UNIVERSITÉ DU QUÉBEC

MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INGÉNIERIE

PAR

JEAN-GABRIEL MAILLOUX

PROTOTYPAGE RAPIDE DE LA COMMANDE VECTORIELLE SUR FPGA À L'AIDE DES OUTILS SIMULINK - SYSTEM GENERATOR

MARS 2008

Bibliothèque
Paul-Émile-Boulet

UQAC
Université du Québec
à Chicoutimi

Mise en garde/Advice

Afin de rendre accessible au plus grand nombre le résultat des travaux de recherche menés par ses étudiants gradués et dans l'esprit des règles qui régissent le dépôt et la diffusion des mémoires et thèses produits dans cette Institution, **l'Université du Québec à Chicoutimi (UQAC)** est fière de rendre accessible une version complète et gratuite de cette œuvre.

Motivated by a desire to make the results of its graduate students' research accessible to all, and in accordance with the rules governing the acceptation and diffusion of dissertations and theses in this Institution, the **Université du Québec à Chicoutimi (UQAC)** is proud to make a complete version of this work available at no cost to the reader.

L'auteur conserve néanmoins la propriété du droit d'auteur qui protège ce mémoire ou cette thèse. Ni le mémoire ou la thèse ni des extraits substantiels de ceux-ci ne peuvent être imprimés ou autrement reproduits sans son autorisation.

The author retains ownership of the copyright of this dissertation or thesis. Neither the dissertation or thesis, nor substantial extracts from it, may be printed or otherwise reproduced without the author's permission.

RÉSUMÉ

Avec la mécatronique, la recherche effectuée dans ce domaine touche nécessairement à plusieurs disciplines. Cela engendre des problèmes de recherche applicationnels très diversifiés. C'est pourquoi il est important d'établir une plateforme de travail à l'aide d'outils académiques modulaires qui permet de passer d'un algorithme ou d'un problème d'application vers un prototype fonctionnel à tester en laboratoire sur FPGA. Une méthodologie intégrant des outils comme Xilinx System Generator, SimPowerSystems et GAPPA offre un environnement de développement organique qui peut évoluer avec la direction de la recherche, tout en évitant les coûts d'une solution commerciale intégrée. De plus, l'accessibilité de ces outils et l'élaboration de cette méthode permettent d'éviter les coûts et le temps de formation du personnel nécessaire avec des outils commerciaux plus complexes. Dans un cadre académique, cela facilite l'intégration de nouveaux étudiants dans un projet de recherche.

La méthode de prototypage rapide est développée afin d'offrir à l'usager de la plateforme de développement une procédure optimale de conception, de validation, de co-simulation et d'optimisation du modèle. Une application de commande vectorielle des moteurs à induction devant répondre à des conditions d'exécution sévères (un court temps de réponse, une taille sur FPGA décente, etc.) est ici proposée afin de développer et tester cette méthode de conception. Les temps de développement sont évalués et la précision des résultats est analysée afin de confirmer la viabilité de cette méthode de prototypage.

REMERCIEMENTS

C'est avec plaisir que je prends ces quelques lignes, si peu nombreuses, afin de remercier les personnes qui m'ont soutenu durant ce projet de recherche.

De façon spéciale, je tiens à remercier mon directeur M. Rachid Beguenane qui a cru en moi et m'a permis de faire ce projet au sein de l'équipe de recherche en micro-électronique et traitement informatique des signaux (ERMETIS). Son support, son professionnalisme et surtout, sa disponibilité ont rendu possible la réalisation de ce travail. Également, je veux remercier mon collègue Stéphane Simard qui a dût répondre à un nombre incalculable de questions. Sans son savoir et sa connaissance des ressources d'information disponibles, la réalisation de ce travail eût été bien difficile.

Je remercie aussi ma charmante et jolie conjointe, qui ne cesse jamais d'être une source de réconfort et d'encouragement.

Table des matières

Table des figures

Liste des tableaux

CHAPITRE 1

Introduction et problématique

1.1 Introduction générale

Cette recherche s'inscrit dans le cadre d'une thèse de maîtrise en ingénierie, dont le sujet consiste à réaliser la commande vectorielle des moteurs à induction sur une seule puce reconfigurable tout en respectant une contrainte de temps de calcul de quelques microsecondes. Cela est fait pour remédier aux implémentations DSP (*digital signal processors)* nécessitant des temps d'exécution assez longs, notamment si l'on inclut l'algorithme de calcul PWM (*pulse-width modulation)* ainsi que des estimateurs complexes de grandeur difficilement mesurables. De plus, cette recherche vise la mise en place d'une plateforme de travail et d'une méthode de développement standardisées de manière à fournir une technique de conception optimale pour des applications futures.

1.2 Contexte et revue bibliographique

Actuellement, les fonctions algorithmiques sont généralement implémentées sur deux types de plateforme programmable, soit les DSPs et les FPGA *(field programmable gate arrays)*. Les FPGAs peuvent exécuter de multiples opérations en parallèle, ce qui leur donne un avantage sur les DSPs dans le domaine de la mécatronique comme indiqué dans [1]. L'intérêt de ce domaine d'ingénierie interdisciplinaire est de concevoir des systèmes automatiques puissants et de permettre le contrôle de systèmes hybrides complexes. Des plateformes DSP incluant une carte comme la DS1103 de dSPACE sont d'excellents outils encore aujourd'hui utilisés pour le prototypage rapide de contrôle, tel que dans [2] et [3]. Comme ces cartes utilisent des DSPs à virgule flottante avec un langage logiciel de haut niveau, ce sont des outils moins appropriés pour supporter le contrôle d'applications hautement modulaires. Avec des langages à description matérielle comme le VHDL et des outils de synthèse pour FPGAs reconfigurables peu coûteux, le

prototypage rapide d'algorithmes modulaires et complexes constitue une solution efficace déjà observable dans [4][5][6] .

Cependant, le développement sur ces puces n'est pas simple et une plateforme de travail intégrée devient nécessaire afin de concevoir, modéliser, simuler et optimiser. Comme il s'agit d'une plateforme de recherche, il est intéressant de vérifier la viabilité des outils académiques accessibles et maintenus indépendamment en assurant les liens entre chaque outil. De cette manière, chaque module de développement peut évoluer et faire bénéficier à la méthode des nouvelles fonctionnalités et performances sans délai important.

Pour le contrôle vectoriel, les modules clés comme l'estimateur de flux, la transformée de Park, les régulateurs PI *(proportional-integral)* et le générateur PWM sont particulièrement exigeants en calcul. Leur réalisation matérielle avec l'outil de conception DSP System Generator de Xilinx (XSG) œuvrant dans l'environnement Matlab/Simulink constitue le cœur de cette thèse. Particulièrement, la recherche s'intéresse à la méthode de conception qui assure la détermination rapide des précisions d'entrée et de sortie en termes de bits pour chacun des nombreux blocs constituant le contrôle vectoriel. L'intégration de l'outil GAPPA pour accélérer l'établissement et l'optimisation des précisions d'un modèle XSG au sein d'une méthode de prototypage n'a pas, au mieux de notre connaissance, de précédents dans la littérature. Un exemple du type d'utilisation actuelle de GAPPA est présenté dans [7].

Pour le design et la simulation, les plateformes commerciales offertes par les compagnies canadiennes RTDS Technologies [8] et OPAL-RT [9] proposent des performances intéressantes, mais les coûts sont très élevés. De plus, ces solutions intégrées force l'usager à ne pas profiter de la flexibilité de gérer des outils académiques indépendamment. Comme alternative aux coûts importants des systèmes commerciaux, les auteurs de [10][11] proposent une solution peu coûteuse qui se nomme VTB-RT *(Virtual Test Bed Real Time)*. Cette méthode utilise du matériel commun (des PCs ordinaires) et le système d'exploitation Linux. Il est cependant nécessaire d'effectuer la configuration de ces systèmes et de migrer les modèles vers ces outils, ce qui force l'usager à dépendre une fois de plus d'une solution d'un tiers parti. La méthode de développement de cette thèse va plutôt regarder l'intérêt d'intégrer les modèles XSG à des

modèles SPS (*SimPowerSystems)* dans l'environnement Simulink et de simuler à l'aide du mode de co-simulation appelé « co-simulation libre » qui utilise un FPGA dans la boucle.

Les résultats de cette thèse serviront comme base de conception pour le prototypage rapide de toutes sortes d'algorithmes de contrôle haute performance déjà existants ou à venir, ainsi que d'autres applications diverses (MEMS, biologie, etc.).

1.3 Problématique

Cette recherche vise le développement d'un contrôle vectoriel avec PWM et génération d'ondes entièrement sur FPGA, ainsi que d'une plateforme de travail pour toutes les phases du développement à l'aide d'outils académiques accessibles et gérés individuellement. La méthode de prototypage rapide doit être suffisamment flexible pour être réutilisée lors de projets futurs de mise en œuvre d'algorithmes complexes, et ce, tout en exploitant la plateforme de travail de façon optimale.

Le temps d'exécution du contrôle vectoriel ne doit pas dépasser quelques µs, même si la physique de l'électronique de puissance ne nous fournit qu'une contrainte de l'ordre de 25 µs pour des raisons de dissipation d'énergie chez les modules de puissance [12]. En respectant une condition temporelle aussi sévère, on démontre qu'il est possible d'utiliser cette technique pour des algorithmes encore plus complexes. Aussi, cette méthode de développement anticipe les avancements technologiques au niveau de l'électronique de puissance qui permettraient alors de profiter d'un contrôle d'une telle rapidité.

La méthode proposée couvre la dernière étape pour un prototype jugé fonctionnel après simulation, soit l'intégration avec le matériel physique pour tests en laboratoire. Pour ce faire, le prototype doit être placé dans la logique du FPGA (Virtex II pro) contenu sur la carte PCI Amirix AP1000, fournie au laboratoire par la Canadian Microelectronics Corporation (CMC). Cette carte communiquera avec la carte AIO PMC66-16AISS8AO4 de General Standards Corporation pour recevoir les signaux de l'électronique de puissance. La communication entre les deux cartes

s'effectue via l'interface PMC. Personne n'ayant précédemment développé une application pour la carte Amirix utilisant la communication PMC avec une carte d'entrée/sortie, c'est en collaboration avec mon collègue Stéphane Simard que le pilote logiciel et la logique permettant la communication en question sont développés. La partie logique (sur FPGA) sera discutée en détail dans cette recherche puisque son développement est requis pour assurer le bon fonctionnement de la plateforme de travail. Cette logique supplémentaire est développée dans cette recherche afin de communiquer avec le pilote logiciel ainsi que la carte AIO. L'achat de cette carte AIO a d'ailleurs influencé CMC dans le choix d'une carte d'acquisition à coupler à la carte Amirix, et notre système de communication entre ces cartes fait l'objet d'un contrat entre CMC et l'UQAC.

Le prototypage rapide de la commande vectorielle sur FPGA présente avant tout une problématique de méthodologie. Pour un système simple en boucle ouverte, l'implémentation, l'analyse des résultats et la correction du modèle posent peu de difficultés, car les entrées et les sorties peuvent rapidement être comparées aux résultats souhaités d'un modèle de référence. L'opération est plus difficile lors du prototypage rapide d'un système complexe en boucle fermée. La nature modulaire d'un système complexe développé avec les outils Simulink - XSG (dont les licences sont fournies par CMC) demande le débogage individuel des différents modules. Cependant, les entrées et les résultats d'un bloc dépendent nécessairement des autres blocs du modèle. Les sorties du système y sont réinjectées et affectent les résultats subséquents. Comme il n'est pas possible de tester chaque module individuel en boucle fermée immédiatement après sa conception, la technique de prototypage rapide doit tout de même permettre de vérifier la validité des blocs avant l'assemblage final.

La méthode de prototypage doit également fournir une manière d'évaluer la précision du modèle. Comme les valeurs de chaque module devront être comparées à un résultat théorique idéal provenant du modèle de référence Matlab en virgule flottante, cette comparaison doit être juste. Cela serait facile à vérifier s'il s'agissait d'un système en boucle ouverte, mais dans le cas d'une boucle fermée, les valeurs désirées doivent être les mêmes à l'entrée et à la sortie du bloc. En résumé, la méthode doit traiter chaque bloc en boucle ouverte, mais en lui injectant des signaux de boucle fermée provenant du modèle de référence. Une fois l'assemblage du prototype terminé, la réponse de la commande vectorielle est comparée à la référence, et les résultats sont

analysés. Il y aura nécessairement des variations entre les résultats, surtout à l'initialisation du système, mais l'analyse doit en tenir compte et juger si le prototype offre la réponse souhaitée, avec la précision demandée. Aussi, la précision doit être évaluée et ajustée en fonction de tous les cas possibles pour une certaine gamme de valeurs (ex : pour couvrir les paramètres d'un large éventail de moteurs différents).

La méthode de prototypage rapide doit donc satisfaire les caractéristiques suivantes :

- la **généralité** (pour être facilement adaptée à un autre algorithme de contrôle ou à un autre moteur);
- la **flexibilité d'analyse** (pour produire des résultats différents, comme la précision des calculs, avec un minimum de modifications);
- l'**exactitude des résultats** (pour la réponse du système et la précision).

Une fois la méthode de prototypage développée, l'obtention des résultats pour l'analyse et le débogage se fait par simulation. Ici encore, la complexité du contrôle pose problème. Si l'on désire effectuer le prototypage d'un contrôle dont le temps de calcul est très court, on doit aussi utiliser des vitesses d'échantillonnage très courtes. Le nombre de calculs fait ainsi croître le temps de simulation. Un exemple chiffré donne une bonne idée du problème.

Soit un contrôle nécessitant 200 coups d'horloge. À une vitesse de 100 MHz, on obtient un temps de contrôle de 2 µs. Si un contrôle doit être déclenché à toutes les 25 µs, il faut $\frac{5s}{25\mu s} \times 200 = 40\,000\,000$ coups d'horloge simulés pour observer les résultats du moteur pendant 5 secondes. Le contrôle comporte un nombre élevé d'opérations mathématiques et une grande précision (un grand nombre de bits pour la partie entière et fractionnaire des opérandes), ce qui le rend trop lent (plusieurs jours) à simuler par des logiciels comme Matlab - Simulink. De plus, ces tests impliquent qu'une quantité massive de résultats soit accumulée dans Matlab, ce qui a pour conséquence d'alourdir davantage les opérations. On trouve une solution partielle à la lenteur des simulations dans la co-simulation offerte par XSG qui utilise le FPGA pour effectuer le travail du prototype. Il est une fois de plus nécessaire de trouver une manière optimale d'intégrer cet outil à la méthode de prototypage rapide, car la préparation de chaque co-simulation est relativement longue. Bien que Xilinx fournisse des exemples de modèles pour illustrer le gain de temps avec la

co-simulation, il n'est pas possible de trouver une étude dans la littérature qui nous permet de bien estimer les temps impliqués dans l'utilisation de cet outil. La recherche permet de mieux estimer les cas où la co-simulation XSG est un bon choix et le temps que cela va impliquer dans le processus de développement. La plateforme de co-simulation utilisée est la carte Xilinx ML402 fournie par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG).

Le contrôle ne doit pas seulement être simulé à l'aide de valeurs théoriques. Il faut pouvoir le tester dans un environnement réel, c'est-à-dire, en le reliant à un moteur. Évidemment, vu le coût des équipements et la sécurité du personnel, il serait illogique et totalement irresponsable de brancher le prototype au moteur et d'observer ce qui se passe. Ce moteur doit avant tout être simulé, et c'est le toolbox SPS de Matlab-Simulink qui sera employé à cet effet. Cependant, le couplage des blocs SPS et XSG n'est pas une simple question de branchements. Afin de simuler les deux entités simultanément, les contraintes temporelles doivent être considérées. Le moteur est simulé en termes de temps, alors que le prototype XSG est simulé en pas d'horloge. Un couplage temporel doit donc se faire pour assurer une bonne simulation. Une fois ce couplage achevé, l'intégration des outils de co-simulation de XSG devient critique pour accélérer la simulation.

Les moteurs et modules de puissance (CRSNG) sont installés au laboratoire ainsi que les filtres, une charge résistive qui permet de varier les couples de charge et les connecteurs manquant pour assurer la communication entre tous les appareils.

1.4 Objectifs et méthodologie de recherche

La recherche est amorcée par une revue de la littérature concernant le contrôle vectoriel, les implémentations de commande des moteurs électriques sur FPGA, et l'implémentation *space vector PWM* (SVPWM) sur FPGA. En parallèle, les détails de la méthode de prototypage rapide sur FPGA sont définis de manière brouillonne. Pendant ce temps, l'outil XSG est exploré en détail afin de bien maîtriser ses fonctionnalités.

Avec les étapes de la préparation complétées, le développement du premier prototype du contrôle vectoriel est amorcé en se basant sur les équations théoriques de ce type de contrôle (voir annexe A). La méthode de prototypage rapide est modelée afin d'obtenir des résultats cohérents et de perdre un minimum de temps pendant le développement. Les erreurs commises et les outils non appropriés permettent de particulièrement bien cerner le flux optimal de conception. Pour le contrôle vectoriel, les blocs individuels sont développés et testés, ainsi que l'ensemble du modèle. La réponse du système obtenue est comparée à celle du modèle de référence de commande vectorielle, et la précision est optimisée en conséquence. Cette étape d'ajustement de la précision du modèle avec l'outil GAPPA (section 4) devient un point critique de la méthode avant d'entamer le processus de développement.

Une fois le prototype apte à respecter toutes les conditions temporelles, il peut être synthétisé et ensuite chargé dans un FPGA. L'outil de co-simulation de XSG est utilisé pour remplacer le modèle du prototype dans Matlab-Simulink par un bloc unique qui communique avec le FPGA via un lien ETHERNET.

Une version SPS de l'équipement (moteur et module de puissance) est conçue pour connecter au bloc du prototype à co-simuler. Pour une première estimation du modèle en conditions physiques réelles, le contrôle théorique (une version modifiée du modèle Matlab de référence) est couplé au moteur virtuel afin d'estimer la réponse attendue. Lorsque le moteur fonctionne bien avec ce contrôle, celui-ci est remplacé par la version XSG (le prototype). Ici, un travail est effectué afin de respecter les contraintes temporelles, car les deux systèmes sont vus différemment par Matlab-Simulink. Le prototype est simulé en termes de pas d'horloge, et le moteur en temps effectif.

Pour les étapes précédentes du développement, la co-simulation matérielle ne peut être employée, car un grand travail de débogage doit encore être effectué. Pour éviter que les simulations s'éternisent pendant des semaines, une précision de contrôle moins agressive est utilisée pendant les tests du système de commande vectorielle (pour la génération de signaux PWM). L'idée est d'obtenir de bonnes courbes de réponse sans toutefois s'occuper de la précision qui sera ajustée plus tard.

Lorsque le couplage est fonctionnel, la co-simulation matérielle est utilisée pour accélérer le temps de calcul, afin de vérifier une plus grande plage de résultats et d'évaluer la précision de ces derniers.

Comme le prototype est développé pour FPGA, l'espace et le temps sont considérés. Pour l'espace, la cellule logique (CL) est l'unité de mesure utilisée. Pour le temps, deux échelles sont nécessaires; lorsque l'ensemble du système est considéré, incluant moteur, contrôle et PWM, c'est la µs, et pour les composants implémentés strictement sur le FPGA, les coups d'horloges font office de référence.

En ce qui concerne le prototypage, plusieurs informations doivent être compilées. Les temps d'exécution de chaque composant du contrôle, en coup d'horloge, sont déterminés en analysant le chemin critique d'un bloc et en confirmant par simulation. Les ressources du FPGA nécessaires au bloc sont évaluées grâce à un outil d'estimation de XSG qui fournit un résultat après synthèse, placement et routage. Finalement, des estimations de temps de simulation sont calculées selon les diverses techniques utilisées. Toutes ces informations sont aussi amassées pour le système en entier, une fois qu'il est fonctionnel et débogué. Avec ces statistiques, il devient possible de fournir une échelle réaliste pour accompagner les étapes de la méthode de prototypage, facilitant l'estimation en gestion de projet. Cela permet par exemple au développeur utilisant la méthode de prédire à l'avance la taille du FPGA nécessaire à son application, et d'approximer le temps de développement que le projet demandera. Cela donne aussi une idée de l'effort qui doit être fournit dans l'optimisation du prototype aussi bien en termes de densité logique que de temps d'exécution.

CHAPITRE 2

Algorithme et plateforme

2.1 Introduction et schématisation de l'algorithme

Avant d'amorcer le développement du prototype, il faut étudier l'algorithme utilisé. Dans le cas présent, il s'agit de la commande vectorielle. La recherche bibliographique permet d'identifier des articles et des sources qui expliquent la commande afin d'en tirer les équations mathématiques. On peut alors schématiser la commande vectorielle sous forme de boîtes connectées par des fils. Chacun de ces blocs va produire des sorties en appliquant les entrées aux équations du bloc.

2.1.1 Contrôle vectoriel

Le choix du contrôle vectoriel est intéressant comme application utilisée afin de créer une méthode de prototypage optimale. C'est sa complexité et sa modularité qui offrent un défi intéressant ainsi qu'un bon terrain d'évaluation pour des outils de co-simulation. La présence d'une racine carrée et d'une division offre aussi l'avantage de pouvoir appliquer et vérifier l'implémentation de ces opérations selon un modèle non restaurant plus économique en terme de ressources logiques (voir section 3.4). Les résultats de l'implémentation de ces opérations ont permis de publier deux articles sur le sujet que l'on retrouve à l'annexe I.

