CHAPITRE 2

Programmation en VHDL

2.1 Introduction au VHDL

Le **VHDL** (VHSIC Hardware Description Language) est un langage de description du matériel conçu dans les années 1980 dans le cadre du programme *VHSIC (Very High Speed Integrated Circuit)*. Il permet de **modéliser**, **simuler** et **synthétiser** des systèmes numériques complexes avant leur implémentation physique sur **FPGA** ou **ASIC**.

Contrairement aux langages de programmation classiques comme `C` ou `Java`, qui décrivent des algorithmes exécutés de manière séquentielle, le VHDL décrit le fonctionnement de circuits numériques qui s'exécutent de manière **concurrente**. Cela reflète fidèlement la nature parallèle du matériel électronique.

2.2 Objectifs du VHDL

Le VHDL poursuit trois objectifs principaux.

— **Modélisation** : représenter des circuits logiques à différents niveaux d'abstraction.

— **Simulation** : vérifier le fonctionnement logique et temporel du système.

— **Synthèse** : traduire la description en une implémentation matérielle réelle.

Tout dabord, la **modélisation** permet de représenter les circuits numériques à différents niveaux dabstraction. Au niveau logique, on peut décrire directement des portes élémentaires comme `AND`, `OR` ou `NOT`. Au niveau RTL (Register Transfer Level), on exprime le transfert des données entre registres en tenant compte des horloges et des signaux de contrôle. Enfin, au niveau système, il est possible de modéliser des blocs fonctionnels plus complexes tels quun processeur ou une mémoire.

Ensuite, la **simulation** sert à vérifier le fonctionnement logique et temporel du système avant toute implémentation matérielle. Elle permet dobserver lévolution des signaux au cours du temps sous forme de chronogrammes, de détecter des erreurs logiques ou temporelles, et de valider que les résultats produits correspondent bien aux attentes (par

exemple, vérifier quun additionneur 4 bits génère la bonne somme pour chaque combinaison dentrées).

Enfin, la **synthèse** consiste à traduire la description VHDL en une implémentation matérielle réelle. Le synthétiseur transforme les instructions en un réseau de portes logiques qui peut être chargé sur un FPGA ou intégré dans un ASIC. Ainsi, une simple instruction comme `Y <= A AND B;` sera concrètement réalisée par une porte physique `AND` dans le circuit.

2.3 Caractéristiques principales

Langage standardisé (IEEE 1076). Le VHDL est défini par la norme IEEE 1076.

Approche hiérarchique et modulaire. Le VHDL encourage une structuration en niveaux : du système complet jusquaux sous-blocs et composants élémentaires. Il sépare l*interface* (`entity`, qui définit les ports) de l*implémentation* (`architecture`, qui décrit le comportement). Les conceptions se construisent par composition de composants réutilisables, reliés par des ports et éventuellement paramétrés par des *génériques*. Cette hiérarchie favorise la réutilisation, les tests ciblés à chaque niveau et lévolution incrémentale du design.

Description précise du comportement logique et temporel. Le VHDL modélise le matériel comme un ensemble de processus *concurrents* avec une sémantique temporelle explicite. Les signaux évoluent dans le temps, permettant de décrire fidèlement la logique combinatoire et séquentielle (synchronisée par lhorloge). Le *timing* sexprime via des délais (`after`) et les sémantiques *inertielle* (filtrage des glitches) et *transport* (transport idéal). La logique multi-valeurs (`'0'`, `'1'`, `'Z'`, `'X'`, etc.) et la détection de fronts (par ex. `rising_edge`) offrent une représentation réaliste du comportement et des contraintes temporelles du matériel.

2.4 Structure dune description VHDL simple

Un opérateur élémentaire, un circuit intégré, une carte électronique ou encore un système complet est défini par deux aspects fondamentaux :

— les signaux dentrées et de sorties (interface de communication) ;

— la fonction réalisée en interne (comportement).

Ces deux aspects se retrouvent à tous les niveaux de la hiérarchie dune application. En VHDL, lélément de base de toute description, appelé *design entity*, est constitué du couple **entité / architecture** :

— **Lentité** : décrit linterface externe du circuit (les entrées et sorties).

