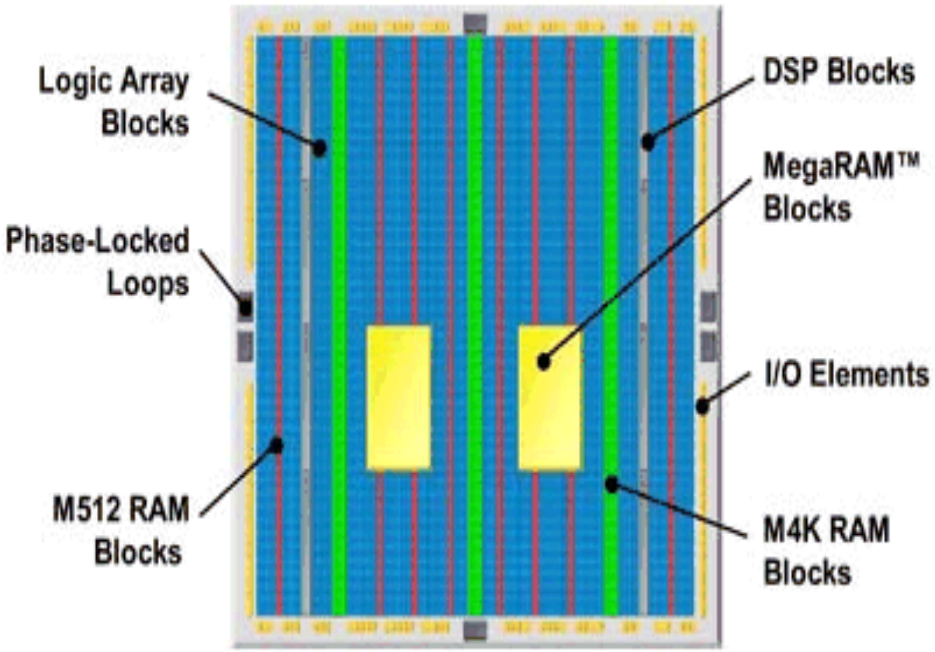
Introduction au langage VHDL Synthétisable



Jean Louis BOIZARD
Maître de conférences IUFM Midi-Pyrénées

Master 1 ISME
Version 1.3

SOMMAIRE

1) GENERALITES

1.1) INTRODUCTION

Le langage VHDL signifie:

VHSIC (Very High Speed Integrated Circuit)
Hardware
Description
Language

C'est un standard de L'IEEE (Institute of Electrical and Electronics Engineers). C'est un langage de description de haut niveau pour la simulation et la synthèse de circuits électroniques numériques.

1.2) HISTORIQUE

- 1980: projet du DoD (Department of Defense) américain de créer un langage standard de description des circuits intégrés à haute vitesse (programme VHSIC).

- 1987: IEEE adopte VHDL comme une norme (norme IEEE 1076).

- 1993: Révision et enrichissement de la norme (la norme devient IEEE 1076-93 puis IEEE-1164)

1.3) TERMINOLOGIE

- **HDL**: le langage de description matérielle est un **logiciel de programmation** permettant la description d'un ensemble ou sous-ensemble matériel.

- **description comportementale**: le circuit est décrit fonctionnellement et ne fait pas appel à des structures électroniques.
(Cette description est utilisable aussi bien pour la simulation que pour la synthèse).

- **description structurelle**: le circuit est décrit par des interconnexions de composants ou primitives de plus bas niveau.

1.4) VHDL et les autres standards

- **VHDL**: " Dites-moi comment vous voulez que votre application se comporte et je vous fournirai les circuits correspondants".

- **ABEL, PALASM, AHDL**: " Dites-moi quels circuits vous voulez et je vous les fournirai".

1.5) SYNTHESE DU VHDL

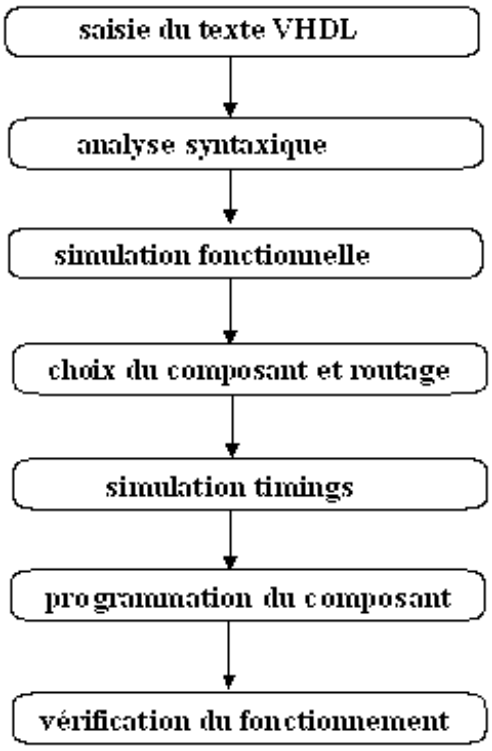
Il ne faut pas oublier que pour l'électronicien la finalité d'une description en langage VHDL est la réalisation d'un ou plusieurs circuits électroniques réalisant les fonctions souhaitées. On distingue alors deux aspects:

- le VHDL synthétisable pour lequel le compilateur sait générer la circuiterie adéquate.
- le VHDL non synthétisable pour lequel le compilateur ne sait pas traduire en circuits les fonctions demandées.