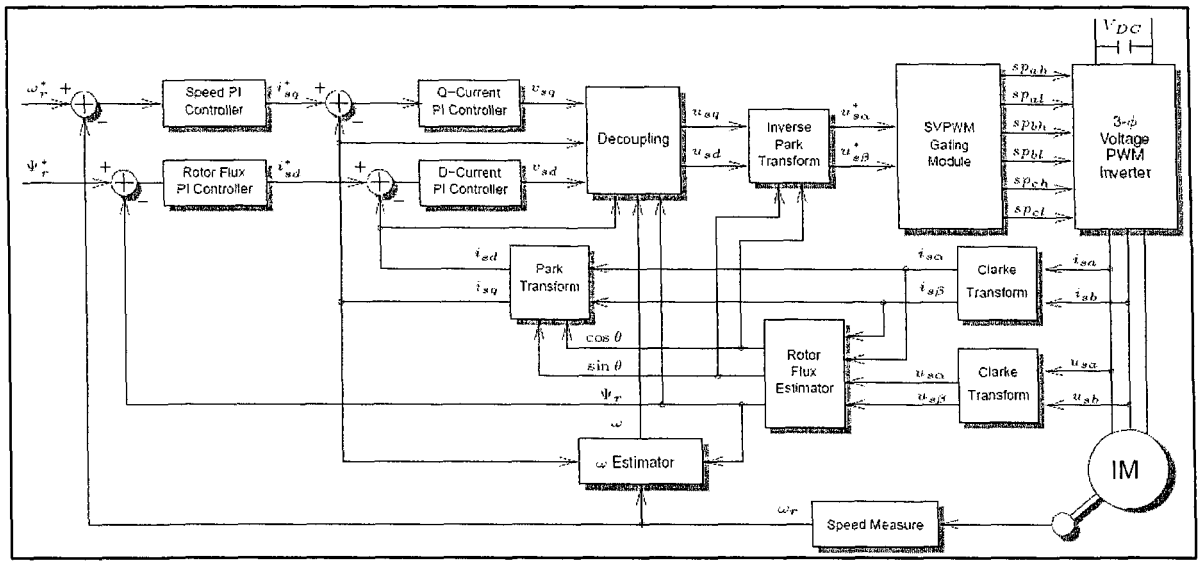


figure 1 - Contrôle vectoriel

La figure 1 montre comment le contrôle vectoriel est scindé sous forme modulaire. Les équations de chaque module se retrouvent à l'annexe A. Dans cette recherche, la lecture de la vitesse se fait avec capteur.

2.1.2 Survol du portillonnage (délais d'activation et temps de maintien)

Avant la venue des larges FPGAs disponibles aujourd'hui, les méthodes utilisées, incluant l'utilisation de DSPs seuls ou en combinaison avec des microcontrôleurs spécialisés, permettaient un temps de cycle total de plus de 100 µs. Il n'y avait alors pas de manière efficace d'implémenter le SVPWM, ce qui causait l'obtention d'une période de commutation aux alentours de 1-5 kHz, produisant ainsi du bruit audible.

L'introduction de FPGAs plus puissants durant les années 90 a permis pour la première fois de diminuer la quantité de traitement à gérer par le DSP à l'aide de coprocesseurs PWM sur un FPGA de petite taille. Dans le domaine du contrôle de moteur, cela a permis d'augmenter la fréquence de commutation à 10-15 kHz, réduisant nettement le bruit perceptible.

Avec les FPGAs d'aujourd'hui, un contrôle large et complexe comprenant le générateur SVPWM au complet peut se trouver sur une seule puce, tout en supportant des fréquences de l'ordre de 50 kHz. Il serait même possible d'obtenir des fréquences aussi élevées que 100 kHz (transistors MOSFET), mais l'utilité d'une telle vitesse demeure pour l'instant questionnable vu que la dissipation d'énergie est proportionnelle à la fréquence de commutation et que le courant généré par les transistors de puissance (IGBT, GTO) diminue selon la même proportion. Même à 20 kHz, la dissipation d'énergie est importante.

Par exemple, l'équipement utilisé dans notre laboratoire supporte, selon le manufacturier, une fréquence de commutation maximale de 20 kHz. Cependant, il est recommandé de ne pas utiliser une fréquence au-delà de 16 kHz sauf si absolument nécessaire, comme dans le cas d'un contrôle nécessitant une très large bande. Une fréquence de 16 kHz offre l'avantage d'une tension de sortie avec moins de distorsions causées par l'insertion du temps mort. Les simulations du contrôle vectoriel en boucle fermée de cette recherche sont donc paramétrées afin de reproduire une telle fréquence.

Ces désavantages limitent l'intérêt d'utiliser des onduleurs à très haute fréquence de commutation. Un exemple d'application où la fréquence doit être limitée est l'opération de gros moteurs industriels lorsqu'un fort courant est requis de l'onduleur. Pour des applications sur plateformes autonomes (ex. : robots mobiles), c'est la dissipation d'énergie qui doit être minimisée. Une fréquence de 16 kHz semble donc largement suffisante pour la plupart des applications de contrôle, tout en demeurant à l'extérieur de la bande de fréquence audible.

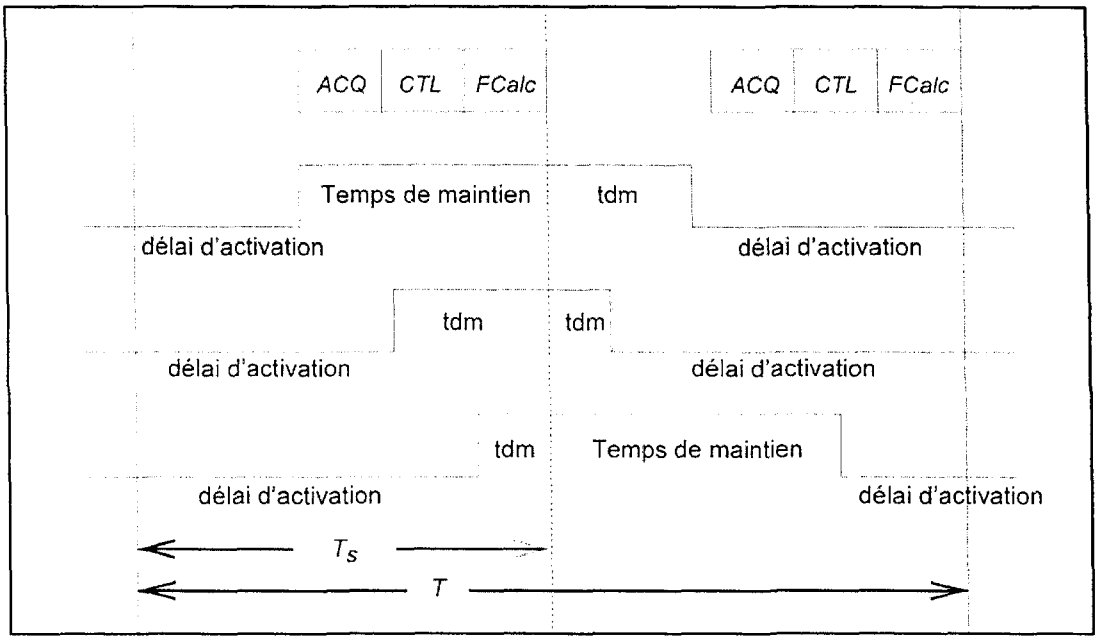


figure 2 - Portillonnage PWM asymétrique

Ces contraintes sont prises en considération lors de la maximisation de la fréquence d'échantillonnage et de contrôle. Afin d'optimiser cette dernière sans augmenter la fréquence de commutation, un portillonnage PWM asymétrique est employé. Ceci permet d'obtenir une période d'échantillonnage T_s qui est égale à la moitié de la fréquence de commutation T. La figure 2 illustre le cycle d'acquisition (ACQ), de contrôle (CTL) et de calcul des délais d'activation (FCalc) qui sont observables à la fin de la période T_s. Le délai d'activation T_f représente le temps nécessaire avant d'allumer les transistors de puissance, alors que le temps de maintien T_d représente le temps où ces transistors continuent de conduire. Ces temps suivent l'équation $T_f + T_d = T_s$.

Même s'il existe sur le marché des transistors de puissance qui atteignent presque 100 kHz (MOSFET), les calculs de ce travail seront effectués en considérant une fréquence de commutation de 16 kHz exigée par le module de puissance IGBT utilisé dans cette recherche. Cette fréquence offre un temps de commutation de 62.5 µs, soit un échantillonnage de 31.25 µs. Il est important de s'assurer d'avoir suffisamment de temps pour l'achèvement du cycle de temps T_{cycle} qui se représente selon l'équation $T_{cycle} = T_{acq} + T_{ctl} + T_{Fcalc}$. La section 3.3 illustre que le

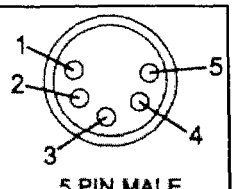
$T_{ctl} + T_{Fcalc}$ nécessite environ 2 µs. Avec des convertisseurs analogues numériques (ADC) qui supportent 1 million d'échantillons par seconde (MSPS), soit $T_{acq} = 1$ µs, cela donne un $T_{cycle} = 3$ µs, ce qui est très rapide.

Ceci veut dire que la technologie permet de réduire le temps de calcul d'un algorithme complexe de contrôle à quelques µs. Des algorithmes divers pourront donc être modifiés et adaptés afin de bénéficier de cette accélération. Un exemple est illustré à la figure 2 où l'utilisation d'un générateur SVPWM asymétrique permet d'effectuer deux cycles de contrôle complets au lieu d'un seul à l'intérieur d'une période de commutation tout en respectant la physique.

2.1.3 Lecture de vitesse provenant d'un encodeur optique

Le moteur utilisé comporte un encodeur optique afin de fournir l'information concernant la vitesse de rotation actuelle de l'engin. Il y a 5 sorties observées, soit : A, B, INDEX, +5 et GND. Chaque signal correspond à un fil de couleur, illustré à la figure 3.

PIN #	SIGNAL	COLOR
1	A	GREEN
2	B	ORANGE
3	INDEX	WHITE
4	+5VDC	RED
6	GND	BLACK



5 PIN MALE CONNECTOR (ENCODER)

figure 3 - Fils de l'encodeur optique

L'encodeur optique traduit la position angulaire en signal électrique. Pour chaque rotation du moteur, 1000 lignes d'encodage produisent 1000 changements d'état sur les lignes A et B. Donc, plus la période d'un signal sur A ou B est longue, plus le moteur tourne lentement. Le retard entre les signaux A et B permet quant à lui de déterminer la direction de rotation de l'engin.

En cherchant dans la littérature, on trouve l'article [13] qui propose une méthode de lecture de signaux d'un encodeur optique optimale sur FPGA. Les lectures de vitesses sont alors précises pour un grand nombre de scénarios vu la possibilité d'ajuster la méthode de calcul selon la fréquence à traiter. En basse vitesse, l'implémentation utilise une méthode de comptage de période alors qu'en haute vitesse, c'est un comptage de fréquence.

Dans le cadre de ce travail, un simple décodeur sera employé afin de pouvoir plus rapidement passer à la phase de tests. Ce décodeur ne va pas changer, qu'il opère en basse ou en haute vitesse. Vu la présence de 1000 changements d'état par rotation, un intervalle de 1 ms est observé par le FPGA, et la vitesse moyenne est estimée en calculant la valeur de rotation par minute (RPM) correspondante à l'aide de l'équation $X \times \frac{60\ s}{1000\ lignes} = Y\ rpm$ où X est le nombre de lignes détecté par seconde.

Cette méthode de lecture moyenne agit comme un petit filtre sur la vitesse mesurée. Un échantillon de 1 ms est suffisant pour la dynamique du moteur qui sera testé en laboratoire, alors que le filtrage devrait empêcher les vibrations de trop affecter les résultats.

2.2 Référence Matlab de l'algorithme

Même en connaissant les équations qui forment l'algorithme, il est nécessaire de vérifier la validité du prototype. Il faut établir une référence Matlab de l'algorithme; une version fonctionnelle de l'algorithme de contrôle dont les résultats peuvent être accumulés dans le logiciel Matlab. Comme les données numériques sont faciles à importer dans l'environnement Matlab, nos données de référence peuvent être générées dans n'importe quel logiciel qui peut exporter ses résultats. Souvent, cette référence sera le résultat d'une autre recherche ou de l'aboutissement d'autres travaux. Dans le cas présent, un modèle de commande vectoriel développé sous Matlab par le professeur Rachid Beguenane [14] sera utilisé comme référence. Le fait d'établir une telle référence a une conséquence importante pour la réalisation des objectifs : si le prototype donne, avec les mêmes entrées que pour la référence, des sorties qui approchent

celles de la référence en respectant une contrainte d'erreur déterminée, le prototype peut être optimisé et co-simulé. En d'autres termes, le prototype est considéré comme étant valide.

2.2.1 Contrôle Vectoriel

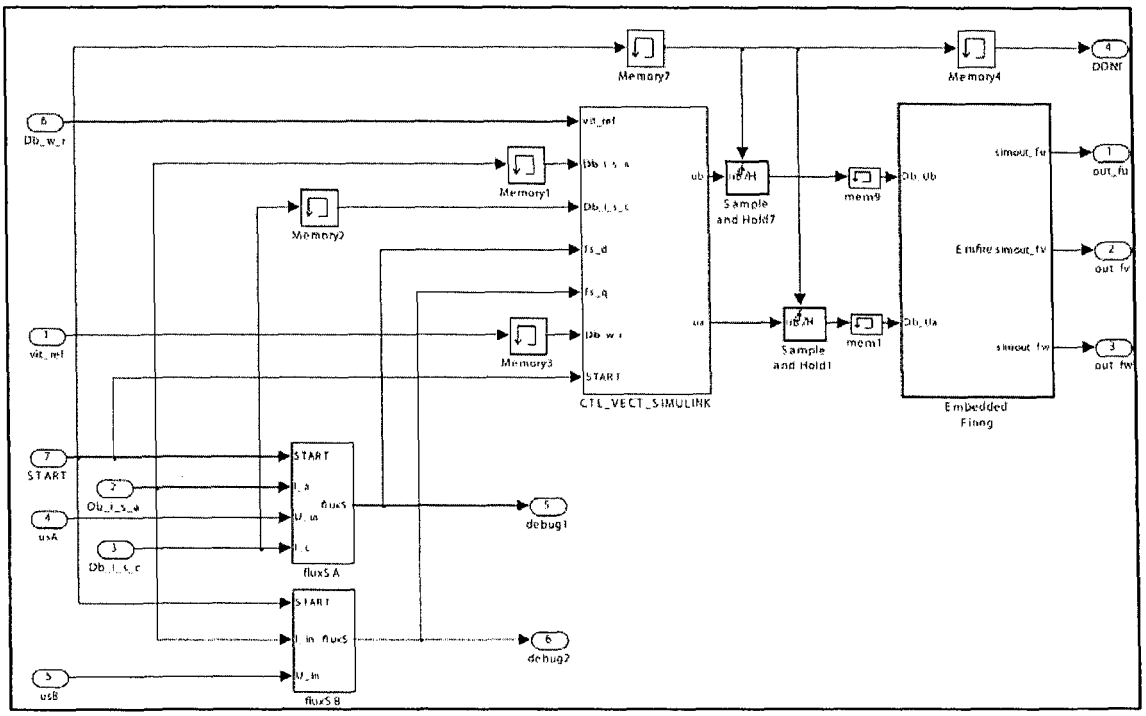


figure 4 - Interface du contrôle Simulink (référence)

La figure 4 représente l'implémentation de référence Simulink du contrôle vectoriel basé sur le modèle de [15]. Comme il s'agit d'une référence, la précision maximale de Matlab est utilisée. L'interface à la sortie et à l'entrée du bloc qui contient le modèle correspond exactement à l'interface de contrôle vectoriel implémenté en utilisant l'outil XSG. Ceci permet de facilement substituer ces systèmes dans l'environnement de test qui comprend les engins SPS ainsi que le bloc de portillonnage (voir section 4.9).

2.2.2 Portillonnage (délai d'activation et temps de maintien)

Pour le calcul des délais d'activation ainsi que pour le portillonnage, la référence Matlab est facilement transférable vers un bloc de fonction embarquée (code à l'annexe C dans l'environnement Simulink. Ces blocs (figure 5) sont utilisés dans la simulation en boucle fermée de la référence Simulink couplée aux équipements modélisés par SPS. De plus, le bloc portillonnage sera réutilisé lors de la co-simulation du contrôle vectoriel XSG car il s'agit d'un processus séparé qui ne peut être co-simulé dans le cadre de cette application (voir explication à la section 4.8).

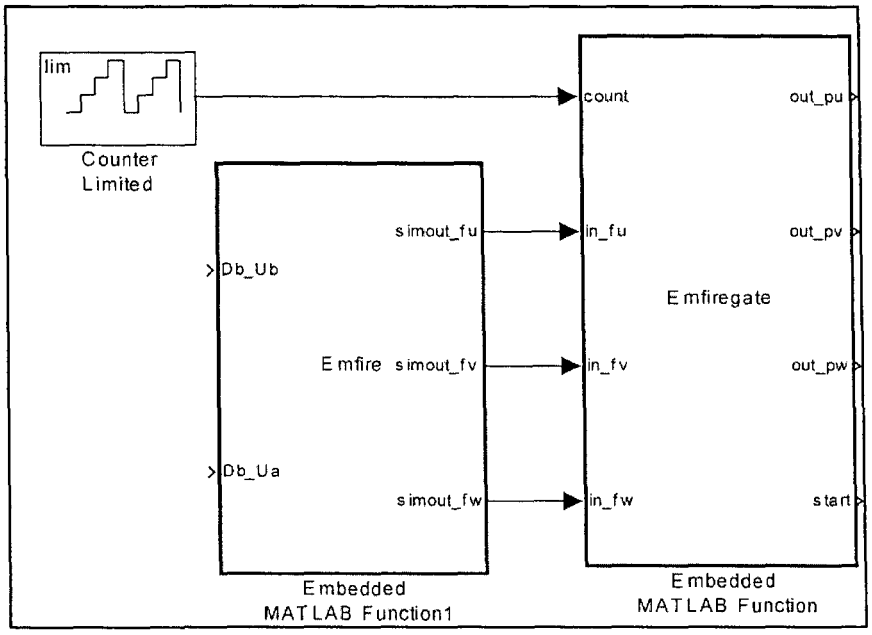


figure 5 - Calcul des délais d'activation et portillonnage (embedded Matlab functions)

2.2.3 Moteur et électronique de puissance

Le toolbox SPS est utilisé afin de modéliser le moteur et le module de puissance dans un environnement de contrôle vectoriel complet. Le développement du modèle XSG de contrôle vectoriel dépend fortement des paramètres utilisés pour identifier le moteur SPS simulé. C'est

pourquoi les paramètres de ce moteur partagent la nomenclature des variables utilisées pour le modèle XSG. Il n'y a donc pas de fichier d'initialisation particulier pour le bloc SPS illustré à la figure 6.

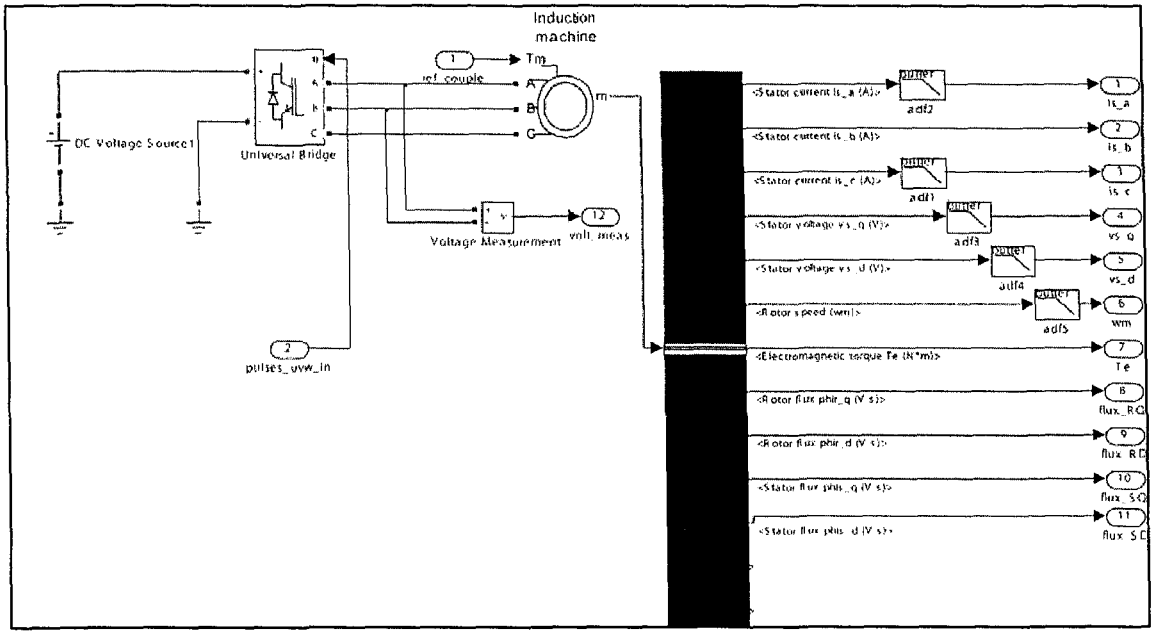


figure 6 - Module de puissance et moteur à induction (SPS)

Lors du développement, un premier ensemble de paramètres est utilisé qui correspond au moteur retrouvé dans un tutoriel de SPS. Ce moteur est choisi pour faciliter le test du portillonnage en comparant les résultats avec ceux du tutoriel SPS de SVPWM. Une fois le modèle XSG fonctionnel, les paramètres du moteur disponible en laboratoire qui sera la cible de tests physiques sont utilisés. Ces paramètres sont illustrés dans le tableau 1.

