LE LANGAGE DE DESCRIPTION VHDL

T. BLOTIN
Lycée Paul-Eluard
93206 SAINT-DENIS

SOMMAIRE

I. Le VHDL pour qui, pourquoi , quand, comment ?

A. Le VHDL !

VHDL ➔ VHSIC Hardware Description Language

Développé dans les années 80 aux États-Unis, le langage de description VHDL est ensuite devenu une norme IEEE numéro 1076 en 1987. Révisée en 1993 pour supprimer quelques ambiguïtés et améliorer la portabilité du langage, cette norme est vite devenue un standard en matière d'outils de description de fonctions logiques. A ce jour, on utilise le langage VHDL pour :

- ☞ concevoir des ASIC,
- ☞ programmer des composants programmables du type PLD, CPLD et FPGA,
- ☞ concevoir des modèles de simulations numériques ou des bancs de tests.

B. Pourquoi un langage de description ?

L'électronicien a toujours utilisé des outils de description pour représenter des structures logiques ou analogiques. Le schéma structurel que l'on utilise depuis si longtemps et si souvent n'est en fait qu'un outil de description graphique. Aujourd'hui, l'électronique numérique est de plus en plus présente et tend bien souvent à remplacer les structures analogiques utilisées jusqu'à présent. Ainsi, l'ampleur des fonctions numériques à réaliser nous impose l'utilisation d'un autre outil de description. Il est en effet plus aisé de décrire un compteur ou un additionneur 64 bits en utilisant l'outil de description VHDL plutôt qu'un schéma.

Le deuxième point fort du VHDL est d'être “un langage de description de haut niveau”. D'autres types de langage de description, comme l'ABEL par exemple, ne possèdent pas cette appellation. En fait, un langage est dit de haut niveau lorsqu'il fait le plus possible abstraction de l'objet auquel ou pour lequel il est écrit. Dans le cas du langage VHDL, il n'est jamais fait référence au composant ou à la structure pour lesquels on l'utilise. Ainsi, il apparaît deux notions très importantes :

➔ **portabilité** des descriptions VHDL, c'est-à-dire, possibilité de cibler une description VHDL dans le composant ou la structure que l'on souhaite en utilisant l'outil que l'on veut (en

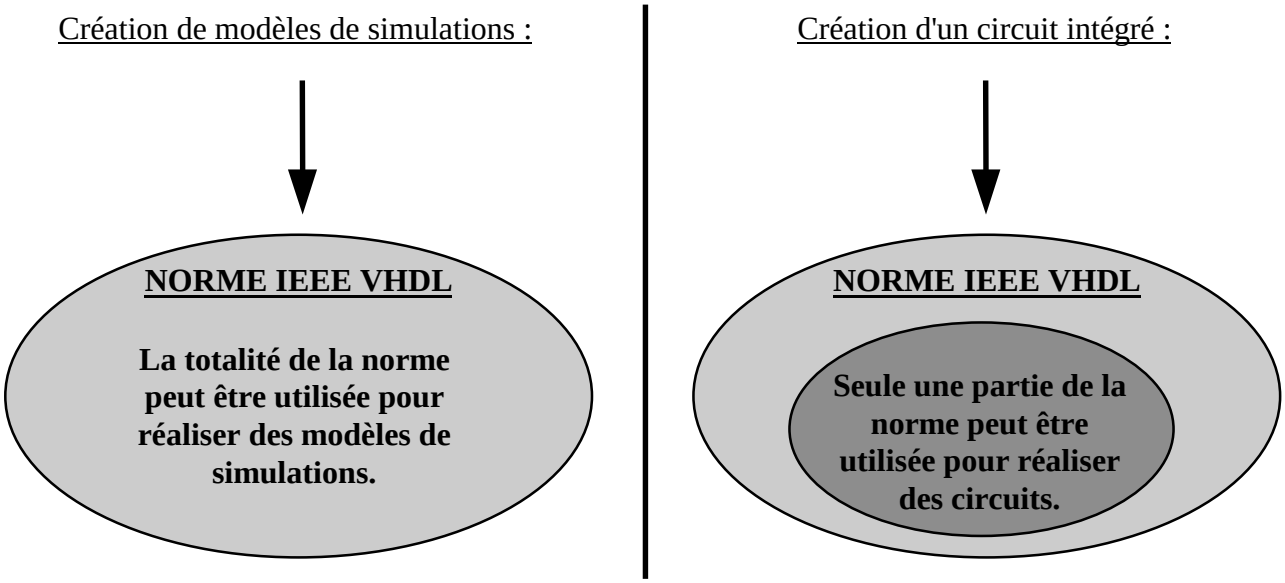
supposant, bien sûr, que la description en question puisse s'intégrer dans le composant choisi et que l'outil utilisé possède une entrée VHDL) ;

➔ **conception de haut niveau**, c'est-à-dire qui ne suit plus la démarche descendante habituelle (du cahier des charges jusqu'à la réalisation et le calcul des structures finales) mais qui se "limite" à une description comportementale directement issue des spécifications techniques du produit que l'on souhaite obtenir.

C. Les limites actuelles

La norme qui définit la syntaxe et les possibilités offertes par le langage de description VHDL est très ouverte. Il est donc possible de créer une description VHDL de systèmes numériques non réalisable, tout au moins, dans l'état actuel des choses. Il est par exemple possible de spécifier les temps de propagations et de transitions des signaux d'une fonction logique, c'est-à-dire créer une description VHDL du système que l'on souhaite obtenir en imposant des temps précis de propagation et de transition. Or les outils actuels de synthèses logiques sont incapables de réaliser une fonction avec de telles contraintes. Seuls des modèles théoriques de simulations peuvent être créés en utilisant toutes les possibilités du langage.

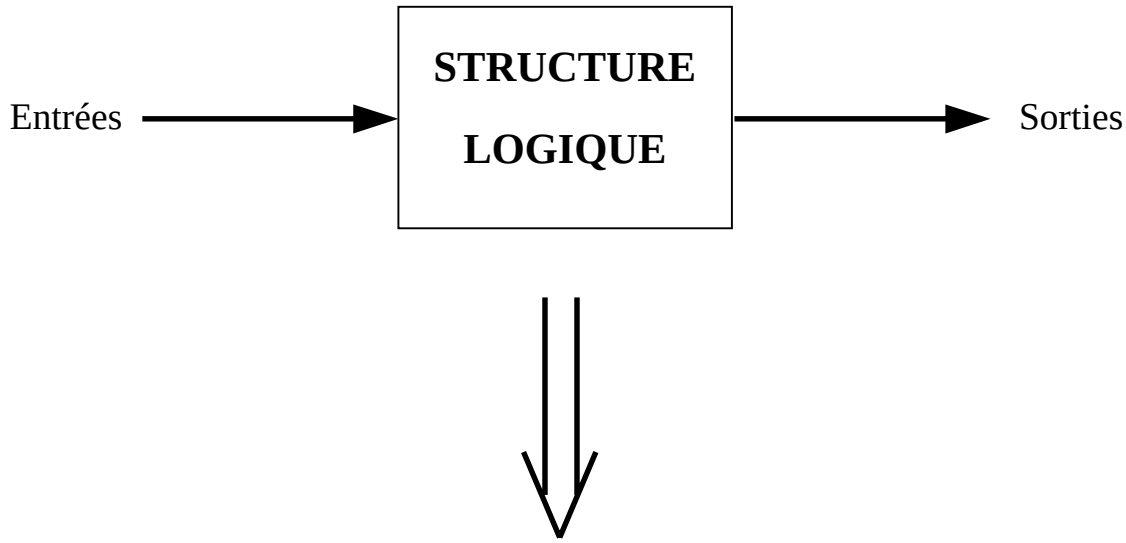
La situation peut donc se résumer de la façon suivante :



II. Structure d'une description VHDL

A. Entité et architecture

En VHDL, une structure logique est décrite à l'aide d'une entité et d'une architecture de la façon suivante :



ENTITE

ENTITY *Nom de l'entité* **IS**

Description des entrées et des sorties de la structure en explicitant pour chacune d'entre elles le nom, la direction (IN, OUT et INOUT) et le type.

END *Nom de l'entity* **;**

ARCHITECTURE

ARCHITECTURE *Nom de l'architecture* **OF** *Nom de l'entité* **IS**

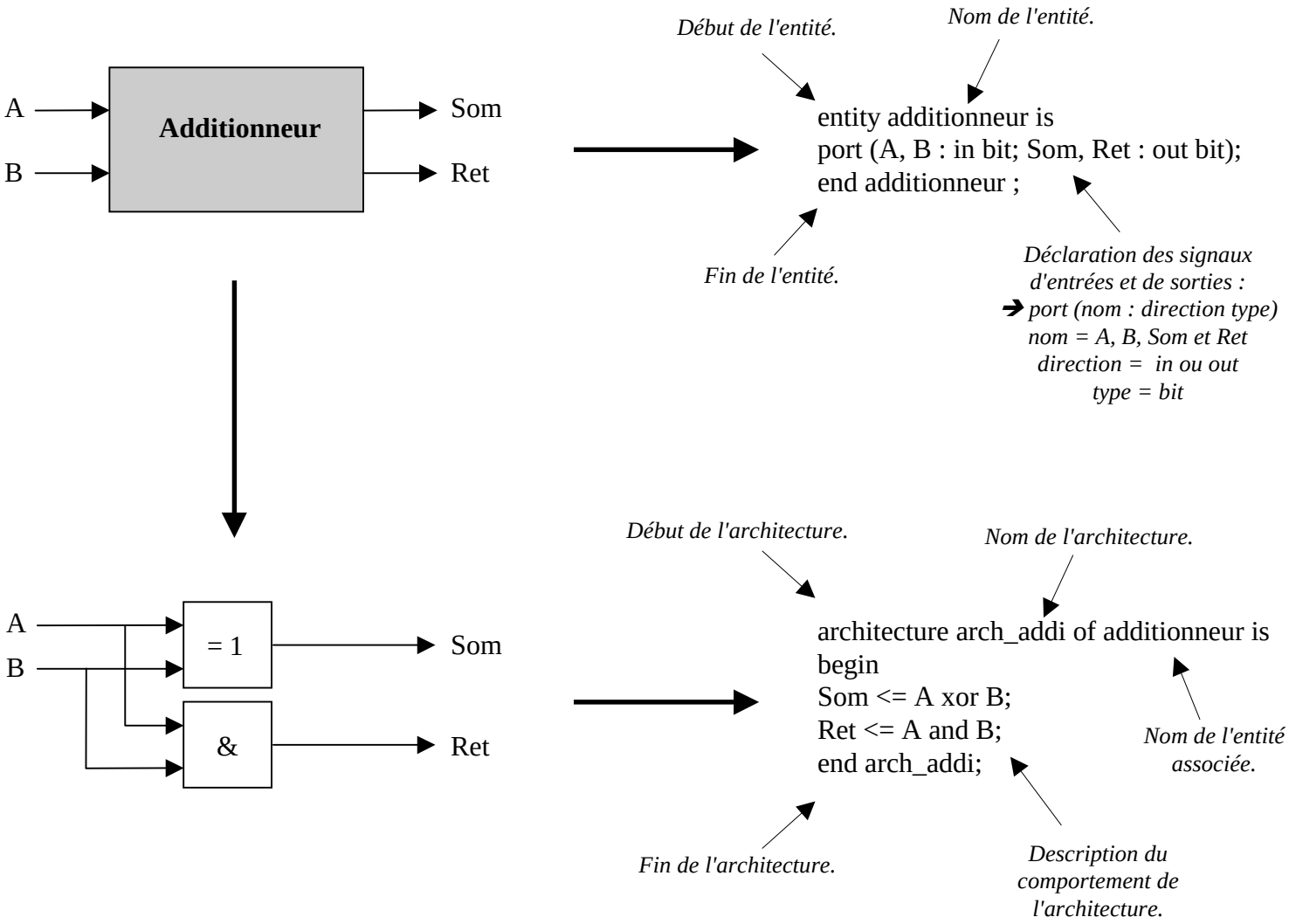
Zone de déclaration.

BEGIN

Description de la structure logique.

END *nom de l'architecture* **;**

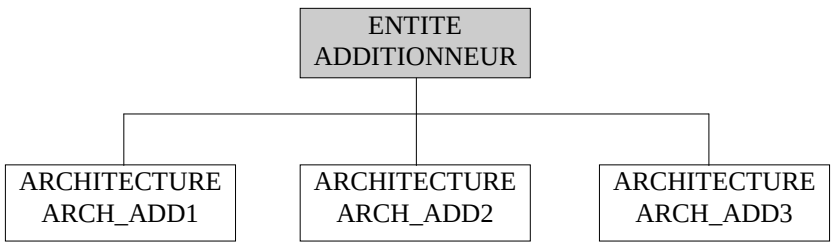
Prenons l'exemple d'un additionneur 1 bit.



☞ **L'entité donne les informations concernant les signaux d'entrées et de sorties de la structure ainsi que leurs noms et leurs types.**

☞ **L'architecture décrit le comportement de l'entité.**

☞ **Il est possible de créer plusieurs architectures pour une même entité. Chacune de ces architectures décrira l'entité de façon différente.**



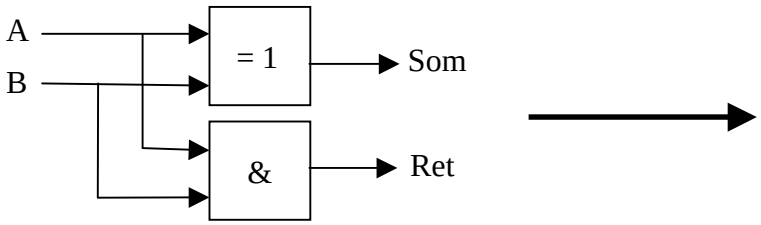
B. Description comportementale et structurelle

Il existe deux façons distinctes de décrire une structure. L'une dite **comportementale** et l'autre **structurelle**.

Description comportementale

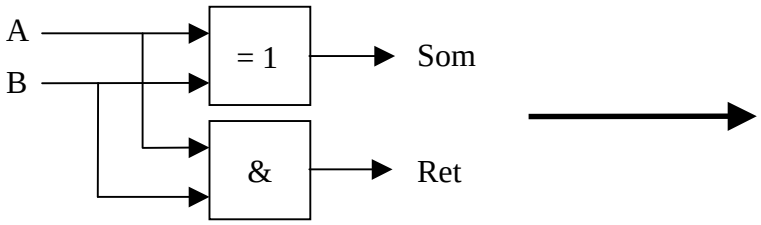
Dans ce type de description, le comportement de la structure est directement inscrit dans l'architecture à l'aide d'instructions séquentielles ou sous forme de flow de données.

