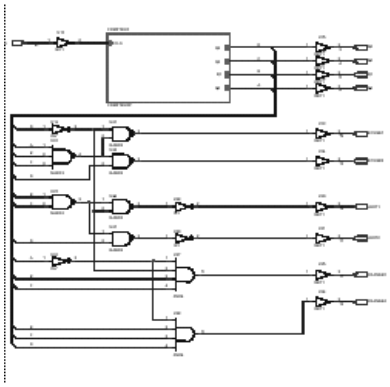
Présentation du langage VHDL

VHDL signifie VHSIC Hardware Description Language (VHSIC : Very High Speed Integrated Circuit). Ce langage a été écrit au milieu des années 80 pour réaliser la simulation de circuits électroniques. On l'a ensuite étendu en lui rajoutant des extensions pour permettre la conception (synthèse) de systèmes numériques.

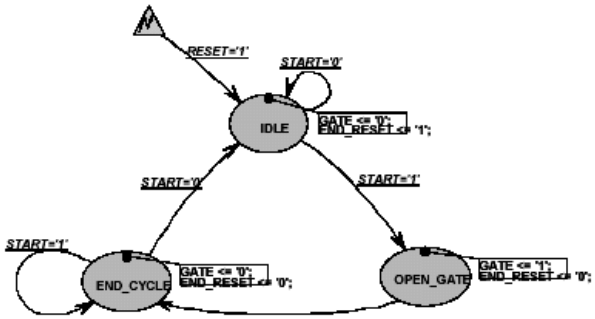
Le VHDL est un langage de haut niveau qui n'est pas lié à une cible technologique (AISC ou FPGA). Un autre de ces avantages est qu'il permet de décrire de façon très lisible et compacte le fonctionnement du système à concevoir. Il faut par contre ne jamais perdre de vue que l'exécution du code VHDL ne va pas effectuer les opérations que l'on attend du système, mais bien définir le système lui-même qui sera capable de les effectuer. En clair, les instructions VHDL seront traduites au final par des portes logiques et des bascules mais ne vont pas exécuter de tâches comme le ferait un programme en langage C par exemple. Cette confusion est pourtant très fréquente…

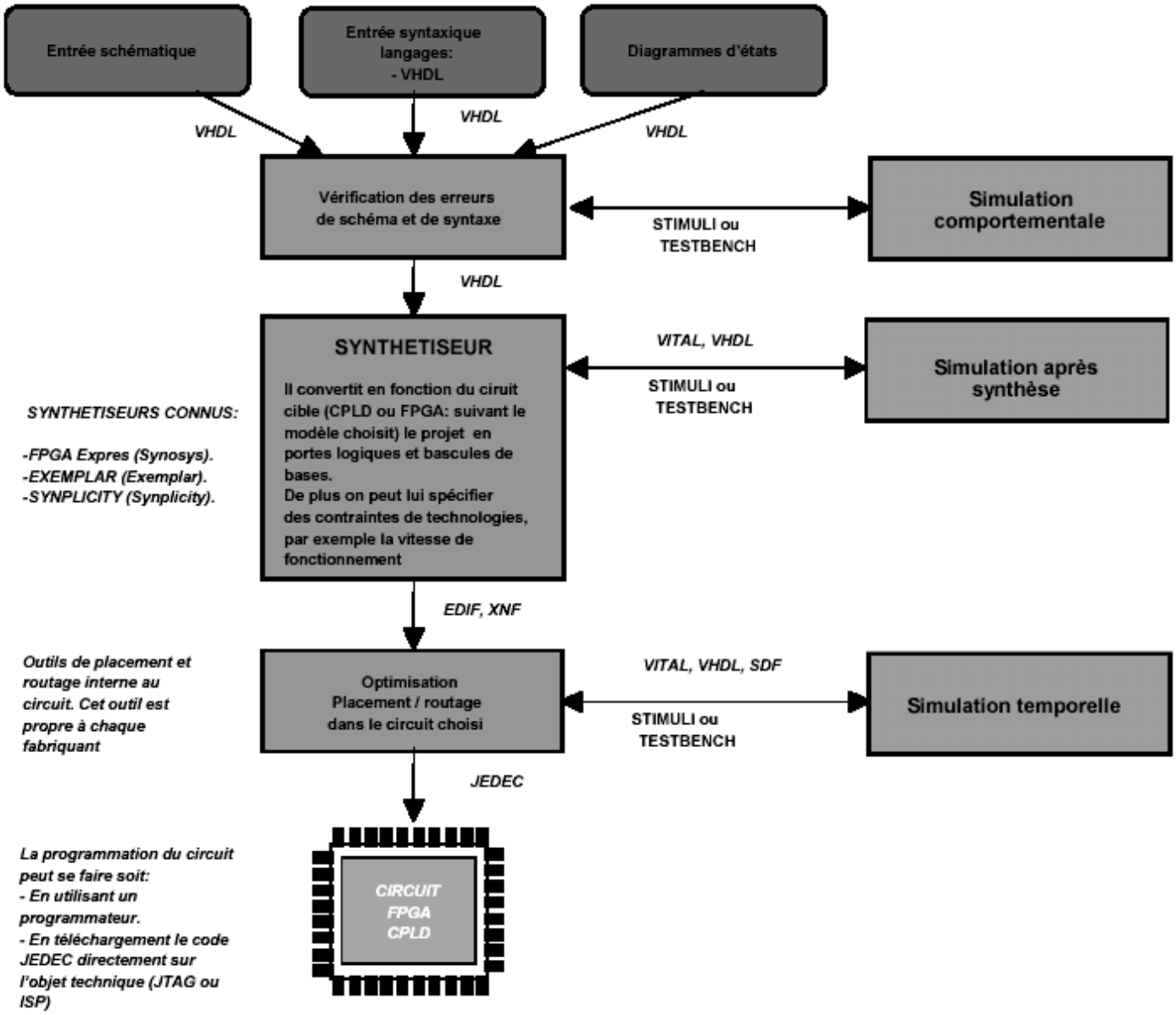
Ce document est une introduction à la synthèse de circuit en VHDL et présente la syntaxe de ce langage illustrée par quelques exemples. C'est un extrait du document rédigé par Philippe LETENNEUR et Philippe LECARDONNEL qui peut être consulté (ftp://ftp.discip.crdp.ac-caen.fr/discip/crgelec/Cours/vhdl.pdf) pour plus d'informations et d'exemples.

