

# Le langage VHDL

ENSIAS Rabat

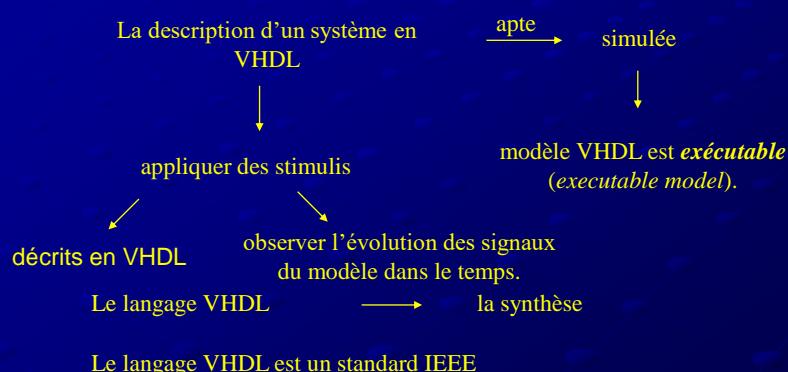
Z. ALAOUI ISMAILI

1

## 2.1. Qu'est ce que le langage VHDL?

VHICL Hardware Description Language, où VHICL signifie Very High-Speed Integrated Circuit.

Le langage VHDL permet la description de tous les aspects d'un système matériel (*hardware system*): son comportement, sa structure et ses caractéristiques temporelles.



Le langage VHDL est un standard IEEE

ENSIAS Rabat

Z. ALAOUI ISMAILI

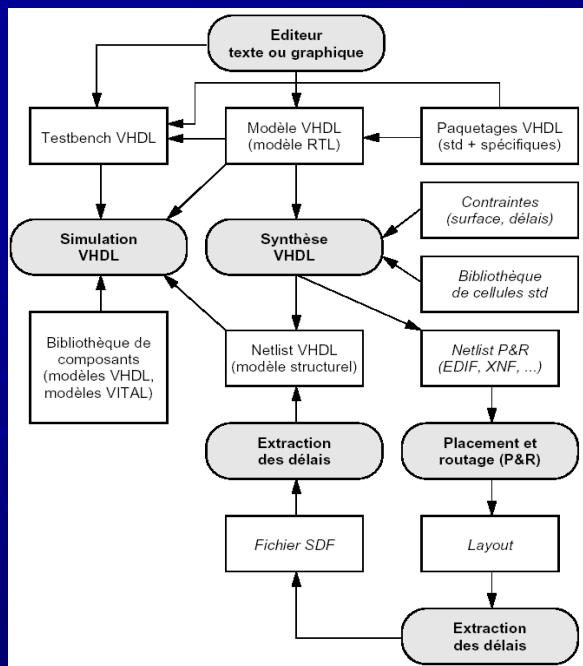
2

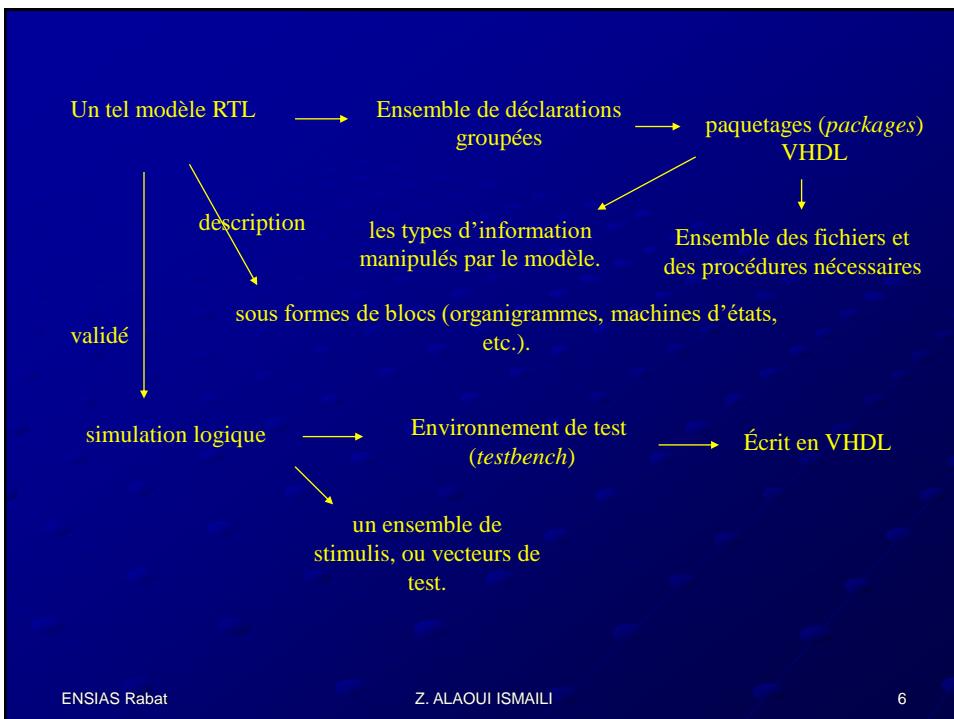
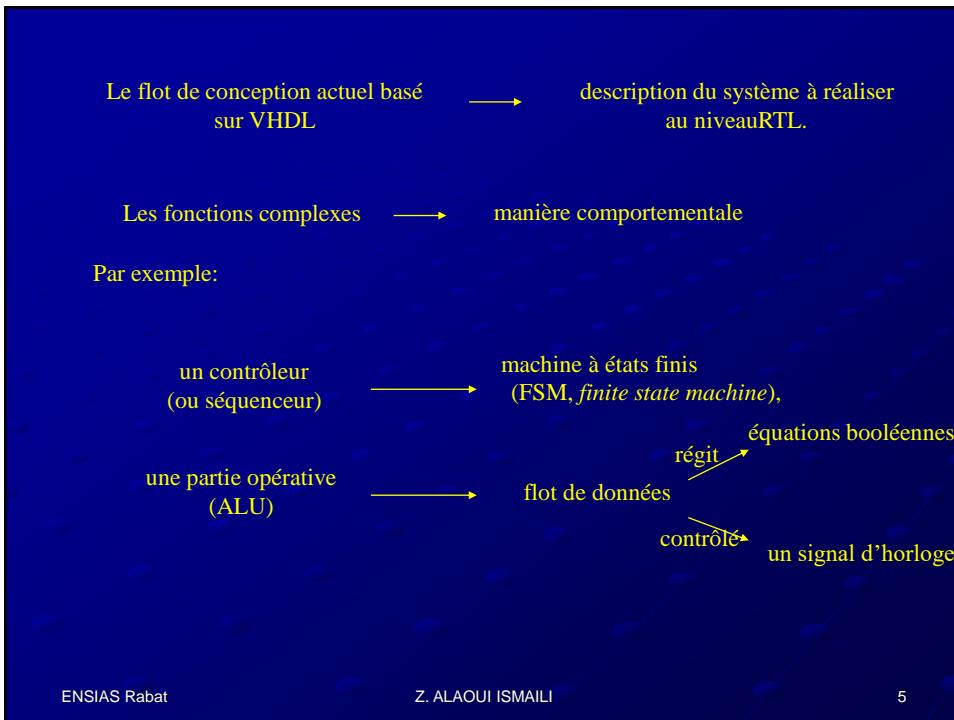
## 2.2. Flot de conception basé sur VHDL

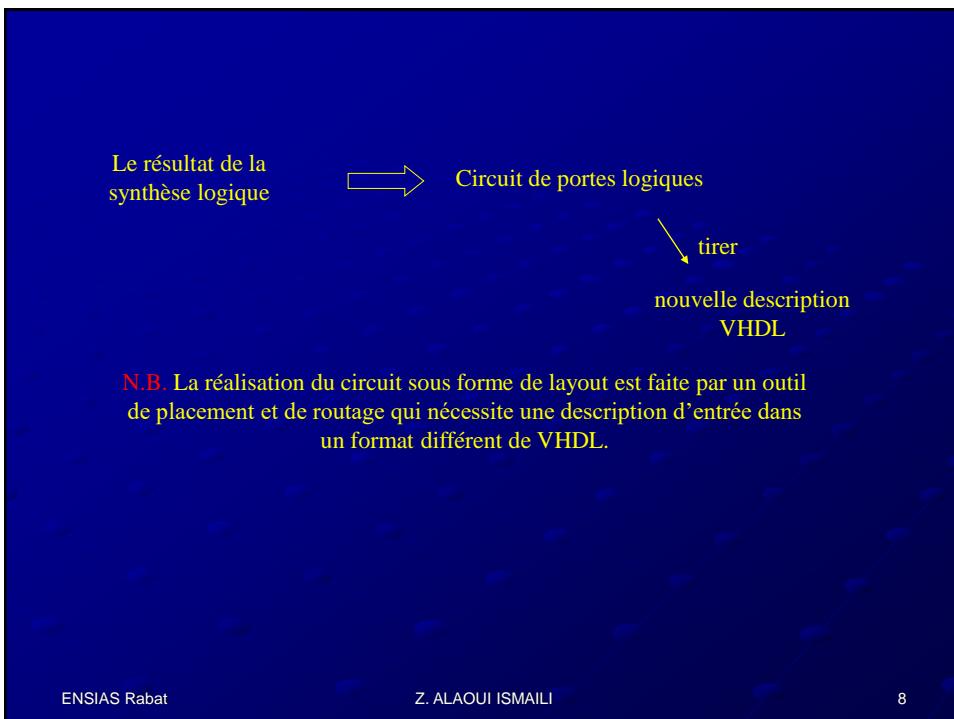
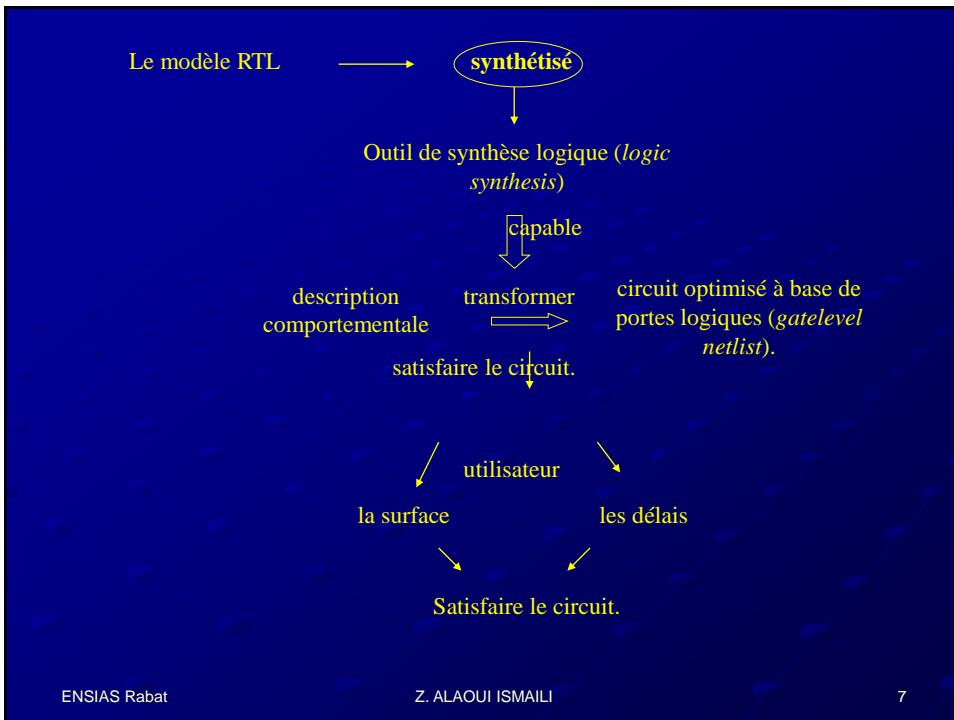
La Figure 2.1 illustre les différentes étapes du flot de conception ainsi que les outils utilisés et les informations (données de conception) nécessaires.

