UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

IMPLÉMENTATION SUR FPGA DE LA FORMULE DE SHERMAN-MORRISON À FAIBLE LATENCE POUR DU MATÉRIEL DANS LA BOUCLE

MÉMOIRE PRÉSENTÉ

COMME EXIGENCE PARTIELLE DE LA

MAÎTRISE EN GÉNIE ÉLECTRIQUE

PAR

SLIM ASSALI

OCTOBRE 2024

Université du Québec à Trois-Rivières

Service de la bibliothèque

Avertissement

L'auteur de ce mémoire, de cette thèse ou de cet essai a autorisé l'Université du Québec à Trois-Rivières à diffuser, à des fins non lucratives, une copie de son mémoire, de sa thèse ou de son essai.

Cette diffusion n'entraîne pas une renonciation de la part de l'auteur à ses droits de propriété intellectuelle, incluant le droit d'auteur, sur ce mémoire, cette thèse ou cet essai. Notamment, la reproduction ou la publication de la totalité ou d'une partie importante de ce mémoire, de cette thèse et de son essai requiert son autorisation.

UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES
MAÎTRISE EN GÉNIE ÉLECTRIQUE (M. Sc. A.)

Direction de recherche :

Daniel Massicotte
Prénom et nom directeur de recherche

Jury d'évaluation

Daniel Massicotte
Prénom et nom UQTR

François Nougarou
Prénom et nom UQTR

Alexandre Robichaud
Prénom et nom UQAT (Membre externe)

REMERCIEMENTS

C’est avec un grand plaisir que j’aimerais réserver mes modestes phrases en signe de gratitude et de profonde reconnaissance pour adresser mes vifs remerciements aux personnes qui m’ont permis de réaliser ce projet. Je tiens à manifester l’expression de mes respects les plus distingués à monsieur le professeur Daniel MASSICOTTE, mon directeur de recherche pour avoir accepté de m’aiguiller patiemment et pour la qualité de l’encadrement qu’il m’a fourni ainsi que son aide, ses précieux conseils, la disponibilité et la sympathie.

Ma profonde reconnaissance à tous les professeurs de l’Université du Québec à Trois-Rivières pour la formation d’excellence qu’ils m’ont offerte, pour leurs conseils et leurs contributions à entamer ma carrière avec des bases solides. Mes remerciements s’adressent à toute l’équipe du Laboratoire des signaux et systèmes intégrés (LSSI) pour leurs idées et leurs conseils, ainsi que pour leur aide précieuse. Ainsi que la Chaire de recherche sur les signaux et l’intelligence des systèmes haute performance et ses partenaires industriels, plus particulièrement OPAL-RT et Hydro-Québec, pour leur support financier de mes travaux de recherche et la CMC pour les outils et les équipements de recherche pour l’implémentation sur FPGA Qu’ils trouvent ici ma grande reconnaissance et mon sincère respect.

RÉSUMÉ

Le domaine des simulations en temps réel joue un rôle crucial dans les recherches académiques et industrielles, en particulier pour la conception et le test des convertisseurs en électronique de puissance, ainsi que pour les réseaux électriques. Parmi les approches les plus prometteuses, le matériel dans la boucle (*Hardware-in-the-Loop, HIL*) se distingue, permettant des simulations en temps réel grâce à la résolution discrète d'équations en temps réel. Les principaux défis associés à la conception de systèmes HIL résidents dans la recherche des meilleurs compromis entre la latence des calculs, qui définissent le temps discret minimal, la précision des résultats, et les ressources matérielles requises.

Dans le cadre de ce projet, nous explorons l'utilisation de FPGA (*Field Programmable Gate Arrays*) pour le développement de systèmes HIL à très faible latence. L'objectif principal de cette mémoire est de concevoir et d'implémenter sur FPGA des algorithmes à faible latence adaptés aux contraintes du temps réel. Ces algorithmes, issus de travaux de recherche antérieurs, ont nécessité une adaptation complexe pour une mise en œuvre efficace sur FPGA.

Par ailleurs, cette étude intègre l'application de la formule de Sherman-Morrison (FSM) pour la simulation de deux types d'onduleurs triphasés : les onduleurs à deux niveaux (2-L) et le Back-to-Back (B2B). De plus, une comparaison approfondie est effectuée entre deux types d'arithmétique, à savoir la virgule fixe et la virgule flottante, afin d'évaluer leur impact respectif sur les performances et la précision des calculs dans un contexte de simulation en temps réel. Nous cherchons à minimiser la latence des calculs et garantissant une précision conforme aux exigences des applications, en utilisant une représentation en virgule fixe.

Mots-clés : Circuit électrique, FPGA, virgule fixe, Onduleur triphasé, Simulation temps réel, Formule de Sherman-Morrison.

TABLE DES MATIÈRES

Liste des tableaux viii

Liste des figures x

Liste des abréviations xii

Chapitre 1 - Introduction 1

1.1 Contexte et motivation 1

1.2 Problématiques 3

1.3 Les objectifs 6

1.4 Méthodologie 7

Chapitre 2 - État de l'art 9

2.1 Calcule en virgule fixe 9

2.1.1 Codage des données en virgule fixe 10

2.1.2 Comparaison arithmétiques virgule flottante et virgule fixe 12

2.1.3 Comparaison du RSBQ des deux codages 12

2.2 Field Programmable Gate Array (FPGA) 13

2.2.1 Constitution interne 14

2.2.2 Apport des FPGA dans la simulation numérique temps réel et implémentation 15

2.3 Conception du HIL 16

2.3.1 Architecture de simulateur en temps réel basé sur FPGA 16

2.3.2 Test HIL en temps réel avec accélération FPGA 18

2.4 Calcul matriciel 19

2.4.1 Architecture de multiplication matricielle 19

2.4.2 Architecture d'inversion matricielle méthode de Gauss Jordan 21

2.4.3 Inversion matricielle selon la formule de Sherman-Morrison 22

2.5 Revue de littératures 23

2.5.1 Exploration des méthodes d'inversion matricielle en virgule fixe sur FPGA 25

2.6 Conclusion 26

Chapitre 3 - FSM sur FPGA – Précision 27

3.1 Introduction 27

3.2 FSM et dynamique des calculs 29

3.2.1 Revue de l'algorithme de FSM 29

3.2.2 Reformulation de la FSM 31

3.2.3 Dynamique des calculs 32

3.2.4 Application de la mise à l'échelle (Scaling) 34

3.3 Calcul en virgule fixe 35

3.3.1 Méthode de conversion 35

3.3.2 FSM en virgule fixe-performance 37

3.3.3 FSM choit du facteur d'échelle et de la configuration binaire......... **Erreur ! Signet non défini.**

3.4 Résultats de synthèse de la FSM 55

3.5 Conclusion 61

Chapitre 4 - FSM sur FPGA – implémentation 63

4.1 Introduction 63

4.2 Implémentation HLS 64

4.2.1 Tests préliminaires d'implémentations 65

4.2.2 Implémentation de FSM 70

4.3 Implémentation SysGen 73

4.3.1 Théorie derrière la conception du processeur élémentaire 73

4.3.2 Ordonnancement 78

4.3.1 Architecture 82

4.3.2 Résultats et discussions 88

4.4 Conclusion 103

Conclusion 105

Références 107

Liste des tableaux

Tableau 2-1 Comparaison du RSBQ. 13

Tableau 2-2 Exploration des méthodes d'inversion matricielle 24

Tableau 3-1 Erreur selon (3-8) et (3-9) en fonction de la configuration binaire et le facteur d’échelle pour le 2-L pour Wl fixe de 90 bits. 45

Tableau 3-2 Erreurs selon (3-8) et (3-9) en fonction de la configuration binaire et le facteur d’échelle pour le B2B pour Wl fixe de 90 bits. 50

Tableau 4-1 Test additionneur virgule fixe en HLS sur le Kintex-7 de Xilinx. 66

Tableau 4-2 Test d’multiplieur en virgule fixe en VHDL et en HLS sur le Kintex-7. 67

Tableau 4-3 Test produit vectoriel de dimension 16 en HLS sur le Kintex-7 de Xilinx. 68

Tableau 4-4 Test Produit matrice-vecteur de dimension 16 en HLS sur le Kintex-7. 70

Tableau 4-5 Implémentation HLS de FSM sur le Kintex-7 de Xilinx. 72

Tableau 4-6 Résumés des délais en virgule fixe 200MHz en SysGen sur le Kintex-7 de Xilinx. 89

Tableau 4-7 E/L mémoire de la 1ère itération SM en virgule fixe 200 MHz en SysGen sur le Kintex-7. 89

Tableau 4-8 Calcul de la 1-ére itération en virgule fixe FSM à 200 MHz en SysGen sur le Kintex -7. 90

Tableau 4-9 Rétroaction de la 1-ère itération en virgule fixe FSM 200 MHz en SysGen sur le Kintex-7. 90

Tableau 4-10 E/L mémoire du 2-ème itération FSM en virgule fixe 200 MHz en SysGen sur le Kintex-7. 91

Tableau 4-11 Calcul du 2-éme itération en virgule fixe FSM 200 MHz en SysGen sur le Kintex-7. 91

Tableau 4-12 Rétroaction de la 2-éme itération en virgule fixe FSM 200 MHz en SysGen sur le Kintex-7. 92

Tableau 4-13 Résumés des délais virgule flottante 200MHz en SysGen sur le Kintex -7. 92

Tableau 4-14 E/L mémoire de la 1 ère itération FSM en virgule flottante 200 MHz en SysGen sur le Kintex-7. 93

Tableau 4-15 Calcul de la 1ère itération en virgule flottante de FSM 200 MHz en SysGen sur le Kintex-7. 93

Tableau 4-16 Rétroaction de la 1 ère itération en virgule flottante FSM 200 MHz en SysGen sur le Kintex-7. 93

Tableau 4-17 E/L mémoire de la 2-éme itération en virgule flottante SM 200 MHz en SysGen sur le Kintex-7. 94

Tableau 4-18 Calcul de la 2-éme itération en virgule flottante de FSM 200 MHz en SysGen sur le Kintex-7. 95

Tableau 4-19 Rétroaction de la 2-éme itération en virgule flottante de FSM 200 MHz en SysGen sur le Kintex-7. 95

Tableau 4-20 Résumés des délais en virgule fixe 400MHz en SysGen sur le Kintex -7.....96

Tableau 4-21 Résumés des délais virgule flottante 400MHz en SysGen sur le Kintex-7. ..96

Tableau 4-22 Temps d'exécution de FSM sur SysGen en virgule fixe de 32 bits et virgule flottante simple à 400 MHz (T = 2.5 ns)...98

Tableau 4-23 Synthèses des temps de calcul de FSM sur SysGen et HLS en virgule fixe 32 bits et flottante sur FPGA Kintex-7. 102

Tableau 4-24 Synthèse des ressources de FSM sur SysGen et HLS en virgule fixe 32 bits et flottante sur FPGA Kintex-7..103

Liste des figures

Figure 1-1 Environnements et méthodologie du projet..6

Figure 2-1 Représente la configuration de codage des données en virgule fixe [5]. 11

Figure 2-2 Configuration en virgule fixe désirée [5]. 11

Figure 2-3 Structure interne du SoC FPGA: Virtex-7 XC7VX485T [10]. 14

Figure 2-4 Schémas fonctionnels d'un simulateur HIL [13] 17

Figure 2-5 Propositions de multiplication matricielle – Architecture 1 [23].......................20

Figure 2-6 Proposition de multiplication matricielle – Architecture 2 [23] 20

Figure 2-7 Gauss Jordan Method [23] 21

Figure 3-1 Onduleur à 2 niveaux. 28

Figure 3-2 Onduleur *Back-to-Back* (B2B). 28

Figure 3-3 Distribution de l'inverse de la matrice **A** (16×16) pour le 2 niveaux. 33

Figure 3-4 Distribution de l'inverse de la matrice **A** (26×26) pour le B2B. 33

Figure 3-5 Erreur absolue selon (3-4) pour le convertisseur 2-L avec un mot binaire de 72 bits et différentes parties entières de 16, 18, 20 et 32 bits. L'erreur maximum de (3-4) en virgule flottante 32 bits est de 1×10^{-7}. 39

Figure 3-6 Erreur absolue selon (3-4) pour le convertisseur 2-L pour les configurations différentes (wl, wlInt): (36, 18), (54,18), (72, 18), (90,18) et (108, 18). L'erreur max de (3-4) en virgule flottante 32 bits est de 1×10^{-7}. 40

Figure 3-7 Erreur absolue selon (3-4) pour le convertisseur B2B avec un mot binaire de 77 bits et différentes parties entières de 12, 18, 24 et 32 bits. L'erreur max de (3-4) en virgule flottante 32 bits est de 1×10^{-5}..42

Figure 3-8 Erreur absolue selon (3-4) pour le convertisseur B2B pour une partie entière de 18 bits et un mot binaire multiple de 18. L'erreur max de (3-4) en virgule flottante 32 bits est de 1×10^{-5}. 43

Figure 3-9 Courbe 3D représentant l'erreur maximale en fonction du facteur d'échelle et le Wint pour le 2-L selon le tableau 3-1 pour Wl=90 bits. 47

Figure 3-10 Erreur en fonction de la longueur de la partie entière et le facteur d'échelle pour le 2-L selon le tableau 3-1 pour Wl=90 bits...48

Figure 3-11 Erreur en fonction du facteur d'échelle et de la longueur de la partie entière et pour le 2-L selon le tableau 3-1 pour Wl=90 bits. 49

Figure 3-12 Erreur en fonction du facteur d'échelle et de la longueur de la partie entière pour le 2L pour Wint=12 bits 49

Figure 3-13 Courbe 3D représentant l'erreur maximale en fonction du facteur d'échelle et le Wint pour le B2B selon le tableau 3-2 pour Wl=90 bits. 52

Figure 3-14 Erreur en fonction de la longueur de la partie entière et le facteur d'échelle pour le B2B selon le tableau 3-2 pour Wl=90 bits. 53

Figure 3-15 Erreur en fonction du facteur d'échelle et de la longueur de la partie entière pour le B2B selon le tableau 3-2 pour Wl=90 bits 53

Figure 3-16 Erreur en fonction du facteur d'échelle et de la longueur de la partie entière pour le B2B pour Wint=12 bits 54

Figure 3-17 Résultats de SM pour le 2-L pour les configurations différentes (Wl, Wint): (36,18), (54,18), (72, 18), (90, 18) et (108, 18) bits. 56

Figure 3-18 Résultats de SM pour le B2B pour les configurations différentes (Wl, Wint): (36, 18), (54,18), (72, 18), (90, 18) et (108, 18) bits. 57

Figure 4-1 Arborescence composée d’unité de multiplication et d’addition 69

Figure 4-2 Processeur élémentaire. 76

Figure 4-3 Design et Architecture désirée. 77

Figure 4-4 Design et description de la méthode 78

Figure 4-5 Organigramme général de FSM. 79

Figure 4-6 Organigramme de chargement de sigma. 80

Figure 4-7 Organigramme de calcul de la partie mis à jour. 82

Figure 4-8 Partie Sigma. 83

Figure 4-9 Partie contrôleur. 84

Figure 4-10 Gestion des BRAM. 85

Figure 4-11 L’organigramme du calcul de la mise à jour de la matrice inverse. 86

Figure 4-12 Schéma de visualisation du processus de décomposition. 88

Figure 4-13 Séquencements des itérations de FSM pour l’implémentation en Fixe. 100

Liste des abréviations

A/N	Analogique/Numérique.
ADC	Analog/Digital Converter.
ASIC	Application Specific Integrated Circuit
B2B	Back-to-Back
CPU	Central Processing Unit.
DSP	Digital Signal Processing
DSP48	48-bit Digital Signal Processing
DP16	Dot Product 16
FF	Flip-flop
FPGA	Field Programmable Gate Array
FSM	Formule de Sherman-Morrison
HIL	Hardware-in-the-Loop
HLS	High-Level Synthesis
LSSI	Laboratory of Signal and System Integration
LUT	Look Up Table
VHDL	Very High Description Language
VLSI	Very Large-Scale Integration
MCU	Microcontroller Unit.
RAM	Random Access Memory
RSBQ	Rapport Signal à Bruit de Quantification
SysGen	System Generator
2-L	Two Leve

Chapitre 1 - Introduction

Ce chapitre passe en revue les principales notions et contextes des travaux de recherches en relation avec le concept de la simulation avec du matériel dans la boucle (HIL – *Hardware-in-the-Loop*). Il s'agit d'une technique utilisée dans le test et le développement des systèmes électromécaniques complexes en temps réel. Il est bien évidemment nécessaire de présenter le contexte pour ensuite positionner la problématique du projet de recherche.

1.1 Contexte et motivation

Le HIL est de plus en plus présent dans différents domaines de la recherche et de l'industrie, notamment, les réseaux électriques, les véhicules électriques. Son contexte d'intérêt principal est la simulation à faible latence de calcul avec une grande précision. Un cas qui présente ce défi est celui des convertisseurs de puissance. Les convertisseurs de puissance et leurs contrôleurs sont devenus de plus en plus complexes, nécessitant un besoin de trouver des moyens rapides, fiables, économiques et précis pour tester plusieurs scénarios critiques en temps réel du circuit dans son application. De plus, la complexité grandissante de ces convertisseurs implique également des équations plus complexes afin de pouvoir les simuler. Celles-ci forment une matrice dont l'inverse peut être précalculé à partir de la résolution continue d'un ensemble d'équations en temps réel. Les convertisseurs de puissance modernes doivent gérer des densités de puissance plus élevées et des fréquences de commutation plus élevées. Ce projet propose une technique d'inversion matricielle basée

sur la FSM (Formule de Sherman-Morrison), qui permet d'atteindre le temps d'exécution optimal pour la résolution d'un tel système en temps réel. Dans ce contexte, une méthode d'approximation des inverses de matrices basée sur l'implémentation de la FSM est envisagée. L'implémentation vise une optimisation en virgule fixe afin de minimiser la latence de calcul. La FSM permet de réduire la mémoire utilisée lors du calcul de l'inverse comparativement au stockage de toutes les inverses. L'objectif est d'optimiser le temps de calcul des architectures matérielles dédiées aux simulations en temps réel, tout en préservant une précision conforme aux exigences. Cependant, la mise en œuvre de ces algorithmes sur FPGA (*Field Programmable Gate Array*) n'est pas toujours évidente. La performance des implémentations FPGA dépend de plusieurs facteurs, dont la complexité de l'algorithme, la précision du calcul voulue, la structure de l'architecture matérielle et les contraintes temporelles.

La mise en œuvre de systèmes DSP (*Digital Signal Processing*, soit le traitement numérique des signaux) sur FPGA est un défi en raison de la complexité de l'algorithme, de la précision de calcul requise et des contraintes temporelles. Les implémentations FPGA sont limitées par des contraintes matérielles, comme la mémoire intégrée, le nombre de blocs DSP, LUTs et FFs. Ce travail de recherche se concentrera sur l'implémentation sur FPGA d'algorithmes à virgule fixe afin d'atteindre une faible latence où l'objectif est de minimiser cette latence tout en assurant la précision de calcul nécessaire pour les applications. Nous allons explorer des méthodes pour améliorer les performances de l'implémentation FPGA, comme la parallélisation et la virgule fixe, afin de réduire sa complexité matérielle. Cela permet de définir les enjeux fondamentaux qui orientent et structurent nos travaux de recherche.

1.2 Problématiques

La problématique qu'on peut déduire est que l'implémentation d'algorithmes à faible latence sur des FPGA à temps réel est un défi technique complexe. Nos travaux sont basés sur la FSM qui permet de réduire l'intensité du calcul de l'inversion de matrices en temps réel pour les convertisseurs de puissance sur FPGA. La FSM permet de calculer l'inverse d'une matrice par une série d'opérations plus économiques. Elle permet de recalculer seulement une partie de l'inverse à partir d'un inverse connu réduisant ainsi sa complexité de calcul. Cela évite le stockage complet de toutes les matrices inverses en mémoire, seules quelques matrices clés seront sélectionnées pour être enregistrées en mémoire et serviront de bases pour recalculer les autres inverses. Ceci permet de réduire drastiquement le stockage mémoire nécessaire étant particulièrement bénéfique lorsque la mémoire est limitée. La formule permet donc d'effectuer une mise à jour itérative de l'inverse de la matrice en fonction des perturbations sur la matrice à inverser par rapport à une matrice connue. Le convertisseur de puissance est modélisé mathématiquement, puis un modèle discret est implémenté sur un appareil numérique, le FPGA, qui effectue une simulation du convertisseur de puissance. Cependant, cette implémentation doit être spécifiée en virgule fixe sous contraintes d'optimisation du temps d'exécution du code en prenant en considération l'architecture hardware donc la problématique de ce mémoire de recherche est de proposer une implémentation sur FPGA pour minimiser la latence des calculs dans une implémentation. Pour ce faire, l'arithmétique à virgule fixe sera appliquée tout en assurant d'atteindre une bonne précision de calcul. Mes travaux s'insèrent et poursuivent ceux de Jérémy Poupart et Michel Lemaire [1-3], notamment en suivant les optimisations proposées par Jérémy Poupart pour la simulation en temps réel sur FPGA d'un onduleur triphasé, ainsi

comparé à la méthodologie et l'architecture générique développées par Michel Lemaire pour la simulation temps réel d'électronique de puissance sur une plateforme hétérogène.

La FSM et les techniques d'inversion matricielle classiques (décomposition LU, QR, ou l'algorithme de Kant & Kimura [24-25] [27]) présentent des caractéristiques fondamentales différentes, rendant leur comparaison difficile dans un contexte d'implémentation matérielle sur FPGA.

La FSM est spécialement conçue pour résoudre le problème de la mise à jour de l'inverse d'une matrice déjà connue et servant de référence. Cette méthode repose sur des opérations simples, principalement des vecteurs et scalaires plutôt que des matrices complètes, ce qui réduit significativement les opérations arithmétiques et les besoins en ressources matérielles lors de son implémentation sur FPGA. Contrairement aux autres méthodes, la FSM calcule l'inverse à chaque pas de calcul, éliminant ainsi le besoin de précalculer hors ligne toutes les matrices inverses possibles d'un circuit. Cela rend la FSM particulièrement adaptée aux systèmes temps réel implémentés sur FPGA, où elle permet la simulation de circuits avec une faible quantité de mémoire, mais avec des pas de calcul dans les centaines de nanosecondes.

En revanche, les méthodes telles que la décomposition LU, QR ou l'algorithme de Kant & Kimura sont des approches générales d'inversion matricielle. Ces méthodes nécessitent impérativement des étapes de factorisation et de résolution de systèmes triangulaires, ce qui mobilise des blocs DSP et des BRAM supplémentaires pour les multiplications matricielles et le stockage des matrices intermédiaires. Bien qu'elles offrent un faible pas de calcul, elles imposent une utilisation massive de mémoire pour stocker toutes les inverses précalculées, rendant leur implémentation impraticable sur FPGA. Par ailleurs, ces méthodes présentent

une latence beaucoup plus élevée, incomparablement supérieure à celle de la méthode de Sherman-Morrison (quelques centaines de microsecondes).

L’arithmétique à virgule fixe permet d’effectuer des calculs mathématiques dans une logique combinatoire plus simples et plus rapides que pour les nombres à virgule flottante. Elle permet également d’utiliser des structures de données plus compactes, ce qui réduit les besoins en mémoire et accélère les opérations de stockage et d’accès aux données. Cela peut entraîner une réduction de la latence lors de l’application de la FSM à une matrice de grande taille. En effet, le traitement des données sera plus rapide et plus efficace. Les questions suivantes seront abordées pour minimiser la latence des calculs:

- Comment choisir l’arithmétique entière ou flottante appropriée pour garantir la précision des résultats tout en minimisant la latence?
- Comment concevoir l'architecture matérielle pour tirer parti des capacités de parallélisation de l’architecture sur FPGA tout en respectant les ressources disponibles?
- Quels sont les impacts sur la performance et le temps de développement dont les outils d’implémentation sur FPGA?

Le développement d’une méthodologie efficace pour l’implémentation sur FPGA d’algorithmes à faible latence passe par la résolution de ces questions. Il faut donc s’assurer d’un calcul en virgule fixe pour optimiser la largeur des données et ensuite procéder au mapping de l’algorithme sur FPGA afin de minimiser la surface du circuit. Pour la validation, il est nécessaire d’obtenir des résultats qui démontrent la performance de la méthode. Dans ce contexte, la question qui se pose est comment utiliser les propriétés de nos savoir-faire pour assurer le fonctionnement désiré. Ce dernier présente un grand défi : celui de l’émergence d’une telle implémentation composée de plusieurs entités hétérogènes en

interaction mutuelle. De plus, le nombre et la distribution des variables impliquées dans le calcul matriciel et l'optimisation de cette implémentation s'avère donc très coûteuse en mémoire et en temps tel que détaillé dans la figure 1-1

La figure 1-1 représente l'interface entre les différentes parties de l'implémentation et la méthodologie de validation des résultats, qui est basée sur les contraintes imposées par l'environnement matériel FPGA de Xilinx. De nombreuses études ont été faites à ce sujet ; on en discutera plus en détail au chapitre 2 de la revue de littérature.

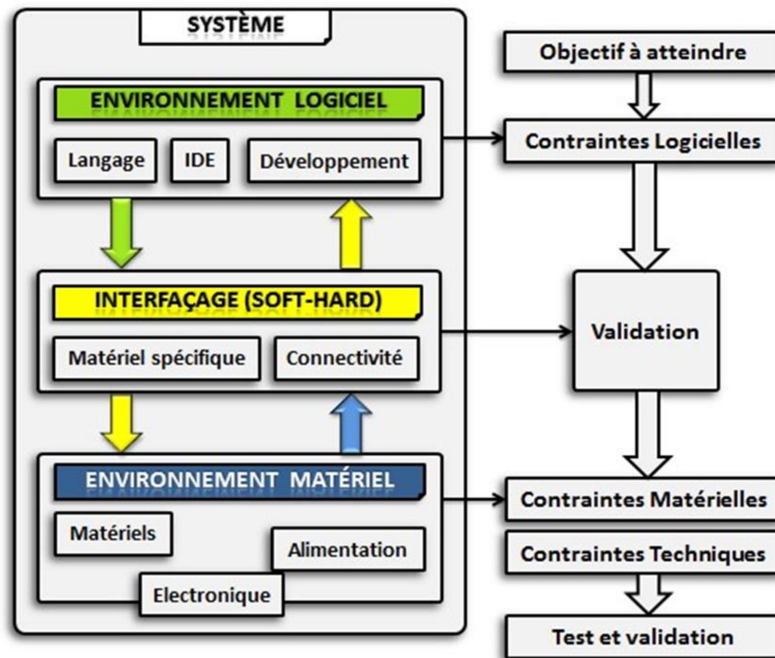

Figure 1-1 Environnements et méthodologie du projet.

1.3 Les objectifs

L'objectif de ce mémoire de recherche est de réaliser l'implémentation de la FSM sur FPGA pour la résolution d'onduleurs triphasé en temps réel. L'implémentation doit répondre aux exigences des applications en termes de latence, ressource et précision des calculs. Ainsi,

deux outils d'implémentations de Xilinx seront comparés (Vivado-HLS et System Generator for DSP (Sysgen)) ainsi que deux arithmétiques, virgule fixe et flottante.

Les objectifs spécifiques de ce mémoire de recherche sont :

- Évaluer les performances de la FSM en termes de précision de calcul en virgule fixe et flottante.
- Réaliser l'implémentation sur FPGA de la FSM par l'utilisation de deux outils : langage de haut niveau (HLS) en exploitant les instructions de parallélisme dans la conception architectural exploitant manuellement le parallélisme avec SysGen.
- Comparer les performances en termes de temps d'exécution, de ressources et de précision et des calculs pour deux cas d'applications d'onduleurs triphasé : le deux niveau (2-L) et le *Back-to-Back* (B2B).

1.4 Méthodologie

Pour atteindre l'objectif principal et répondre aux besoins des sous-objectifs mentionnés dans la section précédente, la méthodologie de recherche s'inscrit dans une analyse et une décomposition du projet en plusieurs étapes. Chacune étant réalisée de manière itérative et incrémentale étant donné que chacune des étapes de cette méthodologie peut être adaptée à nos besoins spécifiques et aux résultats préliminaires de chaque phase du projet.

Étude bibliographique : Réalisation d'une recherche de la littérature spécifique aux sous-objectifs du projet. Le chapitre 2 résume le résultat de cette étape.

Étudier la FSM selon l'arithmétique de calcul : Il est important de comprendre l'algorithme à implémenter. À savoir, les conditions dans lesquelles la FSM s'applique pour réduire la quantité mémoire, comment la virgule fixe et flottante impactent sur la précision,

les ressources et le temps d’exécution et finalement comment la FSM peut être implémentée efficacement sur une technologie FPGA cible?

Modélisation et validation fonctionnelle: Pour atteindre l'objectif d'implémenter une FSM sur FPGA en utilisant HLS et SysGen, il faut d'abord modéliser la FSM en deux versions. Avec HLS, la FSM est codée en exploitant le parallélisme automatique à travers des directives spécifiques. Avec SysGen, la conception est réalisée manuellement où l'architecte doit structurer et paralléliser les blocs fonctionnels pour maximiser l'efficacité. Une fois les deux modèles prêts, des simulations sont effectuées pour valider leur fonctionnement.

Synthèse et évaluation des performances : Après validation, chaque modèle est synthétisé et implémenté sur FPGA. Ensuite, les performances sont comparées en termes de temps d'exécution et l'utilisation des ressources FPGA, pour les deux cas d'onduleurs (2-L et B2B). Cette analyse permet de déterminer quelle approche offre le meilleur compromis pour chaque application. Envisager des optimisations supplémentaires (ajustement des directives HLS, optimisation manuelle de la conception SysGen, etc.).

Le projet s’inscrit dans un grand projet de recherche réalisé à la Chaire de recherche sur les signaux et l’intelligence des systèmes haute performance, ainsi qu’au Laboratoire de Signaux et Systèmes intégrés (LSSI) de l’UQTR. Le choix de la méthode de gestion de projet dépendra des préférences de l’équipe de recherche ainsi que des caractéristiques spécifiques du projet. Toutefois, dans le cadre de ce projet qui implique à la fois des tâches de développement matériel et logiciel, nous pouvons suggérer l’utilisation d’une méthode hybride combinant des éléments de la méthodologie en V et du développement agile, comme Scrum, une approche itérative et incrémentale serait toutefois plus appropriée.

Chapitre 2 - État de l'art

Ce chapitre est consacré à réaliser une recherche bibliographique pour maîtriser les trois concepts clés de ce mémoire : HIL, calculs en virgule fixe sur FPGA et la FSM. En profitant des dernières avancées technologiques, nous allons pouvoir résoudre la problématique soulevée précédemment. Le concept du HIL bénéficie d'une vaste littérature à ce sujet.

2.1 Calcule en virgule fixe

Une bonne majorité des implémentations VLSI utilisent l'arithmétique de calcul en virgule fixe (*fixed-point* en anglais) pour réduire la surface et la consommation d'énergie et ainsi obtenir un matériel rentable. Cela entraîne toutefois une possibilité de détérioration ou de dégradation de la précision des calculs en raison du nombre limité de bits utilisés dans la représentation des données. Il est évident que le codage en virgule fixe introduit des bruits de quantification lors de la réduction de la longueur binaire des données, par exemple lors des opérations fortement non linéaires. De plus, il pose un grand problème de débordement à chaque fois que la longueur du mot de code de la partie entière est insuffisante pour représenter la dynamique entière des variables de calcul.

Cette section vise à présenter et décrire en détail les spécifications de l'arithmétique à virgule fixe, ainsi que les différentes méthodes permettant d'évaluer sa précision. Elle explique également comment distinguer les arithmétiques, virgule fixe et virgule flottante, qui constituent les arithmétiques binaires de données les plus couramment utilisées dans les cartes et plates-formes embarquées modernes. Les arithmétiques à virgule fixe existent

depuis les premiers calculateurs électroniques et ordinateurs. L'objectif est de choisir le format adéquat pour cette arithmétique, tout en tenant compte de la puissance de calcul des processeurs et de la mémoire nécessaire en raison de la latence et du coût d'implantation des algorithmes complexes basés sur des tables précalculées (LUT – *Look-Up-Table*).

Aussi, l'utilisation de la virgule fixe apporte une réduction de la surface de silicium, du temps d'exécution et de la consommation totale d'énergie de la carte électronique par rapport à l'arithmétique flottante. Par contre, l'utilisation de la virgule fixe entraîne la création de bruits de quantification causés par une longueur binaire insuffisante. Ces bruits nuisent à la qualité des calculs dans l'ensemble de l'application. Ainsi, avant d'entamer l'application en virgule fixe l'étape primordiale est l'évaluation de la précision de la conversion de format flottant au format fixe.

