RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITÉ MOULOUD MAMMERI DE TIZI-OUZOU

**Faculté de Génie Électrique et d'Informatique**
**Département d'Automatique**

# Mémoire de fin d'études de Master Académique

Spécialité : AUTOMATIQUE ET INFORMATIQUE INDUSTRIELLE

Présenté par :

**Hayet ZIDANI**
*et*
**Kahina TEKLAL**

## Thème
## Réalisation d'un microprocesseur 8 bits sur une carte FPGA

Mémoire soutenu publiquement le 10 Juillet 2018, devant le jury :

| | | |
|---|---|---|
| **Président** | **M. Prénom NOM** | *Professeur, Université de Tizi-Ouzou* |
| **Encadreur** | **M. Redouane KARA** | *Professeur, Université de Tizi-Ouzou* |
| **Examinateurs** | **M. Prénom NOM** | *Professeur, Université de Tizi-Ouzou* |
| | **M. Prénom NOM** | *Professeur, Université de Tizi-Ouzou* |

# Dédicaces

Avec un énorme plaisir, un cœur ouvert et une immense joie, que je dédie mon travail à mes très chers, respectueux et magnifiques parents, pour tous leurs sacrifices, leur amour, leur tendresse, leur soutien et leurs prières tout au long de mes études.

A mes frères, ma sœur et ma belle-sœur, pour leurs appuis et leur encouragement.

A tous mes amis et toute ma promotion, et en particulier mon binôme KAHINA.

A tous ce qui de loin ou de près n'ont pas cessé de m'apporter leur soutien, pendant mes années d'études.

Que ce travail soit l'accomplissement de vos vœux tant allégués, et le fruit de votre soutien infaillible.

HAYET

# *Dédicaces*

*Avec un énorme plaisir, un cœur ouvert et une immense joie, que je dédie mon travail à mes très chers, respectueux et magnifiques parents, pour tous leurs sacrifices, leur amour, leur tendresse, leur soutien et leurs prières tout au long de mes études.*

*A mon frère et mes sœurs, pour leurs appuis et leur encouragement.*

*A tous mes amis et toute ma promotion, et en particulier mon binôme HAYET.*

*A tous ce qui de loin ou de près n'ont pas cessé de m'apporter leur soutien, pendant mes années d'études.*

*Que ce travail soit l'accomplissement de vos vœux tant allégués, et le fruit de votre soutien infaillible.*

*KAHINA*

# Remerciements

Nous remercions tout d'abord ALLAH le tout puissant pour nous avoir donné la force, la santé, et la patience pour arriver là où nous sommes.

Nous tenons également, à exprimer nos forts remerciements à notre encadreur, Monsieur KARA, pour son accueil, ses conseils et son aide, et qui a mis à notre disposition tous les nécessaires pour réaliser ce travail.

Nous remercions les membres de jurys qui ont bien voulu accepter d'examiner ce mémoire et à cet égards nous leurs présentons nos profonds respects, et notre grande gratitude.

Nous rendons grâce à tous les enseignants du département Automatique, qui nous ont suivis durant notre cursus, et tous ceux qui ont contribué, à ce que ce travail prenne forme.

Nous exprimons aussi nos remerciements à tout le personnel du laboratoire L2CSP.

Nous tenons à remercier : nos parents et tous nos amis, pour leur pensé et leur soutien tout le temps de travail.

# Table des matières

**Table des figures** VI

**Table des tableaux** VII

**Abréviations** VIII

**Introduction générale** 1

**1 Généralités sur les circuits logiques programmables** 2

1.1 Introduction . . . . . 2
1.2 Les technologies de mémorisation . . . . . 2
1.3 Présentation de la Logique Programmée . . . . . 4
1.3.1 Liste des Circuits Logiques Programmables . . . . . 4
1.3.1.1 Les PALs . . . . . 4
1.3.1.2 Les GALs . . . . . 5
1.3.1.3 Les EPLDs . . . . . 5
1.3.1.4 Les CPLDs . . . . . 6
1.3.1.5 Les FPGAs . . . . . 6
1.4 Présentation des FPGAs . . . . . 7
1.5 Architecture interne d'un FPGA . . . . . 7
1.5.1 Interfaces d'entrées/sorties (IOB) . . . . . 8
1.5.2 Les blocs logiques de base (CLB) . . . . . 9
1.5.3 Les Interconnexions . . . . . 10
1.6 Classification des FPGAs . . . . . 12
1.7 Différents domaines d'application des FPGAs . . . . . 13
1.8 Les outils de développement des FPGAs . . . . . 13
1.9 FPGA, points faibles-points forts . . . . . 13
1.10 Conclusion . . . . . 14

**2 Le langage de description VHDL** 15

2.1 Introduction . . . . . 15
2.1.1 Définition du langage VHDL . . . . . 15

2.1.2 Structure d'un programme VHDL . . . 15
2.1.2.1 Déclaration des bibliothèques . . . 16
2.1.2.2 Déclaration de l'entité . . . 16
2.1.2.3 Déclaration de l'architecture . . . 18
2.1.3 Les différents styles de descriptions d'une architecture . . . 20
2.1.3.1 Description de type flot de données . . . 20
2.1.3.2 Description comportementale . . . 20
2.1.3.3 Description structurelle . . . 23
2.1.4 Les éléments du langage VHDL . . . 23
2.1.4.1 Les constantes : . . . 23
2.1.4.2 Les signaux : . . . 24
2.1.4.3 Les variables : . . . 24
2.1.5 Les opérations de base sur les objets . . . 24
2.1.5.1 Les opérations logiques : . . . 24
2.1.5.2 Les opérations relationnelles : . . . 25
2.1.5.3 Les opérations d'addition : . . . 25
2.1.5.4 Les opérations de signe : . . . 25
2.1.5.5 Les opérations de multiplication : . . . 25
2.1.5.6 Les opérations NOT, ABS et ** : . . . 25
2.1.6 Règles d'écriture du langage VHDL . . . 25
2.1.7 Les avantage du langage VHDL . . . 26
2.2 Machine à état . . . 26
2.2.1 Machine de Moore . . . 27
2.2.2 Machine de Mealy . . . 28
2.3 Conclusion . . . 29

**3 Conception d'un microprocesseur 30**
3.1 Introduction . . . 30
3.2 Définition du microprocesseur . . . 30
3.3 Architecture de base d'un microprocesseur . . . 30
3.3.1 Unité de commande . . . 31
3.3.2 Unité de traitement . . . 32
3.3.3 Les bus . . . 32
3.4 Jeu d'instructions . . . 32
3.5 Notion d'architecture RISC et CISC . . . 33
3.5.1 Architecture CISC . . . 33
3.5.2 Architecture RISC . . . 33
3.5.3 Comparaison entre les deux architectures . . . 34
3.6 Architecture de Von Neumann . . . 34

3.7 Architecture de Harvard . . . . . 35
3.8 Conception d'un microprocesseur étape par étape . . . . . 35
3.8.1 Microprocesseur linéaire basique (Version 1) . . . . . 36
3.8.2 Microprocesseur avec accumulateur (Version 2) . . . . . 39
3.8.3 Microprocesseur avec accumulateur et indirection (Version 3) . . . . 42
3.9 Conclusion . . . . . 45

**4 Réalisation et résultats expérimentaux 46**
4.1 Introduction . . . . . 46
4.2 Présentation de la carte DE2 . . . . . 46
4.2.1 Les périphériques de la carte DE2 . . . . . 47
4.2.1.1 Boutons poussoirs . . . . . 47
4.2.1.2 Interrupteurs . . . . . 48
4.2.1.3 Led . . . . . 48
4.2.1.4 Afficheurs 7-segments . . . . . 48
4.2.2 Utilisation de la carte DE2 . . . . . 49
4.3 Présentation du logiciel de programmation « Quartus II » . . . . . 49
4.3.1 Création d'un projet . . . . . 49
4.3.2 Saisie d'un projet . . . . . 52
4.3.3 Simulation . . . . . 53
4.3.4 Programmation d'un circuit . . . . . 54
4.3.5 Configurer le composant au programme . . . . . 55
4.4 Test et validation des résultats . . . . . 56
4.4.1 Première version . . . . . 56
4.4.1.1 Résultats de simulation . . . . . 56
4.4.1.2 Résultats pratiques . . . . . 58
4.4.2 Deuxième version . . . . . 59
4.4.2.1 Résultats de simulation . . . . . 59
4.4.2.2 Résultats pratiques . . . . . 61
4.4.3 Troisième version . . . . . 62
4.4.3.1 Résultats de simulation . . . . . 63
4.4.3.2 Résultats pratiques . . . . . 65
4.5 Conclusion . . . . . 66

**Conclusion générale 67**

**Bibliographie 69**

**Annexes 72**

# Table des figures


1.1 Les différents types de mémoires. . . . . 3
1.2 Structure générale d'un PAL. . . . . 5
1.3 Physionomie des CPLDs. . . . . 6
1.4 Résumé graphique des familles de PLD. . . . . 7
1.5 Architecture interne d'un FPGA. . . . . 8
1.6 Bloc d'entrées/sorties. . . . . 9
1.7 Architecture interne d'un CLB. . . . . 10
1.8 Structure générale des interconnexions. . . . . 11
1.9 Structure générale d'un FPGA. . . . . 12

2.1 Déclaration d'une bibliothèque. . . . . 16
2.2 Déclaration d'une entité. . . . . 17
2.3 Le mode d'un port. . . . . 17
2.4 Déclaration d'une architecture. . . . . 19
2.5 Structure d'un programme VHDL . . . . . 19
2.6 Description de type flot de données d'un demi additionneur. . . . . 20
2.7 Schéma d'exécution des PROCESS. . . . . 21
2.8 Description avec l'instruction IF. . . . . 22
2.9 Description avec l'instruction CASE. . . . . 23
2.10 Machine d'état de Moore. . . . . 27
2.11 Exemple d'une machine de Moore. . . . . 27
2.12 Machine d'état de Mealy. . . . . 28
2.13 Exemple d'une machine de Mealy. . . . . 29

3.1 Éléments principaux constituant un microprocesseur. . . . . 31
3.2 Architecture de Von Neumann. . . . . 34
3.3 Architecture de Harvard. . . . . 35
3.4 Schéma synoptique du microprocesseur à réaliser. . . . . 35
3.5 Graphe d'états de la première version. . . . . 38
3.6 Architecture du microprocesseur basique. . . . . 38
3.7 Graphe d'états de la deuxième version. . . . . 41

3.8 Architecture du microprocesseur avec accumulateur. . . . . . . . . . . . . . . 42
3.9 Graphe d'état de la troisième version. . . . . . . . . . . . . . . . . . . . . . . . . 44
3.10 Architecture de microprocesseur avec un accumulateur et indirection. . . . 45

4.1 Aperçu de la carte DE2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
4.2 Boutons poussoirs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.3 Interrupteurs (les Switchs). . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.4 LEDs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.5 Afficheurs 7-segments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.6 Présentation de l'environnement de conception Quartus II. . . . . . . . . . . 50
4.7 La fenêtre qui permet la configuration du projet. . . . . . . . . . . . . . . . . 51
4.8 La fenêtre apparait après la validation des choix . . . . . . . . . . . . . . . . 52
4.9 Fenêtre de choix de la saisie . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
4.10 Fenêtre de programmation. . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
4.11 Fenêtre de Hardware Setup. . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
4.12 Résultats de simulation de la première version. . . . . . . . . . . . . . . . . . 57
4.13 Codes des instructions (version 1) . . . . . . . . . . . . . . . . . . . . . . . . 58
4.14 Résultats pratique de la première version. . . . . . . . . . . . . . . . . . . . 58
4.15 Résultats de simulation de la deuxième version. . . . . . . . . . . . . . . . . 60
4.16 Codes des instructions (version 2) . . . . . . . . . . . . . . . . . . . . . . . . 61
4.17 Résultats pratique de la deuxième version (première opération). . . . . . . 61
4.18 Résultats pratique de la deuxième version (deuxième opération). . . . . . . 62
4.19 Résultats de simulation de la troisième version. . . . . . . . . . . . . . . . . 64
4.20 Résultats pratique de la troisième version (première opération). . . . . . . 65
4.21 Résultats pratique de la troisième version (deuxième opération). . . . . . . 66

# Liste des tableaux

1.1 Tableau récapitulatif des différentes familles des PLDs. . . . . . . . . . . . 4
1.2 Avantages et inconvénients des FPGAs. . . . . . . . . . . . . . . . . . . . . . . . 14

2.1 Liste des états du type STD_LOGIC. . . . . . . . . . . . . . . . . . . . . . . 18
2.2 Comparaison entre variable et signal. . . . . . . . . . . . . . . . . . . . . . . 24

3.1 Comparaison entre l'architecture RISC et CISC. . . . . . . . . . . . . . . . . 34
3.2 Organisation de la mémoire du microprocesseur basique. . . . . . . . . . . 36
3.3 Code d'instruction du microprocesseur basique. . . . . . . . . . . . . . . . . 36
3.4 Exemple d'organisation de la mémoire du microprocesseur avec accumulateur. 39
3.5 Code d'instruction du microprocesseur avec accumulateur. . . . . . . . . . 40
3.6 Code d'instruction du microprocesseur avec accumulateur et indirection. . 43

4.1 Brochage des boutons poussoirs . . . . . . . . . . . . . . . . . . . . . . . . . 76
4.2 Brochage des afficheurs 7-segments . . . . . . . . . . . . . . . . . . . . . . . 77
4.3 Brochage des afficheurs 7-segments . . . . . . . . . . . . . . . . . . . . . . . 78
4.4 Mots résérvés du langage VHDL . . . . . . . . . . . . . . . . . . . . . . . . . 78

# Abréviations

- PLD : Programmable Logic Device.
- PAL : Programmable Array Logic.
- FPGA : Field Programmable Gate Array.
- GAL : Generic Array Logic.
- EPLD : Erasable Programmable Logic Device.
- CMOS : Complementary Metal Oxide Semiconductor.
- CPLD : Complex Programmable Logic Device.
- IOB : Input Output Blocks.
- CLB : Configurable Logic blocks.
- LUT : Look Up Table.
- SRAM : Static Random Access Memory.
- ROM : Read Only Memory.
- VHDL : Very High Speed Integrated Circuit Hardware Description Language.
- DOD : United States Department of Defense.
- IEEE : Institute of Electrical and Electronics Engineers.
- CAO : Conception Assistée par Ordinateur.
- RISC : Reduced Instruction Set Computer.
- CISC : Complex Instruction Set Computer.

# Introduction générale

Le processeur est le cerveau de l'ordinateur. Il permet de manipuler des informations numériques, c'est-à-dire les informations codées sous forme binaire, et d'exécuter les instructions stockées en mémoire, couplée au processeur.

La réalisation d'un microprocesseur, est une tâche passionnante qui, à première vue, peut paraitre très compliquée, puisqu'elle nécessite des connaissances dans des domaines très variés, tels que l'informatique et l'électronique.

Notre travail consiste à concevoir un microprocesseur 8 bits, pour cela, nous allons procéder par étapes, et le construire petit à petit. Ce dernier a été synthétisé en utilisant le langage de description matériel "VHDL". La mise en œuvre de ce microprocesseur a été effectuée sur un circuit logique programmable de type "FPGA", pour se faire, on a utilisé la carte DE2 d'Altera et son environnement de programmation "Quartus II".

L'organisation de ce mémoire suit cette structuration :

- Le premier chapitre est dédié à la présentation, d'une manière générale, des circuits logiques programmables, ainsi que les circuits FPGA et leurs architectures internes.
- Le second chapitre est consacré à la présentation du langage de description matériel VHDL utilisé pour la programmation de la carte DE2.
- Le troisième chapitre s'intéresse à la conception du microprocesseur étape par étape, ainsi que leur architecture, et leur jeu d'instructions.
- Le quatrième chapitre est consacré à la présentation de la carte DE2 et ses différents composants. Aussi, nous avons présenté l'outil de développement "Quartus II". Enfin, on termine par les résultats de simulations et expérimentaux de notre projet, ainsi que l'interprétation de chaque résultat.

Enfin, nous terminons ce mémoire par une conclusion générale.

# Chapitre 1

# Généralités sur les circuits logiques programmables

## 1.1 Introduction

Les circuits logiques programmables représentent une solution incontournable dans le domaine de l'électronique numérique. En effet, la possibilité de programmer un composant pour qu'il puisse fonctionner selon les besoins du concepteur est une aide précieuse pour pouvoir élaborer efficacement un circuit complexe. Contrairement aux circuits logiques prédéfinis par les fabricants tels que les portes logiques ou les fonctions logiques qui ont des entrées sorties fixe, une architecture figée, un encombrement étendu, une consommation en énergie élevée réduit l'usage de ces composants pour de petites applications simple et des circuits à densités moyenne. Les PLDs (Programmable Logic Device) offrent la possibilité de programmer l'architecture interne du composant, pour réaliser une fonction souhaitée par l'utilisateur et de configurer les pins d'entrées/sorties de celui-ci. Les PLDs ont connu une évolution technologique au fil du temps depuis l'apparition du premier PAL (Programmable Array Logic ou réseau logique programmable) jusqu'à l'aboutissement aux FPGAs (Field Programmable Gate Array ou réseau de cellules logiques programmables) qui sont les circuits logiques programmables les plus performant qui existent en ce moment.

## 1.2 Les technologies de mémorisation

Voici d'une manière générale l'arborescence des mémoires disponibles à nos jours :

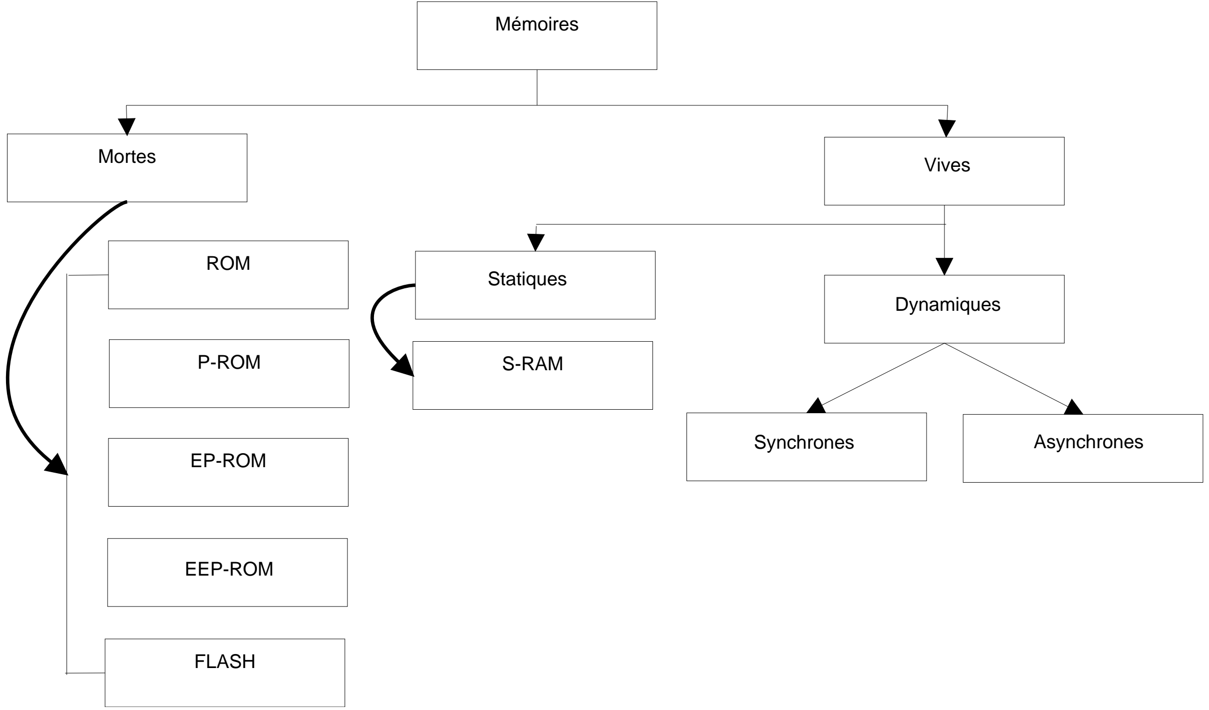



Figure 1.1 – Les différents types de mémoires.

L'ensemble des caractéristiques de ces mémoires sont récapitulés comme suit :

— **ROM (Read Only Memory) :** Mémoire figée par le concepteur à la lecture seule et non modifiable.
— **PROM (Programmable Read Only Memory) :** Mémoire programmable une fois par l'utilisateur avec un équipement spécialisé.
— **EPROM (Erasable Programmable Read Only Memory) :** Mémoire programmable électriquement et effaçable par des rayons Ultra-Violet.
— **EEPROM (Electrically Erasable Programmable Read Only Memory) :** Mémoire programmable électriquement à lecture seule, effaçables électriquement (quelques millisecondes).
— **Mémoires FLASH :** Elle est l'une des versions les plus évoluée des EEPROM avec avantages d'être plus facile à programmer et à effacer.
— **SRAM (Static Random Memory) :** Mémoire volatile avec cellule de base à plusieurs transistors (accès rapide, consommation plus couteuse). La volatilité correspond au non disponibilité de l'information lorsqu'il n'y a pas d'alimentation.
— **RAM dynamiques :** Mémoire volatile qui nécessite rafraichissement périodique de l'information afin de la conserver avec cellule de base à un transistor (densité forte, accès lent).[1]

## 1.3 Présentation de la Logique Programmée

Actuellement les Objets Techniques utilisent de plus en plus la logique programmée. Ces structures ont besoin de s'interfacer entre elles. Pour réaliser ces interfaces, les structures utilisent généralement des fonctions logiques élémentaires, compteurs, registres. Le nombre de circuits nécessaires pour remplir ces fonctions peut devenir très vite important. Pour diminuer les couts de fabrication, de développement et de maintenance, les fabricants de circuits intégrés ont donné naissance aux Circuits Logique Programmable ou encore P.L.D (Programmable Logic Device), qui sont devenus au fil des années, de plus en plus évolués.

### 1.3.1 Liste des Circuits Logiques Programmables

La classification des PLDs est un peu délicate et difficile. Néanmoins, on peut les classer suivant leurs structures internes à savoir ; le nombre d'entrées, de sorties, de connexions programmables et le niveau d'intégration. Le tableau suivant représente certaines de ces familles :

| Type | Nombre de porte intégrée | Matrice ET | Matrice OU | Effaçable |
|---|---|---|---|---|
| PAL | 10 à 100 | Programmable | Fixe | Non |
| GAL | 10 à 100 | Programmable | Fixe | Électriquement |
| EPLD | 100 à 3000 | Programmable | Fixe | U-V |
| FPGA | Plus de 50.000 | Programmable | Programmable | Électriquement |

TABLE 1.1 – Tableau récapitulatif des différentes familles des PLDs.

#### 1.3.1.1 Les PALs

La structure de base des PALs est formée par une matrice "ET" suivie d'une matrice "OU". Les circuits constituants le PAL sont placés sous la forme d'une matrice de type lignes/colonnes (figure1.2). La programmation s'effectue par destruction de fusible. La disposition adoptée pour les circuits permet de réaliser de nombreuses fonctions logiques. A l'achat d'un PAL, tous les fusibles sont vierges.