I) Organisation fonctionnelle de développement d'un circuit de logique programmable



```
library IEEE;
use IEEE.std_logic_1164.all;


entity CNT_4B is
    port (
        CLK: in STD_LOGIC;
        RESET: in STD_LOGIC;
        ENABLE: in STD_LOGIC;

        FULL: out STD_LOGIC;
        Q: out STD_LOGIC_VECTOR
(3 downto 0)
```





II) Structure d'une description VHDL

Une description VHDL est composée de 2 parties indissociables :

- L'entité (*ENTITY*) où l'on définit les entrées et les sorties
- L'architecture (*ARCHITECTURE*) qui va décrire le fonctionnement attendu

Exemple : DECODEUR/DEMULTIPLEXEUR 2 VERS 4

```
library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
```

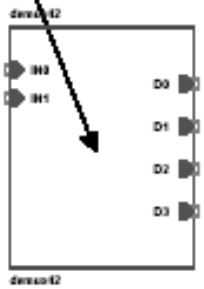
Déclaration des bibliothèques

Commentaires, en VHDL ils commencent par - -

```
-- démultiplexeur / décodeur
-- Deux vers quatre
```

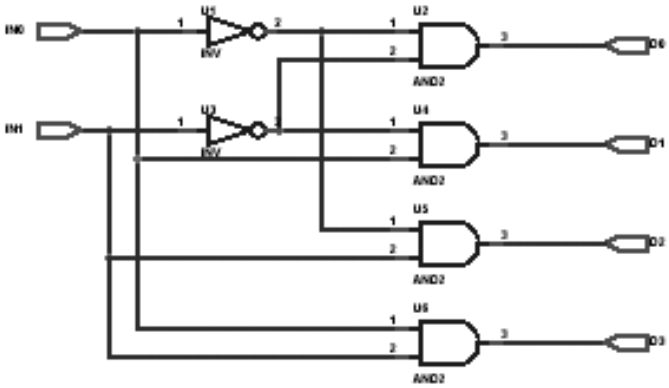
Déclaration de l'entité du démultiplexeur
Correspondance schématique

```
entity DEMUX2_4 is
      port(IN0, IN1: in std_logic;
           D0, D1, D2, D3: out std_logic);
end DEMUX2_4;
```



Déclaration de l'architecture du démultiplexeur
Correspondance schématique

```
architecture DESCRIPTION of DEMUX2_4 is
begin
      D0 <= (not(IN1) and not(IN0));
      D1 <= (not(IN1) and IN0);
      D2 <= (IN1 and not(IN0));
      D3 <= (IN1 and IN0);
end DESCRIPTION;
```



1) Déclaration des bibliothèques

Toute description **VHDL** utilisée pour la synthèse a besoin de bibliothèques. L'**IEEE** (**I**nstitut of **E**lectrical and **E**lectronics **E**ngineers) les a normalisées et plus particulièrement la bibliothèque **IEEE1164**. Elles contiennent les définitions des types de signaux électroniques, des fonctions et sous programmes permettant de réaliser des opérations arithmétiques et logiques,...