**Les composants avec texte en italique représentent des données qui ne peuvent pas être représentées en VHDL.**

**Les composants grisés représentent des outils EDA**

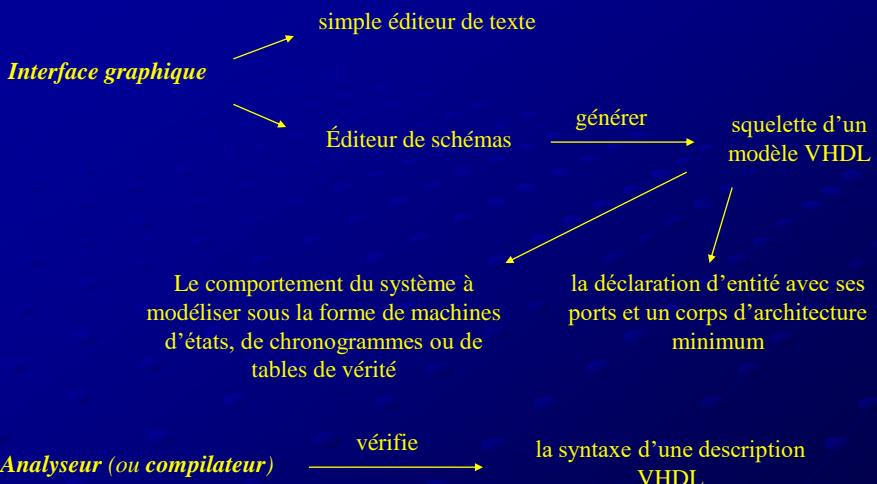
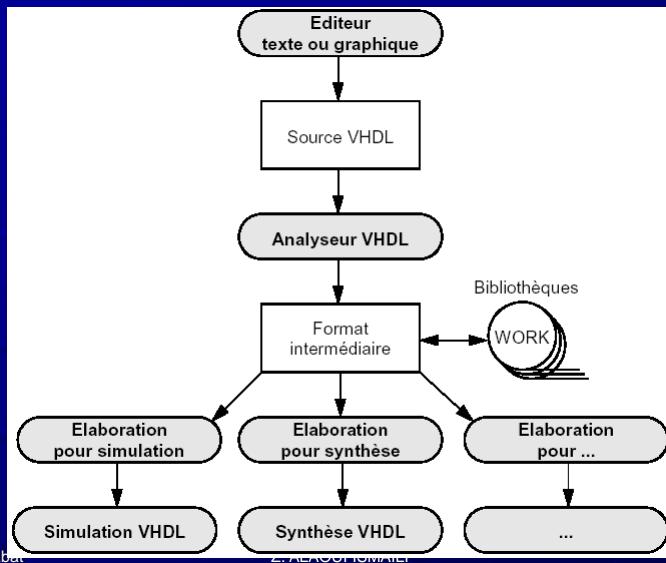


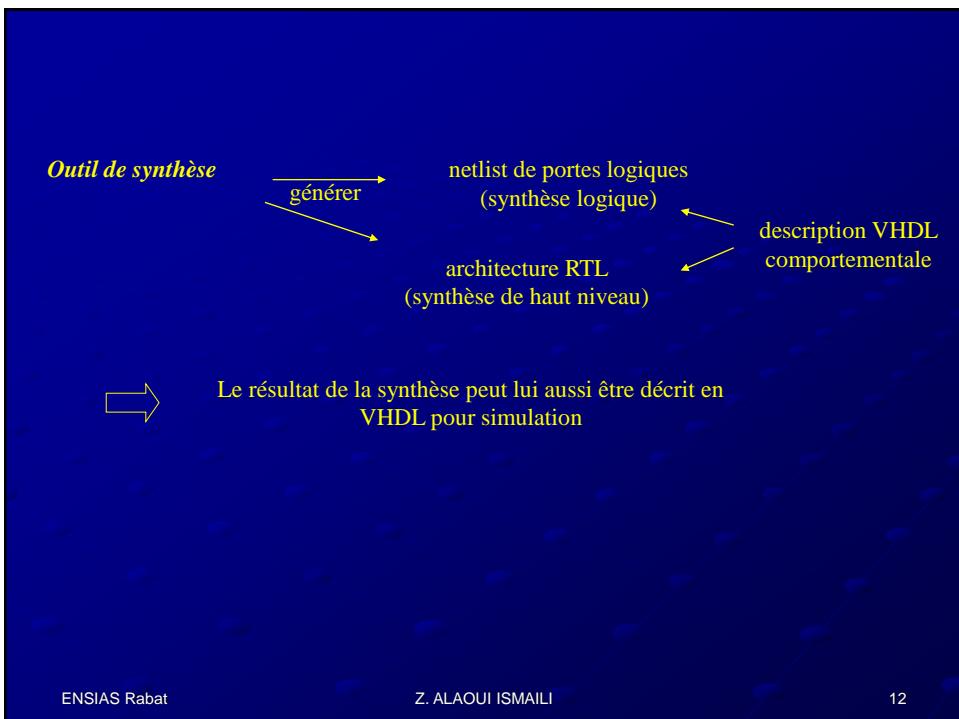
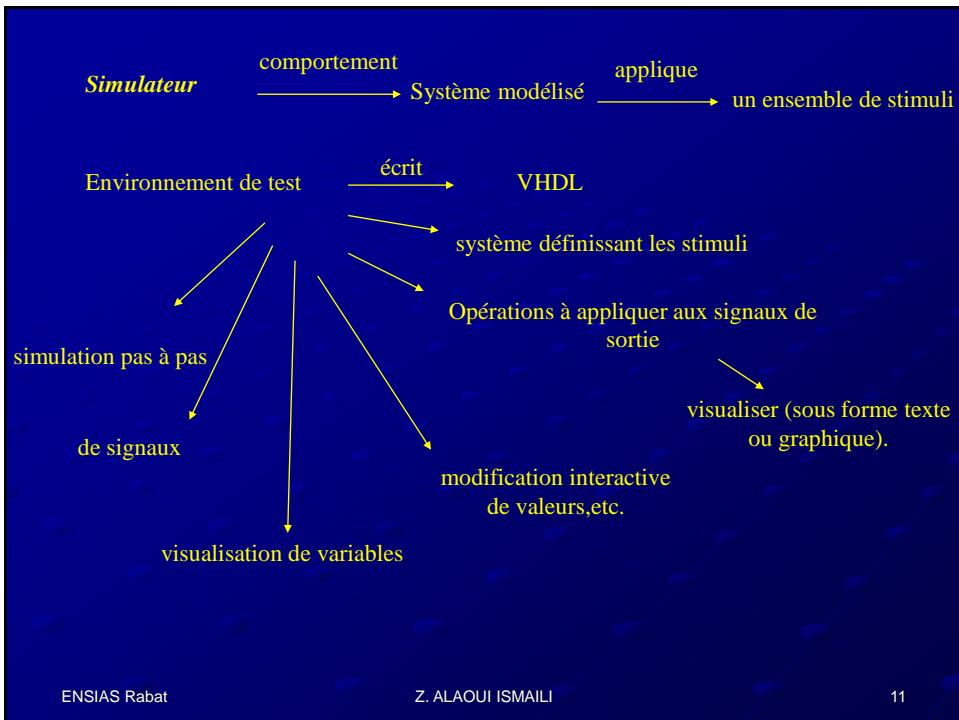




### 2.3. Environnement de travail VHDL

La Figure 2.2 illustre l'environnement de travail type de VHDL et les différentes phases d'édition, d'analyse, d'élaboration et d'exécution liées au langage.





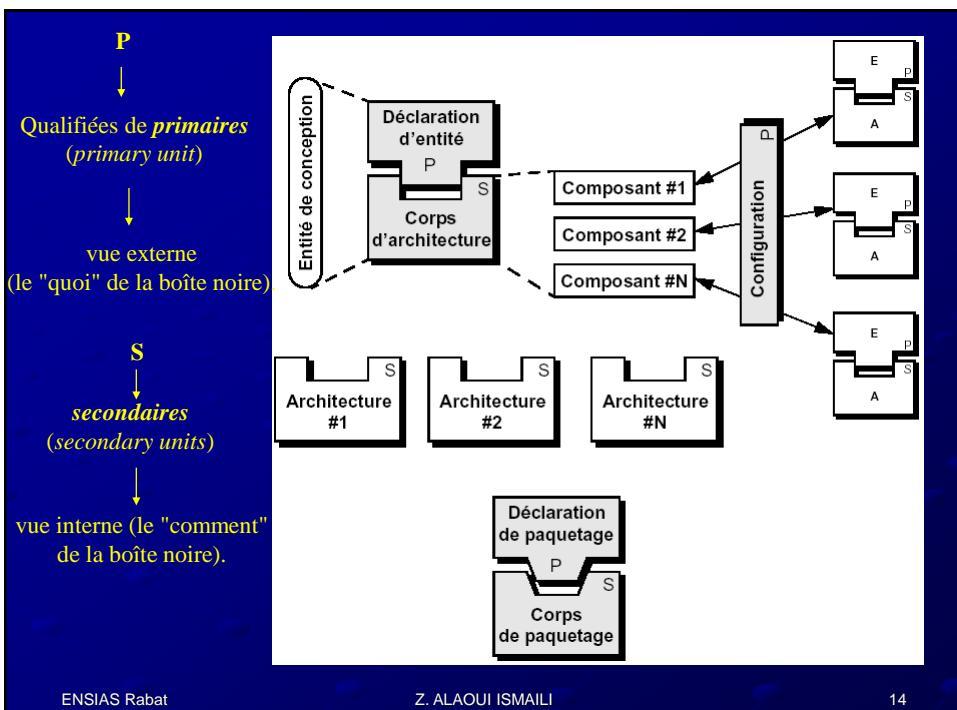
## 2.4. Organisation d'un modèle VHDL

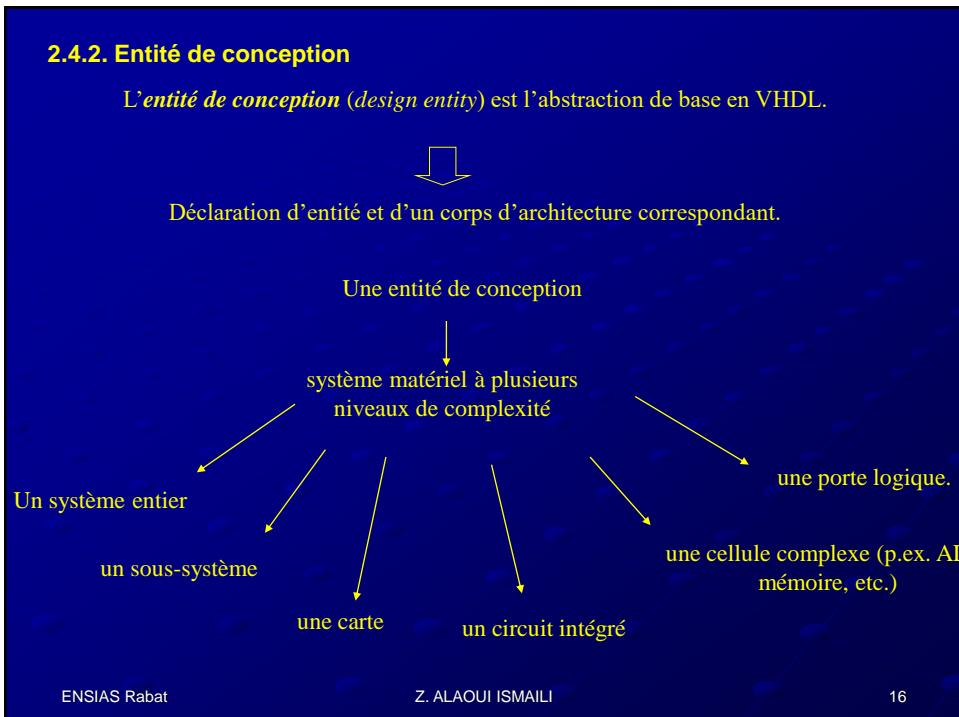
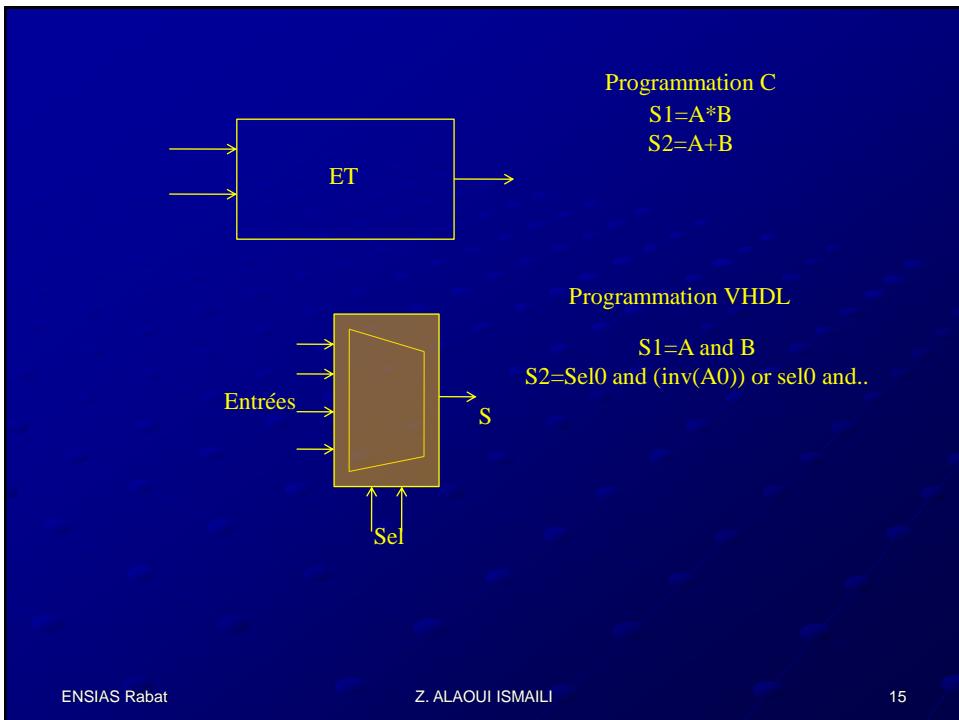
### 2.4.1. Unités de conception

L'*unité de conception* (*design unit*) est le plus petit module compilable séparément.

VHDL offre cinq types d'unités de conception (Figure 2.3):

- ✓ la déclaration d'entité (*entity declaration*);
- ✓ le corps d'architecture (*architecture body*), ou plus simplement architecture;
- ✓ la déclaration de configuration (*configuration declaration*);
- ✓ la déclaration de paquetage (*package declaration*);
- ✓ le corps de paquetage (*package body*).





### 2.4.3. Bibliothèques de conception

VHDL possède deux bibliothèques logiques prédefinies:

- La bibliothèque de nom logique WORK est le dépositaire de toutes les unités de conception compilées.



