Université Mohamed Khider de Biskra
Faculté des Sciences et de la Technologie
Département de génie électrique

MÉMOIRE DE MASTER

Sciences et Technologies
Electronique
Réseaux et télécommunications

Réf. : .....

Présenté et soutenu par :
AOUICHE Mounir

Le : lundi 08 juillet 2019

Conception et implémentation d'un Microcontrôleur de 64 bits sur FPGA

Jury :

Mr.	ARIF Ali	MCA	Université de Biskra	Président
Mr.	DHIABI Fathi	MCB	Université de Biskra	Encadreur
Mr.	GUETTAF Abderrazak	MCA	Université de Biskra	Examinateur

Année universitaire : 2018 - 2019

الجمهورية الجزائرية الديمقراطية الشعبية
République Algérienne Démocratique et Populaire
وزارة التعليم العالي و البحث العلمي
Ministère de l'enseignement Supérieur et de la recherche scientifique

Université Mohamed Khider Biskra
Faculté des Sciences et de la Technologie
Département de Génie Electrique
Filière : Electronique

Option : Réseaux et télécommunications

Mémoire de Fin d'Etudes
En vue de l'obtention du diplôme:

MASTER

Thème

Conception et implémentation d'un Microcontrôleur de 64 bits sur FPGA

Présenté par :
AOUICHE Mounir

Avis favorable de l'encadreur :
DHIABI Fathi

Avis favorable du Président du Jury

.....

Cachet et signature

الجمهورية الجزائرية الديمقراطية الشعبية
République Algérienne Démocratique et Populaire
وزارة التعليم العالي و البحث العلمي
Ministère de l'enseignement Supérieur et de la Recherche scientifique

Université Mohamed Khider Biskra
Faculté des Sciences et de la Technologie
Département de Génie Electrique
Filière : Electronique

Option : Réseaux et télécommunication

Thème :

Conception et implémentation d'un Microcontrôleur de 64 bits sur FPGA

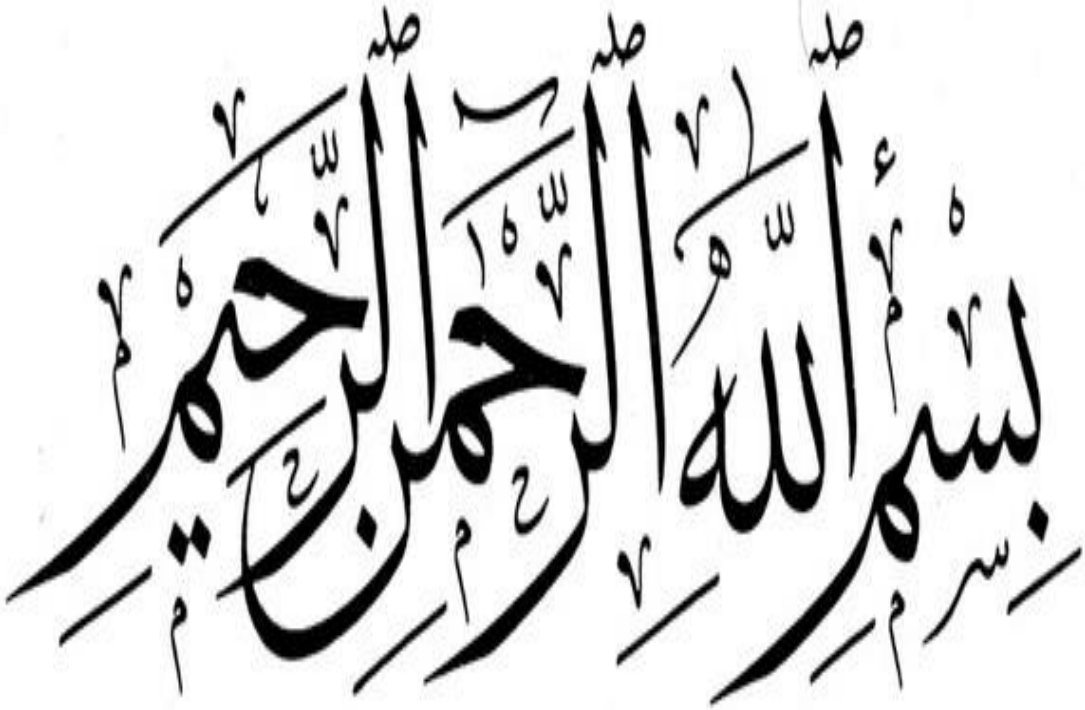
Proposé par : DHIABI FATHI
Dirigé par : DHIABI FATHI

Résumé

Le travail que nous présentons dans ce mémoire vise la transposition de l'architecture et de la fonction du microcontrôleur de 64 bits vers une architecture implantable sur FPGA du type CYCLONE II de ALTERA. Afin réaliser la conception et modélisation de ce microcontrôleur, et mettre en œuvre quelques fonctions de ses composants internes, on a utilisé le langage de description VHDL après avoir expliqué ses notions et ses fonctionnalités. Après, on a implémenté les programmes de quelques fonctions des composants du µC 64 bits sur FPGA dans l'environnement Quartus II, qui est un logiciel développé par la société Altéra permettant la gestion complète d'un flot de conception.

ملخص

يهدف العمل المقدم في هذه المذكرة للتخرج إلى تحويل بنية و وظيفة الميكرومراقب 64 بت إلى بنية قابلة للتنفيذ على الدارة المنطقية القابلة للبرمجة FPGA من النوع سيكلون 2 من شركة ALTERA. من أجل تحقيق تصميم ووصف هذا الميكرومراقب 64 بت ، و لتنفيذ بعض وظائف مكوناته الداخلية ، استخدمنا لغة الوصف VHDL بعد شرح مفاهيمها وميزاتها، ثم قمنا بتنفيذ برامج لبعض وظائف مكونات الميكرومراقب 64 بت على FPGA في بيئة Quartus II وهو برنامج تم تطويره من قبل شركة Altéra مما يسمح في التحكم الشامل للتصميم.


بسم الله الرحمن الرحيم

Dédicaces

Je dédie ce modeste mémoire de Master :

À ma mère et mon père

À mes frères et mes sœurs

À mon épouse et mes chers enfants

À toute ma famille et à tous mes amis

A mon encadreur Mr DHIABI Fathi

AOUICHE
Mounir

Remerciements

Avant tout, je remercie ***Allah*** *le tout puissant et miséricordieux, qui ma a donné le courage, la force et la foi de mener à terme ce modeste travail.*

Je tiens d'abord à remercier très chaleureusement mon encadreur Monsieur ***DHIABI Fathi*** *d'avoir accepté de m' encadrer pour mon projet ainsi que pour son soutien, ses remarques pertinentes, son encouragement, sa disponibilité, sa confiance et les conseils précieuses durant la réalisation de ce projet.*

Je remercie également ***les membres du jury*** *monsieur* ***ARIF Ali*** *et monsieur* ***GUETTAF Abderrazak*** *de m'avoir honoré en acceptant de juger et évaluer mon mémoire, et de me faire part de leurs précieuses remarques.*

Je voudrais remercier tous ceux qui m'ont aidé et encouragé de près ou de loin

Liste des figures

Liste des Abréviations

A

ASIC : Application **S**pecific **I**ntegrated **C**ircuit.

C

CLB : Configurable **L**ogic **B**lock

CISC : Complex **I**nstructions **S**et **C**onstruction.

CPU: **C**entral **P**rocessing **U**nit

CPLD : **C**omplex **P**rogrammable **L**ogic **D**evice

D

DSP : Digital **S**ignal **P**rocessor

E

EPROM : Erasable **P**rogrammable **R**ead **O**nly **M**emory

EEPROM : Electricall**y-E**rasable **P**rogrammable **R**ead **O**nly **M**emory

F

FPGA : Field **P**rogrammable **G**ate **A**rrays

I

IEEE : Institut of **E**lectrical and **E**lectronics **E**ngineers

IOB : Input **O**utput **B**lock

J

JTAG : Joint **T**est **A**ction **G**roup

L

LCA : Logic **C**ell **A**rray

LC : Logic **C**ell.

LE : Logic **E**lement.

LUT : Look-**U**p-**T**able

P

PLD : Programmable **L**ogic **D**evice.

R

RAM : Random **A**ccess **M**emoy

ROM : Read **O**nly **M**emory

RISC : Reduced **I**nstructions **S**et **C**onstruction.

RTL : Register **T**ransfert **L**anguage.

S

SRAM : Static **R**andom **A**ccess **M**emory

U

UAL : Unité **A**rithmétique et **L**ogique.

UC : Unité de **C**ontrôle

V

VHDL : VHSIC **H**ardware **D**escription **L**anguage.

VHSIC : Very **H**igh **S**cale **I**ntegrated **C**ircuit.

VLSI : Very **L**arge **S**cale **I**ntegration

SOMMAIRE

Chapitre 2 : Le circuit FPGA

Chapitre 3 : Notice QURATUS

Chapitre 4 : Le Microcontrôleur

INTRODUCTION GENERALE

Introduction générale

Les années passées ont vu l'explosion du marché des systèmes embarqués dans de nombreux domaines industriels comme par exemple les télécommunications, les satellites, les appareils médicaux, l'électroménager…etc. Ces besoins de plus en plus importants génèrent une compétition industrielle féroce où les facteurs comme le coût, les performances deviennent prépondérants pour le succès d'un produit. [**1**]

Dans ce contexte, le FPGA (Field Programmable Gate Array) avec ses grandes capacités d'intégration et de reconfiguration en font un composant clé.

Réduire les coûts, augmenter les performances et la fiabilité d'un circuit, diminuer les délais de mise en place d'un prototype opérationnel, réduire l'encombrement des composants sur circuit, sont des atouts conjugués dans un FPGA. Ces atouts justifient le succès de ces composants dans tous les domaines d'applications de l'électronique. [**1**]

Dans l'objectif d'encourager la large diffusion de ce type de circuits, il est nécessaire d'améliorer les environnements de développement pour les rendre plus accessibles. Aussi, il est possible de tester rapidement la validité de concepts architecturaux nouveaux: l'implémentation complète d'un microcontrôleur sur des circuits FPGA est aujourd'hui à notre portée, entrainant ainsi plus de possibilités d'évaluation que celle offertes par des simulateurs logiciels.

L'étude que nous présentons dans ce mémoire vise la transposition de l'architecture de la fonction de µC 64 bits vers une architecture implantable sur FPGA du type CYCLONE II de ALTERA. L'objectif est de garder la souplesse et la flexibilité du processeur d'une part et de profiter de ressources matérielles disponibles dans le FPGA afin d'augmenter les performances en temps réel d'autre part.

L'outil utilisé pour la programmation de FPGA est le langage VHDL dans l'environnement du logiciel Quartus. Le VHDL est un langage de description du matériel destiné à représenter le comportement ainsi que l'architecture d'un système électronique numérique.

L'objectif de ce travail est d'implémenter quelques fonctionnalités du microcontrôleur dans le FPGA de CYCLONE II de ALTERA. Il s'agit de programmer les fonctions des éléments internes du microcontrôleur dans un FPGA en utilisant le langage de description matériel VHDL.

Ce travail est structuré en quatre chapitres :

- Le premier chapitre présente les notions de base du langage de description de matériel le VHDL.
- Le deuxième chapitre est consacré à l'étude du circuit FPGA.
- Le troisième chapitre montre les étapes è suivre pour l'exécution du logiciel Quatus II.
- Le quatrième chapitre représente une étude théorique sur le microcontrôleur, ainsi que la programmation des codes VHDL de quelques fonctions de ses composants.

Chapitre 1

Notions sur
LE LANGAGE
V.H.D.L

I. I - Introduction

VHDL est un langage de description de matériel, destiné à représenter le comportement ainsi que l'architecture d'un système numérique. [2]

L'acronyme VHDL signifie Very High Speed Integrated Circuit(VHSIC) Hardware Description Language(HDL). [3]

Ce langage a trouvé ses premières applications dans la modélisation et la simulation de circuits électroniques, on l'a ensuite étendu en lui rajoutant des extensions pour permettre la conception et la synthèse des circuits logiques programmables (P.L.D. Programmable Logic Device) ou FPGA (Field Programmable Gate Array).Il fallait à ces architectures devenues complexes un langage descriptif de haut niveau en remplacement des langages de première génération aux fonctionnalités limitées.[4]

La description d'un circuit logique avec un schéma est limitée aux circuits les plus simples. Il est difficile de dessiner un circuit complexe avec un schéma de portes logiques. Il est encore plus difficile de le modifier. Parfois, un simple changement dans une équation booléenne du circuit se répercute par une grande quantité de connexions à corriger. De plus, il est difficile, voire impossible, d'utiliser des variables en guise de paramètres d'un circuit représenté par un schéma. Pour ces raisons, les langages de description matérielle (Hardware Description Language – HDL) ont vu le jour au début des années 1980 lorsque la complexité des circuits à concevoir a rendu impossible l'utilisation exclusive de schémas.[3]

I.2-Historique

Dans les années 80, et dans le cadre de l'initiative de développement des circuits intégrés à très haute vitesse VHSIC, le département de la défense des Etats-Unis lance un appel d'offre pour un langage de description de matériel unique qui permettrait de décrire tous les systèmes électroniques utilisés.

C'est en 1983, que d'importantes société telles que IBM, Intermetrics et Texas instruments se sont investies dans ce projet. En 1985 le développement de la première version officielle du VHDL(version 7.2), en 1986 VHDL est donné à IEEE (**I**nstitut of **E**lectrical and **E**lectronics **E**ngineers) pour la standardisation et la norme VHDL 1076 a été approuvée le 10 décembre 1987 par l'IEEE sous la référence IEEE1076.

En 1993, IEEE définit le standard IEEE 1164 normalisant la représentation des signaux logiques multivaleurs. la dernière évolution de la norme est la norme IEEE 1076-2001. [2] [4] [6] [7]

I.3- Les avantages du langage VHDL

Le langage VHDL est un:

- **Langage complet** : dès l'origine, l'un des objectifs majeurs de VHDL fut de couvrir les différents étapes de la conception des systèmes numériques : spécification, modélisation, simulation et synthèse ; un système unique pour ces différentes phases.

- **Langage indépendant** : standard IEEE, il n'appartient à personne en particulier, qui aurait la mainmise sur son évolution et son utilisation. Son indépendance vis-à-vis des architectures et des technologies est totale : son emploi n'est pas réservé à une certaine catégorie d'architectures ou certaine technologies.

- **Langage flexible** :VHDL permet à l'utilisateur d'effectuer des descriptions en se plaçant à différents niveaux d'abstraction : niveau comportemental, niveau intermédiaire avec utilisation d'équations logiques par exemples, niveau composant avec interconnexion de portes logiques élémentaires. Flexibilité aussi dans l'assemblage des descriptions avec ou sans hiérarchie, c'est selon le souhait de l'utilisateur.

- **Langage moderne** :Sa syntaxe est particulièrement lisible, et permet fréquemment de faire l'économie de commentaires. Langage fortement typé et contrôlé, il minimise les risques d'erreurs et leur propagation insidieuse et favorise la qualité.

- **Langage standard** :C'est un gage de compatibilité, de stabilité et pérennité. . Le langage VHDL est normalisé par l'IEEE. La première norme remonte à 1987. Des mises à jour ont eu lieu en 1993, 2000, 2002 et 2008.

- **Langage ouvert** : VHDL supporte les mécanismes de fonctions et procédures permettant aux concepteurs d'étendre les fonctionnalités du langage, en toute portabilité et dans le respect des règles strictes. [4]

I.4- Structure d'une description VHDL simple

un opérateur élémentaire, un circuit intégré, une carte électronique ou un système complet est complètement défini par des signaux d'entrées et de sorties et par fonction réalisée de façon interne. Ce double aspect - signaux de communications et fonction- se retrouve à tous les niveaux de la hiérarchie d'une application. L'élément essentiel de toute description en VHDL, nommé **design entity** dans le langage, est formé par le couple **entité-architecture**, qui décrit l'apparence externe d'une unité de conception et son fonctionnement interne.

- **L'entité:** décrit l'interface externe du circuit : signaux d'entrées et de sorties.
- **L'architecture** : décrit le fonctionnement interne du circuit. Elle est toujours associée à une entité.
  Une même entité peut avoir plusieurs architecture. [3][4]

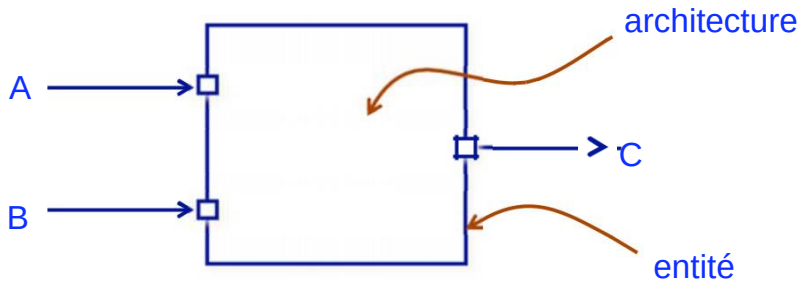


Figure I.1: Entité et architecture [5]

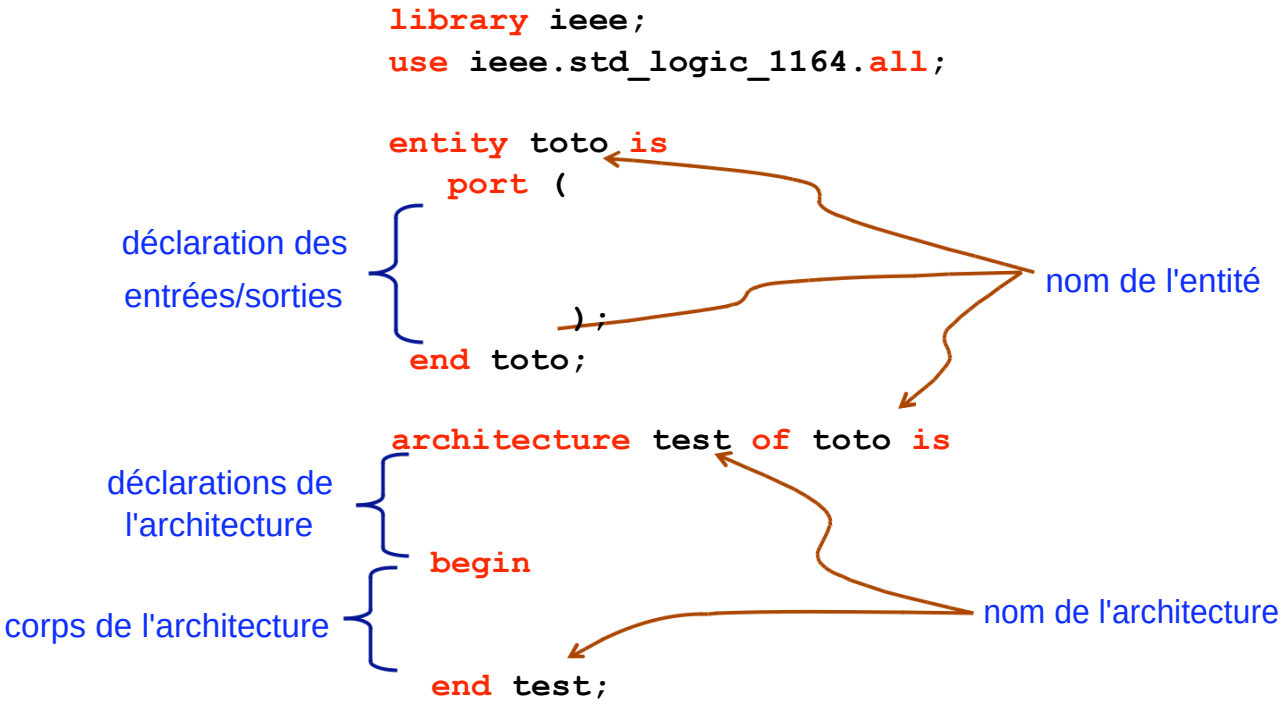


Figure I.2: Structure d'un programme VHDL[5]

I.4.1- Déclaration des bibliothèques

Toute description VHDL utilisée pour la synthèse a besoin de bibliothèques. L'**IEEE** (**I**nstitut of **E**lectrical and **E**lectronics **E**ngineers) les a normalisées et plus particulièrement la bibliothèque **IEEE1164** . Elles contiennent les définitions des types de signaux électroniques, des fonctions et sous programmes permettant de réaliser des opérations arithmétiques et logiques:

- **Library ieee;** Déclaration standard.
- **Use ieee.std_logic_1164.all;** Pour définir les types std_logic ('X','U','L','H','0','1')
- **Use ieee.std_logic_arith.all;** Pour utiliser les opérateurs arithmétiques '+','-','*','/' pour les types std_logic
- **Use ieee.std_logic_signed.all;** Le type std_logic avec valeur signée.
- **Use ieee.std_logic_usigned.all;** Le type std_logic avec valeur non signée.
- **Use ieee.numeric_std.all;** Utilisation des valeurs décimales pour le type std_logic.
- **Use ieee.std_logic_textio.all;** Utilisation des valeurs ASCII pour le type std_logic.
- **Use ieee.numeric_bit.all;** Utilisation des valeurs décimales pour le type std_logic.
- **Use ieee.math_real.all;**
- **Use ieee.math_complex.all;**

Note :

IEEE 1164 : référence de multivaleur système publiée par ieee dans l'année 1993.

IEEE 1076 : référence de langage VHDL publiée par ieee dans l'année 1987.

1. La directive **Use** permet de sélectionner les bibliothèques à utiliser.
2. La directive **All** pour utiliser toutes les applications.
3. La différence entre **Signed et Unsigned (**Un vecteur représente alors un nombre signé ou non signé**).** [6][8]

I.4.2- Déclaration de l'entité

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 associer 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.[3]

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. [9]

Exemple de syntaxe d'une déclaration d'entité :

```
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;
```

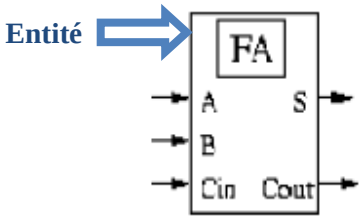


Figure I-3 : représentation schématique[9]

I.4.2.1- Ports:

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.

I.4.2.1.1 - Le nom du signal : le nom de chaque signal est choisi par l'utilisateur est connu à l'intérieur de toutes les architectures qui font référence à l'entité correspondante. il est constitué par une chaine de caractères alphanumériques qui commence par une lettre et qui peut contenir le caractère souligné(_). VHDL ne distingue pas les majuscules des miniscules AmI et aMi représentent donc le même objet. [3] [4]

I.4.2.1.2- Le mode du signal

le mode précise le sens du signal :

- **In** : pour un signal en entrée.
- **Out** : pour un signal en sortie.
- **inout** : pour un signal en entrée sortie
- **buffer** : pour un signal en sortie mais utilisé comme entrée dans la Description

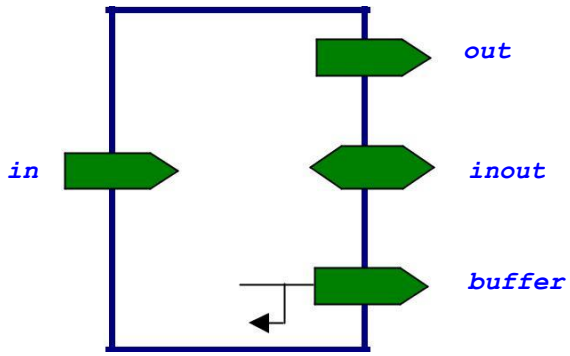


Figure I.4: les quatre modes de VHDL [4]

I.4.2.1.3- Le type du signal

Le type utilisé pour les signaux d'entrées / sorties est :

- le **std_logic** pour un signal.
- le **std_logic_vector** pour un bus composé de plusieurs signaux.

Les valeurs que peuvent prendre un signal de type ***std_logic*** et ***std_logic_vector*** sont :

- 0 : Niveau logique **bas à basse impédance** (mise à la masse via une faible impédance)
- 1 : Niveau logique **haut à basse impédance** (mise à Vcc via une faible impédance)
- Z : Niveau logique **flottant** (entrée déconnectée)
- L : Niveau logique **bas à haute impédance** (mise à la masse via une résistance de *pull-down*)
- H : Niveau logique **haut à haute impédance** (mise à Vcc via une résistance de *pull-up*)
- W : Niveau logique **inconnu à haute impédance** (pouvant être 'L', 'Z' ou 'H')
- X : Niveau logique **inconnu** (pouvant être '0', 'L', 'Z', 'H' ou '1')
- U : **Non défini**
- - : **N'importe quel niveau logique** (renvoie toujours *true* lors d'une comparaison avec les 8 autres niveaux logiques). [10]

I.4.3- Déclaration de l'architecture

L'architecture décrit le fonctionnement souhaité pour un circuit ou une partie du circuit. En effet le fonctionnement d'un circuit est généralement décrit par plusieurs modules VHDL. Il faut comprendre par module le couple ENTITE/ARCHITECTURE. Dans le cas de simples **PLDs** on trouve souvent un seul module.

