Université de Boumerdès

Automatique

Module: FPGA & VHDL

Chapitre 1: VHDL

Dr. Belkacem Samia

Plan

1. Introduction

Introduction

Qu'est ce que ça veut dire ?

Vhsic **H**ardware **D**escription **L**anguage

Vhsic : Very High Speed Integrated Circuit

Langage de description de systèmes matériels

Langage : **de description structurelle et comportementale de la conception des dispositifs matériels en électronique numérique (ASICs, CPLD, FPGA, logique câblée)**

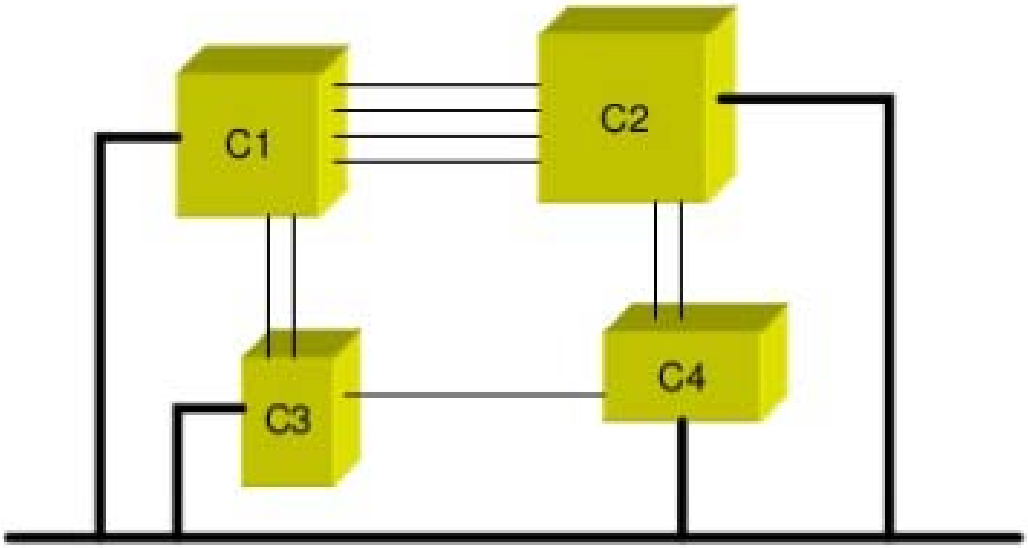
Introduction

Qu’es ce qu’un langage de description?

Sert à décrire un matériel et à créer un circuit

Un HDL, qu'est ce que c'est ?

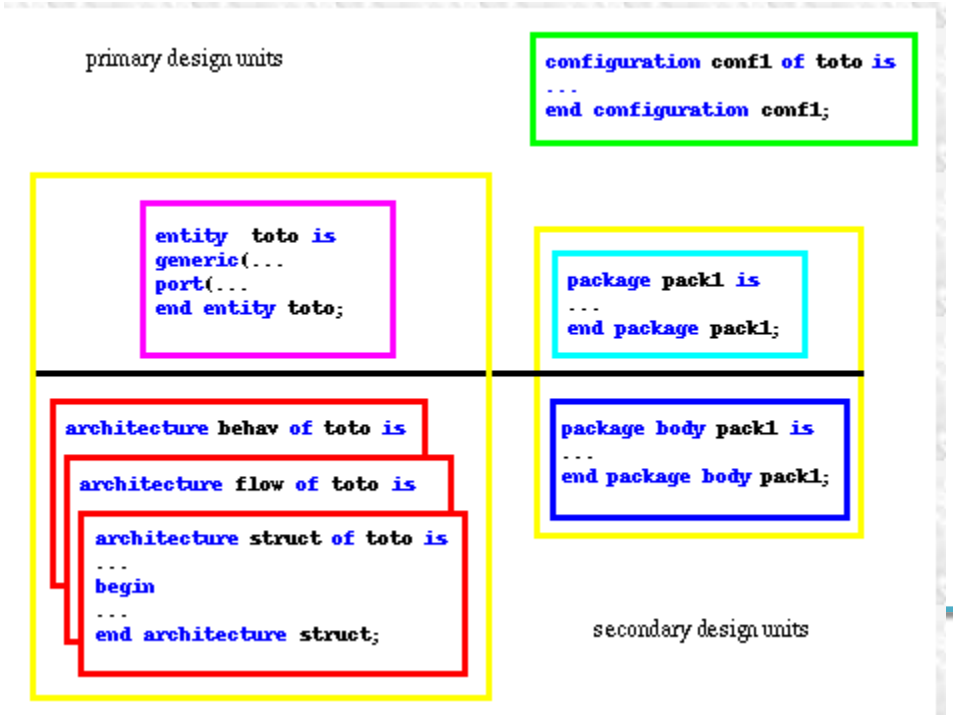
- moyen de décrire un **système matériel** :
  - ✓ qu'est qu'un système matériel ?
    - en général, il s'agit d'un schéma mettant en œuvre :
      - un certain nombre de composants
      - des connexions entre composants



Unités de conception

Les modèles VHDL sont construits à partir de différents types d'unités. Il existe deux classes d'unités de conception VHDL: les unités de conception principales et secondaires.

- Les unités de conception principales sont l'entité, le package et la configuration.
- Les unités de conception secondaires sont la ou les architectures et la déclaration du corps du package.



Unités de conception

- 5 unités de conception disponible en VHDL :

Unités de conception

- Il existe cinq types d'unités de conception en VHDL:
  - Entités
  - Architectures
  - paquetage
  - Corps de paquetage
  - Configurations
- Les entités et les architectures sont les deux seules unités de conception que vous devez avoir dans n'importe quelle conception VHDL
- Les packages et la configuration sont optionnels.