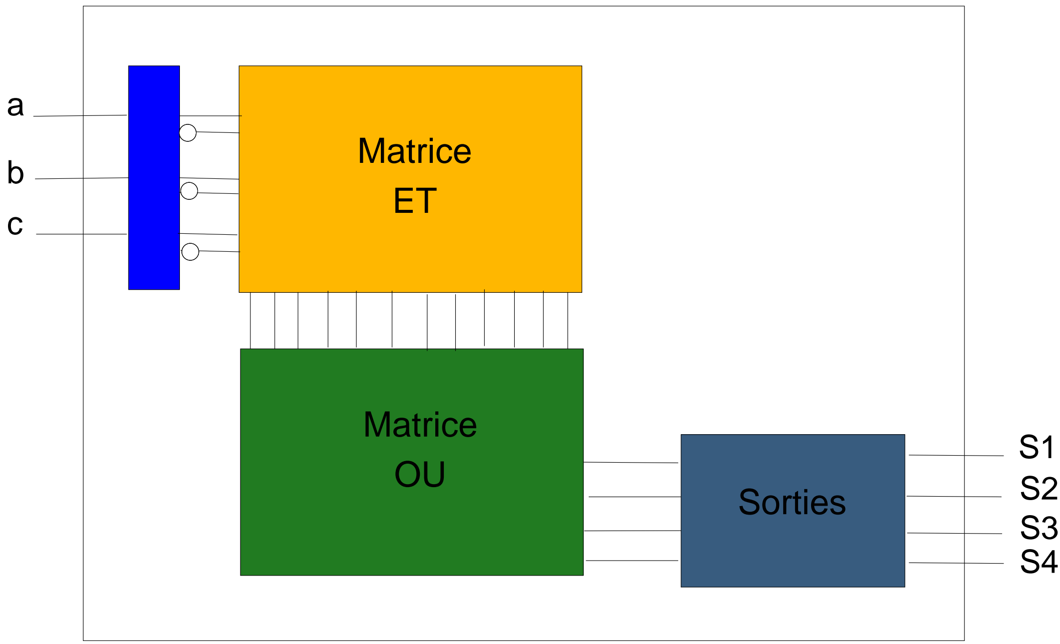



FIGURE 1.2 – Structure générale d'un PAL.

#### 1.3.1.2 Les GALs

Les PALs possèdent un inconvénient majeur qui réside dans l'impossibilité de les programmer au-delà d'une seule fois. Alors, la firme LATTICE a opté pour le remplacement des fusibles irréversibles (fusibles bipolaire) des PALs par des transistors MOSFET pouvant être rgénérer. Ce qui donne naissance au GAL que l'on pourrait traduire par « Réseau Logique Générique ». Donc ces circuits peuvent être reprogrammer sans se soucier de leur durée de vie restreinte.

#### 1.3.1.3 Les EPLDs

Les EPLD sont des circuits programmables électriquement et effaçables, soit par exposition aux UV pour les plus anciens, soit électriquement. Ces circuits, développés en premier par la firme ALTERA, sont arrivés sur le marché en 1985. Les EPLD sont une évolution importante des PAL CMOS. Ils sont basés sur le même principe pour la réalisation des fonctions logiques de base. Les procédés physiques d'intégration permis par les EPLD sont nettement plus importants que ceux autorisés par les PAL CMOS. En effet, les plus gros EPLD actuellement commercialisés intègrent plusieurs dizaines de milliers de portes utilisables par l'utilisateur.

#### 1.3.1.4 Les CPLDs

Les CPLDs sont composés de plusieurs PAL élémentaires reliés entre eux par une zone d'interconnexion (matrice d'interconnexion) (figure 1.3). Grâce à cette architecture, ils permettent d'atteindre des vitesses de fonctionnement élevées (plusieurs centaines de Mhz). La physionomie est généralement très structurée. Un certain nombre de macro-cellules de base (une zone de porte logique et une bascule) regroupées pour former des blocs logiques. La complexité, le nombre de macro-cellules dans un bloc ainsi le nombre de blocs varient d'un composant à un autre.

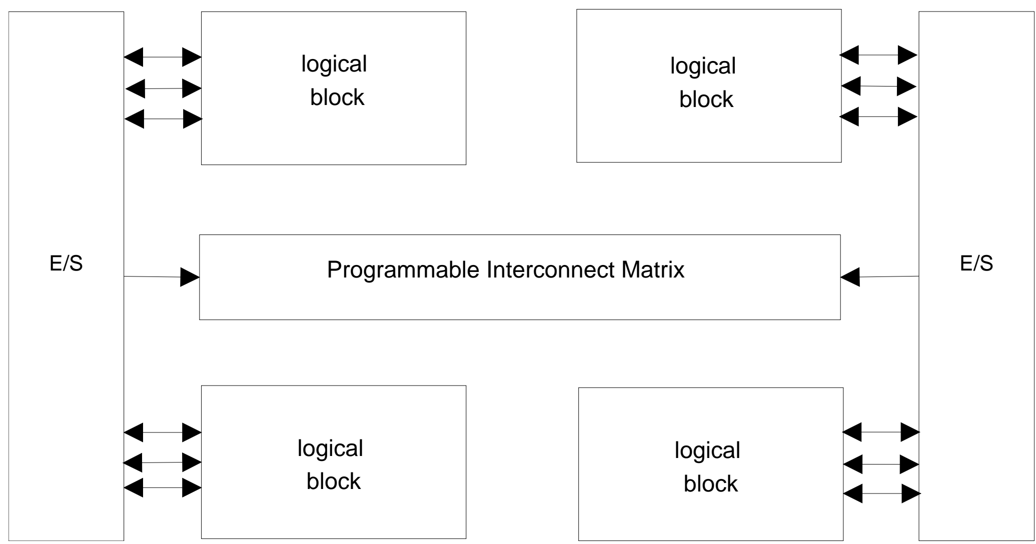



FIGURE 1.3 – Physionomie des CPLDs.

#### 1.3.1.5 Les FPGAs

Un FPGA est un circuit logique reprogrammable et configurable afin de mettre en œuvre des fonctionnalités matérielles personnalisées, sans avoir jamais besoin d'utiliser une maquette ou un fer à souder. Il suffit de développer des tâches de traitement numérique par logiciel et de les compiler sous forme de fichier de configuration.

La figure qui suit résume toutes ces familles :

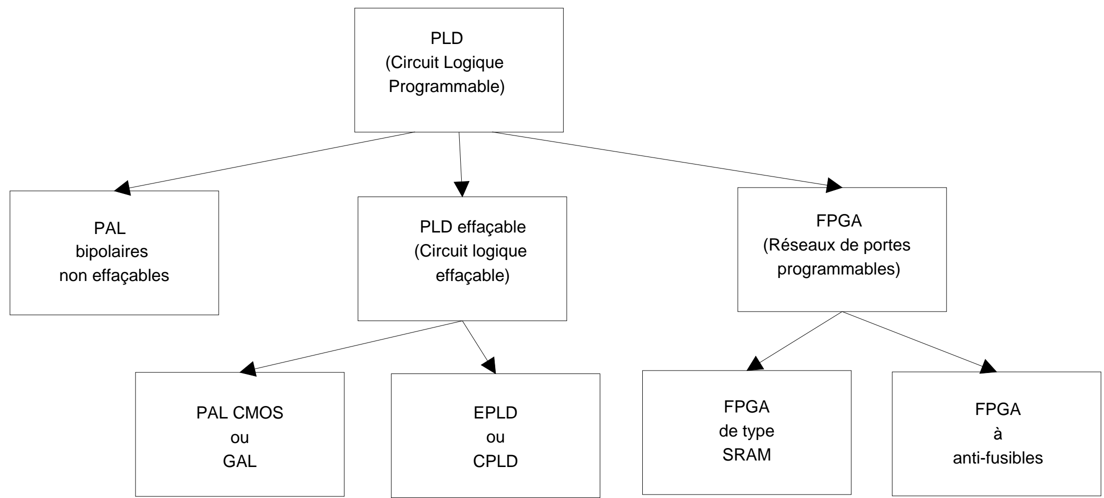



FIGURE 1.4 – Résumé graphique des familles de PLD.

## 1.4 Présentation des FPGAs

Un FPGA constitue une évolution des circuits logiques programmables CPLDs, il se compose d'une matrice de blocs logiques programmables entouré de blocs d'entrées/sorties et l'ensemble est relié par un réseau d'interconnexions programmables. Les blocs logiques peuvent exécuter n'importe quelle fonctionnalité depuis des fonctions logiques de base (comme AND, OR, XOR,...) jusqu'aux fonctions complexes combinatoires. La principale caractéristique des FPGAs est la reconfiguration qui peut être, effectuée même au cours de fonctionnement.

Il y a plusieurs constructeurs de composants FPGA tels que ACTEL, XILINX et ALTERA, ces constructeurs utilisent différentes technologies pour la réalisation des FPGAs.

## 1.5 Architecture interne d'un FPGA

Les circuits FPGAs sont structurés sous forme de matrices d'éléments logiques de base, constitués de porte logique. Les circuits FPGA utilisent deux types de cellules de base, les cellules d'entrées/sorties appelées IOB et les cellules logiques appelés CLB. Ces différentes cellules sont reliées entre elles par un réseau d'interconnexions configurable (figure 1.5). [2]

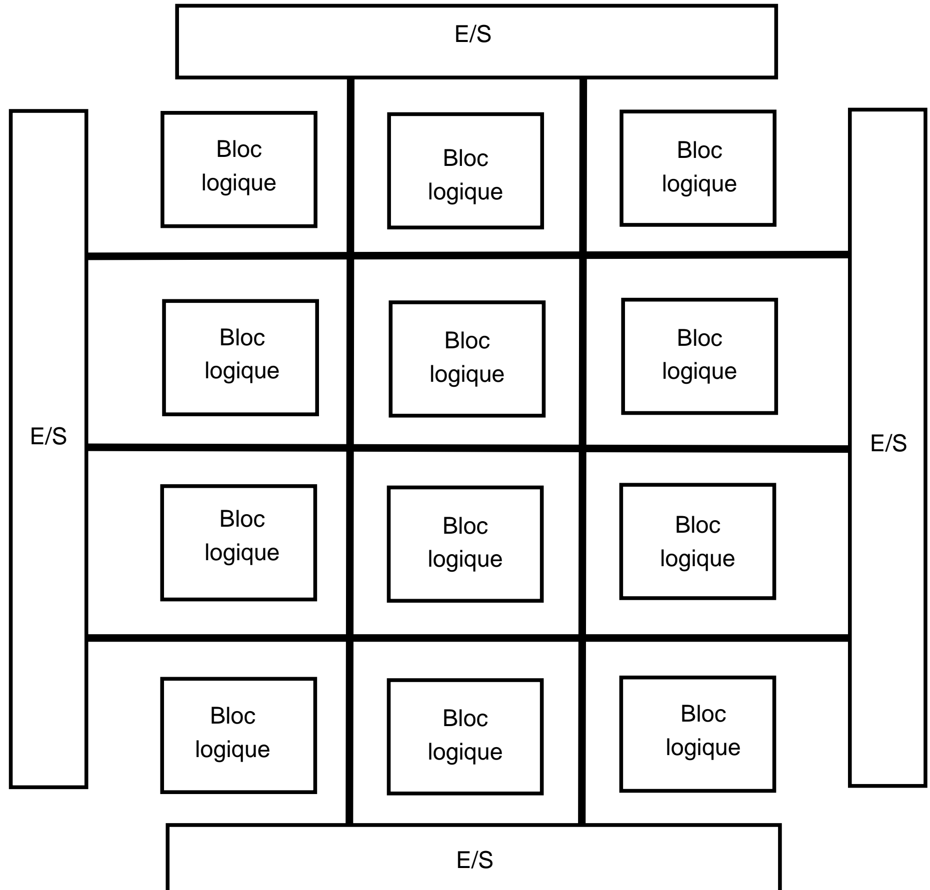



FIGURE 1.5 – Architecture interne d'un FPGA.

### 1.5.1 Interfaces d'entrées/sorties (IOB)

La figure qui suit présente la structure de ce bloc. Ces blocs entrées/sorties permettent l'interface entre les broches du composant FPGA et la logique interne développée à l'intérieur du composant. Ils sont présents sur toute la périphérie du circuit FPGA. Chaque IOB contrôle une broche du composant et il peut être défini en entrée, en sortie, en signaux bidirectionnels ou être inutilisé (haute impédance).

Les blocs d'entrées/sorties comportent une porte 3-états, une bascule D pour la sortie, une autre bascule D pour la synchronisation de l'entrée et des fonctionnalités nouvelles. Ces fonctionnalités sont : l'utilisation des résistances de rappel d'une broche d'entrée/sortie à la tension d'alimentation (pull-up resistor) ou à la masse (pull-down resistor), la programmation du temps de monté d'un signal de sortie.[1]

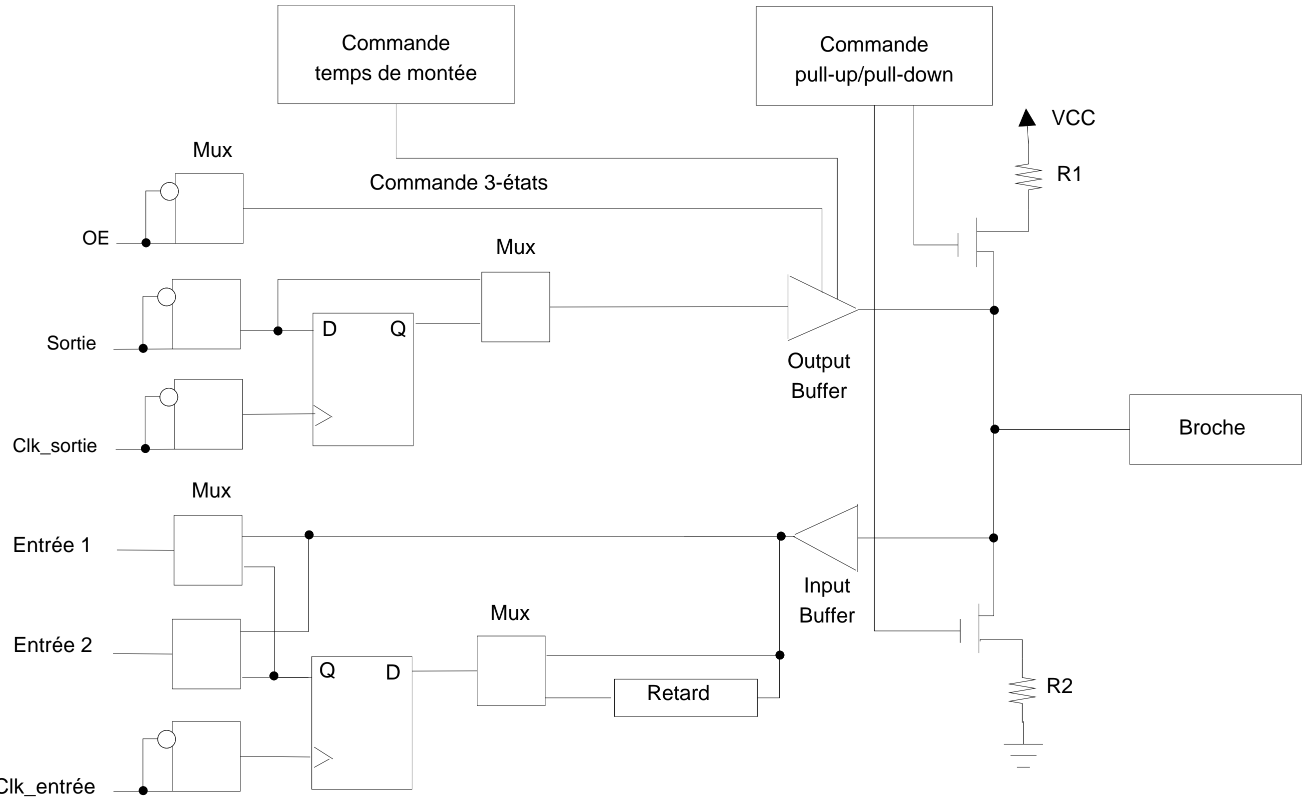



FIGURE 1.6 – Bloc d'entrées/sorties.

## 1.5.2 Les blocs logiques de base (CLB)

Les blocs logiques de base sont les principaux éléments d'un FPGA. Le concept de ces blocs est basé sur l'utilisation des LUTs (Look-Up-Table) comme élément combinatoire de la cellule de base. En première approximation, cette LUT peut être vue comme une mémoire (16 bits en général) qui permet de créer n'importe quelle fonction logique combinatoire de quatre variables d'entrées. La figure suivante représente l'architecture interne d'un CLB.

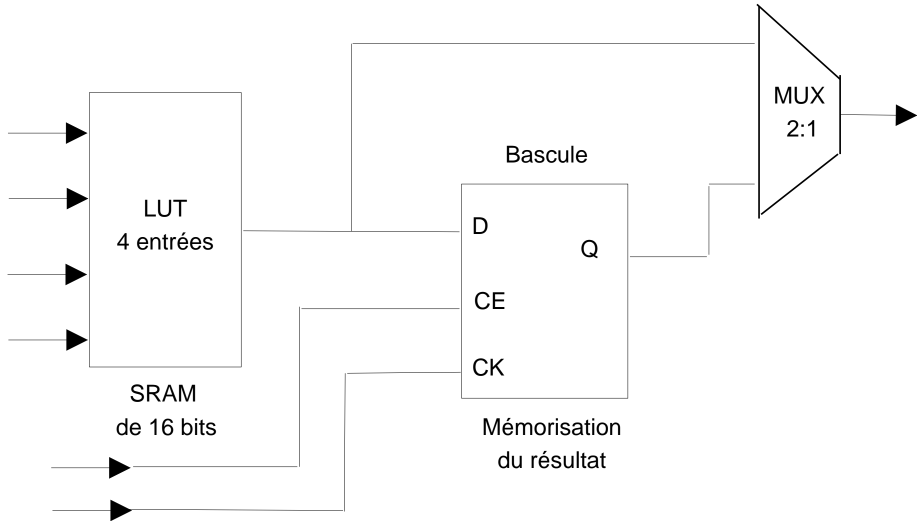



FIGURE 1.7 – Architecture interne d'un CLB.

Elle comprend une LUT de quatre (04) entrées et une bascule D qui permet la réalisation de fonctions logiques séquentielles. La configuration du multiplexeur 2 vers 1 de sortie, autorise la sélection des deux types de fonction, combinatoire ou séquentielle.

### 1.5.3 Les Interconnexions

Les lignes d'interconnexion au sein d'un FPGA, permettent la connexion des CLBs et des IOBs. Les connexions internes dans les circuits FPGAs 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. Le rôle de ces interconnexions est de relier avec un maximum d'efficacité les blocs logiques et les entrées/sorties afin que le taux d'utilisation dans un circuit donné soit le plus élevé possible.

Il existe plusieurs types de lignes :

**- Les lignes directes :**

Ces interconnexions permettent la liaison entre les CLBs et les IOBs. Il est aussi possible de connecter directement certaines entrées d'un CLB aux sorties d'un autre .

**- Les lignes longues :**

Qui sont des lignes verticales et horizontales, qui n'utilisent pas de matrices de communication. Elles parcourent toutes les zones d'interconnexion. Elles sont utilisées pour véhiculer avec un minimum de retard, les signaux qui doivent parcourir de long trajet.

Ces lignes conviennent pour véhiculer les signaux d'horloge.

**- Matrice d'interconnexion :**

Ce sont des aiguilleurs situés à chaque intersection. Leur rôle est de raccorder les longues lignes entre elles selon diverses configurations. Ces interconnexions sont utilisées pour relier un CLB à n'importe quel autre CLB sur le FPGA.

Chaque CLB est connectable à chacune des synchronisation de ces lignes par des points de connexion. Un point de connexion est un ensemble de six transistors commandés.

La structure générale des interconnexions est présentée sur la figure (1.8).[**6**]

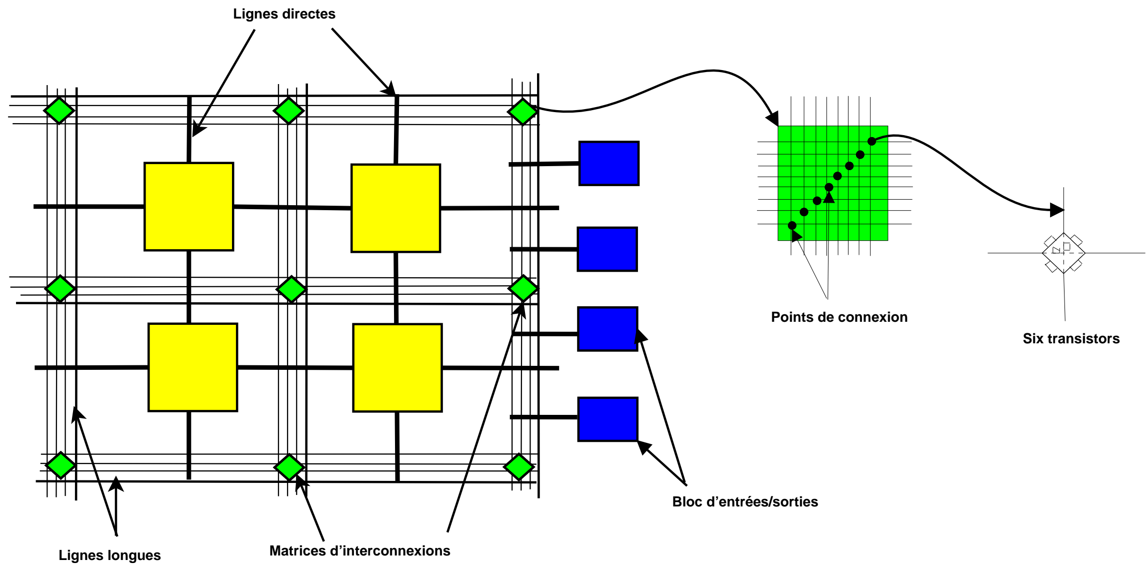



FIGURE 1.8 – Structure générale des interconnexions.

La figure (1.9) illustre la structure générale d'un FPGA.

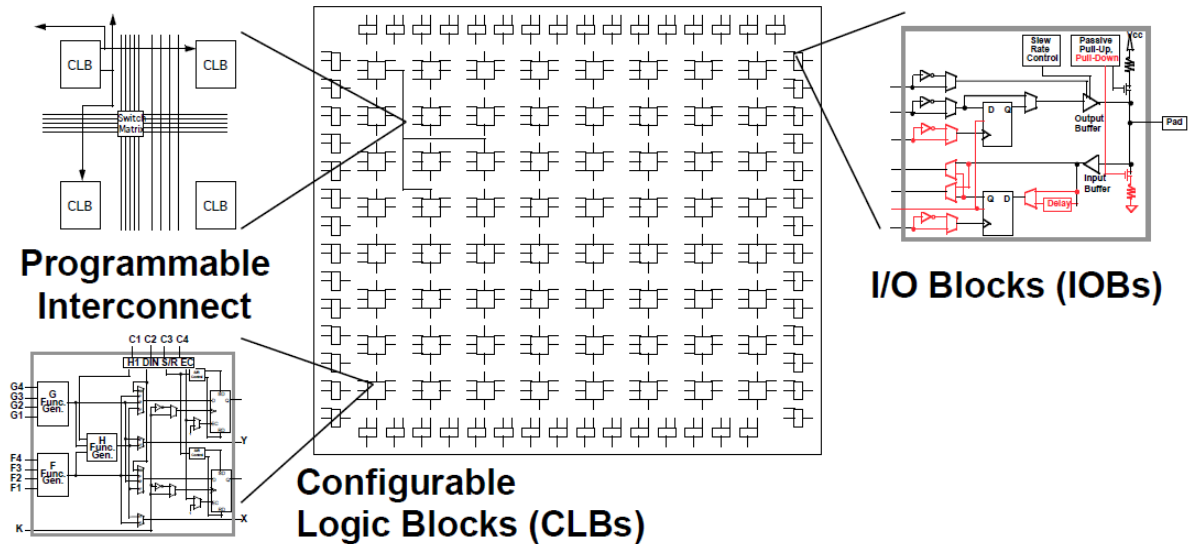



Figure 1.9 – Structure générale d'un FPGA.

