UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN GÉNIE ÉLECTRIQUE

PAR
SALIM BOUKAKA

RÉALISATION D'UNE BIBLIOTHÈQUE DE LOIS DE COMMANDE ADAPTATIVE POUR MSAP

SEPTEMBRE 2015

Résumé

Un contrôleur performant d'une machine à courant alternatif (CA) nécessite généralement des opérations complexes et un long temps de calcul, ce qui rend la tâche d'implémentation plus complexe. L'implémentation FPGA (*Field Programmable Gate Array*) de ces contrôleurs, y compris les contrôleurs intelligents, a beaucoup d'avantages : reprogrammable, outil logiciel pratique, rendement élevé, densité d'intégration très élevée. Ce travail présente l'implémentation sur FPGA de plusieurs lois de commande à base d'intelligence artificielle d'une machine synchrone à aimant permanent (*MSAP*) en utilisant le langage VHDL.

La machine synchrone à aimant permanent est connue par son modèle complexe et non linéaire. Les contrôleurs à base d'intelligence artificielle représentent une solution efficace face aux problèmes associés au modèle mathématique de la *MSAP*. Les algorithmes de commande proposés dans ce travail ont été retravaillés et restructurés dans le but d'en avoir un schéma d'implémentation simple et bien structuré. La modélisation et la simulation de ce schéma sont faites en langage VHDL.

La validation des schémas a été effectuée en utilisant une cosimulation entre ModelSim et Simulink™/Matlab®. Les résultats de simulation et de synthèse montrent que les schémas non seulement conservent les avantages des modélisations en virgule flottante sur Simulink™/Matlab®, mais aussi donnent des schémas structurels simples qui consomment peu d'espace lors de l'implémentation, donc ça permet d'implémenter

la boucle de régulation et le générateur SVPWM (*Space-Vector Pulse Width Modulation*) sur la même carte laissant de l'espace pour d'autres fonctions.

Abstract

High performance motor controllers usually require complex operations and long computation time, which makes the implementation task more complex. FPGA (Field Programmable Gate Array) implementation of these controllers, including intelligent controllers, has many advantages: re-programmable, convenient software tool, high efficiency, very high significant integration density. This work presents implementation on FPGA of many Intelligent Adaptive Controls of Permanent Magnet Synchronous Machines (PMSMs) using Hardware Description Language VHDL.

The permanent magnet synchronous machine is known for its complex and non-linear model. Artificial intelligence based controllers are an effective solution to face problems associated with the PMSM's mathematical model. The control algorithms proposed in this work have been revised and restructured in order to have a simple and well-structured implementation scheme. Modeling and simulations are made in VHDL.

The validation of the algorithm is performed using co-simulation between ModelSim and Simulink™/Matlab®. The simulation results show that the scheme not only preserves the advantages of the modeling on Simulink™/Matlab® with floating point, but has a simple structural scheme and consumes few resources during implementation, so it permits to implement the control law on the same board and leaving sufficient free resources for implementing additional functions.

Table des matières

Résumé ii

Abstract iv

Table des matières v

Liste des tableaux x

Liste des figures xi

Liste des abréviations xv

Liste des symboles xvii

Chapitre 1 - Introduction 1

1.1 Problématique 2

1.2 Objectifs 4

1.3 Méthodologie 5

1.4 Structure du mémoire 6

Chapitre 2 - Logique Floue 8

2.1 Introduction 8

2.1.1 Généralités 8

2.1.2 Exemple Introductif 9

2.2 Définitions 11

2.2.1 Univers de discours 11

2.2.2 Variable linguistique 11

2.2.3 Fonction d'appartenance 12

2.3 Opérateurs et normes 13

2.3.1 Opérateur NON 13

2.3.2 Opérateur ET 13

2.3.3 Opérateur OU 13

2.4 Unités d'un bloc logique flou 14

2.4.1 La fuzzification 15

2.4.2 Base de règles 15

2.4.3 Mécanisme d'inférence 16

2.4.4 La défuzzification 17

2.5 Conclusion 18

Chapitre 3 - Field Programmable Gate Array « FPGA » 19

3.1 Introduction 19

3.2 Histoire des FPGA 23

3.3 Principe de fonctionnement des FPGA 25

3.3.1 Une simple fonction programmable 26

3.3.2 Technologie de connexion à base fusibles 27

3.3.3 Technologie de connexion à base d'anti-fusibles 28

3.3.4 ROM 28

3.3.5 RAM 30

3.4 Intégration sur VHDL 31

3.4.1 Nombre entier positif 32

3.4.2 Nombre entier négatif 33

3.4.3 Complément à deux 34

3.4.4 Représentation des données numériques 34

3.4.5 Modélisations des opérations de base 35

3.5 Conclusion 44

Chapitre 4 - Systèmes d'entrainement électromécanique 45

4.1 Introdcution 45

4.2 Machines Synchrones à Aimants Permanents (MSAP) 46

4.2.1 Équations générales de la machine synchrone à aimant permanent 47

4.3 Onduleur triphasé à deux niveaux 50

4.3.1 Topologie d'un onduleur triphasé à deux niveaux 50

4.3.2 Technique SVPWM 52

4.3.3 Simulation et discussion 56

4.4 Approximation de la SVPWM 60

4.4.1 Algorithme de l'approximation 61

4.4.2 Simulations et résultats 63

4.4.3 Implémentation 68

4.5 Conclusion 72

Chapitre 5 - Réalisation d'une bibliothèque de lois de commande adaptative pour MSAP 73

5.1 Introduction 73

5.2 1ere Loi implémentée : *Adaptive Fuzzy Logic Control of Permanent Magnet Synchronous Machines with Nonlinear Friction* 74

5.2.1 Configuration du contrôleur flou adaptatif 75

5.2.2 Implémentation et simulation 80

5.3 2eme Loi: *PMSM control based on adaptive fuzzy logic and sliding mode [5.2]* 87

5.3.1 Configuration du Régulateur mode glissant 88

5.3.2 Implémentation et simulation 89

5.4 3eme Loi: *Adaptive Fuzzy Logic Control Structure of PMSMs [5.3]* 92

5.4.1 Implémentation et simulation 92

5.4.2 Interprétation des résultats de simulation 94

5.5 Conclusion 95

Chapitre 6 - Conclusion Générale 96

Références 99

Annexe A – Extraits des codes de simulation 106

Annexe B - Résultats de synthèse des lois de commande.....136

Annexe C - Publications et contributions.....144

Liste des tableaux

Tableau 3-1 Progression des FPGA 25

Tableau 4-1 Vecteurs de tensions pour chaque combinaison possible des interrupteurs..... 52

Tableau 4-2 Détection du secteur..... 56

Tableau 4-3 Résumé de l'utilisation des ressources pour l'implémentation de la SVPWM. 59

Tableau 4-4 Résumé de l'utilisation des ressources pour l'implémentation de l'approximation de la SVPWM. 72

Tableau 5-1 Table des règles d'inférence. 76

Tableau 5-2 Sommaire d'utilisation des ressources pour l'implémentation de la loi de commande..... 87

Tableau 5-3 Distribution des ressources. 87

Tableau 5-4 Comparaison des trois lois en termes de consommation. 95

Liste des figures

Figure 1.1 Processus de simulation Matlab/ModelSim/Matlab 6

Figure 2.1 Exemple de réglage de vitesse d'un véhicule selon la logique classique. 9

Figure 2.2 Exemple de réglage de vitesse d'un véhicule selon la logique floue. 10

Figure 2.3 Exemple d'une fonction sigmoïde. 12

Figure 2.4 Schéma des différentes opérations dans un bloc logique floue. 14

Figure 3.1 Architecture traditionnelle d'une plateforme FPGA à base de mailles [3.6]. 26

Figure 3.2 Exemple d'une simple fonction programmable [3.4]. 27

Figure 3.3 Exemple d'une connexion à base de fusibles [3.4] 28

Figure 3.4 Exemple d'une connexion à base d'anti-fusibles [3.4] 28

Figure 3.5 Cellule principale d'une mémoire à grille flottante et son schéma symbolique. 30

Figure 3.6 Exemple d'une mémoire RAM statique et dynamique 30

Figure 3.7 Représentation des données. 35

Figure 3.8 Bloc d'addition. 36

Figure 3.9 Prise en compte du débordement sur VHDL pour un additionneur. 37

Figure 3.10 Résultats d'un additionneur à 16 bits 38

Figure 3.11 Bloc de Multiplication. 39

Figure 3.12 Résultats d'un multiplieur à 8 bits. 40

Figure 3.13 Bloc de Soustraction. 40

Figure 3.14 Prise en compte du débordement sur VHDL pour un soustracteur. 41

Figure 3.15 Résultats d'un soustracteur à 16 bits. 41

Figure 3.16 Bloc de décalage. 42

Figure 3.17 Résultats d'un registre à 8 bits. 43

Figure 3.18 Bloc de troncation. 43

Figure 3.19 Résultats d'un troncateur à 16bits-à-8bits. 44

Figure 4.1 Croissance du marché des systèmes d'entrainement électromécanique (Million $/ Année) [4.3]. 45

Figure 4.2 Dispositif d'entrainement électromécanique typique. 46

Figure 4.3 Représentation de la machine synchrone à aimants permanents. 47

Figure 4.4 Circuit équivalent de la MSAP. 49

Figure 4.5 Onduleur triphasé à deux niveaux. 51

Figure 4.6 Vecteurs d'espace de tension. 53

Figure 4.7 Forme des Signaux PWM dans le secteur I. 55

Figure 4.8 RTL Schéma fonctionnel de l'onduleur sur Simulink. 57

Figure 4.9 Entrées/sorties du circuit SVPWM réalisé sur une FPGA (CLK_PWM=Ts) 57

Figure 4.10 Schéma du circuit SVPWM réalisé sur FPGA. 58

Figure 4.11 Tension de sortie Phase A 58

Figure 4.12 Signaux PWM pour trois périodes Ts 59

Figure 4.13 Représentation des harmoniques d'une phase. 59

Figure 4.14 Méthodologie utilisée pour identifier le modèle approximatif. 62

Figure 4.15 Relation entre le coefficient A et la tension V_{dc} 63

Figure 4.16 Tension de sortie Phase A 64

Figure 4.17 Représentation des harmoniques 65

Figure 4.18 Signaux PWM pour trois périodes T_s. 65

Figure 4.19 Tensions triphasées pour un système déséquilibré. 67

Figure 4.20 Simulation de l'approximation avec une loi de commande..... 68

Figure 4.21 Schéma du circuit du modèle approximatif réalisé sur FPGA. 69

Figure 4.22 Entrées/sorties du circuit SVPWM réalisé sur FPGA (CLK_ApproximationModel=Ts). 69

Figure 4.23 Tension de sortie Phase A..... 70

Figure 4.24 Représentation des harmoniques d'une phase. 70

Figure 4.25 Signaux PWM pour trois périodes Ts..... 71

Figure 5.1 Structure de contrôle. 75

Figure 5.2 Fonctions d'appartenance. 76

Figure 5.3 Structure de l'unité d'adaptation de la logique floue..... 79

Figure 5.4 Schéma du circuit Calcul du pourcentage d'appartenance réalisé sur un FPGA. 81

Figure 5.5 Schéma du circuit de la partie d'adaptation réalisé sur un FPGA. 82

Figure 5.6 Entrées/Sorties du circuit global de contrôle réalisé sur un FPGA. 82

Figure 5.7 Schéma du circuit FLC adaptatif réalisé sur un FPGA..... 83

Figure 5.8 Entrées/Sorties du circuit Générateur de sin/cos réalisé sur un FPGA..... 84

Figure 5.9 Sinus et cosinus à base de la série de Taylor. 84

Figure 5.10 Schéma du circuit Générateur SinCos réalisé sur un FPGA..... 85

Figure 5.11 Entrées/sorties du circuit SVPWM réalisé sur un FPGA. 86

Figure 5.12 Résultats de cosimulation. 86

Figure 5.13 Structure de contrôle de la commande LF-MG. 88

Figure 5.14 Entrées/sorties du circuit Régulateur Mode Glissant réalisé sur un FPGA. 90

Figure 5.15 Entrées/Sorties du circuit global de contrôle réalisé sur un FPGA. 90

Figure 5.16 Résultats de cosimulation. 91

Figure 5.17 Structure de contrôle de la commande LF. 92

Figure 5.18 Entrées/Sorties du circuit global de contrôle réalisé sur un FPGA. 93

Figure 5.19 Résultats de cosimulation. 94

Liste des abréviations

ASIC	*Application-Specific Integrated Circuits*
ASSP	*Application Specific Standard Product*
CG	*Grille de contrôle*
COG	*Center Of Gravity*
COM	*Center Of Maximum*
D	*Drain*
EEPROM	*PROM Effaçable Électriquement*
EPROM	*PROM Effaçable*
FG	*Grille Flottante*
FLC	*Fuzzy Logic Controller*
FPGA	*Field-Programmable Gate Arrays*
IC	*Integred Circuit*
ITGE	*Intégration à Très Grande Échelle*
LF-MG	*Logique Flou- Mode Glissant*
MLI	*Modulation de Largeur d'Impulsion*
MOM	*Means Of Maximum*

MSAP	*Machine Synchrone à Aimants Permanents*
PLD	*Programmable Logic Device*
PMSM	*Permanent Magnet Synchronous Machine*
PROM	*ROM Programmables*
RNA	*Réseaux de Neurones Artificiels*
ROM	*Read-Only Memory*
RTL	*Register Transfer Level*
S	*Source*
SLF	*Systèmes à Logique Floue*
SRAM	*Static Random-Access Memory*
SVPWM	*Space-Vector Pulse Width Modulation*
VHDL	*VHSIC Hardware Description Language*
VLSI	*Very Large Scale Integration*

Liste des symboles

d-q	Référentiel fixe
e_ω	Erreur de vitesse
$\dot{e}_\omega$	Variation de l'erreur
i_d, i_q	Courants dans le repère d-q
$[i_s]$	Vecteur de courant statorique
J	Inertie
L_d, L_q	Inductances dans le repère d-q
$[L_s]$	Matrice inductance du stator
m_i	Nombre de sous-ensembles flous
n	Nombre de variables d'entrée
p	Nombre de paires de pôles
r_{max}	Nombre maximum de règles
R	Résistance statorique
$[R_s]$	Matrice résistance du stator
s	Modèle d'erreur de la vitesse
V_d, V_q	Tensions dans le repère d-q
(S_a, S_b, S_c)	États des trois bras de l'onduleur

T_s	Période d'échantillonnage pour la SVPWM
V_{dc}	Tension du bus continu
$[V_s]$	Vecteur de tension statorique
$\overrightarrow{V_s}$	Vecteur d'espace de tension
α-β	Référentiel rotatif
λ	Flux magnétique permanent
λ_d, λ_q	Flux dans le repère d-q
θ	Position du rotor.
φ	Angle entre le vecteur $\overrightarrow{V_1}$ et $\overrightarrow{V_s}$.
$[\phi_s]$	Vecteur de flux statorique
τ	Couple électromagnétique
τ_f	Couple de frottement
τ_L	Couple de charge.
ω	Vitesse mécanique du rotor
μ	Fonction d'appartenance

Chapitre 1 - Introduction

Le développement rapide dans la technologie de la microélectronique, l'électronique de puissance, les lois de commande et surtout dans le domaine des matériaux magnétiques, a permis au moteur synchrone à aimant permanent (*MSAP*, en anglais *Permanent Magnet Synchronous Motor*, *PMSM*) de remplacer le moteur asynchrone et le moteur à courant continu dans de nombreuses applications industrielles, comme: les outils de fabrication à haute résolution, la robotique et les lecteurs de disque dur [1.1, 1.2, 1.3]. La popularité des *MSAP* vient de leurs avantages par rapport aux moteurs à courant continu et aux moteurs asynchrones, comme: leur faible bruit, faible inertie, rapport couple/courant élevé, efficacité élevée, robustesse, et leur faible coût de maintenance [1.3, 1.4]. Cependant, les non linéarités et les incertitudes internes et externes de la *MSAP* représentent de sérieux obstacles pour le contrôle en vitesse d'une *MSAP*. Afin d'obtenir les performances satisfaisantes, de nombreux chercheurs ont proposé divers concepts de commande, par exemple: le contrôle adaptatif, le contrôle avec linéarisation par rétroaction, la commande robuste, et le contrôle basé sur l'observateur de perturbations [1.1].

Plusieurs techniques de contrôle adaptatif ont été adoptées pour remplacer les méthodes conventionnelles qui ont le principal inconvénient de sensibilité envers le changement de paramètres de système. Mais pour la plupart de ces nouvelles techniques adaptatives, leurs algorithmes reposent sur une estimation en ligne des paramètres du moteur. Cette estimation peut paraitre comme un inconvénient majeur surtout en termes d'implémentation

ou il peut être difficile d'intégrer, loi de contrôle, algorithme de la *SVPWM* et aussi un algorithme d'estimation sur une seule plateforme. Les algorithmes de contrôle basés sur l'intelligence artificielle; comme le contrôle à base de la logique floue, et de réseaux de neurones sont considérés comme des solutions qui pourraient résoudre les problèmes associés à la non linéarité et l'incertitude du modèle mathématique du système, sans avoir besoin de procéder à une estimation en ligne [1.5].

Les techniques de commande basées sur l'intelligence artificielle conçoivent un modèle mathématique adaptatif du système pour résoudre le problème de contrôle concernant la non linéarité et l'incertitude des paramètres [1.6]. La capacité de s'adapter avec la variation et la non linéarité du modèle et aussi avec les perturbations externes a fait des contrôleurs intelligents une très bonne option dans le domaine de contrôle des systèmes électriques, spécialement les moteurs électriques [1.7].

1.1 Problématique

Dans la littérature, les approches classiques de commande de ces systèmes non-linéaires (*MSAP*) se basent sur un modèle mathématique précis du système et ont tendance à bien fonctionner en théorie. Mais, leurs performances se dégradent en présence de conditions de fonctionnement variables, d'incertitudes dynamiques, variation des paramètres et des perturbations externes [1.8, 1.9].

La commande exploitant les outils d'intelligence artificielle, comme les réseaux de neurones artificiels (RNA) et les systèmes à logique floue (SLF), a le potentiel de répondre à ces contraintes. Ces outils ont en effet la capacité de générer une approximation robuste de systèmes mal définis mathématiquement, avec incertitudes structurées et non structurées.

Cependant, les performances élevées sont obtenues avec ces outils au prix de calculs lourds [1.10]. Il est donc essentiel, pour profiter des avantages apportés par ces outils et pour les faire accepter par le milieu industriel, d'offrir des solutions pour l'implémentation matérielle efficace des algorithmes [1.11, 1.12].

Dans la littérature, on trouve plusieurs approches qui intègrent différentes technologies d'implémentation des lois de commande des machines électriques. Les microprocesseurs et les microcontrôleurs ont été les solutions les plus adoptées pour des applications simples qui n'ont pas besoin d'un algorithme de commande complexe [1.10]. Comme les algorithmes de contrôle deviennent de plus en plus précis, alors leurs implémentations demandent plus de ressources et une vitesse de calcul plus élevée. Pour résoudre ce problème, les algorithmes de contrôle complexes sont désormais implémentés sur la technologie *Very Large Scale Integration* (VLSI) qui est plus rapide en termes d'exécution des opérations [1.13]. Par contre, pas tous les circuits VLSI sont propices à un changement de paramètres de configuration après l'avoir conçu, et pour cette raison les chercheurs ont adopté les *Field Programmable Gate Array* (FPGA) comme technologie rivale [1.14].

La réalisation des différents circuits de commande sur une seule carte est devenue possible grâce au développement rapide et la technologie avancée utilisée dans l'industrie micro-électronique. Cependant, *Application-specific integrated circuit* (ASIC) n'est pas la seule technologie d'implémentation qui a bénéficié de ce progrès dans le domaine de la micro-électronique. La technologie FPGA a aussi réussi à augmenter la densité des cartes et donc à améliorer leur coût. Donc, la technologie FPGA n'est plus juste une technologie de prototypage rapide mais elle représente dorénavant une solution définitive qui permet de raccourcir considérablement le temps de mise sur le marché [1.15].

1.2 Objectifs

Le but de ce projet est de simplifier et raffiner des structures de commande à base d'intelligence artificielle et de démontrer que même après avoir retravaillé ces algorithmes et les avoir implémentés à virgule fixe par prototypage rapide et sur circuit de commande programmable (FPGA), ils gardent leurs performances. Ces commandes utilisent des algorithmes d'apprentissage ou d'adaptation en ligne avec stabilité garantie. Les algorithmes d'adaptation sont souvent coûteux lors d'une implémentation et nécessitent des fréquences de calculs très élevées. L'adaptation et l'apprentissage hors ligne sont des solutions efficaces pour l'implémentation, mais les contraintes liées à la dynamique et aux changements de paramètres du modèle des machines électriques excluent cette solution. Alors les algorithmes à implémenter doivent être de meilleurs compromis entre les différents critères algorithmiques et d'ITGE.

Les objectifs de ce travail portent sur la proposition d'une librairie de schémas de commande à base d'intelligence artificielle pour le contrôle d'une machine synchrone à aimant permanent. Ces schémas ont une architecture hautement parallèle, ce qui permet d'un côté de changer la loi de commande seulement en activant ou en désactivant un bloc, de l'autre côté d'utiliser la même loi de commande mais pour une autre gamme de machines, en changeant minimalement le code. Au final on fait une comparaison entre les différentes lois qui ont été implémentées, en tirant des conclusions sur leurs performances côtés précision, fiabilité et implémentation, consommation et temps de calcul, pour décider laquelle des lois est la plus appropriée et efficace pour une implémentation.

1.3 Méthodologie

Pour contrôler unw *MSAP* en temps réel, notre contrôleur doit tenir compte des incertitudes internes et externes et de ses non linéarités. De plus, il faut prendre en considération les limites de la plateforme et du banc de test.

En premier lieu, les algorithmes à implémenter doivent être retravaillés et simplifiés le plus possible tout en gardant un minimum de précision. Cette étape consiste généralement à plusieurs opérations : normalisation, restructuration de l'algorithme pour avoir une structure parallèle, utilisation de multiples signaux d'horloge (*Clock*), approximation des fonctions complexes, soit par des méthodes mathématiques comme l'interpolation ou directement en les remplaçant par des fonctions moins complexes.

La deuxième étape est de passer du calcul à virgule flottante à la virgule fixe. Le critère de précision utilisé dans cette étape est pour les opérations de base comme l'addition, la soustraction, et la multiplication. L'erreur doit être inférieure à l'erreur correspondant à 1bit. Donc le défi ici est de s'assurer que l'erreur ne s'accumule pas. Pour faire face à ce défi, une méthode de validation très complexe est employée afin de valider des plus petites parties de l'algorithme à l'assemblage final de ces parties. Le schéma de la figure 1.1 décrit les différentes étapes à suivre pour valider un bloc VHDL.

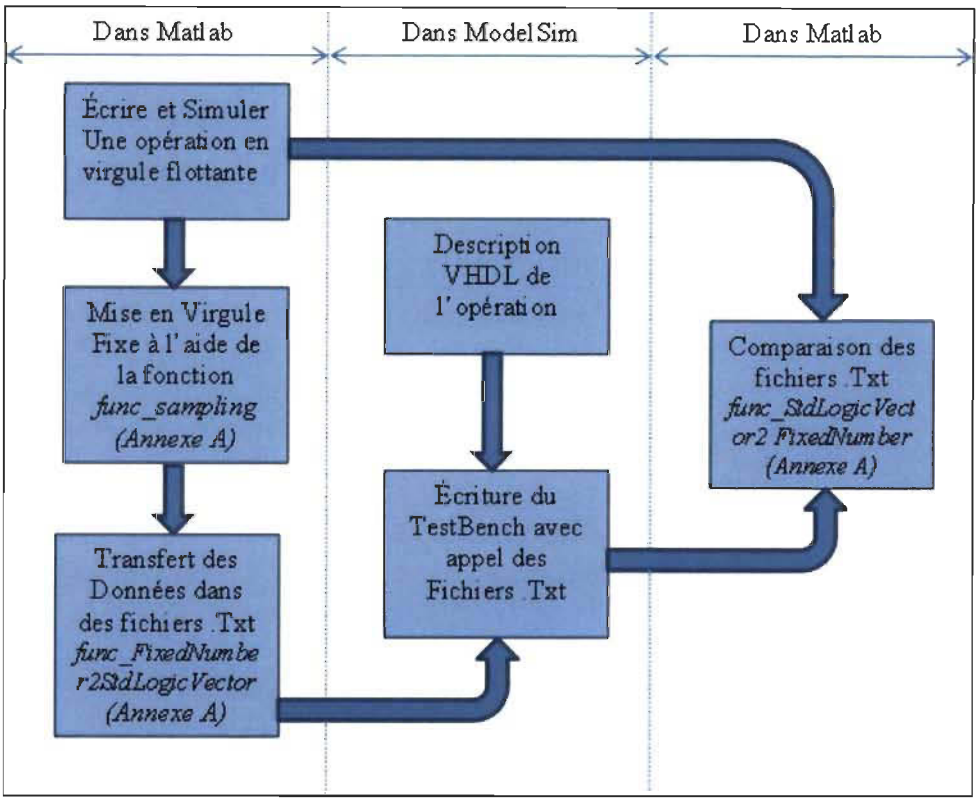


Figure 1.1 Processus de simulation Matlab/ModelSim/Matlab

1.4 Structure du mémoire

Le deuxième chapitre contient une étude sur la logique floue, la définition des différents termes utilisés, les opérateurs et les normes applicables dans la logique floue et les différents éléments constituant un bloc de logique floue.

Le troisième chapitre a pour but d'expliquer les différents principes des FPGA, présenter des généralités, les différents éléments électroniques qu'on peut trouver sur un FPGA, une étude de quantification, et un survol sur l'implémentation des différente opérations arithmétiques de base en utilisant le langage de programmation VHDL.

Le quatrième chapitre présente, une synthèse des connaissances dans le domaine des entrainements électriques, un survol sur les machines synchrones à aimant permanent, la

modélisation et la commande des convertisseurs cc-ca, et le choix d'une méthode de commande et l'implémentation.

Le cinquième chapitre est consacré pour présenter les différentes lois de commande qui ont été implémentées. Dans ce chapitre la théorie de ces lois de commande est présentée brièvement afin de consacrer la grande partie pour les détails et les résultats d'implémentation.

Finalement dans la partie conclusion générale, nous présenterons une discussion des principaux résultats, de la contribution du projet et des travaux futurs.

Chapitre 2 - Logique Floue

2.1 Introduction

2.1.1 Généralités

Dans le cadre de recherche pour le développement de nouvelles technologies basées sur l'intelligence artificielle, la logique floue a reçu un immense intérêt par les chercheurs et les industriels depuis le début de la deuxième moitié du dernier siècle. Avant cette tournure dans la définition de la logique, tous les algorithmes et les procédés se basaient sur la logique classique [2.1]. La logique classique n'admet aucun état entre le vrai et faux, ou aucune valeur entre le 0 et 1, contrairement à la logique floue qui a l'avantage de traiter même les valeurs entre 0 et 1 en se basant sur le raisonnement humain [2.1-2.3].

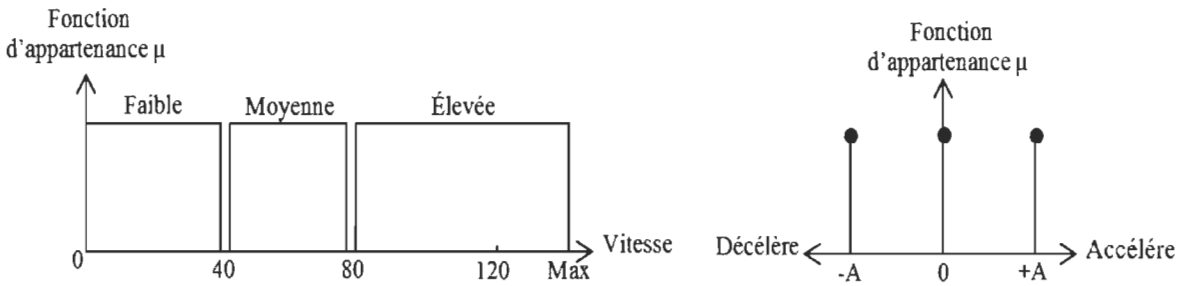
Les travaux de Lotfi A. Zadeh (Professeur de l'Université de Californie de Berkeley) ont été les premiers qui présentaient la logique floue comme une méthode de classification pertinente en ayant établi les principes de bases [2.4]. Initialement, la logique floue a été désignée comme un algorithme de classification qui a été utilisé pour l'analyse de données, surtout dans des domaines comme le commerce et la médecine [2.5]. Cependant, son principe de raisonnement inspiré du raisonnement humain a attiré Mamdani qui a été le

premier qui a introduit la notion de la logique floue dans le domaine de contrôle en 1974 [2.6, 2.7].

2.1.2 Exemple Introductif

Afin de bien comprendre le principe de base de la logique floue, dans cette partie on expose un exemple simple de régulation [2.7]. Nous prenons l'exemple de régulation de vitesse de déplacement d'un véhicule. Dans un premier temps, la vitesse de déplacement pourra être estimée soit : basse, moyenne, ou élevée. Si nous nous basons sur la logique classique pour réguler la vitesse du véhicule à une vitesse désirée moyenne, nous nous retrouvons avec un algorithme simple comme le montre la figure 2.1:

- Si la vitesse du véhicule est faible : Accélérer;
- Si la vitesse du véhicule est moyenne : Ne rien faire;
- Si la vitesse du véhicule est élevée : Décélérer.



a- Exemple de répartition des zones de vitesse en trois classes

b- Choix de la décision

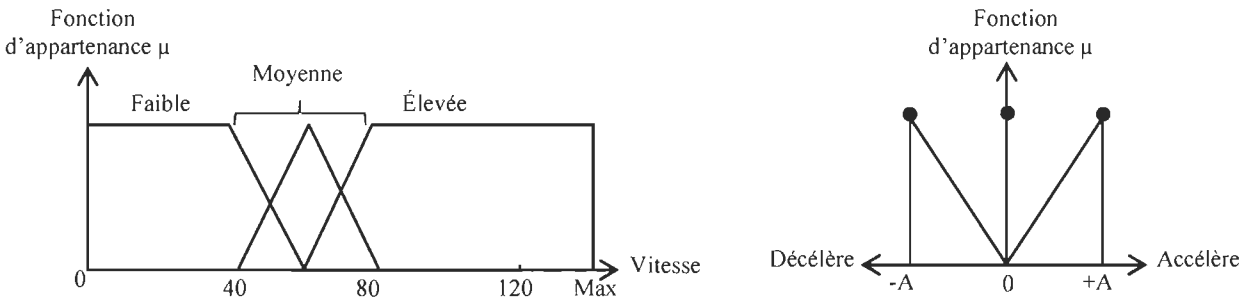
Figure 2.1 Exemple de réglage de vitesse d'un véhicule selon la logique classique.

Nous constatons que ce raisonnement est loin d'être semblable au raisonnement humain qui est capable d'être plus précis au niveau de classification. Une vitesse de 38 km/h d'après la logique booléenne appartient à l'intervalle vitesse faible, alors que le cerveau

humain voit qu'elle pourrait appartenir autant à l'intervalle vitesse moyenne, qu'à l'intervalle vitesse faible. C'est de là que le principe de la logique floue a été inspiré, le degré d'appartenance peut varier entre 0 et 1 d'une façon que la somme de tous donne 1. La vitesse du véhicule n'est pas toujours soit nettement faible, moyenne ou élevée, elle peut être par exemple à 80% faible, 20% moyenne et 0% élevée, et dans ce cas la décision ne va pas être la même qu'avec la logique classique, parce que la logique floue prend en considération le passage d'intervalle.

Dans l'exemple considéré (Fig. 2.2) :

- Pour une vitesse inférieure à 40Km/h, la vitesse est faible;
- Pour une vitesse inférieure à 60Km/h et supérieure à 40Km/h, on hésite entre faible et moyenne;
- Pour une vitesse inférieure à 80Km/h et supérieure à 60Km/h, on hésite entre moyenne et élevée;
- Pour une vitesse supérieure à 80Km/h, la vitesse est élevée.



a- Exemple de répartition des zones de vitesse en trois classes

b- Choix de la décision

Figure 2.2 Exemple de réglage de vitesse d'un véhicule selon la logique floue.

Maintenant qu'on a une classification précise de la vitesse de déplacement, ça aidera l'expert d'en déduire une correction plus précise et qui tient compte du passage entre les intervalles. Par exemple, lors du passage de l'intervalle 'Faible' à l'intervalle 'Moyenne', l'accélération n'est plus constante. L'accélération est une image de la fonction d'appartenance comme le montre la figure 2.2.

2.2 Définitions

2.2.1 Univers de discours

L'univers de discours est la plage de variation de la grandeur mesurée ou de commande. Reprenons l'exemple introductif : l'univers de discours de la vitesse du véhicule est [0, Max[. Il est imposé par les caractéristiques du processus à commander, dans ce cas, la voiture. L'univers de discours de l'accélération est l'ensemble des trois valeurs, -A, 0 et +A. L'univers de discours de la réponse est fixé par l'expert selon la dynamique qu'il cherche à obtenir [2.8, 2.9].

2.2.2 Variable linguistique

Chaque univers de discours est divisé en plusieurs sous-ensembles. Une fois que la valeur est assignée au sous-ensemble auquel elle appartient, elle est représentée par le symbole ou par le mot qui représente ce sous-ensemble. Ce mot ou ce symbole est la variable linguistique. Dans l'exemple introductif : faible, moyenne et élevée sont les variables linguistiques [2.10].

2.2.3 Fonction d'appartenance

La fonction d'appartenance est l'ensemble de degrés d'appartenance à chaque valeur linguistique [2.2, 2.11].

$$\mu(Faible) = \begin{cases} 1 & Si\ vitesse \leq 40Km/h \\ \frac{vitesse}{(40-60)} - \frac{60}{(40-60)} & Si\ 40Km/h \leq vitesse \leq 60Km/h \end{cases} \quad (2.1)$$

$\mu(Faible)$ est la fonction d'appartenance de la valeur linguistique *Faible* dans l'exemple.

Comme le montre l'exemple, il existe plusieurs formes de fonctions d'appartenance.

Exemples de fonction d'appartenance :

- Fonction d'échelon (Figure 2.1.b);
- Fonction triangulaire (Figure 2.2.a);
- Fonction Carrée, ou Trapézoïdale (Figure 2.2.a);
- Fonction sigmoïde (Figure 2.3), etc…

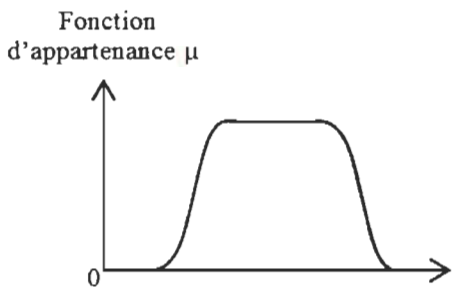


Figure 2.3 Exemple d'une fonction sigmoïde.

