les deux exemples précédents placent toujours la même équation sur la sortie, mais si le concepteur veut réaliser un multiplexeur, il devra assigner, suivant le code de sélection, la valeur des entrées à la sortie. Il existe en VHDL une commande qui permet cela, le mot réservé *when*. Elle permet de donner une condition lors d'un assignement.

Prenons le cas d'un multiplexeur à 8 entrées et 3 bits de sélection. Le symbole est représenté sur la figure 16.

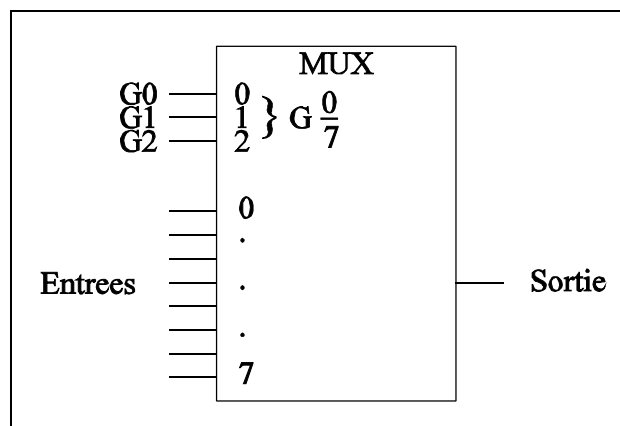


Image 16

Code VHDL :

```

ENTITY mux8 IS
  GENERIC (delai: TIME := 5 ns); -- delai par defaut, modifiable
  PORT (e7, e6, e5, e4, e3, e2, e1, e0: IN bit;
         G2, G1, G0: IN bit;
         sortie: OUT bit);
END mux8;

ARCHITECTURE flux_de_donnees OF mux8 IS
  SIGNAL adresse : bit_vector(2 DOWNTO 0);

BEGIN
  adresse <= G2&G1&G0;           -- concatenation en un seul vecteur

  WITH adresse SELECT
    sortie <= e0 AFTER delai WHEN "000",
             e1 AFTER delai WHEN "001",
             e2 AFTER delai WHEN "010",
             e3 AFTER delai WHEN "011",
             e4 AFTER delai WHEN "100",
             e5 AFTER delai WHEN "101",

```

```

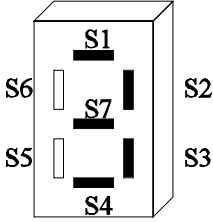
e6 AFTER delai WHEN "110",
e7 AFTER delai WHEN "111";
END flux_de_donnees;

```

3.4.1.4 La table de vérité

De même, il est aussi possible d'utiliser avantageusement ce mot réservé when pour créer une table de vérité. L'exemple utilisé est très simple à comprendre puisqu'il s'agit d'afficher sur un affichage 7 segments la valeur d'un code BCD. Les cas non désirés afficheront un X.

D3	D2	D1	D0	S7	S6	S5	S4	S3	S2	S1
0	0	0	0	0	1	1	1	1	1	1
0	0	0	1	0	0	0	0	1	1	0
0	0	1	0	1	0	1	1	0	1	1
0	0	1	1	1	0	0	1	1	1	1
0	1	0	0	1	1	0	0	1	1	0
0	1	0	1	1	1	0	1	1	0	1
0	1	1	0	1	1	1	1	1	0	1
0	1	1	1	0	0	0	0	1	1	1
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	0	1	1	1	1
1	0	1	0	1	1	1	0	1	1	0
1	0	1	1	1	1	1	0	1	1	0
1	1	0	0	1	1	1	0	1	1	0
1	1	0	1	1	1	1	0	1	1	0
1	1	1	0	1	1	1	0	1	1	0
1	1	1	1	1	1	1	0	1	1	0



0011 → 1001111

```

ENTITY sept_seg IS
  GENERIC (delai: TIME := 5 ns); -- delai par default, modifiable
  PORT (code_binaire: IN bit_vector(3 DOWNTO 0); -- code d'entree
        S7, S6, S5, S4, S3, S2, S1: OUT bit); -- affichage 7 seg.

```

```

END sept_seg;

ARCHITECTURE TDV_flot_de_donnees OF sept_seg IS
  SIGNAL code_sept_seg : bit_vector(6 DOWNT0 0);

BEGIN

  WITH code_binaire SELECT
    code_sept_seg <= "0111111" AFTER delai WHEN "0000", -- 0
                    "0000110" AFTER delai WHEN "0001", -- 1
                    "1011011" AFTER delai WHEN "0010", -- 2
                    "1001111" AFTER delai WHEN "0011", -- 3
                    "1100110" AFTER delai WHEN "0100", -- 4
                    "1101101" AFTER delai WHEN "0101", -- 5
                    "1111101" AFTER delai WHEN "0110", -- 6
                    "0000111" AFTER delai WHEN "0111", -- 7
                    "1111111" AFTER delai WHEN "1000", -- 8
                    "1101111" AFTER delai WHEN "1001", -- 9
                    "1110110" WHEN OTHERS; -- X

  S7 <= code_sept_seg(6);
  S6 <= code_sept_seg(5);
  S5 <= code_sept_seg(4);
  S4 <= code_sept_seg(3);
  S3 <= code_sept_seg(2);
  S2 <= code_sept_seg(1);
  S1 <= code_sept_seg(0);

END TDV_flot_de_donnees;

```

3.4.1.5 *Résumé*

```

USE work.bibliotheques_necessaires.ALL;

ENTITY nom_de_l_entite IS
  GENERIC (parametres_generiques: type := Valeur_par_defaut);
  PORT (ports_d_entree      : IN   type_ports_d_entree;
        ports_de_sortie    : OUT  type_ports_de_sortie;
        ports_d_entree_sortie : INOUT type_ports_entree_sortie);
END nom_de_l_entite;

ARCHITECTURE type_de_description OF nom_de_l_entite IS
BEGIN
  signal_sortie_1 <= action_1;
  signal_sortie_2 <= action_2 WHEN condition_1,
                    action_3 WHEN condition_2,
                    action_4 WHEN OTHERS;
  ...
END type_de_description;

```

3.4.2 Description structurelle

Cette description correspond à un assemblage de modules. Chacun des modules est décrit de manière comportementale ou par flot de données et fait partie ensuite d'une bibliothèque de modules. Ils sont alors utilisés de la même manière que la structure d'un schéma. Un additionneur 4 bits donnera un bon exemple de ce type de description. Pour ce faire, le module d'un additionneur 1 bit sera utilisé.

Élément se trouvant en bibliothèque : (additionneur 1 bit)

```

ENTITY addit_1_bit IS
  PORT (entree_a: IN bit;
         entree_b: IN bit;
         report_prec: IN bit;
         somme: OUT bit;
         report_suiv: OUT bit);
END addit_1_bit;

ARCHITECTURE flot_de_donnees OF addit_1_bit IS
BEGIN
  report_suiv <= (entree_a AND entree_b) OR (entree_a AND report_prec)
                OR (entree_b AND report_prec);
  somme <= entree_a XOR entree_b XOR report_prec;
END flot_de_donnees;

```

Additionneur 4 bits :

```

ENTITY addit_4_bits IS
  PORT (nombre1, nombre2 : IN bit_vector (3 DOWNTO 0);
         report_in       : IN bit;
         total           : OUT bit_vector (3 DOWNTO 0);
         report_out      : OUT bit);
END addit_4_bits;

ARCHITECTURE structurelle OF addit_4_bits IS

  COMPONENT bloc1 PORT (entree_a, entree_b, report_prec : IN bit;
                        somme, report_suiv : OUT bit);
  END COMPONENT;

  FOR ALL : bloc1 USE ENTITY WORK.addit_1_bit (flot_de_donnees);

  SIGNAL report0, report1, report2 : bit;

BEGIN
  g0 : bloc1 PORT MAP
    (nombre1(0), nombre2(0), report_in, total(0), report0);
  g1 : bloc1 PORT MAP
    (nombre1(1), nombre2(1), report0, total(1), report1);

```

```
g2 : bloc1 PORT MAP
      (nombre1(2), nombre2(2), report1, total(2), report2);
g3 : bloc1 PORT MAP
      (nombre1(3), nombre2(3), report2, total(3), report_out);
END structurelle;
```

3.4.2.1 *Résumé*

```
USE work.bibliotheques_necessaires.ALL;
```

```
ENTITY nom_de_l_entite IS
  GENERIC (parametres_generiques: type := Valeur_par_defaut);
  PORT (ports_d_entree      : IN    type_ports_d_entree;
        ports_de_sortie    : OUT   type_ports_de_sortie;
        ports_d_entree_sortie : INOUT type_ports_entree_sortie);
END nom_de_l_entite;
```

```
ARCHITECTURE type_de_description OF nom_de_l_entite IS
```

```
  COMPONENT nom_du_composant_local PORT (
    entrees_du_comp      : IN    type_des_entrees;
    sorties_du_comp      : OUT   type_des_sorties;
    entrees_sorties_comp : INOUT type_entrees_sorties);
  END COMPONENT;
```

```
  FOR ALL : nom_du_composant_local
    USE ENTITY WORK.nom_entite_comp (nom_arch_composant);
```

```
BEGIN
  nom_de_l_element :nom_du_composant_local PORT MAP
    (entrees_du_comp, sorties_du_comp,
     entrees_sorties_comp);
```

```
END type_de_description;
```

3.4.3 *Description comportementale*

La description comportementale se fait à l'aide d'un processus. Les instructions à l'intérieur de celui-ci s'exécutent de manière séquentielle.

Un comparateur est un élément combinatoire avantageusement décrit de façon comportementale.

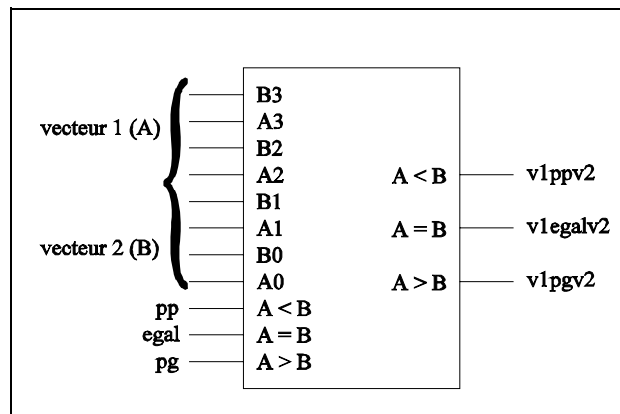


Image 18

```

ENTITY compareteur IS
  GENERIC (delai: TIME := 7 ns); -- delai par default, modifiable
  PORT (vecteur1, vecteur2 : IN bit_vector (3 DOWNTO 0);
        pg, egal, pp : IN bit;
        v1pgv2, v1egalv2, v1ppv2 : OUT bit);
END compareteur;

ARCHITECTURE comportementale OF compareteur IS
BEGIN
  PROCESS (vecteur1, vecteur2, pg, egal, pp)
  BEGIN

    v1pgv2 <= '0';      -- initialisation obligatoire,
    v1egalv2 <= '0';   -- sinon des bascules sont
    v1ppv2 <= '0';     -- rajoutees pour synchroniser les
                      -- signaux

    IF vecteur1 > vecteur2 THEN
      v1pgv2 <= '1' AFTER delai;
      v1egalv2 <= '0' AFTER delai;
      v1ppv2 <= '0' AFTER delai;
    ELSIF vecteur1 = vecteur2 THEN
      v1pgv2 <= pg AFTER delai;
      v1egalv2 <= egal OR NOT(pg OR pp) AFTER delai;
      v1ppv2 <= pp AFTER delai;
    ELSIF vecteur1 < vecteur2 THEN
      v1pgv2 <= '0' AFTER delai;
      v1egalv2 <= '0' AFTER delai;
      v1ppv2 <= '1' AFTER delai;
    END IF;
  END PROCESS;
END comportementale;

```

Le fonctionnement de ce petit programme demande quelques explications. C'est un processus dans lequel il y a trois branches où l'on compare les deux nombres. Dans chacun

des cas, des valeurs de sorties appropriées sont données après un délai. Chaque fois qu'une des valeurs se trouvant entre parenthèses après le mot-clé *process* est modifiée, le processus est activé et toutes les instructions sont exécutées séquentiellement.

3.4.3.1 *Résumé*

Pour décrire un programme sous forme comportementale d'un système combinatoire, la structure est la suivante :

```

USE work.bibliotheques_necessaires.ALL;

ENTITY nom_de_l_entite IS
  GENERIC (parametres_generiques: type := Valeur_par_defaut);
  PORT (ports_d_entree      : IN    type_ports_d_entree;
        ports_de_sortie     : OUT   type_ports_de_sortie;
        ports_d_entree_sortie : INOUT type_ports_entree_sortie);
END nom_de_l_entite;

ARCHITECTURE type_de_description OF nom_de_l_entite IS
BEGIN
  PROCESS (valeurs_permettant_d_entrer_dans_le_processus)
  BEGIN
    description du code;
  END PROCESS;
END type_de_description;

```

3.5 *LE VHDL POUR UN SYSTÈME SÉQUENTIEL*

La plupart des systèmes logiques sont séquentiels et peuvent être écrits en VHDL. A nouveau, la présentation de cette partie se fera selon les besoins et non les outils.

Systeme	Type de description	Page
bascule simple (D par exemple)	comportementale, flot de données	36, 41

Systeme	Type de description	Page
bascule D avec set synchrone, reset asynchrone et un enable	comportementale	18
machine d'états	machine d'états	47
compteurs	structurelle (flot de données)	52
composé d'éléments en bibliothèque	structurelle	52
éléments en cascade (compteur)	structurelle	52

Les systèmes séquentiels se décomposent en trois parties : le décodeur d'état futur (combinatoire), le registre (séquentiel) et le décodeur de sortie (combinatoire) selon la figure 19. Pour faciliter l'écriture et surtout la lecture du code VHDL, cette structure sera, dans la mesure du possible, respectée. Elle offre également l'avantage de n'avoir que peu ou pas de modifications à apporter au processus décrivant le registre pour passer d'un développement à l'autre. Cette partie du code étant en effet la plus difficile à représenter, il vaut mieux la simplifier au maximum et minimiser ainsi le risque d'erreurs.

Cependant, il arrive parfois que les décodeurs d'état futur et de sortie peuvent très facilement s'écrire si cette description se fait dans un seul et même processus ; c'est le cas pour une description à partir d'un graphe d'états. Dans ces cas, il sera avantageux de regrouper les deux processus combinatoires.

Selon le type d'architecture utilisée (flip-flops D-CE par exemple), il sera nécessaire de regrouper différemment les processus, mais cela n'entre pas dans le cadre d'une introduction au VHDL et nous laisserons de côté ces particularités liées aux synthétiseurs.

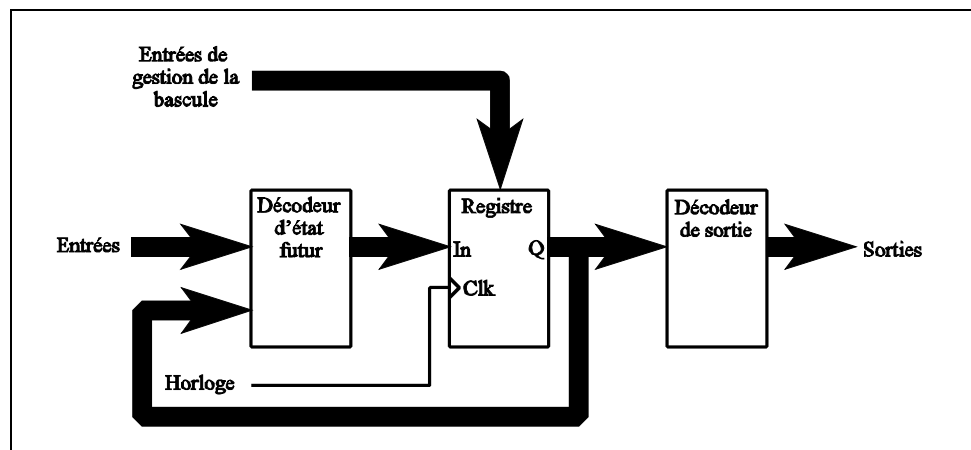


Image 19

Voici quelques avantages de cette méthode :

- La structure du code VHDL est toujours la même pour un système séquentiel, ce qui facilite la création d'un fichier chablon.

- On sépare l'action de l'horloge des actions que l'on veut sur les signaux.
- Il n'est pas utile de vérifier chaque fois la structure du code. Une fois correcte, elle le reste.

Cette structure est la suivante :

```

USE work.bibliotheques_necessaires.ALL;

ENTITY nom_de_l_entite IS
  GENERIC (parametres_generiques: type := Valeur_par_defaut);
  PORT (ports_d_entree      : IN    type_ports_d_entree;
         ports_de_sortie    : OUT   type_ports_de_sortie;
         ports_d_entree_sortie : INOUT type_ports_entree_sortie);
END nom_de_l_entite;

ARCHITECTURE type_de_description OF nom_de_l_entite IS

  SIGNAL present, futur : type_des_signaux;

BEGIN
-----
-- ce premier processus est celui qui decode l'etat futur de la
-- bascule, on l'appellera processus d'entree
-----

  PROCESS (liste_de_sensibilite_du_processus)
  BEGIN
    Comportement_du_decodeur_d_etat_futur;
  END PROCESS;

-----
-- ce processus est celui qui s'occupe de la gestion des bascules
-- on l'appellera processus de memorisation
-----

  PROCESS
  BEGIN
    WAIT UNTIL clock'EVENT and clock = '1';
    sortie_basculer <= valeur_d_entree AFTER delai;
  END PROCESS;

-----
-- le processus suivant est celui du decodeur de sortie
-- on l'appellera processus de sortie
-----

  PROCESS (liste_de_sensibilite_du_processus)
  BEGIN
    Comportement_du_decodeur_de_sortie;
  END PROCESS;
END type_de_description;

```

Chaque partie est représentée par un processus. Il est bien sûr possible de découper les decodeurs d'état futur et de sortie en plusieurs processus pour permettre le développement de petites entités réutilisables pour d'autres développements. Le processus de mémorisation

est en quelque sorte une de ces entités: conçu une fois pour toutes, le processus n'est plus touché pour des développements ultérieurs.