L'architecture établit à travers les instructions les relations entre les entrées et les sorties. On peut avoir un fonctionnement purement combinatoire, séquentiel voire les deux séquentiel et combinatoire. [6]

L'architecture est divisée en deux parties: une zone déclarative et une zone d'instructions. ces instructions sont concurrentes, elles s'exécutent en parallèles. cela signifie que les instructions d'une architecture peuvent être écrites dans un ordre quelconque, le fonctionnement ne dépend pas de cet ordre.[3]

Trois styles de description peuvent être utilisés en VHDL :

- le style "**structurel**" : interconnexion de composants, chacun d'eux étant une instance de couple entité/architecture.
- le style "**dataflow**" : correspond grosso modo à un **RTL (R**egister **T**ransfert **L**anguage) , ensemble d'instructions sur signaux qui décrivent les connexions entre portes logiques et les chargements de registres.
- le style "**comportemental**" : ensemble de processus qui expriment le comportement du système. [12]

Une définition d'architecture a la forme suivante :

```
architecture nom_de_l_architecture of nom_de_l_entité is
    déclarations
begin
    instructions-concurrentes
end nom_de_l_architecture;
```

I.5- Les opérateurs du langage

le tableau ci-dessous liste les opérateurs standard et le type des opérandes manipulés: [4]

<table>
<tr><th>CATEGORIE</th><th colspan="2">TYPE OPERANDE</th><th>SIGNIFICATION</th></tr>
<tr><td>Opérateurs logiques
and
nand
or
nor
xor
xnor
not</td><td colspan="2">boolean
bit ou
bit_vector_</td><td>Et
Non-et
Ou
Non-ou
Ou exclusif
Non ou exclusif
Non</td></tr>
<tr><td rowspan="2">Opérateurs relationnels
=
/=
<
<=
>
>=</td><td>entrée</td><td>résultat</td><td rowspan="2">Egal
Non égal
Inférieur
Inférieur ou égal
Supérieur
Supérieur ou égal</td></tr>
<tr><td>Tout type scalaire</td><td>boolean</td></tr>
<tr><td rowspan="2">Opérateurs arithmétiques
+
-
*
/
Abs
**
Mod
rem</td><td colspan="2">Integer, real</td><td rowspan="2">+ unaire (signe +) ou addition
- unaire (signe -) ou soustraction
Multiplication
Division
Valeur absolue
Exponentiation
Modulo
Reste</td></tr>
<tr><td colspan="2">integer</td></tr>
<tr><td>Décalages
Sll
Srl
Rol
Ror</td><td colspan="2">bit_vector (amplitude : integer)</td><td>Logique gauche
Logique droit
Circulaire gauche
Circulaire droit</td></tr>
<tr><td>Autres
&</td><td>Bit,
bit_vector</td><td>bit_vector</td><td>concaténation</td></tr>
</table>

I.6-Fonctionnements concurrents et séquentiels

I.6.1- Fonctionnement concurrent

Le comportement d'un circuit peut être décrit par un ensemble d'actions s'exécutant en parallèle. C'est pourquoi VHDL offre un jeu d'instructions dites concurrentes.

Une instruction concurrente est une instruction dont l'exécution est indépendante de son ordre d'apparition dans le code VHDL.

Par exemple, prenons le cas d'un simple latch, comme le montre la figure :

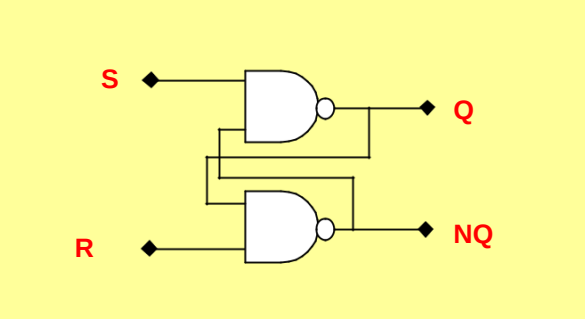


Figure I.5: Schéma d'un Latch

Les deux portes constituant ce latch fonctionnent en parallèle. Une description possible de ce circuit est donnée dans le code suivant (seule l'architecture est donnée).

```
architecture comportement of VERROU is
begin
        Q<= S nand NQ;
        NQ <= R nand Q;
end comportement;
```

Ces deux instructions s'exécutent en même temps. Elles sont concurrentes ; leur ordre d'écriture n'est pas significatif ; quel que soit l'ordre de ces instructions, la description reste inchangée. [12]

I.6.2 Fonctionnement séquentiel

En VHDL, les instructions séquentielles ne s'utilisent qu'à l'intérieur des processus **process.**

I.6.2.1- Définition d'un process

Un **process** est une partie de la description d'un circuit dans laquelle les instructions sont exécutées séquentiellement, c'est à dire les unes à la suite des autres (contrairement aux instructions concurrentes). Il permet d'effectuer des opérations sur les signaux en utilisant les instructions standard de la programmation structurée comme dans les systèmes à microprocesseurs.

Syntaxe :

```
[Nom_du_process :] process (Liste_de_sensibilité_nom_des_signaux)
                    begin
                       -- instructions du process
                    end process [Nom_du_process] ; [8]
```

I.6.2.2- Règles de fonctionnement d'un process :

1) L'exécution d'un **process** est déclenchée par un ou des changements d'états de signaux logiques. Le nom de ces signaux est défini dans **la liste de sensibilité** lors de la déclaration du **process**.

2) Les instructions du **process** s'exécutent séquentiellement.

3) Les modifications apportées aux valeurs de signaux par les instructions prennent effet à la fin du processus. [4][8]

L'exemple de la description suivante montre une architecture (seule) d'une bascule **D** contenant un processus qui est exécuté lors du changement d'état de l'horloge CLK. [12]

```
architecture comportement of basc_D is
begin
   Process (CLK)
   Begin
       If ( CLK= '1') then
          Q <= D;
       End if ;
   End process ;
end comportement;
```

I.7-INSTRUCTIONS

I.7.1- Instructions en mode concurrent:

I.7.1.1- Assignation inconditionnelle

sa forme générale:

signal <= expression;

I.7.1.2- Assignation conditionnelle

l'assignation conditionnelle se présente comme suit:

signal <= exp_1 when $condition_1$ else
exp_2 when $condition_2$ else
.....
exp_{N-1} when $condition_{N-1}$ else exp_N;

I.7.1.3- Assignation sélective:

a la syntaxe suivante:

with expression select
signal <= exp_1 when $choix_1$,
exp_2 when $choix_2$,
....
exp_N when $choix_N$;

I.7.1.4- Instanciation du composant:

Consiste à utiliser un sous-ensemble décrit en VHDL comme composant dans un ensemble plus vaste

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); [13]

I.7.1.5- Instruction generate

-- structure répétitive :

etiquette : for variable in valeur_debut to valeur_fin generate

instructions_concurrentes

end generate [etiquette] ;

ou :

-- structure conditionnelle :

```
etiquette : if condition generate
                instructions_concurrentes
            end generate [etiquette] ;
```

I.7.2- Instruction en mode séquentiel

I.7.2.1- Assignation inconditionnelle de variable

forme générale

```
var := expression;
```

I.7.2.2- Instruction wait :

L' instruction **wait** suspend l'exécution d'un **process** jusqu'à ce qu'un événement, une condition ou une clause de temps écoulé (time out) soit vraie. Si aucune clause de réveil n'est stipulée, le processus s'arrête définitivement. [4]

« **wait** » peut être utilisé des manières suivantes:

wait on [nomSignal1, nomSignal2…] **;**

wait until [expressionBooléenne] **;**

wait for [expressionTemps] **;**

I.7.2.3- Instructions conditionnelles :

Peuvent être exprimées au moyen des instructions ***if*** et ***case***

- Syntaxe de **if** :

```
if condition1 then instructions
  [elsif condition2 then instructions]
  . . .
  [else instructions]
end if ;
```

Il est possible d'imbriquer plusieurs boucles ***if*** les unes dans les autres:

```
if ... then
      if ... then
        elsif ... then
      end if;
 else
end if;
```

- Syntaxe de **case:**

```
case signal_de_slection is
      when valeur_de_sélection => instructions
      [when others => instructions]
end case;
```

I.7.3- Les boucles :

Les boucles permettent de répéter une séquence d'instructions. Trois catégories de boucles existent en VHDL, suivant le schéma d'itération choisi :

− Les boucles simples, sans schéma d'itération, dont on ne peut sortir que par une instruction « **exit** ».

− Les boucles « **for** », dont le schéma d'itération précise le nombre d'exécution.

− Les boucles « **while** », dont le schéma d'itération précise la condition de maintien dans la boucle. [14]

Syntaxe « for » :

```
for parametre in minimum to maximum loop
      séquence d'instructions
end loop;
```

Ou :

```
for parametre in maximum downto minimum loop
      séquence d'instructions
end loop [ etiquette ] ;
```

Syntaxe " while " :

```
while condition loop
     séquence d'instructions
end loop [ etiquette ] ;
```

Instructions exit et next

- Pour ne pas rester indéfiniment dans une boucle simple, une condition de sortie est nécessaire **(exit)**

Syntaxe :

exit [etiquette] when condition;

- L'instruction **next** permet de passer à l'itération suivante dans une boucle :

next [etiquette] when condition;

I.8- Sous programmes

Les sous programmes sont le moyen par lequel le programmeur peut se constituer une bibliothèque d'algorithmes séquentiels qu'il pourra inclure dans une description.

Les deux catégories de sous programmes, procédures et fonctions, diffèrent par les mécanismes d'échanges d'informations entre le programme appelant et le sous programme.

I.8.1- Les fonctions

Une fonction retourne au programme appelant une valeur unique, elle a donc un type. Elle peut recevoir des arguments, exclusivement des signaux ou des constantes, dont les valeurs lui sont transmises lors de l'appel.

Une fonction ne peut en aucun cas modifier les valeurs de ses arguments d'appel. [14]

Syntaxe:

FUNCTION nom de la fonction (*liste des paramètres de la fonction avec leur type***)**
RETURN *type du paramètre de retour* **IS**
zone de déclaration des variables;
BEGIN
instructions séquentielles;
RETURN *nom de la variable de retour ou valeur de retour*;
END; [8]

*Le corps d'une fonction ne peut pas contenir d'instruction **wait**, les variables locales, déclarées dans la fonction, cessent d'exister dès que la fonction se termine.

I.8.2- Les procédures

Une procédure, comme une fonction, peut recevoir du programme appelant des arguments : constantes, variables ou signaux. Mais ces arguments peuvent être déclarés de modes « in », « inout » ou « out » (sauf les constantes qui sont toujours de mode « in »), ce qui autorise une procédure à renvoyer un nombre quelconque de valeurs au programme appelant.

Syntaxe :

Procedure *nom de la procédure* **(***liste des paramètres de la procédure avec leur direction et type***)**
RETURN IS
zone de déclaration des variables;
BEGIN
instructions séquentielles;
END [*nom de la procédure***];** [8]

Le corps d'une procédure peut contenir une instruction ***wait***, les variables locales, déclarées dans la procédure, cessent d'exister dès que la procédure se termine.

Une procédure peut être appelée par une instruction concurrente ou par une instruction séquentielle, mais si l'un de ses arguments est une variable, elle ne peut être appelée que par une instruction séquentielle. [14].

I.9 - Conclusion:

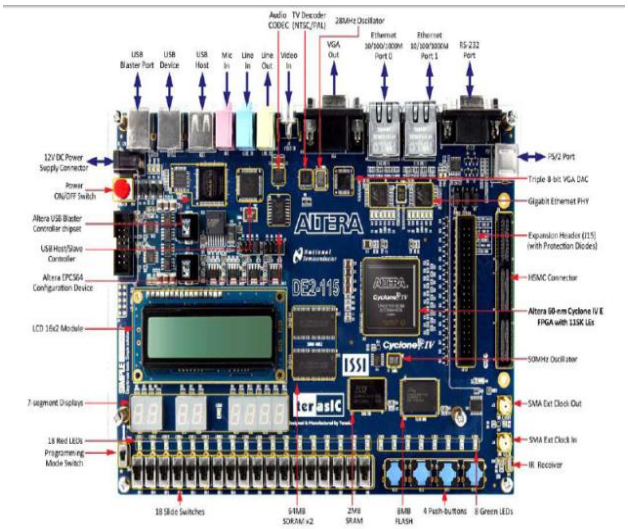
Dans ce chapitre, on a présenté les notions de base du langage de description de matériel VHDL, sa structure, ses opérateurs, ses instructions et les avantages qu'il présente car c'est un langage complet qui couvre les différentes étapes de la conception des systèmes numériques.

Chapitre 2

Le circuit

(*F*ield-*P*rogrammable *G*ate *A*rray)

F.P.G.A



II.1 - Introduction

FPGA est l'abréviation de Field Programmable Gate Arrays ou "réseaux logiques programmables". Inventés par la société Xilinx en 1985, les FPGA sont des composants logiques de haute densité et reconfigurables qui permettent, après programmation, de réaliser des fonctions logiques, des calculs, et des générations de signaux. Il s'agit d'un circuit intégré qui peut être programmé pour fonctionner selon la conception prévue. Cela signifie qu'il peut fonctionner comme un microprocesseur, ou comme une unité de cryptage, ou une carte graphique, ou même tous ces trois à la fois.

L'avantage de ce genre de circuit est sa grande souplesse qui permet de les réutiliser à volonté dans des algorithmes différents en un temps très court. Les FPGAs peuvent être utilisés pour implémenter n'importe quelle fonction logique que les Circuits intégrés spécifiques ASICs (Application Specific Integrated Circuit) peuvent implémenter. Leur reconfiguration, qui peut être effectuée un nombre arbitraire de fois, représente l'un de leurs avantages majeurs par rapport aux ASICs. [15] [16]

Les conceptions fonctionnant sur des FPGA sont généralement créées à l'aide de langages de description de matériel tels que VHDL et Verilog.

II.2- Application des FPGA

Les FPGA sont utilisés dans de nombreuses applications, on en cite dans ce qui suit quelques unes:

- Prototypage de nouveaux circuits ;
- Fabrication de composants spéciaux en petite série ;
- Adaptation aux besoins rencontrés lors de l'utilisation ;
- Systèmes de commande à temps réel ;
- DSP (Digital Signal Processor) ;
- Imagerie médicale. [16]

II.3- Architecture des FPGA

Les principaux éléments composant les FPGAs sont :

* Les CLB (Configurable Logic Block) : Les CLBs constituent le coeur du FPGA. Ce sont des cellules constituées d'éléments logiques programmables où l'on trouve des bascules (registres), des LUTs (Look-Up Table), des multiplexeurs et des portes logiques disposées sous forme matricielle.

• Les IOBs (Input Output Block) : Ces cellules d'entrées-sorties permettent d'interfacer le FPGA avec l'environnement extérieur.

• Les ressources d'interconnexion des cellules : Pour pouvoir réaliser des fonctions complexes à partir des fonctions de base que représentent les CLBs, il est nécessaire de disposer de ressources d'interconnexion entre ces différentes cellules. Ce sont des bus qui remplissent cette fonction. Il existe différents types de bus d'interconnexion en fonction du type de signal à propager (Les horloges ont en général des bus de transmission qui leur sont dédiés). [21]

voici l'architecture interne d'un FPGA de type matrice symétrique

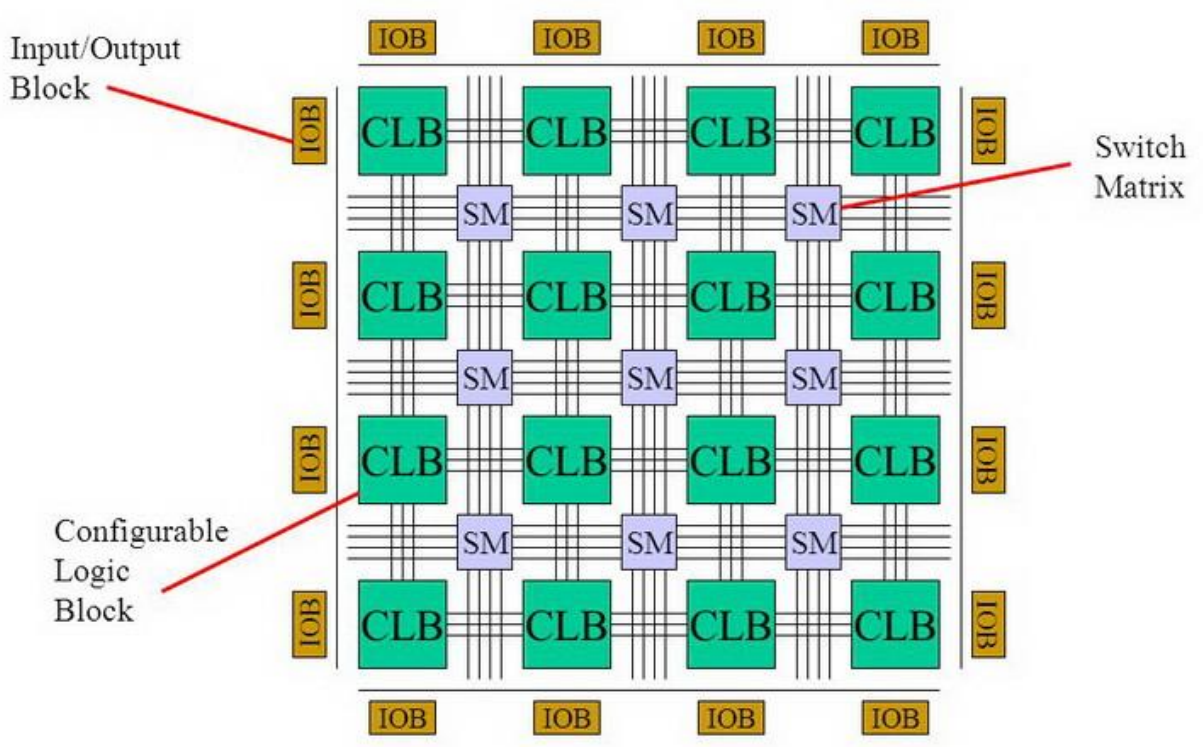


Figure II.1 structure d'un FPGA. [15]

Les circuits FPGA de Xilinx utilisent deux types de cellules de base :

- les cellules d'entrées/sorties appelés IOB (input output bloc),
- les cellules logiques appelées CLB (configurable logic bloc). Ces différentes cellules sont reliées entre elles par un réseau d'interconnexions configurable.

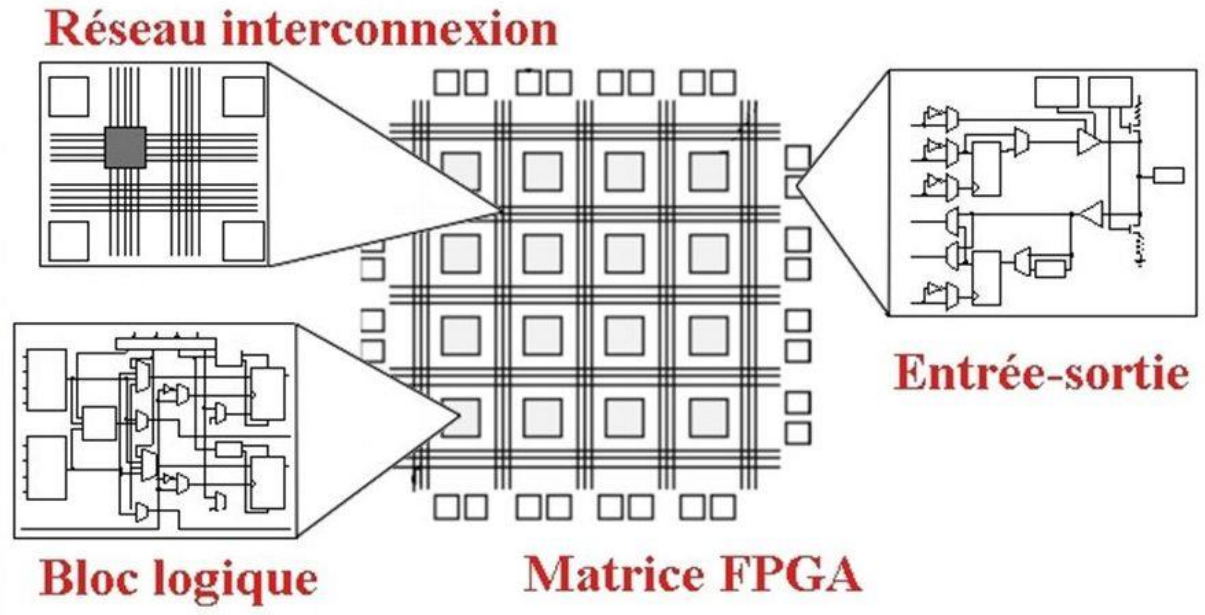


Figure II.2 - Architecture interne d'un FPGA. [15]

II.3-1 Le Bloc logique configurable : la structure du bloc logique change selon le fabricant, la famille, etc..., un bloc logique configurable CLB d'un FPGA typique consiste en une table de correspondance ou consultation **LUT** (**L**ook-**U**p-**T**able) RAM pour implémenter une fonction combinatoire et une bascule D (**F**lip-**F**lop). L'ensemble de la logique combinatoire (ET, OU, NON ET,etc...) est implémentée sous forme de tables de vérités dans la mémoire LUT. Et comme on le sait une table de vérité est une liste prédéfinie d'états de sorties pour chaque combinaison d'entrée. Donc La fonction de la LUT est de stocker la table de vérité de la fonction combinatoire à implémenter dans la cellule.

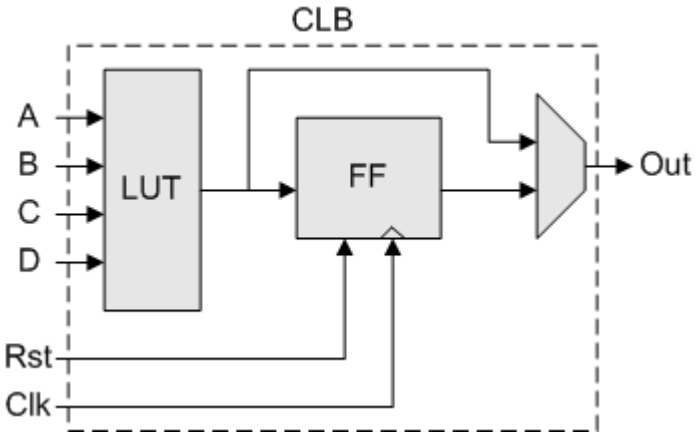


Figure II.3 Schéma bloc d'une cellule [17]

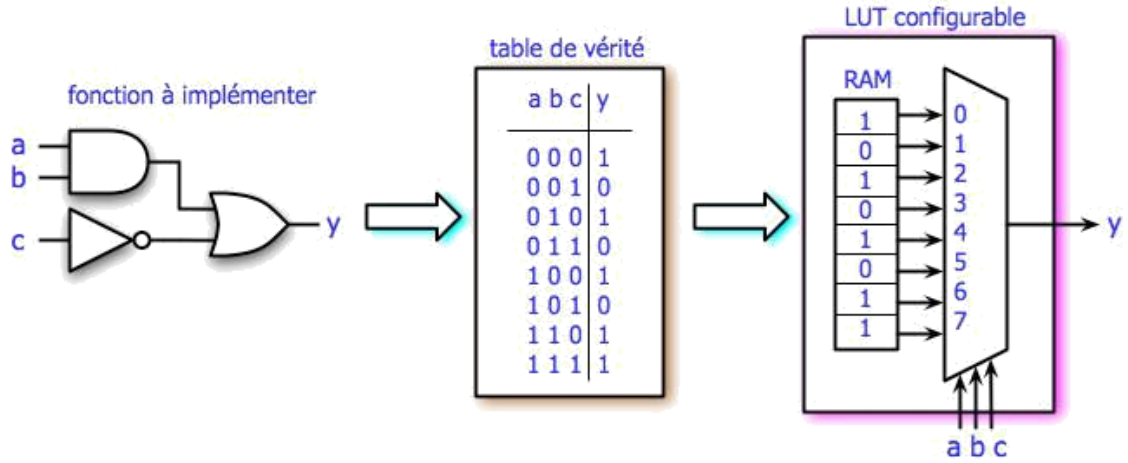


Figure II.4 Stockage de la table de vérité de la fonction à implémenté dans la cellule logique

II.3.2- Les IOB (input output bloc):

Les blocs d'entrées-sorties permettent l'interconnexion de la logique interne aux ports d'entrées et de sorties du FPGA. Les IOB ont leur propre mémoire de configuration, elles stockent les standards de tension et la direction des ports. Ces blocs sont présents sur toute la périphérie du circuit FPGA. Chaque bloc IOB contrôle une broche du composant et il peut être défini en entrée, en sortie, en signaux bidirectionnels ou être inutilisé.