2.1.1 Codage des données en virgule fixe

Dans ce paragraphe de codage, nous allons décrire les spécifications de notre arithmétique à virgule fixe ainsi que les différents choix de codage pour évaluer la précision du calcul. Bien entendu, la mise en œuvre d'un algorithme sur un FPGA nécessite l'utilisation de la virgule fixe, car elle présente des avantages en termes de coûts et de latence de calcul [4]. Par conséquent, les applications conçues en flottant simples ou doubles doivent être converties en virgule fixe. Cette opération de conversion est un processus délicat, parmi les étapes de base, on compte l'évaluation de la précision du calcul en arithmétique entière ou à virgule fixe [5].

Cette étape du changement de format des données de calcul implique souvent une perte de bits, ce qui entraîne un bruit de quantification systématique non négligeable. Par la suite, la

position de la virgule peut alors être repérée par un entier *n* relatif au bit de poids le plus faible (LSB – *Least Significant Bit*) ou par un entier *m* relatif au bit de poids fort (MSB – *Most Significant Bit*) [5]. Dans ce cas, on peut représenter le nombre qu'on veut convertir en virgule fixe selon la figure 2-1.

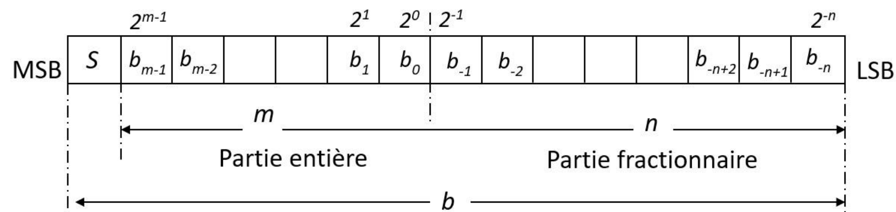

Figure 2-1 Représente la configuration de codage des données en virgule fixe [5].

Si le nombre a un bit de signe, il aura *m* bits pour sa partie entière et *n* bits pour sa partie fractionnaire. Le nombre de chiffres utilisés est noté $b = m + n + 1$.

$$x = -2^m S + \sum_{i=-n}^{m-1} b_i 2^i \qquad (2\text{-}1)$$

La figure 2-2 représente le format des données en virgule fixe désirée, qui est entièrement défini par la représentation choisie et la largeur de ses parties entières et fractionnaires. L'objectif est d'avoir une seule unité dans la partie entière pour le bit de signe (m=0), et tout le reste des bits pour la partie fractionnelle afin d'obtenir la plus grande précision possible.

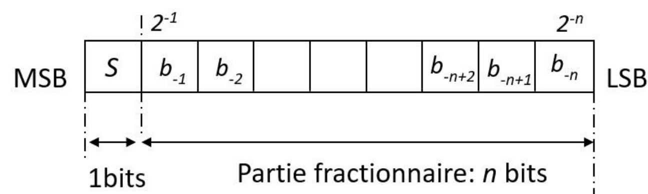

Figure 2-2 Configuration en virgule fixe désirée [5].

Étant donné que cette notation de l'équation (2-1) en virgule fixe, qui est simple et qui d'exécuter des calculs rapidement, présente toutefois un taux d'erreur. Ce taux se mesure à l'aide du rapport signal sur bruit quantifié (RSBQ, SQNR – *Signal-to-Quantization Noise Ratio*) souvent exprimé en décibel (dB) dans le domaine du traitement numérique du signal.

2.1.2 Comparaison arithmétiques virgule flottante et virgule fixe

Ce paragraphe compare les systèmes de numérotation en virgule flottante et en virgule fixe afin d'en mettre en évidence les avantages et les inconvénients. Cette comparaison est fondée sur l'analyse de deux volets réalisés dans [7] : le premier concerne l'aspect arithmétique, tandis que la seconde partie sur l'implantation hardware ou matérielle. C'est ce que vise ce projet en mettant en évidence la qualité du codage et le dynamisme de chacun des deux codes. Nous allons aussi analyser le rapport signal sur bruit de quantification. Ce sujet est abordé dans la prochaine section.

2.1.3 Comparaison du RSBQ des deux codages

Il est à noter que, dans la plupart des applications de traitement numérique du signal, la précision des calculs est mesurée le plus souvent par le critère de RSBQ. La précision des calculs est généralement déterminée par l'exigence du RSBQ suivant :

$$\mathrm{RSBQ} = 10\log\left(\frac{Px}{Pe}\right) \qquad (2\text{-}2)$$

Le RSBQ est défini comme le rapport entre la puissance du signal Px et la puissance d'erreur due au bruit Pe. Il s'agit généralement d'une quantification exprimée sur une échelle logarithmique et on doit s'assurer de l'absence de débordement dans l'intégralité de l'algorithme. On doit donc bien déterminer la longueur de la partie entière de chaque variable.

Une conception courante est de normaliser toutes les variables en s'assurant qu'elles sont présentes dans l'intervalle [-1,1].

Comme présenté dans [8], le tableau 2-1 présente une comparaison du RSBQ des deux formats : virgule flottante et virgule fixe. On remarque qu'en utilisant le format virgule fixe avec un nombre correctement échelonné, il est possible d'obtenir un RSBQ plus grand que celui de l'arithmétique virgule flottante. Le format RSBQ est une précision simple de 151 dB et une précision double de 326 dB. Il utilise des virgules fixes avec 32 bits (194 dB) et 64 bits (387 dB). En raison de ses nombreux avantages en matière d'implémentation sur le hardware (surface, vitesse, consommation), il faut étudier pour trouver un compromis optimal entre la longueur des mots binaires utilisés et les performances en précision dans cet environnement de calcul [8] en raison de leur capacité à représenter et à manipuler les chromosomes de manière binaire. Plusieurs implémentations matérielles, notamment sur des FPGA ou des ASICs (*Application Specific Integrated Circuit*), sont présentées dans la littérature [6-9].

Tableau 2-1 Comparaison du RSBQ.

Arithmétique	**Longueur binaire**	**RSBQ (dB)**
Virgule flottante simple	32 bits	151
Virgule flottante double	64 bits	326
Virgule fixe	32 bits	194
Virgule fixe	64 bits	387

2.2 Field Programmable Gate Array (FPGA)

Dans cette section, on présentera une revue des FPGA, servant à plusieurs applications. Il s'agit de circuits logiques configurables dont la configuration par programmation est très différente des technologies CPU.

2.2.1 *Constitution interne*

Comme la montre [10], les FPGA sont construits à partir d'une matrice de bloc logique reconfigurable (CLB – *Configurable Logic Block*). Ces derniers se composent eux-mêmes de quatre tranches (*Slices*). Chacune de ces tranches contient deux cellules logiques (LC – *Logic Cell*). La figure 2-3 présente la structure d'un CLB ; on y voit que la LC est composée d'une table de conversion (LUT) à quatre entrées, d'un multiplexeur, d'un registre et d'une gestion de la retenue. La LC constitue l'élément le plus petit dans un FPGA. La gestion de la retenue (carry en anglais) est interne à chaque LUT dans chaque LC. Chaque LUT est composé de seize (16) cellules de mémoire vive statique (SRAM – *Static Random Access Memory*). Par contre, lors de l'implémentation, le bloc des LUTs peut être instancié sous différentes formes. Une LUT standard à quatre entrées ou un registre ce qui permet de disposer d'une architecture Hardware adapté à la fonction souhaitée.

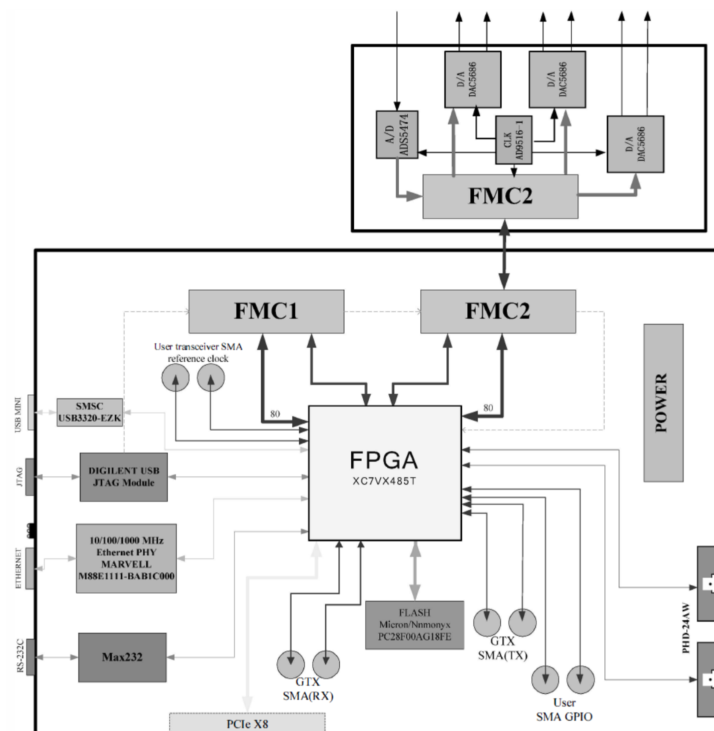

Figure 2-3 Structure interne du SoC FPGA: Virtex-7 XC7VX485T [10].

Xilinx inclut également des blocs multiplicateurs pour des opérations sur des nombres 18 bits à virgule fixe. La famille Virtex-7utilise une architecture 28 nm optimisés pour l'efficacité énergétique et atteinte une densité de 2 millions de cellules logiques. La figure 2-3 montre le modèle XC7VX485T de la série Virtex-7, supportant jusqu'à 12,5 Gb/s de GTX, et une mémoire FLASH de 1 Go pour le stockage des configurations et programmes.

2.2.2 Apport des FPGA dans la simulation numérique temps réel et implémentation

Les FPGA jouent un rôle crucial dans la simulation numérique en temps réel, permettant de répondre aux exigences de systèmes dynamiques, allant de lentes à ultrarapides. Leur capacité de calcul parallèle permet d'accélérer les performances temporelles, rendant possible la simulation en temps réel des systèmes électriques tout en conservant le parallélisme des algorithmes. Cela entraîne une réduction considérable du temps de calcul, permettant l'utilisation de pas de temps très courts (de l'ordre d'une microseconde ou moins), essentiels pour détecter les commutations rapides et améliorer la synchronisation des simulations [11].

En exploitant les ressources matérielles des FPGA, telles que les blocs logiques, mémoires, blocs DSP et E/S rapides, il est possible de mettre en œuvre des modèles complexes sur une seule plateforme. Cette approche est particulièrement adaptée à la simulation en temps réel de convertisseurs de puissance comportant de nombreux commutateurs et une taille de mot binaire flexible, optimisant ainsi l'utilisation des ressources internes du FPGA [10]. De plus, les FPGA permettent d'intégrer des cœurs de processeur (softcore ou hardcore) et des convertisseurs analogiques numériques, ce qui les rend adaptés aux applications HIL [12].

L'implémentation des FPGA passe par la génération de code à partir de modèles, facilitée par des outils Xilinx, tels que Vivado-HLS et Xilinx System Generator for DSP (XSG) sous Simulink. Dans notre projet, nous avons choisi d'utiliser HLS et XSG pour générer automatiquement le code à partir de nos modèles, simplifiant ainsi le processus de développement. Après la validation de l'algorithme, la conception de l'architecture FPGA est optimisée en utilisant des techniques comme le pipeline, afin de réduire les retards de propagation et d'augmenter la fréquence d'horloge, ce qui diminue le temps de calcul. Une fois l'architecture codée en VHDL ou Verilog [13][14], la validation fonctionnelle est réalisée à l'aide d'outils comme HLS de Xilinx, en comparant les résultats de simulation obtenus avec ceux générés par Matlab/Simulink lors du développement de l'algorithme. Cela garantit une architecture matérielle optimisée, à la fois en termes de performances temporelles et d'utilisation des ressources, tout en restant rentable.

2.3 Conception du HIL

Cette section présente quelques détails du concept du HIL.

2.3.1 Architecture de simulateur en temps réel basé sur FPGA

Pour les systèmes de conversion à plusieurs niveaux (comme dans notre cas, le système 2-L et le B2B), qui sont principalement destinés aux applications de test HIL. Le simulateur décrit dans la référence [13] répond aux défis mentionnés en mettant en œuvre une architecture matérielle de pipeline ou personnalisée sur FPGA. Ce qui permet d’obtenir des pas de temps de simulation de l’ordre de dizaines à quelques centaines de nanosecondes [13].

La figure 2-4 illustre le schéma d'un simulateur HIL. On y trouve deux schémas : a) un système embarqué connecté à un simulateur HIL, et b) représente un simulateur HIL simple. La simulation HIL se développe rapidement selon un critère de conception détaillé dans [14]. Elle est souvent utilisée comme outil puissant dans plusieurs domaines, notamment ceux des réseaux électriques (comme notre cas de projet), des avions, des missiles et des véhicules sans pilote, le système embarqué doit fonctionner en temps réel grâce au simulateur HIL, qui imite les conditions du monde réel avec des entrées et des sorties réelles.

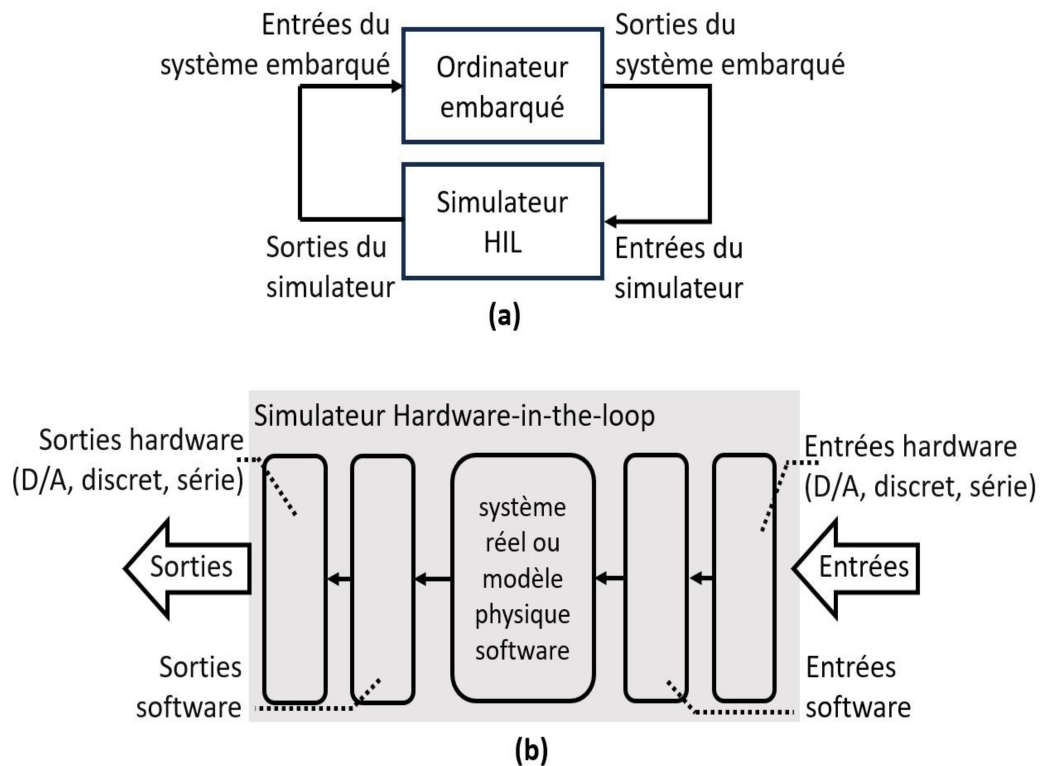

Figure 2-4 Schémas fonctionnels d'un simulateur HIL [13]

Dans [15], la simulation HIL inclut également des simulations de contrôleur dans la boucle. Habituellement, les tests du système et son évaluation se font en temps réel, avec une entrée

de commande fournie à l'intervalle d'échantillonnage souhaité dans le système embarqué. Cependant, il faut noter que le signal de commande est essentiel à la stabilité du système.

2.3.2 Test HIL en temps réel avec accélération FPGA

Dans cette section, nous allons détailler le concept du test HIL, qui est une simulation en temps réel, elle nous permet de tester le code embarqué sans le matériel réel du système afin de détecter des anomalies et des défauts de conditions qui pourraient endommager le matériel [15]. L'article de référence [16] montre que la validation du code embarqué pour les prototypes comporte beaucoup de défis, en effet, il existe un risque de dommages matériels qui empêchent le système d'être testé sur une gamme complète. Le principal avantage du système est son exécution en temps réel, ce qui rend difficile sa validation, la difficulté à valider des systèmes HIL se manifeste principalement dans le contexte des systèmes critiques où les décisions doivent être prises rapidement et où les résultats doivent être fournis dans des délais stricts. Les tests HIL sont utilisés pour vérifier la fonctionnalité de l'interface synchronisée de simulation en temps réel. Le testeur peut tester les signaux internes et configurer le design d'autre part. Il est associé à des cœurs ou noyaux spécifiques, et lorsqu'il s'agit de simuler un système en temps réel, on pense systématiquement à la simulation HIL.

Le simulateur HIL basé sur FPGA décrit dans l'article de référence [17] permet non seulement la simulation précise et en temps réel des convertisseurs de forte puissance, mais aussi la simulation en temps réel des convertisseurs électroniques de puissance à haute fréquence. Cela correspond très bien à notre cas de projet, ce qui rend cet article très pertinent comme référence. Et bien évidement ce qui importe, c'est que le coût du simulateur temps réel basé sur le FPGA est très inférieur à celui du simulateur multiprocesseur existant tout en offrant au moins une amélioration des performances.

En effet, il est évident que certains problèmes rendent l'exécution d'un modèle en temps réel difficile et dépendent du temps de pas minimum. Pour les systèmes dynamiques plus complexes, tels que l'électronique de puissance, où le pas de temps réel est de l'ordre de la microseconde, on ne sait pas si le CPU ou le FPGA est meilleur pour simuler les systèmes avec une dynamique de commutation. Cependant, lors du choix de la bonne combinaison, il est avantageux d'utiliser les FPGA dans les tests HIL dû au niveau d'intégration entre outils de simulation et outils FPGA. Dans ce cas, Xilinx Vivado, où les outils FPGA sont intégrés à la conception du système et ne sont pas accessibles directement [18] [19].

2.4 Calcul matriciel

Le calcul matriciel est un élément clé de notre projet. Dans cette section, nous allons détailler des architectures spécialisées pour la multiplication et l'inversion de matrices. Deux opérations essentielles largement utilisées dans de nombreux algorithmes et applications. L'architecture de multiplication matricielle est conçue pour optimiser le calcul efficace des produits de matrices. Quant à elle, l'architecture d'inversion matricielle, en particulier la méthode de Gauss-Jordan et le FSM, permet de calculer l'inverse d'une matrice.

2.4.1 Architecture de multiplication matricielle

Dans ce paragraphe, les articles de référence [20-23] présentent une conception basée sur un modèle pour la multiplication de matrices. Ils évoquent également le développement et l'évaluation d'un système basé sur un modèle pour la multiplication de matrices à virgule flottante ou à virgule fixe à l'aide de trois différentes architectures. La première architecture utilise N éléments de traitement identique (PE – *Processing Element*), comme illustré dans la figure 2-5 pour une taille de matrice de N × N. L'approche présente un avantage majeur :

elle utilise la fonction de parallélisme du FPGA, ce qui permet d'obtenir une sortie plus rapide au prix d'une augmentation des ressources.

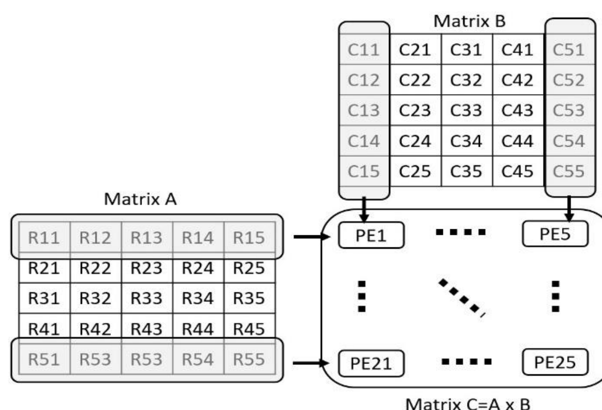

Figure 2-5 Propositions de multiplication matricielle – Architecture 1 [23]

La deuxième architecture utilise un PE différent avec N. PE identique pour une matrice de dimension N × N (voir figure 2-6). Elle fait appel à un multiplexeur dans la PE pour lire les lignes de la matrice **A** et réutilise les ressources de la PE pour calculer les éléments de sortie. Cette approche permet d'optimiser l'utilisation des ressources au détriment de la vitesse de traitement. Ces approches de multiplication matricielles proposées peuvent être facilement étendues à des tailles de matrices plus grandes en augmentant le nombre de PE pour la première et la deuxième approche.

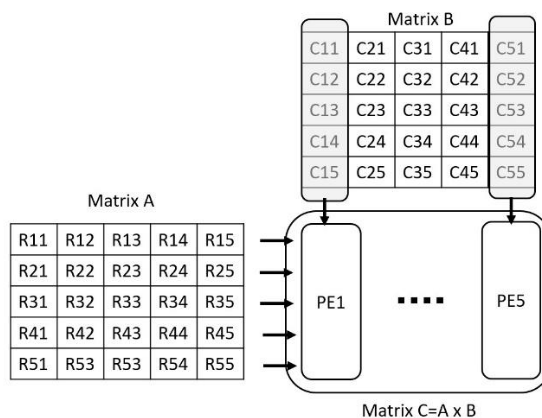

Figure 2-6 Proposition de multiplication matricielle – Architecture 2 [23]

2.4.2 Architecture d'inversion matricielle méthode de Gauss Jordan

La méthode décrite dans [23] est comparée à [24] et [25] qui proposent une conception basée sur un modèle d'inversion des matrices. La conception d'un système basé sur un modèle pour l'inversion de matrice à virgule flottante ou fixe à l'aide de la méthode de Gauss Jordan est possible. Pour inverser la matrice pleine **A** de taille N × N, on commence par l'ajouter à une matrice identité **I** de même dimension, comme le montre la figure 2-7.

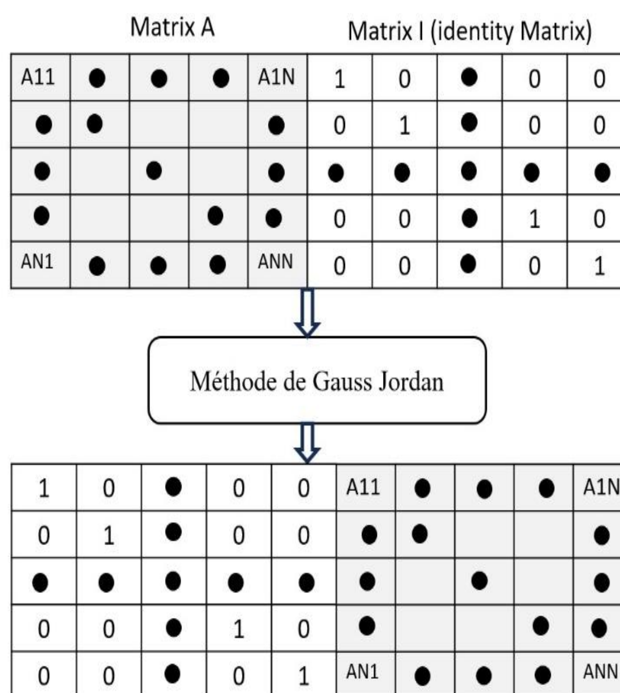

Figure 2-7 Gauss Jordan Method [23]

On applique des opérations élémentaires sur les lignes (soustraction, division, multiplication et échanges) afin que les éléments de la matrice ajoutée deviennent une matrice identité et que l'inverse de la matrice **A** soit obtenu dans les éléments de la matrice identité initiale. Il s'agit ici d'un nombre de cycle et de ressources trop important pour répondre aux exigences des applications de calcul temps réel des onduleurs triphasés visés.

2.4.3 Inversion matricielle selon la formule de Sherman-Morrison (FSM)

Dans cette section, on présentera le principe de l'inversion matricielle à partir de la FSM qui relie l'inverse d'une matrice après une petite perturbation de rang à l'inverse de la matrice originale et on analysera ensuite la structure des différentes formules de FSM et finalement, nous passerons en revue ces formules et nous examinerons quelques applications dans lesquelles elles sont très utiles.

Bien évidemment, la méthode proposée dans l'article de référence [4] montre clairement que la FSM consomme moins de ressources pour le calcul que celui qui consiste à inverser directement la matrice **A** de taille N×N et cela permet également de réduire le temps de calcul et donc la durée du pas de temps. En effet cette formule calcule l'inverse de la somme d'une matrice inversible **A** et le produit extérieur $\mathbf{uv}^T$ de vecteurs **u** et **v** et si l'inverse de **A** est déjà connu, comme dans la plupart des applications) la formule offre un moyen numériquement peu coûteux de calculer l'inverse et de corrigé **A** par la matrice $\mathbf{uv}^T$ (bien entendu que la correction peut être considérée comme une perturbation ou comme une mise à jour du rang) donc le calcul est relativement peu coûteux parce que l'inverse de $\mathbf{A}+\mathbf{uv}^T$ n'a pas besoin d'être calculé à partir de zéro (ce qui est généralement coûteux). Il suffit de perturber (ou de corriger) $\mathbf{A}^{-1}$ et on présente l'équation (2-3) de [4].

$$(\mathbf{A}+\mathbf{uv}^{\mathrm{T}})^{-1} = \mathbf{A}^{-1} - \frac{\mathbf{A}^{-1}\mathbf{uv}^{\mathrm{T}}\mathbf{A}^{-1}}{1+\mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}\mathbf{u}} \quad (2\text{-}3)$$

Par contre, la dynamique de nos variables des matrices peut varier en fonction de l'état des commutateurs du circuit à différents pas de temps et c'est ce qui rend l'approche FSM plus avantageuse que d'autres méthodes qui ne sont pas efficaces en termes de calcul et si on suppose que **A** est une matrice de N × N inversible et que **u** et **v** sont des vecteurs colonnes

de taille N, alors $\mathbf{A} + \mathbf{u}\mathbf{v}^T$ est inversible si $(1+\mathbf{v}^T\mathbf{A}^{-1}\mathbf{u})$ est non nul. Dans nos conditions de projet on peut décrire un autre problème avec les formules et que de petites erreurs s'accumulent sur de nombreux pas de temps entre l'état présent et l'état précédent des commutateurs et ces petites erreurs sont causées par l'arrondissement des valeurs fractionnaires lors de chaque opération sur tout en virgule fixe, par exemple, l'addition et la multiplication sont inhérente à effectuer en virgule fixe dans des circuits numériques comme proposer à l'article [20] en le comparant à la méthode dans [22].

2.5 Revue de la littérature

Les articles examinés proposent diverses méthodes pour l'inversion matricielle en virgule fixe sur des plateformes FPGA, mettant en avant plusieurs techniques, dont la méthode de Gauss-Jordan, la FSM, la décomposition QR et l'adjonction de colonnes. Ces approches sont utilisées pour optimiser les performances matérielles, mais chacune présente des avantages et des inconvénients, qui sont soigneusement analysés dans les travaux de recherche.

Les résultats des articles présentés dans le tableau 2-3 montrent que toutes ces approches améliorent significativement les performances par rapport aux implémentations logicielles classiques sur CPU, avec des réductions substantielles du temps de calcul, parfois de plusieurs ordres de grandeur. Cependant, les performances dépendent fortement de la méthode utilisée, du type de matrice, et des caractéristiques de la plateforme FPGA employée. Par exemple, les FPGA avec davantage de blocs DSP et de mémoire peuvent mieux gérer des matrices plus grandes et plus complexes. Certaines méthodes, comme Gauss-Jordan ou la décomposition QR, sont plus adaptées aux matrices de grande taille, mais peuvent nécessiter une gestion plus précise des erreurs de quantification en virgule fixe, ce qui limite leur efficacité dans certaines applications. En revanche, des méthodes comme

l'adjonction de colonnes sont plus simples à implémenter mais ne conviennent pas aussi bien aux matrices de grande taille ou aux systèmes qui nécessitent une haute précision.

Tableau 2-2 Exploration des méthodes d'inversion matricielle

Réf.	Méthode de l'article	Résultats principaux
[24]	Mode de communication indépendant de la taille de la matrice, atténuant les problèmes d'évolutivité. Architecture matérielle basée sur l'algorithme Kant and Kimura's pour l'inversion de matrice sur FPGA. Mode de communication indépendant de la taille de la matrice, atténuant les problèmes d'évolutivité.	Résultats expérimentaux : matrice 8×8 inversée en 1,3 µs, 64×64 en 153 µs.
[25]	Implémentation FPGA de l'algorithme de décomposition QR pour les systèmes de communication numériques. Comparaison des compromis entre virgule fixe et virgule flottante dans les conceptions FP	Le cœur QRD amélioré avec DPR réduit la latence, la surface et la puissance Débits : 2,4 M et 2,11 M de décompositions par seconde atteintes.
[26]	Mise à jour des matrices à virgule flottante et à virgule fixe pour la mise en œuvre de BFGS-QN - Vérification des propriétés des matrices et schémas de mise à l'échelle de précision pour la réduction des ressources Schémas d'échelle de précision proposés pour un compromis entre précision des ressources et précision.	Jusqu'à 10,9 % de LUT, 20,2 % de FF et 18,1 % de BRAM ont été atteints. La vitesse d'entraînement n'est pas satisfaisante avec la conception de mise à jour de la matrice **B** à point fixe.
[27]	Implémentation FPGA pour une inversion matricielle efficace sur le plan matériel dans des systèmes adaptatifs complexes. Utilise la décomposition QR et la méthode Gram-Schmidt modifiée pour l'optimisation.	Taille de matrice jusqu'à 23×23 implémentée sur Xilinx Spartan3 XC3S1000 avec un temps de calcul de 253 µs.
[28]	Détection compressée pour la récupération de signaux épars à l'aide d'une inversion matricielle basée sur FPGA. Nouvelle architecture FPGA avec méthode de type Chebyshev pour une précision élevée.	Haute précision avec une erreur de 0,0001 en moyenne. Architecture FPGA basée sur la méthode de type Chebyshev pour l'inversion de matrice.
[29]	Cellules systoliques hybrides à approximation polynomiale par morceaux - Technique de segmentation hybride pour la mise en œuvre de cellules systoliques polynomiales par morceaux	La conception proposée permet d'obtenir 7,51 méga-matrices par seconde pour effectuer des opérations matricielles 4×4 avec une latence de 12 cycles d'horloge ; la conception matérielle ne nécessite que 1474 registres de tranches, 1458 LUT dans un FPGA Virtex-5
[30]	Cette méthode consiste à décomposer la matrice en un produit d'une matrice triangulaire inférieure et de sa transposée, puis à résoudre deux systèmes triangulaires pour obtenir l'inverse.	Les résultats expérimentaux sur FPGA montrent qu'il est possible d'atteindre un facteur de vitesse de 2,6 et une puissance maximale de 41 par rapport au processeur Pentium Dual
[31]	Outil de simulation pour déterminer la longueur des mots et les facteurs d'échelle Mesure des performances à l'aide de résultats de simulation à virgule fixe pour l'inversion de matrice	Coût matériel réduit de 25% par rapport à la mise en œuvre en virgule flottante. Inversion matricielle réussie à l'aide de la décomposition de Cholesky avec simulation en virgule fixe.

2.5.1 Exploration des méthodes d'inversion matricielle en virgule fixe sur FPGA

Variété des méthodes d'inversion : les articles présentent une diversité de méthodes d'inversion matricielles. On y retrouve des approches classiques telles que Gauss-Jordan, QR et Cholesky, ainsi que des méthodes novatrices comme la décomposition polynomiale et les techniques itératives de type Chebyshev. Cette diversité met en évidence l'importance de sélectionner la méthode la mieux adaptée en fonction des contraintes matérielles et des exigences de précision.

Adaptation aux contraintes des FPGA : les auteurs s'efforcent d'adapter les algorithmes d'inversion matricielle aux spécificités des FPGA. Ils prennent en compte les limitations des ressources matérielles ainsi que la nécessité de minimiser la latence. Cela se traduit par des architectures matérielles sophistiquées et des techniques de conception innovantes pour optimiser les performances et la consommation de ressources.

Optimisation des performances et des ressources : Les architectures FPGA présentées dans les articles visent à améliorer les performances en termes de vitesse de calcul, de précision et d'efficacité énergétique tout en minimisant l'utilisation des ressources matérielles (Luts, registres et mémoire BRAM). Cette approche témoigne des efforts constants pour exploiter pleinement le potentiel des FPGA dans les applications d'inversion matricielle.