Description comportementale à l'aide d'instructions séquentielles :




```
architecture Arch_comp1 of entité_X is
begin
        process  (A,  B)
        begin
                if  (A = '0'  and  B  = '0')  then
                        Som  <=  '0';  Ret  <=  '0';
                if  (A = '0'  and  B  = '1')  then
                        Som  <=  '1';  Ret  <=  '0';
                if  (A = '1'  and  B  = '0')  then
                        Som  <=  '1';  Ret  <=  '0';
                if  (A = '1'  and  B  = '1')  then
                        Som  <=  '1';  Ret  <=  '1';
                end  if;
        end  process;
end arch_comp1;
```

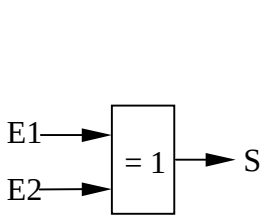
Description comportementale sous forme de flow de données :



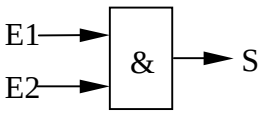

```
architecture Arch_comportementale of entité_X is
begin
        Som <= A xor B;
        Ret <= A and B;
end arch_comportementale;
```

Description structurelle

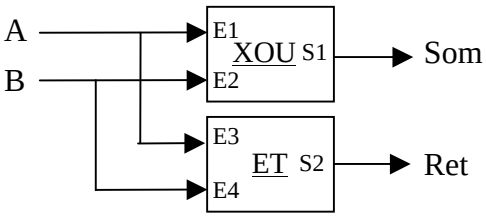
Dans ce type de description, on relie entre eux des composants préalablement décrits en VHDL.




```
entite XOU is
port (E1, E2 : in bit; S1 : out bit);
end XOU;

architecture A_XOU of XOU is
begin
S1 <= E1 XOR E2;
end A_XOU;
```




```
entite ET is
port (E2, E3 : in bit; S2 : out bit);
end ET;

architecture A_ET of ET is
begin
S2 <= E3 AND E4;
end A_ET;
```




```
Entité add is
port (A, B : in bit; Som, Ret : out bit);
end add;

architecture Arch_structurelle of add is
```

Déclaration du composant XOU.

```
component XOU
port (E1, E2 : in bit; S : out bit);
end component;
```

Déclaration du composant ET.

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

Spécification de l'entité et de l'architecture ‡ utiliser pour les composants ET et XOU.

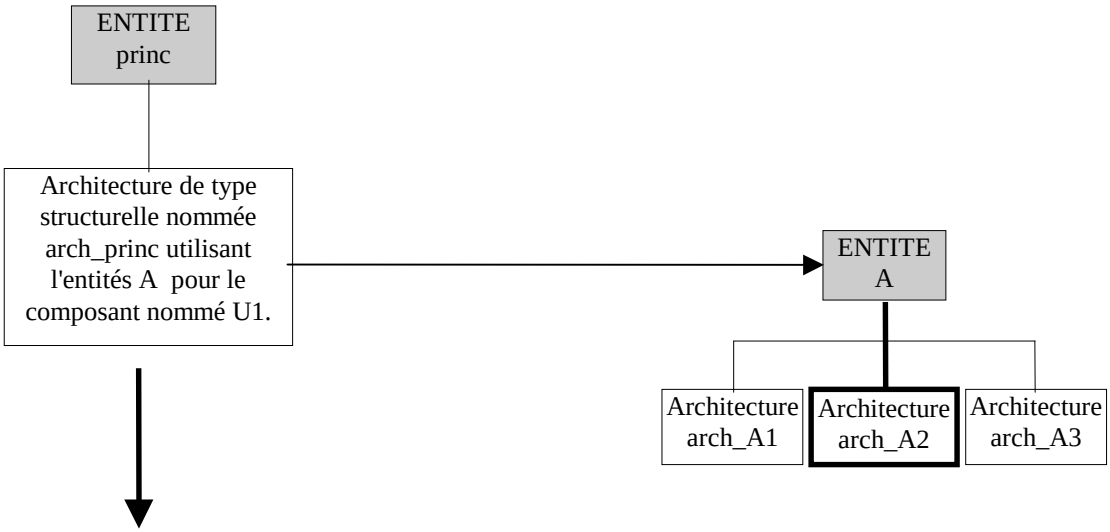
```
for U1 : XOU use entity work.XOU(A_XOU);
for U2 : ET use entity work.ET(A_ET);

begin
```

Câblage des composants XOU et ET appelé U1 et U2.

```
U1 : XOU port map (A, B, Som);
U2 : ET port map (A, B, Ret);
end arch_structurelle;
```

C. Configuration, package et package body

Admettons que plusieurs architectures aient été créées pour l'entité A utilisée au sein de l'architecture arch_princ de l'entité princ. Lors de la synthèse ou de la simulation de l'entité princ, il va être nécessaire de spécifier quelles sont les architectures à utiliser. Cette spécification peut être faite grâce à l'utilisation d'une configuration.




```
Configuration conf_princ of Princ is
for arch_princ
        for U1 : A use entity
        work.A(arch_A2)
        end for;
end for;
end conf_princ;
```

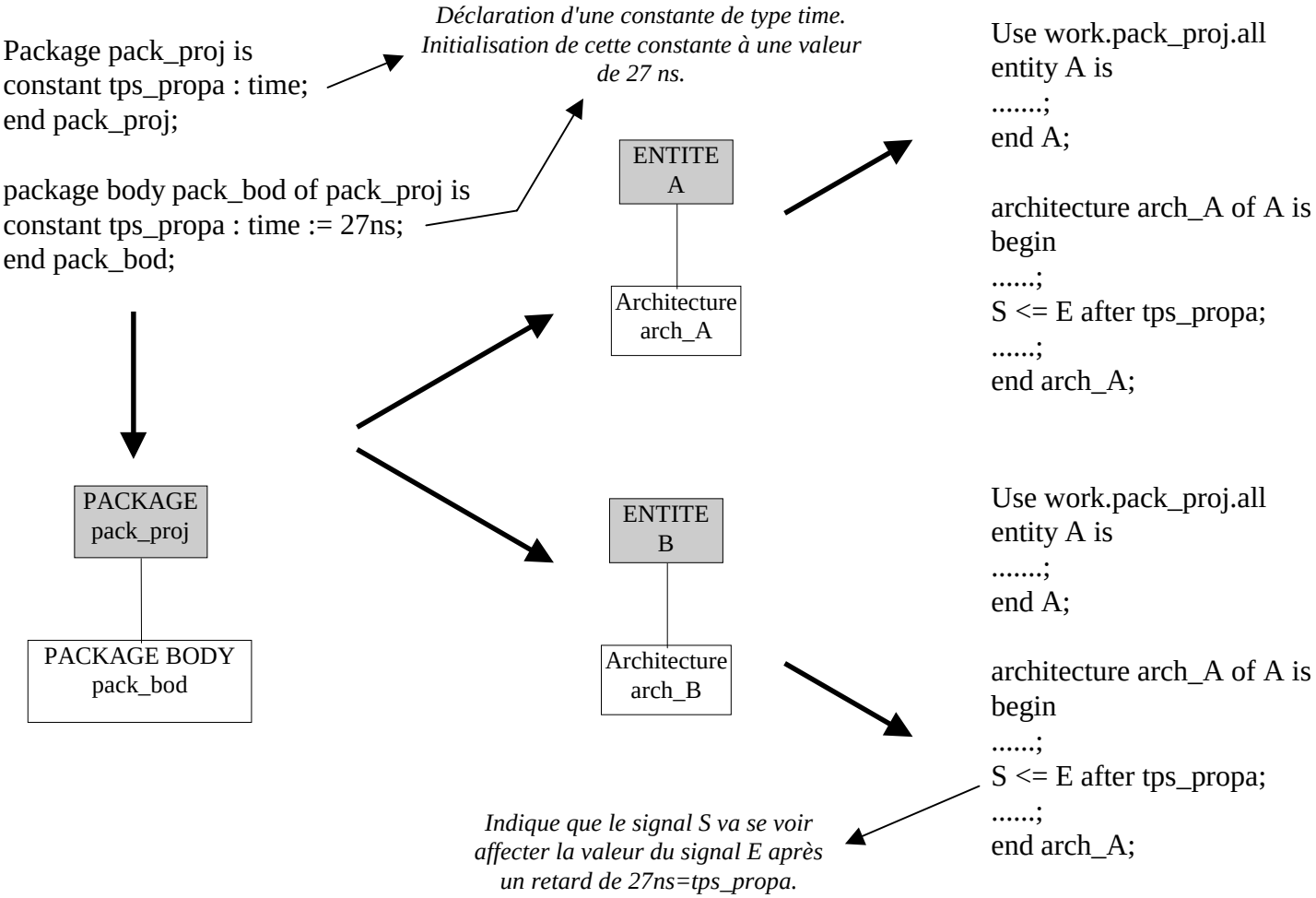
Se lit pour le composant appelé U1 de l'architecture arch_princ de l'entité nommée princ, utiliser l'architecture arch_A2 de l'entité A.

☞ **La configuration permet, comme son nom l'indique, de configurer l'entité à laquelle elle est associée. On peut, par exemple, spécifier les architectures à utiliser pour tous les composants déclarés au sein d'une description.**

☞ **Pour qu'une description soit portable, c'est-à-dire synthétisable ou simulable sur des outils différents et pour des composants cibles différents, il est préférable de n'avoir qu'une seule architecture par entité. Les configurations deviennent alors inutiles.**

Dans le langage VHDL, il est possible de créer des “package” et “package boy” dont les rôles sont de permettre le regroupement de données, variables, fonctions, procédures, etc., que l'on souhaite pouvoir utiliser ou appeler à partir d'architectures. Il est ainsi possible de se créer une sorte de bibliothèque d'objets (constantes, variables particulières, fonctions ou procédures spécifiques, etc.) que l'on pourra appeler à partir de plusieurs descriptions. Cette possibilité est particulièrement intéressante lorsque l'on utilise, au sein d'un projet contenant plusieurs entités, des ressources (constantes, variables particulières, fonctions...) identiques.

Imaginons, par exemple, qu'un projet contenant deux entités A et B aux architectures respectives arch_A et arch_B utilise la constante de temps tps_propa = 27ns. L'utilisation d'un package nommé pack_proj et d'un package body nommé pack_bod donnerait :

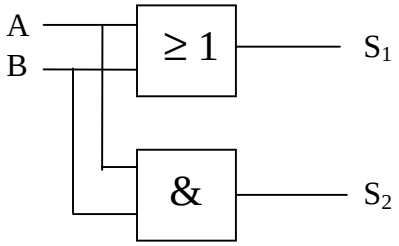


☞ **La ligne de commande “*use work.pack_proj.all*” qui se trouve en tête des déclarations des entités a pour but de prévenir l'analyseur ou le synthétiseur que l'on va utiliser des éléments du package pack_proj, en l'occurrence la constante de type time tps_propa initialisée à 27ns.**

III. Les instructions concurrentes et séquentielles

A. La nécessité d'utiliser des instructions concurrentes

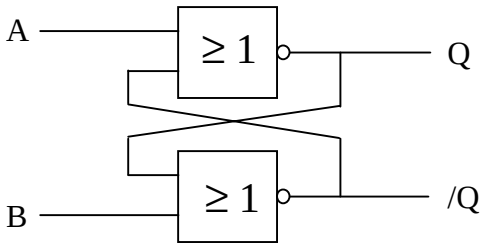
Lorsque l'on étudie une structure logique, il apparaît à l'évidence que les signaux qui la constituent évoluent de façon indépendante et parallèle. Prenons l'exemple simple suivant :




```
architecture ET_OU of ET_OU is
begin
        S2 <= A and B;
        S1 <= A or B;
end ET_OU;
```

Dans cet exemple, les signaux S_1 et S_2 évoluent de façon indépendante et parallèle. Il est donc impossible de décrire leur évolution de façon séquentielle. Il faut avoir recours à des instructions de types concurrents qui s'exécutent en parallèle indépendamment de leur ordre d'écriture. Ainsi, à l'exécution de la description ci-dessus, c'est-à-dire lors de la simulation, les signaux S1 et S2 seront réactualisés en continu, en fonction des valeurs de A et de B et de façon parallèle. **On parle alors d'instructions concurrentes.**

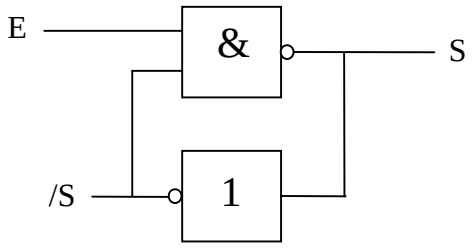
Mais alors que se passe-t-il si la structure présente des rebouclages avec ses sorties comme la bascule RS, par exemple ? En fait, le simulateur regarde en permanence si le changement de valeur prévu d'un signal ne va pas entraîner celui d'un ou plusieurs autres signaux et réagit en conséquence. Dans le cas de la bascule RS,



si le changement d'état de l'entrée A entraîne celui de la sortie Q, alors le simulateur va calculer l'état futur de la sortie /Q et, si cela est nécessaire, calculer à nouveau l'état de Q, puis celui de /Q, etc., jusqu'à arriver à une situation stable. On peut résumer le fonctionnement du simulateur de la façon suivante :

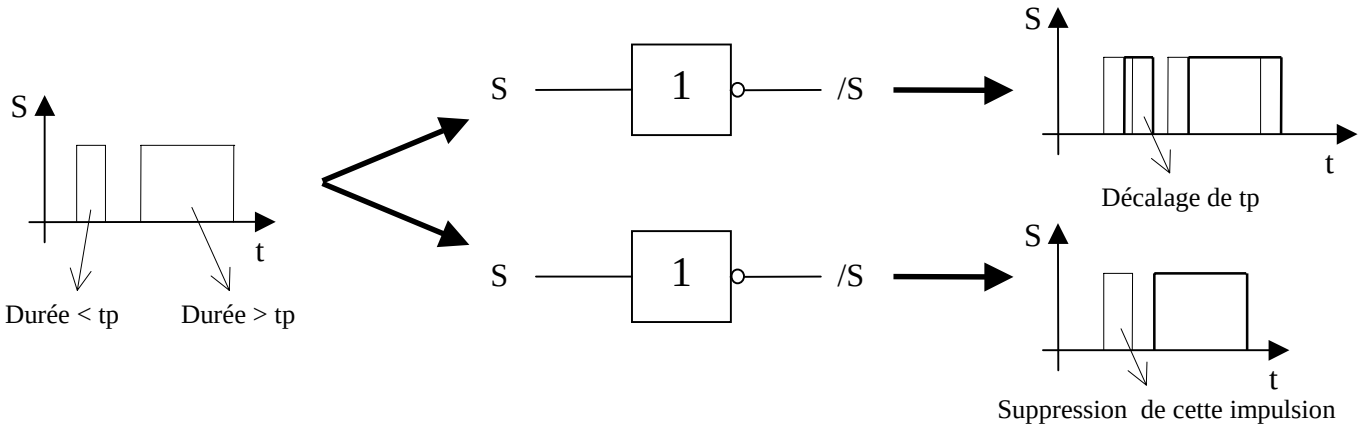
☞ **A chaque changement d'état d'un ou de plusieurs signaux d'entrées, le simulateur calcule l'état des signaux de sorties en tenant compte des rebouclages jusqu'à aboutir à une situation stable. C'est à ce moment, et seulement à ce moment, que les valeurs des signaux de sorties sont réactualisées.**

Notons que cette règle de fonctionnement laisse entrevoir des possibilités simples de blocage du simulateur. Prenons l'exemple du montage suivant :



Lors de la simulation de ce montage, tant que E = '0' le simulateur donne S = '1' mais dès que E = '1', le simulateur n'arrive pas à résoudre l'équation "S <= not (E and not (S))". Pour que ce montage puisse être simulé, il faudrait introduire des temps de propagation aux portes logiques. On peut alors rajouter à la règle de fonctionnement précédente que le simulateur stocke en mémoire les évolutions futures des signaux du montage. Dans notre exemple, le signal /S va évoluer après le temps tp de propagation de la porte NON. Le simulateur va donc mémoriser que le signal /S doit évoluer dans tp secondes après l'évolution de S. En prolongeant la simulation, on pourra constater l'évolution du signal /S.