C'est en fait la seule bibliothèque dans laquelle il est possible d'ajouter et de modifier des éléments.

- La bibliothèque de nom logique STD est prédefinie et contient deux unités:

- ✓ le paquetage STANDARD, qui inclut les définitions des types, opérateurs et sous-programmes prédefinis;
- ✓ le paquetage TEXTIO, qui inclut les définitions et les sous-programmes relatifs au traitement de fichiers textes.

### 2.5. Modèles VHDL d'un registre 4 bits

Le langage VHDL n'est pas sensible à la casse (minuscule, majuscule) des lettres.

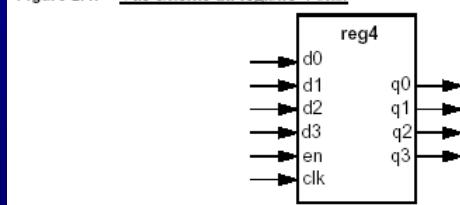
Plusieurs versions possibles d'un modèle de registre 4 bits en VHDL

→ Illustrer l'usage des différentes unités de conception et des styles de description disponibles en VHDL.

#### 2.5.1. Déclaration d'entité

La Figure 2.4 illustre la vue externe du registre 4 bits.

Figure 2.4. Vue externe du registre 4 bits.



**Code 2.3:** Déclaration d'entité du registre 4 bits.

```
entity reg4 is
  port (
    d0, d1, d2, d3: in bit;      -- données d'entrée
    en, clk: in bit;            -- signal de contrôle et horloge
    q0, q1, q2, q3: out bit    -- données de sortie
  );
end entity reg4;
```

Les mots-clés réservés du langage sont indiqués en caractères gras.

- ✓ Les **ports** (*ports*) définissent les canaux de communication entre le modèle et le monde extérieur.

les signaux d'entrée (resp. de sortie) sont définis par le **mode in** (resp. par le **mode out**).

la manière dont les signaux sont traités dans le modèle.

les types de signaux transitant par ces canaux ainsi que les directions de transition.

Dans l'exemple, tous les signaux sont du type prédéfini bit

prendre la valeur '0' ou '1'

**Rq:**

Le signal associé à un port

d'entrée ne peut qu'être lu dans le modèle.

de sortie ne peut que recevoir une valeur dans le modèle.

## Architecture

Une architecture décrit le contenu d'une entité VHDL à trois styles d'architecture qui peuvent être combinée dans le corps d'une architecture

- Comportemental
- Flot de données
- Structurel

Le même circuit peut être décrit en utilisant n'importe lequel des trois styles

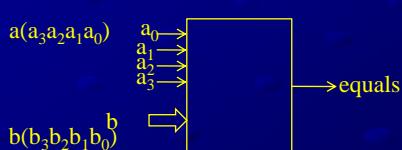
**Rq.** Le même circuit peut être décrit en utilisant n'importe lequel des trois styles

## Description comportementale

Une description comportementale fournit un algorithme qui modélise le fonctionnement du circuit  
Utilisation des algorithmes séquentielles

Une déclaration de processus contient un algorithme

- Elle commence par une étiquette (optionnel), le mot clé “process” et une liste des signaux actifs (*sensitivity list*)
- La liste des signaux actifs indique quels signaux provoqueront l'exécution du processus



```

library ieee;
use ieee.std_logic_1164.all;
entity eqcomp4 is port(
    a, b:  in std_logic_vector(3 downto 0)
    equals: out std_logic);
end eqcomp4;

architecture behavioral of eqcomp4 is
begin
    comp: process (a, b)
    begin
        if a = b then
            equals <= '1';
        else
            equals <= '0';
        end if;
    end process comp;
end behavioral;

```

Initialisation: a=0; b=0;

Prog C	$a=1;$	$\Rightarrow$	$b=6;$	
Prog VHDL	$a \leq 1;$	$\Rightarrow$	$b=5;$	

S1 $\leq a$  and  $b$ ;      S2 $\leq a$  or  $b$ ;  
 S2 $\leq a$  or  $b$ ;      S1 $\leq a$  and  $b$ ;

## La particularité du HDL (VHDL) : la notion de concurrence

### Process

Un PROCESS peut être sensible à une liste de signaux ou à un WAIT

Les PROCESS contiennent seulement des déclarations séquentielles qui sont exécutées dans l'ordre spécifié.

**Syntaxe**

```
[label]: PROCESS [(sensitive_signal_name {,sensitive_signal_name})]
  [constant_declarations]
  [variable_declarations]
BEGIN
  [sequential_statements]
END PROCESS [label];
```

**Exemple**

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY nor2 IS
  PORT (a,b : IN std_logic:=‘0’;
        qn : OUT std_logic);
END nor2;

ARCHITECTURE proc_behv OF nor2 IS
BEGIN
  ex1: PROCESS(a,b) -- ce PROCESS est décrit avec une liste de sensibilité
  BEGIN
    qn <= a NOR b;
  END PROCESS ex1;

  ex2: PROCESS -- ce PROCESS n'a pas de liste de sensibilité
  BEGIN
    qn <= a NOR b;
    WAIT ON a,b; -- WAIT permet d'attendre un événement sur a ou b
  END PROCESS ex2;
END proc_behv;
```

ENSIAS Rabat

Z. ALAOUI ISMAILI

24

**Exemple**

```

SENSE_PROC: PROCESS (CLK)
begin
    if CLK'event and CLK='1' then
        Q2 <= D2;
    END IF;
END PROCESS SENSE_PROC;

WAIT_PROC: PROCESS
begin
    WAIT UNTIL CLK'event and CLK='1';
    Q1 <= D1;
END PROCESS WAIT_PROC;

```

**Description flot de données**

Utilisation d'équations concurantes

- ✓ Une description flot de données spécifie comment la donnée est transférée de signal à signal sans utiliser d'affectations séquentielles (comme dans la description comportementale)
- ✓ Cette description ne nécessite pas de déclaration de processus
- ✓ Toutes les déclarations s'exécutent en même temps ("concurrent signal assignment")

```

architecture dataflow of eqcomp4 is
begin
    equals <= '1' when (a = b) else '0';
end dataflow;

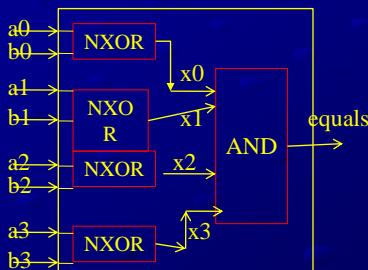
```

## Descriptions structurelles

Création d'un lien avec un modèle

- Les composants sont instanciés et connectés.
- Ils doivent être définis dans un package et compilés dans une bibliothèque
- Les bibliothèques sont attachées par une déclaration

xnor2 and and4 sont dans la bibliothèque work.gatespkg.



```

library ieee;
use ieee.std_logic_1164.all;
entity eqcomp4 is port(
  a, b:  in std_logic_vector(3 downto 0)
  equals:  out std_logic);
end eqcomp4;

use work.gatespkg.all;
architecture struct of eqcomp4 is
  signal x : std_logic_vector(0 to 3)
begin
  u0: xnor2 port map (a(0),b(0),x(0));
  u1: xnor2 port map (a(1),b(1),x(1));
  u2: xnor2 port map (a(2),b(2),x(2));
  u3: xnor2 port map (a(3),b(3),x(3));
  u4: and4 port map (x(0),x(1),x(2),x(3),equals);
end struct;
  
```

### Port Map

PORT MAP est utilisé pour associer les broches d'un COMPONENT avec les signaux du montage.

Deux méthodes d'affectation sont possibles :

- par l'association des noms ;
- par l'association des positions (la plus courante).

#### *Syntaxe*

PORT MAP (pin\_name => signal\_name {, pin\_name => signal\_name});  
 PORT MAP (signal\_name {, signal\_name});

## Component

COMPONENT permet d'écrire d'un programme sous la forme structurelle. En effet, il faut déclarer et définir les composants de la fonction puis dans le corps de l'architecture, il suffit de les connecter en suivant un schéma structurel.

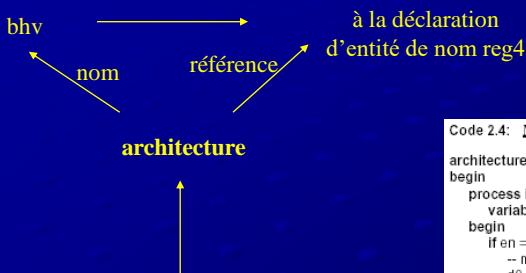
Cette méthode d'écriture permet aussi la description hiérarchique.

Quatre blocs sont à distinguer dans une telle définition :

- recensement de tous les composants du schéma. Ceci se fait par la syntaxe **COMPONENT ... END COMPONENT ;**
- affectation des liaisons du schéma aux broches des composants. On utilise **PORT MAP**. Cette phase correspond à réaliser la **NETLIST** du schéma ;
- attribution des circuits à leur ENTITY. Cette phase permet de contrôler le modèle comportemental qui va déterminer du fonctionnement du composant. Cette phase est réalisée avec **CONFIGURATION** ;
- définition comportementale de chaque composant associé. Ici on procédera à une simple structure d'écriture basée autour d'**ENTITY** et d'**ARCHITECTURE**.

### **2.5.2. Architecture comportementale**

Le Code 2.4 donne le corps d'architecture définissant un modèle comportemental du registre.



Tous les objets déclarés dans l'entité (ici les signaux d'interface) sont visibles dans l'architecture sans déclaration supplémentaire.

```
Code 2.4: Modèle comportemental du registre 4 bits.
architecture bhv of reg4 is
begin
process is
variable d0_reg, d1_reg, d2_reg, d3_reg: bit;
begin
if en = '1' and clk = '1' then
-- mémorisation des entrées
d0_reg := d0;
d1_reg := d1;
d2_reg := d2;
d3_reg := d3;
end if;
-- modification des signaux de sortie
q0 <= d0_reg after 5 ns;
q1 <= d1_reg after 5 ns;
q2 <= d2_reg after 5 ns;
q3 <= d3_reg after 5 ns;
-- attente du prochain événement sur l'un des signaux d'entrée
wait on d0, d1, d2, d3, en, clk;
end process;
end architecture bhv;
```

**Rq:** Le comportement du registre est défini à l'aide d'un processus unique délimité par les mots-clés **process** et **end process**.

Le processus définit une séquence d'instructions qui:

1. mémorisent conditionnellement l'état des entrées du registre lorsque le signal en et l'horloge clk sont actifs (ici égaux à la valeur '1'),

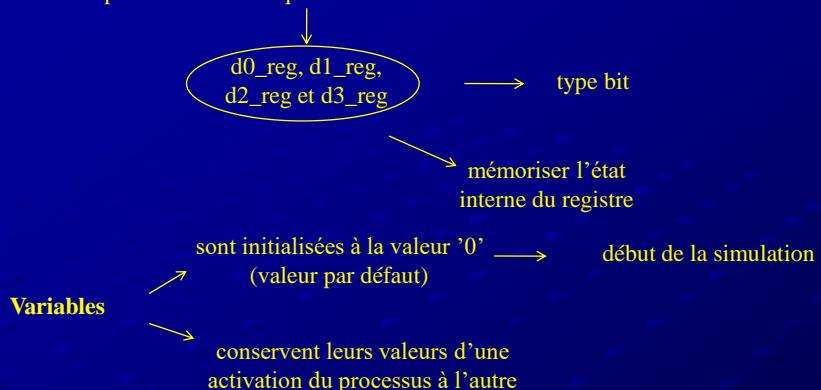
2. modifient les valeurs des signaux de sortie du registre q0, q1, q2 et q3 avec un délai de 5 ns (clause **after**);

→ ce délai modélise le fait que le registre réel ne réagira pas de manière instantanée,

3. mettent le processus en état de veille jusqu'à un prochain événement *sur l'un quelconque* des signaux d'entrée (instruction **wait**).

→ Un événement sur l'un des signaux d'entrée du registre réactivera le processus qui exécutera ses instructions en repartant de la première instruction après le motclé **begin**.

Le processus déclare quatre variables locales :

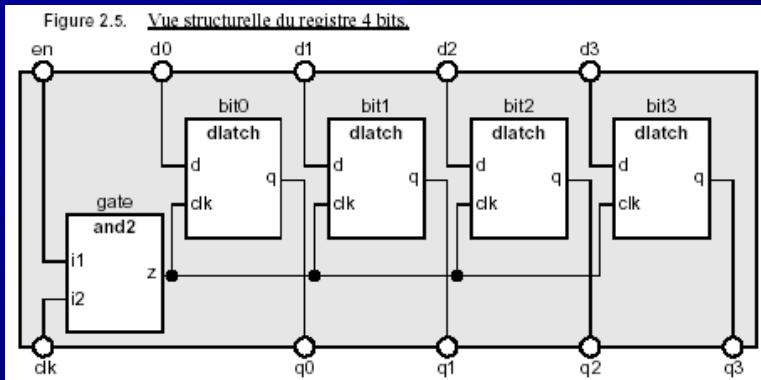


**N.B.** Une variable ne peut être déclarée et utilisée que dans le corps d'un processus.

**Rq:** Le corps d'architecture peut être stocké dans un fichier propre dont le nom suggéré est reg4\_bh.vhd.

### 2.5.3. Architecture structurelle

La Figure 2.5 représente une vue structurelle du registre 4 bits composée de composants (portes) de types latch et AND interconnectés.



Le Code 2.5 donne l'architecture VHDL correspondante.

L'architecture nommée str fait référence à la même déclaration d'entité que le modèle comportemental précédent, ce qui est normal vu que la vue externe du modèle reste la même.

Code 2.5: Modèle structurel du registre 4 bits.