II.3.3- Les ressources de communications:

Les ressources de communications ou des interconnexions au sein d'un FPGA permettent la connexion arbitraire des CLB et des IOB. Les connexions internes dans les circuits FPGA sont composées de segments métallisés. Parallèlement à ces lignes, nous trouvons des matrices programmables réparties sur la totalité du circuit, horizontalement et verticalement entre les divers CLB. Elles permettent les connexions entre les diverses lignes, celles-ci sont assurées par des transistors MOS dont l'état est contrôlé par des cellules de mémoire vive ou RAM.

II.4 -Architecture détaillée d'un FPGA de Xilinx:

L'architecture, retenue par Xilinx, se présente sous forme de deux couches :

- une couche active appelée circuit configurable,
- une couche de configuration. [22]

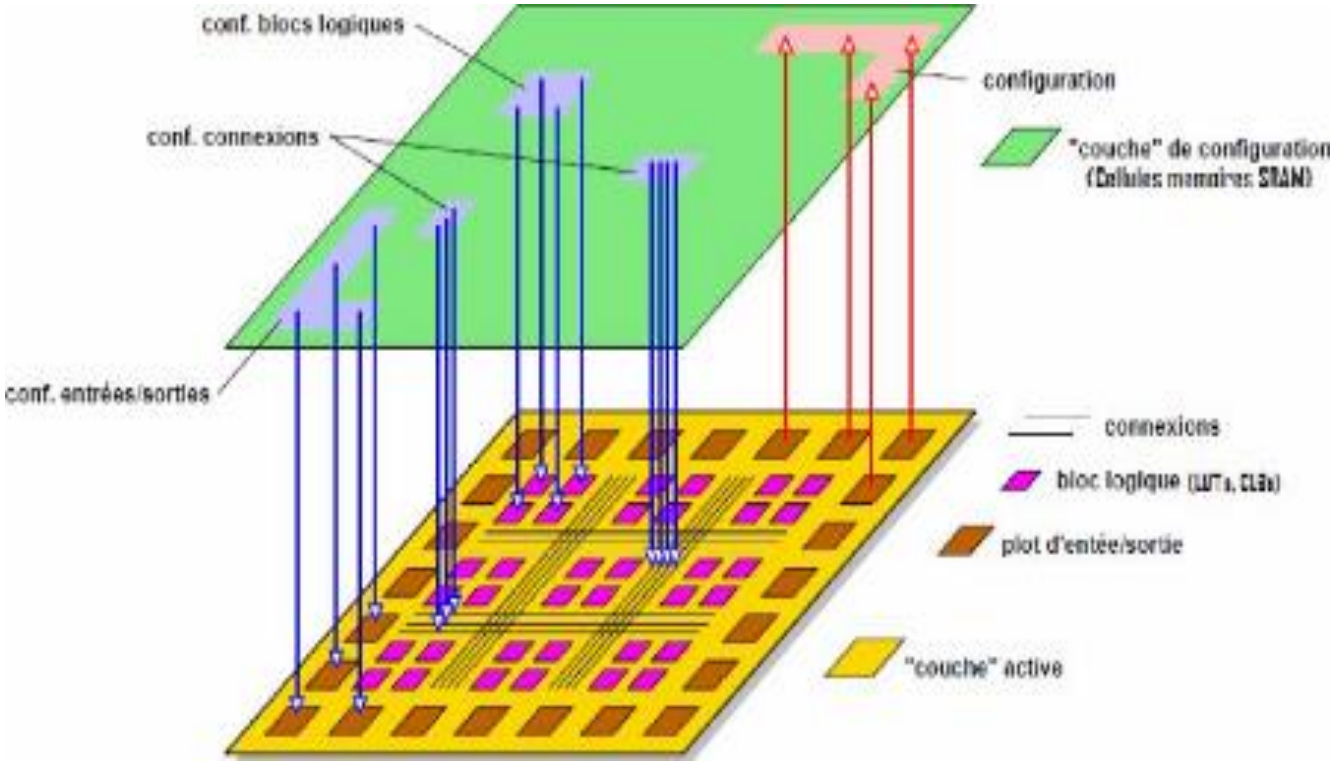


Figure II.5 Architecture d'un FPGA [22]

II.4.1 - La couche active : ou bien la couche **circuit configurable** est constituée d'une matrice de **blocs logiques configurables CLB** permettant de réaliser des fonctions combinatoires et des fonctions séquentielles . Tout autour de ces blocs logiques configurables, nous trouvons des **blocs entrées/sorties IOB** dont le rôle est de gérer les entrées-sorties réalisant l'interface avec les modules extérieurs. La programmation du circuit FPGA appelé aussi LCA (logic cells arrays) consistera par le biais de l'application d'un potentiel adéquat sur la grille de certains transistors à effet de champ à interconnecter les éléments des CLB et des IOB afin de réaliser les fonctions souhaitées et d'assurer la propagation des signaux. Ces potentiels sont tout simplement mémorisés dans le réseau mémoire SRAM. [15]

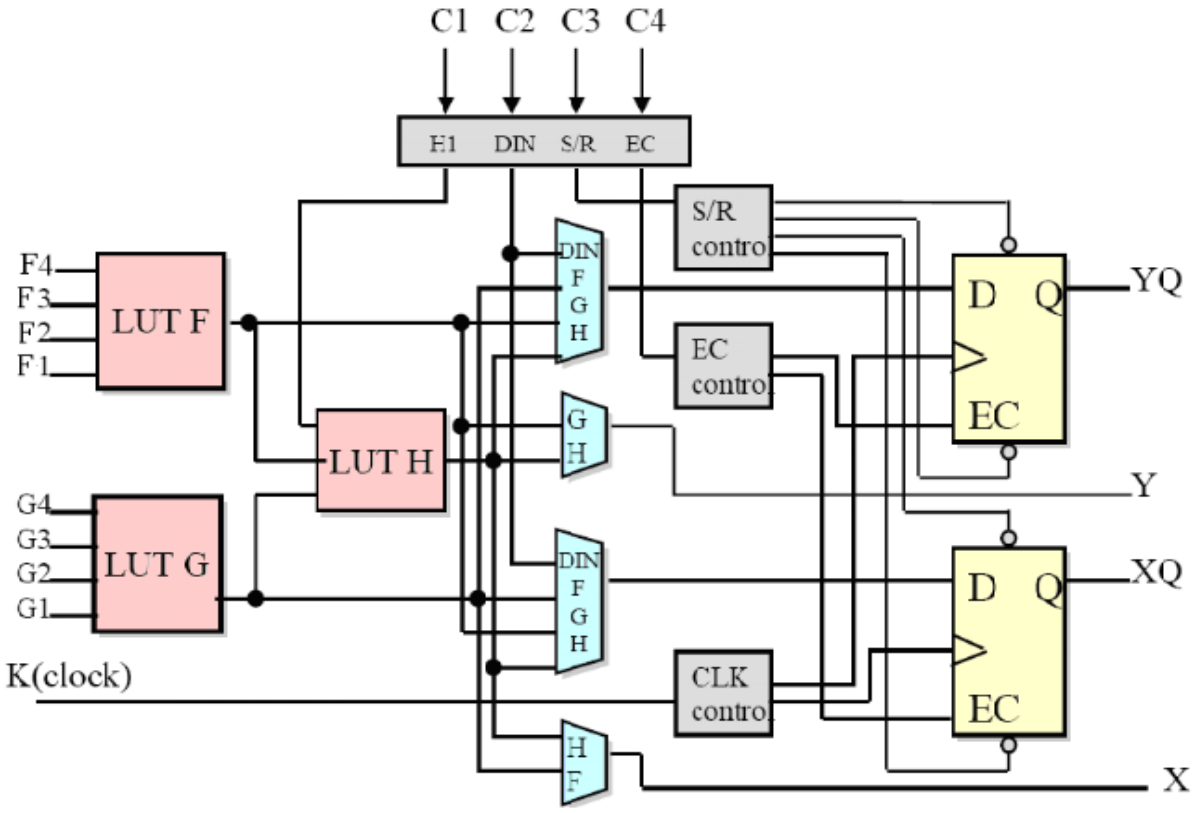


Figure II.6 Cellule logique (CLB). [18]

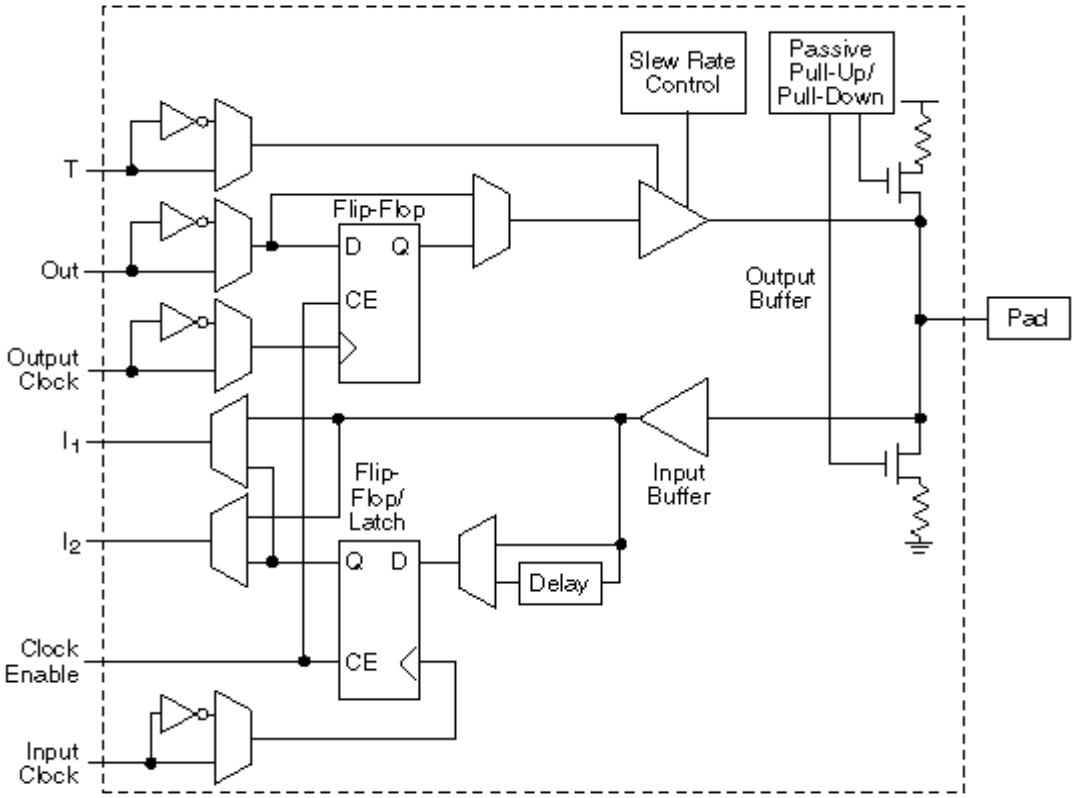


Figure II.7: Input Output Block (IOB). [18]

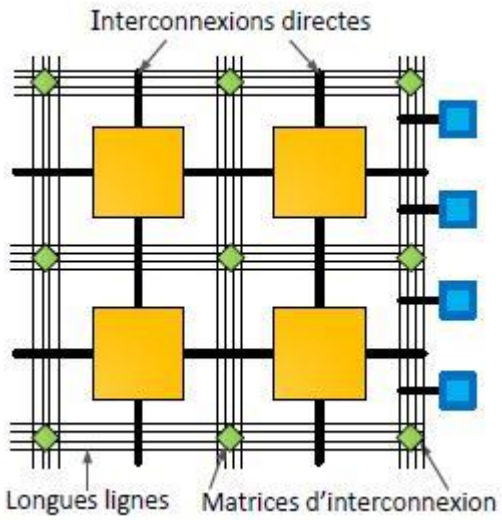


Figure II.8 : Les interconnexions entre les cellule d'un FPGA [18]

II.4.2- La couche de configuration: encore appelée *mémoire de configuration* et permettant de programmer électriquement les caractéristiques des ressources de la couche active. En effet, toutes les ressources logiques du FPGA sont contrôlées par le contenu de la «mémoire de configuration» . [22]

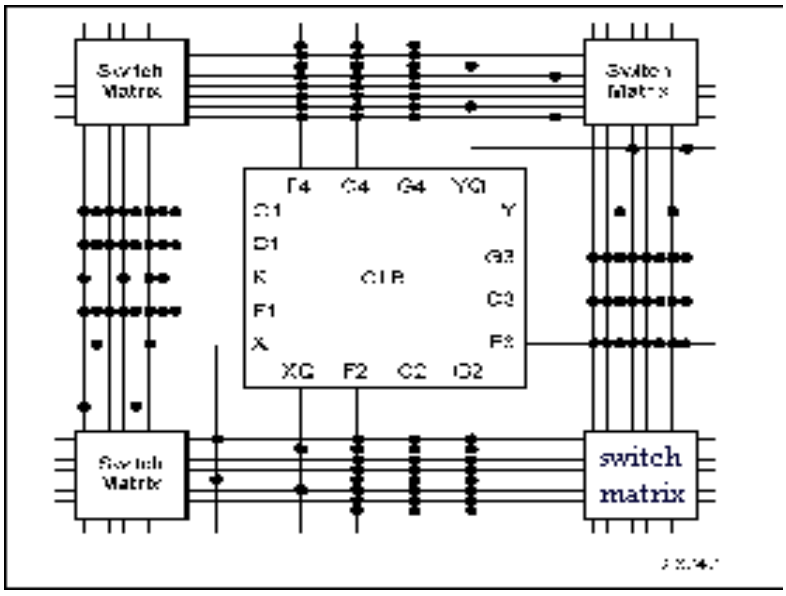
II.5- Les différents types d'interconnexions:

On dispose trois sortes d'interconnexions selon la longueur et la destination des liaisons:

- d'interconnexions à usage général,
- d'interconnexions directes,
- de longues lignes. [15]

II.5.1-Les interconnexions à usage général:

Ce système fonctionne en une grille de cinq segments métalliques verticaux et quatre segments horizontaux positionnés entre les rangées et les colonnes de CLB et de l'IOB.



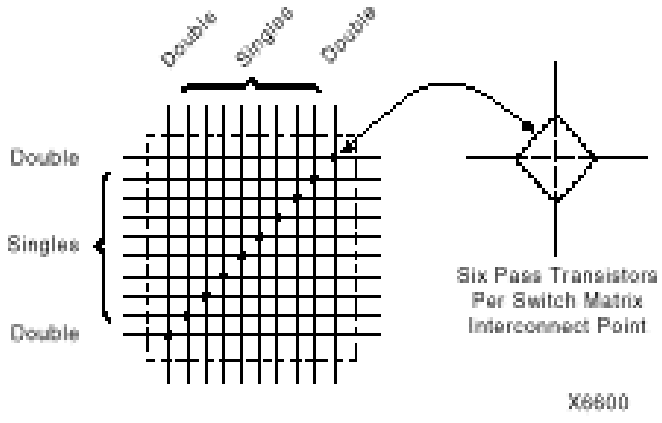


Figure II.9 : Connexions à usage général et détail d'une matrice de commutation.

Des aiguilleurs appelés aussi matrices de commutation sont situés à chaque intersection. Leur rôle est de raccorder les segments entre eux selon diverses configurations, ils assurent ainsi la communication des signaux d'une voie sur l'autre. Ces interconnexions sont utilisées pour relier un CLB à n'importe quel autre. Pour éviter que les signaux traversant les grandes lignes ne soient affaiblis, nous trouvons généralement des buffers implantés en haut et à droite de chaque matrice de commutation.

II.5.2- Les interconnexions directes:

Ces interconnexions permettent l'établissement de liaisons entre les CLB et les IOB avec un maximum d'efficacité en terme de vitesse et d'occupation du circuit. De plus, il est possible de connecter directement certaines entrées d'un CLB aux sorties d'un autre.

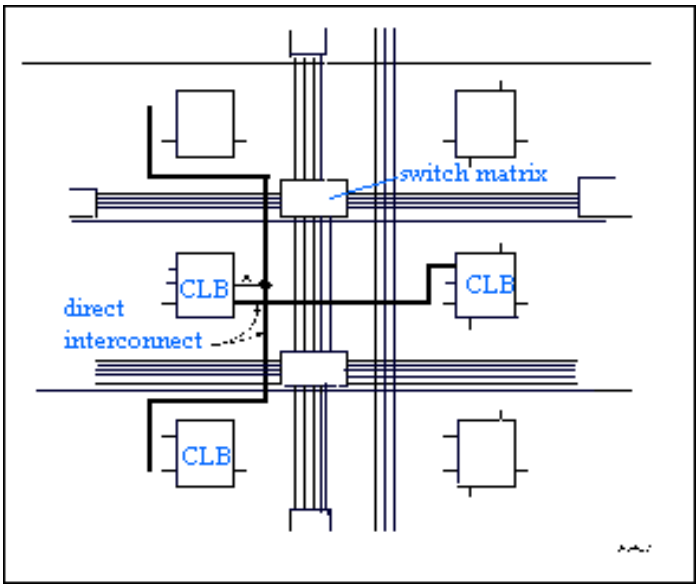


Figure II.10 : Les interconnexions directes.

II.5.3- Les longues lignes:

Les longues lignes sont de longs segments métallisés parcourant toute la longueur et la largeur du composant, elles permettent éventuellement de transmettre avec un minimum de retard les signaux entre les différents éléments dans le but d'assurer un synchronisme aussi parfait que possible. De plus, ces longues lignes permettent d'éviter la multiplicité des points d'interconnexion.

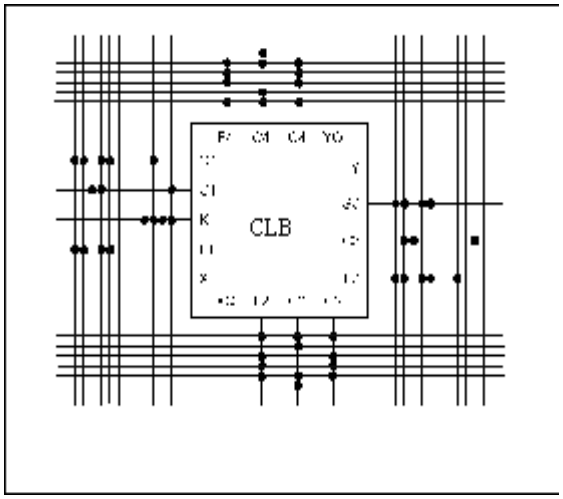


Figure II.11 : Les longues lignes.

II.6- Performances des interconnexions:

Les performances des interconnexions dépendent du type de connexions utilisées. Pour les interconnexions à usage général, les délais générés dépendent du nombre de segments et de la quantité d'aiguilleurs employés. Le délai de propagation de signaux utilisant les connexions directes est minimum pour une connectique de bloc à bloc. Quant aux segments utilisés pour les longues lignes, ils possèdent une faible résistance mais une capacité importante. De plus, si on utilise un aiguilleur, sa résistance s'ajoute à celle existante.

II.7- L'oscillateur à quartz:

Placé dans un angle de la puce, il peut être activé lors de la phase de programmation pour réaliser un oscillateur. Il utilise deux IOB voisins, pour réaliser l'oscillateur dont le schéma est présenté ci-dessous. Cette oscillateur ne peut être réalisé que dans un angle de la puce où se trouve l'amplificateur prévu à cet effet. Il est évident que si l'oscillateur n'est pas utilisé, les deux IOB sont utilisables au même titre que les autres IOB.

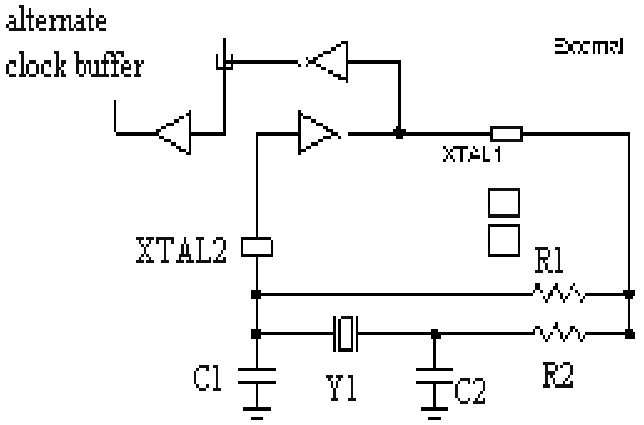


Figure II.12 : L'oscillateur à quartz.

II.8- Les différentes technologies utilisées pour les FPGA:

II.8.1- Static RAM: Les connexions sont réalisées en rendant les transistors passant. L'avantage de cette technologie est qu'elle permet une reconfiguration rapide au sein même du circuit. Le principal désavantage est la surface nécessaire pour la SRAM. Il nécessitent l'utilisation d'une mémoire standard chargée à l'initialisation. Ils font appel à la technologie CMOS.

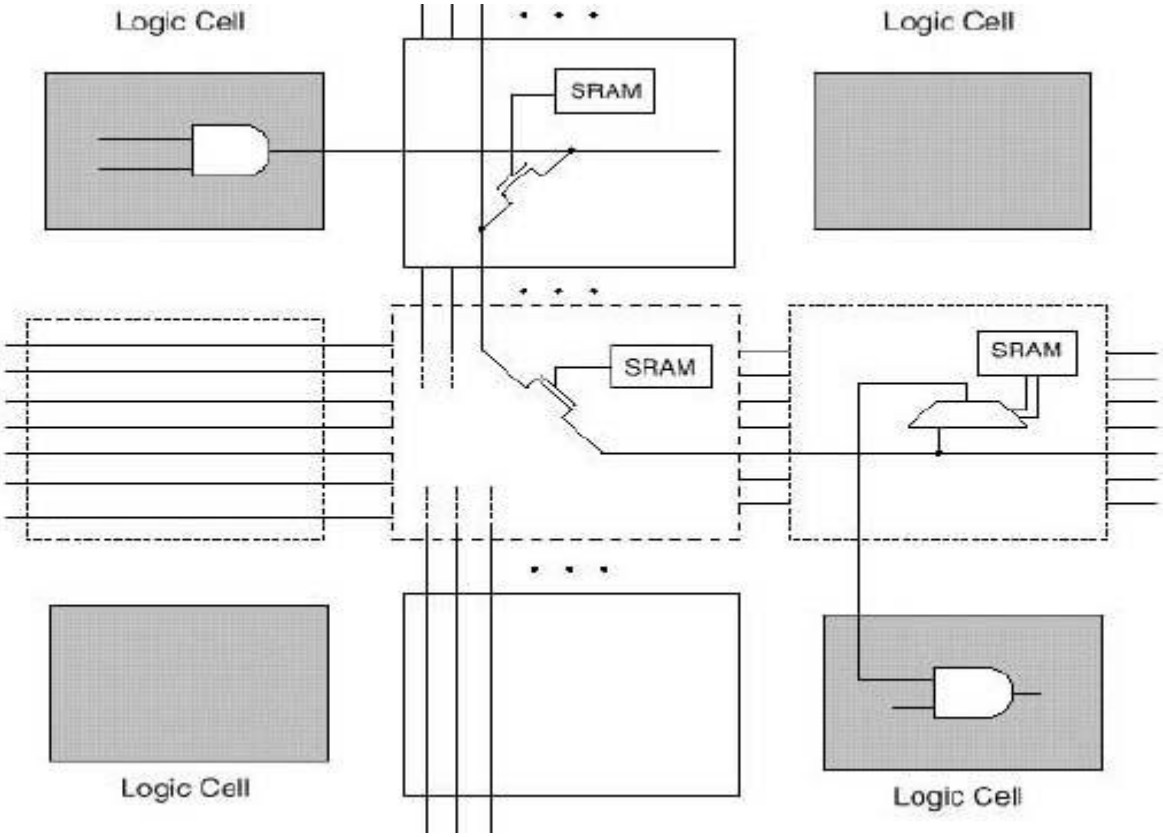


Figure II.13 Technologie Static Ram [15]

II.8.2- La technologie anti-Fuse: Un état anti-fuse réside en un état d'haute impédance. Il peut être programmé dans un état de faible impédance ou état "fused". Il s'agit d'une technologie moins chère que la SRAM, elle permet d'atteindre des vitesses plus élevées et occupe moins de place sur le circuit. Par contre, un tel FPGA ne peut être programmé qu'une seule fois.