## 1.6 Classification des FPGAs

Il existe plusieurs types de FPGA dont notamment :

1. **Les FPGAs basés sur les SRAMs :** Introduit dès 1985 par Xilinx, la configuration du composant réside en deux points : la définition des interconnexions définies par des cellules SRAM et les fonctions booléennes définies dans des mémoires de type SRAM. La mémorisation de la configuration est donc volatile. Ces boitiers sont programmés après chaque mise sous tension. Les données de programmation sont issues soit directement d'une ROM (mémoire à lecture seul), soit d'un système intelligent (microprocesseur ou microcontrôleur).

2. **Les FPGAs à anti fusibles :** Les blocs logiques des FPGAs du type anti-fusible sont généralement beaucoup moins complexes que ceux des SRAMs. Les points d'interconnexions sont de type ROM, c'est-à-dire la modification du point est irréversible.

On notera que contrairement aux cellules de type SRAM, le nombre de sortie est réduit. Pour une fonction donnée, le nombre de blocs logiques nécessaire est supérieur au nombre de blocs nécessaire dans un FPGA de type SRAM.

L'avantage des FPGAs à anti-fusible est le nombre important de points de connexion à cause de la surface réduite du fusible.

L'inconvénient est la non réversibilité de la liaison.

## 1.7 Différents domaines d'application des FPGAs

Leurs utilisations actuelles couvrent les deux domaines : civil et militaire. Parmi ces applications nous citons :

1. **Informatique :** périphériques spécialisés.
2. **Machinerie industrielle :** contrôleur pour machine.
3. **Télécommunications :** traitement d'image, filtrage.
4. **Instrumentation :** Équipement médical, prototypage.
5. **Transport :** contrôle d'avions et métros.
6. **Aérospatiale :** satellites.
7. **Militaire :** radar, communication protégée, la détection ou la surveillance.

## 1.8 Les outils de développement des FPGAs

Ils sont composés en général de deux outils :

1. **Le système de développement :** La conception des architectures de commande s'effectue graphiquement ou via un langage de description matériel de haut niveau, nommé également HDL. Deux langages HDL sont les plus couramment utilisés, à savoir le VHDL et le VERILOG. Ces deux langages sont standardisés et offrent au concepteur différents niveaux de description, et surtout l'avantage d'être portable et compatible avec toutes les technologies FPGA.

2. **Le programmateur :** Il permet de :
   — Vérifier la virginité du circuit.
   — Programmer le circuit.
   — Vérifier la programmation.

## 1.9 FPGA, points faibles-points forts

Un circuit FPGA présente, comme tout autre circuit, un ensemble d'avantages et d'inconvénients. Ces points sont cités dans ce qui suit :

| **Les Avantages** | **Les inconvénients** |
|---|---|
| - Technologie facile à maitriser<br>- Temps de développement réduit.<br>- Programmable et reprogrammable.<br>- cout de réalisation très faible. | - Cout unitaire élevé.<br>- Risque de panne élevé.<br>- Performances limitées.<br>- temps de réponse lent par rapport aux ASIC |

Table 1.2 – Avantages et inconvénients des FPGAs.

## 1.10 Conclusion

Les circuits logiques programmables, ont permis de développer les méthodes de conception des systèmes logiques. Ce chapitre avait pour objectif de présenter la structure des composants suivant une complexité croissante : mémoire, PLD, CPLD et FPGA ; où on s'est intéressé plus particulièrement au FPGAs.

# Chapitre 2

# Le langage de description VHDL

## 2.1 Introduction

Au début des années 80, le département de la défense Américaine (DOD) désire standardiser un langage de description et de documentation des systèmes matériels ainsi qu'un langage logiciel afin d'avoir une indépendance vis-à-vis de leurs fournisseurs. C'est pourquoi le DOD a décidé de définir un langage de spécification. Il a ainsi mandaté des sociétés pour établir un langage. Parmi les langages proposés, le DOD a retenu le langage VHDL qui fut ensuite normalisé par IEEE en 1987.[4]

### 2.1.1 Définition du langage VHDL

Le langage VHDL n'est pas un langage de programmation mais un langage de description de matériel électronique. Il a été défini et introduit dans les outils de CAO électronique pour apporter de la méthode et de la rigueur dans le cycle de développement des systèmes matériels.[7]

Il est considéré comme étant un langage de description de haut niveau car il permet une description globale sans se soucier des détails d'implantation. De plus c'est un langage modulaire et hiérarchique.

Le langage VHDL, contrairement à d'autres langages, est un langage de spécification, de simulation et également de conception.

Avec un langage de description matérielle et un FPGA (Field Programmable Gate Array), un concepteur peut développer rapidement et simuler un circuit numérique sophistiqué, de l'implémenter sur une carte de prototypage, et de vérifier son fonctionnement.

### 2.1.2 Structure d'un programme VHDL

Pour décrire en VHDL un circuit, on aura à spécifier trois parties essentielles qui sont : les bibliothèques, l'entité et l'architecture.

#### 2.1.2.1 Déclaration des bibliothèques

Toute description VHDL utilisée pour la synthèse a besoin de bibliothèques. L'IEEE ( Istitut of Electrical and Electronics Engineers) 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. [10]
C'est par la bibliothèque qu'on entre dans le monde VHDL, elle permet de sauvegarder les résultats de compilation.
La bibliothèque STD est une bibliothèque standard fournie avec le langage. Elle contient des définitions, des types et des fonctions de base (integer, bit, boolean,...).
Les librairies IEEE principales sont :

- `IEEE.std_logic_1164`
- `IEEE.numeric_std`
- `IEEE.std_logic_arith`

Attention, il ne faut pas utiliser les librairies numeric_std et std_logic_arith en même temps, la librairie std_logic_arith est en fait une version améliorée de la numeric_std, elle indique les opérations relatives d'arithmétique et de comparaison, elle contient également plusieurs fonctions de conversion de données, qui permettent à un type d'être converti en d'autres :

conv_integer (a), conv_unsigned (a,b), conv_signed (a,b), conv_std_logic_vector (a,b). [10]

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Use IEEE.numeric_std.all;
Use IEEE.std_logic_unsigned.all;
```

FIGURE 2.1 – Déclaration d'une bibliothèque.

Le mot clé « ***Use*** » permet de sélectionner les bibliothèques à utiliser.

#### 2.1.2.2 Déclaration de l'entité

L'entité correspond au circuit vu de l'extérieur et comprend essentiellement les entrées-sorties du circuit, elle permet de définir les ports par où sont véhiculées les signaux. La première ligne d'une entité indique le nom du circuit.

```
entity   NOM DE L'ENTITE   is
              port   (Description des signaux d'entrées/sorties,
                      leurs noms, leurs types et leurs modes);

end  NOM DE L'ENTITE  ;
```

FIGURE 2.2 – Déclaration d'une entité.

Les entrées/sorties du circuit sont des ports qui doivent avoir le format suivant : **NOM_DU_PORT** : *mode* **TYPE**;

**1. Le MODE** : il peut être :

- **IN** pour un signal en entrée.
- **OUT** pour un signal en sortie.
- **INOUT** pour un signal en entrée et sortie (port bidirectionnel).
- **BUFFER** pour un signal en sortie mais utilisé comme entrée dans la description.

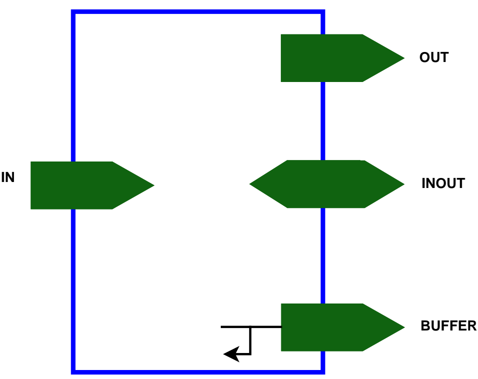



FIGURE 2.3 – Le mode d'un port.

**2. Le TYPE :** il existe deux types principalement utilisés pour les signaux. Ce sont les types :

— **Bit** et **Bit_Vector**.
— **Std_Logic** et **Std_Logic_Vector**.

Les extensions « **_Vector** » indiquent un vecteur composé d'un ensemble de bits.

**2.1.Type Std_Logic** : utilisé pour modéliser des signaux dans un circuit, nous donnons ci-dessous les neufs états de ce type :

| Etat | Définition |
|---|---|
| 0 | Niveau logique 0, forçage fort |
| 1 | Niveau logique 1, forçage fort |
| Z | Haute impédance |
| U | Niveau non initialisé |
| X | Niveau inconnu, forçage fort |
| - | Niveau quelconque |
| L | Niveau logique 0, forçage faible |
| H | Niveau logique 1, forçage faible |
| W | Niveau inconnu, forçage faible |

Table 2.1 – Liste des états du type STD_LOGIC.

**2.2.Type Bit** : Le bit est définit d'une telle façon qu'il peut uniquement prendre comme valeur deux états logiques :

- '0', état logique forcé à 0.
- '1', état logique forcé à 1.

#### 2.1.2.3 Déclaration de l'architecture

L'architecture du circuit correspond au circuit vu de l'intérieur, sa déclaration commence par un en-tête où sont précisés le nom de l'architecture et celui de l'entité dont elle décrit le comportement.L'architecture est divisée en deux parties :

— **Une zone déclarative :** Placée entre un en-tête et le mot-clé BEGIN, elle permet de définir des types, des objets (signaux, constantes) locaux au bloc considéré. Elle permet aussi de déclarer des noms d'objets externes utilisés dans le corps de l'architecture (composants d'une librairie, par exemple).

— **Une zone d'instructions :** Compris entre les mots-clés BEGIN et END. Les instructions du corps de l'architecture sont concurrentes, elles s'exécutent en parallèle.[8]

```
Architecture   nom de l'architecture   of nom de l'entité   is

        Zone de déclaration (facultative ou optionnelle)

Begin

        Description de la structure logique

End   nom de l'architecture   ;
```

FIGURE 2.4 – Déclaration d'une architecture.

La figure suivante résume les trois parties essentielles du langage VHDL citées précédemment.

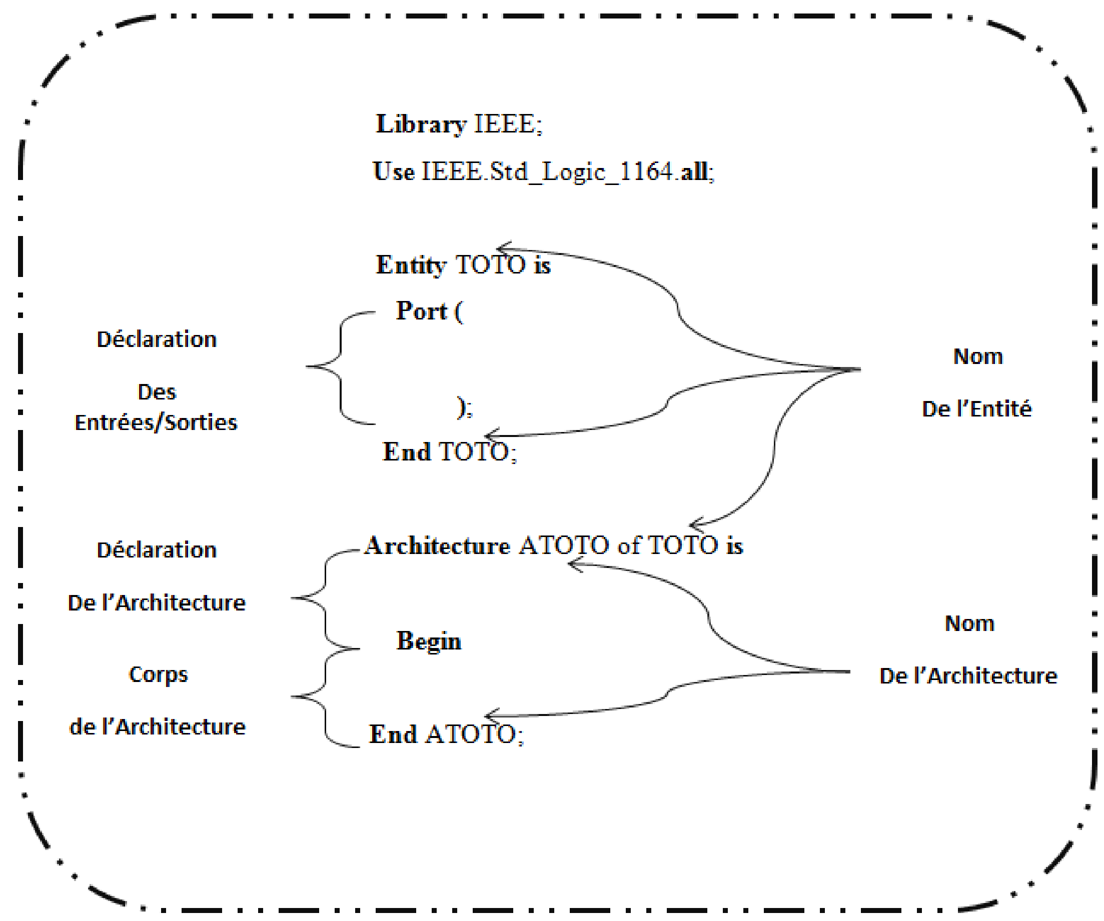



FIGURE 2.5 – Structure d'un programme VHDL .

### 2.1.3 Les différents styles de descriptions d'une architecture

En VHDL, il existe plusieurs styles de description d'une architecture qui diffèrent les unes des autres. Les trois descriptions principales sont :

#### 2.1.3.1 Description de type flot de données

Elle est appelée "flot de donnée" (DATA FLOW) parce que le comportement est décrit à partir du flot des données transitant à l'intérieure de la boite noire (l'entité) que constitue la vue externe. Mais elle est aussi appelée description fonctionnelle parce que l'objectif est d'exprimer les fonctions des sorties par des équations booléennes. C'est donc une description basée sur les opérateurs logiques. [9]

A titre d'exemple la figue ci-dessous donne la spécification d'entité et l'architecture de type flot de données d'un demi additionneur appelé DEMI_ADD.

```
Entity  DEMI_ADD  is

        Port   (A, B:  in std_logic   ;
                S, R:  out std_logic   ) ;

End  DEMI_ADD   ;

Architecture   FLOT  of  DEMI_ADD  is
Begin

   S <=  A XOR B  ;
   R  <=  A AND B  ;

End  FLOT ;
```

FIGURE 2.6 – Description de type flot de données d'un demi additionneur.

#### 2.1.3.2 Description comportementale

Appelée aussi description procédurale, elle décrit le comportement d'un circuit selon des conditions (**IF**), des cas (**CASE**), et des boucles. C'est une description qui comporte des processus (**PROCESS**) qui s'exécutent en parallèles. Les instructions à l'intérieur d'un processus s'exécutent séquentiellement.

**- Le PROCESS :**

On utilise le PROCESS pour faciliter la modélisation d'un système. On peut le considérer comme une boite où il y'a des instructions à l'intérieur qui s'exécutent séquentiellement.[11]

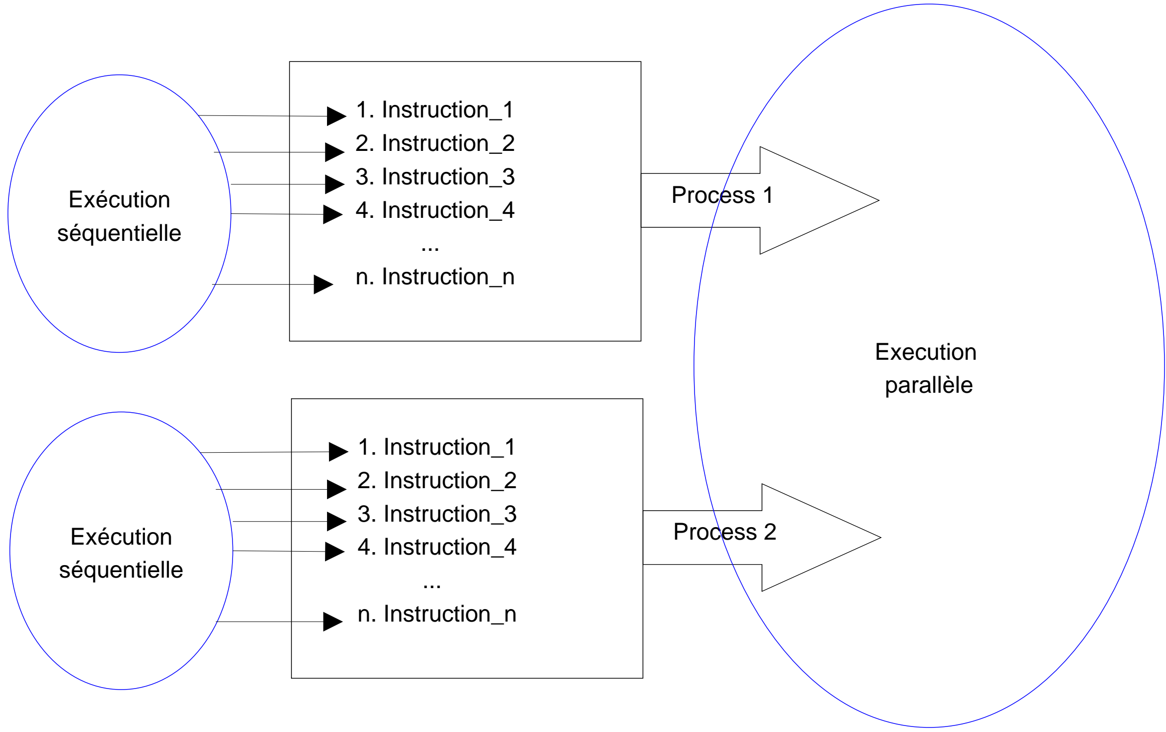



FIGURE 2.7 – Schéma d'exécution des PROCESS.

**1. Description par l'instruction « IF »**

L'architecture est décrite sous forme d'un petit programme informatique avec des instructions séquentielles constituant un processus et traduisant un algorithme. La description avec l'instruction if commence avec le mot réservé « if »,ensuite vient la condition, suivie par le mot réservé « then »,et se termine avec les deux mots réservés « end if ».[11]

L'architecture de la figure ci-dessous en est un exemple.

```
If condition_1 then
séquence_instructions_1;
elsif condition_2 then
séquence_instructions_2;
else
séquence_instructions_3;
end if;
```

FIGURE 2.8 – Description avec l'instruction IF.

**2. Description par l'instruction CASE**

La description avec l'instruction « case » est utilisée lorsque la valeur d'un signal peut être utilisée pour choisir entre un certain nombre d'actions. Cette description débute avec le mot réservé « case » suivi par une instruction, et le mot-clé « is ». L'instruction entre « case » et « is », renvoie une valeur qui correspond à un des choix qui se trouvent dans les déclarations «WHEN» et «OTHERS».[11]

```
Case instruction is

When choix_1 => Instruction_1;

When choix_2 => Instruction_2;