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
```

La directive **Use** permet de sélectionner les bibliothèques à utiliser.

2) Déclaration de l'entité et des entrées / sorties (I/O)

Elle permet de définir le NOM de la description **VHDL** ainsi que les entrées et sorties utilisées, l'instruction qui les définit c'est **port :**

```
Syntaxe:
entity NOM_DE_L_ENTITE is

     port ( Description des signaux d'entrées /sorties …);

end NOM_DE_L_ENTITE;

Exemple :
entity SEQUENCEMENT is
port (
     CLOCK     : in std_logic;
     RESET     : in std_logic;
     Q         : out std_logic_vector(1 downto 0)
     );
end SEQUENCEMENT;
```

Remarque : Après la dernière définition de signal de l'instruction `port` il ne faut jamais mettre de point virgule.

L'instruction **port** .

```
Syntaxe:   NOM_DU_SIGNAL   : sens     type;

Exemple:   CLOCK: in       std_logic;
           BUS   : out     std_logic_vector (7 downto 0);
```

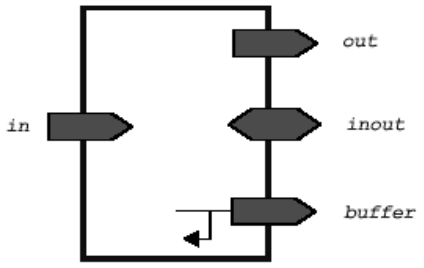
On doit définir pour chaque signal : le *NOM_DU_SIGNAL*, le ***sens*** et le ***type***.

➢ *nom du signal :*

Le nom du signal est composé de caractères, le premier caractère doit être une lettre, sa longueur est quelconque, mais elle ne doit pas dépasser une ligne de code. Le **VHDL** n'est pas sensible à la « casse », c'est à dire qu'il ne fait pas la distinction entre les majuscules et les minuscules.

➢ *sens :*

- ***in*** : pour un signal en entrée.
- ***out*** : pour un signal en sortie.
- ***inout*** : pour un signal en entrée sortie
- ***buffer*** : pour un signal en sortie mais utilisé comme entrée dans la description.



➢ *type :*

Un grand nombre de types sont définis en VHDL en fonction des opérations permises sur les signaux. Le type recommandé pour les signaux d'entrées / sorties est :

- le ***std_logic*** pour un signal.
- le ***std_logic_vector*** pour un bus composé de plusieurs signaux.

Un changement de type est toujours possible dans la description de l'architecture pour permettre localement certaines opérations (arithmétiques,…) sur le signal.

Par exemple un bus d'entrée de 5 bits s'écrira :

LATCH : in std_logic_vector (4 downto 0) ;

où ***LATCH(4)*** correspond au **MSB** et ***LATCH(0)*** correspond au **LSB**.

Les valeurs que peuvent prendre un signal de type ***std_logic*** sont au nombre de 9:

- ***'0' ou 'L'*** : pour un niveau bas.
- ***'1' ou 'H'*** : pour un niveau haut.
- **'Z'** : pour état haute impédance.
- **'W'** : pour un niveau inconnu forçage faible.
- **'X'** : pour un niveau inconnu forçage fort.
- **'U'** : pour un signal non initialisé.
- **'-'** : quelconque, c'est à dire n'importe quelle valeur.

3) Déclaration de l'architecture

L'architecture décrit le fonctionnement de l'entité correspondante.

Exemples :

```
-- Opérateurs logiques de base
entity PORTES is
      port (A,B :in std_logic;
      Y1,Y2,Y3,Y4,Y5,Y6,Y7:out std_logic);
end PORTES;

architecture DESCRIPTION of PORTES is
begin
      Y1 <= A and B;
      Y2 <= A or B;
      Y3 <= A xor B;
      Y4 <= not A;
      Y5 <= A nand B;
      Y6 <= A nor B;
      Y7 <= not(A xor B);
end DESCRIPTION;

-- Décodeurs 7 segments
entity DEC7SEG4 is
      port (DEC :in std_logic_vector(3 downto 0);
      SEG:out std_logic_vector(0 downto 6));
end DEC7SEG4;