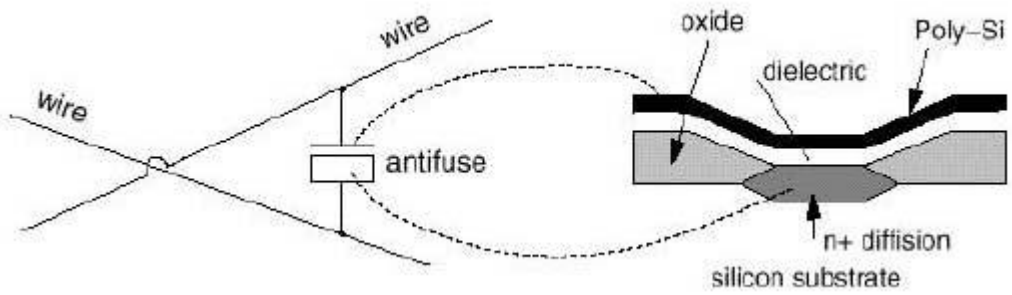


Figure II.14: Technologie Anti fuse [15]

II.8.3- EPROM/EEPROM: Cette méthode est la même que celle utilisée dans les mémoires EPROM. L'EEPROM est reprogrammable. La puce fonctionne seule. La surface moyenne et les caractéristiques électriques sont semblables à la SRAM.

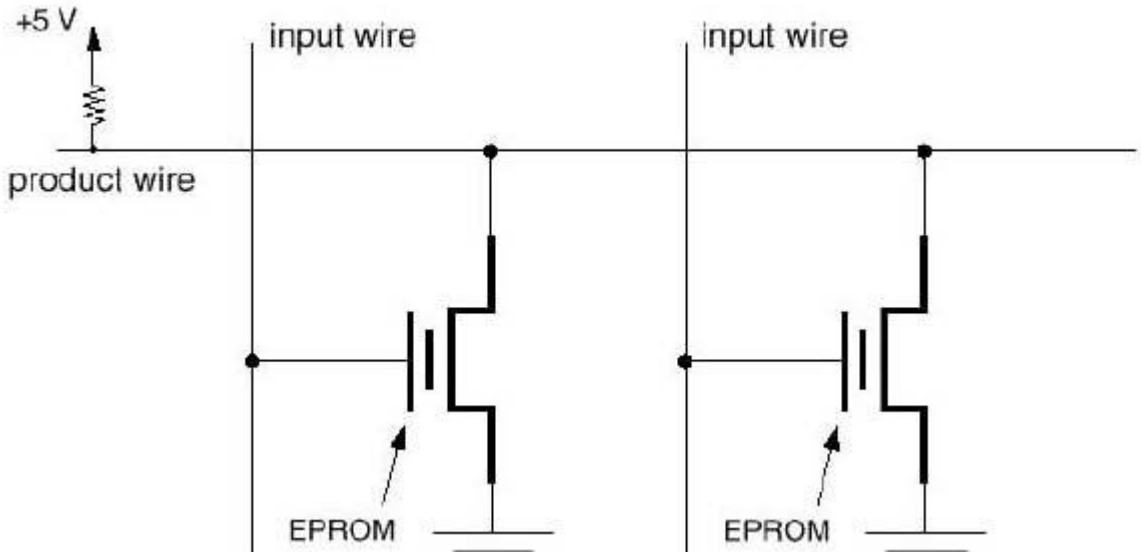


Figure II.15 : Technologie EPROM/EEPROM. [15]

II.9 - Configuration des FPGA:

La configuration du circuit est mémorisée sur la couche réseau SRAM et stockée dans une ROM externe. Un dispositif interne permet à chaque mise sous tension de charger la SRAM interne à partir de la ROM. Ainsi on conçoit aisément qu'un même circuit puisse être exploité successivement avec des ROM différentes puisque sa programmation interne n'est jamais définitive. On voit tout le parti que l'on peut tirer de cette souplesse en particulier lors d'une phase de mise au point. Une erreur n'est pas rédhibitoire, mais peut aisément être réparée.

II.9.1 Technique de configuration des FPGA:

La méthode de configuration du FPGA peut être divisée en:

- Mode Maître.
- Mode esclave.
- Mode JTAG (**J**oint **T**est **A**ction **G**roup). [22]

II.9.1.1- Modes maîtres

En mode maître, les données de configuration sont stockées dans des mémoires externes non volatiles telles que SPI FLASH, FLASH parallèle, PROM, etc. Pendant le processus de configuration, les données sont chargées dans les blocs logiques configurables du FPGA pour fonctionner comme une application spécifique. L'horloge de configuration est fournie par FPGA en mode maître.

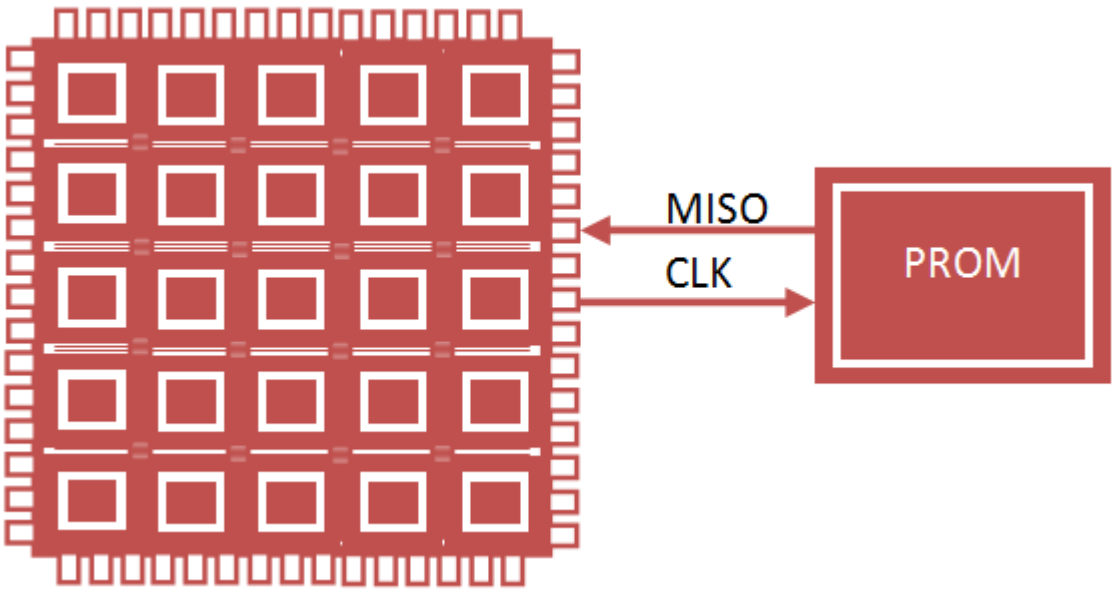


Figure II.16 : Configuration FPGA via une PROM externe. [22]

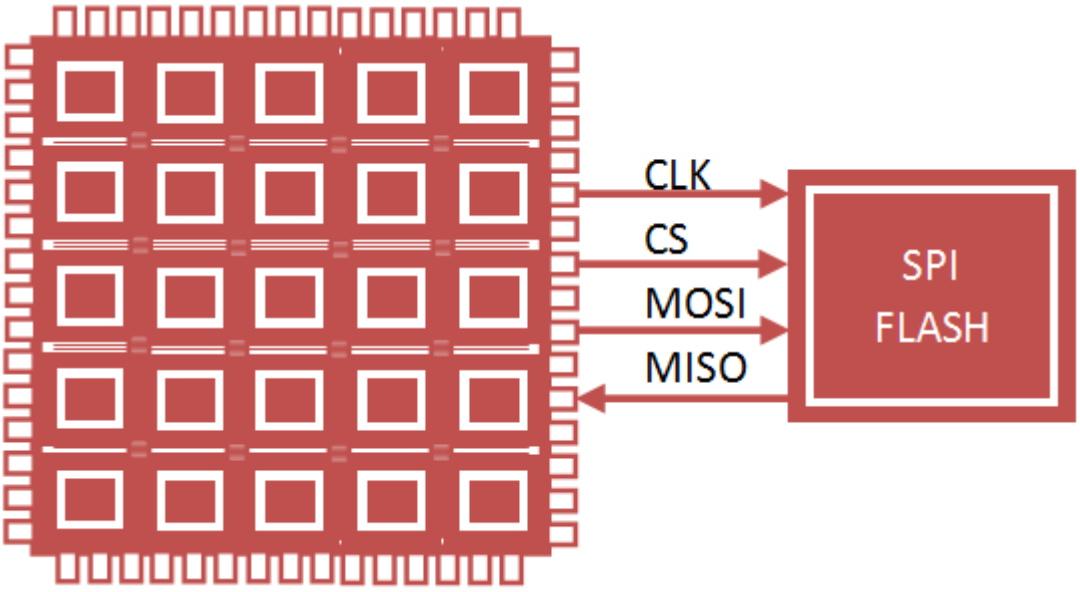


Figure II.17: Configuration FPGA via la mémoire SPI FLASH. [22]

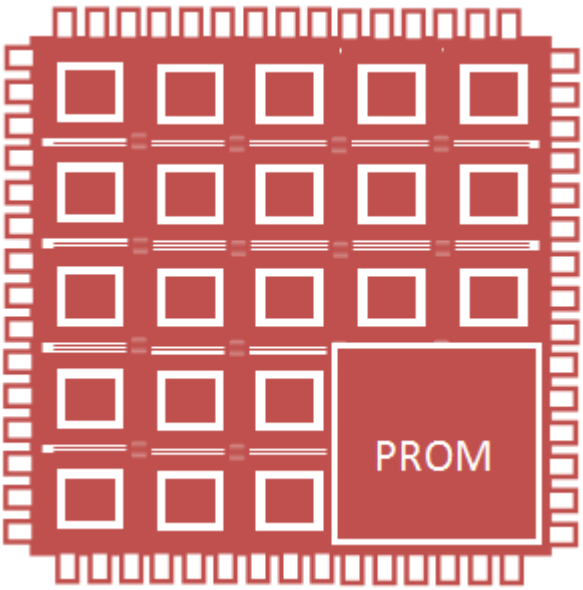


Figure II.18 : Configuration FPGA via FLASH interne. [22]

II.9.1.2- Mode esclave:

En mode esclave, l'ensemble du processus de configuration est contrôlé par un périphérique externe. Ces périphériques externes peuvent être des processeurs, des microcontrôleurs, etc. La configuration peut être effectuée en série ou en parallèle. L'entrée Clock est fournie par le périphérique externe pour le mode Slave.

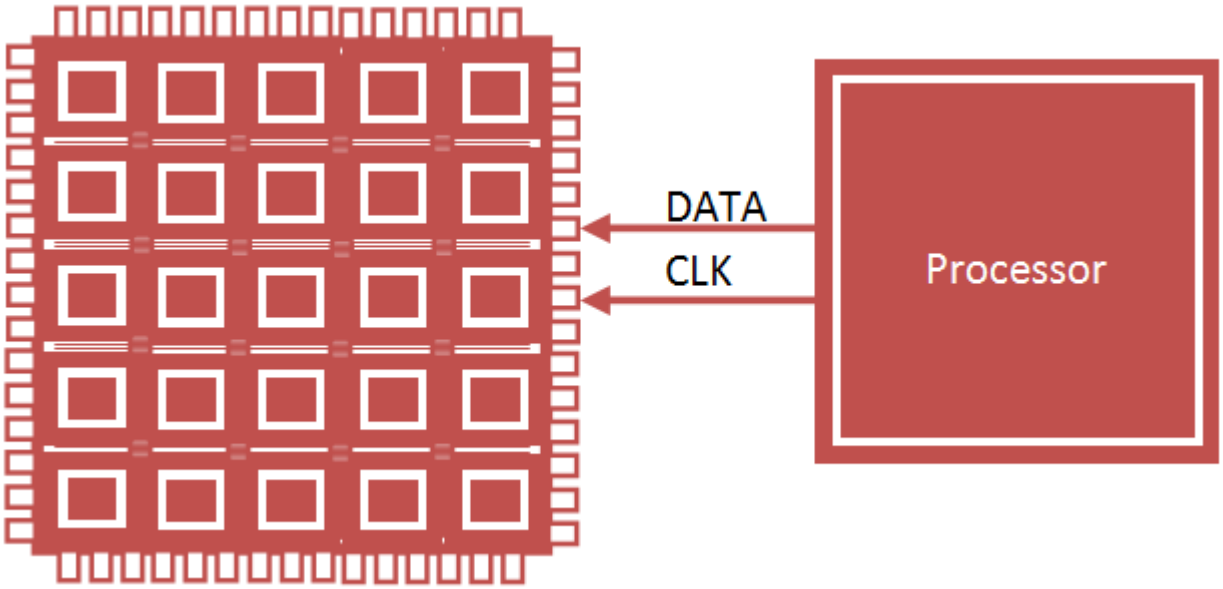


Figure II.19 : Configuration FPGA mode esclave. [22]

II.9.1.3- Connexion JTAG

L'interface JTAG à quatre fils est commune aux testeurs embarqués et au matériel de débogage. FPGA utilise principalement l'interface JTAG pour le téléchargement et le débogage de prototypes. JTAG comprend les lignes TCK, TMS, TDI et TDO pour la communication.

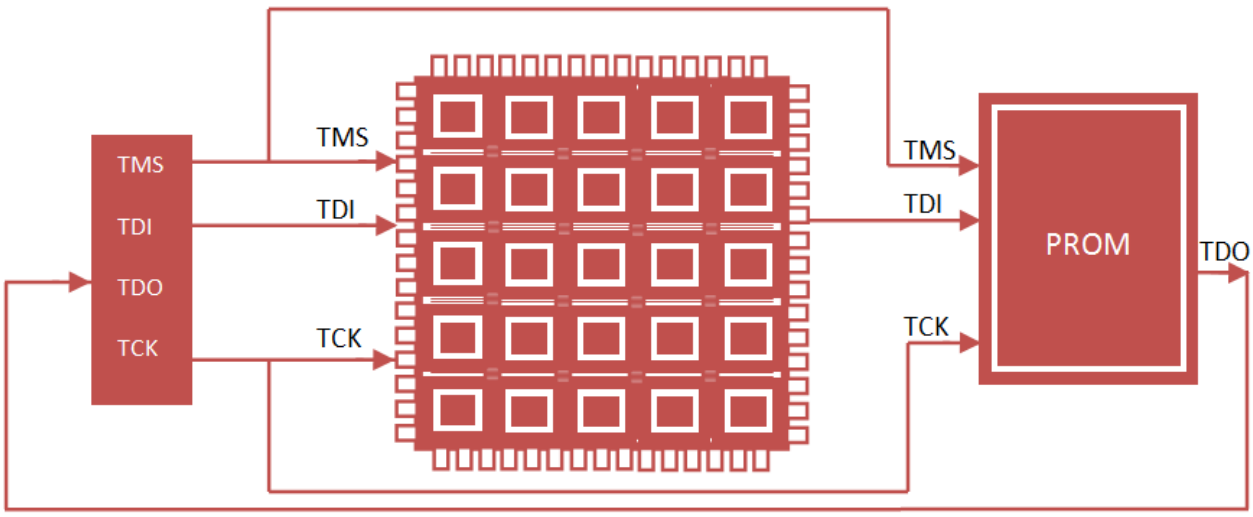


Figure II.20 : Configuration FPGA mode JTAG. [22]

II.9.2- Programmation des FPGA:

On commence par décrire le design soit en utilisant un langage de description matériel (tel VHDL, Verilog...) soit en rentrant directement le schématique. Le synthétiseur va générer la netlist, ensuite il faudra placer tous ces composants (si c'est possible, en effet, certains FPGA ne permettent pas d'émuler des Latchs) dans un FPGA et effectuer le routage entre les différentes cellules logiques. Au terme de ces étapes, le synthétiseur aura générer le bitstream qui sera prêt à être envoyer vers le FPGA. C'est seulement à ce moment là que la programmation proprement dite pourra avoir lieu.

II.10- Conclusion:

Dans ce chapitre, on a fait une brève étude sur le FPGA, on a présenté sa structure et son architecture détaillé et les techniques de configuration de ces circuits qui offrent des performances que n'on trouve pas dans d'autre circuits du coté flexibilité, configuration et reconfiguration, ils possèdent une gestion dynamique de connectivité à leurs environnements ce qui permet d'implémenter de différents algorithmes adéquats à des applications diverses tout en gardant une grande flexibilité de communication.

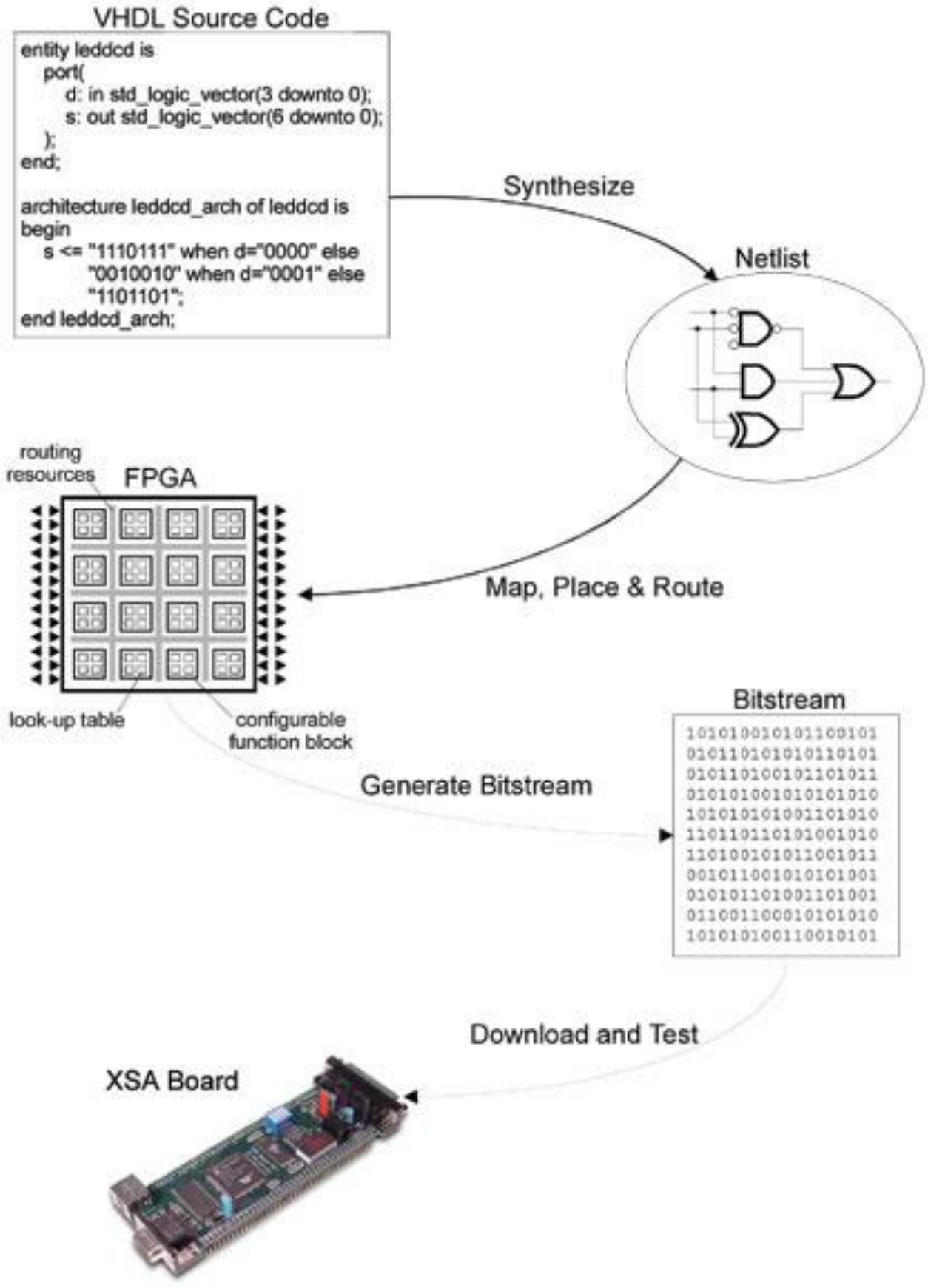


Figure II.21 : programmation d'un FPGA [15]

Chapitre 3

Notice

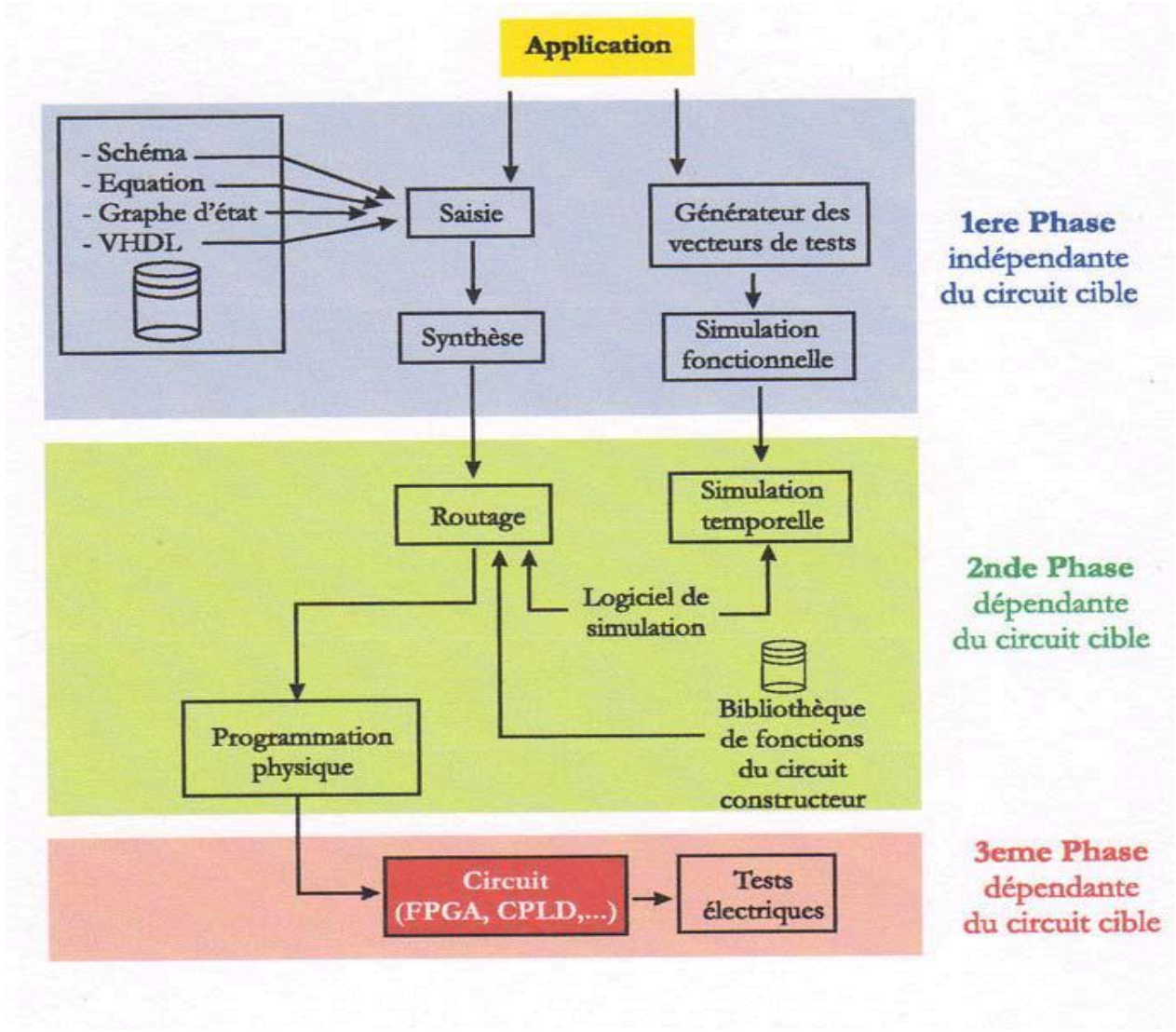
QUARTUS

III.1 - Présentation

Quartus est un logiciel développé par la société Altera, permettant la gestion complète d'un flot de conception CPLD ou FPGA. Ce logiciel permet de faire une saisie graphique ou une description HDL (VHDL ou verilog) d'architecture numérique, d'en réaliser une simulation, une synthèse et une implémentation sur cible reprogrammable.