On peut alors chercher à savoir ce qui se passe lorsque le signal S évolue plus rapidement que le temps de propagation de la porte. La réponse est simple, il y a en VHDL deux cas de figure :



☞ **Dans le premier cas, on parle de mode “transport” : toutes les transitions en entrée sont transmises en sortie avec un décalage égal au temps de propagation.**

☞ **Dans le deuxième cas, on parle de mode “inertiel” : les transitions présentes en entrée et plus courtes que le temps de propagation ne sont pas transmises.**

Le langage VHDL propose ces deux modes d'affectation. Le mode inertiel est celui par défaut, alors que le mode transport doit être rajouté dans l'instruction d'affectation :

/S <= **transport** S after 10ns;

On peut dissocier trois types d'instructions concurrentes :

➔ les instructions d'affectation utilisée au sein des exemples précédents qui correspondent à l'écriture d'une équation booléenne,

➔ les instructions conditionnelles du type "affectation When condition else ...",

➔ les instructions conditionnelles du type "With signal select affectation When condition else ...",

L'instruction When else

Cette instruction permet d'affecter un signal à une expression (expression = signal, équation ou valeur) pour certaines conditions (conditions sur les signaux de type IN ou INOUT). Voici un exemple d'utilisation de ce type d'instruction lors de la description d'un multiplexeur :

```
entity exemple is
port (A, B, C : IN bit; Sel : IN bit_vector (1 downto 0); S : OUT bit);
end exemple;

architecture exemple of exemple is
begin
S <= A when (Sel = "00") else
     B when (Sel = "01") else
     C when (Sel = "10") else
     '0';
end exemple;
```

Ici il s'agit de la définition d'un bus de signaux. Cette déclaration est équivalente à S1 : IN bit ; S2 : IN bit; S comporte donc deux éléments identiques à S1 et S2 (voir paragraphe B.3 ci-après).

L'instruction With select when

Cette instruction est semblable à la précédente avec en plus une précision préalable du signal sur lequel vont se porter les conditions. En reprenant l'exemple précédent, on obtient donc :

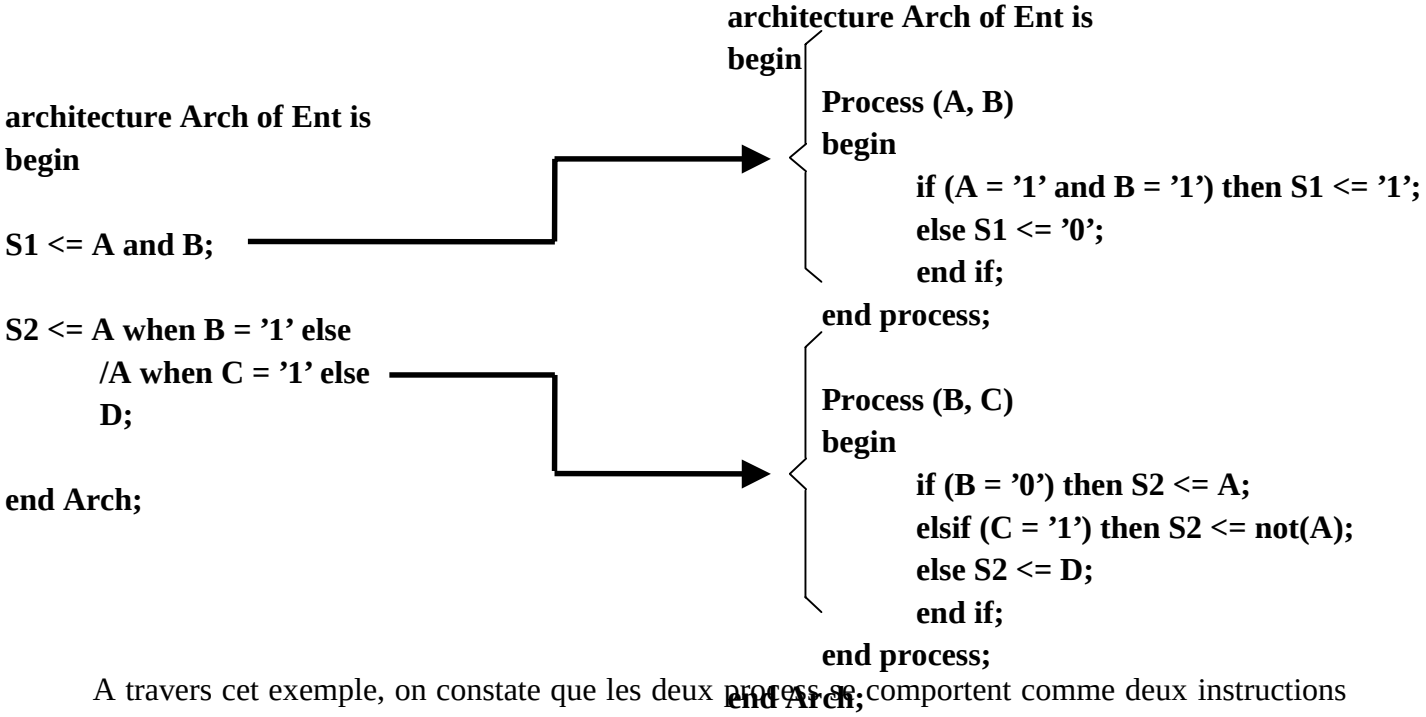
```
entity exemple is
port (A, B, C : IN bit; Sel : IN bit_vector (1 downto 0); S : OUT bit);
end exemple;

architecture exemple of exemple is
begin
With Sel select
S <= A when "00",
     B when "01",
     C when "10",
     '0' when others;
end exemple;
```

Ici il s'agit de la définition d'un bus de signaux. Cette déclaration est équivalente à S1 : IN bit ; S2 : IN bit; S comporte donc deux éléments identiques à S1 et S2 (voir paragraphe B.3 ci-après).

B. Les instructions séquentielles

B. 1. Les process

Par opposition aux instructions concurrentes présentées au paragraphe précédent, le langage VHDL propose un jeu très complet d'instructions séquentielles identiques à celles que l'on trouve dans les langages de programmation évolués C++, Pascal, etc. Mais, comme on a pu le constater au paragraphe précédent, ces types d'instructions ne peuvent être utilisés pour décrire les phénomènes (évolution de signaux indépendants) que l'on rencontre dans les montages à base de composants logiques. Pourtant, il aurait été dommage de se passer de la puissance et du confort qu'offre ce type d'instructions. Pour palier l'incompatibilité qui existe entre les instructions séquentielles et le fonctionnement des montages que l'on souhaite décrire, le langage VHDL propose une solution simple qui consiste à créer des ensembles appelés “process”, regroupant des instructions séquentielles et se comportant, d'un point de vue externe, comme des instructions concurrentes. Voici un exemple d'équivalence entre deux descriptions, l'une à base d'instructions concurrentes et l'autre à base d'instructions séquentielles regroupées au sein d'un process.



A travers cet exemple, on constate que les deux process se comportent comme deux instructions concurrentes.

☞ **L'exécution d'un process est concurrente même si les instructions qu'il contient sont séquentielles.**

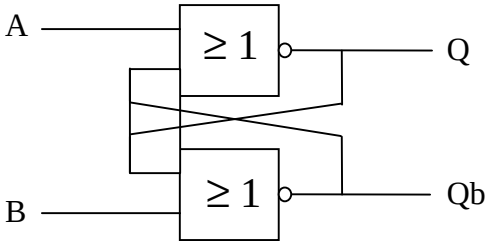
Notons que la déclaration d'un process débute par le mot clé “process”, suivi d'une liste de noms de signaux. Cette liste est appelée “liste de sensibilité”, elle contient le nom des signaux

dont le changement d'état va provoquer l'exécution du process. En d'autres termes, le process dont la déclaration est la suivante : process (A, B) va être exécuté à chaque changement d'état de l'un (ou des deux) signal(aux) A et B.

☞ **L'exécution d'un process n'a lieu qu'en cas de changement d'état de l'un (ou de plusieurs) signal(aux) compris dans la liste de sensibilité.**

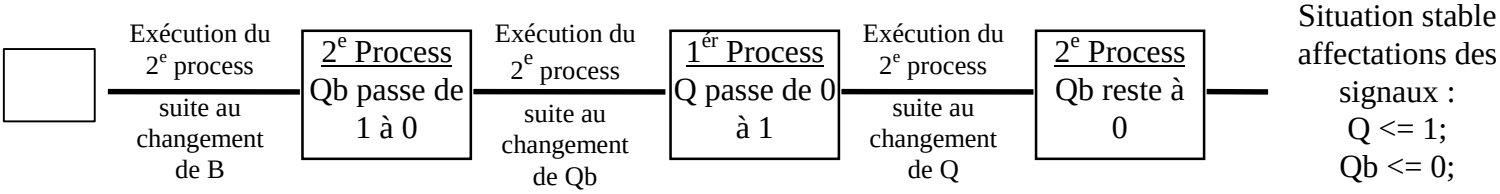
Au paragraphe précédent, nous avons constaté, grâce à l'exemple de la bascule RS, que l'exécution d'une instruction concurrente pouvait entraîner celle d'une autre instruction de même type, qui elle-même pouvait entraîner…, etc. Ce principe s'applique aux process dont l'exécution peut, par conséquent, entraîner celle d'un autre process, qui elle-même peut entraîner…, etc.

Si l'on reprend l'exemple de la bascule RS décrite à l'aide de deux process, on obtient :




```
Entity RS is
port (A, B : in bit; Q, Qb : inout bit);
end RS;

architecture RS of RS is
begin
        process (A, Qb)
        begin
        Q <= not (A or Qb);
        end process;

        process (Q, B)
        begin
        Qb <= not (B or Q);
        end process;
end RS;
```

On suppose au départ que A et B = '0' avec Q = '0' et Qb = '1' puis B passe à '1'. L'exécution des deux process de l'architecture RS peut se représenter de la façon suivante :



B. 2. Les boucles et instructions if, then, else, for, case...

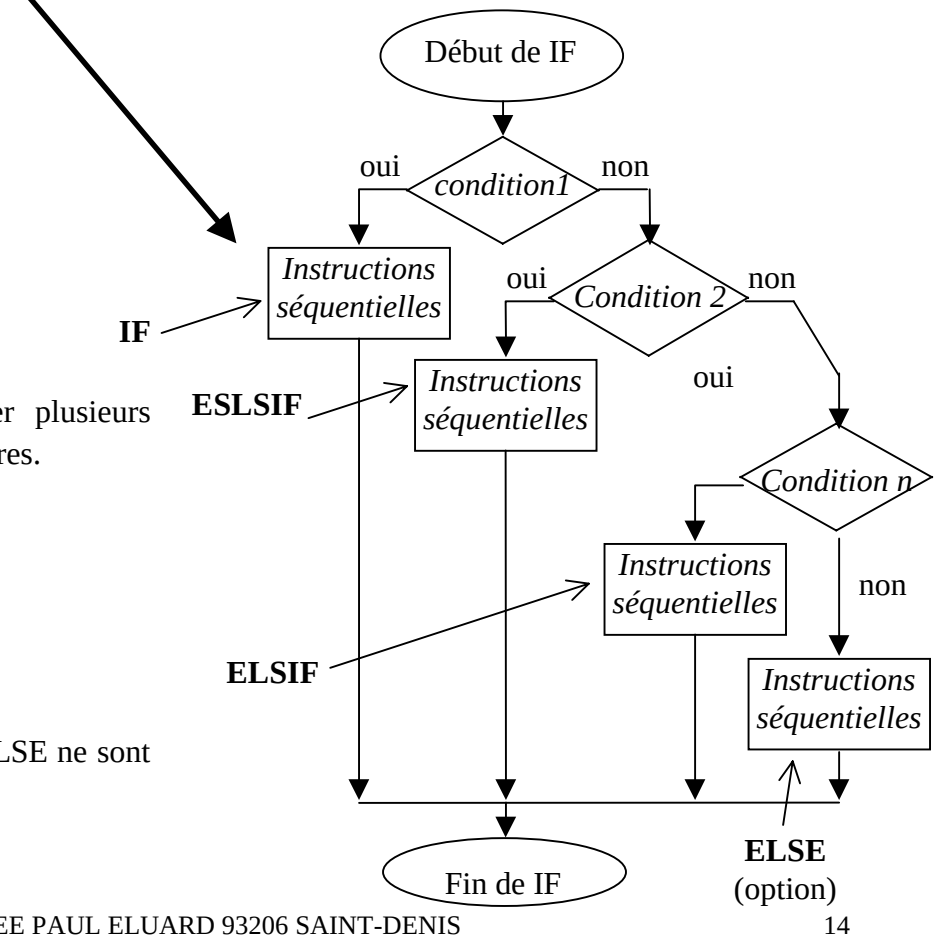
Les instructions séquentielles du type if, then, else... sont très puissantes et permettent la réalisation de boucles conditionnelles intéressantes. De plus, la syntaxe qui leur est associée ne présente pas de difficultés particulières et permet une écriture des descriptions très lisible. Ainsi,

pour bien utiliser ce type d'instructions, il suffit souvent de comprendre leurs significations. Vous trouverez ci-dessous quelques exemples de traductions de boucles ou d'ensembles d'instructions conditionnelles.