— **Larchitecture** : décrit le fonctionnement interne du circuit.

Ainsi, une même entité peut être associée à plusieurs architectures différentes, selon la manière dont on souhaite implémenter son comportement interne.

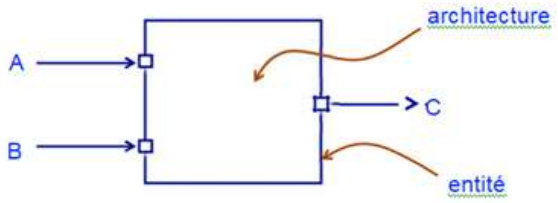

Figure 2.1 – Entité et architecture en VHDL

La structure minimale dun programme VHDL suit donc ce schéma :

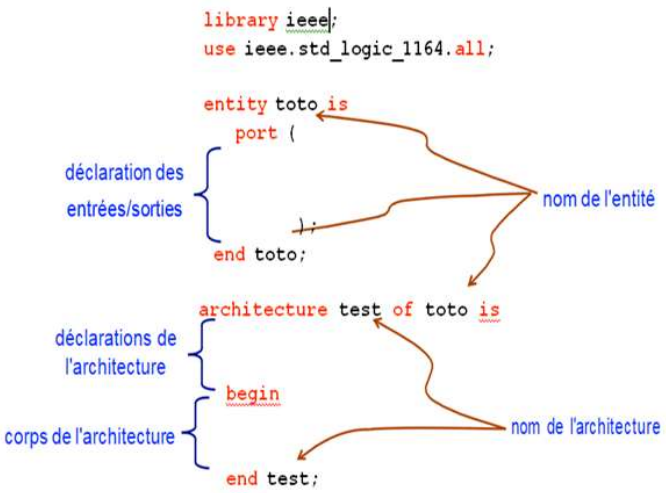

Figure 2.2 – Structure générale dun programme VHDL

Le code suivant illustre la description d'une porte logique **AND** en VHDL :

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

entity Porte_AND is
    Port (
        A,B : in STD_LOGIC;
        Y : out STD_LOGIC
    );
end Porte_AND;

