

Jean-luc.dekeyser@lifl.fr  
Version 2013

## LANGAGES DE DESCRIPTION D'ARCHITECTURES

### Pourquoi utiliser des Langages HDL?

- Pour accélérer la conception de circuits (raisons économiques)
- Pour permettre la conception de circuits très complexes (milliards de portes logiques)
- Pour pouvoir représenter les systèmes numériques selon différents axes d'abstraction
- Pour utiliser et/ou améliorer le code pour des systèmes futures

Le VHDL: Qu'est-ce que c'est, et à quoi cela sert-il ?

- VHDL: VHSIC Hardware Description Language
- VHSIC: Very High Speed Integrated Circuit (projet de grande envergure du DoD (Department of Defense) Américain, mis en place dans les années '80)
- Principe de base: Définir un langage de description de matériel qui puisse être utilisé pour simuler du matériel numérique
- Extension: Utilisation du même langage pour la synthèse automatique de circuits

### Langages de référence

- VHDL
  - Origine : DoD américain
  - Syntaxe ADA
  - Meilleure abstraction
- Verilog
  - Origine : industrie
  - Syntaxe proche du C
  - Plus simple d'apprentissage, plus concret
- Saisie de schémas
  - Outil dépendant

### VHDL: Est-ce le seul HDL?

- Il existe plusieurs autres langages de description de matériel, entre autres:
  - Verilog (Très populaire aux États-Unis, utilisé aussi en Europe, au Japon et au Canada)
  - UDL/1 (Utilisé à un certain moment au Japon)
  - Estérel (langage académique – Français)
  - HardwareC (langage académique – Stanford)
- Verilog est plus simple que le VHDL, mais est un peu moins utilisé

### Soft/hard

- Software Programming Language
  - Compilation en langage machine
  - Exécution séquentielle
- Hardware Description Language
  - Synthèse en « portes et câbles »
  - Exécution concurrente
  - Exécution séquentielle en simulation (Test benches)

## Savoir-faire

- Pièges
  - Exécution concurrente
  - « Optimisations »
  - Synthèse (guidage, asynchronisme...)
  - Problèmes hardware (Routage, clock skew...)
- Remèdes
  - Simulation
  - Rapports de synthèse
  - Visualisation à l'oscilloscope, analyseur logique

## utilité du VHDL

- langage standard et international
- permet la description sous forme texte d'un système numérique
  - pour implantation dans un FPGA
  - pour implantation dans un ASIC (circuit intégré spécifique)
  - pour simulation avec niveau d'abstraction plus ou moins poussé (niveau système ou niveau porte logique)
- langage indépendant
  - de l'environnement utilisé (Quartus, ISE...)
  - de la cible finale (EPLD Altera, FPGA Xilinx, ASIC, etc...)

# VHDL

- intérêt et principes du VHDL
- structure et syntaxe du langage
- exemples

## flot de conception en VHDL

1. description sous forme texte

```
...
x <= a AND b;
...
```

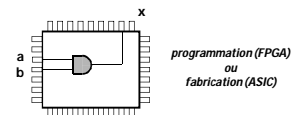
↓ compilation

2. synthèse logique



↓ placement / routage

3. implantation physique



## points abordés

- Présentation
  - intérêt du VHDL
  - flot de conception
  - niveaux de description
  - différence avec un langage de programmation
- Syntaxe
  - structure d'une description VHDL
  - description d'un système combinatoire
  - description d'un système séquentiel (bascules, compteurs, machines d'état)

## VHDL

### Very High Speed Integrated Circuit Hardware Description Language

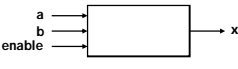
Lancé dans les années 80

#### Objectifs:

- répondre à une complexité croissante des systèmes numériques
- rester indépendant de la technologie utilisée (analogie: un programme est écrit en C sans se soucier de l'ordinateur sur lequel tournera l'application...)