Moteur Motorsoft 2928	
Puissance	250 W
Vitesse	4000 RPM
Voltage	42 V
P_p	1
R_S	8.1 Ω
L_S, L_R	0.8155 H
R_R	3.8048 Ω
J	0.0014 kg m^2

tableau 1 - Paramètres du moteur en laboratoire

Un des objectifs de la plateforme de développement établie dans cette recherche étant la versatilité du modèle généré, le changement de paramètres entre deux moteurs si complètement différents démontre la flexibilité offerte par la plateforme de travail. Cela démontre aussi que l'optimisation n'est pas trop agressive, ce qui limiterait l'étendue des valeurs que peuvent prendre les paramètres du moteur.

Les tensions et les courants sont filtrés avant leur introduction au contrôle vectoriel. Lors du développement du modèle, des filtres d'ordre 4 et de fréquence de 5000 Hz sont utilisés afin de faciliter le débogage du modèle et de mieux évaluer la précision de sortie de celui-ci. Une fois le modèle assez performant selon les standards du développement, des filtres de fréquence de 500 Hz sont utilisés, ce qui correspond à ceux fabriqués par le technicien de département pour le montage en laboratoire.

2.3 Collecte des données de la référence

La dernière étape préparatoire en ce qui concerne l'algorithme consiste à effectuer la collecte des données de la référence. Comme le prototype XSG est développé bloc par bloc, ces derniers

sont nécessairement reliés par des signaux intermédiaires. Ceux-ci sont bien identifiés lors de la schématisation de l'algorithme, mais ils peuvent être difficiles à identifier dans le modèle de référence. En effet, ce dernier peut être un code complexe sans forme graphique qui se prête mal à l'identification rapide de signaux. Il faut malgré cela isoler chacun de ces signaux (si possible) dans la référence et accumuler les résultats pendant une simulation (figure 7). Ces données seront utilisées pour tester individuellement chaque bloc du prototype avec XSG. Vu la complexité de l'algorithme, il est clair que la quantité de données produites sera énorme. Heureusement, il n'est pas nécessaire d'accumuler une trop grande plage de données pour les signaux intermédiaires. Il s'agit ici de vérifier que nos équations reflètent le comportement de la référence, sans plus. Les cas extrêmes (valeurs trop grandes, trop petites, dépassements, etc.) seront gérés plus tard dans le projet à l'aide de l'outil GAPPA (voir section 4.2). Cependant, il faut être plus rigoureux pour l'ensemble du système. Il faut simuler plus longtemps en accumulant les entrées et sorties de l'ensemble de l'algorithme, sans les signaux intermédiaires (figure 8). Dans le cas de la commande vectorielle, un profil de vitesse et de charge est utilisé pour tester le contrôle. La vitesse de référence est changée, ainsi que la charge, et les résultats sont notés. Ceci présente l'avantage de pouvoir afficher la réponse du système sous forme de graphiques qui peuvent ensuite être comparés avec ceux du prototype.

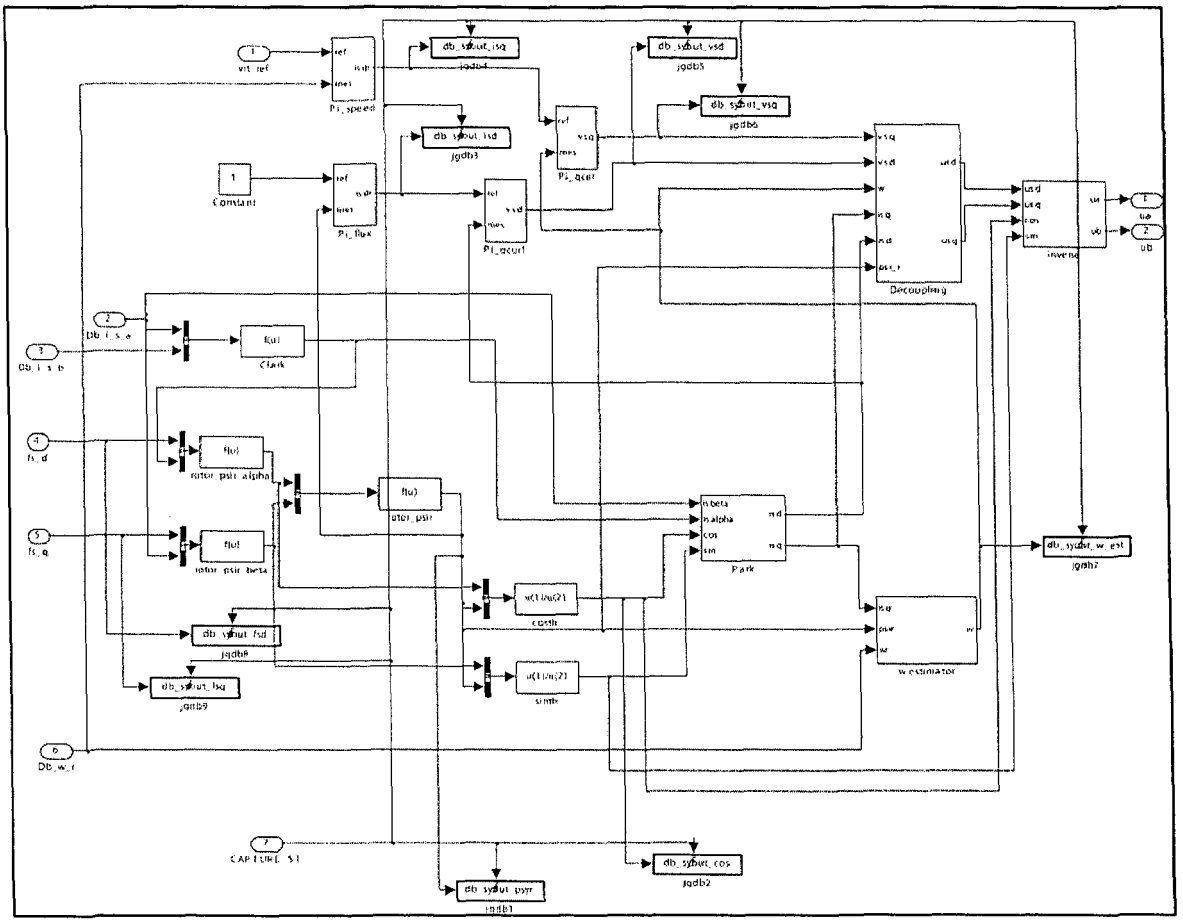


figure 7 - Collecte des données de référence (Contrôle Simulink)

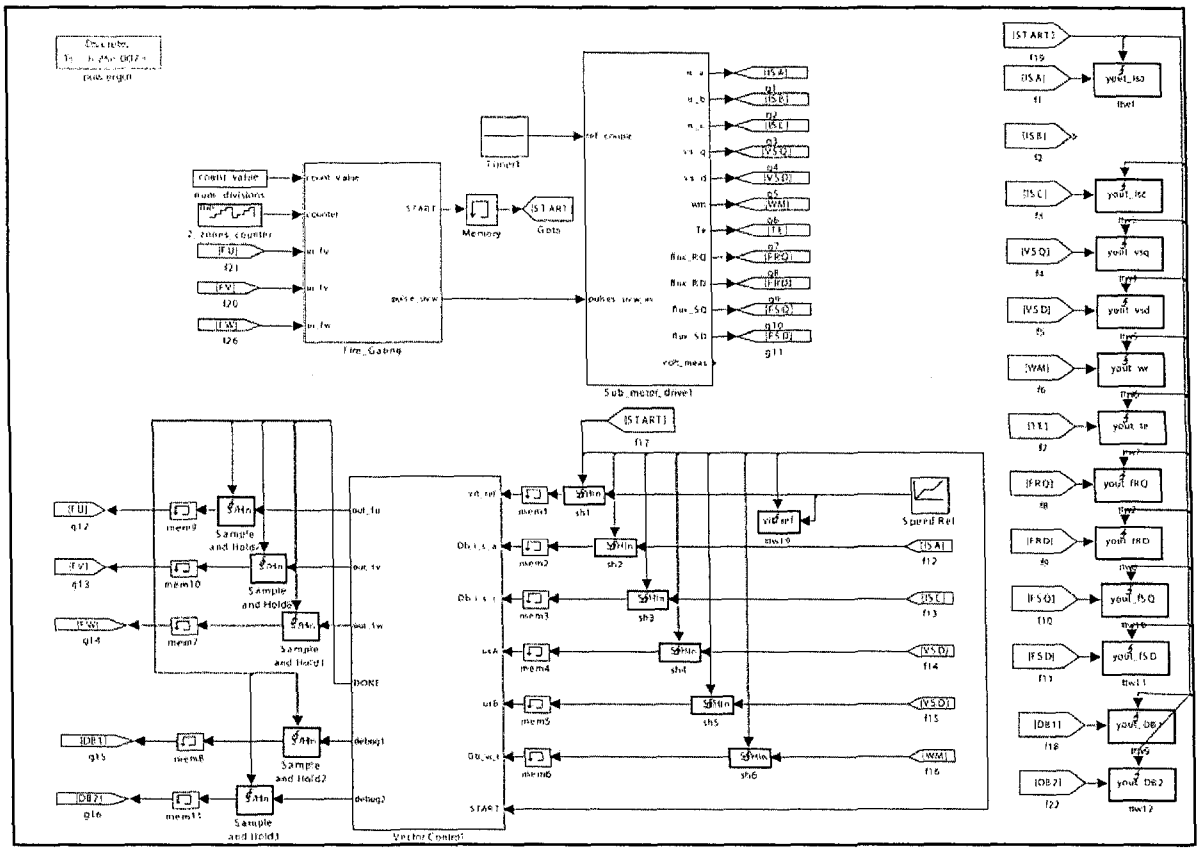


figure 8 - Collecte des données de référence (réponse du moteur)

2.4 Plateforme Amirix

Bien qu'il s'agisse de la dernière étape de développement qui sera suggérée dans la méthode de prototypage rapide produite dans ce travail, il est nécessaire de tout de suite explorer les détails de la plateforme de test en laboratoire. Cette dernière est composée de deux cartes, soit une carte de développement Amirix AP1000 et une carte d'acquisition analogique numérique PMC66-16AISS8AO4 de General Standards. Cette dernière se branche à la carte Amirix via le bus PCI mezzanine (PMC). La recherche mentionne aussi une carte Xilinx ML402. Cette carte est utilisée strictement pour la co-simulation car sa configuration pour cette étape est plus rapide.

Notre achat de carte AIO a influencé CMC qui cherchait une carte d'acquisition à joindre à la plateforme Amirix AP1000 afin d'offrir une solution complète aux groupes de recherche académiques de son réseau. Comme la documentation et les exemples fournis par Amirix ne contiennent pas de projets utilisant la communication via PMC (ni sur la plateforme Linux embarquée, ni sur le *Baseline* de la carte), un contrat entre CMC et l'UQAC (travail réalisé par notre équipe de recherche) fut signé avec pour objectif de produire le nécessaire pour assurer la communication des deux cartes, ainsi qu'une méthodologie simple pour utiliser la carte AIO à partir de la logique matérielle du FPGA. Cet ajout au *Baseline* de la carte Amirix est maintenant utilisé à l'échelle nationale dans la recherche des circuits intégrés et le domaine de MEMS (Micro Electro Mechanical Systems). Comme il s'agit d'une étape de la méthode de prototypage de cette recherche, les détails de développement de la communication sur la carte Amirix sont présentés dans l'annexe B.

CHAPITRE 3

Modèle System Generator

3.1 Brève description

Dans le cadre de cette recherche, le développement du prototype s'effectue avec le logiciel XSG. Il s'agit d'un toolbox développé par Xilinx pour être intégré dans l'environnement Matlab-Simulink et qui laisse l'usager créer des systèmes hautement parallèles pour FPGA. Les modèles créés sont affichés sous forme de blocs, et peuvent être raccordés aux autres blocs et autres toolbox de Matlab-Simulink comme SPS. Une fois le système complété, le code VHDL généré par l'outil *XSG* reproduit exactement le comportement observé dans Matlab. Pour le prototypage rapide, le choix de cet outil est facilement explicable. Le système de contrôle devant être vérifié et simulé souvent et rapidement pendant tout le développement, il est beaucoup plus simple d'analyser les résultats avec Matlab qu'avec les outils habituellement associés au VHDL, tel que Modelsim. Aussi, le modèle peut ensuite être couplé à des moteurs virtuels (à l'aide du toolbox SPS) et des simulations en boucle fermée sont réalisables. Quand le prototype fonctionne, le passage vers la plateforme matérielle pour des tests sur le terrain est rapide, ce qui rend la validation du prototype un projet réalisable à court terme.

3.2 Méthode de développement

Avec une banque de résultats provenant de la référence établie, le développement des blocs du prototype peut se faire. L'outil XSG est utilisé afin de produire un modèle qui va tout de suite fonctionner sur le matériel une fois achevé et validé.

3.2.1 Développement de bloc individuel et validation de premier niveau

L'avantage d'utiliser XSG pour le prototypage rapide devient plus évident lorsqu'il est nécessaire de tester un bloc achevé. Il suffit de brancher aux entrées les données intermédiaires obtenues de la référence. En simulant, on recueille les données à la sortie pour ensuite les comparer avec les données de la référence. Notons qu'il peut y avoir une certaine erreur, car une précision arbitraire selon le nombre de bits des opérandes est employée. L'outil GAPPA va plus tard permettre d'ajuster cette précision (voir section 4.2). Pour l'instant, il faut seulement vérifier le comportement du bloc.

Lorsqu'un bloc se comporte bien, ce test s'effectue assez rapidement, car le temps de simulation est plus court. Nous avons déjà mentionné que les signaux intermédiaires ne sont pas emmagasinés en trop grand nombre, de façon à rendre le temps de simulation acceptable. Cependant, lorsqu'un bloc ne répond pas comme il se doit, le travail devient plus important. Dans le cas de la commande vectorielle, plusieurs blocs sont d'une complexité importante comme le bloc découplage ou le bloc estimateur de flux du rotor. La provenance de l'erreur à l'intérieur du bloc est souvent difficile à identifier. Il faut alors reprendre notre méthode en traitant le bloc comme un algorithme séparé. On retourne vers la référence afin d'identifier les signaux que l'on retrouve à l'intérieur du bloc problématique, et une nouvelle collecte de données est effectuée. Ensuite, de nouvelles sorties temporaires sont instanciées à l'intérieur du bloc fautif pour tenter d'identifier la source du problème.

Afin de rapidement identifier si le comportement du modèle XSG correspond au modèle théorique Simulink, les résultats des deux modèles sont comparés. Cependant, il s'agit de tests qui ne serviront pas seulement lors du développement initial, mais qui seront sollicités lors de toutes les phases de débogage. Ainsi, il est important d'établir certaines règles afin d'accéder rapidement à cet environnement lorsqu'un problème survient. Deux méthodes sont ici proposées : la méthode simultanée et la sainte croyance.

Voyons premièrement les conditions communes aux deux méthodes.

- *Le bloc XSG utilise le même fichier d'initialisation que le modèle final*

  Il est important d'utiliser le même fichier d'initialisation que le modèle XSG final afin de ne pas avoir à créer d'autres copies de ce fichier. Chaque fichier supplémentaire d'initialisation doit être modifié si un changement de constante, de précision, etc. est effectué. Lorsqu'un modèle est composé de nombreux blocs, maintenir un nombre égal de scripts d'initialisation devient une source d'erreurs trop importante.

- *Les données (entrées et sorties) sont toutes dans une même source*

  Dans les deux méthodes proposées, les entrées proviennent de l'enregistrement des signaux lors d'un test en boucle fermée du système de référence (Simulink et SPS). Comme tous ces signaux sont enregistrés au même endroit (ex. : fichier MAT), il est important de toujours utiliser cette source, même si une seule portion des signaux est utilisée. Encore une fois, la maintenance de multiples fichiers doit être évitée pour minimiser les causes d'erreur et faciliter la réutilisation de l'environnement de bloc individuel lors de débogage avancé.

- *Les entrées respectent la nomenclature des signaux du modèle de référence Simulink*

  Dans un système complexe, plusieurs signaux sont insérés dans des blocs différents afin de subir plusieurs traitements. Il convient donc d'utiliser le même nom pour un signal injecté dans plusieurs blocs différents. Dans l'environnement de bloc individuel, les entrées sont souvent des signaux provenant du *workspace* ou d'un fichier. Comme la deuxième règle spécifie que l'on doit utiliser un seul et même fichier de source, les noms des entrées vont devoir correspondre dans chaque environnement de test. Les sorties ne vont pas se plier exactement à cette règle selon la méthode choisie.

- *Les blocs sont encapsulés exactement comme dans le modèle final ou de référence*

  Il est important d'utiliser le même routage de fils afin de pouvoir supprimer et remplacer un bloc rapidement lors de modifications à vérifier.

- *L'analyse des résultats se trouve dans un fichier Matlab unique au bloc, exécutable en tout temps*

  Chaque bloc doit avoir son fichier Matlab d'analyse qui va générer les résultats (et souvent, les graphiques) permettant de valider le bloc XSG. Un seul fichier par bloc assure que l'utilisateur n'a pas besoin de faire fonctionner les autres blocs (ou de commenter des lignes du fichier Matlab) pour faire une analyse. Le fichier doit également pouvoir produire une analyse en tout temps, afin que l'usager puisse faire une pause dans sa simulation et analyser les résultats obtenus jusqu'à cette pause, sans devoir compléter un temps donné de simulation.

3.2.1.1 *La méthode simultanée*

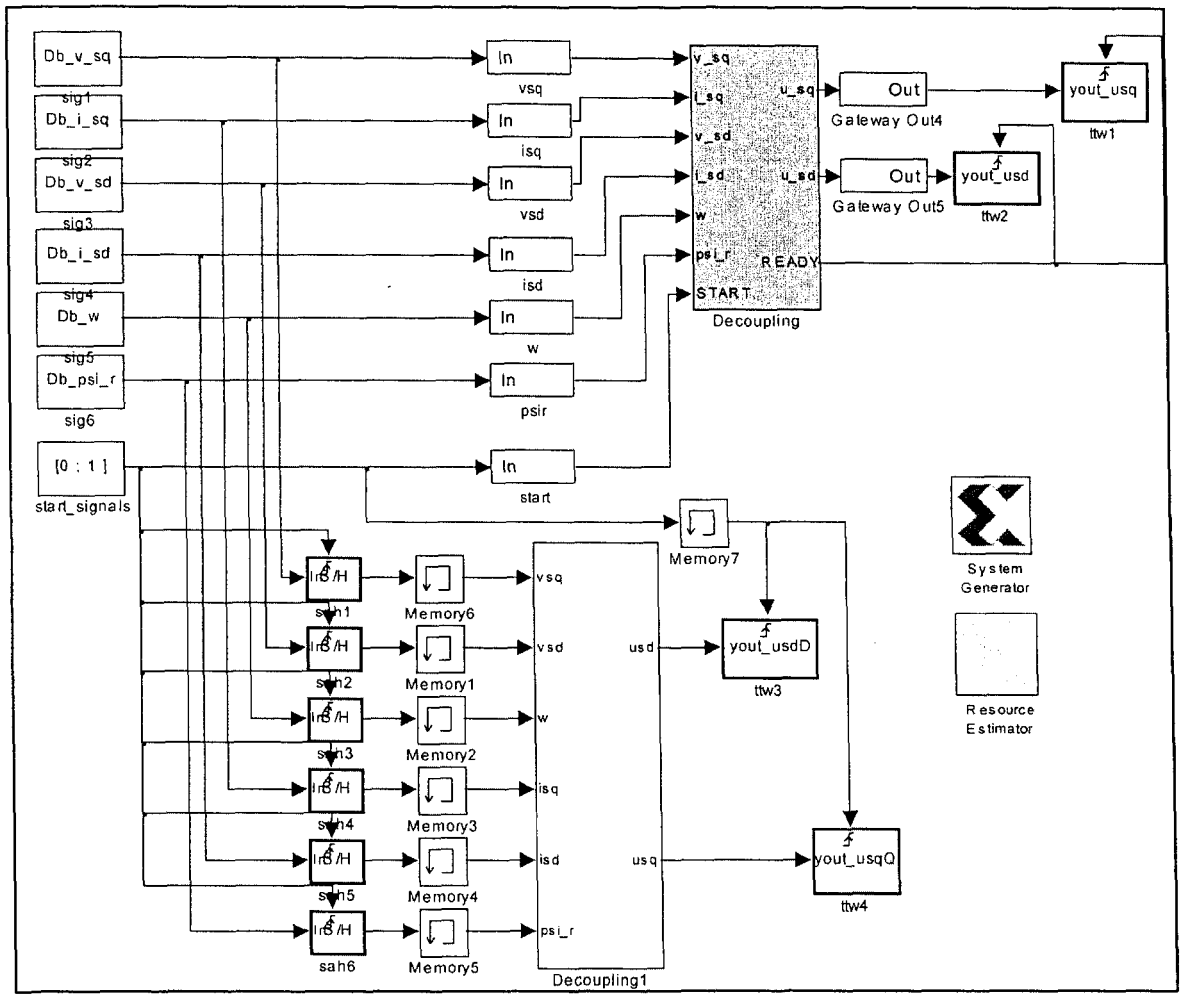


figure 9 - méthode simultanée (bloc individuel)

L'environnement contient une source d'entrée, deux blocs et deux ensembles de sorties. La source d'entrée commune aux deux blocs provient du modèle de référence Simulink. Ces signaux sont ensuite injectés dans le bloc XSG ainsi que son bloc Simulink correspondant, en parallèle. Les deux sorties (XSG et référence) sont ensuite enregistrées et comparées à l'aide du fichier d'analyse Matlab créé pour le bloc. Cette méthode est la plus efficace pour tester la correspondance des

deux blocs, car ceux-ci sont tous les deux isolés et doivent traiter les mêmes entrées. Les connexions et délais entre les blocs du système ne sont donc pas une cause d'erreur possible dans ce scénario.