architecture Comportement of Porte_AND is
```

```
begin
    Y <= A AND B;
end Comportement;
```

Listing 2.1 – Exemple de description d'une porte AND en VHDL

Cet exemple montre la structure fondamentale d'un programme VHDL :

— **Entity** : définit l'interface (entrées/sorties).

— **Architecture** : décrit le comportement interne.

Ainsi, le VHDL constitue un outil incontournable pour passer du **concept théorique** au **prototype matériel**.

2.4.1 Déclaration des bibliothèques

Toute description VHDL utilisée pour la synthèse nécessite lutilisation de bibliothèques. L**IEEE** (*Institute of Electrical and Electronics Engineers*) a normalisé plusieurs bibliothèques, dont la plus importante est la bibliothèque `IEEE1164`, qui contient les définitions des types de signaux électroniques ainsi que des fonctions et sous-programmes permettant de réaliser des opérations arithmétiques et logiques.

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

Listing 2.2 – Exemple de déclaration de bibliothèques

Les principales bibliothèques utilisées en VHDL sont les suivantes :

`IEEE.STD_LOGIC_1164` Définition du type `STD_LOGIC` et des valeurs multivaluées (X, U, L, H, 0, 1).

`IEEE.STD_LOGIC_ARITH` Opérateurs arithmétiques (+, , *, /) pour `STD_LOGIC`.

`IEEE.STD_LOGIC_SIGNED` Interprétation dun vecteur `STD_LOGIC` comme nombre signé.

`IEEE.STD_LOGIC_UNSIGNED` Interprétation dun vecteur `STD_LOGIC` comme nombre non signé.

`IEEE.NUMERIC_STD` Opérations arithmétiques normalisées sur `SIGNED`/`UNSIGNED`.

`IEEE.STD_LOGIC_TEXTIO` E/S texte (ASCII) avec `STD_LOGIC`.

`IEEE.NUMERIC_BIT` Types et opérateurs pour vecteurs binaires.

`IEEE.MATH_REAL` Nombres réels.

`IEEE.MATH_COMPLEX` Nombres complexes.

Remarques :

1. La directive `use` permet de sélectionner les bibliothèques à utiliser.
2. La directive `all` permet dimporter toutes les entités disponibles de la bibliothèque.

Enfin, il convient de rappeler que :

— La norme **IEEE 1076** (1987) définit le langage VHDL.

— La norme **IEEE 1164** (1993) définit le système de valeurs multivaluées pour le type `STD_LOGIC`.

Note importante : Dans les versions anciennes de VHDL, plusieurs bibliothèques non normalisées étaient souvent utilisées, telles que : `IEEE.STD_LOGIC_SIGNED`, `IEEE.STD_LOGIC_UNSIGNED` et `IEEE.STD_LOGIC_ARITH`. Elles permettaient deffectuer des opérations arithmétiques directement sur des vecteurs du type `STD_LOGIC_VECTOR`. Cependant, ces bibliothèques ne font pas partie du standard IEEE et leur usage peut entraîner des problèmes de portabilité et des ambiguïtés dans linterprétation des données.

Aujourdhui, il est fortement recommandé dutiliser la bibliothèque normalisée `IEEE.NUMERIC_STD`, qui définit explicitement deux types distincts : `signed` (nombres signés, pouvant être positifs ou négatifs) et `unsigned` (nombres non signés, uniquement positifs). Cette approche rend le code plus clair, plus robuste et pleinement conforme aux normes IEEE.

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

entity Adder is
    Port (
        A  : in signed(3 downto 0);   -- Entrée signée (peut être négative ou
           positive)
        B  : in unsigned(3 downto 0); -- Entrée non signée (toujours positive)
        S1 : out signed(4 downto 0);  -- Résultat signé
        S2 : out unsigned(4 downto 0) -- Résultat non signé
    );
end Adder;

architecture Behavioral of Adder is
begin
    -- Exemple : A = "1110" = -2 (signed), B = "1110" = 14 (unsigned)
    -- Résultat signé : -2 + 14 = 12
    -- Résultat non signé : 14 + 14 = 28

    S1 <= resize(A, 5) + signed(resize(B, 5));
    S2 <= resize(B, 5) + unsigned(resize(B, 5));
```

```
end Behavioral;
```

Listing 2.3 – Exemple moderne avec NUMERIC_STD

Dans cet exemple :

— Lentrée `A` est interprétée comme un **nombre signé**, ce qui signifie que la valeur binaire `"1110"` est traduite en -2.

— Lentrée `B` est interprétée comme un **nombre non signé**, la même valeur binaire `"1110"` est traduite en 14.

— En sortie, `S1` montre laddition signée ($-2+14 = 12$), tandis que `S2` montre laddition non signée ($14 + 14 = 28$).

Ainsi, la même combinaison de bits peut avoir des significations différentes selon quelle est interprétée comme `signed` ou `unsigned`.

2.4.2 Déclaration de lentité

La déclaration d'entité décrit l'interface entre le monde extérieur et une unité de conception : signaux d'entrées et de sorties et, éventuellement, paramètres génériques (constantes dont la valeur est fixée par l'environnement de l'unité considérée). La déclaration d'entité peut également contenir des instructions concurrentes passives, c'est-à-dire qui ne contiennent aucune affectation de signaux ; de telles instructions ne génèrent pas de circuits lors du processus de synthèse.

Une même entité peut être associée à plusieurs architectures différentes. Elle décrit alors une classe d'unités de conception qui présentent au monde extérieur le même aspect, avec des fonctionnements internes différents.

L'entité précise :

— le nom du circuit ;

— les ports d'entrée-sortie :

  — leurs noms ;

  — leurs directions (`in`, `out`, `inout`, ...) ;

  — leurs types (`bit`, `bit_vector`, `integer`, `std_logic`, ...) ;

— les paramètres éventuels pour les modèles génériques.