INSTRUCTION IF

SI *condition1* **ALORS**
*instructions séquentielles***;**
SINON SI *condition2* **ALORS**
*instructions séquentielles***;**
SINON SI *condition3* **ALORS**
*instructions séquentielles***;**
⋮
SINON
*instructions séquentielles***;**
FIN DU SI;

IF *condition1* **THEN**
*instructions séquentielles***;**
ELSIF *condition2* **THEN**
*instructions séquentielles***;**
ELSIF *condition3* **THEN**
*instructions séquentielles***;**
⋮
ELSE
*instructions séquentielles***;**
END IF;



Remarques :

❶ Il est possible d'imbriquer plusieurs boucles IF les unes dans les autres.

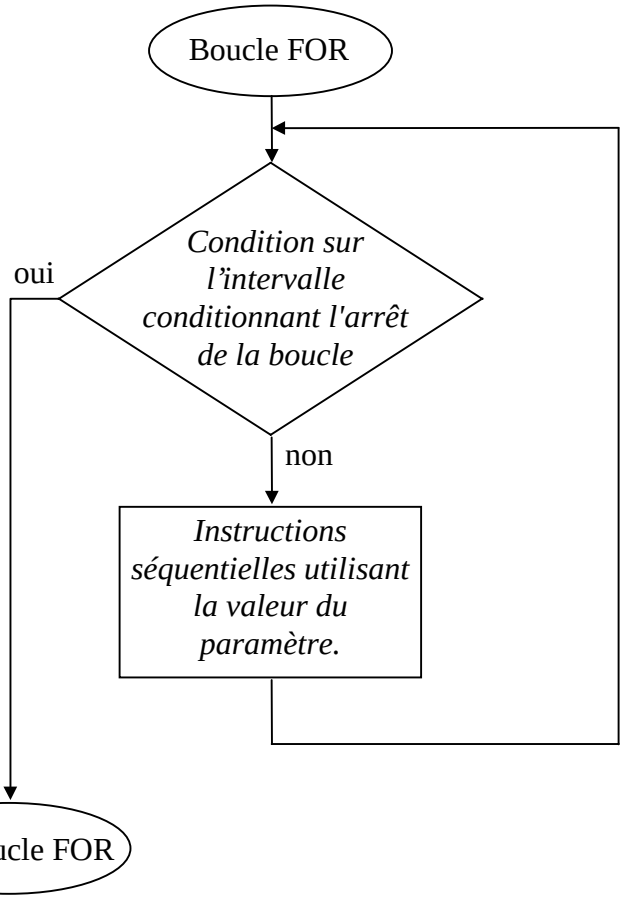
```
IF ... THEN
    IF ... THEN
    ELSIF ... THEN
    END IF;
ELSE
END IF;
```

❷ Les instructions ELSIF et ELSE ne sont pas obligatoires.

BOUCLE FOR

FOR *paramètre* **IN** *intervalle* **LOOP**
*instructions séquentielles***;**
END LOOP;

POUR *le paramètre* **COMPRIS**
dans l'intervalle **EXECUTER**
*instructions séquentielles***;**
FIN DE LA BOUCLE;



Exemple :

```
FOR i IN 0 to 3 LOOP
        IF (A = i) THEN
        S <= B;
        END IF;
END LOOP;
```

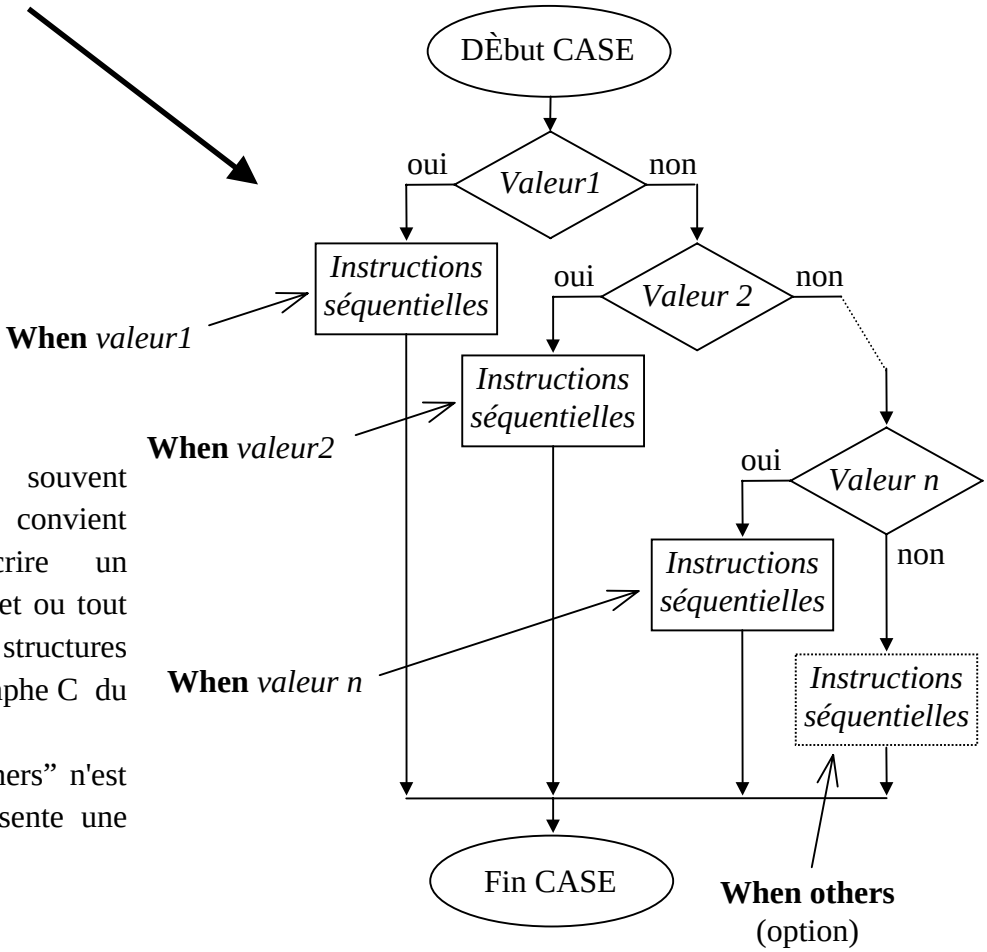
Exécuter, pour i = 0, i = 1, i = 2 puis i = 3
les instructions suivantes :

```
        SI (A = i) THEN
        S <= B;
        END IF;
FIN DE LA BOUCLE;
```

INSTRUCTION CASE

CASE *signal* **IS**
WHEN *valeur1* => *instructions séquentielles*;
WHEN *valeur2* => *instructions séquentielles*;
WHEN *valeur3* => *instructions séquentielles*;
WHEN OTHERS => *instructions séquentielles*;
END CASE;

CAS *possibles de l'expression* **EST**
LORSQUE *signal = valeur1* => *instructions séquentielles*;
LORSQUE *signal = valeur2* => *instructions séquentielles*;
LORSQUE *signal = valeur3* => *instructions séquentielles*;
LORSQUE *signal* = **AUTRES** => *instructions séquentielles*;
FIN DE CAS;



Remarques :

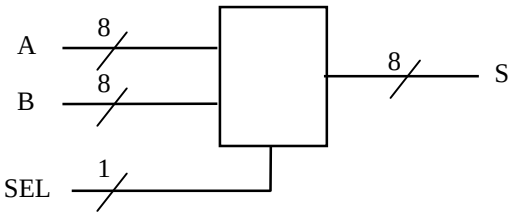
❶ L'instruction CASE, souvent appelée "switch case", convient parfaitement pour décrire un diagramme d'état, un grafcet ou tout autre formalisme de structures séquentielles (voir paragraphe C du chapitre VII).

❷ L'instruction "When others" n'est pas obligatoire, elle représente une facilité d'écriture.

B. 3. Les vecteurs de signaux

Au sein des exemples précédents, nous n'avons utilisé que des signaux uniques et jamais de vecteurs ou de bus de signaux. Or le langage de description VHDL permet la déclaration et la manipulation de ce type de signaux. Voici un exemple commenté de description qui utilise des vecteurs ou des bus de signaux :

Multiplexeur de vecteurs de signaux :



S = A si SEL = ’1’ et
S = B si SEL = ’0’.

```
Entity MUX is
      port (SEL : in bit;
      A : in bit_vector (7 downto 0);
      B : in bit_vector (7 downto 0);
      S : out bit_vector (7 downto 0));
end MUX;

architecture Arch_MUX of MUX is
begin
      S <= A when SEL = '1' else
      S <= B when SEL <= '0';
end Arch_MUX;
```

Déclaration de trois signaux correspondant à des bus (un vecteur de signaux) comportant 8 signaux A(0), A(1) , ..., A(7) et B(0), B(1) , ..., B(7) ainsi que S(0), S(1) , ..., S(7).

➔ *Pour assigner une valeur à un des éléments d'un bus, on utilise le nom du bus suivi du numéro de l'élément, exemple :*
B(3) <= '1'; l'élément 3 du bus B prend la valeur '1'.

➔ *Pour assigner une valeur à l'ensemble du bus, plusieurs méthodes sont possibles, exemples:*

❶ *A <= "11001010"; assigne la valeur 11001010 aux éléments A(7), A(6), ..., A(0).*

❷ *A <= (7 => '1', 6 => '1', 5 downto 4 => '0', 3 => '1', 2 => '0', 1 => '1', 0 => '0'); assigne la valeur 11001010 aux éléments A(7), A(6), ..., A(0).*

❸ *S <= (7 => '1', others => 'z'); assigne la valeur 1ZZZZZZZ au vecteur S.*

B. 4. La déclaration GENERIC

Au sein de la description VHDL d'une structure logique, il est possible de rajouter une déclaration de GENERIC correspondant à des paramètres. Ces paramètres pourront, par la suite, être modifiés lors de l'utilisation de la description en temps que composant. Typiquement, on utilise une déclaration de GENERIC pour spécifier des temps de propagation et autres délais de portes logiques. Ces temps seront alors modifiables lors de l'utilisation de ces portes logiques dans une description structurelle. Voici un exemple d'utilisation de déclarations de GENERIC :

```
Entity OU is
GENERIC (TP : time := 20 ns);
port (E1, E2 : in bit; S1 : out bit);
end OU;

architecture OU of OU is
begin
S1 <= E1 or E2 after TP;
end OU;
```

```
Entity ET is
GENERIC (TP : time := 0 ns);
port (E1, E2 : in bit; S1 : out bit);
end ET;

architecture ET of ET is
begin
S1 <= E1 and E2 after TP;
end ET;
```

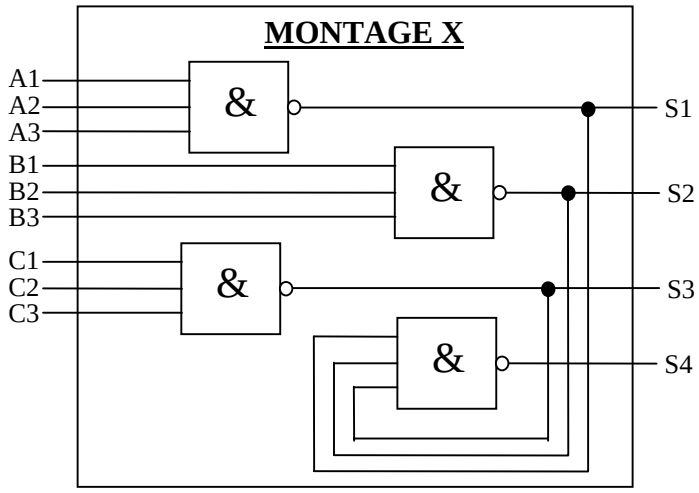
```
Entity ET_OU is
port (A, B : in bit; X, Y : out bit);
end ET_OU;

architecture ET_OU of ET_OU is

component OU
generic (TP : time);
port (E1, E2 : in bit; S1 : out bit);
end component;

component ET
generic (TP : time);
port (E1, E2 : in bit; S1 : out bit);
end component;

for U1 : OU use entity work.OU(OU);
for U2 : ET use entity work.ET(ET);

begin
U1 : OU     generic map ( TP => 5 ns )
            port map (A, B, X);

U2 : ET     generic map ( TP => 15 ns )
            port map (A, B, Y);

end ET_OU;
```

IV. Les fonctions et procédures

A. Rôle, principe et fonctionnement

Le langage VHDL permet l'utilisation et la création de fonctions ou de procédures que l'on peut appeler à partir d'une architecture. Le rôle de ces fonctions ou procédures est de permettre, au sein d'une description, la création d'outils dédiés à certaines tâches pour un type déterminé de signaux (voir chapitre suivant concernant les types de signaux). Cette notion d'objets est typique aux langages évolués de programmation.

Dans le montage X suivant, on utilise à quatre reprises la fonction logique NONET à trois entrées. On décide donc de décrire ce montage en créant une description VHDL comportant une fonction nommée NONET.




```
Entity  montage_X  is
port (A1, A2, A3, B1, B2, B3, C1, C2, C3 :  IN  bit ;
        S1, S2, S3 : INOUT  bit ;  S4  :  OUT  bit) ;
end montage_X ;

Architecture  Arch  of  montage_X  is

function NONET (A, B, C : bit) return bit is
variable result : bit;
begin
result := not (A and B and C);
return result;
end;

begin
S1  <=  NONET  (A1,  A2,  A3);
S2  <=  NONET  (B1,  B2,  B3);
S3  <=  NONET  (B1,  B2,  B3);
S4  <=  NONET  (S1,  S2,  S3);
end Arch;
```

Déclaration des signaux S1, S2 et S3 avec la direction INOUT car ils sont utilisés au sein de la description

création de la fonction NONET

Utilisation de la fonction NONET

La syntaxe d'une fonction est donc la suivante :

FUNCTION *nom de la fonction* **(***liste des paramètres de la fonction avec leur type***) RETURN** *type du paramètre de retour* **IS**
*zone de déclaration des variables***;**
BEGIN
.....;
*instructions séquentielles***;**
.....;
RETURN *nom de la variable de retour ou valeur de retour***;**
END;

Une fonction reçoit donc des paramètres d'entrées et renvoie un paramètre de sortie. Dans l'exemple précédent, la fonction reçoit trois paramètres A, B et C et retourne un paramètre S. Le mot clé RETURN permet d'associer au paramètre de sortie une valeur. Une fonction peut donc contenir plusieurs RETURN, exemple :

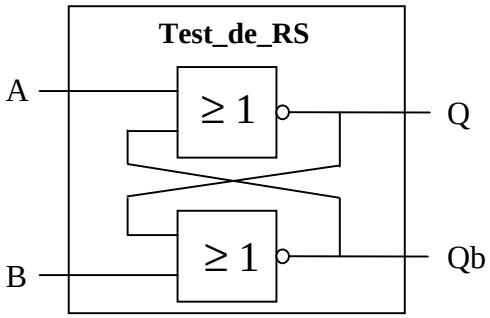
```
FUNCTION  bool_vers_bit  (X : boolean)  RETURN  bit  IS
BEGIN
        if  X  then
                RETURN  '1';
        else
                RETURN  '0';
        end if;
END;
```

Cette fonction, qui convertit un signal de type booléen en un signal de type bit, possède deux RETURN. L'un pour renvoyer la valeur '1' et l'autre pour la valeur '0'. Notons qu'il est possible de déclarer une variable interne à la fonction que l'on pourra affecter au paramètre de sortie (voir exemple précédent), ce type d'affectation "signal <= variable" est impossible en dehors des fonctions, procédures ou process.