Unités de conception

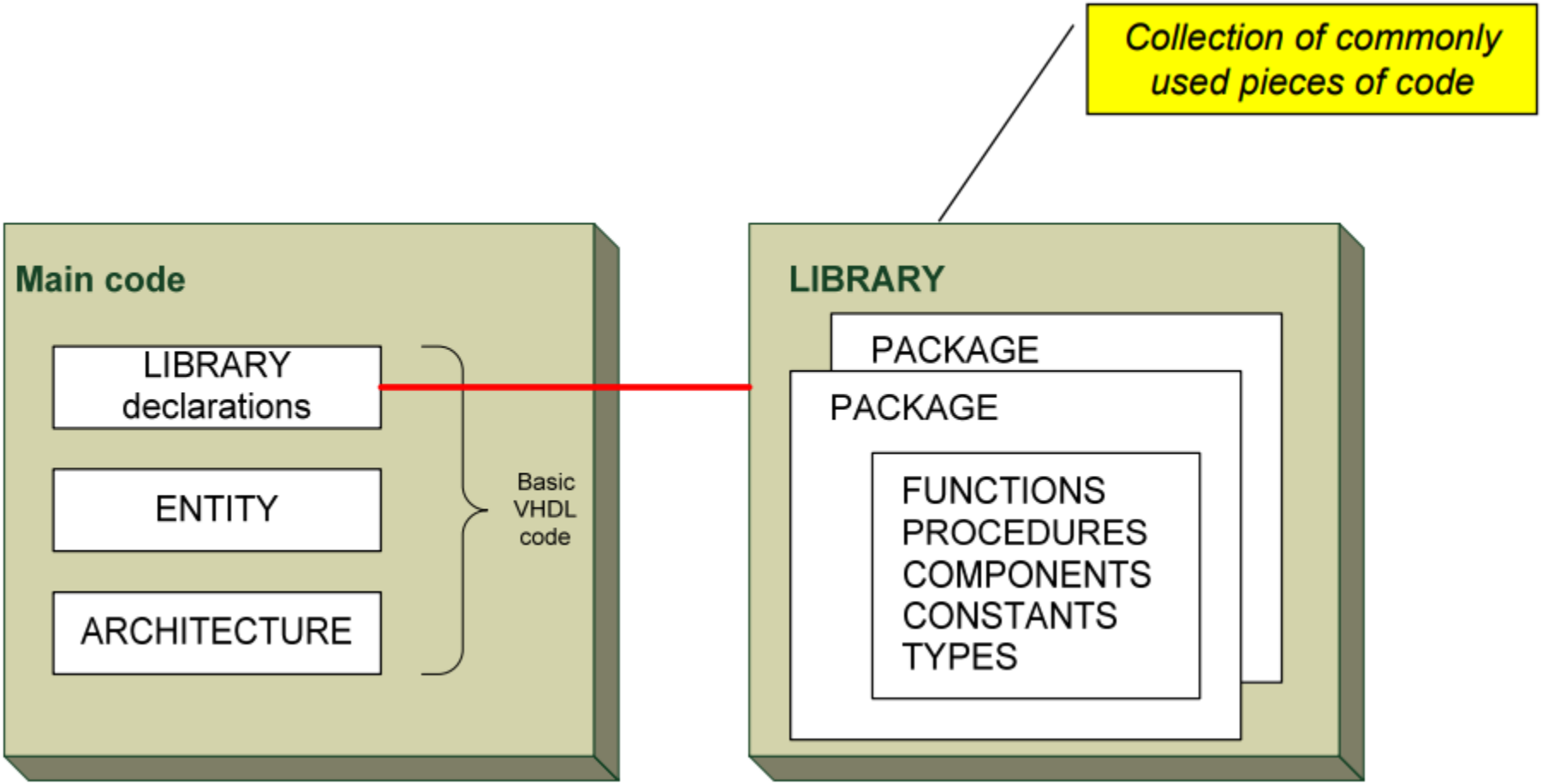
Unités de conception primaires: correspondent à la vue externe des objets.

1. Déclaration d'entité
2. Déclaration du paquetage
3. Déclaration de configuration

Unités de conception secondaires: correspondent aux algorithmes des modèles et des sousprogrammes:

1. Corps d'architecture
2. Corps du paquetage.

Structure de code VHDL



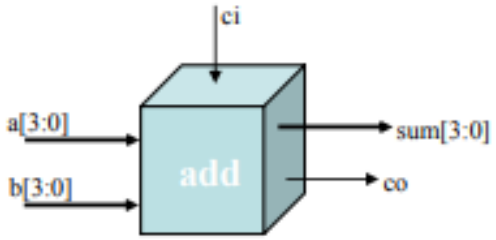
Entité

- Le bloc entity définit le nom d'un module ainsi que son interface avec le monde extérieur.

```
entity nom-de-l-entité is
    generic (
        paramètre1 : type := valeur-par-défaut;
        paramètre2 : type := valeur-par-défaut;
        …
    );
    port (
        port1 : direction type;
        port2 : direction type;
        …
    );
end nom-de-l-entité;
```

Entité

- Description des entrées sorties
  - Comme le schéma d'un composant
  - Description des "ports" d'un composant (les "pattes" d'E/S du schéma)
  - **entity** <name> **is** . . . **end** <name>;
  - EXEMPLE : un additionneur 4 bits



Schéma

```
entity add4 is port(
   a, b:  in std_logic_vector(3 downto 0);
   ci:    in std_logic;
   sum: out std_logic_vector(3 downto 0);
   co:    out std_logic);
end add4;
```

Entité

- Les modes des ports

Les trois directions les plus populaires sont :

- `in` pour les ports d'entrée;
- `out` pour les ports de sortie; et,
- `inout` pour les ports bidirectionnels.

Architecture

Le bloc architecture définit le comportement d'une entité, c'est-à-dire son fonctionnement intérieur.

- Une architecture décrit le contenu d'une entité
- VHDL a 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

Architecture

- Le bloc architecture comporte deux parties : une partie déclarative et un corps.
  - Dans la partie déclarative, on peut retrouver des déclarations de signaux, de constantes, de composantes et de types. On peut aussi y définir des sous-routines sous la forme de fonctions et de procédures.
  - Dans le corps de l'architecture on retrouve des énoncés concurrents qui décrivent le comportement de l'entité à laquelle l'architecture est rattachée.

Architecture

Syntaxe

```
architecture nom-de-l-architecture of nom-de-l-entité is
    [déclarations de signaux, constantes, composantes, types]
begin
    [énoncés concurrents]
end nom-de-l-architecture;
```

Architecture

- Décrit le comportement de l'entité
- Doit être associé à une entité spécifique
- Une seule entité peut avoir plusieurs architectures

Configuration

- Utilisé pour lier des entités à des architectures pour la simulation.
- Requis pour simuler des conceptions structurelles (avec des composants sous-jacents - netlists) mais n'est pas obligatoire pour les conceptions purement comportementales.
- Les configurations constituent généralement l'unité de conception finale à compiler.
- La configuration (CONFIGURATION) est une unité de conception primaire qui permet de créer un couple entité-architecture (plusieurs architectures pouvant être associées à une entité). Dans les cas simple, elle peut être remplacée par une ligne comportant la clause USE...

Librairie et Package

En pratique, on trouve trois types de bibliothèques :

- ***WORK*** : bibliothèque de travail,
- ***STD*** : bibliothèque standard VHDL,
- ***IEEE*** : bibliothèque standard IEEE.

Les bibliothèques WORK et STD font l'objet d'une clause library implicite, c'est-à-dire que toute unité est comme précédée de la clause :

```
LIBRARY work, std;
USE std.standard.all;
```

Pour la librairie IEEE, il est nécessaire de la déclarer ainsi que les paquetages utilisés :

```
LIBRARY IEEE;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;  --opérations  non signées
ou USE ieee.std_logic_signed.all;      --opérations signées
ou USE ieee.std_logic_arith.all;       -- opérations ALU
```

Notion de paquetage: toolbox

- Un package contient l'ensemble des fonctions, procédures, constantes, variables … utilisées par plusieurs descriptions.
- 2 parties : spécification + description interne

Syntaxe :

```
PACKAGE nom_du_package IS
        <<Déclaration des composants>>
        <<Déclaration des types>>
        <<Déclaration des signaux>>
END nom_du_package;
```

Un package est toujours associé à une librairie dont la visibilité se fait par l'instruction ***LIBRARY*** et celle du package par ***USE***.

```
LIBRARY nom_de_la_librairie;
USE nom_de_la librairie.nom_du_package.ALL;
```

Exemple: Paquetage

Considérons une constante utilisée par deux architectures :

Tps : temps de propagation = 27 ns

Déclaration :