When choix_3 => Instruction_3;
...
When Others => Instruction_n;
```

Figure 2.9 – Description avec l'instruction CASE.

#### 2.1.3.3 Description structurelle

Les deux premières descriptions d'une architecture que nous venons d'étudier, à savoir la description de type flot de données et la description de type comportementale, sont les descriptions de base pour décrire le fonctionnement d'un circuit électronique. La description de type structurelle est plutôt un complément qui va permettre de décrire un assemblage de différents circuits électroniques, chacun de ceux-ci étant une "boite noire" donc une entité. À partir de certain niveau de complexité d'un système logique, il est très difficile de ne pas faire appel à une description de type structurelle des différents blocs, chacun d'eux étant une entité donnant lieu à une architecture décrite dans le type flot de donnée ou le type comportemental.[9](Comme expliqué dans **Annexe** ≪ **1** ≫).

### 2.1.4 Les éléments du langage VHDL

Les trois types de base utilisés lors d'une description VHDL sont :

#### 2.1.4.1 Les constantes :

Ce sont des objets bien connus des langages de programmation. Leur valeurs, une fois définie pendant l'initialisation, reste inchangée par l'exécution, elles peuvent être déclarée dans un **PROCESS** (constante locale), ou en dehors de celui-ci (constante globale). Elles sont déclarée par le mot-clé ≪ **Constant** ≫.[10]

#### 2.1.4.2 Les signaux :

Les signaux sont des objets utilisés pour connecter des éléments concurrents, de la même façon que les fils sont utilisés pour connecter des composants dans un circuit électronique. Les signaux sont principalement utilisée lorsqu'on a besoin de déplacement d'informations entre les différentes parties fonctionnelles d'une conception, comme par exemple entre deux composants connectés. La déclaration d'un signal se fait avec le mot réservé ≪ **SIGNAL** ≫, ce dernier doit avoir au minimum un nom et un type. Si plusieurs signaux du même type sont nécessaires dans une conception, on peut les spécifier dans une seule déclaration.[11]

***Remarque :*** Les signaux doivent être déclarés avant le mot "BEGIN".

#### 2.1.4.3 Les variables :

Une variable est un objet auquel on peut affecter, à tout moment, la valeur qu'on désire, elle ne peut être déclarée qu'à l'intérieur d'un **PROCESS ;** elles sont locales à leur bloc de déclaration. Les variables ne peuvent apparaitre que dans le domaine de programmation séquentielle**.[11]**

La comparaison entre les variable et les signaux est illustrée sur le tableau (2.2).

| Type d'objet | Utilité | Usage | Assignation |
|---|---|---|---|
| Signal | Représente un fil de connexion | Dans les entités et les architectures | <= |
| Variable | Représente une information locale | Dans des exécutions séquentielles | : = |

TABLE 2.2 – Comparaison entre variable et signal.

### 2.1.5 Les opérations de base sur les objets

Le langage VHDL comporte six classes d'opérations avec un niveau de priorité défini pour chaque classe. Lors de l'évaluation d'une expression, l'opération dont la classe a la plus haute priorité est effectuée en premier. Étudions l'une après l'autre, chacune de ces classes en les donnant par ordre de priorité croissante, les opérations logiques ayant donc le niveau de priorité le plus faible.[9]

#### 2.1.5.1 Les opérations logiques :

Ce sont les opérations and, or, nand, nor, xor, soit les fonctions logiques : ET, OU, NON-ET, NON-OU, OU exclusif. Les cinq opérateurs logiques ont la même priorité.

#### 2.1.5.2 Les opérations relationnelles :

Ce sont les relations égal, inégal, plus petit, plus petit ou égal, plus grand, plus grand ou égal (=, /=,<, <=, >, =>), Le résultat de comparaison est de type booléen. Pour rendre l'écriture plus lisible, il est utile et conseillé de mettre entre parenthèses l'opération relationnelle.

#### 2.1.5.3 Les opérations d'addition :

Elles sont au nombre de trois : l'addition, la soustraction et la concaténation de symboles respectifs : +, - et &. La concaténation est définie pour des chaines de bits et des chaines de caractères.

#### 2.1.5.4 Les opérations de signe :

Ce sont les signes **+** ou **-**. Ces opérations sont définies, c'est-à-dire valides, pour les objets de type entier.

***Remarque :***
L'opérateur signe est moins priorité que l'opérateur multiplication qui est *. Ainsi 5*-2 (5 multiplié par -2) est une opération impossible. Il faut écrire : 5*(-2).

#### 2.1.5.5 Les opérations de multiplication :

Ce sont les opérations suivantes :
- La multiplication, de symbole *
- La division, de symbole /
- Le modulo, de symbole mod

#### 2.1.5.6 Les opérations NOT, ABS et ** :

Ce sont les trois opérations de plus haute priorité. L'opérateur NOT prend le compliment de la valeur d'un objet de type bit, boolean, bit_vector et également d'une chaine d'éléments de type booléen.

### 2.1.6 Règles d'écriture du langage VHDL

L'écriture d'une description VHDL obéir aux règles suivantes :

**1. Les commentaires :**
A l'intérieur d'un programme, un commentaire est introduit par deux tirets « **--** » et se prolonge jusqu'à la fin de la ligne. Si le commentaire doit se poursuivre jusqu'à la deuxième ligne,un doubles tiret doit être utilisé au début de cette ligne[12].

**2. Majuscules et minuscules :**

Le langage VHDL ne distingue pas les majuscules et les minuscules. Pour lui les écritures suivantes représentent le même mot : is, IS ou Is. on dit que, le VHDL n'est pas sensible à la casse[12].

**3. Les identificateurs :**

Un identificateur est un nom qui désigne un objet pouvant être : une entité, une architecture, un signal, un processus,...etc. Ce nom est constitué d'une suite de caractères alphabétiques, de caractères numériques ou du caractère "_" souligné. Le premier caractère doit être toujours une lettre, le caractère souligné ne doit pas terminer le nom, ni figurer deux fois consécutives. Ce nom ne doit pas être un mot réservé de VHDL et sa longueur ne doit pas dépasser une ligne[12].

***Remarque :*** Une liste de mot réservé à la syntaxe du langage VHDL, ces mots, interdits comme identificateurs, sont donnée en Annexe (3).

### 2.1.7 Les avantage du langage VHDL

1. **Puissance et flexibilité :** Le VHDL est un langage de haut niveau, moderne, puissant permettant l'écriture du code décrivant des circuits logiques complexes. Il offre en plus la possibilité de créer des bibliothèques de composants réutilisables d'un projet à l'autre ; l'utilité est grande car le code de chaque composant peut être validé par simulation, la recherche des erreurs dans un composant complexe est donc simplifié.

2. **Design indépendant du circuit :** Le VHDL permet de créer une fonction sans se préoccuper du composant dans laquelle elle sera implémentée. Il ne faut pas donc devenir intimement familier avec l'architecture d'un circuit pour l'utilisation des ressources et les performances.

3. **Portabilité :** Portabilité des descriptions VHDL, c'est-à-dire, possibilité de cibler une description VHDL dans le compteur ou la structure qu'on l'on souhaite en utilisant l'outil que l'on veut.

## 2.2 Machine à état

Utilisée pour décrire des systèmes séquentiels quelconques. Il existe deux familles de machines d'état : machine de Moore et machine de Mealy.

### 2.2.1 Machine de Moore

- Les sorties d'une machine de Moore dépendent de l'état présent (synchrones, elles changent sur un front d'horloge).
- L'état futur est calculé à partir des entrées et de l'état présent.

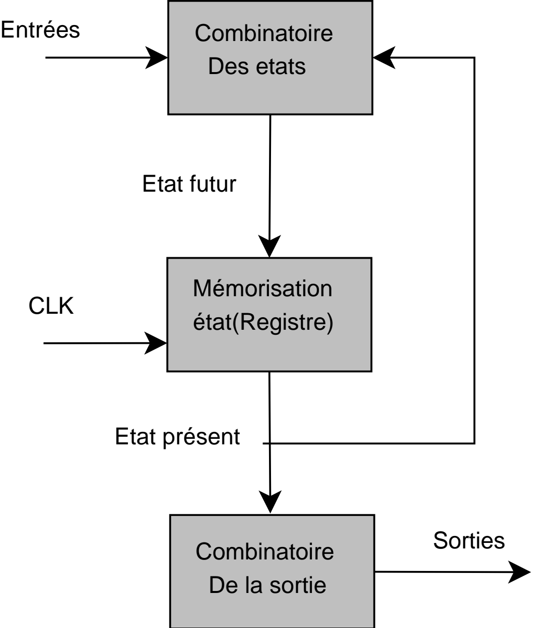



FIGURE 2.10 – Machine d'état de Moore.

- **Exemple :**

Dans ce qui suit on traite un exemple d'une machine séquentielle de Moore, et son code sous VHDL.

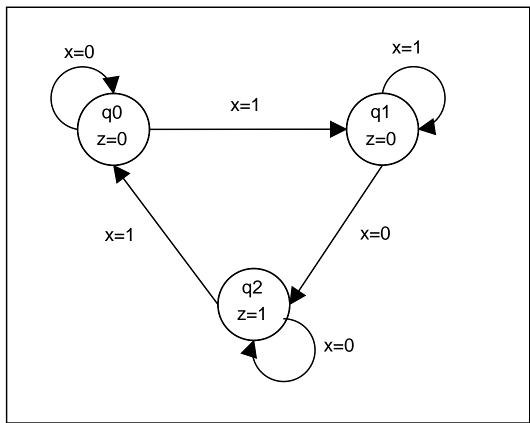



FIGURE 2.11 – Exemple d'une machine de Moore.

```
Architecture A_Moore
of Moore is
Type etat is
(q0,q1,q2);
Signal state,
N_state: etat;
Begin
Process(Clk,Reset)
Begin
if Reset='1' then
State<=q0;
elsif (Clk'event
and Clk='1') then
State<=N_state;
End if;
End Process;
Process(x,state)
Begin
Case state is
When q0=>
z<='0';
If x='0' then
N_state<=q0;
Else
N_state<=q1;
End if ;
When q1=>
z<='0';
If x='1' then
N_state<=q1;
else
N_state<=q2;
End if ;
When q2=>
z<='1';
If x='0' then
N_state<=q2;
Else
N_state<=q0;
End if ;
End case;
End process ;
End A_Moore;
```

### 2.2.2 Machine de Mealy

- L'état futur est calculé à partir des entrées et de l'état présent.
- Les sorties d'une machine de Mealy dépendent de l'état présent et des entrées.
- Mémorisation synchrone des états ( c'est-à-dire un front d'horloge).
- La sortie dépend directement de l'entrée et ceci indépendamment de l'horloge (clk)
- Nombre d'états plus réduit que pour une machine de Moore.

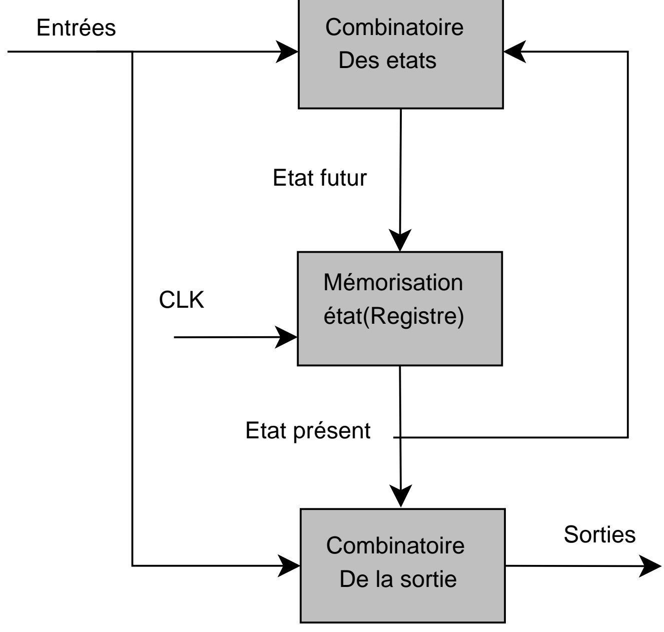



FIGURE 2.12 – Machine d'état de Mealy.

— **Exemple :**

Dans ce qui suit on traite un exemple d’une machine séquentielle de Mealy, et son code sous VHDL.

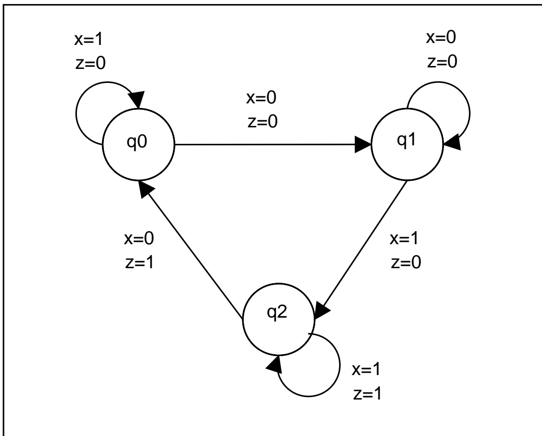



Figure 2.13 – Exemple d’une machine de Mealy.