Cependant, la comparaison des deux systèmes occasionne souvent des problèmes de « timing ». Le nombre de pas nécessaire pour le traitement n'est pas forcément le même pour les deux blocs, ce qui engendre l'insertion de blocs supplémentaires pour synchroniser les résultats à comparer et introduit conséquemment une nouvelle source d'erreur. Mais le plus gros désavantage de cette méthode est que, advenant un changement dans le bloc XSG, les contraintes temporelles peuvent être appelées à changer, ce qui signifie que l'usager doit modifier le mécanisme de synchronisation entre les deux blocs. Un tel ouvrage ralentit l'usager lors de tests de débogage et détériore la facilité de réutilisation de l'environnement.

3.2.1.2 La sainte croyance

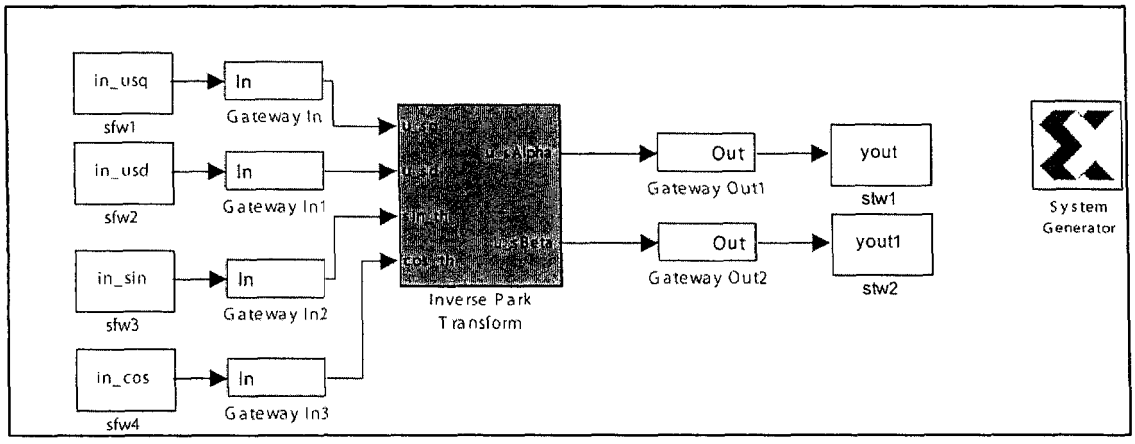


figure 10 - méthode de la sainte croyance

Cette méthode ne contient qu'un seul bloc par environnement, soit le bloc XSG. Pour des entrées qui proviennent de la simulation du modèle Simulink théorique entier, le bloc XSG traite ces données et produit des résultats qui sont ensuite comparés aux résultats enregistrés aux

sorties du même bloc dans le modèle théorique entier. On suppose ici que le bloc du modèle théorique est juste, et que les sorties théoriques utilisées sont fiables pour l'analyse.

L'avantage d'une telle méthode est la rapidité d'utilisation. Un seul bloc doit être simulé par l'ordinateur, un seul ensemble de sorties doit être enregistré, et l'environnement ne change jamais, peu importe les modifications apportées au bloc XSG. La réutilisation de cet environnement est donc très grande, et le débogage s'effectue rapidement.

Cependant, un changement des constantes et paramètres de n'importe quel bloc du système affecte les résultats du système de référence Simulink. Donc, pour chaque changement de constante, la simulation du modèle de référence en entier doit à nouveau être effectuée. Cela n'est pas toujours nécessaire en utilisant la méthode simultanée, car pour une même entrée, les deux traitements sont simulés. Pour un modèle complexe (figure 11), ce désavantage peut coûter du temps plutôt qu'en sauver. Le plus grand désavantage est au niveau de la précision d'identification, car cette méthode ne permet pas de détecter un problème qui vient des interconnexions entre les blocs. Par exemple, un usager pourrait chercher longtemps le problème de correspondance entre deux sorties, alors qu'avec la méthode simultanée, l'erreur serait la même à la sortie des deux traitements, ce qui confirme que le bloc XSG démontre le même comportement que son bloc de référence. Bien que plus rare, ce cas peut se produire et ralentir l'identification d'un problème surtout dans un système qui comporte de nombreux délais, registres et bascules d'entrée et de sortie.

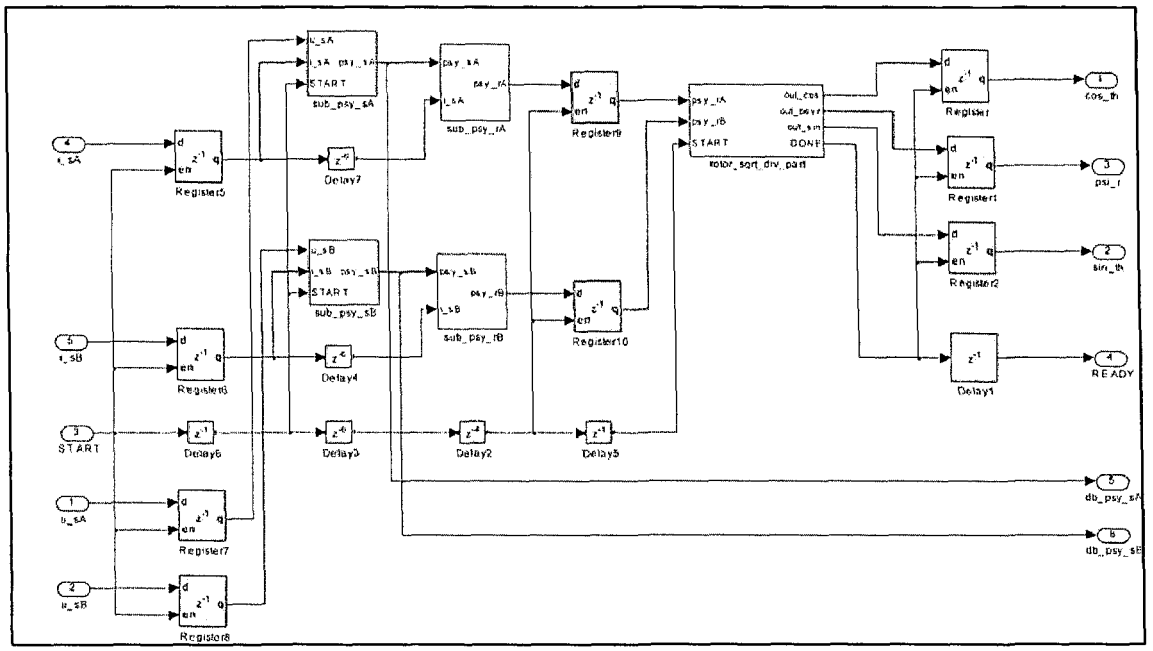


figure 11 - Estimateur de flux rotorique (bloc complexe)

3.2.2 Développement d'ensemble, liaisons et validation de deuxième niveau

Le test de chaque bloc individuel peut sembler laborieux, mais le temps investi dans cette étape se rentabilise lors du débogage du système complet. En effet, il serait laborieux de chercher le bloc fautif qui rend le système complet inopérant. Plus le test des blocs individuels est rigoureux, moins le test du système complet est laborieux. Souvent, comme dans le cas de la commande vectorielle, le défi est alors de bien synchroniser les blocs entre eux. C'est ici que des signaux « START » et « READY » peuvent être implémentés afin de s'assurer que le traitement des données se fait dans le bon ordre. Il ne s'agit plus ici d'un problème d'ordre mathématique, mais plutôt d'ordre logistique.

3.3 Contrôle vectoriel

Le contrôle vectoriel XSG est divisé selon les mêmes blocs schématisés lors de la prise de connaissance de l'algorithme de contrôle. De cette façon, il est plus facile de comparer et de tester les blocs de manière individuelle durant le développement. Le résultat d'assemblage de tous les composants est illustré à la figure 12. Des schémas internes de chaque sous-système composant le contrôle vectoriel se trouvent à l'annexe E.

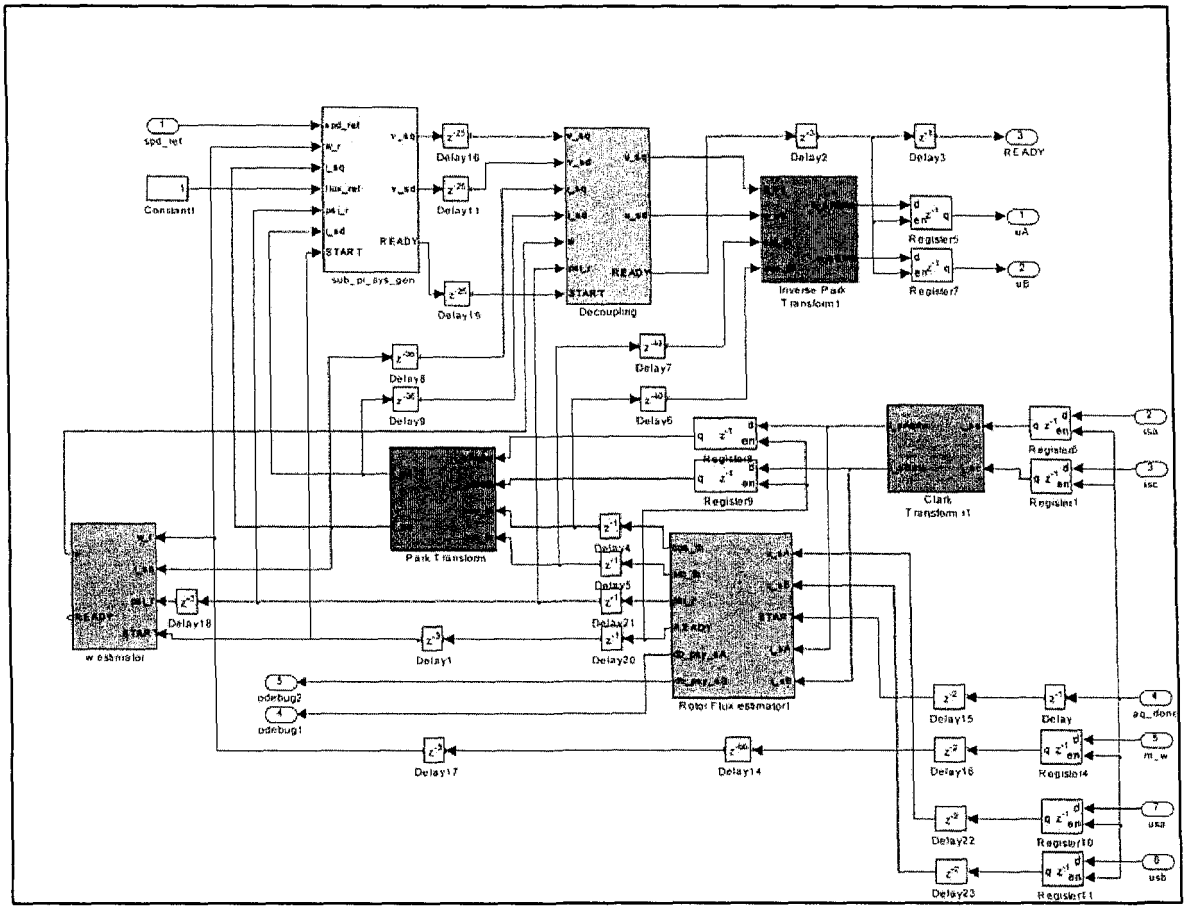


figure 12 - Contrôle vectoriel complet XSG

En incluant le calcul des délais d'activation (*firing*), le contrôle demande 122 pas d'horloge avec une précision adéquate pour donner les résultats observés à la section 4.10. Comme illustré dans la section 2.1.2, le contrôle ne s'effectue que deux fois pour une période de commutation, ce qui signifie qu'à une vitesse d'horloge de 62.5 MHz (environs 2 µs), il a amplement le temps de se compléter. Advenant le cas où il serait désirable d'effectuer des contrôles en série, il faut considérer qu'on doit attendre 70 pas d'horloge entre chaque échantillonnage. Ceci est dû au fait que l'opération de division et celle de racine carrée présentes dans le système doivent produire un résultat avant de traiter une nouvelle valeur (voir section 3.4).

La figure 13 illustre les propriétés d'un bloc d'opération XSG où la précision peut être changée par une valeur ou une variable. La section 4.2 explique l'importance d'utiliser des variables et l'outil GAPPA pour déterminer ces précisions. La précision des signaux d'entrée de tout le système (courants, tensions, vitesse) est représentée sur 16 bits, car il s'agit de la précision offerte par la carte AIO utilisée au laboratoire.

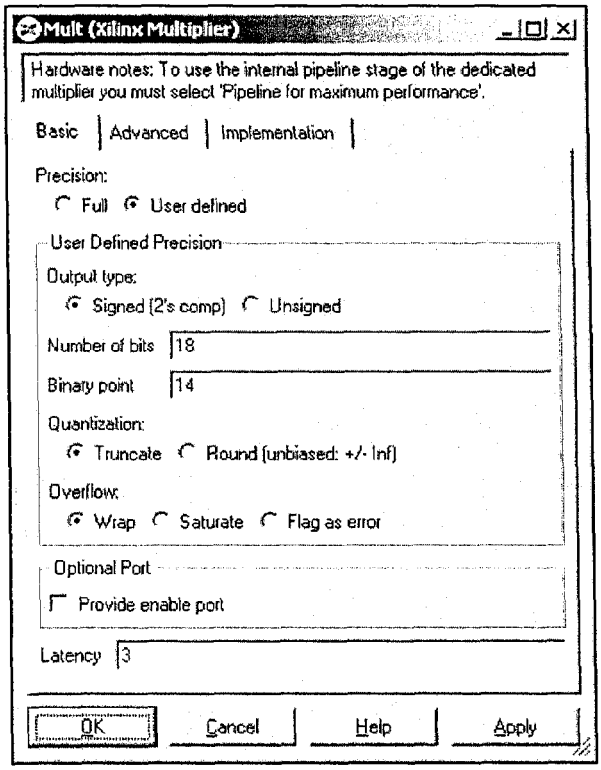


figure 13 - propriétés d'un bloc XSG

3.4 Division et Racine carrée

Il est possible d'utiliser une version non restaurante de la racine carrée et de la division afin de garder petite la taille du système et d'obtenir une meilleure précision. En effet, les opérations de division et de racine carrée fournies par XSG sont des implémentations CORDIC lourdes et peu précises. Après de nombreux tests en utilisant les versions 7, 8.1 et 8.2 de XSG (les résultats n'ont pas été vérifiés avec la nouvelle version 9), l'opération de racine carrée CORDIC de XSG ne donne qu'une grossière approximation du résultat lorsqu'une grande précision fractionnaire est demandée. De plus, la lourde implémentation de cette opération nécessite un long temps de synthèse lors de la génération d'un bloc co-simulé ou du fichier BIN pour des tests sur le terrain. Dans le cas du contrôle vectoriel, la précision offerte par l'opérateur racine carrée n'est pas suffisante, et le contrôle diverge. La division CORDIC offre pour sa part une précision très satisfaisante et permet le bon fonctionnement d'un algorithme comme le contrôle vectoriel. Cependant, sa grande taille et sa vitesse d'exécution moins grande peut la rendre moins intéressante pour une application très complexe qui nécessite déjà beaucoup de ressources du FPGA.

Les versions non restaurantes décrites dans l'annexe I de ces deux opérations offrent la précision désirée, mais elles ne peuvent pas recevoir plusieurs valeurs à traiter avant l'accomplissement de l'opération. Pour une valeur X à l'entrée de la racine carrée, il n'est pas possible de changer X pour une valeur Y tant que le résultat de la racine de X n'est pas obtenu. Heureusement, comme le contrôle vectoriel s'effectue très rapidement et s'insère confortablement entre deux échantillonnages, les versions non restaurantes peuvent être utilisées sans effet négatif. Les opérations sont complétées longtemps avant qu'un nouvel échantillonnage doive être traité. De plus, le nombre de pas nécessaire pour effectuer une racine carrée est coupé de moitié avec cette nouvelle implémentation, tandis que la division nécessite le même nombre de pas d'horloge.

Les tailles et performances des opérateurs sont comparées au tableau 2. Il s'agit de versions qui n'ont jamais été optimisées pour les FPGAs auparavant. Les schémas XSG des versions non restaurantes crées dans XSG pour cette recherche se trouvent à l'annexe F.

32 bits (14 binary point)						
	Sqrt NON CORDIC	Sqrt CORDIC	Diff.	Division NON CORDIC	Division CORDIC	Diff.
Slices	60	1377	2295%	124	1806	1456%
FFs	82	2289	2791%	96	3132	3263%
BRAMs	0	0	n.a.	0	0	n.a.
LUTs	103	2425	2354%	218	2981	1367%
IOBs	0	0	n.a.	0	0	n.a.
Emb. Mults	0	6	600%	0	3	300%
TBUFs	0	0	n.a.	0	0	n.a.
Nbr. de pas	24	51	213%	48	47	98%
Vitesse	238.112 MHz	58.666 MHz	25%	173.408 MHz	76.963 MHz	25%

tableau 2 - Tailles des opérateurs division et racine carrée

3.5 Portillonnage (délai d'activation et temps de maintien)

Les blocs de portillonnage précédemment implémentés à l'aide du bloc de fonction embarquée (voir section 2.2.2) dans Simulink sont maintenant créés avec XSG (figure 14 et figure 15).