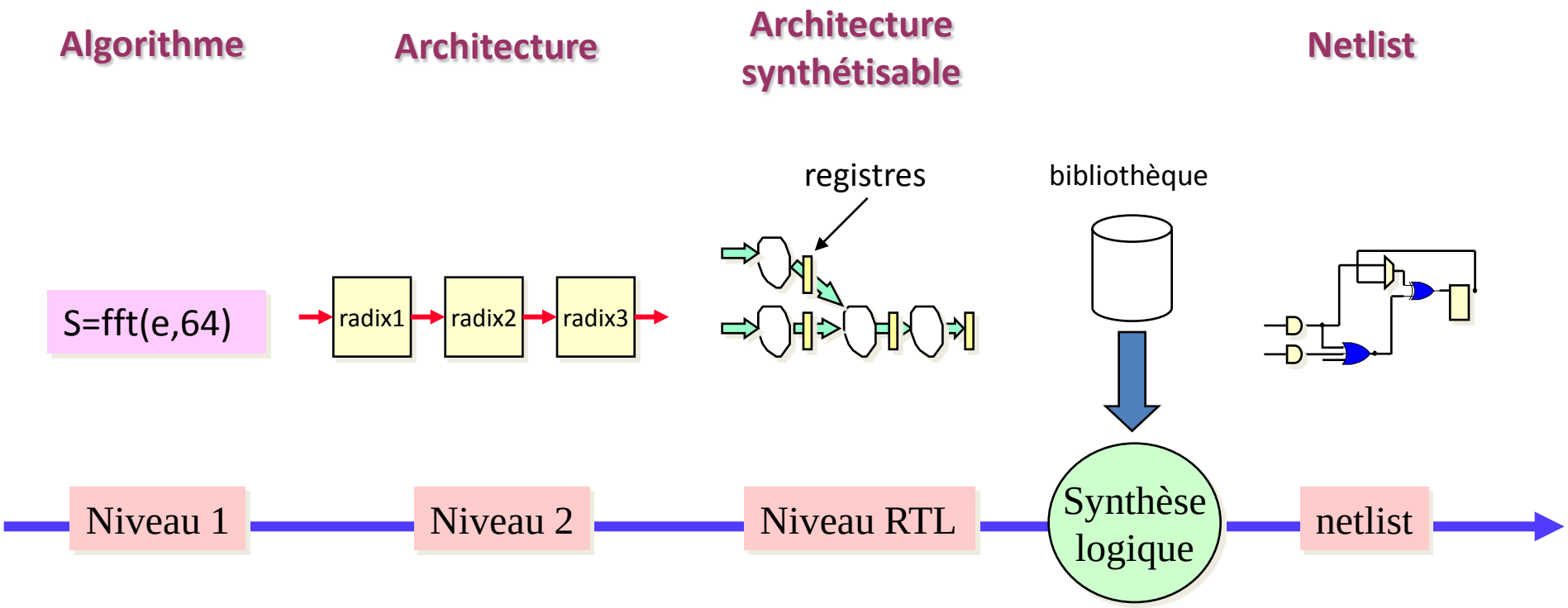
```
PACKAGE Pack1 IS
CONSTANT tps : TIME;
END Pack1;
...
PACKAGE BODY Pack1_Bod OF Pack1 IS
CONSTANT tps : TIME : = 27 ns;
END Pack1_Bod;
```

Utilisation :

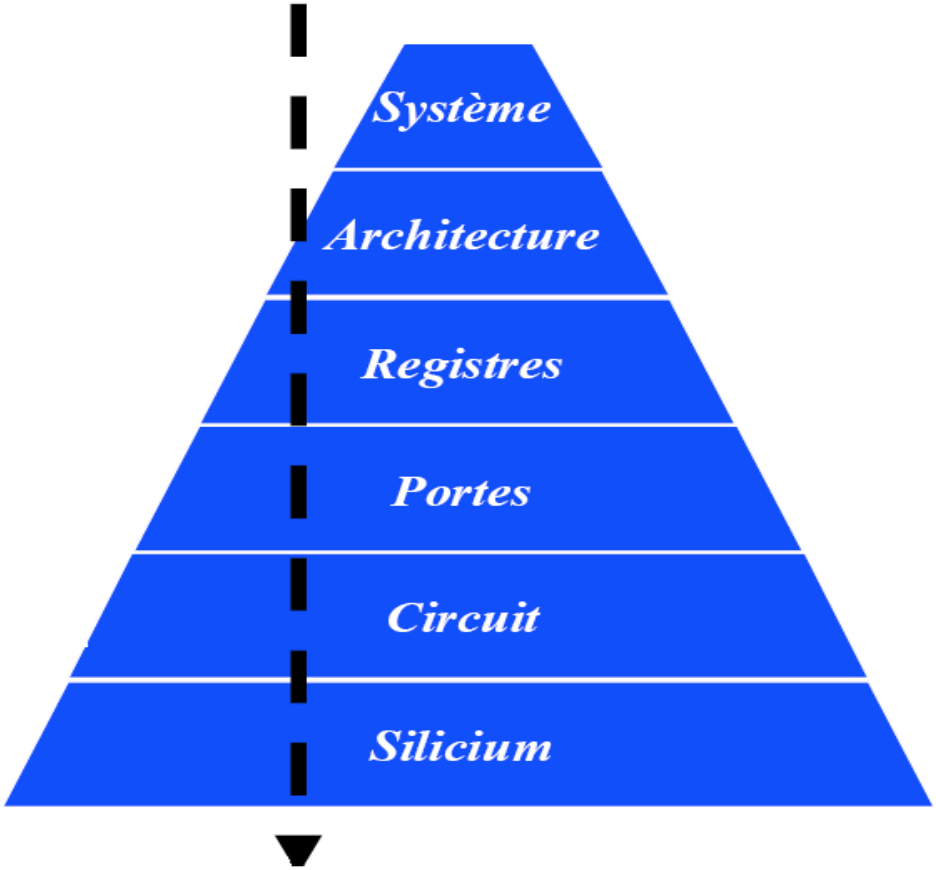
```
USE WORK.Pack1.all;
ENTITY A IS ...
END A;
ARCHITECTURE arch_A OF A IS ...
        S <= E1 AND E2 AFTER tps;
        ...
END arch_A;
```

Niveau d'abstraction

Le codage en VHDL peut se faire selon 4 niveaux d'abstraction, ce qui permet une conception descendante



Niveau d’abstraction



Niveau d'abstraction

- Le niveau système : cette description regroupe l'ensemble des spécification du circuit décrites sous la forme de comportements vis-à-vis de l'environnement associé au circuit, de performances à satisfaire, de contraintes d'utilisation. Cette description est de nature purement externe par rapport au circuit.

- Le niveau fonctionnel : la description au niveau fonctionnel exprime le premier niveau interne de la solution, sous la forme d'un ensemble de fonctions +/- interdépendantes et de complexité variée. Cette description est orientée application ou objectif à satisfaire.

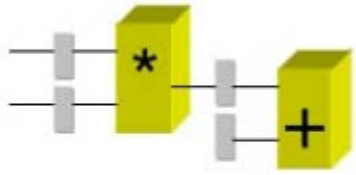
Niveau d'abstraction

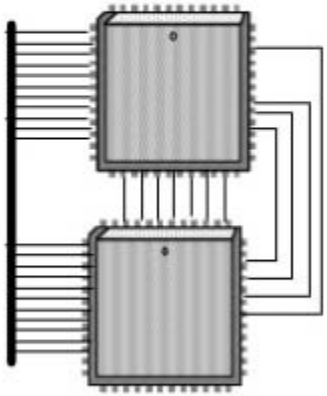
- Le niveau architectural : la description architecturale exprime les modules exécutifs et les interconnections nécessaires entre ceux-ci pour satisfaire la fonctionnalité globale du niveau supérieur. Cette description est donc orientée réalisation.
- Le niveau logique: la description représente un circuit par un ensemble interconnecté de fonctions logiques de faible complexité.
- Le niveau électrique : la description électrique représente le circuit sous la forme d'éléments microélectroniques (transistors essentiellement) interconnectés de façon à assurer la fonctionnalité du circuit spécifiée par les niveaux supérieurs.

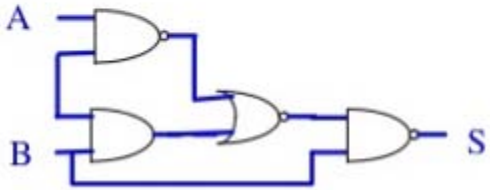
Les niveaux de description

- les niveaux de description peuvent être variables :
  - ✓ niveau logique
  - ✓ niveau transfert de registres
  - ✓ niveau système

A l'heure actuelle, il existe deux standards IEEE qui permettent de la modélisation RTL (ou plus bas niveau) : Verilog et VHDL.







