Chapitre 1

Le langage VHDL

1.1 Introduction

En VHDL (VHSIC Hardware Description Language), les unités de conception sont des blocs de code qui permettent de définir la structure et le comportement d'un composant électronique ou d'un système numérique. Les unités de conception sont utilisées pour modéliser différents niveaux d'abstraction, de la description du matériel jusqu'à la description du système complet. Les principales unités de conception en VHDL sont les suivantes :

Entity (Entité) :

L'entité est une unité de conception de haut niveau qui définit l'interface d'un composant ou d'un module. Elle spécifie les ports d'entrée et de sortie, ainsi que le nom de l'entité. L'entité ne décrit pas le comportement interne du composant, elle se contente de définir son interface.

Architecture (Architecture) :

L'architecture est une unité de conception qui spécifie le comportement interne d'une entité. Il existe deux types d'architectures en VHDL : l'architecture structurelle et l'architecture comportementale. L'architecture structurelle décrit la connexion des composants internes, tandis que l'architecture comportementale décrit le comportement du composant en utilisant des descriptions de processus et de signaux.

Package (Paquet) :

Un paquet est une unité de conception qui permet de regrouper des déclarations de types, de constantes, de fonctions et de procédures qui peuvent être réutilisées dans plusieurs parties du code VHDL. Les paquets facilitent la modularité et la réutilisation du code.

Configuration (Configuration) :

Une configuration est une unité de conception qui permet de spécifier comment différentes entités sont interconnectées pour former un système complet. Elle associe des entités à des architectures spécifiques et définit les liaisons entre les ports.

Package Body (Corps de paquet) :

Le corps de paquet est utilisé pour définir le comportement des fonctions et des procédures déclarées dans un paquet. Il sert à implémenter les fonctionnalités définies dans le paquet.

Testbench (Banc de tests) :

Bien que ce ne soit pas une unité de conception au sens strict, un banc de tests est essentiel pour vérifier le comportement d'une conception VHDL. Il consiste en un ensemble de stimuli et de vérifications qui permettent de tester le composant ou le système.
En utilisant ces unités de conception, vous pouvez modéliser des systèmes électroniques de manière structurée et hiérarchique en VHDL, ce qui facilite la conception, la simulation et la vérification de vos projets.

1.2 Les niveaux de description

En VHDL (VHSIC Hardware Description Language), il existe plusieurs niveaux de description qui vous permettent de modéliser un système électronique à différents niveaux d'abstraction. Ces niveaux de description sont généralement utilisés pour représenter les différents aspects d'un système électronique. Voici les niveaux de description couramment utilisés en VHDL :

1.2.1 Niveau de l'abstraction comportementale

Comportemental :

Ce niveau de description se concentre sur le comportement global du système sans détailler sa structure interne. Il est souvent utilisé pour décrire le comportement algorithmique d'un composant ou d'un système. Les descriptions comportementales utilisent généralement des processus et des assignations pour spécifier le comportement.

1.2.2 Niveau de l'abstraction structurelle

Ce niveau de description se concentre sur la structure interne du système en spécifiant comment les composants sont interconnectés. Il est utilisé pour décrire comment les différents éléments matériels sont liés les uns aux autres pour former un système complet.

1.2.3 Niveau de l'abstraction de la porte logique

Ce niveau de description se concentre sur la représentation du système en utilisant des portes logiques et des interconnexions. Il est utilisé pour une modélisation de très bas niveau et est rarement utilisé dans la conception moderne de systèmes numériques.

1.2.4 Niveau de l'abstraction de la commutation

Ce niveau de description est encore plus bas que le niveau de la porte logique et se concentre sur la modélisation des éléments de commutation individuels tels que les transistors et les interconnexions. Il est rarement utilisé dans la conception VHDL, sauf dans le cadre de la conception de circuits intégrés personnalisés (ASIC).
Lors de la conception d'un système électronique en VHDL, vous pouvez choisir le niveau de description qui convient le mieux à votre objectif. Par exemple, vous pouvez commencer par une description comportementale pour définir le comportement souhaité du système, puis passer à une description structurelle pour spécifier comment les composants sont interconnectés. La conception VHDL est souvent réalisée en utilisant une approche de conception hiérarchique, où les différents niveaux de description sont imbriqués pour représenter le système de manière structurée et modulaire.