Il comprend une suite de fonctions de conception au niveau système, permettant d'accéder à la large bibliothèque d'IP d'Altera et un moteur de placement-routage intégrant la technologie d'optimisation de la synthèse physique et des solutions de vérification.

De manière générale, un flot de conception ayant pour but la configuration de composants programmables se déroulent de la manière suivante :



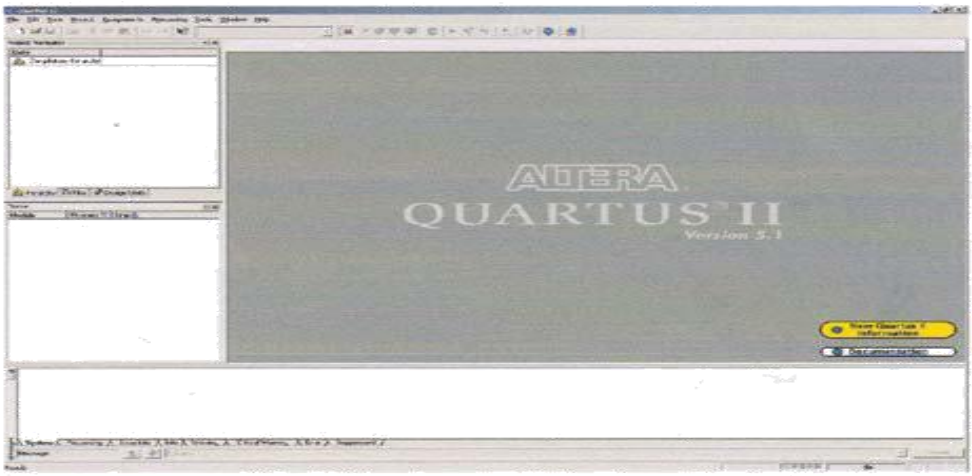
III.2 - Création d'un projet

Quartus est un logiciel qui travaille sous forme de projets c'est à dire qu'il gère un design sous forme d'entités hiérarchiques. Un projet est l'ensemble des fichiers d'un design que ce soit des saisies graphiques, des fichiers VHDL ou bien encore des configurations de composants (affectation de pins par exemple).

Pour lancer le logiciel, on cliquera sur :

Démarrer —Programmes —>Altera —>Quartus II

La fenêtre suivante s'ouvre :

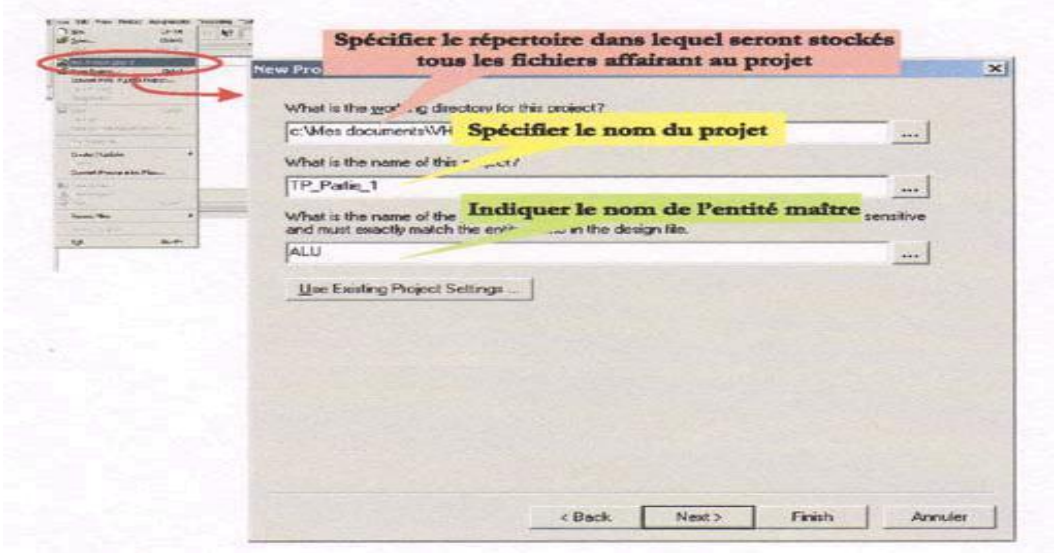


Afin de créer un nouveau projet aller dans le menu :

File —>New Project Wizard

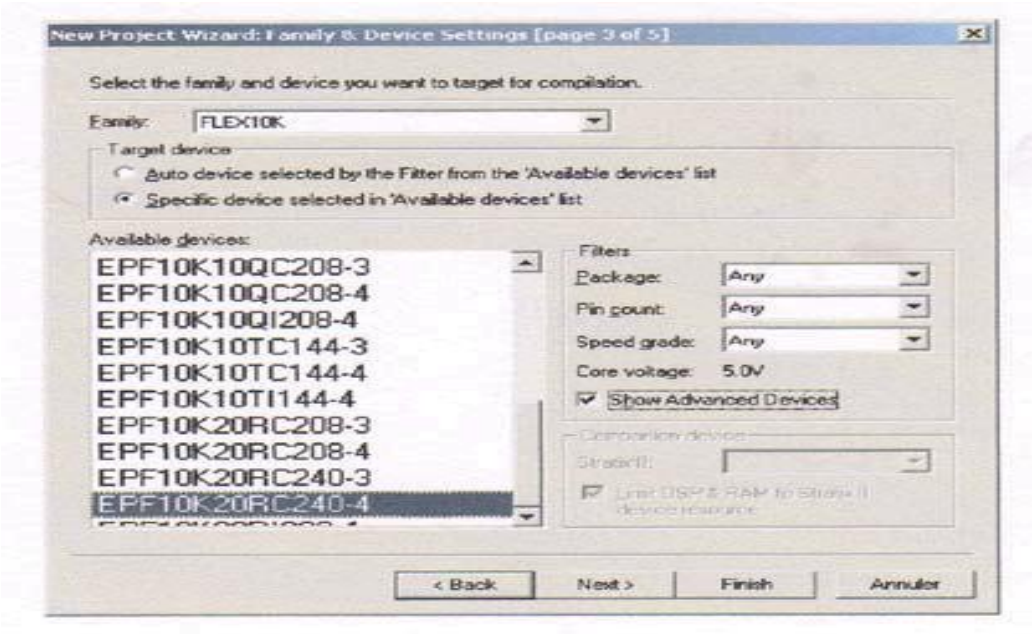
Puis se laisser guider. Une nouvelle fenêtre permettant de configurer le projet apparaît.

Dans cette dernière trois champs sont à renseigner :

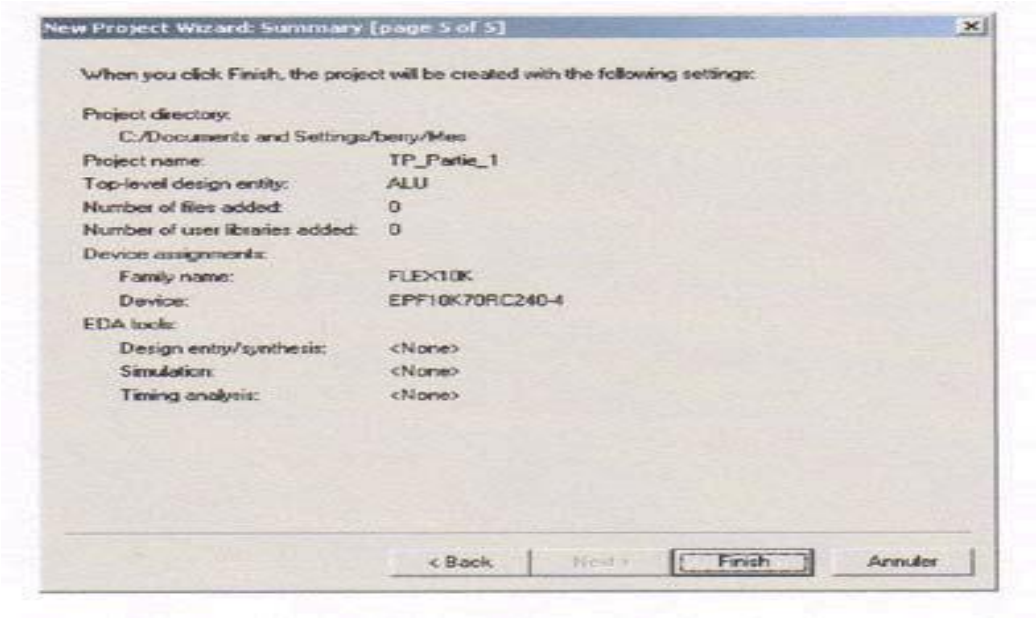


- ✓ L'emplacement du répertoire où seront stockés tous les fichiers. Il est conseillé de créer un répertoire propre afin d'assurer la sauvegarde des données d'une séance sur l'autre,
- ✓ Le nom du projet,
- ✓ Le nom de l'entité maître du projet: On appelle entité maître, le design qui correspond à la couche hiérarchique la plus haute. Par exemple s'il on conçoit un schéma nommé additionneur dans lequel il y aura plusieurs composants graphiques ou décrit en VHDL, alors additionneur sera l'entité maître. En d'autres termes, additionneur ne dépend pas d'autres fichiers, c'est le niveau le plus haut dans le design.

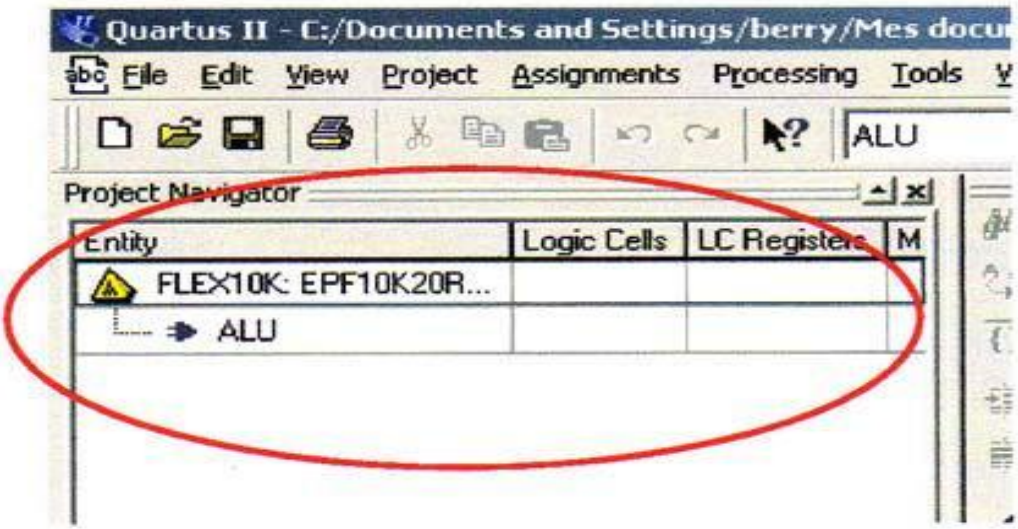
Cliquer sur **Next**, puis quand la fenêtre **Add Files** apparaît recliquer sur **Next**. Dans la fenêtre suivante intitulée **Family & Device Settings**, choisir le circuit logique programmable que l'on souhaite utiliser.



Quand la fenêtre **EDA Tool Settings** apparaît cliquer sur Next. Une fenêtre récapitulative apparaît :



Valider les choix par **Finish** ou bien faire **Back** pour des modifications éventuelles. Dans le navigateur de Projet, un onglet avec le type composant et l'entité maître apparaît :



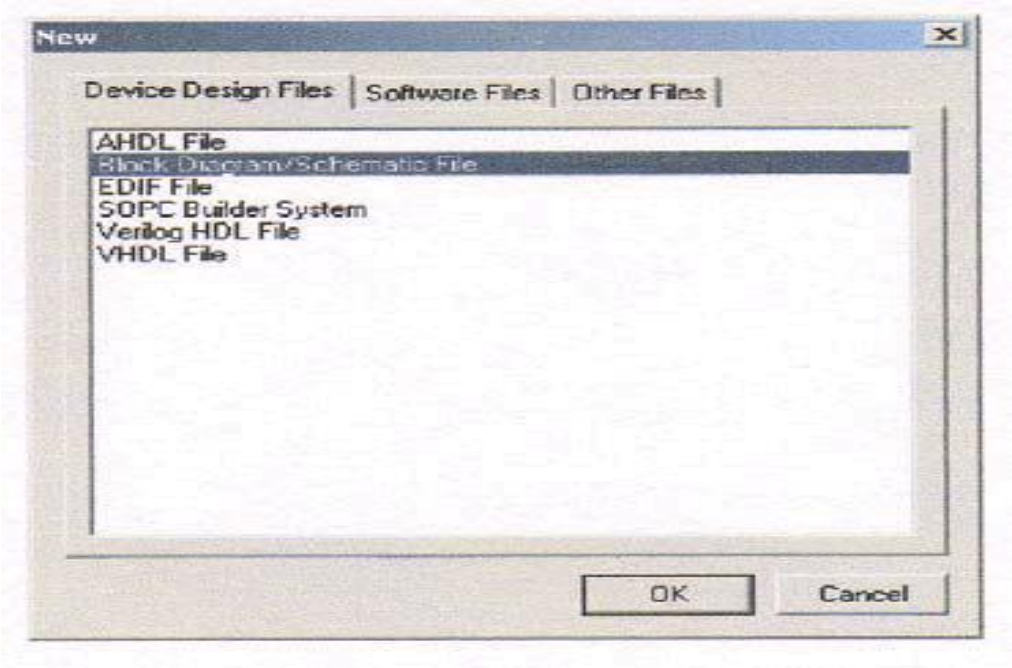
III.3- Saisie d'un projet

Cette étape permet de définir et configurer les différentes parties du projet. Quartus accepte plusieurs types de saisie à savoir :

- une saisie graphique en assemblant des symboles,
- une saisie textuelle à l'aide de différents langages (VHDL, Verilog, AHDL,...)

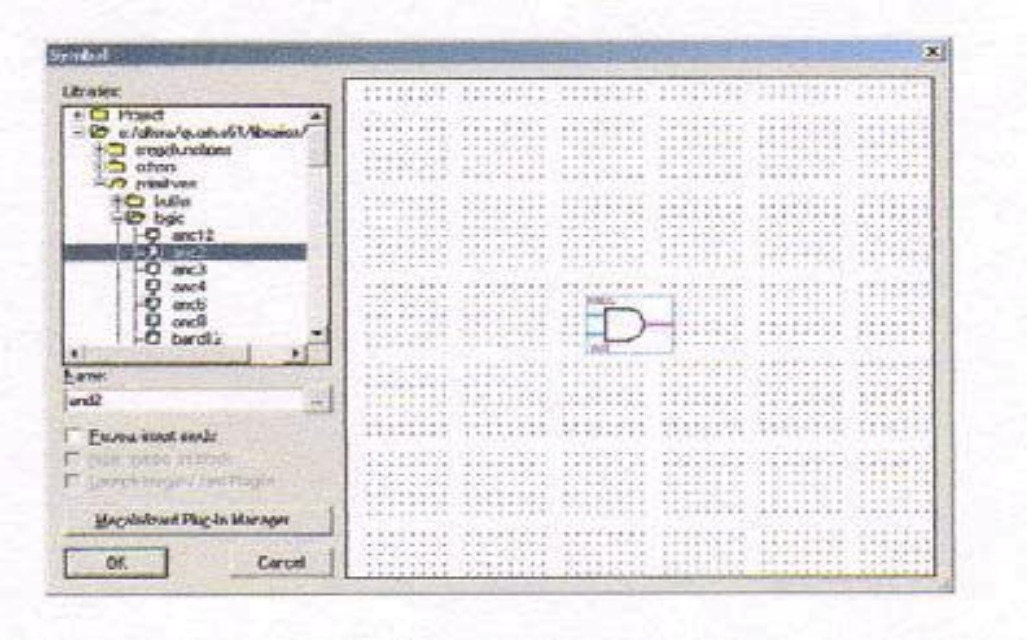
III.3.1. Saisie graphique

Pour saisir un projet en mode graphique, aller dans le menu : **File —>New**
La fenêtre suivante apparaît :



Choisir **Block Diagram/Schematic File** et faire **OK**. Une feuille blanche se crée intitulée **Block1.bdf**. On prendra soin de sauver cette feuille sous le nom de l'entité maître (ALU exemple). C'est maintenant cette feuille de saisie graphique qui a la hiérarchie la plus haute dans le projet.
Il convient maintenant d'insérer des symboles dans notre feuille. Pour cela, nous pouvons soit choisir des composants de la librairie Altera soit en créer en les décrivant en VHDL.

L'insertion d'un symbole se fait en cliquant dans la feuille avec le bouton de droite et en allant dans **Insert —>Symbol**. La fenêtre suivante s'ouvre :



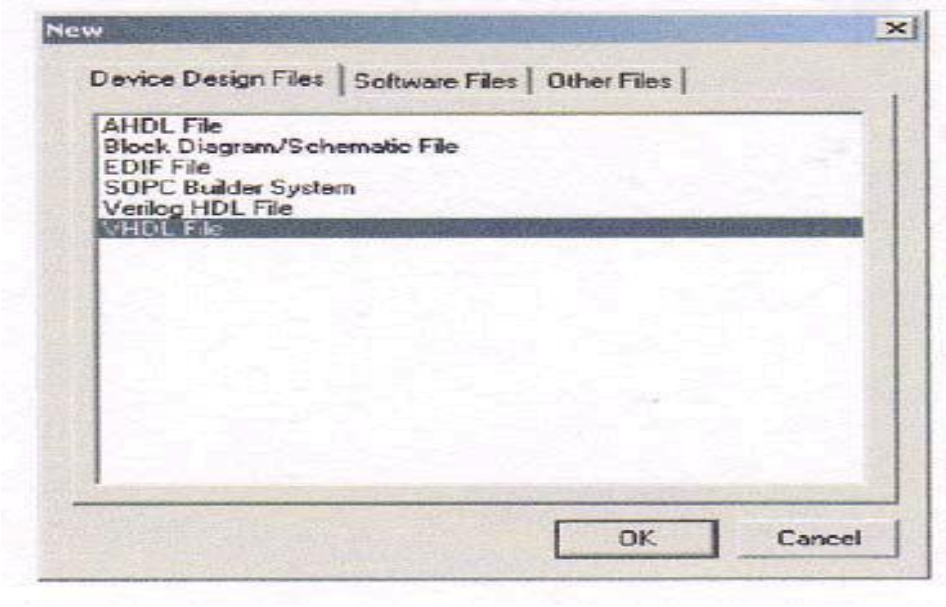
On remarquera sur l'onglet de gauche (Librairies), les bibliothèques propres au projet (personnel) et les bibliothèques natives d'Altera.

III.3.2. Saisie textuelle en VHDL

La saisie d'un composant VHDL se fait de la même manière que précédemment. Pour cela, aller dans le menu :

File —>New

La fenêtre suivante apparaît :



Choisir **VHDL File** et faire **OK**. Un petit éditeur de texte apparaît. Afin de fixer les idées, nous allons créer un petit composant réalisant un ET logique sur 4 bits par le programme suivant :

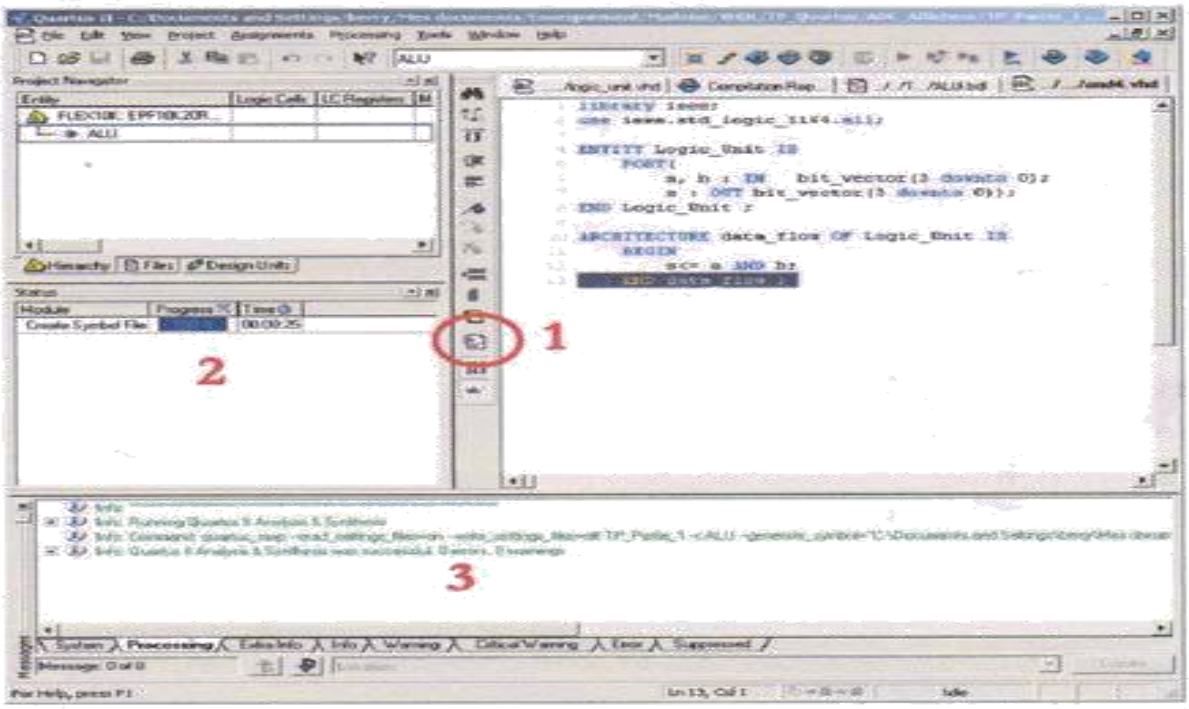
```
library ieee;
use ieee.std_logic_1164.all;

        ENTITY ET4 IS
                PORT(
                        a, b : IN   bit_vector(3 downto 0);
                        s : OUT bit_vector(3 downto 0));
         END ET4 ;

         ARCHITECTURE data_flow OF ET4 IS
                BEGIN
                        s<= a AND b;
                END data_flow ;
```

Une fois le code VHDL saisie, il convient de le sauver (**File —>Save** sous le nom ET4 par exemple) puis d'en vérifier la syntaxe. Pour cela, on cliquera sur l'icône entouré noté 1 et l'on suivra l'évolution de la vérification en 2 et les différentes informations s'afficheront en 3.

Il est important de sauver le fichier sous le même nom que l'entité. Bien que cela ne soit pas indispensable comme sous Maxplus, cela évite des intersections d'entité entre fichiers. Dans notre cas, l'entité s'appelle ET4 donc nous sauverons le fichier sous ET4.vhdl. On prendra aussi garde à ne pas appeler une entité du même nom qu'un composant natif Altera comme par exemple AND4.



Une fois que le fichier est OK, on peut alors créer un symbole graphique qui nous permettra d'insérer le composant dans la feuille graphique initiale (ALU). Pour cela, aller dans

File —»Create/Update —>Create Symbol File from Current File

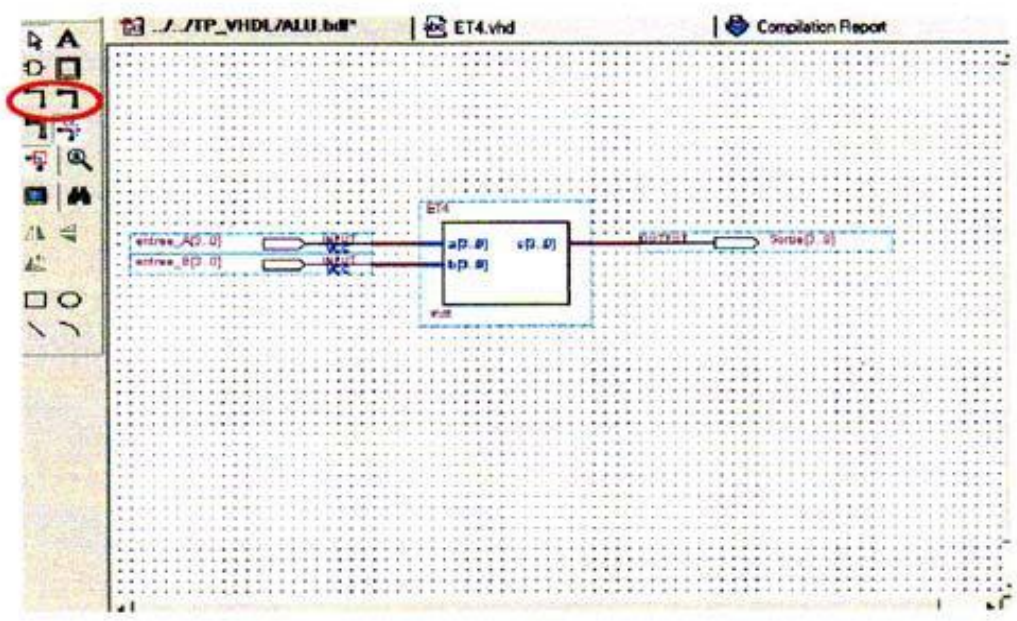
Le symbole correspondant à notre composant VHDL est maintenant créé. Nous pouvons l'instancier dans la feuille graphique ALU en l'insérant comme cela est expliqué précédemment. Normalement, le composant ET4 doit se trouver sous l'onglet Projet.