ENSSAT - Université de Rennes 1 - France - Année universitaire 2003 - 2004

6

- Les avantages de VHDL :
  - indépendant du constructeur
  - indépendant de la technologie
  - indépendant de la démarche
  - indépendant du niveau de conception

⇒ Portabilité

Les objets du langage VHDL

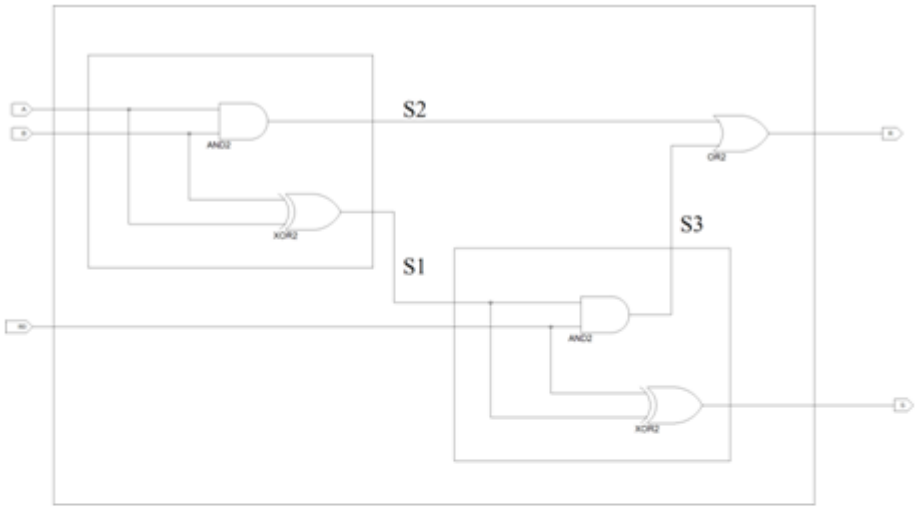
- Les objets sont des éléments de base du langage VHDL. Il ya trois familles d'objets.
  - Les constantes.
  - Les signaux.
  - Les variables.
- Chaque objet est bien repéré par un nom appelé identificateur. Celui-ci doit respecter les conventions d'écritures suivantes :
- Convention d'écritures
  - Il n'ya pas de différence entre minuscule et majuscule.
  - La longueur d'un identificateur est limitée à celle d'une ligne
  - Un identificateur est composé de lettre, du symbole « _ » (underscore et non tiret) et de chiffres, mais il doit commencer par une lettre ; il n'accepte pas deux symboles « _ » successifs.
  - A l'intérieur d'un programme un commentaire est introduit par deux tirets (--).
  - Une liste de mots est réservée à la syntaxe du langage VHDL, ces mots sont interdits comme identificateurs.

Les objets du langage VHDL

- **Les signaux**
  - Les signaux représentent les données physiques échangées entre des blocs logiques.
  - Un signal étant un nœud intermédiaire, ni une entrée, ni une sortie, ni une entrée/sortie.
  - Les signaux doivent être déclarés avant le mot Begin.
- Syntaxe : signal nom1, nom2 :type ;
- Exemple : Schéma d'un additionneur 1bit

Les objets du langage VHDL

Les signaux




```
architecture flot of add is
signal S1, S2, S3:STD_LOGIC;
begin
S1<=A xor B;
S2<=A and B;
S3<=S1 and R0;
S<=S1 xor R0;
R<=S2 or S3;
end flot;
```

Les objets du langage VHDL

Les variables

- Une variable se comporte comme un récipient que l'on peut remplir ou vider.
- L'affectation de la valeur d'une variable B à une variable A s'écrit : A :=B.
- La liaison entre les deux variables est temporaire ce qui veut dire que si la valeur de B change ensuite celle de A est inchangé.
- La valeurs d'une variable peut être affectée à un signal, est inversement, si leurs types sont compatibles.
- Une variable ne peut être déclarée et utilisée que dans un processus ou un sous programme.

Les objets du langage VHDL

Les constantes

Une constante est un objet qui est initialisé à une valeur donné lors de sa création.

Il ne peut être modifié.

Syntaxe :

Exemple :

```
Constant  k :int16 :=5 ;
Constant r :real :=5.7489 ;
Constant retard :time :5ns ;
```

Les opérateurs

Classe	Symbole	Fonction	Défini pour
Opérateurs divers	Not ** abs	Complément Exponentiel Valeur absolue	Bit, booléen, entier, réel, numérique
Opérateurs multiplicatifs	* / Mod rem	Multiplication Division Modulo Reste	Numérique Entier
Signe (unaire)	+ -	Positif Négatif	Numérique
Opérateurs additifs (binaire)	+ - &	Addition Soustraction Concaténation	Numérique 1 dimension
Opérateurs relationnels	= /= < > <= >=	Égal Différent Inférieur Supérieur Inférieur ou égal Supérieur ou égal	Tous les types Retourne un booléen
Opérateurs logiques (binaire)	And Or Nand Nor Xor	ET OU NON ET NON OU OU exclusif	Bit Booléen vecteur

Les opérateurs

Opérateurs logiques	VHDL
Et	**And**
Non et	**Nand**
Ou	**Or**
Non ou	**Nor**
Ou exclusif	**Xor**
Non ou exclusif	**Xnor**
Décalage à gauche	**SLL**
Décalage à droite	**SRL**
Rotation à gauche	**ROL**
Rotation à droite	**ROR**

Opérateurs arithmétiques	VHDL
Addition	+
Soustraction	-
Multiplication	*
Division	/

Remarque

Pour pouvoir utiliser ces operateurs, il faut rajouter les bibliothèques suivantes :

```
use IEEE.NUMERIC_std.ALL;
Use IEEE.STD_LOGIC_ARITH.ALL;
```

Les types des données

- VHDL est un langage fortement typé: tout objet doit être déclaré avant utilisation
- Les types predéfinis sont:
  - scalaire: *integer*, *real*, *enumerated*, *physical*
  - composé: *array*, *record*
  - pointeur: *acces*
  - I/O: *file*

integer	: entier négatif ou positif
natural	: entier positif ou nul
positive	: entier positif
bit	: énuméré dont les deux seules valeurs possibles sont '0' et '1'
bit_vector	: composite tableau représentant un vecteur de bits
boolean	: énuméré dont les deux valeurs possibles sont false et true
real	: flottant compris entre -1.0E38 et 1.0E38

Les types des données

std_logic : 9 valeurs décrivant tous les états d'un signal logique

'U'	: non initialisé**
'X'	: niveau inconnu, forçage fort**
'0'	: niveau 0, forçage fort
'1'	: niveau 1, forçage fort
'Z'	: haute impédance *
'W'	: niveau inconnu, forçage faible**
'L'	: niveau 0, forçage faible
'H'	: niveau 1, forçage faible
'-'	: quelconque (don't care) *

(**) Utile pour la simulation

(*) Utile pour la synthèse

std_logic_vector : vecteur de std_logic

Pour utiliser ces types il faut inclure les directives suivantes dans le code source.