1.3 Organisation en bibliothèque

En VHDL, l'organisation en bibliothèques est une technique importante pour gérer et organiser vos différentes unités de conception, packages et autres éléments de code. Les bibliothèques permettent de regrouper et de catégoriser les différents éléments de votre projet VHDL, ce qui facilite la gestion, la réutilisation et la collaboration sur le code. Voici comment organiser vos designs en bibliothèques en VHDL :

Créez une bibliothèque :

1. Vous pouvez créer une bibliothèque en utilisant la directive 'library' suivie du nom de la bibliothèque, suivi de 'is' (figure 1.1).

2. Ajoutez des fichiers source à la bibliothèque : Une fois que vous avez créé une biblio-

```
library my_lib is
end library;
```

FIGURE 1.1 – Library

thèque, vous pouvez y ajouter des fichiers source VHDL. Ces fichiers peuvent contenir des entités, des architectures, des packages, des configurations, etc.

Utilisez la directive 'use' pour lier un fichier source à une bibliothèque spécifique (figure 1.2).

3. Créez des éléments dans la bibliothèque : Une fois que vous avez ajouté des fichiers

```
library my_lib;
use my_lib.my_package.all;
```

FIGURE 1.2 – Library utilisant use

source à une bibliothèque, vous pouvez y créer des éléments tels que des entités, des architectures, des packages, etc. Pour ce faire, utilisez le nom de la bibliothèque comme préfixe (Figure 3).

Notez que 'work' est une bibliothèque spéciale en VHDL qui représente la bibliothèque de travail par défaut. Vous pouvez également créer des éléments directement dans la bibliothèque de travail si vous ne spécifiez pas de bibliothèque.

```
my_lib.entity_name : entity work.entity_name ( ... );
```

FIGURE 1.3 – Formes de Library

Utilisez les éléments de la bibliothèque :
Une fois que vous avez créé des éléments dans une bibliothèque, vous pouvez les utiliser dans d'autres parties de votre code VHDL en faisant référence à la bibliothèque et à l'élément spécifique. Par exemple : L'organisation en bibliothèques en VHDL vous per-

```
entity_name : my_lib.entity_name port map ( ... );
```

FIGURE 1.4 – Formes de Library

met de gérer efficacement les dépendances entre les différents éléments de votre projet, de réutiliser du code dans plusieurs projets et de créer une structure modulaire pour votre conception. Elle est particulièrement utile dans les projets VHDL complexes où de nombreux fichiers et éléments doivent être gérés. Le langage VHDL (VHSIC Hardware Description Language) est un langage de description matériel (HDL) utilisé pour modéliser et concevoir des systèmes électroniques. Voici une liste des principaux éléments du langage VHDL :
1. Entités (Entities) : Les entités définissent l'interface d'un composant électronique en spécifiant ses ports d'entrée et de sortie. Elles servent à décrire la structure externe du composant.
2.Architectures (Architectures) : Les architectures définissent le comportement interne d'une entité. Il peut y avoir plusieurs architectures pour une même entité, chacune décrivant une implémentation différente du même composant.
3. Signaux (Signals) : Les signaux représentent des valeurs qui circulent à l'intérieur d'une architecture. Ils peuvent être utilisés pour transmettre des données entre les différents éléments d'une conception.
4. Processus (Processes) : Les processus décrivent le comportement séquentiel ou concurrentiel à l'intérieur d'une architecture. Ils sont souvent utilisés pour définir le comporte-

ment d'un composant en réponse à des événements ou des conditions.

5. Affectations (Assignments) : Les affectations permettent de définir la valeur d'un signal à un instant donné. Les affectations peuvent être utilisées pour mettre à jour les signaux en fonction de leur comportement.

6. Types de données (Data Types) : VHDL propose différents types de données, notamment les types scalaires (entiers, réels, booleans), les types composites (tableaux, enregistrements) et les types d'énumération. Vous pouvez également définir vos propres types personnalisés.

7. Packages (Packages) : Les packages sont des regroupements de déclarations de types, de fonctions, de procédures et de constantes qui peuvent être réutilisés dans plusieurs parties du code VHDL. Ils facilitent la modularité et la réutilisation du code.