### différents niveaux de description possibles

- haut niveau / comportemental**  
*On définit la réponse des sorties aux entrées, sans se préoccuper de ce qu'il faut pour le réaliser*



```

...
IF enable = '1' THEN
  IF (a=1) OR (b=1) THEN
    x <= '1';
  ELSE '0';
  END IF;
ELSE '0';
END IF;
...

```

- bas niveau / structurel**  
*La description est proche du circuit électrique, on descend au niveau des portes logiques*

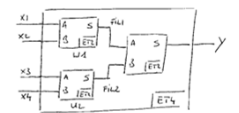
```

...
x <= (a OR b) AND enable;
...

```

### VHDL: concepts de base

- Les questions à se poser
  - On identifie les fonctions et on les dessine *sur papier*
  - On repère et nomme les entrées de chaque blocs (on évite d'utiliser les mêmes noms)
  - On répertorie les signaux INTERNES (mot clé SIGNAL)
  - Le bloc est-il combinatoire ou séquentiel?
    - Si séquentiel alors description avec le mot clé PROCESS + instructions autorisées
  - Le bloc est-il utilisé plusieurs fois
    - Si oui il vaut mieux créer un composant (entity+ architecture)
    - Sinon le bloc est synthétiser par les lignes de codes directement



### différents niveaux de description possibles (suite)

description de haut niveau

↓

- permet la simulation du comportement dans les toutes premières phases de conception
- permet l'échange d'informations sous forme standardisée entre différentes équipes de conception
- ne permet pas forcément de synthétiser la logique correspondante

### Structure d'un fichier VHDL

```

library ieee;
use ieee.std_logic_1164.all;

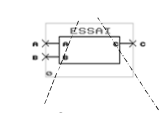
entity essai is
  port (a,b : in std_logic;
        c : out integer);
end essai;

architecture archi1 of essai is
  Signal s1, s2 : std_logic;
begin
  .
end archi1;

```

Permet d'accéder à d'autres descriptions définies dans des diverses bibliothèques.  
 Rem. : Plusieurs couples entity/architecture sont possibles au sein d'un même fichier. Les clauses library et use ne sont pas globales. Leur effet s'arrête dès la déclaration d'une nouvelle entity. D'office il y a Standard et Work

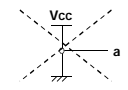
Description de l'interface



Description du comportement de l'entité. Pour une même entité plusieurs architectures sont possibles.

### ⓘ Le VHDL n'est pas un langage de programmation

le VHDL décrit la structure matérielle (figée) d'un système numérique


programmation:	description en VHDL:
<p>les instructions sont exécutées de façon <b>séquentielle</b> (les unes après les autres)</p> <pre> ... a = 1; ... a = 0; ... </pre> <p><b>correct:</b>            d'abord a prend la valeur 1, puis a prend la valeur 0</p>	<p>les fonctions sont réalisées de façon <b>concurrente</b> (tout en parallèle)</p> <pre> ... a &lt;= '1'; ... a &lt;= '0'; ... </pre> <p><b>incorrect:</b>            (car contradictoire)</p> 

### ENTITY

1<sup>ère</sup> étape: définir un **symbole** pour le composant à décrire

↳ boîte noire + ports d'entrée/sortie

exemple: additionneur 1 bit



```

ENTITY adder IS
  PORT ( A,B,Cin : IN BIT;
        Sum,Cout : OUT
        BIT );
END adder;

```

remarques:

- le caractère de fin de ligne est ;
- pas de distinction majuscule / minuscule

### PORT

PORT ( nom1, nom2 : mode type );

direction des données par rapport au composant

IN (port lu par le composant)  
 OUT (port écrit par le composant)  
 BUFFER (flot bidirectionnel mais 1 seule source à la fois)  
 INOUT (flot bidirectionnel mais plusieurs sources possibles - exemple d'un bus)  
 LINKAGE (flot inconnu)

BIT  
 BIT\_VECTOR  
 BOOLEAN  
 INTEGER, REAL  
 TIME

remarques:

- le séparateur entre les noms est ,
- après -- tout est pris comme commentaire jusqu'à la fin de la ligne