Limitations et défis : Les articles présentent une évaluation détaillée des performances des architectures proposées bien que les approches matérielles présentent certains avantages, elles comportent également des limites. On retrouve entre autres des contraintes liées aux ressources FPGA, des problèmes de précision dus à la quantification en virgule fixe et des difficultés liées à la taille de la matrice gérée par le matériel FPGA.

2.6 Conclusion

En résumé, les travaux de recherche sur l'inversion matricielle en virgule fixe sur FPGA présentent un potentiel prometteur pour accélérer les calculs matriciels sur des plates-formes matérielles tout en conservant une précision acceptable. Toutefois, il est nécessaire d'effectuer des efforts supplémentaires pour surmonter les limitations spécifiques à chaque approche et pour adapter ces méthodes à différentes applications et plates-formes matérielles.

Dans cette section, nous avons abordé plusieurs aspects importants liés à l'utilisation des calculs en virgule fixe et des (FPGA) dans le contexte de la simulation numérique en temps réel et du calcul matriciel. Nous avons notamment examiné le codage des données en virgule fixe et comparé leurs avantages et leurs inconvénients par rapport à la virgule flottante. Nous avons également étudié l'architecture interne des FPGA. Ensuite, nous avons examiné le calcul matriciel, en étudiant de près les architectures de multiplication et d'inversion matricielles, y compris les formules de FSM. Enfin, nous avons effectué une revue de littérature approfondie sur les méthodes d'inversion matricielle en virgule fixe sur FPGA.

En conclusion, ce chapitre souligne l'importance croissante des plates-formes FPGA dans l'accélération des calculs matriciels en virgule fixe. Les résultats de ces travaux de recherche sont précieux et permettent de mieux comprendre les progrès réalisés dans ce domaine. Ils ouvrent également la voie à de nouvelles possibilités pour exploiter pleinement le potentiel des FPGA dans différents domaines d'application.

Chapitre 3 - FSM sur FPGA – Précision

Le projet propose une approche innovante pour l'inversion de matrices en exploitant les capacités des FPGA, offrant une solution optimisée pour atteindre des latences ultra-faibles dans la résolution. Offrant ainsi des possibilités inédites pour une simulation précise et réactive des systèmes d'onduleurs.

3.1 Introduction

Ce chapitre examine les performances de notre méthode d'inversion matricielle appliquée aux diverses méthodes de simulation et de calcul pour les deux types de circuits de conversion de puissance (deux niveaux et *Back-to-Back*, B2B). Nous étudierons différentes techniques employées pour réaliser l'inversion matricielle basée sur la FSM, dont nous évaluerons l'efficacité dans nos cas particuliers de nos circuits de puissance. Pour cela, nous nous concentrerons sur la méthode de FSM en virgule fixe et ses répercussions sur la dynamique du calcul, en soulignant son importance pour la gestion efficace des applications qui exigent une grande précision numérique. Par la suite, nous aborderons également une méthode de correction stratégique destinée à améliorer la précision de calcul dans nos simulations en considérant les erreurs et les approximations engendrées par les méthodes de calcul. Nous terminerons cette analyse par un résumé des principaux points abordés et une mise en évidence des considérations clés y compris un aperçu approfondi des performances de notre technique d'inversion matricielle sur FPGA. La figure 3.1 présente un onduleur à deux niveaux. Ce type d'onduleur est composé de deux niveaux de tension, généralement un

niveau positif et un niveau négatif. Il offre une solution simple et rentable pour convertir le courant continu en courant alternatif.

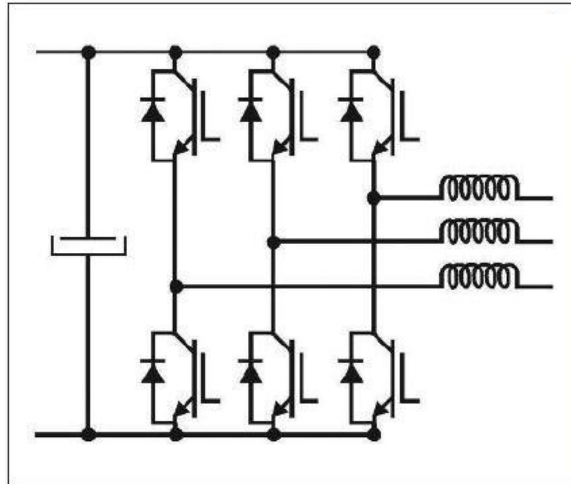

Figure 3-1 Onduleur à 2 niveaux(2-L).

En complément, le convertisseur « Back-to-Back », illustré à la figure 3.2, fait également l'objet d'une étude approfondie. Cette configuration consiste en l'interconnexion en série de deux convertisseurs, permettant ainsi un transfert bidirectionnel d'énergie entre deux systèmes en alternatif ou entre un système en courant alternatif.

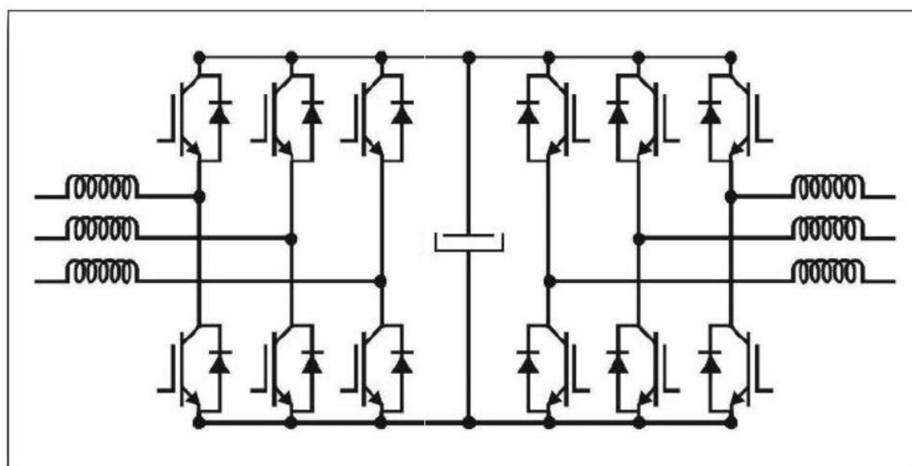

Figure 3-2 Onduleur *Back-to-Back* (B2B).

Pour analyser et comprendre le comportement électrique global d’un tel circuit d’onduleur, il faut passer par l’étape de modélisation en identifiant les éléments du circuit et leurs équivalences en termes d’impédances. En appliquant, une analyse nodale où l’interrupteur est modélisé par leur impédance équivalente aux deux états (ON est OFF) par Ron et Roff, on construit une matrice d’impédance du circuit d’un tel onduleur ainsi que des modèles de commutation à deux états pouvant servir dans des simulations en temps réel ayant des périodes très courtes.

3.2 FSM et dynamique des calculs

Pour résoudre le problème d'inversion matricielle, nous utilisons le FSM et celles-ci nous permettent de calculer l’inverse de la matrice **A** à chaque pas de calcul en temps réel. La version originale de ces formules réduit l’intensité du calcul d’inversion matriciel dans certaines conditions, mais elle n'est pas directement applicable dans nos cas de figure. En effet on propose une version modifiée des FSM spécifiquement conçue pour la simulation en temps réel basée sur FPGA de convertisseurs d'électroniques de puissance. Nous optimisons aussi la solution proposée grâce à de nouvelles techniques additionnelles de correction étudiée à la dernière section de ce chapitre qui nous permet d’obtenir un compromis entre précision et quantité de mémoire.

3.2.1 Revue de l’algorithme de FSM

Nous avons développé notre approche basée sur la FSM, qui permet de diminuer considérablement le temps nécessaire au calcul de l’inversion matricielle en temps réel des convertisseurs de puissance basés sur FPGA. En effet, cette méthode permet d’effectuer une série de mesures plus simples afin d’obtenir le résultat souhaité. L’objectif consiste à éviter

de devoir recalculer intégralement l'inverse d'une matrice lorsqu'on y apporte un changement minime, comme l'ajout ou la suppression d'une ligne ou d'une colonne. Cette technique permet ainsi d'éviter le stockage complet des matrices inversées en mémoire, ce qui peut s'avérer très couteux en termes de ressources. Seules quelques matrices sont enregistrées et on effectue la mise à jour itérative de la matrice inverse à partir de celles-ci.

Comme nous le montre l'équation (2-3) une mise à jour de $\mathbf{A}^{-1}$ implique une addition, une multiplication et une division matricielles. Cela demande beaucoup de calculs et dépend aussi de la taille N de la matrice **A** de taille $N \times N$ qui influence le nombre de lignes et de colonnes affectées dans **A,** avec n est le pas de calcul. Mais, dans la simulation en temps réel des convertisseurs de puissance, le nombre des commutateurs du circuit peut être grand ou très grand selon leur état à différentes itérations. Lorsqu'ils sont grands, la formule présentée n'est pas efficace pour son implantation. Un autre problème limitant de la formule est l'instabilité numérique. Vu que l'inverse de la matrice est mis à jour à l'itération i en fonction de sa valeur précédente à l'itération $i-1$, ce qui provoque l'accumulation des petites erreurs sur de nombreuses itérations. Cette erreur s'ajoute à une autre, causée par l'arrondissement des valeurs fractionnaires lors de chaque opération arithmétique dans notre cas de figure, soit l'addition ou la multiplication sur la configuration numérique choisie soit une virgule fixe ou flottante. Dans ce contexte une itération est une étape de calcul où une seule perturbation de rang 1 est appliquée à la matrice de référence **A** et utilisé pour mettre à jour son inverse $\mathbf{A}^{-1}$. Dans chaque itération, la formule de mise à jour exploite les vecteurs **u** et **v** pour intégrer cette perturbation. Le processus se répète autant de fois qu'il y a de perturbations successives à traiter, cette perturbation est dû au changement d'état des commutateurs de l'onduleur. Donc pour éviter ce genre de problème tout en profitant des avantages de la FSM en termes

de latence et de quantité de données stockées dans la mémoire, nous proposons la modification suivante.

3.2.2 Reformulation de la FSM

Pour simuler en temps réel, nous devons calculer l'inverse des matrices plus rapidement et efficacement que la méthode traditionnelle basée sur la formule fondamentale de FSM, qui présente des limitations, comme il est mentionné dans la section précédente. Une autre écriture de la FSM est proposée dans les travaux de maîtrise de Jérémy Poupart, tels que présentés dans son mémoire [1] et dans l'article [2], à savoir :

$$(\mathbf{A} + \mathbf{u}\mathbf{v}^{\mathrm{T}})^{-1} = \mathbf{A}^{-1} - \sigma\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1} \tag{3-1}$$

Où σ est un scalaire exprimé par

$$\sigma = \frac{1}{1+\mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}\mathbf{u}} \tag{3-2}$$

A est la matrice de référence et **u** et **v** sont deux vecteurs colonnes qui représentent les perturbations appliquées à la matrice **A**. Selon FSM, décrite au chapitre 1, la matrice **A** et son inverse $\mathbf{A}^{-1}$ sont précalculés afin qu'elles puissent servir de références pour les matrices à l'itération suivante. Quand on ajoute une perturbation $\mathbf{u}\mathbf{v}^{\mathrm{T}}$ à la matrice **A**, celle-ci permet de calculer rapidement l'inverse mis à jour en modifiant le référentiel inverse $\mathbf{A}^{-1}$. Cette méthode est applicable si on considère chaque modification comme étant composée de plusieurs perturbations de rang 1 et itérer la formule jusqu'à ce que toutes les perturbations soient traitées. Dans ce cas, la matrice de référence doit traduire fidèlement le système pour minimiser les différences entre la matrice de référence et la nouvelle matrice afin de faciliter le calcul. Et quand on a fait l'analyse détaillée des matrices et vecteurs de la FSM spécifique à notre circuit de puissance, on a remarqué que **v** est un vecteur rempli de seulement 0 sauf

en deux éléments de 1 et -1 sur l'élément correspondant aux deux colonnes sélectionnées. Le vecteur **u** représente les éléments qui diffèrent entre la matrice de référence et la nouvelle matrice à inverser dans la colonne concernée, sélectionnée par le vecteur **v**. on peut alors se débarrasser du calcul complet du produit de vecteur **v** et ne sélectionner que les colonnes et les lignes des éléments correspondant à l'emplacement où se trouvent les éléments non nuls de **v** et **u** cela facilite énormément le calcul ou le précalcul de σ décrit par l'équation (3-2), détaillée plus haut, et la quantité $\mathbf{v}^T\mathbf{A}^{-1}\mathbf{u}$ deviendrait un produit scalaire entre les deux lignes de la matrice $\mathbf{A}^{-1}$ sélectionnées par **u** et **v**.

3.2.3 Dynamique des calculs

La dynamique des variables matricielles peut varier considérablement lorsque nous effectuons une simulation en temps réel des convertisseurs de puissance. Cela dépend de l'état des commutateurs du circuit à différents pas de temps. Cette dynamique est caractérisée par des variations importantes dans les valeurs des variables matricielles. Une analyse plus poussée est donc nécessaire pour comprendre et modéliser précisément le comportement du système. Les figures 3-3 et 3-4 montrent la distribution de l'inverse de la matrice d'admittance **A**, qui représente un convertisseur statique, pour les systèmes à deux niveaux et (B2B). Elles mettent en évidence les fluctuations significatives dans les valeurs des matrices, ce qui souligne l'importance d'en tenir compte lors de la simulation. Cette dynamique peut résulter de plusieurs facteurs, notamment des changements rapides dans l'état des commutateurs, des variations de charge ou d'autres éléments influençant les caractéristiques électriques du système. La compréhension et le développement d'un modèle de cette dynamique sont essentiels pour mettre au point des stratégies de contrôle efficaces et assurer le bon fonctionnement du convertisseur de puissance dans des conditions variées.

Il faut noter que la matrice **A** est une matrice carrée de dimension N, et que **u** et **v** doivent être des vecteurs colonnes de taille N. Ainsi, $\mathbf{A} + \mathbf{u}\mathbf{v}^T$ est inversible si $(1+\mathbf{v}^T\mathbf{A}^{-1}\mathbf{u})$ est différent de zéro.

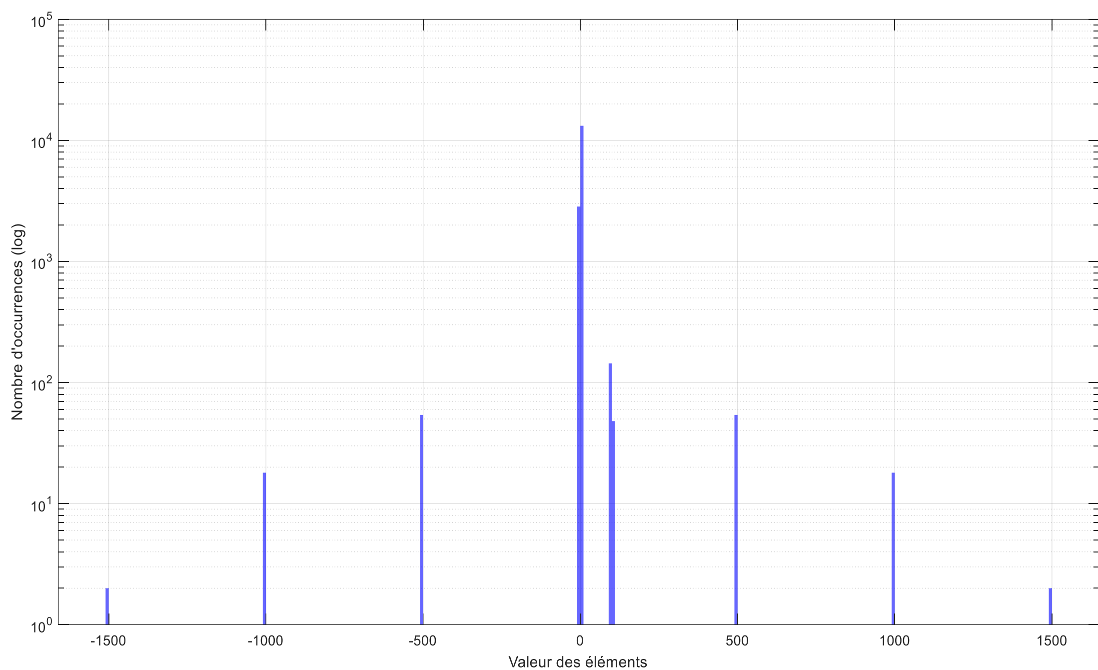

Figure 3-3 Distribution de l'inverse de la matrice **A** (16×16) pour le 2-L.

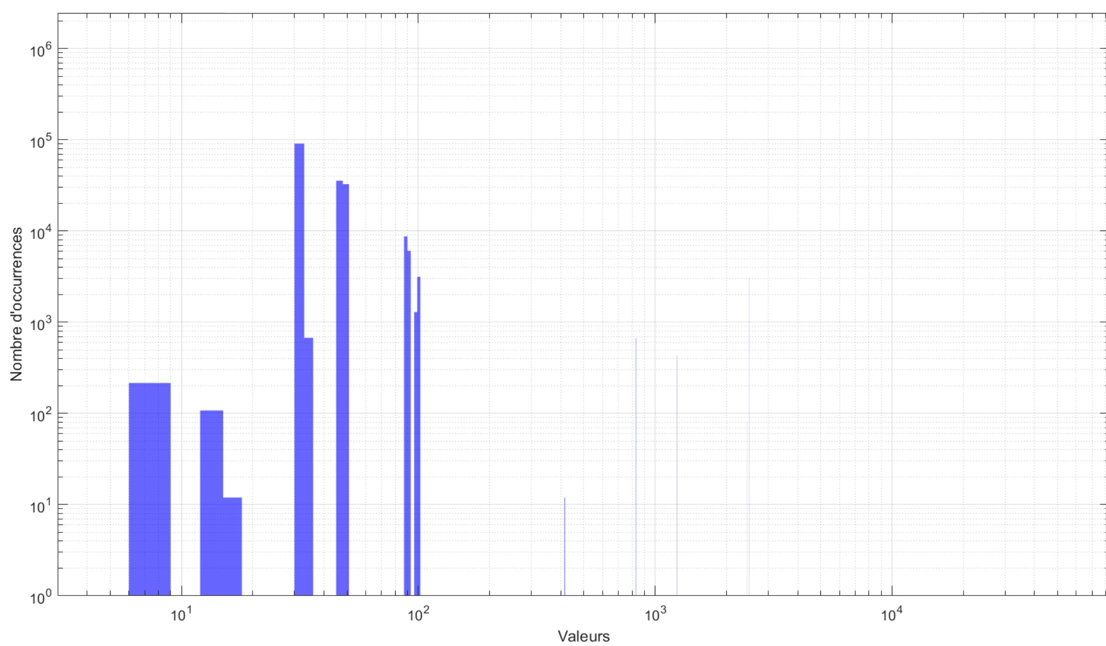

Figure 3-4 Distribution de l'inverse de la matrice **A** (26×26) pour le B2B.

3.2.4 Application de la mise à l'échelle (Scaling)

Lorsque nous effectuons des calculs impliquant des matrices dont la plage de variation des valeurs est très importante, il est nécessaire de réduire cette dynamique pour pouvoir effectuer des opérations sur des nombres en arithmétique entière, à savoir, lorsque nous disposons d'un nombre limité de bits pour représenter les nombres. La technique du *Scaling* ou mise à l'échelle est souvent utilisée pour normaliser les valeurs des variables et réduire leur plage de distribution.

L'idée principale du facteur d'échelle consiste à multiplier toutes les valeurs des variables par un facteur d'échelle approprié afin de réduire leur amplitude. Cela permet de ramener les valeurs des variables dans une plage plus restreinte de nombre entier, ce qui facilite leur manipulation avec un nombre fixe de bits. Par exemple, on peut recourir à un facteur de mise à l'échelle pour ramener cette plage à des valeurs plus petites, comme -1 et 1, en multipliant toutes les valeurs par un facteur d'échelle approprié. Ainsi des petites erreurs peuvent s'accumuler au fil des itérations en raison de l'arrondi des valeurs fractionnelles pendant les opérations arithmétiques. Cependant, en réduisant la partie entière, cela réduit les risques de débordement et améliorer la précision des calculs en virgule fixe.

$$(\mathbf{A}_{\text{norm}} + \mathbf{u}\mathbf{v}_{\text{norm}}{}^{\text{T}})^{-1} = \mathbf{A}_{\text{norm}}{}^{-1} - \frac{\mathbf{A}_{\text{norm}}{}^{-1}\mathbf{u}\mathbf{v}_{\text{norm}}{}^{\text{T}}\mathbf{A}_{\text{norm}}{}^{-1}}{1 + \mathbf{v}_{\text{norm}}{}^{\text{T}}\mathbf{A}_{\text{norm}}{}^{-1}\mathbf{u}} \tag{3-3}$$

où $\mathbf{A}_{\text{norm}} = {}^{\mathbf{A}}/_{\beta}$ et $\mathbf{v}_{\text{norm}} = {}^{\mathbf{v}}/_{\beta}$.

Le facteur d'échelle, β, est crucial dans le contexte des opérations en virgule fixe dans les circuits numériques, tel qu'expliqué dans [15]. En effet, il assure la précision des calculs. Il est illustré dans l'équation (3-3), où $\mathbf{A}_{\text{norm}}$ représente la matrice normalisée à virgule fixe obtenue en divisant chaque élément de la matrice **A** par un facteur d'échelle approprié. De

mêmes $\mathbf{v}_{\mathbf{norm}}$ c'est le vecteur **v** normalisé, calculé en divisant chaque élément de **v** par le facteur d'échelle. Les vecteurs colonnes **u** et **v** sont aussi normalisés en fonction de la perturbation de la matrice. Cette normalisation permet de réduire la plage de valeurs des variables, ce qui facilite leur représentation sous forme binaire avec un nombre limité de bits, sans sacrifier la précision des calculs. L'intégration du facteur d'échelle dans les opérations en virgule fixe nous permet de garantir la stabilité et la précision des calculs, même lorsque les variables matricielles présentent une grande dynamique.

3.3 Calcul en virgule fixe

Comme on a bien expliqué le principe du calcul en virgule fixe dans la section 2.1, le processus de conversion d'algorithmes à virgules flottantes en versions à virgules fixes est essentiel dans le contexte de MATLAB pour les implémentations FPGA d'applications de traitement du signal numérique. Cette conversion a pour but de préserver la fiabilité de l'algorithme, tout en limitant l'utilisation des ressources matérielles et les erreurs de quantification dans des plages spécifiques. En outre, l'arithmétique en virgules fixes est souvent privilégiée dans les conceptions pratiques de FPGA en raison de son faible coût et de sa simplicité. En effet tout au long de cette section on va ce concentré sur l'application du calcul en virgule fixe sur l'algorithme de FSM appliqué à nos cas de convertisseur de puissance les deux niveaux et le B2B.

3.3.1 Méthode de conversion

La représentation par virgule fixe réduit la précision des calculs en raison du nombre limité de bits utilisés pour représenter les données. Cela se traduit par une augmentation du bruit de quantification lorsqu'on élimine des bits via des opérations de saturation, ainsi qu'un

problème de débordement important lorsque la longueur de mot de la partie entière est insuffisante pour représenter la dynamique entière des variables de calcul, une configuration binaire idéale avec une virgule fixe ramène toutes les valeurs des variables entre 0 et ±1. L'objectif est d'avoir un seul bit dans la partie entière pour le bit de signe et tous les autres bits pour la partie fractionnaire afin d'obtenir la plus grande précision possible, comme l'indique l'équation.

Fonction *quantizer* versus fonction *fi*

La fonction *quantizer*(•) de MATLAB est un outil puissant pour la conversion en virgule fixe, offrant une approche simplifiée pour la quantification des nombres, elle est conçue pour les tâches de quantification simples et offre une interface conviviale pour les utilisateurs qui recherchent une solution rapide et efficace. En comparaison, la fonction *fi*offre un contrôle plus détaillé sur la représentation en virgule fixe. La fonction *quantizer* permet de convertir les nombres à virgule fixe en spécifiant simplement la longueur totale du mot en bits et le nombre de bits après la virgule. Cela offre une approche simplifiée de la conversion en virgule fixe qui ne requiert aucune spécification détaillée des paramètres, comme la gestion des dépassements ou des modes de sur dépassement. Cette fonction est idéale pour les tâches de quantification simples qui n'exigent pas un contrôle détaillé. Bien que cette fonction offre une approche simplifiée de la conversion en virgule fixe, *quantizer* manque de flexibilité par rapport à la fonction *fi*. Cette dernière permet de spécifier si le nombre est signé ou non, alors que *quantizer* ne supporte pas cette fonctionnalité. En outre, *quantizer* ne propose pas de fonctionnalités avancées telles que la gestion des dépassements ou l'arrondi, limitant ainsi sa capacité à traiter des scénarios complexes, mais l'une des principales forces est sa facilité

d'utilisation un syntaxe simple et intuitive permet aux utilisateurs de convertir facilement des nombres en virgule fixe, sans qu'ils aient à spécifier de nombreux paramètres.

La fonction *fi*(•) de MATLAB est très polyvalente. Elle permet de représenter des nombres avec une virgule fixe tout en offrant un contrôle précis sur leur précision et leur plage de même on spécifier des paramètres comme la longueur totale du mot en bits, la position de la virgule (nombre de bits après celle-ci) ainsi que s'il s'agit d'un nombre positif ou négatif, de plus, elle propose des fonctionnalités avancées, telles que la gestion des débordements, l'arrondit et le contrôle des modes de sur débordement donc elle est couramment employée pour des applications qui exigent une grande précision et un contrôle minutieux sur la représentation en virgule fixe parfaite pour notre cas d'application qu'on va détaillé l'effet de la quantification en virgule fixe sur la performance en particulier sur l'erreur relative de FSM en fonction des différentes configurations binaires.

3.3.2 FSM en virgule fixe-performance

Dans cette partie on va passer en revue sur l'étude de performance de FSM en virgule fixe. Plusieurs aspects sont étudiés pour trouver le meilleur compromis entre le nombre de bits utilisé versus la précision des calculs et dans ce cadre, il est important de bien comprendre et de gérer la précision des calculs. Dans cette analyse, nous examinons l'impact de la quantification en virgule fixe sur l'erreur relative de FSM pour les deux convertisseurs de puissance le 2-L et le B2B. Pour chacune de ces configurations, nous évaluons l'erreur relative de la FSM en comparant les résultats quantifiés aux résultats non quantifiés.

$$\text{Erreur Absolue} = Max \left\| \mathbf{A}^{-1} - \mathbf{A}_{SM_fixe}^{-1} \right\| \tag{3-4}$$

Pour les différentes figures qui suivent, les matrices sont classées en abscisse en fonction de leur erreur croissante en ordonnée. Celle-ci est calculée à partir de la formule (3-4) de l'erreur absolue maximale, définie comme la valeur absolue de la différence entre l'inverse exact de la matrice **A** et son inverse approximatif calculé par FSM avec différentes configurations binaires.

Tests du convertisseur à deux niveaux (2-L)

Le convertisseur 2-L possède 6 interrupteurs. Pour modéliser tous les cas possibles, il faut 2^6 matrices. Le seuil d'erreur ou la précision recherchée est de l'ordre de 10^{-7} comparable à celle obtenue en utilisant la précision flottante simple précision (32 bits).

La figure 3-5 évalue la précision de la représentation en virgule fixe avec une longueur de mot totale fixée à 72 bits. Nous avons mené une série de tests en variant la longueur de la partie entière du mot binaire et on a examiné quatre configurations différentes, avec des longueurs de partie entière de 16, 18, 20 et 32 bits respectivement, et pour chaque configuration, on a calculé l'erreur absolue associée à la représentation en virgule fixe.

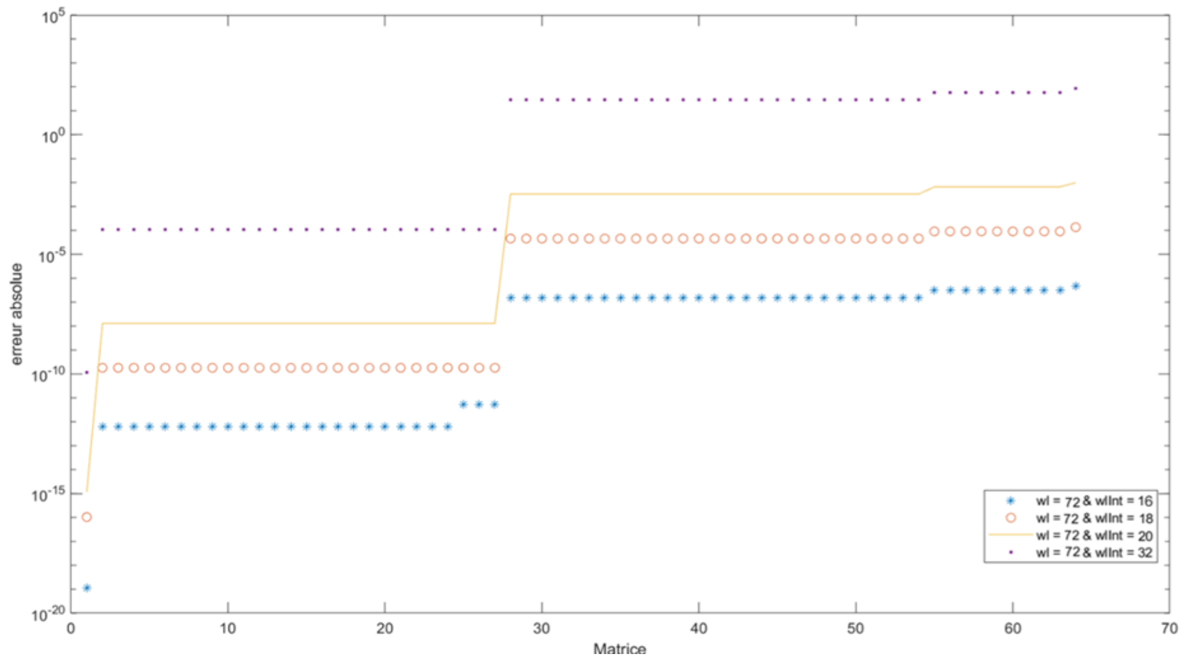

Figure 3-5 Erreur absolue selon (3-4) pour le convertisseur 2-L avec un mot binaire de 72 bits et différentes parties entières de 16, 18, 20 et 32 bits. L'erreur maximum de (3-4) en virgule flottante 32 bits est de 1×10^{-7}.

Et par suite de nos tests, on a généré des courbes d'erreur absolue pour chaque configuration, afin de visualiser les performances de la représentation en virgule fixe. On a observé que la configuration avec une longueur de partie entière de 18 bits (Wint = 18) a présenté l'erreur absolue la plus faible parmi les configurations testées. Cette configuration, avec une longueur de mot totale de 72 bits et une longueur de partie entière de 18 bits, a démontré une meilleure précision de représentation par rapport aux autres configurations.

La figure 3-6 étudie la précision de la représentation en virgule fixe avec une longueur de partie entière fixée à 18 bits et en variant la longueur totale du mot binaire (Wl), nous avons effectué une série de tests en examinant cinq configurations différentes (Wl, Wint): (36, 18), (54,18), (72, 18), (90,18) et (108, 18) bits. Pour chacune de ces configurations, nous avons généré des courbes d'erreur absolue afin de comparer les performances de représentation. Nos résultats ont montré que la configuration avec une

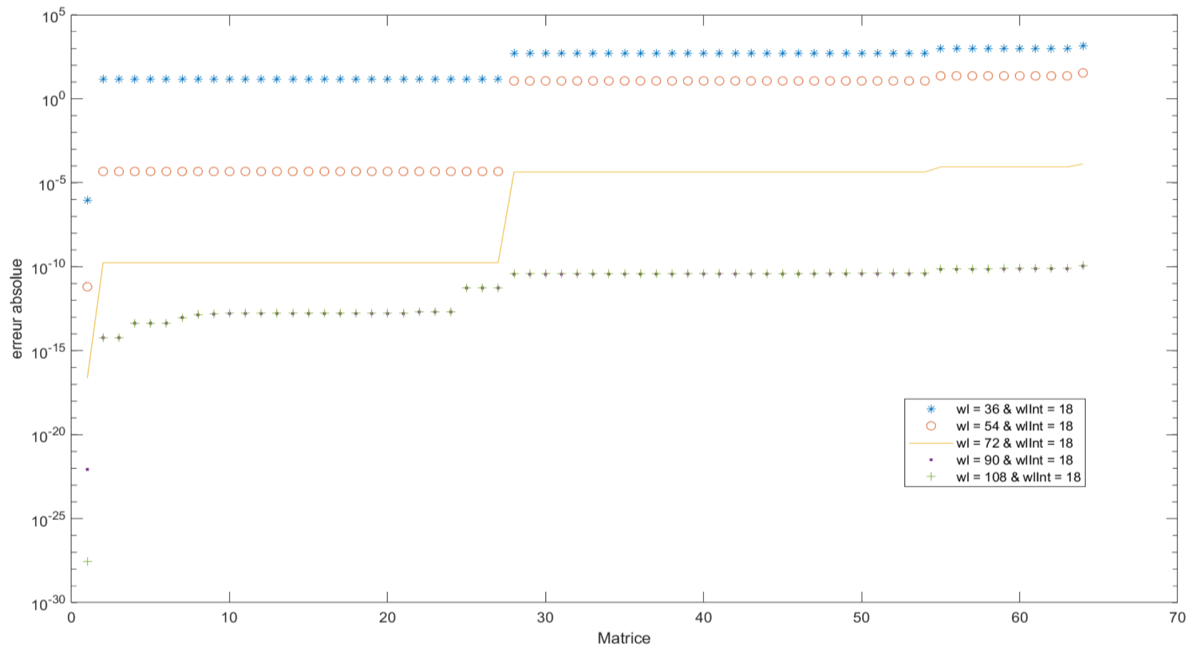

Figure 3-6 Erreur absolue selon (3-4) pour le convertisseur 2-L pour les configurations différentes (wl, wlInt): (36, 18), (54,18), (72, 18), (90,18) et (108, 18). L'erreur max de (3-4) en virgule flottante 32 bits est de 1×10^{-7}.

longueur totale du mot binaire de 90 bits et une longueur de partie entière de 18 bits présente l'erreur absolue la plus faible. On remarque la superposition pour wl=90 et 108 bits.