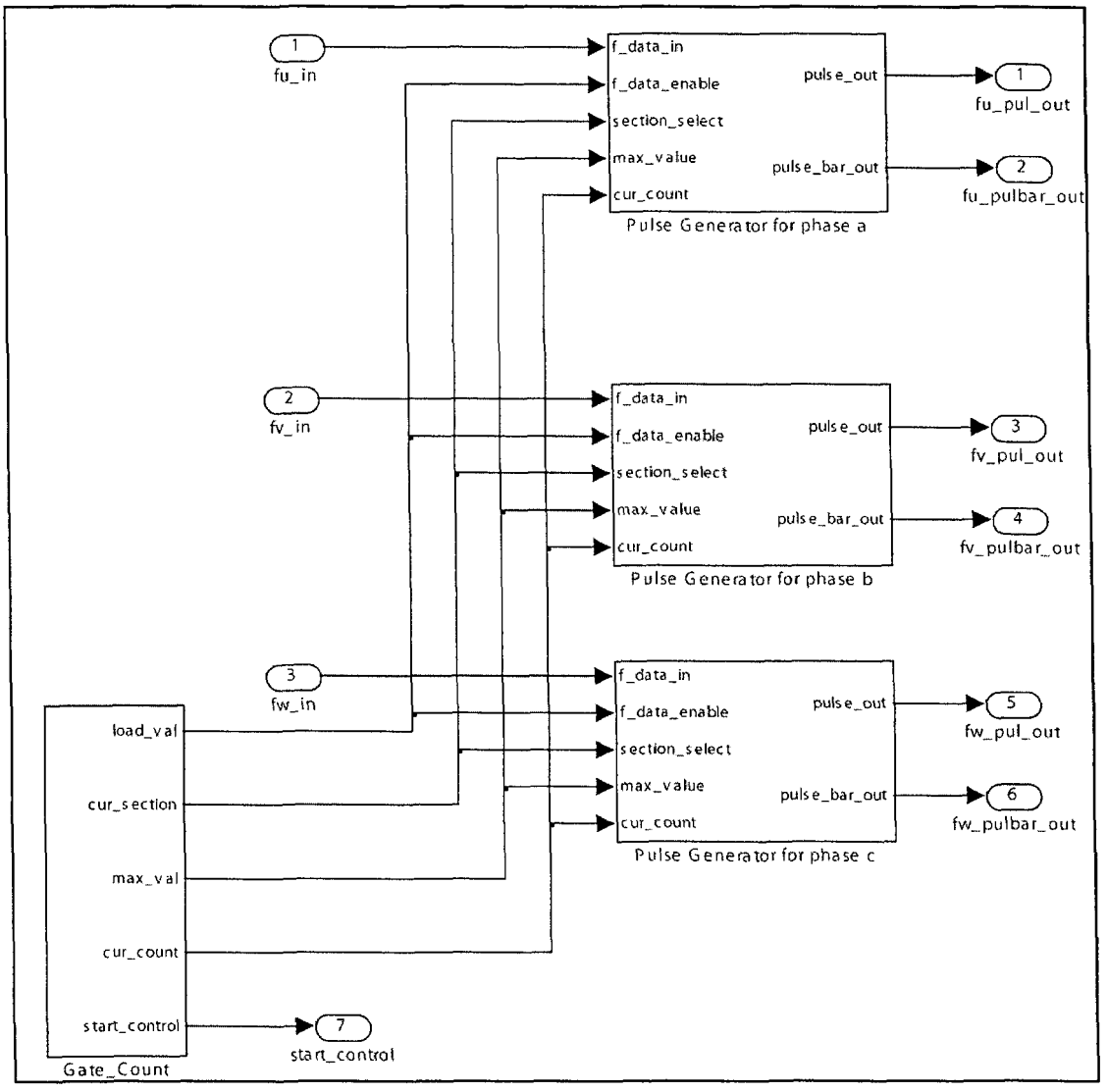


figure 14 - Portillonnage

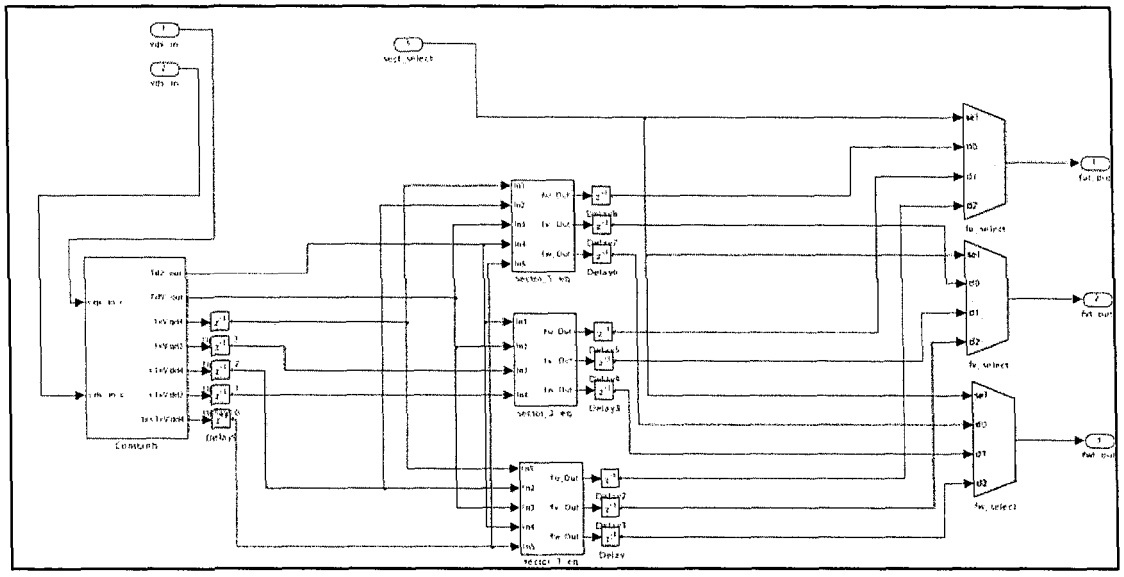


figure 15 - Calcul des délais d'activation

Le calcul du temps mort (figure 16) qui s'effectue à l'intérieur du bloc de portillonnage est basé sur un algorithme développé par OPAL-RT dans [9] et adapté à la situation de la recherche. Le fichier d'initialisation Matlab dicte le temps mort désiré avec la constante `dead_time`.

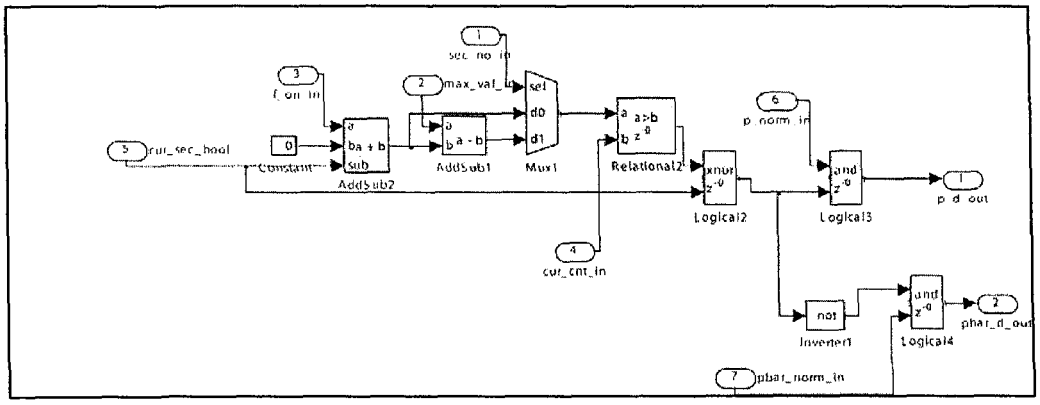


figure 16 - Calcul du temps mort

Afin de tester le bon fonctionnement du portillonnage avant son intégration au sein du projet entier, celui-ci est inséré dans un tutoriel fournit par SPS. Il remplace alors le portillonnage de base

s'y trouvant, et le débogage peut s'effectuer jusqu'à l'obtention d'une précision adéquate. Les résultats sont observables à la figure 17.

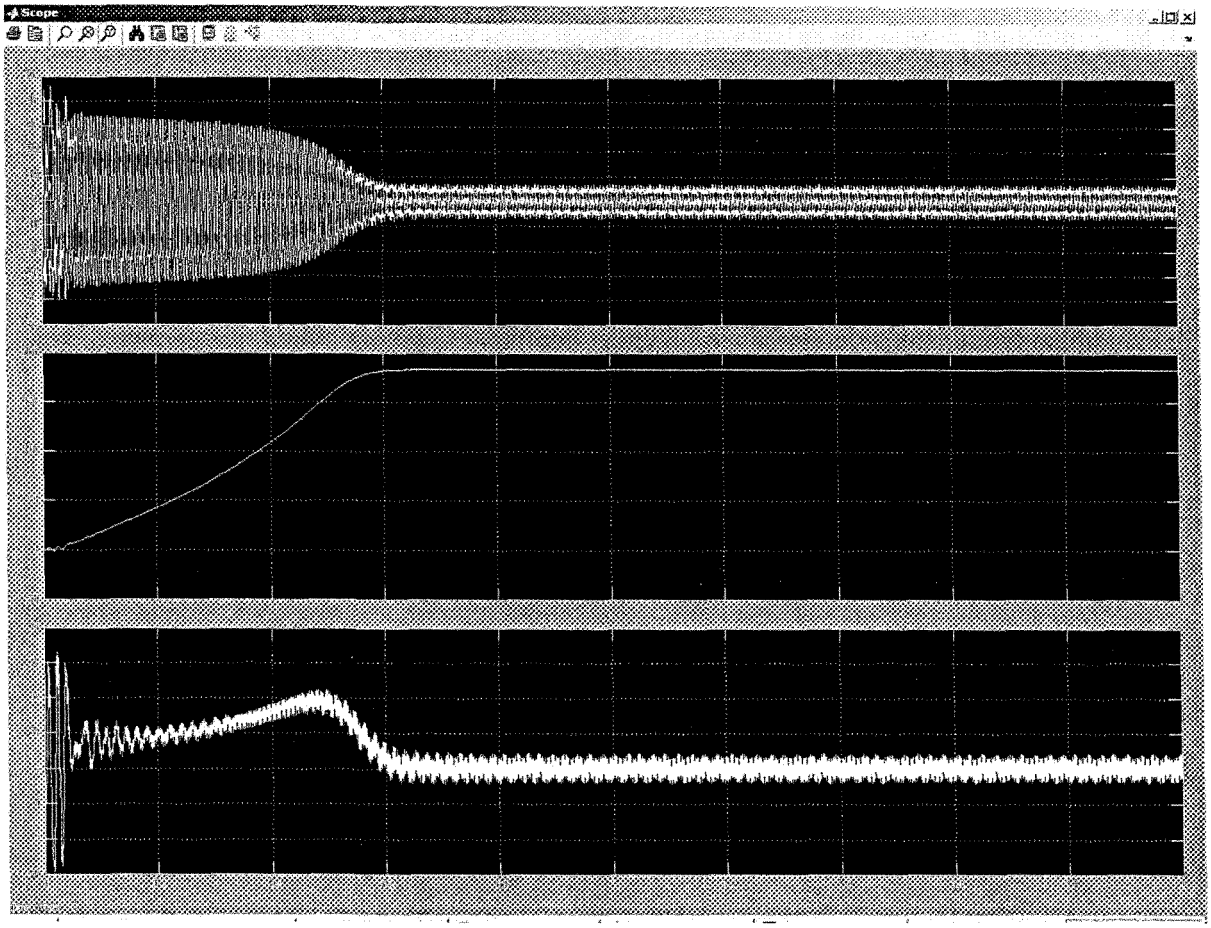


figure 17 - Courants, vitesse et torque du moteur avec simulation du portillonnage XSG

Le profil suivant donne les résultats de la figure 19.

```
freq=10;
Ua=0;Ub=0;
for k=Ts:Ts:3

    Wref=2*pi*(freq/p);


    if k > (0.3)
         freq=30;
    end
```

```
if k > (0.8)
     freq=10;
end

    if k > (1.7)
     freq=50;
end

if k > (2.5)
     freq=5;
end
```

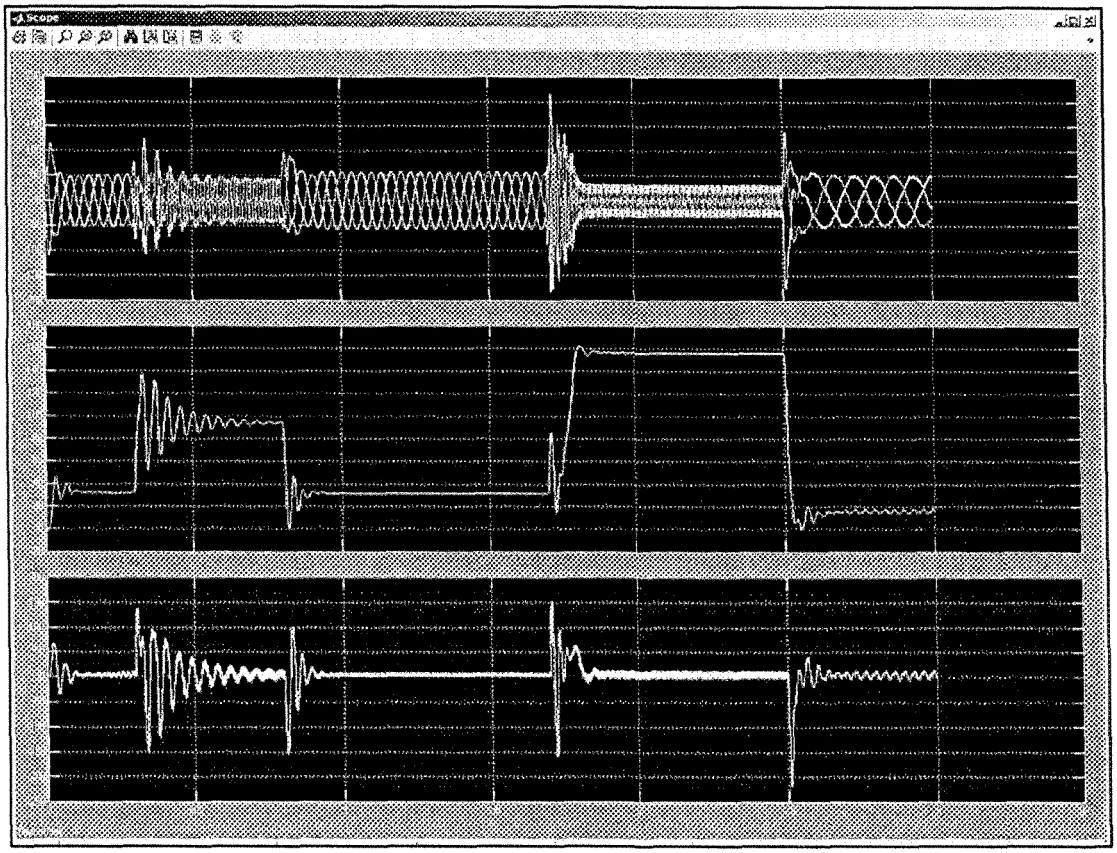


figure 18 - Courants, vitesse et torque du moteur avec simulation du portillonnage XSG

3.6 Lecture de vitesse provenant d'un encodeur optique

L'implémentation de la lecture de vitesse simple se trouve à la figure 19. Il s'agit d'un filtre simple qui calcule la vitesse moyenne pour une plage de temps spécifiée. Une implémentation beaucoup plus efficace décrite dans [12] sera éventuellement développée et testée avant d'effectuer des tests plus poussés en laboratoire. Pour des tests préliminaires et la vérification des connexions, cette méthode est suffisante.

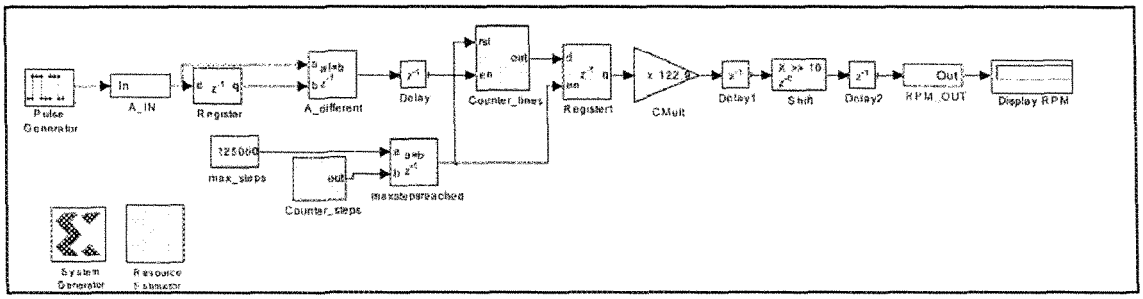


figure 19 - Lecture de vitesse d'un encodeur optique (modèle simple)

CHAPITRE 4

GAPPA et les outils de vérification XSG

4.1 Introduction à GAPPA

Un outil fort intéressant appelé GAPPA (Génération Automatique de Preuves de Propriétés Arithmétiques) est développé au Laboratoire de l'Informatique du Parallélisme en France. Cet outil permet de vérifier et prouver les propriétés numériques de programmes qui font de l'arithmétique en virgule flottante et en point fixe. C'est ce dernier volet qui nous intéresse pour la présente recherche, car il s'agit du volet qui peut grandement réduire les temps de développement avec XSG (ou même un projet codé à la main en langage HDL).

Le programme est utilisé pour trouver l'intervalle des valeurs possibles pour les équations des modèles afin d'utiliser le nombre minimal de bits pour les opérandes. Il permet aussi de connaître, pour une équation donnée, l'erreur du résultat pour un nombre de bits alloué à la partie fractionnaire d'un opérande. Cette erreur peut être calculée en fonction de l'équation théorique ou en la comparant à une équation dont le nombre de bits pour chaque opérande diffère. En conséquence, cela permet non seulement d'optimiser davantage le modèle, mais également de déterminer rapidement comment arriver à une précision désirée sans devoir longuement simuler ou co-simuler dans Matlab. L'intégration de cet outil au sein d'une méthode de prototypage rapide est un apport important de cette recherche.

4.2 Premier niveau - Pre-XSG

4.2.1 Scripts

Avant de commencer la fabrication du modèle XSG, les équations et sous-systèmes du contrôle vectoriel sont représentés avec des scripts GAPPA. Ces scripts vont permettre de connaître les précisions requises de chaque opérateur XSG afin d'obtenir des résultats d'une précision adéquate dans tous les cas possibles pour une étendue de valeur d'entrée.

Bien que la syntaxe GAPPA contenue dans la documentation du logiciel soit vaste, seuls certains aspects s'appliquent dans cette recherche.

Les opérateurs XSG comportent toujours deux paramètres de précision, soit un nombre de bits alloués à la partie binaire d'un résultat, et un nombre de bits total (partie binaire + partie entière). Dans un script GAPPA, la partie binaire est indiquée avec la notation de virgule fixe, soit :

```
@clark_sub_isb_Cmult1 = fixed<-14,dn>; #Binary point de la multiplication
```

Ce qui signifie que l'opérateur Cmult1 du bloc sub_isb dans le système Clark va avoir 14 bits pour sa valeur binaire de sortie.

Ensuite, lorsqu'une opération mathématique est effectuée, la contrainte de point fixe est appliquée :

```
clark_res clark_sub_isb_Cmult1 = isa*isb;
```

Ici, *clark_res* est le résultat de la multiplication des deux signaux *isa* et *isb* avec une précision de point binaire de 14 bits.

Une fois que toutes les équations et variables sont établies, les questions sont posées au logiciel.

```
{ isa in [-100,100] /\
  isb in [-100,100]
  ->
  clark_res in ? /\
  clark_res - CLARK_RES_THEORIQUE in ?
}
```

Cela signifie que pour des entrées *isa* et *isb* de n'importe quelle valeur entre -100 et 100, nous voulons connaître l'étendue des valeurs de réponse possible de *clark_res*. Aussi, nous demandons l'erreur maximale en comparant *clark_res* à sa valeur théorique (il suffit de coder la ligne « `CLARK_RES_THEORIQUE = isa*isb;` »).

4.2.2 Migration vers Matlab/XSG

Il est important de connaitre à l'avance l'étendue possible des constantes de multiplication utilisées dans les équations traitées par GAPPA. Lorsqu'un algorithme complexe est modélisé, le nombre de ces constantes grandit rapidement, ce qui force l'usager à produire des scripts GAPPA contenant un plus grand nombre de constantes. De plus, lorsque le nombre de blocs augmente dans notre modèle complet, le nombre de scripts GAPPA augmente non linéairement car il y a souvent plusieurs scripts pour un bloc donné.

Lorsque le processus de prototypage est suivi une première fois pour des conditions de tests données, la difficulté qui sera expliquée ci-dessous n'apparait pas nécessairement. C'est-à-dire que lorsque le premier débogage du système est effectué, il est souvent nécessaire d'utiliser des constantes de multiplication qui ne changent pas puisque l'on effectue des tests sur un moteur qui ne change pas non plus. Une fois que les changements de vitesse et d'autres profils sont effectués avec succès pour le moteur de test, il convient de refaire des tests pour un moteur différent. C'est ici que les constantes sont changées pour des valeurs qui peuvent parfois être totalement différentes.

Même si la planification des scripts GAPPA d'origine est supposée anticiper des changements aussi radicaux de valeurs, il se peut que certains chemins logiques du modèle n'allouent pas le nombre nécessaire de bits pour assurer la propagation de résultats d'une précision souhaitée. L'apparition de nouveaux cas problèmes lors de changement de paramètres nécessite de revoir la taille de certains signaux afin d'accommoder les nouvelles conditions. Malheureusement, l'identification rapide de la source (ou des sources multiples) peut s'avérer difficile même si la complexité du modèle est modeste, et s'avérer impossible lors de cas très complexes. C'est pourquoi une seconde analyse GAPPA est souvent souhaitable, et le grand nombre de constantes rend l'identification des signaux encore plus difficile.

Il est donc important d'établir un système de correspondance entre les scripts GAPPA et les scripts Matlab d'initialisation. Supposons ici qu'un fichier Matlab est utilisé strictement pour définir les constantes des opérateurs, afin de ne pas mélanger ces valeurs avec d'autres types

d'initialisation telle que la définition des profils. Le fichier Matlab ne doit contenir que les valeurs des constantes fixes utilisées dans le modèle, qui ne changeront pas lors du fonctionnement pour un moteur donné. Aussi, il doit contenir pour chaque opérateur XSG qui le demande le nombre de bits pour la partie entière et fractionnaire (2 constantes par opérateur). Dans le cas des multiplicateurs ou autres opérateurs qui nécessitent un plus long délai afin de répondre aux exigences temporelles du modèle, ces délais pourront aussi se trouver dans ce fichier. Finalement, les mêmes noms doivent correspondre entre les scripts GAPPA et Matlab, afin de faciliter l'identification lors de modifications dans l'un ou l'autre de ces outils. À titre d'exemple, voici le script GAPPA de la transformé de Clark dont l'équation est : $i_{s\beta} = \frac{1}{\sqrt{3}} i_{sa} + \frac{2}{\sqrt{3}} i_{sb}$

```
# GAPPA
# Optimisation de Park Transform

# Operateurs

@clark_sub_isb_Cmult1 = fixed<-14,dn>; # Binary point de la
multiplication
@clark_sub_isb_AddSub = fixed<-14,dn>; # Binary point de l'addition

# Constantes et entrees

@isa_bp = fixed<-14,dn>;
@isb_bp = fixed<-14,dn>;
@clark_sub_isb_s3_cte = fixed<-14,dn>;

# Equations Theoriques

s3     = 0.57735026918963;
ISBETA = (isa + (2*isb)) * s3;

# Conversion des entrees et cte

gisa  = isa_bp(isa);
gisb  = isb_bp(isb);
gs3   = clark_sub_isb_s3_cte(s3);

# Equation Pratiques

add_res clark_sub_isb_AddSub   = gisa + (2*gisb);
clark_res clark_sub_isb_Cmult1 = add_res*gs3;

# Analyse
```

```
{   isa in [-100,100] /\
    isb in [-100,100]
    ->
    ISBETA in ? /\
    clark_res in ? /\
    add_res in ? /\
    clark_res - ISBETA in ?
}
```

La figure 20 montre le résultat de l'exécution du script précédent avec l'outil GAPPA.