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

std_logic_vector(N-1 downto 0) –> Vecteur de N std_logic

Les types des données

- **La librairie IEEE**
  - A mettre au début de votre description
  - Pour rajouter les types étendues **std_logic** et **std_logic_vector**
    - ✓ use IEEE.STD_LOGIC_1164.all;
  - DORENAVANT nous remplacerons *SYSTEMATIQUEMENT*
    - ✓ BIT par STD_LOGIC
    - ✓ BIT_VECTOR par STD_LOGIC_VECTOR
  - Pour utiliser des fonctions arithmétiques sur ces STD_LOGIC_VECTOR
    - ✓ USE IEEE.NUMERIC_STD.ALL;
    - ✓ Et aussi USE IEEE.LOGIC_UNSIGNED.ALL; ou USE IEEE.LOGIC_UNSIGNED.ALL;

Exemples Applicatifs

- Q<=Q+1; -- Q étant par exemple un std_logic_vector et 1 est un entier!!
- A<B -- A et B des std_logic_vector
- oData<=CONV_STD_LOGIC_VECTOR(TEMP,8); avec TEMP integer range 0 to 255;

Les attributs

- Les attributs :
  - ils permettent de connaître les caractéristiques
    - ✓ des signaux
    - ✓ des tableaux
    - ✓ des types
  - très utilisés pour rendre les descriptions génériques

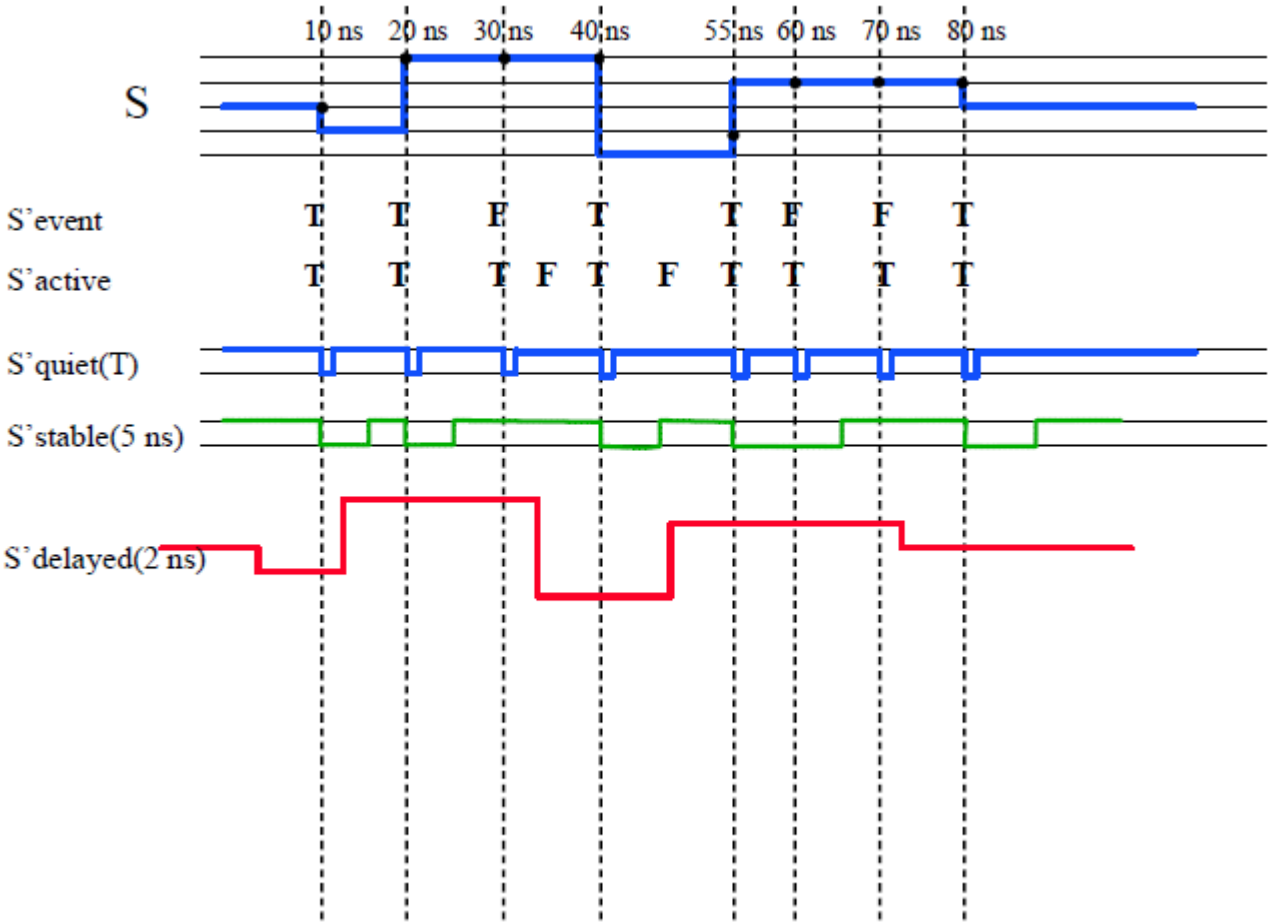
Les attributs

- **Attributs de signaux :**
  - ✓ **S'event :**
    - boolean
    - true si un événement vient d'arriver sur S pendant le cycle de simulation en cours
  - ✓ **S'active :**
    - boolean
    - true si il y a eu une transaction (affectation) sur le signal dans le cycle de simulation en cours
  - ✓ **S'quiet(T) :**
    - boolean
    - true si le signal a eu ni transaction ni événement pendant un temps T
  - ✓ **S'stable(T) :**
    - boolean
    - true si il n'y a pas eu d'événement sur le signal pendant le temps T
  - ✓ **S'transaction :**
    - signal
    - c'est un signal de type bit qui change d'états pour chaque transaction du signal

Les attributs

- ✓ **S'delayed :**
  - signal
  - c'est un signal identique à S mais retardé de T
- ✓ **S'last_event :**
  - time
  - rend le temps écoulé depuis le dernier événement sur le signal
- ✓ **S'last_active :**
  - time
  - rend le temps écoulé depuis la dernière transaction
- ✓ **S'last_value :**
  - type du signal
  - rend la valeur du signal immédiatement avant le dernier changement de S

Les attributs



Les différentes descriptions d'une architecture

- Structurel
  - ✓ ne fait pas intervenir le temps
  - ✓ décrit la structure de la fonction réalisée
  - ✓ décrit un schéma, des connexions entre composants
- Comportemental
  - ✓ algorithme
  - ✓ le temps peut intervenir
- Flot de données
  - ✓ exprime le flot de données sortants par rapport au flot entrant

Possibilité de mélanger ces modèles de description

Les différentes descriptions d'une architecture

❑ **Description flot de données : instructions concurrentes d'assignation de signal**

- Description de la manière dont les données circulent de signal en signal, ou d'une entrée vers une sortie
- Trois types d'instructions :

```
étiquette : ... <= ...
étiquette : ... <= ... when ... else ...
étiquette : with ... select ... <= ... when ...
```

- L'ordre d'écriture des instructions d'assignation de signaux est quelconque

Les différentes descriptions d'une architecture

❑ **Description structurelle : instructions concurrentes d'instanciation de composant**

- Interconnexion de composants (components), à la manière d'un schéma, mais sous forme d'une liste
- Dans l'architecture utilisatrice, un composant peut être considéré comme une boîte noire
- Un composant est instancié à l'aide d'une instruction d'appel de composant :

```
étiquette : nom_composant port map ( liste_des_entrées_et_sorties );
```

- L'ordre d'écriture des instructions d'instanciation de composants est quelconque
- Par définition, un composant est aussi un système logique (un sous-sytème) ; à ce titre, il doit aussi être décrit par un couple (entity, architecture) dans lequel sont définis ses entrées-sorties et son comportement
- Le corps d'architecture du composant (le comportement) est décrit selon un ou plusieurs styles ; par exemple, un composant peut faire appel à d'autres composants
- Un composant peut être rangé dans une bibliothèque