Tests du convertisseur Back-to-back (B2B)

Dans ce paragraphe on va suivre les mêmes démarches pour tester les performances de FSM en virgule fixe sur le convertisseur B2B avec la particularité que l'approximation de l'inverse de la matrice qui définissent l'états des interrupteurs, est à 12 commutateurs comme déjà présenté au deuxième chapitre donc pour modéliser tous les cas possibles il faut 2^{12} matrices ce qui fait 4096 matrices à tester. Le seuil d'erreur ou la précision recherchée est de l'ordre de 10^{-5} comparable à celle obtenue en utilisant la précision flottante simple précision (32 bits).

La figure 3-7 vise à évaluer la précision de la représentation en virgule fixe, tout en maintenant une longueur totale du mot binaire (Wl) à 77 bits, nous avons entrepris une série de tests en variant la longueur de la partie entière (Wint). Quatre configurations distinctes ont été examinées pour Wint : 12, 18, 24 et 32 bits. Pour chacune de ces configurations, nous avons calculé l'erreur absolue maximale, représentant la différence absolue entre l'inverse exact de la matrice **A** et son inverse approximatif obtenu grâce à la représentation en virgule fixe. L'analyse des courbes d'erreur absolue associées à chaque configuration a révélé que la configuration avec une longueur de partie entière de 18 bits (Wint = 18) offrait la meilleure résolution, présentant l'erreur absolue maximale la plus faible parmi les configurations testées.

Enfin, la figure 3-8 illustre qu'en maintenant une longueur de partie entière de 18 bits et en variant la longueur totale du mot binaire (Wl), nous avons exploré cinq configurations différentes. Après une analyse des courbes d'erreur absolue, il est apparu que la configuration avec Wl = 108 bits et Wint = 18 bits se démarquaient en offrant la meilleure résolution, avec une erreur absolue maximale minimale parmi les configurations testées.

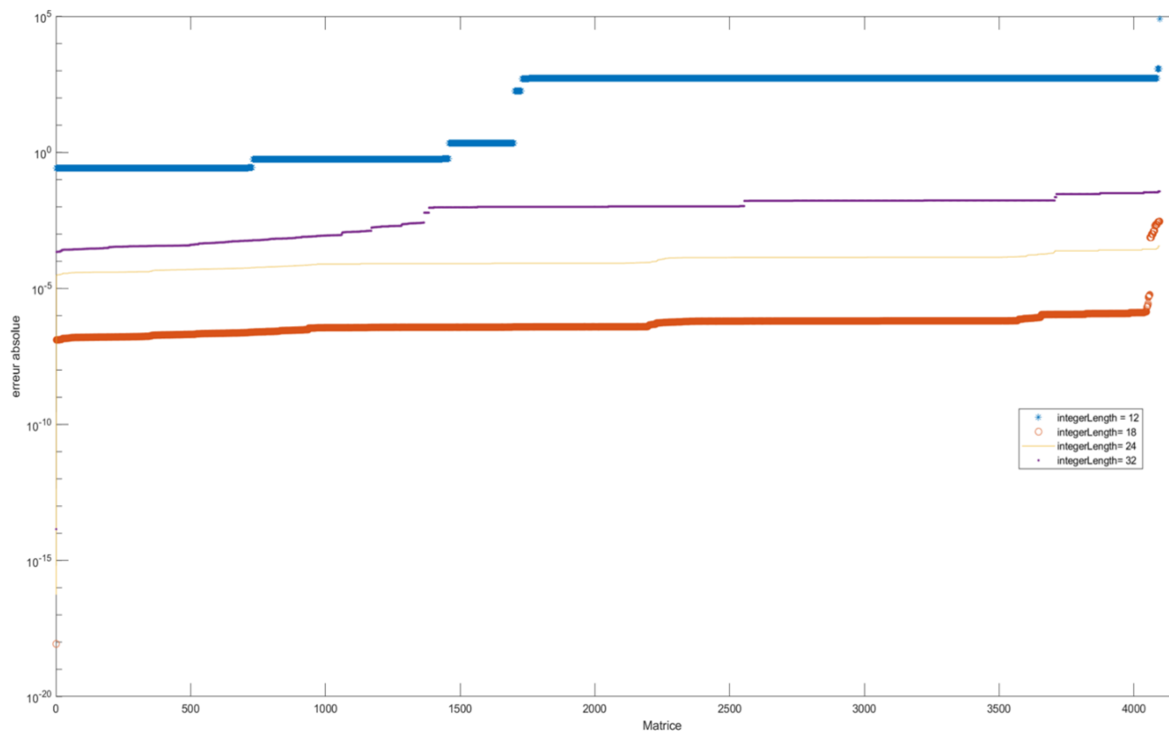

Figure 3-7 Erreur absolue selon (3-4) pour le convertisseur B2B avec un mot binaire de 77 bits et différentes parties entières de 12, 18, 24 et 32 bits. L'erreur max de (3-4) en virgule flottante 32 bits est de 1×10^{-5}.

Ces résultats soulignent l'importance de choisir judicieusement les paramètres de représentation en virgule fixe pour garantir une précision optimale dans diverses applications.

On observe que l'erreur relative de FSM a tendance à augmenter lorsqu'on utilise une quantification en virgule fixe. Toutefois, l'ampleur de cette augmentation varie selon la configuration binaire choisie. Les configurations ayant un plus grand nombre total de bits ainsi qu'un plus grand nombre de bits après la virgule présentent une erreur relative moins grande; toutefois, cette précision accrue peut nécessiter une charge de calcul plus importante en termes de quantité mémoire. Cette analyse montre que la quantification en virgule fixe a un impact significatif sur l'erreur relative de FSM, et nos constatations mettent en évidence l'importance du choix d'une configuration binaire adéquate pour l'utilisation d'un tel algorithme.

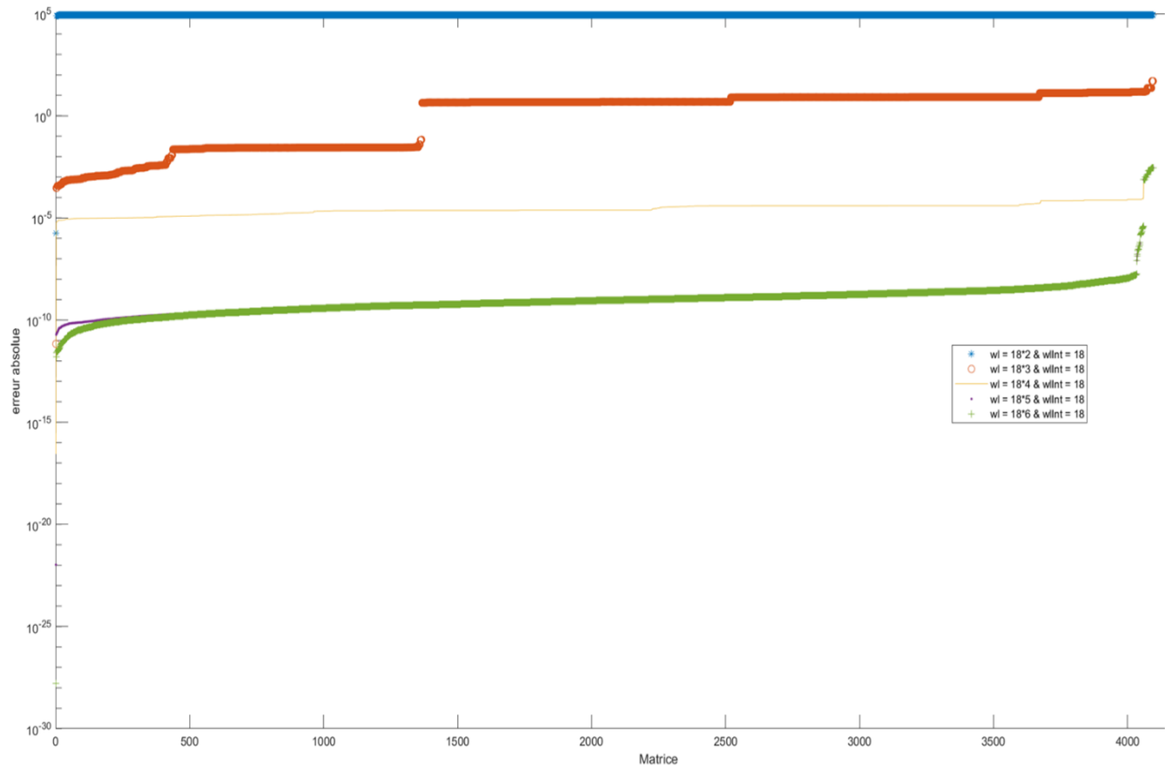

Figure 3-8 Erreur absolue selon (3-4) pour le convertisseur B2B pour une partie entière de 18 bits et un mot binaire multiple de 18. L'erreur max de (3-4) en virgule flottante 32 bits est de 1×10^{-5}.

Un nombre total de bits plus élevé ainsi qu'un nombre de bits après la virgule plus grand peuvent améliorer la précision, mais au détriment d'une charge de calcul plus importante. Il est donc essentiel de trouver un compromis entre la précision désirée et l'efficacité du calcul c'est pour cela qu'on a recoure à une méthode de facteur d'échelle qui limite la plage dynamique des calcule mais le choix du facteur d'échelle n'est pas aussi évident, nous détaillant dans la section suivante une étude de synthèse qui évalue la performance de FSM en fonction du facteur d'échelle et la configuration binaire chosiste.

3.3.3 FSM - facteur d'échelle et configuration binaire

Comme on a déjà initié à la section précédente le choix du facteur d'échelle optimal n'est pas toujours évident vu qu'il dépend fortement des caractéristiques spécifiques de

l'application et des compromis entre la précision et les contraintes de ressources matérielles. Nous avons donc entrepris une étude de synthèse approfondie pour mieux comprendre l'impact du facteur d'échelle sur la performance de FSM en virgule fixe. Les configurations binaires étudiées comprennent différentes combinaisons de plusieurs longueurs de partie entière (Wint) et de longueurs de mot binaire (Wl) fixe à 90 bits. Pour chaque configuration, l'erreur maximale (Erreur max) décrite par l'équation (3-5) et l'erreur moyenne décrite par l'équation (3-6) sont rapportées.

Les facteurs d'échelle β varient de 2^{-2} à 2^{20}.

$$\text{Erreur Max} = Max \left\| I - \mathbf{A} \times \mathbf{A}_{SM_fixe}^{-1} \right\| \tag{3-5}$$

$$\text{Erreur Mean} = Mean \left\| I - \mathbf{A} \times \mathbf{A}_{SM_fixe}^{-1} \right\| \tag{3-6}$$

Analyse de précision du 2-L

Le Tableau 3-1 présente une analyse détaillée de l'erreur en fonction de la configuration binaire et du facteur d'échelle. Les résultats présentés par ce tableau montrent des tendances intéressantes, chaque combinaison d'échelle et de Wint est accompagnée d'une erreur correspondante, mettant en lumière la variabilité des performances en fonction des paramètres choisis. Les résultats indiquent que les erreurs présentent une tendance à diminuer lorsque l'on utilise des échelles inférieures, telles que 2^{-2}. Dans ces cas, les erreurs demeurent relativement faibles, ce qui peut être interprété comme une indication de précision accrue pour des valeurs de Wint plus élevées.

Tableau 3-1 Erreur selon (3-5) et (3-6) en fonction de la configuration binaire et le facteur d'échelle pour le 2-L pour Wl fixe de 90 bits.

Facteur d'échelle (β)	**Wint pour Wl 90 bits**	**Erreur Max**	**Erreur Mean**	**Facteur d'échelle** (β)	**Wint pour Wl 90 bits**	**Erreur Max**	**Erreur Mean**
2^{-2}	20	2.6×10^{3}	1.9×10^{2}	2^{8}	20	1.5×10^{4}	0.8×10^{3}
	18	3.7×10^{-1}	3.1×10^{-2}		18	2.6×10^{3}	1.9×10^{2}
	16	2.3×10^{-9}	1.6×10^{-10}		16	3.7×10^{-1}	3.1×10^{-2}
	14	1.0×10^{-9}	0.6×10^{-10}		14	2.6×10^{3}	1.9×10^{2}
	12	1.0×10^{-9}	0.6×10^{-10}		12	2.6×10^{3}	1.9×10^{2}
	10	1.0×10^{-9}	0.6×10^{-10}		10	6.9×10^{3}	5.3×10^{2}
	8	2.3×10^{-9}	0.7×10^{-8}		8	6.9×10^{3}	5.3×10^{2}
	6	2.3×10^{-9}	0.7×10^{-10}		6	1.5×10^{4}	0.7×10^{3}
2^{0}	20	2.6×10^{3}	1.9×10^{2}	2^{12}	20	5.4×10^{6}	2.1×10^{4}
	18	3.7×10^{-1}	3.1×10^{-2}		18	1.5×10^{4}	0.8×10^{3}
	16	1.0×10^{-9}	0.6×10^{-10}		16	2.6×10^{3}	1.9×10^{2}
	14	1.0×10^{-9}	0.6×10^{-10}		14	2.6×10^{3}	1.9×10^{2}
	12	1.0×10^{-9}	0.6×10^{-10}		12	6.9×10^{3}	5.3×10^{2}
	10	2.3×10^{-9}	1.6×10^{-10}		10	6.9×10^{3}	5.3×10^{2}
	8	2.3×10^{-9}	1.6×10^{-10}		8	1.5×10^{4}	0.8×10^{3}
	6	4.6×10^{-9}	3.9×10^{-10}		6	1.5×10^{4}	0.8×10^{3}
2^{1}	20	2.6×10^{3}	1.9×10^{2}	2^{14}	20	5.4×10^{6}	2.1×10^{4}
	18	3.7×10^{-1}	3.1×10^{-2}		18	1.5×10^{4}	0.7×10^{3}
	16	1.0×10^{-9}	0.6×10^{-10}		16	6.9×10^{3}	5.3×10^{2}
	14	1.0×10^{-9}	0.6×10^{-10}		14	6.9×10^{3}	5.3×10^{2}
	12	1.0×10^{-9}	0.6×10^{-10}		12	1.5×10^{4}	0.8×10^{3}
	10	2.3×10^{-9}	1.6×10^{-10}		10	1.5×10^{4}	0.8×10^{3}
	8	2.3×10^{-9}	1.6×10^{-10}		8	6.2×10^{4}	4.9×10^{3}
	6	4.6×10^{-9}	3.9×10^{-8}		6	5.4×10^{6}	2.1×10^{4}
2^{2}	20	2.6×10^{3}	1.9×10^{2}	2^{16}	20	5.4×10^{6}	2.1×10^{4}

	18	3.7×10^{-1}	3.9×10^{-2}		18	1.5×10^{4}	0.8×10^{3}
	16	1.0×10^{-9}	0.6×10^{-10}		16	6.9×10^{3}	5.3×10^{2}
	14	2.3×10^{-9}	1.6×10^{-10}		14	1.5×10^{4}	0.8×10^{3}
	12	2.3×10^{-9}	1.6×10^{-10}		12	1.5×10^{4}	0.8×10^{3}
	10	4.6×10^{-9}	3.9×10^{-10}		10	6.2×10^{4}	4.9×10^{3}
	8	4.6×10^{-9}	3.9×10^{-10}		8	6.2×10^{4}	4.9×10^{3}
	6	3.7×10^{-1}	3.1×10^{-2}		6	5.4×10^{6}	2.1×10^{4}
2^{4}	20	2.6×10^{3}	1.9×10^{2}	2^{20}	20	5.4×10^{6}	2.1×10^{4}
	18	3.7×10^{-1}	3.1×10^{-2}		18	1.5×10^{4}	0.8×10^{3}
	16	2.3×10^{-9}	1.6×10^{-10}		16	6.9×10^{3}	5.3×10^{2}
	14	2.3×10^{-9}	1.6×10^{-10}		14	6.9×10^{3}	5.3×10^{2}
	12	4.6×10^{-9}	3.9×10^{-10}		12	1.5×10^{4}	0.8×10^{3}
	10	4.6×10^{-9}	3.9×10^{-10}		10	6.2×10^{4}	6.2×10^{4}
	8	3.7×10^{-1}	3.9×10^{-2}		8	5.4×10^{6}	8.9×10^{4}
	6	2.6×10^{3}	1.9×10^{2}		6	5.7×10^{6}	9.2×10^{4}
Erreur en Virgule flottante double				Max 8.1×10^{-9}		Mean $0,3\times10^{-10}$	
Erreur en Virgule flottante simple				**Max $1,6\times10^{-7}$**		**Mean $6,2\times10^{-8}$**	

À l'opposé, les échelles supérieures, notamment 2^{20} sont associées à des erreurs considérablement plus élevées, avec des valeurs d'erreur atteignant des niveaux alarmants, tels que 5.4×10^{6} pour certaines Wint. Cela souligne une sensibilité accrue des erreurs aux changements dans la langueur de la partie entière, en particulier à mesure que le facteur d'échelle augmente. Ces résultats mettent en évidence la nécessité d'une sélection minutieuse des valeurs de facteur d'échelle et Wint lors de la modélisation. En effet, le choix de l'échelle peut avoir un impact profond sur la qualité des résultats, suggérant qu'une évaluation approfondie des compromis entre nombre de bits utilisé et précision est essentielle. Pour ce faire, nous avons généré une courbe 3D illustrée par la figure 3-9 qui résume le tableau et qui nous aide d'analysé en visuel les résultats.

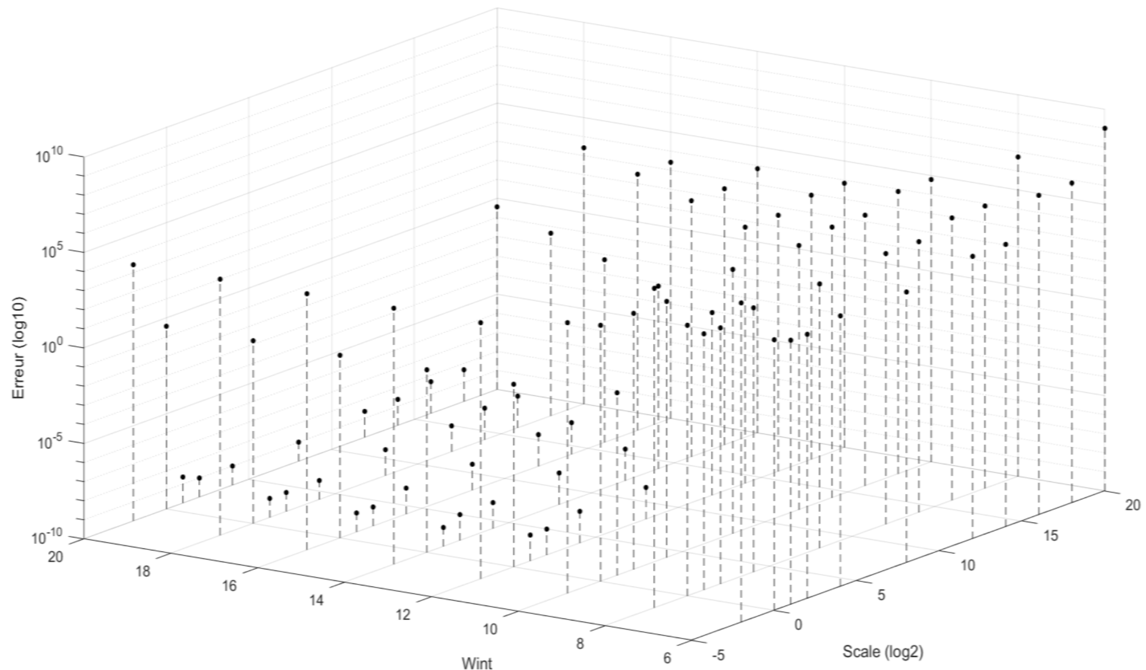

Figure 3-9 Courbe 3D représentant l'erreur maximale en fonction du facteur d'échelle et le Wint pour le 2-L selon le tableau 3-1 pour Wl=90 bits, selon l'équation (3-5).

Les résultats obtenus suggèrent que l'erreur mentionnée est principalement influencée par la longueur en bits de la partie entière du mot binaire. Afin d'approfondir cette observation, une nouvelle série d'expériences a été menée, focalisée sur l'analyse de la partie entière dans la configuration binaire décrite aux figures 3-10 et 3-11. Cette étude systématique vise à quantifier l'influence conjointe de la longueur de la partie entière du mot binaire (Wint) et du facteur d'échelle maximale sur l'erreur absolue, dans le contexte de la représentation en virgule fixe. Pour ce faire, nous avons généré une courbe représentant l'erreur maximale absolue en fonction de la longueur de la partie entière et du facteur d'échelle. Nos paramètres de test pour Wint étaient [6, 8, 10, 12, 14, 16, 18, 20] bits, tandis que nous avons varié le facteur d'échelle sur une échelle logarithmique avec des valeurs de [2^{-2}, 2^{0}, 2^{1}, 2^{2}, 2^{4}, 2^{8}, 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}]. Et cette approche nous a permis d'explorer une gamme étendue de configurations pour la représentation en virgule fixe.

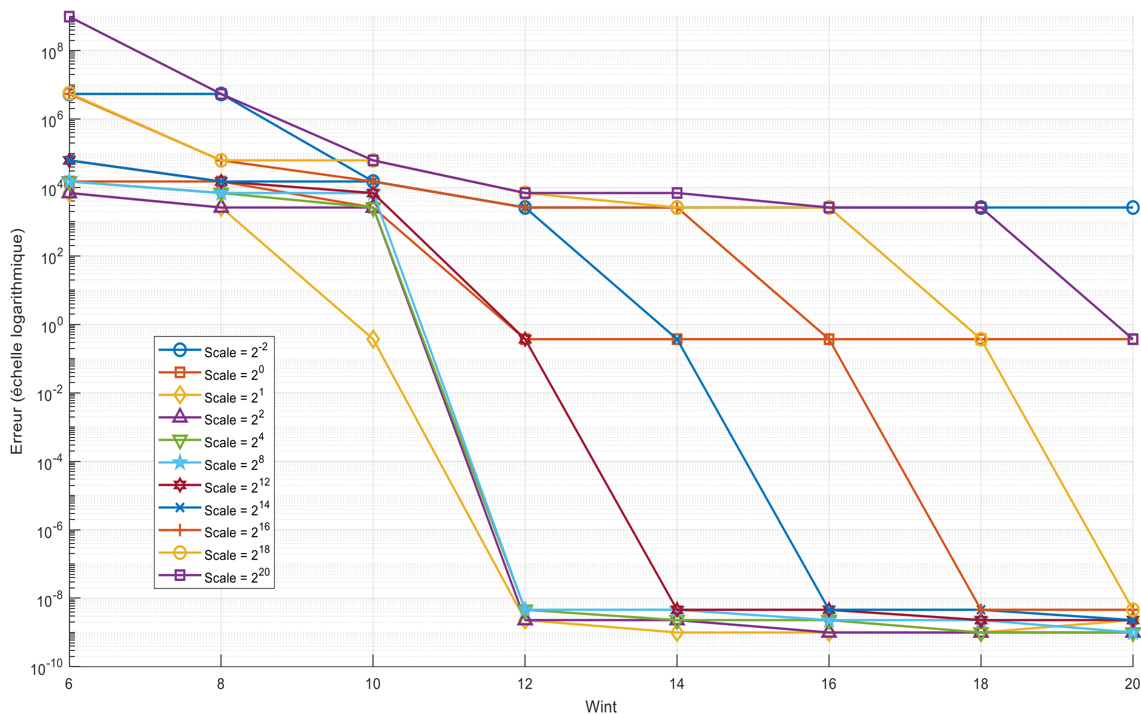

Figure 3-10 Erreur Max en fonction de la longueur de la partie entière et le facteur d'échelle pour le 2-L selon le tableau 3-1 pour Wl=90 bits, selon l’équation (3-5).

En analysant les deux courbes résultantes, nous pouvons visualiser comment l'erreur maximale absolue varie en fonction de ces deux paramètres clés, ce qui confirme que l’erreur présentent une tendance à diminuer lorsque l'on utilise des valeurs de facteur d’échelle inferieurs, Une telle analyse fournit des informations cruciales pour comprendre comment la précision de la représentation en virgule fixe est influencée par le choix de la longueur de la partie entière et du facteur d'échelle.

Le tableau 3-2 présente les erreurs moyennes et maximales pour différentes configurations binaires et facteurs d'échelle pour les matrices du convertisseur B2B (*Back-to-Back*) spécifique. Dans chaque configuration, la longueur de la partie entière (Wint), la longueur du mot binaire (Wl) sont variées et la longueur totale du mot binaire (Wl) est fixée à 90 bits les résultats montrent que le choix du facteur d'échelle influence considérablement les performances du système, tant en termes d'erreur moyenne que maximale.

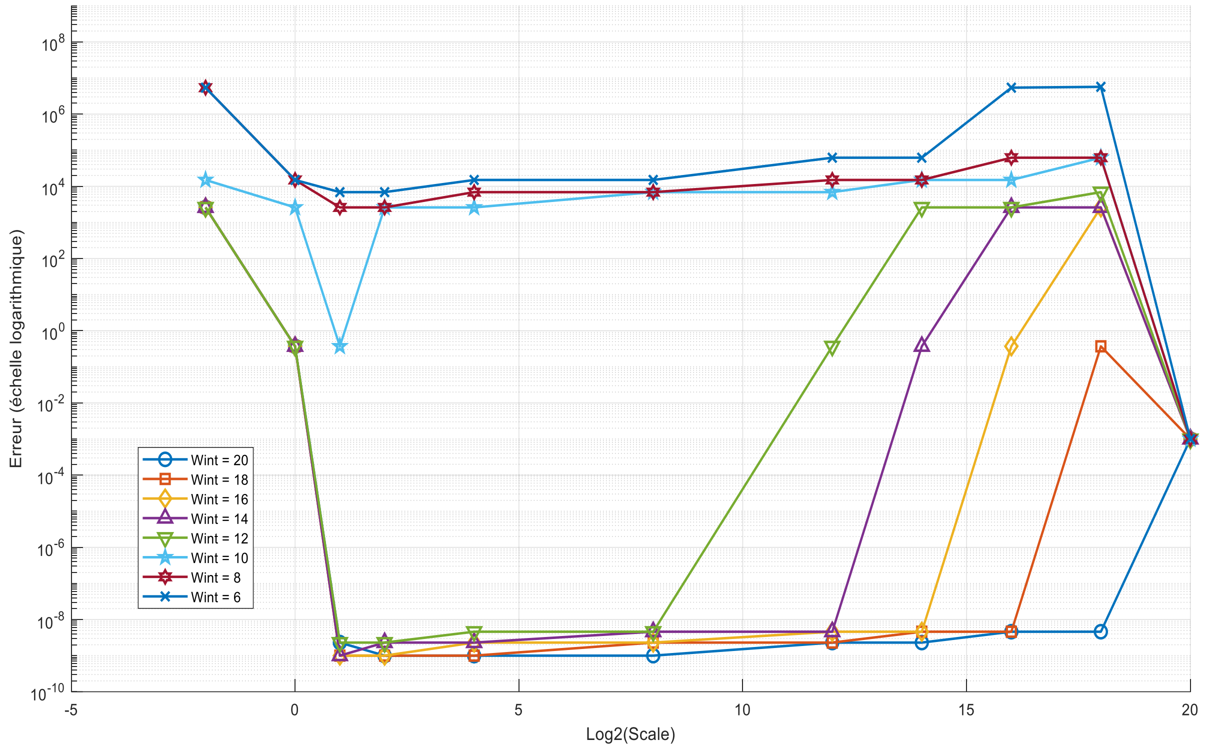

Figure 3-11 Erreur Max en fonction du facteur d'échelle et de la longueur de la partie entière et pour le 2-L selon le tableau 3-1 pour Wl=90 bits, selon l'équation (3-5).

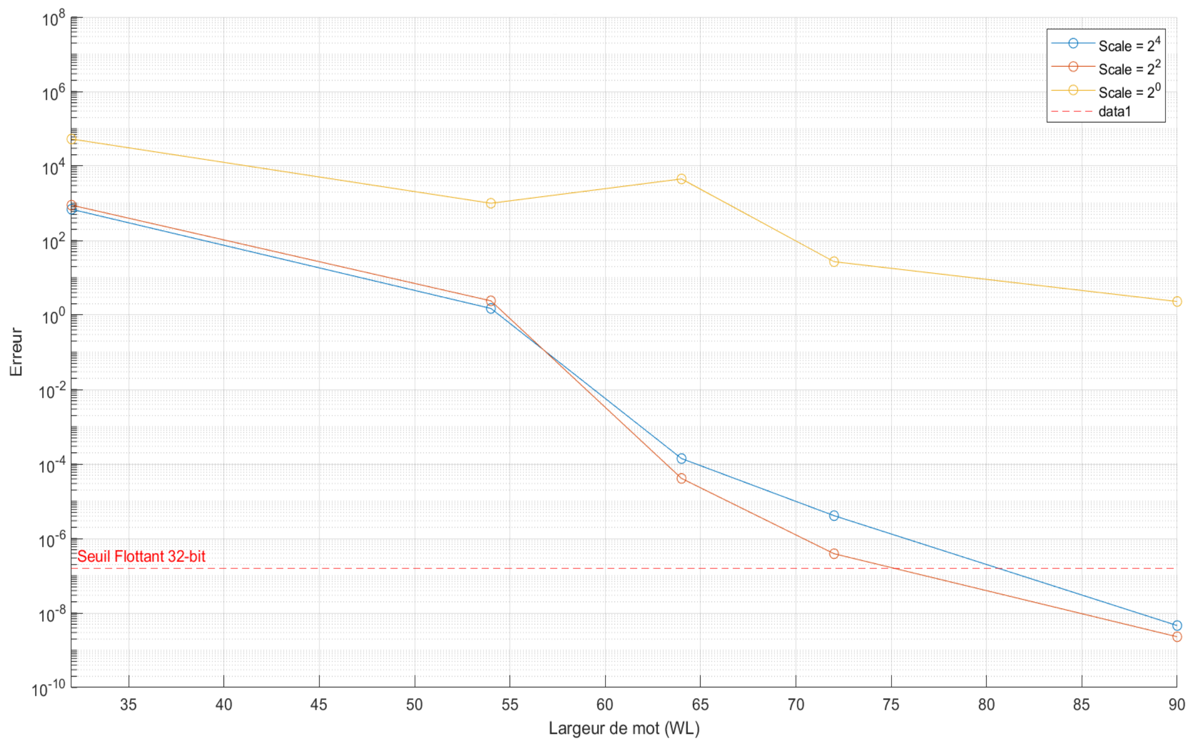

Figure 3-12 Erreur Max en fonction du facteur d'échelle et de la longueur de la partie entière pour le 2L pour Wint=12 bits, selon l'équation (3-5).

Analyse de précision du B2B

Le tableau 3-2 présente les erreurs moyennes et maximales pour différentes configurations binaires et facteurs d'échelle pour les matrices du convertisseur B2B (*Back-to-Back*) spécifique. Dans chaque configuration, la longueur de la partie entière (Wint), la longueur du mot binaire (Wl) sont variées et la longueur totale du mot binaire (Wl) est fixée à 90 bits les résultats montrent que le choix du facteur d'échelle influence considérablement les performances du système, tant en termes d'erreur moyenne que maximale.