Afin d'en faire une simulation, on lui adjoindra des composants d'entrées/sorties (IO) permettant de repérer les signaux. Ces derniers se trouvent dans la librairie :

Altera —>Primitives —>Pin

La liaison entre les composants se fait à l'aide de l'icône entouré en rouge dans la fenêtre ci-dessous. On notera la différence entre les équipotentielles simples (trait fin) et les bus (trait large).

On doit se trouver avec un schéma de ce type :



Il est possible de renommer les pins en double-cliquant dessus. Une fenêtre de propriétés apparaît et il suffit de se laisser guider.

III.4 - Compilation

Cette étape consiste maintenant à compiler le schéma précédemment réalisé.

Durant la compilation, Quartus va réaliser 4 étapes :

La transformation des descriptions graphiques et textuelles en une structure électronique à base de portes et de registres : C'est la synthèse logique.

- L'étape de Fitting (ajustement) consiste à voir comment les différentes portes et registres (produit par la synthèse logique) peuvent être placés en fonction des ressources matérielles du circuit cible: C'est la synthèse physique.

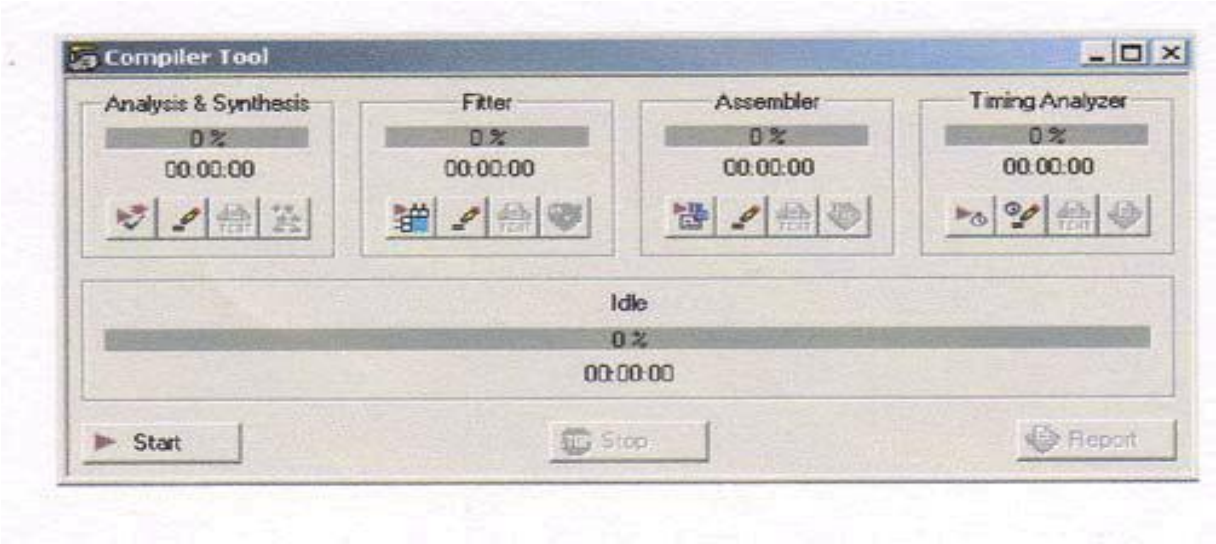
- L'assemblage consiste à produire les fichiers permettant la programmation du circuit. Ce sont des fichiers au format Programmer Object Files (.pof), SRAM Object Files (.sof), Hexadecimal (Intel-Format) Output Files (.hexout), Tabular Text Files (.ttf), and Raw Binary Files (.rbf). Dans notre cas, nous utiliserons toujours le format SOF pour les FPGA et le format POF pour les CPLD.

- L'analyse temporelle permet d'évaluer les temps de propagation entre portes et le long des chemins choisis lors de l'étape de « fitting ».

Pour lancer la compilation, cliquer sur :

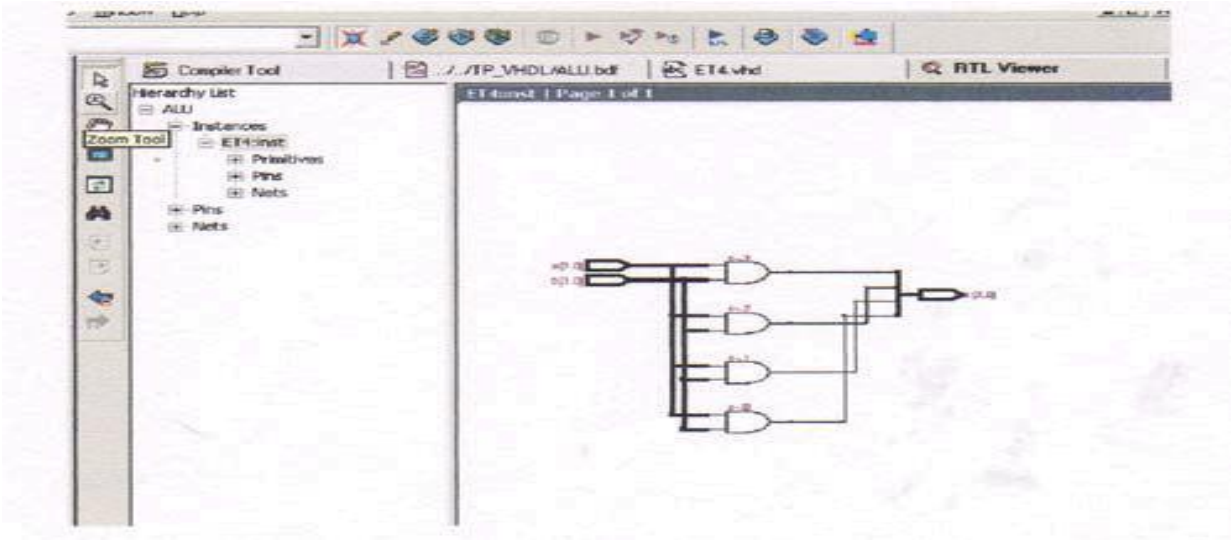
Processing —>Compiler Tool.

La fenêtre suivante apparait. On cliquera sur Start pour la compilation.



Normalement il ne doit pas y avoir de warning ou d'erreur. Si ce n'est pas le cas vérifiez dans la zone Processing (en bas où s'affichent les messages) la source du problème. En cliquant sur **Report,** on trouve une multitude d'information entre autres le pourcentage d'occupation du schéma dans le circuit programmable, les temps de propagation, les fréquences maximums d'utilisation etc…

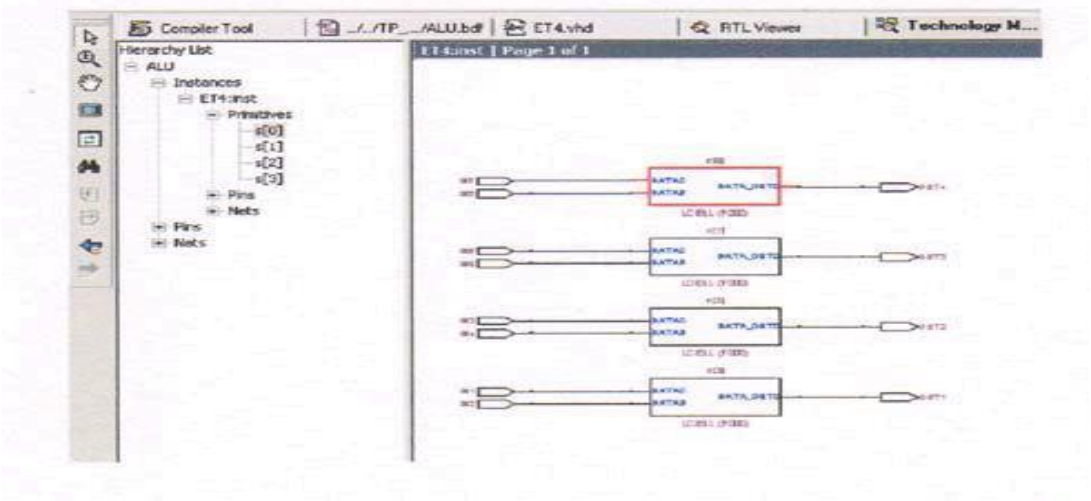
Afin d'illustrer cette partie, on pourra par exemple lancer la commande **Tools —>RTL Viewer** qui permet de voir comment le schéma (ALU) contenant le code VHDL a été transformé en portes et bascules. Cela permet de voir comment la synthèse logique s'est déroulée. Dans notre cas, nous retrouverons bien les 4 portes AND.



De même, nous pouvons visualiser la synthèse physique en utilisant la commande :

Tools —>Technology Map Viewer

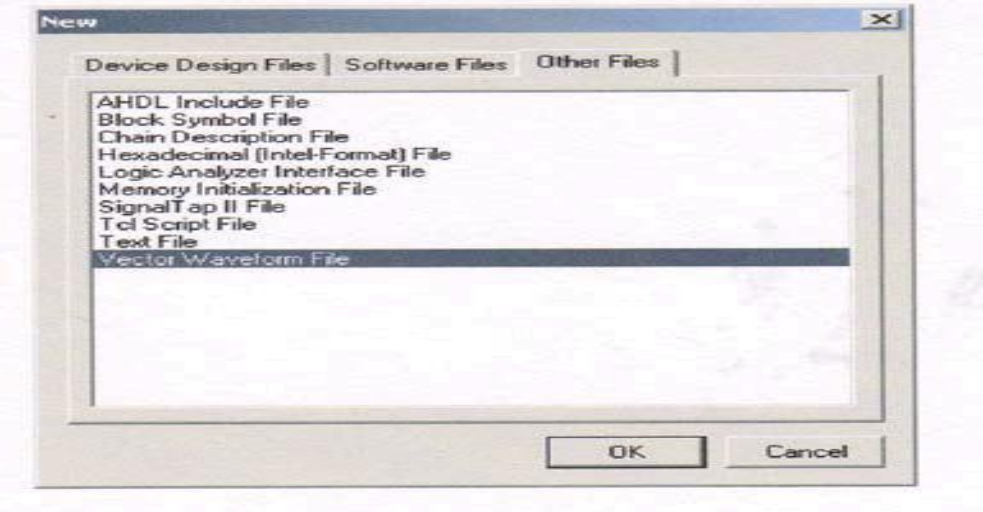
On retrouve les 4 instances placées dans le circuit et repérées par leurs références.



III.5 - Simulation

Afin de simuler le design réalisé, il convient de lui injecter des stimuli. Lorsque ces stimuli sont générés à partir d'un fichier on dit que l'on utilise un fichier de Bench.

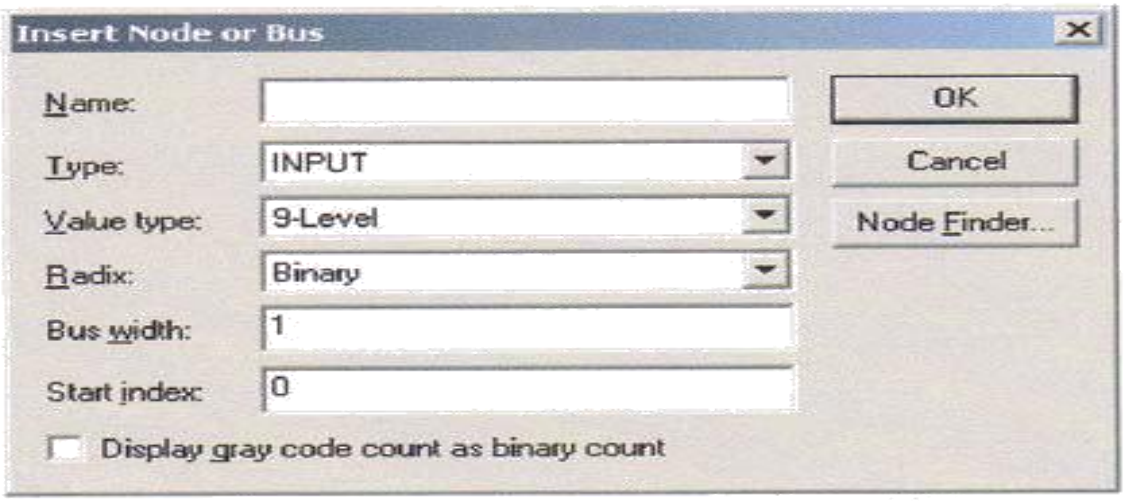
Dans le cas présent, nous allons simuler notre schéma en générant les stimuli à partir du Wave *Editor*. Pour cela, faites **File —>New**, aller dans l'onglet **Other Files** et sélectionnez **Vector Waveform File** (Fichier de simulation).



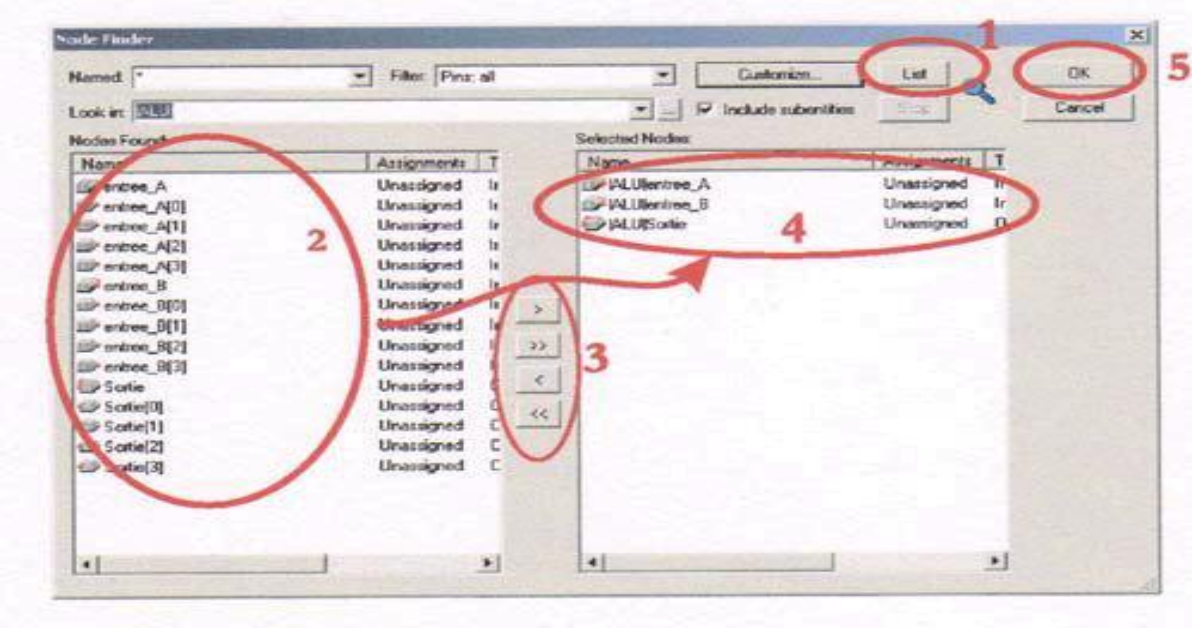
Faire une sauvegarde du fichier par **File —>Save** en lui donnant un nom compréhensible (par exemple Simul_ALU). Le fichier sera du type .vwf.

Pour changer la durée de simulation, on fera **Edit —>End Time** et pour réaliser des zooms ou voir toute la simulation, il est possible de cliquer dans la fenêtre avec le bouton de droite et de choisir **Zoom** dans le menu.

Il ne reste plus qu'à insérer les différents signaux de simulation c'est à dire les signaux d'entrée et les sorties. Pour cela, on fera **Edit —>Insert Node or Bus**. Dans la fenêtre



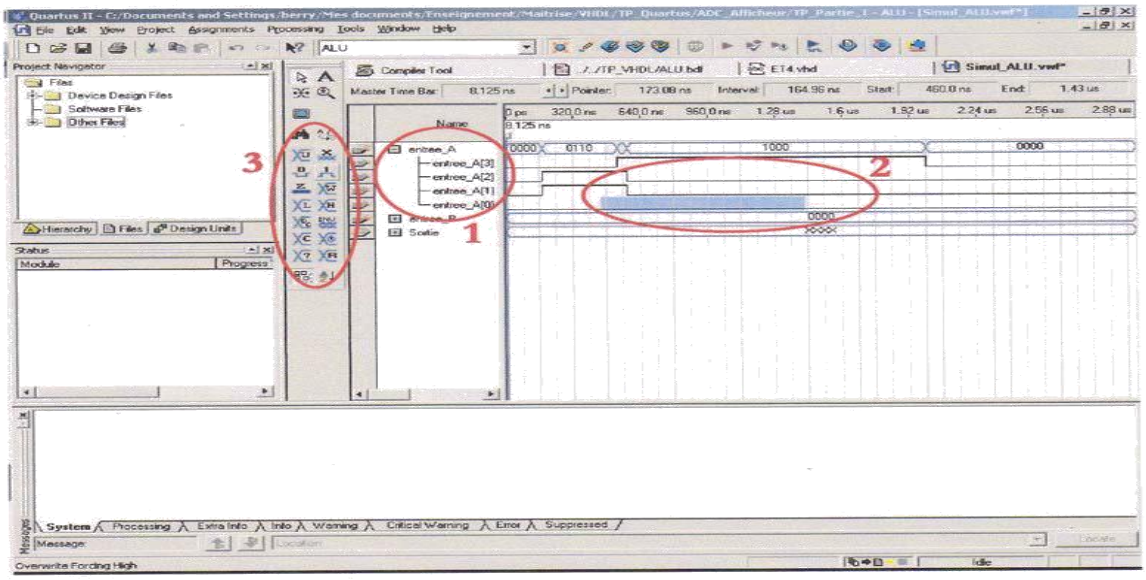
Cliquer sur **Node Finder**, ce qui permet de lancer le navigateur de signaux



Dans le **Node Finder**, cela se déroule en 5 étapes :

1. Cliquer sur list afin de faire apparaître les signaux du design,
2. Sélectionner les signaux voulus,
3. Cliquer sur la flèche correspondante
4. Vérifier que tous les signaux que vous voulez visualiser sont dans Selected Nodes
5. Valider par OK Cliquer sur OK dans la fenêtre Insert Node or Bus.

Les signaux apparaissent maintenant dans le visualiseur de signal. Si des bus ont été sélectionnés, il est possible de les "déplier" en bit à bit en cliquant sur le + (Zone 1).



Afin de donner des valeurs de stimuli, on sélectionne à la souris une partie du signal (2) et avec le menu (3) on lui attribue une valeur. On notera la possibilité d'insérer automatiquement une horloge (clock) ou un compteur.

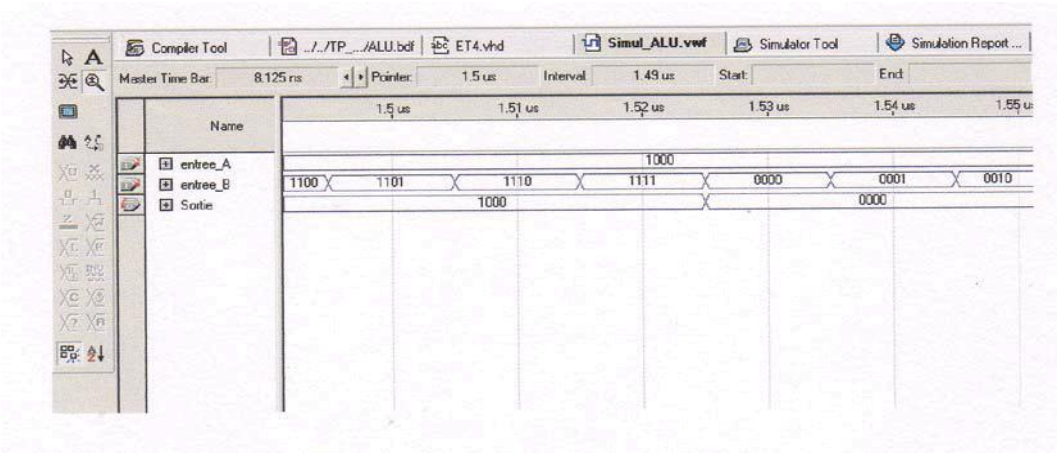
En dernier lieu, il ne reste plus qu'à lancer la simulation par la commande :

Tools →Simulator Tool.

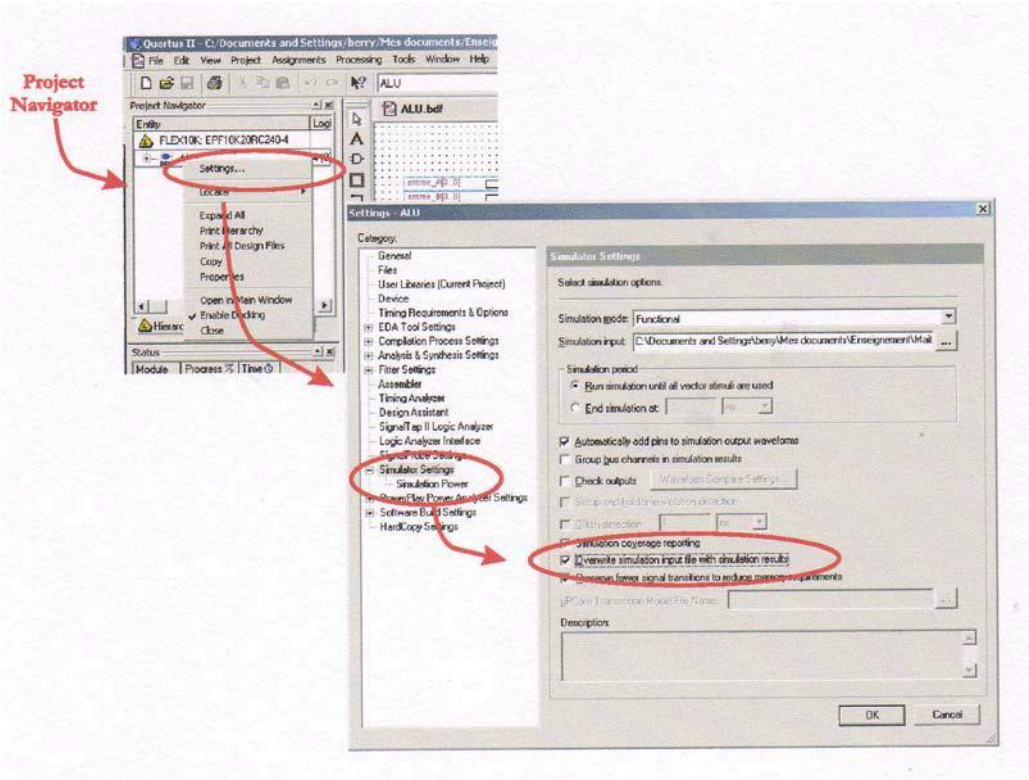
Dans cette fenêtre, il faut tout d'abord spécifier le fichier de simulation (par exemple Simul_ALU.wvf) dans la partie 1.

La zone 2 permet de choisir si l'on veut prendre en compte les délais (timing) ou non (functional). Dans ce dernier cas, il faut générer un modèle fonctionnel en cliquant sur le bouton 3.

Le bouton **Start**(4) permet de lancer la simulation et il est possible de voir le résultat en cliquant sur **Report**(5).