En ce qui concerne l'appel d'une fonction, il peut être fait à partir d'instructions séquentielles ou concurrentes. Dans le cas des instructions concurrentes, la fonction sera toujours vérifiée.

En ce qui concerne les procédures, elles diffèrent des fonctions par le fait qu'elles acceptent des paramètres dont la direction peut être IN, INOUT et OUT. Une procédure ne possède donc pas un ensemble de paramètres d'entrées et un paramètre de sortie mais un

ensemble de paramètres d'entrées-sorties et aucun paramètre spécifique de sortie. Prenons l'exemple de la description d'une bascule RS à travers une procédure :




```
Entity  test_de_rs is
port (E1,  E2  :  IN  bit;  S1,  S2  INOUT  bit);
end test_de_RS;

Architecture  test_de_RS  of  test_de_RS  is

PROCEDURE  RS  (signal  A,  B  :  IN  bit;  signal  Q,  Qb  :  INOUT  bit)  IS
BEGIN
Q  <=  not (A  or  Qb);
Qb  <=  not (B  or  Q);
END;

begin
RS  (E1,  E2,  S1,  S2);
end test_de_RS;
```

On constate que les paramètres A, B, Q et Qb de la procédure ont été explicitement décrits. Ainsi, la liste des paramètres contient :

– *le genre de paramètre*, variable (valeur par défaut), **signal** ou **constante**,

– *le nom des paramètres*, dans notre cas, A, B, Q et Qb (ces noms ne sont connus en temps que paramètres qu'à l'intérieur de la procédure),

– *la direction de chaque paramètre*, IN, **OUT** ou **INOUT** (dans notre cas **INOUT** pour Q et Qb qui sont utilisés en lecture **IN** et écriture **OUT** au sein de la procédure),

– ***le type des paramètres*** (voir chapitre V concernant les types), **bit** dans notre cas.

Remarque :

Dans l'exemple ci-dessous, les paramètres de la procédure étaient de genre signal, ce qui impliquait les affectations "<=" propres aux signaux. Dans le cas où les paramètres sont de genre

variable, il faut utiliser les symboles ":=" propres aux variables. Si l'on reprend cet exemple, en utilisant des paramètres de genre variable, on obtient une description moins lisible qui est la suivante :

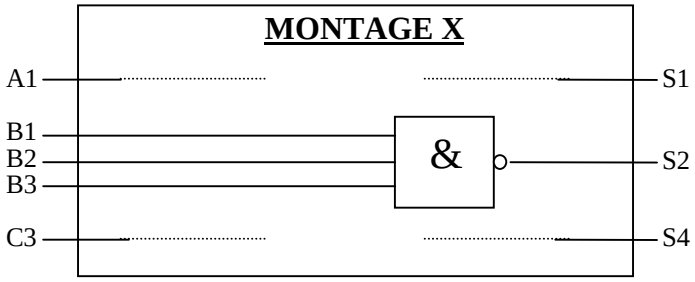
```
architecture test_rs of test_rs is

        PROCEDURE RS (signal A, B : IN bit;  variable Q, Qb : INOUT bit) IS
        BEGIN
        Q  :=  not (A  or  Qb);
        Qb  :=  not (B  or  Q);
        END;

begin
        process  (E1,  E2)
        variable  Q1,  Q2 :  bit;
        begin
        RS (E1,  E2,  Q1,  Q2);
                if (Q1 = '1') then S1 <= '1';
                else S1 <= '0'; end if;
                if (Q2 = '1') then S2 <= '1';
                else S2 <= '0'; end if;
        end process;
end test_rs;
```

Affectation de variable ":=" du type signal vers variable impossible en dehors de procédure ou fonction.

Les deux IF permettent de convertir les variables Q1 et Q2 en deux signaux S1 et S2.

B. Déclaration des fonctions et procédures au sein de packages

Dans les exemples qui précèdent, les fonctions ou procédures ont été insérées dans des architectures. Or, dans ce cas précis, les fonctions ou procédures en question ne sont accessibles que dans l'architecture dans laquelle elles ont été décrites. Pour les rendre accessibles à partir de plusieurs architectures, il est nécessaire de les déclarer au sein de packages et de déclarer en amont de l'architecture vouloir utiliser ce package. En reprenant l'exemple de la création d'une fonction NONET et en insérant cette fonction au sein d'un package, voici ce que l'on obtient :



Déclaration de la fonction NONET →

```
Package pack_NONET is
        function NONET (A, B, C : bit) return bit;
end pack_NONET;

Package body  pack__NONET  is
        function NONET (A, B, C : bit) return bit is
                variable result : bit;
        begin
                result := not (A and B and C);
                return result;
        end;
end pack_NONET;
```

Création de la fonction NONET →

Déclaration de la librairie et du membre utilisé →

```
Library user;
use user.pack_NONET.all;

Entity  montage_X  is
port (E1, E2, E3, ..... : IN bit; S1, ..... : OUT bit);
end montage_X;

Architecture  Arch  of  montage_X  is
begin
......
S1 <= NONET (E1, E2, E3);
.....
end Arch;
```

Utilisation de la fonction NONET →

Le nom de cette librairie correspond au nom donné au répertoire utilisateur où sont stocké les fichiers simulables issus de l'analyse des descriptions VHDL. Pour certains outils ce répertoire est par défaut appelé work. On aura alors les lignes suivantes:
Library work;
use work.pack_NONET.all;

Notons que la fonction NONET déclarée dans le package est décrite dans le package body. Cette procédure est obligatoire et valable aussi pour les procédure :

☞ **Toute fonction ou procédure déclarée au sein d'un package est décrite dans le package body associé à ce package.**

V. Les types prédéfinis ou non, surcharge des opérateurs, fonction de résolution

A. Les types prédéfinis et les opérateurs associés

Depuis le début de ce document, nous utilisons des signaux que l'on déclare en utilisant la syntaxe "S : out bit;". Cette déclaration signifie que le nom du signal est S, que sa direction est out (sortie) et qu'il est de type bit. Nous allons, au cours de ce paragraphe, nous intéresser tout particulièrement au type d'un signal.

Le type d'un signal définit l'ensemble auquel appartient un signal au même titre que l'on dit du nombre 3 qu'il appartient à l'ensemble des entiers. Le langage VHDL initial propose des types prédéfinis, c'est-à-dire des types pour lesquels l'ensemble est clairement défini. Voici les types prédéfinis proposés par le langage VHDL initial :

Nom du type	Définition de l'ensemble
BIT	Deux valeurs possibles '0' ou '1'
INTEGER	Entiers (nombre positif ou négatif sur 32 bits)
REAL	Réels
BOOLEAN	Deux valeurs possibles True ou False
CHARACTER	Caractères a, b, c ..., 1, 2 ...
TIME	Nombre réel de temps fs, ps ..., min, hr.

On peut donc déclarer un signal de la façon suivante :

S : out integer;

Dans ce cas précis, le signal S est de type entier et peut, par conséquent, prendre l'une des valeurs de l'ensemble des nombres entiers. On peut alors s'interroger sur l'existence d'opérateurs capables de gérer tous ces différents types. En effet, si l'opération A or B semble compréhensible lorsque les signaux A et B sont de types bit, elle devient stupide lorsque les signaux sont de types entiers, sans parler du cas ou les signaux sont de types différents.

En fait, le langage VHDL initial propose avec ses types prédéfinis des opérateurs dédiés capables de faire la somme, la soustraction d'entiers ou de réels, etc. Notons que, en l'absence de fonctions ou de procédures créées par l'utilisateur (voir chapitre IV), il n'existe pas d'opérateur prédéfini capable de manipuler des signaux de types différents.

☞ **Le langage VHDL initial propose des types prédéfinis ainsi que des opérateurs associés que l'utilisateur peut utiliser au sein des descriptions.**

On peut donc écrire :

A + B ➔ avec A et B de type entiers et "+" une fonction d'addition entre entiers.

A + B ➔ avec A et B de type réels et "+" une fonction d'addition entre réels.

Dans ces deux exemples, les fonctions "+" d'addition ont le même nom mais ne sont pas identiques, car elles concernent des types distincts.

En plus des différents types listés dans le tableau précédent, le langage VHDL propose trois types très utiles dont voici la définition.

Type énumération

Ces types sont utilisés lorsque l'on souhaite créer un nouveau type de signal (voir paragraphe C ci-dessous) ou lorsque l'on souhaite créer un signal ou une variable dont les valeurs possibles sont explicites. Prenons l'exemple d'un signal dont les valeurs sont significatives des instructions lues dans une ROM programme. Pour clarifier la description, il est possible de définir un type (voir paragraphe suivant concernant la définition d'un type) de signal dont les valeurs possibles portent les noms des instructions.

```
TYPE Signal_instruction IS (MOVE, ADD, JNE, .... , BRS, RTS) ;
SIGNAL Instruction : Signal_instruction ;
```

Dans cet exemple, la première ligne correspond à la définition d'un type de nom Signal_instruction et la deuxième à la création d'un signal nommé Instruction et de type Signal_instruction.

Type tableau

Le langage VHDL autorise la définition de tableau de valeurs de type quelconque. L'instruction suivante correspond à la définition d'un type nommé Tableau_d'entiers :

```
TYPE Tableau_d'entiers IS ARRAY (7 downto 0) OF integer ;
```

L'instruction suivante va créer un signal de nom Tableau et de type Tableau_d'entiers :

```
SIGNAL Tableau : Tableau_d'entiers ;
```

Le signal Tableau comportera donc 8 éléments de type entier que l'on pourra manipuler individuellement ou tous en même temps (voir chapitre III, paragraphe B.3). Pour créer un tableau à plusieurs lignes, la syntaxe suivante est possible :

```
TYPE Tableau_8xbit IS ARRAY (7 downto 0) OF bit ;
TYPE Tableau_4x8xbit IS ARRAY (3 downto 0) OF Tableau_8xbit ;
```

ou encore :

```
TYPE Tableau_4x8xbit IS ARRAY (3 downto 0, 7 downto 0) OF bit ;
```

Type enregistrement

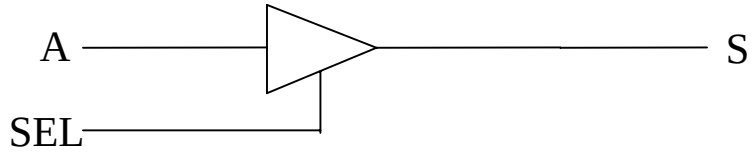
Il est possible, en VHDL comme dans la plupart des langages évolués, de créer un type constitué de plusieurs éléments différents. Les éléments ainsi juxtaposés pourront, par la suite, être accessibles grâce à l'utilisation du "." comme pour les adresses électroniques.

```
TYPE Bus_micro IS RECORD
        Nbr_bit_adresse : integer ;
        Nbr_bit_donnee  : integer ;
        Vitesse : time ;
END RECORD ;
```

```
Bus_micro.Nbr_bit_adresse <= 24 ;
Bus_micro.Nbr_bit_donnee <= 16 ;
Bus_micro.vitesse <= 10us ;
```

B. Définition de types et surcharge des opérateurs

Les types prédéfinis proposés par le langage VHDL initial sont vite devenus insuffisants dans la plupart des cas. Il suffit, pour s'en convaincre, d'essayer de décrire à l'aide de signaux de type bit la porte trois états ci-dessous :



Avec un signal S de type bit pouvant prendre deux valeurs '0' et '1', la description de cette structure est impossible. Il faut utiliser un type qui permette au signal S de prendre la valeur 'Z' haute impédance.

En fait, on pourrait citer des exemples plus complexes et montrer qu'il est nécessaire de créer des types qui puissent proposer un grand nombre de valeurs. L'objectif final étant de pouvoir décrire avec le plus de vraisemblance toutes les structures logiques possibles. C'est dans cette optique que les types **std_logic** et **std_ulogic** conformes au standard IEEE 1164 ont été créés. Ces types proposent les valeurs suivantes :

'**U**' = Non initialisé,
'X ' = inconnu forçage fort,
'0' = forçage fort,
'1' = forçage fort,
'Z ' = haute impédance,
'W' = inconnu forçage faible,
'L' = forçage faible,
'H' = forçage faible,
et **'-'** = quelconque.

La notion de forçage correspondant à la prise en compte d'éléments de rappel à la masse ou à l'alimentation (résistance de pull-up ou pull-down par exemple).

Ainsi, on va désormais pouvoir déclarer des signaux en spécifiant type = std_logic ou std_ulogic, on obtiendra par exemple :

E1 : std_logic; ou **E2 : std_ulogic**;

Mais on peut alors se poser la question suivante : que va-t-il se passer si l'on écrit "**E1 and E2**" ? L'opérateur prédéfini **and** que l'on utilisait dans les exemples précédents avec des signaux de type bit est-il capable de résoudre l'opération E1 and E2 ?
La réponse est non, il va falloir créer de nouveaux opérateurs and, or, xor, etc. adaptés aux types std_logic et std_ulogic. Mais ces nouveaux opérateurs porteront les mêmes noms que les

opérateurs prédéfinis du langage VHDL initial. Heureusement, le langage VHDL prévoit ce cas de figure et se comporte de la façon suivante :

☞ **L'analyseur (ou le compilateur) VHDL va utiliser l'opérateur qui supporte les types utilisés dans l'expression.**

Par exemple, devant l'expression "**E1 or E2**" avec **E1** et **E2** de type **bit** l'analyseur VHDL va utiliser l'opérateur prédéfini **or** alors que devant l'expression "**E1 or E2**" avec **E1** et **E2** de type **std_logic** l'analyseur VHDL va utiliser l'opérateur **or** prévu pour le type **std_logic**.