L'exécution des processus est la suivante :

- Tous les processus s'exécutent de manière parallèle entre eux.
- L'exécution à l'intérieur d'un processus se fait de manière séquentielle.

Un processus est activé dès qu'un des signaux de la liste de sensibilité change d'état. Cette liste de sensibilité peut prendre deux formes : soit elle apparaît entre parenthèses juste après le mot-clé *process* (a), soit elle fait partie d'une ligne *wait until* à l'intérieur du processus (b). La nouvelle norme préconise plutôt la deuxième, mais elle n'est pour l'instant pas très bien acceptée par les synthétiseurs.

```
a - PROCESS (liste de sensibilité)
    ....
    END PROCESS;
```

```
b - PROCESS
    ....
    WAIT UNTIL (liste de signaux)
    ....
    END PROCESS;
```

Cette liste de sensibilité doit comprendre tous les signaux susceptibles de modifier le comportement du système.

3.5.1 Description comportementale

Le premier exemple VHDL séquentiel et le plus classique est celui d'une bascule D. Son code est le suivant :

```
-----
-- Nom du fichier : basculeD.vhd
-- Auteur       : C. Guex
-- Date        : avril 1997
-- Version     : 1.0
-- Modifications :
-- Auteur      :                               Date :
-----
-- But          : exemple du cours VHDL d'une bascule D
-----

ENTITY bascule_d IS
  GENERIC (delai: time := 7 ns);
  PORT (entree_D, clock : IN bit;
```

```

        sortie : OUT bit);
END bascule_d;

ARCHITECTURE flot_de_donnees OF bascule_d IS

    SIGNAL sortie_bascule : bit;    -- signal intermediaire
    SIGNAL entree_bascule : bit;    -- signal intermediaire

BEGIN

-----
-- ce premier processus est celui qui decode l'etat futur de la
-- bascule, on l'appellera processus d'entree
-----

    PROCESS (entree_D)
    BEGIN
        entree_bascule <= entree_D;
    END PROCESS;

-----
-- ce processus est celui qui s'occupe de la gestion des bascules
-- on l'appellera processus de memorisation
-----

    PROCESS
    BEGIN
        WAIT UNTIL clock'EVENT and clock = '1';
        sortie_bascule <= entree_bascule AFTER delai;
    END PROCESS;

-----
-- le processus suivant est celui du decodeur de sortie
-- on l'appellera processus de sortie
-----

    PROCESS (sortie_bascule)
    BEGIN
        sortie <= sortie_bascule;
    END PROCESS;
END flot_de_donnees;

```

Dans ce cas particulier, les processus d'entrée et de sortie ne font rien. Il n'y a donc aucun délai à spécifier pour ces deux processus. Par contre, le processus de mémorisation a un délai correspondant au temps de transfert d'une bascule. Ce délai est à choisir en fonction du circuit pressenti pour le développement, ce pour améliorer la qualité de la simulation. Il ne faut pas oublier que ce délai n'a une valeur que pour la simulation. Il n'est pas possible de définir une sorte d'horloge interne au circuit.

Etant donné que les processus d'entrée et de sortie ne font rien, le code se simplifiera de la manière suivante :

```

-----
-- Nom du fichier : basculeD.vhd
-- Auteur         : C. Guex
-- Date          : avril 1997

```

```

-- Version      : 1.0
-- Modifications :
-- Auteur       :                               Date :
-----
-- But          : exemple du cours VHDL d'une bascule D
-----

ENTITY bascule_d IS
  GENERIC (delai: time := 7 ns);
  PORT (entree_D, clock : IN bit;
        sortie : OUT bit);
END bascule_d;

ARCHITECTURE flot_de_donnees OF bascule_d IS

BEGIN

-- processus du decodeur d'etat futur inexistant

-----

-- ce processus est celui qui s'occupe de la gestion des bascules
-- on l'appellera processus de memorisation
-----

PROCESS
BEGIN
  WAIT UNTIL clock'EVENT and clock = '1';
  sortie <= entree_D AFTER delai;
END PROCESS;

-- processus du decodeur de sortie inexistant

END flot_de_donnees;

```

Une bascule un peu plus complète sera donnée comme deuxième exemple. Il s'agit d'une bascule D avec enable et deux entrées supplémentaires que sont le set synchrone et le reset asynchrone. Le schéma, découpé selon les trois processus est le suivant (figure 22):

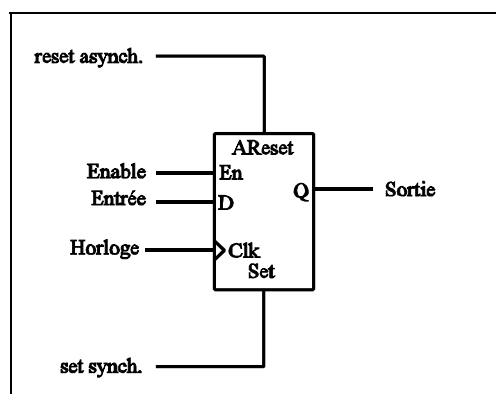


Image 20

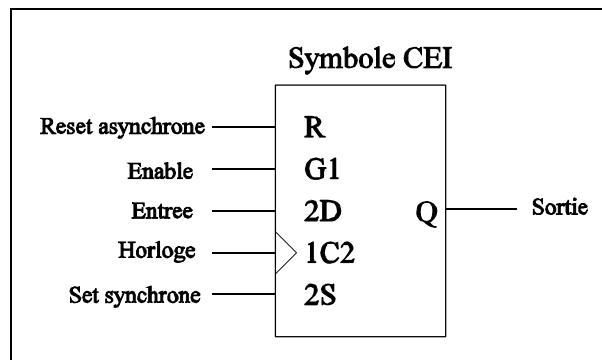


Image 21

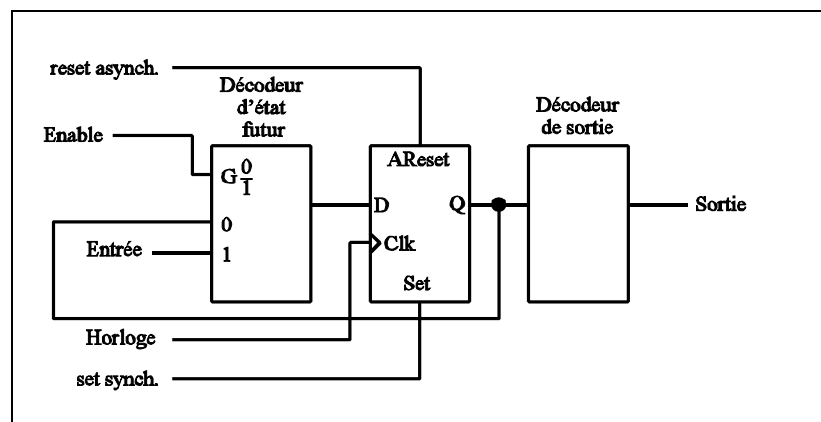


Image 22

L'enable n'a pas été placé directement sur la bascule, car il n'a pas de rapport direct avec elle. Il s'agit en fait d'un multiplexeur qui sélectionne soit la valeur d'entrée, soit l'ancienne valeur de la bascule selon l'état du signal *enable*. Le processus de mémorisation s'occupe lui de traiter l'horloge et les informations indépendantes de l'entrée, comme le set et le reset. Pour terminer, le décodeur de sortie est trivial et vaut sa propre entrée. Ce qui donne le code VHDL suivant :

```
-----
-- Nom du fichier : b_D_RSE.vhd
-- Auteur       : C. Guex
-- Date        : avril 1997
-- Version     : 1.0
-- Modifications :
-- Auteur      :
-- Date       :
-----
-- But          : exemple du cours VHDL d'une bascule D avec reset
--              : asynchrone, set synchrone et enable pour valider
--              : le transfert d'une entree sur la sortie
-----
```

```
ENTITY bascule_d_ar_ss_en IS
  GENERIC (delai: time := 7 ns); -- delai de transfert d'une porte
```

```

    PORT (entree_D : IN bit;      -- entree de la bascule D
          clock    : IN bit;      -- horloge de la bascule D
          a_reset  : IN bit;      -- reset asynchrone de la bascule D
          s_set    : IN bit;      -- set synchrone de la bascule D
          enable   : IN bit;      -- validation de la bascule D
          sortie   : OUT bit);    -- sortie de la bascule D
END bascule_d_ar_ss_en;

-----
-- Architecture comportementale d'une bascule D avec reset asynchrone
-- et set synchrone. Un enable autorise ou non le transfert de
-- l'entree D.
-----

ARCHITECTURE comportementale OF bascule_d_ar_ss_en IS

    SIGNAL entree_basculer : bit; -- valeur de l'etat futur de la bascule
    SIGNAL sortie_basculer : bit; -- valeur intermediaire que prendra
                                -- la sortie

BEGIN

    -----
    -- ce premier processus est celui qui decode l'etat futur de la
    -- bascule, on l'appellera processus d'entree
    -----

    PROCESS (entree_D, sortie_basculer, enable)
    BEGIN
        IF enable = '1' THEN
            entree_basculer <= entree_D;
        ELSE
            entree_basculer <= sortie_basculer;
        END IF;
    END PROCESS;

    -----
    -- ce processus est celui qui s'occupe de la gestion des bascules
    -- on l'appellera processus de memorisation
    -----

    PROCESS
    BEGIN
        WAIT UNTIL (a_reset = '1') OR (clock'EVENT AND clock = '1');
        IF a_reset = '1' THEN
            sortie_basculer <= '0' AFTER delai;    -- reset asynchrone
                                                    -- prioritaire
        ELSE
            IF s_set = '1' THEN
                sortie_basculer <= '1' AFTER delai; -- set synchrone en
                                                    -- deuxieme priorite
            ELSE
                sortie_basculer <= entree_basculer AFTER delai;
            END IF;
        END IF;
    END PROCESS;
END PROCESS;

```



```

-----
-- le processus suivant est celui du decodeur de sortie
-- on l'appellera processus de sortie
-----

PROCESS
BEGIN
    sortie <= sortie_bascule;
END PROCESS;

END comportementale;

```

Toutes les bascules peuvent être construites selon le modèle précédent. Il est aussi possible de conserver cette bascule en bibliothèque sous forme d'élément réutilisable. Quelques autres exemples de modules séquentiels vont encore être donnés pour illustrer ce langage. Il existe également une autre forme d'écriture, moins pratique, que nous allons voir dans le prochain paragraphe.

3.5.2 Description par flot de données

En effet, il est aussi possible de décrire des modules séquentiels par flot de données. Ces descriptions peuvent prendre deux formes, selon les exemples donnés ci-après. Cependant, ce type de description n'est pas des plus approprié pour un module séquentiel et il n'en sera pas fait grand état.

Cette description est très semblable à celle utilisée dans la partie combinatoire. Il suffit d'ajouter aux équations la synchronisation avec l'horloge de la manière suivante :

```
sortie <= entree WHEN (horloge = '1' AND NOT horloge'STABLE) ELSE sortie;
```

Le handicap d'une telle méthode est qu'il faut écrire pour chaque sortie cette condition. Il existe un moyen d'éviter ceci en créant un bloc. Toutes les opérations se trouvant alors dans ce bloc ne se dérouleront que si la condition d'entrée est vraie. Ce qui veut dire que si nous plaçons l'attente du flanc montant de l'horloge comme condition d'entrée, tous les signaux internes au bloc seront synchronisés.

L'exemple d'une bascule D est donné ci-après [selon les pages 180-181 de Navabi]. Deux architectures sont écrites pour montrer la différence entre une synchronisation de l'horloge sur un seul signal ou sur tout un bloc.

```

ENTITY d_flipflop IS
    GENERIC (delai1: time := 4 ns; delai2: time := 5 ns);
    PORT(d, c : IN bit;
         q, qb : OUT bit);

```

```

END d_flipflop;

-- exemple de signaux gardes (bloc)
ARCHITECTURE guarding OF d_flipflop IS

BEGIN
  ff: BLOCK(c='1' AND NOT c'STABLE)
  BEGIN
    q <= GUARDED      d AFTER delai1;
    qb <= GUARDED NOT d AFTER delai2;
  END BLOCK ff;
END guarding;

-- exemple de signaux assignes
ARCHITECTURE assignement OF d_flipflop IS

  SIGNAL etat_interne : bit;

BEGIN
  etat_interne <= d WHEN (c='1' AND NOT c'STABLE) ELSE etat_interne;
  q <=      etat_interne AFTER delai1;
  qb <= NOT etat_interne AFTER delai2;
END assignement;

```

L'exemple d'un bloc est celui correspondant à la première architecture. Le bloc commence par un *begin* et se termine par un *end*. A l'intérieur de celui-ci, tous les signaux qui ont le mot clé *guarded* sur leur affectation ne changent pas tant que la condition de bloc n'est pas remplie (en l'occurrence, le flanc montant de l'horloge). Ce qui veut dire que les signaux q et qb ne réagissent qu'au flanc de l'horloge. On dit de ces signaux qu'ils sont gardés. Contrairement à la première architecture, la deuxième ne modifie que le signal *etat_interne* au flanc de l'horloge, les deux signaux q et qb changent uniquement lors d'une variation de *etat_interne*.

Pour mieux comprendre le principe, un fichier de test et son résultat graphique sont donnés ci-dessous :

```

ENTITY flipflop_test IS
END flipflop_test;

ARCHITECTURE input_output OF flipflop_test IS

  COMPONENT
    flop PORT (d, c : IN bit;
              q, qb : OUT bit);
  END COMPONENT;

  FOR c1: flop USE ENTITY WORK.d_flipflop (assignement);
  FOR c2: flop USE ENTITY WORK.d_flipflop (guarding);

  SIGNAL dd, cc, q1, q2, qb1, qb2 : bit;

BEGIN

```

```

-- inversion de cc toutes les 400ns jusqu'a 2us
cc <= NOT cc AFTER 400 ns WHEN NOW < 2 us ELSE cc;
-- inversion de dd toutes les 1000ns jusqu'a 2us
dd <= NOT dd AFTER 1000 ns WHEN NOW < 2 us ELSE dd;
c1: flop PORT MAP (dd, cc, q1, qb1);
c2: flop PORT MAP (dd, cc, q2, qb2);
END input_output;

```

Résultat graphique :

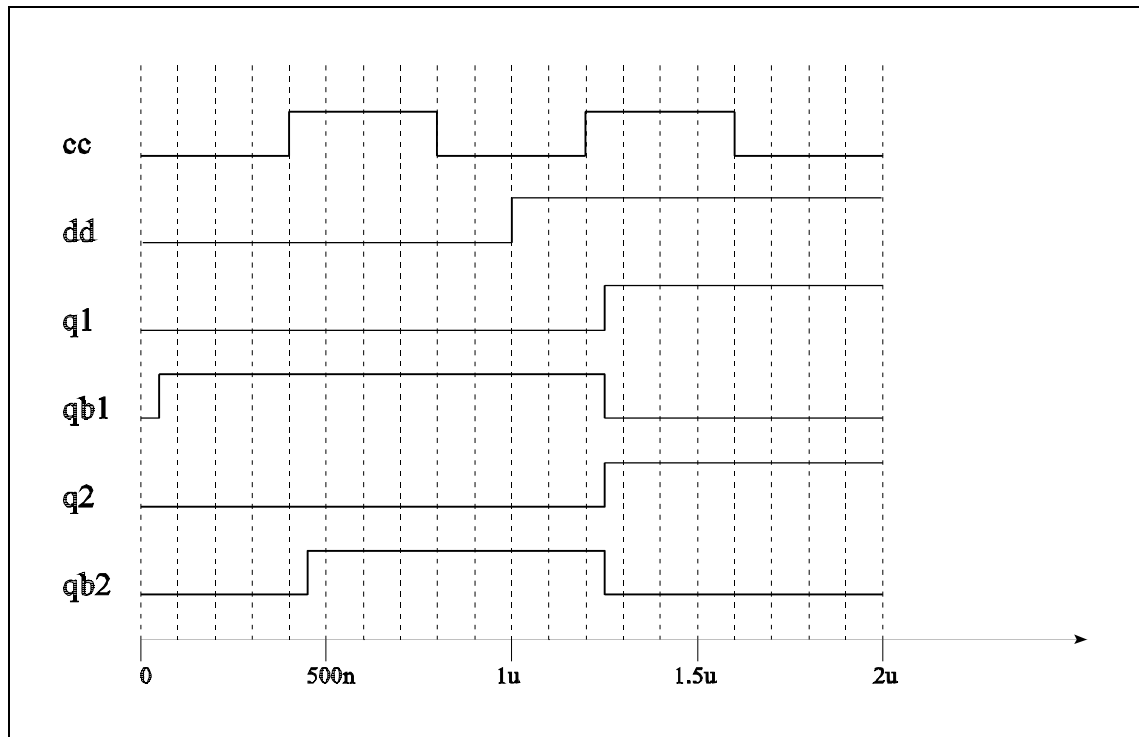


Image 23

Ce graphique révèle la différence entre qb1 et qb2 selon la description VHDL. le signal qb1 est bel est bien l'inverse du signal q1, contrairement à qb2 par rapport à q2.

Mais comme cela a déjà été dit, cette méthode pour décrire un système séquentiel n'est pas des plus lisible et n'est que peu employée, on l'évitera donc.

3.5.3 Description d'une machine séquentielle à partir d'un graphe d'états ou d'un organigramme

Un graphe d'états ou un organigramme se traduit tout naturellement en VHDL, en utilisant les structures **CASE** et **IF...THEN...ELSE**. La figure 24 montre la forme générale de l'architecture des décodeurs d'état futur et de sortie, la mémorisation de l'état étant décrite dans un processus séparé.