Tableau 3-2 Erreurs selon (3-5) et (3-6) en fonction de la configuration binaire et le facteur d’échelle pour le B2B pour Wl fixe de 126 bits.

Facteur d’échelle (β)	**Wint pour Wl 90 bits**	**Erreur max**	**Erreur Mean**	**Facteur d’échelle** (β)	**Wint pour Wl 90 bits**	**Erreur max**	**Erreur Mean**
2^{-2}	20	4.5×10^{-2}	2.3×10^{-3}	2^{4}	20	3.2×10^{-2}	1.2×10^{-3}
	18	4.5×10^{-2}	2.3×10^{-3}		18	3.2×10^{-2}	1.2×10^{-3}
	16	4.5×10^{-2}	2.3×10^{-3}		16	3.2×10^{-2}	1.2×10^{-3}
	14	4.5×10^{-2}	2.3×10^{-3}		14	3.2×10^{-2}	1.2×10^{-3}
	12	3.2×10^{-2}	1.2×10^{-3}		12	3.2×10^{-2}	1.2×10^{-3}
	10	8.9×10^{2}	7.2×10^{3}		10	8.9×10^{2}	7.2×10^{1}
	8	2.1×10^{2}	7.4×10^{3}		8	8.9×10^{2}	7.2×10^{1}
	6	9.7×10^{4}	1.5×10^{3}		6	9.7×10^{4}	1.5×10^{3}
2^{0}	20	1.1×10^{-2}	5.3×10^{-3}	2^{8}	20	3.2×10^{-2}	1.2×10^{-3}
	18	1.1×10^{-2}	5.3×10^{-3}		18	3.2×10^{-2}	1.2×10^{-3}
	16	1.1×10^{-2}	5.3×10^{-3}		16	3.2×10^{-2}	1.2×10^{-3}
	14	1.1×10^{-2}	5.3×10^{-3}		14	3.2×10^{-2}	1.2×10^{-3}
	12	1.1×10^{-2}	5.3×10^{-3}		12	7.1×10^{1}	6.5×10^{-1}
	10	4.2×10^{3}	9.5×10^{2}		10	4.5×10^{2}	3.4×10^{1}
	8	9.3×10^{3}	5.8×10^{2}		8	3.2×10^{2}	7.8×10^{1}
	6	3.4×10^{4}	1.9×10^{3}		6	9.7×10^{2}	1.5×10^{1}
2^{1}	20	1.3×10^{-4}	2.9×10^{-5}	2^{14}	20	3.2×10^{-2}	1.2×10^{-1}

<table>
<tr><td rowspan="7"></td><td>18</td><td>1.3×10^{-4}</td><td>2.9×10^{-5}</td><td rowspan="7"></td><td>18</td><td>3.2×10^{-2}</td><td>1.2×10^{-1}</td></tr>
<tr><td>16</td><td>3.2×10^{-4}</td><td>1.2×10^{-5}</td><td>16</td><td>2.1×10^{-1}</td><td>9.8×10^{-1}</td></tr>
<tr><td>14</td><td>1.3×10^{-4}</td><td>2.9×10^{-5}</td><td>14</td><td>3.1×10^{1}</td><td>9.5×10^{0}</td></tr>
<tr><td>12</td><td>1.3×10^{-4}</td><td>2.9×10^{-5}</td><td>12</td><td>3.1×10^{1}</td><td>9.5×10^{0}</td></tr>
<tr><td>10</td><td>3.2×10^{-2}</td><td>7.8×10^{-3}</td><td>10</td><td>7.8×10^{2}</td><td>8.5×10^{1}</td></tr>
<tr><td>8</td><td>3.9×10^{1}</td><td>2.6×10^{0}</td><td>8</td><td>6.1×10^{2}</td><td>3.4×10^{1}</td></tr>
<tr><td>6</td><td>5.2×10^{2}</td><td>6.9×10^{1}</td><td>6</td><td>9.0×10^{2}</td><td>6.2×10^{1}</td></tr>
<tr><td rowspan="8">2^{2}</td><td>20</td><td>3.2×10^{-2}</td><td>1.2×10^{-3}</td><td rowspan="3">2^{18}</td><td>20</td><td>0.9×10^{-2}</td><td>3.6×10^{-2}</td></tr>
<tr><td>18</td><td>3.2×10^{-2}</td><td>1.2×10^{-3}</td><td>18</td><td>5.3×10^{-1}</td><td>9.8×10^{-2}</td></tr>
<tr><td>16</td><td>3.2×10^{-2}</td><td>1.2×10^{-3}</td><td>16</td><td>7.4×10^{-1}</td><td>9.5×10^{-1}</td></tr>
<tr><td>14</td><td>3.2×10^{-2}</td><td>1.2×10^{-3}</td><td rowspan="5"></td><td>14</td><td>8.9×10^{1}</td><td>7.2×10^{-1}</td></tr>
<tr><td>12</td><td>3.2×10^{-2}</td><td>1.2×10^{-3}</td><td>12</td><td>8.9×10^{1}</td><td>7.2×10^{-1}</td></tr>
<tr><td>10</td><td>7.1×10^{2}</td><td>4.5×10^{1}</td><td>10</td><td>1.1×10^{2}</td><td>9.1×10^{0}</td></tr>
<tr><td>8</td><td>3.2×10^{2}</td><td>7.8×10^{1}</td><td>8</td><td>1.0×10^{2}</td><td>9.3×10^{0}</td></tr>
<tr><td>6</td><td>5.2×10^{3}</td><td>6.9×10^{2}</td><td>6</td><td>8.9×10^{2}</td><td>7.3×10^{0}</td></tr>
<tr><td colspan="3">Erreur en Virgule flottante double</td><td colspan="2">Max 6.4×10^{-7}</td><td colspan="3">Mean 1.2×10^{-6}</td></tr>
<tr><td colspan="3">Erreur en Virgule flottante simple</td><td colspan="2">Max $2,6\times10^{-4}$</td><td colspan="3">Mean $2,6\times10^{-5}$</td></tr>
</table>

L'analyse du tableau des erreurs en fonction des paramètres révèle une relation complexe influencée par la précision numérique de calcul, ainsi présenté par la figure 3-13, une courbe 3D, offre une représentation visuelle de l'erreur maximale absolue en fonction du facteur d'échelle et de la longueur de la partie entière. Cette visualisation permet d'identifier les tendances générales de l'erreur en fonction de ces deux paramètres.

Globalement, on observe que les petites valeurs relatives de facteur d'échelle par exemple, 2^{-2} à 2^{2} entraînent des erreurs relativement faibles, généralement de l'ordre de 10^{0} à 10^{1}, indiquant une précision accrue dans cette plage de mise à l'échelle. À mesure que le facteur d'échelle augmente particulièrement 2^{14} et 2^{18} les erreurs s'amplifient considérablement, atteignant des valeurs élevées jusqu'à 10^{5}, particulièrement pour des valeurs réduites de Wint.

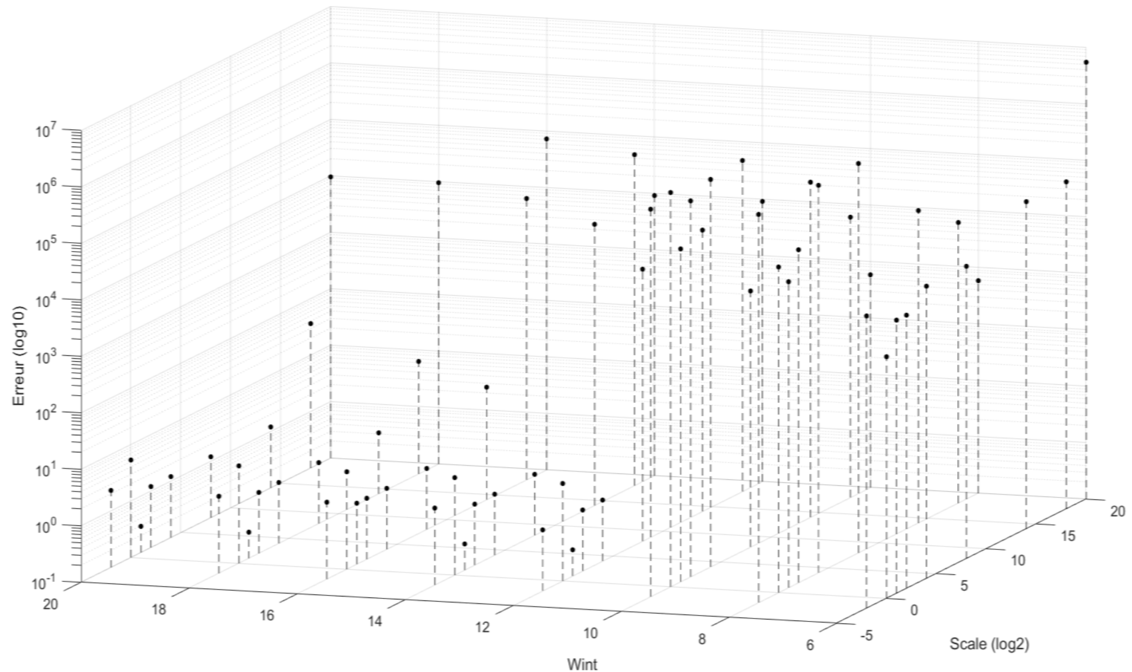

Figure 3-13 Courbe 3D représentant l'erreur maximale en fonction du facteur d'échelle et le Wint pour le B2B selon le tableau 3-2 pour Wl=126 bits, selon l'équation (3-5).

Ce phénomène peut être attribué à une amplification des erreurs d'arrondi ou à une instabilité numérique accrue à des échelles plus élevées vu la plage dynamique étendu des valeurs des matrices d'impédance étudiée, qui exacerbent la perte de précision lors des opérations sur les données. Les deux figures 3-14 et 3-15 présentent une analyse de l'erreur absolue dans le calcul des inverses des matrices du convertisseur B2B, en tenant compte à la fois de la longueur de la partie entière et du facteur d'échelle offrant une perspective supplémentaire sur l'impact spécifique du facteur d'échelle sur l'erreur. En combinant les informations des deux figures, des valeurs élevées de Wint, supérieures ou égales à 18, tendent à maintenir des erreurs faibles même pour des valeurs modérées de facteur d'échelle, mais l'association de valeurs faibles de Wint, par exemple, 6 et 8, avec des valeurs de facteur d'échelle très élevées produit les erreurs maximales suggérant une précision accrue dans ces conditions extrêmes. En conséquence, pour assurer une précision optimale des résultats, il est conseillé d'utiliser

des valeurs de facteur d'échelle limitées à un intervalle modéré et d'éviter les très petites valeurs de Wint, surtout lorsqu'elles sont combinées avec des mises à l'échelle élevées.

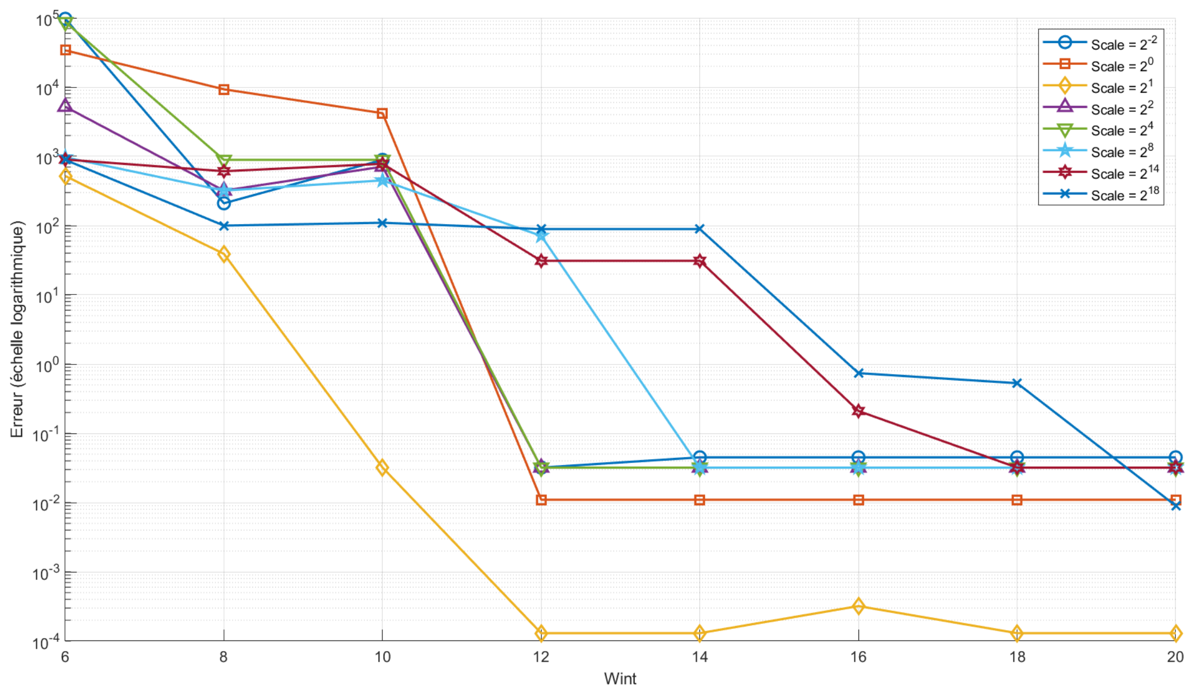

Figure 3-14 Erreur Max en fonction de la longueur de la partie entière et le facteur d'échelle pour le B2B selon le tableau 3-2 pour Wl=126 bits, selon l'équation (3-5).

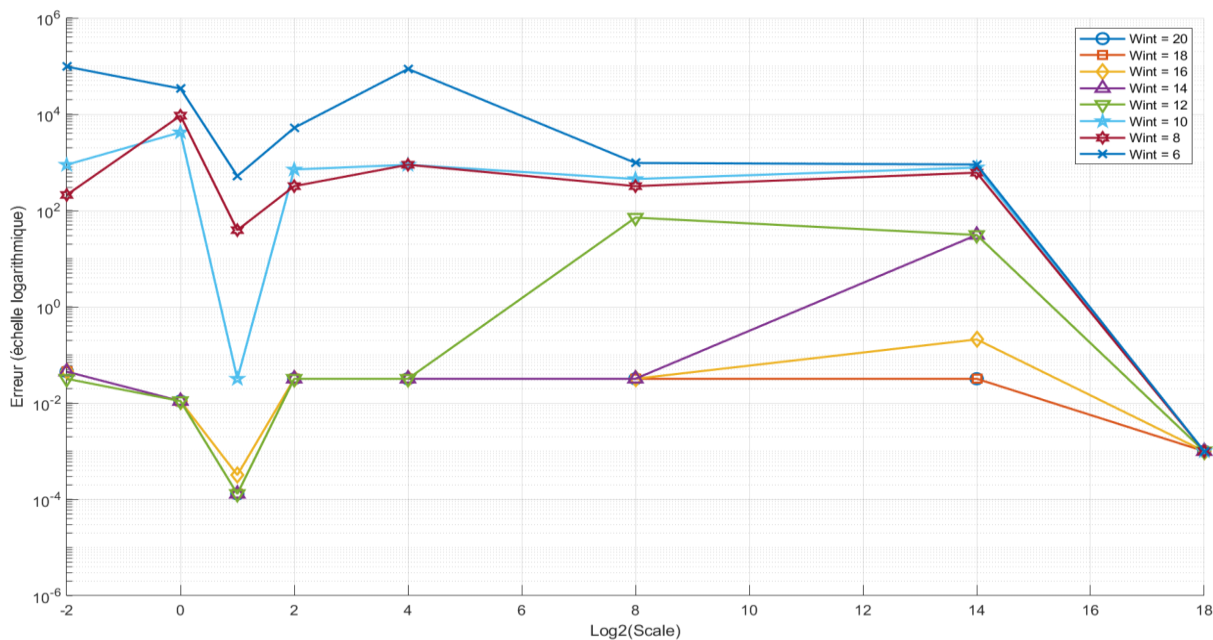

Figure 3-15 Erreur Max en fonction du facteur d'échelle et de la longueur de la partie entière pour le B2B selon le tableau 3-2 pour Wl=126 bits, selon l'équation (3-5).

La figure 3-16 montre que l'erreur de calcul de l'inverse avec FSM pour Les résultats indiquent que, pour la matrice B2B, l'erreur de calcul de l'inverse utilisant la méthode FSM diminue à mesure que le facteur d'échelle augmente, particulièrement pour les largeurs de mot élevées. Pour garantir une précision élevée, il est recommandé d'utiliser une largeur de mot de 126 bits, une partie entière de 12 bits et un facteur de mise à l'échelle de 2^1. Cette configuration permet d'approcher une précision en virgule fixe proche de celle de la virgule flottante. En revanche, des facteurs d'échelle plus faibles induisent des erreurs significatives, notamment l'efficacité des représentations binaires fixes. Cela met en lumière la nécessité d'ajuster avec précision les paramètres de quantification (Wl et facteur d'échelle) pour un compromis optimal entre la précision et les ressources, telles que la mémoire et la complexité de calcul, dans des systèmes à capacité limitée.

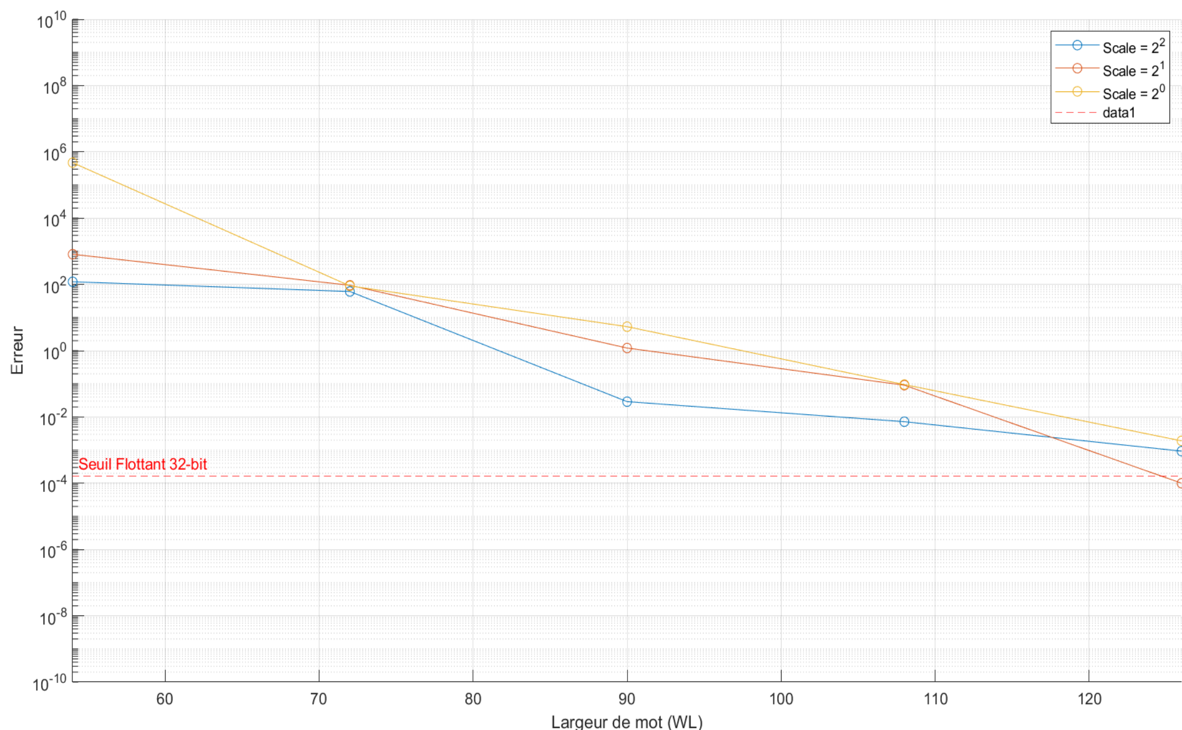

Figure 3-16 Erreur Max en fonction du facteur d'échelle et de la longueur de la partie entière pour le B2B pour Wint=12 bits, selon l'équation (3-5).

Les résultats présentés montrent qu'après plusieurs tests de configurations binaires pour le 2-L et le B2B, y compris l'étude approfondie de la mise en échelle on se retrouve avec un nombre très élevé de bits de calcul en virgule fixe et une faible précision de calcul et Pour atteindre une précision d'erreur la plus proche de celle en virgule flottante dans le cas du convertisseur 2L, il est nécessaire d'utiliser une largeur de mot (Wl) de 72 bits, une partie entière (Wint) de 12 bits et un facteur de mise à l'échelle de 2^2. En revanche, pour le convertisseur B2B, la configuration optimale pour obtenir une précision similaire en termes d'erreur requiert une largeur de mot de 126 bits, une partie entière de 12 bits et un facteur de mise à l'échelle de 2^1.

3.4 Résultats de synthèse de la FSM

Les résultats de synthèse de la FSM mettent en lumière plusieurs aspects cruciaux de son implémentation et de ses performances il est également essentiel de mentionner que les performances du codage en virgule fixe de la FSM sont comparées avec celles de la virgule flottante à double précision ce qui permet de bien comprendre l'impact de la représentation numérique sur la précision des calculs et d'évaluer la performance de l'approche en virgule fixe dans le contexte spécifique de cette application.

Les figures 3-17 et 3-18 présentent les résultats d'implémentation de la FSM bien décrit dans la section 3-2 les graphiques montrent l'erreur absolue entre l'inverse matriciel exact et l'inverse calculée selon la FSM en virgule fixe pour diverses configurations de mot binaires. Le calcule d'erreur est fait avec l'équation (3-4) Pour ces tests, la longueur de la partie entière est fixée à 18 bits, ce qui correspond au nombre de bits les multiplicateurs à l'intérieur du DSP48 du FPGA utilisé. Les DSP48 (*Digital Signal Processing Blocks*) sont des blocs matériels spécialisés présents sur les FPGAs, jouant un rôle clé dans le traitement numérique

du signal et offrant des performances optimales pour des opérations arithmétiques telles que les multiplications. La longueur de la partie entière a été choisie ainsi afin d'obtenir une meilleure précision lors de l'approximation.

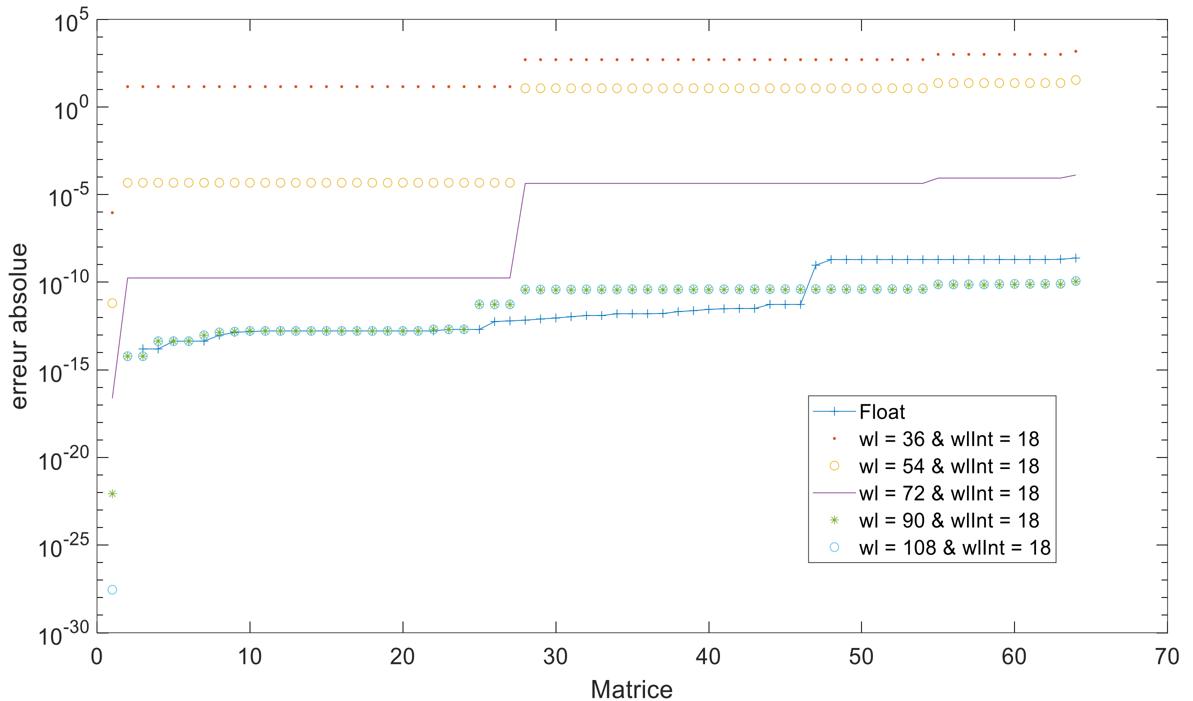

Figure 3-17 Résultats de FSM pour le 2-L pour les configurations différentes (Wl, Wint): (36,18), (54,18), (72, 18), (90, 18) et (108, 18) bits, erreur Max selon l'équation (3-4).

Pour étudier l'impact de la longueur totale du mot binaire, la WordLength a été modifiée par paliers de 18 bits. Les résultats montrent que la meilleure résolution est obtenue pour une WordLength de 72 bits et une partie entière de 18 pour le système à deux niveaux, alors que pour le système B2B, la configuration optimale est une WordLength de 90 bits avec une partie entière de 18. Cela souligne l'importance de trouver un compromis entre la précision souhaitée et le nombre de bits utilisés, l'idée est de se retrouver avec une précision de calcul optimale similaire à celle obtenue en virgule flottante.

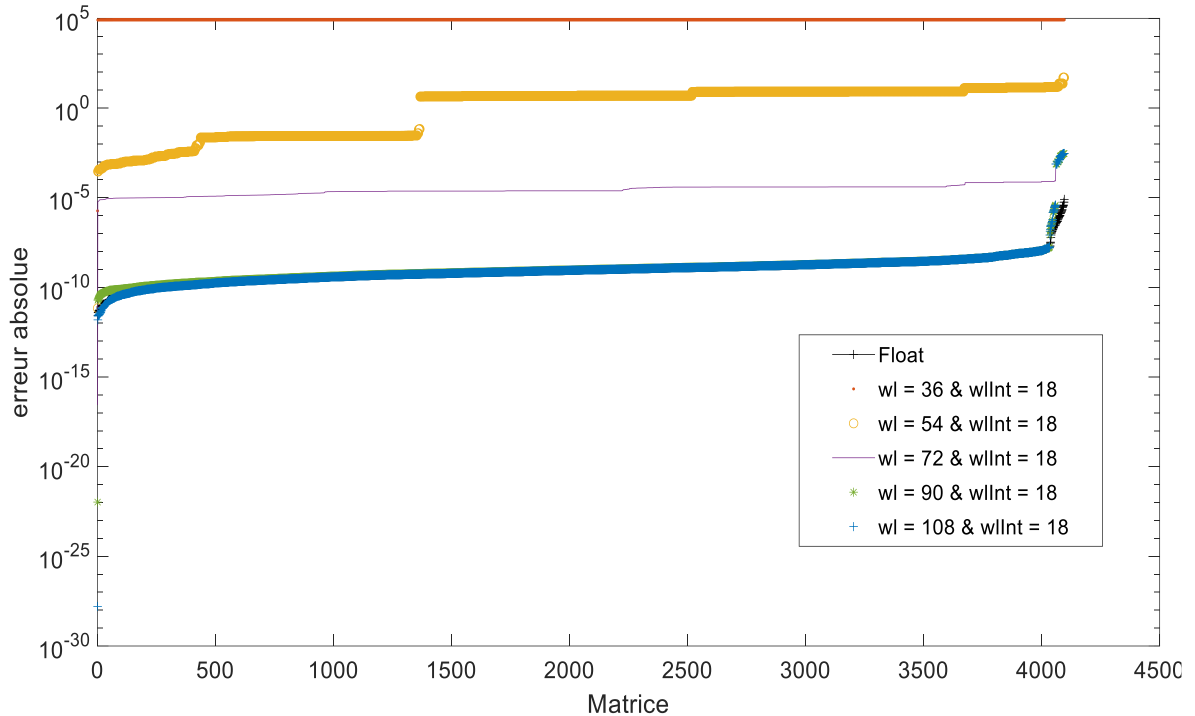

Figure 3-18 Résultats de FSM pour le B2B pour les configurations différentes (Wl, Wint): (36, 18), (54,18), (72, 18), (90, 18) et (108, 18) bits, erreur Max selon l'équation (3-4).

3.5 Étude comparative d'un cas de simulation d'un 2-L

Cette partie s'attache à comparer les performances des calculs en virgule flottante et fixe pour les courants triphasés, et à évaluer les seuils d'erreur acceptables associés. Cela est crucial pour voir l'effet de l'erreur présenté par les figures précédentes sur un cas réel d'application et permet d'optimiser la méthode de calcul en fonction des contraintes du projet.

Ces simulations visent à évaluer l'exactitude des calculs numériques dans un cas de simulation d'un circuit temps réel. Une simulation réalisée avec Simulink est utilisée pour calculer les trois courants triphasés. Cette simulation permet de comparer les erreurs liées au calcul de ces courants à l'aide de trois méthodes différentes pour l'inversion de matrices. Trois méthodes de calcul d'inverse sont utilisées ; Double: l'inverse est calculé par MATLAB en

double précision (virgule flottante 64 bits), FSM Simple: matrice inverse calculée avec la FSM en virgule flottante simple 32 bits et finalement FSM Fix: matrice inverse calculée avec la FSM en virgule fixe 32 bits.

Il s’agit de l’onduleur 2-L de la figure 3-1, dont la valeur de Ron est 1×10^{-3} Ohms, Roff est 1×10^{9} Ohms, l'inductance L est 5×10^{-3} H, et la charge est RL en série dont R est 1×10^{-2} Ohms, et L est 5×10^{-3} H. chaque sortie de courant est connectée à une charge RL série distincte. Les trois charges RL série sont indépendantes et reliées en parallèle aux sorties des trois phases. De plus, la tension d'entrée utilisée pour les simulations est de 15V. Le circuit Simulink est structuré autour de plusieurs éléments principaux. Les entrées génèrent des signaux triphasés simulés (tension et courant), servant à alimenter la matrice à inverser. Ces sources permettent de modéliser des scénarios réalistes, comme des tensions ou courants alternatifs triphasés. Le calcul de l'inverse de matrices est assuré par trois sous-systèmes distincts, chacun correspondant à une méthode spécifique : la méthode en double précision MATLAB (64 bits), utilisée comme référence pour sa précision maximale ; la méthode FSM en virgule flottante simple (32 bits) et la méthode FSM en virgule fixe (32 bits). Une fois les matrices inversées, les systèmes résolvent des équations pour calculer les courants triphasés en temps réel, simulant un cas concret d'application.

La figure 3-19 met en évidence les performances des différentes méthodes d’inversion. Pour le courant I1, les courbes obtenues avec la méthode Double (en bleu) et la méthode FSM Simple (en vert) sont pratiquement identiques, démontrant la grande précision de la méthode FSM Simple par rapport à la référence. En revanche, la courbe rouge, correspondant à la méthode FSM Fixe, reste proche des autres, mais présente de légères ondulations supplémentaires autour des pics et des creux, traduisant une précision légèrement inférieure

en virgule fixe. Concernant le courant I2, les méthodes Double et FSM Simple continuent de fournir des résultats très bien alignés, ce qui confirme la capacité de la méthode FSM Simple à reproduire fidèlement les résultats en double précision. Cependant, la méthode FSM Fixe montre des écarts un peu plus visibles, particulièrement dans les variations rapides du signal, bien que ces différences restent modérées. Enfin, pour le courant I3, les courbes bleue et verte restent étroitement alignées, confirmant une fois de plus la fiabilité de la méthode FSM Simple. La courbe rouge, quant à elle, révèle des écarts plus prononcés dans les zones de transition rapide, mais elle suit globalement la même tendance que les deux autres méthodes. Cependant, la méthode FSM Fixe continue de révéler des écarts visibles, en particulier dans les zones de transition rapide, où elle semble moins performante. Les écarts introduits par la virgule fixe sont plus notables notamment autour des points d'inflexion des sinusoïdes, mais restent modérés, particulièrement dans la partie centrale des sinusoïdes, confirmant que cette méthode demeure acceptable malgré une précision légèrement inférieure.