8. Fonctions (Functions) : Les fonctions sont des blocs de code qui effectuent des calculs et renvoient une valeur. Elles peuvent être utilisées pour effectuer des opérations sur les données dans une architecture.

9. Procédures (Procedures) : Les procédures sont similaires aux fonctions, mais elles ne renvoient pas de valeur. Elles sont utilisées pour effectuer des opérations de traitement dans une architecture.

10. Configuration (Configuration) : Les configurations sont utilisées pour spécifier comment différentes entités sont interconnectées pour former un système complet. Elles sont souvent utilisées pour définir l'architecture de haut niveau d'une conception.

11. Instantiations (Instantiations) : Les instantiations permettent de créer des instances de composants (entités) dans une architecture. Cela permet de réutiliser des composants existants dans une nouvelle conception.

12. Commentaires (Comments) : Les commentaires sont utilisés pour documenter le code VHDL et fournir des informations sur son fonctionnement. Les commentaires sont ignorés par le compilateur.

Ces éléments de base du langage VHDL sont utilisés pour décrire et modéliser des systèmes électroniques à différents niveaux d'abstraction, de la spécification du comportement à la description de la structure interne.

En VHDL (VHSIC Hardware Description Language), les objets sont des éléments du langage qui permettent de stocker et de manipuler des données. Les objets sont utilisés pour représenter des signaux, des variables, des constantes et d'autres types de données dans

une conception VHDL. Voici les principaux types d'objets en VHDL :

1. Signaux (Signals) : Les signaux sont des objets qui représentent des valeurs qui circulent à l'intérieur d'une architecture. Ils sont utilisés pour transmettre des données entre les différents éléments d'une conception. Les signaux ont une durée de vie plus longue que les variables et sont utilisés pour modéliser la propagation des données dans le temps.

2. Variables (Variables) : Les variables sont des objets qui stockent des données temporaires à l'intérieur d'un processus ou d'une procédure. Contrairement aux signaux, les variables sont locales à un processus donné et ne sont pas utilisées pour la communication entre les processus. Les variables sont souvent utilisées pour effectuer des calculs temporaires.

3. Constantes (Constants) : Les constantes sont des objets dont la valeur ne peut pas être modifiée après leur déclaration. Elles sont utilisées pour représenter des valeurs constantes dans une conception, telles que des paramètres de configuration ou des coefficients de filtre.

4. Type (Type) : Les types sont des objets qui définissent la nature des données stockées dans les signaux, les variables et les constantes. VHDL propose plusieurs types de données prédéfinis, tels que les entiers, les réels, les booleans, les tableaux, les enregistrements, etc. Vous pouvez également définir vos propres types personnalisés.

5. Files (Files) : Les files sont des objets qui permettent de stocker des données de manière dynamique. Les files sont similaires aux tableaux, mais leur taille peut varier pendant l'exécution. Elles sont principalement utilisées pour la manipulation de données dans des simulations.

6. Accès (Access) : Les objets d'accès sont utilisés pour créer des références à des objets existants. Ils sont souvent utilisés pour passer des données de manière efficace à l'intérieur et à l'extérieur des processus, en évitant la copie de données.

7. Variables de temps (Time Variables) : Les variables de temps sont utilisées pour représenter des valeurs de temps dans VHDL. Elles sont utilisées pour définir des délais, des horloges et d'autres opérations liées au temps. Ces objets du langage VHDL servent à représenter et à manipuler les données dans une conception VHDL. Ils sont utilisés dans la description du comportement, de la structure et de la synchronisation des systèmes électroniques. Chaque objet a ses propres règles de portée et de durée de vie, ce qui les