On peut s'interroger quant à l'intérêt d'une description VHDL qui ne serait pas synthétisable. La raison tient en fait à la puissance et au succès de ce langage. Très vite avec l'évolution des besoins et la complexité des circuits électroniques, une méthode rigoureuse et puissante de description des circuits est devenue indispensable. On peut faire le parallèle à ce sujet avec l'évolution du génie logiciel et les apparitions des méthodes de spécification (SADT, SART, OMT, UML...). Un des avantages fondamentaux de VHDL est qu'il peut traduire un cahier des charges complexe et en assurer la simulation: il devient donc un outil de spécification et de simulation. C'est du ressort de l'électronicien de voir comment une description non synthétisable a priori par le compilateur peut être malgré tout traduite en circuits. C'est également une des raisons pour lesquelles le langage VHDL est avant tout un **outil d'électronicien.**

Nota: Les plateformes de développement de FPGA (LATTICE, ALTERA, XILINX, ACTEL...) n'acceptent que du langage VHDL synthétisable.

1.6) ETAPES DE DEVELOPPEMENT



2) BASES DE VHDL

2.1) INTRODUCTION

- Tout comme la saisie de schéma, le langage VHDL fait appel à des **LIBRAIRIES**.
- Il peut effectuer de la **simulation** et de la **synthèse.**
- Il utilise des **mots clés**:

* les lignes de commande sont terminées par " **;** " .
* les "blancs" n'ont pas d'effet (utilisés pour la lisibilité).
* les commentaires sont précédés de " -- ".
* On peut utiliser indifféremment les majuscules ou les minuscules cependant les mots clés sont souvent mis en majuscules.

- les modèles peuvent être décrits en
  * comportemental
  * structurel
  * mixte (les deux précédents).
- Il utilise les concepts de base suivants:

*** L'ENTITE**	**(ENTITY)**
*** L'ARCHITECTURE**	**(ARCHITECTURE)**
*** LA CONFIGURATION**	**(CONFIGURATION)**
*** Le BOÎTIER**	**(PACKAGE)**

2.2) LES LIBRAIRIES

Celles-ci sont déclarées en début du fichier VHDL. Les plus usuelles sont:

- IEEE.std_logic_1164.all;
- IEEE.std_logic_arith.all;
- IEEE.std_logic_signed.all;
- IEEE.std_logic_unsigned.all;

Ainsi pour une application faisant appel à de la logique combinatoire et séquentielle (le plus fréquent) on trouvera en début de fichier:

```
Library IEEE;
        use IEEE.std_logic_1164.all;
```

Cette librairie (issue de la révision de 1993) permet de travailler sur des signaux, variables, entrées-sorties de type BIT, BOOLEEN, ENTIER, REEL et supporte tous les opérateurs associés.

Afin d'optimiser les temps de développement et le "time to market" les constructeurs de circuits programmables proposent des librairies de macro fonctions.

exemple:
La librairie **LPM** (Library of Parametrized Modules) de chez ALTERA propose un ensemble de macro fonctions (additionneurs n bits, multiplieurs...) optimisées pour ses circuits.
Pour l'appeler:

```
library lpm;
        use lpm.lpm_components.all;
```

Par ailleurs les constructeurs proposent sur leurs sites Web un bon nombre de macro fonctions mises en libre service (filtres numériques, arbitre de bus VME, codeurs, cœurs de CPU...).

2.3) L'ENTITE

Elle donne une vue extérieure du modèle VHDL réalisé. Elle s'apparente à un symbole.
Syntaxe:

ENTITY <nom_entité> **IS**
Déclarations génériques;
Déclaration des ports;
END <nom_entité>;

où **<nom_entité>** est le nom donné au modèle VHDL (ex: multiplexeur, additionneur...).

Remarque: dans ALTERA le fichier VHDL doit porter le même nom que l'entité. En conséquence si l'entité s'appelle **multiplexeur** le fichier s'intitulera: **multiplexeur.vhd**

- **Déclarations génériques**:

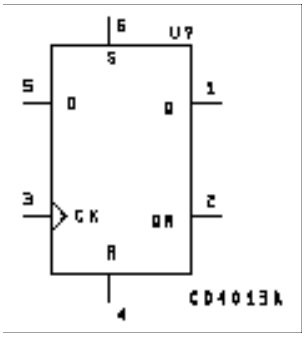
Utilisées pour passer des informations au modèle. C'est le cas lorsqu'on veut avoir un modèle générique réutilisable (exemple: générateur de parité de taille N. La valeur de N est passée en paramètre générique).
Exemple :

```
LIBRARY ieee;
        USE ieee.std_logic_1164.all;
ENTITY parite_taille IS
        generic (taille : integer :=16);
        PORT (op1       : IN  bit_vector (taille-1 downto 0);
              parite    : OUT bit);
END parite_taille;
```

- **Déclaration des ports**:

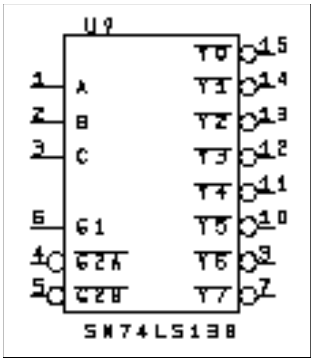
Elle permet de décrire les broches d'entrées et de sorties du modèle.

exemple 1: bascule D CD4013A avec Set et Reset




```
Library IEEE;
        USE IEEE.std_logic_1164.all;
ENTITY  CD4013A IS
        port    ( CK, D, S, R : in std_logic;
                  Q, QN : out std_logic
                  );
END CD4013A ;
```

exemple 2: décodeur type 74LS138




```
Library IEEE;
        USE IEEE.std_logic_1164.all;
ENTITY 74LS138 IS
        port    (A, B, C, G2AN, G2BN, G1 : in std_logic ;
                 YN : out std_logic_vector (7 downto 0)
                 );
```

END 74LS138 ;

Remarque: Souvent le type déclaré pour une entrée ou une sortie dépend de la manière dont on décrit l'architecture.