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

entity FA is
  port (
    a, b, cin : in std_logic;
    s, cout   : out std_logic
  );
end FA;
```

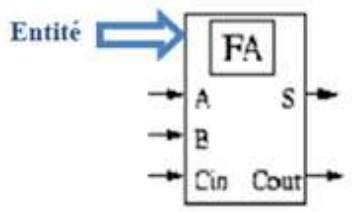

Figure 2.3 – Exemple de syntaxe d'une déclaration d'entité

2.4.2.1 Les ports d'une entité en VHDL

Les signaux d'interface d'une entité constituent, dans la terminologie VHDL, un `port`. Chaque signal du port doit posséder un **nom**, un **mode** et un **type**.

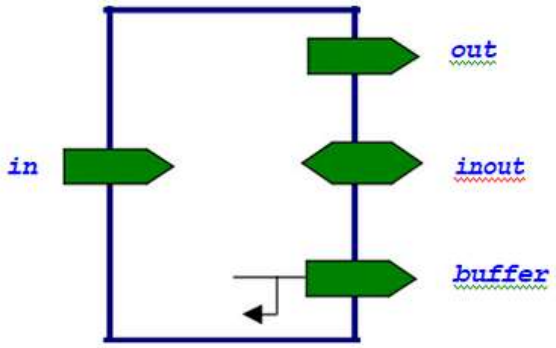

Figure 2.4 – Les quatre modes de signal en VHDL

Le nom du signal : Le nom de chaque signal est choisi par l'utilisateur et reste connu à l'intérieur de toutes les architectures qui font référence à l'entité correspondante. Il est constitué d'une chaîne de caractères alphanumériques qui commence par une lettre et qui peut contenir le caractère souligné (_). Le langage VHDL ne distingue pas les majuscules des minuscules : ainsi `AmI` et `aMi` représentent le même objet.

Le mode du signal : Le mode précise le sens du signal dans l'entité :

- `in` : signal d'entrée ;
- `out` : signal de sortie ;
- `inout` : signal bidirectionnel (entrée/sortie) ;
- `buffer` : signal de sortie pouvant également être lu comme une entrée dans la description.

Le type du signal : Le type détermine l'ensemble des valeurs que peut prendre le signal. Les types les plus couramment utilisés sont :
- `std_logic` pour un signal unique ;
- `std_logic_vector` pour un bus composé de plusieurs signaux.

Les valeurs possibles pour un signal de type `std_logic` ou `std_logic_vector` sont les suivantes :
- `0` : niveau logique bas à basse impédance (masse via faible impédance) ;
- `1` : niveau logique haut à basse impédance (Vcc via faible impédance) ;
- `Z` : niveau logique flottant (entrée déconnectée) ;
- `L` : niveau logique bas à haute impédance (masse via résistance de pull-down) ;
- `H` : niveau logique haut à haute impédance (Vcc via résistance de pull-up) ;
- `W` : niveau logique faible ou inconnu à haute impédance (peut être `L`, `Z` ou `H`) ;
- `X` : niveau logique inconnu (peut être `0`, `L`, `Z`, `H` ou `1`) ;
- `U` : non initialisé (undefined) ;
- `-` : indifférent, correspond à n'importe quel niveau logique (retourne toujours `true` lors d'une comparaison).

2.4.3 Déclaration de larchitecture

L**architecture** décrit le fonctionnement souhaité pour un circuit ou une partie du circuit. En VHDL, le fonctionnement dun circuit est généralement décrit par plusieurs **modules**, chacun correspondant à un couple `ENTITY/ARCHITECTURE`. Dans le cas de simples PLDs (Programmable Logic Devices), on trouve souvent un seul module.

Larchitecture établit, à travers les instructions, les relations entre les entrées et les sorties. Elle peut décrire un fonctionnement :
- purement combinatoire,
- séquentiel,
- ou bien une combinaison des deux (séquentiel et combinatoire).

Une **architecture** est composée de deux zones principales :
- une zone déclarative,
- une zone d'instructions concurrentes.

Les instructions concurrentes sexécutent en parallèle, ce qui signifie que leur ordre décriture ninfluence pas le comportement du circuit.

Trois styles de description peuvent être utilisés en VHDL :
- **Style structurel** : interconnexion de composants, chacun étant une instance dun couple `entité/architecture` ;
- **Style dataflow** : correspond globalement au RTL (Register Transfer Level), cest-à-dire un ensemble dinstructions décrivant les connexions entre portes logiques et registres ;

— **Style comportemental** : basé sur des processus exprimant le comportement du système.

La forme générale dune définition darchitecture est la suivante :