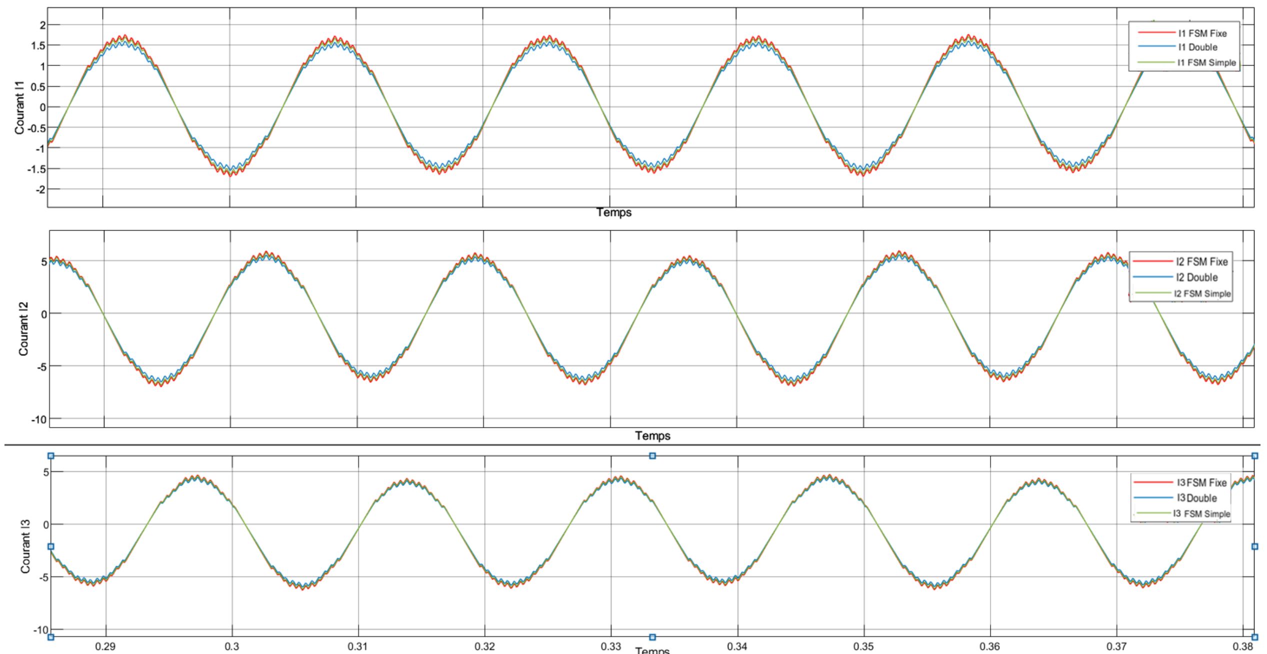

Figure 3-19 Comparaison des méthodes d'inversion de matrices pour l'estimation des courant triphasés I1, I2 et I3

Le tableau 3-3 compare les erreurs RMSE (3-7) et RRMSE (en pourcentage) (3-8) entre deux configurations binaires de SM, virgule flottante simple et virgule fixe. Il en ressort que le FSM virgule flottante simple offre une meilleure précision globale pour un nombre N de 2×10^4 points calculés.

L'analyse des résultats met en évidence une différence notable entre les méthodes FSM Simple et FSM Fixe en termes de précision et de performance. Pour I1, la méthode FSM en virgule flottante simple donne un RMSE de 0,0724 (6,5 % en RRMSE), nettement inférieur au RMSE de 0,11 (10 % en RRMSE) obtenu avec la virgule fixe. Des tendances similaires sont observées pour I2 et I3, où la virgule flottante simple présente des RMSE respectifs de 0,18 (4 %) et 0,11 (3,2 %), contre des valeurs plus élevées avec la virgule fixe (0,28 et 0,18 avec des RRMSE de 7 % et 5 %).

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_{i^{Réel}} - y_{i^{Approx}})^2} \tag{3-7}$$

Et

$$RRMSE = \frac{RMSE}{\sqrt{Mean({y_{i}}_{Réel}{}^2)}} \times 100 \tag{3-8}$$

Dans les deux configurations, le courant admet des erreurs modérées. Cependant, dans un cas réel d'application, la méthode en virgule fixe permet de maintenir des erreurs relativement acceptables, ce qui en fait une option viable et pertinente lorsque des compromis entre précision et économie de ressources sont nécessaires, en particulier dans des environnements à fortes contraintes matérielles comme les FPGA.

Tableau 3-3 Erreurs calculées pour trois courants I1, I2 et I3 à partir des résultats de simulation en utilisant SM avec deux configurations binaires : virgule flottante simple et virgule fixe.

Courant	Méthodes	RMSE	RRMSE (%)
I1	FSM virgule flottante simple	0.0724	6.5
	FSM virgule Fixe 32 bits	0.11	10
I2	FSM virgule flottante simple	0.18	4
	FSM virgule Fixe 32 bits	0.28	7
I3	FSM virgule flottante simple	0.11	3.2
	FSM virgule Fixe 32 bits	0.18	5

3.6 Conclusion

Dans ce chapitre, nous avons exploré divers aspects liés à la représentation en virgule fixe et à la correction des erreurs de calcul associées. Dans un premier temps, nous avons revisité la FSM en reformulant ses équations pour faciliter l'analyse de la dynamique des calculs. Ces étapes nous ont permis de mieux comprendre la structure des opérations au sein de la FSM, en particulier les mécanismes de propagation et d'accumulation des erreurs lors des opérations matricielles. Cette compréhension a ensuite été renforcée par l'introduction de la mise à l'échelle pour réduire l'erreur de calcul et optimiser la précision des calculs tout en tenant compte des limitations en ressources matérielles.

La conversion des calculs en virgule fixe nécessite une attention particulière afin de minimiser la perte de précision. Nous avons ainsi examiné les performances de la FSM dans ce contexte, en tenant compte des contraintes en termes de largeur des mots binaires et des

ressources matérielles disponibles. Nous avons démontré qu'il est très coûteux en termes de longueur binaire en virgule fixe pour assurer une performance de l'estimation des matrices via FSM comparativement à la virgule flottante simple 32 bits.

Bien que la méthode de FSM n'ait pas démontré une précision totalement satisfaisante à 32 bits à virgule fixe dans l'analyse théorique de la totalité des matrices, elle a montré, dans un cas pratique de simulation des courants triphasés d'un convertisseur 2-L, une erreur relative acceptable. Cela en fait une solution intéressante pour les applications normales de simulations d'un convertisseur, particulièrement dans des contextes où l'optimisation des ressources et des pas de calculs sont prioritaires.

Dans la suite de l'étude, nous allons viser l'implémentation sur FPGA en comparant le calcul en virgule flottante simple (32 bits) versus une implémentation en virgule fixe de même longueur de 32 bits. Ceci afin de démontrer les latences et ressources sachant que 32 bits à virgule fixe est insuffisant en termes de précision dans l'application 2-L et B2B.

Dans le prochain chapitre, nous allons comparer les performances en implémentation FPGA de la FSM entre une implémentation utilisant l'outil SysGen et l'outil HLS.

Chapitre 4 - FSM sur FPGA – implémentation

Après avoir étudié les performances de FSM et corrigé l'erreur de quantification on s'intéresse à l'implémentation sur des circuits FPGA, on met l'emphase sur les techniques d'ordonnancement des calculs et les outils de conception matériels HLS et SysGen. Ce chapitre explore les différentes approches pour traduire efficacement FSM en une implémentation matérielle. Les objectifs sont de fournir des solutions matérielles rapides et efficaces pour l'inverse matriciel, tout en offrant une compréhension détaillée des processus et des avantages spécifiques de chaque approche.

4.1 Introduction

L'implémentation de FSM sur FPGA via des outils comme le System Generator et HLS (*High-Level Synthesis*) constitue un progrès significatif dans la conception de systèmes matériels destinés au calcul de matrices. Cette méthode permet de concevoir des architectures matérielles efficaces en exploitant la parallélisation et l'optimisation des opérations, tout en intégrant des innovations technologiques telles que la création de nouveaux modèles basés sur le principe du processeur élémentaire. La combinaison de cette puissante technique avec les capacités massives de calcul parallèle proposées par les FPGA vise à offrir des solutions matérielles extrêmement performantes répondant à une variété d'applications nécessitant des calculs intensifs de matrices. Ce chapitre commencera par une introduction générale, présentant l'importance de l'implémentation matérielle de FSM sur FPGA, notamment pour les applications nécessitant des calculs matriciels rapides et efficaces. Nous aborderons

ensuite les différentes stratégies d’ordonnancement des calculs, qui sont essentielles à la conception de circuits matériels optimisés, Nous aborderons ensuite deux approches spécifiques pour implanter le FSM sur FPGA. L’une est l’approche HLS et l’autre est l’approche SysGen, une méthode basée sur des outils de conception matérielle, il présente l’architecture, le processus d’ordonnancement des calculs et les avantages spécifiques de chacune. Pour l'implémentation de FSM sur FPGA, nous avons opté pour une approche comparative en utilisant deux outils principaux : Vivado HLS et System Generator pour DSP. Cette décision découle de la nécessité d'évaluer les compromis entre deux approches distinctes : d'une part, Vivado HLS offre une mise en œuvre rapide et polyvalente, mais avec moins de contrôle sur l'architecture synthétisée ; d'autre part, System Generator permet de spécifier une architecture sur mesure, mais nécessite un temps de développement plus long. Cette comparaison nous permettra de mieux comprendre les avantages et les inconvénients de chaque approche en termes de performances, de flexibilité et de complexité de conception, et ainsi de prendre des décisions éclairées pour l'implémentation finale sur FPGA.

4.2 Implémentation HLS

Vivado HLS est un outil de synthèse de haut niveau qui permet de convertir du code C en une implémentation RTL (*Register Transfer Level*) et de réaliser l’évaluation des ressources et du timing pour une conception FPGA. Pour notre projet, nous avons débuté en implémentant de FSM avec Vivado HLS et du code C++. Cette méthode nous a permis d’obtenir une mise en œuvre rapide grâce au cadre préétabli et au code aisément modifiable pour tester divers paramètres. Parmi les avantages de Vivado HLS, notons la possibilité de développer l’algorithme dans un langage C facile à modifier. Nous pouvons aussi effectuer des tests rapides grâce aux banques d’essais en C. En partant du code C, il est également

possible d'indiquer certaines directives pour orienter la conception de l'implémentation, ce qui permet des optimisations telles qu'un pipeline ou un déroulement de boucles. En comparaison avec System Generator (SysGen), où les optimisations doivent être prises en compte dès la conception, Vivado HLS permet de comparer rapidement différentes solutions d'optimisation.

4.2.1 Tests préliminaires d'implémentations

Les tests d'implantation préliminaires sur FPGA sont cruciaux pour vérifier et valider le bon fonctionnement de conceptions matérielles sur des dispositifs réels notamment pour savoir la latence et le hardware utilisé pour les opérations élémentaires dans le calcul vectoriel et matriciel et pour ce faire nous avons utilisé deux principaux outils d'implantation dans notre processus :

Xilinx Vivado : cet outil est essentiel pour concevoir des circuits destinés aux FPGA de Xilinx. Il propose une suite complète de fonctionnalités pour la synthèse sur FPGA, l'implémentation et la vérification sur les différentes familles de puces Xilinx. Vivado est largement utilisé pour créer et optimiser des conceptions matérielles pour une grande variété d'applications.

Vivado HLS (*High-Level Synthesis*) : Vivado HLS permet une synthèse de haut niveau à partir du code source en langages C, C++ ou SystemC. Contrairement à une implémentation RTL traditionnelle, Vivado HLS génère automatiquement une implémentation RTL à partir du code source, offrant ainsi une conception plus rapide et optimale. Il permet également de créer plusieurs implémentations à partir du même code source, ce qui facilite l'exploration des différentes architectures matérielles.Tous les résultats des tests ont été implémentés sur

le Virtex®-7 de Xilinx, avec une fréquence d'horloge de 100 MHz, utilisant à la fois un format à virgule fixe et à virgule flottante simple précision. L'implémentation en virgule fixe est indispensable dans de nombreuses applications en raison de contraintes de précision, notamment lorsque la matrice est mal conditionnée, comme c'est souvent le cas dans les modèles d'électronique de puissance utilisés dans le matériel dans la boucle (HIL), nécessitant une latence en virgule fixe, les calculs sont effectués avec des nombres réels, et les résultats sont validés en les comparant avec l'implémentation Matlab de SM.

Les tableaux 4-1 et 4-2 présentent les résultats des tests d'implémentation d'un multiplicateur et d'un additionneur en VHDL sur FPGA. À la suite des tests de précision de FSM, nous nous concentrons sur la configuration binaire Q16.32 pour les 2-L. Une comparaison entre les résultats des tests et les calculs théoriques révèle une différence de seulement 1 DSP entre les résultats de synthèse et la théorie, l'aspect théorique sera détaillé par la suite et cela indique une bonne concordance entre les résultats obtenus sur la plateforme FPGA et les prévisions théoriques, ce qui renforce la validité et la fiabilité de l'implémentation matérielle de l'algorithme.

Tableau 4-1 Test additionneur virgule fixe en HLS sur le Kintex-7 de Xilinx.

Configuration binaire			**Latences**		**Ressources**	
(Wint)	(Wlf)	(Wl)	Nbre de cycle	Clock (MHz)	FF	LUT
16	16	32	1	500	132	59
18	36	54	1	500	250	73
18	54	72	1	500	334	91
18	72	90	1	400	416	109
18	90	108	1	400	502	127
18	108	126	1	333	574	145

Les résultats d'implémentation des tableaux 4-1 à 4-6 ont été obtenus en virgule fixe sur FPGA avec l'outil Vivado HLS de Xilinx. Ce processus d'implémentation a permis d'évaluer les performances et la précision des calculs sur une plateforme matérielle réelle, en tenant compte des spécifications de l'application et des contraintes matérielles. Pour tous les tests, des blocs DSP48 ont été utilisés, dont l'architecture matérielle est composée d'un multiplicateur à deux opérandes, complété à deux, sur 18 bits chacun, suivi de multiplexeurs et d'un additionneur à trois entrées sur 48 bits avec extension de signe. Cette implémentation se réfère au DSP48 de la série FPGA 7, connue sous le nom de Kintex-7.

Tableau 4-2 Test d'un multiplieur en virgule fixe en VHDL et en HLS sur le Kintex-7 de Xilinx.

Configuration binaire			**Tests en VHDL sur Vivado**		**Tests en HLS**				
			Ressources		Latence		Ressources		
Wint	Wlf	Wl	DSP48	LUT	Nbre de cycle	Clock (MHz)	DSP48	FF	LUT
18	36	54	10	75	1	100	10	275	177
18	54	72	17	160	1	100	17	363	194
18	72	90	26	293	1	100	26	363	194
18	90	108	37	406	1	100	37	363	194
18	108	126	50	867	1	100	50	363	194

Pour estimer les ressources nécessaires à l'implémentation en virgule fixe d'un multiplicateur, nous avons varié le nombre de bits des mots binaires à multiplier en utilisant les LUT et les DSP de Xilinx. Pour cette implémentation, nous avons utilisé des blocs DSP48, dont l'architecture matérielle comprend un multiplicateur à deux opérandes, complémentés à deux, sur 18 bits chacun, suivi de multiplexeurs et d'un additionneur à trois entrées sur 48 bits avec extension de signe et dans le calcul théorique, nous avons défini la longueur du mot binaire (wl) avec des valeurs allant de 32 à 126 bits. Étant donné que les DSP de Xilinx ont une

longueur binaire de 18 bits, nous avons utilisé la formule suivante pour estimer le nombre de DSP nécessaires (DSP_theorique) : DSP_theorique = ceil(wl/18)^2. Ainsi, nous obtenons les valeurs suivantes : [4 9 16 25 36 49], et lors de l'implémentation, les résultats réels du nombre de DSP utilisés (DSP_synth) sont légèrement différents : [5 10 17 26 37 50]. L'écart entre les résultats de synthèse et la théorie est négligeable dans tous les cas, ne dépassant pas 1 DSP. Cette cohérence entre les résultats théoriques et pratiques démontre une bonne correspondance entre les estimations initiales et les résultats réels, ce qui valide l'approche de conception et d'implémentation utilisée et près avoir tester la multiplication et l'addition nous entamons les tests pour le produit vectoriel présenté par le tableau 4-3

Tableau 4-3 Test produit vectoriel de dimension 16 en HLS sur le Kintex-7 de Xilinx.

Configuration binaire			**Latence**		**Ressources**					
(Wint)	(Wlf)	(Wl)	Nbrs Cycles	Clock (MHz)	DSP48		FF		LUT	
					Nbr	%	Nbr	%	Nbr	%
16	16	32	3	100	64	2	5243	1	4236	1
18	54	72	3	100	272	9	7868	1	4480	1
18	72	90	3	100	416	14	8822	1	4894	1
Virgule flottante 32 bits			18	100	64	2	4651	1	4137	1

Le tableau présente les résultats des tests d'implémentation d'un produit vectoriel, où chaque vecteur est de 16 éléments, ce produit vectoriel est essentiel dans la construction d'une architecture, illustrée dans la figure 4-1, qui est utilisée pour réaliser le produit matrice-vecteur avec une structure arborescente prédéfinie. Cette architecture arborescente a été reconnue pour ses performances, notamment en termes de latence de calcul, comme le démontre le tableau 4-3 et, on constate que le nombre de cycles nécessaires pour effectuer le produit matrice-vecteur en virgule fixe est considérablement inférieur à celui requis en

virgule flottante. Cette réduction significative du nombre de cycles en virgule fixe indique une amélioration de l'efficacité de calcul et de la vitesse de traitement par rapport à la représentation en virgule flottante. Ainsi, l'utilisation de la représentation en virgule fixe offre des avantages en termes de performance et de rapidité avec l'architecture arborescente utilisée pour le produit matrice-vecteur.

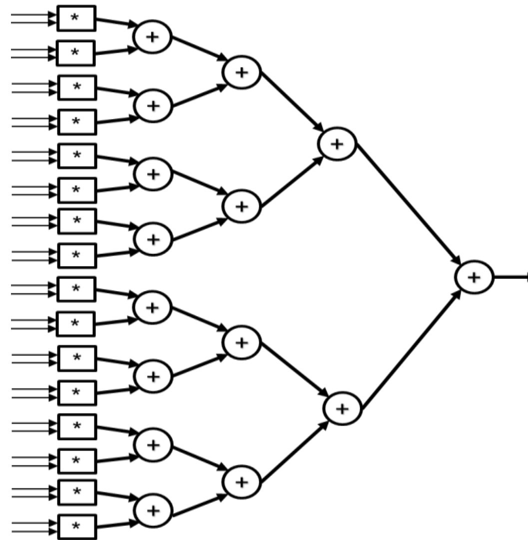

Figure 4-1 Arborescence composée d’unité de multiplication (carrée) et d’addition (cercle).

Le produit matrice-vecteur, dont les résultats sont répertoriés dans le tableau 4-4, représente la dernière étape de calcul après l'exécution de FSM lorsque le produit matrice-vecteur est nécessaire. L'architecture associée à cette étape de calcul montre l'utilisation d'un multiplexeur pour lire séquentiellement les lignes de la matrice, ce qui permet de réutiliser les ressources du produit vectoriel pour calculer d'autres éléments de la matrice. Cette approche séquentielle vise à minimiser l'utilisation des ressources matérielles, mais au prix d'une latence de sortie accrue et cette méthode séquentielle présente ses propres avantages et inconvénients. D'un côté, elle permet une utilisation minimale des ressources matérielles, ce qui est particulièrement avantageux dans des environnements où les ressources sont limitées.

D'un autre côté, cette approche peut entraîner une latence de sortie plus élevée, ce qui peut être un inconvénient dans des applications nécessitant des résultats en temps réel.

Tableau 4-4 Test Produit matrice-vecteur de dimension 16 en HLS sur le Kintex-7 de Xilinx.

Nbre de DP16	**Latence**			**Ressources**							
	Nbre de cycles		Clock (MHz)	32 bits		72 bits		90 bits		Flottante	
	Virgule fixe	Virgule flottante		DSP48	%	DSP48	%	DSP48	%	DSP48	%
1	55	70	100	64	2	272	9	416	14	64	2
2	29	44	100	128	4	544	18	832	28	128	4
3	21	36	100	256	8	816	27	1248	42	192	6
4	17	32	100	512	16	1088	36	1664	56	512	8

En effet, il est important de noter que cette approche de calcul du produit matrice-vecteur peut être facilement adaptée à des matrices de taille plus importante tout en conservant les avantages de latence par rapport à la représentation en virgule flottante. Cependant, les résultats des tests montrent que cette implémentation est relativement gourmande en ressources. Par exemple, sur les FPGA de la série 7 de Xilinx, qui offrent jusqu'à 3 600 DSP par dispositif, cette implémentation consomme 1 248 DSP pour 3 DP16 avec des mots de 90 bits, ce qui représente environ 42% des DSP disponibles sur la carte. Ainsi, le choix d'utiliser cette approche doit être soigneusement évalué en fonction des exigences spécifiques de conception et des contraintes matérielles.

4.2.2 Implémentation de FSM

Dans cette phase d'implémentation en virgule fixe de FSM, nous suivrons les formules et les codes MATLAB détaillés dans la section 2 du chapitre précédent. Cette approche garantira une cohérence entre la conception théorique et la mise en œuvre pratique sur FPGA.

En effet nous nous efforcerons également d'optimiser cette implémentation en utilisant des directives pour permettre une parallélisation complète à chaque itération de l'algorithme. Cette parallélisation sera appliquée de manière stratégique aux différentes boucles de calcul afin de minimiser la latence. De plus, nous veillerons à partitionner entièrement les tableaux de données afin de faciliter l'accès à la mémoire et de réduire davantage la latence. Une attention particulière sera accordée à la parallélisation des produits scalaires, qui sont des opérations critiques dans FSM. À cet effet, un arbre d'addition détaillé sera mis en place pour chaque produit scalaire, permettant ainsi une réutilisation efficace des ressources FPGA disponibles.

En adoptant cette approche méthodique et en mettant en œuvre des techniques d'optimisation appropriées, nous visons à obtenir une implémentation en virgule fixe de FSM qui allie efficacement précision, performances et utilisation optimale des ressources matérielles sur FPGA.

Le tableau 4-5 offre un aperçu des latences et des ressources associées à FSM. Il révèle que les implémentations en virgule fixe proposées présentent des latences inférieures à celles des implémentations FSM à virgule flottante comparables pour toutes les configurations binaires à virgule fixe. Ces résultats confirment la capacité des implémentations à virgule fixe à offrir des performances compétitives par rapport à leurs homologues à virgule flottante tout en conservant une précision adéquate.

La configuration binaire joue un rôle crucial dans les performances et l'utilisation des ressources de l'algorithme. Les résultats du tableau indiquent que les différentes configurations binaires entraînent des variations significatives de la latence et des ressources nécessaires.

Tableau 4-5 Implémentation HLS de FSM sur le Kintex-7 de Xilinx.

Configuration binaire			**Latence**		**Ressources**	
Wint	Wlf	Wl	Nbre de Cycles	Clock (MHz)	DSP48	
					Nbr	%
16	16	32	33	100	656	23
16	32	48	34	100	1556	55
16	48	64	35	100	2112	75
16	64	72	35	100	2516	89
Virgule flottante (32 bits)			50	100	410	14

Par exemple, pour une configuration binaire avec une longueur de mot binaire de 16 bits pour la partie entière et 16 bits pour la partie fractionnaire (Wint = 16, Wlf = 16), la latence est de 33 cycles d'horloge 23% des DSP48. En comparaison, lorsque la longueur du mot binaire est augmentée à 32 bits (Wlf = 32), la latence augmente à 34 cycles d'horloge et l'utilisation des ressources DSP48 est plus importante, avec 55% des DSP48. De même pour (Wlf = 48), la latence et l'utilisation des ressources continuent d'augmenter, avec 35 cycles d'horloge et 75% des DSP48 utilisés. Cependant, pour une configuration binaire en virgule flottante, la latence est légèrement plus élevée, avec 50 cycles d'horloge nécessaires, mais l'utilisation des ressources DSP48 est considérablement moindre de 14% des DSP48 utilisés.

Ces résultats soulignent l'importance de choisir judicieusement la configuration binaire en fonction des exigences de performance et de ressources spécifiques à chaque application. Une configuration binaire bien choisie peut permettre d'optimiser à la fois la latence et l'utilisation des ressources, garantissant ainsi une efficacité maximale de l’implémentation sur FPGA de FSM.

4.3 Implémentation SysGen

Le logiciel SysGen de Xilinx est un outil puissant et polyvalent pour l'implantation de circuits sur FPGA. Il permet de concevoir et de simuler des systèmes sur FPGA en exploitant les possibilités combinées de sur MATLAB® Simulink®, De plus, il offre une plate-forme complète pour développer et vérifier des algorithmes, puis effectuer la simulation de modèles, tout en générant du code VHDL ou Verilog spécifique au matériel pour les conceptions sur FPGA. En effet, l'un des principaux avantages de SysGen est qu'il facilite considérablement le processus de développement en autorisant l'utilisateur à voir et à manipuler ses conceptions par l'intermédiaire d'une interface graphique conviviale. Cette approche visuelle simplifie grandement la modélisation des systèmes, ce qui accélère le temps nécessaire pour mettre au point et vérifier ces derniers. Des fonctions avancées de simulation intégrées à MATLAB et Simulink permettent également d'évaluer les performances du système dans différents contextes simulés, ce qui aide à déceler et corriger les erreurs potentielles avant le déploiement sur du matériel réel.

4.3.1 Théorie derrière la conception du processeur élémentaire

Pour expliquer la théorie derrière l'idée de processeur élémentaire qui permet de réaliser le calcul de la FSM avec une architecture systolique ou semi systolique et pour bien expliquer la méthode on se base sur une matrice de 3×3, et on se concentre sur la partie mise à jour de l'inverse de la matrice à chaque itération définie par l'équation

$$\mathbf{B} = \sigma \mathbf{A}^{-1}\mathbf{u}\mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1} \tag{4-1}$$

Et

$$(\mathbf{A} + \mathbf{u}\mathbf{v}^{\mathrm{T}})^{-1} = \mathbf{A}^{-1} - \mathbf{B} \tag{4-2}$$

Donc toute la partie de mise à jour de la matrice sera remplacée par la matrice **B** et la théorie derrière le processeur élémentaire se concentre sur comment calculer la matrice **B** la matrice de mise à jour vu qu'après le calcule de **B** il nous reste que soustraire cette perturbation de l'inverse de la matrice de référence et on développant cette formule pour une matrice 3×3 ca sera plus facile de plonger dans les détails des opérations mathématiques requises pour ce processus ce qui nous offre une compréhension approfondie des algorithmes et des manipulations matricielles nécessaires pour concevoir une architecture matérielle qui satisfait les exigences du projet en termes de précision et latence de calcul.

Si on détaille la matrice **B** on a :

$$\mathbf{v} = \begin{pmatrix} v1 \\ v2 \\ v3 \end{pmatrix} \qquad \mathbf{u} = \begin{pmatrix} u1 \\ u2 \\ u3 \end{pmatrix} \qquad \mathbf{A}^{-1} = \begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix}$$

Et on détaille les calcules

$\mathbf{vA}^{-1}$=(v1×a11+v2×a21+v3×a31   v1×a12+v2×a22+v3×a32   v1×a13+v2×a23+v3×a33)

$\mathbf{A}^{-1}\mathbf{u}$=(a11×u1+a12×u2+a13×u3 a21×u1+a22×u2+a23×u3   a31×u1+a32×u2+a33×u3)

Il est a noté que le résultat du produit **Au** est un vecteur ligne

Donc le produit vectorielle (**Au**)(**vA**) nous donne

$$\mathbf{B} = \begin{pmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \end{pmatrix}$$

Avec:

b11= (a11×u1 + a12×u2 + a13×u3) × (v1×a11 + v2×a21 + v3×a31)

$$b12 = (a11\times u1 + a12\times u2 + a13\times u3) \times (v1\times a12 + v2\times a22 + v3\times a32)$$

$$b13 = (a11\times u1 + a12\times u2 + a13\times u3) \times (v1\times a13 + v2\times a23 + v3\times a33)$$

$$b21 = (a21\times u1 + a22\times u2 + a23\times u3) \times (v1\times a11 + v2\times a21 + v3\times a31)$$

$$b22 = (a21\times u1 + a22\times u2 + a23\times u3) \times (v1\times a12 + v2\times a22 + v3\times a32)$$

$$b23 = (a21\times u1 + a22\times u2 + a23\times u3) \times (v1\times a13 + v2\times a23 + v3\times a33)$$

$$b31 = (a31\times u1 + a32\times u2 + a33\times u3) \times (v1\times a11 + v2\times a21 + v3\times a31)$$

$$b32 = (a31\times u1 + a32\times u2 + a33\times u3)\ v\ (v1va12 + v2\times a22 + v3\times a32)$$

$$b33 = (a31\times u1 + a32\times u2 + a33\times u3) \times (v1\times a13 + v2\times a23 + v3\times a33)$$

Dans le cas des systèmes à deux niveaux et de Back-to-Back, lorsque l'on considère les conditions pour les vecteurs **u** et **v** de FSM, on constate que seuls deux éléments sont non nuls, à savoir 1 et -1, avec les mêmes positions pour ces éléments non nuls dans les deux vecteurs. Cette configuration simplifiée facilite l'application de FSM dans notre contexte spécifique. En ayant seulement deux éléments non nuls dans les vecteurs **u** et **v** l'opération de mise à jour de l'inverse de la matrice devient plus concise et efficace, ce qui contribue à la simplicité et à l'efficacité de l'implémentation de l'algorithme, et pour nos calculs des matrices simplifiées de 3×3 à titre d'exemple on représente **u** et **v** comme suite :

$$\mathbf{v} = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} \qquad \mathbf{u} = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}$$

Ce qui nous donne

$$b11 = (a11 - a13)(a11 - a31)$$

Après avoir obtenu les nouvelles valeurs de la matrice **B** en remplaçant les éléments des vecteurs **v** et **u** par les valeurs réelles non nulles pour seulement deux éléments comme

mentionné précédemment, l'idée consiste à concevoir un processeur élémentaire capable de satisfaire le calcul de chaque élément de **B**. Ce processeur élémentaire doit être conçu de manière à pouvoir manipuler les éléments de la matrice de manière efficace et à effectuer les opérations nécessaires pour mettre à jour les valeurs de **B** conformément à l'équation de FSM. En mettant en œuvre ce processeur élémentaire, il est possible de construire un système qui peut calculer rapidement et avec précision les mises à jour de l'inverse de la matrice après une perturbation donnée détaillé dans la figure 4-2.

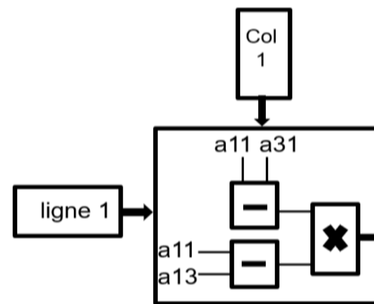

Figure 4-2 Processeur élémentaire.

En répétant le design du processeur élémentaire autant de fois que nécessaire en fonction de la dimension de la matrice à inverser, on peut construire un système complet capable de gérer des matrices de différentes tailles. Chaque processeur élémentaire est responsable du calcul des mises à jour pour un élément spécifique de la matrice résultante. En répliquant ce processeur pour chaque élément de la matrice, on crée un système parallèle capable de gérer efficacement des calculs complexes sur des matrices de grande taille, en prenant l’exemple du deux niveau les matrices a inversées sont des matrices de 16×16 présenté par la figure 4-3 qui décrit bien notre méthode qui permet une mise à l'échelle efficace du système en fonction de la taille de la matrice, tout en maintenant des performances élevées.

Figure 4-3 Design et Architecture désirée.

Le design que nous allons présenter par la suite par la figure 4-4 est en accord avec toutes les caractéristiques de l'architecture carrée précédemment décrite. Cependant, nous allons le considérer colonne par colonne. Chaque colonne de notre design contient 16 processeurs élémentaires, chacun responsable du calcul d'un élément spécifique de la nouvelle matrice inverse. Le résultat final sera sous forme de vecteurs lignes, chaque vecteur contenant 16 éléments qui contribuent à la construction de la nouvelle matrice inverse. Une fois que toutes les colonnes ont été traitées, il suffit de soustraire les deux matrices pour obtenir la nouvelle matrice inversée. Cette approche parallèle et systolique permet un traitement efficace des calculs pour des matrices de grande taille, tout en maintenant une précision élevée et des performances optimales. Cette approche permet une évaluation détaillée des performances et de l'efficacité de la conception, en mettant l'accent sur l'importance des colonnes individuelles et leur contribution à la formation de la matrice résultante. En adoptant une évaluation par colonne, la conception garantit une analyse approfondie de la contribution de

chaque segment à la construction globale de la matrice, facilitant ainsi une compréhension globale de sa fonctionnalité et de son efficacité.