```

architecture str of reg4 is
    -- déclaration des composants utilisés
    component and2 is
        port (i1, i2: in bit; z: out bit);
    end component and2;

    component dlatch is
        port (d, clk: in bit; q: out bit);
    end component dlatch;

begin
    -- instances de latches
    bit0: dlatch port map (d => d0, clk => int_clk, q => q0);
    bit1: dlatch port map (d => d1, clk => int_clk, q => q1);
    bit2: dlatch port map (d => d2, clk => int_clk, q => q2);
    bit3: dlatch port map (d => d3, clk => int_clk, q => q3);
    -- génération de l'horloge interne
    gate: and2 port map (en, clk, int_clk);
end architecture str;

```

L'architecture reflète fidèlement la structure de la Figure 2.5.

```

    graph TD
        C[Une déclaration de composant] --> P1[ses ports formels  
(formal ports)]
        C --> P2[ports effectifs  
(actual ports)]
        P1 --> D1[définis dans sa  
déclaration]
        P2 --> D2[les signaux d'interface  
(signal local du registre)]
        C --> A[Chaque instance de  
composant]
        A --> B[définit les  
associations]
        B --> P1
        B --> P2
    
```

Elle déclare dans un premier temps les composants nécessaires latch et and2 dont un certain nombre d'instances seront "placées" dans le modèle.

Elle ne lie pas encore ses instances à des modèles effectifs.

Une déclaration de composant ne constitue qu'une définition virtuelle de ce dont le modèle a besoin.

Chaque instance de composant → définit les associations

Z. ALAOUI ISMAILI

35

**Rq**

Dans l'exemple

1. les instances de latches      l'instance de la porte and2

association par nom  
(named association)

une association par position  
(positional association)

2. L'étiquette (label) → bit0 → obligatoire pour nommer chaque instance de manière unique.

3. L'ordre de déclaration des instances → peut être quelconque.

ENSIAS Rabat

Z. ALAOUI ISMAILI

36

Rq

Les modèles des composants utilisés  
ne sont pas définis à ce niveau

une étape supplémentaire de ***configuration*** sera nécessaire pour rendre le modèle structurel du registre simulable.

Associer une entité de conception (paire entité/architecture)  
à chaque instance de composant

Comme les noms des composants déclarés et de leurs ports formels peuvent être a priori quelconques

N.B

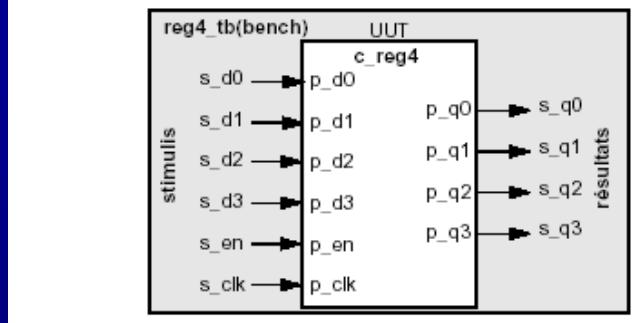
Le corps d'architecture peut être stocké dans un fichier propre dont le nom suggéré est reg4\_str.vhd.

Il peut être compilé et le résultat de la compilation sera placé dans la bibliothèque de travail WORK.

#### 2.5.4. Environnement de test

L'environnement de test (*testbench*) d'un modèle VHDL peut être lui-même décrit comme un modèle VHDL.

Figure 2.6. Environnement de test du registre 4 bits.



La Figure 2.6 illustre la structure d'un tel environnement.

- Le modèle à tester est instancié comme un composant d'un modèle qui ne possède pas d'interactions avec le monde extérieur.
- Le reste du modèle définit les stimuli à appliquer au composant testé et un traitement éventuel des résultats obtenus aux sorties du composant.
- Un environnement de test VHDL est usuellement un mélange de structure et de comportement.

Le Code 2.6 donne le modèle VHDL de l'environnement de test du modèle du registre 4 bits.

<b>entité</b> <b>Analyse du code</b> <b>L'horloge</b> <b>stimulis</b> <b>Remarques</b>	<pre>Code 2.6: Modèle VHDL de l'environnement de test du registre 4 bits.  entity reg4_tb is end entity reg4_tb;  architecture bench of reg4_tb is  component c_reg4 is     port (p_d0, p_d1, p_d2, p_d3, p_en, p_clk: in bit;           p_q0, p_q1, p_q2, p_q3: out bit); end component c_reg4;  signal s_d0, s_d1, s_d2, s_d3, s_en, s_clk, s_q0, s_q1, s_q2, s_q3: bit; begin     -- composant à tester     UUT: c_reg4 port map (p_d0 =&gt; s_d0, p_d1 =&gt; s_d1, p_d2 =&gt; s_d2, p_d3 =&gt; s_d3,                            p_en =&gt; s_en, p_clk =&gt; s_clk,                            p_q0 =&gt; s_q0, p_q1 =&gt; s_q1, p_q2 =&gt; s_q2, p_q3 =&gt; s_q3);     -- stimuli     s_clk &lt;= not s_clk after 20 ns; -- période de 40 ns     process     begin         s_en &lt;= '0';         s_d0 &lt;= '1'; s_d1 &lt;= '1'; s_d2 &lt;= '1'; s_d3 &lt;= '1';         wait for 40 ns;         s_en &lt;= '1';         wait for 40 ns;         s_d0 &lt;= '0'; s_d2 &lt;= '0';         wait for 40 ns;         s_en &lt;= '0'; s_d1 &lt;= '0'; s_d3 &lt;= '0';         wait for 40 ns;         s_en &lt;= '1';         wait; -- stop définitif     end process; end architecture bench;</pre>
ENSIAS Rabat	Z. ALAOUI ISMAILI

La déclaration d'entité se limite à sa plus simple expression puisque le modèle de test est un environnement fermé.



Code 2.6: Modèle VHDL de l'environnement de test du registre 4 bits.

```
entity reg4_tb is
end entity reg4_tb;

architecture bench of reg4_tb is

component c_reg4 is
    port (p_d0, p_d1, p_d2, p_d3, p_en, p_clk: in bit;
          p_q0, p_q1, p_q2, p_q3: out bit);
end component c_reg4;

signal s_d0, s_d1, s_d2, s_d3, s_en, s_clk, s_q0, s_q1, s_q2, s_q3: bit;
begin
    -- composant à tester
    UUT: c_reg4 port map (p_d0 => s_d0, p_d1 => s_d1, p_d2 => s_d2, p_d3 => s_d3,
                           p_en => s_en, p_clk => s_clk,
                           p_q0 => s_q0, p_q1 => s_q1, p_q2 => s_q2, p_q3 => s_q3);
    -- stimuli
    s_clk <= not s_clk after 20 ns; -- période de 40 ns
    process
    begin
        s_en <= '0';
        s_d0 <= '1'; s_d1 <= '1'; s_d2 <= '1'; s_d3 <= '1';
        wait for 40 ns;
        s_en <= '1';
        wait for 40 ns;
        s_d0 <= '0'; s_d2 <= '0';
        wait for 40 ns;
        s_en <= '0'; s_d1 <= '0'; s_d3 <= '0';
        wait for 40 ns;
        s_en <= '1';
        wait; -- stop définitif
    end process;
end architecture bench;
```

➤ Le composant à tester est déclaré puis instancié.

➤ Pour l'instant rien ne lie l'instance de composant avec une entité de conception particulière.

➤ Il s'agit encore de définir une **configuration** pour rendre l'environnement de test simulable.



Code 2.6: Modèle VHDL de l'environnement de test du registre 4 bits.

```
entity reg4_tb is
end entity reg4_tb;

architecture bench of reg4_tb is

component c_reg4 is
    port (p_d0, p_d1, p_d2, p_d3, p_en, p_clk: in bit;
          p_q0, p_q1, p_q2, p_q3: out bit);
end component c_reg4;

signal s_d0, s_d1, s_d2, s_d3, s_en, s_clk, s_q0, s_q1, s_q2, s_q3: bit;
begin
    -- composant à tester
    UUT: c_reg4 port map (p_d0 => s_d0, p_d1 => s_d1, p_d2 => s_d2, p_d3 => s_d3,
                           p_en => s_en, p_clk => s_clk,
                           p_q0 => s_q0, p_q1 => s_q1, p_q2 => s_q2, p_q3 => s_q3);
    -- stimuli
    s_clk <= not s_clk after 20 ns; -- période de 40 ns
    process
    begin
        s_en <= '0';
        s_d0 <= '1'; s_d1 <= '1'; s_d2 <= '1'; s_d3 <= '1';
        wait for 40 ns;
        s_en <= '1';
        wait for 40 ns;
        s_d0 <= '0'; s_d2 <= '0';
        wait for 40 ns;
        s_en <= '0'; s_d1 <= '0'; s_d3 <= '0';
        wait for 40 ns;
        s_en <= '1';
        wait; -- stop définitif
    end process;
end architecture bench;
```

Le comportement du signal d'horloge clk est défini au moyen d'une affectation concurrence de signal

Le signal clk a une valeur initiale par défaut égale à '0', qui est la première valeur du type énuméré prédefini bit.

Le signal d'horloge aura une période de 40 ns.



ENSIAS Rabat

Code 2.6: Modèle VHDL de l'environnement de test du registre 4 bits.

```
entity reg4_tb is
end entity reg4_tb;

architecture bench of reg4_tb is

component c_reg4 is
    port (p_d0, p_d1, p_d2, p_d3, p_en, p_clk: in bit;
          p_q0, p_q1, p_q2, p_q3: out bit);
end component c_reg4;

signal s_d0, s_d1, s_d2, s_d3, s_en, s_clk, s_q0, s_q1, s_q2, s_q3: bit;
begin
    -- composant à tester
    UUT: c_reg4 port map (p_d0 => s_d0, p_d1 => s_d1, p_d2 => s_d2, p_d3 => s_d3,
                           p_en => s_en, p_clk => s_clk,
                           p_q0 => s_q0, p_q1 => s_q1, p_q2 => s_q2, p_q3 => s_q3);

    -- stimuli
    s_clk <= not s_clk after 20 ns; -- période de 40 ns
    process
    begin
        s_en <= '0';
        s_d0 <= '1'; s_d1 <= '1'; s_d2 <= '1'; s_d3 <= '1';
        wait for 40 ns;
        s_en <= '1';
        wait for 40 ns;
        s_d0 <= '0'; s_d2 <= '0';
        wait for 40 ns;
        s_en <= '0'; s_d1 <= '0'; s_d3 <= '0';
        wait for 40 ns;
        s_en <= '1';
        wait; -- stop définitif
    end process;
end architecture bench;
```

Z. ALAOUI ISMAILI

43

Les autres stimuli sont définis dans un processus

Plusieurs séquences de stimuli délimitées par des instructions **wait** sont définies.

Lorsque tous les stimuli ont été appliqués, la dernière instruction **wait** sans paramètre stoppe le processus indéfiniment.



ENSIAS Rabat

Code 2.6: Modèle VHDL de l'environnement de test du registre 4 bits.

```
entity reg4_tb is
end entity reg4_tb;

architecture bench of reg4_tb is

component c_reg4 is
    port (p_d0, p_d1, p_d2, p_d3, p_en, p_clk: in bit;
          p_q0, p_q1, p_q2, p_q3: out bit);
end component c_reg4;

signal s_d0, s_d1, s_d2, s_d3, s_en, s_clk, s_q0, s_q1, s_q2, s_q3: bit;
begin
    -- composant à tester
    UUT: c_reg4 port map (p_d0 => s_d0, p_d1 => s_d1, p_d2 => s_d2, p_d3 => s_d3,
                           p_en => s_en, p_clk => s_clk,
                           p_q0 => s_q0, p_q1 => s_q1, p_q2 => s_q2, p_q3 => s_q3);

    -- stimuli
    s_clk <= not s_clk after 20 ns; -- période de 40 ns
    process
    begin
        s_en <= '0';
        s_d0 <= '1'; s_d1 <= '1'; s_d2 <= '1'; s_d3 <= '1';
        wait for 40 ns;
        s_en <= '1';
        wait for 40 ns;
        s_d0 <= '0'; s_d2 <= '0';
        wait for 40 ns;
        s_en <= '0'; s_d1 <= '0'; s_d3 <= '0';
        wait for 40 ns;
        s_en <= '1';
        wait; -- stop définitif
    end process;
end architecture bench;
```

Z. ALAOUI ISMAILI

44

Le modèle ne contient pas d'instructions particulières pour le traitement des résultats, c'est-à-dire des signaux q0, q1, q2 et q3. On se contente dans ce cas de visualiser les formes d'ondes dans l'environnement de simulation.

Un environnement de test plus élaboré pourrait contenir des composants supplémentaires pour la génération des stimuli et le traitement des signaux de sortie.

Comme la déclaration d'entité est très simple, il est recommandé de maintenir la déclaration d'entité et le corps d'architecture dans le même fichier dont le nom suggéré est reg4\_tb.vhd.

### 2.5.5. Déclaration de configuration

La déclaration de configuration définit la vue de plus haut niveau d'un modèle.

Associer une instance de composant à un couple entité/architecture.

Elle définit les associations globales entre les instances de composants d'une architecture et les entités de conception disponibles dans une bibliothèque (WORK ou autre).

Le Code 2.7 donne la déclaration de configuration de l'environnement de test du registre 4 bits utilisant le modèle comportemental du registre (Code 2.4).

**Code 2.7:** Déclaration de configuration pour le test du modèle comportemental du registre 4 bits.