rend adaptés à des utilisations spécifiques dans le modèle de conception VHDL
En VHDL (VHSIC Hardware Description Language), les données sont généralement classées en deux catégories principales : les types de données scalaires et les types de données composites. Voici une brève description de chacune de ces catégories :
Types de données scalaires : Bit : Le type de données le plus élémentaire, représentant un seul bit pouvant avoir les valeurs 0 ou 1.
Boolean : Un type de données logiques avec les valeurs "true" ou "false".
Integer : Représente un nombre entier signé.
Natural : Représente un nombre entier non négatif (0 et les entiers positifs).
Positive : Représente un nombre entier positif (1 et les entiers positifs).
Real : Représente un nombre à virgule flottante.
Time : Représente une durée temporelle.
Types de données composites : Array : Permet de définir des tableaux unidimensionnels ou multidimensionnels de types scalaires ou composites.
Record : Permet de définir une structure de données composée de plusieurs champs de différents types.
Enumeration : Permet de créer un ensemble fini de valeurs possibles pour une variable (par exemple, les jours de la semaine).
Access : Permet de définir des pointeurs vers des objets dans la mémoire.
En VHDL, vous pouvez également créer vos propres types de données personnalisés à l'aide de types scalaires et composites existants, ce qui permet une grande flexibilité dans la définition de types de données adaptés à votre application. Ces types de données sont utilisés pour déclarer des signaux, des variables, des constantes, des paramètres de fonctions, etc., dans vos conceptions VHD.
En VHDL, la modélisation par paramètres génériques permet de créer des composants réutilisables en paramétrant leur comportement à l'aide de valeurs génériques lors de leur instantiation. Les génériques permettent de personnaliser les caractéristiques d'un composant sans avoir à modifier sa définition interne. Voici comment utiliser les génériques en VHDL :
1. **Déclaration de génériques** : Vous pouvez déclarer des génériques dans la déclaration d'un composant, d'une entité ou d'un package en utilisant le mot-clé generic. Par exemple :

2. **Utilisation des génériques dans une instance** : Lorsque vous instanciez un com-

```
entity MonComposant is
  generic (
    largeur : integer := 8;   -- Un paramètre générique de largeur avec une
    profondeur : integer := 16  -- Un autre paramètre générique de profonde
  );
  port (
    -- Déclaration des ports
  );
end entity MonComposant;
```

FIGURE 1.5 – Generic

posant ou une entité, vous pouvez spécifier les valeurs des génériques à utiliser pour cette instance particulière. Par exemple : Ici, l'instance instance1 de MonComposant uti-

```
instance1 : MonComposant generic map (
  largeur => 4,      -- Vous spécifiez une valeur différente pour 'largeur'
  profondeur => 32  -- Vous spécifiez une valeur différente pour 'profondeu
)
port map (
  -- Connexions de ports
);
```

FIGURE 1.6 – Instantiation

lise des valeurs spécifiques pour les génériques largeur et profondeur, qui remplacent les valeurs par défaut définies dans la déclaration de l'entité.

3. **Utilisation des génériques dans le code interne** : À l'intérieur du code du composant ou de l'entité, vous pouvez utiliser les valeurs des génériques pour paramétrer le comportement du composant. Par exemple, vous pouvez utiliser les génériques pour déterminer la taille des tableaux ou la résolution d'un composant.

4. **Passage de génériques de niveau supérieur** : Vous pouvez également définir des génériques au niveau supérieur de votre conception, par exemple dans un fichier de test,

pour personnaliser le comportement global de votre système en fonction des besoins.
Les génériques sont un outil puissant pour la création de composants réutilisables et paramétrables en VHDL, ce qui simplifie la conception et favorise la modularité.
Les sous programmes : En VHDL, les sous-programmes sont des blocs de code réutilisables qui permettent d'effectuer des opérations spécifiques de manière modulaire. Les deux principaux types de sous-programmes en VHDL sont les procédures et les fonctions.
Voici une explication plus détaillée de chaque type de sous-programme :
1. **Procédures** : Une procédure est une unité de code autonome qui peut effectuer une série d'instructions. Les procédures sont principalement utilisées pour effectuer des actions ou des opérations, mais elles ne renvoient pas de valeur en sortie.
Une procédure peut avoir des paramètres d'entrée (arguments) qui permettent de passer des valeurs à la procédure pour être utilisées dans son exécution. Exemple de déclaration de procédure en VHDL :
2. **Fonctions** : Une fonction est également une unité de code autonome, mais contraire-

```
procedure MaProcedure(Argument1 : in integer; Argument2 : out integer) is
begin
  -- Corps de la procédure
  Argument2 := Argument1 * 2;
end procedure MaProcedure;
```