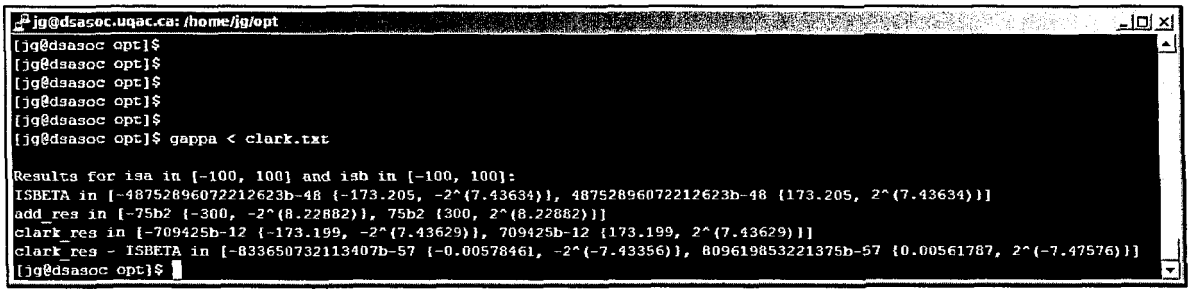


figure 20 - Résultat de l'exécution GAPPA

Comme l'erreur est satisfaisante et les valeurs entières maximales sont maintenant connues, le script d'initialisation Matlab peut être généré.

```
isa_bp = 14;
isa_bp_T = isa_bp + ceil(log2((100)+1));

isb_bp = 14;
isb_bp_T = isb_bp + ceil(log2((100)+1));

clark_sub_isb_s3_cte = 14;
clark_sub_isb_s3_cte_T = clark_sub_isb_s3_cte + ceil(log2((0)+1));

clark_sub_isb_Cmult1 = 14;
clark_sub_isb_Cmult1_T = clark_sub_isb_Cmult1 + ceil(log2((173)+1));

clark_sub_isb_AddSub = 14;
clark_sub_isb_AddSub_T = clark_sub_isb_AddSub + ceil(log2((300)+1));
```

La figure 21 montre comment ces variables sont utilisées dans XSG pour rendre les changements de précisions facilement et rapidement variables.

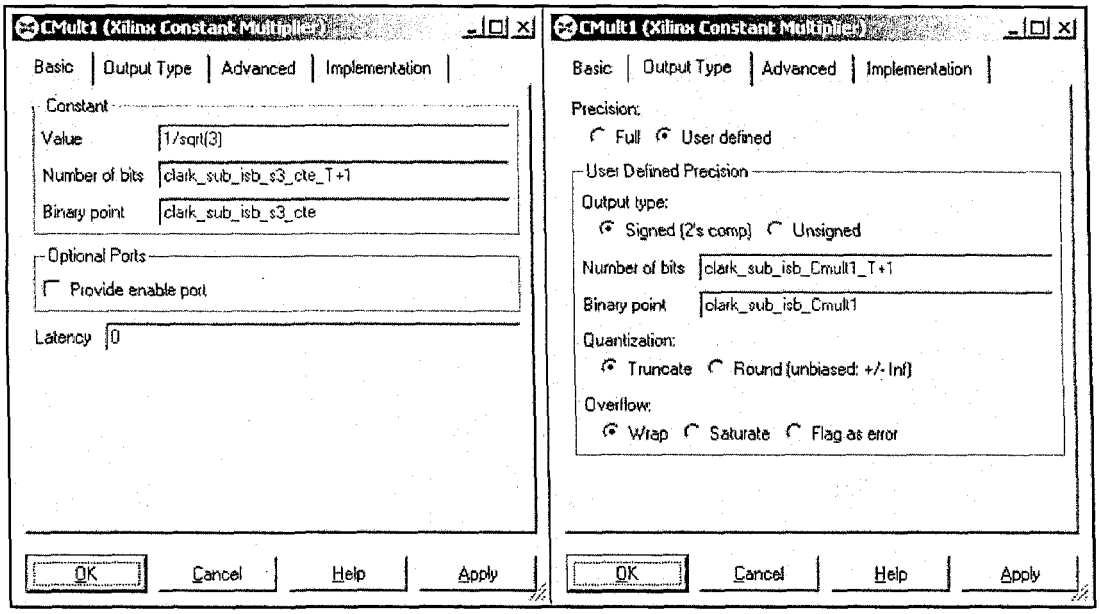


figure 21 - Précision variable dans un bloc XSG

L'exécution de l'équation en version XSG confirme bien les prédictions de GAPPA (figure 22). Il est donc possible de déterminer les précisions optimales (le minimum de bits) avec plusieurs exécutions de GAPPA qui sont presque instantanées, sans jamais devoir refaire des simulations XSG qui sont plus coûteuses en temps et plus difficiles à analyser afin de couvrir toutes les possibilités et cas critiques.

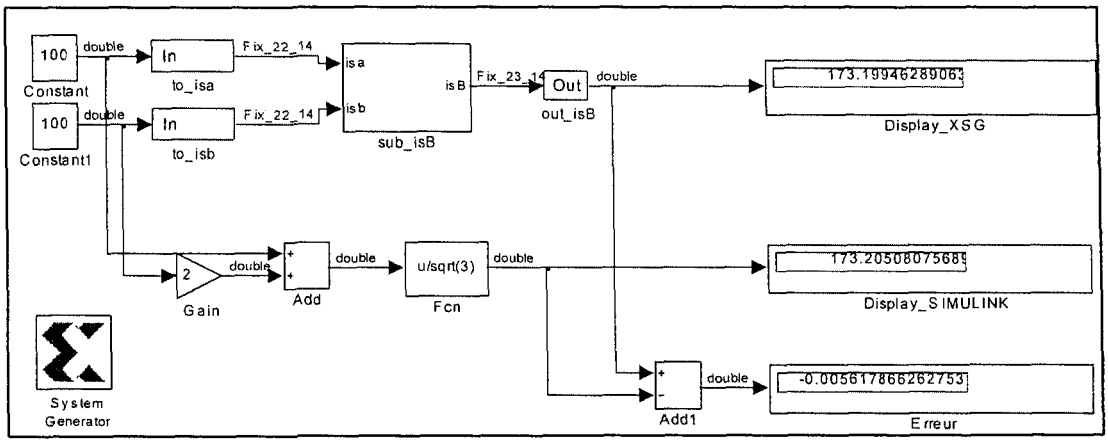


figure 22 - Résultat d'exécution XSG

La visualisation du processus à suivre laisse voir qu'une automatisation pourrait être considérée. Un parseur qui analyserait les scripts GAPPA afin de trouver les différentes constantes et précisions pourrait ensuite générer les constantes à insérer dans le fichier d'initialisation de constantes Matlab. En plus d'accélérer le développement et faciliter le débogage, un tel outil éviterait les cas où la nomenclature entre les deux outils ne correspond pas.

Tous les scripts GAPPA de cette recherche se trouve à l'annexe G.

4.3 Deuxième Niveau - Post-XSG

4.3.1 Optimisation de bloc individuel

Une fois que le prototype complet répond de la même manière que la référence, on passe à l'étape de deuxième niveau. Cette étape n'est pas toujours nécessaire, mais deux cas précis y font appel. Premièrement, si l'usager désire optimiser l'espace sur FPGA pour un moteur donné (les paramètres autrement variables du contrôle seront donc toujours les mêmes), les scripts GAPPA originaux peuvent être modifiés en diminuant l'étendue possible des constantes. Les précisions (en nombre de bits) sont donc diminuées au minimum requis par la configuration sélectionnée pour une marge d'erreur voulue. La taille de la logique au sein du FPGA est alors grandement optimisée.

L'autre cas arrive lors de la situation inverse. Supposons qu'un moteur complètement différent doit être contrôlé, et que ses paramètres ne sont pas gérés par les scripts GAPPA d'origine. Des modifications doivent alors être apportées afin d'éviter les dépassements au niveau des opérateurs. C'est une situation qui, sans l'aide de GAPPA, peut prendre des semaines de travail lorsqu'on modifie les valeurs directement dans le modèle XSG. Les tests GAPPA sont quasi instantanés et couvrent tous les cas possibles. Les simulations du système entier sont longues et il est difficile de couvrir tous les cas extrêmes. De plus, les scripts GAPPA vont tout de suite indiquer, après chaque exécution, si l'erreur provient du script actuel. Dans la simulation du modèle entier,

il faut analyser les signaux intermédiaires pour trouver les sources d'erreur, ce qui est difficile dans un système complexe avec rétroaction.

4.3.2 Validation de premier niveau et de deuxième niveau

Contrairement à la section 3.2, il est recommandé après une intervention GAPPA post-XSG d'effectuer une validation de deuxième niveau avant de retourner aux tests de blocs individuels. Comme il s'agit ici d'optimisation ou de correction de bogues de tailles d'opérateur, les modifications aux scripts GAPPA permettent de rapidement identifier les causes de dépassement et les optimisations possibles. Du coup, il est souvent suffisant de revérifier la réponse du modèle XSG dans son ensemble pour voir si les changements ont porté fruit. Advenant le cas où ces simulations sont sans succès, des évaluations de blocs individuels (en utilisant la méthode simultanée ou la sainte croyance, section 3.2.1) sont envisageables.

Dans le cas d'une optimisation, il est également possible de comparer la nouvelle réponse optimisée à l'ancienne, sans faire appel aux données accumulées à partir du modèle de référence (Simulink ou Matlab, voir section 2.3).

4.4 Analyse temporelle (Outil XSG)

Le prototype est jugé fonctionnel quand sa réponse et sa précision sont adéquates. Mais, avant de pouvoir le charger dans le FPGA ou même de le simuler avec des outils comme *ModelSim*, il faut procéder à l'analyse temporelle. Il s'agit d'une des étapes les plus longues du développement. Lorsque les blocs du prototype sont développés, on désire faire fonctionner un bloc sans se soucier des contraintes de temps. Mais une fois que son fonctionnement est vérifié, on doit prendre le temps d'enregistrer les entrées et sorties ainsi que d'ajouter des registres de délai afin de respecter les contraintes temporelles établies. Dans le cas de la commande vectorielle, supposons un contrôle qui doit s'effectuer en environs 3 µs. Si le contrôle nécessite approximativement 122 pas d'horloge, cela signifie qu'il doit fonctionner à une vitesse d'au moins 41 MHz. On doit également s'assurer que la carte cible est capable d'assumer une telle vitesse, ce qui n'est pas un problème dans la présente recherche. Pour effectuer ces vérifications, XSG propose l'outil d'analyse temporelle (timing analysis tool). Cet outil montre à l'utilisateur de manière graphique (figure 25) et textuelle (figure 24) les chemins critiques (les plus lents) ainsi que ceux qui ne réussissent pas à respecter les contraintes (ex : si un signal ne peut se rendre de la sortie d'un élément synchrone à l'entrée d'un autre élément synchrone en un pas d'horloge). L'utilisateur peut ensuite trouver des tactiques en vue d'accélérer ces chemins.

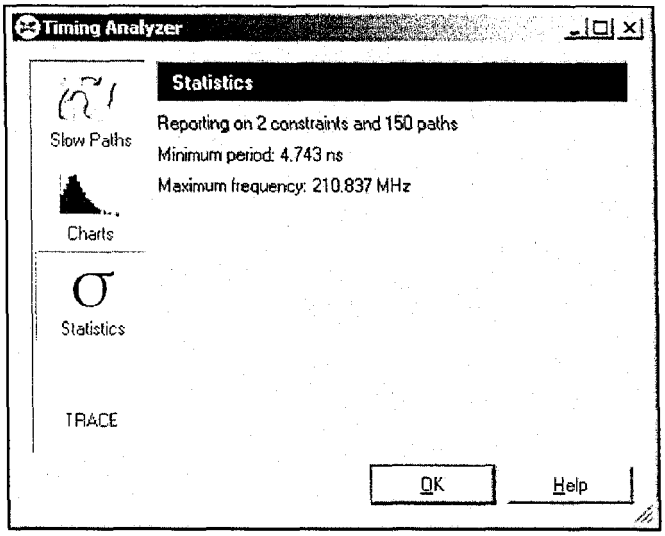


figure 23 - Analyse temporelle (vitesse maximum)

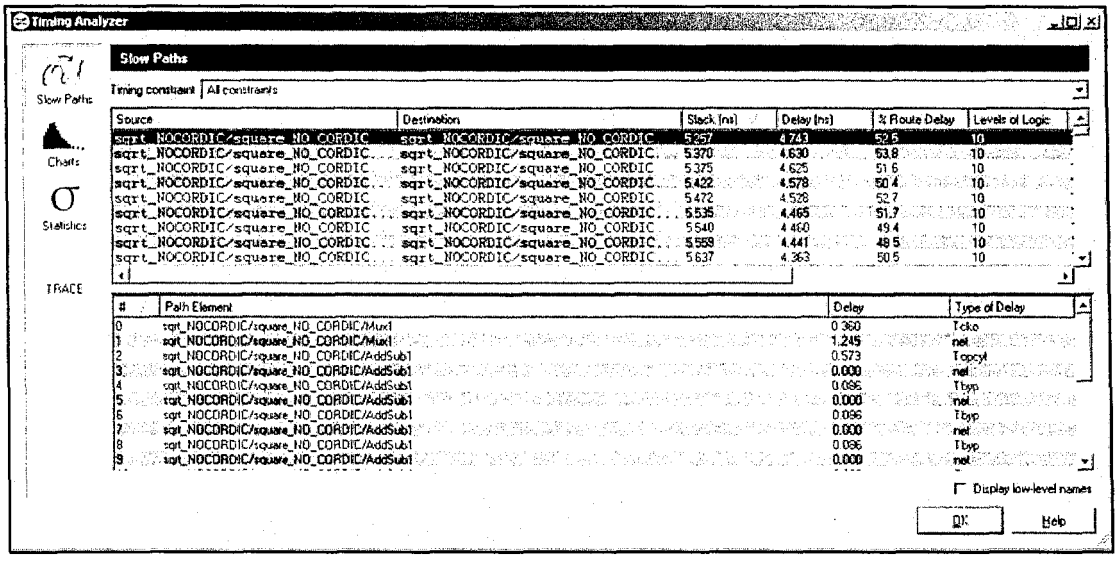


figure 24 - Analyse temporelle (détails des délais)

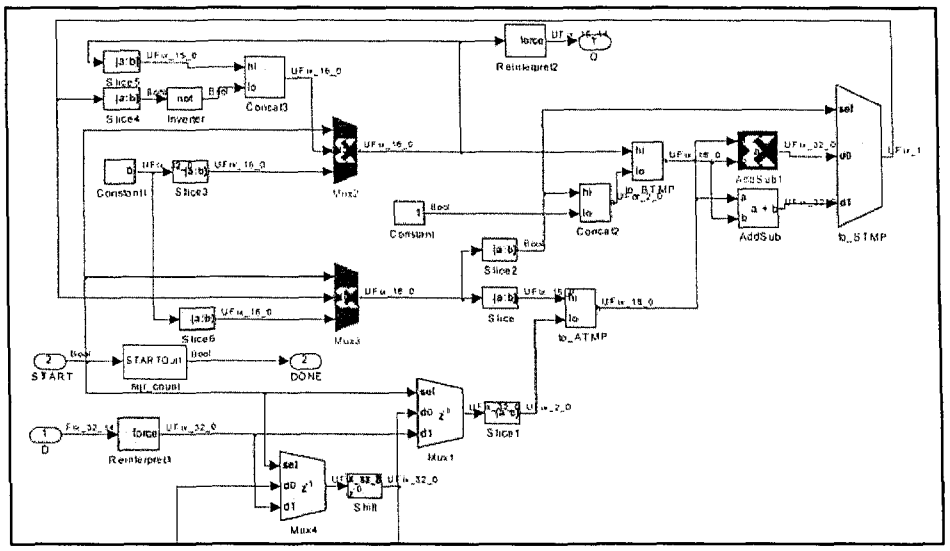


figure 25 - Analyse temporelle (identification des délais sur le modèle XSG)

Le problème est que l'analyse temporelle est longue à effectuer. Plus un système est complexe, plus l'analyse prend du temps à être générée. De plus, il faut faire l'analyse de chaque bloc individuellement, faire les ajustements et revérifier. Chaque analyse peut facilement prendre de 10 à 15 minutes et il faut souvent en effectuer plusieurs sur un même bloc avant de trouver le problème. Finalement, on effectue cette analyse sur le système complet, ce qui demande une période de temps encore plus importante.

4.4.1 Analyse de bloc individuel

Le résultat de l'analyse temporelle permet à l'usager d'insérer les délais et registres de façon manuelle aux bons endroits dans un système XSG ainsi que d'ajuster la latence de certains opérateurs comme la multiplication. La figure 26 représente le bloc estimateur de ω XSG avant l'analyse temporelle. Après avoir utilisé l'outil, des délais sont ajoutés (figure 27) pour assurer le respect des contraintes temporelles.

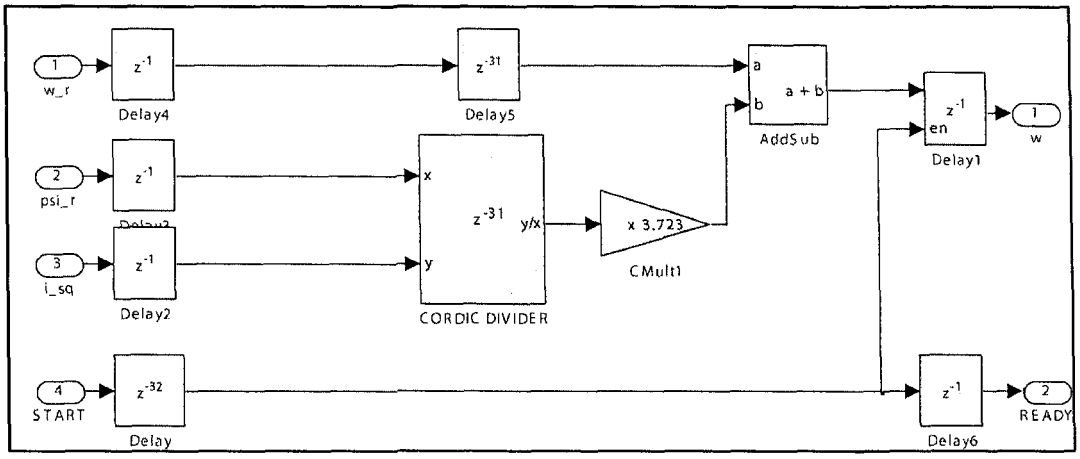


figure 26 - Bloc XSG avant analyse temporelle

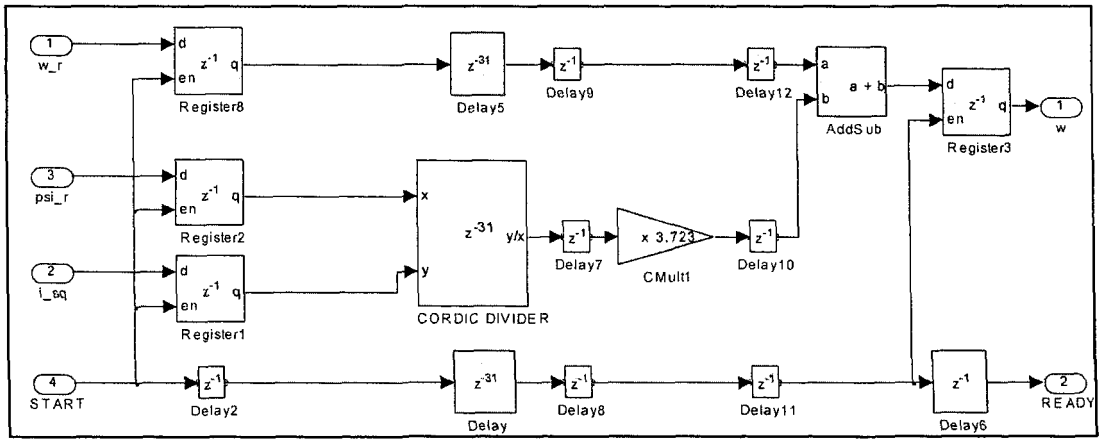


figure 27 - Bloc XSG après analyse temporelle

Tous les blocs figurants à l'annexe E ont passé par le processus d'analyse temporelle. C'est pourquoi on peut y observer beaucoup de blocs de délais. Il est également bon de noter que dans plusieurs situations, il est avantageux d'insérer de multiples délais d'un pas d'horloge plutôt qu'un seul délai de plusieurs pas d'horloge. Cela permet à XSG de mieux optimiser la logique générée et d'obtenir une taille plus petite. Cependant, les expériences conduites dans le cadre de cette recherche indiquent que ces gains d'espaces sont en général très minimes et ne valent pas nécessairement le désavantage d'un modèle XSG plus encombré à cause des blocs supplémentaires.