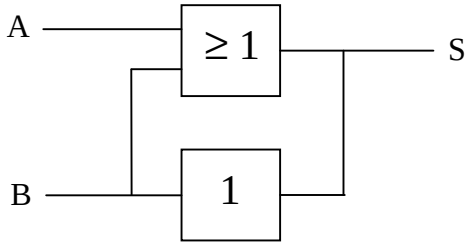
Tout ceci sous-entend qu'il est possible de créer des types et des opérateurs associés à ces types. Voici l'exemple d'une création de type et d'opérateur associés :

```
package pack_test is
type my_logic is ('X', 'Z', '0', '1');
function "AND" (v1, v2: my_logic) return my_logic;
end pack_test;

package body pack_test is
type my_table is array (my_logic'left to my_logic'right,
        my_logic'left to my_logic'right) of my_logic;
function "AND"          (v1, v2: my_logic)   return my_logic is
    constant and_t: my_table := -- 'X' 'Z' '0' '1'
                                (('X','X','0','X'), -- 'X'
                                 ('X','X','0','X'), -- 'Z'
                                 ('0','0','0','0'), -- '0'
                                 ('X','X','0','1'));-- '1'
begin
    return and_t (v1, v2);
end;
end pack_test;
library user;
use user.pack_test.all;
entity test_my_logic is
    port (A, B : in my_logic;
    S : out my_logic);
end test_my_logic;

architecture arch_test of test_my_logic is
begin
S <= A AND B;
end arch_test;
```

L'instruction "**TYPE** nom **IS** (valeurs possibles)" permet de définir un type. On constate donc, dans cet exemple, qu'un type nommé my_logic a été créé. Ce type propose quatre valeurs possibles : "X, Z, 0 et 1". Notons que la déclaration de ce type a été faite au sein d'un package pour pouvoir la rendre accessible par plusieurs entités. En effet, il suffit de rajouter en amont de l'entité les lignes "library user;" et "use user.pack_test.all;" pour que l'analyseur comprenne que vous souhaitez utiliser des éléments compris dans le package spécifié.
L'instruction "**fonction "AND" (v1, v2: my_logic) return my_logic;**" permet de créer un nouvel opérateur **and** que l'analyseur utilisera lorsqu'il rencontrera une instruction "**E1 and E2**" avec **E1** et **E2** du type **my_logic**.
☞ **Dans un tel cas de figure, on parle de surcharge d'opérateur. Dans cet exemple, l'opérateur *and* est surchargé. Le nom de la fonction est compris entre guillemets lorsqu'il s'agit d'une surcharge.**

C. Fonction de résolution d'un type

Dans l'exemple ci-dessous, on constate que le signal S possède plusieurs sources. La description VHDL d'une telle structure ne pose pas de difficultés particulières.



Or, pour qu'une telle description puisse être simulée il faut que le simulateur sache résoudre les différents cas possibles d'état du signal S. On peut par exemple se demander quelle est la valeur de S si la sortie de l'une des portes logiques est à '0' alors que l'autre est à '1'.

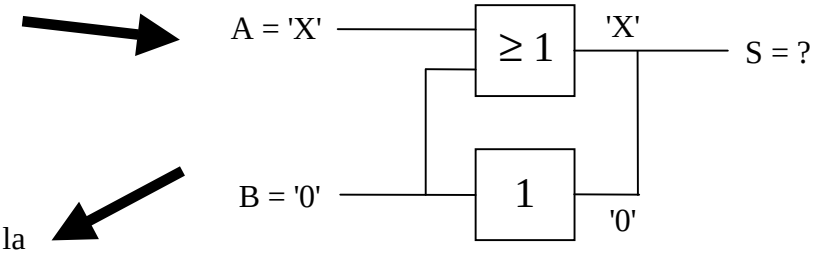
☞ **Pour permettre au simulateur de résoudre une affectation multiple d'un signal, le langage VHDL permet à un type d'être résolu, c'est-à-dire d'énumérer au sein de la description tous les cas possibles d'associations et les résultats qui en découlent.**
En reprenant l'exemple du paragraphe précédent on obtient :

```
package pack_test is
type   my_logic   is ('X', 'Z', '0', '1');
type my_logic_vector is array (integer range<>) of my_logic;
function res (inputs: my_logic_vector) return my_logic;
```

```
subtype  my_rlogic   is  res  my_logic;
function "and"  (v1, v2: my_logic)     return my_logic;
end pack_test;

package body pack_test is
type my_table is array (my_logic'left to my_logic'right,
my_logic'left to my_logic'right) of my_logic;
---------------------------------------------------------------
function res (inputs: my_logic_vector) return my_logic is
constant merge: my_table := -- 'X' 'Z' '0' '1'
(('X','X','X','X'), -- 'X'
('X','Z','0','1'), -- 'Z'
('X','0','0','X'), -- '0'
('X','1','X','1'));-- '1'
variable result : my_logic := 'Z';
begin
for i in inputs'range loop
result := merge (result, inputs (i));
end loop;
return result;
end;
---------------------------------------------------------------
function "and" (v1, v2: my_logic) return my_logic is
constant and_t: my_table := -- 'X' 'Z' '0' '1'
(('X','X','0','X'), -- 'X'
('X','X','0','X'), -- 'Z'
('0','0','0','0'), -- '0'
('X','X','0','1'));-- '1'
begin
return and_t (v1, v2);
end;
end pack_test;
```

Dans cet exemple, la fonction "**res**" permet de donner au simulateur le résultat d'une multiaffectation d'un signal. Concrètement, dans l'exemple de la structure ci-dessous, la fonction "**res**" va utiliser le tableau "**merge**" à deux reprises pour déterminer la valeur du signal S.

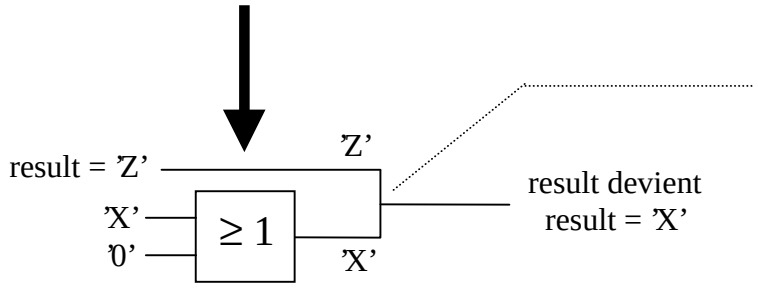
```
library user;
use user.pack_test.all;
entity test_my_logic is
        port (A, B : in my_rlogic;
        S : out my_rlogic);
end test_my_logic;

architecture arch_test of test_my_logic is
begin
S <= A OR B;
S <= B;
end arch_test;
```



Variable "result" de la fonction res initialisée à "Z". Cette variable représente le résultat de la fonction.

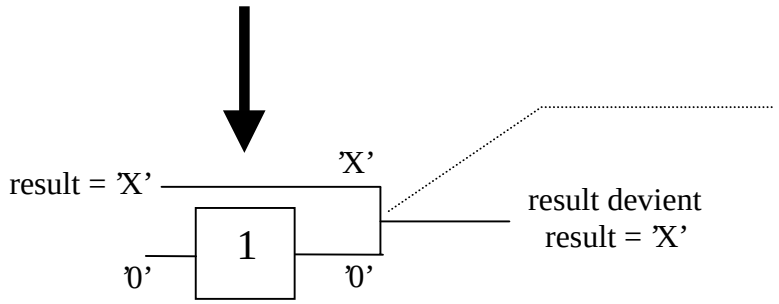
Utilisation du tableau merge

```
   X   Z   0   1
(('X','X','X','X'), --X
('X','Z','0','1'),    --Z
('X','0','0','X'),    --0
('X','1','X','1'));   --1
```



Utilisation du tableau merge

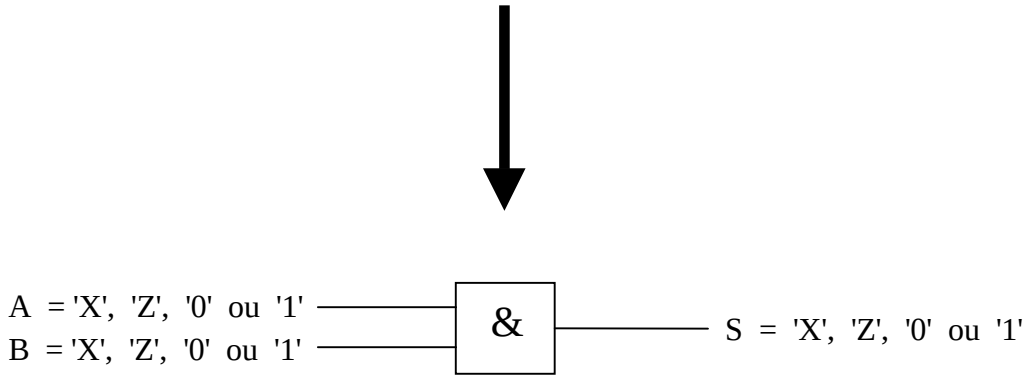
```
   X   Z   0   1
(('X','X','X','X'), --X
('X','Z','0','1'),    --Z
('X','0','0','X'),    --0
('X','1','X','1'));   --1
```

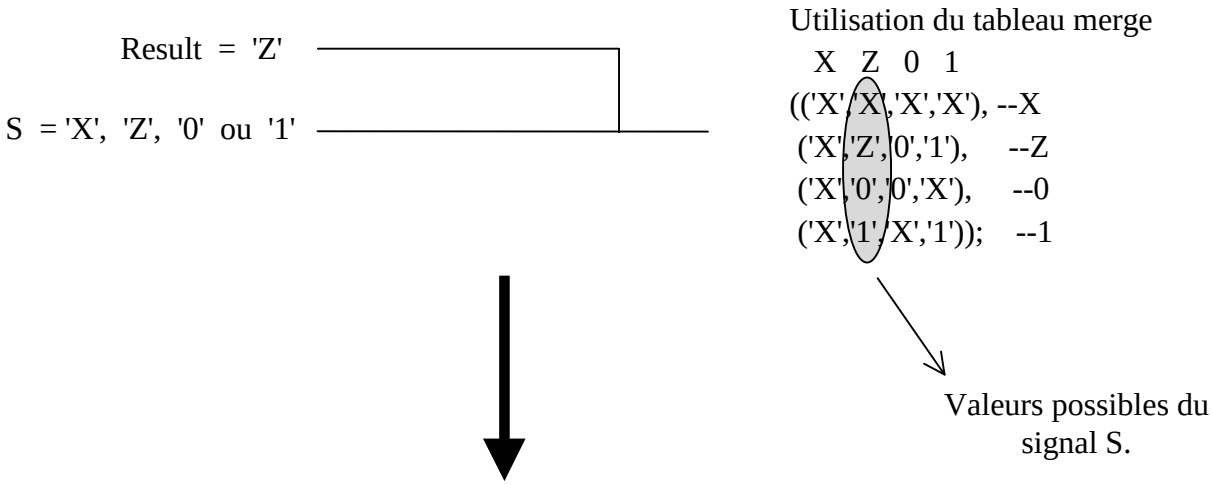


result = 'X' ➔ S <= 'X'

Tout se passe comme si l'affectation multiple du signal S était décomposée en deux affectations. La première de ces décompositions n'ayant pour but que de prévoir le cas d'une affectation unique. En effet, dans le cas d'une affectation unique, on obtient la situation suivante :

```
Architecture Arch of Ent is
begin
  S <= A and B;
end Arch;
```






```
S <= result
```

D. Le package "IEEE standard logic 1164"

Lors de l'évolution du langage VHDL vers le standard IEEE 1164, un package a été ajouté au langage VHDL initial. Ce package est constitué d'un ensemble de fonctions et de déclarations de types que l'on peut utiliser et appeler au sein de descriptions. Il propose, entre autres, des fonctions de conversion de types, des fonctions de surcharge d'opérateurs, la fonction de résolution du type std_logic, etc. Les types std_logic, std_ulogic, std_logic_vector et std_ulogic_vector utilisés à ce jour par les concepteurs sont créés dans ce package standard IEEE nommé "std_logic_1164". Pour utiliser ces types, il faut donc déclarer la librairie "library IEEE;" et le package "use IEEE.std_logic_1164.all;" en amont de l'entité de la description VHDL.

En étudiant ce package IEEE et son package body, on constate que la lettre "u", qui différencie les noms de types std_logic et std_logic_vector des noms de types std_ulogic et std_ulogic_vector, signifie "**unresolved**" pour rappeler que ces types ne sont pas résolus. Il est donc impossible d'avoir, par exemple, un signal de type std_ulogic avec une affectation multiple (plusieurs sources). En fait, le type std_logic est un sous-type résolu du type std_ulogic comme le type my_rlogic est un sous-type résolu du type my_logic dans l'exemple du paragraphe précédent.
En ce qui concerne les fonctions proposées au sein du package IEEE standard logic 1164, on retrouve, comme cela a été mentionné ci-dessus :

– toutes les fonctions de surchages and, nand, or, nor, xor, et not adaptées au types std_logic, std_ulogic, std_logic_vector et std_ulogic_vector,

– un ensemble de fonctions de conversion de types :

bit ou bit_vector➔ std_logic, std_ulogic, std_logic_vector et std_ulogic_vecto,
std_logic, std_ulogic, std_logic_vector et std_ulogic_vecto ➔ logique 3 états X, 0 et 1,
logique 3 états X, 0 et 1 ➔ std_logic, std_ulogic, std_logic_vector et std_ulogic_vecto,
std_logic, std_ulogic, std_logic_vector et std_ulogic_vecto ➔ logique 4 états X, 0, 1 et Z,
logique 4 états X, 0, 1 et Z ➔ std_logic, std_ulogic, std_logic_vector et std_ulogic_vecto,
std_logic, std_ulogic, std_logic_vector et std_ulogic_vecto ➔ logique 4 états U, X, 0 et 1,
logique 4 états U, X, 0 et 1 ➔ std_logic, std_ulogic, std_logic_vector et std_ulogic_vecto,
détection de fronts montants sur un signal ou un vecteur de type std_logic ou std_ulogic,

En réalité, la plupart des outils de synthèses logiques ou de simulations logiques proposent des packages d'aide à la conception dans lesquels se trouvent des fonctions de conversion ou tout simplement des fonctions ou procédures du type utilitaire (additionneur, soustracteur, incrémenteur...).