```
Architecture AMealy
of Mealy is
Type etat is
(q0,q1,q2);
Signal state,
N_state: etat;
Begin
Process(Clk,Reset)
Begin
if Reset=’1’ then
State<=q0;
elsif (Clk’event
and Clk=’1’) then
State<=N_state;
End if;
End Process;
Process (x,state)
Begin
Case state is
When q0 =>
If x=’0’ then z<=’0’;
N_state<=q1;
Else z<=’0’;
N_state<=q0;
End if;
When q1 =>
If x=’0’ then z<=’0’;
N_state<=q1;
Else z<=’0’;
N_state<=q2;
End if;
When q2 =>
If x=’1’ then z<=’1’;
N_state<=q2;
Else z<=’1’;
N_state<=q0;
End if;
End case;
End Process;
End AMealy;
```

## 2.3 Conclusion

Ce chapitre avait pour objectif de présenter les principes de base sur lesquels repose le développement d’un système à l’aide d’une description VHDL. Aussi nous avons présenté la structure du langage VHDL et les éléments fondamentaux de ce langage, ainsi que les différentes méthodes de programmation.

# Chapitre 3

# Conception d’un microprocesseur

## 3.1 Introduction

Le microprocesseur est le composant principal et essentiel de chaque carte mère. Il est le cœur des systèmes automatiques. C’est lui qui gère et contrôle la totalité des procédures et des enchainements, exécute les instructions des programmes. Sans lui, un fonctionnement opérationnel et interactif des autres composants n’est pas possibles, il est relié à tous les autres composants de la carte mère. La structure interne des microprocesseurs s’est considérablement développée au cours de l’évolution du PC (Programme Center), dans ce chapitre nous allons étudier les différentes architectures du microprocesseur en ses différentes versions et fonctionnements.

## 3.2 Définition du microprocesseur

C’est un circuit complexe, qui est capable de réaliser toutes les fonction logiques, mais une seule à la fois. En effet le microprocesseur reçoit un mot binaire de 8 bits ou 16 bits pour exécuter une fonction logique, ce mot binaire est une instruction puisqu’elle instruit le microprocesseur de la fonction à exécuter. Pour faire fonctionner un système à microprocesseur, il faut prévoir une suite d’instructions ordonnées appelée ”Programme”. Le programme est logé dans une mémoire externe.[14]

## 3.3 Architecture de base d’un microprocesseur

Un microprocesseur est construit autour de deux éléments principaux :

— **Unité de commande.**
— **Unité de traitement.**

Associés à des registres chargés de stocker les différentes informations à traiter. Ces trois éléments sont reliés entre eux par des bus internes permettant les échanges d'informations (figure 3.1).

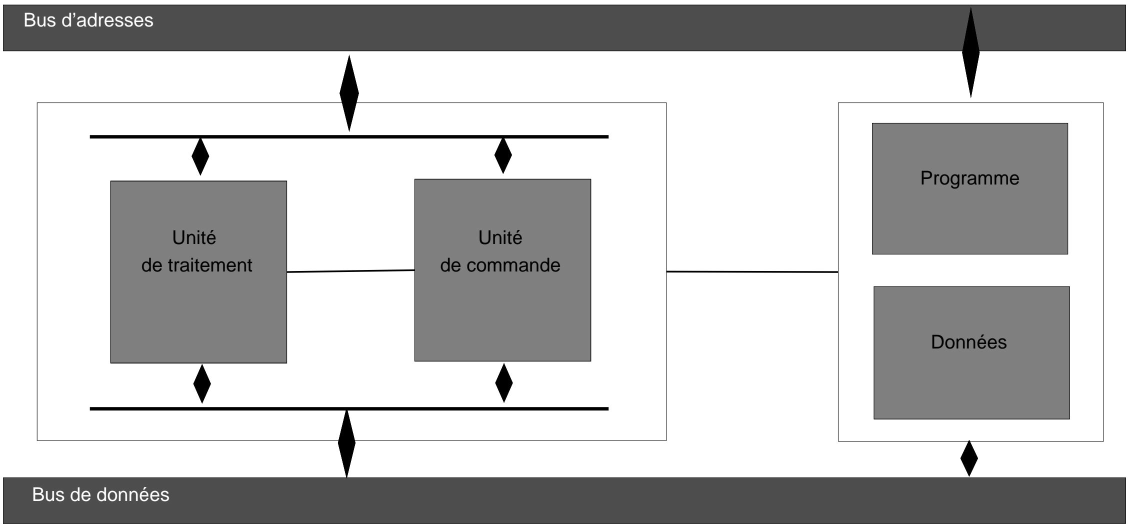



FIGURE 3.1 – Éléments principaux constituant un microprocesseur.

### 3.3.1 Unité de commande

Elle permet de séquencer le déroulement des instructions. Comme chaque instruction est codée sous forme binaire, elle en assure le décodage pour enfin réaliser son exécution puis effectue la préparation de l'instruction suivante. Pour cela, elle est composée par :

— **Compteur de programme (PC) :** constitué d'un registre dont le contenu est initialisé avec l'adresse de la première instruction du programme. Il contient toujours l'adresse de l'instruction à exécuter.

— **Registre d'instruction :** contient l'instruction qui doit être traité par le microprocesseur, cette instruction est recherchée en mémoire puis placée dans ce registre.

— **Registre d'adresse :** c'est un registre qui contient l'adresse du mot à accéder en mémoire centrale. A chaque accès mémoire, l'adresse recherchée est stockée dans ce registre. Il a la taille d'une adresse qui est la même que celle du bus d'adresse ce qui permet de déterminer le nombre de mots mémoires adressables et l'espace mémoire adressable.

— **Séquenceur :** il organise l'exécution des instructions au rythme d'une horloge. Il élabore tous les signaux de synchronisations internes ou externes (bus de commande) du microprocesseur en fonction des divers signaux de commandes provenant du décodeur d'instruction ou du registre d'état par exemple.

### 3.3.2 Unité de traitement

C'est le cœur du microprocesseur. Elle regroupe les circuits qui assurent les traitements nécessaires à l'exécution des instructions :

— **Unité Arithmétique et Logique (UAL) :** est un circuit complexe qui assure les fonctions logiques (ET, OU, Comparaison, Décalage, ...etc.) ou arithmétique (addition, soustraction), afin d'effectuer les calculs et les opérations logiques des différentes instructions à exécuter, les données a traiter se présentent aux entrées de l'UAL, sont traitées, puis le résultat est fourni en sortie.

— **Registre accumulateur :** c'est un registre de travail très important de l'UAL, dans la plupart des opérations l'accumulateur contient un des opérandes avant l'exécution et les résultat après. L'accumulateur a la même taille qu'un mot mémoire.

### 3.3.3 Les bus

Un bus est un ensemble de fils qui assure la transmission du même type d'information. On retrouve trois (03) types de bus véhiculant des informations en parallèle dans un système de traitement programmé de l'information :

— **Bus de données :** bidirectionnel qui assure le transfert des informations entre le microprocesseur et son environnement, et inversement. Son nombre de lignes est égal à la capacité de traitement du microprocesseur.

— **Bus d'adresse :** unidirectionnel qui permet la sélection des informations à traiter dans un espace mémoire qui peut avoir 2n emplacements, avec n est le nombre de ligne du bus d'adresses.

— **Bus de commande :** constitué par quelques conducteurs qui assurent la synchronisation des flux d'informations sur les bus de données et des adresses. [13]

## 3.4 Jeu d'instructions

La première étape de la conception d'un microprocesseur est la définition de son jeu d'instruction. Le jeu d'instruction décrit l'ensemble des opérations élémentaires que le

microprocesseur pourra exécuter. Les instructions que l'on retrouve dans chaque microprocesseur peuvent être classées en trois (03) groupes :

— **Transfert de données :** pour charger ou sauvegarder en mémoire, effectuer des transferts de registre à registre, ...etc.
— **Opérations arithmétiques :** addition, soustraction, division, multiplication.
— **Opérations logiques :** ET, OU, NON, NAND, comparaison, test, ...etc.

## 3.5 Notion d'architecture RISC et CISC

Actuellement l'architecture des microprocesseurs se composent de deux grandes familles :

— Architecture CISC (Complex Instruction Set Computer).
— Architecture RISC (Reduced Instruction Set Computer).

### 3.5.1 Architecture CISC

Les processeurs CISC possèdent un grand nombre d'instructions complexes. Chacune de ces instructions peut effectuer plusieurs opérations élémentaires comme, charger une valeur en mémoire, faire une opération arithmétique et ranger le résultat en mémoire.

### 3.5.2 Architecture RISC

les processeurs RISC possèdent un jeu d'instructions réduit où chaque instruction effectue une seule opération élémentaire. Le jeu d'instructions d'un processeur RISC est plus uniforme. Toutes les instructions sont codées sur la même taille et toutes s'exécutent dans le même temps (un cycle d'horloge en général). Les accès à la mémoire s'effectuent seulement à partir de deux instructions (Load et Store).

### 3.5.3 Comparaison entre les deux architectures

| **Architecture RISC** | **Architecture CISC** |
|---|---|
| Instructions simples ne prenant qu'un seul cycle | Instructions complexes prenant plusieurs cycles |
| Instruction au format fixe | Instructions au format variable |
| Décodeur simple(câblé) | Décodeur complexe (micro code) |
| Beaucoup de registres | Peu de registres |
| Seules les instructions LOAD et STORE ont accès à la mémoire | Toutes les instructions sont susceptibles d'accéder à la mémoire |
| Peu de modes d'adressage | Beaucoup de modes d'adressage |
| Compilateur complexe | Compilateur simple |

TABLE 3.1 – Comparaison entre l'architecture RISC et CISC.

## 3.6 Architecture de Von Neumann

L'architecture, dite architecture de Von Neumann, est un modèle pour microprocesseur qui utilise une structure de stockage unique pour conserver à la fois les instructions et les données requises ou générées par le calcul. La séparation entre le stockage et le processeur est implicite dans ce modèle. Une même instruction permet l'accès au programme ou aux données, cependant pas simultanément. Cette architecture est maintenant principalement utilisée pour la conception des processeurs d'ordinateurs.

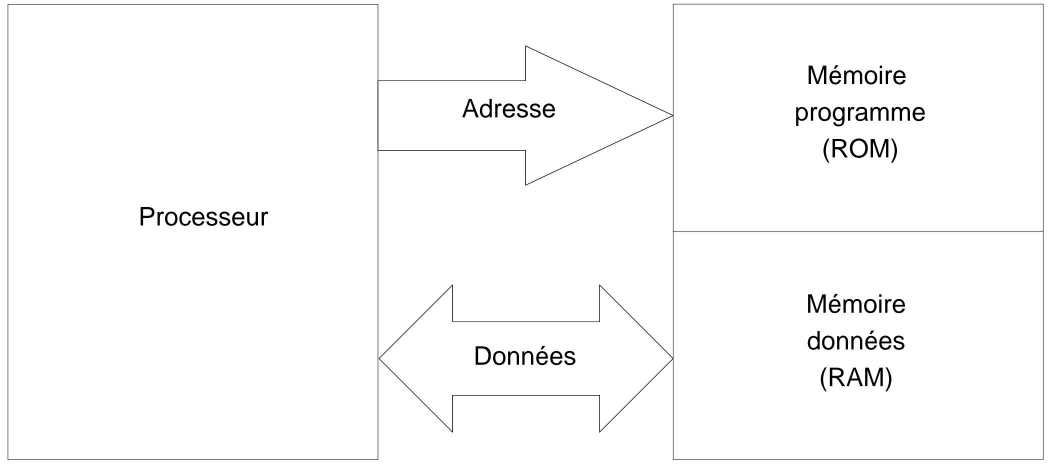



FIGURE 3.2 – Architecture de Von Neumann.

Ce type d'architecture nécessite plusieurs cycles d'horloge pour exécuter une instruction.

## 3.7 Architecture de Harvard

L'architecture de type Harvard est une conception de microprocesseurs qui sépare physiquement la mémoire de données et la mémoire programme. L'accès à chacune des deux mémoires s'effectue via deux bus distincts. Cette structure permet un accès simultané aux données et aux instructions l'exécution des programmes est plus rapide. En revanche elle nécessite des instructions différentes pour accéder à la mémoire programme et à la mémoire de données. Cette architecture très employée pour la conception des processeurs de traitement de signal (DSP) est de plus en plus utilisée pour les microprocesseur d'usage généraux.

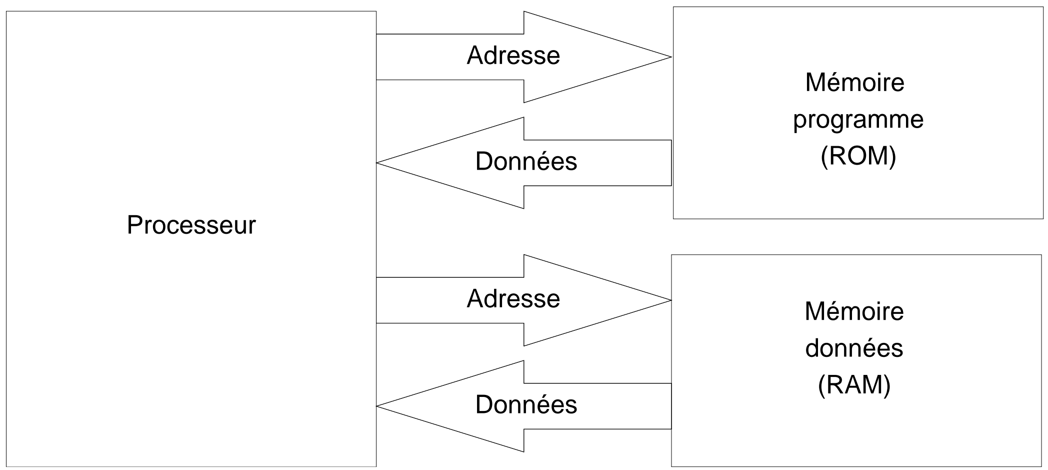



FIGURE 3.3 – Architecture de Harvard.

## 3.8 Conception d'un microprocesseur étape par étape

Dans cette section, nous aborderons les différentes versions du microprocesseur en utilisant l'architecture de Von Neumann[14].

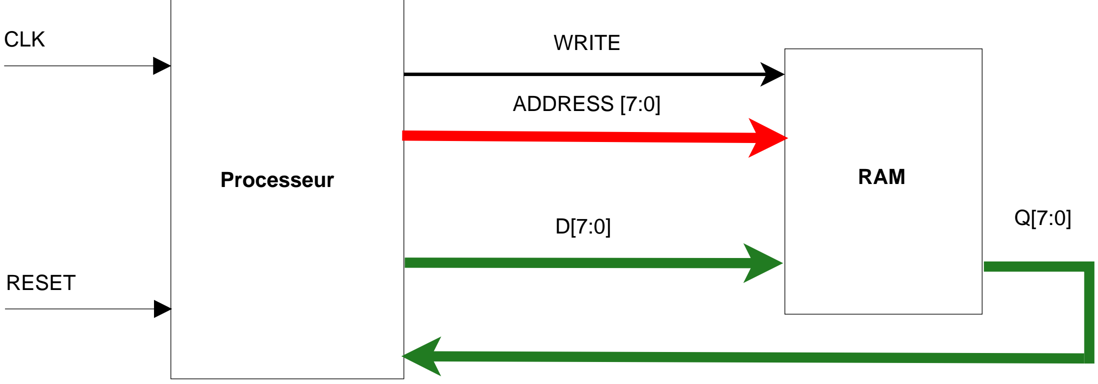



FIGURE 3.4 – Schéma synoptique du microprocesseur à réaliser.

Une RAM de 256 octets est reliée au processeur par ses bus de données, bus d'adresses et la ligne de WRITE, comme le montre la figure précédente.

### 3.8.1 Microprocesseur linéaire basique (Version 1)

On voudrait réaliser un microprocesseur le plus simple qui soit. Pour cela, le processeur sera relié à une mémoire vive (RAM)[14].

**1. Organisation de la mémoire :**

On suppose que le programme (opérations à effectuer) ainsi que les données sont déjà chargées dans la mémoire, et qu'ils respectent le format suivant :

| Adresse | Type de mot stocké | Exemple |
|---|---|---|
| 0 | Instruction | + |
| 1 | Donnée(premier opérande) | 3 |
| 2 | Donnée(deuxième opérande) | 4 |
| 3 | Donnée(résultat) | X |
| 4 | Instruction | - |
| 5 | Donnée(premier opérande) | 12 |
| 6 | Donnée(deuxième opérande) | 4 |
| 7 | Donnée(résultat) | X |

TABLE 3.2 – Organisation de la mémoire du microprocesseur basique.

Le "X" indique que la RAM ne contient rien de valide à cet endroit là. C'est au microprocesseur d'aller y écrire le résultat correct.

**2. Jeu d'instruction :**

On a utilisé des opérations arithmétiques et logiques, ces opérations sont codées sur 8 bits.

Le tableau ci-dessous représente le code de ces opérations.

| Code (binaire sur 8 bits) | Instruction |
|---|---|
| 00000001 | Addition |
| 00000010 | Soustraction |
| 00000011 | AND |
| 00000100 | OR |

TABLE 3.3 – Code d'instruction du microprocesseur basique.

**3. Fonctionnement du microprocesseur :**

Vu l'organisation de la RAM qui a été choisie, le fonctionnement de l'automate est simple. À chaque coup d'horloge, l'automate aura pour fonction de trouver successivement une instruction, puis le premier opérande, puis le deuxième opérande, calculer le résultat et le stocker. Le premier cycle étant fait, la reprise à partir du premier coup d'horloge permet de boucler un autre cycle.

- **Fonctionnement détaillé :**
  - Premier coup d'horloge **:** le compteur présente l'adresse "0" à la RAM. Sur la RAM, le contenu de l'adresse "0" est la première instruction.
  - Deuxième coup d'horloge : le compteur incrémente l'adresse qu'il présente à la RAM ("1"). Sur la RAM, le contenu de l'adresse "1" est le premier opérande.
  - Troisième coup d'horloge : le compteur incrémente l'adresse qu'il présente à la RAM ("2"). Sur la RAM, le contenu de l'adresse "2" est le deuxième opérande. À ce moment là, le microprocesseur dispose de toutes données nécessaire pour le calcul : l'instruction, et les deux opérandes.
  - Quatrième coup d'horloge : le compteur incrémente l'adresse qu'il présente à la RAM ("3"). Sur le bus de donnée en entrée de la RAM, on trouve le résultat issu du troisième coup d'horloge, aussi la ligne WRITE de la RAM se met à l'état haut. Ce qui fait, de stocker ce résultat à l'adresse "3" de la mémoire.

**4. Fonctionnement de la machine à état :**

Vu l'organisation de la mémoire, la machine à état a un fonctionnement linéaire, l'ordre des actions effectuées est toujours le même :

- **État 1 :** chercher une instruction.
- **État 2 :** chercher le premier opérande.
- **État 3 :** chercher le deuxième opérande.
- **État 4 :** stocker le résultat du calcul.

La figure (3.5) présente le graphe d'états de la première version.

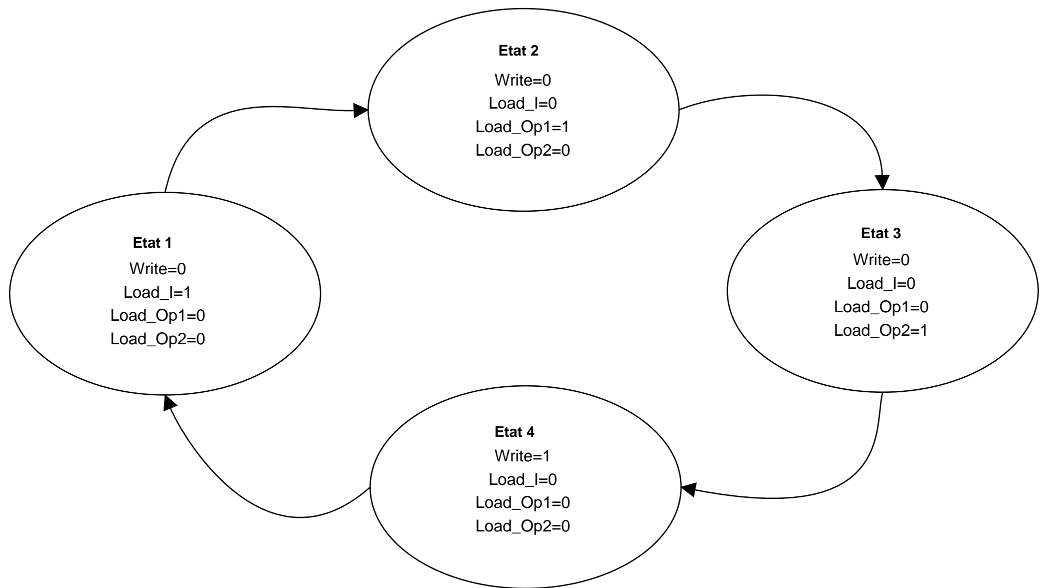



FIGURE 3.5 – Graphe d'états de la première version.

L'architecture de ce premier modèle est la suivante :

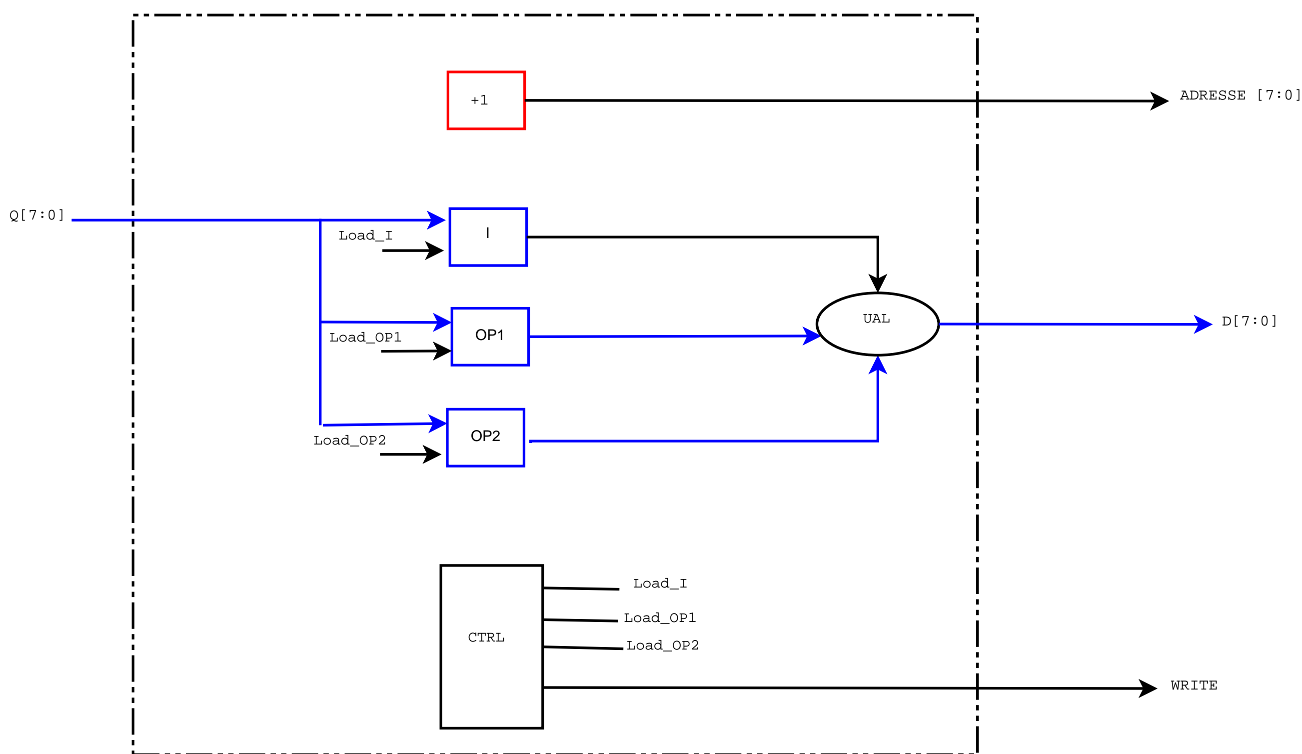



FIGURE 3.6 – Architecture du microprocesseur basique.

Avec : **CTRL,** La machine à état qui séquence les opérations.

**Inconvénient :**

L'architecture actuelle ne permet pas d'enchainer les calculs. Pour le faire, on supprime les opérations sur deux (02) opérandes en RAM. Toutes les opérations à deux (02) opérandes se feront entre un opérande en RAM, et une donnée sauvegardée dans un registre accumulateur. Pour rendre cela possible, on définit deux nouvelles instructions :

— Chargement de l'accumulateur à partir d'une donnée en RAM.
— Stockage du contenu de l'accumulateur en RAM.

### 3.8.2 Microprocesseur avec accumulateur (Version 2)

Comme cité dans la version précédente, la solution qu'on va apporter à l'inconvénient est d'ajouter un accumulateur à l'architecture de la version actuelle.

On ajoute donc deux instructions à notre processeur :

**Load :** chargement de l'accumulateur à partir de la mémoire.
**Store :** stockage du contenu de l'accumulateur dans la mémoire.

**1. Organisation de la mémoire :**

Le contenu de la mémoire se présente comme suit :

| Adresse | Type de mot stocké | Exemple | Effet |
|---|---|---|---|
| 0 | Instruction | Load | |
| 1 | Donnée | 3 | L'accumulateur contient maintenant '3' |
| 2 | Instruction | + | |
| 3 | Donnée | 4 | L'accumulateur contient maintenant '7' |
| 4 | Instruction | - | |
| 5 | Donnée | 1 | L'accumulateur contient maintenant '6' |
| 6 | Instruction | Store | |
| 7 | Donnée | X | Après l'exécution du programme cet emplacement en RAM contiendra '6' |

Table 3.4 – Exemple d'organisation de la mémoire du microprocesseur avec accumulateur.

Pour effectuer "3 + 4 -1" :

1. Chargement de 3 dans l'accumulateur.
2. Addition du résultat de l'accumulateur avec un opérande en RAM ("4").
3. Soustraction du résultat de l'accumulateur avec un opérande en RAM ("1").
4. Stockage du contenu de l'accumulateur en RAM.

**2. Jeu d'instruction :**

Le nouveau jeu d'instruction devient donc :

| Code | Instruction | Effet |
|---|---|---|
| 00000001 | Addition | Additionner le contenu de l'accumulateur à une donnée en RAM, le résultat est stocké dans l'accumulateur. |
| 00000010 | Soustraction | Soustraire le contenu de l'accumulateur une donnée en RAM, le résultat est stocké dans l'accumulateur. |
| 00000011 | AND | Effectuer un ET bit à bit entre le contenu de l'accumulateur et une donnée en RAM, le résultat est stocké dans l'accumulateur. |
| 00000100 | OR | Effectuer un OU bit à bit entre le contenu de l'accumulateur et une donnée en RAM,le résultat est stocké dans l'accumulateur. |
| 00000101 | Load | Charger dans l'accumulateur une donnée en RAM. |
| 00000110 | Store | Stocker le contenu de l'accumulateur en RAM. |

Table 3.5 – Code d'instruction du microprocesseur avec accumulateur.

**3. Fonctionnement du microprocesseur :**

A chaque instruction, le processeur effectue ceci :

1. **Pour une instruction arithmétique ou logique :**
   (a) Chercher l'instruction en mémoire, la stocker dans le registre I.
   (b) Lire l'opérande en mémoire, effectuer le calcul et stocker le résultat dans l'accumulateur.

2. **Pour un LOAD :**
   (a) Chercher l'instruction en mémoire, la stocker dans I.
   (b) Lire l'opérande en mémoire, et le stocker dans l'accumulateur.

3. **Pour un STORE :**
   (a) Chercher l'instruction en mémoire, la stocker dans I.
   (b) Écrire le contenu de l'accumulateur en mémoire à l'adresse courante.

— **Principe de l'accumulateur :**

Lors du second cycle l'accumulateur peut subir trois traitements différents :

1. Pour une opération (+, -, xor, or, ...), l'accumulateur est modifié avec le résultat de l'opération.
2. Pour un load, l'accumulateur est chargé par la donnée sortant de la mémoire.

3. Pour un store, l'accumulateur n'est pas modifié.

En entrée de l'accumulateur, on mettra donc un multiplexeur qui présentera soit le résultat de l'opération en cours (si on exécute une opération normale) soit le contenu de la RAM (si on exécute un load). La sortie de l'accumulateur est branchée en même temps : sur le bus d'entrée de la RAM et sur l'ALU.

**4. Fonctionnement de la machine à état :**

Pour effectuer des calculs, le processeur a besoin de deux informations, l'instruction et l'opérande. On garde donc le registre d'instruction (8 bits) qui stocke l'instruction à effectuer. Maintenant, on utilise une machine à deux états :

— **État 1 :** lire en mémoire l'instruction pointée par le compteur programme et la transférer dans le registre d'instruction RI, puis incrémenter le PC pour qu'il pointe sur l'instruction suivante.
— **État 2 :** l'opérande est lue en mémoire, l'opération UAL est exécutée, et le résultat est rangé dans l'accumulateur.

La figure (3.7) présente le graphe d'états de la deuxième version.

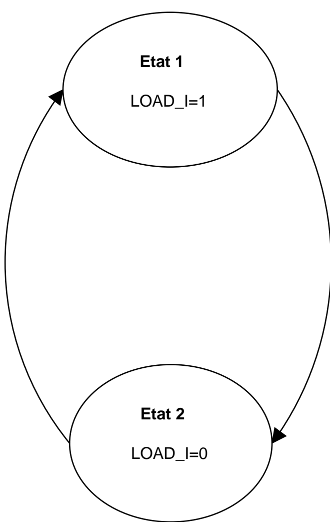



FIGURE 3.7 – Graphe d'états de la deuxième version.

L'architecture de cette version est la suivante :

FIGURE 3.8 – Architecture du microprocesseur avec accumulateur.

**Inconvénient :**

L'architecture actuelle ne permet pas de :

— Faire tourner un même code sur des données différentes.
— Faire tourner différents codes sur des mêmes données.

### 3.8.3 Microprocesseur avec accumulateur et indirection (Version 3)

Dans cette architecture les adresses du code sont globalement linéaire, celles de données ne le sont pas, d'où on a ajouté un multiplexeur pour présenter sur le bus d'adresse RAM soit :

— Le compteur d'adresse pendant les deux premiers cycles, ce compteur doit être piloté par un signal INCR_PC.
— Le contenu du registre d'adresse.

**1. Organisation de la mémoire :**

On n'accède plus à la RAM d'une façon linéaire. Le contenu de la mémoire se présente comme suit :

| Adresse | Type du mot stocké | Exemple | Zone |
|---|---|---|---|
| 0 | Instruction | load | Zone de code |
| 1 | Adresse de l'opérande | 100 | |
| 2 | Instruction | + | |
| 3 | Adresse de l'opérande | 101 | |
| 4 | Instruction | store | |
| 5 | Adresse de l'opérande | 104 | |
| ... | ... | ... | |
| 100 | donnée | 3 | Zone de donnée |
| 101 | donnée | 4 | |
| 104 | donnée | X | |

Table 3.6 – Code d'instruction du microprocesseur avec accumulateur et indirection.

**2. Fonctionnement du microprocesseur :**

Pour chaque front montant de l'horloge, on a le fonctionnement suivant :

- Premier coup d'horloge :
  - La mémoire présente l'instruction (1) qui est : "LOAD" ;
  - L'instruction "LOAD" est affectée au registre d'instruction.
- Deuxième coup d'horloge :
  - La mémoire présente l'adresse de l'opérande (100) ;
  - L'adresse de l'opérande est affectée au registre d'adresse.
- Troisième coup d'horloge :
  - La mémoire présente l'opérande qui est à l'adresse (101) ;
  - La donnée est chargée dans l'accumulateur, qui passera à l'UAL.
- Quatrième coup d'horloge :
  - La mémoire présente l'instruction (2) qui est : "Addition" ;
  - L'instruction "Addition" est affectée au registre d'instruction.
- Cinquième coup d'horloge :
  - La mémoire présente l'adresse de l'opérande (104) ;
  - L'adresse de l'opérande est affectée au registre d'adresse.
- Sixième coup d'horloge :
  - La mémoire présente l'opérande qui est à l'adresse (104) ;
  - La donnée est chargée dans l'accumulateur, effectuer l'opération et stocker le résultat dans l'accumulateur.

- Septième coup d'horloge :
  - La mémoire présente l'instruction (3) qui est : "STORE" ;
  - L'instruction "STORE" est affectée au registre d'instruction.

- Huitième coup d'horloge :
  - La mémoire présente l'adresse de l'opérande (106) ;
  - L'adresse de l'opérande est affectée au registre d'instruction.

- Neuvième coup d'horloge :
  - Écriture du résultat obtenu dans l'adresse (106).

**3. Fonctionnement de la machine à état :**

L'automate doit maintenant pour chaque instruction :

- **État 1 :** Chercher l'instruction (la stocker dans le registre d'instruction).
- **État 2 :** Chercher l'adresse de l'opérande (la stocker, dans le "registre d'adresse").
- **État 3 :** Chercher l'opérande, en lisant l'adresse de la RAM obtenue à l'état précédent.

On a donc une machine qui possède un état de plus (opérande en RAM). Comme le montre la figure suivante :

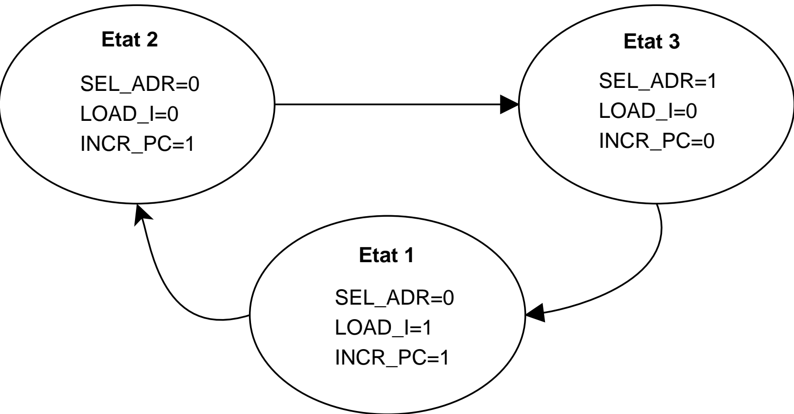



FIGURE 3.9 – Graphe d'état de la troisième version.

L'architecture de cette version est la suivante :

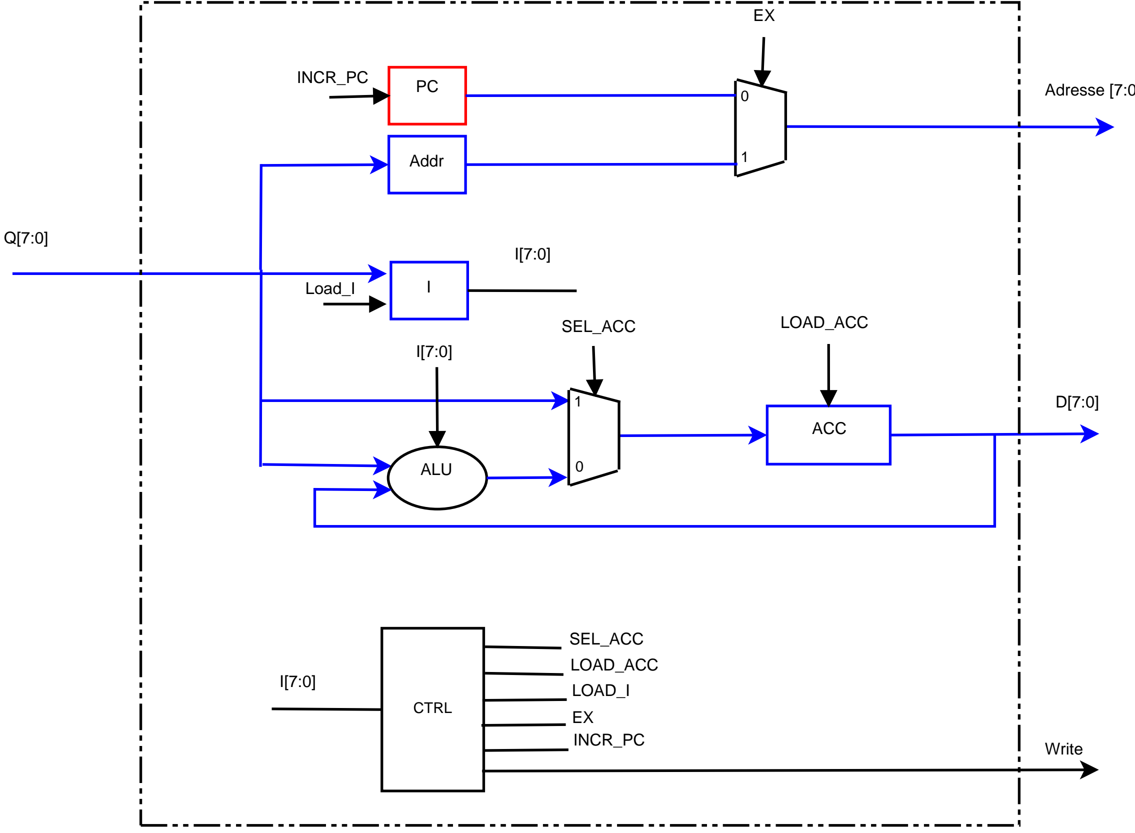



FIGURE 3.10 – Architecture de microprocesseur avec un accumulateur et indirection.

## 3.9 Conclusion

Ce chapitre nous a permis de présenter l'architecture de base d'un microprocesseur et ses constituants. Aussi, de présenter les trois (03) différentes versions du microprocesseur et leur fonctionnement, ainsi que le fonctionnement des machines à état.

Le chapitre suivant sera consacré pour la réalisation de ce microprocesseur, ainsi que leur implémentation sur la carte FPGA.

# Chapitre 4

# Réalisation et résultats expérimentaux

## 4.1 Introduction

Afin de mettre en œuvre la réalisation des différentes versions du microprocesseur précédemment décrites, on aura à utiliser un outil de développement appelé "Quartus II", qui est le compilateur du FPGA intégré sur la carte. Après validation de ces différentes architecture, l'implémentation se fera sur une carte DE2.

## 4.2 Présentation de la carte DE2

La carte DE2 est un outil de développement et de prototypage pour circuits spécifiques développée par la société ALTERA. Au cœur de la carte se trouve le circuit FPGA CYCLONE II EP2C35F672C6, elle est dotée de plusieurs périphériques externes permettant de tester diverses solutions.

La figure ci-dessous nous montre la disposition des composants de la carte DE2 et indique l'emplacement des connecteurs et des composants.

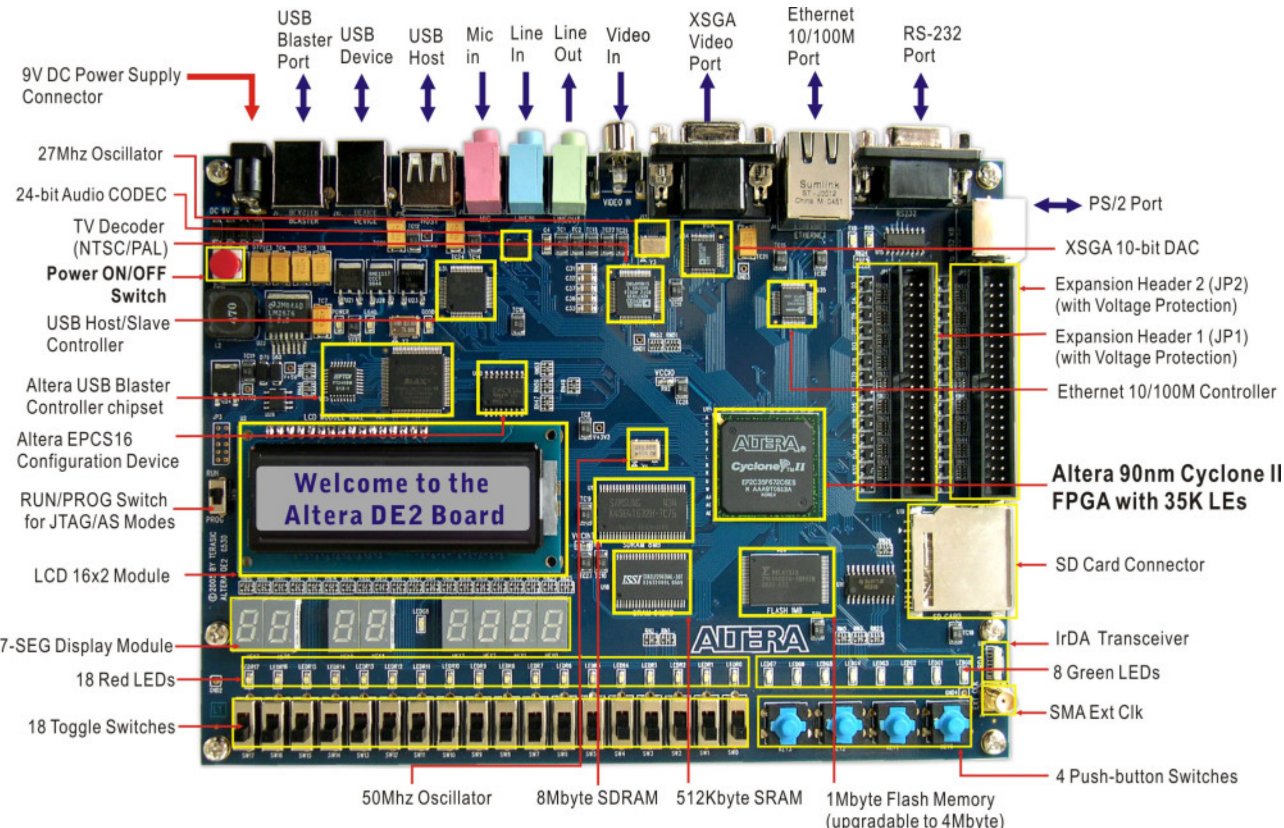



FIGURE 4.1 – Aperçu de la carte DE2.

### 4.2.1 Les périphériques de la carte DE2

— 18 interrupteurs.
— 4 Boutons poussoirs.
— 8 afficheurs 7 segments.
— 18 diodes.
— 27 LEDS (18 rouges et 8 vertes).
— 1 afficheur LCD 2 rangées de 16 caractères.
— 1 entrée/sortie clavier/souris et RS232.
— 2 ports USB.
— 1 sortie moniteur VGA.
— 1 entrée vidéo (NTSC/PAL/multi-format).
— Un codec Audio.
— Un bouton d'allumage.

#### 4.2.1.1 Boutons poussoirs

— 4 boutons poussoirs reliés à 4 broches du cyclone.

— Quand le bouton est dans la position relâchée, la broche reçoit un 1 logique.
— Quand le bouton est dans la position appuyée, la broche reçoit un 0 logique.

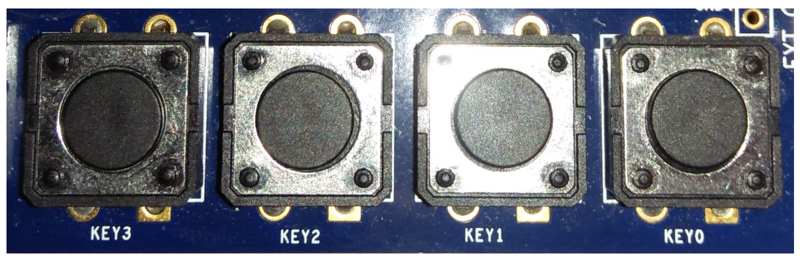



FIGURE 4.2 – Boutons poussoirs .

#### 4.2.1.2 Interrupteurs

— 18 interrupteurs reliés à 18 broches du cyclone.
— Quand l'interrupteur est du coté du bord de la carte, il retourne un 0 logique.
— Quand l'interrupteur est du coté du centre de la carte, il retourne un 1 logique.

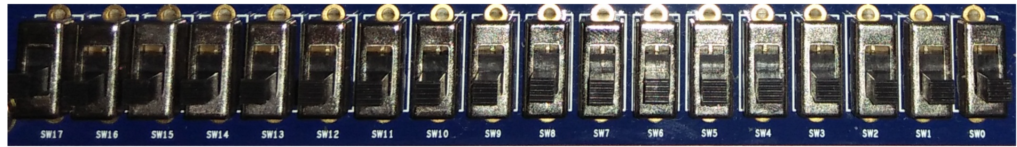



FIGURE 4.3 – Interrupteurs (les Switchs).

#### 4.2.1.3 Led

27 Leds au total, reliées à 27 sorties du cyclone. Mettre à "1" la sortie allume la Led, mettre à "0" la sortie éteint la Led.

FIGURE 4.4 – LEDs .

#### 4.2.1.4 Afficheurs 7-segments

— 8 afficheurs à 7 segments anode commune, disposés en deux paires et un groupe de 4.
— Chaque segment d'un afficheur est relié à une sortie du cyclone.
— Les segments sont numérotés de 0 à 7 comme indiqué dans la figure suivante.

FIGURE 4.5 – Afficheurs 7-segments .

Mettre à 0 une sortie du segment permet de l'allumer, mettre à 1 permet de l'éteindre.

### 4.2.2 Utilisation de la carte DE2

Deux alternatives :

1. Travailler directement avec les outils accompagnant la DE2 (Utiliser le logiciel Quartus II ).
2. Travailler avec le logiciel Active VHDL, ensuite exporter vers le logiciel Quartus II.

## 4.3 Présentation du logiciel de programmation « Quartus II »

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 et d'en réaliser une simulation, une synthèse et une implémentation sur cible reprogrammable.[?]

### 4.3.1 Création d'un projet

Quartus est un logiciel qui travaille sous forme de projet, 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 clique sur : **Démarrer → Programmes → Altera → Quartus II.**

Lors de l'ouverture du logiciel, l'espace de travail illustré à la figure (4.6) est présenté à l'utilisateur. Cet espace de travail contient :

1. La barre des menus.
2. La barre d'outils.
3. Le navigateur de projet permet de voir les fichiers, les projets et leur hiérarchie.
4. La fenêtre de statut vous permet de connaitre l'état d'avancement de la compilation de votre circuit.
5. La fenêtre de message vous avertit des erreurs et autres avertissements durant la compilation ou la programmation.

Si la fenêtre utilitaire "Messages" n'est pas visible, pour la faire apparaître, on choisi le menu : **View** → **Utility Windows** → **Messages (Alt-3)**

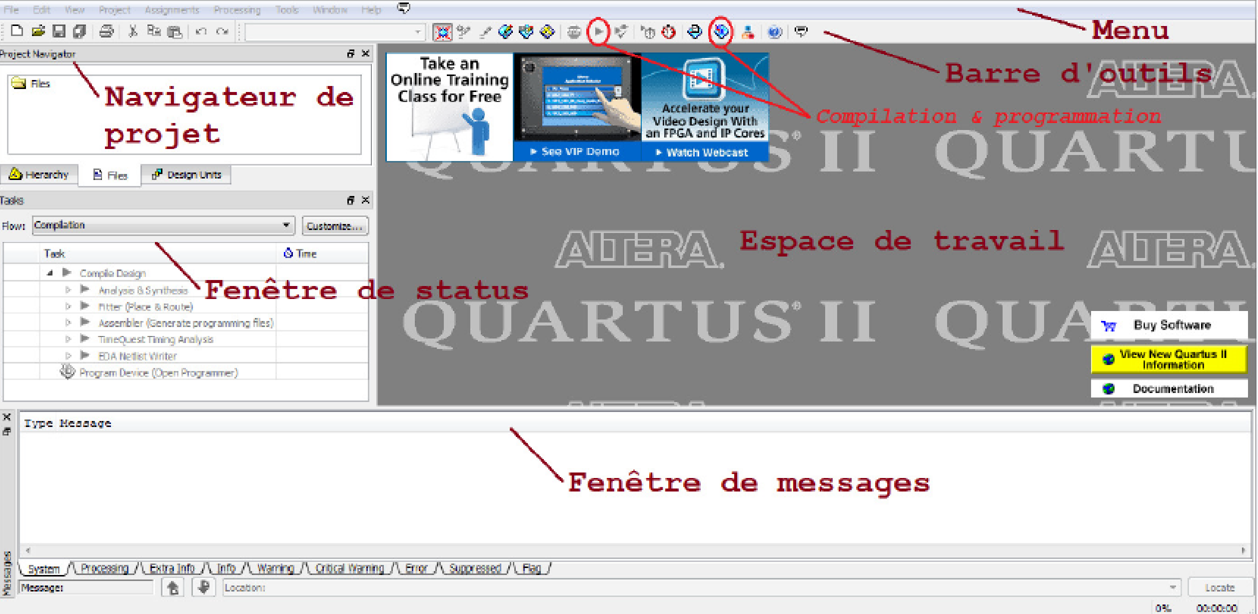



FIGURE 4.6 – Présentation de l'environnement de conception Quartus II.

Afin de créer un nouveau projet, on procède par le menu :
**File** → **New Project Wizard,** puis se laisser guider. Une nouvelle fenêtre comme le montre la figure ci-dessous permettant de configurer le projet apparait. Dans cette dernière trois champs sont à renseigner :

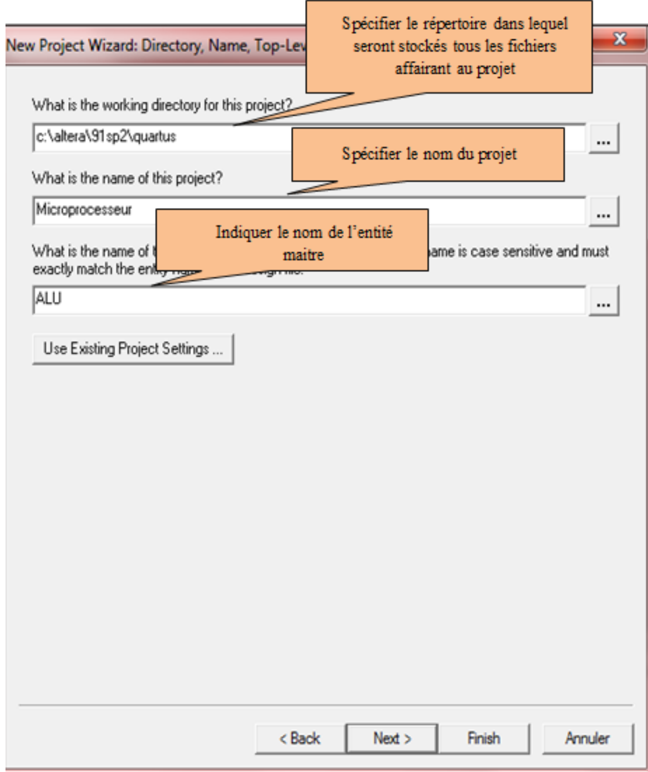



FIGURE 4.7 – La fenêtre qui permet la configuration du projet.

-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.

Cliquer sur **Next**, puis quand la fenêtre **Add Files** apparait recliquer sur **Next**. Dans la fenêtre suivante intitulée **Family & Device Settings**, choisir le circuit logique programmable que l'on souhaite utiliser. Dans notre cas, nous choisirons un FPGA de la famille « **Cyclone II** ».

Par la suite, dans la liste « **Available Devices** » sélectionnons « **EP2C35F672C6** », qui est le FPGA que nous utilisons. Quand la fenêtre **EDA Tool Settings** apparait cliquer sur **Next**. Une fenêtre récapitulative apparait, 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, comme montré dans la figure suivante.

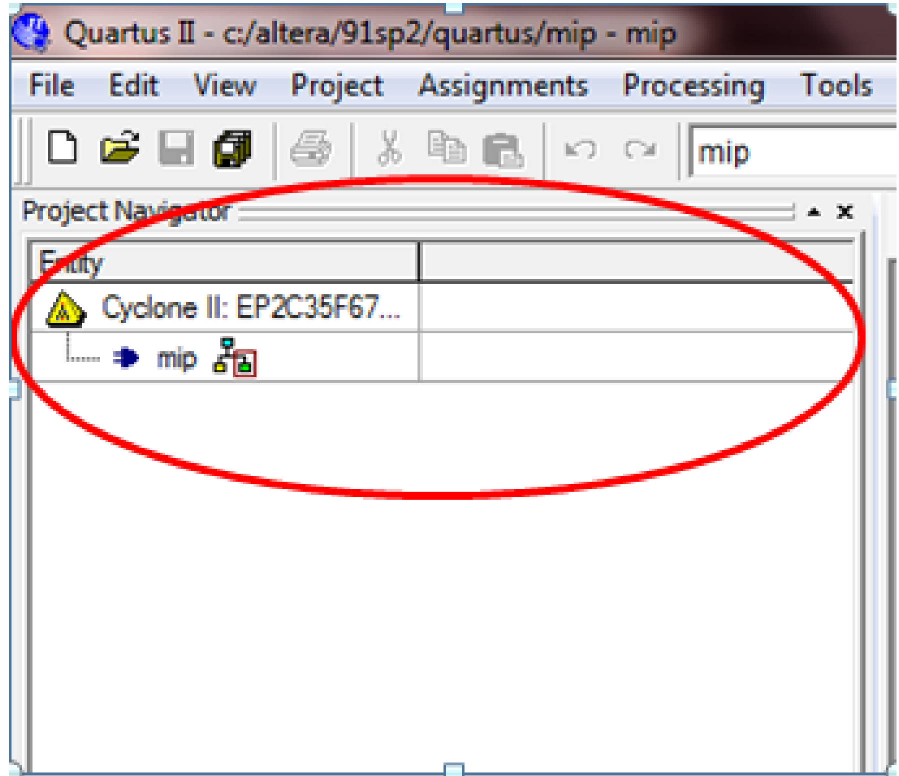



FIGURE 4.8 – La fenêtre apparait après la validation des choix .

### 4.3.2 Saisie d'un projet

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

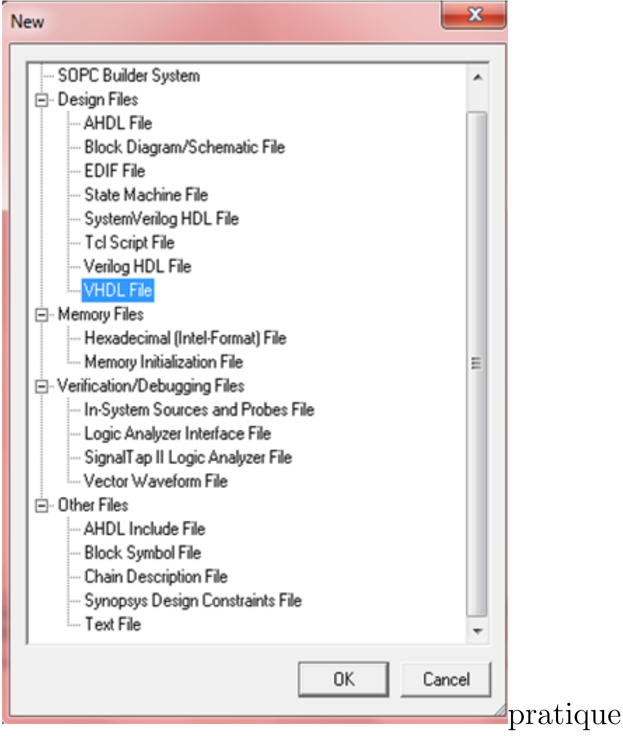
- Une saisie graphique en assemblant des symboles.

- Une saisie textuelle à l'aide de différents langages (VHDL, Verilog,...).

Dans notre projet nous utilisons la saisie textuelle.

— **Saisie textuelle en VHDL** : La saisie d'un composant VHDL se fait de la manière suivante :

Aller dans le menu : **File** → **New.** La fenêtre suivante apparait :



pratique

FIGURE 4.9 – Fenêtre de choix de la saisie .

Choisir **VHDL File** et faire **OK**. Un petit éditeur de texte apparait.

### 4.3.3 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, faire **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). 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 qui apparait cliquer sur **Node Finder**, ce qui permet de lancer le navigateur de signaux, puis suivre les cinq étapes suivantes :

1. Cliquer sur **list** afin de faire apparaitre les signaux du design.
2. Sélectionner les signaux voulus.
3. Cliquer sur la flèche correspondante.

4. Vérifier que tous les signaux qu'on veut visualiser sont dans **Selected Nodes**.
5. Valider par **OK,** cliquer sur **OK** dans la fenêtre **Insert Node or Bus**.

Afin de donner des valeurs de stimuli à un signal, on sélectionne le signal dans la colonne **Name** puis pour insérer une horloge on sélectionne menu **Edit** → **Value** → **Clock**

En dernier lieu, il ne reste plus qu'à lancer la simulation par la commande : **Assignements** → **Settings,** puis dans la colonne **Category** sélectionner **Simulator Settings** par la suite suivre les étapes suivantes :

— Choisir une simulation fonctionnelle (Fonctional Simulation) pour le mode de simulation (Simulation Mode).
— Ajouter le fichier (Par exemple Simul_ALU) pour les paramètres d'entrées de la simulation (Simulation Input).
— Valider par **OK.**
— Dans le menu **Processing**, sélectionner « **Generate Fonctional Simulation Netlist** » puis par la fin sélectionner **Processing** → **Start Compilation**.

### 4.3.4 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.

— **Affectation des pins :**

Afin de choisir quelle broche physique du circuit doit être connectée, lancer l'outil d'assignement de pins par : **Assignements** → **Assignements Editor,** dans la fenêtre qui apparait, il est possible de choisir différents types d'assignement. Dans notre cas, nous sélectionnerons dans la partie **Category** l'onglet **Pin**.

Dans le menu de gauche, prendre garde à valider le bouton permettant de faire afficher toutes les pins et bien valider le bouton ne permettant l'affichage que des pins utilisées. Afin de réaliser l'affectation, on double clique dans la colonne **Location** au niveau de la pin voulue de manière à faire apparaitre un menu déroulant où sont répertoriées les broches disponibles du circuit. La liste des broches utilisables dans notre projet est données en « **Annexe**2 ». On notera que certaines broches sont réservées pour des horloges, des reset,... ou sont directement câbles sur des périphériques tels que les afficheurs, les boutons poussoirs,...etc.

### 4.3.5 Configurer le composant au programme

Pour configurer le composant afin d'obtenir le fonctionnement décrit par le projet, il faut impérativement que :

— La maquette DE2 soit reliée à l'ordinateur par un câble USB.
— Elle soit sous tension (allumer le bouton rouge).
— Le commutateur RUN/PROG soit positionné sur RUN.

configurer le composant de la maquette DE2, on sélectionne l'icône de programmation ou bien **Tools→ Programmer.** La fenêtre de configuration suivante apparait.

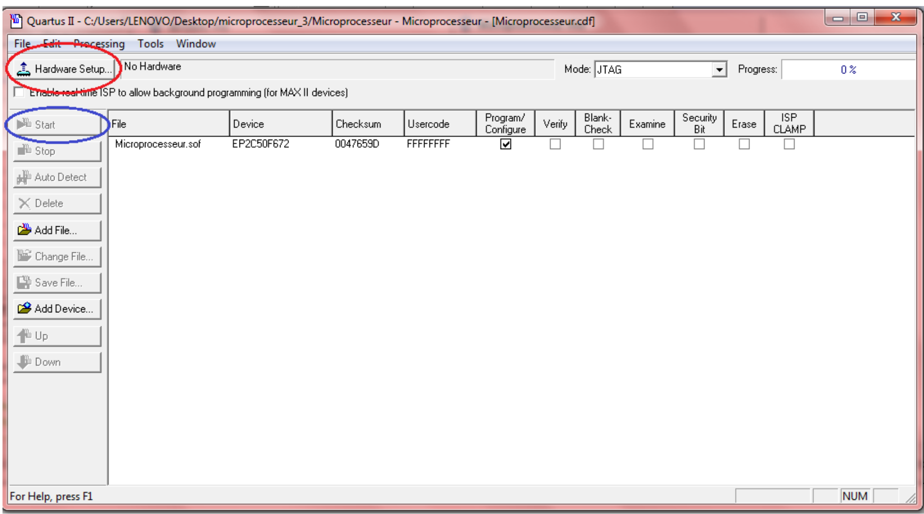



FIGURE 4.10 – Fenêtre de programmation.

Cliquer sur **Hardware Setup** (encerclé en rouge), la fenêtre suivante apparait, on sélectionne **USB-Blaster** dans le menu **Currently Selected Hardware.**

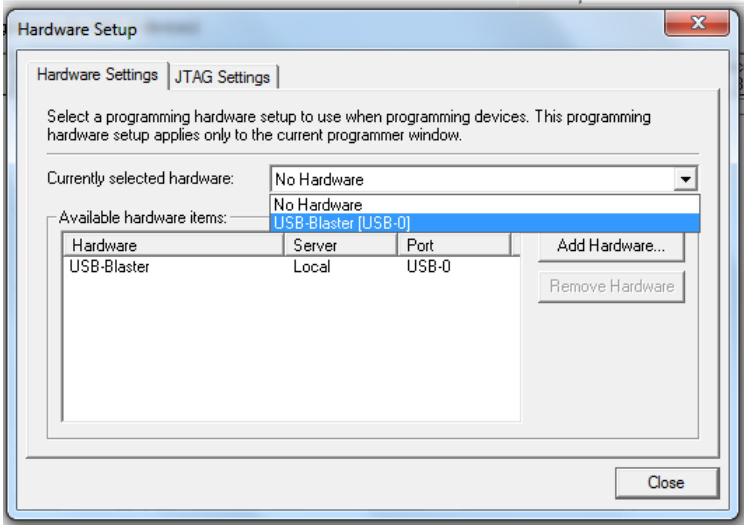



FIGURE 4.11 – Fenêtre de Hardware Setup.

Enfin pour lancer la configuration du circuit de la maquette DE2, cliquer sur **Start** (encerclé en bleu dans la **figure 4.10**).

## 4.4 Test et validation des résultats

### 4.4.1 Première version

Le programme exécuté en RAM est :