### ARCHITECTURE (suite)

autre description équivalente:

```

ENTITY adder IS
  PORT ( A, B, Cin : IN BIT ;
        Sum, Cout : OUT BIT );
END adder;
ARCHITECTURE arch2 OF adder IS
  BEGIN
    Sum <= '1' WHEN (A='1' AND B='0' AND Cin='0') OR (etc...) ELSE '0';
    Cout <= '1' WHEN (etc...) ELSE '0';
  END arch2;
  
```

arch1: description structurelle (proche du schéma électrique)  
 arch2: description comportementale (le schéma ne se déduit pas directement)

### PORT (suite)

exemples:

- PORT ( d, ck : IN BIT ); -- 2 bits en entrée
- PORT ( q : OUT BIT\_VECTOR(3 DOWNTO 0) ); -- 1 bus 4 bits en sortie
- PORT ( q : OUT BIT\_VECTOR(0 TO 3) ); -- idem ci-dessus
- PORT ( q : OUT INTEGER RANGE 0 TO 15 ); -- 1 bus 16 bits en sortie

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
PORT ( d, ck : IN STD_LOGIC;
      q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) );
  
```

-- même chose  
 -- que  
 -- ci-dessus

### Les paramètres génériques

- Ils sont déclarés dans l'entity
- Ils permettent de paramétrer des composants
- une valeur par défaut peut être définie (":=")
- l'instanciation se fait grâce à "generic map(...)"

```

entity mux is
  generic (width : integer := 8);
  port (sel : in std_logic;
        a, b : in std_logic_vector (width-1 downto 0);
        c : out std_logic_vector (width-1 downto 0));
end mux;

architecture behav of mux is
  begin
    c <= a when sel='0' else b;
  end behav;
  
```

```

comp1 : mux generic map ( 4 )
port map(sel=>seldata, a=> dataa, b=> datab, c=>data);
comp2 : mux port map (sel=>seladr, a=>adrA, b=> adrB, c=>adr);
  
```

### ARCHITECTURE

- décrit le fonctionnement du composant
- similaire au schéma du composant
- l'ARCHITECTURE possède son nom propre, ≠ nom de l'ENTITY

```

ENTITY adder IS
  PORT ( A, B, Cin : IN BIT ;
        Sum, Cout : OUT BIT );
END adder;
ARCHITECTURE arch1 OF adder IS
  BEGIN
    Sum <= A XOR B XOR Cin;
    Cout <= (A AND B) OR (Cin AND (A XOR B));
  END arch1;
  
```

### Opérateurs

principaux opérateurs :

```


AND
OR
NOT
XOR
NAND
NOR
& (concaténation)
+ (addition)
- (soustraction)
/= (différent)
* (multiplication)
/ (division - limité à des puissances de 2 pour ALTERA)
  
```

### ARCHITECTURE (suite)

pour de la logique combinatoire, il faut prévoir TOUTES les lignes de la table de vérité, sinon le compilateur VHDL génère une fonction de mémorisation (logique séquentielle)

```

ENTITY composant IS
  PORT ( in1, in2 : IN BIT ;
        out1, out2 : OUT BIT );
END composant;
ARCHITECTURE demo OF composant IS
  BEGIN
    s1 <= in1 WHEN (in2=1) ELSE 0';
    s2 <= in1 WHEN (in2=1) ;
  END demo;
  
```



avec ELSE  
sans ELSE

### PROCESS (IF ... ELSE)

PROCESS: suite d'instructions que le compilateur doit lire en totalité pour générer la logique correspondante

```

ENTITY mux4_1 IS
  PORT (e3, e2, e1, e0 : IN BIT;
        c : IN INTEGER RANGE 0 TO 3;
        s : OUT BIT);
END mux4_1;
ARCHITECTURE archi OF mux4_1 IS
  BEGIN
    PROCESS(c)
      BEGIN
        IF (c = 0) THEN s <= e0;
        ELSIF (c = 1) THEN s <= e1;
        ELSIF (c = 2) THEN s <= e2;
        ELSIF (c = 3) THEN s <= e3;
        END IF;
      END PROCESS;
    END archi;
  
```

sans PROCESS  
avec PROCESS

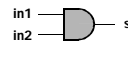
1 fonction logique  
1 autre fonction logique

### ARCHITECTURE (suite)

logique synthétisée:

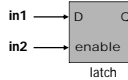
```

out1 <= in1 WHEN (in2=1) ELSE 0';
  
```

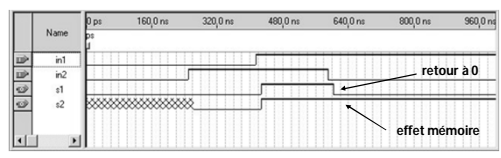