VI. Les attributs

A. Présentation des attributs, leurs rôles

Le langage VHDL propose un outil appelé attribut que l'on peut, en quelque sorte, assimiler à des fonctions. Placé auprès d'un signal, l'attribut " 'event " va, par exemple, retourner une information vraie ou fausse (donc de type booléen) sur le fait que le signal en question ait subi ou non un événement (un changement de valeur). Le fonctionnement de cet attribut ressemble au comportement d'une fonction dont le paramètre de retour serait de type booléen, à ceci près que l'attribut 'event en question possède une notion de mémoire. En effet il détecte un changement d'état qui nécessite d'avoir au préalable mémorisé l'état précédent. Il existe en réalité deux sortes d'attributs.

❶ Les attributs destinés à retourner des informations concernant le comportement d'un signal (événements, dernière valeur...).

❷ Les attributs destinés à retourner des informations concernant les caractéristiques d'un vecteur (longueur, dimension...).

Pour ces deux sortes d'attributs, on parle d'attributs prédéfinis, au même titre que l'on parle de types prédéfinis (bit, boolean, etc.). Voici une liste non exhaustive des différents attributs prédéfinis proposés par le langage VHDL.

Attributs	Définitions - informations de retour
'high	Placé près d'un nom de tableau, il retourne la valeur du rang le plus haut (la valeur de retour est de type entier).
'low	Placé près d'un nom de tableau, il retourne la valeur du rang le plus bas (la valeur de retour est de type entier).
'left	Placé près d'un nom de vecteur ou tableau, il retourne la valeur du rang le plus à gauche (la valeur de retour est de type entier).
'right	Placé près d'un nom de vecteur ou tableau, il retourne la valeur du rang le plus à droite (la valeur de retour est de type entier).
'range	Placé près d'un signal, il retourne la valeur de l'intervalle spécifié par l'instruction range lors de la déclaration du signal ou du type utilisé.
'reverse_range	Placé près d'un signal, il retourne la valeur de l'intervalle inverse spécifié par l'instruction range lors de la déclaration du signal ou du type utilisé.
'length	Retourne X'high - X'low + 1 (sous la forme d'un entier).
'event	Vrai ou faux si le signal auquel il est associé vient de subir un changement de valeur.

Attributs	Définitions - informations de retour
'active	Vrai ou faux si le signal auquel il est associé vient d'être affecté.
'last_value	Retourne la valeur précédente du signal auquel il est associé.
'last_event ou 'last-active	Retournent des informations de temps concernant la date d'affectation ou de transition des signaux auxquels ils sont affectés
'stable(T)	Vrai ou faux si le signal auquel il est associé n'est pas modifié pendant la durée T.
'quiet	Vrai ou faux si le signal auquel il est associé n'est pas affecté pendant la durée T.
'transaction	Retourne une information de type bit qui change d'état lorsque le signal auquel il est associé est affecté.

Parmi tous ces attributs prédéfinis, les plus couramment utilisés sont certainement les attributs relatifs aux caractéristiques d'un vecteur. Prenons l'exemple d'une fonction nommée FONC que l'on souhaite pouvoir appeler à partir de plusieurs instructions d'une même architecture et qui possède un paramètre de type std_logic_vecteur, la dimension est variable suivant l'instruction appelant. Pour que la dimension du paramètre vecteur s'adapte à l'appel, il faut utiliser les attributs 'left et 'right.

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

entity ex_attrib is
  port (vecteur1 : in std_logic_vector(0 to 7);
        vecteur2 : in std_logic_vector(0 to 15);
        Svecteur1 : out std_logic_vector(0 to 7);
        Svecteur2 : out std_logic_vector(0 to 15));
end ex_attrib;

Architecture ex_attrib of ex_attrib is
function FONC (vect_A : std_logic_vector) return
        std_logic_vector is
variable vect_result : std_logic_vector (vect_A'left
        to vect_A'right);
begin
    for I in vect_A'left to vect_A'right loop
        if (vect_A(i) = '1') then vect_result(i) := '1';
        else vect_result(i) := '0';
        end if;
        end loop;
      return vect_result;
end;

begin
Svecteur1 <= FONC (vecteur1);
Svecteur2 <= FONC (vecteur2);
end ex_attrib;
```

Dans cet exemple, l'utilisation d'attributs lors de la déclaration de la variable vect_result permet d'adapter les dimensions de ce vecteur à 8 ou 16 bits suivant qu'il s'agit de vecteur1 ou de

vecteur2. Ainsi, la fonction s'adapte aux dimensions du vecteur paramètre. Cette méthode est très employée au sein des packages pour généraliser un paramètre vecteur de fonction en un vecteur de dimension quelconque.

B. Définition des attributs

Au même titre qu'il est possible de définir des types, il est également possible de définir des attributs. La différence réside dans le fait que le simulateur va ignorer ces nouveaux attributs et ne reconnaître que les attributs de type prédéfinis. La définition d'un nouvel attribut ne sert qu'à ajouter des informations à une description. Ces informations seront par la suite utilisées par d'autres outils. Les outils de synthèses logiques se servent souvent de cette procédure de passage d'informations et profitent de cette opportunité pour introduire des paramètres de synthèse et spécifier, entre autres, le type de boîtier ou le type de composant à utiliser pour la synthèse d'une description.

Voici l'exemple d'une création d'attribut destiné à préciser le type de boîtier de deux composants.

```
type IC_PACKAGE is (DIL, PLCC, PGA);
attribute PTYPE : IC_PACKAGE;
attribute PTYPE of U1 : component is PLCC;
attribute PTYPE of U2 : component is DIL;
```

Création d'un type nommé IC_PACKAGE pouvant prendre les valeurs DIL, PLCC et PGA.

Création d'un attribut de type IC_PACKAGE.

VII. Synthèse d'une description VHDL

La synthèse d'une description VHDL représente une étape essentielle dans le processus de conception d'un composant programmable ou d'un ASIC. Le résultat de cette synthèse va déterminer l'ampleur ou, tout au moins, les caractéristiques du composant cible. Il est donc important et même crucial, dans la plupart des cas, de savoir orienter une synthèse de façon à en optimiser le résultat.

Mais optimiser ne signifie pas forcement diminuer le nombre de portes logiques équivalentes. On peut, par exemple, choisir de diminuer au maximum le nombre de bascules D au détriment du nombre de portes logiques, ou encore privilégier certaines structures plutôt que d'autres, sachant que ces structures optimisent l'intégration dans le composant cible. Ceci étant, il est possible de créer deux descriptions équivalentes à la simulation dont la synthèse ne donnera pas le même résultat. Il y a donc tout un savoir-faire à acquérir pour rédiger une description VHDL de façon à orienter le résultat de la phase de synthèse.

Dans les deux paragraphes qui suivent, vous trouverez quelques éléments de réflexion concernant la synthèse de fonctions simples. En appliquant ces notions à des descriptions plus complexes, vous parviendrez certainement à optimiser votre description suivant vos critères.

A. Fonctions combinatoires

Une fonction combinatoire est une structure pour laquelle chaque combinaison d'entrée fait correspondre en sortie une combinaison déterminée et indépendante des combinaisons d'entrées et de sorties précédentes. On peut donc décrire ces structures à l'aide d'équations booléennes qui seront toujours vérifiées comme le sont les instructions concurrentes. En effet, l'instruction "**S <= A and B or C**" est toujours vérifiée et correspond à l'équation booléenne S = A.B + C. Mais il est aussi possible de décrire une structure combinatoire en utilisant les instructions séquentielles d'un process. L'équation précédente peut, par exemple, s'écrire :

```
process (A, B, C)
begin
if (C = '1') then S <= '1';
elsif (A = '1' and B = '1') then S <= '1';
else S <= '0';
end if;
end process;
```

```
S <= A and B or C
```

L'issue de synthèse de ces deux types de description est identique, fort heureusement. Toutefois, on peut être amené à se poser la question de savoir quel type de description est à utiliser. En

réalité, même si le résultat de la synthèse est identique, il y a une différence fondamentale entre ces deux descriptions. L'une n'est que le reflet des équations logiques de la structure combinatoire à décrire, alors que l'autre se base sur l'expression de son comportement.
Pour tenter de résumer les différences existant entre ces deux méthodes de description des fonctions combinatoires, on peut lister les avantages et les inconvénients de chacune d'elles.

➔ Description de fonctions combinatoires à l'aide d'instructions concurrentes :

Avantages :

– la description obtenue est lisible et simple,
– la description obtenue reflète bien la réalité.

Inconvénients :

– décrire une fonction combinatoire en utilisant des instructions concurrentes sous-entend que la taille de sa table de vérité le permet ou que les simplifications de cette dernière ont été faites.

➔ Description de fonctions combinatoires à l'aide de process :

Avantages :

– pour des fonctions compliquées dans lesquelles les équations possèdent de nombreux termes, cette méthode est parfois plus simple. Lorsque la description comportementale est plus simple que la description algorithmique, cette méthode est avantageuse ;
– il n'est pas nécessaire de faire les simplifications de la table de vérité, elles sont faites automatiquement par le synthétiseur lors de la synthèse.

Inconvénients :

– dans certains cas, cette écriture est peu lisible et par conséquent très difficilement modifiable par une personne extérieure.

Il n'y a donc pas de règle générale mais plutôt la nécessité de procéder à une démarche logique d'observation avant la description pour choisir la méthode de description adéquate.

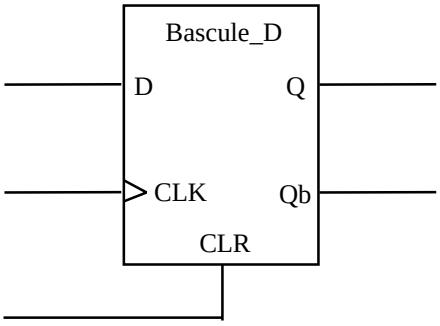
B. Fonctions séquentielles

La synthèse d'une fonction logique séquentielle est beaucoup plus complexe que celle d'une fonction combinatoire. Les possibilités de description offertes par le langage VHDL sont vastes et bien souvent irréalisables dans l'état actuel des technologies. Prenons l'exemple des attributs, ils sont puissants, efficaces, mais posent, pour certains, de gros problèmes aux outils de synthèse qui ne savent pas comment réaliser des structures capables de représenter le fonctionnement de ces attributs. L'attribut 'quiet(T), qui retourne une information de type booléen qui est vraie lorsque le signal auquel il est associé n'a pas été affecté depuis le temps T, ne

représente aucune structure électronique simple. Les synthétiseurs actuels refusent donc cet attribut faute de savoir quoi en faire.

Par contre, il existe des attributs qui sont très utilisés et qui représentent même un véritable standard. La description d'un front est très simple à réaliser si l'on utilise l'attribut 'event associé à une condition de niveau pour la détection d'un front montant ou descendant. L'exemple ci-dessous qui correspond à la description d'une bascule D ne posera donc aucun problème de synthèse.

```
Library ieee;
use ieee.std_logic_1164.all;
entity bascule_D is
port (D, clk, clr : in std_logic; Q, Qb : out std_logic);
end bascule_D;

architecture bascule_D of bascule_D is
BEGIN
        process (clk, clr)
        begin
        if (clr = '1') then                 Entrée CLR active à '1'
           Q <= '0';
           Qb <= '1';
        elsif (clk'event and clk = '1') then
           Q <= D;
           Qb <= D;                 Détection d'un front montant
        end if;
        end process;
end bascule_D;
```

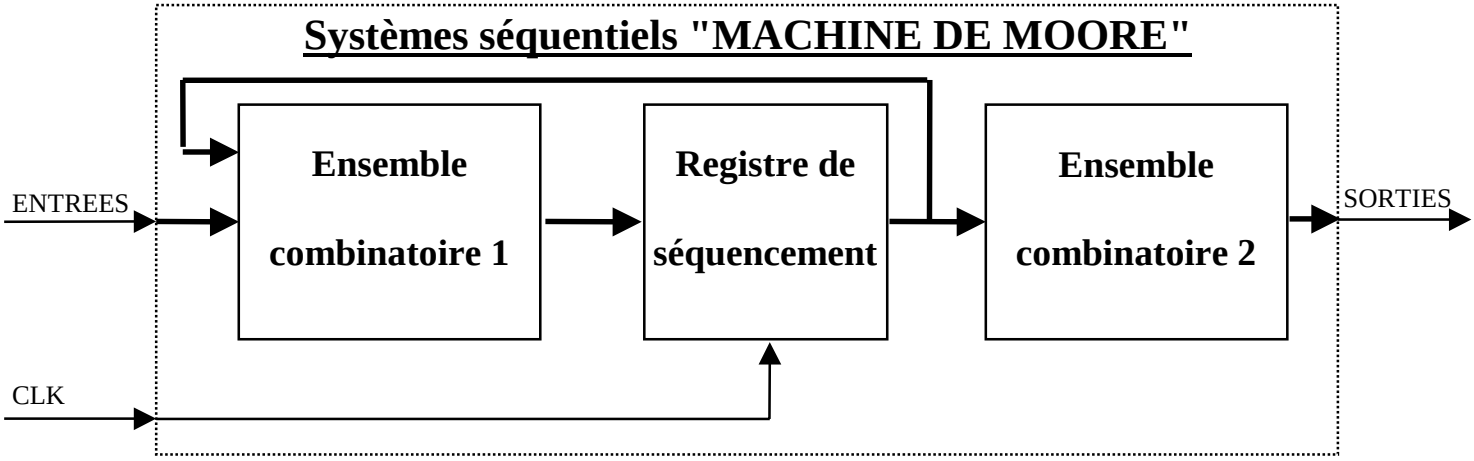


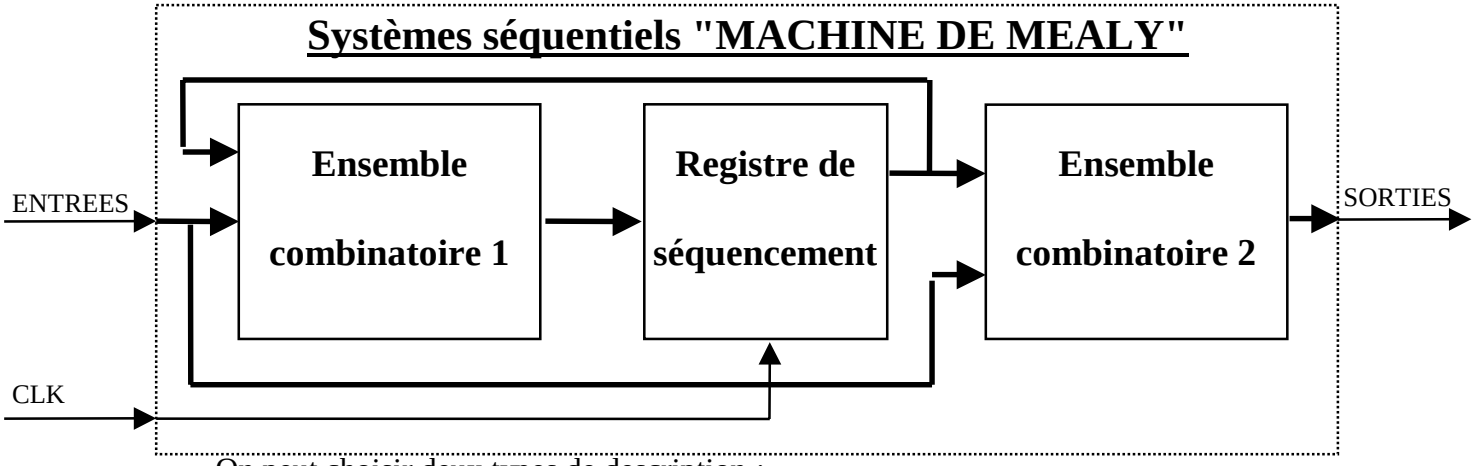
Dans le process précédent, on aurait tout aussi bien pu utiliser l'instruction "wait" qui permet de figer l'exécution d'un process jusqu'à l'arrivée d'une condition ou d'un événement sur un signal, il aurait alors fallu écrire :