2.3 Opérateurs et normes

Dans certain cas, on trouve que la réponse de la logique floue dépend de plus qu'une condition (deux ou plus). Dans la logique classique il existe des opérations qui permettent de gérer le chevauchement de plusieurs conditions. Les mêmes opérateurs utilisés pour la logique booléenne sont encore utilisables pour la logique floue [2.1].

2.3.1 Opérateur NON

Selon la logique classique, la négation est définie par :

$$c = NON(a) = \bar{a} \tag{2.2}$$

Dans le cas de la logique floue, cette expression peut être écrite d'une façon plus générale :

$$NON\ (\mu_a(x)) = 1 - \mu_a(x) \tag{2.3}$$

2.3.2 Opérateur ET

L'opérateur ET dans la logique floue correspond à l'opération minimum. L'application de cette opération sur deux fonctions d'appartenances $\mu_a(x)$ *et* $\mu_b(x)$, donne la fonction $\mu_c(x)$ qui représente l'intersection entre les deux fonctions.

$$\mu_c(x) = min\ [\mu_a(x),\ \mu_b(x)] \tag{2.4}$$

2.3.3 Opérateur OU

Selon la théorie des ensembles, l'opération OU est utilisée souvent pour exprimer l'union entre deux ensembles. Dans la logique floue, cette opération est définie par l'opération maximum. L'application de cette opération sur deux fonctions d'appartenance $\mu_a(x)$ *et* $\mu_b(x)$, donne la fonction $\mu_c(x)$ qui représente l'union des deux fonctions.

$$\mu_c(x) = max\ [\mu_a(x),\ \mu_b(x)] \tag{2.5}$$

2.4 Unités d'un bloc logique flou

Un contrôleur logique flou typique est représenté dans la figure 2.4. Basé sur la logique floue, le noyau du dispositif de commande se trouve dans sa base de connaissances, constituée de règles floues qui décrivent la réaction du régulateur, et un système d'inférence qui combine les règles actives selon les entrées présentées au contrôleur [2.12, 2.13]. Comme les règles et le système d'inférence agissent sur des sous-ensembles flous, il est nécessaire de convertir les données nécessaires à la régulation provenant du monde extérieur en valeurs linguistiques qui peuvent être manipulées par les fonctions de la base de connaissances de la logique floue [2.14]. Les décisions prises par la base de connaissances sont en général des valeurs linguistiques qui doivent être converties en valeurs numériques afin de les appliquer sur le processus à contrôler. De ce fait, un bloc logique flou peut être divisé sous 5 sous-blocs :

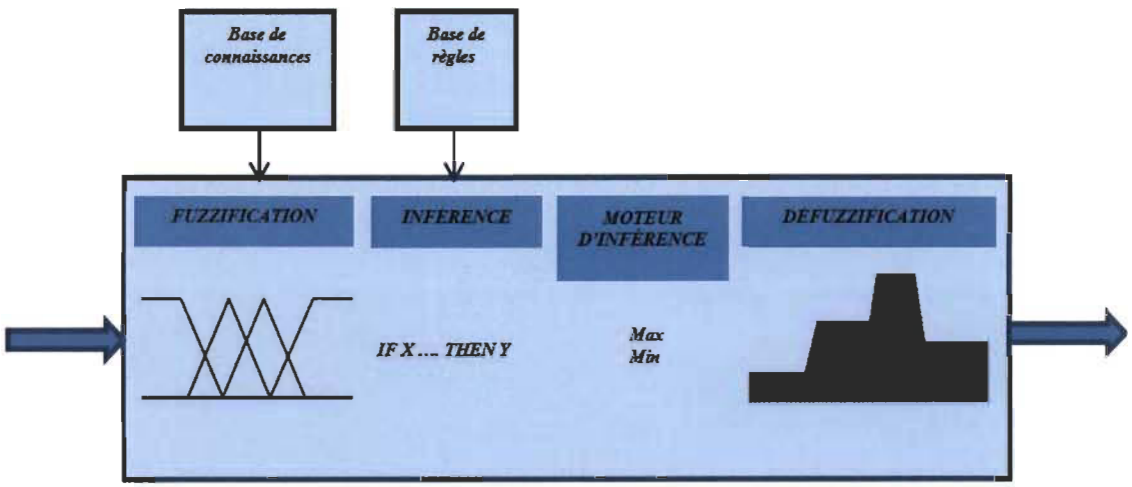


Figure 2.4 Schéma des différentes opérations dans un bloc logique flou.

2.4.1 La fuzzification

La fuzzification est la première étape dans le FLC qui transforme les entrées numériques x_i en un ensemble de valeurs d'appartenance dans l'intervalle [0,1] à des ensembles flous correspondants μ_{xi}. La fonction d'appartenance est une représentation graphique de l'amplitude de participation de chaque entrée. Il existe de nombreux types de fonctions d'appartenance. Parmi eux, les deux plus couramment utilisés dans la pratique sont les fonctions triangulaires et trapézoïdales [2.15, 2.16]. Le nombre de fonctions d'appartenance à définir pour chaque variable linguistique est défini à l'aide d'expertise humaine. Plus l'univers de discours contient de sous-ensembles flous, plus le régulateur flou est précis.

2.4.2 Base de règles

En logique floue, les règles sont souvent formulées en basant sur la connaissance de l'expert du comportement et de la dynamique du système. Ces règles stipulent la relation entre les ensembles flous d'entrée et les ensembles flous de commande correspondante [2.15, 2.17, 2.18]. Une règle prend habituellement la forme d'instruction IF -THEN comme suit :

IF x is a AND y is b THEN z is c

Par conséquent, le nombre de règles définies dépend directement du nombre de sous-ensembles défini pour chaque variable d'entrée et de sortie. Si on définit n variables d'entrée et de sortie X_i dans notre système, et pour chaque univers de discours de ces variables on a m_i sous-ensembles flous, le nombre maximum de règles est défini par la relation suivante [2.2] :

$$r_{max} = \prod_{i=1}^{n} m_i \tag{2.6}$$

2.4.3 Mécanisme d'inférence

Aussi appelé le moteur d'inférence ou inférence floue, le mécanisme d'inférence est l'élément clé dans l'algorithme de logique floue qui émule la prise de décision de l'expert dans l'interprétation et l'application de connaissances sur la meilleure façon de contrôler le système [2.3].

Après avoir décidé quelles sont les règles à appliquer, maintenant, cette étape consiste à définir les degrés d'appartenance de la variable de sortie aux ensembles flous [2.6]. Il existe deux méthodes fondamentales qui permettent de calculer ces degrés d'appartenance. Les autres méthodes permettant d'y arriver sont nombreuses, la différence entre elles se définie essentiellement par la façon de réaliser les opérateurs flous (ET, OU et NON).

2.4.3.1 La méthode de Mamdani

Pour les moteurs d'inférences basés sur la méthode de Mamdani, les degrés d'appartenance de la variable de sortie, sont calculés à l'aide des opérateurs ET et OU réalisés par les deux fonctions *Min* et *Max* respectivement.

Dans cette méthode, les règles s'écrivent de la façon suivante:

Si x_1 *est* A_1^l *et* x_2 *est* A_2^l *et ... et* x_n *est* A_n^l *Alors* y_1 *est* C_1^l *et* y_2 *est* C_2^l *et ... et* y_m *est* C_m^l

où x_i, *et* y_j sont les variables linguistiques d'entrée et de sortie, respectivement; A_i^l et C_j^l, $i = 1, ..., n$, $j = 1, ..., m$, sont les ensembles flous entrées et sorties, respectivement; n *et* m sont les nombres d'entrées et de sorties du bloc logique floue, respectivement.

Dans ce cas les degrés d'appartenance correspondant aux variables de sortie sont calculés à l'aide de la relation suivante :

$$\mu_{y^l} = \max(\min(\mu_{A_i}, \mu_{R^l})) \qquad 2.7$$

où, R^l est la relation floue entre les entrées x_i et les sorties y_i, $l = 1,\ldots, r$ et r est le nombre des règles [2.12]

2.4.3.2 La méthode de Takagi-Sugeno

Cette méthode utilise la même forme de règles que la méthode de Mamdani, *Si..... Alors,* où l'antécédent est toujours une variable linguistique, mais le conséquent utilise des variables numériques. Le conséquent peut être calculé à partir de m'importe quelle formule mathématique, une constante, un polynôme, ou une fonction de manière générale; tout dépend du comportement voulu.

Si x_1 *est* A_1^l *et* x_2 *est* A_2^l *et ... et* x_n *est* A_n^l *Alors* $y_1 = C_1^l(x)$ *et* $y_2 = C_2^l(x)$ *et ... et* $y_m = C_m^l(x)$

où C_j^l, $i = 1,\ldots, n$, $l = 1,\ldots, r$, sont les ensembles flous de sorties [2.12].

2.4.4 La défuzzification

C'est la dernière étape de la logique floue. Avant que les sorties du moteur d'inférence soient appliquées sur le processus à contrôler, ces dernières qui sont représentées comme des degrés d'appartenance aux fonctions de la sortie, doivent être converties [2.19]. Alors l'étape de la défuzzification consiste à convertir ces valeurs floues en variables réelles qui peuvent être utilisées. Dépendamment de la forme de la sortie voulue, du type de contrôle,

du type des fonctions d'appartenance de la sortie, il existe trois méthodes fondamentales de défuzzification [2.18].

2.4.4.1 Méthode de centre de gravité (COG)

C'est une des méthodes les plus utilisées grâce à la haute précision qu'elle offre. Cette méthode calcule la totalité de l'espace actif sous les fonctions d'appartenance. Mais le fait de calculer plusieurs surfaces présente un inconvénient côté complexité et temps de calcul.

2.4.4.2 Méthode de centre de maximum (COM)

Cette méthode est connue pour sa simplicité vu qu'elle prend en considération que le sommet de l'espace actif sous la fonction d'appartenance. Dans le cas où les fonctions d'appartenance sont de type échelon, la précision de cette méthode est similaire à la première.

2.4.4.3 Méthode de moyenne de maximum (MOM)

Dans cette méthode, le résultat de sortie correspond à la moyenne entre les maximums de chaque fonction d'appartenance. Elle est utilisée beaucoup plus dans les cas où la grandeur de sortie a des valeurs fixes et discontinues.

2.5 Conclusion

Dans ce chapitre, on a présenté les différentes étapes pour la conception d'un régulateur flou. La difficulté de la conception réside dans la configuration de chaque bloc vu que cela dépend de l'expertise humaine. Dans le cinquième chapitre on présentera la configuration de chaque étape qui a été présenté dans ce chapitre.

Chapitre 3 - Field Programmable Gate Array « FPGA »

3.1 Introduction

Les contrôleurs analogiques, numériques ou analogiques et numériques peuvent être utilisés pour contrôler n'importe quelle grandeur physique désirée (température, pression, tension électrique, position mécanique, etc.). Au début de l'ère de contrôle, la plupart des systèmes physiques et spécialement les systèmes électriques se contrôlèrent à l'aide des contrôleurs analogiques, mais leur taille, poids, manque de flexibilité et complexité de les concevoir ont poussé les chercheurs à se concentrer sur la technologie numérique. Cependant, avec le développement de la technologie numérique, la conception de systèmes de contrôle est devenue plus facile ainsi que plus économique [3.1].

Un autre progrès récent dans les systèmes de traitement informatique, qui a contribué à l'augmentation de la capacité de traiter et résoudre les problèmes très complexes liés au contrôle des systèmes électriques, est dans la façon de traitement, où la tâche de calcul est partagée entre plusieurs processeurs qui peuvent communiquer les uns avec les autres d'une manière efficace. Les processeurs dits individuels peuvent résoudre les sous-problèmes qui constituent le problème original. L'assemblage des résultats de chaque sous-problème d'une certaine manière ordonnée, permet à arriver à la solution du problème global. Étant donné que de nombreux processeurs peuvent être incorporés pour exécuter les calculs, il est possible de résoudre des problèmes importants et complexes rapidement et efficacement [3.2].

Aujourd'hui avec le développement de la technologie ITGE, les systèmes de contrôle numérique sont de plus en plus utilisés dans différents domaines d'application en raison de leur exactitude, précision, haute vitesse. De plus, il est possible de concevoir des circuits miniaturisés à haute performance, en termes de puissance et de vitesse, à faible coût. Grâce aux avantages qu'offre la technologie ITGE, ce domaine a gagné une énorme popularité au cours des deux dernières décennies [3.1, 3.3].

Les familles les plus connues de la technologie ITGE sont : PLD (*Programmable Logic Device*), FPGA (*Field-Programmable Gate Array*), ASIC (*Application Specific Integrated Circuit*) et ASSP (*Application Specific Standard Product*). Ces différentes familles ont leurs propres avantages et inconvénients.

Les PLD sont des dispositifs dont l'architecture interne est prédéterminée par le fabricant, mais sont conçus d'une manière telle que les ingénieurs spécialisés peuvent les configurer pour effectuer une variété de fonctions différentes. L'architecture interne d'un dispositif FPGA est plus développée que celle des PLD; elle comporte un nombre très élevé de portes logiques et les fonctions qui peuvent être réalisées par chacune de ces portes logiques sont beaucoup plus vastes et plus complexes que celles d'un PLD. À l'autre extrémité du spectre se trouvent les ASIC et ASSP qui peuvent contenir des centaines de millions de portes logiques et peuvent être utilisés pour créer des fonctions incroyablement vastes et complexes. ASIC et ASSP sont basés sur les mêmes processus de conception et de technologies de fabrication. Les deux sont conçus sur mesure pour répondre à une application spécifique, la seule différence étant que l'ASIC est conçu et construit à l'ordre pour une utilisation par une entreprise spécifique, tandis qu'un ASSP est commercialisé à

plusieurs clients [3.4]. Comme dans plusieurs publications, nous nous permettons dans le reste du chapitre de croire que les dispositifs ASSP appartiennent à la famille ASIC.

Bien que l'ASIC offre le rapport nombre de transistors/taille le plus élevé, le temps de calcul le plus rapide, et moins de consommation par rapport aux autres familles d'ITGE, la technologie des ASIC n'est pas la plus adaptée aux différentes applications, et ce, à cause du temps, de la complexité et du coût de fabrication très élevés des ASIC avec l'inconvénient supplémentaire et majeur que la conception finale est "gelée dans le silicium" et ne peut être modifiée sauf si le fabriquant crée une nouvelle version du circuit [3.5]. De son côté l'architecture interne d'un FPGA présente une flexibilité et re-programmabilité. Une telle architecture peut être utilisée pour exécuter diverses applications. De même le produit final peut être corrigé (s'il y a des erreurs) ou amélioré (s'il y a des mises à jour) en reprogrammant simplement le FPGA. Un FPGA permet aussi une reconfiguration dite partielle, où seulement une partie du FPGA est reconfigurée tandis que d'autres parties du même FPGA sont toujours en train de fonctionner. La reconfiguration partielle est utile dans la conception de systèmes qui nécessitent de s'adapter fréquemment selon les contraintes des différents modes de fonctionnement [3.6]. Par rapport à d'autres technologies tel que les ASIC, les applications à base de FPGA ont moins de coût de conception par unité et une commercialisation plus rapide. Ainsi, les FPGA occupent une position intermédiaire entre PLD et ASIC parce que leur fonctionnalité peut être personnalisée comme avec un PLD, mais ils peuvent contenir des millions de portes logiques et être utilisés pour mettre en œuvre des fonctions extrêmement importantes et complexes qui ne pouvaient auparavant être réalisées qu'en utilisant uniquement des ASIC. Ces avantages font des contrôleurs à base de FPGA des contrôleurs efficaces et

économiques, mais cela à un coût important en matière de taille, de temps de calcul et de consommation d'énergie: un FPGA nécessite environ 20 à 35 fois plus d'espace qu'une cellule ASIC standard, a une performance de vitesse à peu près 3 à 4 fois plus lent qu'un ASIC et consomme environ 10 fois plus d'énergie dynamique [3.5].

Au début des années 1980 les FPGA commencèrent à apparaitre sur la scène des IC (*Integred Circuit*). Jusqu'à cette époque les FPGA ont été largement utilisés pour mettre en œuvre les différentes fonctions de la logique binaire et combinatoire, des machines d'état de complexité moyenne, et des tâches relativement limitées pour le traitement de données. Au début de 1990, comme la taille et la sophistication des FPGA ont commencé à augmenter, les domaines d'utilisation des FPGA ont connu un élargissement spectaculaire.

Au début du XXIe siècle, les deux leaders de fabrication et de commercialisation des dispositifs logiques programmables, Altera et Xilinx, à la fois déclarent des revenus supérieurs à 1 milliard de dollars US. Les FPGA ont connu une croissance régulière de plus de 20 % dans la dernière décennie, dépassant les ASIC de 10%. Cela vient du fait que les FPGA ont réussi de gagner de nombreuses nouvelles caractéristiques en commun avec les ASIC, tels que la réduction de la taille, la dissipation de puissance, et un temps de calcul plus rapide, en gardant les mêmes avantages par rapport aux ASIC [3.7]. Du coup, aujourd'hui les FPGA sont utilisés dans toutes les applications qui nécessitent une plateforme de traitement de données rapide, efficace, pas cher, et pas volumineuse.

Dans ce chapitre on commencera par présenter un survol de l'histoire de développement des FPGA, ensuite on présente le principe de fonctionnement des *FPGA* en expliquant les éléments de base qu'on peut trouver sur une carte *FPGA*. Dans la dernière

partie du chapitre on abordera l'intégration de données, et on présentera quelques exemples sur *VHDL* des opérations de base (addition, multiplication,... etc.).

3.2 Histoire des FPGA.

Les FPGA ont fait leur apparition au début des années 1960 après la publication des travaux de Gerald Estrin qui ont été connus par "*fixed plus variable structure computer*" [3.8]. A cette époque l'idée de la configurabilité était d'utiliser une régularité structurale et une flexibilité fonctionnelle. Les réseaux cellulaires étaient généralement constitués d'un tableau à deux dimensions de cellules logiques simples avec une communication fixe point à point. La programmation des cellules logiques de ces réseaux, tels que 'Maitra cascades', se faisait par métallisation pendant la fabrication pour mettre en œuvre un ensemble de fonctions logiques à deux entrées. Cependant, après l'introduction de la technique '*Cutpoint*', à la fin des années 1960, la possibilité de modifier la fonction logique d'une puce après le processus de fabrication a été atteinte. Bien que la communication entre les différentes cellules logiques fût toujours fixe, la fonctionnalité de chaque cellule logique dans le réseau peut être déterminée par l'état des fusibles programmables. Ces fusibles peuvent être programmés grâce à l'utilisation d'une intensité bien définie de courant ou par la propriété de la photoconductivité [3.5].

En 1970 une nouvelle technologie de conception des FPGA qui se base sur le principe des ROM (*Read-Only Memory*) est apparue. Une série de ROM programmables (PROM) de N-entrées, est capable d'implémenter certaines fonctions logiques complexes. Comme l'indique leur appellation, cette catégorie de mémoires mortes se caractérise par le fait qu'elle est programmable une seule fois après la fabrication. Il existe plusieurs types de PROM qui se distinguent surtout par la façon de les programmer [3.5]. Cette technologie

basée sur les PROM a été combinée en 1977 avec les technologies utilisées jusqu'à ce moment pour rajouter aux nombreux avantages la flexibilité vu que les PROM une fois programmées sont des mémoires mortes.

Le concept moderne du FPGA est connu depuis 1985 avec la commercialisation de deux plateformes qui ont profité des progrès spectaculaires dans le domaine des semi-conducteurs. La première est la famille XC2064TM FPGA de Xilinx, qui était basée sur SRAM (*static Random-access memory*). La deuxième, EP1200 d'Altera, avait une très haute densité (3-μm CMOS) et utilisait des PROM effaçables (EPROM) [3.9]. Ces deux produits ont été le véritable lancement de la technologie FPGA à l'échelle de prototypage rapide et de nos jours, elle rivalise avec l'ASIC à l'échelle de produit final. Au courant des deux dernières décennies plusieurs familles de FPGA ont été commercialisées par les différents fabricants, Xilinx, Altera, Actel, Lattice, Crosspoint, etc. Pour résumer cette période, Dr.Steve Trimberger qui est un des célèbres experts de Xilinx, a divisé les étapes de progression des FPGA comme dans le tableau 3.1 [3.9].

Tableau 3-1 Progression des FPGA

Période	Étape	Caractéristiques
1984-1991	Invention	➢ Technologie était limitée; ➢ Outils de conception étaient primitifs; ➢ Architecture était efficace.
1992-1999	Développement	➢ Taille de FPGA se rapprochait de la taille du problème; ➢ Facilité de conception devenait critique.
2000-2007	Accumulation	➢ La taille des FPGA ne pose plus de problème; ➢ Technologie des I/O.

3.3 Principe de fonctionnement des FPGA

Un FPGA, comme l'indique son nom, se compose d'une matrice de blocs logiques reprogrammables. La topologie d'interconnexion entre les différents blocs logiques fait distinguer deux structures : la topologie arborescente et la topologie maillée. La première se base sur une structure qui relie les blocs logiques d'une façon hiérarchique et récursive. La deuxième est une structure organisée sur la forme de mailles et les blocs logiques de chaque ligne sont reliés en cascade. La première topologie offre une taille plus miniature pour la plateforme, mais en revanche une simplicité critique pour l'opération de routage. C'est pour cette raison que la plupart des fabricants adoptent la topologie maillée qui est totalement le contraire. La figure 3.1. montre l'architecture d'un FPGA qui adopte la topologie maillée [3.6].

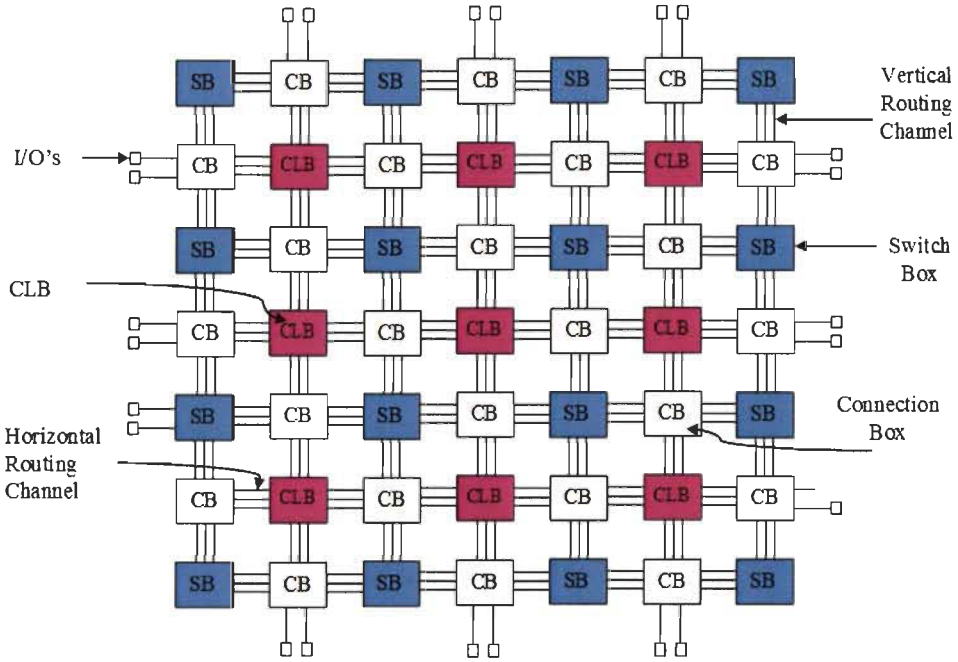


Figure 3.1 Architecture traditionnelle d'une plateforme FPGA à base de mailles [3.6].

Un FPGA peut contenir différents blocs logiques entre les blocs d'entrée et les blocs de sortie, tout dépend des technologies utilisées et à quelle application il est désigné. En plus, ce que la figure 3.1 montre, un FPGA peut contenir aussi des mémoires, des multiplicateurs et des additionneurs déjà implémentés. Dans la partie suivante nous présenterons une partie des composants qui constituent le mécanisme qui nous permet de configurer (programmer) une puce de silicium.

3.3.1 Une simple fonction programmable

Pour présenter les différents composants, nous commençons par proposer une simple fonction à deux entrées et une sortie.

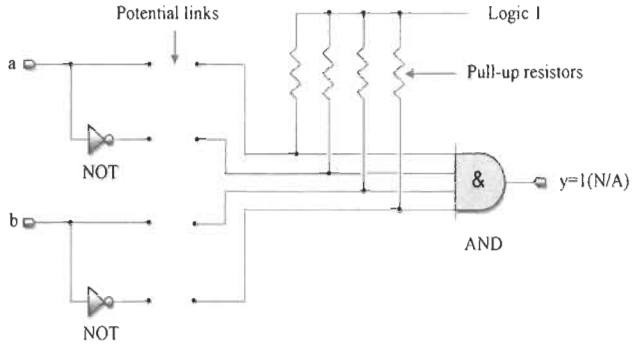


Figure 3.2 Exemple d'une simple fonction programmable [3.4].

Les inverseurs à l'entrée de la fonction donnent une large plage de fonctionnalités pour cette fonction de base. Les connexions potentielles constituent le mécanisme de programmation de cette fonction. En disposant de deux entrées avec leurs états complémentaires, même une fonction simple comme AND peut être dérivée en 2^2=4 fonctions. Des résistances de sécurité sont reliées sur l'état 1 pour la sécurité du circuit si cette fonction n'a pas été configurée.

3.3.2 Technologie de connexion à base de fusibles

Dans cette technologie, les fonctions logiques reprogrammables sont fabriquées avec toutes les connexions activées. Le dispositif de connexion est un fusible, semblable au fusible électrique utilisé pour sécuriser les montages électriques. L'idée c'est de faire griller volontairement le fusible correspondant à l'entrée indésirable, comme le montre la figure 3.3 [3.7].

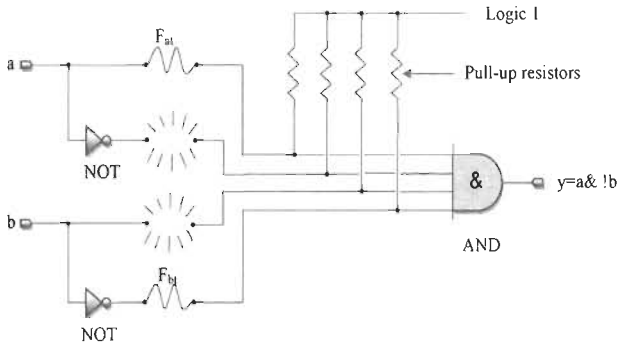


Figure 3.3 Exemple d'une connexion à base de fusibles [3.4].

3.3.3 Technologie de connexion à base d'anti-fusibles

L'anti-fusible est un dispositif à deux bornes qui a le même principe que le fusible, mais d'une façon inverse. L'anti-fusible dans son état normal est isolant contrairement au fusible et il est conducteur lorsqu'il subit une haute tension sur ses deux bornes [3.10].

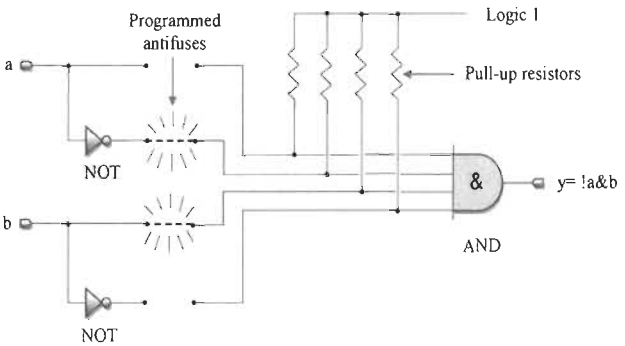


Figure 3.4 Exemple d'une connexion à base d'anti-fusibles [3.4].

3.3.4 ROM

Dans chaque système électronique, certaines informations doivent être stockées de façon permanente, c'est à dire qu'elles doivent être conservées même lorsque le système est hors tension.

Depuis une longue période, il y avait deux catégories différentes de mémoires non volatiles qui sont programmables électriquement: de l'EPROM et l'EEPROM (PROM effaçable électriquement).

EPROM ont une cellule de mémoire d'un seul transistor donc ils peuvent fournir une haute densité et rentabilité, cependant leur structure ne permet pas d'effacer les données stockées.

De par sa structure l'EEPROM a la capacité d'effacer électriquement des données. Toutefois, en raison de la structure complexe d'une cellule de mémoire, leur cout est assez élevé d'un côté, et de l'autre côté cette technologie offre une densité beaucoup plus faible qu'EPROM.

Avec le développement de la technologie des semi-conducteurs une troisième catégorie de mémoires non volatiles est devenue la ROM la plus demandée du marché. Cette technologie qui s'appelle *Flash memory* est de plus en plus utilisée ces dernières années sous forme de nouvelles architectures. L'architecture de base est représentée sur la figure 3.5 [3.7].

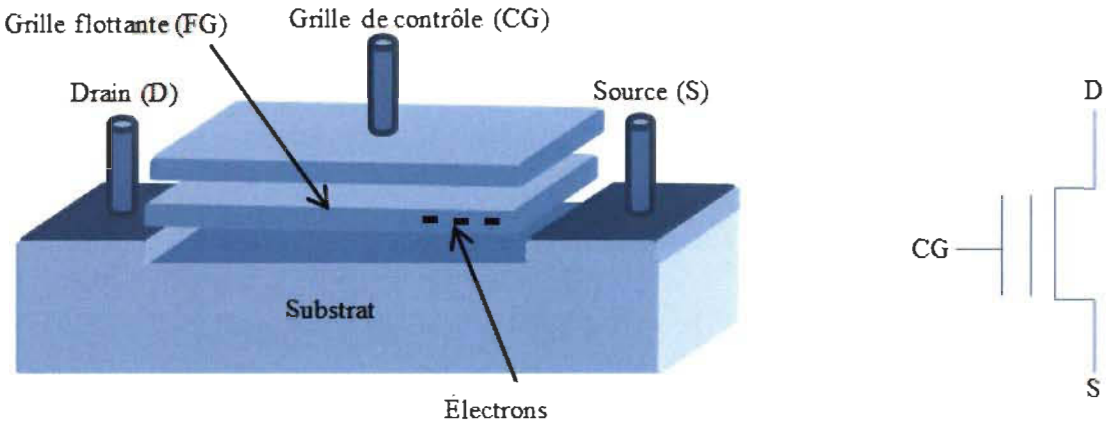


Figure 3.5 Cellule principale d'une mémoire à grille flottante et son schéma symbolique [3.7].

3.3.5 RAM

Les mémoires vives statiques sont constituées d'éléments statiques (transistors et diodes) d'où vient son appellation. Elles ont toujours joué un rôle essentiel dans la majorité des systèmes VLSI. Cependant, cette catégorie de RAM présente des faiblesses à basse échelle de tension; pour remédier à ce problème les mémoires vives dynamiques ont été proposées. À la base de ces deux catégories, plusieurs variétés ont été proposées au fil du temps.

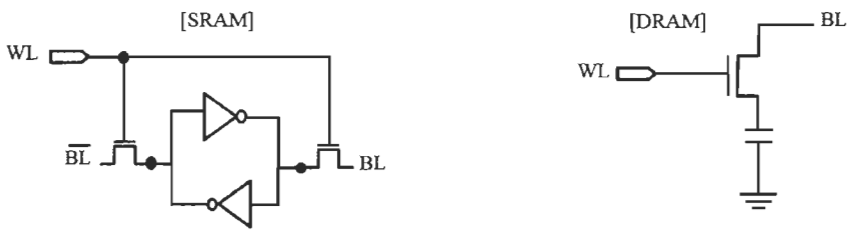


Figure 3.6 Exemple d'une mémoire RAM statique et dynamique [3.7].

3.4 Intégration sur VHDL

Dans cette section, nous expliquons la façon de concevoir les circuits logiques qui effectueront les opérations arithmétiques. Dans l'arithmétique numérique deux principes de conception fondamentaux sont d'une grande importance: la représentation des nombres et la mise en œuvre des opérations algébriques [3.7]. Chacune des deux représentations qui sont utilisées habituellement pour l'implémentation numérique ont des avantages et des inconvénients. La représentation en virgule flottante peut, avec le même nombre de bits, couvrir une plage plus large de nombres décimaux avec une précision supérieure à la virgule fixe [3.2]. Cependant cette haute performance se paie au prix de consommation d'espace et le temps de calcul élevé par rapport à une représentation en virgule fixe. Du coup on a opté pour la représentation la moins couteuse [3.9]. Après avoir choisi la représentation la plus convenable à notre application et à la base de cette représentation nous procéderons à la réalisation des différentes opérations arithmétiques de base.

Dans une représentation numérique à point fixe, la variable est représentée par un vecteur de données. Chacun des éléments du vecteur est appelé un bit. Le nombre de bits n est appelé la précision de la représentation [3.11]. Pour effectuer des opérations arithmétiques sur les nombres à virgule fixe au niveau des algorithmes, une représentation spécifique de nombre est nécessaire. Les nombres les plus simples à représenter sont les entiers, alors nous commençons par représenter un nombre entier et positif. Après nous aborderons les différentes représentations d'un nombre réel et négatif.

3.4.1 Nombre entier positif

La représentation numérique suit le même principe que la représentation décimale qu'on connait, où le nombre est représenté par un vecteur qui comporte *n* chiffres décimaux. En général un nombre décimal est représenté comme suite [3.12]: $D = d_{n-1}\ d_{n-1} \ldots d_1\ d_0$

Ce qui donne un nombre entier qui a une valeur égale a :

$$V(D) = d_{n-1}*10^{n-1} + d_{n-2}*10^{n-2} \ldots\ldots d_1*10^1 + d_0*10^0 \qquad (3.1)$$

Dans le cas de la représentation numérique, un nombre est représenté sous la forme suivante :

$$B = b_{n-1}\ b_{n-1} \ldots b_1\ b_0 \qquad (3.2)$$

où b_i est un nombre binaire.

La valeur du nombre entier est calculée à l'aide de la relation suivante :

$$V(B) = b_{n-1}*2^{n-1} + b_{n-2}*2^{n-2} \ldots\ldots b_1*2^1 + b_0*2^0 \qquad (3.3)$$

$$V(B) = \sum_{i=0}^{n-1} b_i * 2^i \qquad (3.4)$$

où *n* est le nombre de bits utilisé, b_0 est le bit le moins significatif, et b_{n-1} est le plus significatif.

Il faut noter que la représentation à point fixe peut représenter un intervalle limité de nombres entiers. Le nombre de bits est l'image de l'intervalle que la représentation peut couvrir. En général n-bits peuvent représenter les nombres entiers qui appartiennent à

l'intervalle [0, 2^n-1]. Par exemple, le maximum qu'une représentation à 3 bit peut représenter est (111 binaire = 7 décimal).