2.4) L'ARCHITECTURE

- Elle a pour rôle de décrire le **comportement** du composant VHDL. Elle correspond au **schéma électrique interne** du composant et est nécessairement associée à une entité.
- Toutes les instructions qui la composent sont exécutées de manière **concurrente** (ou simultanée).
- Les descriptions peuvent être comportementales (elles ne font pas appel à des structures matérielles) ou structurelles (netlist au niveau composant) ou mixtes (un mélange des deux).

Syntaxe générale:

ARCHITECTURE <nom_arch> OF <nom_entité> IS
Zone de déclaration
BEGIN
Description de la structure logique
END <nom_arch>;

Zone de déclaration:

Elle permet d'identifier des signaux, des constantes ainsi que leurs caractéristiques utilisés par l'architecture.

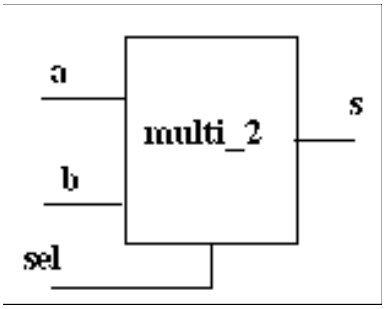
exemple:

signal sig_Ut : integer range 0 to 9 ;

Après avoir créé dans l' **ENTITE** du composant les broches de sorties correspondant à Ut, on crée maintenant dans l'architecture le signal correspondant. Au signal sig_U sera affectée, à l'intérieur du composant, une (des) **équipotentielle(s)**.
En général il est nécessaire de déclarer un signal lorsque celui-ci est utilisé comme une entrée dans une équation logique alors qu'il n'est pas une entrée du circuit (non déclaré dans la zone de **PORT**) : par exemple les sorties d'un compteur seront nécessairement des signaux puisque ils participent à la détermination des états du compteur.

Description de la structure logique:
C'est une suite d'instructions concurrentes.

exemple 1: multiplexeur deux entrées a et b et une sortie s.



Library IEEE;
USE IEEE.std_logic_1164.all;

ENTITY multi_2 IS
Port (a, b, sel : in std_logic;

```
            s : out std_logic
            );
END multi_2 ;

ARCHITECTURE arch_multi_2 OF multi_2 IS
BEGIN
      s <= (a AND NOT sel) OR (b AND sel);     -- représentation structurelle
END arch_multi_2 ;
```

OU ENCORE:

```
      s <=    a WHEN sel = '0' ELSE
              b ;                              -- représentation comportementale
```

OU ENCORE:

```
      WITH sel SELECT
            s <=    a WHEN '0' ,
                    b WHEN '1',
                    '0' WHEN OTHERS;          -- autre représentation
comportementale
```

2.5) MODELISATION DES ARCHITECTURES

2.5.1) Les SIGNAUX

Ils représentent des **équipotentielles enterrées** reliant différentes fonctions. Ils doivent obligatoirement être déclarés à l'intérieur de l'architecture. L'affectation du signal utilise le symbole : **<=**

exemple de déclaration:

SIGNAL sig_Drampe : std _logic_vector (5 downto 0) ;

=> génère 6 signaux : sig_Drampe(5) à sig_Drampe(0)

Initialisation du signal:
Il existe plusieurs possibilités:

- **sig_Drampe <= "100111";** (on utilise les doubles cotes quand il y a plusieurs bits).
- **sig_Drampe <= X "27";** pour de l'hexa.

Pour initialiser un bit (le 3 par ex.):

- **sig_Drampe(3) <= '0' ;** (on utilise les simples cotes quand il n'y a qu'un bit).

Attention !!! Les directives d'initialisation des signaux sont à utiliser avec précaution : ne pas oublier que l'objectif est la synthèse d'un circuit donc une réalisation matérielle. Préférer une phase d'initialisation qui opère sur l'apparition d'un signal réel (signal de reset, RAZ, … par exemple).

2.5.2) LES OPERATEURS

+, -, <, >, <=, >=, /=, *, / sont définis pour des entiers (type integer).
AND, OR, NOT, XOR, & (concaténation) sont définis pour des bits.

2.5.3) AFFECTATION CONDITIONNELLE DES SIGNAUX

Syntaxe:

```
nom_signal  <=  valeur_signal  WHEN condition1 ELSE
                valeur_signal  WHEN condition2 ELSE
                valeur_signal  WHEN condition3 ELSE
                .....
                valeur_signal ;
```

2.5.4) AFFECTATION SELECTIVE DES SIGNAUX

Syntaxe:

```
WITH <expression> SELECT
    nom_signal  <=  <valeur_signal> WHEN condition1,
                    <valeur_signal> WHEN condition2,
                    ....
                    <valeur_signal> WHEN OTHERS ;
```

WHEN OTHERS prend en compte toutes les combinaisons qui n'ont pas été décrites.

2.5.5) LE PROCESS

2.5.5.1) LES VARIABLES

Les variables font partie intégrante d'un process et doivent être déclarées à l'intérieur de celui-ci. Elles correspondent à un stockage temporaire.

Syntaxe:

```
VARIABLE <nom_variable> := <type> := <valeur> ;
```

La variable est mise à jour immédiatement à l'intérieur d'un process alors que le signal est mis à jour à la fin d'un process.

2.5.5.2) SYNTAXE D'UN PROCESS

```
PROCESS (a,b)
BEGIN
    liste des instructions
END PROCESS ;
```

Remarque 1: le process ne s'exécute que si un évènement se produit sur a ou b.
on appelle **(a, b)** la liste de sensibilité du process.