```

out2 <= in1 WHEN (in2=1) ;
  
```



latch

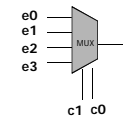


### PROCESS (CASE)

```

ENTITY mux4_1 IS
  PORT (e3, e2, e1, e0 : IN BIT;
        c : IN INTEGER RANGE 0 TO 3;
        s : OUT BIT);
END mux4_1;
ARCHITECTURE archi2 OF mux4_1 IS
  BEGIN
    PROCESS(c)
      BEGIN
        CASE c IS
          WHEN 0 => s <= e0;
          WHEN 1 => s <= e1;
          WHEN 2 => s <= e2;
          WHEN 3 => s <= e3;
        END CASE;
      END PROCESS;
    END archi2;
  
```

même exemple que précédemment



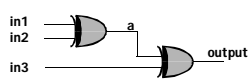
liste des signaux dont un changement d'état influe sur les signaux calculés dans le PROCESS (liste de sensibilité)

### SIGNAL

SIGNAL : nœud ou bus interne permettant un calcul intermédiaire

```

ENTITY xor3 IS
  PORT ( in1, in2, in3 : IN BIT ;
        output : OUT BIT );
END xor3;
ARCHITECTURE archi OF xor3 IS
  SIGNAL a : BIT;
  BEGIN
    a <= in1 XOR in2;
    out <= in3 XOR a ;
  END archi;
  
```



### PROCESS (IF ... THEN / CASE)

- remarques concernant les tests (avec IF ... THEN ou CASE):
  - il faut lister tous les cas possibles, sinon une fonction de mémorisation est créée
  - il est possible d'utiliser la syntaxe **WHEN OTHERS => ...**
  - on peut introduire un IF au sein d'un CASE ou inversement
  - on peut tester des INTEGER (exemple: IF c = 3 -- base 10  
IF c = 16#13# -- base 16)  
des BIT (exemple: IF s = '1')
  - des BUS (exemples: IF s = «1100» -- base 2  
IF s = X«2C» -- hexa)
- remarques concernant les PROCESS
  - il est possible de définir plusieurs PROCESS dans une même architecture: ils sont alors indépendants et concurrents (chacun donne lieu à une logique indépendante des autres)

### PROCESS (IF ... THEN / CASE)

exercices:

- décrire un décodeur 7 segments

x3  
x2  
x1  
x0

DECODEUR  
7 SEGMENTS

a  
b  
c  
d  
e  
f  
g

- décrire un comparateur de 2 mots de 4 bits (utiliser un PROCESS + IF...)

P 3  
Q 3

P > Q

s

s = 1 dans les cas suivants:

- P(3)=1 et Q(3)=0
- [P(3)=Q(3)] et [P(2)=1 et Q(2)=0]
- [P(3)=Q(3)] et [P(2)=Q(2)] et [P(1)=1 et Q(1)=0]
- [P(3)=Q(3)] et [P(2)=Q(2)] et [P(1)=Q(1)] et [P(0)=1 et Q(0)=0]

### BOUCLE, variable de boucle

- sert à effectuer une description itérative (simplifie l'écriture)
- ne correspond à rien de physique

exemple n°1:

e3  
e2  
e1  
e0

reseau\_OR

s3 = e3 OR e2  
s2 = e2 OR e1  
s1 = e1 OR e0  
s0 = e0

### VARIABLE

Une variable

- sert à effectuer un calcul intermédiaire
- permet au compilateur de générer la logique correspondante
- ne correspond à rien de physique

exemple:

e 3

compte\_1

s 2

la sortie s indique en binaire le nombre de bits d'entrée à 1

### BOUCLE, variable de boucle (suite)