4.4.2 Analyse d'ensemble

Après la vérification que chaque sous-système respecte bien les consignes temporelles, il est important d'utiliser l'outil sur le système entier. Dans le cas du contrôle vectoriel, on observe tous les délais insérés après l'analyse temporelle du système dans la figure 12. Le contrôle vectoriel développé dans cette recherche fonctionne à une vitesse supérieure à 100 MHz, ce qui est nécessaire pour la co-simulation sur carte ML402 (voir section 4.7). En ce qui concerne la carte Amirix, le contrôle vectoriel ne va tourner qu'à une vitesse de 62.5 MHz. Le modèle respecte donc amplement cette condition.

4.5 Comparaison à AccelDSP

La compagnie Xilinx produit depuis peu un nouvel outil qui se nomme AccelDSP. Cet outil permet de passer d'un modèle en code Matlab vers un bloc XSG ou un bloc HDL prêt à intégrer dans un projet de l'usager. Cet outil propose également une manière optimisée de vérification d'un modèle selon un niveau de précision désiré.

Dans AccelDSP, l'usager peut spécifier une précision de X bits pour son modèle HDL. Une version en langage C est alors générée afin d'accélerer la simulation des résultats pour comparaison avec un modèle théorique en virgule flottante. Le désavantage de cette méthode est que l'usager ne peut utiliser cette technique de simulation à l'intérieur d'une boucle comme va le permettre la co-simulation XSG. Aussi, l'outil GAPPA permet d'optimiser tous les opérateurs à l'intérieur d'un bloc individuellement. Ceci est plus long, certes, mais peut permettre une meilleure optimisation de l'espace lors de problèmes complexes.

Finalement, l'outil AccelDSP force l'usager à écrire son code en Matlab, ce qui n'est pas l'objet de cette recherche. Dans le cas présent, c'est l'outil XSG qui permet de rapidement implémenter un projet avec une orientation matérielle. AccelDSP s'adresse plutôt aux développeurs qui sont moins familier avec l'aspect physique d'une implémentation. Dans un tel cas, l'outil AccelDSP fait un travail fabuleux.

4.6 Introduction à la co-simulation (XSG)

Pour simuler la partie XSG du système, il est possible d'utiliser deux techniques de co-simulation qui utilisent le FPGA de la plateforme ML402 afin de gérer la logique de contrôle. Un mode pas à pas (single step), dans lequel Simulink dicte au FPGA lorsqu'un pas d'horloge doit être simulé, ainsi qu'un mode de co-simulation libre (free running) où la logique est exécutée sur le

FPGA à pleine vitesse de 100 MHz. La communication entre Simulink et le FPGA est assurée par un lien ETHERNET gigabit dans les deux modes d'opérations.

Il est très difficile dans la littérature d'obtenir des informations sur l'efficacité et sur le temps nécessaire lors de l'utilisation de ces méthodes. Cette recherche permet de mieux évaluer la performance de cet outil, ce qui fait d'ailleurs l'objet d'un article de conférence présenté en Floride à l'été 2007 et disponible à l'annexe I. L'utilisation du mode de co-simulation libre dans cette recherche pour déjouer le simulateur de Matlab est un apport important de ce travail.

4.7 Co-simulation de modèles avec attente

Le mode pas-à-pas de co-simulation permet d'améliorer le temps de simulation lorsqu'une partie d'un gros système en boucle ouverte est remplacée. Cela est d'autant plus vrai si la partie du système qui n'est pas créée avec XSG ne contient que des blocs qui peuvent être accélérés par Simulink.

En boucle fermée, ce bénéfice de performance est aussi observable, mais seulement si le système XSG ne nécessite pas un grand nombre de pas pour effectuer son traitement des données. Si le nombre de pas nécessaire pour un traitement est plus grand que le nombre de pas entre deux valeurs à traiter, un ajustement dispendieux doit être effectué. La période Simulink pour la simulation d'un pas d'horloge du FPGA (un pas pour le modèle XSG) doit être réduite, alors que le reste du modèle Simulink doit continuer de fonctionner à la même vitesse. Dans un environnement où la taille d'un pas est fixe, cette taille doit être suffisamment réduite pour que le système XSG puisse compléter son opération entre deux acquisitions de données. L'exemple qui suit permet de mieux comprendre cette problématique.

Pour l'implémentation du contrôle vectoriel choisie dans le cadre de cette maîtrise, 122 pas sont nécessaires pour un contrôle sur des données acquises. Pour une fréquence d'échantillonnage de 16 kHz, la génération des signaux PWM est divisée en 100 sections, soit deux zones de 50 divisions, ce qui offre un bon compromis entre la précision du contrôle et le temps de

simulation. Comme nous avons vu à la section 2.1.2, bien que ce ne soit pas le nombre de sections qui sera utilisé dans des conditions physiques réelles, il s'agit d'un bon compromis pour réduire le temps de simulation à un stade où la précision absolue n'est pas encore nécessaire.

Ici, la taille de pas fixe est 625 ns ($\frac{1}{16\,kHz \times 100}$), ce qui est déjà assez petit pour causer un long temps de simulation du système. Puisque la génération de signaux PWM est divisée en deux, pour chaque tranche de 50 divisions, un contrôle de 122 pas est effectué. Donc, le contrôle vectoriel doit s'effectuer 2.44 fois ($\frac{122}{50}$) plus rapidement que le reste du système Simulink. La taille de pas fixe est donc divisée par 2.44 et devient environ 256 ns, ce qui rend la simulation encore plus longue, sans compter le temps perdu à cause du surdébit nécessaire pour synchroniser les pas du FPGA avec les pas Simulink. En d'autres mots, comme la génération de signaux PWM et la simulation du moteur SPS s'effectuent en peu de pas alors que le contrôle demande beaucoup de pas, le couplage des deux entités force un pas de simulation très petit et une simulation très longue.

La co-simulation libre permet au FPGA de fonctionner à pleine vitesse en tout temps. Ce n'est plus l'environnement Simulink qui dicte au bloc co-simulé lorsque son horloge doit être activée, ce qui réduit le surdébit de la communication entre FPGA et Matlab. En accord avec l'objectif de développer une méthode intéressante en milieu académique, la plateforme utilisée pour la co-simulation est le *Virtex 4 ML402 SX XtremeDSP Evaluation Platform* (figure 29). Il s'agit d'une plateforme accessible, offrant assez de puissance et de flexibilité pour assurer la co-simulation de modèles complexes. C'est également une plateforme déjà supportée par XSG pour la co-simulation, ce qui assure un bon fonctionnement lors de l'utilisation de la méthode détaillée dans cet écrit. Il n'est pas nécessaire de configurer une nouvelle cible de co-simulation dans XSG, ce qui accélère davantage le développement. La vitesse d'horloge de cette plateforme lorsqu'utilisée en co-simulation libre est de 100 MHz. Pour 122 pas, cela signifie que le contrôle vectoriel s'effectue en 1.22 µs. Le contrôle peut donc être complété aisément entre deux acquisitions de données alors que le reste du système Simulink fait son travail. Peu importe le pas de temps Simulink utilisé, il fut impossible d'observer des problèmes où la co-simulation n'avait pas le temps de traiter les données qui lui étaient envoyées.

Le désavantage de cette méthode de simulation est que, contrairement à la co-simulation pas à pas, la synchronisation doit être assurée par le modèle XSG et non par Simulink. Dans le cas du contrôle vectoriel, un simple signal de démarrage est envoyé au bloc afin d'amorcer son traitement. Comme il est impossible de prédire combien de signaux seront alors vus par le bloc XSG, ce dernier utilise un simple mécanisme (figure 28) afin de ne s'activer qu'à la fin d'un signal de démarrage. Celui-ci peut donc être d'une durée arbitraire et un seul traitement sera lancé.

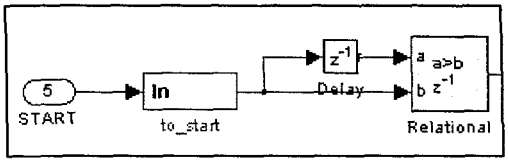


figure 28 - Simple synchronisation d'un bloc XSG co-simulé

Le signal de démarrage est doublé, et un délai est inséré dans un des deux signaux. Ceux-ci sont enfin comparés afin de déterminer la fin du signal envoyé, et c'est alors qu'une pulsation est communiquée au contrôle vectoriel.

4.8 Test sur matériel

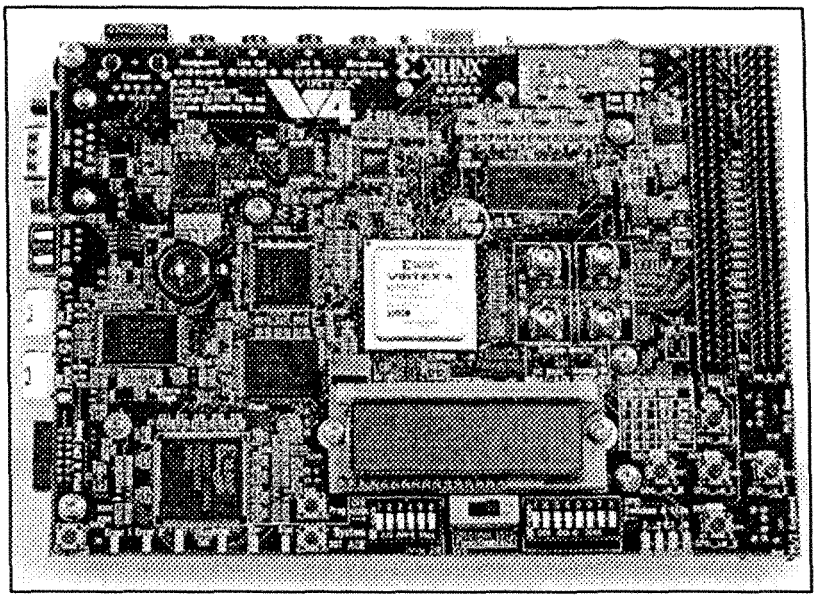


figure 29 - Carte de co-simulation Xilinx ML402

L'option de co-simulation offerte par XSG assure également que le modèle testé va répondre correctement une fois implémenté sur la plateforme matérielle ciblée. Une fois qu'un prototype XSG a été suffisamment testé dans l'environnement Simulink et que l'analyse temporelle assure à l'usager que le modèle peut fonctionner à la vitesse désirée, il est nécessaire de reconfigurer un FPGA avec la logique du design pour vérifier que la transition vers le domaine matériel n'occasionne pas de bogues imprévus. Une fois sur FPGA, le design est souvent analysé avec un outil tel que Chipscope qui permet d'analyser les signaux internes de la puce en pleine action. Chipscope est d'ailleurs disponible depuis XSG, ce qui facilite grandement son utilisation au sein d'un projet.

Dans le cas d'une co-simulation libre, la partie XSG du système fonctionne entièrement sur le FPGA de la carte de co-simulation utilisée, et ce, à pleine vitesse. Le comportement sur matériel peut donc tout de suite être analysé, débogué et optimisé. Si un problème non présent lors de simulations dans l'environnement Simulink (du modèle XSG) est introduit en co-simulation, l'outil chipscope (qui s'intègre facilement dans XSG et Simulink) ou de simples sorties temporaires de débogage permettent de rapidement identifier la cause de la complication. Même si l'usager

désire analyser un grand nombre de signaux internes lors de la co-simulation, l'utilisation de la communication Ethernet gigabit offre une large bande passante qui le permet.

Dans le cas de l'application du contrôle vectoriel développé dans cette recherche, cette technique a permis d'identifier un problème de dépassement lors de l'utilisation du bloc soustraction XSG où la gestion du dépassement doit être mise à *wrap*. Une fois ces problèmes réglés, l'analyse *FPGA-in-the-loop* peut être lancée.

4.9 FPGA-in-the-loop; co-simulation avec SimPowerSystems

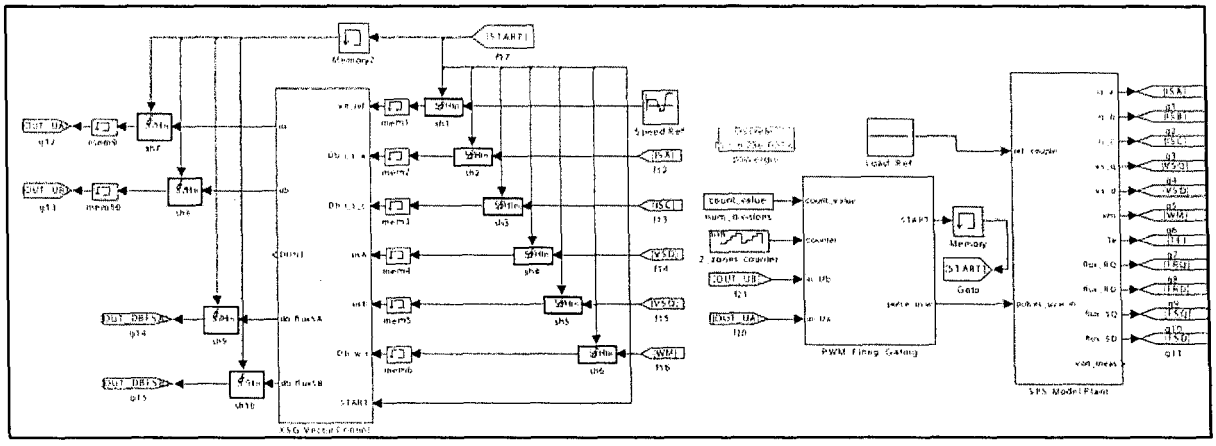


figure 30 - Co-simulation avec portillonnage, moteur et module de puissance

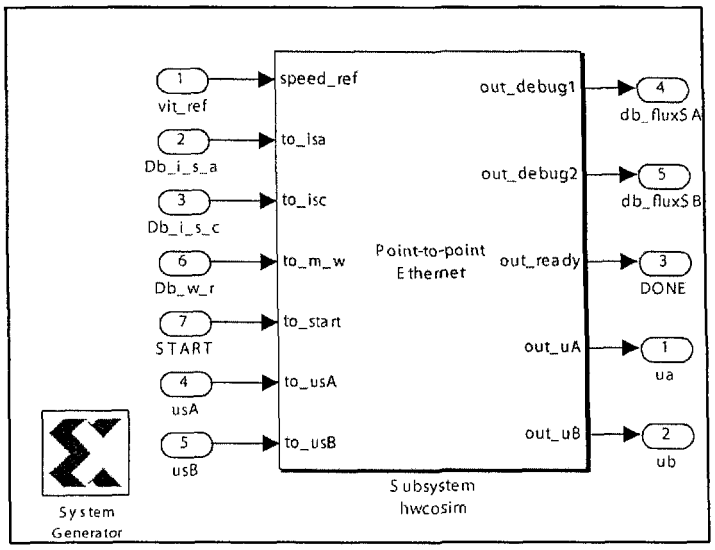


figure 31 - Interface du bloc XSG co-simulé

Lors des tests et de l'analyse d'un système de contrôle fait avec XSG, une des difficultés majeures est de vérifier ce design en boucle fermée. Bien qu'une simulation en boucle ouverte avec des valeurs de référence soit efficace pour confirmer le bon fonctionnement de l'algorithme, cette méthode comporte deux désavantages. Premièrement, elle ne représente pas comment le système va se comporter avec rétroaction, et deuxièmement, il faudra de toute façon effectuer un test en boucle fermée avant de brancher le système à l'équipement de laboratoire.

Le test en condition de boucle fermée utilise le module de puissance et le moteur simulé avec SPS (voir section 2.2.3). Le portillonnage PWM est simulé par le bloc de « fonction embarquée » Matlab. Ici, pour des raisons de synchronisation, on ne peut co-simuler le portillonnage sur FPGA car il doit fonctionner selon la même horloge que le bloc SPS.

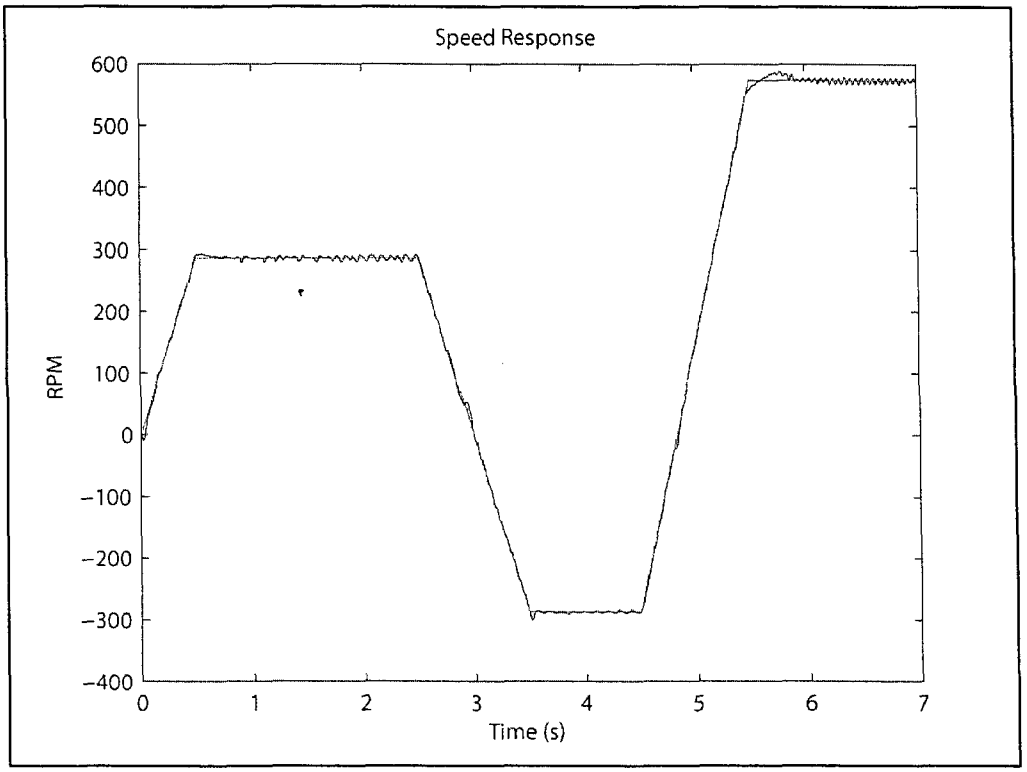


figure 32 - Réponse du système XSG co-simulé (rouge, lisse = profil, bleu = réponse)

Type de simulation	Temps de simulation
co-simulation libre	1734 s
co-simulation pas à pas	174610 s (48 heures)
Simulation du modèle Simulink en mode accéléré	763 s

tableau 3 - Temps des 3 méthodes de simulation

Le tableau 3 affiche les performances de simulation pour un profil de vitesse donné (qui est le même pour les trois tests, voir figure 32). Les tests sont tous effectués sur la même plateforme, soit un PC Intel P4 3.4GHz, pour une même précision de 50 divisions par demi-période de commutation (voir section 2.1.2). Les résultats montrent que si l'on compare avec un modèle de contrôle entièrement Simulink (en mode accéléré de Simulink), une simulation du modèle XSG avec co-simulation pas à pas demande environs 229 fois plus de temps. La simulation sans aucune co-simulation est encore plus longue, ce qui rend ces deux modes de fonctionnement non

envisageables pour un développement rapide. En revanche, la co-simulation libre ne demande qu'environ 2.27 fois le temps nécessaire à la simulation Simulink. Des tests avec différents profils et des blocs individuels (plutôt que le contrôle entier) confirment un facteur d'environs 2 ou 3 fois en temps de simulation.

Les résultats des expériences conduites dans le cadre de cette recherche permettent de conclure que si le temps de simulation Simulink est acceptable pour l'usager, la co-simulation libre représente une bonne option. Si le temps Simulink est déjà inacceptable pour l'usager, il doit alors envisager une autre solution de simulation (plateforme en temps réel, etc.).

4.10 Résultats de profils simulés

Le premier profil de vitesse (figure 33) représente une co-simulation du système entier avec des filtres théoriques de $4^{ème}$ ordre et de fréquence 5000 Hz. Le nombre de divisions d'une demi-période de commutation est à 256, ce qui offre une grande précision, mais rallonge le temps de simulation. Les articles de conférence à l'annexe I contiennent également d'autres profils et résultats obtenus grâce au travail produit dans cette recherche.

Pour chaque figure, la ligne ROUGE (lisse) identifie le profil et la ligne BLEUE (plus sinueuse), la réponse du moteur.