```
architecture nom_de_l_architecture of nom_de_l_entité is
  -- Zone déclarative
begin
  -- Instructions concurrentes
end nom_de_l_architecture;
```

Exemples des styles de description en VHDL

Pour illustrer les différents styles de description, considérons lexemple dun **demi-additionneur** (Half Adder) qui calcule la somme (`S`) et la retenue (`Cout`) à partir de deux bits dentrée `A` et `B`.

1) Style structurel

Dans ce style, larchitecture est décrite comme une **interconnexion de composants**, chacun étant une instance dun couple `entité/architecture` déjà défini.

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

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

architecture Structurel of HalfAdder is
  component xor2
    port (x, y : in std_logic; z : out std_logic);
  end component;

  component and2
    port (x, y : in std_logic; z : out std_logic);
  end component;
begin
  U1: xor2 port map(A, B, S);
  U2: and2 port map(A, B, Cout);
end Structurel;
```

2) Style dataflow (RTL)

Ce style correspond au niveau **Register Transfer Level**. On décrit les relations logiques directes entre les signaux.

```
architecture Dataflow of HalfAdder is
begin
  S    <= A xor B;
  Cout <= A and B;
end Dataflow;
```

3) Style comportemental

Dans ce style, on exprime le fonctionnement sous forme d**algorithmes séquentiels**, à laide de processus et dinstructions conditionnelles.

```
architecture Comportemental of HalfAdder is
begin
  process (A, B)
  begin
    if (A = '0' and B = '0') then
      S    <= '0'; Cout <= '0';
    elsif (A = '0' and B = '1') then
      S    <= '1'; Cout <= '0';
    elsif (A = '1' and B = '0') then
      S    <= '1'; Cout <= '0';
    else
      S    <= '0'; Cout <= '1';
    end if;
  end process;
end Comportemental;
```

Résumé :

- **Structurel** : description comme un schéma, basée sur linstanciation et la connexion de composants.
- **Dataflow (RTL)** : description par équations logiques et transferts de registres.
- **Comportemental** : description algorithmique par processus séquentiels.

2.5 Les types en VHDL

Le langage VHDL est fortement typé : chaque signal, constante ou variable doit être déclaré avec un type. Les principaux types utilisés en synthèse sont les suivants :

2.5.1 Types scalaires

— `bit` : prend les valeurs `0` ou `1`. Exemple :

```
signal clk : bit := '0';
```

— `std_logic` : type défini dans `IEEE.std_logic_1164`, capable de représenter 9 états logiques (0, 1, Z, X, L, H, W, U, -). Exemple :

```
signal a, b : std_logic;
```

— `integer` : valeurs entières dans un intervalle défini. Exemple :

```
signal count : integer range 0 to 255 := 0;
```

— `boolean` : valeurs logiques `true` ou `false`. Exemple :

```
signal flag : boolean := false;
```

2.5.2 Types composés

— `bit_vector` : tableau de bits. Exemple :

```
signal bus1 : bit_vector(7 downto 0);
```

— `std_logic_vector` : tableau de `std_logic`, très utilisé pour les bus. Exemple :

```
signal data : std_logic_vector(15 downto 0);
```

— `signed` et `unsigned` : définis dans `IEEE.numeric_std`, utilisés pour les opérations arithmétiques. Exemple :

```
signal x : unsigned(7 downto 0);
signal y : signed(7 downto 0);
```

Remarque :

— Pour les bus logiques, il est recommandé dutiliser `std_logic_vector`.

— Pour les calculs arithmétiques, il est préférable dutiliser `signed` et `unsigned` avec la bibliothèque `IEEE.numeric_std`.