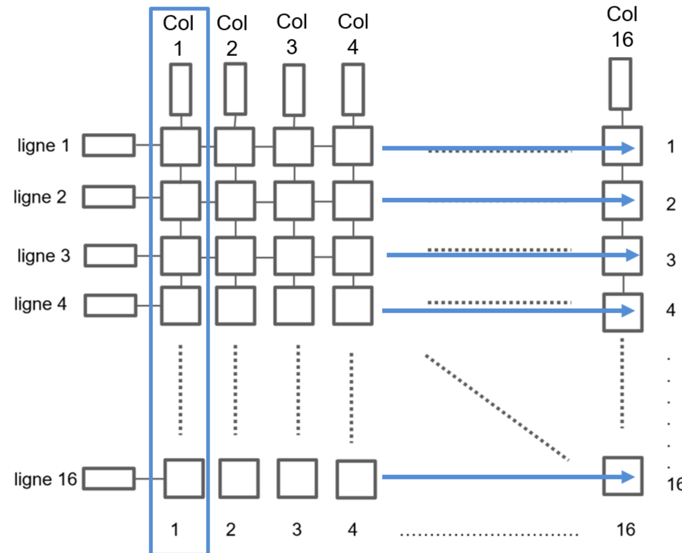

Figure 4-4 Design et description de la méthode.

De plus, cette architecture semi-systolique offre une modularité qui permet son adaptation à différentes tailles de matrices et à d'autres systèmes, assurant ainsi une flexibilité et une réutilisabilité accrues dans divers contextes d'application.

4.3.2 Ordonnancement

Cette partie se concentre sur l'ordonnancement de l'algorithme de FSM pour l'implémentation dans SysGen, nous suivons l'architecture semi-systolique basée sur le concept du processeur élémentaire décrit précédemment, en effet cet ordonnancement peut être organisé de manière séquentielle tout en respectant l'ordre des opérations telles que décrites dans l'algorithme original. La figure 4-5 présente l'organigramme général de l'algorithme qui détaille la séquence des étapes nécessaires pour effectuer les opérations de mise à jour de l'inverse de la matrice en mettant en évidence la parallélisation des opérations

à travers l'utilisation de processeurs élémentaires comme détaillé a le paragraphe précédent, chaque étape est organisée de manière logique et séquentielle, permettant ainsi une implémentation efficace et précise de l'algorithme.

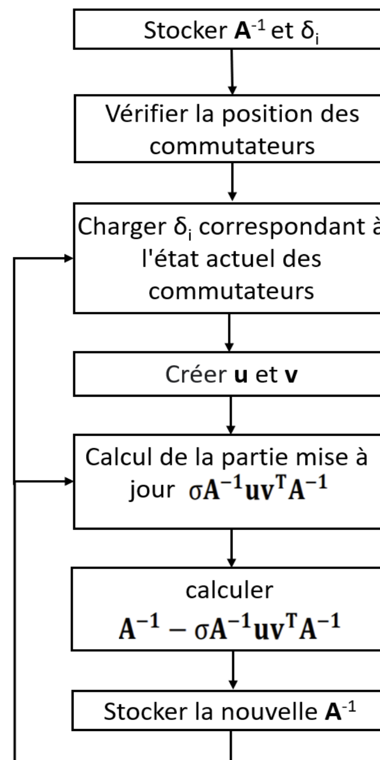

Figure 4-5 Organigramme général de FSM.

Le FSM suit un processus bien défini pour mettre à jour la matrice inverse d'une manière incrémentale. Tout d'abord, la matrice de référence $\mathbf{A}^{-1}$ est déjà pré stocker ainsi que les valeurs δi calculées à partir de la matrice de référence et les vecteurs **u** et **v** pour chaque état d'interrupteurs et stockées en mémoire comme il est déjà détaillé dans le chapitre 3 a la section reformulation de l'algorithme de FSM. Ensuite, l'état des commutateurs est vérifié pour déterminer la configuration actuelle du système. En fonction de cette configuration, les valeurs appropriées sont chargées. En utilisant ces paramètres, les vecteurs **u** et **v** sont calculés. Ensuite, une étape cruciale de l'algorithme consiste à calculer la partie de mise à

jour $\sigma\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^{T}\mathbf{A}^{-1}$ qui est calculé à partir des valeurs actuelles de **u**,**v** et δi. Cette partie est soustraite de la valeur actuelle de $\mathbf{A}^{-1}$, ce qui nous donne la nouvelle valeur de $\mathbf{A}^{-1}$, cette nouvelle valeur est alors stockée en mémoire pour être utilisée dans la prochaine itération de l'algorithme et ce processus est répété à chaque itération, permettant ainsi une mise à jour itérative et efficace de la matrice inverse $\mathbf{A}^{-1}$ en fonction des changements de l'état des interrupteurs du système. La figure 4-6 montre l'organigramme de chargement de sigma qui commence par le stockage de deux tableaux, un premier pour les valeurs de sigma et un deuxième pour les adresses qui permet d'accéder correctement aux valeurs des sigmas stockées, en effet, pour accéder aux valeurs dans la première table, nous utilisons l'indice [switch_value + 64×I], où switch_value représente la valeur des interrupteurs combinées, variant de 0 à 64, et I correspond à l'itération en cours.

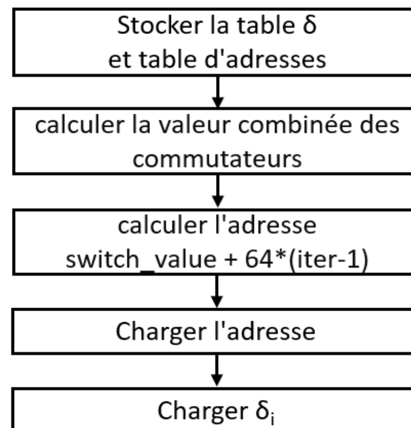

Figure 4-6 Organigramme de chargement de sigma.

En multipliant l'itération par le nombre total de valeurs pouvant être stockées dans une ligne de la table (64), nous obtenons l'indice de la ligne dans laquelle nous devons rechercher la valeur. En ajoutant switch_value à cet indice, nous ajustons correctement la position dans la table en tenant compte de la valeur des interrupteurs. Cette approche garantit que nous accédons à la valeur appropriée dans la table en fonction de l'état d'interrupteur à un moment

donné. Les deux tables sont stockées réellement dans des blocs RAMs (BRAM), la valeur à la sortie de la première BRAM représente l'adresse de sélection de la seconde BRAM pour la sélection du sigma, nous n'avons pas besoin d'effectuer de calculs supplémentaires sur l'indice. Nous utilisons simplement la valeur de switch_value telle quelle, car elle correspond directement à l'indice approprié pour sélectionner le bon sigma dans la deuxième table donc cette approche assure une sélection précise du sigma en fonction de l'état actuel des interrupteurs, la partie de gestion des BRAM sera détaillée en profondeur dans la section architecture suivante. Après avoir fait l'analyse profonde des calculs des matrices et vecteurs itération par itération et interrupteur par interrupteur de la FSM spécifique à notre circuit de puissance, on a remarqué que **v** est un vecteur rempli de seulement 0 sauf en deux éléments de 1 et -1 sur l'élément correspondant aux deux colonnes sélectionnées. Le vecteur **u** représente les éléments qui diffèrent entre la matrice de référence et la nouvelle matrice à inverser dans la colonne concernée, sélectionnée par le vecteur **v** et qui est aussi rempli de seulement 0 sauf en deux éléments de 1 et -1 on peut alors se débarrasser du calcul complet de la formule en se basant sur la figure 4-7, on montre l'organigramme de calcul de la partie de mise à jour jours de la matrice inverse. L'idée essentielle est de simplifier l'opération de mise à jour jours de la matrice qui repose sur l'exploitation des vecteurs qui ne contient que deux valeurs non nulles, 1 et -1, placées à des positions précises., en effet multiplier une matrice par un vecteur rempli de 0 sauf en deux points de valeur 1 et -1 revient juste à choisir les deux colonnes de la matrice correspondante aux positions des éléments non nuls de la vecteur et vu que les deux valeurs sont 1 et -1 il suffit de soustraire les deux vecteurs choisit et cette opération elle peut se faire en parallèle pour le produit $\mathbf{A}^{-1}\mathbf{u}$ et $\mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}$, ce qui nous donne un vecteur ligne pour $\mathbf{A}^{-1}\mathbf{u}$ et un vecteur colonne pour $\mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}$, alors le produit des

deux vecteurs nous donne une matrice $\mathbf{A}^{-1}\mathbf{u}\ \mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}$ et finalement il nous reste que multiplier par sigma pour obtenir la matrice de mise à jour avant l'étape finale décrite précédemment par l'organigramme général de l'algorithme.

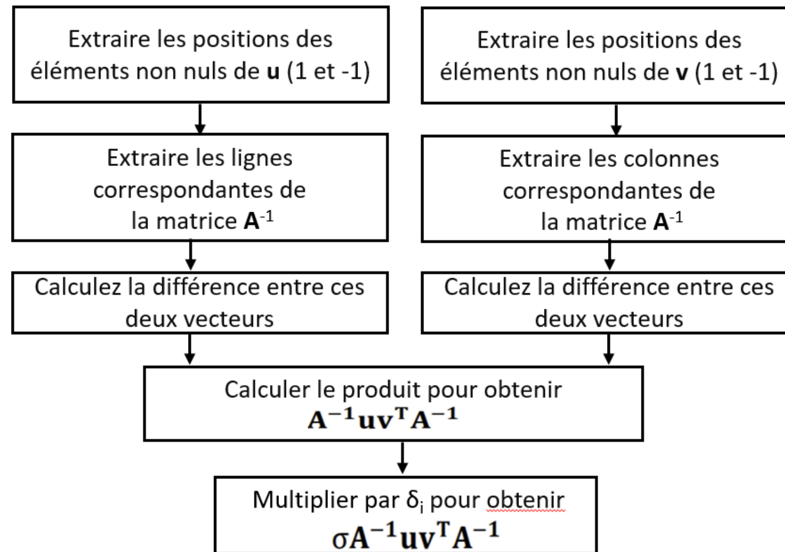

Figure 4-7 Organigramme de calcul de la partie mis à jour.

4.3.1 Architecture

L'implémentation de l'algorithme à l'aide de SysGen a pour principal objectif de réduire la latence tout en offrant un contrôle accru sur l'architecture de l'algorithme. Elle permet également de comparer les conceptions personnalisées avec celles générées par Vivado-HLS. En effet, l'utilisation de blocs fonctionnels dans SysGen permet de créer des conceptions similaires à la programmation avec les IP de base de Vivado, ce qui facilite le développement rapide de conceptions personnalisées par rapport au codage en VHDL, bien que cela prenne encore plus de temps que Vivado-HLS. Pour mettre en œuvre l'algorithme, une reformulation des formules détaillées dans la section 2 du chapitre 3 est nécessaire. Cette reformulation met l'algorithme en pipeline avec une structure d'opérations parallèles. Elle permet d'optimiser les ressources ou de réduire la latence en fonction des besoins spécifiques. Chaque itération

de l'algorithme repose sur la différence entre la matrice d'origine et la nouvelle matrice calculée à chaque itération, et par la suite on va détailler chaque partie de notre architecture. La partie Sigma présentée par la figure 4-8 de l'architecture matérielle est cruciale pour le calcul de la valeur σ à chaque itération, une composante essentielle de l'algorithme SM, cette section reçoit en entrée l'état des interrupteurs ainsi que les vecteurs **u** et **v**.

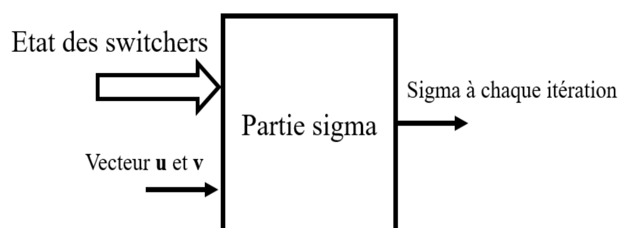

Figure 4-8 Partie Sigma.

En effet l'état des interrupteurs influence directement quelles lignes et colonnes de la matrice de référence sont sélectionnées et donc modifiées, les vecteurs **u** et **v** contenant les différences entre la matrice de référence et la nouvelle matrice, jouent un rôle clé dans ces modifications et le calcul de σ s'appuie sur l'état actuel des interrupteurs et les vecteurs **u** et **v** pour ajuster la matrice inverse de manière ciblée et précise, garantissant une mise à jour rapide et efficace. Ce processus optimise l'utilisation des ressources matérielles en permettant une mise à jour en parallèle des éléments concernés, et ce, à chaque itération, assurant ainsi que la matrice inverse reste précise et à jour en temps réel. Alors la partie Sigma joue un rôle déterminant en intégrant l'état dynamique des interrupteurs avec les vecteurs de mise à jour pour calculer σ, ce qui est indispensable pour la performance globale de l'algorithme SM.

Le contrôleur illustré par la figure 4-9, surveillant l'état des interrupteurs en tant qu'entrées principales, ces informations sont essentielles pour déterminer les adresses

nécessaires aux BRAM (Block RAM), tant pour les lignes que pour les colonnes et en fonction de l'état détecté, le contrôleur génère les adresses mémoires appropriées.

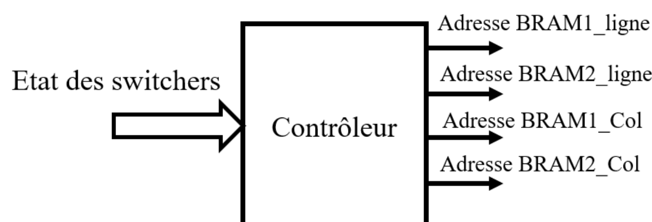

Figure 4-9 Partie contrôleur.

Celles-ci sont essentielles pour cibler spécifiquement les lignes et les colonnes qui doivent être lu ou mises à jour. Ainsi, le contrôleur assure une gestion efficace des accès mémoire, permettant un fonctionnement optimisé et précis du système Sigma.

En effet, seules les lignes et colonnes 1, 2, 3, 4 et 5 changent en fonction de l'état des interrupteurs, selon les règles suivantes : la ligne et la colonne 1 changent si s1, s3 ou s5 changent ; la ligne et colonne 2 changent si s2, s4 ou s6 changent ; la ligne et colonne 3 changent si s1 ou s2 changent ; la ligne et colonne 4 changent si s3 ou s4 changent ; la ligne et colonne 5 changent si s5 ou s6 changent ; et enfin, la ligne et colonne 6 changent si s5 ou s2 changent.

La figure 4-10 représente la partie gestion des BRAM (Block RAM) qui contient la matrice inverse de référence, cette architecture est composée de quatre modules BRAM, chacun gérant une partie spécifique de la matrice inverse de référence. Dont les modules sont organisés en deux paires : BRAM1_ligne et BRAM1_col, ainsi que BRAM2_ligne et BRAM2_col. En effet la BRAM1_ligne reçoit la matrice de référence inverse et une adresse spécifique (Adresse BRAM1_ligne), extrayant soit la ligne 1 soit la ligne 2 en fonction de l'adresse fournie, de même, BRAM1_col reçoit la matrice de référence inverse et une adresse

spécifique (Adresse BRAM1_col), extrayant soit la colonne 1 soit la colonne 2. BRAM2_ligne, quant à lui, extrait soit la ligne 3, soit la ligne 4, soit la ligne 5 en fonction de l'adresse fournie, et BRAM2_col extrait soit la colonne 3, soit la colonne 4, soit la colonne 5, par ailleurs cette organisation permet une séparation claire des adresses et des données, facilitant une sélection ciblée et efficace des éléments nécessaires pour les opérations ultérieures de l'algorithme et une gestion parallèle des lignes et des colonnes par ces modules distincts permet d'effectuer des opérations simultanément en augmentant ainsi l'efficacité du système.

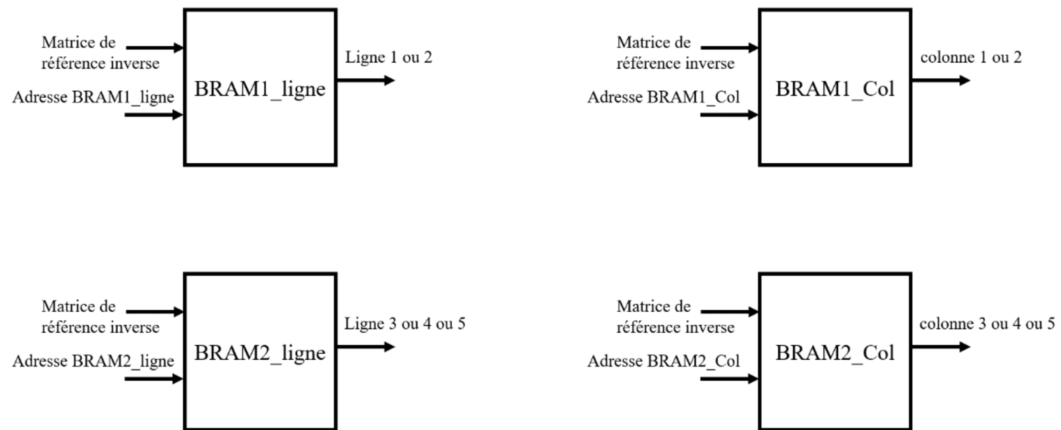

Figure 4-10 Gestion des BRAM.

Mais cette architecture est spécifiquement conçue pour des applications où seules certaines lignes et colonnes de la matrice changent fréquemment permettant ainsi des mises à jour efficaces sans recalculer l'ensemble de la matrice inverse. En se concentrant sur des lignes et des colonnes spécifiques. Notre conception optimise l'utilisation des ressources matérielles, réduisant ainsi la charge de calcul et le temps de traitement, avantage particulièrement notable dans des systèmes embarqués ou des environnements où les ressources sont limitées. L'organigramme du calcul de la mise à jour de la matrice inverse, présenté dans la figure 4-11 montre un processus détaillé et itératif pour la mise à jour d'une

matrice inverse, notre architecture extraite se compose de plusieurs modules spécialisés, chacun traitant une partie spécifique du calcul donc une structure modulaire permet une mise en œuvre efficace et maintenable de l'algorithme en permettant des optimisations et des ajustements ciblés pour chaque étape du processus.

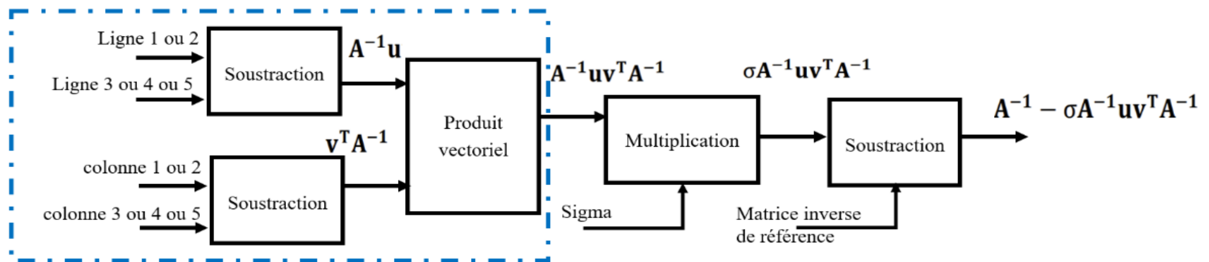

Figure 4-11 L'organigramme du calcul de la mise à jour de la matrice inverse.

Étant donné que le vecteur u et v représente les différences entre la matrice de référence et la nouvelle matrice à inverser, ils sont composés majoritairement de zéros, avec seulement deux positions spécifiques prenant des valeurs différentes (1 ou -1) et cette particularité permet de simplifier le calcul de la mise à jour de la matrice, car multiplier une matrice par un tel vecteur revient à sélectionner deux colonnes ou deux lignes de la matrice correspondant aux positions de ces éléments et on sait que les deux valeurs sont 1 et -1 il suffit de soustraire les deux colonnes choisies. On peut effectuer cette opération en parallèle pour le produit $\mathbf{A}^{-1}\mathbf{u}$ et $\mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}$ ainsi, pour obtenir $\mathbf{A}^{-1}\mathbf{u}$, on soustrait la sortie de la première BRAM de celle de la deuxième sur les lignes pertinentes et la même chose pour $\mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}$. Ensuite, on calcule le produit vectoriel des deux vecteurs résultants ce qui nous donne $\mathbf{A}^{-1}\mathbf{u}\,\mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}$, puis on multiplie le tout par σ pour obtenir la matrice de mise à jour $\sigma\mathbf{A}^{-1}\mathbf{u}\,\mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}$. Enfin, on soustrait cette matrice de mise à jour de la matrice inverse de référence déjà stockée pour obtenir la nouvelle matrice inverse. Cela se fait vecteur par vecteur, car le résultat de la matrice de mise à jour sort également vecteur par vecteur.

La partie pointillée en bleu du schéma représente la mise en application efficace de la théorie derrière le processeur élémentaire, bien décrite dans la section précédente. Cette méthode met en parallèle plusieurs processeurs élémentaires afin de vérifier l'architecture sous forme de vecteur, cette évaluation par colonne garantit un examen approfondi de la contribution de chacun des segments à la structure globale de la matrice favorisant ainsi une meilleure compréhension de son fonctionnement et de son efficacité. De plus, cette architecture semi-systolique présente une modularité lui permettant d'être adaptable à différentes dimensions de matrices et à divers systèmes, ce qui augmente considérablement sa flexibilité et sa réutilisabilité dans divers contextes d'application.

En effet on va utiliser cette flexibilité et cette modularité pour adapter cette architecture pour la matrice de l'onduleur de deux niveaux d'une matrice carre 16×16 à partir de quatre blocs de 8×8 se qui déroule en quatre étapes clés. D'abord, la matrice $\mathbf{A}_{16x16}$ est divisé en quatre sous-matrices de taille 8×8, notées $\mathbf{A}_{11}$, $\mathbf{A}_{12}$, $\mathbf{A}_{21}$, $\mathbf{A}_{22}$. Avec cette division on peut traiter chaque sous-matrice de manière indépendante. Les résultats de chaque étape sont combinés pour obtenir l'inverse complet de la matrice **A** et pour atteindre cet objectif, il est nécessaire de quadrupler les ressources requises pour réaliser la décomposition en blocs de calcule parallèles des matrices 8×8 qui va être bien détaillé dans la section suivante. Ce procédé permet d'obtenir les résultats de l'inverse calculer avec le FSM de la grande matrice en un temps équivalent à celui nécessaire pour inverser une matrice 8×8. Cette méthode modulaire maximise l'efficacité du processus de décomposition en favorisant l'évolutivité et la flexibilité, ce qui est particulièrement bénéfique pour minimiser le temps de calcule mais avec un cout en ressources qui sera multiplié par quatre illustrer par la figure 4-12.

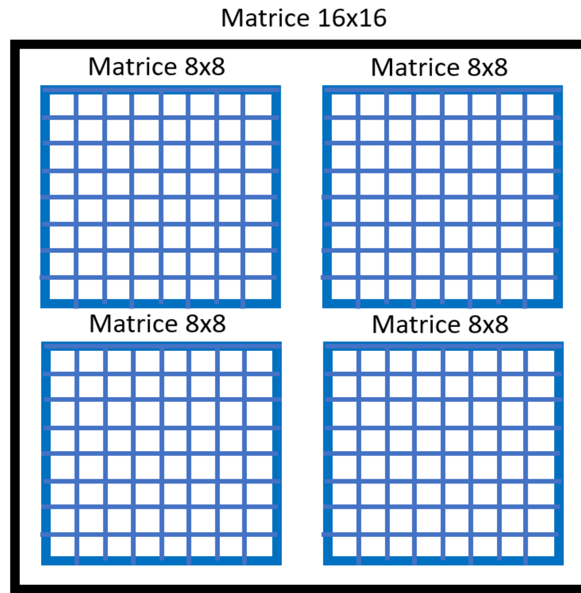

Figure 4-12 Schéma de visualisation du processus de décomposition.

4.3.2 Résultats et discussions

L'implémentation des résultats a été réalisée sur le kit d'évaluation KC705 de Xilinx, équipé du Kintex 7. Cette mise en œuvre a été effectuée en testant plusieurs configurations d'horloge et en utilisant à la fois un format à virgule flottante simple précision et un format à virgule fixe. Les deux formats sont indispensables dans de nombreuses applications en raison des contraintes de précision, notamment lorsque la matrice est mal conditionnée, comme c'est souvent le cas dans les modèles d'électronique de puissance utilisés dans le matériel dans la boucle (HIL). Les calculs sont effectués avec des nombres réels, et pour valider les résultats obtenus, ils sont comparés avec l'implémentation de référence réalisée avec Matlab de FSM. Cette validation croisée garantit la fiabilité et l'exactitude des résultats produits par l'implémentation sur FPGA, assurant ainsi sa pertinence et son utilité dans des applications réelles. Nous présentant les tableaux des résultats de Scheduling des étapes de calcul avec les délais d’une implémentation SysGen de l’algorithme de FSM pour une matrice 8×8 en

virgule fixe 32 bits avec une horloge de 200MHz les résultats des timings peuvent être résumé dans le tableau 4-6 suivant. Ce tableau décrit les deux itérations du processus de calcul de FSM dont chaque itération comprenant plusieurs étapes et délais associés.

Tableau 4-6 Résumés des délais en virgule fixe 200MHz en SysGen sur le Kintex-7 de Xilinx.

1 ère itération			**2 (N)-ème itération**			
L-E mémoire	Calcul	Résultat	Délai	L-E mémoire	Calcul	Résultat
[0-9]T	[9-19]T	[19-27]T	[19-22]T	[22-24]T	[24-34]T	[34-42]T

Pour les calculs en virgule fixe 32 bits toujours avec une fréquence d'horloge de 200 MHz, la première itération nécessite 9 cycles pour les opérations de lecture et d'écriture en mémoire décrite dans le tableau 4-7, suivis de 10 cycles pour les calculs, incluant des multiplications, des soustractions, des additions détaillées dans le tableau 4-8 et des opérations de transfert entre registres. Le résultat est obtenu entre les cycles 19 et 27, avec un délai de synchronisation de 3 cycles entre les étapes de calcul et de résultat puis une étape de rétroaction décrite par le tableau 4-9.

Tableau 4-7 E/L mémoire de la 1ère itération FSM en fixe 200 MHz en SysGen sur le Kintex-7.

Étape	**Délai (T)**	**Timing**
Choix de vecteur de référence Adressage des registres	2T	[0-2]T
Écriture du vecteur de référence sur le port B du RAM	8T	[2-10]T
Lecture du vecteur de référence sur le port B du RAM	8T	[9-17]T

Chaque itération comprend une phase de lecture et écriture en mémoire suivie d'une opération de calcul puis la récupération du résultat, en effet les délais entre ces étapes sont spécifiés avec des plages de temps indiquées entre crochets, par exemple [0-9]T signifies que la phase de lecture/écriture en mémoire de la première itération prend entre 0 et 9 unités de temps(période) et pareil pour les autres étapes des itérations avec une configuration binaire en virgule fixe de 32 bits et une fréquence d'horloge de 200MHz.

Tableau 4-8 Calcul de la 1-ére itération en virgule fixe FSM à 200 MHz en SysGen sur le Kintex -7.

<table>
<tr><th colspan="2">Étape</th><th>Délai</th><th>Timing</th><th colspan="2">Étape</th><th>Délai</th><th>Timing</th></tr>
<tr><td rowspan="3">Au</td><td>Multiplication</td><td>3T</td><td>[9-12]T</td><td rowspan="3">δAv</td><td>Soustraction</td><td>1T</td><td>[9-10]T</td></tr>
<tr><td>Addition</td><td>1T</td><td>[12-13]T</td><td>Multiplication</td><td>3T</td><td>[10-13]T</td></tr>
<tr><td>S to P</td><td>1T</td><td>[13-14]T</td><td>P to S</td><td>2T</td><td>[13-15]T</td></tr>
<tr><th colspan="2">Étape</th><th colspan="3">Délai</th><th colspan="3">Timing</th></tr>
<tr><td>AuδAv</td><td>Multiplication</td><td colspan="3">3T</td><td colspan="3">[15-18]T</td></tr>
<tr><td>A - AuδAv</td><td>Soustraction</td><td colspan="3">1T</td><td colspan="3">[18-19]T</td></tr>
</table>

Tableau 4-9 Rétroaction de la 1-ère itération en virgule fixe FSM 200 MHz en SysGen sur le Kintex-7.

Étape	**Délai**	**Timing**
Adressage Lecture des registres	2T	[0-2]T
Écriture sur le port B du RAM du vecteur Ai	8T	[2-10]T
Lecture sur le port B du RAM du vecteur Ai	8T	[10-18]T
Delay	8T	[10-18]T

Les résultats de la deuxième (ou la nième) itération présentée par les tableaux 4-10,4-11 et 4-12 montrent une optimisation dans la gestion de la mémoire avec seulement 2 cycles pour les opérations de lecture et d’écriture, tandis que les calculs prennent le même nombre de cycles que la première itération. Le résultat est obtenu entre les cycles 34 et 42, avec un délai notable de 8 cycles pour la lecture en mémoire mais qui n’as pas un grand effet sur le timing vu que l’opération de lecture et écriture mémoire se faire en parallèle avec les calculs à partir de la deuxième itération.

Tableau 4-10 E/L mémoire du 2-ème itération FSM en virgule fixe 200 MHz en SysGen sur le Kintex-7.

Étape	**Délai (t)**	**Timing**
Choix du vecteur (mux)	1T	[18-19]T
Adressage et lecture des RAMs	2T	[19-21]T
Lecture du vecteur de référence sur le port B du RAM	8T	[24-32]T

Tableau 4-11 Calcul du 2-éme itération en virgule fixe FSM 200 MHz en SysGen sur le Kintex-7.

Étape		**Délai**	**Timing**	**Étape**		**Délai**	**Timing**
Av	Multiplication	3T	[24-27]T	δ**Av**	Soustraction	1T	[24-25]T
	Addition	1T	[27-28]T		Multiplication	3T	[25-28]T
	S to P	1T	[28-29]T		P to S	2T	[28-30]T

Étape		**Délai**	**Timing**
Auδ**Av**	Multiplication	3T	[30-33]T
A - **Au**δ**Av**	Soustraction	1T	[33-34]T

Tableau 4-12 Rétroaction de la 2-éme itération en virgule fixe FSM 200 MHz en SysGen sur le Kintex-7.

Étape	Délai	Timing
Décalage de 8 Delay	8T	[18-26]T
Écriture et Lecture sur le port A du RAM du vecteur Ai	8T	[26-34]T
Delay	7T	[26-33]T

Dans ce paragraphe l'analyse se concentre sur le calcul pour une matrice 8x8 en virgule flottante à une fréquence de 200MHz y compris les étapes incluent les cycles de lecture/écriture en mémoire, les calculs, le chargement des résultats et les délais associés.

Le tableau 4-13 montre que les calculs en virgule flottante nécessitent plus de cycles comparativement aux calculs en virgule fixe. La première itération utilise 9 cycles pour les opérations de mémoire et 16 cycles pour les calculs, incluant des multiplications, des soustractions, des additions, et des transferts. Le résultat final est obtenu entre les cycles 25 et 33, avec un délai de 2 cycles pour la synchronisation détaillée dans les tableaux 4-14, 4-15 et 4-16.

Tableau 4-13 Résumés des délais en virgule flottante 200MHz en SysGen sur le Kintex -7.

1 ère itération			**2 (N)-ème itération**			
L-E mémoire	Calcul	Résultat	Délai	L-E mémoire	Calcul	Résultat
[0-9]T	[9-25]T	[25-33]T	[25-27]T	[27-29]T	[29-45]T	[45-53]T

Tableau 4-14 E/L mémoire de la 1 ère itération FSM en virgule flottante 200 MHz en SysGen sur le Kintex-7.

Étape	**Délai**	**Timing**
Choix de vecteur de référence & Adressage des registres	2T	[0-2]T
Écriture du vecteur de référence sur le port B du RAM	8T	[2-10]T
Lecture du vecteur de référence sur le port B du RAM	8T	[9-17]T

Tableau 4-15 Calcul de la 1ère itération en virgule flottante de FSM 200 MHz en SysGen sur le Kintex-7.

<table>
<tr><th colspan="2">Étape</th><th>Délai</th><th>Timing</th><th colspan="2">Étape</th><th>Délai</th><th>Timing</th></tr>
<tr><td rowspan="3">Au</td><td>Multiplication</td><td>3T</td><td>[9-12]T</td><td rowspan="3">δAv</td><td>Soustraction</td><td>4T</td><td>[9-13]T</td></tr>
<tr><td>Addition</td><td>4T</td><td>[12-16]T</td><td>Multiplication</td><td>3T</td><td>[13-16]T</td></tr>
<tr><td>S to P</td><td>1T</td><td>[16-17]T</td><td>P to S</td><td>2T</td><td>[16-18]T</td></tr>
<tr><th colspan="3">Étape</th><th colspan="2">Délai</th><th colspan="3">Timing</th></tr>
<tr><td>AuδAv</td><td colspan="2">Multiplication</td><td colspan="2">3T</td><td colspan="3">[18-21]T</td></tr>
<tr><td>A - AuδAv</td><td colspan="2">Soustraction</td><td colspan="2">4T</td><td colspan="3">[21-25]T</td></tr>
</table>

Tableau 4-16 Rétroaction de la 1 ère itération en virgule flottante FSM 200 MHz en SysGen sur le Kintex-7.