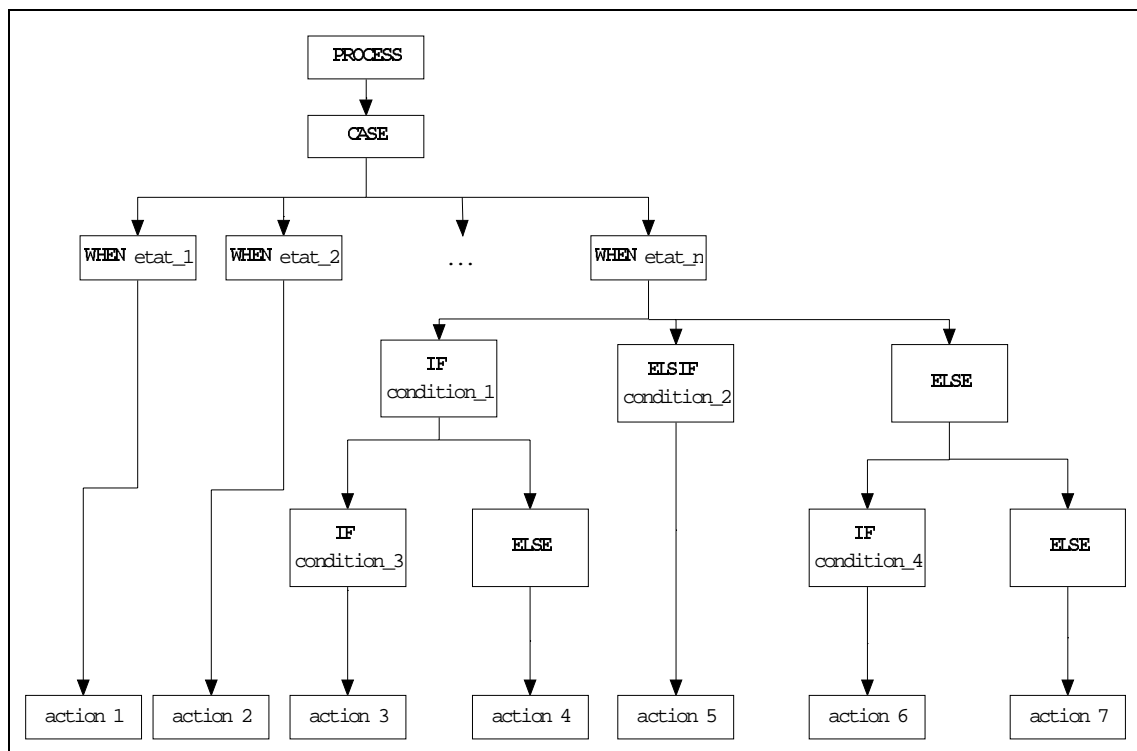


Image 24

Sous forme VHDL, cela donne :

```

ARCHITECTURE machine_etats OF entite_machine_etats IS

  TYPE etat IS (cas_1, ..., cas_n);

  SIGNAL present : etat := cas_1;  -- etat present, par default cas_1
  SIGNAL futur   : etat;          -- etat futur

  PROCESS (present, entrees)
  BEGIN
    CASE present IS
      WHEN cas_1 =>
        IF condition_d_entrees THEN                -- selon les entrees

```

```

        sorties <= valeur_a_mettre_sur_les_sorties;
        futur <= etat_futur_dans_ce_cas;
    ELSIF not condition_d_entrees THEN
        sorties <= valeur_a_mettre_sur_les_sorties;
        futur <= etat_futur_dans_ce_cas;
    ELSE
        valeur_avant_synchro <= maintien_de_la_sortie;
        futur <= etat_de_trappe;
    END IF;

    .
    .
    .

WHEN cas_n =>
    IF autre_condition_d_entrees then
        sorties <= valeur_a_mettre_sur_les_sorties;
        futur <= etat_futur_dans_ce_cas;
    ELSIF not autre_condition_d_entrees THEN
        sorties <= valeur_a_mettre_sur_les_sorties;
        futur <= etat_futur_dans_ce_cas;
    ELSE
        valeur_avant_synchro <= maintien_de_la_sortie;
        futur <= etat_de_trappe;
    END IF;
END CASE;
END PROCESS;
END machine_etats;

```

3.5.3.1 *Transitions inconditionnelles*

Regardons de manière plus précise les différentes transitions, conditionnelles ou non pour les coder en VHDL.

Une transition inconditionnelle s'écrit de la manière suivante :

```

WHEN etat_1 =>
    futur <= etat_futur_dans_ce_cas;

```

Aucune condition *if* n'est définie pour passer à l'état suivant. Par conséquent, une fois arrivé dans l'état 1, le décodeur d'état futur passera de toute manière à un seul état futur, celui indiqué à droite de l'assignation.

3.5.3.2 *Transitions conditionnelles*

Une transition conditionnelle se définit par l'instruction *if ... then ... else*. Il faut placer l'état futur dans la branche correspondant à sa condition.

```
WHEN etat_1 =>
  IF condition_d_entrees THEN
    futur <= etat_futur_dans_ce_cas;
  ELSIF autre_condition_d_entrees THEN
    futur <= etat_futur_dans_ce_cas;
  ELSE
    futur <= etat_de_trappe;
  END IF;
```

3.5.3.3 *Sorties inconditionnelles*

Quant aux sorties inconditionnelles, de la même manière que pour une transition inconditionnelle, on place directement à droite de l'assignation la valeur que la sortie doit prendre.

```
WHEN etat_1 =>
  sorties <= valeur_a_mettre_sur_les_sorties;
```

3.5.3.4 *Sorties conditionnelles*

Les sorties conditionnelles peuvent s'écrire sous deux formes. L'une où l'on donne la condition désirée en lieu et place d'une valeur, l'autre en insérant l'assignation de la sortie dans une branche correspondant à la condition souhaitée. Il est aussi possible de cumuler ces deux méthodes pour placer une condition sur les sorties, alors qu'elles se trouvent déjà dans une branche de condition.

```
WHEN etat_1 =>
  sorties <= condition_a_mettre_sur_les_sorties;
```

ou

```

WHEN etat_1 =>
  IF condition_d_entrees THEN                                -- selon les entrees
    sorties <= valeur_ou_condition_a_mettre_sur_les_sorties;
  ELSIF autre_condition_d_entrees THEN
    sorties <= autre_valeur_ou_condition_a_mettre_sur_les_sorties;
  ELSE
    valeur_avant_synchro <= maintien_de_la_sortie;
  END IF;

```

3.5.3.5 *Résumé*

Un résumé avec toutes les possibilités apparaît ci-dessous :

```

WHEN etat_1 =>
  sortie <= valeur_1;                                         -- sortie inconditionnelle
  sortie <= condition_1;                                     -- sortie conditionnelle
                                                           -- (condition 1)
  IF condition_2 THEN
    sortie <= valeur_2;                                       -- sortie conditionnelle
                                                           -- (condition 2)
    futur <= etat_futur_dans_ce_cas; -- transition conditionnelle
  ELSIF condition_3 THEN
    sortie <= condition_4;                                     -- sortie conditionnelle
                                                           -- (cond. 3 et 4)
    futur <= etat_futur_dans_ce_cas; -- transition conditionnelle
  ELSE
    valeur_avant_synchro <= maintien_de_la_sortie;
    futur <= etat_de_trappe;
  END IF;
WHEN etat_2 =>
  futur <= etat_futur_dans_ce_cas; -- transition inconditionnelle

```

3.5.3.6 *Sorties post-synchronisées*

Pour post-synchroniser une sortie, il faut créer une sortie intermédiaire qui sera ensuite placée sur l'entrée d'une bascule.

3.5.3.7 Exemple : détecteur de sens

L'exemple utilisé pour représenter une machine à états en VHDL est l'un des plus classiques, puisqu'il s'agit d'un détecteur de sens.

Son fonctionnement est le suivant :

Un disque peut tourner dans deux sens. Deux capteurs se trouvent d'un côté du disque pour lire la position de celui-ci. Il s'agit alors de déterminer dans quel sens tourne le disque.

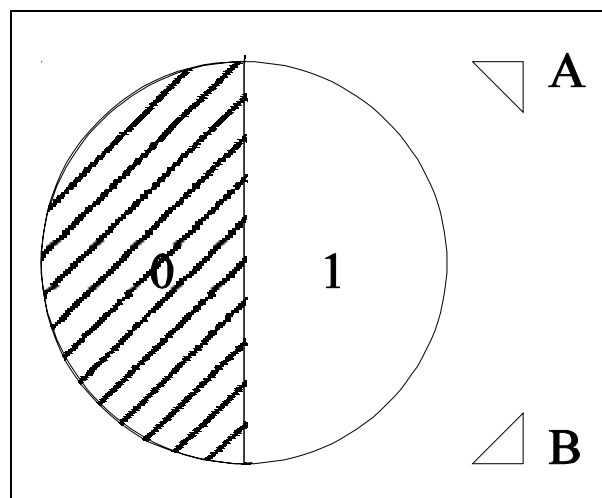


Image 25

La sortie est active tant qu'il tourne dans le sens anti-horaire. (Sortie de Mealy post-synchronisée active haute).

Il faut noter que le graphe d'états présenté ci-dessous n'est pas le graphe le plus optimal, mais il permet de présenter différents aspects du codage VHDL en un seul exemple.

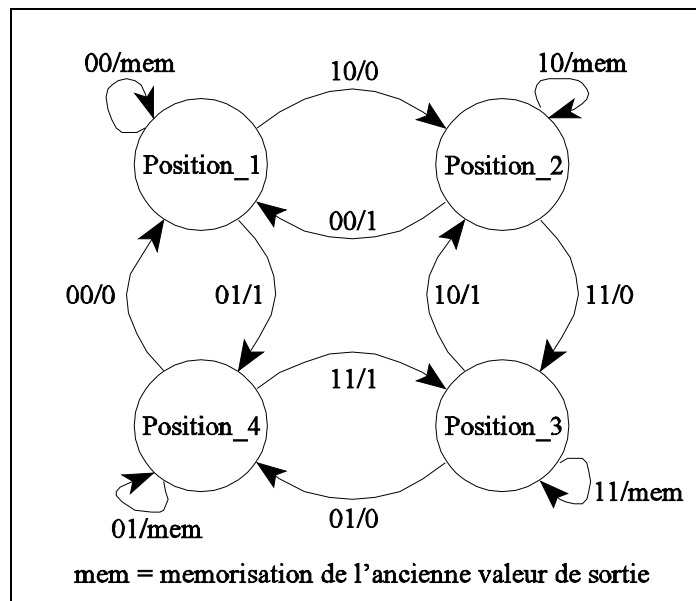


Image 26

```

-----
-- Nom du fichier : m_etats.vhd
-- Auteur       : C. Guex
-- Date        : avril 1997
-- Version     : 1.0
-- Modifications :
-- Auteur      :
-- Date       :
-----

```

```

-- But : exemple du cours VHDL d'une machine d'etats avec
--       reset asynchrone et set synchrone
-----

```

```

ENTITY detecteur_sens IS
  GENERIC (delai: time := 7 ns);-- delai de transfert d'une porte
  PORT (entree_a: IN bit;      -- entree a du detecteur de sens
        entree_b: IN bit;      -- entree b du detecteur de sens
        clk      : IN bit;      -- horloge de la bascule D
        reset    : IN bit;      -- reset asynch. de la bascule D
        set      : IN bit;      -- set synchrone de la bascule D
        sens_anti_horaire : OUT bit);-- sens de rotation
END detecteur_sens;

```

```

-----
-- Architecture comportementale d'une bascule D avec reset asynchrone
-- et set synchrone. Un enable autorise ou non le transfert de
-- l'entree D.
-----

```

```

ARCHITECTURE comportementale OF detecteur_sens IS

```

```

TYPE etat IS (position_1, position_2, position_4, position_3);

-- attribut existant sur certains compilateurs pour le codage des
-- etats, mais ceci ne fait pas partie de la norme vhdl
--type etat is (position_1, position_2, position_3, position_4);
--attribute enum_encoding of etat: type is
--  "00 " &          -- position_1
--  "01 " &          -- position_2
--  "11 " &          -- position_3
--  "10 " ;         -- position_4

SIGNAL present : etat := position_1; -- etat present,
-- depart: position_1
SIGNAL futur   : etat;              -- etat futur
SIGNAL sortie_bascule : bit;       -- valeur intermediaire que
-- prendra la sortie
SIGNAL valeur_avant_synchro : bit;  -- valeur de l'etat futur de la
-- bascule

BEGIN

-----
-- processus des decodeurs d'etat futur et de sortie
-----

PROCESS (present, entree_a, entree_b)
BEGIN
  CASE present IS
    WHEN position_1 =>
      IF entree_a = '1' AND entree_b = '0' THEN
        valeur_avant_synchro <= '0';
        futur <= position_2;
      ELSIF entree_a = '0' AND entree_b = '1' THEN
        valeur_avant_synchro <= '1';
        futur <= position_4;
      ELSE
        valeur_avant_synchro <= sortie_bascule;
        futur <= position_1;
      END IF;

    WHEN position_2 =>
      IF entree_a = '1' AND entree_b = '1' THEN
        valeur_avant_synchro <= '0';
        futur <= position_3;
      ELSIF entree_a = '0' AND entree_b = '0' THEN
        valeur_avant_synchro <= '1';
        futur <= position_1;
      ELSE
        valeur_avant_synchro <= sortie_bascule;
        futur <= position_2;
      END IF;

    WHEN position_3 =>
      IF entree_a = '0' AND entree_b = '1' THEN

```

```

        valeur_avant_synchro <= '0';
        futur <= position_4;
    ELSIF entree_a = '1' AND entree_b = '0' THEN
        valeur_avant_synchro <= '1';
        futur <= position_2;
    ELSE
        valeur_avant_synchro <= sortie_bascule;
        futur <= position_3;
    END IF;

    WHEN position_4 =>
        IF entree_a = '0' AND entree_b = '0' THEN
            valeur_avant_synchro <= '0';
            futur <= position_1;
        ELSIF entree_a = '1' AND entree_b = '1' THEN
            valeur_avant_synchro <= '1';
            futur <= position_3;
        ELSE
            valeur_avant_synchro <= sortie_bascule;
            futur <= position_4;
        END IF;
    END CASE;
END PROCESS;

```

```

-----
-- processus de memorisation de l'etat interne et de la post-synchro
-- de la sortie
-----

```

```

PROCESS
BEGIN
    WAIT UNTIL (clk'EVENT AND clk='1') OR (reset='1');
    IF reset = '1' THEN
        sortie_bascule <= '0' AFTER delai; -- reset asynchrone
                                           -- prioritaire
        present <= position_1 AFTER delai; -- retour a l'etat initial
    ELSE
        IF set = '1' THEN
            sortie_bascule <= '1' AFTER delai; -- set en 2eme priorite
            present <= position_1 AFTER delai; -- retour a l'etat initial
        ELSE
            sortie_bascule <= valeur_avant_synchro AFTER delai;
            present <= futur AFTER delai; -- transfert de l'etat
        END IF;
    END IF;
END PROCESS;

```

```

-----
-- En creant une variable intermediaire sortie_bascule, il est
-- possible de maintenir la valeur de la bascule tout en gardant
-- la variable sens_anti_horaire comme etant une sortie et non une
-- entree-sortie.
-- Il faut neanmoins assigner cette variable intermediaire a la sortie
-----

```

```
PROCESS (sortie_bascule)
BEGIN
    sens_anti_horaire <= sortie_bascule;
END PROCESS;

END comportementale;
```

On peut remarquer dans cet exemple qu'il est possible et même très recommandé de donner un nom mnémonique (un nom rappelant le rôle joué par cet état dans le fonctionnement de la MSS) à chaque état. Une fois ces noms choisis, il faut déclarer un type énuméré pour définir ces différents états. Pour le codage de ceux-ci, plusieurs compilateurs offrent une fonction `enum_encoding`, malheureusement, elle n'est pas standard et par conséquent n'est pas utilisable partout. Si elle n'existe pas, il faut placer les états dans l'ordre de leur code dans la déclaration du type (A VERIFIER).

De même, un état d'initialisation peut être choisi, ainsi qu'un état de trappe. Les états non utilisés peuvent aussi avoir des noms prévus.

S'il faut décrire plusieurs machines d'états en parallèle, il vaut mieux créer des fichiers VHDL pour chacune des machines et ensuite utiliser une description structurelle pour lier le tout.

3.5.4 Description structurelle

Il n'y a aucune différence entre une description structurelle d'un système séquentiel et celle d'un système combinatoire. Un tel type de description n'a normalement un sens que pour un système modulaire. L'exemple sera celui d'une cascade de compteurs 1 bit pour réaliser un compteur 4 bits.

Élément se trouvant en bibliothèque : (compteur 1 bit)

```
ENTITY compt_1_bit IS
    PORT (horloge      : IN  bit;
          report_prec  : IN  bit;
          sortie       : OUT bit;
          report_suiv  : OUT bit);
END compt_1_bit;

ARCHITECTURE flot_de_donnees OF compt_1_bit IS

    SIGNAL int_sortie : bit;
    SIGNAL int_report_suiv : bit;

BEGIN
```

```

int_report_suiv <= int_sortie AND report_prec;
int_sortie <= int_sortie XOR report_prec WHEN
    (horloge = '1' AND NOT horloge'STABLE) ELSE int_sortie;
report_suiv <= int_report_suiv;
sortie <= int_sortie;
END flot_de_donnees;

```

Compteur 4 bits :

```

ENTITY compt_4_bits IS
  PORT (horloge      : IN bit;
         report_in    : IN bit;                               -- enable du
compteur
         valeur_compteur : OUT bit_vector (3 DOWNTO 0);
         report_out     : OUT bit);
END compt_4_bits;

```

```

ARCHITECTURE structurelle OF compt_4_bits IS

```

```

  COMPONENT bloc1 PORT (horloge, report_prec : IN bit;
                        sortie, report_suiv : OUT bit);
  END COMPONENT;

```

```

FOR ALL : bloc1 USE ENTITY WORK.compt_1_bit (compt_flux_de_donnees);

```

```

SIGNAL report0, report1, report2 : bit;

```

```

BEGIN

```

```

  g0 : bloc1 PORT MAP
    (horloge, report_in, valeur_compteur(0), report0);

```

```

  g1 : bloc1 PORT MAP
    (horloge, report0, valeur_compteur(1), report1);

```

```

  g2 : bloc1 PORT MAP
    (horloge, report1, valeur_compteur(2), report2);

```

```

  g3 : bloc1 PORT MAP
    (horloge, report2, valeur_compteur(3), report_out);

```

```

END structurelle;

```

Voilà les différents principes de base pour décrire un circuit en VHDL. Mais ce n'est pas tout, car l'un des intérêts majeurs du langage est la possibilité de simuler par le même langage les différents modules. Sur les quelques pages suivantes, nous verrons donc les bases pour créer un fichier de test pour la simulation.