Remarque 2: Une architecture peut avoir plusieurs process. Ceux-ci s'exécutent simultanément et les liens entre process sont effectués par les signaux.

Remarque 3: les instructions à l'intérieur d'un process sont séquentielles.

2.5.5.3) AFFECTATION DES SIGNAUX

Les affectations des signaux peuvent se faire à l'intérieur ou à l'extérieur du process. Elles ne prendront effet qu'à la fin de celui-ci.

2.5.5.4) INSTRUCTIONS SEQUENTIELLES

Le process permet au sein d'une architecture de décrire de **manière séquentielle** le comportement de fonctions. Il permet de faire appel à des instructions puissantes rencontrées dans les langages informatiques (C par exemple).

Instructions utilisables:

- **IF ...THEN...ELSE**
- **FOR** paramètre **IN** intervalle **LOOP**
- **CASE** signal **IS**
  WHEN valeur1 => instructions séquentielles;
  WHEN valeur2 => instructions séquentielles;
  ...
  END CASE;

2.5.6) EXEMPLES

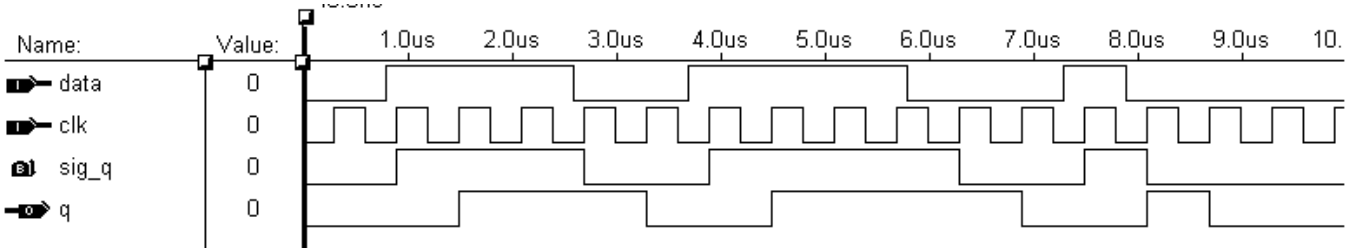
2.5.6.1) INSTRUCTION **IF ..THEN..ELSE** : bascule D CD4013

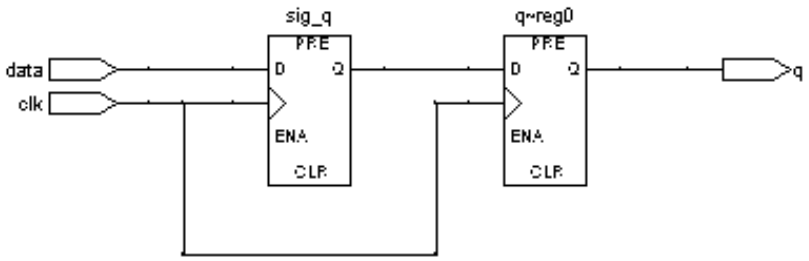
```
library IEEE;
	use IEEE.std_logic_1164.all;

-- description des entités du CD4013

ENTITY CD4013 IS
	port	(ck, d, s, r : in std_logic ;
			q, qn : out std_logic
			);
END CD4013;

-- description de l'architecture

ARCHITECTURE arch_CD4013 of CD4013 IS
SIGNAL sig_q : std_logic ;
SIGNAL sig_qn : std_logic ;
BEGIN
	PROCESS (ck, r, s)
	BEGIN
		IF r= '1' THEN
			sig_q <= '0' ;
		ELSIF s = '1' THEN
			 sig_q <= '1' ;
```

```
            ELSIF ck 'event and ck = '1' THEN
            sig_q <= d ;
            END IF;
        sig_qn <= NOT sig_q ;
        END PROCESS;
        q <= sig_q ;
        qn <= sig_qn ;
END arch_CD4013 ;
```

Bascule cd4013 : MAUVAISE REPRESENTATION

```
ENTITY cd4013b is
        port    (clk, data : in std_logic;
                        q : out std_logic);
end cd4013b;
ARCHITECTURE arch_cd4013b of cd4013b is
        signal sig_q : std_logic;
begin
        process (clk)
        begin
            if clk 'event and clk = '1' then
            sig_q <= data ;          -- un registre utilisé (affectation conditionnée par clk)
            q <= sig_q;     -- un autre registre utilisé  (affectation conditionnée par clk)
            End if ;
        end process;
END arch_cd4013b;
```





2.5.6.2) INSTRUCTION **FOR…LOOP**: Compteur CD 4040

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

-- description des entités du CD4040 (compteur 12 bits)
```

```
ENTITY CD4040 IS
	port	(clock, reset : in std_logic ;
			q : out std_logic_vector (11 downto 0)
			);
END CD4040;

-- description de l'architecture

ARCHITECTURE arch_CD4040 of CD4040 IS
SIGNAL sig_q : std_logic_vector (11 downto 0) ;
BEGIN
	PROCESS (clock, reset)
	VARIABLE var_q : std_logic_vector (11 downto 0) ;
	VARIABLE carry : std_logic ;
	BEGIN
		IF reset = '1' THEN
			sig_q <= "000000000000" ;
		ELSIF clock 'event and clock = '0' THEN
		carry := '1' ;
			FOR i in 0 to 11 LOOP
				var_q (i) := sig_q (i) xor carry ;
				carry := sig_q (i) and carry ;
			END LOOP;
		END IF;
		sig_q <= var_q ;
	END PROCESS;
	q <= sig_q ;
END arch_CD4040 ;
```

autre façon de décrire ce composant:

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

-- description des entités du CD4040 (compteur 12 bits)

ENTITY CD4040v1 IS
	port	(clock, reset : in std_logic ;
			q : out integer range 0 to 4095 -- on assimile la sortie à un entier
			);
END CD4040v1;

-- description de l'architecture

ARCHITECTURE arch_CD4040v1 of CD4040v1 IS
SIGNAL sig_q : integer range 0 to 4095 ;
BEGIN
	PROCESS (clock, reset)
	BEGIN
		IF reset = '1' THEN
			sig_q <= 0 ;
```