architecture DESCRIPTION of DEC7SEG4 is
begin
      SEG <= "1111110" when DEC = 0
      else "0110000" when DEC = 1
      else "1101101" when DEC = 2
      else "1111001" when DEC = 3
      else "0110011" when DEC = 4
      else "1011011" when DEC = 5
      else "1011111" when DEC = 6
      else "1110000" when DEC = 7
      else "1111111" when DEC = 8
      else "1111011" when DEC = 9
      else "-------";
end DESCRIPTION;
```

III) Le mode combinatoire

Pour une description **VHDL** toutes les instructions sont évaluées et affectent les signaux de sortie en même temps. **L'ordre dans lequel elles sont écrites n'a aucune importance**. En effet la description génère des structures électroniques, c'est la grande différence entre une description **VHDL** et un langage informatique classique.

Dans un système à microprocesseur, les instructions sont exécutées les unes à la suite des autres. En **VHDL** il faut essayer de penser à la structure qui va être générée par le synthétiseur pour écrire une bonne description, ce qui n'est pas toujours évident…

1) Les opérateurs

a) L'affectation simple <=

Dans une description VHDL c'est certainement l'opérateur le plus utilisé. En effet il permet de modifier l'état d'un signal en fonction d'autres signaux et/ou d'autres opérateurs.

Exemple avec des portes logiques : *S1 <= E2 and E1 ;*

Les valeurs numériques que l'on peut affecter à un signal sont au nombre de 9 :

- **'1'** ou **'H'** pour un niveau **haut** avec un signal de **1** bit.
- **'0'** ou **'L'** pour un niveau **bas** avec un signal de **1** bit.
- **'W'** pour un niveau inconnu forçage faible avec un signal de **1** bit.
- **'X'** pour un niveau inconnu forçage fort avec un signal de **1** bit.
- **'Z'** pour un état haute impédance avec un signal de **1** bit.
- **'U'** pour un signal non initialisé avec un signal de **1** bit.
- **'-'** pour un état quelconque, c'est à dire **'0'** ou **'1'**. Cette valeur est très utilisée avec les instructions : ***when … else*** et ***with …. Select ….***
- Pour les **signaux** composés de plusieurs bits on utilise les guillemets **" … "** , voir les exemples ci dessous :
- Les bases numériques utilisées pour les bus peuvent être :

BINAIRE, exemple : ***BUS <= "1001" ;*** -- *BUS = 9 en décimal*
HEXA, exemple : ***BUS <= X"9" ;*** -- *BUS = 9 en décimal*
OCTAL, exemple : ***BUS <= O"11" ;*** -- *BUS = 9 en décimal*

Exemple:

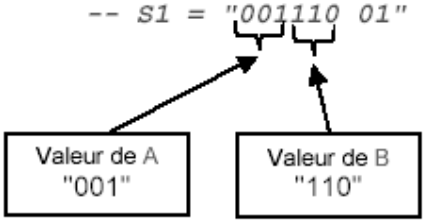
```
Library ieee;
Use ieee.std_logic_1164.all;

entity AFFEC is
port (
      E1,E2              : in std_logic;
      BUS1,BUS2,BUS3     : out std_logic_vector(3 downto 0);
      S1,S2,S3,S4        : out std_logic);
end AFFEC;

architecture DESCRIPTION of AFFEC is

begin
S1 <= '1'; -- S1 = 1
S2 <= '0'; -- S2 = 0
S3 <= E1;  -- S3 = E1
S4 <= '1' when (E2 ='1') else 'Z'; -- S4 = 1 si E1=1 sinon S4
                                   -- prend la valeur haute impédance
BUS1 <= "1000";                    -- BUS1 = "1000"
BUS2 <= E1 & E2 & "10";            -- BUS2 = E1 & E2 & 10
BUS3 <= x"A";                      -- valeur en HEXA -> BUS3 = 10(déc)

end DESCRIPTION;
```

*b) <u>Opérateurs de concaténation **&**</u>*

Cet opérateur permet de joindre des signaux entre eux.

Exemple :

```
--   Soit A et B de type 3 bits et S1 de type 8 bits
--   A = "001"  et B ="110"
S1 <= A & B & "01" ;
-- S1 prendra la valeur suivante après cette affectation
-- S1 = "001110 01"
```



c) <u>Opérateurs logiques</u>

Opérateur	VHDL
ET	`and`
NON ET	`nand`
OU	`or`
NON OU	`nor`
OU EXCLUSIF	`xor`
NON OU EXCLUSIF	`xnor`
NON	`not`
DECALAGE A GAUCHE	`sll`
DECALAGE A DROITE	`srl`
ROTATION A GAUCHE	`rol`
ROTATION A DROITE	`ror`

Exemples :

S1 <= A sll 2 ; -- S1 = A décalé de 2 bits à gauche.

S2 <= A rol 3 ; -- S2 = A avec une rotation de 3 bits à gauche

S3 <= not (R); -- S3 = R

Remarque : Pour réaliser des décalages logiques en synthèse logique, il est préférable d'utiliser les instructions suivantes :

Décalage à droite :

```
-- Si A est de type std_logic_vector(7 downto 0)
S1 <= '0' & A(7 downto 1); -- décalage d'un bit à droite
S1 <= "000" & A(7 downto 3); -- décalage de trois bits à droite
```

Décalage à gauche :