3.6 LA SIMULATION

Pour être efficace, une simulation doit être la plus complète possible, sans être redondante. Une bonne simulation épurée de tous les tests doublés permettra une bonne économie tant pour le temps consacré à cette simulation que pour les tests sur circuit ensuite. De plus, le fichier écrit pour la simulation peut être avantageusement utilisé pour tous les tests ultérieurs (sur le prototype, en fabrication, etc).

Comme précité, un système de test fournissant les stimuli et vérifiant les réponses peut aussi être écrit en VHDL. Un module de simulation (test bench) sera créé et à l'intérieur de celui-ci, nous intégrerons le module à tester. En code VHDL, nous aurons ceci :

```

ENTITY sim_nom_entite IS
END sim_nom_entite;

-----
-- Architecture de la simulation
-----

ARCHITECTURE simulation OF sim_nom_entite IS
  COMPONENT nom_composant PORT (entrees           : IN type_entrees;
                                   entrees_sorties : INOUT
                                   type_entree_sortie;
                                   sorties         : OUT type_sorties);
  END COMPONENT;

FOR    ALL    :nom_composant    USE    ENTITY    WORK.ent_mod_a_tester
(arch_mod_a_tester);

BEGIN

  a1: nom_composant PORT MAP (entrees, entrees_sorties, sorties);

  PROCESS
    -- simulation
  END PROCESS;

END simulation;

```

Pour tester plusieurs parties distinctes d'un même fichier VHDL, il est possible de créer plusieurs processus de simulation. Par contre, il est préférable pour une meilleure compréhension d'avoir un déroulement temporel pour tous les signaux ayant un rapport entre eux. Expliquons-nous : en VHDL, il est possible de donner toutes les caractéristiques temporelles d'un signal en une seule fois, ce qui permet de décrire toutes les entrées de manière parallèle comme l'exemple ci-dessous :

```

ARCHITECTURE simulation_par_signal OF sim_nom_entite IS

    SIGNAL entree_1 : bit := '0';
    SIGNAL entree_2 : bit := '0';
    SIGNAL entree_3 : bit := '1';

BEGIN

    entree_1 <= '0', '1' AFTER 23 ns, '1' AFTER 35 ns, '1' AFTER 45 ns;
    entree_2 <= '1' AFTER 17 ns, '0' AFTER 35 ns, '1' AFTER 52 ns;
    entree_3 <= '0', '1' AFTER 10 ns, '0' AFTER 14 ns, '1' AFTER 25 ns;

END simulation_par_signal;

```

Malheureusement, il est très difficile de se retrouver avec une telle simulation, car il faut vérifier tous les signaux pour voir s'il y a un changement ou non. La chronologie des événements n'est absolument pas évidente et il faudrait reclasser les signaux pour en avoir une idée (1ère action sur les entrées 1 et 3, puis deux fois sur la 3, ensuite une action sur l'entrée 2 pour continuer par l'entrée 1, etc.). Il vaut donc mieux avoir une simulation chronologique, parfois plus longue à écrire, mais nettement plus compréhensible et réutilisable. Cette simulation peut se trouver sous deux formes, l'une référencée au temps t_0 et l'autre au dernier événement qui a eu lieu.

Code VHDL pour une simulation chronologique référencée au temps t_0 :

```

ENTITY sim_nom_entite IS
END sim_nom_entite;

ARCHITECTURE simulation_chronologique_t0 OF sim_nom_entite IS

    SIGNAL entree_1 : bit := '0';
    SIGNAL entree_2 : bit := '0';
    SIGNAL entree_3 : bit := '1';

BEGIN

    PROCESS
    BEGIN
        entree_1 <= TRANSPORT '0';
        entree_3 <= TRANSPORT '0';
        entree_3 <= TRANSPORT '1' AFTER 10 ns;
        entree_3 <= TRANSPORT '0' AFTER 14 ns;
        entree_2 <= TRANSPORT '1' AFTER 17 ns;
        entree_1 <= TRANSPORT '1' AFTER 23 ns;
        entree_3 <= TRANSPORT '1' AFTER 25 ns;
        entree_1 <= TRANSPORT '0' AFTER 35 ns;
        entree_2 <= TRANSPORT '0' AFTER 35 ns;
        entree_1 <= TRANSPORT '1' AFTER 45 ns;
        entree_2 <= TRANSPORT '1' AFTER 52 ns;
        WAIT;
    END PROCESS;

END simulation_chronologique_t0;

```

Code VHDL pour une simulation chronologique référencée au dernier événement :

```
ENTITY sim_nom_entite IS
END sim_nom_entite;

ARCHITECTURE simulation_chronologique OF sim_nom_entite IS

    SIGNAL entree_1 : bit := '0';
    SIGNAL entree_2 : bit := '0';
    SIGNAL entree_3 : bit := '1';

BEGIN
    PROCESS
    BEGIN
        entree_1 <= '0';
        entree_3 <= '0';
        WAIT FOR 10 ns;
        entree_3 <= '1';
        WAIT FOR 4 ns;
        entree_3 <= '0';
        WAIT FOR 3 ns;
        entree_2 <= '1';
        WAIT FOR 6 ns;
        entree_1 <= '1';
        WAIT FOR 2 ns;
        entree_3 <= '1';
        WAIT FOR 10 ns;
        entree_1 <= '1';
        entree_2 <= '0';
        WAIT FOR 10 ns;
        entree_1 <= '1';
        WAIT FOR 7 ns;
        entree_2 <= '1';
    END PROCESS;

END simulation_chronologique;
```

Si la première forme semble plus lisible, c'est la deuxième qui est utilisée pour créer des procédures facilitant une simulation par cycles d'horloge. En effet, pour faciliter la simulation, deux procédures et un processus peuvent être créés. Une procédure pour définir un cycle d'horloge, l'autre pour définir le pas de simulation, et le processus pour définir la séquence d'un cycle d'horloge. Les simulations pourront alors s'écrire sous la forme suivante :

```
PROCESS
BEGIN

    cycle; -- cycle inactif obligatoire

    entree_1 <= 'valeur_de_l_entree';
```



```

    entree_2 <= 'valeur_de_l_entree';

sim(10);    -- pas de simulation de duree 10 * l'unite minimale

    controle_sortie_1 <= sortie_1 XOR 'valeur_desiree_de_la_sortie';
    controle_sortie_2 <= sortie_2 XOR 'valeur_desiree_de_la_sortie';
    entree_1 <= 'valeur_de_l_entree';
    entree_3 <= 'valeur_de_l_entree';

cycle;      -- termine le cycle d'horloge commence par le sim

    controle_sortie_1 <= sortie_1 XOR 'valeur_desiree_de_la_sortie';
    controle_sortie_2 <= sortie_2 XOR 'valeur_desiree_de_la_sortie';
    entree_2 <= 'valeur_de_l_entree';

cycle;      -- cycle d'horloge

    controle_sortie_1 <= sortie_1 XOR 'valeur_desiree_de_la_sortie';
    controle_sortie_2 <= sortie_2 XOR 'valeur_desiree_de_la_sortie';
    entree_1 <= 'valeur_de_l_entree';
    entree_2 <= 'valeur_de_l_entree';

cycle(5);   -- 5 cycles d'horloge

    controle_sortie_1 <= sortie_1 XOR 'valeur_desiree_de_la_sortie';
    controle_sortie_2 <= sortie_2 XOR 'valeur_desiree_de_la_sortie';

sim_end <= true; -- fin de la simulation

END PROCESS;

```

Remarque : le premier cycle est inactif et il faut le mettre. Il sert à se positionner correctement au début de la simulation.

Les signaux à déclarer et les procédures sont les suivantes (attention, seul ce qui est en gras peut être modifié) :

```

CONSTANT periode : time := 100 ns;  -- periode d'horloge
CONSTANT unite_min : time := 1 ns;  -- unite de temps minimale

SIGNAL sim_end : boolean := false;  -- fin de la simulaion
SIGNAL horloge : bit;               -- signal d'horloge
SIGNAL debut_cycle : bit;          -- indique le debut d'un cycle d'horloge

-----
-- Permet des pas de simulation avec l'unite de temps minimale
-----

PROCEDURE sim (nombre_d_unites_de_temps : Integer := 1) IS
BEGIN

```

```

    WAIT FOR (nombre_d_unites_de_temps * unite_min);
END sim;

-----
-- Procedure permettant de laisser passer plusieurs cycles d'horloge
-- Le premier appel de la procedure termine le cycle precedent si
-- celui-ci n'etait pas complet (par exemple : si on a fait quelques
-- pas de simulation non synchronises avant, reset asynchrone,
-- combinatoire, ...)
-----

PROCEDURE cycle (nombre_de_cycles : Integer := 1) IS
BEGIN
    FOR i IN 1 TO nombre_de_cycles LOOP
        WAIT UNTIL debut_cycle'EVENT AND debut_cycle = '1';
    END LOOP;
END cycle;

BEGIN

-----
-- ce processus genere une horloge pour la simulation
-----

PROCESS
BEGIN
    IF NOT sim_end THEN
        horloge <= '0',
            '1' AFTER periode / 4,
            '1' AFTER 2 * periode / 4,
            '0' AFTER 3 * periode / 4;
        debut_cycle <= '1',
            '0' AFTER unite_min;
        WAIT FOR periode;
    ELSE WAIT;
    END IF;
END PROCESS;

```

Pour illustrer ces fichiers de simulation, un fichier de test pour le détecteur de sens vu auparavant est donné ci-dessous :

```

-----
-- Nom du fichier : m_etats.vhd
-- Auteur          : C. Guex
-- Date           : avril 1997
-- Version        : 1.0
-- Modifications  :
-- Auteur         :                               Date :
-----
-- But           : exemple du cours VHDL d'une simulation d'une
--                machine d'etats avec reset asynchrone et set
--                synchrone

```

```
-----  
ENTITY sim_detecteur_sens IS  
END sim_detecteur_sens;  
  
-----  
-- Architecture du module de simulation du detecteur de sens  
-----  
ARCHITECTURE simul_det_sens OF sim_detecteur_sens IS  
  COMPONENT detect_sens PORT (entree_a, entree_b, clk, reset, set : IN  
bit;  
                                sens : OUT bit);  
  END COMPONENT;  
  
  FOR ALL : detect_sens USE ENTITY WORK.detecteur_sens (det_sens_etats);  
  
  CONSTANT periode : time := 100 ns;  
  CONSTANT unite_min : time := 1 ns;  
  
  SIGNAL sim_end : boolean := false;  
  SIGNAL entree_a, entree_b : bit;  
  SIGNAL set, reset : bit;  
  SIGNAL horloge : bit;      -- signal d'horloge  
  SIGNAL debut_cycle : bit;  -- indique le debut d'un cycle d'horloge  
  SIGNAL verif_sortie : bit;  
  SIGNAL erreur : bit;  
  
-----  
-- Permet des pas de simulation avec l'unite de temps minimale  
-----  
PROCEDURE sim (nombre_d_unites_de_temps : Integer := 1) IS  
BEGIN  
  WAIT FOR (nombre_d_unites_de_temps * unite_min);  
END sim;  
  
-----  
-- Procedure permettant plusieurs cycles d'horloge  
-- Le premier appel de la procedure termine le cycle precedent si  
-- celui-ci n'etait pas complet (par exemple : si on a fait quelques  
-- pas de simulation non synchronises avant, reset asynchrone,  
-- combinatoire, ...)  
-----  
PROCEDURE cycle (nombre_de_cycles : Integer := 1) IS  
BEGIN  
  FOR i IN 1 TO nombre_de_cycles LOOP  
    WAIT UNTIL debut_cycle'EVENT AND debut_cycle = '1';  
  END LOOP;  
END cycle;  
  
BEGIN
```

```
-----  
-- ce processus genere une horloge pour la simulation  
-----  
  
PROCESS  
BEGIN  
  IF NOT sim_end THEN  
    horloge <= '0',  
              '1' AFTER periode / 4,  
              '1' AFTER 2 * periode / 4,  
              '0' AFTER 3 * periode / 4;  
    debut_cycle <= '1',  
                  '0' AFTER unite_min;  
    WAIT FOR periode;  
  ELSE WAIT;  
  END IF;  
END PROCESS;  
  
a1: detect_sens PORT MAP  
    (entree_a, entree_b, horloge, reset, set, verif_sortie);  
  
PROCESS  
BEGIN  
  
-- Attention, le premier cycle doit etre double pour un fonctionnement  
-- correct ou alors, le plus simple est de toujours commencer par un  
-- cycle inactif  
  
  cycle; -- cycle inactif  
  
  reset <= '1';  
  sim(10);  
  reset <= '0';  
  cycle;  
  set <= '0';  
  entree_a <= '0';  
  entree_b <= '0';  
  cycle;  
  entree_a <= '0';  
  entree_b <= '1';  
  cycle(4);  
  erreur <= verif_sortie XOR '1';  
  entree_a <= '1';  
  entree_b <= '1';  
  cycle;  
  erreur <= verif_sortie XOR '1';  
  entree_a <= '1';  
  entree_b <= '0';  
  cycle;  
  erreur <= verif_sortie XOR '1';  
  entree_a <= '0';  
  entree_b <= '0';  
  cycle;  
  erreur <= verif_sortie XOR '1';
```

```
    entree_a <= '1';
    entree_b <= '0';
cycle;
    erreur <= verif_sortie XOR '0';
    entree_a <= '1';
    entree_b <= '1';
cycle;
    erreur <= verif_sortie XOR '0';
    entree_a <= '0';
    entree_b <= '1';
cycle;
    erreur <= verif_sortie XOR '0';
    entree_a <= '0';
    entree_b <= '0';
cycle;
    erreur <= verif_sortie XOR '0';

sim_end <= true;

END PROCESS;

END simul_det_sens;
```

Pour tester les sorties, la solution la plus simple est d'utiliser une fonction ou exclusif et de comparer la valeur réelle de la sortie par rapport à la valeur souhaitée. La valeur du signal *erreur* devrait toujours valoir 0. Nous aurons donc :

```
erreur <= sortie XOR 'valeur souhaitee';
```

Note : Un paquetage a été créé afin de faciliter la simulation. On le retrouve dans le chapitre consacré aux fichiers de simulation.

3.7 POLARITÉ DES ENTRÉES - SORTIES

La polarité des signaux en VHDL peut s'écrire de manière très simple si l'on se donne une norme. Pour décrire le fonctionnement d'un système, il est beaucoup plus aisé d'écrire en polarité positive, plutôt qu'en polarité mixte. C'est donc dans cette optique qu'une interface peut être créée entre le module décrivant le fonctionnement du système (écrit en polarité positive) et le monde extérieur.

La représentation graphique de cette interface est la suivante :

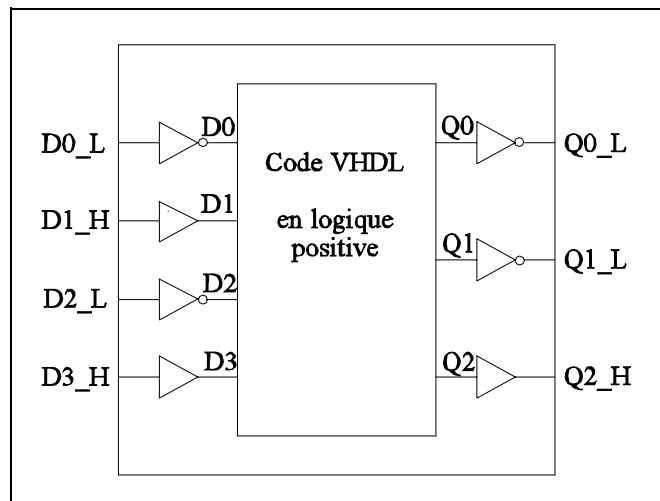


Image 27

et le code VHDL est celui-ci :

```

-----
-- Nom du fichier : polarite.vhd
-- Auteur       : C. Guex
-- Date        : mai 1997
-- Version     : 1.0
-- Modifications :
-- Auteur      :
-- Date       :
-----
-- But          : interface pour polariser les signaux
-----

ENTITY module_exterieur IS
    PORT (d0_L : IN bit;   -- entree 0 active basse
          d1_H : IN bit;   -- entree 1 active haute
          d2_L : IN bit;   -- entree 2 active basse
          d3_H : IN bit;   -- entree 3 active haute
          q0_L : OUT bit;  -- sortie 0 active basse
          q1_L : OUT bit;  -- sortie 1 active basse
          q2_H : OUT bit); -- sortie 2 active haute
END module_exterieur;

-----
-- Architecture pour la polarite des signaux
-----

ARCHITECTURE polarite OF module_exterieur IS

    COMPONENT module_interieur
        PORT (d0, d1, d2, d3 : IN bit;
              q0, q1, q2     : OUT bit);
    END COMPONENT;

    FOR ALL : module_interieur

```

```

        USE ENTITY WORK.ent_module_interieur (arch_module_interieur);

SIGNAL d3, d2, d1, d0, q2, q1, q0 : bit;

BEGIN

    poll : module_interieur PORT MAP (d0, d1, d2, d3, q0, q1, q2);

    -- entrees
    d0 <= NOT d0_L;
    d1 <=    d1_H;
    d2 <= NOT d2_L;
    d3 <=    d3_H;

    -- sorties
    q0_L <= NOT q0;
    q1_L <= NOT q1;
    q2_H <=    q2;

END polarite;

```

Comme il est possible de le voir ci-dessus, cette norme est la suivante :

- aucun des signaux d'un module intérieur ne porte d'indication de sa polarité.
- tous les modules intérieurs sont décrits en polarité positive.
- une interface est créée pour désigner la polarité d'un signal.
- le module intérieur est englobé dans l'interface.
- vis de l'extérieur, tous les signaux ont leur polarité indiquée dans le nom du signal de la manière suivante :

<i>nom_du_signal_</i> H	<i>pour une polarité positive</i>
<i>nom_du_signal_</i> L	<i>pour une polarité négative</i>

- si les deux polarités d'un signal sont utilisées à l'extérieur du module, alors on créera 2 signaux à polarité inverse dans l'interface, mais on n'utilisera pas les signaux du module intérieur (ex: *clk_L* et *clk_H*).