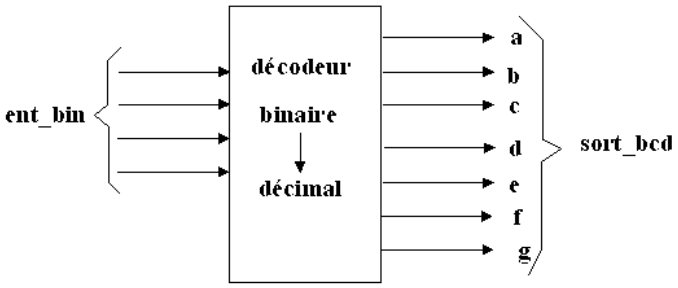
```
        ELSIF clock 'event and clock = '0' THEN
                sig_q  <= sig_q + 1 ;   -- on peut facilement faire une addition sur
        END IF;                         -- des entiers
    END PROCESS;
    q <= sig_q ;
END arch_CD4040v1 ;
```

2.5.6.3. Utilisation d'un paramètre générique:

Le paramètre générique permet de disposer d'un seul modèle pour un type de composant.
Exemple : compteur modulo « modulo »

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

-- description des entités du CD40xx (compteur n bits)

ENTITY CD40xxv2 IS
    generic (modulo : integer := 255) ;
    port    (clock, reset : in std_logic ;
                q : out integer range 0 to modulo
                );
END CD40xxv2;

-- description de l'architecture

ARCHITECTURE arch_CD40xxv2 of CD40xxv2 IS
SIGNAL sig_q : integer range 0 to modulo ;
BEGIN
    PROCESS (clock, reset)
    BEGIN
        IF reset = '1' THEN
            sig_q <= 0 ;
        ELSIF clock 'event and clock = '0' THEN
            sig_q  <= sig_q + 1 ;
        END IF;
    END PROCESS;
    q <= sig_q ;
END arch_CD40xxv2 ;
```

2.5.6.4) Instruction **CASE**: décodeur binaire -7 segments.



```
--      décodeur BCD - 7 segments
--      a       b       c       d       e       f       g
--      0       1       2       3       4       5       6

library IEEE;
        use IEEE.std_logic_1164.all;
--      description des entités du décodeur

ENTITY decodeur IS
        port    (ent_bin : in std_logic_vector (3 downto 0) ;
                        sort_bcd : out std_logic_vector (6 downto 0)
                        );
END decodeur;

--       description de l'architecture (sorties actives à 1)

ARCHITECTURE arch_decodeur of decodeur IS
BEGIN
        PROCESS (ent_bin)
        BEGIN
                CASE ent_bin IS
                        WHEN "0000" =>      sort_bcd <= "0111111";      -- 0
                        WHEN "0001" =>      sort_bcd <= "0000110";      -- 1
                        WHEN "0010" =>      sort_bcd <= "1011011";      -- 2
                        WHEN "0011" =>      sort_bcd <= "1001111";      -- 3
                        WHEN "0100" =>      sort_bcd <= "1100110";      -- 4
                        WHEN "0101" =>      sort_bcd <= "1101101";      -- 5
                        WHEN "0110" =>      sort_bcd <= "1111101";      -- 6
                        WHEN "0111" =>      sort_bcd <= "0000111";      -- 7
                        WHEN "1000" =>      sort_bcd <= "1111111";      -- 8
                        WHEN "1001" =>      sort_bcd <= "1101111";      -- 9
                        WHEN others =>      sort_bcd <= "0000000";      -- éteint
                END CASE;
        END PROCESS;
END arch_decodeur ;
```

2.5.6.5) Additionneur 4 bits

2.5.6.5.1) représentation structurelle

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

-- description des entités de l'additionneur

ENTITY adder_4 IS
        port    (a0,a1,a2,a3,b0,b1,b2,b3 : in std_logic ;
                        s0,s1,s2,s3,s4 : out std_logic
                        );
END adder_4;
```

```
-- description de l'architecture

ARCHITECTURE arch_adder_4 of adder_4 IS
SIGNAL sig_r0 : std_logic ;
SIGNAL sig_r1 : std_logic ;
SIGNAL sig_r2 : std_logic ;
SIGNAL sig_r3 : std_logic ;

BEGIN
        s0 <= a0 XOR b0;
        sig_r0 <= a0 and b0;
        s1 <= sig_r0 xor a1 xor b1;
        sig_r1 <= (a1 and b1) or (sig_r0 and (a1 or b1));
        s2 <= sig_r1 xor a2 xor b2;
        sig_r2 <= (a2 and b2) or (sig_r1 and (a2 or b2));
        s3 <= sig_r2 xor a3 xor b3;
        sig_r3 <= (a3 and b3) or (sig_r2 and (a3 or b3));
        s4 <= sig_r3;

END arch_adder_4 ;
```

2.5.6.5.2) Représentation par un process

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

-- description des entités de l'additionneur 2n entrées

ENTITY adder_n IS
generic (n: integer := 8);
        port    (a,b : in std_logic_vector (n downto 1) ;
                        s : out std_logic_vector (n downto 1);
                        r : out std_logic
                        );
END adder_n;