3.4.2 Nombre entier négatif

Le vecteur qui représente un nombre entier négatif est divisé sur deux parties: la première partie qu'est le bit le plus significatif représente le signe du nombre, et la deuxième que sont les *n-1* bits restants représente le module du nombre entier [3.13]. Alors la relation (3.4) devient :

$$B = \begin{cases} \sum_{i=0}^{n-2} b_i * 2^i & B \geq 0 \\ -\sum_{i=0}^{n-2} b_i * 2^i & B < 0 \end{cases} \qquad (3.5)$$

Dans ce cas, n-bits peuvent représenter les nombres entiers qui appartiennent à l'intervalle [-(2^{n-1}-1), (2^{n-1}-1)]. Cette représentation a l'avantage de détecter facilement un débordement lors d'une opération arithmétique: il suffit de mettre en œuvre un algorithme qui supervise le signe du résultat. Cependant, cette représentation complique l'exécution des opérations arithmétiques vu que le bit de signe doit être traité séparément des bits de magnitude. Par exemple, considérons l'addition des deux nombres +18 (00010010) et -19 (10010011) en utilisant la représentation « entier négatif ». Les deux nombres ont des signes différents, les résultats alors devrait porter le signe du nombre le plus grand, dans ce cas le (-19), et être (10000001) ce qui n'est pas le cas si on l'applique de cette façon :

00010010	10010011
+ 10010011	- 00010010
100100101	100000001

Dans ce cas, on est obligé de modifier l'algorithme de l'addition pour avoir un résultat logique.

Pour éviter de compliquer les algorithmes des opérations de base, on préfère utiliser une autre représentation des nombres négatifs. Deux représentations sont connues: « *complément à un* » et « *complément à deux* ». Pour notre travail, on a opté pour la deuxième vu la simplicité qu'elle offre au niveau des opérations arithmétiques.

3.4.3 Complément à deux

Cette représentation consiste aussi à représenter le signe par le bit le plus significatif. Cependant dans le cas où le nombre est négatif on ne change pas seulement le signe, les n-1 bits restants représenteront le complément à deux de la magnitude [3.13].

3.4.4 Représentation des données numériques

Dans ce travail nous essayons de ramener toutes les grandeurs qui sont présentes dans la loi de commande à varier entre $[-1, +1]$. En arrivant à faire ça, on évite la complexité de représenter un nombre réel; la virgule fictive sera placée juste après le bit du signe.

Toutes les données dépassant cet intervalle seront forcées à la valeur de l'intervalle la plus proche.

La figure 3.7 montre un exemple.

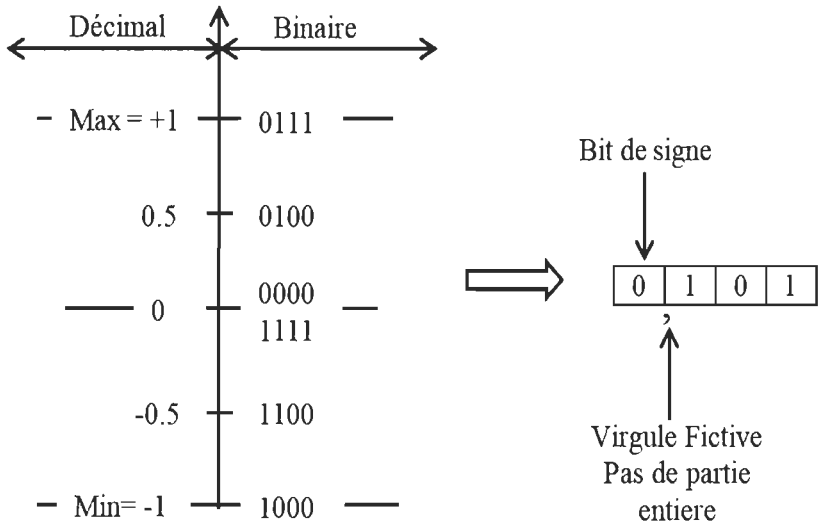


Figure 3.7 Représentation des données.

Même en utilisant la représentation complément à deux, qui d'après la plupart des chercheurs est la méthode la plus appropriée pour l'intégration des lois de commande, pour représenter les nombre négatifs, on risque toujours d'obtenir des faux résultats lors de l'exécution des opérations si on ne prend pas en considération le problème du débordement.

3.4.5 Modélisations des opérations de base

Avant de procéder à modéliser les architectures des différents modules utilisés dans les lois de commande, on tient à présenter l'architecture détaillée de chacune des opérations de base qu'on peut utiliser le long de ce travail.

Le processus de vérification sera le même pour les différentes opérations.

- On commence par générer des données aléatoires en virgule flottante à l'aide de Matlab;

- On transforme ces données en virgule fixe, ensuite on compare les deux pour s'assurer que l'erreur maximale est toujours inférieure ou égale à l'erreur correspondant à un bit;
- Une fois la transformation virgule flottante => virgule fixe vérifiée, on exécute la partie qui transforme les données virgule fixe en données binaires;
- Les données binaires vont être transférées et après récupérées par le programme VHDL qui décrit l'architecture de l'opération;
- Les résultats qu'on obtient du programme VHDL vont être comparés à la fin avec les résultats obtenus par l'exécution de la même opération en virgule fixe et flottante; on essaye toujours de fixer l'erreur résultante d'un bit comme critère.

Les instructions utilisées pour le transfert entre les deux langages ainsi que les instructions qui font la conversion des données sont notées dans les Annexes.

3.4.5.1 Modélisation de l'additionneur à 16 BITS

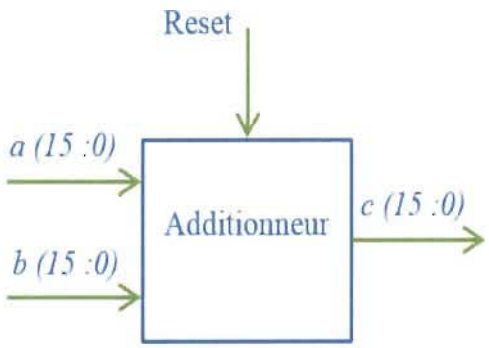


Figure 3.8 Bloc d'addition.

La figure 3.8 représente l'architecture externe d'un additionneur sur une carte FPGA, où

- **a** et **b** sont deux entrées sur 16 bits;
- **c** est une sortie sur 16 bits ;
- **Reset** : mise à zéro (initialisation).

Pour prendre en compte la possibilité du débordement, on a mis en œuvre un algorithme de saturation comme montré sur la figure 3.9.

```
33
34        if C_MSim1(15)='1' and A_MSim(15)='0' and B_MSim(15)='0' then
35             C_MSim1 :="0111111111111111";
36
37
38         elsif C_MSim1(15)='0' and A_MSim(15)='1' and B_MSim(15)='1' then
39             C_MSim1 :="1000000000000000";
40        end if;
41     end if;
```

Figure 3.9 Prise en compte du débordement sur VHDL pour un additionneur.

Les résultats de simulation d'un additionneur à 16bits, à l'aide du Matlab et ModelSim, sont présentés sur la figure 3.10.

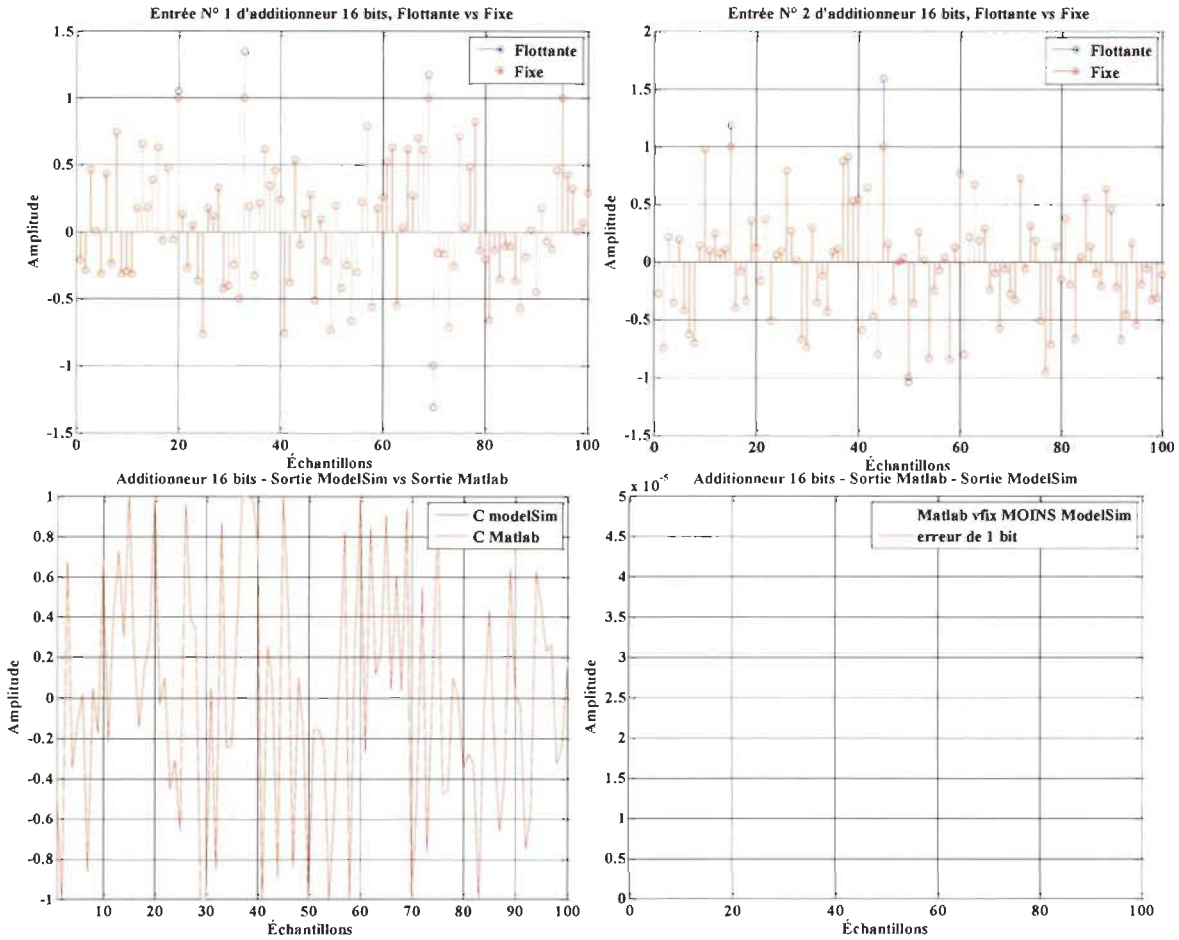


Figure 3.10 Résultats d'un additionneur à 16 bits.

Sur les deux premières figures on présente les deux entrées d'additionneur, en virgule fixe et flottante. La transformation des données en virgule fixe ne déforme pas ces données, sauf les échantillons qui dépassent l'intervalle [*-1, 1*]. Les deux dernières figures présentent une comparaison entre les sorties d'additionneur. Sur la dernière, on remarque que l'erreur entre le modèle sous Matlab celui avec ModelSim n'atteint même pas l'erreur tolérée, correspondant à un bit.

3.4.5.2 Modélisation du multiplieur à 8 BITS

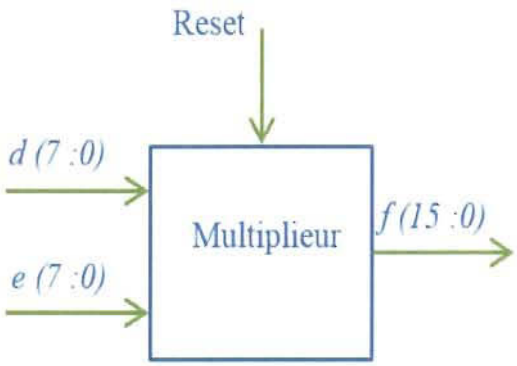


Figure 3.11 Bloc de Multiplication.

La figure 3.11 représente l'architecture externe d'un multiplieur sur une carte FPGA, où

- ***d*** *et* ***e*** sont deux entrées sur 8 bits;
- ***f*** est une sortie sur 16 bits ;
- ***Reset*** : mise à zéro (initialisation).

Vu que les entrées ***d*** *et* ***e*** $\in$ [*-1, 1*], la sortie ***f*** ne risque pas de déborder de cette intervalle, donc il n'y a aucune utilité d'une prise en compte de débordement comme avec l'additionneur et le soustracteur.

Les résultats obtenus sont représentés sur la figure 3.12.

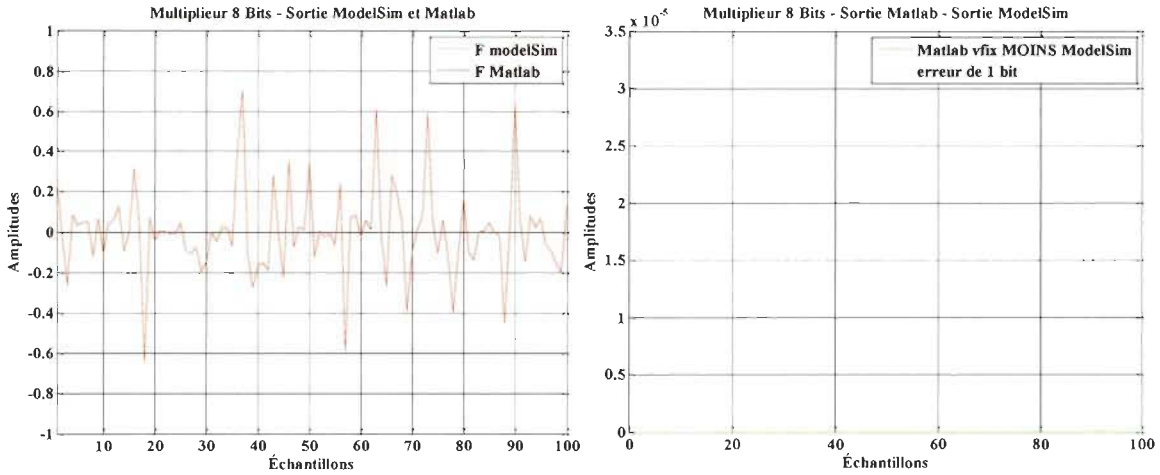


Figure 3.12 Résultats d'un multiplieur à 8 bits.

L'erreur entre la sortie Matlab et la sortie ModelSim est toujours inférieure à l'erreur tolérée.

3.4.5.3 Modélisation du soustracteur à 16 BITS

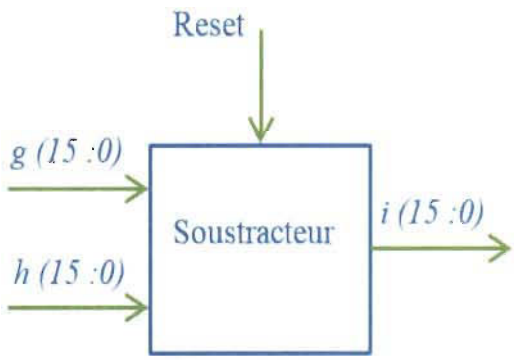


Figure 3.13 Bloc de Soustraction.

Le figure 3.13 représente l'architecture externe d'un soustracteur sur une carte FPGA, où

- ***g*** et ***h*** sont deux entrées sur 16 bits;
- ***i*** est une sortie sur 16 bits ;

- ***Reset*** : mise à zéro (initialisation).

Pour prendre en compte la possibilité du débordement on a mis en œuvre un algorithme de saturation comme montré sur la figure 3.14.

```
37
38              if temp1(15)='0' and Sous1(15)='1' and Sous2(15)='0' then
39              temp1 :="1000000000000000";
40
41
42              elsif temp1(15)='1' and Sous1(15)='0' and Sous2(15)='1' then
43              temp1 :="0111111111111111";
44               end if;
45
```

Figure 3.14 Prise en compte du débordement sur VHDL pour un soustracteur.

L'exécution des deux modèles, Matlab et ModelSim, donne les résultats montrés sur la figure 3.15, où on peut remarquer que l'erreur entre les deux est toujours moins que l'erreur qu'on peut tolérer.

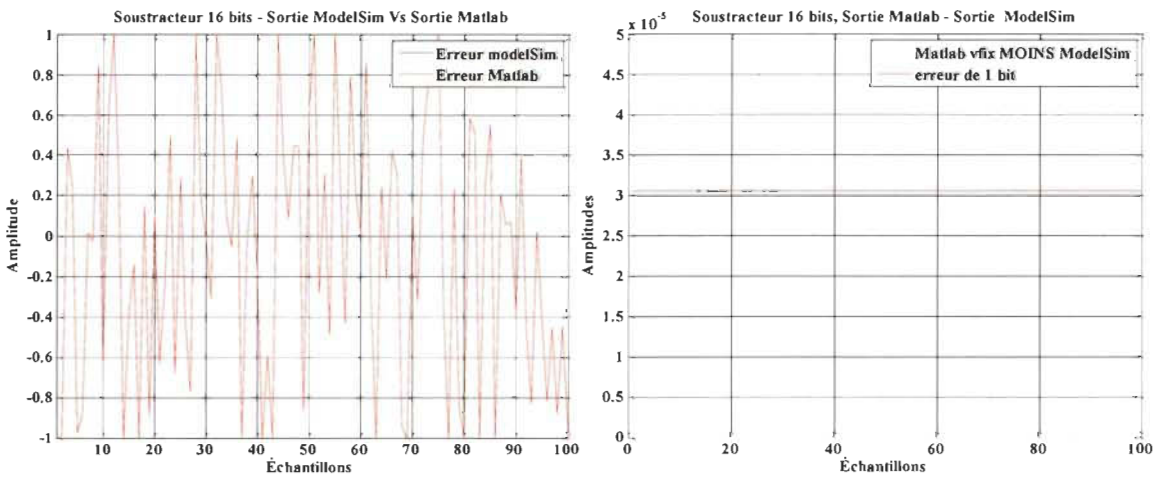


Figure 3.15 Résultats d'un soustracteur à 16 bits.

3.4.5.4 Modélisation du registre

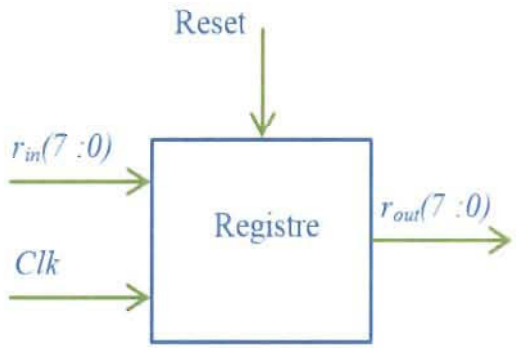


Figure 3.16 Bloc de décalage.

La figure 3.16 représente l'architecture externe d'un registre à décalage sur une carte FPGA, où

- r_{in} est une entrée 8 bits;
- ***Clk*** est l'horloge;
- r_{out} est une sortie 8 bits;
- ***Reset*** : mise à zéro (initialisation)

Comme il est reconnu, un registre décale la transmission d'une donnée par une période d'horloge. Donc r_{in} apparaît en sortie, r_{out}, à chaque front montant de ***clk*** comme il est représenté sur la figure 3.17.

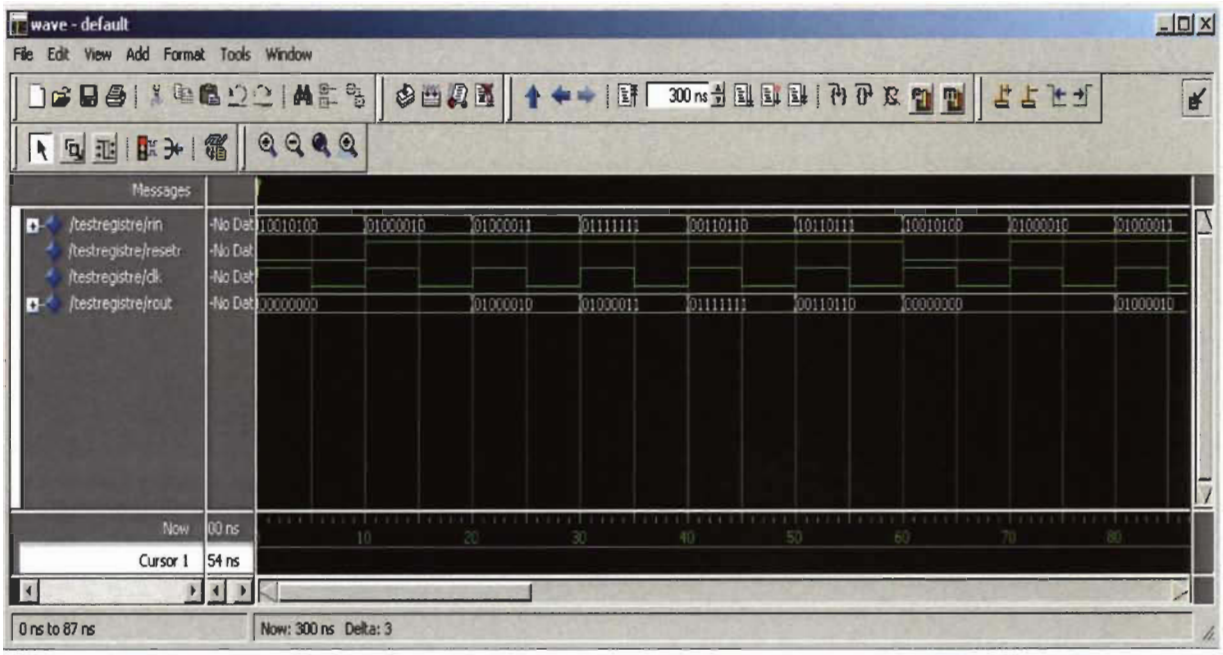


Figure 3.17 Résultats d'un registre à 8 bits.

3.4.5.5 Modélisation du troncateur

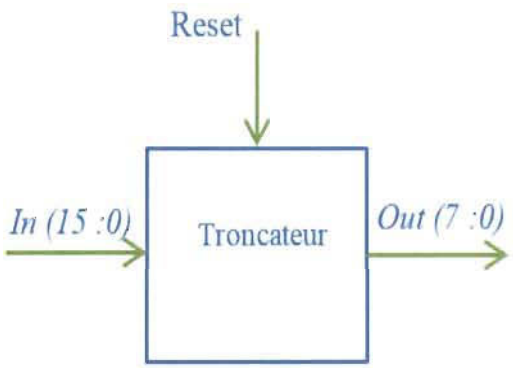


Figure 3.18 Bloc de troncation.

La figure 3.18 représente l'architecture externe d'un troncateur sur une carte FPGA, où

- ***In*** est une entrée à 16 bits;
- ***Out*** est une sortie à 8 bits.

Le troncateur a comme rôle de réduire la précision d'une donnée en coupant les *N/2* bits les moins significatifs.

Les résultats VHDL obtenus d'un toncateur 16bits-à-8bits sont représentés sur la figure 3.19.

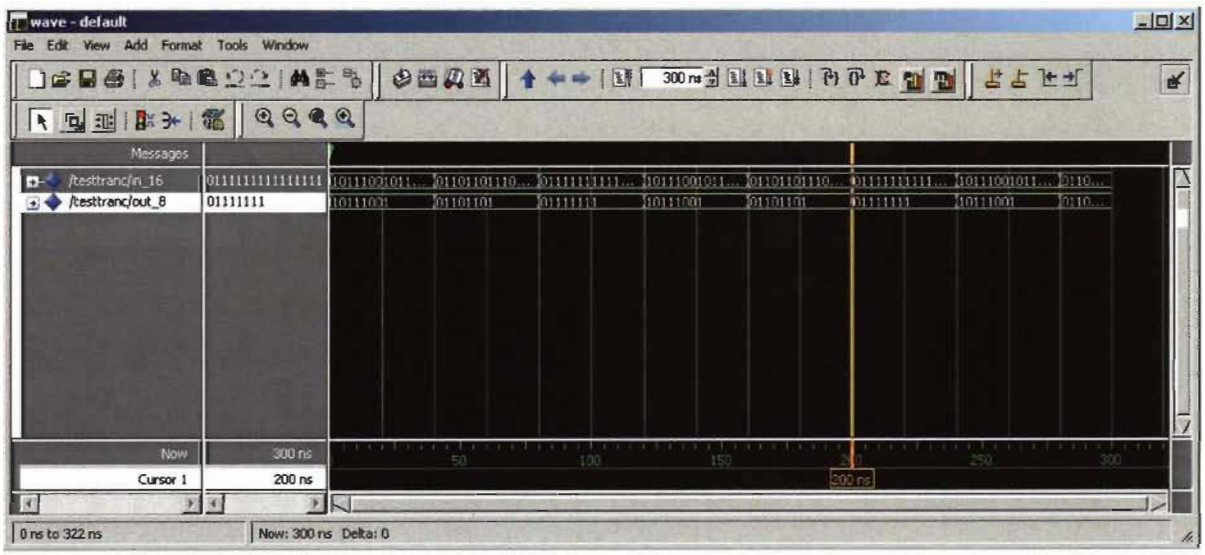


Figure 3.19 Résultats d'un troncateur 16bits-à-8bits.

3.5 Conclusion

Dans ce chapitre, après avoir présenté un survol sur la technologie des *FPGA,* nous nous sommes focalisés sur le côté algorithmique. La méthode de représentation numérique choisie a été réalisée sur MATLAB. A l'aide de ce code, des données numériques ont été fournies pour les modules de base afin de les valider. Les résultats présentés montrent une précision moins que l'erreur qui correspond à un seul bit. Cela va nous permettre d'éviter que l'erreur s'accumule après l'assemblage des modules pour concevoir le module global de contrôle.

Chapitre 4 - Systèmes d'entrainement électromécanique

4.1 Introduction

Les systèmes d'entrainement électromécanique reçoivent une croissance d'intérêt par la plupart des applications, que ce soit domestiques ou industrielles [4.1]. Ces applications ont chacune des exigences particulières de vitesse, de couple ou même de qualité d'énergie. Pour faire face à ces exigences, une grande variété de solutions a été proposée pour faire face à ces exigences [4.2]. La croissance du marché des systèmes d'entrainement électromécanique est illustrée dans l'histogramme de la figure 4.1 [4.3].

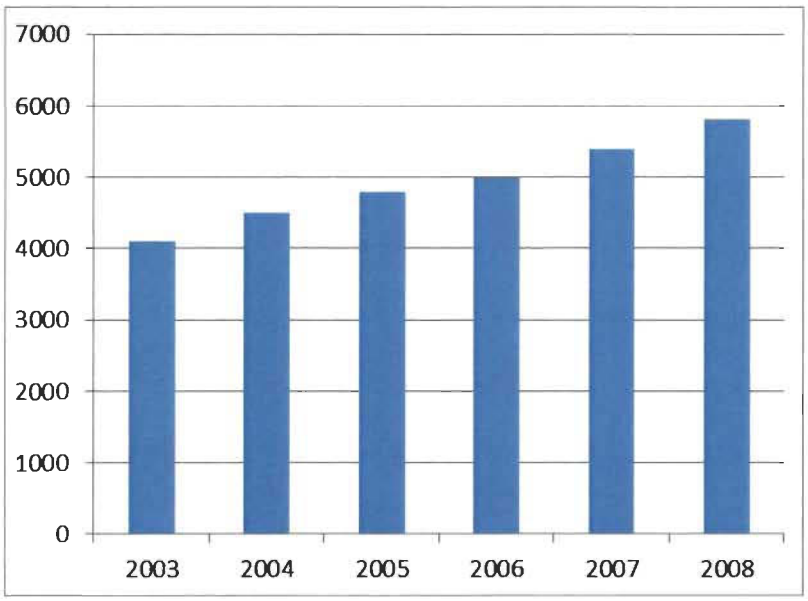


Figure 4.1 Croissance du marché des systèmes d'entrainement électromécanique (Million $/ Année) [4.3].

Un dispositif d'entrainement électromécanique, comme représenté sur la figure 4.2, permet d'entrainer et de contrôler une charge mécanique ou même un procédé mécanique

en transformant l'énergie électrique en une énergie mécanique le plus efficacement possible [4.4].

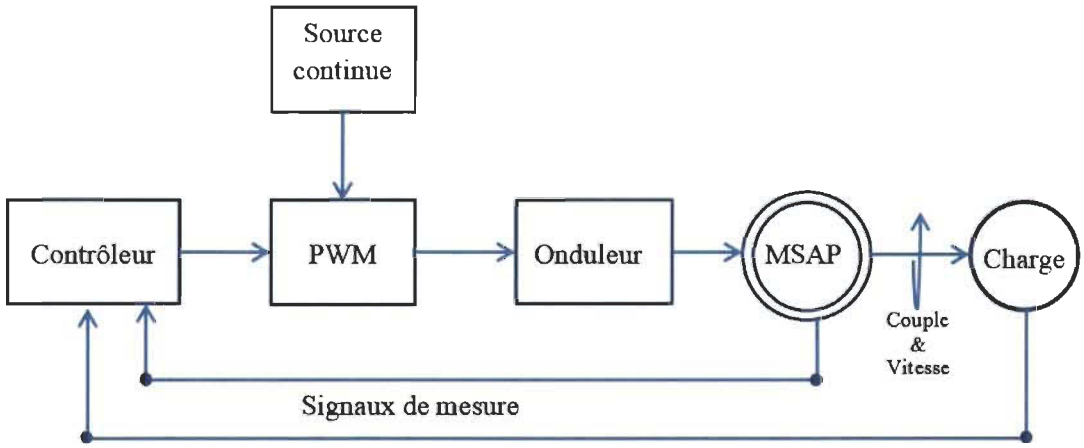
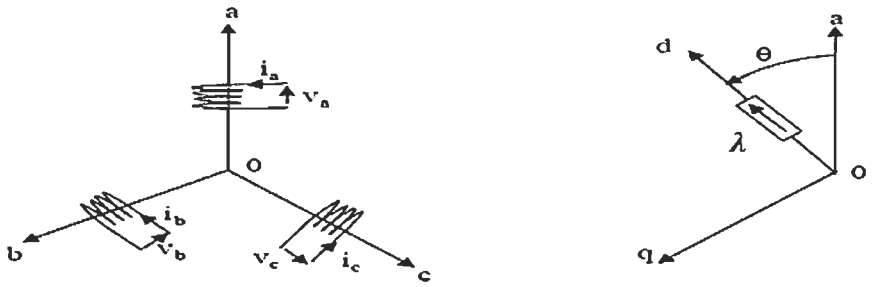


Figure 4.2 Dispositif d'entrainement électromécanique typique.

4.2 Machines Synchrones à Aimant Permanent (MSAP)

Tout comme la machine à induction, la machine synchrone a un stator avec enroulements triphasés mais l'enroulement du rotor est remplacé par un aimant permanent. Donc, les équations électriques peuvent être déduites de la même façon que la machine à induction à partir du schéma qui représente les enroulements statoriques et rotoriques.

4.2.1 Équations générales de la machine synchrone à aimant permanent



Stator Rotor

Figure 4.3 Représentation de la machine synchrone à aimant permanent.

A partir de la figure 4.3, qui représente le schéma symbolique des enroulements d'une machine synchrone triphasée à aimants permanents, on écrit les équations en notation matricielle qui représentent ces circuits :

$$\left[V_s\right] = \left[R_s\right]\ \left[i_s\right] + \frac{d\phi_s}{dt} \tag{4.1}$$

$$\phi_s = \left[L_s\right]\ \left[i_s\right] + \left[\lambda\right] \tag{4.2}$$

où : λ est le flux magnétique permanent

$[L_s] = [L_{ss}] + [L_m]$: Matrice d'inductance du stator

$\left[V_s\right] = \left[V_a \quad V_b \quad V_c\right]^T$: Vecteur de tension statorique

$\left[i_s\right] = \left[i_a \quad i_b \quad i_c\right]^T$: Vecteur de courant statorique

$\left[\phi_s\right] = \left[\phi_a \quad \phi_b \quad \phi_c\right]^T$: Vecteur de flux statorique

$$[R_S]=\begin{bmatrix} R_S & 0 & 0 \\ 0 & R_S & 0 \\ 0 & 0 & R_S \end{bmatrix}$$: Matrice résistance du stator

$$[L_{ss}]=\begin{bmatrix} L_a & M_{ab} & M_{ac} \\ M_{ab} & L_b & M_{bc} \\ M_{ac} & M_{bc} & L_c \end{bmatrix} \tag{4.3}$$

$$[L_m] = Lm\begin{bmatrix} \cos(2\theta) & \cos 2(\theta-\frac{2\pi}{3}) & \cos 2(\theta-\frac{4\pi}{3}) \\ \cos 2(\theta-\frac{2\pi}{3}) & \cos 2(\theta-\frac{4\pi}{3}) & \cos(2\theta) \\ \cos 2(\theta-\frac{4\pi}{3}) & \cos(2\theta) & \cos 2(\theta-\frac{2\pi}{3}) \end{bmatrix} \tag{4.4}$$

$$[X_{dqo}] = [T_\theta][X_{abc}] \tag{4.5}$$

où

X représente soit un courant, une tension ou un flux,

θ représente la position du rotor.

Les termes X_d, X_q représentent les composantes longitudinale et transversale, respectivement du vecteur X.

Pour simplifier la représentation du modèle mathématique de la MSAP, on doit passer du repère triphasé au repère biphasé en appliquant la transformée de Park.

$$[T_\theta] = \frac{2}{3}\begin{bmatrix} \cos(\theta) & \cos(\theta-\frac{2\pi}{3}) & \cos(\theta-\frac{4\pi}{3}) \\ -\sin(\theta) & -\sin(\theta-\frac{2\pi}{3}) & -\sin(\theta-\frac{4\pi}{3}) \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \tag{4.6}$$

Après l'application de (4.6) qui représente la transformée de Park dans un repère lié au rotor, on obtient un système d'équations plus simple pour modéliser une MSAP.

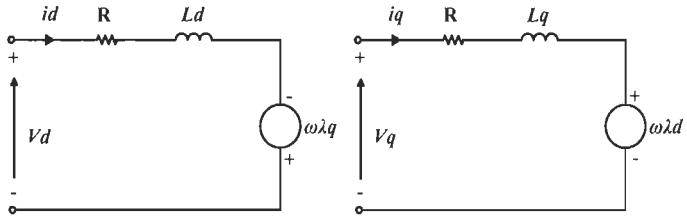


Figure 4.4 Circuit équivalent de la MSAP.

$$V_d = Ri_d + L_d\frac{d}{dt}i_d - L_q p\omega i_q \tag{4.7}$$

$$V_q = Ri_q + L_q\frac{d}{dt}i_q + L_d p\omega i_d + p\lambda\omega \tag{4.8}$$

$$\tau = \frac{3}{2}p\left[(L_d - L_q)i_d i_q + \lambda i_q\right] \tag{4.9}$$

$$\frac{d}{dt}\omega = \frac{1}{J}(\tau - \tau_f - \tau_L) \tag{4.10}$$

où

V_d, V_q Tensions dans le repère d-q

i_d, i_q Courants dans le repère d-q

L_d, L_q Inductances dans le repère d-q

R Résistance statorique

p Nombre de paires de pôles

λ Flux de l'aimant permanent

λ_d, λ_q Flux dans le repère d-q

τ Couple électromagnétique

ω Vitesse mécanique du rotor

J Inertie

τ_f Couple de frottement

τ_L Couple de charge.