4 *STRUCTURE DU VHDL*

4.1 *STRUCTURE GÉNÉRALE D'UN PROGRAMME VHDL*

Un programme VHDL se structure de la manière suivante :

```
-- -----  
-- declaration et utilisation des bibliotheques necessaires      -  
-- -----  
  
USE work.bibliotheques_necessaires.ALL;  
  
-- -----  
-- declaration de l'entite et de ses ports d'entree-sortie      -  
-- -----  
  
ENTITY nom_de_l_entite IS  
    GENERIC (parametres_generiques: type := Valeur_par_defaut);  
    PORT (ports_d_entree : IN type_ports_d_entree;  
          ports_de_sortie : OUT type_ports_de_sortie;  
          ports_d_entree_sortie : INOUT type_ports_entree_sortie);  
END nom_de_l_entite;  
  
-- -----  
-- architecture du programme, structure interne                  -  
-- -----  
  
ARCHITECTURE type_de_description OF nom_de_l_entite IS  
BEGIN  
    -- programme interne  
  
END type_de_description;
```


4.2 LES BIBLIOTHÈQUES

Définition : Une *bibliothèque* est le regroupement d'un ensemble d'éléments destinés à être réutilisés dans un développement ultérieur.

Déclaration :

```
USE work.bibliotheques_necessaires.ALL;
```

4.2.0.1 *Que trouve-t-on dans une bibliothèque ?*

Le but d'une bibliothèque est de regrouper de façon organisée un ensemble d'éléments (entités, paquetages) de manière à pouvoir les utiliser dans nos programmes VHDL. C'est en quelque sorte un magasin de pièces.

Il existe deux types de bibliothèques :

- les bibliothèques de l'utilisateur (user library) : Il s'y trouve tout ce qu'il a créé et analysé en VHDL. L'utilisateur crée lors de chaque projet sa (ou ses) bibliothèque(s) utilisateur.
- Les bibliothèques système : Elles font partie du système logiciel mis à disposition de l'utilisateur, et sont exploitables dans tout projet.

4.2.0.2 *Comment déclare-t-on une bibliothèque dans un fichier VHDL ?*

Pour utiliser des éléments dans un fichier VHDL, il faut d'abord savoir si l'on veut utiliser un élément de notre espace de travail (fichier VHDL créé par l'utilisateur) ou si l'on veut utiliser un élément qui se trouve dans une bibliothèque externe.

4.2.0.2.1 *L'élément à utiliser fait partie de notre espace de travail (projet).*

En VHDL, on assimile l'espace de travail à une bibliothèque (bien qu'il n'en soit pas une). Pour y utiliser un élément, la plupart des compilateurs VHDL emploient le mot **WORK** comme nom de bibliothèque. Dès lors, tous les éléments créés par l'utilisateur peuvent être réutilisés dans un autre fichier VHDL du même projet : il suffit de déclarer les éléments de la bibliothèque WORK que l'on désire utiliser.

Par exemple, si l'utilisateur a créé un paquetage (fichier VHDL dans lequel on trouve une série de variables, constantes, signaux, fonctions et procédures), il peut l'utiliser en déclarant la ligne suivante au début de son nouveau fichier VHDL :

```
USE WORK.nom_du_paquetage.ALL;
```

Lorsque c'est un élément que l'on désire utiliser, la déclaration est la suivante :

```
FOR ALL : nom_element USE ENTITY WORK.entite_element(arch_element);
```

Note : Certains compilateurs placent le résultat de l'analyse dans une bibliothèque préalablement créée par l'utilisateur. Dès ce moment, s'il compile un paquetage, celui-ci se retrouvera dans l'espace de travail, mais aussi dans la bibliothèque créée par l'utilisateur. Pour utiliser un élément de son paquetage, il aura deux manières de le déclarer :

```
USE WORK.nom_du_paquetage.ALL;
```

ou

```
USE nom_bibliotheque_utilisateur.nom_du_paquetage.ALL;
```

Exemple :

```
USE WORK.paquetage_portes_logiques.ALL;
```

ou

```
USE user.paquetage_portes_logiques.ALL;
```

4.2.0.2.2 L'élément à utiliser fait partie d'une autre bibliothèque

Pour utiliser un élément d'une bibliothèque, il faut d'abord déclarer celle-ci dans notre fichier VHDL, puis il faut spécifier quels sont les parties de la bibliothèque à utiliser.

Ce qui donne le code suivant :

```
LIBRARY nom_de_bibliotheque;
```

```
USE nom_de_bibliotheque.nom_du_paquetage.ALL;
```

Exemple pour utiliser le paquetage de simulation (la description suit dans quelques pages):

```
LIBRARY einev;
```

```
USE einev.simulation_pkg.ALL;
```

5 FICHIERS DE SIMULATION DES EXEMPLES

5.1 PAQUETAGE SIMULATION

Ce paquetage a été créé pour faciliter la simulation en VHDL et la rendre plus structurée. Plusieurs de ces fonctions et procédures sont utilisées dans les exemples de simulation qui se trouveront ensuite.

5.1.1 Procédures et fonctions utilisables

Nous trouverons ci-dessous toutes les procédures et fonctions utilisables dans ce paquetage:

Les éléments entre crochets [] sont des éléments optionnels. Il n'est pas nécessaire de les mettre.

sim;

sim avance d'un pas de simulation. La valeur par défaut est de 100ns.

sim(a);

sim exécute un certain nombre de pas de simulation.

a nombre de pas de simulation.

Exemple : *sim(5);*

sim(b unite_temps);

sim exécute un pas de simulation.
b durée du pas de simulation.
unite_temps unité de temps d'un pas de simulation.

Exemple : `sim(10 ns);`

`sim(a, b unite_temps);`

sim exécute un certain nombre de pas de simulation.
a nombre de pas de simulation.
b durée du pas de simulation.
unite_temps unité de temps d'un pas de simulation.

Exemple : `sim(5,10 ns);`

`cycle;`

cycle Exécute un cycle d'horloge. La forme du cycle doit être spécifiée dans le fichier de simulation. Dans le cas où l'on se trouvait au milieu d'un cycle, cette commande terminera le cycle en cours.

`cycle(n);`

cycle Exécute un certain nombre de cycles d'horloge. La forme du cycle doit être spécifiée dans le fichier de simulation. Dans le cas où l'on se trouvait au milieu d'un cycle, cette commande terminera le cycle en cours.

n nombre de cycles d'horloge effectués.

Exemple : `cycle(5);`

`echo("texte" [, "nom_de_fichier"]);`

echo	Commande d'écriture dans un fichier.
"texte"	Texte que l'on désire écrire dans le fichier.
"nom_de_fichier"	Nom du fichier dans lequel on désire écrire le texte. Cet élément est optionnel. S'il n'est pas indiqué, le fichier par défaut est <i>resultat.txt</i> .

Exemples : `echo("Debut de la simulation");`
`echo("Fin","compteur.txt");`

`echo(valeur [, "nom_de_fichier"]);`

echo	Commande d'écriture dans un fichier.
valeur	Valeur d'un signal que l'on désire écrire dans le fichier.
"nom_de_fichier"	Nom du fichier dans lequel on désire écrire la valeur du signal. Cet élément est optionnel. S'il n'est pas indiqué, le fichier par défaut est <i>resultat.txt</i> .

Exemples : `echo(etat0);`
`echo('1');`
`echo(etat0,"compteur.txt");`

`echo("texte", valeur [, "nom_de_fichier"]);`

echo	Commande d'écriture dans un fichier.
"texte"	Texte que l'on désire écrire dans le fichier.
valeur	Valeur d'un signal que l'on désire écrire dans le fichier.
"nom_de_fichier"	Nom du fichier dans lequel on désire écrire. Cet élément est optionnel. S'il n'est pas indiqué, le fichier par défaut est <i>resultat.txt</i> .

Exemples : `echo("bit d'etat 0 = ", etat0);`
`echo("entree = ", '1');`
`echo("bit d'etat 0 = ", etat0, "compteur.txt");`

```
echo_v(vecteur [, string'("nom_de_fichier"))];
```

echo_v Commande d'écriture dans un fichier.

vecteur Vecteur que l'on désire écrire dans le fichier.

"nom_de_fichier" Nom du fichier dans lequel on désire écrire la valeur du signal. Cet élément est optionnel. S'il n'est pas indiqué, le fichier par défaut est *resultat.txt*.

Exemples :

```
echo_v(vecteur_a);
echo_v ("10001");
echo_v (vecteur_a, "compteur.txt");
echo_v ("10011", string' ("compteur.txt"));
```

```
echo_v("texte ", vecteur [, "nom_de_fichier"]);
```

echo_v Commande d'écriture dans un fichier.

vecteur Vecteur que l'on désire écrire dans le fichier.

"nom_de_fichier" Nom du fichier dans lequel on désire écrire la valeur du signal. Cet élément est optionnel. S'il n'est pas indiqué, le fichier par défaut est *resultat.txt*.

Exemples :

```
echo_v("le vecteur a vaut : ", vecteur_a);
echo_v ("le vecteur a = ", vecteur_a, "compteur.txt");
```

Attention : il ne faut pas mettre

```
echo_v("le vecteur a = ", "0010");
```


mais

```
echo("le vecteur a = 0010");
```

```
vecteur_a xor vecteur_b;
```

xor Opérateur logique **xor** utilisable pour des vecteurs. L'opération ou exclusif se fait bit à bit. Les dimensions du vecteur de sortie sont celles du premier vecteur. De ce fait, il est possible d'écrire `vecteur_a <= vecteur_a xor vecteur_b;` mais pas `vecteur_b <= vecteur_a xor vecteur_b;`

vecteur_a Premier vecteur.

vecteur_b Deuxième vecteur.

Exemples :

```
somme <= compteur xor "1001";
somme <= "0011" xor compteur;
somme <= compteur_a xor compteur_b;
vecteur_a <= vecteur_a xor vecteur_b;
```

```
check (vecteur_a, valeur_a_tester [, "nom_de_fichier"]);
```

check Commande indiquant dans un fichier s'il y a une différence entre une valeur ou un vecteur calculé dans un fichier VHDL et un résultat supposé par l'utilisateur. Attention, la valeur calculée doit toujours se mettre en premier.

vecteur_a Valeur ou vecteur de sortie d'un fichier VHDL.

valeur_a_tester Valeur que l'on désire comparer à la valeur ou au vecteur précédent.

"nom_de_fichier" Nom du fichier dans lequel on désire écrire. Cet élément est optionnel. S'il n'est pas indiqué, le fichier par défaut est *resultat.txt*.

Exemples :

```
check (compteur, "1001");
check (compteur, "1001", "compteur.sim");
```

```
indiquer_si_erreur(vecteur [, "nom_de_fichier"]);
```

indiquer_si_erreur Ecrit dans un fichier "en ordre" si le vecteur à tester vaut zéro et "erreur sur le pas de simulation précédant" si le vecteur à tester est non nul. S'utilise après une vérification des valeurs de sortie par un XOR.

vecteur Vecteur à tester.

"nom_de_fichier" Nom du fichier dans lequel on désire écrire. Cet élément est optionnel. S'il n'est pas indiqué, le fichier par défaut est *resultat.txt*.

Exemples : `indiquer_si_erreur(somme);`
`indiquer_si_erreur(somme, "compteur.txt");`

`cmp_vect`(vecteur_a, vecteur_b [, ecrire [, "nom_de_fichier"]]);

<code>cmp_vect</code>	Compare deux vecteurs. S'ils sont égaux, le résultat de cette fonction sera un signal de type bit et de valeur '0' et s'il ne sont pas égaux, la valeur restituée sera '1'. De plus, si l'utilisateur le désire, il écrit dans un fichier si les vecteurs sont égaux ou non.
<code>vecteur_a</code>	Premier vecteur à comparer.
<code>vecteur_b</code>	Deuxième vecteur à comparer.
<code>ecrire</code>	Indique si l'on veut écrire le résultat dans un fichier ou non. Pour écrire dans un fichier, il faut spécifier <i>oui</i> . Cette valeur est optionnelle et ne s'indique pas si l'on ne désire pas écrire le résultat dans un fichier.
<code>"nom_de_fichier"</code>	Nom du fichier dans lequel on désire écrire. Cet élément est optionnel. S'il n'est pas indiqué, le fichier par défaut est <i>resultat.txt</i> .

Exemples : `test <= cmp_vect (etat, "1001");`
`test <= cmp_vect (etat, "1001", oui);`
`test <= cmp_vect (etat, "1001", oui, "compteur.txt");`

5.1.2 Constantes

<code>pas_de_simulation_min</code>	est la durée par défaut d'un pas de simulation. Elle a été définie à 100 ns .
<code>nom_fichier</code>	représente le nom du fichier par défaut pour tous les enregistrements. Ce nom par défaut est resultat.txt .
<code>unite_de_temps_min</code>	est l'unité de temps minimale dans la simulation. Elle a été définie à 1 ns .

5.1.3 *Types*

accepter est un type énuméré avec les deux valeurs suivantes : **oui**, **non**.

5.1.4 *Signaux*

debut_cycle est un signal qui doit être utilisé pour des simulations séquentielles.

5.2 *CODE DU PAQUETAGE DE SIMULATION*

```

-----
-- Auteur      : Claude Guex
-- Date       : octobre 1997
-- But        : faciliter la simulation en VHDL
-- Remarques  : 1) Plusieurs procedures et fonctions utilisent le paquetage
--               standard textio pour les acces aux fichiers. Ce paquetage
--               etant refuse par plusieurs compilateurs, il sera peut-
--               etre necessaire de les modifier. Il s'agit de :
--               - toutes les procedures ECHO (3 procedures)
--               - de la procedure INDIQUER_SI_ERREUR
--               - de la fonction CMP_VECT
-----
-- Modifications :
-- Auteur      :                               Date :
-- But        :
-----
-- Ajout de procedures et fonctions :
-- Auteur      :                               Date :
-- But        :
-----

USE std.textio.ALL;

PACKAGE simulation_pkg IS

-----
-- ZONE MODIFIABLE PAR L'UTILISATEUR
-----
CONSTANT pas_de_simulation_min : time := 100 ns;
CONSTANT nom_fichier : string := "resultat.txt";
-----
-- FIN DE LA ZONE MODIFIABLE PAR L'UTILISATEUR
-----

```

```
TYPE accepter IS (oui,non);

CONSTANT unite_de_temps_min : time := 1 ns;

SIGNAL debut_cycle : bit; -- indique le debut d'un cycle d'horloge

-----
-- Procedure permettant des pas de simulation avec l'unite minimale
-----

PROCEDURE sim (nombre_d_unites_de_temps : integer := 1 ;
              unite_min : time := pas_de_simulation_min);

-----
-- Procedure permettant des pas de simulation avec l'unite minimale
-----

PROCEDURE sim (unite_min : time;
              nombre_d_unites_de_temps : integer := 1 );

-----
-- Procedure permettant plusieurs cycles d'horloge
-- Le premier appel de la procedure termine le cycle precedent si celui-ci
-- n'etait pas complet (par exemple : si on a fait quelques pas de
-- simulation non synchronises avant, reset asynchrone, combinatoire, ...)
-----

PROCEDURE cycle (nombre_de_cycles : integer := 1);

-----
-- Procedure qui indique dans un fichier si un vecteur est nul ou non
-----

PROCEDURE indiquer_si_erreur (somme : IN bit_vector;
                             nom_du_fichier: string := nom_fichier);

-----
-- Procedure permettant d'ecrire dans un fichier un texte.
-----

PROCEDURE echo (texte_a_ecrire: string;
               nom_du_fichier: string := nom_fichier);

-----
-- Procedure permettant d'ecrire dans un fichier la valeur d'un signal.
-----

PROCEDURE echo (valeur_a_ecrire: bit;
               nom_du_fichier: string := nom_fichier);

-----
-- Procedure permettant d'ecrire dans un fichier un vecteur.
-----

PROCEDURE echo_v (vecteur_a_ecrire: bit_vector;
                 nom_du_fichier: string := nom_fichier);

-----
-- Procedure permettant d'ecrire dans un fichier un texte et la valeur
-- d'un signal.
-----

PROCEDURE echo (texte_a_ecrire: string;
               valeur_a_ecrire: bit;
               nom_du_fichier: string := nom_fichier);

-----
```

```

-- Procedure permettant d'ecrire dans un fichier un texte et un vecteur.
-----
PROCEDURE echo_v (texte_a_ecrire: string;
                  vecteur_a_ecrire: bit_vector;
                  nom_du_fichier: string := nom_fichier);
-----
-- Procedure qui compare ce que l'utilisateur desire et le calcul resultant
-- de la description VHDL. Indique la valeur fausse s'il y en a.
-----
PROCEDURE check (valeur_1, valeur_2 : bit;
                 nom_du_fichier: string := nom_fichier);
-----
-- Procedure qui compare ce que l'utilisateur desire et le calcul resultant
-- de la description VHDL. Indique la valeur fausse s'il y en a.
-----
PROCEDURE check (vecteur_1, vecteur_2 : bit_vector;
                 nom_du_fichier: string := nom_fichier);
-----
-- Fonction qui verifie l'identicite de deux vecteurs.
-- Elle ne fait qu'indiquer si les deux vecteurs sont identiques (0) ou
-- non (1) a l'aide d'un bit.
-----
FUNCTION cmp_vect (vecteur_1, vecteur_2 : bit_vector;
                  ecrire_fichier: accepter := non;
                  nom_du_fichier: string := nom_fichier) RETURN bit;
-----
-- Fonction permettant de faire l'operation "ou exclusif" sur des vecteurs
-- Le "ou exclusif" se fait bit a bit.
-- PAS AU POINT. A TERMINER
-----
FUNCTION "xor" (vecteur_1, vecteur_2 : bit_vector) RETURN bit_vector;

END simulation_pkg;

PACKAGE BODY simulation_pkg IS
-----
-- Procedure permettant des pas de simulation avec l'unite minimale
-----
PROCEDURE sim (nombre_d_unites_de_temps : integer := 1 ;
              unite_min : time := pas_de_simulation_min) IS

BEGIN
  WAIT FOR (nombre_d_unites_de_temps * unite_min);
END sim;
-----
-- Procedure permettant des pas de simulation avec l'unite minimale
-----
PROCEDURE sim (unite_min : time;