```
-- Si A est de type std_logic_vector(7 downto 0)
S1 <= A(6 downto 0) & '0'; -- décalage d'un bit à gauche
S1 <= A(4 downto 0) & "000"; -- décalage de trois bits à gauche
```

(cette autre écriture évite ainsi l'utilisation de portes logiques !)

e) Opérateurs relationnels

Ils permettent de modifier l'état d'un signal ou de signaux suivant le résultat d'un test ou d'une condition. En logique combinatoire ils sont souvent utilisés avec les instructions :

- when … else …
- with …. Select ….

Opérateur	VHDL
Egal	=
Non égal	/=
Inférieur	<
Inférieur ou égal	<=
Supérieur	>
Supérieur ou égal	>=

2) Les instructions

a) Affectation conditionnelle

Cette instruction modifie l'état d'un signal suivant le résultat d'une condition logique entre un ou des signaux, valeurs, constantes.

SIGNAL <= **expression when condition**
[else expression when condition]
[else expression];

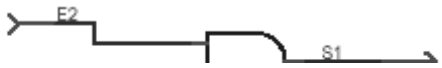
Remarque : l'instruction **[else expression]** n'est pas obligatoire mais elle est fortement conseillée car elle permet de définir la valeur du **SIGNAL** dans le cas où la condition n'est pas remplie.

On peut mettre en cascade cette instruction (voir l'exemple N°2).

Exemple N°1 :

```
-- S1 prend la valeur de E2 quand E1='1' sinon S1 prend la --
valeur '0'
S1 <= E2 when ( E1= '1') else '0';
```

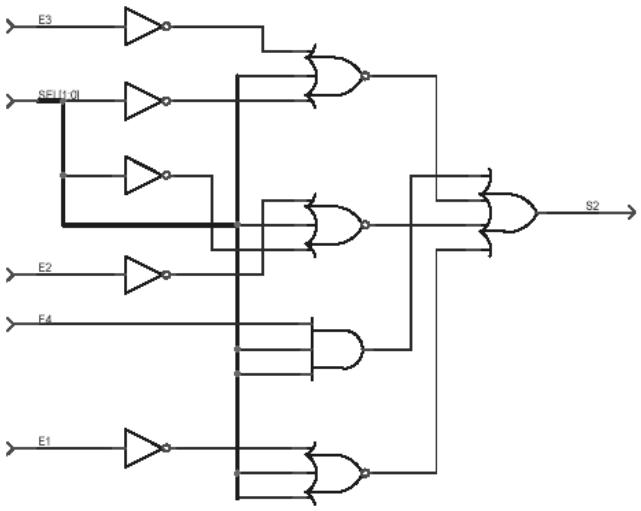
Schéma correspondant :



Exemple N°2 :

```
-- Structure évoluée d'un multiplexeur 4 vers 1
S2 <= E1 when (SEL="00" ) else
      E2 when (SEL="01" ) else
      E3 when (SEL="10" ) else
      E4 when (SEL="11" )
      else '0';
```

Schéma correspondant après synthèse:



b) Affectation sélective

Cette instruction permet d'affecter différentes valeurs à un signal, selon les valeurs prises par un signal dit de sélection.

with SIGNAL_DE_SELECTION **select**
SIGNAL <= **expression when valeur_de_selection,**
[expression when valeur_de_selection,]
[expression when others];

Remarque: l'instruction **[expression when others]** n'est pas obligatoire mais fortement conseillée car elle permet de définir la valeur du **SIGNAL** dans le cas où la condition n'est pas remplie.

Exemple N°1 :

```
-- Multiplexeur 4 vers 1
with SEL select
    S2 <=   E1  when "00",
            E2  when "01",
            E3  when "10",
            E4  when "11",
            '0' when others;
```

Remarque: when others est nécessaire car il faut toujours définir les autres cas du signal de sélection pour prendre en compte toutes les valeurs possibles de celui-ci.

IV) Le mode séquentiel

1) Définition d'un PROCESS

Un **process** est une partie de la description d'un circuit dans laquelle les instructions sont exécutées séquentiellement c'est à dire les unes à la suite des autres. Il permet d'effectuer des opérations sur les signaux en utilisant les instructions standards de la programmation structurée comme dans les systèmes à microprocesseurs.

L'exécution d'un **process** est déclenchée par un ou des changements d'états (***event***) de signaux logiques. Le nom de ces signaux est défini dans **la liste de sensibilité** lors de la déclaration du **process**.

[Nom_du_process :] process (Liste_de_sensibilité_nom_des_signaux)
Begin
    -- instructions du process
end process [Nom_du_process] ;

Remarque: Le nom du **process** entre crochet est facultatif, mais il peut être très utile pour repérer un **process** parmi d'autres lors de phases de mise au point ou de simulations.

Règles de fonctionnement d'un process :

1) L'exécution d'un **process** a lieu à chaque changement d'état d'un signal de la liste de sensibilité.
2) Les instructions du **process** s'exécutent séquentiellement.
3) Les changements d'état des signaux par les instructions du **process** sont pris en compte à la **fin** du **process**.

2) Les 2 principales structures utilisées dans un process

L'assignation conditionnelle	L'assignation sélective

`if condition then`
`  instructions`
`[elsif condition then instructions]`
`[else instructions]`
`end if ;`
 	
Exemple:
`if (RESET='1') then SORTIE <= "0000";`
`end if ;`
	`case signal_de_slection is`
	`when valeur_de_sélection => instructions`
	`[when others => instructions]`
	`end case;`
	
	Exemple:
	`case SEL is`
	`  when "000" => S1 <= E1;`
	`  when "001" => S1 <= '0';`
	`  when "010" | "011" => S1 <='1';`
	`  -- La barre | permet de réaliser`
	`  -- un ou logique entre les deux`
	`  -- valeurs "010" et "011"`
	`  when others => S1 <= '0';`
	`end case;`

3) Exemples de process

Exemple N°1 : Déclaration d'une bascule D.