4.3 Onduleur triphasé à deux niveaux

Pour les applications d'entrainement électrique, les onduleurs de source de tension sont les plus utilisés. Il existe plusieurs topologies d'onduleur triphasé dans la littérature. Chacune de ces topologies répond aux critères et aux exigences de certaines applications comme: complexité, puissance et qualité de l'onde. Cependant, avec l'apparition des méthodes de modulation vectorielle, l'obtention d'une source de tension alternative puissante avec un taux d'harmonique faible, à l'aide d'un onduleur à deux niveaux, est devenu possible.

4.3.1 Topologie d'un onduleur triphasé à deux niveaux

Le schéma structurel d'un onduleur triphasé à deux niveaux et de sa charge est illustré par la Figure 4.5 Chaque groupe transistor–diode assemblé en parallèle forme un interrupteur bi-commandable.

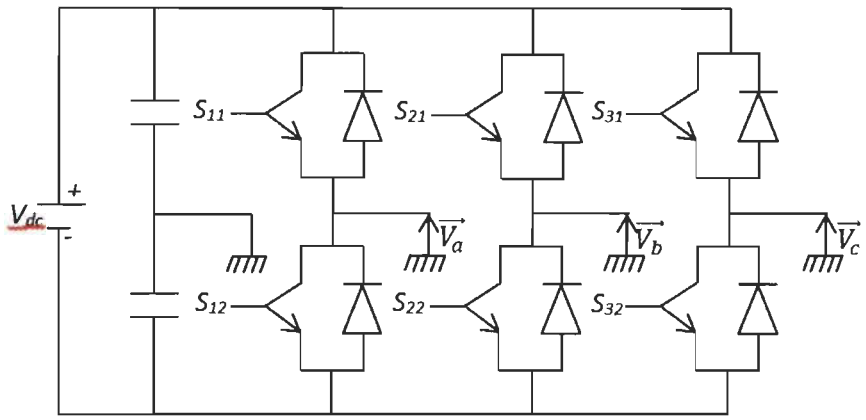


Figure 4.5 Onduleur triphasé à deux niveaux.

Les trois tensions d'alimentation V_a, V_b et V_c sont représentées dans le plan complexe, par un seul vecteur d'espaces $\vec{V_s}$ défini par la relation (4.11) [4.5]:

$$\vec{V_s}(t) = \overline{V_a}(t)e^{j0} + \overline{V_b}(t)e^{j2\pi/3} + \overline{V_c}(t)e^{j4\pi/3} \qquad (4.11)$$

Le vecteur tension $\vec{V_s}$ est délivré par un onduleur de tension triphasé, dont l'état des interrupteurs, supposés parfaits, est représenté en théorie par trois grandeurs booléennes de commande.

$S_a = 1$: Interrupteur haut est fermé et celui en bas est ouvert.

$S_a = 0$: Interrupteur haut est ouvert et celui en bas est fermé.

À partir de la combinaison des 3 grandeurs (S_a, S_b et S_c), le vecteur de tension $\vec{V_s}$ peut se retrouver dans huit positions fixes correspondant aux huit configurations possibles des

interrupteurs [4.6]. Ces huit états du vecteur d'espace définissent les limites de 6 secteurs dans le plan complexe. Deux des huit états sont des vecteurs nuls : $\overrightarrow{V_0}$ et $\overrightarrow{V_7}$ [4.7, 4.6].

Tableau 4-1 Vecteurs de tensions pour chaque combinaison possible des interrupteurs.

	S_a	S_b	S_c	V_α	V_β
$\overrightarrow{V_0}$	0	0	0	0	0
$\overrightarrow{V_1}$	1	0	0	$2V_{dc}/3$	0
$\overrightarrow{V_2}$	1	1	0	$V_{dc}/3$	$V_{dc}/\sqrt{3}$
$\overrightarrow{V_3}$	0	1	0	$-V_{dc}/3$	$V_{dc}/\sqrt{3}$
$\overrightarrow{V_4}$	0	1	1	$-2V_{dc}/3$	0
$\overrightarrow{V_5}$	0	0	1	$-V_{dc}/3$	$-V_{dc}/\sqrt{3}$
$\overrightarrow{V_6}$	1	0	1	$V_{dc}/3$	$-V_{dc}/\sqrt{3}$
$\overrightarrow{V_7}$	1	1	1	0	0

4.3.2 Technique SVPWM

La technique de modulation vectorielle (SVPWM) consiste à reconstituer le vecteur tension de référence pendant une période d'échantillonnage T_s par les vecteurs tensions adjacents correspondant aux huit états possibles de l'onduleur.

On se place par exemple dans le cas où le vecteur de référence est situé dans le secteur 1 (Figure 4.6). Dans ce cas, la tension de référence $\overrightarrow{V_s}$ est reconstituée en faisant une moyenne temporelle des tensions $\overrightarrow{V_1}$ et $\overrightarrow{V_2}$.

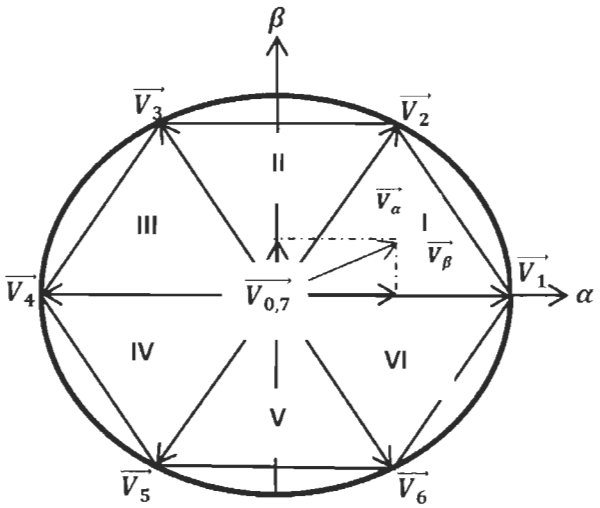


Figure 4.6 Vecteurs d'espace de tension.

Après, il suffit de déterminer la position du vecteur de référence $\overrightarrow{V_s}$ dans le repère (α, β) et le secteur i dans lequel il se trouve. Pour une fréquence de commutation $1/T_s$ suffisamment élevée, le vecteur d'espace de référence est considéré constant pendant un cycle de commutation [4.8, 4.9].

$$\overrightarrow{V_s} = \frac{t_1}{T_s}\overrightarrow{V_1} + \frac{t_2}{T_s}\overrightarrow{V_2} \tag{4.12}$$

avec :

t_1: Temps d'application du vecteur $\overrightarrow{V_1}$

t_2: Temps d'application du vecteur $\overrightarrow{V_2}$

T_s: Période de commutation de l'onduleur.

Dans le cas où $(t_1+t_2 < T_s)$, on doit compléter par les vecteurs nuls ($\overrightarrow{V_0}$ et $\overrightarrow{V_7}$).

Alors :

$$\overrightarrow{V_s} = \frac{t_1}{T_s}\overrightarrow{V_1} + \frac{t_2}{T_s}\overrightarrow{V_2} + \frac{t_0}{2T_s}\overrightarrow{V_0} + \frac{t_0}{2T_s}\overrightarrow{V_7} \tag{4.13}$$

t_0: Temps d'application des deux vecteurs nuls $\overrightarrow{V_0}$ et $\overrightarrow{V_7}$.

Après décomposition sur les deux axes du plan complexe (α, β) [4.10]:

$$T_1 = \frac{\sqrt{3}V_s}{V_{dc}} T_s \sin\left(\frac{\pi}{3} - \varphi\right) \tag{4.14}$$

$$T_2 = \frac{\sqrt{3}V_s}{V_{dc}} T_s \sin(\varphi) \tag{4.15}$$

$$T_0 = \frac{1}{2}(T_s - T_1 - T_2) \tag{4.16}$$

φ : est l'angle entre les vecteurs $\overrightarrow{V_1}$ et $\overrightarrow{V_s}$.

Les mêmes règles s'appliquent pour les secteurs 2 jusqu'à 6.

L'équilibrage des commandes nous permet de réduire au minimum les pertes par commutations et de diminuer le taux d'harmonique; ce sont encore d'autres avantages de la MLI vectorielle.

Afin de diminuer le taux d'harmonique, il faut respecter les deux conditions suivantes :

— Tous les interrupteurs de chaque demi-bras doivent avoir le même état au milieu et aux extrémités de la période.

— Les impulsions de commande des interrupteurs sont centrées au milieu de la période de commutation [4.4].

Sur la Figure 4.7 on présente la combinaison lorsque le vecteur est situé dans le secteur 1 délimité par $\overrightarrow{V_1}$ et $\overrightarrow{V_2}$.

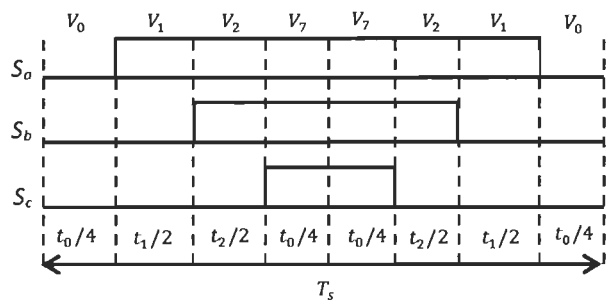


Figure 4.7 Forme des Signaux PWM dans le secteur I.

Le secteur est généralement déterminé par l'angle α où $\alpha = \tan^{-1}\left(\frac{V_\beta}{V_\alpha}\right)$. Dans ce travail le secteur est déterminé par une méthode simple basée sur les tensions V_α, V_β. La détermination est faite comme dans le tableau 4.2, où A_0 est le signe de V_β, A_1 égale à 1 si $V_\beta > \sqrt{3}\, V_\alpha$ sinon égale à 0, et A_2 égale à 1 si $V_\beta > -\sqrt{3}\, V_\alpha$ sinon égale à 0.

Tableau 4-2 Détection du secteur.

A_0	A_1	A_2	Secteur
0	0	0	5
0	0	1	6
0	1	0	4
1	0	1	1
1	1	0	3
1	1	1	2

4.3.3 Simulation et discussion

L'onduleur a été modélisé à l'aide de Simulink™ / Matlab® utilisant la structure représentée sur la figure 4.8.

L'algorithme de SVPWM a été décrit en VHDL et synthétisé avec le logiciel Xilinx ISE Design (Suite 13,4) (figures 4.9 et 4.10). Le modèle VHDL a été évalué en cosimulation avec ModelSim et Simulink™ / Matlab®. ModelSim exécute tous les calculs de la

SVPWM avec une précision de mot égale à 8 bits, de la génération du vecteur de tension de référence à la transformation de Concordia, la détection du secteur, calcul des temps de commutation, jusqu'à la génération des signaux de commutation.

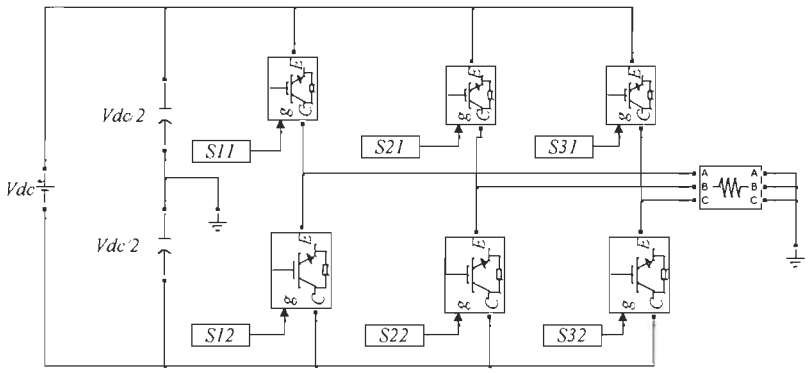


Figure 4.8 RTL Schéma fonctionnel de l'onduleur sur Simulink.

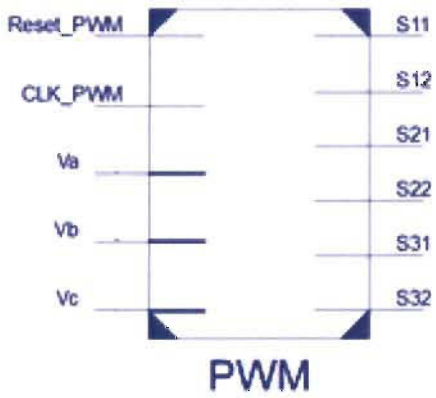


Figure 4.9 Entrées/sorties du circuit SVPWM réalisé sur une FPGA (CLK_PWM=Ts).

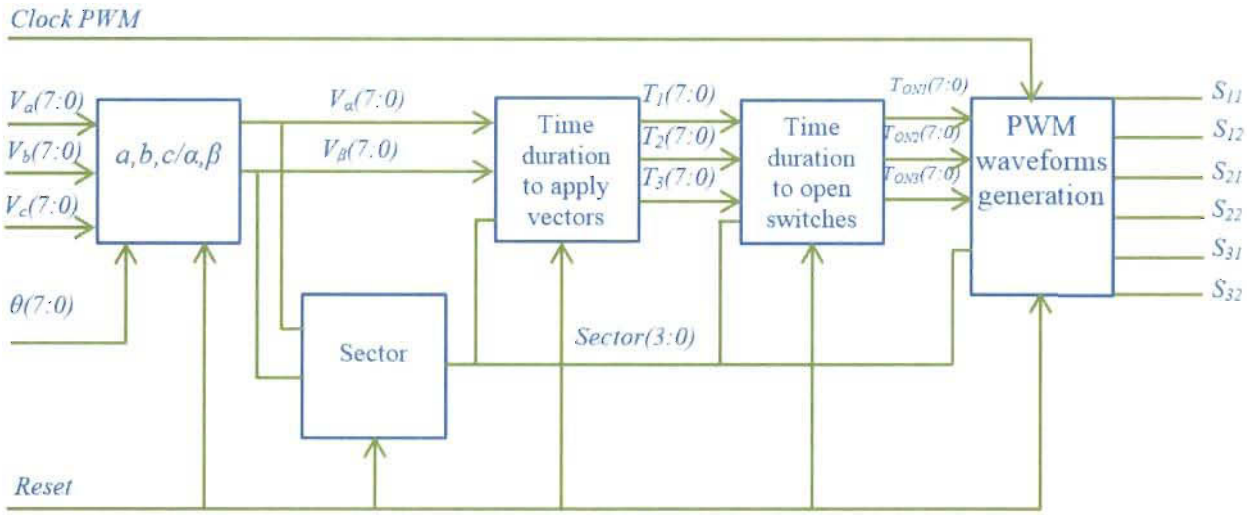


Figure 4.10 Schéma du circuit SVPWM réalisé sur FPGA.

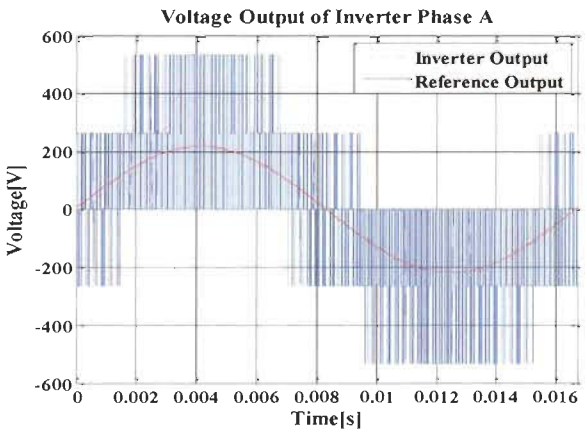


Figure 4.11 Tension de sortie Phase A.

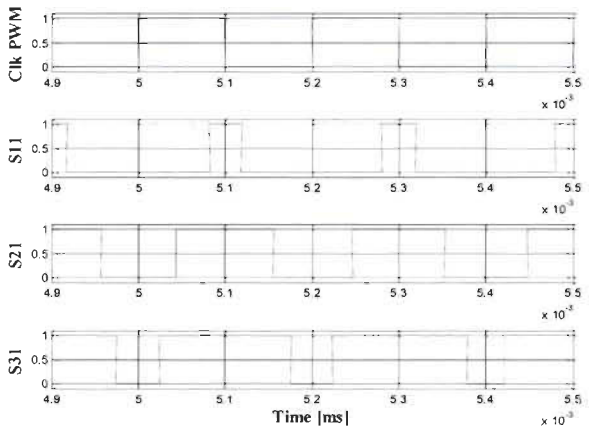


Figure 4.12 Signaux PWM pour trois périodes Ts.

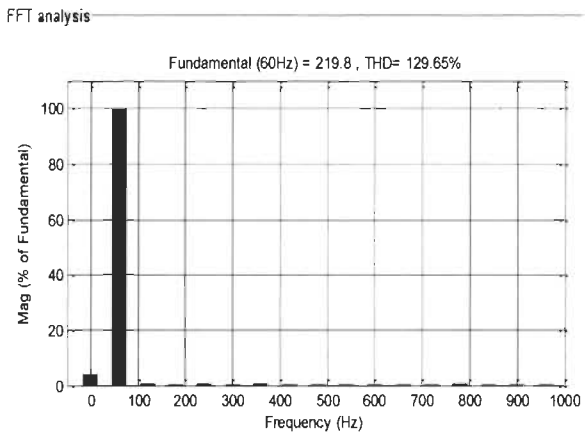


Figure 4.13 Représentation des harmoniques d'une phase.

Tableau 4-3 Résumé de l'utilisation des ressources pour l'implémentation de la SVPWM.

Spartan III			
	Utilisé	Disponible	Utilisation
Number of slices	358	23872	1.5%
Number of slices Flip Flops	110	47744	0.2%
Number of input LUTs	666	47744	1.4%
Number of bonded IOBs	32	469	6.8%
Number GCLKs	2	24	8.3%
Number DSP 48s	11	126	8.7%

La figure 4.11 montre la forme d'onde de tension (phase A) à la sortie de l'onduleur (avec une charge résistive) et le signal de référence de 60 Hz envoyé au modèle VHDL.

Les signaux de commande sont représentés sur la figure 4.12 sur laquelle on peut constater que les contraintes de forme d'onde définies sur la figure 4.7 sont respectées. Du coup, on peut remarquer sur la figure 4.13 le spectre d'harmonique de la sortie qui est excellent: l'amplitude de la fréquence fondamentale est égale à 219.8V, une erreur de 0.1% par rapport au signal de référence 220V, ce qui est acceptable étant donné que la commande serait en boucle fermée, donc elle ne sera pas affectée par une telle petite erreur de 0,1%. Le spectre harmonique n'a pas été affecté par la précision des données.

4.4 Approximation de la SVPWM

La SVPWM est l'algorithme le plus sophistiqué et le plus approprié pour les lois de contrôle à haute performance, en raison de sa capacité à générer une tension alternative avec amplitude variable pour le même bus à courant continu [4.11] et pour sa capacité à minimiser les harmoniques et les pertes aux niveaux des commutateurs [4.12]. Cependant, cet algorithme nécessite classiquement des transformations linéaires et un grand nombre d'équations ayant des fonctions trigonométriques, ce qui augmente le coût et le temps de calcul algorithmique. Du coup la majorité des ressources de calcul, sur une carte, est utilisée pour générer les signaux PWM, laissant moins de ressources pour les boucles de commande.

Vu que les critères de performance des contrôleurs deviennent plus strictes, les algorithmes de contrôle sont devenus complexes et en général ils ne sont pas adaptés à la mise en œuvre directe, ainsi ils ont besoin d'être retravaillés [4.13]. La mise en œuvre des

boucles de régulation et du générateur PWM consomme plus de ressources et nécessite une vitesse d'exécution élevée pour exécuter les fonctions complexes en particulier celles de la SVPWM [4.14]. Il existe plusieurs approches qui intègrent plus qu'une carte pour mettre en œuvre une loi de contrôle. De l'autre côté, il y a plusieurs chercheurs qui ont opté pour la simplification des algorithmes.

Dans [4.15], les auteurs proposent une comparaison entre la performance de la mise en œuvre des techniques SVPWM et SPWM (sinusoïdales) pour trouver un compromis entre la qualité du signal et la simplicité de l'algorithme. Dans [4.16], les auteurs ont proposé une solution à la complexité de la technique SVPWM en utilisant un réseau neuronal artificiel (RNA). Cependant, la conception d'un RNA est un processus itératif, et elle est fondée sur l'application des non-linéarités et d'un grand nombre d'opérations, qui peuvent toutefois être évaluées en parallèle.

La mise en œuvre d'un nouveau schéma approximatif de technique de SVPWM est proposée pour commander un onduleur à deux niveaux triphasé. Les résultats de simulation montreront que cette approximation conserve les avantages de la technique SVPWM, à savoir de minimiser la distorsion harmonique et les pertes de commutation. La synthèse, en utilisant Xilinx ISE, sera présentée et comparée aux résultats obtenus dans la section précédente. Cette nouvelle application permettra de réduire considérablement les ressources consommées.

4.4.1 Algorithme de l'approximation

L'algorithme de SVPWM procède à calculer le temps d'ouverture et de fermeture des commutateurs de puissance à partir des tensions de référence. La durée du temps de

conduction des commutateurs dépend de l'amplitude de la tension de référence et de la tension V_{dc}, et la fréquence de variation dépend de la fréquence de référence.

Pour ce travail, nous avons développé des relations polynomiales directes entre chaque tension de référence et la période de temps de conduction dans les bras du convertisseur.

Pour approcher l'algorithme de SVPWM présenté dans la section précédente, nous avons suivi la méthodologie montrée sur la figure 4.14. En utilisant le modèle Simulink™ / Matlab® développé dans la section précédente, nous générons les temps de commutation pour plusieurs valeurs d'amplitude de référence.

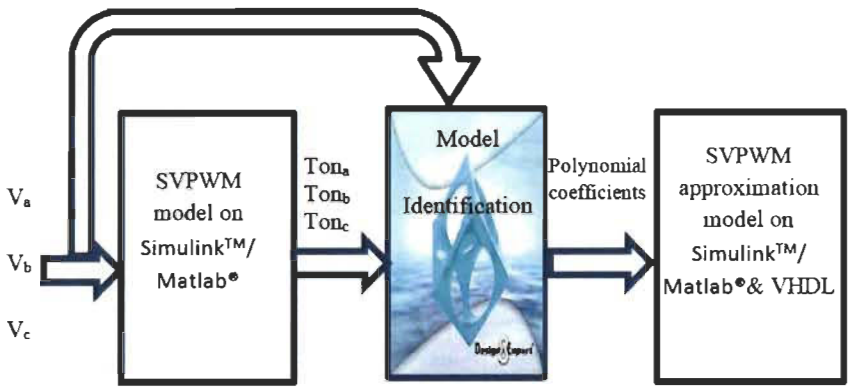


Figure 4.14 Méthodologie utilisée pour identifier le modèle approximatif.

Les paires de données (V_a, T_{ona}), (V_b, T_{onb}) et (V_c, T_{onc}) sont stockées dans le logiciel de conception-Expert®. Des fonctions d'optimisation développées par Design-Expert® à partir des données fournies, sont utilisées pour obtenir un modèle qui se rapproche de la relation entre le signal de référence et la durée du temps de conduction. Ce modèle est représenté par un polynôme du premier ordre pour chaque phase:

$$Ton_{a,b,c} = A * \boldsymbol{V}^{*}_{\boldsymbol{a,b,c}} + B \qquad (4.17)$$

où B est une constante qui est la même pour les trois phases et A est une constante dans le cas où V_{dc} est constante. Cependant, en faisant varier la tension du bus cc, qui est bien représentatif de la pratique, nous constatons que le contenu harmonique dans la sortie du convertisseur varie également. Du coup, nous avons fait varier la tension V_{dc} de *400 V à 1200 V*, tout en ajustant le coefficient A afin d'avoir le taux d'harmonique le plus bas possible. Les valeurs (V_{dc}, A) trouvées sont présentées dans la figure 4.15.

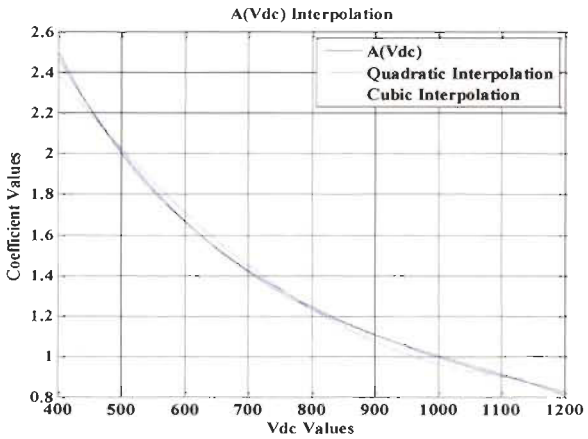


Figure 4.15 Relation entre le coefficient A et la tension V_{dc}.

Selon les résultats obtenus par Design-Expert®, la fonction $A\ (V_{dc})$ la plus appropriée est un polynôme d'ordre 3 de la forme:

$$A = a * V_{dc}^{\ 3} + b * V_{dc}^{\ 2} + c * V_{dc} + d \tag{4.18}$$

4.4.2 Simulations et résultats

L'algorithme d'approximation de SVPWM et l'onduleur ont été modélisés en utilisant Simulink™ / Matlab®.

Dans un premier temps, le modèle proposé est utilisé pour la génération d'un système de tensions triphasées équilibrées. La tension de référence imposée à l'entrée du modèle

approximatif est la même qui a été imposée dans la section précédente: l'amplitude est égale à 220 volts, et la fréquence est de 60 Hz. Sur la figure 4.16, on observe que la tension de sortie de l'onduleur (avec une charge résistive), qui a été contrôlée par le modèle approximatif Simulink™ / Matlab®, est en phase avec la tension de référence. La figure 4.17 montre le spectre de la sortie de tension de l'onduleur. Dans le premier test, l'amplitude de la fréquence fondamentale est égale à 219.2V, une erreur par rapport au signal de référence 220V de 0,4%. En comparant cette erreur à l'erreur obtenue dans la section précédente, qui est égale à 0,1%, on constate que, compte tenu de la simplicité de cette approximation, cette erreur est acceptable étant donné que la commande est en boucle fermée, donc elle ne sera pas affectée par une telle petite erreur.

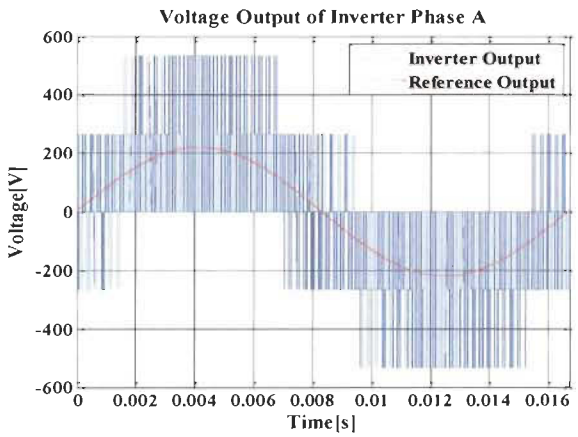


Figure 4.16 Tension de sortie Phase A.

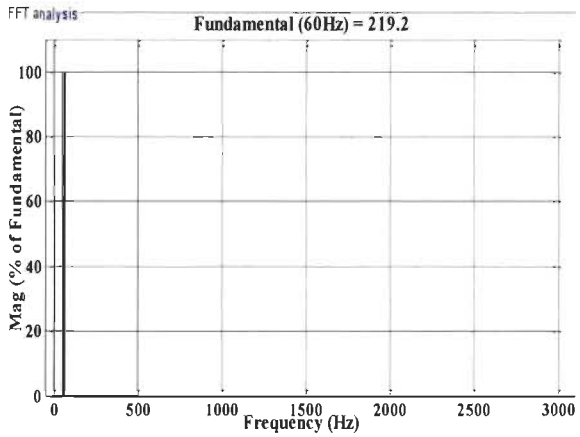


Figure 4.17 Représentation des harmoniques

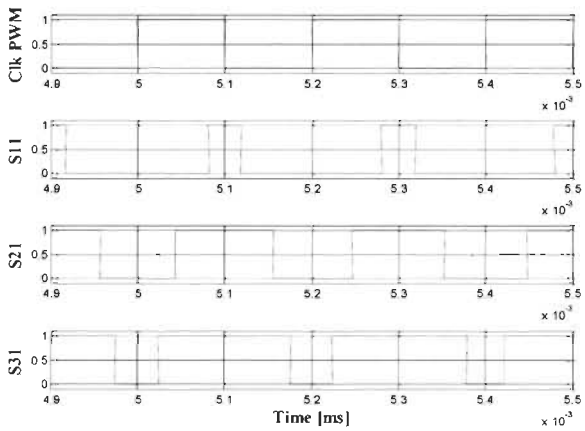


Figure 4.18 Signaux PWM pour trois périodes T_s.

Les signaux de commande sont indiqués sur la figure 4.18, ce qui démontre que les contraintes de forme d'onde définies sur la figure 4.7 sont respectées par l'approximation. Le deuxième test vérifie la capacité du nouveau modèle à générer un système de tensions asymétrique. Pour cela, nous avons imposé un système de tensions déséquilibrées comme ondes de référence.

$$V_a = 220*0.8 * sin\ (2*\pi*60)$$

$$V_b = 220 * sin\ (2*\pi*60 - 0.9465*2*\pi/3)$$

$$V_c = 220 * sin\ (2*\pi*60 + 0.9465*2*\pi/3)$$

La figure 4.19 montre que, même dans le cas d'un système déséquilibré, les sorties de l'onduleur sont synchronisées avec la tension de référence, et l'erreur est égale à 0,5% par rapport au 0,8 * 220V.

Pour tester l'adaptabilité du modèle proposé avec les lois de contrôle, nous avons remplacé le bloc SVPWM dans le travail [4.17] avec le modèle approximatif. Les résultats obtenus montrent que l'approximation est efficace et précise, même dans un état transitoire (Figure 4.20).

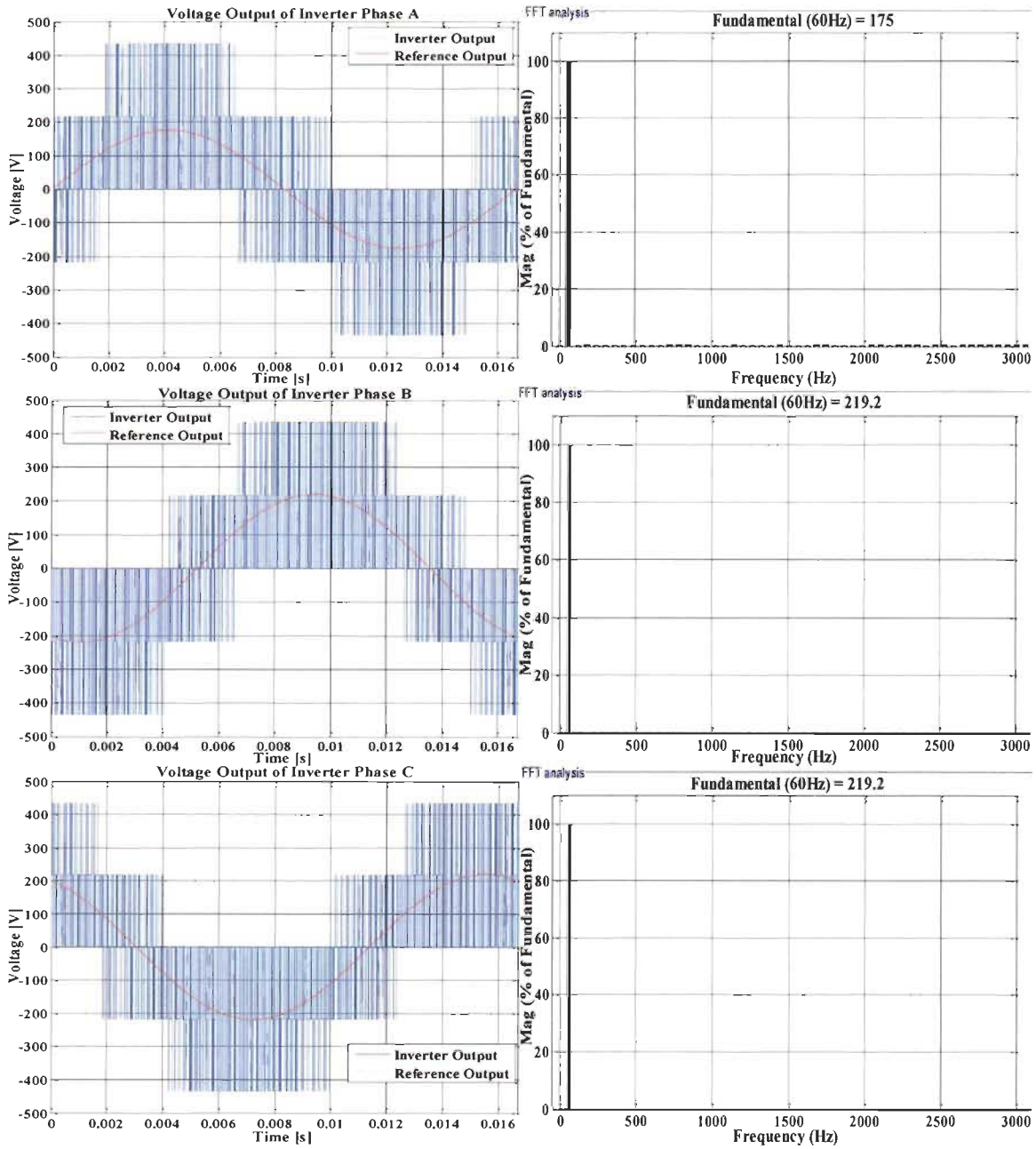


Figure 4.19 Tensions triphasées pour un système déséquilibré.

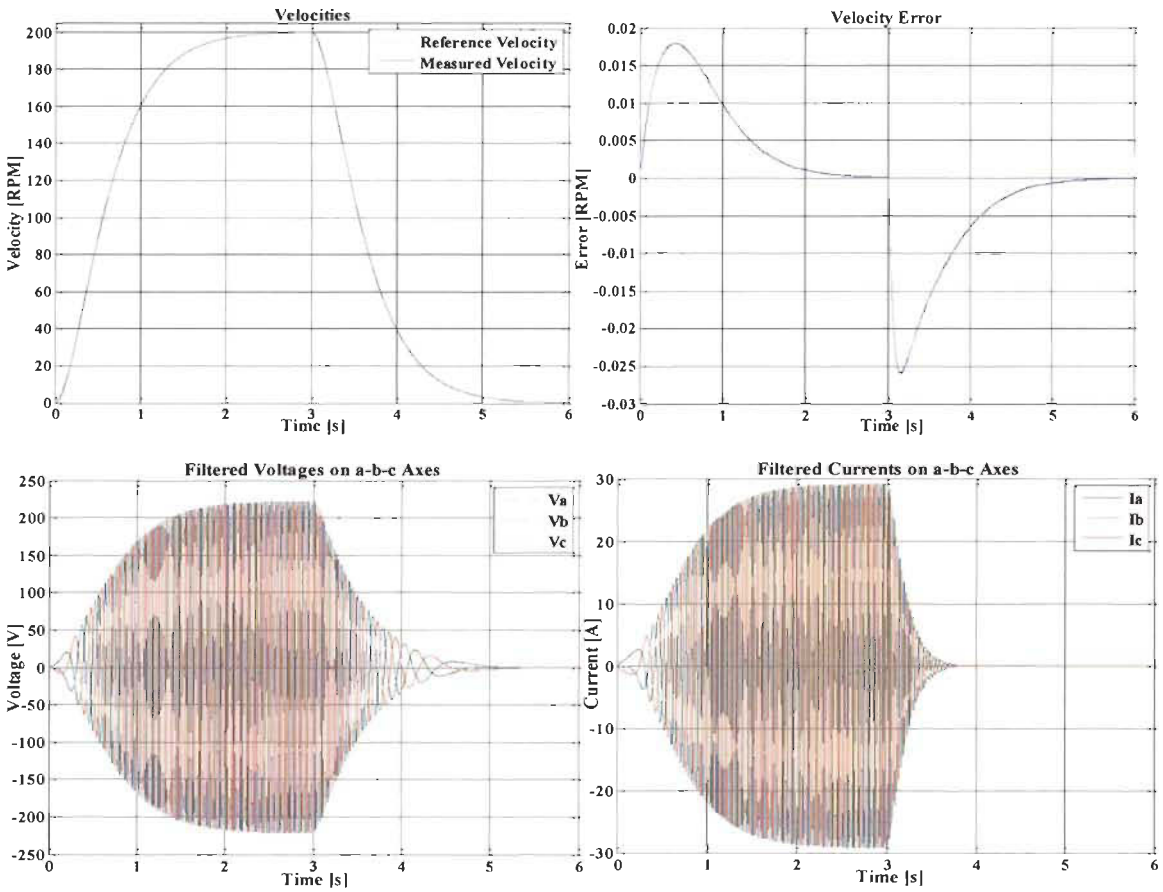


Figure 4.20 Simulation de l'approximation avec une loi de commande.