Afin de permettre le rafraîchissement automatique du visualiseur de signal, il suffit de modifier une option en cliquant avec le bouton de droite sur l'entité maître (ALU) dans le Project Navigator, puis en choisissant le menu **Settings**. Dans la nouvelle fenêtre aller dans l'onglet **Simulator Settings** et cocher la case **Overwrite simulation input file with simulation results**



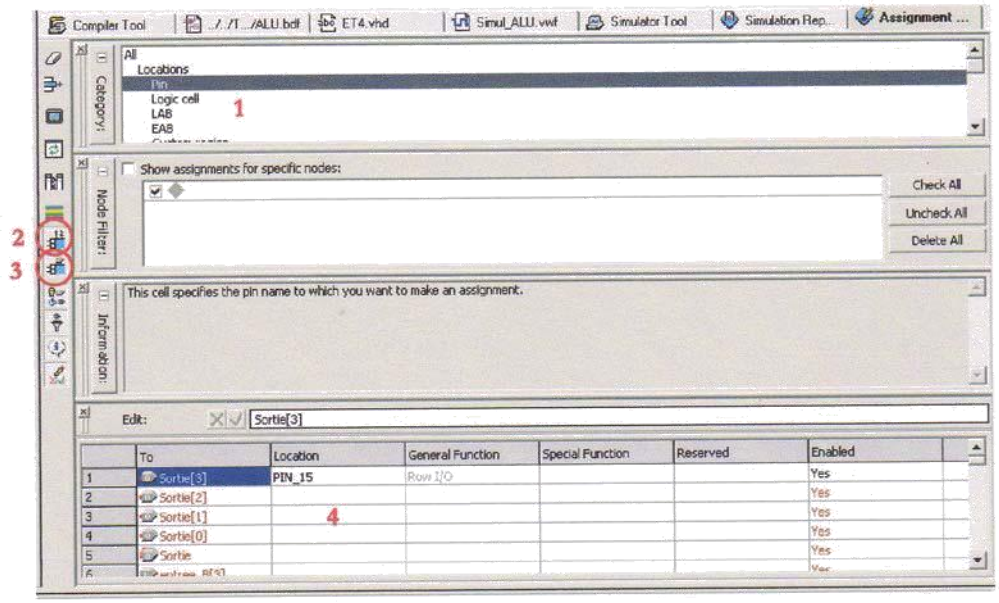
III.6- Programmation d'un circuit

C'est l'étape ultime. Pour cela, il faut assigner les pins d'entrée/sortie du design aux broches du circuit physique. Si rien n'est spécifié, Quartus affectera comme bon lui semble des pins (A éviter!).

III.6.1- Affectation des pins

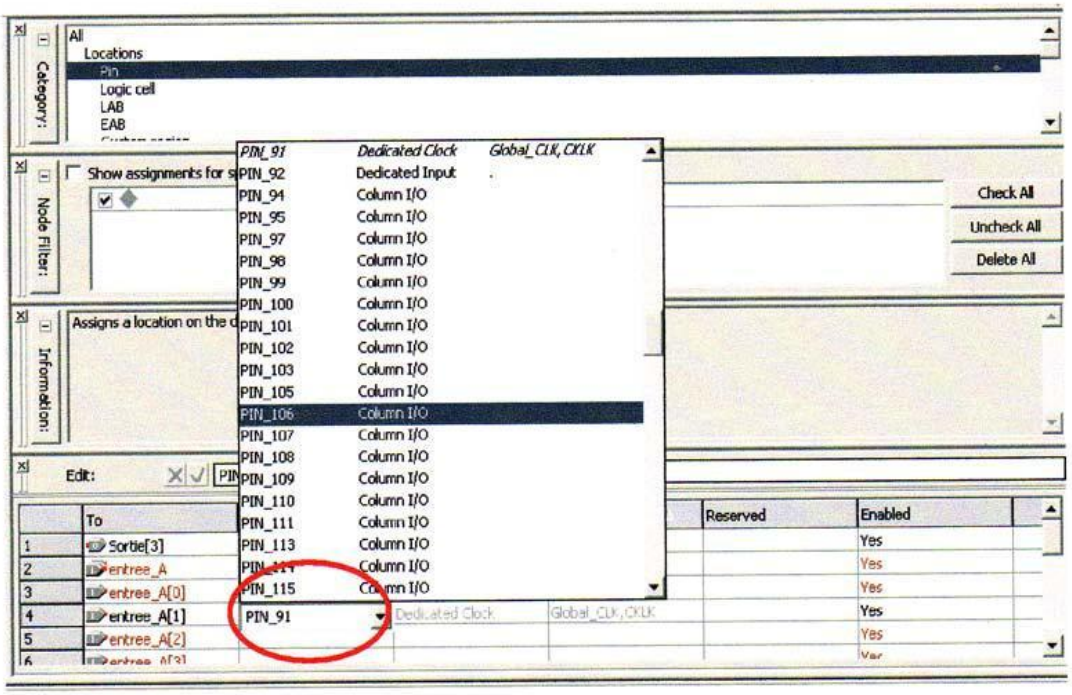
Afin de choisir quelle broche physique du circuit doit être connectée, lancer l'outil d'assignement de pins par :

Assignements →Pins



Dans la fenêtre correspondante, il est possible de choisir différents types d'assignement. Dans notre cas, nous sélectionnerons dans la partie 1 l'onglet Pin. Dans le menu de gauche, prendre garde à dévalider le bouton (2) permettant de faire afficher toutes les pins et bien valider le bouton (3) ne permettant l'affichage que des pins utilisées. De cette manière dans la partie 4 ne doivent s'afficher que les pins utilisées dans le design.

Afin de réaliser l'affectation, on double clique dans la colonne **Location** de la partie 4 au niveau de la pin voulue de manière à faire apparaître un menu déroulant où sont répertoriées les broches disponibles du circuit.



La liste des broches utilisables pour le FPGA et sortant sur les connecteurs est données en annexe. On notera que certaines broches sont réservées pour des horloges, des resets, des CS (Chip Select),... ou sont directement câbles sur des périphériques tels que les afficheurs, les boutons poussoirs,...

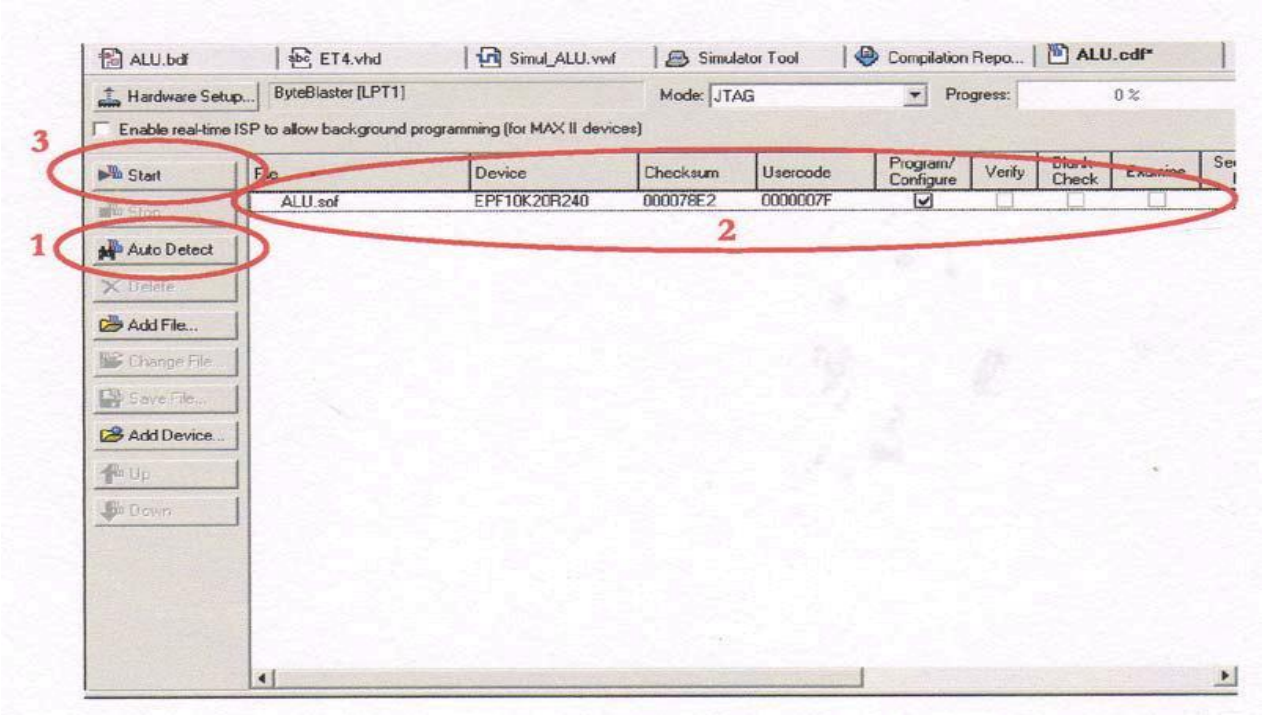
Dans le cas du design de la porte NAND, on peut par exemple câbler sur chacun des poussoirs un bit de l'entrée A et un bit de l'entrée B, puis câbler la sortie sur une led de l'afficheur.

On notera que l'état bas est à 5V et l'état haut est à 0V sans mettre des inverseurs aux entrées et sorites.

III.6.2- Programmation du circuit

La programmation du circuit se fait via le protocole JTAG (Joint Test Action Group). Pour cela vérifier que les connections entre le PC (port parallèle) et la carte via le module **ByteBlaster** sont opérationnelles. Si tout est bon et que la carte Altera est sous tension, lancer le programmateur par **Tools →Programmer**, puis cliquer sur le bouton **Autodetect**

(1). Si le PC ne détecte pas la carte, une erreur doit apparaître du type Unable to scan device Chain. Hardware is not connected.



Vérifier dans la partie 2 que le fichier .sof est bien là et que la case **Program/Configure** est cochée, puis cliquer sur **Start**. [23]

Chapitre 4

Le Microcontrôleur

IV.1 - Introduction

L'évolution des systèmes électroniques amène de plus en plus souvent les concepteurs à remplacer l'électronique câblée à base de nombreux circuits intégrés par un circuit programmable qui rempli à lui seul toutes les fonctions. Les microcontrôleurs appartiennent à cette famille de circuits. Le microcontrôleur apparaît donc comme un système extrêmement complet et performant, capable d'accomplir une ou plusieurs tâches très spécifiques, pour lesquelles il a été programmé. Ces tâches peuvent être très diverses, si bien qu'on trouve aujourd'hui des microcontrôleurs presque partout: ils sont fréquemment utilisés dans les systèmes embarqués, comme les contrôleurs des moteurs automobiles, (airbags, climatisation, ordinateur de bord, alarme...), les télécommandes, dans les appareils électroménagers (réfrigérateurs, fours à micro-ondes...), les téléviseurs et magnétoscopes, les téléphones sans fil, les périphériques informatiques (imprimantes, scanners...), les avions et vaisseaux spatiaux, les appareils de mesure ou de contrôle des processus industriels, ...

La force du microcontrôleur, qui lui a permis de s'imposer de manière si envahissante en si peu de temps, c'est sa spécialisation, sa très grande fiabilité et son coût assez faible (pour les modèles produits en grande série, notamment pour l'industrie automobile).

IV.2 - Définition du microcontrôleur

Un **microcontrôleur** (en notation abrégée **µc**, ou **MCU** en anglais) est un circuit intégré qui rassemble les éléments essentiels d'un ordinateur : processeur, mémoires (mémoire morte et mémoire vive), unités périphériques et interfaces d'entrées-sorties.

Les microcontrôleurs se caractérisent par un plus haut degré d'intégration, une plus faible consommation électrique, une vitesse de fonctionnement plus faible et un coût réduit par rapport aux microprocesseurs polyvalents utilisés dans les ordinateurs personnels. Les microcontrôleurs sont des circuits programmables capables d'exécuter des programmes et qui possèdent des circuits d'interface intégrés avec le monde extérieur. [24]

IV.3 – Microprocesseur et Microcontrôleur

Les microprocesseurs et les microcontrôleurs sont des puces électroniques programmables typiques utilisées à des fins différentes. La différence clé entre eux est qu'un microprocesseur est un moteur de calcul programmable constitué d'une unité arithmétique et logique, d'un processeur et de registres, capable d'effectuer des calculs et de prendre des décisions. Tandis qu'un microcontrôleur est un microprocesseur spécialisé considéré comme un ordinateur sur une puce car il intègre des composants tels qu'un microprocesseur, une mémoire et des E/S.

Il y a une différence fondamentale entre un microprocesseur et un microcontrôleur :

- ✓ le microcontrôleur intègre dans un même boîtier, un microprocesseur, de la mémoire, et des interfaces entrées/sorties.
- ✓ le microprocesseur se présente sous la forme d'un boîtier qui nécessite des éléments externes, comme de la mémoire et des circuits d'interfaces. [25]

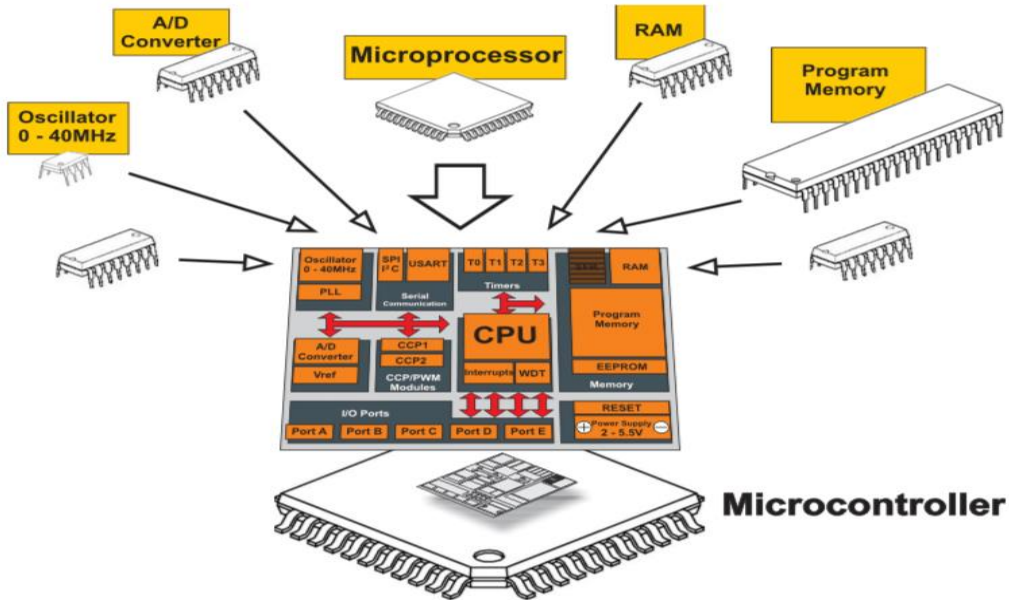


Figure IV.1 : Contenu type d'un microcontrôleur [25]

IV.4- Structure de base d'un microcontrôleur

Voici généralement ce que l'on trouve à l'intérieur d'un microcontrôleur:

- Un processeur (C.P.U.),
- Des bus,
- De la mémoire de donnée (RAM),
- De la mémoire programme (ROM),
- Des interfaces parallèles pour la connexion des entrées / sorties,
- Des interfaces séries (synchrone ou asynchrone) pour le dialogue avec d'autres unités,
- Des timers pour générer ou mesurer des signaux avec une grande précision temporelle. [25]

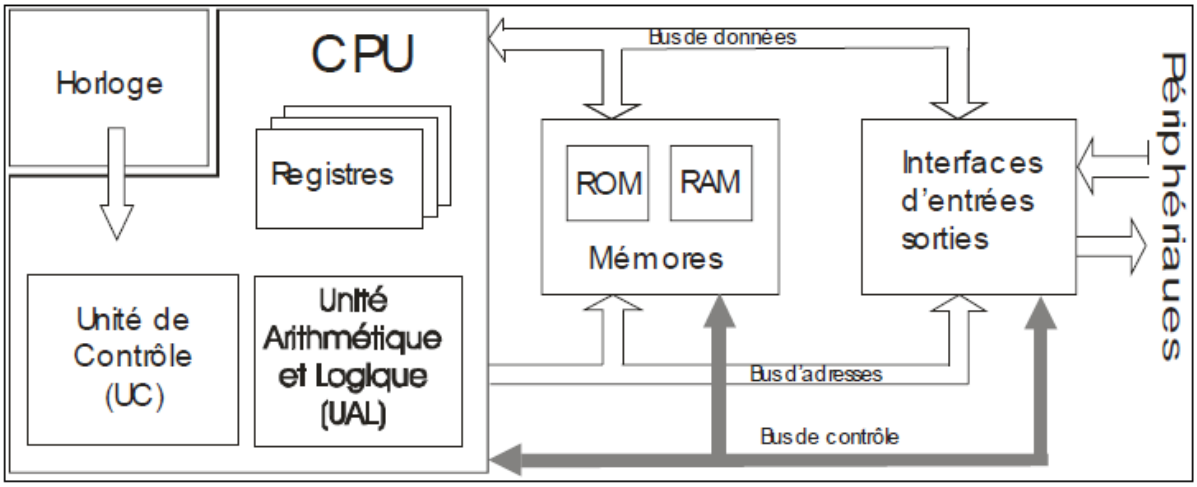


Figure IV.2 : schéma d'un microcontrôleur [25]

IV.4.1 - Le microprocesseur élément de base, est un circuit intégré à très grande échelle d'intégration (VLSI) capable d'exécuter automatiquement des instructions (opérations élémentaires) qu'il ira chercher dans la mémoire. Il remplit donc les fonctions d'une unité centrale de traitement (**CPU**, Central Processing Unit) en un seul boîtier.

IV.4.1.1- Architecture d'un CPU

Le microprocesseur est constitué des unités fonctionnelles suivantes :

- Unité Arithmétique et Logique (UAL ou ALU en anglais),
- des registres,
- une unité de contrôle (UC).

IV.4.1.1.1- Unité arithmétique et logique

Cet organe, interne au microprocesseur, permet la réalisation des opérations arithmétiques (Addition, Soustraction…) et logiques (AND, OR, XOR…). Outre les opérations arithmétiques et logiques, l'UAL réalise aussi les opérations de décalage et de rotation.

IV.4.1.1.2- Les registres

Il existe deux types de registres : les registres à usage général, et les registres d'adresses (ou pointeurs).

IV.4.1.1.2.1- Les registres d'usage général (Registres de travail)

Ce sont des mémoires rapides à l'intérieur du microprocesseur ; ils permettent à l'UAL de manipuler des données à vitesse élevée.
L'adresse d'un registre est associée à son nom (on donne généralement comme nom une lettre **A**, **B**, **C**…).

IV.4.1.1.2.2- Les registres d'adresses (pointeurs)

Ce sont des registres connectés sur le bus adresses. Ils sont utilisés pour l'adressage de la mémoire. On peut citer comme registres :

- Le compteur de programme **PC** (appelé aussi compteur ordinal) : le microprocesseur utilise ce registre pour repérer l'instruction à exécuter à un instant donné. Celui-ci contient toujours l'adresse de la prochaine instruction à exécuter.
- Le pointeur de pile (Stack Pointer **SP**), pointe toujours le sommet de la pile. La pile est une partie de la mémoire de données de type **LIFO** (Last In First Out) utilisée pour sauvegarder l'adresse de retour d'un sous-programme et/ou des variables utilisateurs.
- Les registres pointeurs de données ou d'index : utilisés pour l'adressage indirect de la mémoire de données.

IV.4.1.1.3- L'unité de contrôle (UC)

Elle permet de séquencer le déroulement des instructions. Elle effectue la recherche en mémoire des instructions, le décodage et l'exécution de l'instruction recherchée. Elle est composée essentiellement :

- d'un registre d'instruction (RI), recevant le code de l'instruction à exécuter.
- d'un décodeur d'instruction, permettant de déterminer le type de l'instruction à exécuter.
- d'un Bloc logique de contrôle (ou séquenceur) : Il organise toutes les étapes d'exécution des instructions au rythme d'une horloge et élabore tous les signaux de synchronisation internes et externes du microprocesseur.

Toutes les informations qu'utilise le microprocesseur sont stockées dans des **mémoires**, en particulier le programme ; le fonctionnement du microprocesseur est entièrement conditionné par le contenu de celles-ci. Ces mémoires contiennent deux types d'informations : le **programme** et les **données** nécessaires pour la réalisation d'une tâche précise.

IV.4.2- Les mémoires

Les mémoires sont des circuits intégrés à grande échelle d'intégration, capables de sauvegarder des informations binaires de façon permanente ou temporaire. Elles sont liées étroitement aux microprocesseurs puisqu'elles constituent l'élément de stockage de premier niveau.
On distingué deux type de mémoires :

IV.4.2.1- Les mémoires vives (**RAM** : Random Access Memoy) : Ce sont des mémoires volatiles, le maintien de l'information dépend de la présence de l'alimentation. Toute coupure de l'alimentation provoque la perte des informations. Elles sont utilisées pour stocker généralement les données temporaires.

IV.4.2.2- Les mémoires mortes (**ROM** : Read Only Memory) : gardent les informations même en absence d'alimentation. Ces mémoires contiennent des informations figées (souvent des programmes) et que l'accès ne se fait qu'on lecture seule.

IV.4.3- Les Bus :

Dans un système à microprocesseur, l'interface d'entrée-sortie permet d'assurer la liaison entre l'unité centrale de traitement et l'environnement extérieur (périphériques). Le microprocesseur échange les informations avec les composants qui lui sont associés (mémoire et périphériques I/O) au moyen d'un ensemble des lignes de connexions appelé **bus**.

Un bus : est un ensemble de fils qui assure la transmission du même type d'information. On distingue trois types de bus véhiculant les informations dans un système de traitement à microprocesseur :

IV.4.3.1- Le bus de données : bidirectionnel, assure le transfert des informations entre le microprocesseur et son environnement. Le nombre de lignes du bus de données définie la capacité de traitement du microprocesseur ; selon le microprocesseur la largeur du bus peut être de 8 bits, 16 bits, 32 bits, 64 bits …

IV.4.3.2- Le bus d'adresses : unidirectionnel, permet la sélection de la case contenant l'information à traiter dans un espace mémoire (ou espace adressable). L'espace adressable peut avoir 2^n emplacements, avec *n* est le nombre de lignes du bus d'adresses.

IV.4.3.3- le bus de contrôle : constitué de quelques lignes, assure la synchronisation du flux d'informations sur les bus de données et d'adresses. [25] [26]

IV.5- Différentes architectures

Tous les microcontrôleurs utilisent l'une des 2 architectures nommées **HARVARD** et **VON NEUMANN**. Elles représentent les différentes manières d'échange de données entre le CPU (microprocesseur interne) et la mémoire. [25]

IV.5.1- Architecture VON NEUMAN (du nom d'un des savants qui a contribué à la mise au point des premiers ordinateurs). La mémoire programme, la mémoire de données et les périphériques d'entrées/sorties partagent le même bus s'adresses et de données.

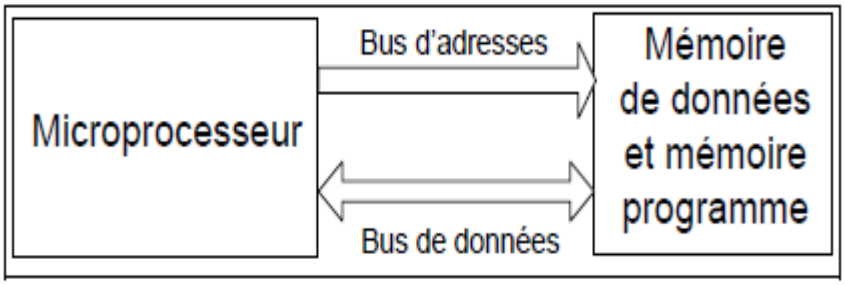


Figure IV.3: Architecture VON NEUMAN [25]

IV.5.2- Architecture HARVARD, sépare systématiquement la mémoire de programme de la mémoire de données : l'adressage de ces mémoires est indépendant. Ce type d'architecture favorise l'accès simultané aux mémoires de programme et de données.