Étape	**Délai**	**Timing**
Adressage Lecture des registres	2T	[0-2]T
Écriture sur le port B du RAM du vecteur Ai	8T	[2-10]T
Lecture sur le port B du RAM du vecteur Ai	8T	[10-18]T
Delay	3T	[18-21]T

La deuxième itération, bien que plus optimisée que la première, nécessite encore un nombre significatif de cycles pour les opérations de mémoire (3 cycles) et de calcul (16 cycles). Le résultat final est produit entre les cycles 45 et 53, avec un délai de 3 cycles pour la synchronisation des opérations bien détaillées dans les tableaux 4-17, 4-18 et 4-19.

Pour les itérations allant de deux à n, ainsi que pour les deux types d'implémentation (fixe ou flottante) de l'algorithme, on observe une réduction des cycles nécessaires aux opérations de mémoire, ce qui témoigne d'une optimisation de la gestion de la mémoire. Cette résultante de la parallélisation des itérations, qui sera détaillée par le modèle de séquencement des opérations présenté à la fin de cette section, permettant ainsi de réduire le temps de calcul total du FSM des matrices 8×8 à 200. MHz.

Tableau 4-17 E/L mémoire de la 2-éme itération en virgule flottante FSM 200 MHz en SysGen sur le Kintex-7.

Étape	Délai	Timing
Choix du vecteur (mux)	1T	[25-26]T
Adressage et lecture des RAMs	2T	[26-28]T
Lecture du vecteur de référence sur le port B du RAM	8T	[29-37]T

En comparant les résultats en termes d'efficacité des cycles, les tests d’implémentations de l’algorithme de de FSM pour faire le calcul de l’inverse d’une matrice en virgule fixe sont plus avantageux d’une part ils nécessitent moins de cycles par itération et d’autre part ont des délais de synchronisation plus courts, ce qui se traduit par une exécution plus rapide, ce pendant l’implémentation des opérations de virgule flottante bien qu'elles soient plus complexes et prennent plus de cycles, offrent une précision accrue nécessaire pour certaines applications, mais avec des délais plus longs dans les calculs de FSM en virgule flottante qui reflètent une complexité et la nécessité d'une synchronisation supplémentaire.

Tableau 4-18 Calcul de la 2-éme itération en virgule flottante de FSM 200 MHz en SysGen sur le Kintex-7.

Étape		**Délai**	**Timing**	**Étape**		**Délai**	**Timing**
Au	Multiplication	3T	[29-32]T	**δAv**	Soustraction	4T	[29-33]T
	Addition	4T	[32-36]T		Multiplication	3T	[33-36]T
	S to P	1T	[36-37]T		P to S	2T	[36-38]T

Étape		**Délai**	**Timing**
AuδAv	Multiplication	3T	[38-41]T
A - AuδAv	Soustraction	4T	[41-45]T

Les tableaux illustrent clairement comment les cycles sont distribués et optimisés pour toutes les itérations, réduisant ainsi le temps de calcul total le maximum possible. En résumé, pour des applications nécessitant une exécution rapide et efficace comme le cas de l'algorithme de SM, les calculs en virgule fixe sont préférables en raison de leur utilisation moindre des cycles, et des délais plus courts. En revanche, l'application de l'algorithme de FSM sur nos cas des convertisseurs de puissance là où la précision est cruciale, est possible avec une bonne précision de calculs même avec un nombre limité de bits de calcule grâce à la méthode de correction présenté dans le chapitre précédant.

Tableau 4-19 Rétroaction de la 2-éme itération en virgule flottante de FSM 200 MHz en SysGen sur le Kintex-7.

Rétroaction		
Étape	**Délai**	**Timing**
Décalage de 8 Delay	8T	[25-33]T
Écriture et Lecture sur le port A du RAM du vecteur Ai	8T	[33-41]T
Delay	8T	[33-41]T

Dans cette partie on va ce concentré sur l'implémentation SysGen de l'algorithme de FSM pour une fréquence de 400MHz, nous observons les différences entre les opérations en virgule fixe et en virgule flottante à travers les tableaux fournis qui montrent une optimisation claire des cycles pour les itérations, réduisant ainsi le temps total de calcul de SM.

Les opérations en virgule fixe sur Kintex 7 nécessitent 4 cycles pour une multiplication et 1 cycle pour une addition/soustraction pour une fréquence d'horloge de 400 MHz. Le tableau 4-20 montre que dans la première itération, la E mémoire de prend 9 cycles (0-9), le calcul prend 12 cycles (9-21), et le résultat est obtenu en 8 cycles (21-29), avec un délai de 3 cycles (21-23). Pour la deuxième itération, la E/L mémoire prend 3 cycles (24-26), le calcul 12 cycles (26-38), et le résultat 8 cycles (38-45) et le même nombre de cycles pour les itérations suivantes.

Tableau 4-20 Résumés des délais en virgule fixe 400MHz en SysGen sur le Kintex -7.

1 ère itération			**2 (N)-ème itération**			
L-E mémoire	Calcul	Résultat	Délai	L-E mémoire	Calcul	Résultat
[0-9]T	[9-21]T	[21-29]T	[21-23]T	[24-26]T	[26-38]T	[38-45]T

Tableau 4-21 Résumés des délais en virgule flottante 400MHz en SysGen sur le Kintex-7.

1 ère itération			**2 –(N)-ème itération**			
L-E mémoire	Calcul	Résultat	Delay	L-E mémoire	Calcul	Résultat
[0-9]T	[9-25]T	[25-33]T	[25-28]T	[28-30]T	[31-47]T	[47-54]T

Les opérations en virgule sur Kintex 7 flottante nécessitent 3 cycles pour une multiplication et 4 cycles pour une addition/soustraction pour une fréquence d'horloge de 400MHz. Dans la première itération, la L-E mémoire de prend également 9 cycles (0-9), le calcul s'étend sur 16 cycles (9-25), et le résultat prend 8 cycles (25-33), avec un délai de 4 cycles (25-28). Pour les itérations suivantes 2 -ème itération ou plus, la L-E mémoire prend 3 cycles (28-30), le calcul 16 cycles (31-47), et le résultat 7 cycles (47-54). En effet l'implémentation en virgule fixe est plus efficace en termes d'utilisation des cycles, nécessitant moins de cycles par itération comparée aux opérations en virgule flottante. les opérations en virgule flottante, bien que plus coûteuses en cycles en raison de leur complexité accrue, Les itérations initiales prennent plus de temps, mais il y a une optimisation des opérations mémoire dans les itérations suivantes pour les deux types d'opérations. Globalement, les opérations en virgule fixe sont plus rapides par itération ce qui valide nos hypothèses théoriques de base.

Le tableau 4-22 présente un résumé de l'analyse le temps les opérations de calcul nécessaire des résultats de FSM à 400 MHz et comparant les temps d'exécution en virgule fixe et en virgule flottante pour les différentes étapes de calcul de l'équation de FSM, ainsi que leur impact sur la planification globale des itérations. La FSM est aussi décomposée en quatre étapes de calculs des sous-équations distinctes, et montre une répartition précise du temps par opération et par itération. On se basant sur les résultats de l'implémentation en virgule fixe, pour l'étape E.1, impliquant la multiplication et l'addition pour calculer $\mathbf{A}^{-1}\mathbf{u}$ les temps requis sont respectivement de 4T et 1T par itération, ce qui totalise 5T×I pour *n* itérations. L'étape E.2, incluant la soustraction et la multiplication pour calculer $\sigma\mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}$ qui

nécessite également un total de 5T×I. L'étape E.3 dans le tableau, correspondant au calcul de $\sigma\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}$ qui demande 4T par itération.

Tableau 4-22 Temps d'exécution de FSM sur SysGen en virgule fixe de 32 bits et virgule flottante simple à 400 MHz (T = 2.5 ns).

Étape	**Opérations**		**Temps d'exécution par itération**		**Temps d'exécution pour I itérations**	
			Virgule fixe	**Virgule flottante**	**Virgule fixe**	**Virgule flottante**
E.1	$\mathbf{A}^{-1}\mathbf{u}$	Multiplication	4T	3T	4T×I	3T×I
		Addition	T	4T	T×I	4T×I
E.2	$\sigma\mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}$	Soustraction	T	4T	T×I	4T×I
		Multiplication	4T	3T	4T×I	3T×I
E.3	$\sigma\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}$		4T	3T	4T×I	3T×I
E.4	$\mathbf{A}^{-1}-\sigma\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}$		T	4T	T×I	4T×I
Délai de synchronisation			2T	2T	2T×I	2T×I
Temps Planification de l'équation de SM			12T	16T	12T×I	16T×I
Stockage de la matrice de référence			9T	9T	9T	9T
Délai entre les itérations			5T	5T	5T×(I-1)	5T×(I-1)
Temps total de planification pour la formule de SM			21T	25T	(3 + 17×I)×T	(4 + 21×I)×T
Pour T = 2.5 ns (400 MHz) et I = 6 itérations			52.5 ns	62.5 ns	262.5 ns	325 ns

Et l'étape E.4, pour le calcul final, ajoute 1T pour une simple soustraction. Le délai de synchronisation ajoute 2T par itération, portant le temps total de planification des équations

de FSM à 12T par itération pour le calcul, soit 12T×I au total, avec I est le nombre d'itérations qui dépend des états des interrupteurs de notre circuit de puissance. En plus, le stockage de la matrice de référence nécessite un délai fixe de 9T, tandis que le délai entre les itérations est de 5T, cumulatif pour (I-1) itérations, soit 5T×(I-1). Au totale, le temps de total pour la FSM est de 21T pour la première itération et 21T +17T×(I-1) pour I itérations soit (3+17×I)×T, ce qui illustre une structure de calcul optimisé, où après une configuration initiale, chaque itération additionnelle ajoute un temps fixe de 17T. De même pour l'analyse de temps d'exécution pour la virgule flottante. Pour l'analyse en virgule flottante, une structure similaire est rappelée, avec des temps de calcul plus courts pour les multiplications 3T mais des délais plus longs pour les additions et soustractions 4T. Cela montre que la virgule flottante, bien que plus rapide dans certains cas, introduit des contraintes supplémentaires pour les opérations basiques telles que l'addition et la soustraction.

Cette analyse révèle l'efficacité d'un modèle parallèle et pipeliné, crucial pour des calculs de matrices complexes, en soulignant l'importance de la gestion des délais et de la synchronisation pour maximiser les performances globales du processus computationnel décrit par ce tableau, donc ce modèle de temps permet de prévoir efficacement les ressources nécessaires pour des calculs impliquant de grandes matrices et de nombreuses itérations, et il met en évidence l'importance de la synchronisation et de la gestion des délais pour optimiser les performances globales du processus, cette synchronisation est bien décrite par un modèle de séquencement qui suit.

La figure 4-13 présente une chronologie détaillée des étapes itératives du séquencement des itérations de calcule de SM, structurée par itération. Chaque itération se décompose en plusieurs phases : le stockage de la matrice de référence, le calcul de SM, la lecture /écriture

mémoire et les délais, ainsi que la production des résultats. Ces phases sont représentées respectivement par des motifs hachurés, des cases vides, des motifs en briques et des hachures diagonales.

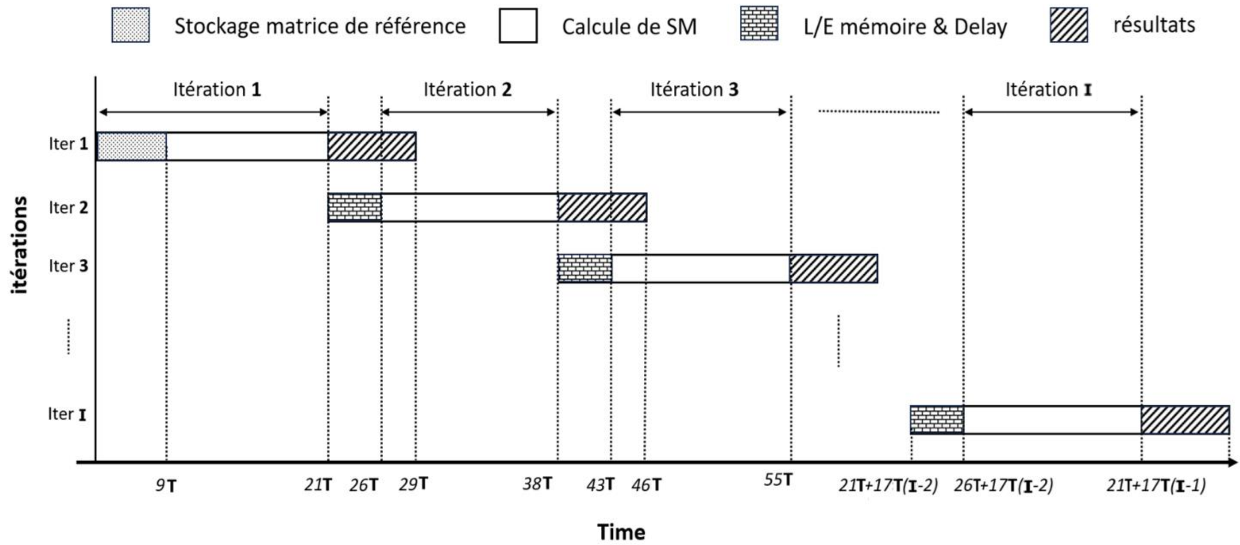

Figure 4-13 Séquencements des itérations de calcule de FSM pour l'implémentation en Fixe.

Les résultats du diagramme coïncident avec celle du tableau 4-22 précédent et montrent clairement que chaque nouvelle itération commence avant la fin de la précédente, illustrant un pipeline d'exécution où les opérations se chevauchent pour optimiser l'utilisation du temps. Par exemple, l'Itération 1 s'étend de 9T à 26T, avec le stockage de la matrice de référence commençant à 9T et les résultats apparaissant autour de 29T. L'itération 2 commence autour de 21T, avant la fin de l'Itération 1, et se termine vers 43T. Ce chevauchement permet de réduire le temps total de traitement en conclusion, ce diagramme montre que l'architecture utilisée pour implémenter l'algorithme de FSM admet une structure itérative et pipelinée optimise le temps de traitement en superposant les étapes des différentes itérations. Chaque itération commence avant que la précédente ne soit complètement terminée, ce qui permet d'exploiter les périodes d'attente et de maximiser l'efficacité du

processus. Les temps finaux calculés (par exemple, 21T + 17T(I-2)) montrent que chaque itération supplémentaire ajoute un temps fixe, illustrant une croissance linéaire du temps total avec le nombre d'itérations.

Les résultats de ce calcul présentés par le tableau 4-22 et la figure 4-13 demeurent valides et applicables à la matrice d'impédance 16×16 de l'onduleur à deux niveaux, grâce à la technique de décomposition en blocs pour le calcul parallèle des matrices 8×8. Ainsi, on peut obtenir les résultats de l'inversion de la grande matrice en un temps équivalent à celui requis pour inverser une matrice 8×8, avec une légère différence pour le temps consacré à la lecture des résultats à la fin du processus de calcul, qui s'étend sur 16 périodes au lieu de 8. Cette méthode est très efficace non seulement pour FSM mais aussi pour les tâches computationnelles nécessitant des étapes répétitives et permet une gestion optimale des ressources et du temps.

Le tableau 4-23 met en évidence la synthèse des temps de calcul entre un onduleur à deux niveaux (2-L) et un onduleur back-to-back (B2B), selon le type d'opérations effectuées (virgule fixe ou virgule flottante) et selon l'environnement de développement utilisé (HLS ou SysGen). Dans une première évidence observée, SysGen affiche des temps d'exécution inférieurs à ceux observés sous HLS pour les deux types d'onduleurs, ce qui souligne son efficacité accumulée. Pour l'onduleur 2-L, le temps d'exécution des opérations en virgule flottante est supérieur à celui des opérations en virgule fixe, avec une différence de 170 ns (31%) en HLS et de 60 ns (18%) en SysGen, ce qui suggère une meilleure optimisation des calculs en virgule flottante dans l'environnement SysGen. En revanche, pour l'onduleur B2B, les écarts sont beaucoup plus significatifs. En HLS, les opérations en virgule flottante de 3010 ns prennent plus de deux fois le temps des opérations en virgule fixe de 1330 ns. Bien

que SysGen améliore les performances de manière notable, la différence reste importante, avec des temps de 640 ns pour la virgule flottante contre 520 ns pour la virgule fixe.

Tableau 4-23 Synthèses des temps de calcul de FSM sur SysGen et HLS en virgule fixe 32 bits et flottante sur FPGA Kintex-7.

Onduleur 2-L Matrice 16×16, 6 itérations				**Onduleur B2B Matrice 26×26, 12 itérations**			
HLS		SysGen		HLS		SysGen	
Virgule fixe	Virgule flottante	Virgule fixe	Virgule flottante	Virgule fixe	Virgule flottante	Virgule fixe	Virgule flottante
330 ns	500 ns	265 ns	325 ns	1330 ns	3010 ns	520 ns	640 ns

À noter que les résultats d'implémentation de la FSM en HLS montrent des opérations en virgule flottante 32 bits et en virgule fixe 32 bits, tandis que dans SysGen, la méthode repose sur une décomposition matricielle similaire à celle utilisée pour l'onduleur 2-L qui a un nombre d'itérations maximales de 6. La matrice d'impédance du B2B, de taille 26×26, est décomposée en sous-matrices de 8×8, ce qui multiplie l'architecture par un facteur 8, tout en conservant la modularité du calcul. Bien que cela revienne à traiter une matrice équivalente à 32×32, la méthode de décomposition permet de préserver la formule de latence, mais de modifier le nombre d'itérations maximales, qui passe à 12 itérations. Cette particularité explique le changement dans le temps d'exécution observé entre les deux onduleurs. Cette variation plus marquée pour l'onduleur B2B peut être attribuée à la complexité accumulée de son architecture, le rendant plus sensible aux opérations en virgule flottante. En conclusion, les opérations en virgule fixe sont globalement plus rapides, particulièrement dans le cas de l'onduleur B2B, et l'environnement SysGen se révèle être le plus performant pour les deux types d'onduleurs et de calculs.

Le tableau 4-24 présente une comparaison de l'utilisation des ressources DSP (*Digital Signal Processor*) pour deux configurations d'onduleurs, à savoir l'onduleur 2-L et l'onduleur B2B, dans différents environnements de conception, HLS et SysGen. Les résultats sont ventilés en deux types de représentations numériques : virgule fixe et virgule flottante. Pour l'onduleur 2-L, en virgule fixe sous HLS, 656 DSP sont utilisés, représentant 23 % des ressources. En virgule flottante, l'utilisation est réduite à 410 DSP, soit 14 %. Sous SysGen, le besoin en DSP pour les deux représentations (fixe et flottante) est beaucoup plus faible, à 176 DSP (7 %). Pour l'onduleur B2B, HLS en virgule fixe montre une utilisation plus élevée, avec 1248 DSP (43 %), alors que la virgule flottante réduit cela à 1064 DSP (38 %). SysGen, similaire à l'onduleur 2-L, montre un usage plus modeste de 704 DSP (25 %) pour les deux types de représentation

Tableau 4-24 Synthèse des ressources de FSM sur SysGen et HLS en virgule fixe 32 bits et flottante sur FPGA Kintex-7.

Onduleur 2-L Matrice 16×16, 6 itérations								**Onduleur B2B Matrice 26×26, 12 itérations**							
HLS				SysGen				HLS				SysGen			
Virgule fixe		Virgule flottante		Virgule fixe		Virgule flottante		Virgule fixe		Virgule flottante		Virgule fixe		Virgule flottante	
DSP	%	DSP	%	DSP	%	DSP	%	DSP	%	DSP	%	DSP	%	DSP	%
656	23	410	14	176	7	176	7	1248	43	1064	38	704	25	704	25

On observe ainsi une tendance générale où SysGen optimise mieux les ressources DSP par rapport à HLS, et où la représentation en virgule flottante tend à utiliser moins de DSP que la virgule fixe grâce à la performance de notre architecture.

4.4 Conclusion

Dans ce chapitre, nous avons étudié l'implémentation de FSM sur FPGA, en utilisant deux approches complémentaires : la synthèse à haut niveau (HLS) et sur *System Generator*

(SysGen). Dans l'implémentation HLS nous avons présenté des tests préliminaires pour vérifier la faisabilité et optimiser les performances, ensuite on a décrit l'implémentation détaillée de l'algorithme montrant les défis et les solutions adoptées pour optimiser la vitesse et l'utilisation des ressources. On a ensuite étudié l'implémentation à l'aide de SysGen, dont l'ordonnancement a été une étape cruciale pour s'assurer que les calculs sont effectués de manière efficace et synchrone. L'architecture détaillée a été présentée, suivie des résultats et discussions,

Les résultats obtenus montrent que l'implémentation de FSM sur FPGA est non seulement faisable mais également très efficace en termes de temps de calcul et d'utilisation des ressources matérielles. L'approche HLS a permis une mise en œuvre rapide et flexible, bien adaptée aux tests et aux optimisations préliminaires. Cependant, l'utilisation de SysGen a offert des optimisations supplémentaires en termes de parallélisme et d'efficacité. Si l'on compare HLS et SysGen pour l'onduleur 2-L, on observe des écarts significatifs en termes de temps d'exécution. En virgule fixe, SysGen atteint un temps de calcul de 265 ns, contre 330 ns pour HLS. En virgule flottante, SysGen maintient une performance élevée avec un temps de 325 ns, tandis que HLS est plus lent avec 500 ns. Cela met en évidence la supériorité de SysGen dans l'exploitation du parallélisme et l'optimisation du temps de calcul, surtout pour les systèmes nécessitant une haute performance grâce à une gestion plus fine de l'ordonnancement et de l'architecture matérielle, et malgré une complexité de développement plus élevée, SysGen a permis d'atteindre de meilleure performance et de compromis en terme de latence et de ressources en exploitant pleinement les capacités de parallélisme du FPGA avec une performance maximale.

Conclusion

Ce travail de recherche a porté sur l’implémentation de la formule de Sherman-Morrison pour l’inversion matricielle en virgule fixe sur FPGA, en explorant deux volets distincts : La performance de la FSM en virgule fixe, incluant l'étude de la quantification, du facteur d'échelle et de la précision, ainsi que son implémentation matérielle sur FPGA avec une faible latence.

En premier lieu, l'évaluation de la précision des calculs en virgule fixe et flottante a révélé des différences significatives en termes de performances, présentant chacune des avantages selon les applications envisagées. Si la virgule flottante offre une meilleure précision pour des systèmes exigeants, la virgule fixe présente des avantages non négligeables en termes de rapidité et d'utilisation réduite des ressources, offrant ainsi un compromis optimal pour des applications avec des contraintes de ressources plus strictes.

Une étude détaillée de la dynamique des calculs de l'algorithme FSM, incluant l'exploration de techniques comme l'optimisation par le facteur d'échelle, visant à améliorer les performances globales. Nous avons prouvé que, même avec les limitations de la représentation en virgule fixe, il est possible d'obtenir une précision adéquate et de minimiser les erreurs grâce à des compromis dédiés.

L'implémentation de la FSM sur FPGA a été réalisée en utilisant Vivado-HLS et *System Generator* for DSP. L'approche en langage de haut niveau (HLS) a permis de bénéficier d'un développement accéléré. En revanche, l'utilisation de SysGen, qui permet un contrôle plus

précis du parallélisme, s'est avérée plus complexe mais plus adaptée aux applications nécessitant une optimisation avancée des ressources et des performances en temps réel.

Les résultats sont encourageants pour les applications qui requièrent des actualisations rapides des matrices inverses, en particulier dans les systèmes de simulation en temps réel. L'emploi de FPGA pour ces opérations répond aux besoins de faible latence et de grande précision, essentiels dans les contextes de simulation Hardware-In-the-Loop (HIL).

Des efforts additionnels pourraient être consentis pour optimiser les architectures FPGA pour d’autres algorithmes matriciels complexes, ainsi que pour explorer de nouvelles méthodes de correction d’erreurs afin d’améliorer encore davantage la précision des calculs en virgule fixe. En outre, l’adaptation de ces méthodes sur différentes plateformes matérielles pourrait ouvrir la voie à de nouvelles applications tout en renforçant leur robustesse et leur efficacité dans divers contextes.

Finalement, ce mémoire a présenté une implémentation de FSM sur FPGA selon deux outils, HLS et SysGen, ainsi qu’une arithmétique en virgule fixe et flottante. Un résultat quantifié a été présenté selon deux cas d’exécutions, onduleur 2-L et B2B.

Références

[1] J. Poupart, "Optimisations des performances de simulation en temps-réel sur FPGA d'un onduleur triphasé", Mémoire maitrise en génie électrique, UQTR, Août 2023.

[2] J. Poupart, M. Lemaire, D. Massicotte, "FPGA Implementation of Sherman-Morrison Formula," 38th Conference on Design of Circuits and Integrated Systems (DCIS), Malaga, Spain, 15-17 Nov. 2023.

[3] M. Lemaire, "Méthode et architecture générique sur FPGA pour la simulation temps réel d'électronique de puissance sur plateforme hétérogène", Thèse de doctorat et génie électrique, UQTR, Janvier 2024.

[4] A. Hadizadeh, M. Hashemi, M. Labbaf et M. Parniani, « Une technique d'inversion de matrice pour la simulation EMT en temps réel basée sur FPGA de convertisseurs de puissance », dans IEEE Transactions on Industrial Electronics , vol. 66, no. 2, pp. 1224-1234, février 2019, doi : 10.1109/TIE.2018.2833058.

[5] A. Burian, J. Takala, and M. Ylinen, "A fixed-point implementation of matrix inversion using Cholesky decomposition," presented at the 2003 46th Midwest Symposium on Circuits and Systems, 2003.

[6] S. K. Chetan Sa, Lekshmi Vb, Sudhakar Sb, Manikandan Ja,c*, "Design and Evaluation of Floating point Matrix Operations for FPGA based system design " presented at the Third International Conference on Computing and Network Communications (CoCoNet'19), 2019.

[7] J. M. R.-C. Hector Daniel Rico-Aniles, Jose de Jesus Rangel-Magdaleno, "FPGA-based Matrix Inversion Using an Iterative Chebyshev-type Method in the Context of Compressed Sensing," IEEE, 2014.

[8] C. I. a. O. Gustafsson, "On Fixed-Point Implementation of Symmetric Matrix Inversion," IEEE, 2015.

[9] M. A.-O. Elie H. SARRAF, and Daniel MASSICOTTE, "FPGA Design and Implementation of Direct Matrix Inversion Based on Steepest Descent Method," IEEE, 2008, doi: 10.1109/MWSCAS.2007.4488771

[10] Wenhua Ye et Huan Li, « Virtex-7 FPGA-based high-speed signal processing hardware platform design », 2013 IEEE 4th International Conference on Electronics Information and Emergency Communication , Beijing, Chine, 2013, pp. 113-116, doi: 10.1109/ICEIEC.2013.6835466.

[11] G. A. Kumar, T. V. Subbareddy, B. M. Reddy, N. Raju, and V. Elamaran, "An approach to design a matrix inversion hardware module using FPGA," presented at the 2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT), 2014.

[12] M. Kumm, O. Gustafsson, F. de Dinechin, J. Kappauf, and P. Zipf, "Karatsuba with Rectangular Multipliers for FPGAs," presented at the 2018 IEEE 25th Symposium on Computer Arithmetic (ARITH), 2018.

[13] P. Lamo, A. de Castro, A. Sanchez, G. A. Ruiz, F. J. Azcondo, and A. Pigazo, "Hardware in the Loop and Digital Control Techniques Applied to Single-Phase PFC Converters," Electronics, vol. 10, no. 13, 2021, doi: 10.3390/electronics10131563.

[14] E. Zamiri, A. Sanchez, M. Yushkova, M. S. Martínez-García, and A. de Castro, "Comparison of Different Design Alternatives for Hardware-in-the-Loop of Power Converters," Electronics, vol. 10, no. 8, 2021, doi: 10.3390/electronics10080926.

[15] Z. Li, P. Wang, Z. Wang, and J. Cheng, "Fixed-point Quantization for Vision Transformer," presented at the 2021 China Automation Congress (CAC), 2021.

[16] J. Y. S. Low and C. Chip-Hong, "A VLSI Efficient Programmable Power-of-Two Scaler " IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 59, no. 12, pp. 2911-2919, 2012, doi: 10.1109/tcsi.2012.2206491.

[17] M. H. Norbert Mitschke, "A Fixed-Point Quantization Technique for Convolutional Neural Networks Based on Weight Scaling," presented at the ICIP 2019, Taipei, Taiwan, 2019, 2019.

[18] A. Przybył, "Fixed-Point Arithmetic Unit with a Scaling Mechanism for FPGA-Based Embedded Systems," Electronics, vol. 10, no. 10, 2021, doi: 10.3390/electronics10101164.

[19] M. Rezaei Larijani and M. R. Zolghadri, "Design and implementation of an ADC-based real-time simulator along with an optimal selection of the switch model parameters," Electrical Engineering, vol. 103, no. 5, pp. 2315-2325, 2021, doi: 10.1007/s00202-021-01237-1.

[20] J. Vázquez-Castillo, A. Castillo-Atoche, R. Carrasco-Alvarez, O. Longoria-Gandara, and J. Ortegón-Aguilar, "FPGA-Based Hardware Matrix Inversion Architecture Using Hybrid Piecewise Polynomial Approximation Systolic Cells," Electronics, vol. 9, no. 1, 2020, doi: 10.3390/electronics9010182.

[21] M. G. B. Sumanasena, "A Scale Factor Correction Scheme for the CORDIC Algorithm," IEEE Transactions on Computers, vol. 57, no. 8, pp. 1148-1152, 2008, doi: 10.1109/tc.2008.41.

[22] N. Sorokin, "Implementation of high-speed fixed-point dividers on FPGA," presented at the JCS&T April 2006, 2006.

[23] Y. Lei, Y. Dou, Y. Dong, J. Zhou, and F. Xia, "FPGA implementation of an exact dot product and its application in variable-precision floating-point arithmetic," The Journal of Supercomputing, vol. 64, no. 2, pp. 580-605, 2013, doi: 10.1007/s11227-012-0860-0.

[24] Andre, Bannwart, Perina., Paulo, Matias., Eduardo, Marques., Vanderlei, Bonato., João, M., G., Lima. (2017). Exploiting Kant and Kimura's Matrix Inversion Algorithm on FPGA. 516-519. doi: 10.1109/DSD.2017.32

[25] Amin-Nejad, Siavash & Basharkhah, Katayoon & Asgari Gashteroodkhani, Tayyebeh. (2019). Floating Point versus Fixed point Tradeoffs in FPGA Implementations of QR Decomposition Algorithm. 10.24018/ejece.2019.3.5.127.

[26] Jia, Liu., Qiang, Liu. (2018). Resource Reduction of BFGS Quasi-Newton Implementation on FPGA Using Fixed-Point Matrix Updating. 301-306. doi: 10.1109/FPL.2018.00058

[27] A., Rosado., Taras, Iakymchuk., M., Bataller., Marek, Wegrzyn. (2012). Hardware-efficient matrix inversion algorithm for complex adaptive systems. 41-44. doi: 10.1109/ICECS.2012.6463562

[28] Hector, Daniel, Rico-Aniles., Juan, Manuel, Ramirez-Cortes., Jose, de, Jesus, Rangel-Magdaleno. (2014). FPGA-based matrix inversion using an iterative Chebyshev-type method in the context of compressed sensing. 983-987. doi: 10.1109/I2MTC.2014.6860890

[29] Javier, Vazquez-Castillo., Alejandro, Castillo-Atoche., Roberto, Carrasco-Alvarez., O., Longoria-Gandara., Jaime, Ortegón-Aguilar. (2020). FPGA-Based Hardware Matrix Inversion Architecture Using Hybrid Piecewise Polynomial Approximation Systolic Cells. Electronics, 9(1):182-. doi: 10.3390/ELECTRONICS9010182

[30] Jie, Zhou., Yong, Dou., Jianxun, Zhao., Fei, Xia., Yuanwu, Lei., Yuxing, Tang. (2009). A Fine-Grained Pipelined Implementation for Large-Scale Matrix Inversion on FPGA. Lecture Notes in Computer Science, 5737:110-122. doi: 10.1007/978-3-642-03644-6_9

[31] Adrian, Burian., Jarmo, Takala., M., Ylinen. (2003). A fixed-point implementation of matrix inversion using Cholesky decomposition. 3:1431-1434. doi: 10.1109/MWSCAS.2003