4.4.3 Implémentation

Le polynôme (4.18) peut être mis en œuvre d'une manière plus optimale à l'aide de l'équation suivante:

$$A = ((a * V_{dc} + b) * V_{dc} + c) * Vdc + d \tag{4.19}$$

Pour calculer les six périodes de commutation on applique la relation (4.17) trois fois et (4.19) une fois donc au total cette approximation peut consommer six multiplicateurs et six additionneurs. Si nous analysons le schéma proposé dans [4.16], nous trouvons que cette approche consomme beaucoup plus, parce que la structure du réseau de neurones utilisé se compose de 20 neurones cachés, ce qui implique l'utilisation d'au moins 20

multiplicateurs et 20 additionneurs, sans tenir compte de la consommation de l'algorithme d'apprentissage. Pour sa part, l'auteur de [4.18] a réussi à simplifier l'algorithme classique de SVPWM en éliminant les calculs d'un bras d'onduleur, mais à partir du tableau 3 de [4.18], qui résume quelques calculs à faire pour calculer les durées de commutation. Nous pouvons voir, dans cette partie seulement, l'utilisation de trois additionneurs et dix multiplicateurs.

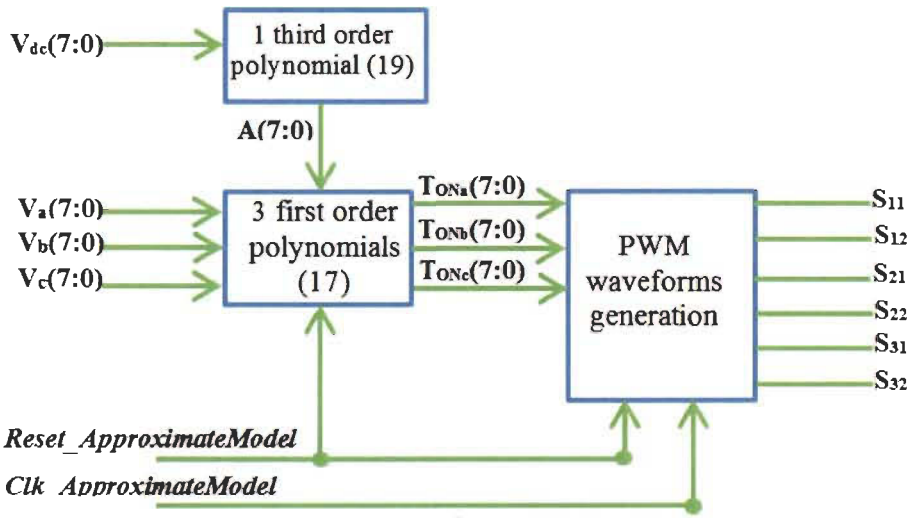


Figure 4.21 Schéma du circuit du modèle approximatif réalisé sur FPGA.

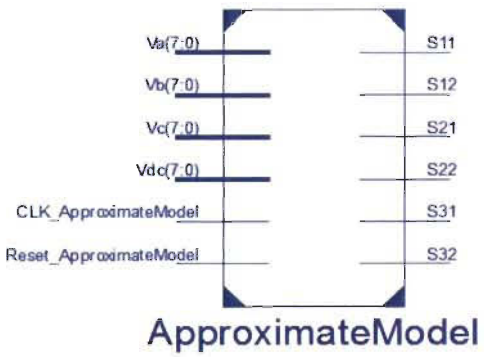


Figure 4.22 Entrées/sorties du circuit SVPWM réalisé sur FPGA (CLK_ApproximationModel=Ts).

L'algorithme du modèle approximatif a été décrit en VHDL et synthétisé avec le logiciel Xilinx ISE Design (Suite 13,4) (figures 4.21 et 4.22). La méthodologie de

simulation et de vérification utilisée est la même que dans la section précédente. Afin d'établir une comparaison raisonnable entre le modèle SVPWM et l'approximation de la SVPWM, les deux modèles ont été mis en œuvre avec une longueur de mot égale à 8 bits. L'onduleur est modélisé avec Simulink™ / Matlab® en utilisant la structure représentée sur la figure 4.8. Le signal de référence désiré est le même que dans la section précédente, avec une amplitude égale à 220 volts et une fréquence de 60 Hz.

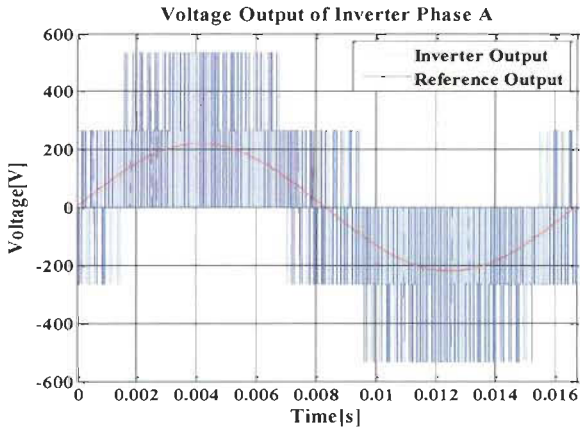


Figure 4.23 Tension de sortie Phase A.

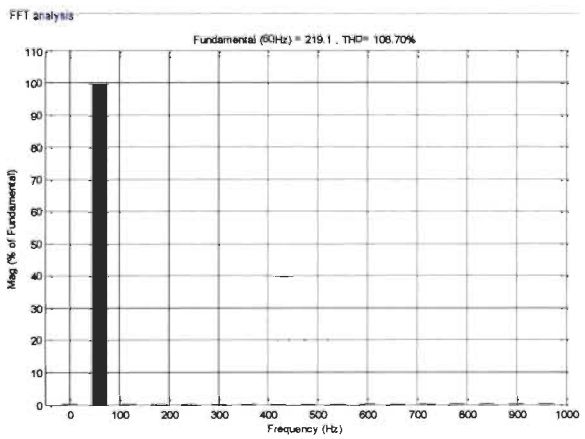


Figure 4.24 Représentation des harmoniques d'une phase.

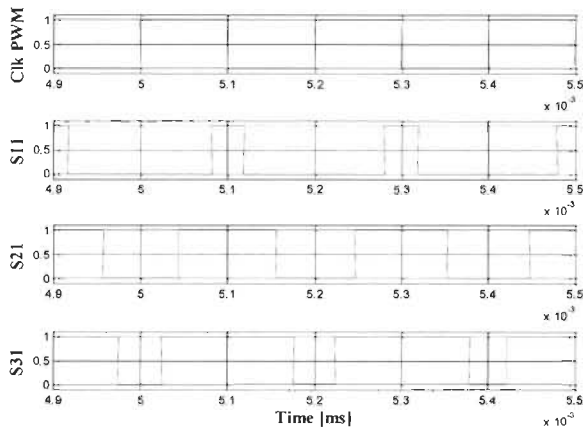


Figure 4.25 Signaux PWM pour trois périodes Ts.

Sur les figures 4.23 et 4.24, on observe que la tension de sortie de l'onduleur (avec une charge résistive), qui a été contrôlée par le modèle VHDL approximatif, est en phase avec la tension de référence avec un taux d'harmonique toujours petit.

En outre, l'approximation proposée conserve les avantages de la méthode traditionnelle. En termes de mise en œuvre, elle est plus efficace car elle consiste à mettre en œuvre un polynôme de troisième ordre et trois polynômes de premier ordre, qui ont les mêmes coefficients. En outre, le calcul du coefficient A peut être effectué à une vitesse inférieure aux premiers polynômes de premier ordre, dans le cas où la tension de bus est bien filtrée.

Après la synthèse nous avons obtenu le tableau 4.4 qui résume les ressources requises par les deux modèles (SVPWM et l'approximation proposée) sur une plateforme Spartan3. L'approximation proposée utilise moins de 30% (96Slice / 358Slices) de l'espace consommé par la SVPWM. Des gains importants sont également observés pour les autres unités de la Spartan III, sauf pour IOBs.

Tableau 4-4 Résumé de l'utilisation des ressources pour l'implémentation de l'approximation de la SVPWM.

Spartan III xc3sd3400a-5fg676			
	SVPWM	Approximation	Disponible
Number of slices	358	96	23872
Number of slices Flip Flops	110	77	47744
Number of input LUTs	666	161	47744
Number of bonded IOBs	32	40	469
Number GCLKs	2	2	24
Number DSP 48s	11	8	126

4.5 Conclusion

Une nouvelle technique pour générer le signal MLI pour contrôler les onduleurs a été implémentée en utilisant VHDL et a été validée à l'aide de cosimulations entre ModelSim et Simulink™ / Matlab®. Cette technique, par ses faibles taux d'harmonique et pertes de commutation d'une part, et sa simplicité d'autre part, peut remplacer le SVPWM dans les applications où les ressources d'intégration sont très limitées. Les résultats de synthèse montrent que la nouvelle technique consomme peu de ressources lors de la mise en œuvre en laissant suffisamment de ressources pour la mise en œuvre des boucles de régulation.

Chapitre 5 - Réalisation d'une bibliothèque de lois de commande adaptative pour MSAP

5.1 Introduction

La principale contribution de ce travail est la conception de trois schémas de circuit sur *FPGA* pour trois lois de commande adaptative d'une machine synchrone à aimants permanents à base de logique floue. Les performances des trois lois, en termes de précision et de fiabilité, en particulier face aux incertitudes internes, ont été validées dans des travaux antérieurs [5.1, 5.2, 5.3]. Dans ce travail, nous comparons les trois lois de commande en termes d'intégration *VLSI* tout en tenant compte de la performance et de la fiabilité. Tous les schémas proposés, grâce aux simplifications qui ont été appliquées aux algorithmes de ces lois, permettent à la fois l'implémentation de la loi de commande, en conservant leurs bonnes performances, et la mise en œuvre de générateur de PWM sur la même carte, laissant libres les ressources suffisantes pour mettre en œuvre des fonctions supplémentaires. Dans les trois lois de commande, la *SVPWM* a été implémentée par l'approximation proposée dans le chapitre précédant. Le premier schéma proposé dans [5.1] sera très utile pour le prototypage rapide des deux autres lois de contrôle qui sont des dérivées de cette loi de contrôle à base de logique floue adaptative. Pour cette raison, la première et la plus grande partie de ce chapitre sera consacrée pour l'implémentation de cette loi. Ensuite, dans les deuxième et troisième parties, on présente les résultats d'implémentation des deuxième et troisième lois de contrôle. Et en conclusion on finira par une comparaison entre les résultats des trois schémas obtenus et on en tirera un compromis pour le schéma le plus efficace en termes d'intégration et de précision.

5.2 1ere Loi implémentée : *Adaptive Fuzzy Logic Control of Permanent Magnet Synchronous Machines with Nonlinear Friction*

La stratégie de contrôle à implémenter (Figure 5.1) a été proposée dans [5.1] pour contrôler une *MSAP* avec un minimum d'informations sur la machine, tout en garantissant la stabilité en présence d'incertitudes internes et externes.

Les variables nécessaires à la commande de vitesse sont les suivantes: la vitesse de référence, la vitesse mesurée et la position du rotor de la *MSAP*. Les entrées du contrôleur adaptatif à base de logique floue sont l'erreur de vitesse $e_\omega = \omega^* - \omega_m$ et la variation de l'erreur $\dot{e}_\omega = e_\omega(k) - e_\omega(k-1)$. La sortie du régulateur est la tension en quadrature désirée V_q^*.

La sortie du modèle de référence (dynamique d'erreur du premier ordre) est utilisée pour l'adaptation de la logique floue. La référence de la composante directe de tension $V_d^* = 0$

est maintenue constante à zéro. La transformation de *PARK* nous permet de passer du référentiel fixe *d-q* au référentiel rotatif α-β. La *SVPWM* génère les six signaux de commande pour l'onduleur de tension à partir des signaux de références V_α^* et V_β^*.

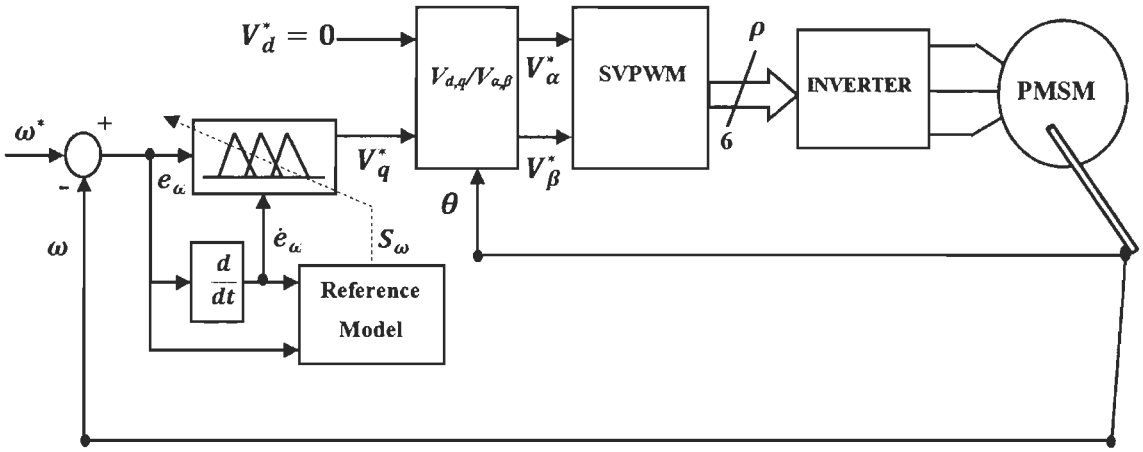


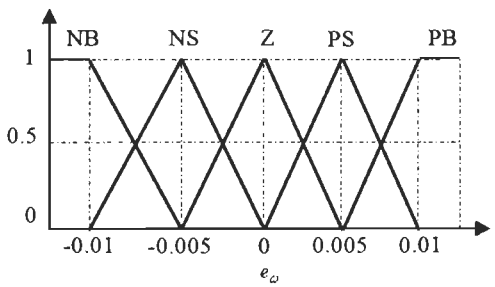
Figure 5.1 Structure de contrôle.

5.2.1 Configuration du contrôleur flou adaptatif

Un contrôleur à logique floue classique se compose de quatre parties, la fuzzification, la base de règles, le moteur d'inférence et la défuzzification. En plus de ces étapes, un contrôleur à logique floue adaptatif comprend un mécanisme d'adaptation.

5.2.1.1 Fuzzification

Afin d'implémenter une structure parallèle, les univers de discours des deux variables e_w et $\dot{e}_w$ sont distribués de la même manière et ont les mêmes fonctions d'appartenance. Les univers de discours sont limités entre -1 et 1, et sont répartis sur cinq fonctions : *BN (Big Negative), SN (Small Negative), Z (Zero), SP (Small Positive), BP (Big Positive)* (Figure 5.2).



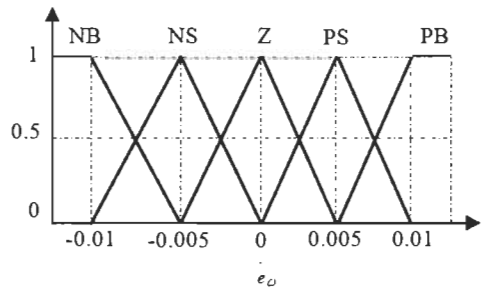


Figure 5.2 Fonctions d'appartenance.

5.2.1.2 Base de règles floues

Cette étape définit la sortie floue pour chaque combinaison d'entrées floues. Ces règles sont souvent fixées en fonction du comportement du contrôleur et du système [5.5]. Pour chaque variable d'entrée on a considéré cinq fonctions d'appartenance, ce qui donne un ensemble de règles de 5x5. Le tableau 5.1 présente les règles utilisées dans ce travail.

Tableau 5-1 Table des règles d'inférence.

		e_ω				
		NL	**NS**	**Z**	**PS**	**PL**
$\dot{e}_\omega$	**NL**	Z	PL	PL	PL	PL
	NS	NL	Z	PS	PS	PL
	Z	NL	NS	Z	PS	PL
	PS	NL	NS	NS	Z	PL
	PL	NL	NL	NL	NL	Z

5.2.1.3 Moteur d'inférence

Le moteur d'inférence est l'élément clé de l'algorithme de la logique floue. Le mécanisme d'inférence, proposé par *Mamdani*, utilise les opérateurs de base minimum et

maximum [5.6]. La méthode MAX-MIN teste les grandeurs de chaque règle et sélectionne les plus élevées [5.1].

$$\mu_{B'}(y) = \coprod_{x \in X} \left[\mu_{A_x}(x) \Pi \mu_{R'}(x, y) \right] \tag{5.1}$$

5.2.1.4 Défuzzification

La défuzzification est la dernière étape d'un contrôleur flou; elle effectue la conversion des valeurs floues en une sortie numérique. C'est l'inverse de l'opération de fuzzification. Dans la littérature, il existe plusieurs méthodes de défuzzification qu'on a présentées au deuxième chapitre. La sortie à l'instant k par la méthode de centre de gravité, qui est la méthode la plus utilisée, est exprimée par :

$$y_j(k) = \frac{\sum_{l=1}^{r} y_l \mu_{B'}(y)}{\sum_{l=1}^{r} y_{l\,B'}(y)} \tag{5.2}$$

où, $l=1, \ldots, r$ et r est le nombre des règles.

5.2.1.5 Commande à logique floue adaptative

Contrairement à d'autres techniques d'intelligence artificielle, la logique floue n'inclut pas explicitement un algorithme d'apprentissage ou d'adaptation. La configuration d'un contrôleur flou se résume à trois étapes, la normalisation des entrées et sorties, le partitionnement de l'univers du discours sur les fonctions d'appartenance et la dernière et la plus importante étape est de bien comprendre le comportement du système en définissant correctement les règles. Cependant, le réglage de ces paramètres peut ne pas être optimal pour une grande plage de fonctionnement. Basé sur l'adaptabilité du réseau de neurones, les

chercheurs ont proposé un régulateur flou adaptatif. Cette technique permet aux contrôleurs flous adaptatifs d'atteindre un fonctionnement optimal sur une très large gamme [5.7].

De la structure de logique floue qui est représentée sur la figure 5.3, les première et seconde couches donnent la partie antécédente des règles floues $\hat{\phi}$, et les troisième et quatrième couches donnent la partie résultante via une matrice de pondération, de sorte que la sortie du régulateur à logique floue peut être écrite en tant que [5.1]:

$$Y = \phi^T W + \varepsilon = \hat{\phi}^T \hat{W} \tag{5.3}$$

où

$\varepsilon = \hat{\phi}^T \hat{W} - \phi^T W$: Erreur de la sortie de la logique floue;

$\hat{\phi} \in R^r$: est le vecteur de la partie antécédente de la logique floue qui est défini comme suit :

$$\hat{\phi} = \frac{U_{B^l}(y)}{\sum_{l=1}^{r} U_{B^l}(y)} \tag{5.4}$$

où

$\hat{W} \in R^{r \times m}$: La matrice du poids.

m : Nombre de sorties, dans ce cas m=1.

Donc, $\hat{W} \in R^{r \times 1} = [y_1, y_2, ..., y_r]$, où y_l est la partie conséquence de la logique floue pour les l règles et $l = 1,...r$. Dans ce cas $r = 5$.

Le modèle d'erreur de la vitesse de consigne (modèle de référence), qui génère le signal d'erreur d'adaptation, est défini comme:

$$s = e_{\omega} - \psi \dot{e}_{\omega} \tag{5.5}$$

Avec ψ étant une constante positive qui détermine la dynamique d'erreur souhaitée.

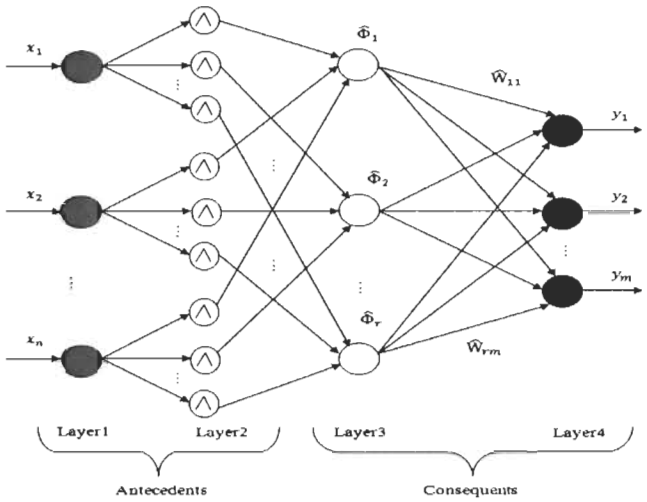


Figure 5.3 Structure de l'unité d'adaptation de la logique floue.

Le vecteur $\widehat{W}$ est ajusté à chaque période du *Clock* en fonction du modèle de l'erreur proposé dans l'équation (5.5). L'équation de récurrence sur le vecteur de poids $\widehat{W}$est donnée par la formule suivante.

$$\Delta\widehat{W} = \delta * \widehat{\Phi} * s \tag{5.6}$$

$$\widehat{W}(k) = \widehat{W}(k-1) + \Delta\widehat{W} \tag{5.7}$$

où δ est une constante positive non nulle.

5.2.2 Implémentation et simulation

Le système utilise deux signaux d'horloge. La première horloge, synchronise les entrées, sorties, et les mémoires. La seconde, *Clock_PWM*, est utilisée pour générer les signaux *PWM*. Seules les instructions primitives et un code modulaire ont été utilisés pour décrire les différentes fonctions. La division a été remplacée par un ensemble d'ajouts, des multiplications et des décalages de bits, ce qui a permis de sauver des ressources.

Le contrôleur de la *MSAP* basé sur l'algorithme adaptatif a été décrit en *VHDL* et synthétisé avec le logiciel *Xilinx ISE Design (Suite 13,4)* (Figures 5.4, 5.5, 5.6 et 5.7 qui seront expliquées ci-après). Le modèle *VHDL* a été évalué par la cosimulation de *ModelSim et Simulink*^{TM} */ Matlab*®. L'onduleur et la *MSAP* ont été modélisés en utilisant *Simulink*^{TM} */ Matlab*®.

- *FLC*

Les entrées du contrôleur flou, l'erreur et la variation de l'erreur sont calculées à partir de la vitesse de référence et de la vitesse mesurée en utilisant deux soustractions de 24 bits, et une mémoire de 24 bits; le choix de la longueur de mot se justifie par les très faibles valeurs à la sortie du mécanisme d'adaptation flou. Les opérations de la fuzzification, table de règles et de la défuzzification ont été réalisées avec une longueur de mot égale à 16 bits. La sortie du régulateur V_q et V_d références, sont représentés en utilisant des mots de 16 bits.

Le calcul du pourcentage d'appartenance des 10 variables linguistiques est contrôlé par un multiplexeur (Figure 5.4). A chaque front montant de l'horloge, nous calculons le degré d'appartenance à une fonction. S'appuyer sur l'autocorrection de la logique floue,

représentée par son adaptabilité qui va corriger une éventuelle erreur, permet d'économiser 90% de l'espace réservé pour le calcul des dix degrés d'appartenance au même temps.

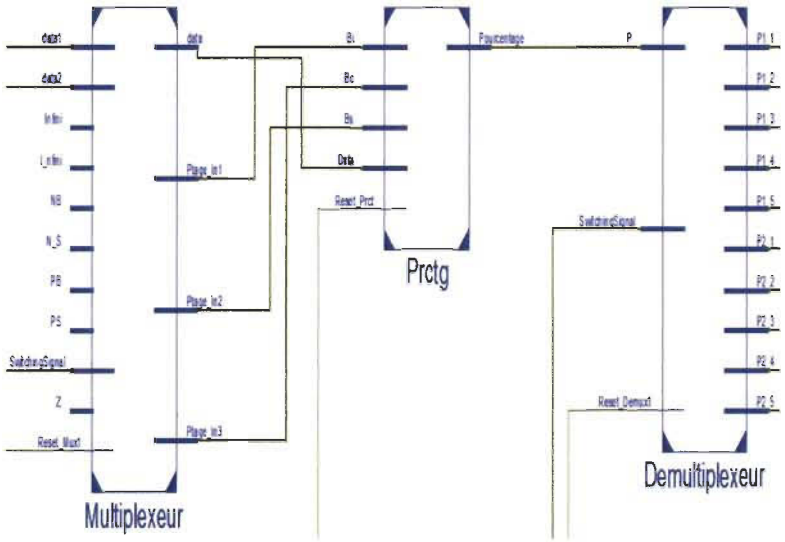


Figure 5.4 Schéma du circuit calcul du pourcentage d'appartenance réalisé sur un FPGA.

Le même principe utilisé dans la mise en œuvre de la fuzzification a été utilisé pour le bloc d'adaptation (Figure 5.5), où nous avons 5 poids à mettre à jour. A chaque front montant de la même horloge un des poids est mis à jour. Il permet d'économiser encore 80% des ressources qui pouvaient être réservées pour mettre à jour les 5 poids en même temps.

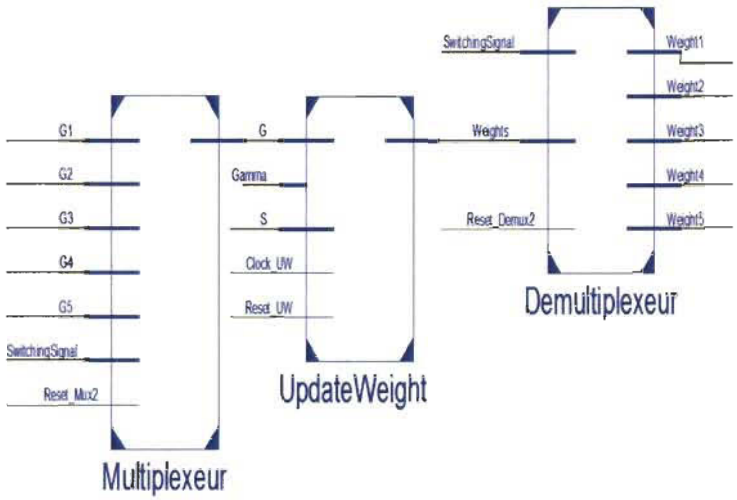


Figure 5.5 Schéma du circuit de la partie d'adaptation réalisé sur un FPGA.

La figure 5.5 montre le schéma proposé pour l'implémentation de l'algorithme d'adaptation. Les entrées G1 a G5 sur le schéma sont les signaux qui représentent les variables μ_{B^i} qui sont définies dans l'équation (5.1) et dans la figure 5.7 sont représentés par les signaux $V_{NB\%}$, $V_{NS\%}$, $V_{Z\%}$, $V_{PS\%}$, $V_{PB\%}$. Le reste des opérations de la logique floue ont été implémentées directement comme représenté mathématiquement, sans modifications. Le schéma global du contrôleur flou sur une carte *FPGA* est représenté sur la figure 5.6.

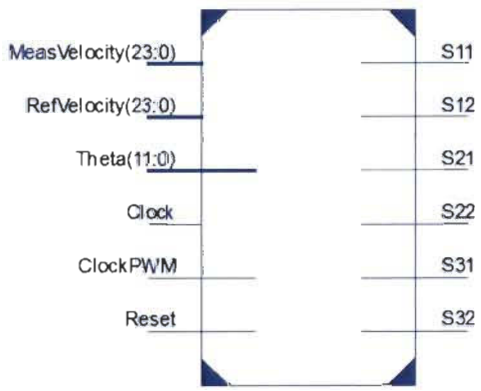


Figure 5.6 Entrées/Sorties du circuit global de contrôle réalisé sur un FPGA.

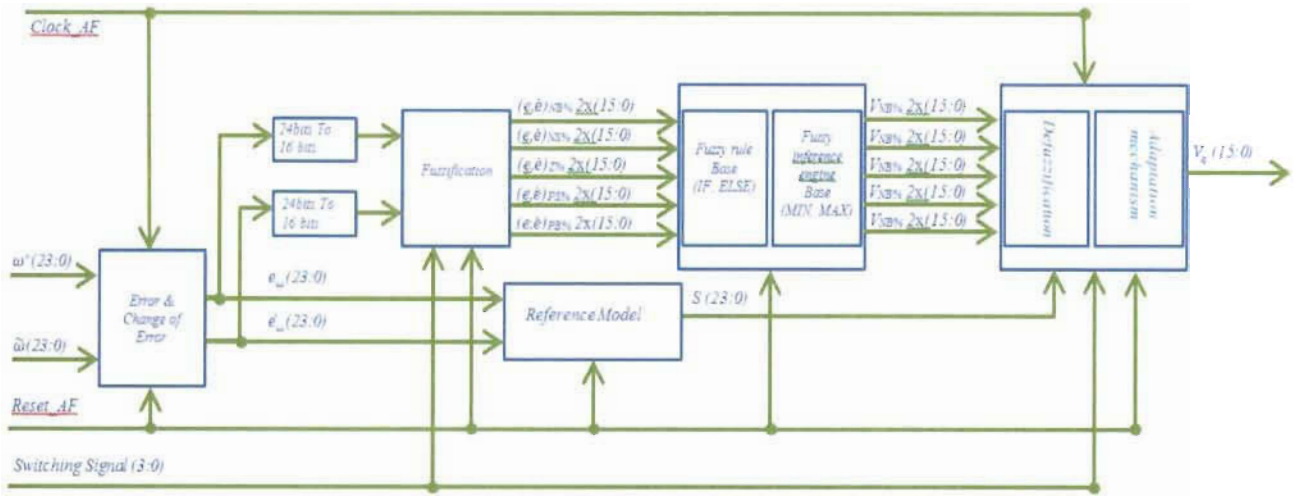


Figure 5.7 Schéma du circuit FLC adaptatif réalisé sur un FPGA (Détails dans l'annexe -B).

- *Transformation de PARK*

La transformation de *Park* est représentée par l'équation suivante :

$$\begin{bmatrix} V_\alpha \\ V_\beta \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & +\cos(\theta) \end{bmatrix} \begin{bmatrix} V_d \\ V_q \end{bmatrix} \tag{5.8}$$

Le calcul des sinus et cosinus de la transformation de *PARK* est mis en œuvre en utilisant la méthode de développement limité en prenant en considération seulement les quatre premiers termes de la série et avec une taille de mot égale à 12 bits (Figure 5.8).

$$\begin{aligned} \sin(\theta) &= \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + \frac{\theta^9}{9!} \\ \cos(\theta) &= 1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \frac{\theta^6}{6!} + \frac{\theta^8}{8!} \end{aligned} \tag{5.9}$$

Le choix du nombre de termes à prendre en considération est justifié par les résultats de simulation présentés dans la Figure 5.9. Notons que, dans le cas de quatre termes, l'erreur relative peut atteindre 7%. Par exemple dans le cas d'une amplitude de tension égale à 100

volts, il y aura une erreur de 7 *volts*, qui peut affecter la précision du contrôleur. L'erreur peut être réduite par la prise en considération de plus de termes dans la série ou bien en exploitant la symétrie des sinus et des cosinus, ce qui est la solution qu'on a choisi dans ce travail.

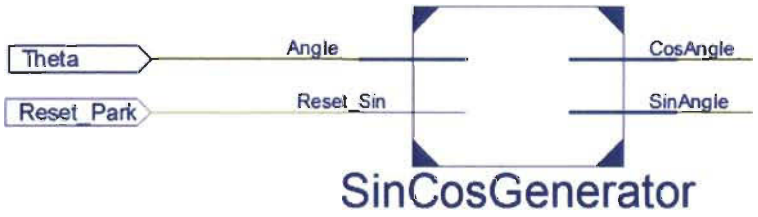


Figure 5.8 Entrées/Sorties du circuit Générateur de sin/cos réalisé sur un FPGA.

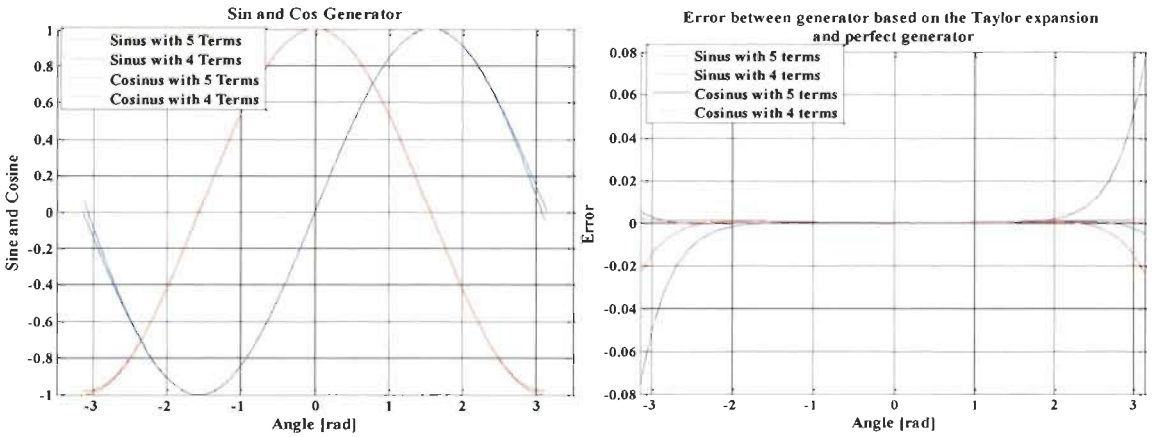


Figure 5.9 Sinus et cosinus à base de la série de Taylor.