```
Library ieee;
Use ieee.std_logic_1164.all;

entity BASCULED is
      port (
            D,CLK : in std_logic;
            S     : out std_logic);
end BASCULED;
architecture DESCRIPTION of BASCULED is
begin
      PRO_BASCULED : process (CLK)
      begin
            if (CLK'event and CLK ='1') then
                  S <= D;
            end if;
      end process PRO_BASCULED;
end DESCRIPTION;
```

Remarques

- Seul le signal ***CLK*** fait partie de la liste de sensibilité. D'après les règles de fonctionnement énoncées précédemment, seul un changement d'état du signal ***CLK*** va déclencher l'exécution du process et par conséquent évaluer les instructions de celui-ci.

- L'instruction ***if (CLK'event and CLK='1') then*** permet de détecter un front montant du signal ***CLK***. La détection de front est réalisée par l'attribut ***event*** appliqué à l'horloge ***CLK***. Si on veut un déclenchement sur un front descendant, il faut écrire l'instruction suivante : ***if (CLK'event and CLK='0')***.

- Les bibliothèques IEEE possèdent deux instructions permettant de détecter les fronts montants***) rising_edge(CLK)*** ou descendants ***falling_edge(CLK).***

- Si la condition est remplie alors le signal de sortie ***S*** sera affecté avec la valeur du signal d'entrée ***D***.

Exemple N°2 : Même exemple que précédemment mais avec des entrées de présélections de mise à zéro *RESET* prioritaire sur l'entrée de mise à un *SET*, toutes les deux sont synchrones de l'horloge *CLK*.

```
Library ieee;
Use ieee.std_logic_1164.all;


entity BASCULEDSRS is
      port (
            D,CLK,SET,RESET   : in std_logic;
            S                 : out std_logic);
end BASCULEDSRS;

architecture DESCRIPTION of BASCULEDSRS is
begin
      PRO_BASCULEDSRS : process (CLK)
      Begin
            if (CLK'event and CLK ='1') then
                  if (RESET ='1') then
                        S <= '0';
                  elsif (SET ='1')then
                        S <= '1';
                  else
                        S <= D;
                  end if;
            end if;
      end process PRO_BASCULEDSRS;
end DESCRIPTION;
```

Exemple N°3 : Même exemple que précédemment mais avec des entrées de présélections, de mise à zéro *RESET* prioritaire sur l'entrée de mise à un *SET*, toutes les deux sont asynchrones de l'horloge *CLK*.

```
Library ieee;
Use ieee.std_logic_1164.all;


entity BASCULEDSRA is
      port (
            D,CLK,SET,RESET   : in std_logic;
            S                 : out std_logic);
end BASCULEDSRA;

architecture DESCRIPTION of BASCULEDSRA is
begin
      PRO_BASCULEDSRA : process (CLK,RESET,SET)
      Begin
            if (RESET ='1') then
                  S <= '0';
            elsif (SET ='1')then
                  S <= '1';
            elsif (CLK'event and CLK ='1') then
                  S <= D;
            end if;
      end process PRO_BASCULEDSRA;
end DESCRIPTION;
```

Remarque : l'entrée ***RESET*** est prioritaire sur l'entrée ***SET***. Elles sont toutes les deux asynchrones car les signaux d'entrée ***SET*** et ***RESET*** sont mentionnés dans la liste de sensibilité du ***process***.

Exemple N°4 : compteur 3 bits avec remise à 0 synchrone

1-) Description recommandée :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity CMP3BITS is
   Port ( CLOCK  : in  STD_LOGIC;
          RESET  : in  STD_LOGIC;
             Q      : out  STD_LOGIC_VECTOR (2 downto 0));
end CMP3BITS;

architecture DESCRIPTION of CMP3BITS is

signal CMP : unsigned (2 downto 0);

begin

Synchrone : Process(CLOCK)
begin
  if (CLOCK='1' and CLOCK'event) then
    if (RESET ='1') then
          CMP  <= "000";
    else
          CMP <= CMP + "001";
           end if;
  end if;
end Process Synchrone;

Q <= std_logic_vector (CMP);

end DESCRIPTION;
```

Pour pouvoir utiliser les opérateurs arithmétiques, il faut rajouter le package normalisé: ieee.numeric_std.all. Ce dernier permet de manipuler uniquement des signaux de type *unsigned* ou *signed*.
Le type *unsigned* représente des valeurs numériques entières entre 0 et 2^{N-1}.
Le type *signed* représente des valeurs numériques entières entre -2^{N-1} et $2^{N-1}-1$ en complément à 2.