```
----------------------------------------1ére Opération
0=> "00000000" ; -- Addition
1=> "00000101" ; -- Donnée (5)
2=> "00000100" ; -- Donnée (4)
3=> "XXXXXXXX" ; -- Espace libre
----------------------------------------2éme Opération
4=> "00000100" ; -- Soustraction
5=> "00000111" ; -- Donnée (7)
6=> "00000110" ; -- Donnée (6)
7=> "XXXXXXXX" ;
----------------------------------------3éme Opération
8=> "00001000" ; -- AND
9=> "00001100" ; -- Donnée (12)
10=> "00001001" ; -- Donnée (9)
11=> "XXXXXXXX" ;
----------------------------------------4éme Opération
12=> "00001100" ; -- OR
13=> "00001010" ; -- Donnée (10)
14=> "00001011" ; -- Donnée (11)
15=> "XXXXXXXX" ;
Others=> "00000000" ;
```

#### 4.4.1.1 Résultats de simulation

Les résultats de simulation de la première version sont illustrés sur la figure (4.12).

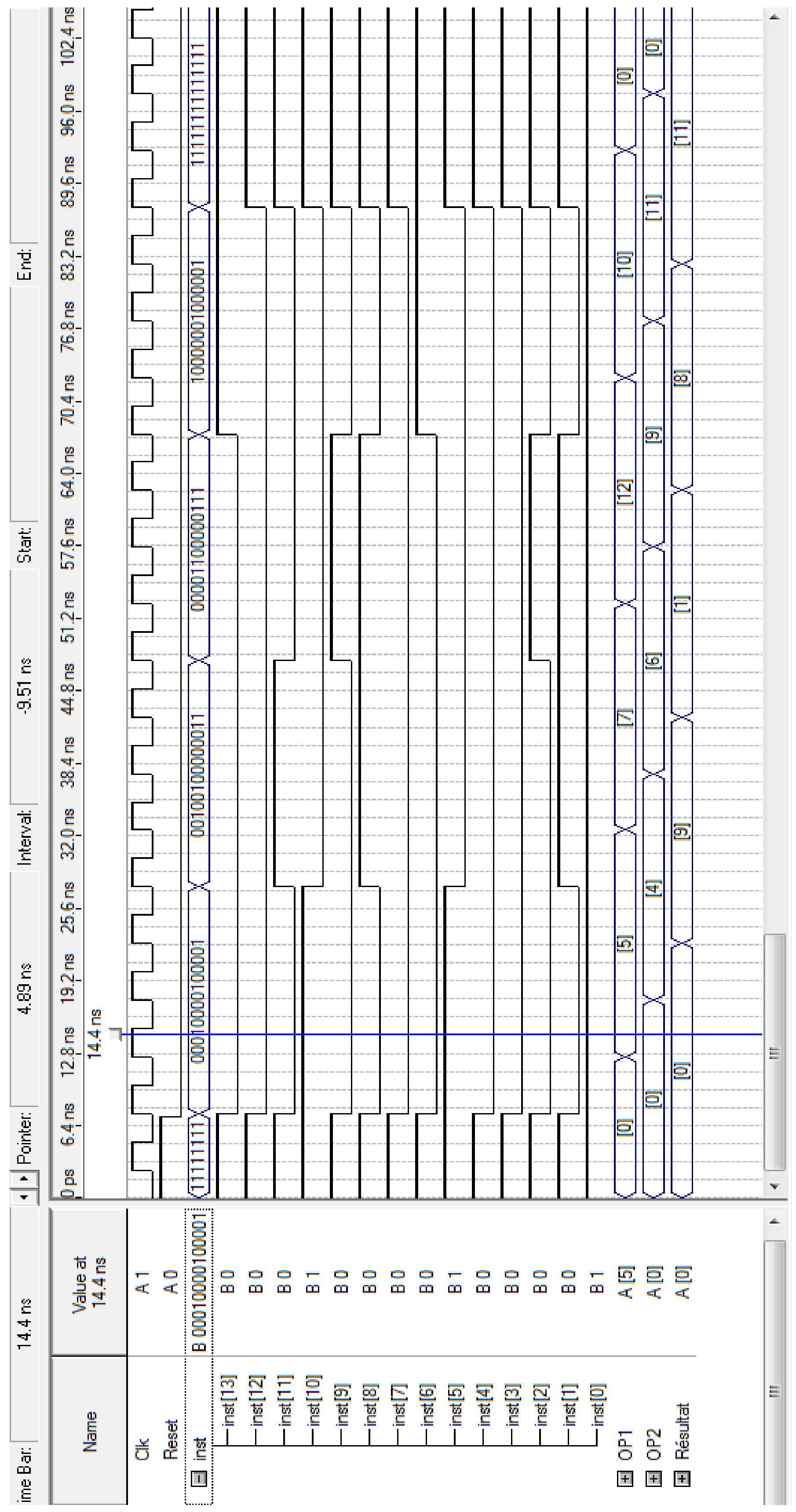



FIGURE 4.12 – Résultats de simulation de la première version.

— **Interprétation des résultats de simulation :**
les résultats de simulation obtenus sont subdivisés en deux parties, la première est la partie détaillée du bus d'instruction « **inst** » codé sur 14bits telle que le montre la figure suivante.

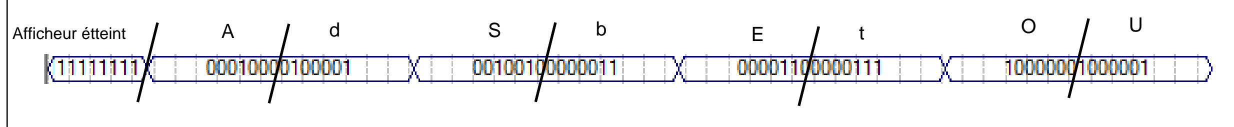



FIGURE 4.13 – Codes des instructions (version 1) .

Sachant que les instructions "ET" et "OU" effectuent des calculs bit a bit comme le montre la deuxième partie, qui sont les données codées en ASCII :

(5 + 4 = 9), ( 7 - 6 = 1), (12 AND 9 = 8), (10 OR 11=11).

### 4.4.1.2 Résultats pratiques

Après implémentation des résultats de simulation (figure 4.12) sur la cible FPGA, on a eu les quatre images illustrées sur la figure suivante.

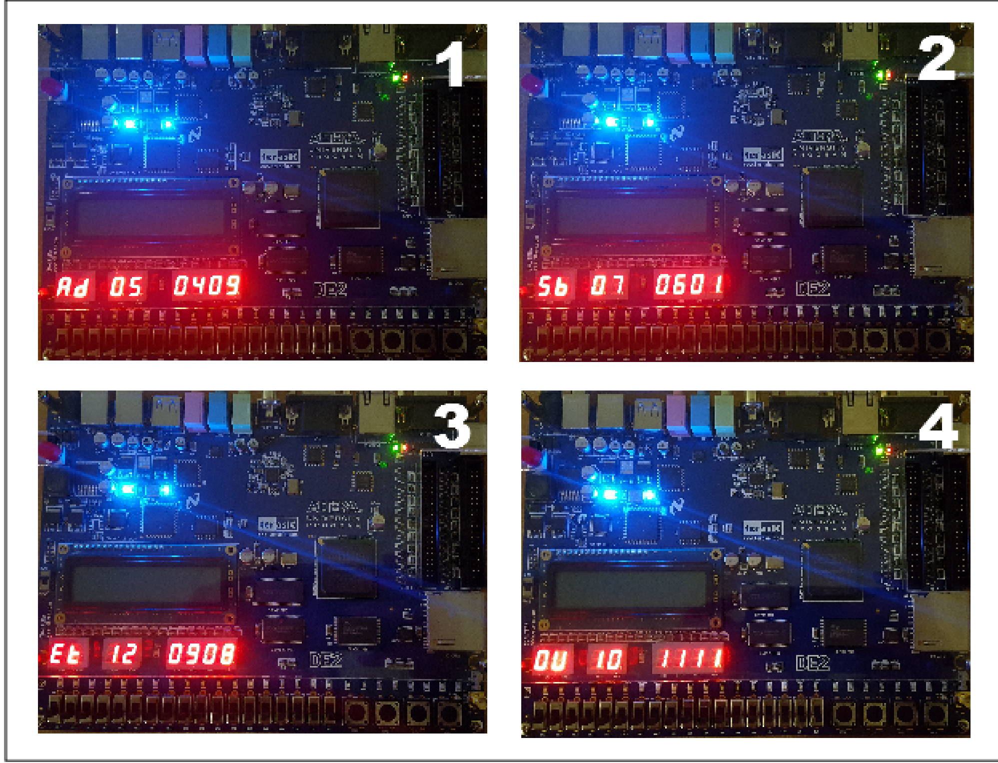



FIGURE 4.14 – Résultats pratique de la première version.

L'organisation des afficheurs 7-segment est comme suit :
- HEX6 et HEX7 affiche le vecteur « inst ».
- HEX4 et HEX5 affiche le vecteur « OP1 ».
- HEX2 et HEX3 affiche le vecteur « OP2 ».
- HEX0 et HEX1 affiche le vecteur « Résultat ».

## 4.4.2 Deuxième version

Le programme exécuté en RAM est :