La variation de l'angle θ est traduite par une variation entre -1 et 1. Cela a impliqué une normalisation des équations (5.9). Lors de l'implémentation, des nouveaux coefficients différents A_1..... A_5 *et* B_1...., B_5, sont calculés et stockés sur 16 bits. A l'aide d'une série de multiplications à 16 bits, on calcule les termes $A_1*\hat{\theta}$, $B_1*\hat{\theta}^2$.... $A_5*\hat{\theta}^9$ qui vont être accumulés pour formuler le sinus et cosinus selon les équations (5.10) et (5.11).

Après simplification les équations (5.9) deviennent :

$$\sin(\theta) = a * \left(A_1 * \hat{\theta} - A_2\hat{\theta}^3 + A_3\hat{\theta}^5 - A_4\hat{\theta}^7 + A_5\hat{\theta}^9\right) \quad (5.10)$$

$$\cos(\theta) = a * \left(B_1 - B_2\hat{\theta}^2 + B_3\hat{\theta}^4 - B_4\hat{\theta}^6 + B_5\hat{\theta}^8\right) \quad (5.11)$$

où :

$\hat{\theta} \in [-1, 1]$, $A_1 = \frac{\pi}{a}$, $A_2 = \frac{\pi^3}{a*3!}$, $A_3 = \frac{\pi^5}{a*5!}$, $A_4 = \frac{\pi^7}{a*7!}$, $A_5 = \frac{\pi^9}{a*9!}$, $B_1 = \frac{1}{a}$, $B_2 = \frac{\pi^2}{a*2!}$, $B_3 = \frac{\pi^4}{a*4!}$, $B_4 = \frac{\pi^6}{a*6!}$ *et* $B_5 = \frac{\pi^8}{a*8!}$

La constante a est choisie de façon que toute les constante $A_1 \ldots A_5$ *et* $B_1 \ldots B_5 \in [-1, 1]$

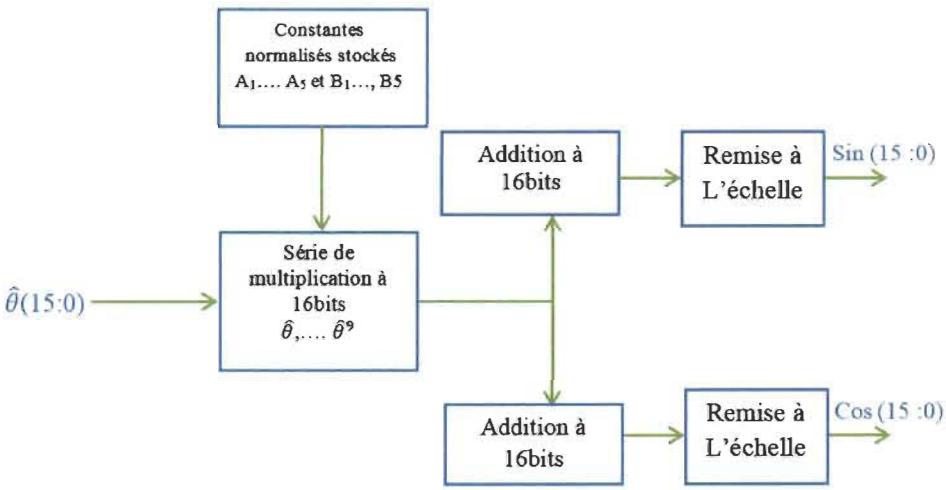


Figure 5.10 Schéma du circuit Générateur SinCos réalisé sur un FPGA.

- *SVPWM*

Les sorties de la transformée de *PARK* sur 8 bits donnent les tensions de référence pour la *SVPWM* (Figure 5.11). L'implémentation de la génération des signaux de commande a été présentée dans le chapitre précédant.

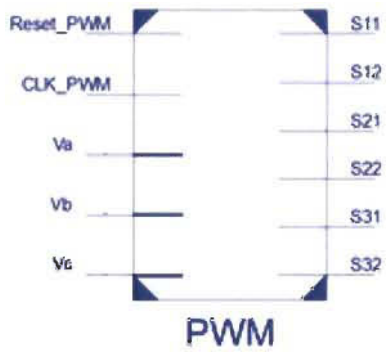


Figure 5.11 Entrées/sorties du circuit SVPWM réalisé sur un FPGA.

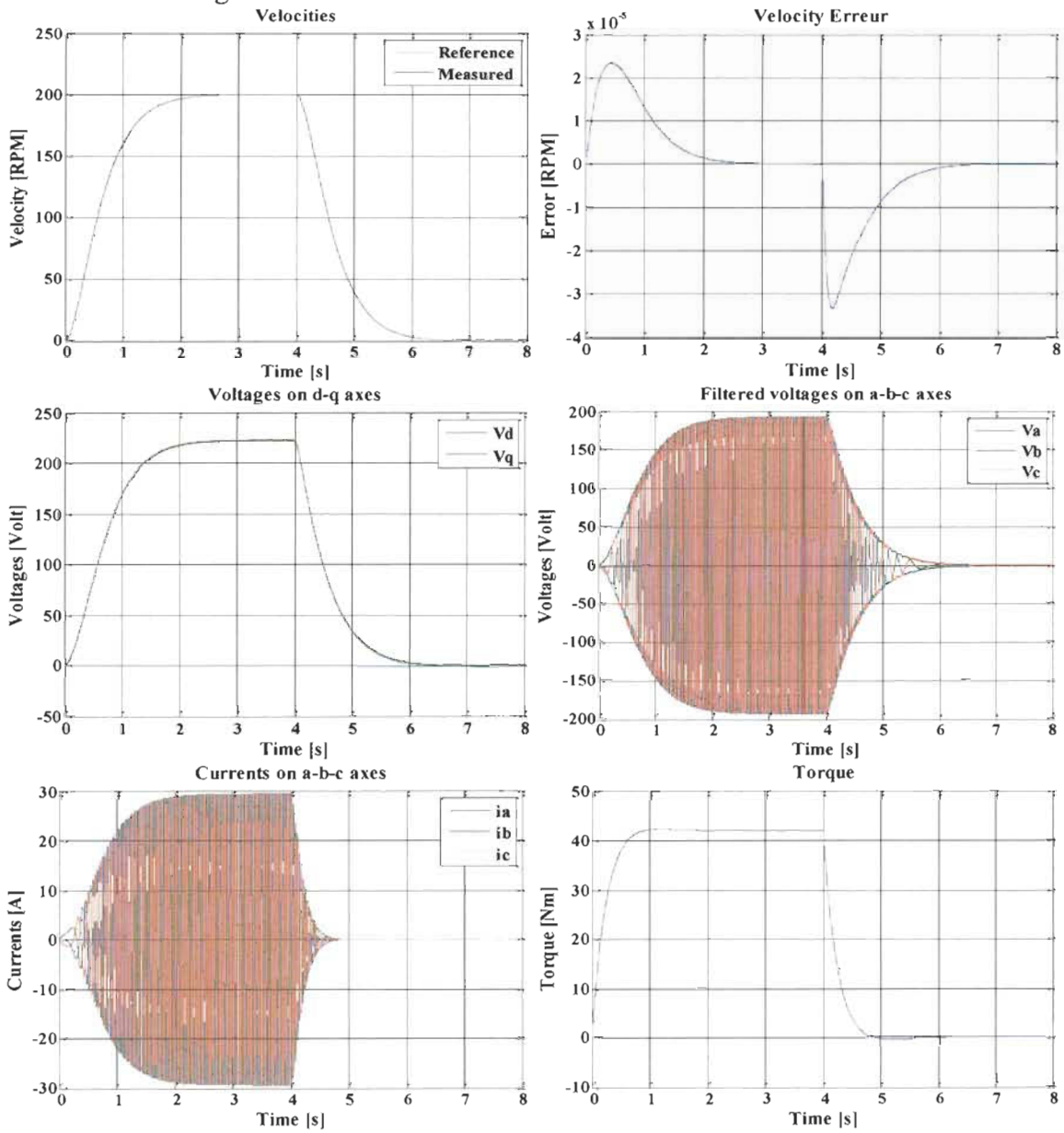


Figure 5.12 Résultats de cosimulation.

Les Figures 5.12 montrent que la vitesse mesurée suit parfaitement sa référence malgré l'erreur qui s'accumule en raison de la quantification et la troncature des données. Les résultats de synthèse montrent que la mise en œuvre de l'algorithme consomme très peu de ressources dans un *FPGA Spartan III* (tableaux.5.2 et 5.3). Désormais, d'autres fonctions peuvent être mises en œuvre sur le même *FPGA*, en particulier une autre boucle de contrôle ou d'estimation de vitesse.

Tableau 5-2 Sommaire d'utilisation des ressources pour l'implémentation de la loi de commande.

Spartan 3A Target Device: xc3sd1800a-4cs484			
	Utilisés	Disponibles	Utilisation
Number of slices	3595	16640	21%
Number of slices Flip Flops	1168	33280	3%
Number of input LUTs	6524	33280	19%
Number of bonded IOBs	153	309	49%
Number GCLKs	8	24	33%
Number DSP 48s	61	84	72%

Tableau 5-3 Distribution des ressources.

Spartan 3A Target Device: xc3sd1800a-4cs484			
	Utilisés	Disponibles	Utilisation
FLC	2666	16640	16%
Tranformation de Park	671	16640	4%
SVPWM	250	16640	1%

5.3 2eme Loi: *PMSM control based on adaptive fuzzy logic and sliding mode [5.2]*

Dans le schéma proposé dans la section précédente, la tension directe V_d a été forcée à zéro, ce qui a donné une structure de contrôle simple d'un côté, et robuste envers les incertitudes internes et externes de l'autre côté. Cependant, cela a causé une réduction de la plage de fonctionnement du moteur. Une des solutions qui a été proposée par notre

laboratoire, est au lieu de forcer la tension V_d à zéro, à l'aide d'un contrôleur mode glissant on force le courant I_d à zéro, ce qui permet de tirer un couple magnétique maximum du moteur.

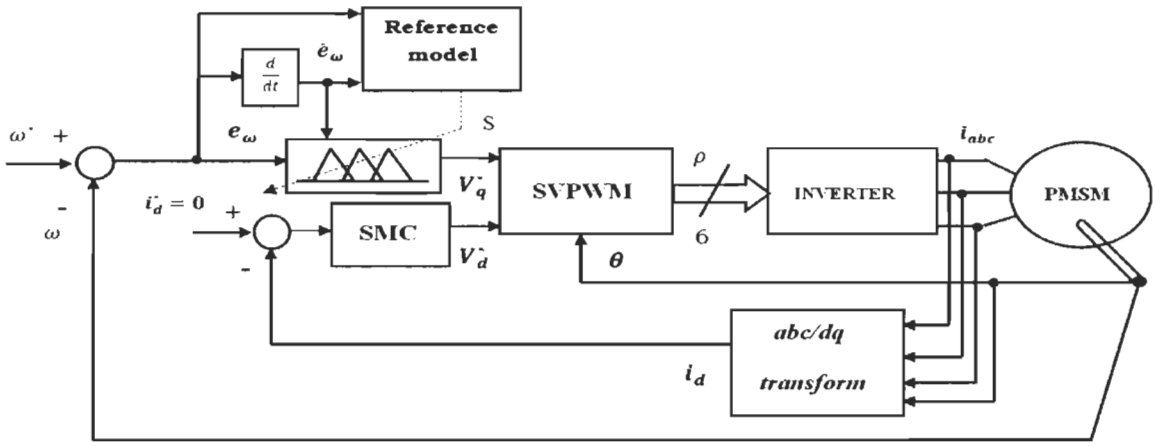


Figure 5.13 Structure de contrôle de la commande LF-MG.

5.3.1 Configuration du Régulateur mode glissant

Le régulateur mode glissant a pour rôle de fournir une tension V_d de référence de sorte qu'elle maintienne un courant $i_d = 0$ pour avoir un couple électromagnétique maximal et ce qui va permettre au moteur de fonctionner dans une plage plus large puisque la tension V_d ne sera pas forcément égale à zéro.

La conception d'un régulateur mode glissant passe par deux étapes:

- La première est de concevoir la surface de glissement pour la grandeur qu'on veut contrôler, dans ce cas le courant i_d;
- La deuxième étape est la conception de la loi de commande à partir de la condition de convergence, qui permettra à la grandeur contrôlée de varier mais le long de la surface de glissement.

Les détails de la conception du régulateur sont présentés dans le travail [5.2]. Dans notre travail nous présenteront seulement les expressions finales obtenues et qu'on va implémenter.

Les deux expressions utilisées pour la conception du régulateur mode glissant sont [5.2] :

$$f(S_s) = \begin{cases} \frac{S_s}{\varepsilon+\delta} & Si\ |S_s| \leq \varepsilon \\ \frac{S_s}{|S_s|+\delta} & Si\ |S_s| > \varepsilon \end{cases} \tag{5.12}$$

$$V_d = \left[-\hat{L}_q p\ \omega i_q\right] + k_d f\left(S_s(i_d)\right) \tag{5.13}$$

où k_d est un coefficient constant.

5.3.2 Implémentation et simulation

Comme on l'avait précisé dans le premier chapitre, un des principaux objectifs de notre travail, est de modéliser un seul schéma qui soit parallèle et qui permet de passer d'une loi de commande à une autre en changeant le minimum de code possible. Donc, pour l'implémentation de cette deuxième loi, on va garder le schéma de la première loi pour la partie contrôle de la tension V_q et la partie contrôle du convertisseur; on a seulement rajouté le module du contrôleur à base de mode glissant. Ce module a été modélisé et validé indépendamment du reste du schéma.

Le régulateur mode glissant a pour rôle de maintenir le courant i_d à zéro, ce qui donne des valeurs trop petites pour les calculs du régulateur et cela nous a obligé d'augmenter la précision sur *VHDL* et d'utiliser une longueur de mot à 24 bits comme il est montré sur la

figure 5.14 qui représente le schéma bloc sur *FPGA* du régulateur mode glissant avec les entrées et les sorties; toutes les grandeurs sont représentées sur 24 bits.

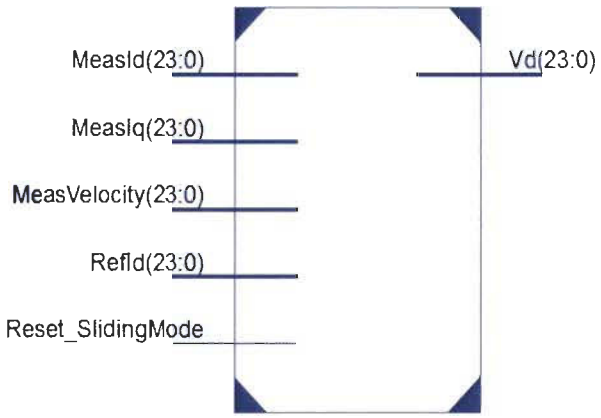


Figure 5.14 Entrées/sorties du circuit Régulateur Mode Glissant réalisé sur un FPGA (Détails dans l'annexe-B).

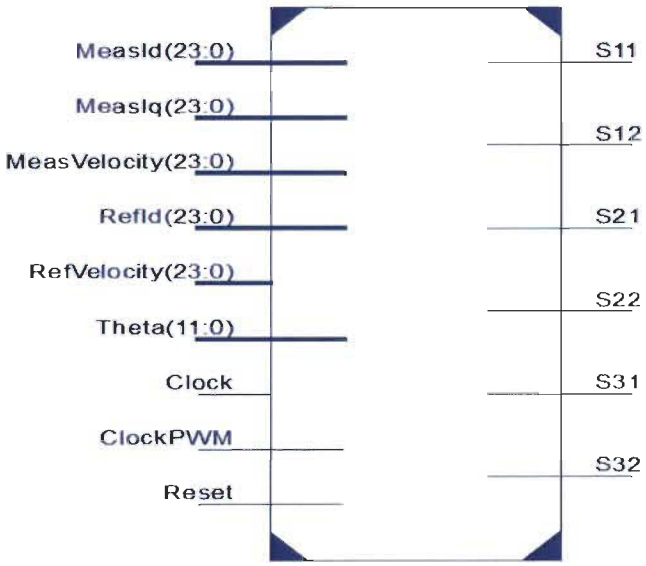


Figure 5.15 Entrées/Sorties du circuit global de contrôle réalisé sur un FPGA.

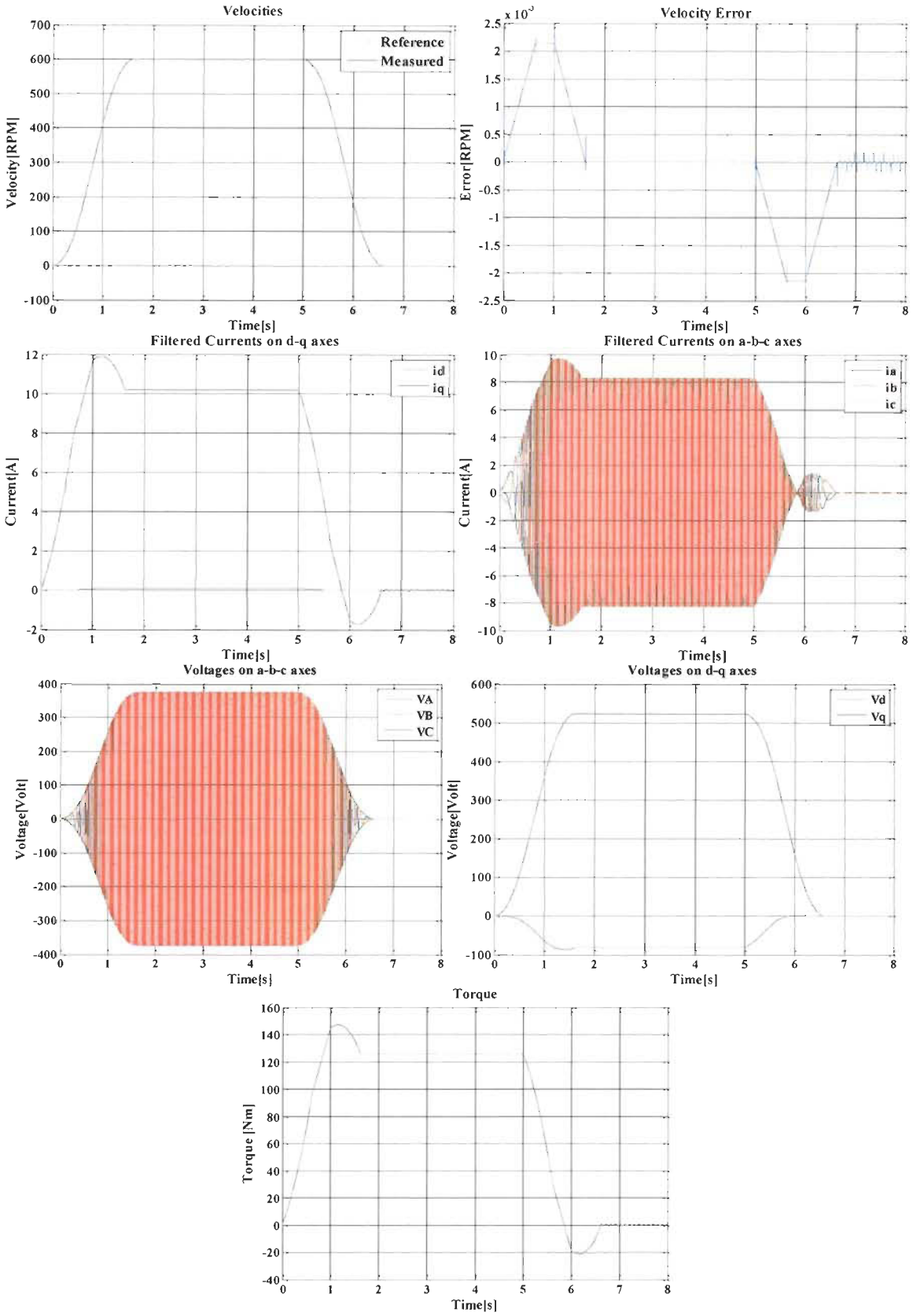


Figure 5.16 Résultats de cosimulation.

5.4 3eme Loi: *Adaptive Fuzzy Logic Control Structure of PMSMs [5.3]*

Malgré son efficacité, le régulateur à mode glissant présente une faible robustesse envers les changements paramétriques de la machine dû à la présence du paramètre L_d dans (5.13). Donc le même auteur a proposé de remplacer le régulateur à mode glissant par un deuxième régulateur flou adaptatif. La figure 5.17 présente la structure proposée.

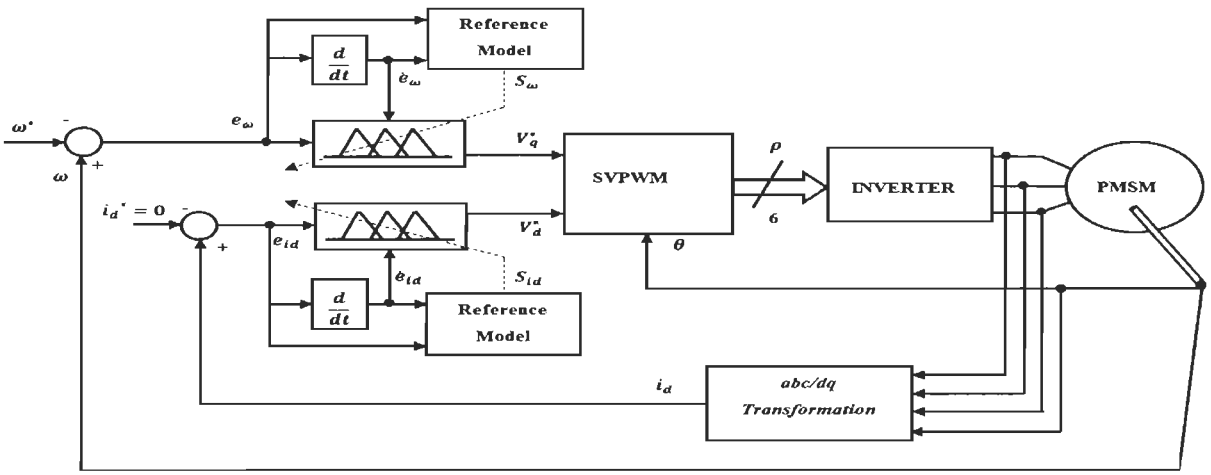


Figure 5.17 Structure de contrôle de la commande LF.

5.4.1 Implémentation et simulation

Dans cette partie la difficulté était de configurer les deux régulateurs flous d'une façon semblable, ça veut dire utiliser le même nombre de variables linguistiques, diviser de la même façon l'univers discours, etc. On a réussi à implémenter un régulateur flou normalisé dans un bloc en *VHDL* qui a été utilisé pour les deux régulateurs, régulateurs de vitesse et de courant. Donc le schéma du circuit du régulateur de courant sur *FPGA* est le même schéma représenté sur la figure 5.7.

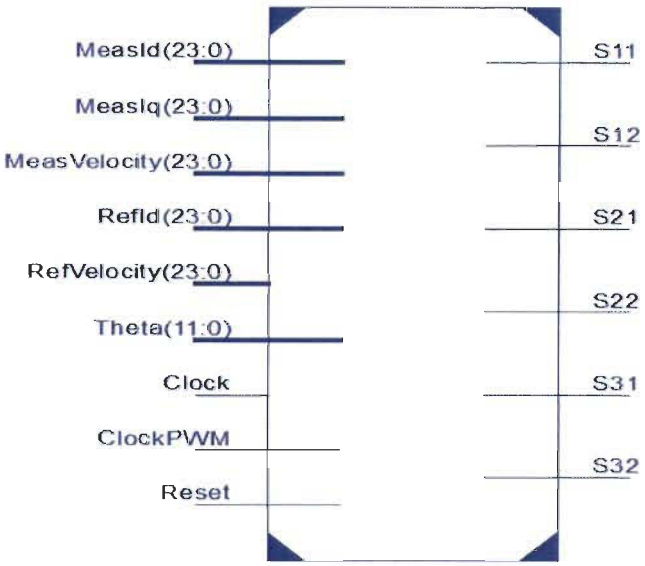


Figure 5.18 Entrées/Sorties du circuit global de contrôle réalisé sur un FPGA.

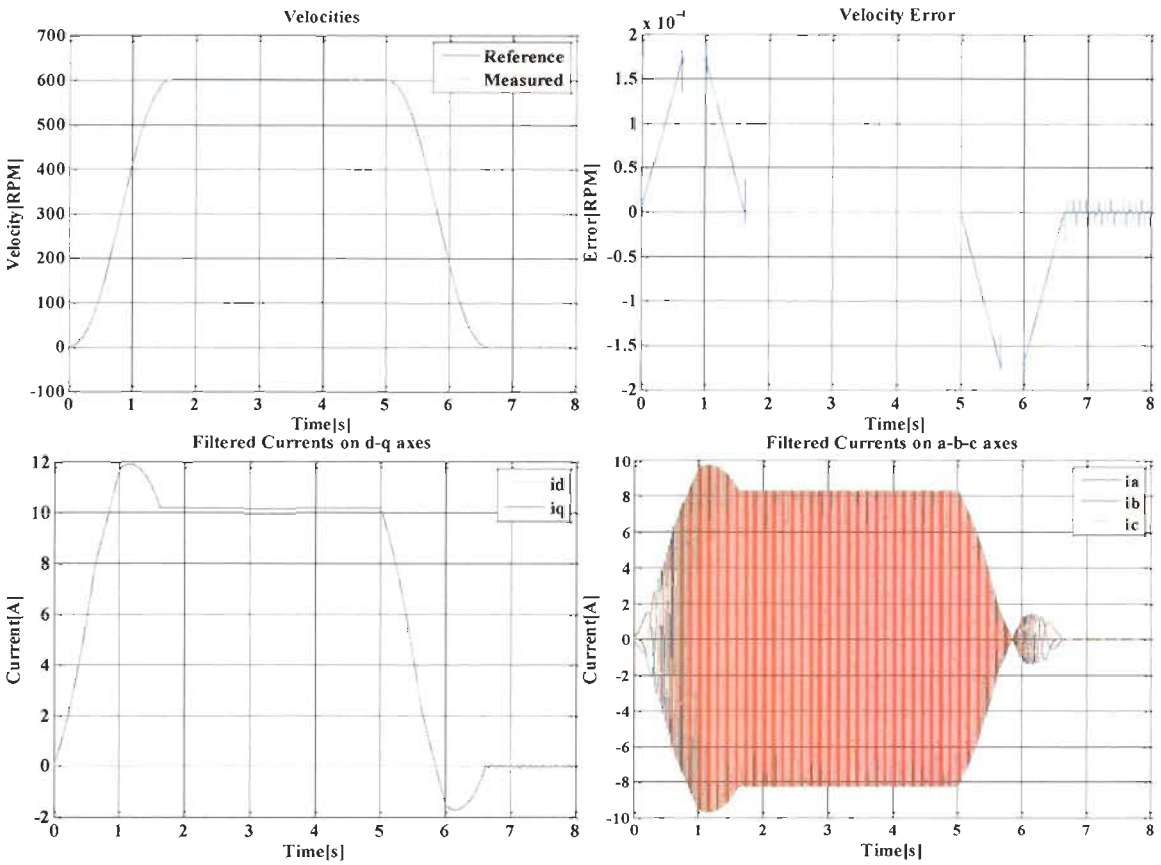


Figure 5.19 Résultats de cosimulation.

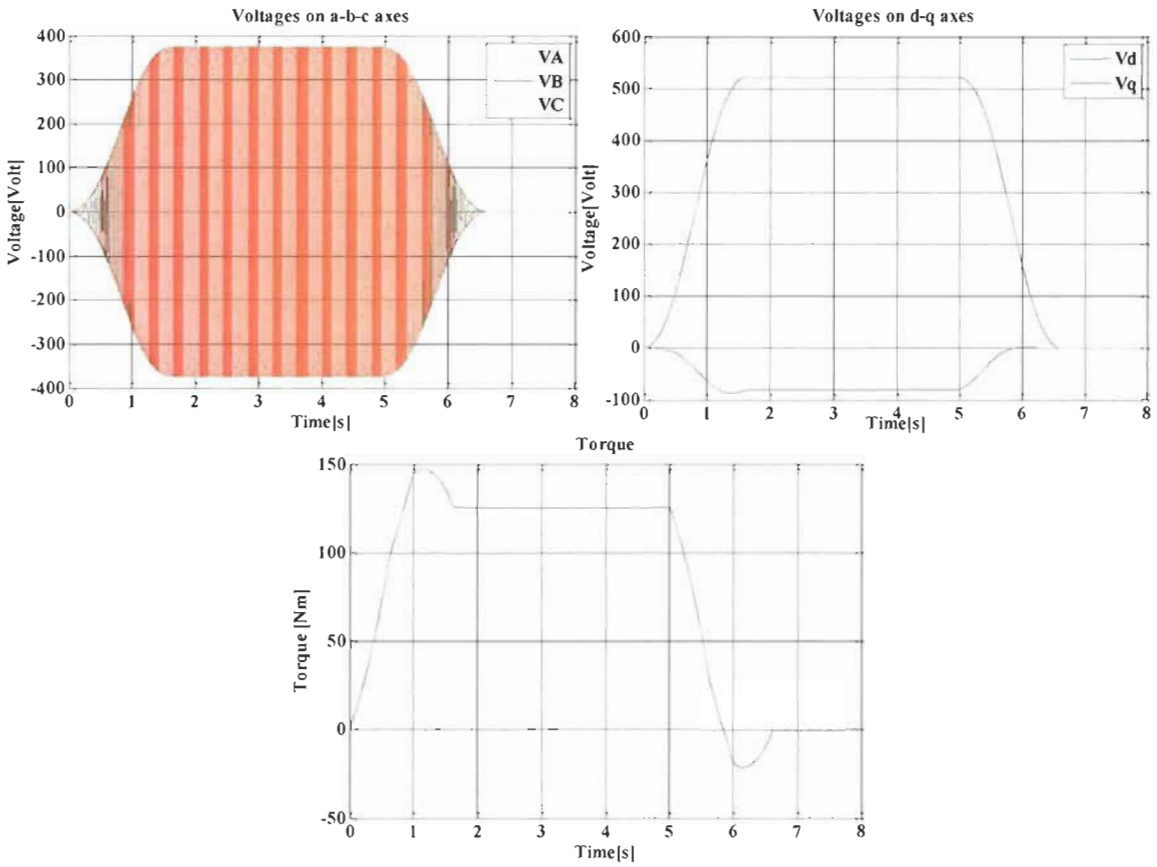


Figure 5.20 Résultats de cosimulation (Suite).

5.4.2 Interprétation des résultats de simulation

Les trois systèmes de contrôle ont été évalués lors de deux modes de fonctionnement, en accélération et freinage. Les trois schémas proposés montrent des bonnes performances au niveau de l'erreur de vitesse. Les performances du premier schéma sont limitées au-delà des 200 tr/min à cause de l'imposition de la composante directe de la tension à zéro. Cependant la figure 5.12 montre l'efficacité de ce schéma lorsque la vitesse est en bas de 200 tr/min, malgré toutes les simplifications qui ont été faites sur l'algorithme original. Les figures 5.16 et 5.19 présentent les résultats des deuxième et troisième schémas. Ces dernières montrent que les lois implémentées peuvent assurer des performances très élevées même avec une vitesse de référence égale à la vitesse nominale. Les résultats au-dessous

prouvent que les modifications apportées et la représentation numérique utilisée n'ont pas nui à la précision des trois schémas.

5.5 Conclusion

En analysant les résultats de cosimulation on constate que l'ajout d'un deuxième régulateur élargit considérablement la plage de fonctionnement de la loi, et le maintien du courant i_d à zéro donne un couple maximum.

En terme de précision, la loi à base de deux régulateurs flous donne une erreur de vitesse inférieure à celle de la loi à base du mode glissant; même le courant i_d réglé par le régulateur flou est plus proche de zéro que celui réglé par le régulateur mode glissant. Cependant en se référant aux résultats de synthèse du code *VHDL* des deux lois, résumés dans le tableau 5.4, on peut remarquer un rapport de plus de 160% entre l'espace utilisé pour l'implémentation de la loi à base de deux régulateurs flous et celle à base du mode glissant.

Tableau 5-4 Comparaison des trois lois en termes de consommation.

Spartan 3A Target Device: xc3sd1800a-4cs484			
	Utilisé	Disponible	Utilisation
Loi N°1	3595	16640	21%
Loi N°2	3746	16640	23%
Loi N°3	6143	16640	37%

Une combinaison entre les résultats de cosimulation et de synthèse nous laisse conclure que la loi à base du régulateur flou et mode glissant est un compromis entre la précision et la consommation de ressources.

Chapitre 6 - Conclusion Générale

L'industrie a connu une importante hausse de demande des systèmes d'entrainement électromécanique dans la dernière décennie. Cela est le fruit des avancées dans le domaine de développement des algorithmes de contrôle d'un côté, et surtout le progrès dans la micro et nanoélectronique. Ces deux points ont rendu exploitable l'implémentation de certaines techniques, classées avant trop complexes pour la mise en œuvre.

Dans ce travail on a opté pour la technologie *FPGA* pour l'implémentation d'algorithmes qui ont été déjà proposés.

La première étape était de choisir la représentation numérique et de développer une fonction *Matlab* qui transforme les données d'une variable à virgule flottante en une variable à virgule fixe, ensuite à une variable numérique. Ici, une fonction déjà existante a été adoptée.

La deuxième étape consistait au développement d'une méthodologie pour la vérification de chaque module *VHDL* développé et sa comparaison avec le module *Matlab* correspondant.

Dans la troisième étape, il a fallu développer et approuver les codes en *VHDL* qui exécutent les opérations arithmétiques de base et de s'assurer que l'erreur est la plus faible possible afin d'éviter qu'elle ne s'accumule après la combinaison des différents modules.

La quatrième étape, l'étape la plus importante, consistait à simplifier et raffiner les structures sans affecter les performances des algorithmes de contrôle.

Après comme avant dernière étape, on a séparé l'algorithme en modules qui ont été développés et approuvés indépendamment afin de créer un code bien structuré, parallèle, et avec des modules accessibles pour être modifiés et même pour être remplacés.

Dans la dernière étape, on a assemblé l'ensemble des modules développés et on les a remodifiés dans la plupart des cas afin que ça donne des résultats les plus proches de ceux à virgule flottante.

Une nouvelle technique pour générer les signaux PWM et contrôler les onduleurs a été proposée dans le quatrième chapitre et a été validée à l'aide Simulink™ / Matlab® dans plusieurs modes de fonctionnement. Cette technique, par son faible taux d'harmoniques et faibles pertes de commutation, d'une part, et sa simplicité d'autre part, peut remplacer le SVPWM dans les applications où les ressources sont très limitées. Les résultats de synthèse montrent que la nouvelle technique consomme peu de ressources, environ 30% (96Slice/358-Slices) de l'espace que la SVPWM consomme dans un FPGA .

Comme futurs travaux pour cette partie on peut explorer la généralisation de la technique approximative pour d'autres structures d'onduleurs où le problème majeur de ces structures est la complexité de l'algorithme de commande, comme dans [6.19, 6.20, 6.21] où les auteurs proposent la mise en œuvre de la SVPWM polyphasique, ou comme dans [6.22, 6.23, 6.24] où les auteurs mettent en œuvre une SVPWM à plusieurs niveaux.