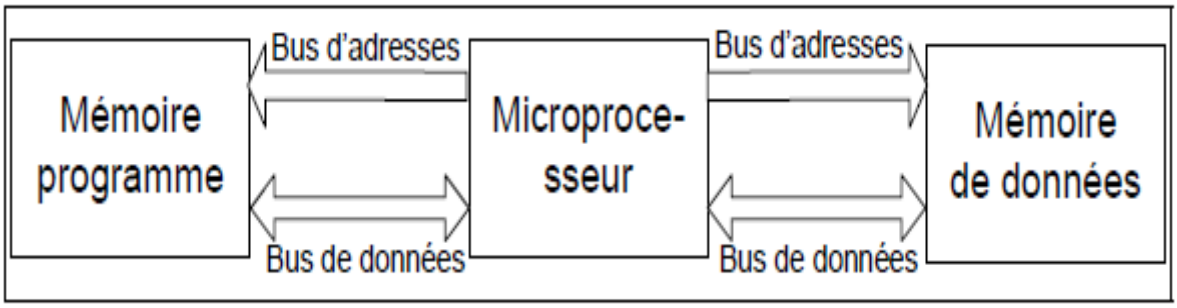


Figure IV.4 : Architecture HARVARD [25]

IV.5.3- Avantages et inconvénients

	VON NEUMANN	HARVARD
Avantages	- Jeu d'instructions riches - Accès à la mémoire facile.	- Jeu d'instructions pauvre, mais facile à mémoriser. - Le codage des instructions est facile, chaque instruction est codée sur un mot et dure un cycle machine. - Le code est plus compact.
Inconvénients	- Le temps pour exécuter une instruction est variable. - Le codage des instructions se fait sur plusieurs octets.	- Le jeu d'instruction est très pauvre, par exemple pour effectuer une comparaison il faut faire une soustraction. - Les accès aux registres internes et la mémoire sont très délicats.

[25]

IV.6- Jeu d'instructions

Toutes les instructions compréhensibles par les microcontrôleurs forment ce que l'on appelle le jeu d'instructions.

Au niveau de jeu d'instructions, les microprocesseurs se réparties en deux grandes catégories appelées CISC et RISC :

IV.6.1- Architecture CISC (Complex Instruction Set Computer) : une instruction peut effectuer plusieurs opérations élémentaires (faire par exemple une opération arithmétique avec chargement du résultat dans la mémoire). La longueur de l'instruction et le temps d'exécution varie d'une instruction à l'autre.

IV.6.2- Architecture RISC (Reduced Instruction Set Computer) : les processeurs RISC possèdent un jeu d'instruction réduit où chaque instruction effectue une opération élémentaire. Seules les instructions **Load** et **Store** accèdent à la mémoire. La plupart des instructions ont la même taille et s'exécute sur un seul cycle d'horloge. [25] [26]

IV.7- Les éléments de choix d'un µC

IV.7.1-Architecture

- ALU (8, 16, 32, 64 bits)
- Structure du processeur (Harvard, Von Neumann)
- Type de processeur (RISC, CISC)
- Taille des mémoires programme et donnée
- Nombre de ports d'entrée/sortie

IV.7.2-Fonctionnalités

- Fonctions analogiques : CAN, CNA, Comparateur, etc.
- Fonctions de timing : Timer, Watchdog, etc.
- Fonctions de communication : UART (Communication série), USB, I2C, etc.
- Facilité de programmation : In-Circuit Serial Programming, Self Programming, etc.

IV.7.3- Caractéristiques électriques :

4. Fréquence d'horloge.
5. Tensions d'alimentation.
6. Consommation d'énergie, modes faible consommation d'énergie, etc.

IV.7.4- Caractéristiques physiques :

-- Type de boîtier : DIL, PLCC, etc.

IV.8 – La Programmation

Le microcontrôleur exécute le programme chargé dans sa mémoire FLASH. Les mots binaires sont considérés par le CPU comme une commande.

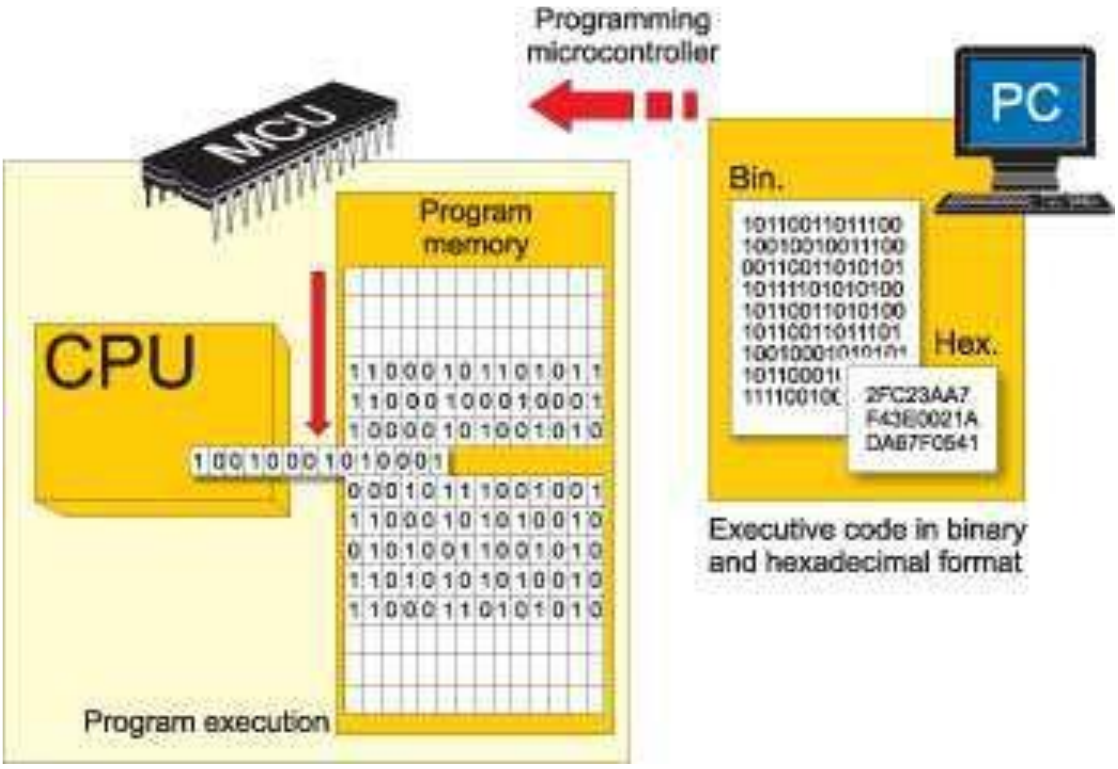


Figure IV.5 : Programmation du microcontrôleur

IV.9 – Les codes VHDL

IV.9.1- Code VHDL Registre 64 bit

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

ENTITY Reg64bits IS
PORT (
  clk,raz :IN std_logic;
  D: in std_logic_vector(63 downto 0));
  Q: out std_logic_vector(63 downto 0));
END Reg64bits;

ARCHITECTURE aReg OF Reg64bits IS
BEGIN
  PROCESS(clk) BEGIN
    IF clk'event and clk='1' THEN
      IF raz='1' THEN
        Q<=(OTHERS=>'0');
      ELSE
        Q <= D;
      END IF;
    END IF;
  END PROCESS;
END aReg;
```

IV.9.2- Code VHDL Counter 64 bit

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

entity counter64bits is
   port(
         Clock: in std_logic;
         Reset: in std_logic;
         Q: inout std_logic_vector(0 to 63));
end counter64bits;

architecture Behavioral of counter64bits is
begin
process(Clock,Reset)
   begin
      if Reset='1' then
         Q <= conv_std_logic_vector(0, 64);
      elsif(rising_edge(Clock)) then
         Q <=Q+1;
      end if;
   end process;
end Behavioral;
```

IV.9.3- code VHDL pour UAL

IV.9.3-1 Table de vérité de l'UAL

F : Fonction de type 6 bits

A : Accumulateur de type 64 bits

B : Registre de données type 64 bits

				F1=1	F1=0	
F5	**F4**	**F3**	**F2**	**Opération logique**	**F0=0**	**F0=1**
0	0	0	0	A = non A	A = A	A = A + 1
0	0	0	1	A = non (A ou B)	A = A ou B	A = (A ou B) + 1
0	0	1	0	A = (non A) et B	A= A ou (non B)	A = (A ou (non B)) + 1
0	0	1	1	A = 0	A = - 1	A = 0
0	1	0	0	A = non (A et B)	A = A + (A et (non B))	A = A + (A et (non B)) + 1
0	1	0	1	A = non B	A = (A ou B) + (A et (non B))	A = (A ou B) + (A et(non B)) + 1
0	1	1	0	A = A xor B	A = A - B - 1	A = A - B
0	1	1	1	A = A et (non B)	A = (A et (non B)) - 1	A = A et (non B)
1	0	0	0	A = (non A) ou B	A = A + (A et B)	A = (A + (A et B)) + 1
1	0	0	1	A = non (A xor B)	A = A + B	A = A + B + 1
1	0	1	0	A = B	A = (A ou (non B)) + (A et B)	A = A ou (non B) + (A et B) + 1
1	0	1	1	A = A et B	A= (A et B) - 1	A = A et B

1	1	0	0	A = 1	A = A + (A << 1)	A = A + A + 1
1	1	0	1	A = A ou (non B)	A = (A ou B) + A	A = (A ou B) + A + 1
1	1	1	0	A = A ou B	A = (A ou (non B)) + A	A = A (non B) plus A plus 1
1	1	1	1	A = A	A = A - 1	A = A

IV.9.3-2 Le Code VHDL

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


entity UAL is
port(
   clk :in std_logic;
   A,B: inout std_logic_vector(63 downto 0);
   F: out std_logic_vector(5 downto 0)
 );
end UAL;

architecture ARCH of UAL_VHDL is
begin

process (Clk)

begin
 if Clk'event and Clk = '1' then
        case Inst is
                when "00001X" =>   A<=not(A);
                when "00011X" =>   A<=not(A or B);
                when "00001X" =>   A<=not(A) and B;
                when "00011X" =>   A<=not(A);
                when "00001X" =>   A<=0;
                when "00011X" =>   A<=not(A and B);
                when "00001X" =>   A<=not(B);
                when "00011X" =>   A<=not(A xor B);
                when "00101X" =>   A<=not(A) and B);
                when "00111X" =>   A<=not(A xor B);
                when "00101X" =>   A<=B;
                when "00111X" =>   A<=A and B;
                when "00101X" =>   A<=1;
                when "00111X" =>   A<=A or not(B);
                when "00101X" =>   A<=not(A) and B);
                when "00111X" =>   A<=not(A xor B);
                when "00101X" =>   A<=A or B;
                when "00111X" =>   A<=A;
                when "000000" =>   A<=A;
                when "000100" =>   A<=A or B;
                when "000000" =>   A<=A or not(B);
                when "000100" =>   A<=-1;
                when "000000" =>   A<=0;
```

```
                when "000100" =>   A<=A+(A and not(B));
                when "000000" =>   A<=(A or B) + (A and (not (B))
                when "000100" =>   A<=A - B - 1;
                when "001000" =>   A<=(A and (not B)) - 1;
                when "001100" =>   A<=A + (A and B);
                when "001000" =>   A<=A + B;
                when "001100" =>   A<= (A or (not B)) + (A and B);
                when "001000" =>   A<=(A and B) - 1;
                when "001100" =>   A<=A + (A << 1);
                when "001000" =>   A<= (A or B) + A;
                when "001100" =>   A<= (A or (not B)) + A;
                when "001000" =>   A<=A - 1;
                when "001100" =>   A<=A;
                when "000001" =>   A<=A + 1;
                when "000101" =>   A<=(A or B) + 1;
                when "000001" =>   A<=(A or not(B)) + 1;
                when "000101" =>   A<=0;
                when "000001" =>   A<= A + (A et (non B)) + 1;
                when "000101" =>   A<=(A and B) + (A and(not B)) + 1;
                when "000001" =>   A<=(A - B);
                when "000101" =>   A<= A and (not B);
                when "001001" =>   A<= (A + (A and B)) + 1;
                when "001101" =>   A<= A + B + 1;
                when "001001" =>   A<=A and not(B) + (A and B) + 1;
                when "001101" =>   A<= A and B;
                when "001001" =>   A<=A + A + 1;
                when "001101" =>   A<=(A and B) + A + 1;
                when "001001" =>   A<= (A or B) + A + 1;
                when "001101" =>   A<=A;
                when others   =>   romout := "--------------";
        end case;

  Insto<=Inst;

 end if;

end process;


end ARCH;
```

IV.9.4- CODE VHDL RAM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.all;

entity Data_Memory_VHDL is
port (
 clk: in std_logic;
 mem_access_addr: in std_logic_Vector(63 downto 0);
 mem_write_data: in std_logic_Vector(63 downto 0);
 mem_write_en,mem_read:in std_logic;
 mem_read_data: out std_logic_Vector(63 downto 0)
);
end Data_Memory_VHDL;

architecture Behavioral of Data_Memory_VHDL is
signal i: integer;
signal ram_addr: std_logic_vector(63 downto 0);
type data_mem is array (0 to 4095 ) of std_logic_vector (63 downto 0);
signal RAM: data_mem :=((others=> (others=>'0')));
begin

 ram_addr <= mem_access_addr(64 downto 1);
 process(clk)
 begin
  if(rising_edge(clk)) then
  if (mem_write_en='1') then
  ram(to_integer(unsigned(ram_addr))) <= mem_write_data;
  end if;
  end if;
 end process;
   mem_read_data <= ram(to_integer(unsigned(ram_addr))) when (mem_read='1') else
x"0000";

end Behavioral;
```

IV.9.5- Code VHDL ROM

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

entity ROM is
  port ( address : in std_logic_vector(63 downto 0);
         data : out std_logic_vector(63 downto 0) );
end entity ROM;

architecture behavioral of ROM is
  type mem is array ( 0 to 4095) of std_logic_vector(63 downto 0);
  constant my_Rom : mem := (
    0  => "0000000000000000000000000000000000000000000000000000000000000000",
    1  => "0000000000000000000000000000000000000000000000000000000000000001",
    2  => "0000000000000000000000000000000000000000000000000000000000000010",
    3  => "0000000000000000000000000000000000000000000000000000000000000011",
    4  => "0000000000000000000000000000000000000000000000000000000000000100",
    5  => "0000000000000000000000000000000000000000000000000000000000000101",
   ........
   ..........
   ...........

   16  => "0000000000000000000000000000000000000000000000000000000000010000",
   17  => "0000000000000000000000000000000000000000000000000000000000010001",
   18  => "0000000000000000000000000000000000000000000000000000000000010010",
   .........
   ............
   60  => "0000000000000000000000000000000000000000000000000000000000111100",
   61  => "0000000000000000000000000000000000000000000000000000000000111101",
   62  => "0000000000000000000000000000000000000000000000000000000000111110",
   63  => "0000000000000000000000000000000000000000000000000000000000111111",
   64  => "1111111111111111111111111111111100000000000000000000000000000000",
   65  => "1111111111111111111111111111111100000000000000000000000000000000",
   ...........
   ........
  4093 => "1111111111111111111111111111111100000000000000000000000000000000",
  4094 => "1111111111111111111111111111111100000000000000000000000000000000",
  4095 => "1111111111111111111111111111111100000000000000000000000000000000"
);
begin
   process (address)
   begin
     case address is
       when "0000000000000000" => data <= my_rom(0);
       when "0000000000000001" => data <= my_rom(1);
       ........
       when "000000000000000F" => data <= my_rom(15);
       when "0000000000000010" => data <= my_rom(16);
       ........
       ........
       when "0000000000000FFE" => data <= my_rom(4094);
       when "0000000000000FFF" => data <= my_rom(4095);
       when others => data <=
"0000000000000000000000000000000000000000000000000000000000000000";
        end case;
  end process;
end architecture behavioral;
```

IV.9.6- Code VHDL pour UC

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity control_unit_VHDL is
port (
  opcode: in std_logic_vector(2 downto 0);
  reset: in std_logic;
  reg_dst,mem_to_reg,alu_op: out std_logic_vector(1 downto 0);
  jump,branch,mem_read,mem_write,alu_src,reg_write,sign_or_zero: out std_logic
 );
end control_unit_VHDL;

architecture Behavioral of control_unit_VHDL is

begin
process(reset,opcode)
begin
 if(reset = '1') then
   reg_dst <= "00";
   mem_to_reg <= "00";
   alu_op <= "00";
   jump <= '0';
   branch <= '0';
   mem_read <= '0';
   mem_write <= '0';
   alu_src <= '0';
   reg_write <= '0';
   sign_or_zero <= '1';
 else
 case opcode is
  when "000" => -- add
    reg_dst <= "01";
    mem_to_reg <= "00";
    alu_op <= "00";
    jump <= '0';
    branch <= '0';
    mem_read <= '0';
    mem_write <= '0';
    alu_src <= '0';
    reg_write <= '1';
    sign_or_zero <= '1';
  when "001" => -- sliu
   reg_dst <= "00";
   mem_to_reg <= "00";
   alu_op <= "10";
   jump <= '0';
   branch <= '0';
   mem_read <= '0';
   mem_write <= '0';
   alu_src <= '1';
   reg_write <= '1';
   sign_or_zero <= '0';
  when "010" => -- j
   reg_dst <= "00";
   mem_to_reg <= "00";
   alu_op <= "00";
   jump <= '1';
   branch <= '0';
   mem_read <= '0';
   mem_write <= '0';
   alu_src <= '0';
```

```
   reg_write <= '0';
   sign_or_zero <= '1';
 when "011" =>-- jal
   reg_dst <= "10";
   mem_to_reg <= "10";
   alu_op <= "00";
   jump <= '1';
   branch <=  '0';
   mem_read <=  '0';
   mem_write <=  '0';
   alu_src <= '0';
   reg_write <=  '1';
   sign_or_zero <= '1';
 when "100" =>-- lw
   reg_dst <= "00";
   mem_to_reg <= "01";
   alu_op <= "11";
   jump <= '0';
   branch <= '0';
   mem_read <= '1';
   mem_write <= '0';
   alu_src <= '1';
   reg_write <= '1';
   sign_or_zero <= '1';
 when "101" => -- sw
   reg_dst <= "00";
   mem_to_reg <= "00";
   alu_op <= "11";
   jump <= '0';
   branch <= '0';
   mem_read <= '0';
   mem_write <= '1';
   alu_src <= '1';
   reg_write <= '0';
   sign_or_zero <= '1';
 when "110" => -- beq
   reg_dst <= "00";
   mem_to_reg <= "00";
   alu_op <= "01";
   jump <= '0';
   branch <= '1';
   mem_read <= '0';
   mem_write <= '0';
   alu_src <= '0';
   reg_write <= '0';
   sign_or_zero <= '1';
 when "111" =>-- addi
   reg_dst <= "00";
   mem_to_reg <= "00";
   alu_op <= "11";
   jump <= '0';
   branch <= '0';
   mem_read <= '0';
   mem_write <= '0';
   alu_src <= '1';
   reg_write <= '1';
   sign_or_zero <= '1';
 when others =>
    reg_dst <= "01";
    mem_to_reg <= "00";
    alu_op <= "00";
    jump <= '0';
    branch <= '0';
```

```
    mem_read <= '0';
    mem_write <= '0';
    alu_src <= '0';
    reg_write <= '1';
    sign_or_zero <= '1';
 end case;
 end if;
end process;

end Behavioral;
```

IV.10- Conclusion:

Dans ce chapitre, on a étudié le microcontrôleur, on a vu sa présentation, sa structure, la différence entre le microcontrôleur et le microprocesseur et ses différentes architectures et enfin les éléments de choix d'un microcontrôleur, puis on a introduits les codes VHDL du quelques fonctions de ses composants.

CONCLUSION GENERALE

Conclusion Générale

L'objectif de ce travail consistait en la conception et l'implémentation d'un microcontrôleur de 64 bits sur un circuit logique programmable FPGA. Pour aboutir à cet objectif qui s'est basé sur la transposition de l'architecture et fonctions des composants internes du microcontrôleur entre autre le microprocesseur comprenant l'unité arithmétique et logique UAL, les registre, l'unité de contrôle et les mémoires RAM et ROM sur FPGA du type cyclone II de Altéra, on a utilisé le langage de description de matériel VHDL qu'on a développé au premier chapitre en présentant ses avantages et ses fonctionnalités, on a démontré que c'est un langage complet destiné à représenter le comportement ainsi que l'architecture des systèmes numériques.

On a étudié aussi les circuits logiques programmable FPGA, on a montré les avantages que présentent de point de vue flexibilité, configuration et , reprogrammation à volonté qui permet la modification et l'amélioration des conceptions réalisées. On a abordé aussi une brève étude sur le microcontrôleur et sa structure, ensuite, on a réalisé les descriptions en VHDL de quelques fonctions des éléments du microcontrôleurs de 64 bits: UAL, UC, Registre, RAM, ROM en utilisant le logiciel Quartus II pour leurs implémentations sur FPGA. Les résultats obtenus nous ont permis de démontrer le bon fonctionnement des circuits logiques programmable FPGA, et de saisir l'importance de leur utilisation.

Comme perspective et travaux futurs, on propose de réaliser d'autres implémentations des fonctionnalités d'autres systèmes numériques sur FPGA.

BIBLIOGRAPHIE

Bibliographie

[1] Debyo SAPTONO **Conception d'un outil de prototypage rapide sur le FPGA pour des applications de traitement d'images** Thèse de doctorat de l'université de Bourgogne le 04 novembre 2011.

[2] https://fr.wikipedia.org/wiki/VHDL (site consulté le 27/01/2019)

[3] Jacques WEBER et Maurice MEAUDRE. **VHDL du langage au circuit du circuit au langage**.1997

[4] Philippe LARCHER .VHDL **Introduction à la synthèse logique**. 2000

[5] Eduardo SANCHEZ EPFL/PDF. **Le langage VHDL**.

[6] Anissa NASRI. **Conception et implémentation d'un PIC 16F48 sur FPGA** Thèse master université de Biskra 2018.

[7] Tangui RISSET. https//: comelec.enst.fr/hdl. **Introduction à VHDL** (site consulté le 27/01/2019)

[8] SUPPORT DE COURS DE Mr DHIABI Fathi. université de Biskra.

[9] Laurent RODRIGUEZ et Benoit MIRAMOND **Cours VHDL** / PDF université de Cergy Pontoise. adresse url: https://docplayer.fr/5927659-Cours-vhdl-iv-l3-s6-universite-de-cergy-pontoise-laurent-rodriguez-benoit-miramond.html (site consulté le 27/01/2019)

[10] https://fr.wikibooks.org/wiki/Conception_et_VHDL (site consulté le 27/01/2019)

[11] Laurence PIERRE. **Modélisation des systèmes numériques - VHDL .**Université Grenoble Alpes adresse url : http://users-tima.imag.fr/sls/lpierre/Teaching/MSN (consulté le 16/02/2019)

[12] C.ALEXANDRE. **Introduction à la conception numérique en VHDL.** pdf 2018 (site consulté le 25/02/2019)

[13] http://hdl.telecom-paristech.fr/vhdl_structurel.html. COURS EN LIGNE VHDL (site consulté le 16/02/2019)

[14] Jaques WEBER et Maurice MEAUDRE. **circuits numériques et synthèse logique un outil VHDL**

[15] http://proxacutor.free.fr/index.htm(site consulté le 25/03/2019)

[16] https://www.cder.dz/vlib/bulletin/pdf/bulletin_024_05.pdf Les circuits FPGA : description et applications GUELLAL Amar Attaché de recherche (site consulté le 25/03/2019).

[17] https://en.wikibooks.org/wiki/Programmable_Logic/FPGAs consulté le 16/02/2019

[18] https://www.memoireonline.com/04/08/1026/m_fpga-and-traffic-network-analysis7.html consulté le 20/03/2019)

[19] https://en.wikipedia.org/wiki/Field-programmable_gate_array#Applications (consulté le 20/03/2019

[20] https://allaboutfpga.com/fpga-configuration-tutorial/ consulté le 25/03/2019

[21] Conception d'un outil de prototypage rapide sur le FPGA pour des applications de traitement d'images Thèse de doctorat **Debyo SAPTONO Universite De Bourgogne NOV. 2011**

[22] https://www.memoireonline.com/10/12/6158/m_Mise-en-oeuvre-de-lauto-reconfiguration-partielle-et-dynamique-sur-FPGA-Xilinx-Virtex-II-pro11.html (consulté le 25/03/2019)

[23] **Notice de prise en main de Quartus II** https://docplayer.fr/5263124-Notice-de-prise-en-main-du-logiciel-quartus-ii.html. (consulté le 13/04/2019à

[24] https://fr.wikipedia.org/wiki/Microcontrôleur (consulté le 17/02/2019)

[25] https://www.electronique-mixte.fr/Cours-Microcontrôleur-microprocesseur (site consulté le 27/03/2019)

[26] https://www.lsis.org/master/documents/micro/ cours d'informatique industrielle (site consulté le 25/03/2019)