```

ENTITY reseau_OR IS
  PORT (e : IN BIT_VECTOR(3 DOWNTO 0);
        s : OUT BIT_VECTOR(3 DOWNTO 0));
END reseau_OR ;

ARCHITECTURE iteratif OF reseau_OR IS
BEGIN
  PROCESS(e)
  BEGIN
    s(0) <= e(0);
    FOR i IN 1 TO 3 LOOP
      s(i) <= e(i) OR e(i-1);
    END LOOP;
  END PROCESS;
END iteratif ;
    
```

### VARIABLE (suite)

```

ENTITY compte_1 IS
  PORT (e : IN BIT_VECTOR(2 DOWNTO 0);
        s : OUT INTEGER RANGE 0 TO 3);
END compte_1;

ARCHITECTURE combinatoire OF compte_1 IS
BEGIN
  PROCESS(e)
  VARIABLE nombre_1 : INTEGER;
  BEGIN
    nombre_1 := 0;
    IF e(2) = '1' THEN nombre_1 := nombre_1 + 1;
    END IF;
    IF e(1) = '1' THEN nombre_1 := nombre_1 + 1;
    END IF;
    IF e(0) = '1' THEN nombre_1 := nombre_1 + 1;
    END IF;
    s <= nombre_1;
  END PROCESS;
END combinatoire ;
    
```

Ici, l'utilisation de la variable revient à écrire plus simplement la table de vérité du système (travail effectué par le compilateur).

Syntaxe du langage 33

### Schémas ou VHDL?

remarque générale: Après compilation d'un système décrit en VHDL, on ne connaît généralement pas le schéma électrique final et les fonctions physiquement utilisées.

<u>description sous forme graphique</u>	<u>description en VHDL</u>
<ul style="list-style-type: none"> <li>■ schéma physique maîtrisé</li> <li>■ proche de la structure matérielle</li> <li>■ vite fastidieux pour des systèmes complexes</li> <li>■ compatibilité non garantie entre différents éditeurs graphiques</li> </ul>	<ul style="list-style-type: none"> <li>■ souple pour des systèmes complexes</li> <li>■ description indépendante de la technologie (réutilisable en cas de changement)</li> <li>■ schéma physique final inconnu (dépend du compilateur)</li> </ul>

C'est l'outil qui choisit!

illustration avec le comparateur vu précédemment:

description graphique (portes XOR et AND):



description VHDL:

### Les Testbenchs

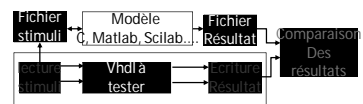
- Pour générer des stimuli et valider le design vhdl.
- Possibilité (nécessité) d'utiliser des ressources VHDL non synthétisable, description uniquement fonctionnelle



Vérification manuelle sur chronogramme:  
- fastidieux, voire impossible si design complexe  
- taux de couverture ?



Vérification automatique:  
- Efficace mais  
- validité du modèle comportementale?  
- Vitesse ?



Vérification automatique:  
- Très efficace

### Génération d'un chronogramme

- la durée s'exprime avec un type physique : fs, ps, ns, us, ms
- `signal <= valeur after durée absolue, valeur after durée absolue, ... ;`

```
H <= "00", "01" after 10 NS, "10" after 20 NS;
clk <= not clk after 50 ns; -- génération d'une horloge à 10 MHz,
attention initialiser clk à la déclaration
```

label: wait on liste\_signal until condition for durée; -- dans des process

- wait on : attente sur événements
- wait until : attente de conditions (s'il a eu un événement sinon la condition n'est pas testée)
- wait for : attente pour un certains temps

```
-- méthode simple mais lourde
constant Period: TIME := 20 NS;
stimulus: process
begin
A <= "0000";
B <= "0000";
wait for Period;
A <= "1111";
wait for Period;
B <= "1111";
wait for Period;
wait;
end process;
```