Dans le reste du travail nous avons proposé une librairie de schémas de commande à base de logique floue principalement pour le contrôle d'une machine synchrone à aimant permanent. Les schémas ont été développés d'une façon qu'ils aient des architectures hautement parallèles, ce qui permet un accès rapide et facile pour changer au minimum le

code pour adapter la loi pour une nouvelle gamme de machines ou même de passer d'une loi à une autre en activant ou désactivant juste le bloc voulu via l'entrée *Reset*. Cela permettra de passer d'une loi de commande à une autre selon la plage de fonctionnement voulu ou les critères de performances demandés.

Références

[1.1] H.H. Choi,, J.W. Jung. "Discrete-Time Fuzzy Speed Regulator Design for PM Synchronous Motor," *IEEE Transactions on Industrial Electronics*, Vol. 60, 600-607, (2013).

[1.2] Y. Mingming, W. Feng, C. Gang, Y. Ke, F. Jun. "A Compound Control for PMSM Based on Fuzzy Sliding-mode and Neural Network," 3rd *International Conference on Digital Manufacturing & Automation*. 2012.

[1.3] A. Mishra, J. Makwana, P. Agarwal, S.P. Srivastava. "Mathematical Modeling and Fuzzy Based Speed Control of Permanent Magnet Synchronous Motor Drive," 7th *IEEE Conference on Industrial Electronics and Applications (ICIEA),* 2012.

[1.4] Z. Liu, Y. Wang, J. Du. "Fuzzy Model Predictive Control of a Permanent Magnet Synchronous Motor in Electric Vehicles," 10th *IEEE International Conference on Control and Automation (ICCA) Hangzhou,* 2013.

[1.5] Y.S. Kung, M.S. Wang, C.C. Huang. "Digital Hardware Implementation of Adaptive Fuzzy Controller for AC Motor Drive," *33th IEEE Industrial Electronics Society Conference (IECON),* 2007.

[1.6] Y.S. Kung, T.W. Tsui, N.H. Shieh. "Design and Implementation of a Motion Controller for XYZ Table based on Multiprocessor SoPC," Control and Decision Conference. CCDC. Chinese, 9th. 2009.

[1.7] K. Hakiki, A. Meroufel, V. Cocquempot, M. Chenafa. "A New Adaptive Fuzzy Vector Control for Permanent Magnet Synchronous Motor Drive," 18th *Mediterranean Conference on Control & Automation Congress Palace Hotel, Marrakech, Morocco,* 2010.

[1.8] H. Teiar, H. Chaoui, and P. Sicard, "PMSM control based on adaptive fuzzy logic and sliding mode," in *Industrial Electronics Society, IECON 2013 - 39th Annual Conference of the IEEE*, 2013, pp. 3048-3053.

[1.9] H. Chaoui, P. Sicard. "Adaptive Fuzzy Logic Control of Permanent Magnet Synchronous Machines with Nonlinear Friction", *IEEE Transactions on Industrial Electronics*, Vol. 59. 2012.

[1.10] O. Lopez, J. Alvarez, J. Doval-Gandoy, and F.D. Freijedo, "Multilevel Multiphase Space Vector PWM Algorithm," *Industrial Electronics, IEEE Transactions on,* vol. 55, pp. 1933-1942, 2008.

[1.11] Z. Zhaoyong, L. Tiecai, T. Takahashi, and E. Ho, "Design of a universal space vector PWM controller based on FPGA," in *Applied Power Electronics Conference and Exposition, 2004. APEC '04. Nineteenth Annual IEEE*, 2004, pp. 1698-1702 Vol.3.

[1.12] R.K. Pongiannan and N. Yadaiah, "FPGA based Space Vector PWM Control IC for Three Phase Induction Motor Drive," in *Industrial Technology, 2006. ICIT 2006. IEEE International Conference on*, 2006, pp. 2061-2066.

[1.13] A. Abu-Khudhair, R. Muresan, S.X. Yang. "FPGA Based Real-Time Adaptive Fuzzy Logic Controller," *IEEE International Conference on Automation and Logistics, Hong Kong and Macau. 2010.*

[1.14] D.N. Oliveira, G.A.L. Henn, J. Du. "Design and Implementation of a Mamdani Fuzzy Inference System on an FPGA using VHDL," *Fuzzy Information Processing Society (NAFIPS), 2010 Annual Meeting of the North American,* 2010.

[1.15] S. Sánchez-Solano, A. J. Cabrera, I. Baturone, F.J. Moreno-Velo, *et al.*, "FPGA Implementation of Embedded Fuzzy Controllers for Robotic Applications," *Industrial Electronics, IEEE Transactions on,* vol. 54, pp. 1937-1945, 2007.

[2.1] L. Baghli, "Contribution à la commande de la machine asynchrone, utilisation de la logique floue, des réseaux de neurones et des algorithmes génétiques," Université Henri Poincaré-Nancy I, 1999.

[2.2] P. Borne, J. Rozinoer, J.-Y. Dieulot, and L. Dubois, Introduction à la commande floue: Éd. Technip, 1998.

[2.3] C. Dualibe, M. Verleysen, and G. Jespers, Design of Analog Fuzzy Logic Controllers in CMOS Technologies: Springer, 2003.

[2.4] E.P. Dadios, "Fuzzy Logic – Controls, Concepts, Theories and Applications," 2012.

[2.5] O. Castillo, J. Kacprzyk, P. Melin, W. Pedrycz, O.M. Ross, and R.S. Cruz, Theoretical advances and applications of fuzzy logic and soft computing vol. 42: Springer, 2007.

[2.6] K.M. Passino, S. Yurkovich, and M. Reinfrank, "Fuzzy control," vol. 42: Addison-Wesley, pp. 15-21, 1998.

[2.7] M. Smyej, "Conception d'un correcteur par logique floue pour un convertisseur cc/cc," Mémoire de maîtrise (M.Sc.A.), ed. Université du Québec à Trois-Rivières, 2000.

[2.8] B. Reusch and K.-H. Temme, Computational intelligence in theory and practice: Springer, 2001.

[2.9] F. Valdés, "Design of a fuzzy logic software estimation process," Thèse de doctorat, École de technologie supérieure, 2011.

[2.10] Z. Li, Fuzzy Chaotic Systems: Modeling, Control, and Applications (Studies in Fuzziness and Soft Computing): Springer-Verlag New York, Inc., 2006.

[2.11] M. Hanss, Applied fuzzy arithmetic: Springer, 2005.

[2.12] E.P. Dadios, "Fuzzy Logic – Emerging Technologies and Applications," 2012.

[2.13] K. Ying-Shieh, T. Tai-Wei, and S. Nan-Hui, "Design and implementation of a motion controller for XYZ table based on multiprocessor SoPC," in Control and Decision Conference, 2009. CCDC '09. Chinese, 2009, pp. 241-246.

[2.14] M. Jothi, N.B. Balamurugan, and R. Harikumar, "Fuzzy processor based on VLSI - A review," in Automation, Computing, Communication, Control and Compressed Sensing (iMac4s), 2013 International Multi-Conference on, 2013, pp. 216-222.

[2.15] S. Singh and K.S. Rattan, "Implementation of a fuzzy logic controller on an FPGA using VHDL," in Fuzzy Information Processing Society, 2003. NAFIPS 2003. 22nd International Conference of the North American, 2003, pp. 110-115.

[2.16] S.P J. Vasantha Rani, P. Kanagasabapathy, and A. Sathish Kumar, "Digital Fuzzy Logic Controller using VHDL," in INDICON, 2005 Annual IEEE, 2005, pp. 463-466.

[2.17] P.T. Vuong, A.M. Madni, and J.B. Vuong, "VHDL Implementation for a Fuzzy Logic Controller," in Automation Congress, 2006. WAC '06. World, 2006, pp. 1-8.

[2.18] A. Abu-Khudhair, R. Muresan, and S.X. Yang, "FPGA based real-time adaptive fuzzy logic controller," in Automation and Logistics (ICAL), 2010 IEEE International Conference on, 2010, pp. 539-544.

[2.19] D.N. Oliveira, G.A. de Lima Henn, and O. da Mota Almeida, "Design and implementation of a Mamdani Fuzzy Inference System on an FPGA using VHDL," in Fuzzy Information Processing Society (NAFIPS), 2010 Annual Meeting of the North American, 2010, pp. 1-6.

[3.1] A. D. M.N. Cirstea, J.G. Khor, M. McCormick, Neural and Fuzzy Logic Control of Drives and Power Systems: Elsevier Science, 2002.

[3.2] H. Chaoui, "Implantation sur FPGA d'une loi de commande adaptative neuronale supervisée pour une articulation flexible," Mémoire de maîtrise, Département d'informatique et d'ingénierie, Université du Québec en Outaouais, 2004.

[3.3] V. A. Chandrasetty, VLSI Design: A Practical Guide for FPGA and ASIC Implementations: Springer New York Dordrecht Heidelberg London, 2011.

[3.4] C. Maxfield, FPGAs: Instant Access: Elsevier Ltd, 2008.

[3.5] R. T. Ian Kuon, Jonathan Rose, FPGA Architecture: Survey and Challenges: now Publishers Inc., 2008.

[3.6] H. M. Husain Parvez, Application-Specific Mesh-based Heterogeneous FPGA Architectures: Springer New York Dordrecht Heidelberg London, 2011.

[3.7] U. Meyer-Baese, Digital Signal Processing with Field Programmable Gate Arrays: Springer Berlin Heidelberg New York, 2007.

[3.8] C. I. Ted Huffmire, Thuy D. Nguyen, Timothy Levin, Ryan Kastner, Timothy Sherwood, Handbook of FPGA Design Security: Springer Dordrecht Heidelberg London New York, 2010.

[3.9] J. M. Roger Woods, Gaye Lightbody, Ying Yi, FPGA-based Implementation of Signal Processing Systems: A John Wiley and Sons, Ltd., Publication, 2008.

[3.10] N. M. Karen Parnell, Programmable Logic Design Quick Start Hand Book, 4 ed.: Xilinx, 2003.

[3.11] G. R. Wilson, "Binary Numbers," in Embedded Systems and Computers Architecture, ed: EBSCO Publishing, 2002.

[3.12] D. M. Ercegovac, Land, Tomas, "Basic Fixed-Point Number Representation Systems," in Digital Arithmetic, ed: EBSCO Publishing, 2004.

[3.13] M. Abd-El-Barr, El-Rewini, Hesham, "Computer Arithmetic," in Fundamentals of Computer Organization and Architecture, ed: EBSCO Publishing, 2005.

[4.1] Hamid A. Toliyat, Subhasis Nandi, Seungdeog Choi, Homayoun Meshgin-Kelk, ELECTRIC MACHINES Modeling, Condition Monitoring,and Fault Diagnosis: CRC Press, 2013.

[4.2] O. Drubel, Converter Applications and their Influence on Large Electrical Machines: Springer Heidelberg New York Dordrecht London, 2013.

[4.3] J. F. Gieras, Advancements in Electric Machines: springer.

[4.4] R. De Doncker, D.W. Pulle, André Veltman, Advanced Electrical Drives Analysis, Modeling, Control: Springer Dordrecht Heidelberg London New York, 2011.

[4.5] N. Mohan, P. Jose, T. Brekken, K. Mohapatra, W. Sulkowski, T. Uneland, "Explaining syntheses of three-phase sinusoidal voltages using SV-PWM in the first power electronics course," IEEE Workshop on Power Electronics Education, pp. 40-44, 2005.

[4.6] T. Lifanf, "Study of the SVPWM Converter Based on TMS320F24X," Third international Conference on Intelligent System Design and Engineering Applications, pp. 1316-1319, 2013.

[4.7] R. Cordero, J. Pinto and J. De Soares, "New Simplification of SV-PWM based on conditional of the reference vector," IEEE International Power Electronics Conference, pp. 2992–2999, 2010.

[4.8] A.O. Rait and P. Bhosale, "FPGA implementation of space vector PWM for speed control of 3-phase induction motor," International conference on Recent advancements in Electrical, Electronics and control engineering, pp. 221-225, 2011.

[4.9] M. Lamich, J. Balcells and D. Gonzalez, "New method for obtaining SV-PWM patterns following an arbitrary reference," IEEE 28th Annual Conference Industrial electronics Society, Vol. 1, pp. 18-22, 2002.

[4.10] Zhou Yuan, Xu Fei-peng and Zhou Zhao-yong "Realization of an FPGA-Based Space-Vector PWM Controller," Power Electronics and motion Control Conference, IPEMC'06, Vol. 1, pp. 1-5, 2006.

[4.11] R. Rajendran and N. Devarajan, "Intelligent Control Fuzzy Logic Applications," Second international Conference on Computer and Electrical Engineering, Vol. 2, pp.422-426, 2009.

[4.12] Zhou Yuan, Xu Fei-peng and Zhou Zhao-yong "Realization of an FPGA-Based Space-Vector PWM Controller," Power Electronics and motion Control Conference, IPEMC'06, Vol. 1, pp. 1-5, 2006.

[4.13] Z. Zhou, T. Li, T. Takahashi, and E. Ho "Design of a Universal Space Vector PWM Controller Based on FPGA," Nineteenth Annual IEEE Applied Power Electronics Conference and Exposition, APEC '04, Vol. 3, pp. 1698-1702, 2004.

[4.14] S. Boukaka, H. Chaoui, P. Sicard "FPGA Implementation of SVPWM", 11th IEEE International New Circuits and Systems Conference (NEWCAS), 2013.

[4.15] A.O. Rait and P. Bhosale, "FPGA implementation of space vector PWM for speed control of 3-phase induction motor," International conference on Recent advancements in Electrical, Electronics and control engineering, pp. 221-225, 2011.

[4.16] J. Osorio, P. Ponce, A. Molina "FPGA-based space vector PWM with Artificial Neural Networks," Nineteenth International Conference on Electrical Engineering, Computing Science and Automatic Control, CCE '09, pp. 1-6, 2012.

[4.17] H. Chaoui, P. Sicard. "Adaptive Fuzzy Logic Control of Permanent Magnet Synchronous Machines with Nonlinear Friction", IEEE Transactions on Industrial Electronics, Vol. 59. 2012.

[4.18] S. Tole, W.J. Hwa, A. Jidin, and N.R.N. Idris. "A Simple Approach of Space-Vector Pulse Width Modulation Realization Based on Field Programmable Gate Array." Electric Power Components and Systems 38, no. 14 (2010): 1546-1557.

[5.1] H. Chaoui, P. Sicard. "Adaptive Fuzzy Logic Control of Permanent Magnet Synchronous Machines with Nonlinear Friction", IEEE Transactions on Industrial Electronics, Vol. 59, No. 2, 1123-1133, 2012.

[5.2] H. Teiar, H. Chaoui, P. Sicard, "PMSM control based on adaptive fuzzy logic and sliding mode," Industrial Electronics Society, IECON 2013 - 39th Annual Conference of the IEEE , pp.3048,3053, 10-13 Nov. 2013.

[5.3] H. Teiar, S. Boukaka, H. Chaoui, P. Sicard, "Adaptive fuzzy logic control structure of PMSMs," Industrial Electronics (ISIE), 2014 IEEE 23rd International Symposium on, pp.745,750, 1-4 June 2014.

[5.4] S. Singh, K.S. Rattan. "Implementation of a Fuzzy Logic Controller on an FPGA using VHDL", 22nd International Conference of the North American Fuzzy Information Processing Society, 2003.

[5.5] S.P. Joy, V. Rani, P. Kanagasabapathy, A.S. Kumar. "Digital Fuzzy Logic Controller using VHDL", IEEE Indicon Conference, Chennai, India. 2005.

[5.6] J.L.G. Vazquez, O. Castillo, L.T.A. Bustos. "A Generic Approach to Fuzzy Logic Controller Synthesis on FPGA", IEEE International Conference on Fuzzy Systems, Vancouver, BC, Canada. 2006.

[5.7] N.J. Patil, R.H. Chile, L.M. Waghmare, "Implementation of Model Reference Adaptive Fuzzy Controller", IET-UK International Conference on Information and Communication Technology in Electrical Sciences. 2007.

[6.1] G. Renukadevi and K. Rajambal "FPGA implementation of SVPWM technique for asymmetrical six-phase VSI," International Conference on Emerging Trends in Electrical Engineering and Energy Management, ICETEEEM, pp. 333 - 338, 2012.

[6.2] J. Alvarez, O. Lopez, F.D. Freijedo, J. Doval-Gandoy "Digital Parameterizable VHDL Module for Multilevel Multiphase Space Vector PWM," IEEE Transactions on Industrial Electronics, Vol. 58, pp. 3946-3957, 2011.

[6.3] P. Sandulescu, L. Idkhajine, S. Cense, F. Colas, X. Kestelyn, E. Semail, A. Bruyere "FPGA implementation of a general Space Vector approach on a 6-leg voltage source inverter," 37th Annual Conference on IEEE Industrial Electronics Society Digital, IECON '37, pp. 3482-3487, 2011.

[6.4] A. Moreno Garzon, J.M. Garzon Rey, G. Ramos, F. Segura-Quijano "Hardware and software performance assessment of a multilevel inverter control," 2013 Workshop on Power Electronics and Power Quality Applications, PEPQA, 2013.

[6.5] Yaofei Han, Xiaohong Fan, Zhangfei Zhao "The study and realization of three-level SVPWM algorithm for high power inverter," International Conference on Electrical and Control Engineering, ICECE, pp. 6177-6180, 2011.

[6.6] P. Thirumurugan, P.S. Manoharan, M. Valanrajkumar "VLSI based space vector pulse width modulation switching control," IEEE International Conference on

Advanced Communication Control and Computing Technologies, ICACCCT, pp. 338-342, 2012.

Annexe A – Extraits des codes de simulation

Dans cette annexe on présente des schémas qui montrent des parties du code utilisé dans ce travail.

A- Code Matlab utilisé pour valider l'additionneur (Première partie)

```
clc;        % Effacement des commandes precedement tappees dans la Command Window

%-- Initialisation ----------------------------------------
Nseq = 100; % nombre de données passant dans l'additionneur

A=0.5*randn(Nseq,1); % Entrées aléatoires de l'entrée A de l'Addtionneur:
B=0.5*randn(Nseq,1); % Entrées aléatoires de l'entree B de l'Addtionneur;

%%=========================================================================
% Additionneur ==> VIRGULE FLOTTANTE (précision maximale)
%%=========================================================================
C=A+B;


%%=========================================================================
% Additionneur ==> VIRGULE FIXE (précision limitée par le nb. de bit)
%%=========================================================================
N_Bit = 16;          % Nombre de bit considerée pour representer les donnees => addition

%--> Quantification des signaux A et B ----------------------
for n=1:Nseq
    A_norm(n,1)=func_sampling_M7(A(n,1),N_Bit);
    B_norm(n,1)=func_sampling_M7(B(n,1),N_Bit);
end

for n=1:Nseq
    C_fixe(n,1)=A_norm(n,1)+B_norm(n,1);
end

%--> Quantification du signal C_vfX -------------------------
for n=1:Nseq
    C_norm(n,1)=func_sampling_M7(C_fixe(n,1),N_Bit);
end

%%=========================================================================
% Courbe de comparaison entre Virgule flottante et fixe
%%=========================================================================
figure;
stem(A);
hold on;
stem(A_norm,'r');
hold off;
title('Entrée A de additionneur ==> flottante vs Fixe')
legend('Flottante','Fixe')

figure;
stem(B);
hold on;
stem(B_norm,'r');
hold off;
title('Entrée B de additionneur ==> flottante vs Fixe')
legend('Flottante','Fixe')

figure;
stem(C);
hold on;
stem(C_norm,'r');
hold off;
title('Sortie C de additionneur ==> flottante vs Fixe')
legend('Flottante','Fixe')


%=== Transfert de données ================================================
%---  ENTREE : A_norm contruction du fichiers de test
fid=fopen('A_matlab.txt','wt');
for i=1:Nseq
    fprintf(fid,func_FixedNumber2StdLogicVector_M7(A_norm(i,1),N_Bit));
    fprintf(fid,'\n');
end
fclose(fid);

%---  ENTREE : B_norm contruction du fichiers de test
fid=fopen('B_matlab.txt','wt');
for i=1:Nseq
    fprintf(fid,func_FixedNumber2StdLogicVector_M7(B_norm(i,1),N_Bit));
    fprintf(fid,'\n');
end
fclose(fid);

%---  SORTIE : C_vfX_norm contruction du fichiers de test
fid=fopen('C_matlab.txt','wt');
for i=1:Nseq
    fprintf(fid,func_FixedNumber2StdLogicVector_M7(C_norm(i,1),N_Bit));
    fprintf(fid,'\n');
end
fclose(fid);

%=========================================================================
```

B- Code Matlab utilisé pour valider l'additionneur (Deuxième partie)

```
clc;        % Effacement des commandes precedement tappees dans la Command Window

%=== Initialisation ================================================
N_Bit=16;

%=== Chemin des fichiers à lire ====================================
path(path,pwd);

%=== Nom des fichiers ==============================================
nom_fichier_modelSim='C_Msim.txt';
nom_fichier_matlab='C_matlab.txt';

%=== Lecture du fichier ModeSim ====================================
pid_yb = fopen(nom_fichier_modelSim,'rb');
i = 1;
while 1
    line_b = fgetl(pid_yb);
    if ~ischar(line_b), break, end
    C_norm_modelSim_bin(i,:) = line_b;
    i=i+1;
end
fclose(pid_yb);

for i=1:size(C_norm_modelSim_bin,1)
    C_norm_modelSim_p2(i,1) = func_StdLogicVector2FixedNumber_M7(C_norm_modelSim_bin(i,:),N_Bit)
end

%=== Lecture du fichier matlab virgule fixe ========================
pid_yb2 = fopen(nom_fichier_matlab,'rb');
i = 1;
while 1
    line_b = fgetl(pid_yb2);
    if ~ischar(line_b), break, end
    C_norm_matlab_bin(i,:) = line_b;
    i=i+1;
end
fclose(pid_yb2);

for i=1:size(C_norm_matlab_bin,1)
    C_norm_matlab_bin_p2(i,1) = func_StdLogicVector2FixedNumber_M7(C_norm_matlab_bin(i,:),N_Bit)
end

%====================================================================
% Courbes de comparaisons virgules flottantes, virgules fixes et ModelSim
%====================================================================

%=== Comparaison des résultats VHDL et MATLAB ======================
figure;
plot(C_norm_modelSim_p2,'b')
hold on
plot(C_norm_matlab_bin_p2,'r')
hold off
legend('C modelSim','C Matlab')
xlabel('Echantillons')
ylabel( 'Amplitudes')
title('Additionneur - Sortie ModelSim et Matlab')
axis([1 length(C_norm_matlab_bin_p2) -1 1])

%=== Comparaison des résultats VHDL et MATLAB ==> erreur ===========
errlbot=2^-(N_Bit-1)*ones(length(C_norm_matlab_bin_p2),1);
% ==> erreur maximale acceptée de 1 bit
figure;
plot(C_norm_matlab_bin_p2-C_norm_modelSim_p2,'g')
hold on
plot(errlbot,'r')
hold off
legend('Matlab vfix MOINS ModelSim','erreur de 1 bit')
xlabel('Échantillons')
ylabel( 'Amplitudes')
title('Additionneur - Sortie Matlab MOINS ModelSim')
%====================================================================
```

C- Code Matlab utilisé pour passer de la virgule flottante à virgule fixe.

```
%%-------------------------------------------------------------------------
function [Data_dec_Norm]=func_sampling_M7(Data_dec,Nbit)


if Data_dec>=1-2^-(Nbit-1)
    Data_dec_Norm=1-2^-(Nbit-1);
elseif Data_dec<-1+2^-(Nbit-1)
    Data_dec_Norm=-1;
else
    temp0=abs(Data_dec);
    temp=temp0;
    temp2=0;
    use_n=zeros(1,Nbit);
    for n=1:(Nbit-1)
        if temp>=2^(-n)
            use_n(1,n+1)=1;
            temp=temp-2^(-n);
            temp2=temp2+2^(-n);
        end
    end

    temp3=temp2+2^-(Nbit-1);
    if abs(temp0-temp3)<abs(temp0-temp2)
        temp_fin=temp3;
    else
        temp_fin=temp2;
    end

    if Data_dec>=0
        Data_dec_Norm=temp_fin;
    else
        Data_dec_Norm=-1*temp_fin;
    end
end

%%-------------------------------------------------------------------------
```

D- Matlab utilisé pour passer du décimal au binaire

```
function [Data_bin]=func_FixedNumber2StdLogicVector_M7(Data_dec_Norm,Nbit)

if Data_dec_Norm>=0
    val_bin=zeros(1,Nbit);
    temp=abs(Data_dec_Norm);
    for n=1:(Nbit-1)
        if temp-2^(-n)>=0
            val_bin(1,n+1)=1;
            temp=temp-2^(-n);
        end
    end
    Data_bin_vec=val_bin;
else
    if Data_dec_Norm==-1
          Data_bin_vec=[1 zeros(1,Nbit-1)];

    else
        val_bin=ones(1,Nbit);
        temp=abs(Data_dec_Norm);
        for n=1:(Nbit-1)
            if temp-2^(-n)>=0
                val_bin(1,n+1)=0;
                temp=temp-2^(-n);
            end
        end
        if val_bin==[ones(1,Nbit)]
            Data_bin_vec=val_bin;
        else
            Data_bin_vec=val_bin;
            chg=0;
            for m=Nbit:-1:2
                if val_bin(1,m)==0 && chg==0
                    Data_bin_vec(1,m)=1;chg=2;
                elseif val_bin(1,m)==1 && chg==0
                    Data_bin_vec(1,m)=0;chg=1;
                elseif val_bin(1,m)==0 && chg==1
                    Data_bin_vec(1,m)=1;chg=2;
                elseif val_bin(1,m)==1 && chg==1
                    Data_bin_vec(1,m)=0;chg=1;
                end
            end
        end
    end
end

Data_bin=sprintf('%01d',Data_bin_vec);
```

E- Code Matlab utilisé pour passer du binaire au décimal

```
%%-------------------------------------------------------------------------
function [Data_dec_conv]=func_StdLogicVector2FixedNumber_M7(Data_bin,Nbit)

conv_dec=bin2dec(Data_bin);
neg=0;
%-- Nombres negatifs ------------------------------------
if conv_dec==2^(Nbit-1)
    Data_dec_conv=-1;
else
    if conv_dec>2^(Nbit-1)
        neg=1;
        conv_dec=conv_dec-2^(Nbit-1);
        temp1=conv_dec;
        use_n=zeros(1,Nbit);
        for n=2:Nbit
            if temp1-2^(Nbit-n)>=0
                use_n(1,n)=1;
                temp1=temp1-2^(Nbit-n);
            else
                use_n(1,n)=0;
            end
        end
        %--- inversion des bits ----
        temp2=use_n;
        for n=1:Nbit
            if temp2(1,n)==0
                temp2(1,n)=1;
            else
                temp2(1,n)=0;
            end
        end
        %--- Complement ------------
        val_bin=temp2;
        chg=0;
        comp_2=val_bin;
        for m=Nbit:-1:2
            if val_bin(1,m)==0 && chg==0
                comp_2(1,m)=1;chg=2;
            elseif val_bin(1,m)==1 && chg==0
                comp_2(1,m)=0;chg=1;
            elseif val_bin(1,m)==0 && chg==1
                comp_2(1,m)=1;chg=2;
            elseif val_bin(1,m)==1 && chg==1
                comp_2(1,m)=0;chg=1;
            end
        end
        out_trt=comp_2;
    else
        %-- Nombres positifs ------------------------------------
        temp3=conv_dec;
        out_trt=zeros(1,Nbit);
        for n=2:Nbit
            if temp3-2^(Nbit-n)>=0
                out_trt(1,n)=1;
                temp3=temp3-2^(Nbit-n);
            else
                out_trt(1,n)=0;
            end
        end
    end
    %-- Conversion binaire => decimal ----------------------
    Data_dec_conv=0;
    for n=1:Nbit-1
        if out_trt(n+1)==1
            Data_dec_conv=Data_dec_conv+2^-(n);
        end
    end
    if neg==1
        Data_dec_conv=Data_dec_conv*(-1);
    end
end
%%-------------------------------------------------------------------------
```

F- Code VHDL utilisé pour réaliser un additionneur

```
-- déclaration des bibliothèques
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

-- déclaration de l'entité
entity ADD_ex_16 is
    port ( A_MSim,B_MSim: in std_logic_vector(15 downto 0);
           reset        : in std_logic;
           C_MSim       : out std_logic_vector(15 downto 0)
         );
 end ADD_ex_16;

-- déclaration de l'architecture
architecture Addition_16 of ADD_ex_16 is

begin
    process (A_MSim,B_MSim,Reset)
        variable C_tmp :std_logic_vector(15 downto 0);
        begin
            if Reset='1' then
                C_tmp:="0000000000000000";
            else
            C_tmp:= A_MSim+B_MSim;
            if A_MSim(15)='0' and B_MSim(15)='0' and C_tmp(15)='1' then
                C_tmp:="0111111111111111";
            elsif A_MSim(15)='1' and B_MSim(15)='1' and C_tmp(15)='0' then
                C_tmp:="1000000000000000";
            end if;
        end if;
            C_MSim <= C_tmp;
        end process;
end Addition_16;
-----------------------------------------
```

G- Exemple d'un code VHDL Test

```
-- déclaration des bibliothèques
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_textio.all;  -- lecture et ecriture de fichiers

use std.textio.all;
--use work.ADD_ex_16;                --Appel de l'additionneur compié

-- déclaration de l'entité
entity test_ADD_ex_16 is
end test_ADD_ex_16 ;
-- déclaration de l'architecture du testBench
architecture stimuli of test_ADD_ex_16 is
 -- lecture fichier txt
 file file_in_A_norm : text open read_mode is
      "C:\Documents and Settings\labo\Bureau\ADD_16bits\A_matlab.txt";
 file file_in_B_norm : text open read_mode is
      "C:\Documents and Settings\labo\Bureau\ADD_16bits\B_matlab.txt";
 -- écriture fichier txt
 file file_out_C_MSim : text open write_mode is
      "C:\Documents and Settings\labo\Bureau\ADD_16bits\C_Msim.txt";
 -- déclaration du composant testé
component ADD_ex_16 is
    port ( A_MSim,B_MSim : in std_logic_vector(15 downto 0);
           reset         : in std_logic;
           C_MSim        : out std_logic_vector(15 downto 0)
         );
end component;
 -- déclaration des signaux du composant
signal A_MSim,B_MSim,C_MSim : std_logic_vector(15 downto 0);
signal reset: std_logic;
constant PERIOD_test: time :=50 ns;
begin
     ADD1 : ADD_ex_16  port map (A_MSim,B_MSim,reset,C_MSim);
     process
     variable L_A_MSIm,L_B_MSIm,L_C_MSIm : line;
     variable value_A_MSIm,value_B_MSIm,value_C_MSIm : std_logic_vector(15 downto 0);
         begin
         reset<='0';
         -- Lecture de A
         readline(file_in_A_norm,L_A_MSIm);
         read (L_A_MSIm,value_A_MSIm);
         A_MSIm<=value_A_MSIm;
         -- Lecture de B
         readline(file_in_B_norm,L_B_MSIm);
         read (L_B_MSIm,value_B_MSIm);
         B_MSIm<=value_B_MSIm;
         wait for PERIOD_test;
         -- Ecriture de C
         value_C_MSIm:=C_MSim;
         write(L_C_MSIm,value_C_MSIm);
         writeline(file_out_C_MSim,L_C_MSIm);
         end process;
end stimuli;
```

H- Code VHDL utilisé pour réaliser un soustracteur

```
-- déclaration des bibliothèques
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;  --pour faire l'addition

-- déclaration de l'entité
entity soustrac_16 is
    port ( Sous1,Sous2: in std_logic_vector(15 downto 0);
           reset_sous         : in std_logic;
           Erreur             : out std_logic_vector(15 downto 0)
          );
 end soustrac_16;

-- déclaration de l'architecture
architecture soustracteur_16 of soustrac_16 is
begin
    process(Sous1,Sous2,reset_sous)
variable temp1 : std_logic_vector(15 downto 0);

       begin


     temp1:= Sous1 - Sous2;

           if reset_sous='1' then
            temp1:="0000000000000000";
         end if;


         if temp1(15)='0' and Sous1(15)='1' and Sous2(15)='0' then
         temp1 :="1000000000000000";


         elsif temp1(15)='1' and Sous1(15)='0' and Sous2(15)='1' then
         temp1 :="0111111111111111";
          end if;


     Erreur <= temp1;

 end process;
end soustracteur_16;
-------------------------------------------
```

I- Code VHDL utilisé pour réaliser un multiplieur

```
-- déclaration des bibliothèques
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;  --pour faire l'addition

-- déclaration de l'entité
entity Mul_16bits is
    port ( D_MSim,E_MSim : in std_logic_vector(15 downto 0);
           reset         : in std_logic;
           F_MSim        : out std_logic_vector(31 downto 0)
          );
 end Mul_16bits;

-- déclaration de l'architecture
architecture multiplication of Mul_16bits is
begin
    process(D_MSim,E_MSim,Reset)
        variable F_tmp :std_logic_vector(31 downto 0);
        begin
        if Reset = '1' then
            F_tmp := "00000000000000000000000000000000";
        else
          if
            D_MSim = "1000000000000000" and E_MSim = "1000000000000000" then
            F_tmp := "01111111111111111111111111111111";

          elsif
            D_MSim = "0111111111111111" and E_MSim = "0111111111111111" then
            F_tmp := "01111111111111111111111111111111";

          elsif
             D_MSim = "0111111111111111" and E_MSim = "1000000000000000" then
             F_tmp := "10000000000000000000000000000000";
           elsif
             D_MSim = "0111111111111111" and E_MSim = "1000000000000000" then
          F_tmp := "10000000000000000000000000000000";
          else
          F_tmp := D_MSim * E_MSim;
          F_tmp := F_tmp(31) & F_tmp(29 downto 0) & '0';
          end if;
          F_MSim <= F_tmp;
        end if;
 end process;
end multiplication;
-------------------------------------------
```

J- Code VHDL utilisé pour réaliser un registre de décalage

```
-- Declaration des bibliotheques -----
library ieee;
use ieee.std_logic_1164.all;

-- Entité ----------------------------
entity Bas_D is
port( Rin: in std_logic_vector(0 to 7);
      RST,CLK: in std_logic;
      Rout: out std_logic_vector(0 to 7));
end Bas_D;

-- Architecture-----------------------
architecture Bas_D_beh of Bas_D is
signal Rout_tmp: std_logic_vector(0 to 7);
begin
    process(RST,CLK)
    begin

    if RST='1' then
    Rout_tmp<="00000000"; -- on peut aussi écrire

    elsif CLK'event and CLK='1' then

       if Rin="XXXXXXXX" then
      Rout_tmp<="00000000";
       else
      Rout_tmp<=Rin;
        end if;

    end if;
    end process;

    Rout<=Rout_tmp;

    end Bas_D_beh;
-----------------------------------------
```

K- Code VHDL utilisé pour réaliser une troncation

```
Library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity troncateur is
    port(tronc_in: in std_logic_vector(15 downto 0);
         tronc_out: out std_logic_vector(7 downto 0));
end troncateur ;


architecture tronc of troncateur is
begin
    tronc_out <= tronc_in(15 downto 8);
 end tronc;
```

Annexe B – Résultats de synthèse des lois de commande

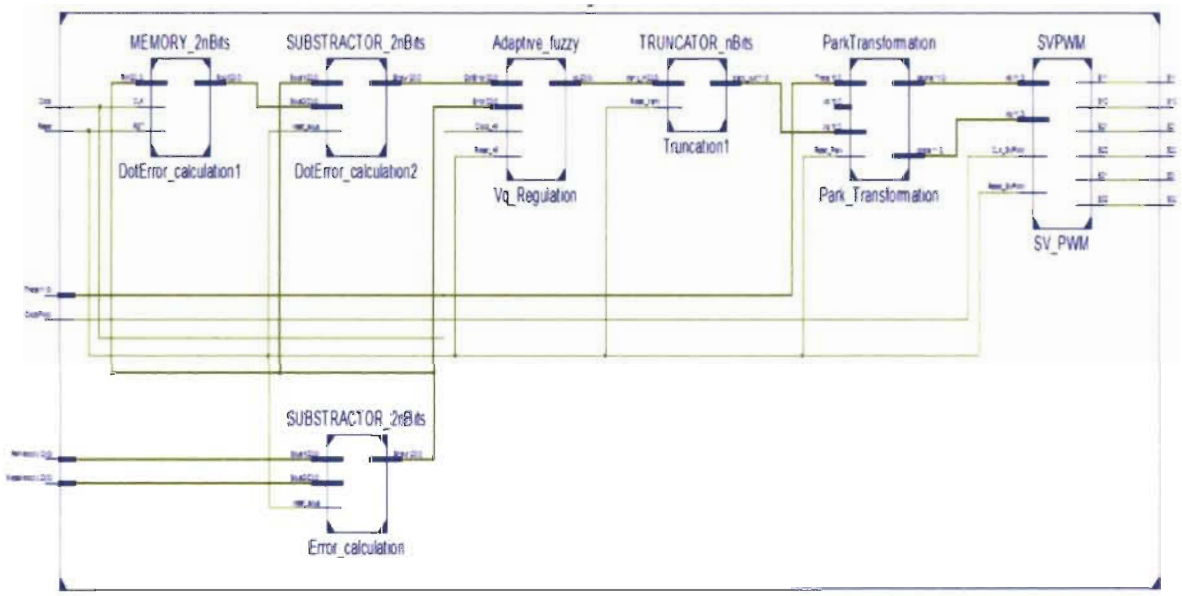
Dans Cette annexe on expose les résultats de synthèse des trois lois de contrôle avec le code VHDL utilisé pour faire sortir ces schémas.