```
configuration reg4_tb_cfg_bhv of reg4_tb is
    for bench
        for UUT: c_reg4 use entity WORK.reg4(bhv)
            port map (d0 => p_d0, d1 => p_d1, d2 => p_d2, d3 => p_d3,
                      en => p_en, clk => p_clk,
                      q0 => p_q0, q1 => p_q1, q2 => p_q2, q3 => p_q3);
    end for; -- UUT
    end for; -- bench
end configuration reg4_tb_cfg_bhv;
```

On suppose que l'entité de conception reg4(bhv) a déjà été compilée sans erreur dans la bibliothèque WORK.

**Code 2.7:** Déclaration de configuration pour le test du modèle comportemental du registre 4 bits.

```
configuration reg4_tb_cfg_bhv of reg4_tb is
  for bench
    for UUT: c_reg4 use entity WORK.reg4(bhv)
      port map (d0 => p_d0, d1 => p_d1, d2 => p_d2, d3 => p_d3,
                 en => p_en, clk => p_clk,
                 q0 => p_q0, q1 => p_q1, q2 => p_q2, q3 => p_q3);
    end for; -- UUT
  end for; -- bench
end configuration reg4_tb_cfg_bhv;
```

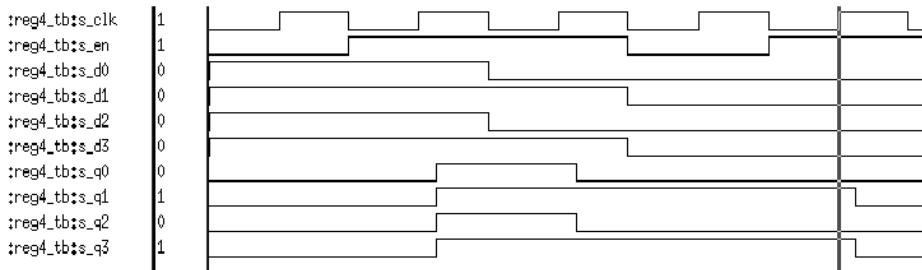
L'association **port map** associe les noms des ports de l'entité (ports formels) aux noms de ports correspondant de la déclaration de composant (ports effectifs)

nécessaire parce que les noms donnés aux ports de la déclaration de composant dans l'environnement de test ne sont pas identiques à ceux de la déclaration d'entité du registre.

Cette association pourrait être omise si les noms étaient identiques.

La Figure 2.7 montre les résultats de la simulation du testbench du Code 2.6 avec la configuration du Code 2.7.

**Figure 2.7.** Résultats de la simulation du testbench.



**N.B.** Le test de l'architecture structurelle du registre requiert la définition d'une autre configuration sans qu'il soit nécessaire de modifier le modèle de l'environnement de test.

Le Code 2.8 donne la nouvelle déclaration de configuration.

**Code 2.8:** Déclaration de configuration pour le test du modèle structurel du registre 4 bits.

```
library GATES;
configuration reg4_tb_cfg_str of reg4_tb is
  for bench
    for UUT: c_reg4 use entity WORK.reg4(str)
      port map (p_d0, p_d1, p_d2, p_d3, p_en, p_clk,
                 p_q0, p_q1, p_q2, p_q3);
    for str
      for all: dlatch use entity GATES.dlatch(bhv); end for;
      for all: and2 use entity GATES.and2(bhv)
        port map (a => i1, b => i2, z => z);
      end for; -- and2
    end for; -- str
  end for; -- UUT
end for; -- bench
end configuration reg4_tb_cfg_str;
```

On suppose que les entités de conception des portes dlatch (Code 2.9) et and2 (Code 2.10) ont été préalablement compilées sans erreur dans la bibliothèque GATES.

**Code 2.9:** Modèle comportemental d'un composant latch.

```
entity dlatch is
  port (d, clk: in bit;
        q : out bit);
end entity dlatch;

architecture bhv of dlatch is
begin
  process is
  begin
    if clk = '1' then
      q <= d after 2 ns;
    end if;
    wait on clk, d;
  end process;
end architecture bhv;
```

**Code 2.10:** Modèle comportemental d'un composant AND2.

```
entity and2 is
  port (a, b: in bit;
        z : out bit);
end entity and2;

architecture bhv of and2 is
begin
  process is
  begin
    z <= a and b after 2 ns;
    wait on a, b;
  end process;
end architecture bhv;
```

### Remarques

La déclaration de configuration comporte dans ce cas un niveau hiérarchique supplémentaire pour associer les instances de portes logiques.

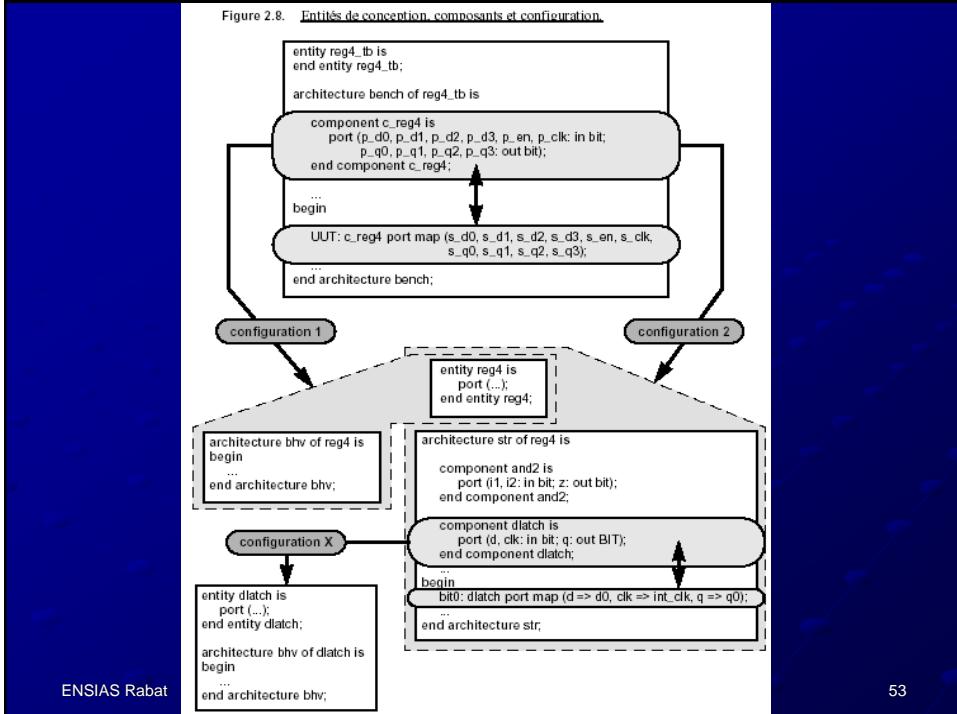
Comme les noms des ports de la déclaration d'entité dlatch sont identiques à ceux de la déclaration du composant du même nom dans l'architecture str, la partie **port map** n'est pas nécessaire.

La déclaration de configuration est stockée dans un fichier dont le nom suggéré est reg4\_tb\_cfg\_str.vhd.

### 2.5.6. Récapitulation

La Figure 2.8 récapitule les liens entre les unités de conception, les composants et la déclaration de configuration.

Figure 2.8. Entités de conception, composants et configuration.



## 2.6. Concurrence et modélisation du temps

## microprocesseur

exhibe un fonctionnement

Un système matériel

essentiellement concurrent

```
graph TD; A["l'unité arithmétique et logique"] --> B["les registres"]; A --> C["les mémoires"]; B --> D["remplissent leur tâches"]; C --> D; D --> E["activation de signaux de contrôle"]; E --> F["décomposées"]; F --> G["Tâches concurrentes"]; G --> H["niveau du matériel"]; H --> I["exécuteront"]; I --> J["indépendamment les unes des autres"]; J --> K["séquencement des opérations requises par le programme assembleur"]
```

Diagram illustrating the decomposition of tasks and their execution by the CPU:

- The CPU (l'unité arithmétique et logique) is composed of registers (les registres) and memories (les mémoires).
- Registers and memories perform their tasks (remplissent leur tâches).
- These tasks lead to the activation of control signals (activation de signaux de contrôle).
- The tasks are decomposed (décomposées) into concurrent tasks (Tâches concurrentes) at the material level (niveau du matériel).
- These tasks will be executed (exécuteront) independently of each other (indépendamment les unes des autres).
- The tasks are sequenced (séquencement des opérations requises par le programme assembleur) at the assembly program level.

Un langage de description de matériel tel VHDL se distingue ainsi des langages de programmation comme Ada ou C par le fait, entre autres, qu'il inclut des instructions pour décrire et simuler des comportements concurrents.

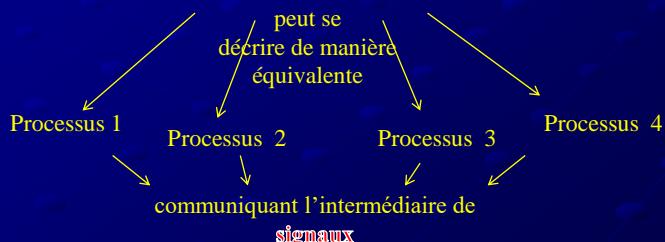
### 2.6.1. Processus

L'instruction concurrente de base en VHDL



*processus (process).*

Tout modèle VHDL



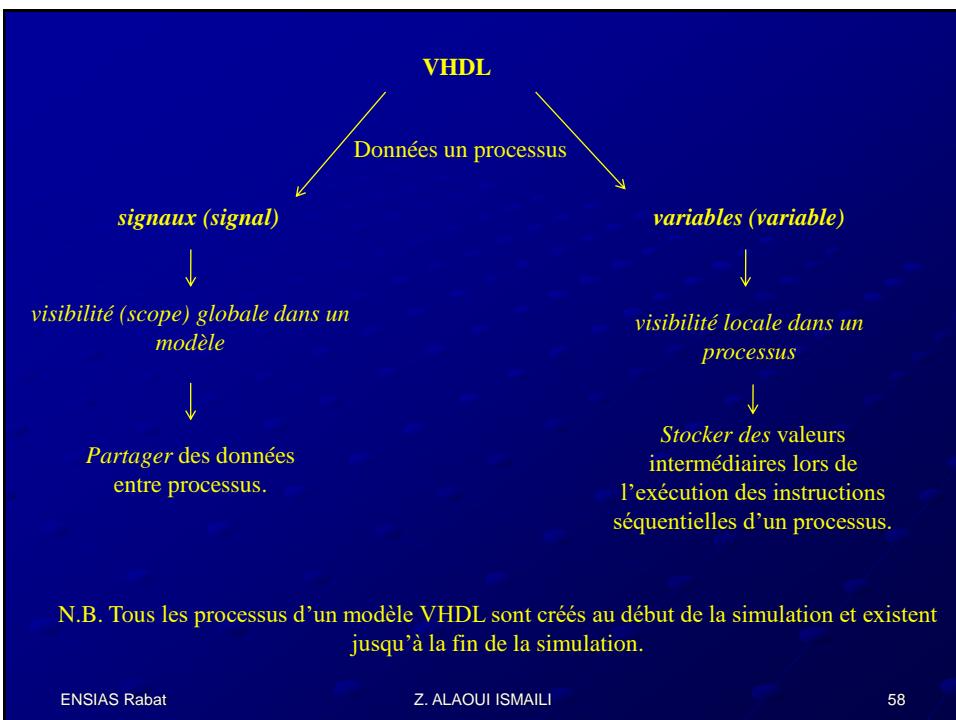
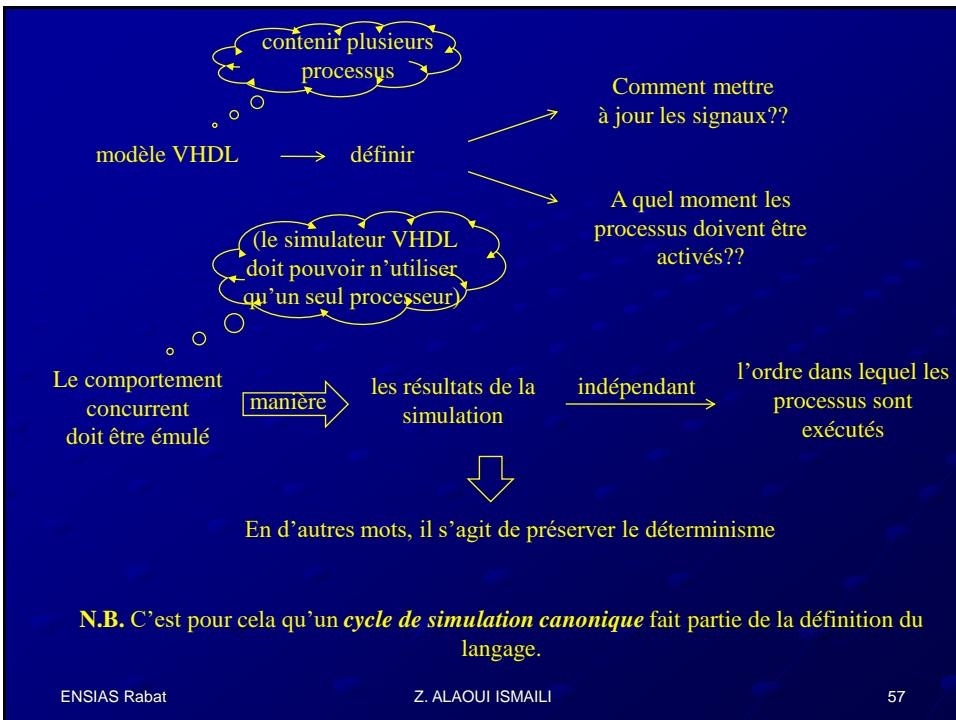
Un processus

Encapsule une séquence d'instructions à exécuter dans l'ordre donné

L'exécution des instructions d'un processus est conditionnée à des *événements* (*event*) sur des signaux

#### *Un signal*

**N.B.** est réputé avoir un événement si sa valeur à un instant donné subit un changement.



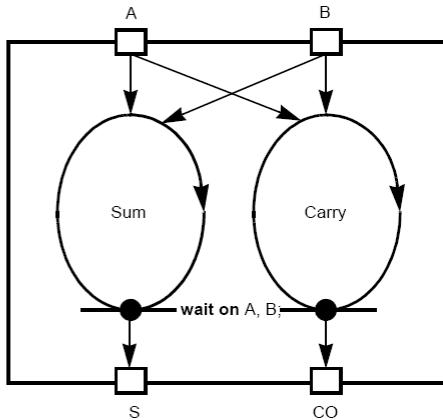
Le Code 2.13 donne l'exemple simple d'un modèle de demi-additionneur 1 bit à deux processus nommés Sum et Carry.

**Code 2.13: Exemple de modèle à deux processus: le demi-additionneur 1 bit.**