```

```

        nombre_d_unites_de_temps : integer := 1 ) IS

BEGIN
    WAIT FOR (nombre_d_unites_de_temps * unite_min);
END sim;

-----
-- Procedure permettant plusieurs cycles d'horloge
-- Le premier appel de la procedure termine le cycle precedent si celui-ci
-- n'etait pas complet (par exemple : si on a fait quelques pas de
-- simulation non synchronises avant, reset asynchrone, combinatoire, ...)
-----

PROCEDURE cycle (nombre_de_cycles : integer := 1) IS
    BEGIN
        FOR i IN 1 TO nombre_de_cycles LOOP
            WAIT UNTIL debut_cycle'EVENT AND debut_cycle = '1';
        end loop;
    END cycle;

-----
-- Procedure qui indique dans un fichier si un vecteur est nul ou non
-----

PROCEDURE indiquer_si_erreur (somme : IN bit_vector;
                             nom_du_fichier: string := nom_fichier) IS
    CONSTANT texte : string := "en ordre";
    CONSTANT texte2 : string := "erreur de fonctionnement sur le pas de simulation
precedent";
    VARIABLE vecteur_int : bit_vector(somme'HIGH DOWNTO somme'LOW):=(OTHERS =>
'0');
    VARIABLE pointeur_pour_fichier : line;
    FILE fichier_resultats : text IS OUT nom_du_fichier;

BEGIN

    IF somme = vecteur_int THEN
        write (pointeur_pour_fichier,texte);
        writeline(fichier_resultats,pointeur_pour_fichier);
    ELSE
        write (pointeur_pour_fichier,texte2);
        writeline(fichier_resultats,pointeur_pour_fichier);
    END IF;
END indiquer_si_erreur;

-----
-- Procedure permettant d'ecrire dans un fichier un texte.
-----

PROCEDURE echo (texte_a_ecrire: string;
               nom_du_fichier: string := nom_fichier) IS
    VARIABLE pointeur_pour_fichier : line;
    FILE fichier_resultats : text IS OUT nom_du_fichier;

BEGIN
    write (pointeur_pour_fichier,texte_a_ecrire);
    writeline(fichier_resultats,pointeur_pour_fichier);
END echo;

-----
-- Procedure permettant d'ecrire dans un fichier la valeur d'un signal.
-----

PROCEDURE echo (valeur_a_ecrire: bit;

```

```

        nom_du_fichier: string := nom_fichier) IS
    VARIABLE pointeur_pour_fichier : line;
    FILE fichier_resultats : text IS OUT nom_du_fichier;

BEGIN
    write (pointeur_pour_fichier,valeur_a_ecrire);
    writeline(fichier_resultats,pointeur_pour_fichier);
END echo;

-----
-- Procedure permettant d'ecrire dans un fichier un vecteur
-----

PROCEDURE echo_v (vecteur_a_ecrire: bit_vector;
                  nom_du_fichier: string := nom_fichier) IS
    VARIABLE pointeur_pour_fichier : line;
    FILE fichier_resultats : text IS OUT nom_du_fichier;

BEGIN
    write (pointeur_pour_fichier,vecteur_a_ecrire);
    writeline(fichier_resultats,pointeur_pour_fichier);
END echo_v;

-----
-- Procedure permettant d'ecrire dans un fichier un texte et la valeur
-- d'un signal.
-----

PROCEDURE echo (texte_a_ecrire: string;
                valeur_a_ecrire: bit;
                nom_du_fichier: string := nom_fichier) IS
    VARIABLE pointeur_pour_fichier : line;
    FILE fichier_resultats : text IS OUT nom_du_fichier;

BEGIN
    write (pointeur_pour_fichier,texte_a_ecrire);
    write (pointeur_pour_fichier,valeur_a_ecrire);
    writeline(fichier_resultats,pointeur_pour_fichier);
END echo;

-----
-- Procedure permettant d'ecrire dans un fichier un texte et un vecteur.
-----

PROCEDURE echo_v (texte_a_ecrire: string;
                  vecteur_a_ecrire: bit_vector;
                  nom_du_fichier: string := nom_fichier) IS
    VARIABLE pointeur_pour_fichier : line;
    FILE fichier_resultats : text IS OUT nom_du_fichier;

BEGIN
    write (pointeur_pour_fichier,texte_a_ecrire);
    write (pointeur_pour_fichier,vecteur_a_ecrire);
    writeline(fichier_resultats,pointeur_pour_fichier);
END echo_v;

-----
-- Procedure qui compare ce que l'utilisateur desire et le calcul resultant
-- de la description VHDL. Indique la valeur fausse s'il y en a.
-----

PROCEDURE check (valeur_1, valeur_2 : bit;
                 nom_du_fichier: string := nom_fichier) IS
    CONSTANT enordre : string := "En ordre";
    CONSTANT erreurl : string := "Vous avez specifie ";

```

```

    CONSTANT erreur2 : string := " et votre fichier VHDL calcule ";
    VARIABLE pointeur_pour_fichier : line;
    FILE fichier_resultats : text IS OUT nom_du_fichier;

BEGIN

    IF valeur_1 = valeur_2 THEN
        write (pointeur_pour_fichier,enordre);
        writeline(fichier_resultats,pointeur_pour_fichier);
    ELSE
        write (pointeur_pour_fichier,erreur1);
        write (pointeur_pour_fichier,valeur_2);
        write (pointeur_pour_fichier,erreur2);
        write (pointeur_pour_fichier,valeur_1);
        writeline(fichier_resultats,pointeur_pour_fichier);
    END IF;
END check;

-----
-- Procedure qui compare ce que l'utilisateur desire et le calcul resultant
-- de la description VHDL. Indique la valeur fausse s'il y en a.
-----

PROCEDURE check (vecteur_1, vecteur_2 : bit_vector;
                 nom_du_fichier: string := nom_fichier) IS
    CONSTANT enordre : string := "En ordre";
    CONSTANT erreur1 : string := "Vous avez specifie ";
    CONSTANT erreur2 : string := " et votre fichier VHDL calcule ";
    VARIABLE pointeur_pour_fichier : line;
    FILE fichier_resultats : text IS OUT nom_du_fichier;

BEGIN
    ASSERT (vecteur_1'LENGTH = vecteur_2'LENGTH)
        REPORT "Attention, les deux vecteurs sont de taille differente"
        SEVERITY error;

    IF vecteur_1 = vecteur_2 THEN
        write (pointeur_pour_fichier,enordre);
        writeline(fichier_resultats,pointeur_pour_fichier);
    ELSE
        write (pointeur_pour_fichier,erreur1);
        write (pointeur_pour_fichier,vecteur_2);
        write (pointeur_pour_fichier,erreur2);
        write (pointeur_pour_fichier,vecteur_1);
        writeline(fichier_resultats,pointeur_pour_fichier);
    END IF;
END check;

-----
-- Fonction qui verifie l'identicite de deux vecteurs.
-- Elle ne fait qu'indiquer si les deux vecteurs sont identiques (0) ou
-- non (1) a l'aide d'un bit.
-----

FUNCTION cmp_vect (vecteur_1, vecteur_2 : bit_vector;
                  ecrire_fichier: accepter := non;
                  nom_du_fichier: string := nom_fichier) RETURN bit IS

    CONSTANT egaux : string := "Les deux vecteurs sont egaux";
    CONSTANT differents : string := "Les deux vecteurs sont differents";
    VARIABLE pointeur_pour_fichier : line;
    FILE fichier_resultats : text IS OUT nom_du_fichier;

BEGIN
    IF vecteur_1 = vecteur_2 THEN

```

```

    IF ecrire_fichier = oui THEN
        write(pointeur_pour_fichier, egaux);
        writeline(fichier_resultats, pointeur_pour_fichier);
    END IF;
    RETURN '0';
ELSE
    IF ecrire_fichier = oui THEN
        write(pointeur_pour_fichier, differents);
        writeline(fichier_resultats, pointeur_pour_fichier);
    END IF;
    RETURN '1';
END IF;
END cmp_vect;

-----
-- Fonction permettant de faire l'operation "ou exclusif" sur des vecteurs
-- Le "ou exclusif" se fait bit a bit.
-- Il est obligatoire de redefinir les vecteurs d'entree, En effet
-- l'utilisateur de cette fonction peut entrer deux vecteurs de meme taille
-- mais avec des bornes differentes :
--     vecteur_1(7 downto 0), vecteur_2(1 to 8)
-- Mais il peut encore entrer un vecteur directement par sa valeur, ce qui
-- implique qu'il n'aura pas de bornes (ex : "0011010").
-- Il est des lors impossible de parcourir les vecteurs d'une borne a
-- l'autre, car elles peuvent etre soit differentes, soit inexistantes !
-- Attention : on peut ecrire     vecteur_a <= vecteur_a xor vecteur_b;
--                               mais pas     vecteur_b <= vecteur_a xor vecteur_b;
-----

FUNCTION "xor" (vecteur_1, vecteur_2 : bit_vector) RETURN bit_vector IS
    ALIAS v1: bit_vector((vecteur_1'LENGTH - 1) DOWNTO 0) IS vecteur_1;
    ALIAS v2: bit_vector((vecteur_2'LENGTH - 1) DOWNTO 0) IS vecteur_2;
    VARIABLE vecteur_intermediaire : bit_vector(vecteur_1'RANGE);
    VARIABLE compteur_interne : integer := vecteur_1'LENGTH - 1;

BEGIN
    ASSERT (vecteur_1'LENGTH = vecteur_2'LENGTH)
        REPORT "Attention, les deux vecteurs sont de taille differente"
        SEVERITY error;
    FOR i IN vecteur_1'RANGE LOOP
        IF v1(compteur_interne) = v2(compteur_interne) THEN
            vecteur_intermediaire(i) := '0';
        ELSE
            vecteur_intermediaire(i) := '1';
        END IF;
        compteur_interne := compteur_interne - 1;
    END LOOP;
    RETURN vecteur_intermediaire;
END "xor";

END simulation_pkg;

```

5.3 CODE DU PAQUETAGE DE SIMULATION POUR LE STD_LOGIC_1164

```

-----
-- Auteur      : Claude Guex
-- Date        : octobre 1997
-- But         : faciliter la simulation en VHDL
-- Remarques   : 1) Plusieurs procedures et fonctions utilisent le paquetage

```

```

--          standard textio pour les acces aux fichiers. Ce paquetage
--          etant refuse par plusieurs compilateurs, il sera peut-
--          etre necessaire de les modifier. Il s'agit de :
--          - toutes les procedures ECHO (3 procedures)
--          - de la procedure INDIQUER_SI_ERREUR
--          - de la fonction CMP_VECT
-----
-- Modifications :
-- Auteur          :                               Date :
-- But              :
-----
-- Ajout de procedures et fonctions :
-- Auteur          :                               Date :
-- But              :
-----

USE std.textio.ALL;
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_textio.ALL;

PACKAGE simulation_std_logic_pkg IS

-----
-- ZONE MODIFIABLE PAR L'UTILISATEUR
-----
CONSTANT pas_de_simulation_min : time := 100 ns;
CONSTANT nom_fichier : string := "resultat.txt";
-----
-- FIN DE LA ZONE MODIFIABLE PAR L'UTILISATEUR
-----

TYPE accepter IS (oui,non);

CONSTANT unite_de_temps_min : time := 1 ns;

SIGNAL debut_cycle : std_logic; -- indique le debut d'un cycle d'horloge

-----
-- Procedure permettant des pas de simulation avec l'unite minimale
-----

PROCEDURE sim (nombre_d_unites_de_temps : Integer := 1 ;
                unite_min : time := pas_de_simulation_min);

-----
-- Procedure permettant des pas de simulation avec l'unite minimale
-----

PROCEDURE sim (unite_min : time;
                nombre_d_unites_de_temps : Integer := 1 );

-----
-- Procedure permettant plusieurs cycles d'horloge
-- Le premier appel de la procedure termine le cycle precedent si celui-ci
-- n'etait pas complet (par exemple : si on a fait quelques pas de
-- simulation non synchronises avant, reset asynchrone, combinatoire, ...)
-----

PROCEDURE cycle (nombre_de_cycles : Integer := 1);

-----
-- Procedure qui indique dans un fichier si un vecteur est nul ou non
-----

```



```
PROCEDURE indiquer_si_erreur (somme : IN std_logic_vector;
                             nom_du_fichier: string := nom_fichier);

-----
-- Procedure permettant d'ecrire dans un fichier un texte.
-----

PROCEDURE echo (texte_a_ecrire: string;
               nom_du_fichier: string := nom_fichier);

-----
-- Procedure permettant d'ecrire dans un fichier la valeur d'un signal.
-----

PROCEDURE echo (valeur_a_ecrire: std_logic;
               nom_du_fichier: string := nom_fichier);

-----
-- Procedure permettant d'ecrire dans un fichier un vecteur.
-----

PROCEDURE echo_v (vecteur_a_ecrire: std_logic_vector;
                 nom_du_fichier: string := nom_fichier);

-----
-- Procedure permettant d'ecrire dans un fichier un texte et la valeur
-- d'un signal.
-----

PROCEDURE echo (texte_a_ecrire: string;
               valeur_a_ecrire: std_logic;
               nom_du_fichier: string := nom_fichier);

-----
-- Procedure permettant d'ecrire dans un fichier un texte et un vecteur.
-----

PROCEDURE echo_v (texte_a_ecrire: string;
                 vecteur_a_ecrire: std_logic_vector;
                 nom_du_fichier: string := nom_fichier);

-----
-- Procedure qui compare ce que l'utilisateur desire et le calcul resultant
-- de la description VHDL. Indique la valeur fausse s'il y en a.
-----

PROCEDURE check (valeur_1, valeur_2 : std_logic;
                nom_du_fichier: string := nom_fichier);

-----
-- Procedure qui compare ce que l'utilisateur desire et le calcul resultant
-- de la description VHDL. Indique la valeur fausse s'il y en a.
-----

PROCEDURE check (vecteur_1, vecteur_2 : std_logic_vector;
                nom_du_fichier: string := nom_fichier);

-----
-- Fonction qui verifie l'identicite de deux vecteurs.
-- Elle ne fait qu'indiquer si les deux vecteurs sont identiques (0) ou
-- non (1) a l'aide d'un std_logic.
-----

FUNCTION cmp_vect (vecteur_1, vecteur_2 : std_logic_vector;
                  ecrire_fichier: accepter := non;
                  nom_du_fichier: string := nom_fichier) return std_logic;
```

```
END simulation_std_logic_pkg;
```

```
PACKAGE BODY simulation_std_logic_pkg IS
```

```
-----  
-- Procedure permettant des pas de simulation avec l' unite minimale  
-----
```

```
PROCEDURE sim (nombre_d_unites_de_temps : Integer := 1 ;  
               unite_min : time := pas_de_simulation_min) IS
```

```
BEGIN  
  WAIT FOR (nombre_d_unites_de_temps * unite_min);  
END sim;
```

```
-----  
-- Procedure permettant des pas de simulation avec l' unite minimale  
-----
```

```
PROCEDURE sim (unite_min : time;  
              nombre_d_unites_de_temps : Integer := 1 ) IS
```

```
BEGIN  
  WAIT FOR (nombre_d_unites_de_temps * unite_min);  
END sim;
```

```
-----  
-- Procedure permettant plusieurs cycles d' horloge  
-- Le premier appel de la procedure termine le cycle precedent si celui-ci  
-- n'etait pas complet (par exemple : si on a fait quelques pas de  
-- simulation non synchronises avant, reset asynchrone, combinatoire, ...)  
-----
```

```
PROCEDURE cycle (nombre_de_cycles : Integer := 1) IS  
  BEGIN  
    FOR i IN 1 TO nombre_de_cycles LOOP  
      WAIT UNTIL debut_cycle'event AND debut_cycle = '1';  
    END LOOP;  
  END cycle;
```

```
-----  
-- Procedure qui indique dans un fichier si un vecteur est nul ou non  
-----
```

```
PROCEDURE indiquer_si_erreur (somme : IN std_logic_vector;  
                             nom_du_fichier: string := nom_fichier) IS  
  CONSTANT texte : string := "en ordre";  
  CONSTANT texte2 : string := "erreur de fonctionnement sur le pas de simulation  
precedent";  
  VARIABLE vecteur_int : std_logic_vector(somme'HIGH DOWNTO somme'LOW):=(OTHERS  
=> '0');  
  VARIABLE pointeur_pour_fichier : line;  
  FILE fichier_resultats : text IS OUT nom_du_fichier;  
  
BEGIN  
  
  IF somme = vecteur_int THEN  
    write (pointeur_pour_fichier, texte);  
    writeline(fichier_resultats, pointeur_pour_fichier);  
  ELSE
```

```

        write (pointeur_pour_fichier, texte2);
        writeline(fichier_resultats, pointeur_pour_fichier);
    END IF;
END indiquer_si_erreur;

-----
-- Procedure permettant d'ecrire dans un fichier un texte.
-----

PROCEDURE echo (texte_a_ecrire: string;
                nom_du_fichier: string := nom_fichier) IS
    VARIABLE pointeur_pour_fichier : line;
    FILE fichier_resultats : text IS OUT nom_du_fichier;

BEGIN
    write (pointeur_pour_fichier, texte_a_ecrire);
    writeline(fichier_resultats, pointeur_pour_fichier);
END echo;

-----
-- Procedure permettant d'ecrire dans un fichier la valeur d'un signal.
-----

PROCEDURE echo (valeur_a_ecrire: std_logic;
                nom_du_fichier: string := nom_fichier) IS
    VARIABLE pointeur_pour_fichier : line;
    FILE fichier_resultats : text IS OUT nom_du_fichier;

BEGIN
    write (pointeur_pour_fichier, valeur_a_ecrire);
    writeline(fichier_resultats, pointeur_pour_fichier);
END echo;

-----
-- Procedure permettant d'ecrire dans un fichier un vecteur
-----

PROCEDURE echo_v (vecteur_a_ecrire: std_logic_vector;
                  nom_du_fichier: string := nom_fichier) IS
    VARIABLE pointeur_pour_fichier : line;
    FILE fichier_resultats : text IS OUT nom_du_fichier;

BEGIN
    write (pointeur_pour_fichier, vecteur_a_ecrire);
    writeline(fichier_resultats, pointeur_pour_fichier);
END echo_v;

-----
-- Procedure permettant d'ecrire dans un fichier un texte et la valeur
-- d'un signal.
-----

PROCEDURE echo (texte_a_ecrire: string;
                valeur_a_ecrire: std_logic;
                nom_du_fichier: string := nom_fichier) IS
    VARIABLE pointeur_pour_fichier : line;
    FILE fichier_resultats : text IS OUT nom_du_fichier;

BEGIN
    write (pointeur_pour_fichier, texte_a_ecrire);
    write (pointeur_pour_fichier, valeur_a_ecrire);
    writeline(fichier_resultats, pointeur_pour_fichier);
END echo;

-----