```
-----------------------------------------1ére Opération
0=> "00001010" ; --load
1=> "00000110" ; --donnée 6
2=> "10000000" ; --addition
3=> "00000111" ; --donnée 7
4=> "00000001" ; --soustraction
5=> "00001000" ; --donnée 8
6=> "00001011" ; --store
7=> "XXXXXXXX" ; --champs libre
-----------------------------------------2éme Opération
8=> "00001010" ; --load
9=> "00001001" ; --donnée 9
10=> "00000010" ; --AND
11=> "00001100" ; --donnee 12
12=> "00000011" ; --OR
13=> "00001101" ; --donnee 13
14=> "00001011" ; --store
15=> "XXXXXXXX" ; --champs libre 13
Others=> "00000000" ;
```

### 4.4.2.1 Résultats de simulation

Les résultats de simulation de la deuxième version sont illustrés sur la figure 4.15.

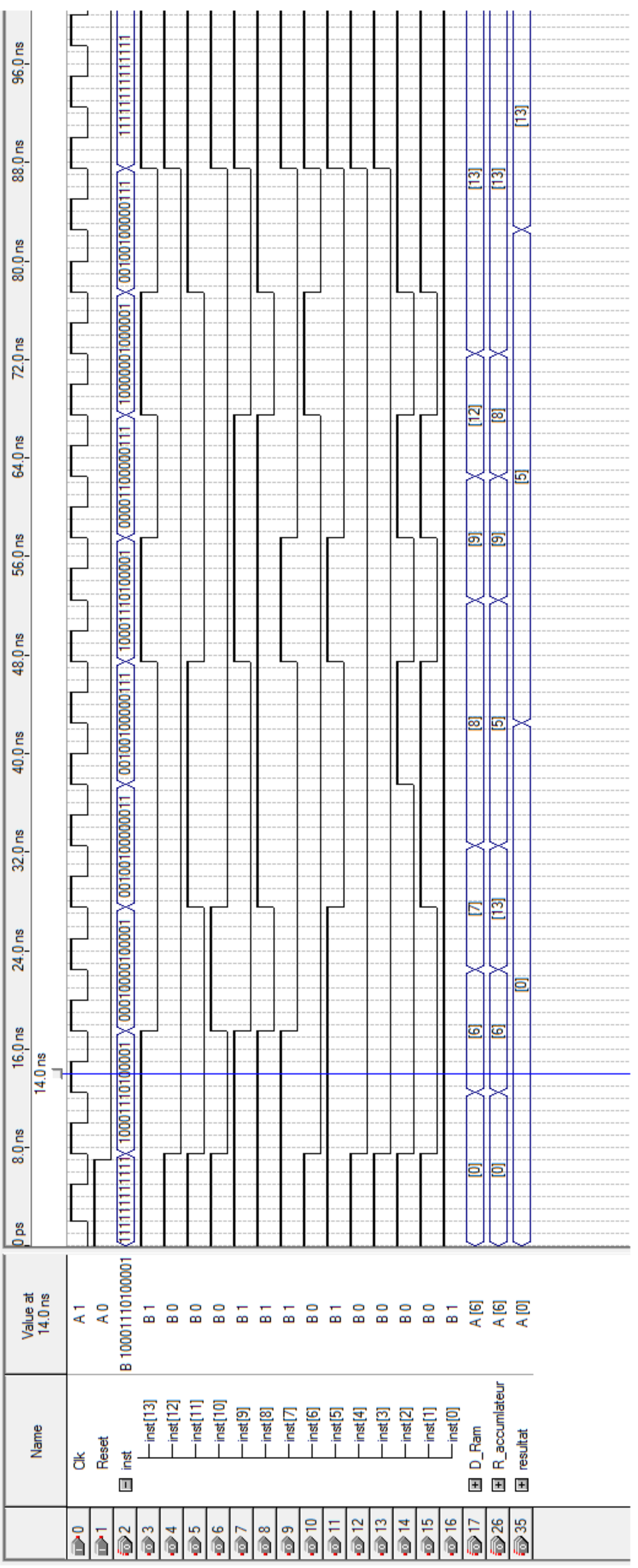



FIGURE 4.15 – Résultats de simulation de la deuxième version.

— **Interprétation des résultats de simulation :**

D'après la figure de simulation (figure 4.15), on a obtenu quatre (4) vecteurs tels que :

- le premier indique le vecteur d'instruction codé sur 14 bit, comme expliqué dans la figure suivante.

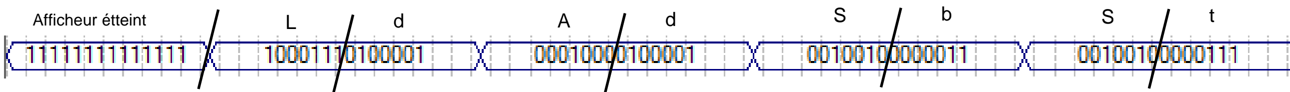



FIGURE 4.16 – Codes des instructions (version 2) .

- Les 3 vecteurs implimentationrestant (« **D_Ram** », « **R_accumulateur** » et « **resultat** ») représentent respectivement, les données en RAM, les données stockées en registre accumulateur et les résultat finals.

### 4.4.2.2 Résultats pratiques

Après implémentation des résultats de simulation (figure 4.15) sur la cible FPGA, les deux opérations effectuées sont illustrée comme suit.

— **Résultat de la première opération :**

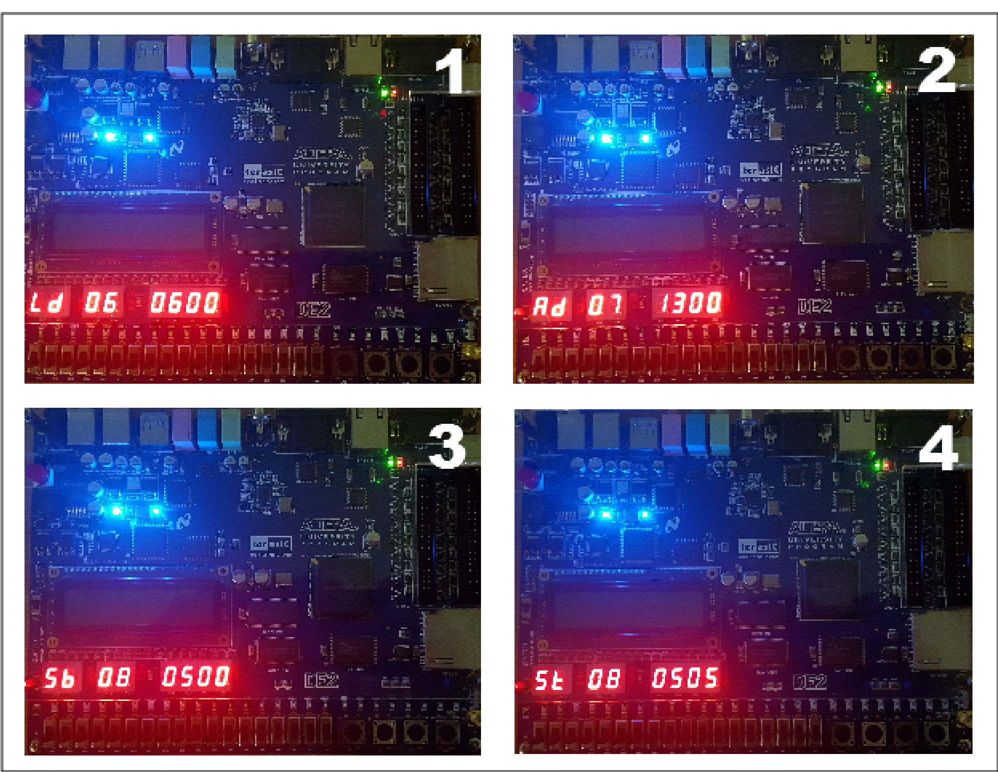



FIGURE 4.17 – Résultats pratique de la deuxième version (première opération).

— **Résultat de la deuxième opération :**

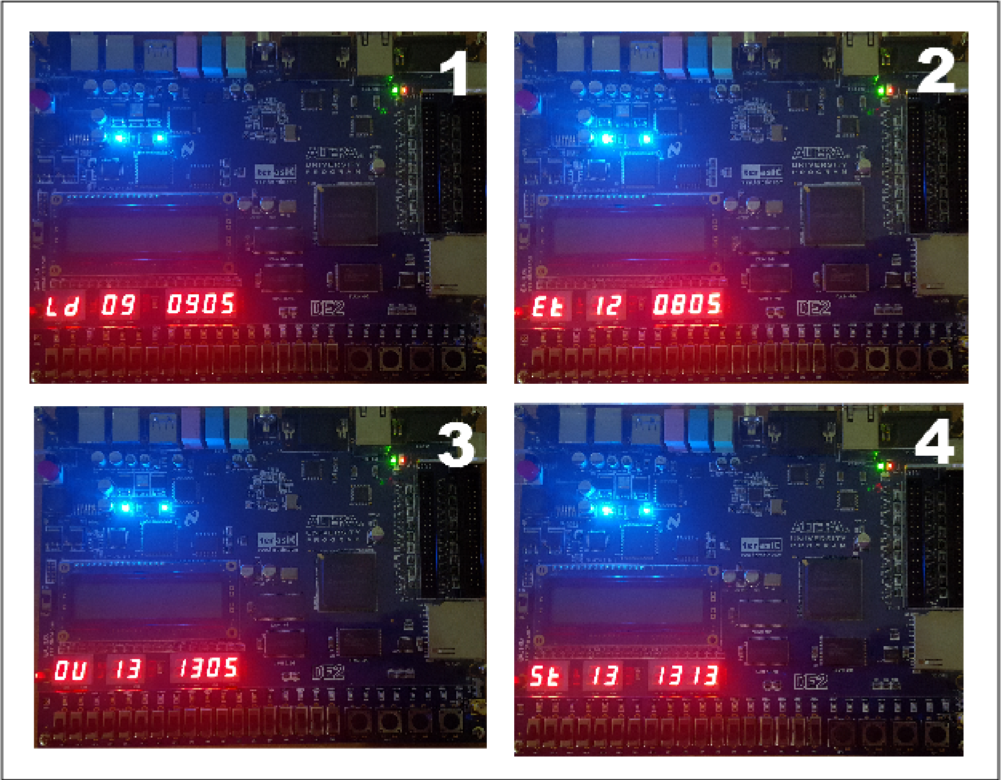



FIGURE 4.18 – Résultats pratique de la deuxième version (deuxième opération).

L'organisation des afficheurs 7-segment est comme suit :
- HEX6 et HEX7 affiche le vecteur « inst ».
- HEX4 et HEX5 affiche le vecteur « D_Ram ».
- HEX2 et HEX3 affiche le vecteur « R_accumulateur ».
- HEX0 et HEX1 affiche le vecteur « resultat ».

### 4.4.3 Troisième version

Le programme exécuté en RAM est :