```
entity half_adder is
    port (A, B: in bit; S, CO: bit);
end entity half_adder;

architecture bhv of half_adder is
begin
    Sum: process
    begin
        S <= A xor B;
        wait on A, B;
    end process Sum;

    Carry: process
    begin
        CO <= A and B;
        wait on A, B;
    end process Carry;
end architecture bhv;
```



Les deux processus sont sensibles à des événements sur le signal *A ou sur le signal B*, sans priorité particulière.



L'ordre d'exécution des deux processus peut être quelconque sans que cela change les résultats de la simulation.

Une fois l'évaluation des nouvelles valeurs des signaux S et CO effectuée, chaque processus se met en veille et attend le prochain événement sur A ou sur B.

La mise  
en veille est exécutée par  
l'instruction **wait**

Une réactivation future des processus fera reprendre le flot d'instructions séquentielles juste après l'instruction **wait** (dans notre exemple, chaque processus recommencera à s'exécuter après le mot-clé **begin**)

Un processus sans instruction **wait** est ainsi condamné à s'exécuter en boucle infinie sans qu'il soit possible de voir quoique ce soit en simulation. C'est donc une situation à éviter et qui est détectée par le compilateur VHDL.

### 2.6.2. Synchronisation de processus

L'instruction **wait** permet de définir la (les) condition(s) d'activation d'un processus. Elle peut prendre plusieurs formes.

#### 1. La forme:

```
wait on liste-de-signaux;
-- p. ex.
wait on S1, S2, S3;
```

active un processus dès qu'un événement survient sur l'un des signaux de la liste.

La liste des signaux forme une *liste de sensibilité* (*sensitivity list*).

#### 2. La forme:

```
wait until condition;
-- p. ex.
wait until CLK = '1';
wait on CLK until CLK = '1'; -- activation sur le flanc montant du signal CLK
-- forme équivalente
```

active un processus dès que la condition est vraie.

Les signaux impliqués dans la condition forment une liste de sensibilité.

**N.B.** Si la liste est vide, l'instruction **wait** stoppe le processus de manière définitive.

#### 3. La forme:

```
wait for délai;
-- p. ex.
wait for 10 ns;
```

active un processus après un temps de suspension spécifié par le délai.

#### 4. La forme générale:

```
wait on liste-de-signaux until condition for délai;
```

→ est possible



Le processus est activé dès la première condition d'activation (événement sur un signal et condition vraie) ou après le temps écoulé.

**N.B.** Une instruction **wait** seule stoppe un processus de manière définitive.

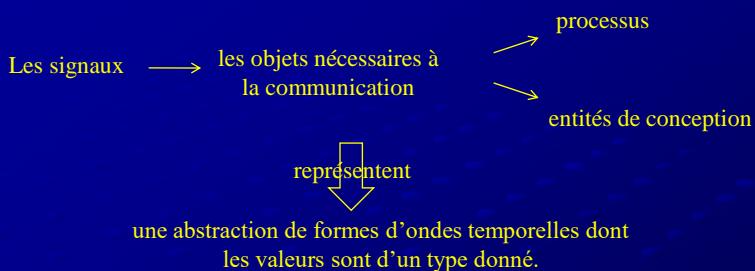
Plusieurs instructions **wait** peuvent exister dans un processus.

Le Code 2.14 donne l'exemple d'un comportement asynchrone contrôlé par deux signaux A et B.

**Code 2.14:** Exemple de processus avec plusieurs instructions wait.

```
process
begin
  wait until A = '1' and B = '1';
  Q <= '1';
  wait until A = '0' and B = '0';
  Q <= '0'
end process;
```

### 2.6.3. Signaux



**Rq.** Un signal possède une structure de donnée complexe appelée un *pilote (driver)* qui stocke la valeur courante du signal et éventuellement ses valeurs futures.

**Code 2.15:** Affectation d'une forme d'onde à un signal.

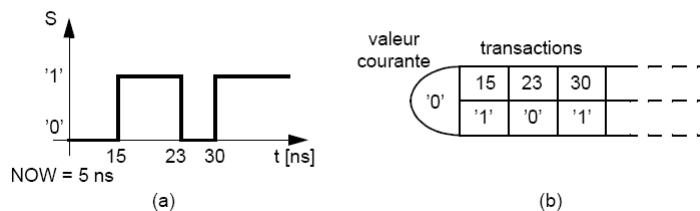
```
signal S: bit;
process
begin
  wait for 5 ns;
  S <= '0', '1' after 10 ns, '0' after 18 ns, '1' after 25 ns;
  wait;
end process;
```

Le Code 2.15 illustre l'affectation d'une forme d'onde à un signal S

La forme d'onde spécifie un certain nombre de ***transactions ordonnées*** dans le temps qui constituent la **forme d'onde projetée (projected waveform)**

Tous les délais sont relatifs au temps auquel l'instruction est exécutée.

Figure 2.9. Forme d'onde résultante de l'affectation du Code 2.15 (a) et pilote du signal S (b).



La Figure 2.9 montre la forme d'onde résultante de l'affectation.

Etant donné l'instruction **wait initiale**, l'affectation se fera au temps **NOW = 5 ns**.

La fonction NOW est prédefinie en VHDL et retourne le temps de simulation courant.

### Remarques:

→ Une variable ne peut exister que dans un contexte séquentiel, elle est affectée immédiatement. Elle n'est pas visible à l'extérieur d'un processus

$x := 1 + 2;$  -- x prend la valeur 3

→ Le signal est le seul objet qui peut être affecté soit de façon concurrente, soit de façon séquentielle selon le contexte. Il est l'agent de communication entre processus.

→ L'affectation du signal est différé à cause de son pilote.

Lors de l'affectation séquentielle

→ le ou les couples valeur\_future: heure\_de\_simulation sont placés dans le pilote.

*La valeur sera (ou dans quelques cas conflictuels, ne sera pas) effectivement passée au signal au moment de la suspension du Process par une instruction WAIT*

*on peut être amené à écrire WAIT FOR 0 ns; pour forcer cette affectation à être immédiate).*

*« C'est le pilote du signal qui est affecté et non le signal lui-même »*

$s \leq 1 + 2$  AFTER 10 ns; -- le pilote de s reçoit le couple de valeurs 3,10

**Illustration 1: différence dans les affectations**

-- à l'heure H = 10 ns , aa prend la valeur 7 , bb prend la valeur 9  
-- entre 0 et 10 ns aa vaut 3 et bb vaut 2

T <sub>aaa</sub>	0	10
P <sub>aaa</sub>	3	1
aaa	3	0
T <sub>bbb</sub>	0	10
P <sub>bbb</sub>	2	11
bbb	2	11

-- à l'heure H = 10 ns , aaa prend la valeur 0 , bbb prend la valeur 11  
-- entre 0 et 10 ns aaa vaut 3 et bbb vaut 2

```

ENTITY varsig IS
END;

ARCHITECTURE exercise OF varsig IS
SIGNAL aa, aaa : INTEGER := 3;
SIGNAL bb, bbb : INTEGER := 2;

BEGIN
  P1: PROCESS
    VARIABLE a: INTEGER := 7;
    VARIABLE b: INTEGER := 6;
  BEGIN
    WAIT FOR 10 ns;
    a := 1; -- a est égal à 1
    b := a + 8; -- b est égal à 9
    a := b - 2; -- a est égal à 7
    aa <= a; -- 7 dans pilote de aa
    bb <= b; -- 9 dans pilote de bb
  END PROCESS;

  P2: PROCESS
    BEGIN
      WAIT FOR 10 ns;
      aaa <= 1; -- 1 dans pilote de aa
      bbb <= aaa + 8; -- 11 dans pilote de b
      aaa <= bbb - 2; -- 0 dans pilote de aaa
    END PROCESS;
  
```

De deux affectations successives d'un même signal, seule la deuxième compte.  
**Le pilote du signal constitue une mémoire associée au signal**

67

**Illustration 2: Mémorisation implicite du signal**

Soit dans un PROCESS la séquence suivante ( sur des type bit)  
WAIT UNTIL h = '1';  
x <= e; -- e est mis dans le pilote de x  
y <= x; -- la valeur actuelle de x est mis dans le pilote de y

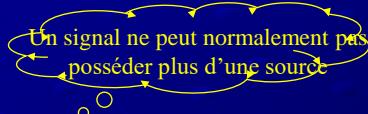
On a réalisé un registre à décalage à 2 étages.  
Alors que la séquence:  
WAIT UNTIL h = '1';  
x := e; -- x prend la valeur e  
y <= x; -- la valeur actuelle de x est mis dans le pilote de y

montre que x n'est qu'un intermédiaire de calcul. En fait, y va recevoir e. C'est une bascule de type D.

N.B.



1. **Un processus** contenant plusieurs instructions d'affectation de signal définit des pilotes (ou des sources) distincts pour chaque signal affecté.



2. L'usage de signaux multi-sources est cependant nécessaire pour modéliser des bus ou des fonctions ET et OU câblées!!!!

→ Ceci est possible en VHDL grâce à un type particulier de signal appelé **signal résolu** (*resolved signal*).



3. Un signal ne prend jamais une nouvelle valeur immédiatement

→ L'affectation définit une transaction qui sera exécutée une fois que le processus est suspendu par une instruction **wait**.

L'exécution d'une transaction induisant un changement de valeur du signal

→ définit un **événement** (*event*) sur le signal.

**N.B.** **Un événement** est toujours le résultat d'une transaction.  
Une transaction ne cause pas forcément un événement

## 2.7. Types et Sous-types

Le VHDL est un langage  
fortement typé



Chaque objet doit être déclaré et appartenir à un type connu (ensemble de valeurs possibles).

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

### 2.7.1 Scalaires

Ils sont ordonnés (=, /=, <, <=, >, >=) et peuvent être restreints « RANGE limite\_basse TO limite\_haute » ou « RANGE limite\_haute DOWNTO limite\_basse »

#### 2.7.1.1 Enumérés

Ensemble de littéraux.

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

#### 2.7.1.2 Entiers

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

#### 2.7.1.3 Flottants

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

#### 2.7.1.4 Physique

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

## 2.7.2 Composites (*Tableaux et enregistrement*)

### 2.7.2.1 Tableaux

Les tableaux (ARRAY) peuvent être à une ou plusieurs dimensions

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

### 2.7.2.2 Enregistrements

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

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

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

**N.B.** On peut accéder à un élément de l'enregistrement en désignant son champ par un point comme dans... date\_de\_naissance.jour ou date\_de\_naissance.mois etc...

→ De manière similaire si un SIGNAL ou une VARIABLE est de type RECORD, il est possible de faire des affectations partielles d'un élément.

### 2.7.3 fichier

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

```
TYPE text IS FILE OF string;
```

#### 2.7.4 Sous-Types

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

SUBTYPE natural IS integer RANGE 0 TO integer'high : *natural* reste compatible avec *integer*  
TYPE nouvel\_entier IS RANGE 0 TO 99: *nouvel\_entier* est un type différent

#### 2.7.5 Conversion de type

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

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

SIGNAL data : donnee_longue;
SIGNAL donnee : donnee_courte;

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

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

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

### 2.8. Structures de contrôle

#### 2.8.1 IF...THEN...ELSIF...ELSE...END IF

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

#### 2.8.2 CASE ... IS...WHEN...

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

## 2.9 ASSERT

L'instruction **ASSERT** est la seule qui permet de générer des messages à l'écran.

**N.B.** Elle n'a donc pas de sens en synthèse (circuit) sinon pendant la phase de mise au point, par contre elle correspond exactement à la démarche du test par assertion.

On vérifie des propriétés du circuit par un ensemble d'affirmation (vraies ou fausses).

ASSERT FALSE REPORT "Ceci n'est qu'un petit bonjour à l'écran";

Ou bien

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

## 2.9 Sous-programmes

Les sous-programmes permettent de regrouper une suite instructions séquentielles pour décrire une seule fois un calcul particulier devant (ou non) se répéter.

**Exemple:** conversion de type, partie de processus, fonction de vérification de timing, fonction de résolution, générateur de stimuli etc...

### Il existe deux types de sous-programmes

#### 2.9.1 Procédures et fonctions

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

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

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

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

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

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

L'utilisation dans une affectation séquentielle ou concrète de signal est possible.

$N \leqslant \text{Convert}(B)$  after 1 ns;

Le passage des paramètres se fait soit par position (ordre prédefini) soit par dénomination ordre quelconque.

Autre exemple:

La fonction de conversion d'un entier en bit\_vector serait :