Les différentes descriptions d'une architecture

❑ Description comportementale : instructions concurrentes d'appel de processus

- Certains comportements peuvent être décrits de façon algorithmique ; il faut alors les définir comme des processus
- Dans l'architecture utilisatrice, un processus est considéré comme une instruction concurrente
- Instruction d'appel de processus :

```
étiquette: process déclarations begin instructions_séquentielles end process;
```

- L'ordre d'écriture des instructions d'appel de processus est quelconque
- Un processus **contient** des instructions séquentielles qui ne servent qu'à traduire simplement et efficacement, sous forme d'un algorithme, le comportement d'un sous-ensemble matériel
- **Des instructions séquentielles (un algorithme) peuvent décrire un système combinatoire ; a contrario, une instruction concurrente peut décrire un système séquentiel !!**
- À l'intérieur d'un processus, trois types d'instruction d'assignation de signaux et deux types d'instruction d'itération :

```
… <= …
if … then … else …
case … when …
for … loop …
while … loop …
```

- L'ordre d'écriture des instructions à l'intérieur du processus est déterminant

Les différentes descriptions: DATAFLOW

- Affectation simple
  - S <= a AND b AFTER 10 ns;
- Affectation conditionnelle
  - neuf <= ' 1' WHEN etat = " 1001" ELSE ' 0'
- Affectation avec sélection
  - WITH ad SELECT

```
s <= e0 WHEN 0,
e1 WHEN 1,
e2 WHEN 2,
e3 WHEN OTHERS;
```

Les différentes descriptions: Structurelle

- Description de type hiérarchique par liste de connexions
- Permet de construire une description à partir de couples entité-architecture
- Le nombre de niveaux de hiérarchie est quelconque
- Une description est structurelle au sens VHDL si elle comporte un ou plusieurs composants (component)

Description structurelle: 3 étapes

- Déclarer
  - un composant (COMPONENT) : Support pour le câblage
  - une liste de signaux (SIGNAL) nécessaires au
  - câblage
- Instancier
  - chaque composant en fixant les paramètres (GENERIC MAP) et le câblage(PORT MAP)
- Configurer
  - Choisir pour chaque composant instancié le modèle correct (couple Entité-architecture). (USE)

Description structurelle

Déclaration et instanciation des composants

Déclaration

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

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

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

Instanciation :

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

Description structurelle

Instanciation :

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

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

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

Exemple

Dans cet exempl , 2 instances de composant "and2" sont appelées pour créer une porte ET à 3 entrées.

Description structurelle

```
entity AND_3 is
 port(
 e1 : in bit;
 e2 : in bit;
 e3 : in bit;
 s : out bit
 );
end entity;

architecture arc of AND_3 is

signal z : bit;
component and2
       port (
       a : bit;
       b : bit;
       s : bit);
end component;

begin
 inst1 : and2 port map (a=>e1, b=>e2 ,
s=>z);
 inst2 : and2 port map (z, e3, s);
end arc
```

La syntaxe des associations est soit

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

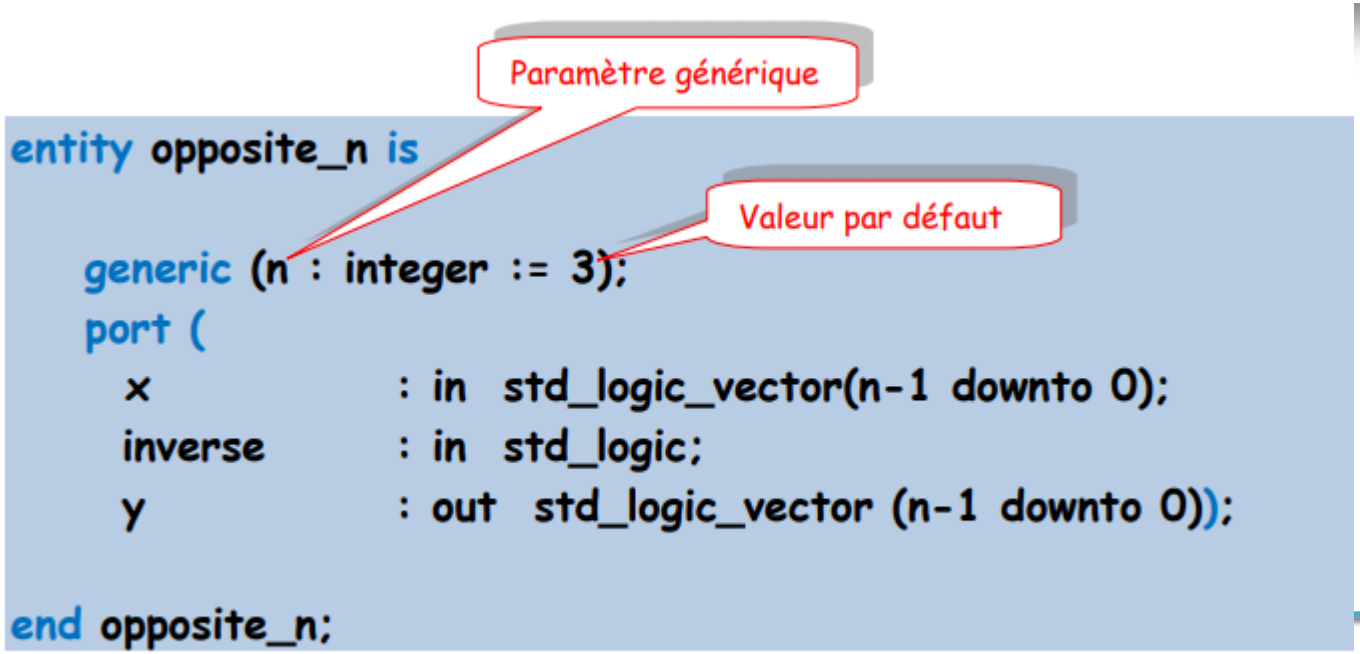
Les paramètres génériques

Un paramètre générique se déclare au début de l'entité, et peut avoir une valeur par défaut :

```
generic (nom : type [ := valeur_par_defaut ] ) ;
```

Au moment de l'instanciation la taille peut être modifiée par une instruction « generic map » :

```
Etiquette : nom generic map ( valeurs )
        port map ( liste_d'association ) ;
```