```
-------------------------------------------------------------------Partie code
-----------------------------------------------1ére opération
0=> "00001010" ;--load
1=> "00010100" ;--Adressee 20
2=> "10000000" ;--inst +
3=> "00010101" ;--Adressee 21
4=> "00000001" ;--inst -
5=> "00011001" ;--Adressee 25
6=> "00001011" ;--store
7=> "00011010" ;--Adressee 26
----------------------------------------------2éme opération
8=> "00001010" ;--load
9=> "00010100" ;--Adressee 20
10=> "00000010" ;--inst and
11=> "00010110" ;--Adressee 22
12=> "00001011" ;--store
13=> "00011100" ;--Adressee 28
----------------------------------------------------------------Partie Données
20=> "00000011" ;-- donnee 3
21=> "00001000" ;--donnee 8
22=> "00000001" ;-- donnee 1
25=> "00000101" ;-- donnee 5
26=> "XXXXXXXX" ;--donnee 6
28=> "XXXXXXXX" ;--donnee 1
Others=> "00000000" ;
```

#### 4.4.3.1 Résultats de simulation

Les résultats de simulation de la troisième version sont illustrés sur la figure 4.19.

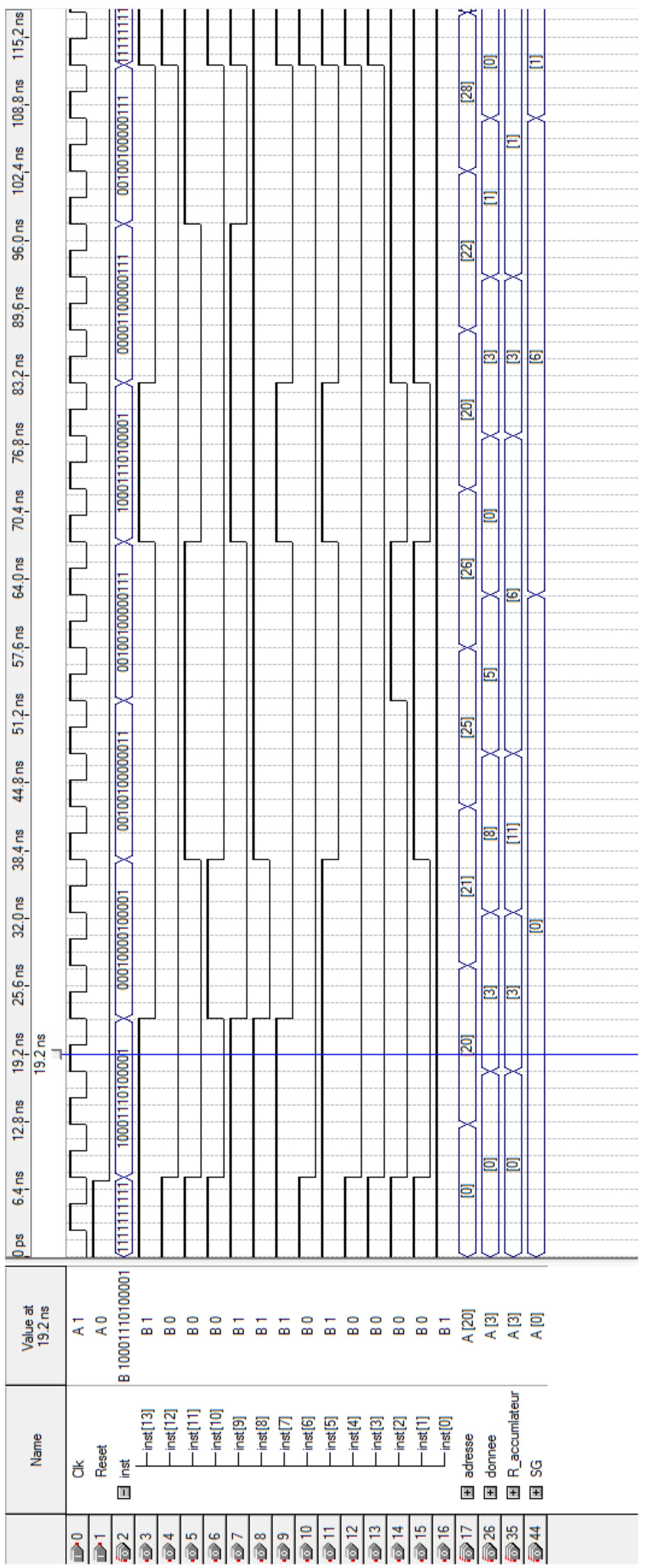



FIGURE 4.19 – Résultats de simulation de la troisième version.

— **Interprétation des résultats de simulation :**
D'après les résultats obtenus après simulation (figure 4.19), le vecteur "inst" est le même que le vecteur "inst" précédent réalisé dans dans l'architecture (2).Il est noté que, la seule différence se résume à la localisation directe de l'opérande dans une ligne mémoire pour la deuxième version. Quant à la dernière version, l'utilisation du bus d'adresse contenant l'opérande à exécuter.

#### 4.4.3.2 Résultats pratiques

Après implémentation des résultats de simulation (figure 4.19) sur la cible FPGA, les deux opérations effectuées sont illustrée comme suit.

— **Résultat de la première opération :**

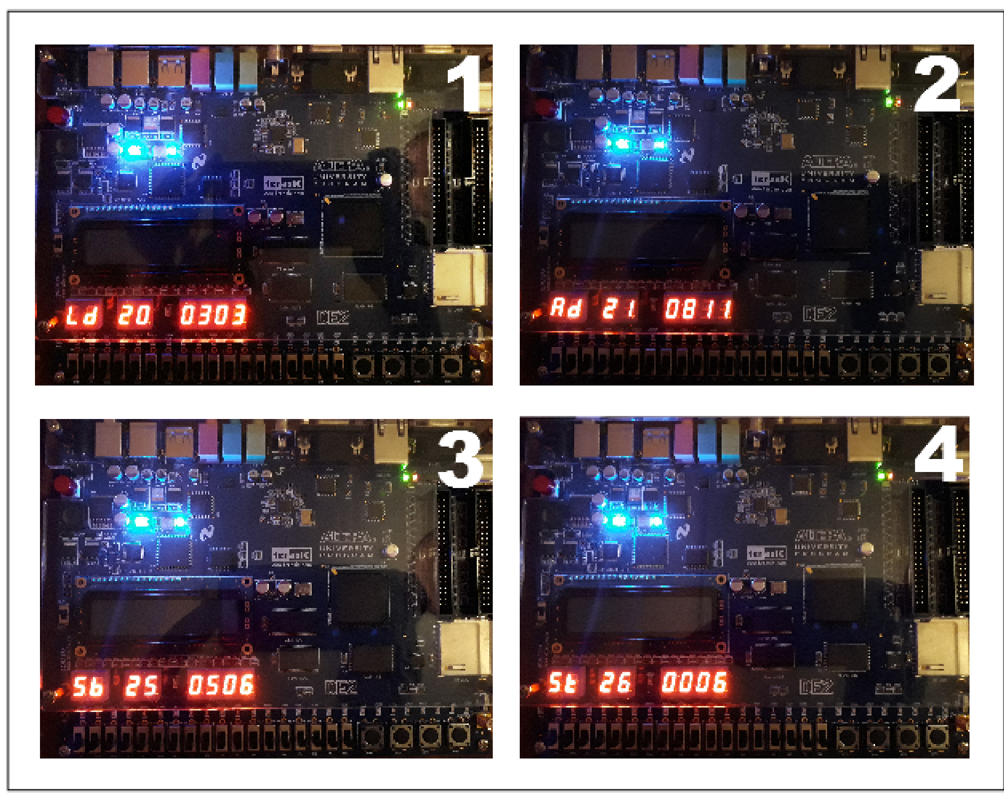



FIGURE 4.20 – Résultats pratique de la troisième version (première opération).

— **Résultat de la deuxième opération :**

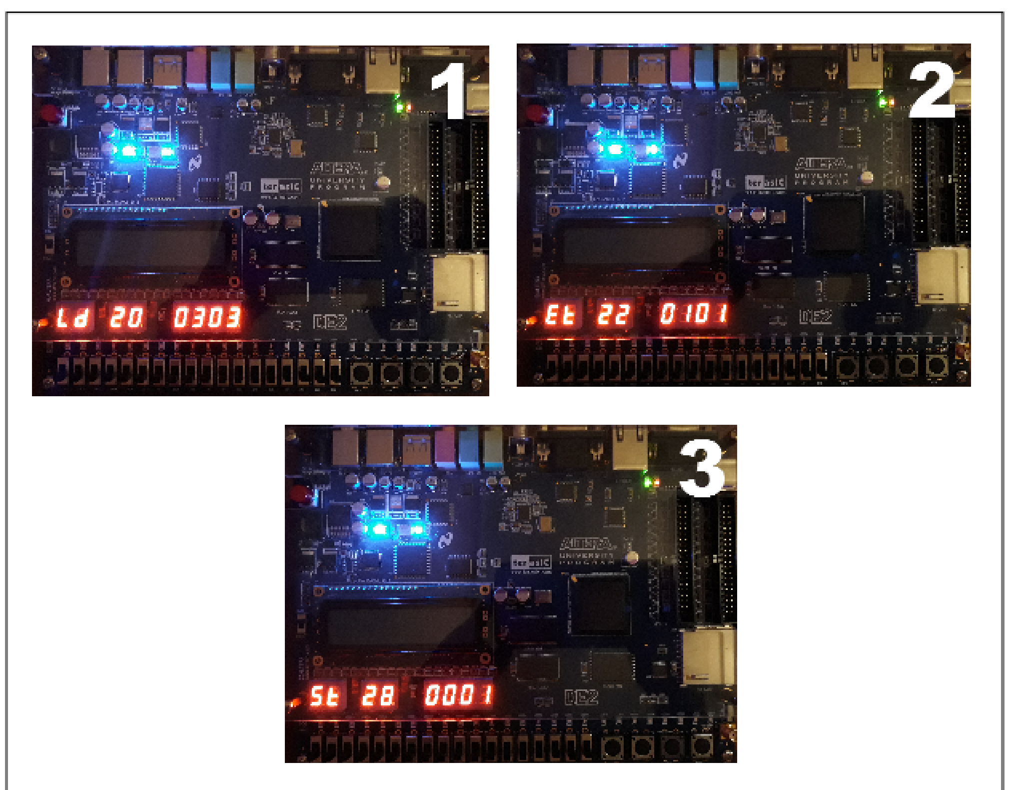



FIGURE 4.21 – Résultats pratique de la troisième version (deuxième opération).

L'organisation des afficheurs 7-segment est comme suit :
- HEX6 et HEX7 affiche le vecteur « inst ».
- HEX4 et HEX5 affiche le vecteur « adresse ».
- HEX2 et HEX3 affiche le vecteur « donnee ».
- HEX0 et HEX1 affiche le vecteur « R_accumulateur ».

## 4.5 Conclusion

Dans ce chapitre, nous avons dans un premier temps, introduit la carte de développement DE2 et ses différents composants. Nous avons, par la suite présenté, le logiciel de développement QUARTUS II. Par la suite, les différents résultats obtenus pour notre application sont présentés et discutés.

# Conclusion générale

Le travail présenté dans ce mémoire porte sur la réalisation d'un microprocesseur 8 bits avec nombre d'instructions réduit. Ce dernier, est interprété par trois versions :

— Première version : consiste à réaliser le processeur le plus simple, il effectue des opérations linéaires non enchainées.
— Deuxième version : cette version a pour objectif de faire une chaine de calculs. Pour se faire, on a ajouté un accumulateur à la version précédente.
— Troisième version : elle résume les deux versions précédentes, son objectif est de faire sauter des lignes dans la mémoire. Pour se faire on a ajouté un registre adresse.

L'objectif était de relier la simulation sur l'outil de développement "Quartus II" ainsi faite par l'implémentation des différentes versions de ce microprocesseur sur un FPGA (carte DE2).

Ainsi, nous avons défini quelques notions sur les circuits logiques programmables en ses technologies de mémorisation ainsi que la présentation des FPGA, architectures internes et la classification de ce circuit.

Puis, nous avons présenté les principes de base sur lesquels repose le développement d'un système à l'aide d'une description avec le langage VHDL, en le définissant, en explicitant ses différentes structures aussi ses avantages.

La conception de ce microprocesseur, nous a permis de suivre trois différentes architectures comme déjà citées. Leurs implémentations sur la carte DE2 est une mise en œuvre pratique caractérisée par leurs description, traduction au langage VHDL, assignements des différents composants puis la visualisation du résultat final obtenu par la carte.

La simulation ainsi faite sur l'outil de développement "Quartus II" concorde parfaitement au résultats obtenus après implémentation de ces différentes architectures sur l'FPGA de la carte DE2.

En perspective, il sera intéressant de réaliser :

— Architecture de microprocesseur de type Harvard ;

— Un microcontrôleur avec un différents périphériques comme l'UART, le chien de garde et les timers.

# Bibliographie


[1] **Arezki CHIBANI :** « Conception d'un contrôleur d'étage de puissance par FPGA », mémoire de magister. Université Mouloud Mammeri de Tizi-Ouzou.

[2] **Djouher SNAOUI :** « Commande numérique en force à base de la carte FPGA d'une architecture de télé-opération à un seul degrés de liberté », mémoire de magister. Université Mouloud Mammeri de Tizi-Ouzou. Département automatique, 2015.

[3] **C.ALEXANDRE :** « Polycopié de cours et de travaux pratiques (Circuits logiques programmables) », Version du 02/10/2013. Conservatoire National des Arts et Métiers.

[4] **Etienne MESSERLI :** « Manuel VHDL synthèse et simulation », Version partielle septembre 2007. Haute École d'Ingénierie et de Gestion du Canton de Vaud.

[5] **Zahir AIT OUALI :** « Application des FPGA à la commande d'un moteur asynchrone », mémoire de magister. Université Mouloud Mammeri de Tizi-Ouzou. Automatique des systèmes continus et productiques, 2011.

[6] **Dominique HOUZET** et **Ludovic BARRABDON** « Conception de circuits en VHDL et VHDL-AMS ». Toulouse, 2006.

[7] **Jacques WABER, Sébastien MOUTAULT** et **Maurice MEAUDRE** « Le langage VHDL du langage au circuits, du circuits au langage », 3éme édition, Paris, Dunod, 2007.

[8] **Michel AUMIAUX** « Initiation au langage VHDL », 2éme édition.

[9] **Rabah RAHEM** « Implémentation d'un algorithme de commande d'un dispositif électromagnétique sur FPGA », mémoire de master. Université Mouloud Mammeri de Tizi-Ouzou.

[10] **V.TOURTCHINE**et **M.IZOUIN**« Modélisation des systèmes numériques dans l'environnement XILINX ISE 13.2 (VHDL) », mémoire de master. Université M'hamed Bougara de BOUMERDES.

[11] **Hamza LAHLOU** « Système de transmission de données sur FPGA », mémoire de master. Université Mouloud Mammeri de Tizi-Ouzou.

[12] **Souad KEBAILI** « », mémoire de master. Université Mouloud Mammeri de Tizi-Ouzou.

[13] **Redouane KARA.** « Système embarqué », cours de troisième année. Univesité de Tizi-Ouzou.

[14] **Jean PROVOST** « Électronique Numérique Intégrée », polycopie de cours Telecom Paris, 2005.

[15] **André DUPONT**« Notice de prise en main du logiciel Quartus II », Université Polytech, Montpellier.

# Annexes

## Annexe 1 :

- Code VHDL de la première version :
  - Programme principal :

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Entity Microprocesseur is
Port ( Clk,Reset: in std_logic;
SG:out std_logic_vector (13 downto 0));
End Microprocesseur;
Architecture AMicroprocesseur
of Microprocesseur is
Signal EN1, EN2, EN3, EN4: std_logic;
Signal SC,SR1,SR2,SR3,SU,SM,SROM:
std_logic_vector (7 downto 0);
Component Compteur
Port( Clk,Reset: in std_logic;
OC:out std_logic_vector (7 downto 0));
End Component;
Component Registre
Port( I1R: in std_logic_vector(7 downto 0);
I2R,Clk,Reset: in std_logic;
O1R: out std_logic_vector(7 downto 0));
End Component;
Component Machine
Port( Clk,Reset: in std_logic;
O1M,O2M,O3M,O4M: out std_logic);
End Component;
Component UAL
Port( I1U,I2U,I3U: in
std_logic_vector(7 downto 0);
OU: out std_logic_vector(7 downto 0));
End Component;
```

```
Component ROM Port( I2ROM,Reset:
in std_logic;
I1ROM: in std_logic_vector(7 downto 0);
OROM: out std_logic_vector(7 downto 0));
End Component;
Component Afficheur
Port( I: instd_logic_vector(7 downto 0);
O: out std_logic_vector(13 downto 0));
End Component;
Begin
C1: Compteur PortMap(Clk=>Clk,Reset=>Reset,OC=>SC);
C2: Registre PortMap(Reset=>Reset,I1R=>SROM,I2R=>EN1,Clk=>Clk,O1R=>SR1);
C3: Registre Port Map(Reset=>Reset,I1R=>SROM,I2R=>EN2, Clk=>Clk,O1R=>SR2);
C4: Registre PortMap(Reset=>Reset,I1R=>SROM,I2R=>EN3, Clk=>Clk,O1R=>SR3);
C5: Registre PortMap(Reset=>Reset,I1R=>SU,I2R=>EN4, Clk=>Clk,O1R=>SM);
C6: Machine PortMap(Clk=>Clk,Reset=>Reset,O1M=>EN1
,O2M=>EN2,O3M=>EN3,O4M=>EN4);
C7: UAL Port Map(I1U=>SR1,I2U=>SR2,I3U=>SR3,OU=>SU);
C8: ROM PortMap(I1ROM=>SC,Reset=>Reset,I2ROM=>Clk,OROM=>SROM);
C9:Afficheur Port Map(I=>SM,O=>SG);
End AMicroprocesseur;
```

— Code VHDL d'un compteur :

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Use IEEE.numeric_std.all;
Use IEEE.std_logic_arith.all;
Entity Compteur is
Port( Clk, Reset: in std_logic;
OC: out std_logic_vector(7 downto 0));
EndCompteur;
Architecture ACompteur of Compteur is
Signal Count: integer:=0;
Begin Process(Clk,Reset)
Begin if (Reset='1') then Count<=0;
elsif (Clk'eventand Clk='1') then
if(Count=255) then Count<=0;
else Count<=Count+1;
End if;
```

```
End if;
OC<=Conv_std_logic_vector(Count,8);
End Process;
End ACompteur;
```

— Code VHDL d'un registre :

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Entity Registre is
Port(CLK, Reset, I2R:in std_logic;
I1R: in std_logic_vector(7 downto 0);
O1R: out std_logic_vector(7 downto 0));
End Registre;
Architecture ARegistre of Registre is
Begin Process (CLK,Reset)
Begin if(Reset='1') then O1R<="00000000";
elsif (CLK'event andCLK='1') then
if(I2R='1') then O1R<=I1R;
End if;
End if;
End Process;
End ARegistre;
```

— Code VHDL d'une machine :

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Entity Machine is
Port( Clk,Reset: in std_logic;
O1M,O2M,O3M,O4M: out std_logic);
End Machine;
Architecture AMachineof Machine is
Type Etat is (q0,q1,q2,q3);
Signal State,NextState:Etat;
Begin
Process(State)
Begin Case State is
When q0 =>
O1M<='1';
```

```
O2M<='0';
O3M<='0';
O4M<='0';
NextState<=q1;
When q1 =>
O1M<='0';
O2M<='1';
O3M<='0';
O4M<='0';
NextState<=q2;
When q2 =>
O1M<='0';
O2M<='0';
O3M<='1';
O4M<='0';
NextState<=q3;
When q3 =>
O1M<='0';
O2M<='0';
O3M<='0';
O4M<='1';
NextState<=q0;
End Case;
End Process;
Process(Clk,Reset)
Begin if Reset='1' then State<=q0;
elsif (Clk'event and Clk='1')
then State<=NextState;
End if;
End Process;
End AMachine;
```

— Code VHDL d'une unité arithmétique et logique (UAL) :

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Use IEEE.Numeric_std.all;
Entity UAL is
Port( I1U,I2U,I3U: in std_logic_vector(7 downto 0);
OU: out std_logic_vector(7 downto 0));
```

```
End UAL;
Architecture AUAL of UAL is
Begin Process(I1U)
Begin
Case I1U is
When "00000000" =>
OU<=std_logic_vector(signed(I2U) + signed(I3U));
When "00000100" =>
OU<=std_logic_vector(signed(I2U) {signed(I3U));
When "00001000" =>
OU<=I2U AND I3U;
When "00001100" =>
OU<=I2U OR I3U;
When others =>NULL;
End Case;
End Process;
End AUAL;
```

# Annexe 2 :

— Brochage des boutons poussoirs :

| Nom du bouton | Broche associée du circuit FPGA |
|---|---|
| Key 0 | PIN_G26 |
| Key 1 | PIN_N23 |
| Key 2 | PIN_P23 |
| Key 3 | PIN_W26 |

TABLE 4.1 – Brochage des boutons poussoirs

— Brochage des afficheurs 7-segments :

| Nom du bouton | Broche associée du circuit FPGA |
| --- | --- |
| HEX0[0] | PIN_AF10 |
| HEX0[1] | PIN_AB12 |
| HEX0[2] | PIN_AC12 |
| HEX0[3] | PIN_AD11 |
| HEX0[4] | PIN_AE11 |
| HEX0[5] | PIN_V14 |
| HEX0[6] | PIN_V13 |
| HEX1[0] | PIN_V20 |
| HEX1[1] | PIN_V21 |
| HEX1[2] | PIN_W21 |
| HEX1[3] | PIN_Y22 |
| HEX1[4] | PIN_AA24 |
| HEX1[5] | PIN_AA23 |
| HEX1[6] | PIN_AB24 |
| HEX2[0] | PIN_AB23 |
| HEX2[1] | PIN_V22 |
| HEX2[2] | PIN_AC25 |
| HEX2[3] | PIN_AC26 |
| HEX2[4] | PIN_AB26 |
| HEX2[5] | PIN_AB26 |
| HEX2[6] | PIN_Y24 |
| HEX3[0] | PIN_Y23 |
| HEX3[1] | PIN_AA25 |
| HEX3[2] | PIN_AA26 |
| HEX3[3] | PIN_Y26 |
| HEX3[4] | PIN_Y25 |
| HEX3[5] | PIN_U22 |
| HEX3[6] | PIN_W24 |
| HEX4[0] | PIN_U9 |
| HEX4[1] | PIN_U1 |
| HEX4[2] | PIN_U2 |
| HEX4[3] | PIN_T4 |
| HEX4[4] | PIN_R7 |
| HEX4[5] | PIN_R6 |
| HEX4[6] | PIN_T3 |
| HEX5[0] | PIN_T2 |
| HEX5[1] | PIN_P6 |
| HEX5[2] | PIN_P7 |
| HEX5[3] | PIN_T9 |
| HEX5[4] | PIN_R5 |
| HEX5[5] | PIN_R4 |
| HEX5[6] | PIN_R3 |

TABLE 4.2 – Brochage des afficheurs 7-segments

| Nom du bouton | Broche associée du circuit FPGA |
|---|---|
| HEX6[0] | PIN_R2 |
| HEX6[1] | PIN_P4 |
| HEX6[2] | PIN_P3 |
| HEX6[3] | PIN_M2 |
| HEX6[4] | PIN_M3 |
| HEX6[5] | PIN_M5 |
| HEX6[6] | PIN_M4 |
| HEX7[0] | PIN_L3 |
| HEX7[1] | PIN_L2 |
| HEX7[2] | PIN_L9 |
| HEX7[3] | PIN_L6 |
| HEX7[4] | PIN_L7 |
| HEX7[5] | PIN_P9 |
| HEX7[6] | PIN_N9 |

TABLE 4.3 – Brochage des afficheurs 7-segments

# Annexe 3 :

— Mots réservés du langage VHDL :

| abs | disconnect | map | rem |
|---|---|---|---|
| access | downto | mod | report |
| after | else | nand | return |
| alias | elsif | new | select |
| all | end | next | severity |
| and | entity | nor | signal |
| architecture | exit | not | subtype |
| array | file | null | then |
| assert | for | of | to |
| attribute | function | on | transport |
| begin | generate | open | type |
| block | generic | or | units |
| body | guarded | others | until |
| buffer | if | out | use |
| Bus | in | package | variable |
| case | inout | port | wait |
| component | is | procedure | when |
| configuration | label | process | while |
| constant | library | range | |
| with | linkage | record | |
| Xor | loop | register | |

TABLE 4.4 – Mots résérvés du langage VHDL

# Résumé

L'objectifde ce travail est de réaliser un microprocesseur 8 bits, puis l'implimenté sur une cible FPGA (carte DE2). Pour se faire, un langage de description matériels VHDL a été utilisé.

Les résultats de simulation et expérimentaux obtenus, indique le bon fonctionnement du système mis en oeuvre

## Mots clefs :

Circuits Logiques Programmables, microprocesseur, VHDL, FPGA, carte DE2, Quartus II.