```
FUNCTION Convert (n:1: NATURAL) RETURN BIT_VECTOR IS
    VARIABLE temp : BIT_VECTOR(1-1 DOWNTO 0);
    VARIABLE valeur : NATURAL := n;
    BEGIN
        FOR i IN temp'RIGHT TO temp'LEFT
        LOOP
            temp(i) := BITVAL( valeur MOD 2 );
            valeur := valeur / 2;
        END LOOP;
        RETURN temp;
    END;
```

- Ecrire une procédure qui permet de détecter le min et le max de quatre valeurs.
- Ecrire une fonction qui ajoute une valeur de 1.0 à une variable a.

### 2.10. Blocs

Le bloc (BLOCK) est élément de base de la structuration de la description. Il permet de définir une hiérarchie.

- Ce que c'est : Une sous unité d'une entitée  
→ C'est un mélange de process et de component : un process avec des ports d'entrée et de sortie
- Rarement utilisés en pratique : alourdi le code. On préconise plutôt l'utilisation des components

## 2.10 Blocs avec Ports

```

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

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

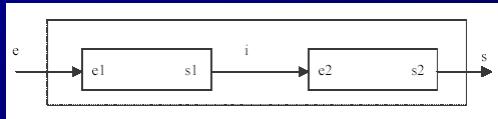
  B2 BLOCK PORT (e2 : IN bit; s2 : OUT bit);
  PORT MAP ( e2 => i, s2 => s);
  BEGIN
    s2 <= e2; -- notations locales
  END BLOCK B2;
END deux_blocs;

```

```

Entity <Nom de l'entité> is
  ...
Begin
  <Nom du bloc> : block
  port (
    <Signaux in et out>
  );
  port map (
    Signal1 => SignalInterne1,
    Signal2 => SignalInterne2
  );
  declarations for <Nom du block>
begin
  <Code utile>
End block <Nom du bloc>;
End Structural;

```



## 2.6 . VHDL supporte trois types de délais

1.  $\xrightarrow{\text{modélise}}$  **délai delta** (*delta delay*)  $\xrightarrow{\text{modélise}}$  **une affectation de signal à délai nul**   
quel que soit l'ordre d'exécution des processus  $\longleftarrow$  **essentiel pour garantir des résultats de simulation corrects**
  
2.  $\xrightarrow{\text{modélise}}$  **délai inertiel** (*inertial delay*)  $\xrightarrow{\text{modélise}}$  **temps de réaction non nul**   
Le mode par défaut lorsque une clause de délai (clause *after*) est spécifiée.  $\longleftarrow$  **Tout événement se passant plus rapidement que le délai inertiel est filtré et est donc annulé.**
  
3.  $\xrightarrow{\text{modélise}}$  **délai de transport** (*transport delay*)  $\xrightarrow{\text{modélise}}$  **réponse fréquentielle infinie**   
**toute impulsion, quelle que soit sa durée, est transmise.**

### Délai inertiel

Le délai inertiel modélise le fait qu'un système matériel réel ne peut pas changer d'état en un temps infiniment court.



En réalité, il faut plutôt appliquer des signaux pendant un temps suffisamment long pour surmonter l'inertie inhérente due au mouvement des électrons.

Le délai inertiel est le mécanisme par défaut en VHDL pour une affectation de signal du type:



$S \leq \text{valeur after délai};$

VHDL-93 le mot-clé **inertial** pour explicitement mentionner **un délai inertiel** et permet de spécifier un temps de rejet plus petit que le délai inertiel:

```
S <= inertial valeur after délai;
S <= reject durée-rejection inertial valeur after délai;
```

La durée du temps de rejet doit être positive et plus petite ou égale au délai inertiel.

Le Code 2.17 donne un extrait de modèle utilisant un délai inertiel pour définir le comportement d'un inverseur

Code 2.17: Exemple d'utilisation du délai inertiel.

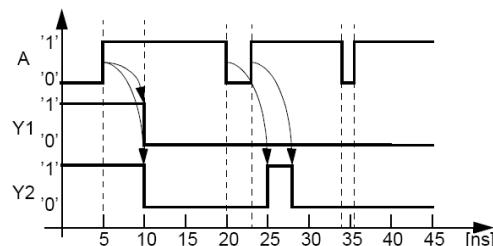
```
inv: process (A) is
begin
  Y1 <= not A after 5 ns; -- ou: Y1 <= inertial not A after 5 ns;
  Y2 <= reject 2 ns inertial not A after 5 ns;
end process inv;
```

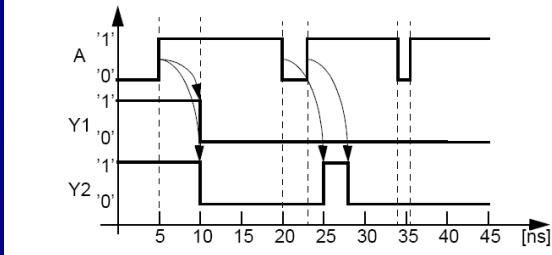
La Figure 2.13 illustre les résultats de l'affectation des signaux Y1 et Y2.

Le changement de valeur du signal A à 5 ns est propagé sur Y1 et Y2 avec un délai de 5 ns.

L'impulsion sur A dès 20 ns et d'une durée de 3 ns n'est pas propagée sur Y1 car sa durée est plus petite que le délai inertiel de 5 ns.

Figure 2.13. Résultat des affectations de signal du Code 2.17.



**Figure 2.13.** Résultat des affectations de signal du Code 2.17.

L'impulsion sur A dès 20 ns et d'une durée de 3 ns propagée sur Y2 car l'affectation spécifie un délai de rejet de 2 ns.

La deuxième impulsion sur A dès 34 ns et d'une durée de 1,5 ns n'est par contre propagée ni sur Y1 ni sur Y2.

### 2.6.8. Délai transport

Le délai transport modélise un comportement idéal pour lequel toute impulsion quelle que soit sa durée, est transmise

Le délai transport est spécifié dans une affectation de signal sous la forme:

`S <= transport valeur after délai;`

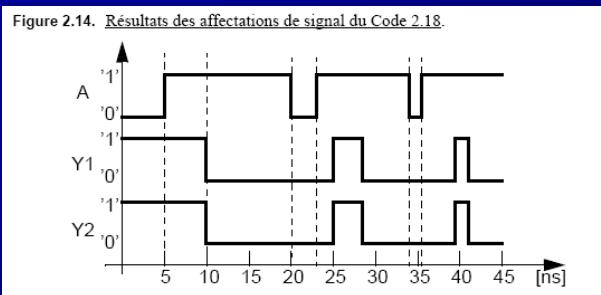
Le Code 2.18 donne un extrait de modèle utilisant un délai transport pour définir le comportement d'un inverseur.

Il montre aussi une formulation équivalente utilisant un délai inertiel et une durée de rejet nulle.

**Code 2.18:** Exemple d'utilisation du délai transport.

```
inv: process (A) is
begin
  Y1 <= transport not A after 5 ns;
  Y2 <= reject 0 ns inertial not A after 5 ns;
end process inv;
```

La Figure 2.14 illustre les résultats de l'affectation des signaux Y1 et Y2.



Les transactions impliquées par des changements de valeurs sur le signal A dans un délai plus petit que 5 ns sont simplement ajoutées au pilote du signal affecté.

## 2.7. La généricité

La généricité peut être vue comme un moyen d'extraire les invariants et les paramètres d'un composant.

Les invariants d'un composant sont codés dans l'architecture.

Tout ce qui peut être amené à évoluer doit être envisagé en tant que "paramètre".

**Exemple :** pour un registre

les invariants sont :

- - le chargement du registre lors du front montant de l'horloge
- - la mise en basse impédance lors de la mise à 1 de la sélection du composant

les paramètres sont :

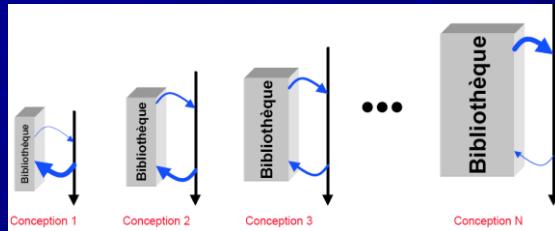
- - la taille des données manipulées
- - le temps de chargement du registre
- - le temps de mise en basse impédance
- - etc

C'est un moyen de transmettre une information à un bloc

- Vu de l'extérieur du bloc, la généricité == paramètre(s)
- Vu de l'intérieur du bloc, paramètres == constantes

Intérêts de la généricité :

- description de composants généraux
- permettre la réutilisation des composants
  - enrichissement progressif de la bibliothèque de travail
  - description de la bibliothèque par des modèles génériques
- assure une plus grande rapidité de développement



Il est important de bien comprendre que plus les composants que vous décrirez seront générique, plus vous serez capable de les réutiliser lors de prochaines conceptions.

**Généralement, c'est l'entité qui est générique**

l'utilisation de générique dans la spécification d'entité

```
entity ..... is
  generic (.....;
            .....;
            .....);
  port (.....;
         .....;
         .....);
end .....;
```

Mais l'architecture doit aussi être générique

⇒ l'utilisation des attributs :

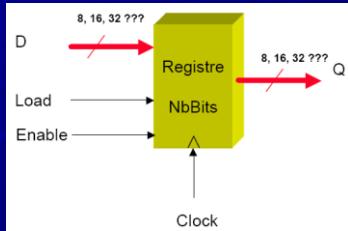
- de tableaux
- de types
- de signaux

others, range, left, event, etc, etc

### Généricité des entités :

Dans de nombreux systèmes électroniques, on est amené à mettre en place des composants ayant les mêmes caractéristiques fonctionnelles, mais travaillant sur des données de tailles différent

Un registre dont le nombre de bascules est générique



Ici nous nous intéressons au registre que nous souhaitons décrire une fois pour toute sans nous soucier de la taille des données (8, 16, 32, ... bits).

Le premier travail consiste à faire la déclaration de l'entité...

Description de l'entité

Remarquons ici que le generic est spécifié avec une valeur par défaut



```

entity registre is
    generic (NbBits : INTEGER := 8);
    port (D : in std_logic_vector (NbBits-1 downto 0);
          Load : in std_logic;
          Enable : in std_logic;
          Clock : in std_logic;
          Q : out std_logic_vector (NbBits-1 downto 0));
    end registre;

```

Cette indication permettra, dans le cas où une instance de cette entité ne préciserait pas la valeur du generic, de s'assurer que l'instance soit créée.

**Remarquons**

Le generic est ensuite utilisé comme un paramètre dans la déclaration des ports d'entrées sorties (à ce niveau le generic est vue comme une constante).

**Utilisation d'une entité générique :**

Soit le système suivant :

des registres sont connectés à des bus dont les tailles sont différentes

L'instanciation d'entité generic doit spécifier les valeurs de chacun des "paramètres" generic.

Imaginons que nous ayons à décrire une système à processeur dans lequel il existe deux bus :

Un premier pour transporter les adresses et une second pour transporter les données

Chacun de ces bus ayant des tailles différentes, il faut pouvoir disposer d'un mécanisme permettant d'instancier 2 entités avec des valeurs de paramètres différentes.

ENSIAS Rabat Z. ALAOUI ISMAILI 93

**Description de l'architecture du système**

Remarquons dans ce listing les instances des 2 registres.

Les paramètres generic sont spécifiés par un mapping.

```

architecture structure of Systeme is
    -- déclaration des composants

    -- déclaration des signaux
    signal BusData : std_logic_vector(15 downto 0);
    signal BusAddress : std_logic_vector(31 downto 0);
    signal SLoadi, SLoadj, SClock, SEnablei, SEnablej : std_logic;

begin
    ...
    Regi : Registre
        generic map (16)
        port map (BusData, SLoadi, SEnablei, SClock, BusData);

    Regj : Registre
        generic map (32)
        port map (BusAddress, SLoadj, SEnablej, SClock, BusAddress);
    ...
end structure;

```

Le composant Regi est bien un composant ayant pour "modèle" l'entité Regitre et la taille des données manipulées est égale à 16 bits.

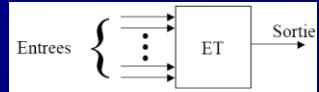
Il faut évidemment s'assurer que les signaux connectés à ce composant ont des tailles cohérentes par rapport à la valeur du generic : BusData est bien un Std\_Logic\_Vector de taille 16.

ENSIAS Rabat Z. ALAOUI ISMAILI 94

### Autre exemple : un ET à N entrées

Ici on s'intéresse à la spécification d'une entité ET logique ayant un nombre d'entrées variable.

La difficulté réside dans le fait que, pour une spécification d'entité, il faut nommer explicitement tous les ports d'entrées sorties.



Or dans ce cas il n'est pas possible de nommer tous ces ports puisque l'on en connaît pas le nombre (cela dépendra des instances).

La solution pour décrire cette entité consiste alors à placer toutes les entrées de l'entité ET logique dans un tableau dont la taille est liée au generic.