-- description de l'architecture

ARCHITECTURE arch_adder_n of adder_n IS
SIGNAL sig_s : std_logic_vector (n downto 1) ;
SIGNAL sig_r : std_logic;

BEGIN
        PROCESS (a,b)
                variable var_s : std_logic_vector (n downto 1);
                variable var_r : std_logic ;
        BEGIN
        var_r:='0';
                for i in 1 to n loop
```

```
                        var_s(i):= a(i) xor b(i) xor var_r;
                        var_r := (a(i) and b(i)) or (var_r and (a(i) or b(i)));
                end loop;
        sig_s <= var_s;
        sig_r <= var_r;
        end process;
s <= sig_s;
r <= sig_r;

END arch_adder_n ;
```

2.5.6.5.3) Additionneur n bits: représentation comportementale

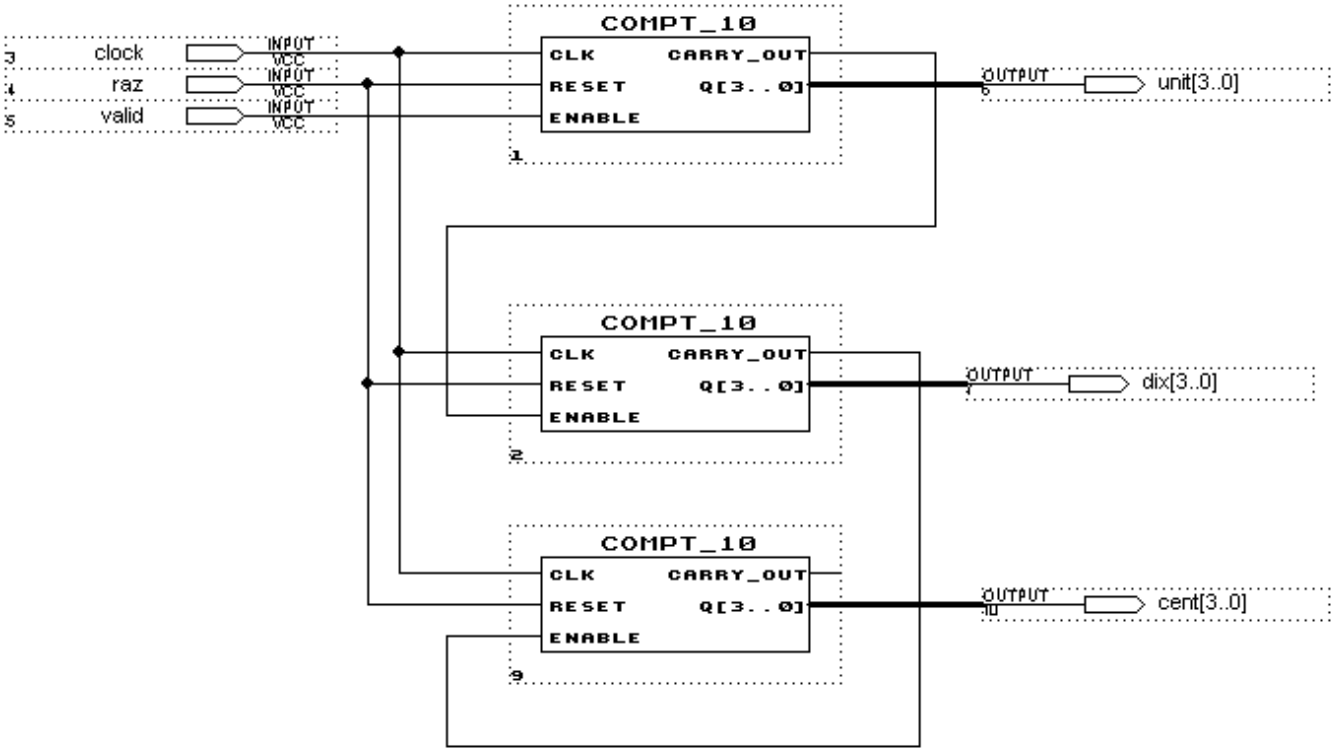
```
library IEEE;
        use IEEE.std_logic_1164.all;
        use IEEE.std_logic_arith.all;

-- description des entités de l'additionneur 2n entrées

ENTITY adder_simple IS
generic (n: integer := 255 );
        port    (a,b : in integer range 0 to n ;
                        s : out integer range 0 to 2*n
                        );
END adder_simple;

-- description de l'architecture

ARCHITECTURE arch_adder_simple of adder_simple IS
BEGIN
s <= a + b ;
END arch_adder_simple
```

3) La conception haut niveau

3.1) La conception hiérarchisée

On l'utilise lorsque le problème posé devient suffisamment complexe. On effectue une décomposition fonctionnelle du problème. On crée une entité pour chacune des fonctions identifiées (si elles sont différentes). On utilise l'instruction **PORT MAP** pour interconnecter les signaux et ports entre eux. Si une fonction est utilisée plusieurs fois (voir ci-dessous) on crée des instances de cette fonction.

Exemple :
Soit à concevoir le circuit de comptage compt_1000 (de 0 à 999) ci-dessous à partir de trois compteurs base 10 :



On décide de créer un compteur base 10 (compt_10) qui sera mis en cascade. L'entité du fichier de description du circuit compt_10 est représentée ci-dessous :

```
ENTITY compt_10 IS
        PORT (clk, reset, enable: in  std_logic ;
                        carry_out : inout std_logic;
                q : inout integer range 0 to 9);
END compt_10;
```

Le fichier VHDL du circuit compt_1000 est le suivant :