A- Code VHDL final utilisé pour implémenter les trois lois de commande

```
LIBRARY ieee;
use ieee.std_logic_1164.all;

ENTITY PMSM_AdaptiveFLC is
Generic (NmbrBits: integer := 12);
PORT( RefVelocity, MeasVelocity, RefId, MeasId, MeasIq   : in std_logic_vector((2*NmbrBits)-1 downto 0);
      Theta   : in std_logic_vector((NmbrBits-1) downto 0);
      Reset, Clock, ClockPWM : in std_logic;
      S11, S12, S21, S22, S31, S32 : out std_logic
    );
END PMSM_AdaptiveFLC;
ARCHITECTURE Archi_PMSM_AdaptiveFLC of PMSM_AdaptiveFLC is
-- Composants du PMSM_AdaptiveFLC--

Component Adaptive_fuzzy is
PORT( Error, DotError   : in std_logic_vector((2*NmbrBits)-1 downto 0);
      Reset_AF, Clock_AF : in std_logic;
      Vq : out std_logic_vector ((2*NmbrBits)-1 downto 0)
    );
end Component ;

Component SlidingMode is

    port ( MeasVelocity, MeasIq, RefId, MeasId  : in std_logic_vector ((2*NmbrBits-1) downto 0);
           Reset_SlidingMode      : in std_logic;
           Vd  : out std_logic_vector ((2*NmbrBits-1) downto 0)
           );
 end Component ;

Component ParkTransformation is
    port ( Theta, Vd, Vq   : in std_logic_vector ((NmbrBits-1) downto 0);
           Reset_Park      : in std_logic;
           Valpha, Vbeta   : out std_logic_vector ((NmbrBits-1) downto 0)
           );
end Component ;

Component SVPWM is
PORT( Vd, Vq : in std_logic_vector((NmbrBits-1) downto 0);
      Reset_SVPWM, CLK_SVPWM : in std_logic;
      S11, S12, S21, S22, S31, S32 : out std_logic
    );
 end Component ;

COMPONENT TRUNCATOR_nBits
    port(tranc_in: in std_logic_vector((2*NmbrBits)-1 downto 0);
         Reset_tranc: in std_logic;
         tranc_out: out std_logic_vector((NmbrBits-1) downto 0));
end COMPONENT ;


COMPONENT SUBSTRACTOR_InBits
    port ( Sous1,Sous2: in std_logic_vector((2*NmbrBits)-1 downto 0);
           reset_sous        : in std_logic;
           Erreur        : out std_logic_vector((2*NmbrBits)-1 downto 0)
         );
```

```
56  END COMPONENT;
57
58  COMPONENT MEMORY_2nBits is
59  port( Rin: in std_logic_vector((2*NmbrBits)-1 downto 0);
60        RST,CLK: in std_logic;
61        Rout: out std_logic_vector((2*NmbrBits)-1 downto 0));
62  END COMPONENT;
63
64
65      signal  Vq_tmp1, Vd_tmp1, Valpha_tmp,  Vbeta_tmp: std_logic_vector ((NmbrBits-1) downto 0);
66      signal  Vq_tmp, Vd_tmp,Error, Error_1, DotError,ErrorId, Error_1Id, DotErrorId : std_logic_vector ((2*NmbrBits)-1 downto 0);
67
68    BEGIN
69
70                                --////// Régulateur de la Tension Vq //////--
71
72  Error_calculation  : SUBSTRACTOR_2nBits port map( Sous1 => RefVelocity, Sous2 => MeasVelocity, reset_sous => Reset, Erreur => Error);
73
74  DotError_calculation1 : MEMORY_2nBits port map( Rin => Error, RST => Reset, CLK => Clock, Rout => Error_1);
75
76  DotError_calculation2  : SUBSTRACTOR_2nBits port map( Sous1 => Error, Sous2 => Error_1, reset_sous => Reset, Erreur => DotError);
77
78  Vq_Regulation  : Adaptive_fuzzy PORT map( Error => Error, DotError => DotError, Reset_AF => Reset, Clock_AF => Clock, Vq => Vq_tmp);
79
80
81                                --////// Régulateur de la Tension Vd //////--
82
83              -- Régulateur Flou
84
85  Error_calculation1  : SUBSTRACTOR_2nBits port map( Sous1 => RefId, Sous2 => MeasId, reset_sous => Reset, Erreur => ErrorId);
86
87  DotError_calculation3 : MEMORY_2nBits port map( Rin => ErrorId, RST => Reset, CLK => Clock, Rout => Error_1Id);
88
89  DotError_calculation4  : SUBSTRACTOR_2nBits port map( Sous1 => ErrorId, Sous2 => Error_1Id, reset_sous => Reset, Erreur => DotErrorId);
90
91  Vd_Regulation  : Adaptive_fuzzy PORT map( Error => ErrorId, DotError => DotErrorId, Reset_AF => Reset, Clock_AF => Clock, Vq => Vd_tmp);
92
93                    -- Régulateur Mode Glissant
94
95  --Vd_Regulation   : SlidingMode port map( MeasVelocity => MeasVelocity, MeasIq => MeasIq, RefId => RefId, MeasId => MeasId, Reset_SlidingMode => Reset, Vd => Vd_tmp);
96
97  Truncation1           :  TRUNCATOR_nBits port map(tranc_in => Vq_tmp, Reset_tranc => Reset,
98                                                    tranc_out => Vq_tmp1);
99  Truncation2           :  TRUNCATOR_nBits port map(tranc_in => Vd_tmp, Reset_tranc => Reset,
100                                                   tranc_out => Vd_tmp1);
101
102             -- Imposition de la Tension Vd a zéro
103
104 --Vd_tmp <= "000000000000";
105
106                               --////// Transformation Park //////--
107
108 Park_Transformation  : ParkTransformation port map( Theta => Theta, Vd => Vd_tmp1, Vq => Vq_tmp1, Reset_Park => Reset, Valpha => Valpha_tmp, Vbeta => Vbeta_tmp);
109
110
111                               --////// SVPWM //////--
112
113
114 SV_PWM   : SVPWM PORT map( Vd => Valpha_tmp,Vq => Vbeta_tmp, Reset_SVPWM => Reset, CLK_SVPWM => ClockPWM, S11 => S11, S12 => S12, S21 => S21, S22 => S22, S31 => S31, S32 =>
115
116
117 END Archi_PWM_AdaptiveFLC;
118
```

B- Schéma du circuit « *Première loi de contrôle* » réalisé sur un FPGA.



C- Code utilisé pour calculé les fonctions d'appartenance

```
LIBRARY ieee;
use ieee.std_logic_1164.all;
ENTITY BELONGING_POURCENTAGES is
Generic (NmbrBits: integer := 12);
PORT( Data1, Data2 : in std_logic_vector((2*NmbrBits-1) downto 0);
      Reset_BelonGP : in std_logic;
      SwitchingSignal : in std_logic_vector (3 downto 0);
      P11, P12, P13, P14, P15,
      P21, P22, P23, P24, P25 : out std_logic_vector ((2*NmbrBits-1) downto 0)
   );
END BELONGING_POURCENTAGES;
ARCHITECTURE Archi_BELONGING_POURCENTAGES of BELONGING_POURCENTAGES is
-- Composants du BELONGING_POURCENTAGES--

Component Pourcentage
    port ( Data,Bi,Bo,Bs      : in std_logic_vector ((2*NmbrBits-1) downto 0);
           Reset_Prct         : in std_logic;
           Pourcentage  : out std_logic_vector ((2*NmbrBits-1) downto 0)
           );
 end Component ;

Component Muxl
    port ( I_nfini, NB, N_S, Z, PS, PB, Infini, data1, data2 : in std_logic_vector((2*NmbrBits)-1 downto 0);
           SwitchingSignal : in std_logic_vector(3 downto 0);
           Reset_Muxl : in std_logic;
           Ptage_In1, Ptage_In2, Ptage_In3, data : out std_logic_vector((2*NmbrBits)-1 downto 0)
           );
 end Component ;

Component Demuxl
    port ( P : in std_logic_vector((2*NmbrBits)-1 downto 0);
           SwitchingSignal : in std_logic_vector(3 downto 0);
           Reset_Demuxl : in std_logic;
           P1_1,P1_2,P1_3,P1_4,P1_5,
           P2_1,P2_2,P2_3,P2_4,P2_5  : out std_logic_vector((2*NmbrBits)-1 downto 0)
           );
 end Component ;


-- Signaux de liaison--
    signal   int1, int2, int3, int4, int5, int6, int7, int8, int9, int10,Ptage_In1,
             Ptage_In2,Ptage_In3, data, Pourcentages: std_logic_vector ((2*NmbrBits-1) downto 0);

    constant P_NB     : std_logic_vector ((2*NmbrBits-1) downto 0) := "1111111010111000010010010"; --"1111111010111001";
    constant P_NS     : std_logic_vector ((2*NmbrBits-1) downto 0) := "1111111101011100000101001";          --1111111101011101
    constant P_Z      : std_logic_vector ((2*NmbrBits-1) downto 0) := "000000000000000000000000";
    constant P_PS     : std_logic_vector ((2*NmbrBits-1) downto 0) := "000000001010001111010111";      --000000001010001l
    constant P_PB     : std_logic_vector ((2*NmbrBits-1) downto 0) := "000000010100011110101110"; --00000001010001l1
    constant Infini   : std_logic_vector ((2*NmbrBits-1) downto 0) := "011111111111111111111111";
    constant I_nfini  : std_logic_vector ((2*NmbrBits-1) downto 0) := "100000000000000000000000";
    constant V_NB     : std_logic_vector ((2*NmbrBits-1) downto 0) := "111111101011100000101001O";
    constant V_NS     : std_logic_vector ((2*NmbrBits-1) downto 0) := "111111101011100000101001";
    constant V_Z      : std_logic_vector ((2*NmbrBits-1) downto 0) := "000000000000000000000000";
    constant V_PS     : std_logic_vector ((2*NmbrBits-1) downto 0) := "000000001010001111010111";


  BEGIN


Multiplexeur : Muxl port map ( I_nfini => I_nfini, NB => P_NB, N_S => P_NS, Z => P_Z, PS => P_PS, PB => P_PB,
                               Infini => Infini, data1 => Data1, data2 => Data2,SwitchingSignal => SwitchingSignal,
                               Reset_Muxl => Reset_BelonGP,Ptage_In1 => Ptage_In1, Ptage_In2 => Ptage_In2,
                               Ptage_In3 => Ptage_In3, data => data );

Prctg_Erreur_1  :  Pourcentage port map ( Data => data,Bi => Ptage_In1,Bo => Ptage_In3,Bs => Ptage_In2,Reset_Prct => Reset_BelonGP,
                                          Pourcentage => Pourcentages);

Demultiplexeur : Demuxl port map ( P => Pourcentages, SwitchingSignal => SwitchingSignal, Reset_Demuxl => Reset_BelonGP
                                  ,P1_1 => P11,P1_2 => P12,P1_3 => P13, P1_4 => P14,P1_5 => P15,P2_1 => P21,P2_2 => P22,
                                   P2_3 => P23,P2_4 => P24,P2_5 => P25);


END Archi_BELONGING_POURCENTAGES;
```

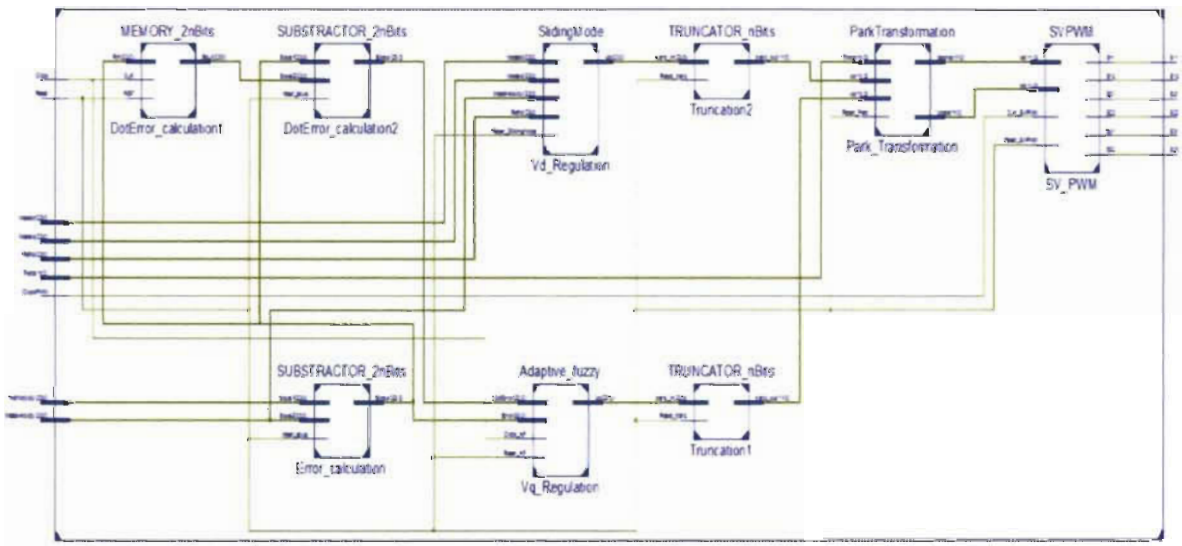
```
-- déclaration des bibliothèques
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;  --pour faire l'addition

-- déclaration de l'entité
entity Pourcentage is
Generic (NmbrBits: integer := 12);
    port ( Data,Bi,Bc,Bs      : in std_logic_vector ((2*NmbrBits-1) downto 0);
           Reset_Prct         : in std_logic;
           Pourcentage  : out std_logic_vector ((2*NmbrBits-1) downto 0)
           );
 end Pourcentage ;

-- déclaration de l'architecture
architecture Pourcentage_1 of Pourcentage is


COMPONENT MULTIPLIER_2nBits
        port(D_MSim, E_MSim: in std_logic_vector((2*NmbrBits-1) downto 0);
             Reset_M: in std_logic;
             F_MSim: out std_logic_vector((4*NmbrBits)-1 downto 0));
END COMPONENT;

COMPONENT SUBSTRACTOR_2nBits
    port ( Sous1,Sous2: in std_logic_vector((2*NmbrBits-1) downto 0);
           reset_sous         : in std_logic;
           Erreur        : out std_logic_vector((2*NmbrBits-1) downto 0)
          );
END COMPONENT;
--
COMPONENT TRUNCATOR_2nBits
    port(tranc_in: in std_logic_vector((4*NmbrBits)-1 downto 0);
         Reset_tranc: in std_logic;
         tranc_out: out std_logic_vector((2*NmbrBits-1) downto 0));
end COMPONENT ;

COMPONENT ADDER_2nBits
port ( A_MSim,B_MSim: in std_logic_vector((2*NmbrBits-1) downto 0);
          reset_A         : in std_logic;
          C_MSim          : out std_logic_vector((2*NmbrBits-1) downto 0)
         );
END COMPONENT;
      signal   Denum : std_logic;
      signal   Num1, Num2, int1, Int2,int3, int5,int6: std_logic_vector ((2*NmbrBits-1) downto 0);
      signal   int4 : std_logic_vector ((4*NmbrBits)-1 downto 0);

      Constant ZeroHuit : std_logic_vector ((2*NmbrBits-1) downto 0) := "011001000000000110100011"; --"011001000000000000000000";-- 01100110011001100 --0.78125
  --      constant ZeroSix : std_logic_vector ((NmbrBits-1) downto 0) := "010100000000000000000000"; --   010011001100100
begin


    process (Data,Bi,Bc,Bs)
    variable Num1_tmp, Num2_tmp : std_logic_vector ((2*NmbrBits-1) downto 0);
    variable Denum_tmp : std_logic;
     variable C1_tmp :std_logic_vector((2*NmbrBits)-2) downto 0) := (others => '0');
     variable C2_tmp :std_logic_vector((2*NmbrBits)-2) downto 0) := (others => '1');
     begin
if (Data > Bi) and (Data < Bs) then

    if (Data <= Bc) and (Data > Bi) then
        Num1  <= Data;
        Num2  <= Bi;

        if (Bi = '1' & C1_tmp) then
        Denum <= '1';
        else
        Denum <= '0';
        end if;
    elsif (Data <= Bs) and (Data > Bc) then
        Num1 <= Bc;
        Num2 <= Data;

        if (Bs = '0' & C2_tmp) then
        Denum <= '1';
        else
        Denum <= '0';
        end if;
    else
    Num1  <= (others => '0');
    Num2  <= (others => '0');
    Denum <='1';
    end if;
else
    Num1  <= (others => '0');
    Num2  <= (others => '0');
    Denum <='1';
end if;
 end process;
 Soustraction_1  : SUBSTRACTOR_2nBits port map (Sous1 => Num1, Sous2 => Num2, Reset_sous => Reset_Prct, Erreur => int2);
process (Denum, int2)
     variable C1_tmp :std_logic_vector((2*NmbrBits)-2) downto 0) := (others => '0');
     variable C2_tmp :std_logic_vector((2*NmbrBits)-2) downto 0) := (others => '1');
begin
if Denum = '1' then                         -- Pour éviter la division sur (Bc-Bi), ya deux cas
int3 <= int1;                           -- Le premier cas: le dénumérateur egale a 0.995, dans le cas ou
int5 <=  '0' & C2_tmp;
                                            -- la valeur infini est présente
else
                                          -- Le deuxieme: est plus fréquent, le dénumertateur egale a 0.005 (le quart de l'univers du discourt)
int2 <= int1(2*NmbrBits-1) & int1(2*NmbrBits-9) downto 0) & "0000000";      -- la on remplace la division par une multiplication par 200,
                                            -- et cette opération est remplacée par un décalage de 8 bits
int3 <= ZeroHuit;
end if;
end process;
Multiplication : MULTIPLIER_2nBits port map (D_MSim => int2, E_MSim => int3, Reset_M => Reset_Prct, F_MSim => int4);
Truncation     : TRUNCATOR_2nBits port map (tranc_in => int4, Reset_tranc => Reset_Prct, tranc_out => int5);
Addition       : ADDER_2nBits port map (A_MSim=>int5, B_MSim=>int6,  Reset_A =>Reset_Prct, C_MSim=> int6);
Pourcentage <= int6;
end Pourcentage_1;
```

D- Code utilisé pour réalisé la tables des lois

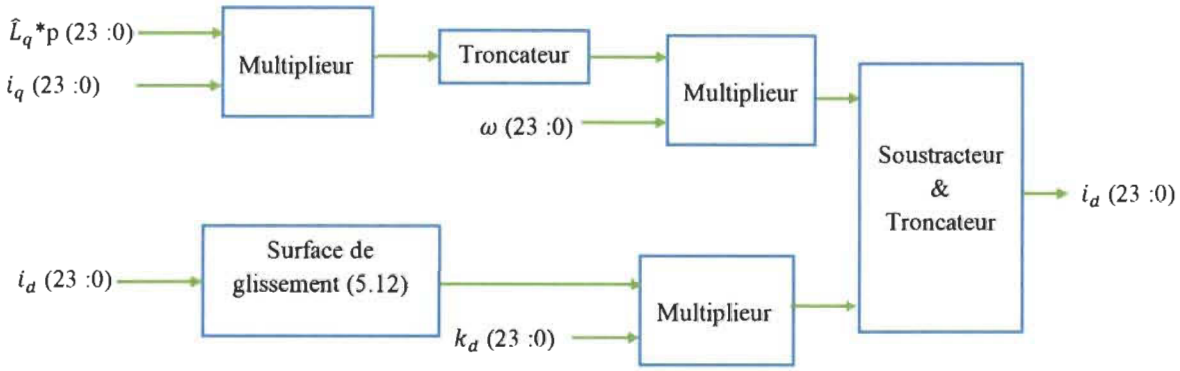
```
LIBRARY ieee;
use ieee.std_logic_1164.all;
ENTITY RulesTable is
Generic (NmbrBits: integer := 12);
PORT( P11, P12, P13, P14, P15,
      P21, P22, P23, P24, P25 : in std_logic_vector((2*NmbrBits-1) downto 0);
      Reset_CS : in std_logic;
      Npw1, Npw2, Npw3, Npw4,Npw5 : out std_logic_vector ((2*NmbrBits-1) downto 0)
    );
END RulesTable;
ARCHITECTURE Archi_RulesTable of RulesTable is
-- Composants du RulesTable--

Component MinMax is
    port ( Entree1, Entree2, Entree3   : in std_logic_vector ((2*NmbrBits-1) downto 0);
           Reset_MinMax       : in std_logic;
           Max           : out std_logic_vector ((2*NmbrBits-1) downto 0)
           );
 end Component ;


-- Signaux de liaison--
     signal   maxNB, maxNB1, maxNB2, maxNB3, maxNB4, maxNB5, maxNB6,
              maxNS, maxNS1, maxNS2, maxNS3, maxNS4, maxNS5, maxNS6,
              maxZ, maxZ1, maxZ2, maxZ3, maxZ4, maxZ5, maxZ6,
              maxPS, maxPS1, maxPS2, maxPS3, maxPS4, maxPS5, maxPS6,
              maxPB, maxPB1, maxPB2, maxPB3, maxPB4, maxPB5, maxPB6
              : std_logic_vector ((2*NmbrBits-1) downto 0);


  BEGIN

maxNB1 <= (others => '0');
maxNS1 <= (others => '0');
maxZ1  <= (others => '0');
maxPS1 <= (others => '0');
maxPB1 <= (others => '0');

--process (P11, P12, P13, P14, P15, P21, P22, P23, P24, P25)
--     variable Min_tmp, Max_tmp : std_logic_vector (15 downto 0);
--     begin


--===== La matrice de sortie ========--
--        P  21 22 23 24 25          --
--       11 |NB NB NB NS Z|          --
--       12 |NB NS NS Z PB|          --
--  Px = 13 |NB NS Z PS PB|          --
--       14 |NB Z PS PS PB|          --
--       15 |Z PS PB PB PB|          --
--                                   --
--=====================================--

----- Calcul de maxNB     ---
MinMax_NB1  :  MinMax port map ( Entree1 => P11, Entree2 => P21, Entree3 => maxNB1, Reset_MinMax => Reset_CS, Max => maxNB2);

MinMax_NB2  :  MinMax port map ( Entree1 => P11, Entree2 => P22, Entree3 => maxNB2, Reset_MinMax => Reset_CS, Max => maxNB3);

MinMax_NB3  :  MinMax port map ( Entree1 => P11, Entree2 => P23, Entree3 => maxNB3, Reset_MinMax => Reset_CS, Max => maxNB4);

MinMax_NB4  :  MinMax port map ( Entree1 => P12, Entree2 => P21, Entree3 => maxNB4, Reset_MinMax => Reset_CS, Max => maxNB5);

MinMax_NB5  :  MinMax port map ( Entree1 => P13, Entree2 => P21, Entree3 => maxNB5, Reset_MinMax => Reset_CS, Max => maxNB6);

MinMax_NB6  :  MinMax port map ( Entree1 => P14, Entree2 => P21, Entree3 => maxNB6, Reset_MinMax => Reset_CS, Max => maxNB);
----- Calcul de maxNS     ---
MinMax_NS1  :  MinMax port map ( Entree1 => P12, Entree2 => P24, Entree3 => maxNS1, Reset_MinMax => Reset_CS, Max => maxNS2);

MinMax_NS2  :  MinMax port map ( Entree1 => P12, Entree2 => P22, Entree3 => maxNS2, Reset_MinMax => Reset_CS, Max => maxNS3);

MinMax_NS3  :  MinMax port map ( Entree1 => P12, Entree2 => P23, Entree3 => maxNS3, Reset_MinMax => Reset_CS, Max => maxNS4);

MinMax_NS4  :  MinMax port map ( Entree1 => P13, Entree2 => P22, Entree3 => maxNS4, Reset_MinMax => Reset_CS, Max => maxNS);
---- Calcul de maxZ     ---
MinMax_Z1  :  MinMax port map ( Entree1 => P12, Entree2 => P25, Entree3 => maxZ1, Reset_MinMax => Reset_CS, Max => maxZ2);

MinMax_Z2  :  MinMax port map ( Entree1 => P12, Entree2 => P24, Entree3 => maxZ2, Reset_MinMax => Reset_CS, Max => maxZ3);

MinMax_Z3  :  MinMax port map ( Entree1 => P13, Entree2 => P23, Entree3 => maxZ3, Reset_MinMax => Reset_CS, Max => maxZ4);

MinMax_Z4  :  MinMax port map ( Entree1 => P14, Entree2 => P22, Entree3 => maxZ4, Reset_MinMax => Reset_CS, Max => maxZ5);

MinMax_Z5  :  MinMax port map ( Entree1 => P15, Entree2 => P21, Entree3 => maxZ5, Reset_MinMax => Reset_CS, Max => maxZ);
---- Calcul de maxPS     ---
MinMax_PS1  :  MinMax port map ( Entree1 => P13, Entree2 => P24, Entree3 => maxPS1, Reset_MinMax => Reset_CS, Max => maxPS2);

MinMax_PS2  :  MinMax port map ( Entree1 => P14, Entree2 => P23, Entree3 => maxPS2, Reset_MinMax => Reset_CS, Max => maxPS3);

MinMax_PS3  :  MinMax port map ( Entree1 => P14, Entree2 => P24, Entree3 => maxPS3, Reset_MinMax => Reset_CS, Max => maxPS4);

MinMax_PS4  :  MinMax port map ( Entree1 => P25, Entree2 => P22, Entree3 => maxPS4, Reset_MinMax => Reset_CS, Max => maxPS);
---- Calcul de maxPB     ---
MinMax_PB1  :  MinMax port map ( Entree1 => P12, Entree2 => P25, Entree3 => maxPB1, Reset_MinMax => Reset_CS, Max => maxPB2);
MinMax_PB2  :  MinMax port map ( Entree1 => P23, Entree2 => P25, Entree3 => maxPB2, Reset_MinMax => Reset_CS, Max => maxPB3);
MinMax_PB3  :  MinMax port map ( Entree1 => P14, Entree2 => P25, Entree3 => maxPB3, Reset_MinMax => Reset_CS, Max => maxPB4);
MinMax_PB4  :  MinMax port map ( Entree1 => P15, Entree2 => P23, Entree3 => maxPB4, Reset_MinMax => Reset_CS, Max => maxPB5);
MinMax_PB5  :  MinMax port map ( Entree1 => P25, Entree2 => P24, Entree3 => maxPB5, Reset_MinMax => Reset_CS, Max => maxPB6);
MinMax_PB6  :  MinMax port map ( Entree1 => P15, Entree2 => P25, Entree3 => maxPB6, Reset_MinMax => Reset_CS, Max => maxPB);
 ---------------------
Npw1 <= maxPB;
Npw2 <= maxNS;
Npw3 <= maxZ;
Npw4 <= maxPS;
Npw5 <= maxNB;
END Archi_RulesTable;
```

```
9   -- déclaration des bibliothèques
10  library ieee;
11  use ieee.std_logic_1164.all;
12  use ieee.std_logic_signed.all;  --pour faire l'addition
13
14  -- déclaration de l'entité
15  entity MinMax is
16  Generic (NmbrBits: integer := 12);
17      port ( Entree1, Entree2, Entree3   : in std_logic_vector ((2*NmbrBits-1) downto 0);
18             Reset_MinMax        : in std_logic;
19             Max                 : out std_logic_vector ((2*NmbrBits-1) downto 0)
20             );
21   end MinMax ;
22
23  -- déclaration de l'architecture
24  architecture Archi_MinMax of MinMax is
25
26  begin
27
28       process (Entree1, Entree2, Entree3)
29       variable Min_tmp, Max_tmp : std_logic_vector ((2*NmbrBits-1) downto 0);
30       begin
31  if Entree1 >= Entree2 then
32
33  Min_tmp := Entree2;
34
35  else
36
37  Min_tmp := Entree1;
38
39  end if;
40
41  if Entree3 >= Min_tmp then
42
43  Max_tmp := Entree3;
44
45  else
46
47  Max_tmp := Min_tmp;
48
49  end if;
50
51  Max <= Max_tmp;
52   end process;
53
```

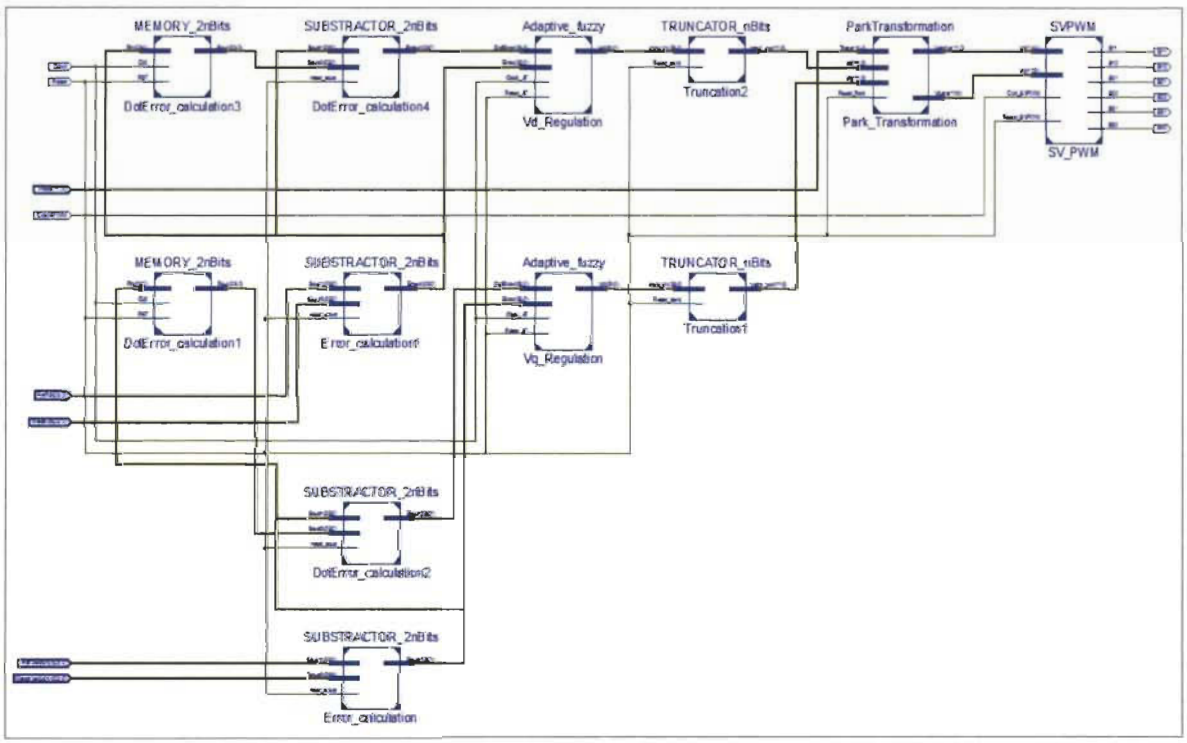
E- Schéma du circuit « *Deuxième loi de contrôle* » réalisé sur un FPGA.



F- Schéma du circuit du régulateur mode glissant réalisé sur un FPGA



G- Schéma du circuit « *Troisième loi de contrôle* » réalisé sur un FPGA.



Annexe C – Publications et Contributions

A- En tant que premier auteur

Boukaka, S.; Chaoui, H.; Sicard, P., "FPGA implementation of SVPWM," *New Circuits and Systems Conference (NEWCAS), 2013 IEEE 11th International* , vol., no., pp.1,4, 16-19 June 2013/NEWCAS.2013.

Boukaka, S.; Teiar, H.; Chaoui, H.; Sicard, P., "FPGA implementation of an adaptive fuzzy logic controller for PMSM," *Power Electronics, Machines and Drives (PEMD 2014), 7th IET International Conference on* , vol., no., pp.1,6, 8-10 April 2014.2014.

Boukaka, S.; Teiar, H.; Sicard, P., "FPGA implementation of a laws library of adaptive control for PMSM," DCIS, 2015.

Boukaka, S.; Massicotte, D.; Sicard, P., "FPGA Implementation of SVPWM Approximation," DCIS, 2015.

Boukaka, S.; Sicard, P., "Polynomial Approximation of SVPWM," Power Electronic Letter. Soumis.

B- En tant que deuxième auteur

Teiar, H.; Boukaka, S.; Chaoui, H.; Sicard, P., "Adaptive fuzzy logic control structure of PMSMs," *Industrial Electronics (ISIE), 2014 IEEE 23rd International Symposium on* , vol., no., pp.745,750, 1-4 June 2014.