```
entity Et_N is
    generic ( N : Natural );
    port ( Entrées : in std_logic_vector ( 1 to N ) ;
           sortie : out std_logic );
end Et_N;
```

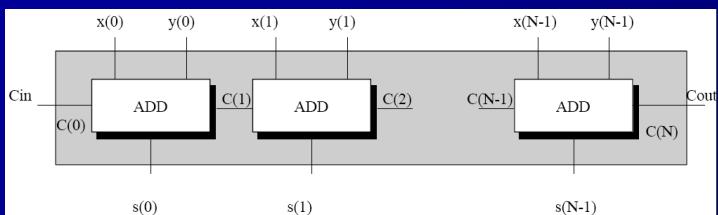
```
architecture comportement of Et_N is
begin
    process
        variable V : std_logic := '1';
    begin
        for i in 1 to N loop
            V := V and Entrées ( i );
        end loop;
        Sortie <= V after 10 ns;
    end process;
end comportement;
```

ENSIAS Rabat

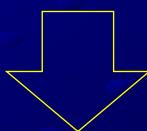
Z. ALAOUI ISMAILI

35

### Autre exemple : additionneur structurelle générifique :



- construit à partir d'un additionneur 1 bits
- assemblage des N additionneurs 1 bits afin de réaliser l'additionneur Complet
- la valeur de N est inconnue avant l'instanciation du composant



ENSIAS Rabat

Z. ALAOUI ISMAILI

96

Lorsque l'on effectue la description d'une architecture d'une entité, on peut souhaiter réaliser cette description de façon structurelle, c'est à dire au travers d'un schéma.

Si l'entité que l'on souhaite spécifier est generic      →      cela peut poser quelques difficultés.

L'additionneur que l'on souhaite rendre generic par rapport à la taille des données manipulées et qui soit décrit à partir d'un schéma de full\_adder.

**PB:** Difficulté lié au fait que lors de l'écriture du code de l'entité, on ne connaît pas le nombre de full\_adder à mettre en place.

⇒ ce nombre dépend des instances, mais pas de l'entité elle-même

Le langage VHDL met à notre disposition une structure permettant de générer automatiquement un schéma à partir d'une boucle

⇒ Il s'agit de la boucle generate

**NB:** Pour que ce type de description puisse être utilisé, il faut que le schéma possède un motif (ou un pattern) qui se répète.

⇒ Dans ce cas, il suffit alors de décrire une seule fois le motif (motif général) et de lier le nombre d'itérations de la boucle au generic.

Il faut en règle générale déclarer des tableaux de signaux et effectuer l'instanciation à partir de l'élément i de ce tableau.

Notons enfin que, comme dans tout cas général, il y a des exceptions.

- ⇒ En effet, les extrémités de la boucle (correspondant aux extrémités de la chaîne des full\_adder), doivent absolument être connectés aux signaux externes que sont Cin et Cout.

Ceci explique les deux affectations concurrentes qui sont réalisées hors de la boucle generate.

On dispose de l'entité Add :

```
entity Add is
  port (
    A, B, Cin : in std_logic;
    S, Cout : out std_logic);
end Add ;
```

L'entité Additionneur générique s'écrit :

```
entity AdditionneurN is
  generic (N : Natural);
  port (
    X, Y : in std_logic_vector (N-1 downto 0);
    Cin : in std_logic;
    S : out std_logic_vector (N-1 downto 0);
    Cout : out std_logic);
end AdditionneurN ;
```

architecture structurelle of AdditionneurN is

```
component Add
  port (
    A, B, Cin : in std_logic;
    S, Cout : out std_logic);
end component;

signal C : std_logic_vector(0 to N);
begin
  for I in 0 to N-1 generate
    Instance : Add
      port map (X(I), Y(I), C(I), S(I), C(I+1));
    end generate;
    C(0) <= Cin;
    Cout <= C(N);
  end structurel;
```

ENSIAS Rabat

Z. ALAOUI ISMAILI

99

### Généricité par les attributs :

Il faut toujours s'assurer que l'architecture supporte la généricité exprimée dans la spécification d'entité.

- ⇒ Ici il est impératif d'utiliser les attributs de façon à être certain que quelque soit la valeur qui sera donnée au generic, le composant aura un fonctionnement correct.

```
architecture comportement of registre is
begin
  process
    variable etat : std_logic_vector(NbBits -1 downto 0);
  begin
    wait on Enable, Clock ;
    if (Clock'event and Clock = '1') then
      if (Load = '1') then
        etat := D;
      end if;
    else
      Q <= etat after 10 ns;
    end if;
  end process;
end comportement;
```

Pas très évolutif  
(si la taille du registre est autre  
que 8 bits, la mise en haute  
impédance ne fonctionne plus)

Cette écriture assure un bon  
fonctionnement quelque soit  
la taille du registre

Q <= (others => 'Z'); after 10 ns;

ENSIAS Rabat

Z. ALAOUI ISMAILI

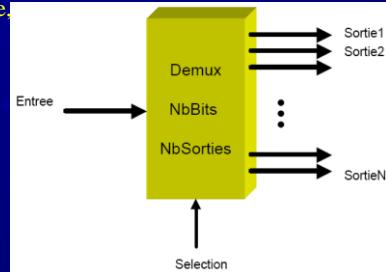
100

### Autre exemple : le démultiplexeur

Ce composant est fonctionnellement immuable,

mais 2 choses, au moins, peuvent varier

il s'agit de la taille des données manipulées, mais aussi le nombre de sorties de ce composant.



→ Il est important de s'efforcer de décrire ce composant une fois pour toute de façon à pouvoir le réutiliser "les yeux fermés" en adaptant simplement ces 2 paramètres.

### Description de l'entité

Dans une première approche, il semblerait que les seuls generic dont nous ayons besoin pour cette entité soient NbBits et NbSorties.

Toutefois, il existe une petite difficulté pour spécifier alors la taille du signal de commande de ce démultiplexeur.

En effet, la taille de la commande est directement liée au nombre de sorties, il s'agit du logarithme à base 2 de NbSorties.



Ne pouvant faire appel à une fonction dans la spécification de l'entité, la seule possibilité qui nous est offerte consiste à mettre en place un generic supplémentaire.

```
entity Demux is
  generic (NbBits : Natural := 8;
           NbCmd : Natural := 3;
           NbSorties : Natural := 16);
  port (
    Entree : in std_logic_vector(NbBits - 1 downto 0);
    Selection : in std_logic_vector(NbCmd - 1 downto 0);
    Sorties : out array (0 to NbSorties - 1) of std_logic_vector(NbBits - 1 downto 0)
  );
end Demux;

architecture comportement of Demux is
begin
  for i in Sorties'range generate
    Sorties(i) <= Entree when Selection = i
      else (others => '0');
  end generate;
end Demux;
```

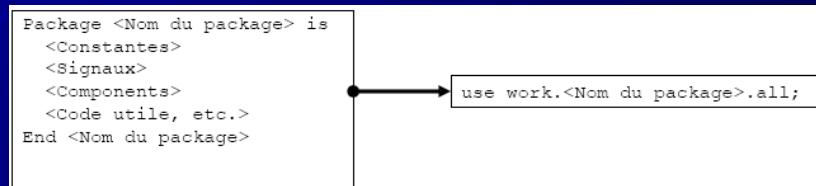
Cette écriture assure une évolution de la taille du démultiplexeur

Ecriture non évolutive :

for i in 0 to 7 generate

## 2.8. Packages

- Les packages sont des librairies de code VHDL précompilées ou non  
→ Analogie avec C/ C++ : #include “package.h”
- Les packages sont généralement déclarés au tout début des fichiers de code source VHDL (avant les entités)
- Employés à plusieurs sauces, par exemple :
  - Modularisation du code
  - Simulation de composantes numériques externes
  - Librairies nécessaires par les simulateurs employés



```

-- Package File Template
-- Purpose: This package defines supplemental types, subtypes,
--           constants, and functions

library IEEE;
use IEEE.STD_LOGIC_1164.all;

package <Package_Name> is

  type <new_type> is
    record
      <type_name>      : std_logic_vector( 7 downto 0);
      <type_name>      : std_logic;
    end record;

  -- Declare constants

  constant <constant_name>          : time := <time_unit> ns;
  constant <constant_name>          : integer := <value>;

  -- Declare functions and procedure

  function <function_name> (signal <signal_name> : in <type_declaration>) return <type_declaration>;
  procedure <procedure_name>      (<type_declaration> <constant_name>          : in <type_declaration>);

end <Package_Name>;

```

```

package body <Package_Name> is
-- Example 1
function <function_name> (signal <signal_name> : in <type_declaration> ) return <type_declaration> is
    variable <variable_name>   :<type_declaration>;
begin
    <variable_name> := <signal_name> xor <signal_name>;
    return <variable_name>;
end <function_name>;

-- Example 2
function <function_name> (signal <signal_name> : in <type_declaration>;
                           signal <signal_name> : in <type_declaration> ) return <type_declaration> is
begin
    if (<signal_name> = '1') then
        return <signal_name>;
    else
        return 'Z';
    end if;
end <function_name>;

-- Procedure Example
procedure <procedure_name> (<type_declaration> <constant_name> : in <type_declaration>) is
begin
    end <procedure_name>;
end <Package_Name>;

```

**Exemple:**

-- Package File Template  
-- Purpose: This package defines supplemental types, subtypes,  
-- constants, and functions

**Package**

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;

-- fichier : composants.vhd
PACKAGE mesportes IS
    COMPONENT et
        PORT(e0,e1 : IN BIT;
             s : OUT BIT);
    END COMPONENT;
    COMPONENT ou
        PORT(e0,e1 : IN BIT;
             s : OUT BIT);
    END COMPONENT;
    COMPONENT inverseur
        PORT(e : IN BIT;
             s : OUT BIT);
    END COMPONENT;
END mesportes;

```

**Package**

```

ENTITY et IS
  PORT(e0,e1 : IN BIT;
        s : OUT BIT);
END et;
ARCHITECTURE aet OF et IS
BEGIN
  s<=e0 AND e1;
END aet;
ENTITY ou IS
  PORT(e0,e1 : IN BIT;
        s : OUT BIT);
END ou;
ARCHITECTURE aou OF ou IS
BEGIN
  s<=e0 OR e1;
END aou;
ENTITY inverseur IS
  PORT(e : IN BIT;
        s : OUT BIT);
END inverseur;
  s<= NOT e;
END ainv;

```

ENSIAS Rabat      Z. ALAOUI ISMAILI      107

**Utilisation:**

```

ENTITY Fct IS
  PORT(e0,e1,e2 : IN BIT;
        s : OUT BIT);
END Fct;
ARCHITECTURE truc OF Fct IS
  SIGNAL e0e1,e2bar : BIT;
BEGIN
  i1:et PORT MAP(e0,e1,e0e1);
  i2:inverseur PORT MAP(e2,e2bar);
  i3:ou PORT MAP(e0e1,e2bar,s);
END truc;

```

ENSIAS Rabat      Z. ALAOUI ISMAILI      108

## 2.9. Attributs

Un attribut est une caractéristique associée à un type ou un objet qui pourra être évalué soit au moment de la compilation soit dynamiquement au cours de la simulation.

Chaque attribut est référencé par son nom consistant en un préfixe, une apostrophe et l'attribut lui-même.



Le préfixe doit être un type, sous-type, tableau, ou bloc.

Exemple montrant comment créer son propre attribut.

### Déclaration

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

### Spécification

Elle associe l'attribut à un ou plusieurs membres d'une «classe d'entité».

```
ATTRIBUTE moitie OF bus_adresse : SIGNAL IS bus_adresse'LENGTH/2;
----- bus_adresse 'moitie -- retourne 8
```

## Attributs prédéfinis

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

- **T'Left** retourne .la limite la plus gauche du type préfixe. T1.Left vaut 1, T2.Left vaut 10, S1.Left vaut 3, S2.Left vaut 4, S3.Left vaut 1.
- **T'Right** retourne la borne droite de définition. T1.Right retourne 10, T2.Right 1, S1.Right 5, S2.Right 3, S3.Right 2.
- **T'High** retourne la plus grande des deux limites du type ou sous-type préfixe. T1.High et T2.High donnent 10, S1.High 5, S2.High 4, S3.High 2.
- **T'low** retourne la plus petite des deux limites du type ou sous-type préfixe. T1.Low et T2.Low retournent 1, S1.low et S2.Low retournent 3 et S3.low retourne 1.

## Conclusions

### Conseils de méthode

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

Pour chaque bloc combinatoire, les instructions concurrentes sont en général plus directement adaptées.

Pour chaque bloc séquentiel, les processus explicites sont en général plus simples à écrire ou à lire. Attention aux mémorisations implicites du signal.

Utiliser au maximum les variables !

En application des  
principes ci-dessus

→ apparaîtra

un ensemble d'instructions  
concurrentes  
→ de processus  
concurrents

➡ la liaison entre ces différents blocs se faisant de manière dynamique (par les signaux) au moment de la simulation ou alors lors de la synthèse du circuit global.

*Une bonne description est une  
description lisible*