```
process
begin
  wait until CLK'event and CLK='1';
  Q <= D;
end process;
```

Dans cette instruction, 'event est redondant avec l'instruction wait, mais certains outils de synthèse impose cette syntaxe. Quoi qu'il en soit, cette instruction déclenche l'exécution du process à chaque front montant du signal clk.

Ainsi, il existe certaines descriptions typiques que le synthétiseur va reconnaître et traduire directement, comme l'attribut 'event de l'exemple précédent. Ceci étant, on peut se poser la question suivante : comment doit-on rédiger la description VHDL d'une fonction séquentielle pour que la synthèse soit optimale et utilise le moins de composants possible ?
Or, là aussi, il n'existe pas de règle de rédaction miracle qui permette de garantir une synthèse optimale. Toutefois, on sait que toute fonction logique séquentielle (synchrone) peut être réalisée grâce à une structure du type machines de MOORE ou de MEALY (appelées aussi séquenceur ou machine à états). La synthèse d'une description de fonctions logiques séquentielles donnera donc une de ces structures (ou une extension de ces structures).





On peut choisir deux types de description :

❶ comportementale, se limitant à décrire le comportement de la structure à l'aide d'instructions IF, THEN, ELSE, CASE, WHEN...,

❷ structurelle ou semi-structurelle, décrivant chacune des parties constituant les machines de MOORE ou de MEALY.

➔ Dans le premier cas, la description est indépendante du choix de la structure finale. En d'autres termes, la description n'impose pas une structure cible, le synthétiseur est donc libre de la choisir

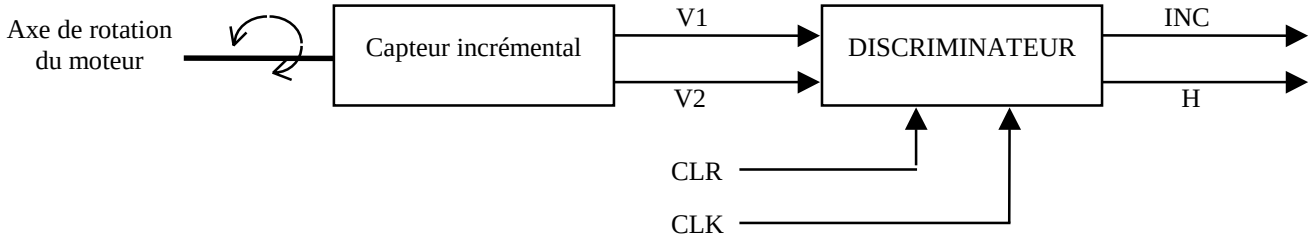
avec ses propres critères. Mais le résultat de ce choix ne correspond pas forcément à une synthèse optimale de la description. Pour des cas simples, ce type de description est souvent suffisant.

➔ Dans le deuxième cas, le choix de la structure cible est implicite ou partiel. Tout ce passe comme si on imposait une structure cible au synthétiseur. Dans ces conditions, le résultat de la synthèse est conforme à la description, et de ce fait optimal si le choix de la structure cible l'est. Notons qu'il est souvent intéressant d'utiliser ce genre de description lorsque l'on souhaite imposer des critères de synthèse du type : utiliser un minimum de bascules, privilégier la rapidité du montage, garantir certaines contraintes de temps, etc.

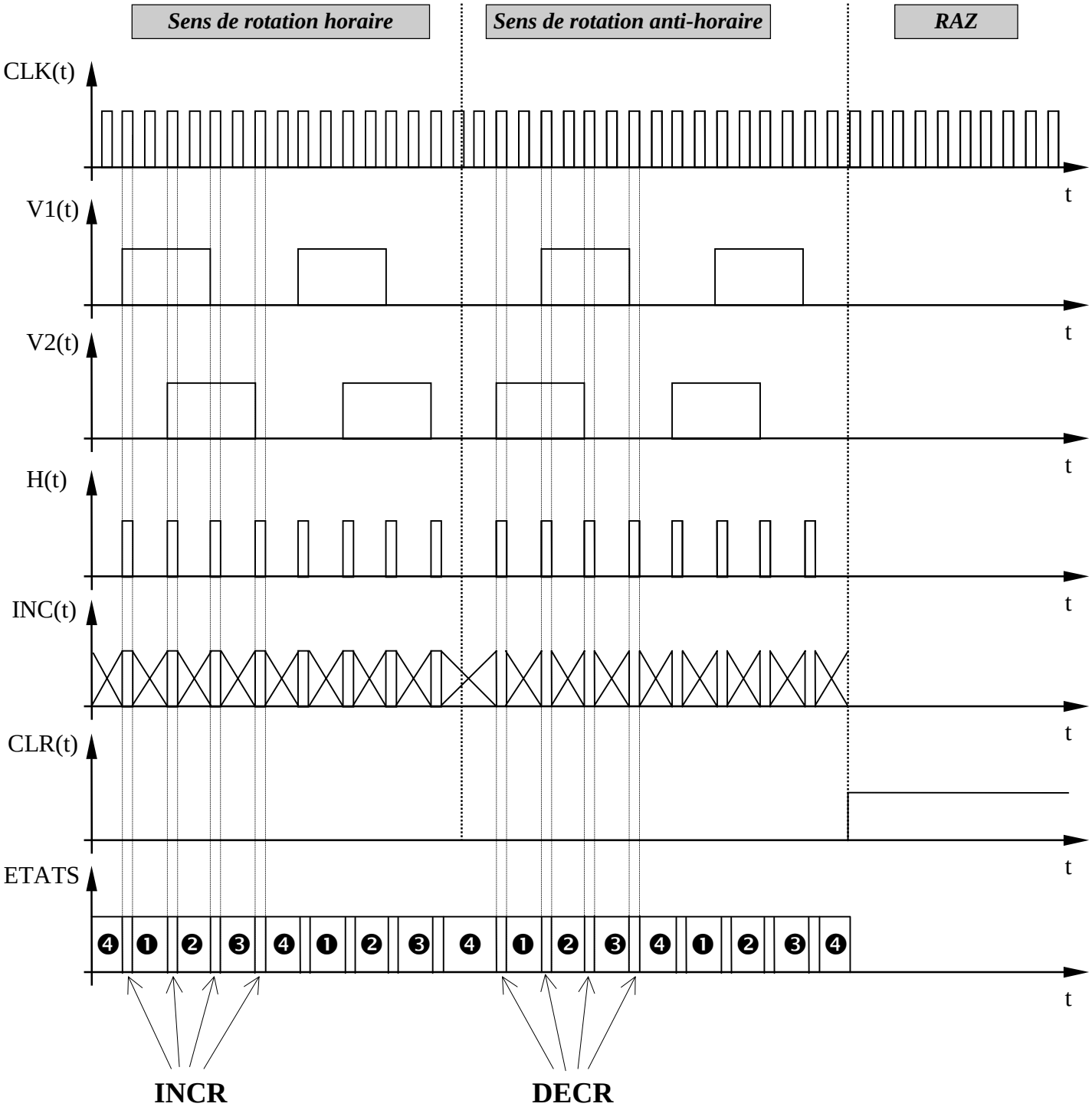
C. Synthèse d'un diagramme d'état

Il existe plusieurs formalismes capables de décrire l'évolution et le comportement d'un système séquentiel. Parmi eux se trouve le diagramme d'états, qui présente l'avantage d'être directement traduisible en langage VHDL.

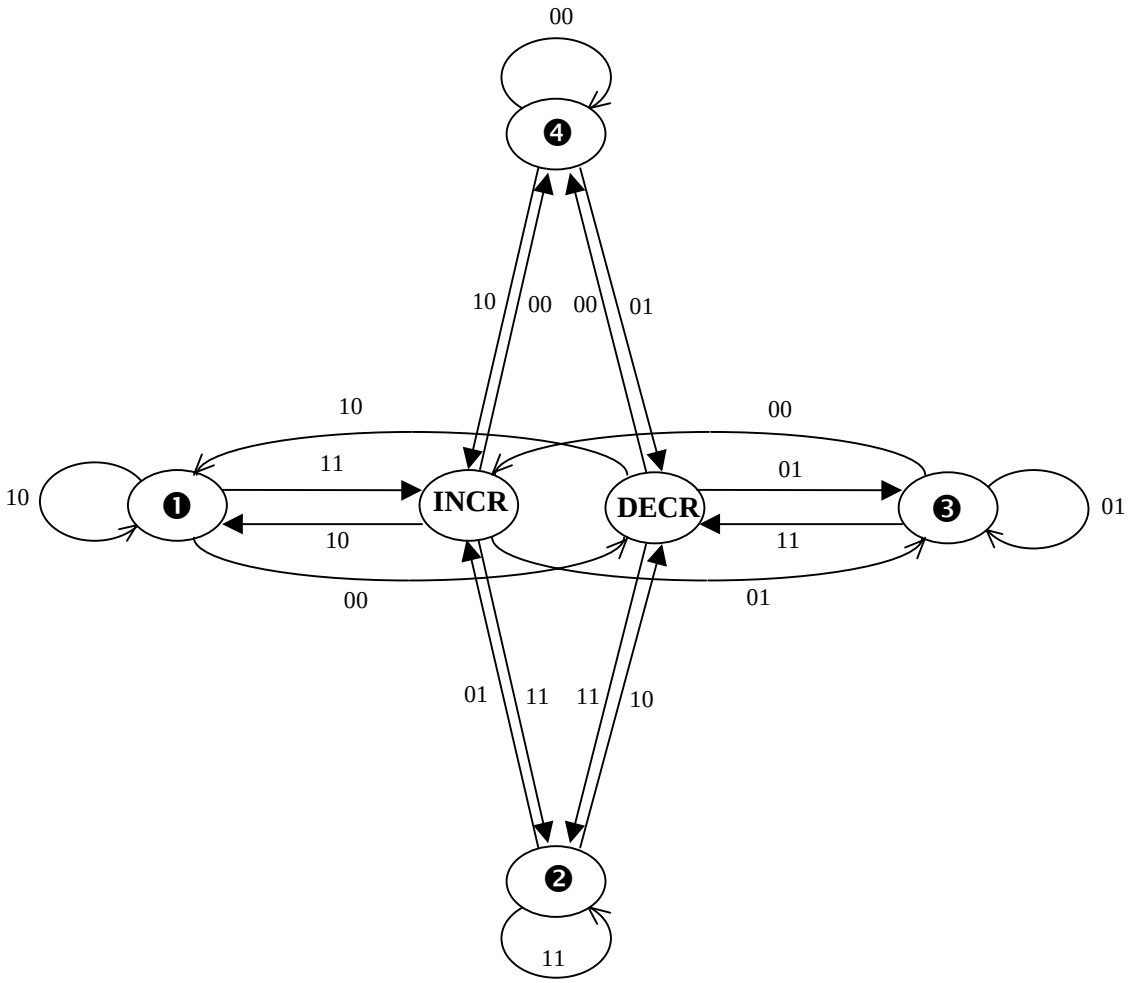
Citons l'exemple de la structure ci-dessous dont le rôle est de transformer deux signaux issus d'un capteur incrémental reliés à un moteur, en deux nouveaux signaux INC et H permettant le comptage et la détection de sens du moteur.



Les chronogrammes correspondant à cette structure sont les suivants :



Le diagramme d'état correspondant à ce système possède donc huit états nommés sur le chronogramme ❶, ❷, ❸, ❹, **INCR** et **DECR** et se représente ainsi :



La description correspondant à ce diagramme d'états est la suivante :

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

ENTITY fs31_comp IS PORT (V1, V2, clk, clr : in std_logic; INC, H : OUT std_logic);
END fs31_comp;

ARCHITECTURE fs31_comp OF fs31_comp IS
type etats is (etat1, etat2, etat3, etat4, incr, decr);
signal etat : etats;
```

```
BEGIN
    process (clk)
    begin
    if (clr='1') then etat <= etat1;
    elsif (clk'event and clk='1') then
        case etat is
        when etat1 =>
            if (V1='1' and V2='0') then etat <= incr;
            elsif (V1='0' and V2='1') then etat <= decr;
            else etat <= etat1;
            end if;
        when etat2 =>
            if (V1='1' and V2='1') then etat <= incr;
            elsif (V1='0' and V2='0') then etat <= decr;
            else etat <= etat2;
            end if;
        when etat3 =>
            if (V1='0' and V2='1') then etat <= incr;
            elsif (V1='1' and V2='0') then etat <= decr;
            else etat <= etat3;
            end if;
        when etat4 =>
            if (V1='0' and V2='0') then etat <= incr;
            elsif (V1='1' and V2='1') then etat <= decr;
            else etat <= etat4;
            end if;
        when incr =>
            if (V1='0' and V2='0') then etat <= etat1;
            elsif (V1='1' and V2='0') then etat <= etat2;
            elsif (V1='1' and V2='1') then etat <= etat3;
            else etat <= etat4;
            end if;
        when decr =>
            if (V1='0' and V2='0') then etat <= etat1;
            elsif (V1='1' and V2='0') then etat <= etat2;
            elsif (V1='1' and V2='1') then etat <= etat3;
            else etat <= etat4;
            end if;
```

```
        end case;
    end if;
    end process;

H <= '0' WHEN ((ETAT = ETAT1) or (ETAT = ETAT2) or (ETAT = ETAT3) or (ETAT = ETAT4))  ELSE  '1';
INC <= '1' WHEN ((ETAT = ETAT1) or (ETAT = ETAT2) or (ETAT = ETAT3) or (ETAT = ETAT4))  ELSE  '0' WHEN (ETAT = DECR) ELSE '1';


END fs31_comp;
```