FIGURE 1.7 – Procedure

ment aux procédures, les fonctions renvoient une valeur en sortie.
Les fonctions sont utilisées pour effectuer des calculs ou des opérations et renvoyer un résultat à l'endroit où elles sont appelées.
Comme les procédures, les fonctions peuvent avoir des paramètres d'entrée (arguments).
Exemple de déclaration de fonction en VHDL (voir figure 1.8). 3. **Appel de sous-programmes** : Pour appeler une procédure ou une fonction en VHDL, vous utilisez le nom de la procédure ou de la fonction suivi des arguments appropriés. Vous pouvez stocker la valeur renvoyée par une fonction dans une variable ou l'utiliser directement dans votre code.
Exemple d'appel de procédure et de fonction en VHDL (Figure 1.9).

```
function MaFonction(Argument1 : in integer) return integer is
  variable Resultat : integer;
begin
  -- Corps de la fonction
  Resultat := Argument1 * 2;
  return Resultat;
end function MaFonction;
```

FIGURE 1.8 – Fonctions

```
variable Entree : integer := 5;
variable Sortie : integer;

MaProcedure(Entree, Sortie); -- Appel de la procédure
Resultat := MaFonction(Entree); -- Appel de la fonction et stockage du résu
```

FIGURE 1.9 – Appel de sous programmes

Structure de base : un fichier VHDL est divisé en plusieurs sections, y compris la bibliothèque, l'entité, l'architecture et le corps. L'entité décrit les entrées et les sorties du circuit, tandis que l'architecture définit le comportement du circuit. Types de données : le VHDL prend en charge plusieurs types de données, notamment les types logiques (bits et vecteurs), les types entiers, les types à virgule flottante, les types enumérés et les types tableaux.

Conception de circuits : le VHDL permet de décrire le comportement d'un circuit numérique en utilisant des instructions logiques telles que AND, OR, XOR et NOT. Il prend également en charge des opérateurs de comparaison tels que l'égalité, l'inégalité et les opérateurs de comparaison arithmétique.

Simulations : le VHDL permet de simuler le comportement d'un circuit numérique avant de le synthétiser dans un FPGA ou un ASIC. Les simulateurs VHDL permettent de vérifier le fonctionnement du circuit, d'optimiser les performances et de détecter les erreurs de conception. Synthèse : la synthèse VHDL permet de transformer une description de circuit en VHDL en un circuit numérique réel dans un FPGA ou un ASIC. Les

outils de synthèse VHDL convertissent la description de circuit en un ensemble de portes logiques et de connexions pour réaliser le circuit numérique.
En conclusion, la programmation VHDL est une compétence essentielle pour les concepteurs de circuits numériques travaillant avec des FPGA, ASIC et CPLD. Le VHDL permet de décrire le comportement d'un circuit numérique, de simuler son fonctionnement et de le synthétiser dans un circuit numérique réel. Les sous-programmes en VHDL sont utiles pour organiser et réutiliser du code, améliorer la lisibilité et la maintenance, et encapsuler des fonctionnalités spécifiques. Ils sont largement utilisés dans la conception de circuits intégrés programmables (FPGA) et de systèmes embarqués.
La simulation fonctionnelle des circuits en VHDL est essentielle pour vérifier et valider le comportement d'un design avant de le synthétiser et de le mettre en place sur un FPGA ou un ASIC. Un test-bench en VHDL est un ensemble de code VHDL qui permet de simuler et de tester le comportement d'un composant ou d'un système VHDL. Voici comment créer un test-bench en VHDL :

1. Créez un fichier de test-bench :

Vous pouvez créer un fichier VHDL distinct pour votre test-bench ou le placer dans le même fichier que votre conception principale. Traditionnellement, les test-benches sont placés dans un fichier distinct pour maintenir la séparation entre la conception et les simulations.