```

```

-- Procedure permettant d'ecrire dans un fichier un texte et un vecteur.
-----

PROCEDURE echo_v (texte_a_ecrire: string;
                  vecteur_a_ecrire: std_logic_vector;
                  nom_du_fichier: string := nom_fichier) IS
  VARIABLE pointeur_pour_fichier : line;
  FILE fichier_resultats : text IS OUT nom_du_fichier;

BEGIN
  write (pointeur_pour_fichier, texte_a_ecrire);
  write (pointeur_pour_fichier, vecteur_a_ecrire);
  writeline(fichier_resultats, pointeur_pour_fichier);
END echo_v;

-----

-- Procedure qui compare ce que l'utilisateur desire et le calcul resultant
-- de la description VHDL. Indique la valeur fausse s'il y en a.
-----

PROCEDURE check (valeur_1, valeur_2 : std_logic;
                 nom_du_fichier: string := nom_fichier) IS
  CONSTANT enordre : string := "En ordre";
  CONSTANT erreur1 : string := "Vous avez specifie ";
  CONSTANT erreur2 : string := " et votre fichier VHDL calcule ";
  VARIABLE pointeur_pour_fichier : line;
  FILE fichier_resultats : text IS OUT nom_du_fichier;

BEGIN

  IF valeur_1 = valeur_2 THEN
    write (pointeur_pour_fichier, enordre);
    writeline(fichier_resultats, pointeur_pour_fichier);
  ELSE
    write (pointeur_pour_fichier, erreur1);
    write (pointeur_pour_fichier, valeur_2);
    write (pointeur_pour_fichier, erreur2);
    write (pointeur_pour_fichier, valeur_1);
    writeline(fichier_resultats, pointeur_pour_fichier);
  END IF;
END check;

-----

-- Procedure qui compare ce que l'utilisateur desire et le calcul resultant
-- de la description VHDL. Indique la valeur fausse s'il y en a.
-----

PROCEDURE check (vecteur_1, vecteur_2 : std_logic_vector;
                 nom_du_fichier: string := nom_fichier) IS
  CONSTANT enordre : string := "En ordre";
  CONSTANT erreur1 : string := "Vous avez specifie ";
  CONSTANT erreur2 : string := " et votre fichier VHDL calcule ";
  VARIABLE pointeur_pour_fichier : line;
  FILE fichier_resultats : text IS OUT nom_du_fichier;

BEGIN
  ASSERT (vecteur_1'LENGTH = vecteur_2'LENGTH)
    REPORT "Attention, les deux vecteurs sont de taille differente"
    SEVERITY error;

  IF vecteur_1 = vecteur_2 THEN
    write (pointeur_pour_fichier, enordre);
    writeline(fichier_resultats, pointeur_pour_fichier);
  ELSE
    write (pointeur_pour_fichier, erreur1);

```

```

        write (pointeur_pour_fichier,vecteur_2);
        write (pointeur_pour_fichier,erreur2);
        write (pointeur_pour_fichier,vecteur_1);
        writeline(fichier_resultats,pointeur_pour_fichier);
    END IF;
END check;

-----
-- Fonction qui verifie l'identicite de deux vecteurs.
-- Elle ne fait qu'indiquer si les deux vecteurs sont identiques (0) ou
-- non (1) a l'aide d'un std_logic.
-----

FUNCTION cmp_vect (vecteur_1, vecteur_2 : std_logic_vector;
                  ecrire_fichier: accepter := non;
                  nom_du_fichier: string := nom_fichier) RETURN std_logic IS

    CONSTANT egaux : string := "Les deux vecteurs sont egaux";
    CONSTANT differents : string := "Les deux vecteurs sont differents";
    VARIABLE pointeur_pour_fichier : line;
    FILE fichier_resultats : text IS OUT nom_du_fichier;

BEGIN
    IF vecteur_1 = vecteur_2 THEN
        IF ecrire_fichier = oui THEN
            write(pointeur_pour_fichier,egaux);
            writeline(fichier_resultats,pointeur_pour_fichier);
        END IF;
        RETURN '0';
    ELSE
        IF ecrire_fichier = oui THEN
            write(pointeur_pour_fichier,differents);
            writeline(fichier_resultats,pointeur_pour_fichier);
        END IF;
        RETURN '1';
    END IF;
END cmp_vect;

END simulation_std_logic_pkg;

```

5.4 FICHIER DE SIMULATION DU DÉCODEUR SEPT SEGMENTS

```

-- Auteur : Claude Guex
-- Date   : octobre 1997
-- But    : Test un comparateur

LIBRARY eiev;
USE eiev.simulation_pkg.ALL;

ENTITY sim_sept_segments IS
END sim_sept_segments;

ARCHITECTURE simulation OF sim_sept_segments IS
    COMPONENT sept_seg_a PORT (code_binaire : IN bit_vector (3 DOWNTO 0);
                               S7, S6, S5, S4, S3, S2, S1 : OUT bit);
    END COMPONENT;

```

```

FOR ALL : sept_seg_a USE ENTITY WORK.sept_seg(tdv_sept_seg);

SIGNAL sim_end : boolean := false;
SIGNAL code_binaire : bit_vector (3 DOWNTO 0);
SIGNAL sept_segments : bit_vector (6 DOWNTO 0);
SIGNAL S7, S6, S5, S4, S3, S2, S1 : bit;
SIGNAL erreur : bit_vector (6 DOWNTO 0);

-----
-- PROGRAMME DE SIMULATION
-----

BEGIN
al: sept_seg_a PORT MAP (code_binaire, S7, S6, S5, S4, S3, S2, S1 );

PROCESS (S7, S6, S5, S4, S3, S2, S1)
BEGIN
    sept_segments(6) <= S7;
    sept_segments(5) <= S6;
    sept_segments(4) <= S5;
    sept_segments(3) <= S4;
    sept_segments(2) <= S3;
    sept_segments(1) <= S2;
    sept_segments(0) <= S1;
END PROCESS;

PROCESS
BEGIN
    echo("---- debut de la simulation");

    echo("---- affiche un 0");
    code_binaire <= "0000";
    sim;
    erreur <= sept_segments XOR "0111111";
    indiquer_si_erreur(erreur);

    echo("---- affiche un 1");
    code_binaire <= "0001";
    sim;
    erreur <= sept_segments XOR "0000110";
    indiquer_si_erreur(erreur);

    echo("---- affiche un 2");
    code_binaire <= "0010";
    sim;
    erreur <= sept_segments XOR "1011011";
    indiquer_si_erreur(erreur);

    echo("---- affiche un 3");
    code_binaire <= "0011";
    sim;
    erreur <= sept_segments XOR "1001111";
    indiquer_si_erreur(erreur);

    echo("---- affiche un 4");
    code_binaire <= "0100";
    sim;
    erreur <= sept_segments XOR "1100110";
    indiquer_si_erreur(erreur);

    echo("---- affiche un 5");
    code_binaire <= "0101";
    sim;

```

```
erreur <= sept_segments XOR "1101101";
indiquer_si_erreur(erreur);

echo("---- affiche un 6");
code_binaire <= "0110";
sim;
erreur <= sept_segments XOR "1111101";
indiquer_si_erreur(erreur);

echo("---- affiche un 7");
code_binaire <= "0111";
sim;
erreur <= sept_segments XOR "0000111";
indiquer_si_erreur(erreur);

echo("---- affiche un 8");
code_binaire <= "1000";
sim;
erreur <= sept_segments XOR "1111111";
indiquer_si_erreur(erreur);

echo("---- affiche un 9");
code_binaire <= "1001";
sim;
erreur <= sept_segments XOR "1101111";
indiquer_si_erreur(erreur);

echo("---- affiche un X");
code_binaire <= "1010";
sim;
erreur <= sept_segments XOR "1110110";
indiquer_si_erreur(erreur);

echo("---- affiche un X");
code_binaire <= "1011";
sim;
erreur <= sept_segments XOR "1110110";
indiquer_si_erreur(erreur);

echo("---- affiche un X");
code_binaire <= "1100";
sim;
erreur <= sept_segments XOR "1110110";
indiquer_si_erreur(erreur);

echo("---- affiche un X");
code_binaire <= "1101";
sim;
erreur <= sept_segments XOR "1110110";
indiquer_si_erreur(erreur);

echo("---- affiche un X");
code_binaire <= "1110";
sim;
erreur <= sept_segments XOR "1110110";
indiquer_si_erreur(erreur);

echo("---- affiche un X");
code_binaire <= "1111";
sim;
erreur <= sept_segments XOR "1110110";
indiquer_si_erreur(erreur);

echo("---- fin de la simulation");
```

```

    sim_end <= true;

    WAIT;

    END PROCESS;
END simulation;

```

5.5 FICHER DE SIMULATION DU COMPAREUR

```

-- Auteur : Claude Guex
-- Date   : octobre 1997
-- But    : Test un compareur

LIBRARY eiev;
USE eiev.simulation_pkg.ALL;

ENTITY sim_comparateur IS
END sim_comparateur;

ARCHITECTURE simulation OF sim_comparateur IS
    COMPONENT compareur1 PORT (vecteur1,vecteur2 : IN bit_vector (3 DOWNTO 0);
                               pg, egal, pp : IN bit;
                               vlpgv2, vlegalv2, vlppv2 : OUT bit);
    END COMPONENT;

    FOR ALL : compareur1 USE ENTITY WORK.compareur(comportementale);

    SIGNAL sim_end : boolean := false;
    SIGNAL vecteur_a, vecteur_b : bit_vector (3 DOWNTO 0);
    SIGNAL pg, egal, pp : bit;
    SIGNAL comparaison_precedente : bit_vector (2 DOWNTO 0);
    SIGNAL vapgvb, vaegalvb, vappvb : bit;
    SIGNAL resultat_comparaison : bit_vector (2 DOWNTO 0);
    SIGNAL erreur : bit_vector (2 DOWNTO 0);

    -----
    -- PROGRAMME PRINCIPAL
    -----

    BEGIN
    al: compareur1 PORT MAP (vecteur_a, vecteur_b, pg, egal, pp, vapgvb, vaegalvb,
    vappvb );

        PROCESS (comparaison_precedente)
        BEGIN
            pg <= comparaison_precedente(2);
            egal <= comparaison_precedente(1);
            pp <= comparaison_precedente(0);
        END PROCESS;

        PROCESS (vapgvb, vaegalvb, vappvb)
        BEGIN
            resultat_comparaison(2) <= vapgvb;

```



```
    resultat_comparaison(1) <= vaegalvb;
    resultat_comparaison(0) <= vappvb;
END PROCESS;

PROCESS
BEGIN

    echo("---- debut de la simulation");
    comparaison_precedente <= "000";

    echo("---- vecteur_a = vecteur_b = 0, pas de report");
    vecteur_a <= "0000";
    vecteur_b <= "0000";
sim;
    erreur <= resultat_comparaison XOR "010"; --pour verifier graphiquement
    check (resultat_comparaison, "010");      --verification par fichier

    echo("---- vecteur_a = vecteur_b = 0, egal = 1");
    comparaison_precedente <= "010";
    vecteur_a <= "0000";
    vecteur_b <= "0000";
sim;
    erreur <= resultat_comparaison XOR "010";
    check (resultat_comparaison, "010");

    echo("---- vecteur_a = vecteur_b = 0, pp = 1");
    comparaison_precedente <= "001";
    vecteur_a <= "0000";
    vecteur_b <= "0000";
sim;
    erreur <= resultat_comparaison XOR "001";
    check (resultat_comparaison, "001");

    echo("---- vecteur_a = vecteur_b = 0, pg = 1");
    comparaison_precedente <= "100";
    vecteur_a <= "0000";
    vecteur_b <= "0000";
sim;
    erreur <= resultat_comparaison XOR "100";
    check (resultat_comparaison, "100");

    echo("---- vecteur_a = vecteur_b = 1, egal = 1");
    comparaison_precedente <= "010";
    vecteur_a <= "0001";
    vecteur_b <= "0001";
sim;
    erreur <= resultat_comparaison XOR "010";
    check (resultat_comparaison, "010");

    echo("---- vecteur_a = vecteur_b = 2, egal = 1");
    vecteur_a <= "0010";
    vecteur_b <= "0010";
sim;
    erreur <= resultat_comparaison XOR "010";
    check (resultat_comparaison, "010");

    echo("---- vecteur_a = vecteur_b = 3, egal = 1");
    vecteur_a <= "0011";
    vecteur_b <= "0011";
sim;
    erreur <= resultat_comparaison XOR "010";
    check (resultat_comparaison, "010");

    echo("---- vecteur_a = vecteur_b = 4, egal = 1");
    vecteur_a <= "0100";
```

```
    vecteur_b <= "0100";
sim;
    erreur <= resultat_comparaison XOR "010";
    check (resultat_comparaison, "010");

    echo("---- vecteur_a = vecteur_b = 5, egal = 1");
    vecteur_a <= "0101";
    vecteur_b <= "0101";
sim;
    erreur <= resultat_comparaison XOR "010";
    check (resultat_comparaison, "010");

    echo("---- vecteur_a = vecteur_b = 6, egal = 1");
    vecteur_a <= "0110";
    vecteur_b <= "0110";
sim;
    erreur <= resultat_comparaison XOR "010";
    check (resultat_comparaison, "010");

    echo("---- vecteur_a = vecteur_b = 7, egal = 1");
    vecteur_a <= "0111";
    vecteur_b <= "0111";
sim;
    erreur <= resultat_comparaison XOR "010";
    check (resultat_comparaison, "010");

    echo("---- vecteur_a = vecteur_b = 8, egal = 1");
    vecteur_a <= "1000";
    vecteur_b <= "1000";
sim;
    erreur <= resultat_comparaison XOR "010";
    check (resultat_comparaison, "010");

    echo("---- vecteur_a = vecteur_b = 9, egal = 1");
    vecteur_a <= "1001";
    vecteur_b <= "1001";
sim;
    erreur <= resultat_comparaison XOR "010";
    check (resultat_comparaison, "010");

    echo("---- vecteur_a = vecteur_b = a, egal = 1");
    vecteur_a <= "1010";
    vecteur_b <= "1010";
sim;
    erreur <= resultat_comparaison XOR "010";
    check (resultat_comparaison, "010");

    echo("---- vecteur_a = vecteur_b = b, egal = 1");
    vecteur_a <= "1011";
    vecteur_b <= "1011";
sim;
    erreur <= resultat_comparaison XOR "010";
    check (resultat_comparaison, "010");

    echo("---- vecteur_a = vecteur_b = c, egal = 1");
    vecteur_a <= "1100";
    vecteur_b <= "1100";
sim;
    erreur <= resultat_comparaison XOR "010";
    check (resultat_comparaison, "010");

    echo("---- vecteur_a = vecteur_b = d, egal = 1");
    vecteur_a <= "1101";
    vecteur_b <= "1101";
sim;
```

```
    erreur <= resultat_comparaison XOR "010";
    check (resultat_comparaison, "010");

    echo("---- vecteur_a = vecteur_b = e, egal = 1");
    vecteur_a <= "1110";
    vecteur_b <= "1110";
sim;
    erreur <= resultat_comparaison XOR "010";
    check (resultat_comparaison, "010");

    echo("---- vecteur_a = vecteur_b = f, egal = 1");
    vecteur_a <= "1111";
    vecteur_b <= "1111";
sim;
    erreur <= resultat_comparaison XOR "010";
    check (resultat_comparaison, "010");

    echo("---- vecteur_a < vecteur_b");
    vecteur_a <= "0000";
    vecteur_b <= "0001";
sim;
    erreur <= resultat_comparaison XOR "001";
    check (resultat_comparaison, "001");

    echo("---- vecteur_a < vecteur_b");
    vecteur_a <= "0000";
    vecteur_b <= "0110";
sim;
    erreur <= resultat_comparaison XOR "001";
    check (resultat_comparaison, "001");

    echo("---- vecteur_a < vecteur_b");
    vecteur_a <= "0000";
    vecteur_b <= "1011";
sim;
    erreur <= resultat_comparaison XOR "001";
    check (resultat_comparaison, "001");

    echo("---- vecteur_a > vecteur_b");
    vecteur_a <= "0100";
    vecteur_b <= "0000";
sim;
    erreur <= resultat_comparaison XOR "100";
    check (resultat_comparaison, "100");

    echo("---- vecteur_a > vecteur_b");
    vecteur_a <= "0111";
    vecteur_b <= "0001";
sim;
    erreur <= resultat_comparaison XOR "100";
    check (resultat_comparaison, "100");

    echo("---- vecteur_a > vecteur_b");
    vecteur_a <= "1110";
    vecteur_b <= "0110";
sim;
    erreur <= resultat_comparaison XOR "100";
    check (resultat_comparaison, "100");

    echo("---- test non complet pour les > et <, mais suffisant");
    echo("---- fin de la simulation");

    echo("---- il reste toute une serie de tests a terminer ",
        string("a_faire.txt"));
```

```

    sim_end <= true;

    WAIT;

    END PROCESS;
END simulation;