C'est la raison pour laquelle des transformations de type sont nécessaires en amont et en aval d'un calcul.

- ✓ Transformation Std_logic vers Unsigned : unsigned(operand)
- ✓ Transformation Std_logic vers Signed : signed(operand)
- ✓ Transformation Unsigned/Signed vers Std_logic: std_logic(operand)

L'incrémentation du compteur est réalisée par l'opérateur + associé à la valeur **1**. Cela est logique pour nous, mais elle l'est beaucoup moins pour les architectures numériques. En

effet, les entrées et sorties sont déclarés de type ***std_logic*** ou ***std_logic_vector***, c'est-à-dire qu'un bit peut prendre comme valeur les états **'1'** ou **'0'** et un bus n'importe quelle valeur, du moment qu'elle est écrite entre deux guillemets **"1010"** ou **X"A"** ou **o"12",** mais pas une valeur comme par exemple **1,2,3,4**. Les valeurs décimales sont interprétées par le synthétiseur comme des valeurs entières (**integer**). Du coup, nous ne pouvons pas par défaut additionner un nombre entier **1** avec un bus de type électronique (***std_logic_vector***). C'est pour cela que l'on rajoute la valeur binaire **"001"**au signal CMP de type unsigned.

La description fait appel à un signal interne ***CMP***. En effet, le signal ***S*** est déclaré comme une sortie dans l'entité et ne peut donc être lu. Pour contourner cette difficulté on utilise un signal interne qui peut être à la fois lu et modifié.

La mise à zéro des sorties du compteur passe par l'instruction : ***CMP <= "0000";***
Une autre façon d'écrire cette instruction est : ***CMP <= (others => '0') ;***
Elle est très intéressante car elle permet de s'affranchir de la taille du bus.

2-) Autre description possible mais fortement déconseillée :