2. Déclarez une entité de test-bench :

Dans le fichier de test-bench, déclarez une entité spéciale pour le test-bench. Cette entité ne représente pas un composant réel, mais elle est utilisée pour définir les signaux d'entrée, les signaux de sortie et les constantes nécessaires pour votre simulation.
Exemple de déclaration d'entité de test-bench :

```
entity TestBench is
end entity TestBench;
```

FIGURE 1.10 – Library

***3. Déclarez les signaux de test* :**

Dans la section de l'architecture du test-bench, déclarez des signaux qui seront utilisés pour fournir des données d'entrée à votre design et pour collecter les données de sortie. Exemple de déclaration de signaux de test :

```
signal EntréeA, EntréeB : integer;
signal Sortie : integer;
```

FIGURE 1.11 – Library

4. Instanciez votre design :

Instanciez votre conception principale (le composant que vous souhaitez tester) dans l'architecture du test-bench. Cela vous permettra de connecter les signaux de test à votre design. Exemple d'instanciation de votre design dans le test-bench :

```
UUT: entity work.VotreDesign
  port map (
    EntréeA => EntréeA,
    EntréeB => EntréeB,
    Sortie => Sortie
  );
```

FIGURE 1.12 – Test-bench

5. Générez des stimuli :

Dans la section de l'architecture du test-bench, générez des valeurs pour les signaux d'entrée de votre design. Vous pouvez utiliser des processus VHDL pour contrôler les valeurs des signaux au fil du temps (Voir figure 1.13).

6. Capturez les résultats :

Utilisez un processus pour capturer les données de sortie de votre design et les comparer avec les résultats attendus. Exemple de capture des résultats (Figure 1.14)

```
process
begin
  EntréeA <= 3;  -- Valeur d'entrée A à t=0
  EntréeB <= 4;  -- Valeur d'entrée B à t=0
  wait for 10 ns;
  EntréeA <= 5;  -- Nouvelle valeur d'entrée A à t=10 ns
  EntréeB <= 2;  -- Nouvelle valeur d'entrée B à t=10 ns
  -- ...
  wait;
end process;
```

FIGURE 1.13 – Génération de valeurs

```
process
begin
  wait for 20 ns;  -- Attendre un certain temps pour la simulation
  assert Sortie = 8
    report "Test failed. Expected output: 8"
    severity error;
  wait;
end process;
```

FIGURE 1.14 – Capture de résultats

7. Exécutez la simulation :

Compilez votre conception principale et le test-bench, puis exécutez la simulation à l'aide de votre outil de simulation VHDL préféré, tel que ModelSim ou Xilinx.
Un test-bench en VHDL permet de simuler le comportement de votre design sous diverses conditions et d'assurer qu'il fonctionne correctement avant de le mettre en œuvre matériellement. Il est essentiel pour la validation et le débogage de vos conceptions VHDL.

Exercice 1 :

Créez un nouveau projet VHDL dans votre environnement de conception (par exemple, Quartus Prime, Xilinx Vivado, ou un autre outil VHDL de votre choix).
- Créez un module VHDL nommé Parite Checker qui prendra un nombre en entrée et renverra un signal pour indiquer si le nombre est pair ou impair. Le module doit avoir les spécifications suivantes :

- Une entrée "Nombre" de type integer.

- Une sortie "Parite" de type boolean, où true indique que le nombre est pair et false indique qu'il est impair.

Écrivez le code VHDL pour le module Parite Checker en utilisant une architecture comportementale. Le code doit effectuer les étapes suivantes :

- Vérifier si le nombre est pair ou impair.

- Affecter la valeur correspondante à la sortie "Parite".

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

entity Parite_Checker is
  port (
    Nombre : in integer;
    Parite : out boolean
  );
end entity Parite_Checker;

architecture Comportementale of Parite_Checker is
begin
  process(Nombre)
  begin
    if Nombre mod 2 = 0 then
      Parite <= true;  -- Le nombre est pair
    else
      Parite <= false; -- Le nombre est impair
    end if;
  end process;
end architecture Comportementale;
```

FIGURE 1.15 – check parite

***Exercice 1* : Types de données en VHDL**

Créez un module VHDL qui définit une entité avec les spécifications suivantes :

Une entrée "EntierA" de type entier.

Une entrée "EntierB" de type entier.

Une sortie "Resultat" de type réel.

Le module doit additionner "EntierA" et "EntierB" et renvoyer le résultat en tant que réel.