2.5.3 Le type `time`

En VHDL, le type `time` est utilisé pour représenter une durée ou un instant temporel. Il est principalement employé dans les **testbenchs** pour décrire le comportement temporel des signaux, par exemple la période dune horloge ou un retard (*delay*).

Les unités de temps supportées sont : `fs` (femtoseconde), `ps` (picoseconde), `ns` (nanoseconde), `us` (microseconde), `ms` (milliseconde), `sec`, `min`, `hr`.

Exemple :

```
signal clk_period : time := 10 ns;

clk_process : process
begin
  clk <= '0';
  wait for clk_period/2;
  clk <= '1';
  wait for clk_period/2;
end process;
```

Dans cet exemple, le signal `clk` est une horloge simulée avec une période de 10 ns.

Remarque : Le type `time` est *non synthétisable*, cest-à-dire quil nest pas utilisé pour générer du matériel réel, mais uniquement pour la simulation et la vérification fonctionnelle.

2.5.4 Le type `character`

En VHDL, le type `character` permet de représenter un seul caractère alphanumérique ou symbolique, encadré par des apostrophes. Il est principalement utilisé pour :

— la manipulation de textes ou de chaînes de caractères dans les **testbenchs** ;
— laffichage de messages à laide de `textio` ;
— le stockage de données sous forme symbolique.

Les valeurs possibles couvrent tous les caractères imprimables de la table ASCII (par exemple : `'A'`, `'z'`, `'0'`, `'*'`, `' '`).

Exemples :

```
-- Déclaration d'un signal de type caractère
signal c1 : character := 'A';
signal c2 : character := '0';

-- Exemple d'utilisation avec une variable
variable lettre : character;
lettre := 'Z';
```

```
-- Exemple d'affichage avec textio
write(output, c1); -- affichera 'A'
```

Remarque : Le type `character` est rarement utilisé en synthèse matérielle, mais il est très pratique pour la simulation, le débogage et la génération de fichiers textes.

2.5.5 Le type `string`

En VHDL, le type `string` est défini comme un tableau (`array`) de caractères de type `character`. Il permet de manipuler et stocker des textes (mots, phrases, messages).

Caractéristiques :

- Chaque chaîne est délimitée par des guillemets doubles (`"..."`) ;
- Sa taille (longueur) doit être spécifiée lors de la déclaration, sauf si cest une constante littérale ;
- Il est très utilisé dans les **testbenchs**, notamment avec la bibliothèque `textio`, pour afficher des messages de simulation ou lire/écrire des fichiers texte.

Exemples :

```
-- Déclaration d'une constante string
constant msg : string := "Hello VHDL";

-- Déclaration d'un signal string de taille fixe (10 caractères)
signal s : string(1 to 10);

-- Affectation d'une chaîne
s <= "FPGA_VHDL";

-- Exemple avec textio
write(output, string'("Simulation OK"));
```

Remarque : Comme le type `character`, le type `string` nest pas destiné à la synthèse matérielle, mais il est indispensable pour la **simulation**, le **débogage** et la **génération de rapports texte**.

> **Note :** Les types `character` et `string` sont **prédéfinis dans le standard VHDL** (IEEE 1076). Ils peuvent être utilisés directement sans bibliothèque supplémentaire. Cependant, pour les opérations dentrée/sortie textuelles (lecture, écriture, affichage de messages), il est nécessaire dutiliser la bibliothèque :

```
library std;
use std.textio.all;
```

Celle-ci fournit les procédures `read`, `write`, `writeline`, etc., qui manipulent les chaînes de caractères et les fichiers texte.