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
Use ieee.std_logic_unsigned.all;
entity CMP3BITS is
PORT (
      CLOCK : in std_logic;
      RESET : in std_logic;
      Q     : out std_logic_vector (2 downto 0));
end CMP3BITS;
architecture DESCRIPTION of CMP3BITS is
signal CMP: std_logic_vector (2 downto 0);
begin
  process (CLOCK)
  begin
    if (CLOCK ='1' and CLOCK'event) then
      if RESET ='1' then
         CMP <= "000";
      else
         CMP <= CMP + 1;
      end if;
    end if;
  end process;
  Q <= CMP;
end DESCRIPTION;
```

ATTENTION : Cette description est déconseillée car elle repose sur le package **ieee.std_logic_unsigned.all** qui n'est pas normalisé.

4) Recommandations

① **Ne pas oublier que l'affectation des signaux n'a lieu qu'à la sortie du process !** (c'est avec les signaux internes qu'on se fait souvent piéger…)

② **Un même signal doit toujours être affecté dans un même process.** Sinon le synthétiseur crée automatiquement un bus car on ne peut relier entre elles 2 sorties de bascules ! (Il peut être préférable d'utiliser le type ***std_ulogic*** au lieu de ***std_logic*** ce qui générera une erreur au moment de la compilation)

V) La description structurelle en VDHL

1) Préambule

La description structurelle d'un circuit complexe en vhdl présente de nombreux avantages :

- Une architecture hiérarchique compréhensible : il est plus simple de séparer un circuit en un ensemble de blocs plus petits, ayant des fonctions bien identifiées. Ces blocs pourront alors être décrits sous forme comportementale, ou bien à leur tour être séparés en blocs encore plus simples.
- Une synthèse logique efficace : la synthèse est un processus lent (en terme de temps de calcul). Plus un bloc est gros et complexe, plus sa synthèse prendra du temps. Il vaut donc mieux travailler sur des blocs plus petits, plus simples à synthétiser, et rassembler le tout à la fin.

L'objectif de cette section est de voir précisément comment coder une représentation structurelle d'un circuit, autrement dit :

- comment déclarer des blocs (qu'on appellera composant)
- comment utiliser un composant (qui devient une "instance") et déclarer la façon dont il est connecté.
- comment choisir l'architecture d'un composant quand il y en a plusieurs (configurations)

2) Elément de base

VHDL permet l'assemblage de "**composants**" ce qui constitue une description structurelle. Ce composant peut être appelé plusieurs fois dans un même circuit. Pour différencier ces mêmes composants, il est nécessaire de leur donner un nom d'"**instance**". L'appel d'un composant se dit aussi "**instanciation**"

De façon à instancier un composant il est nécessaire de connaître :

- Le prototype du composant (ses ports d'entrée et de sortie). La directive component peut être utilisée à cette fin.
- A quelle entité et architecture est lié chaque instance de composant. Ce lien peut être connu grâce à l'unité de configuration.

Il est important de noter :

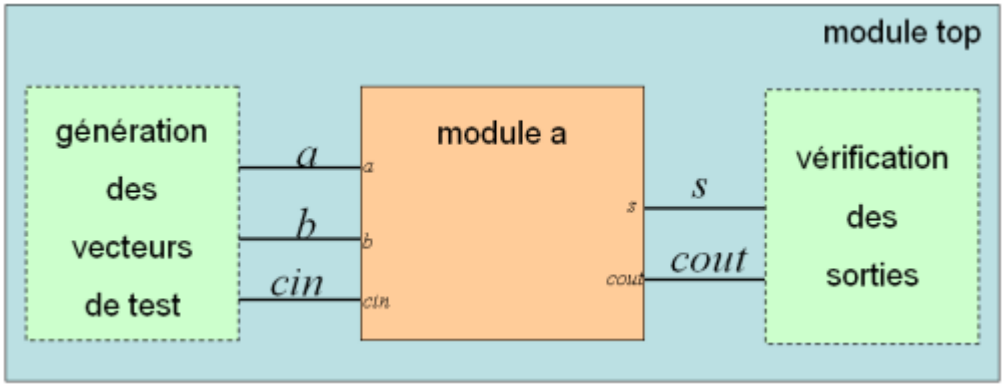
- La déclaration du composant (directive component) est redondante textuellement avec celle de l'entité associée mais permet :

    1. Une compilation indépendante entre l'entité associée au composant et le circuit utilisant le composant.
    2. La conception descendante. Le composant peut être déclaré avant l'entité associée.

- La configuration est une unité de compilation **optionnelle**, très utile pour les gros circuits. Par exemple pour accélérer la simulation, un même composant peut être associé à un couple entité/architecture détaillé et synthétisable ou un autre couple plus abstrait et plus rapide à simuler. Pour ne pas utiliser de configuration, une règle fréquente est d'utiliser le même nom pour le composant et l'entité associée, c'est le cas pour ISIM et les outils de synthèse FPGA.

La description structurelle est nécessaire pour simuler un circuit dont les vecteurs stimulis sont eux mêmes issus d'un modèle VHDL. Le modèle de plus haut niveau fait donc appel au circuit à tester (Device Under Test) et d'un générateur de stimulis. Ces deux objets sont instanciés dans un même circuit, généralement appelé "testbench" (mais ce n'est pas une obligation) qui est autonome : il n'aura *pas* d'entrées ni de sorties.

Exemple : le circuit top, servant à simuler le circuit "module a" doit être autonome : son entité n'a pas d'entrée ni de sortie.



Cas particulier de la simulation : circuit "top" sans entrée ni sortie

3) Déclaration et instanciation des composants

Déclaration

Le mot clé component sert à déclarer le prototype d'interconnexion. La syntaxe est presque identique à celle de l'entité :

```
component AND_2
 port (
 a : in std_logic;
 b : in std_logic;
 s : out std_logic);
end component;
```

Pour créer rapidement un composant, une opération copier/coller de l'entité en enlevant le littéral "IS" suffit.

Instanciation

L'instanciation d'un composant se fait dans le corps de l'architecture de cette façon :

```
<NOM_INSTANCE>:<NOM_COMPOSANT> port map(LISTE DES CONNEXIONS);
```

Exemple:

```
entity AND_3 is
 port(
 e1 : in std_logic;
 e2 : in std_logic;
 e3 : in std_logic;
 s : out std_logic
 );
end entity;
--
architecture arc of AND_3 is
--
signal z : bit;
component and2
       port (
       a : in std_logic;
       b : in std_logic;
       s : out std_logic);
end component;
--
begin
 inst1 : and2 port map (a=>e1, b=>e2 , s=>z);
 inst2 : and2 port map (z, e3, s);
end arc
```

Dans cet exemple, 2 instances de composant "and2" sont appelées pour créer une porte ET à3 entrées.

L'association des ports du composant aux signaux de l'instance se fait à l'aide de la clause *port map*.

La syntaxe des associations est soit

1. par nom où chaque broche du composant est associée à un signal : cas de inst_1
2. positionnelle où l'ordre des signaux correspond à l'ordre des broches : cas de inst_2

V) Bibliographie

J. Weber, S. Moutault, M. Meaudre, *Le langage VHDL – du langage au circuit, du circuit au langage*, 3ème éd., Dunod – 2007.

R. Airiau, J.M. Bergé, V. Olive, J. Rouillard, *VHDL du langage à la modélisation*, Presses Polytechniques et Universitaires Romandes – 1990.

Z. Navabi, *VHDL : Analysis and Modeling of Digital Systems*, McGraw Hill – 1993.