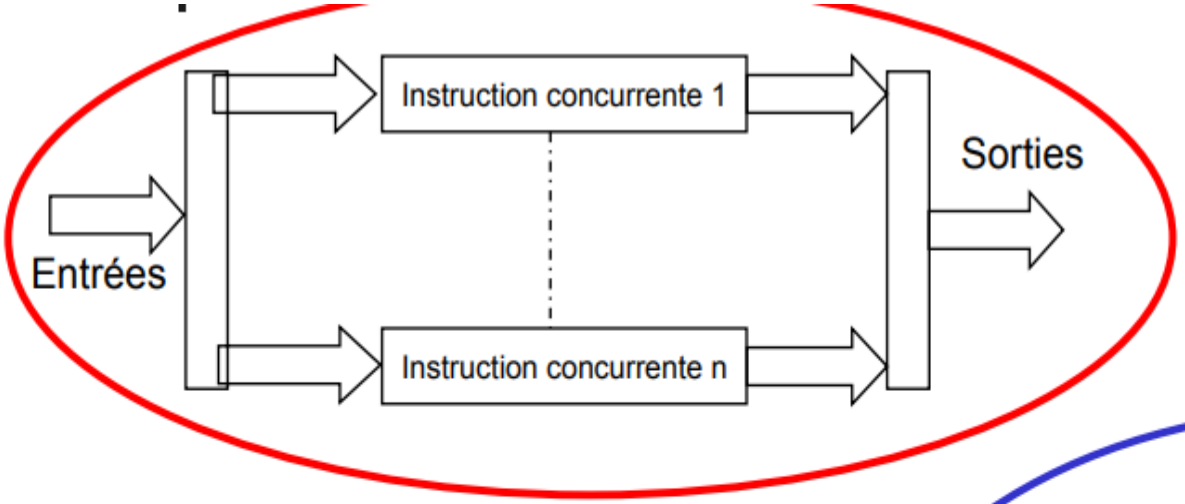
Les types d'instructions

- Il existe deux types d'instructions:
  - Instructions concurrentes

En VHDL, il existe deux types d'instructions

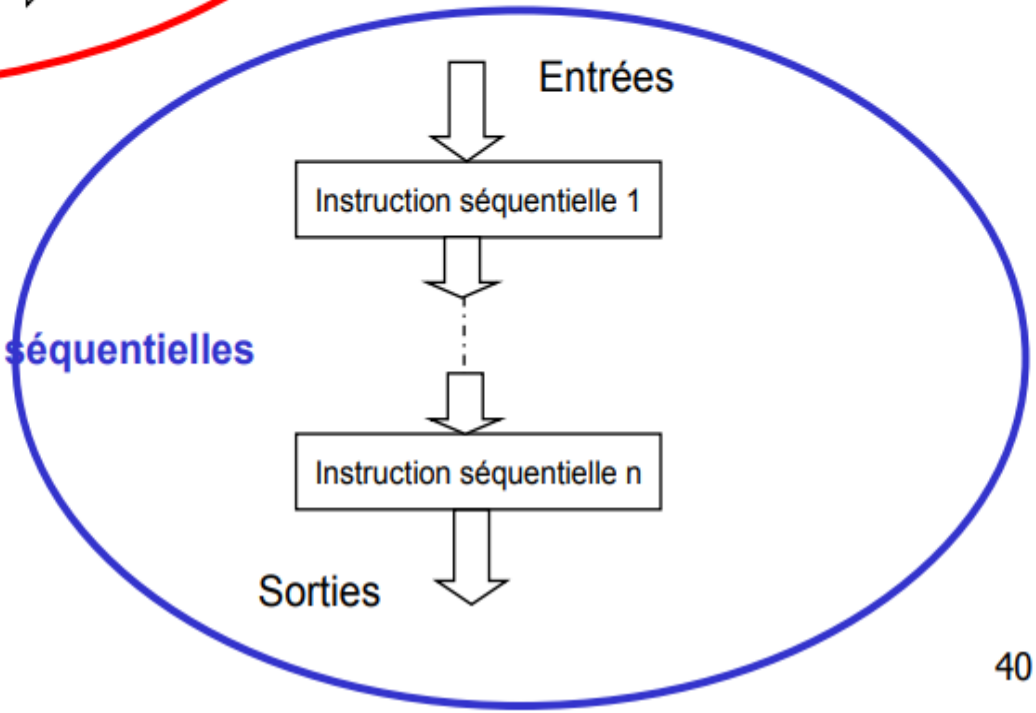
- Instructions concurrentes
  - à écrire dans la zone concurrente de l'architecture
  - elles sont exécutées simultanément
  - l'ordre de l'écriture n'a pas d'importance
- Instruction séquentielles
  - à écrire dans les « process »
  - elles sont exécutées dans l'ordre

Les types d'instructions



Instructions concurrentes

Instructions séquentielles



Les types d'instructions

3.3. Instructions concurrentes

3.3.1. Propriétés

❑ **L'ordre d'écriture des instructions n'a pas d'importance (c'est le parallélisme)**

- Une instruction concurrente décrit une opération qui porte sur des signaux (entrée, interne) pour produire d'autres signaux (interne, sortie)
- Tous les signaux mis en jeu dans l'architecture sont disponibles au même moment

❑ **Le corps d'architecture est décrit dans un ou plusieurs styles**

- Le style n'est pas imposé par le type de logique (combinatoire ou séquentielle) ou le type de traitement des données (parallèle ou séquentiel)
- Le style est choisi pour apporter concision ou par préférence personnelle

Trois styles peuvent coexister au sein d'une même architecture

Les types d'instructions

Les instructions séquentielles

- Ils s'écrivent uniquement dans une zone séquentielle: après le « begin » d'un process
  - instruction d'attente (wait)
  - instruction d'affectation (temporisée)
  - instruction conditionnelle (if)
  - instruction sélective (case)

Les types d'instructions

Les instructions séquentielles

- wait :
  - ✓ suspend l'exécution d'un processus
  - ✓ plusieurs cas :
    - suspension jusqu'à la fin des temps : **wait**
    - suspension pendant un temps donnée **: wait for 10 ns;**
    - suspension jusqu'à événement sur signaux : **wait on S1, S2**
    - idem précédent plus condition : **wait on S1, S2 until condition**
    - idem précédent plus délai maximum d'attente :

      wait on S1, S2 until condition for 5 ms

  - ✓ interdit à l'intérieur d'une fonction (une fonction rend un résultat ***immédiat***, donc pas d'attente)

Process

- Process est une **instruction concurrente** : donc à écrire dans une architecture (après begin),

- Les différents process d'une architecture s'exécutent tous en parallèle

- Process génère une **zone séquentielle**

Les processus sont cycliques

plusieurs processus peuvent être présents dans une architecture

Process

•Syntaxe de process :

```
label  :  -- optionnel
          process (liste de sensibilité)
                    {partie déclarative}
          begin
                    suite d'instructions séquentielles
          end process;
```

•**Notion très importante :** Un process **s'exécute** (se réveille) quand un des signaux de la liste de sensibilité change de valeur.

•Un fois on arrive à la fin du process, celui-ci rendort jusqu'à l'arrivée d'un évènement sur un des signaux de sa liste de sensibilité,

Process

- L'exécution d'un **process** est déclenchée par un ou des changements d'états de signaux logiques. Le nom, de ces signaux est définit dans la liste de sensibilité.
- Dans un processus, un signal n'aura sa valeur d'affectation qu'à la fin du process. A l'inverse, la variable aura sa nouvelle valeur affectée immédiatement. A cause de sa, à l'intérieur d'un process, on préféra d'utiliser des variables intermédiaires plutôt que des signaux intermédiaires.
- Les instructions du process s'exécutent séquentiellement.

Process

D’autre écritures équivalentes

```
process
begin
  <statements>;
wait on (<all_input_signals_separated_by_commas>)
end process;
```

L’instruction **wait** ne peut s’utiliser à l’intérieure d’un process que si celui-ci ne possède pas de liste de sensibilité.
Une liste de sensibilité est équivalente à un **wait on** placé enfin de process.

Sous programmes : fonction, procédure

Le rôle d'une procédure ou d'une fonction est de permettre au sein d'une description la création d'outils dédiés à certaines tâches pour un type déterminé de signaux.

Les Fonctions :

Une fonction reçoit des paramètres d'entrée et renvoie un paramètre de retour.

Les procédures :

Une procédure ne possède pas un ensemble de paramètres d'entrée et un paramètre de sortie mais un ensemble de paramètres d'entrée/sortie.

L'appel d'une procédure est une instruction alors que l'appel d'une fonction est une expression.

Les fonctions de résolution

Ces fonctions peuvent être appelées explicitement (dans une expression) ou implicitement (chaque fois qu'une source concernée doit être résolue).

Fonction

Syntaxe

```
FUNCTION Nom_de_la_fonction (liste des paramètres : type)
RETURN  type du paramètre retourné IS
Zone de déclaration des variables;
BEGIN
        Instructions séquentielles;
        RETURN nom_de_la_variable_de_retour_ou_valeur_de_retour;
END ;
```

Une fonction reçoit des paramètres d'entrée et renvoie un paramètre de retour.