2.5.6 Tableau récapitulatif des types en VHDL

Type	Valeurs possibles	Exemple
`bit`	0, 1	`signal clk : bit := '0';`
`std_logic`	0, 1, Z, X, L, H, W, U, -	`signal a : std_logic;`
`integer`	Ensemble dentiers dans un intervalle	`signal count : integer range 0 to 255;`
`boolean`	`true`, `false`	`signal flag : boolean := false;`
`bit_vector`	Tableau de bits (0/1)	`signal bus1 : bit_vector(7 downto 0);`
`std_logic_vector`	Tableau de std_logic	`signal data : std_logic_vector(15 downto 0);`
`unsigned`	Nombres entiers ≥ 0	`signal x : unsigned(7 downto 0);`
`signed`	Nombres entiers signés	`signal y : signed(7 downto 0);`
`time`	Durées avec unités (fs, ps, ns, us, ms, sec, min, hr)	`signal clk_period : time := 10 ns;`
`character`	Tous les caractères ASCII imprimables	`signal c : character := 'A';`
`string`	Tableau de caractères (`character`)	`signal s : string(1 to 10);`

TABLE 2.1 – Résumé des principaux types en VHDL avec exemples

2.5.7 Définir un type utilisateur en VHDL

Le langage VHDL est fortement typé. En plus des types standards (`bit`, `std_logic`, `integer`, `boolean`, ...), il permet à lutilisateur de créer ses propres types pour rendre la

description plus claire, plus structurée et plus adaptée à lapplication à modéliser.

Les types définis par lutilisateur peuvent être de différentes catégories :

— **Type énuméré** : utilisé pour représenter un ensemble fini de valeurs symboliques (souvent des états logiques).
— **Type tableau (array)** : utilisé pour regrouper plusieurs éléments du même type (vecteurs, bus...).
— **Type enregistrement (record)** : permet de regrouper dans une seule structure plusieurs éléments de types différents.

1) Type énuméré

Le type énuméré permet de définir des symboles représentant des états ou des conditions logiques. Il est souvent utilisé dans les machines à états finis (FSM).

```
-- Définition dun type énuméré
type etat is (Idle, Start, Work, Stop);

-- Déclaration dun signal de ce type
signal current_state : etat := Idle;
```

Dans cet exemple, le signal `current_state` peut prendre une des quatre valeurs symboliques (`Idle`, `Start`, `Work`, `Stop`). Cela rend le code plus lisible et plus intuitif que lutilisation de codes binaires.

2) Type tableau (array)

Le type tableau permet de définir un ensemble ordonné déléments du même type. Il est souvent utilisé pour créer des bus de données personnalisés.

```
-- Définition dun tableau de 16 entiers
type tableau_int is array (0 to 15) of integer;

-- Signal de ce type
signal data_bus : tableau_int;
```

On peut également créer des tableaux de `std_logic` ou de `boolean` pour des structures plus complexes.

3) Type enregistrement (record)

Le type `record` regroupe plusieurs champs de types différents au sein dune même structure. Il est souvent utilisé pour modéliser des objets complexes (pixels, paquets de données, etc.).

```
-- Définition dun type record pour un pixel
type pixel is record
  R : std_logic_vector(7 downto 0);
  G : std_logic_vector(7 downto 0);
  B : std_logic_vector(7 downto 0);
end record;

-- Déclaration dun signal de ce type
signal pixel_in : pixel;

-- Exemple d'utilisation
pixel_in.R <= "11111111"; -- Rouge maximal
pixel_in.G <= "00000000";
pixel_in.B <= "00000000";
```

Remarques importantes :

- Les types personnalisés sont déclarés dans la partie déclarative de larchitecture ou dans un `package` pour être réutilisables dans plusieurs entités.
- Les types énumérés sont souvent utilisés pour la conception de FSM (machines à états finis) car ils améliorent la clarté du code.
- Les types `array` et `record` facilitent la modélisation de structures de données complexes (bus, paquets, structures graphiques...).

Note : Les types définis par lutilisateur ne modifient pas le comportement logique du circuit, mais ils permettent de structurer le code, den améliorer la lisibilité et déviter les erreurs de typage.

2.6 Les opérateurs du langage

Le tableau ci-dessous liste les opérateurs standard du langage VHDL ainsi que le type des opérandes manipulés et leur signification. Ces opérateurs permettent de réaliser différentes opérations logiques, relationnelles, arithmétiques ou de décalage sur les signaux et variables.