```
entity Addition_Entier_Reel is
  port (
    EntierA : in integer;
    EntierB : in integer;
    Resultat : out real
  );
end entity Addition_Entier_Reel;

architecture Comportementale of Addition_Entier_Reel is
begin
  process (EntierA, EntierB)
  begin
    Resultat <= real(EntierA) + real(EntierB);
  end process;
end architecture Comportementale;
```

FIGURE 1.16 – Solution exo1

Exercice 2 : Opérations logiques

Créez un module VHDL qui définit une entité avec les spécifications suivantes :

Une entrée ’A’ de type bit.

Une entrée ’B’ de type bit.

Une sortie ’Resultat’ de type bit.

Le module doit effectuer l’opération logique "ET" entre les entrées "A" et "B" et renvoyer le résultat.

Exercice 3 : Multiplexeur 2 vers 1

Créez un module VHDL qui définit une entité avec les spécifications suivantes :

Deux entrées "Donnee0" et "Donnee1" de type bit.

Une entrée "Selection" de type bit.

Une sortie "Sortie" de type bit.

Le module doit implémenter un multiplexeur 2 vers 1, de sorte que si "Selection" est à 0,

```
entity Operation_ET is
  port (
    A, B : in bit;
    Resultat : out bit
  );
end entity Operation_ET;

architecture Comportementale of Operation_ET is
begin
  process (A, B)
  begin
    Resultat <= A and B;
  end process;
end architecture Comportementale;
```

FIGURE 1.17 – Solution exo2

"Sortie" soit égale à "Donnee0", sinon "Sortie" soit égale à "Donnee1".

Exercice 4 : Compteur asynchrone

Créez un module VHDL qui définit une entité avec les spécifications suivantes :
Une entrée "Horloge" de type bit.
Une entrée "RemiseAZero" de type bit.
Une sortie "Compteur" de type entier.
Le module doit implémenter un compteur asynchrone qui compte de 0 à 15 et revient à 0 lorsque "RemiseAZero" est actif. Le compteur doit être incrémenté à chaque front montant de l'horloge.

Exercice 5 : Convertisseur binaire vers décimal

Créez un module VHDL qui définit une entité avec les spécifications suivantes :
Quatre entrées "Bit3", "Bit2", "Bit1" et "Bit0" de type bit.
Une sortie "Decimal" de type entier.
Le module doit convertir le nombre binaire représenté par les entrées en sa forme décimale correspondante et le renvoyer sur la sortie "Decimal".

```
entity Multiplexeur_2_1 is
  port (
    Donnee0, Donnee1 : in bit;
    Selection : in bit;
    Sortie : out bit
  );
end entity Multiplexeur_2_1;

architecture Comportementale of Multiplexeur_2_1 is
begin
  process (Donnee0, Donnee1, Selection)
  begin
    if Selection = '0' then
      Sortie <= Donnee0;
    else
      Sortie <= Donnee1;
    end if;
  end process;
end architecture Comportementale;
```

FIGURE 1.18 – Solution exo3

```
entity Compteur_Asynchrone is
  port (
    Horloge, RemiseAZero : in bit;
    Compteur : out integer range 0 to 15
  );
end entity Compteur_Asynchrone;

architecture Comportementale of Compteur_Asynchrone is
  signal Compteur_Interne : integer range 0 to 15 := 0;
begin
  process (Horloge, RemiseAZero)
  begin
    if RemiseAZero = '1' then
      Compteur_Interne <= 0;
    elsif rising_edge(Horloge) then
      if Compteur_Interne = 15 then
        Compteur_Interne <= 0;
      else
        Compteur_Interne <= Compteur_Interne + 1;
      end if;
    end if;
  end process;

  Compteur <= Compteur_Interne;
end architecture Comportementale;
```

FIGURE 1.19 – Solution exo4

```
entity Convertisseur_Binaire_Vers_Decimal is
  port (
    Bit3, Bit2, Bit1, Bit0 : in bit;
    Decimal : out integer range 0 to 15
  );
end entity Convertisseur_Binaire_Vers_Decimal;

architecture Comportementale of Convertisseur_Binaire_Vers_Decimal is
begin
  process (Bit3, Bit2, Bit1, Bit0)
  begin
    Decimal <= (to_integer(unsigned(Bit3 & Bit2 & Bit1 & Bit0)));
  end process;
end architecture Comportementale;
```

FIGURE 1.20 – Solution exo5