L'appel d'une fonction peut se faire soit dans les instructions concurrentes soit dans les instructions séquentielles. Dans le cas des instructions concurrentes, la fonction sera toujours vérifiée.

Exemple : fonction convertissant un boolean en un bit.

```
FUNCTION bool_to_bit (X: BOOLEAN) RETURN BIT IS
BEGIN
        IF (X=TRUE) THEN  RETURN '1';
        ELSE RETURN  '0';
        END IF;
END;
```

Exemple: Fonction

```
ENTITY  Montage1 IS
   PORT (  E1, E2, E3, E4 : IN std_logic ;
           S1,S2 : OUT std_logic);
END Montage1;
ARCHITECTURE arch OF Montage1 IS
   FUNCTION NON_ET (A,B,C : STD_LOGIC) RETURN  STD_LOGIC IS
      VARIABLE result : STD_LOGIC;
   BEGIN
           result:= NOT (A AND B AND C):
           RETURN result;
   END;
   BEGIN
           S1 <= NON_ET(E1,E2,E3);
           S2 <= NON_ET(E1,E2,E4);
END arch ;
```

Déclaration

Utilisation

Une fonction ne doit jamais arriver à son mot clé END final, cela provoquerait une erreur. Elle doit toujours rencontrer le mot clé return et rendre la main.

Procédure

Une procédure, à la différence d'une fonction, accepte des paramètres d'entrée dont la direction peut-être IN, OUT, INOUT.

Exemple : la bascule SR

```
ENTITY  Test_SR IS
   PORT(   E1, E2 : IN std_logic ;
           S1,S2 : INOUT std_logic);
END Test_SR;
ARCHITECTURE arch OF Test_SR IS
   PROCEDURE SR (signal A,B : IN STD_LOGIC, signal Qb, Q : INOUT std_logic) IS
   BEGIN
           Q <= NOT (A AND Qb):
           Qb <= NOT( B AND Q);
   END;
   BEGIN
           SR(E1,E2,S1,S2);
END arch ;
```

Procédure

Si les paramètres de la procédure sont des variables l'affectation se fait par := .
Reprenons l'exemple précédent avec des variables :

```
ENTITY Test_SR IS
   PORT( E1, E2 : IN std_logic ;
         S1,S2 : INOUT std_logic);
END Test_SR;
ARCHITECTURE arch OF Test_SR IS
   PROCEDURE SR (signal A,B : IN STD_LOGIC, variable Qb, Q : INOUT std_logic) IS
   BEGIN
         Q := NOT (A AND Qb):
         Qb := NOT( B AND Q);
 END;
BEGIN
         PROCESS (E1,E2)
         VARIABLE Q1,Q2 : std_logic;
         BEGIN
                  SR(E1,E2,Q1,Q2);
                  IF (Q1='1') THEN S1<='1';  ELSE Q1<='0'; END IF;
                  IF (Q2='1') THEN S2<='1';  ELSE Q2<='0'; END IF;
         END PROCESS;
END arch ;
```

Fonctions et procédures au sein des Packages

Afin de rendre accessible les fonctions et les procédures par plusieurs architectures, il est possible de créer un package.

```
PACKAGE Pack_NONET IS
        FUNCTION NONET (A,B,C : std_logic) return std_logic;
END Pack_NONET;

PACKAGE BODY Pack_NONET IS
  FUNCTION NONET (A,B,C : std_logic) return std_logic;
    VARIABLE result : STD_LOGIC;
  BEGIN
        result<= NOT (A AND B AND C):
        RETURN result;
  END;
END Pack_NONET;
```

Exercice : créer un package qui contient deux fonctions l'une permettant de calculer le MAX de deux entiers et l'autre le MIN en valeur absolue.

Pour rappeler la fonction,

```
LIBRARY USER;
USE USER.Pack_NONET.all;
ENTITY  Montage1 IS
   PORT (  E1, E2, E3, E4 : IN std_logic ;
           S1,S2 : OUT std_logic);
END Montage1;
ARCHITECTURE arch OF Montage1 IS
  BEGIN
        S1 <= NON_ET(E1,E2,E3);
        S2 <= NON_ET(E1,E2,E4);
END arch ;
```

Simulation

- Il existe 2 niveaux de simulation : comportementale (behavorial simulation) ;
  - physique (post-route simulation).
  - Dans la version allégée de Xilinx ISE WebPack, seule la simulation comportementale est disponible.

Simulation

- Comment faire une simulation ?
  - instanciation du composant à tester
  - initialisation des signaux d'entrées
  - application d'une séquence de stimuli :
    - ✓ à partir d'un process et d'affectations des signaux d 'entrées
    - ✓ à partir d'un fichier contenant des vecteurs de test
  - analyse des résultats, analyse des transitions des sorties :
    - ✓ affichage des erreurs éventuelles

5.3.1. Simulation comportementale (fonctionnelle)

- Elle montre le comportement théorique (pas de temps de propagation)

Un TestBench (ou banc de test), en simulation, permet de vérifier la validité d'une architecture.

C'est un module VHDL (entity + architecture) qui instancie l'architecture à tester.

Sa déclaration d'entité de contient par d'entrées/sorties !

Le but est de la soumettre à des variations sur les entrées affin de vérifier la correction des sorties générées et de la validité de son comportement temporel.

Ex :

```
entity adder_tb is
end adder_tb;
```

Simulation comportementale

- Pour tester un module VHDL, il faut lui associer un module générateur de signaux (non synthétisable) : le testbench.
- Le testbench est un module VHDL spécial (disponible dans la liste des nouveaux modules lors de l'ajout de fichier source). Il inclut le module à tester comme un composant et lui associe des signaux. L'écriture d'un testbench est automatisée avec Xilinx ISE. Il reste ensuite à écrire un scénario de test animant les signaux d'entrée du module en test. On utilisera alors la même syntaxe qu'un module classique mais avec des instructions spécifiques.

Notion de testbench

- Un test bench est un module VHDL comme un autre, mais qui n'a ni entrées ni sorties, il est autonome.
- Il sert à tester le comportement d'un autre module.
- Il produit lui même les signaux à envoyer au circuit à tester et vérifie que les sorties du circuit sont correctes.

Notion de testbench

- Pour simuler votre conception, vous devez produire un
  - Design ENTITY et ARCHITECTURE.
  - Habituellement appelé banc d'essai
  - Pas du matériel, juste du VHDL supplémentaire

Test bench (banc d’essai)

- Un banc d'essai comprend
  - Entité
    - n'a pas de port (en-tête d'entité vide)
  - Architecture
    - déclare, instancie et relie ensemble le modèle de pilote et le modèle sous test
    - le modèle de pilote fournit un stimulus et vérifie les réponses du modèle
  - Configuration
    - spécifier le scénario de test du modèle de pilote et la version du modèle sous test

test bench du FA

```
[libraries...]
entity full_add1_tb is
end entity;

architecture BEH of full_add1_tb is

  signal w_a, w_b, w_cin, w_s, w_cout : std_logic;

  component full_add1 is
    port (a, b, cin : in std_logic;
          s, cout   : out std_logic);
  end component;

begin

  dut : full_add1 port map(
          a => w_a,
          b => w_b,
          cin => w_cin,
          s => w_s,
          cout => w_cout
        )
```

test bench du FA

```
process
begin
    w_a   <= '0';
    w_b   <= '0';
    w_cin <= '0';
    wait for 5 ns;
    w_a   <= '1';
    w_b   <= '0';
    w_cin <= '0';
    wait for 5 ns;
    w_a   <= '1';
    w_b   <= '1';
    w_cin <= '0';
    wait for 5 ns;
    w_a   <= '1';
    w_b   <= '1';
    w_cin <= '1';
  end process;

end BEH;
```