CATEGORIE	TYPE OPERANDE	SIGNIFICATION
Opérateurs logiques	boolean bit ou bit_vector	`and` : Et `nand` : Non-et `or` : Ou `nor` : Non-ou `xor` : Ou exclusif `xnor` : Non ou exclusif `not` : Non
Opérateurs relationnels	Entrée : Tout type scalaire Résultat : boolean	`=` : Égal `/=` : Non égal `<` : Inférieur `<=` : Inférieur ou égal `>` : Supérieur `>=` : Supérieur ou égal
Opérateurs arithmétiques	integer, real	`+` : addition ou signe positif `-` : soustraction ou signe négatif `*` : multiplication `/` : division `Abs` : valeur absolue `**` : exponentiation `mod` : modulo `rem` : reste
Décalages	bit_vector (amplitude : integer)	`sll` : décalage logique gauche `srl` : décalage logique droit `rol` : rotation gauche `ror` : rotation droite
Autres	bit, bit_vector	`&` : concaténation

En plus des opérateurs standards, VHDL offre un ensemble dopérateurs avancés permettant de manipuler des chaînes, des types énumérés, des vecteurs signés, ainsi que le temps et les conversions de type. Ces opérateurs sont essentiels dans la conception de circuits complexes et la simulation temporelle.

2.6.1 Opérateurs sur les chaînes (String operators)

Opérateur	Type opérande	Signification
`&`	string	Concaténation de chaînes
`=` , `/=`	string	Comparaison dégalité ou de différence
`&`	character + string	Ajoute un caractère au début ou à la fin

Exemple :

```
message <= "Erreur: " & code;
```

—

2.6.2 Opérateurs sur les types énumérés (Enumeration types)

Opérateur	Type opérande	Signification
`=` , `/=` , `<` , `>` , `<=` , `>=`	Tout type énuméré	Comparaison dordre ou dégalité

Exemple :

```
type etat is (idle, start, run, stop);
if etat_actuel = start then ...
```

—

2.6.3 Opérateurs sur les vecteurs signés et non signés

Nécessite la bibliothèque :

```
use IEEE.NUMERIC_STD.ALL;
```

Opérateur	Type opérande	Signification
`+`, `-`, `*`, `/`	signed, unsigned	Opérations arithmétiques
`resize()`	signed, unsigned	Ajuste la taille du vecteur
`to_integer()`	signed, unsigned	Conversion vers entier
`to_signed()`, `to_unsigned()`	integer	Conversion depuis un entier

Exemple :

```
signal A, B : signed(3 downto 0);
signal S : signed(4 downto 0);
S <= resize(A, 5) + resize(B, 5);
```

—

2.6.4 Opérateurs temporels (sur le type time)

Opérateur	Type opérande	Signification
`+`, `-`	time	Addition ou soustraction de durées
`*`, `/`	time, integer	Multiplication ou division par un entier
`after`	Affectation signal	Spécifie un délai de propagation

Exemple :

```
signal clk : std_logic := '0';
clk <= not clk after 10 ns;
```

—

2.6.5 Opérateurs de réduction (non standard, spécifiques à la synthèse)

Opérateur	Exemple	Fonction
`and_reduce`, `or_reduce`, `xor_reduce`	and_reduce(A)	Combine tous les bits dun vecteur par une opération logique

—

2.6.6 Opérateurs de conversion de type

Fonction	Description
`to_integer()`, `to_unsigned()`, `to_signed()`	Conversion entre entiers et vecteurs
`to_bitvector()`, `to_stdlogicvector()`	Conversion entre types de vecteurs binaires
`std_logic'(expr)`	Conversion immédiate dun type vers std_logic

Exemple :

```
signal A : unsigned(3 downto 0);
signal I : integer;
I <= to_integer(A);
```

CHAPITRE A

Annexes

A.1 Acronymes et sigles

Sigle	Définition
PLD	Programmable Logic Device
SPLD	Simple PLD
CPLD	Complex PLD
FPGA	Field Programmable Gate Arrays
CLB	Configurable Logic Block
IOB	Input/Output Block
DSP	Digital Signal Processing
PI	Programmable Interconnect

A.2 Références internes

Figures extraites de *POLYCOPE-FPGA-M2-AUT2-1.pdf* (pages indiquées dans les légendes).