```
-- Compteur modulo 1000 :
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY compt_1000 IS
        PORT (clock, raz, valid: IN  std_logic ;
                unit, dix, cent : inout integer range 0 to 9);
```

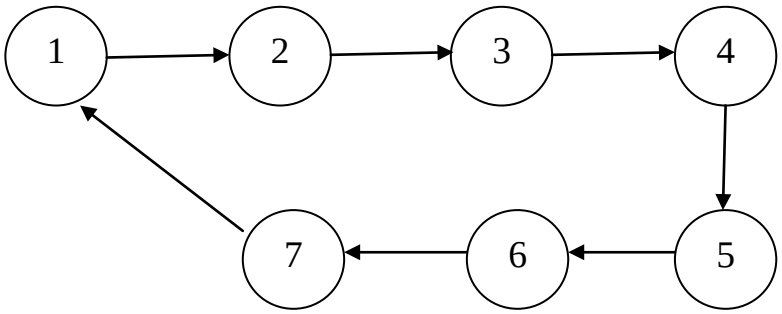
```
END compt_1000;

ARCHITECTURE arch_compt_1000 OF compt_1000 IS
signal cr_dix, cr_cent, cr_mille : std_logic;

        component compt_10
                PORT (clk, reset, enable: IN  std_logic ;
                        carry_out : inout std_logic;
                  q : inout integer range 0 to 9);
        end component;
BEGIN
        U1: compt_10        -- U1 est une instance de compt_10
        port map  (clk => clock, reset=>raz, enable=>valid, carry_out=>cr_dix, q=>unit);

        U2: compt_10
        port map (clk => clock, reset=>raz, enable=>cr_dix, carry_out=>cr_cent, q=>dix);

        U3: compt_10
        port map (clk => clock, reset=>raz, enable=>cr_cent, carry_out=>cr_mille, q=>cent);

END arch_compt_1000;
```

3.2) Synthèse des machines à états

(nota : **Un seul état actif à tout instant**):

Puisqu'un seul état est actif à tout instant, on peut utiliser une variable ou un signal de type **vector** ou **integer**. Si N est le nombre d'états de la machine, alors la taille du type **integer** sera N ou si c'est un type **vector** sa taille sera n de telle sorte que $N \leq 2^n$

Exemple : compteur synchrone modulo 7 (compte de 0 à 6)
Graphe d'état (chaque transition est validée par un front d'horloge non représenté). Un signal de reset remet le circuit dans l'état 1 (non représenté).



Etat	Q2 Q1 Q0
1	000
2	001
3	010
4	011
5	100
6	101
7	110

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY compt_etat_7 IS
        PORT (clk, reset                : IN  std_logic ;
                q0, q1, q2              : OUT std_logic);
END compt_etat_7;
```

```
ARCHITECTURE arch_compt_etat_7 OF compt_etat_7 IS
BEGIN
        process (reset, clk)
        variable etat : integer range 1 to 7;
        begin
        if reset ='1' then
        etat:= 1;
        q0 <= '0'; q1 <= '0'; q2 <= '0';
        elsif clk'event and clk='1' then
        case etat is
        when 1 =>
        etat:=2;
        q0<='1'; q1<='0'; q2<='0';
        when 2 =>
        etat:=3;
        q0<='0'; q1<='1'; q2<='0';
        when 3 =>
        etat:=4;
        q0<='1'; q1<='1'; q2<='0';
        when 4 =>
        etat:=5;
        q0<='0'; q1<='0'; q2<='1';
        when 5 =>
        etat:=6;
        q0<='1'; q1<='0'; q2<='1';
        when 6 =>
        etat:=7;
        q0<='0'; q1<='1'; q2<='1';
        when others =>
        etat :=1;
        q0<='0'; q1<='0'; q2<='0';
        end case;
        end if;
        end process;
END arch_compt_etat_7;
```

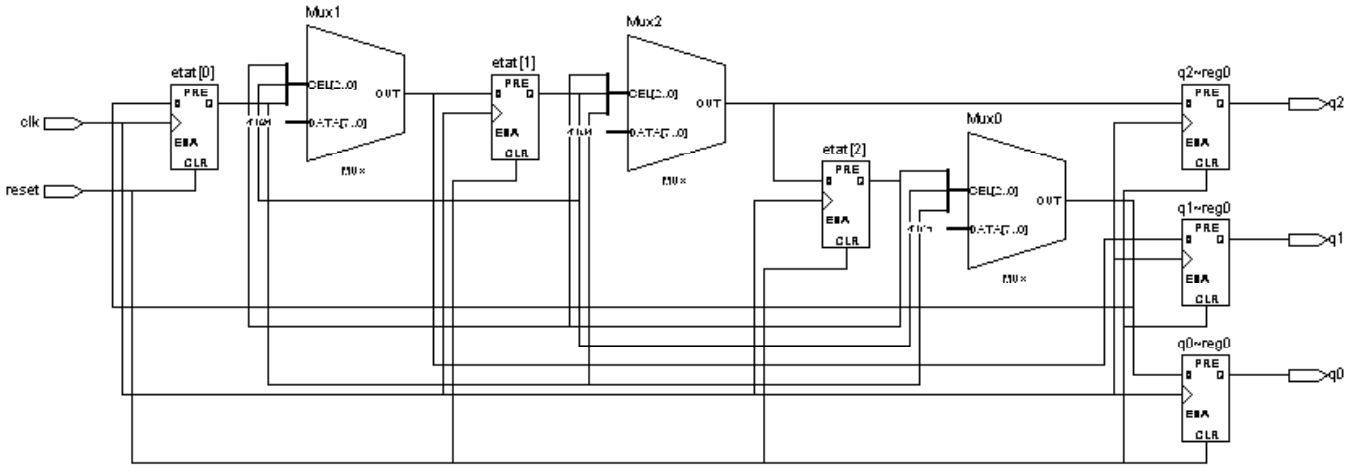
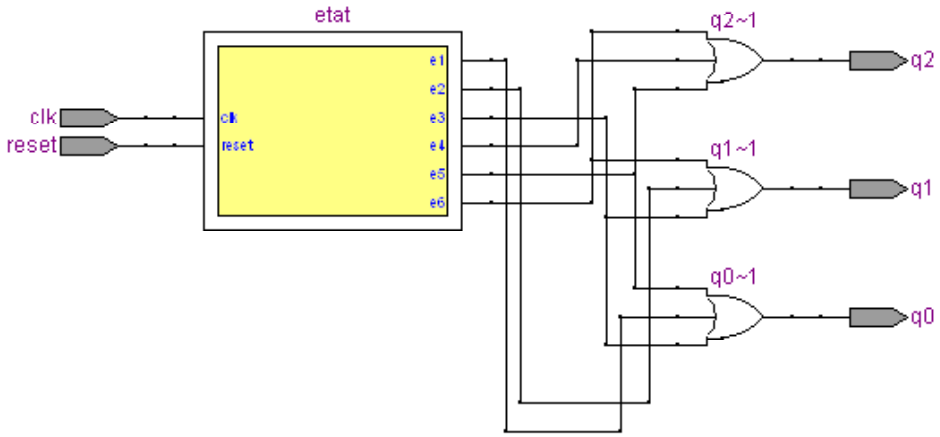