```

5.6 FICHER DE SIMULATION DU COMPTEUR 4 BITS

```

-----
-- Nom du fichier : comptsim.vhd
-- Auteur          : C. Guex
-- Date           : mai 1997
-- Version        : 1.0
-- Modifications  :
-- Auteur         :                               Date :
-----
-- But            : exemple du cours VHDL : simulation d'un compteur
-----

LIBRARY eiev;
USE eiev.simulation_pkg.ALL;

ENTITY sim_compteur IS
END sim_compteur;

-----
-- Architecture pour la simulation du compteur
-----

ARCHITECTURE simul_compteur OF sim_compteur IS
    COMPONENT compteur PORT (horloge, report_in : IN bit;
                             valeur_compteur : OUT bit_vector (3 DOWNTO 0);
                             report_out : OUT bit);
    END COMPONENT;

    FOR ALL : compteur USE ENTITY WORK.compt_4_bits (structurelle);

    CONSTANT periode : time := 100 ns;

    SIGNAL sim_end : boolean := false;

    SIGNAL valeur_compteur : bit_vector (3 DOWNTO 0);
    SIGNAL enable : bit;
    SIGNAL report_sortie : bit;
    SIGNAL horloge : bit; -- signal d'horloge
    SIGNAL erreur : bit_vector (3 downto 0);

BEGIN

-----
-- ce processus genere une horloge pour la simulation
-----

```

```

PROCESS
BEGIN
  IF NOT sim_end THEN
    horloge <= '0',
              '1' AFTER periode / 4,
              '1' AFTER 2 * periode / 4,
              '0' AFTER 3 * periode / 4;
    debut_cycle <= '1',
                  '0' AFTER unite_de_temps_min;
    WAIT FOR periode;
  ELSE WAIT;
  END IF;
END PROCESS;

```

```
a1: compteur PORT MAP (horloge, enable, valeur_compteur, report_sortie);
```

```

PROCESS
BEGIN
-- Attention, le premier cycle doit etre double pour un fonctionnement correct
-- ou alors, le plus simple est de toujours commencer par un cycle inactif

  cycle; -- cycle inactif

  enable <= '1';
  cycle;
  erreur <= valeur_compteur XOR "0001";
  cycle;
  erreur <= valeur_compteur XOR "0010";
  cycle;
  erreur <= valeur_compteur XOR "0011";
  cycle;
  erreur <= valeur_compteur XOR "0100";
  cycle;
  erreur <= valeur_compteur XOR "0101";
  cycle;
  erreur <= valeur_compteur XOR "0110";
  cycle;
  erreur <= valeur_compteur XOR "0111";
  cycle;
  erreur <= valeur_compteur XOR "1000";
  enable <= '0';
  cycle;
  erreur <= valeur_compteur XOR "1000";
  cycle;
  erreur <= valeur_compteur XOR "1000";
  enable <= '1';
  cycle;
  erreur <= valeur_compteur XOR "1001";
  cycle;
  erreur <= valeur_compteur XOR "1010";
  cycle;
  erreur <= valeur_compteur XOR "1011";
  cycle;
  erreur <= valeur_compteur XOR "1100";
  cycle;
  erreur <= valeur_compteur XOR "1101";
  cycle;
  erreur <= valeur_compteur XOR "1110";
  cycle;
  erreur <= valeur_compteur XOR "1111";
  cycle;
  erreur <= valeur_compteur XOR "0000";
  cycle;

```

```
        erreur <= valeur_compteur XOR "0001";

        sim_end <= true;

    END PROCESS;

END simul_compteur;
```

5.7 FICHER DE SIMULATION DU DÉTECTEUR DE SENS

```
-----
-- Nom du fichier   : m_etats.vhd
-- Auteur          : C. Guex
-- Date            : avril 1997
-- Version         : 1.0
-- Modifications   :
-- Auteur          :                               Date :
-----
-- But              : exemple du cours VHDL d'une simulation d'une machine
--                  : d'etats avec reset asynchrone et set synchrone
-----

LIBRARY eiev;
USE eiev.simulation_pkg.ALL;

ENTITY sim_detecteur_sens IS
END sim_detecteur_sens;

-----
-- Architecture pour la simulation du detecteur de sens
-----

ARCHITECTURE simul_det_sens OF sim_detecteur_sens IS
    COMPONENT detect_sens PORT (entree_a, entree_b, clk, reset, set : IN bit;
                               sens : OUT bit);
    END COMPONENT;

    FOR ALL : detect_sens USE ENTITY WORK.detecteur_sens (det_sens_etats);

    CONSTANT periode : time := 100 ns;

    SIGNAL sim_end : boolean := false;
    SIGNAL entree_a, entree_b : bit;
    SIGNAL set, reset : bit;
    SIGNAL horloge : bit;           -- signal d'horloge
    SIGNAL verif_sortie : bit;
    SIGNAL erreur : bit;

BEGIN
```

```
-----
-- ce processus genere une horloge pour la simulation
-----
```

```
PROCESS
BEGIN
  IF NOT sim_end THEN
    horloge <= '0',
              '1' AFTER periode / 4,
              '1' AFTER 2 * periode / 4,
              '0' AFTER 3 * periode / 4;
    debut_cycle <= '1',
                  '0' AFTER unite_de_temps_min;
    WAIT FOR periode;
  ELSE WAIT;
  END IF;
END PROCESS;
```

```
a1: detect_sens PORT MAP (entree_a, entree_b, horloge, reset, set,
verif_sortie);
```

```
PROCESS
BEGIN
```

```
-- Attention, le premier cycle doit etre double pour un fonctionnement correct
-- ou alors, le plus simple est de toujours commencer par un cycle inactif
```

```
cycle; -- cycle inactif

  reset <= '1';
sim(10);
  reset <= '0';
cycle;
  set <= '0';
  entree_a <= '0';
  entree_b <= '0';
cycle;
  entree_a <= '0';
  entree_b <= '1';
cycle(4);
  erreur <= verif_sortie XOR '1';
  entree_a <= '1';
  entree_b <= '1';
cycle;
  erreur <= verif_sortie XOR '1';
  entree_a <= '1';
  entree_b <= '0';
cycle;
  erreur <= verif_sortie XOR '1';
  entree_a <= '0';
  entree_b <= '0';
cycle;
  erreur <= verif_sortie XOR '1';
  entree_a <= '1';
  entree_b <= '0';
cycle;
  erreur <= verif_sortie XOR '0';
  entree_a <= '1';
  entree_b <= '1';
cycle;
  erreur <= verif_sortie XOR '0';
  entree_a <= '0';
  entree_b <= '1';
```

```
    cycle;
    erreur <= verif_sortie XOR '0';
    entree_a <= '0';
    entree_b <= '0';
    cycle;
    erreur <= verif_sortie XOR '0';

    sim_end <= true;
END PROCESS;

END simul_det_sens;
```


6 ANNEXES

6.1 LES OPÉRATEURS VHDL

Tous les opérateurs existants en VHDL se trouvent dans la liste ci-dessous dans l'ordre décroissant de leur priorité :

code VHDL	nom	classe	priorité	exemple
abs	valeur absolue	op. divers	1	abs A
not	non logique	op. divers	1	not A
**	puissance	op. divers	1	A**B
*	multiplication	op. de multiplication	2	A*B
/	division	op. de multiplication	2	A/B
mod	modulo	op. de multiplication	2	A mod B
rem	remainder	op. de multiplication	2	A rem B
+	plus	op. de signe	3	+A
-	moins	op. de signe	3	-A
+	addition	op. d'addition	4	A + B
-	soustraction	op. d'addition	4	A - B
&	concaténation	op. d'addition	4	- D1&D0 - "vh"&"dl"
=	équivalent à	op. relationnels	5	when A=B
/=	différent de	op. relationnels	5	when A/=B
<	plus petit	op. relationnels	5	when A<B
<=	plus petit ou égal	op. relationnels	5	when A<=B
>	plus grand	op. relationnels	5	when A>B

code VHDL	nom	classe	priorité	exemple
<code>>=</code>	plus grand ou égal	op. relationnels	5	when $A \geq B$
and	et	op. logiques	6	A and B
or	ou	op. logiques	6	A or B
nand	et inversé	op. logiques	6	A nand B
nor	ou inversé	op. logiques	6	A nor B
xor	ou exclusif	op. logiques	6	A xor B

6.2 LES MOTS RÉSERVÉS

abs	disconnect	label	package	units
access	downto	library	port	until
after		linkage	procedure	use
alias	else	loop	process	
all	elsif			variable
and	end	map	range	
architecture	entity	mod	record	wait
array	exit		register	when
assert		nand	rem	while
attribute	file	new	report	with
	for	next	return	
begin	function	nor		xor
block		not	select	
body	generate	null	severity	
buffer	generic		signal	
bus	guarded	of	subtype	
		on		
case	if	open	then	
component	in	or	to	
configuration	inout	others	transport	
constant	is	out	type	

6.3 PASSAGE DE ABEL EN VHDL

Abel

Entrées-sorties

```

module route

Declarations

entree PIN;
A1, A0 PIN;
A = [A1, A0];

sortie PIN istype 'reg' ;
S1, S0 PIN istype 'reg';
S = [S1, S0];

entree_sortie PIN istype 'reg';

```

Fonctionnement

```

Declarations
...
Equations
...

```

Déclaration de l'horloge

```

bascule.clk = horloge;

bascule := entree_bascule;

```

Assignment

```
:=, =
```

VHDL

Entrées-sorties

```

entity route is

port (

entree: in STD_LOGIC;
A: in STD_LOGIC_VECTOR(1 downto 0);

sortie: out STD_LOGIC;
S: out STD_LOGIC_VECTOR(1 downto 0);

entree_sortie: inout STD_LOGIC);

end;

```

Fonctionnement

```

architecture behavioral of route is
...
end behavioral;

```

Déclaration de l'horloge

```

process (horloge)
  if horloge'event and horloge = '1'
    then bascule <= entree_bascule;
  end if;
end process;

```

Assignment

```
<=
```

*Abel**VHDL*Logique

&, #, \$, !

Machine à états

State_diagram machine

State etat_1:

```
    sortie = 1;  
    GoTo etat_futur;
```

State etat_2:

```
    sortie = 1;  
    IF condition THEN  
        etat_futur  
    ELSE IF (!condition) THEN  
        etat_1;
```

Logique

and, or, xor, not

Machine à états

case machine **is**

when etat_1 =>

```
        sortie <= 1;  
        machine <= etat_futur;
```

when etat_2 =>

```
        sortie <= 1;  
        if condition then  
            machine <= etat_futur;  
        elsif not condition then  
            machine <= etat_1;  
        else  
            machine <= etat_initial;  
        end if;
```

when others =>

```
        machine <= etat_trappe;  
end case;
```

6.4 *LES PAQUETAGES STANDARDS*

6.4.1 *Standard*

La norme du paquetage Standard peut être commandée à l'adresse suivante :

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street
New York
NY 10017-2394
USA

Référence :

IEEE Standard VHDL Language Reference Manual
ANSI/IEEE Std 1076-1993 (Revision of IEEE Std 1076-1987)
ISBN 1-55937-376-8
version 1994 (une nouvelle édition est publiée tous les 5 ans environ)

6.4.2 *Textio*

La norme de Textio peut être commandée à l'adresse suivante :

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street
New York
NY 10017-2394
USA

Référence :

IEEE Standard VHDL Language Reference Manual
ANSI/IEEE Std 1076-1993 (Revision of IEEE Std 1076-1987)
ISBN 1-55937-376-8

version 1994 (une nouvelle édition est publiée tous les 5 ans environ)

6.4.3 *Std_logic_1164*

La norme du paquetage Std_logic_1164 peut être commandée à l'adresse suivante :

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street
New York
NY 10017-2394
USA

Référence :

IEEE Standard Multivalued Logic System for VHDL Model Interoperability
(Std_logic_1164)
ANSI/IEEE Std 1164-1993
ISBN 1-55937-299-0
version 1993 (une nouvelle édition est publiée tous les 5 ans environ)

7 BIBLIOGRAPHIE

7.1 LES INDISPENSABLES

- [1] VHDL analysis and Modeling of Digital Systems, 1993
Zainalabedin Navabi
Ed. Mc Graw Hill
ISBN : 0-07-112732-1
- [2] VHDL for programmable logic
Kevin Skahill
Ed. Cypress Semiconductor
ISBN : 0-201-89573-0

7.2 LES AUTRES

- [3] A VHDL Primer, 1994
J. Bhasker
Ed. PTR PH
ISBN : 0-13-181447-8
- [4] VHDL du langage à la modélisation, 1990
R. Airiau, J.-M. Bergé, V. Olive, J. Rouillard
Presses Polytechniques et Universitaires Romandes
ISBN : 2-88074-191-2
- [5] Circuit Synthesis with VHDL, 1994
Roland Airiau, Jean-Michel Bergé, Vincent Olive
Kluwer Academic Publishers
ISBN : 0-7923-9429-1

8 INDEX

abréviations	
DOD	2
VHDL	2
VHSIC	2
VISC	15
code VHDL	
after	7
inertial	8
transport	9
when	27
combinatoire	
description comportementale	31
description par flot de données	25
description structurelle	29
Commandes de simulation	
cmp_vect	70, 71
cycle	67
echo	67
indiquer_si_erreur	68, 69
sim	66
description	
comportementale	31, 36
flot de données	25, 41
structurelle	29, 51
DOD	2
mots réservés	5, 96
norme	2
paquetages	
Standard	99
Std_logic_1164	100
Textio	99
séquentiel	
description comportementale	36
description par flot de données	41
description structurelle	51
machines d'état	43
types physiques	6
temps	6
VHDL	2
VHSIC	2

9 *TABLE DES MATIÈRES*

Introduction	1.1 (2)
Bref historique	1.1 (2)
Utilité du VHDL	1.1 (2)
Spécification	1.2 (3)
Simulation	1.2 (3)
Conception	1.3 (4)
Comparaison avec ADA	2.1 (5)
Structure des programmes	2.1 (5)
Les signaux	2.1 (5)
Types physiques	2.2 (6)
Notion de temps	2.2 (6)
Deux types de délai : inertiel et transport	2.3 (7)
Le délai inertiel	2.4 (8)
Le délai transport	2.5 (9)
Utilisation du temps pour exprimer un retard	2.6 (10)
Utilisation du temps pour un module de simulation	2.7 (11)
Événements et transactions	2.7 (11)
Notion d'exécution concurrente et séquentielle	2.8 (12)
Programme à exécution concurrente :	2.9 (13)
Programme à exécution séquentielle	2.11 (15)
Différence entre l'utilisation d'une variable et d'un signal	2.14 (18)
Différence entre le bit et le boolean	2.15 (19)
Débuts en VHDL	3.1 (20)
Méthode de travail	3.1 (20)
Spécification et synthèse	3.3 (22)
L'interface VHDL	3.3 (22)
Le VHDL pour le combinatoire	3.5 (24)
Description par flot de données	3.6 (25)
L'additionneur	3.6 (25)
La bascule RS asynchrone	3.7 (26)
Le multiplexeur	3.8 (27)
La table de vérité	3.9 (28)
Résumé	3.10 (29)
Description structurelle	3.11 (30)
Résumé	3.12 (31)
Description comportementale	3.12 (31)
Résumé	3.14 (33)
Le VHDL pour un système séquentiel	3.14 (33)

Description comportementale	3.17 (36)
Description par flot de données	3.22 (41)
Description d'une machine séquentielle à partir d'un graphe d'états ou d'un organigramme	3.24 (43)
Transitions inconditionnelles	3.26 (45)
Transitions conditionnelles	3.26 (45)
Sorties inconditionnelles	3.27 (46)
Sorties conditionnelles	3.27 (46)
Résumé	3.28 (47)
Sorties post-synchronisées	3.28 (47)
Exemple : détecteur de sens	3.28 (47)
Description structurelle	3.33 (52)
La simulation	3.34 (53)
Polarité des entrées - sorties	3.41 (60)
Structure du VHDL	4.1 (63)
Structure générale d'un programme VHDL	4.1 (63)
Les bibliothèques	4.2 (64)
Que trouve-t-on dans une bibliothèque ?	4.2 (64)
Comment déclare-t-on une bibliothèque dans un fichier VHDL ?	4.2 (64)
L'élément à utiliser fait partie de notre espace de travail (projet).	4.2 (64)
L'élément à utiliser fait partie d'une autre bibliothèque	4.3 (65)
Fichiers de simulation des exemples	5.1 (66)
Paquetage simulation	5.1 (66)
Procédures et fonctions utilisables	5.1 (66)
Constantes	5.6 (71)
Types	5.7 (72)
Signaux	5.7 (72)
Code du paquetage de simulation	5.7 (72)
Code du paquetage de simulation pour le std_logic_1164	5.13 (78)
Fichier de simulation du décodeur sept segments	5.19 (84)
Fichier de simulation du comparateur	5.22 (87)
Fichier de simulation du compteur 4 bits	5.25 (90)
Fichier de simulation du détecteur de sens	5.28 (93)
Annexes	6.1 (95)
Les opérateurs VHDL	6.1 (95)
Les mots réservés	6.2 (96)
Passage de Abel en VHDL	6.3 (97)
Les paquetages standards	6.5 (99)
Standard	6.5 (99)
Textio	6.5 (99)

Std_logic_1164	6.6 (100)
Bibliographie	7.1 (101)
Les indispensables	7.1 (101)
Les autres	7.1 (101)
Index	8.1 (102)
Table des matières	9.1 (103)

Auteur : Claude Guex
E-mail : guex@einev.ch
Ecole : eivd - Ecole d'ingénieurs du Canton de Vaud, rte de Cheseaux 1, 1400 Yverdon-les-Bains, Suisse
Téléphone : +41 (0) 24 423.21.11 accès direct : +41 (0) 24 423.22.51
Fax : +41 (0) 24 425.00.50
Nom du fichier : VHDL_IMP.WPD
Version : 1
Date / heure : 22 juin 1998 / 11h01

Autres personnes à contacter en cas d'absence :

M. Boada Serge	mail : boada@einev.ch	téléphone direct : +41 (0) 24 423.22.53
M. Gaumain Maurice	mail : gaumain@einev.ch	téléphone direct : +41 (0) 24 423.22.53
M. Messerli Etienne	mail : messerli@einev.ch	téléphone direct : +41 (0) 24 423.22.54
M. Bornand Cédric	mail : cedric.bornand@einev.ch	téléphone direct : +41 (0) 24 423.22.54