Schéma électrique correspondant

Autre écriture possible :

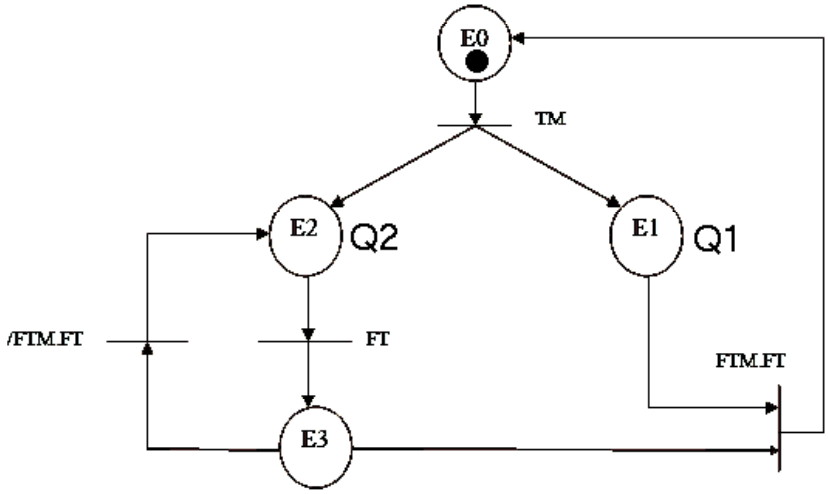
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY compt_etat_7a IS
        PORT (clk, reset            : IN  std_logic ;
                q0, q1, q2          : OUT std_logic);
END compt_etat_7a;

ARCHITECTURE arch_compt_etat_7a OF compt_etat_7a IS
TYPE STATE_TYPE IS (e0,e1,e2,e3,e4,e5,e6);
signal etat : state_type;
BEGIN
        process (reset, clk)
        begin
        if reset ='1' then
        etat<= e0;
        elsif clk'event and clk='1' then
        case etat is
        when e0 =>
        etat<=e1;
        when e1 =>
        etat<=e2;
        when e2 =>
        etat<=e3;
        when e3 =>
        etat<=e4;
        when e4 =>
        etat<=e5;
        when e5 =>
        etat<=e6;
        when others =>
        etat <=e0;
        end case;
        end if;
        end process;
        q0 <= '1' when etat=e1 or etat=e3 or etat=e5 else '0';
        q1 <= '1' when etat=e2 or etat=e3 or etat=e6 else '0';
        q2 <= '1' when etat=e4 or etat=e5 or etat=e6 else '0';
END arch_compt_etat_7a;
```



On peut noter que dans ce mode de représentation, on aura un délai supplémentaire entre l'élaboration des sorties et le front d'horloge (présence des portes OU après les bascules).

3.3) Synthèse des Réseaux de Petri :

Un réseau de Petri pouvant avoir plusieurs places marquées à tout instant, la solution la plus simple est d'associer un signal à chaque place. Dans ce cas les signaux seront nécessairement de type bit.



<u>Représentation VHDL du réseau ci-dessus :</u>
(Nota : pour ne pas surcharger le réseau, les signaux de reset et clk n'apparaissent pas. « reset » remet le réseau dans sa configuration initiale : place E0 marquée).

```
ENTITY petri_1 IS
        PORT (reset, clk, tm, ftm, ft : IN  std_logic;
                        q1, q2 : out std_logic
                        );
END petri_1;

ARCHITECTURE arch_petri_1 OF petri_1 IS
signal  e0, e1, e2, e3 : std_logic ;
Begin
        process (clk, reset)
        begin
```

```
		if reset='0' then
		e0 <='1'; e1 <='0'; e2 <='0'; e3 <='0';
		elsif clk'event and clk ='1' then
			if e0 = '1'  and tm = '1' then
			e0 <='0'; e1 <='1'; e2 <='1';
			end if;
			if e2 = '1'  and ft = '1' then
			e3 <='1'; e2 <='0';
			end if;
			if (e1 and e3 and ftm and ft) ='1' then
			e0 <='1'; e1 <='0'; e3 <='0';
			end if;
			if (e3 and ft and not ftm) = '1' then
			e3 <='0'; e2 <='1';
			end if;
		end if;
	end process;
		q1 <= e1;
		q2 <= e2;
END arch_petri_1;
```