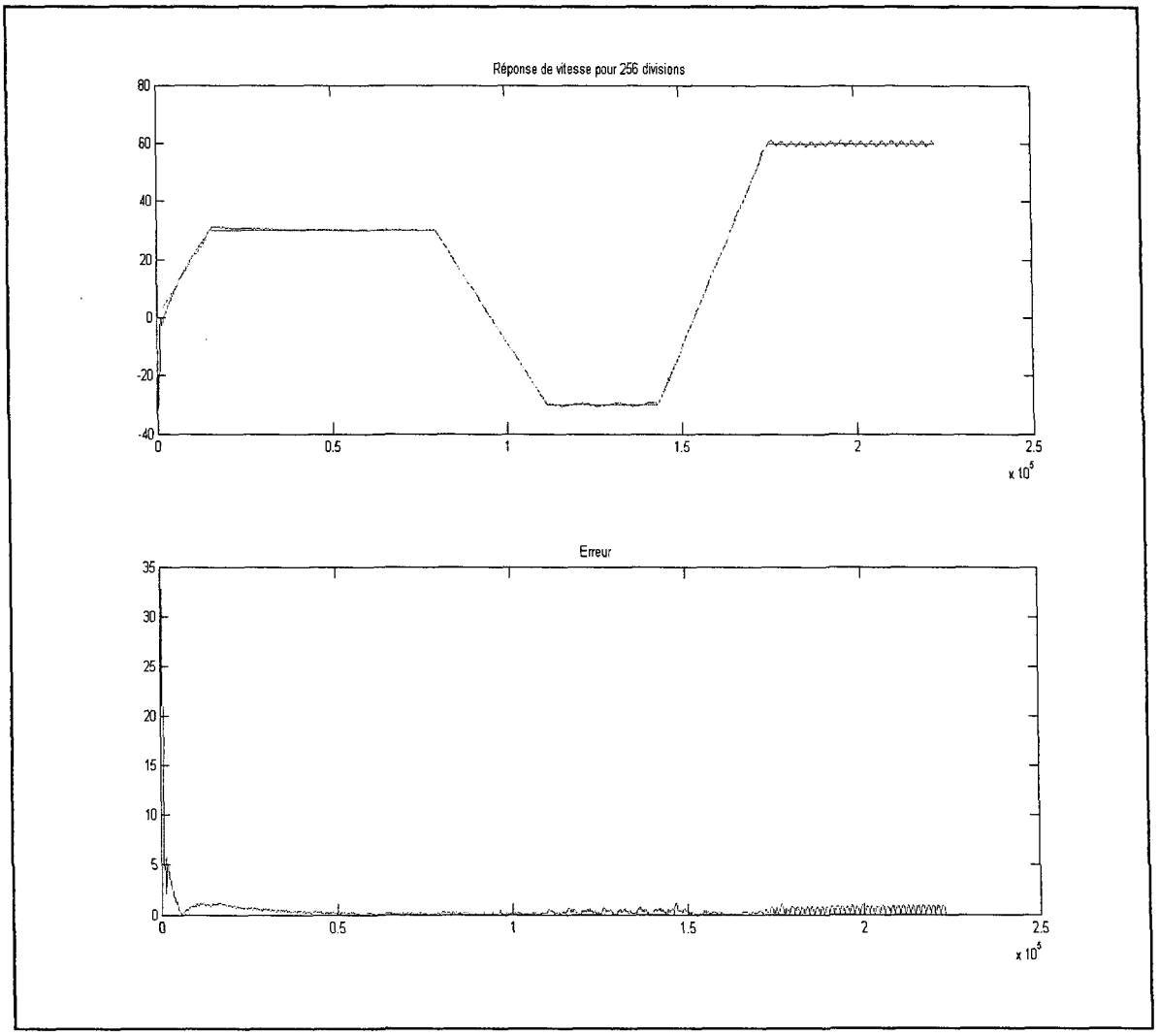


figure 33 - Réponse de vitesse - 256 divisions - Filtre 5000 Hz

La seconde figure 34 représente la même simulation, mais avec un nombre de 100 divisions, ce qui rend la simulation 2.7 fois plus rapide en gardant une allure de courbe très acceptable (pour des fins de simulation). Il s'agit du type de modification de précision que l'usager peut faire afin de tester la réponse de son système dans plusieurs cas critiques comme des dépassements. Une fois le bon comportement confirmé, une simulation qui demande plus de précision peut être lancée.

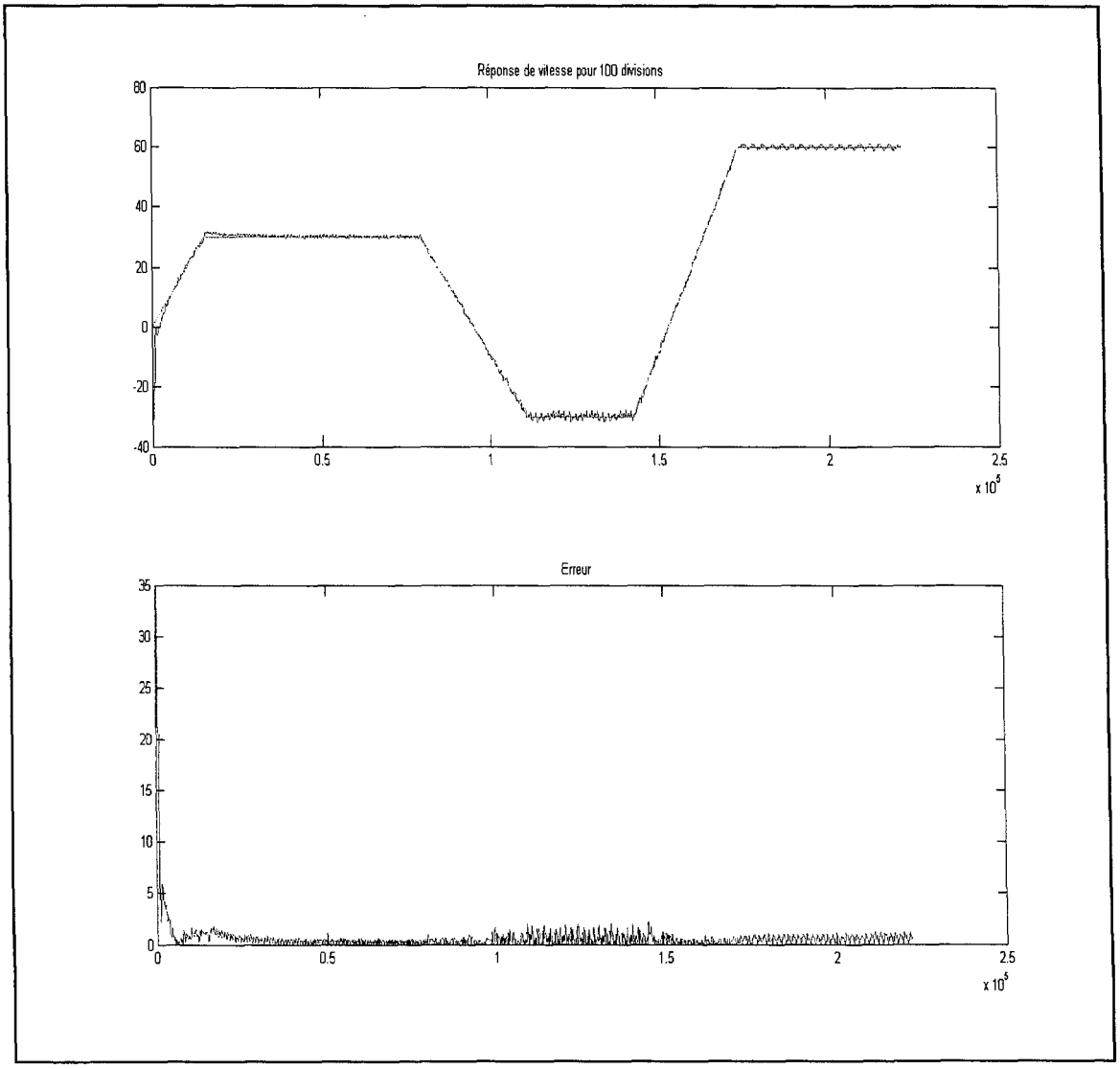


figure 34 - Réponse de vitesse - 100 divisions - Filtre 5000 Hz

Finalement, la figure 35 montre l'effet de changer le filtre pour une fréquence plus réaliste de 500 Hz, ce qui engendre les oscillations attendues à plus grande vitesse.

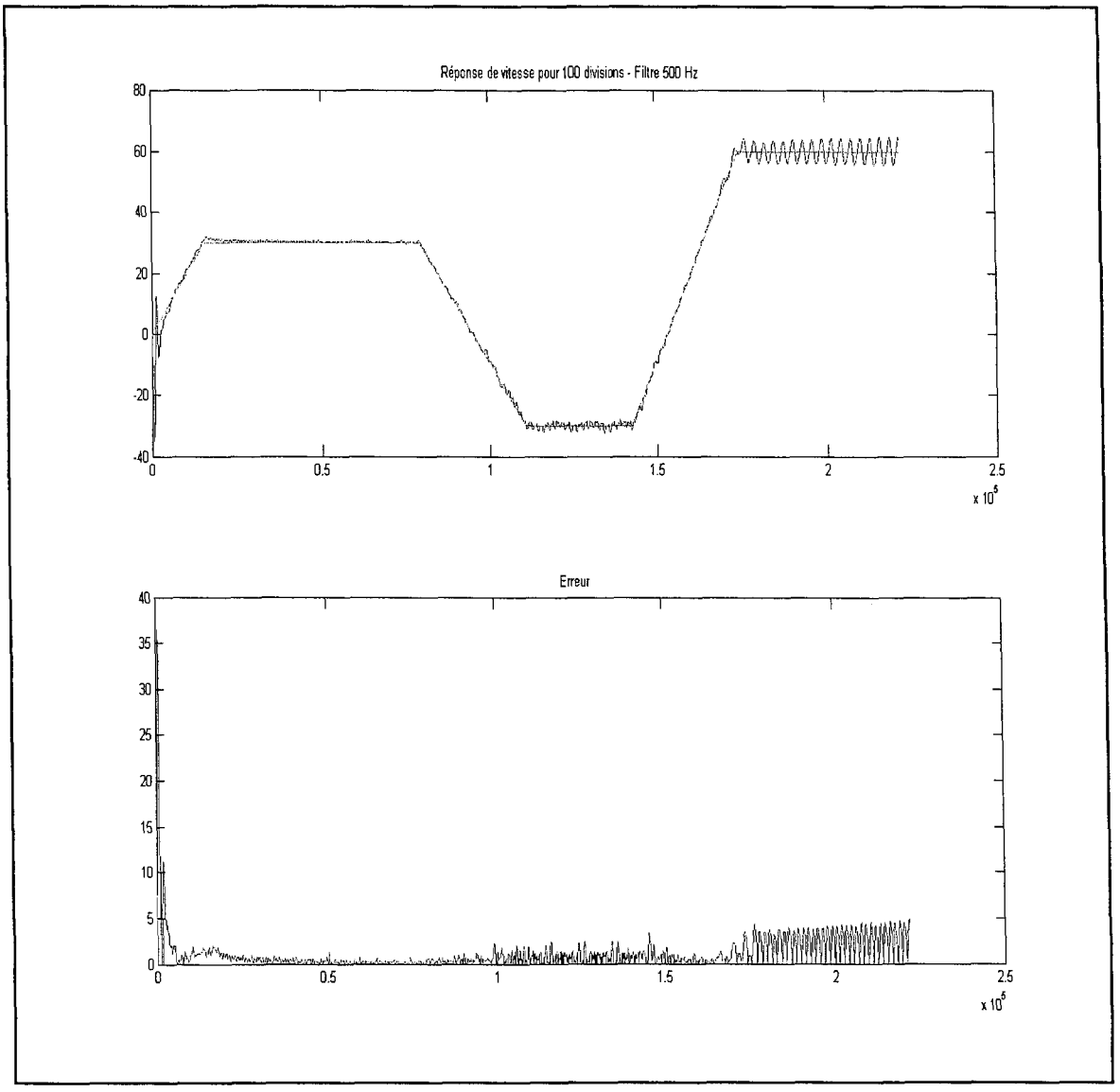


figure 35 - Réponse de vitesse - 100 divisions - Filtre 500 Hz

CHAPITRE 5

Résumé de la méthode, travaux futurs et conclusion

L'établissement d'une méthode de prototypage rapide est un objectif important de ce travail. Il est donc important de bien en isoler les étapes. Cette description modulaire permet à la fois à l'usager de la méthode de suivre le processus de développement proposé par l'auteur de cette recherche, mais aussi de mieux évaluer l'envergure du travail pour chacune des étapes à franchir.

Le prototypage rapide sur la plateforme de développement doit franchir les étapes suivantes :

5.1 Schématisation de l'algorithme

Voir section 2.1

5.2 Référence Matlab et collecte de données

Voir section 2.2

5.3 Création des scripts GAPPA (Pre-XSG) en pire cas possible (PCP)

Voir section 4.2

5.4 Création du fichier d'initialisation Matlab

Voir section 4.2.2 et 3.5

5.5 Création du modèle System Generator

Voir section 3

5.6 Optimisation avec GAPPA (Post-XSG)

Voir section 4.3

5.7 Co-simulation en boucle fermée (FPGA-in-the-loop)

Voir section 4.9

5.8 Instanciation du projet XSG sur la plateforme Amirix

Voir section Annexe B

5.9 Travaux futurs et ajouts intéressants à la méthode

La méthode de développement rapide créée dans cette recherche permet maintenant à l'usager de partir d'un algorithme d'application et de le porter vers une plateforme expérimentale afin de prouver la faisabilité du concept. Les étapes de la méthode sont très détaillées de manière à éviter à l'usager de commettre des erreurs de configuration lors de l'utilisation des outils impliqués dans le processus. Le cheminement est également décrit de manière à ce que l'usager ne perde pas de temps supplémentaire lors des phases de débogage (grâce à GAPPA) et lors de l'implémentation sur la carte Amirix (grâce au livrable CMC).

Une des forces de la méthode de cette recherche réside dans le fait d'utiliser des outils académiques disponibles à l'équipe de recherche, plutôt que de payer un tiers parti pour une solution complète en un produit qui n'offre pas nécessairement les performances désirées. La flexibilité de pouvoir effectuer séparément la mise à jour des différents modules de la méthode permet de rapidement bénéficier des améliorations apportées par ces nouvelles versions. Cependant, certaines tâches pour passer d'un outil à l'autre gagneraient à être automatisées davantage.

Le meilleur exemple est la génération d'un script Matlab et des blocs XSG personnalisés. Actuellement, l'usager doit utiliser la même nomenclature entre ces trois outils et copier les valeurs de GAPPA vers le fichier Matlab, pour ensuite copier les noms de variable dans des blocs XSG correspondants.

Beaucoup de temps serait économisé avec la création d'un outil qui :

- analyse les scripts GAPPA pour déterminer les noms et valeurs des variables de point binaire;
- exécute ces scripts pour ensuite évaluer le nombre de bits nécessaire pour la partie entière des opérandes;
- crée un fichier d'initialisation Matlab;

- crée un fichier MDL contenant les blocs XSG avec leurs paramètres.

Présentement, les co-simulations du contrôle vectoriel fonctionnent et offrent des résultats intéressants. Cependant, les tests avec le moteur du laboratoire n'ont pas encore été effectués. Il faut encore implémenter un bon lecteur de vitesse provenant d'un encodeur optique [13] et obtenir le connecteur d'expansion fabriqué par un technicien pour la carte Amirix AP1000.

5.10 Possibilités futures et applications

Avec une plateforme de développement ainsi qu'une méthode de prototypage pour l'utiliser, il est maintenant possible de traiter de nouveaux problèmes dans des domaines divers afin de rapidement prouver le fonctionnement d'un concept à l'aide d'un prototype sur puce FPGA.

De plus, le contrôle vectoriel conçu avec XSG est utilisé deux fois par période de commutation (voir section 2.1.2), ce qui démontre le fonctionnement de cette méthode pour obtenir une génération de signaux PWM asymétrique. Bien qu'en simulation, les résultats avec et sans cette méthode ne diffèrent pas, il s'agit d'un moyen de réduire les harmoniques fournies au réseau et de produire une onde sinusoïdale plus propre. Il sera intéressant de vérifier le comportement une fois les tests en laboratoire effectués. C'est une méthode qui serait avantageuse particulièrement lors de l'usage d'onduleurs multi-étages, ce qui peut également servir de base pour une recherche intéressante.

5.11 Conclusion

Les objectifs de cette recherche se divisaient en trois volets qui ont été abordés et complétés. Premièrement, l'établissement d'une plateforme de travail à l'aide d'outils académiques modulaires qui permet de passer d'un algorithme ou d'un problème vers un prototype fonctionnel pouvant être testé en laboratoire. Dans le domaine de la recherche, cela est spécialement important car cela permet d'obtenir la précision voulue et la puissance désirée de chaque module de la plateforme. Une solution commerciale intégrée n'offre souvent pas cette flexibilité (sans parler des coûts). Ici, chaque outil ou module matériel peut être changé selon la direction de la recherche. L'utilisation des outils retrouvés dans le milieu académique au lieu d'une coûteuse et complexe solution intégrée d'un tiers parti présente aussi de nombreux avantages dans un cadre de recherche. Un étudiant peut rapidement appliquer ses connaissances actuelles des outils et participer à la réalisation d'un projet sans nécessiter une longue formation. Pour le groupe de recherche, il s'agit d'une économie car le prix d'une telle formation pour du personnel temporaire peut souvent s'avéré très important. De plus, le milieu universitaire a souvent accès aux nouvelles licences des dernières versions de logiciels comme XSG et Matlab.

Ensuite, une méthode de prototypage rapide fut développée afin d'offrir à l'usager de cette plateforme de développement une méthode efficace de conception, de validation et d'optimisation. Ceci permet d'éviter les pertes de temps souvent occasionnées par des erreurs de configuration, un manque de documentation des outils ou un manque de rigueur. Il s'agit aussi d'un outil permettant de mieux estimer les temps de développement, de débogage et de simulation avant de commencer un projet. De plus, cette méthode assure le lien entre les différents outils comme GAPPA, XSG, XPS, etc. Les scripts de migration entre les applications ainsi que les méthodes jugées optimales d'intégration de chaque outil sont fournies afin que l'usager perde le moins de temps possible et puisse simplement se concentrer sur l'application de sa recherche. L'intégration de GAPPA au sein du processus de conception est un apport important de

cette recherche qui permet d'économiser beaucoup de temps, et qui simplifie énormément l'identification de bogues dans un modèle complexe.

Finalement, l'application du contrôle vectoriel devant répondre à des conditions d'exécutions sévères (un court temps de réponse, une taille sur FPGA décente, etc.) offre une réponse simulée très acceptable en respectant les contraintes établies. Ce contrôle pourra maintenant être testé avec l'équipement en laboratoire. C'est aussi cette application qui a permis de développer la méthode efficace de conception, sans quoi, elle n'aurait été qu'une fabrication purement théorique sans preuve à l'appui. Ce problème complexe permet de vérifier la faisabilité d'utiliser la méthode de prototypage pour générer un prototype fonctionnel et des temps de développement que cela implique. Notamment, cet exemple a permis de vérifier l'efficacité d'utiliser la co-simulation libre d'une nouvelle manière afin de simuler des modèles XSG avec attente liés à d'autres modules Simulink ordinaires. Cette recherche vérifie qu'une telle utilisation du mode de co-simulation libre est fiable, et elle offre une idée des temps de simulation auxquels l'usager peut s'attendre.

L'application de contrôle vectoriel a également permis de tester en pratique les versions non restaurantes optimisées pour FPGA de la racine carrée et de la division. Ces résultats furent importants afin de publier les deux articles sur le sujet que l'on retrouve à l'annexe I.

Les divers résultats de cette recherche offrent donc une fondation solide à l'équipe de recherche pour attaquer diverses problématiques d'application, tout en ayant les outils nécessaires pour mieux estimer le temps de développement requis pour ce genre de tâche.

BIBLIOGRAPHIE

[1] E. Monmasson and Y. A. Chapuis, "Contributions of FPGAs to the Control of Electrical Systems - a Review," **IEEE Electronics Society Newsletter**, Vol. 49, No. 4, 2002.

[2] Janne Salomäki, Marko Hinkkanen and Jorma Luomi, **Sensorless Vector Control of an Induction Motor Fed by a PWM Inverter Through an Output LC Filter**, IEEJ Transactions on Industry Applications, Vol. 126 (2006) , No. 4 pp.430-437.

[3] Leppanen, V.M., Luomi, J., "Observer using low-frequency injection for sensorless induction motor control-parameter sensitivity analysis", **Electric Machines and Drives Conference**, 2003. IEMDCapos, IEEE International Volume 1, Issue , 1-4 June 2003 Page(s): 609 - 616 vol.1.

[4] F. Aubepart, P. Poure, Y.A. Chapuis, C. Girerd, F. Braun, "HDL-Based Methodology for VLSI Design of AC Motor Controller" **IEE Proceedings – Circuits, Devices and Systems**, Vol. 150, No. 1, Feb., 2003, pp. 38–44.

[5] M.N. Cirstea, A. Dinu, M. McCormick, and D. Nicula, "VHDL Success Story: Electric Drive System Using Neural Controller," **Proc. of VHDL International Users Forum Fall Workshop**, Orlando, FL, 2000, pp. 118–122.

[6] M.N. Cirstea, A. Aounis, M. McCormick, "Rapid Prototyping of Induction Motor Vector Control System Based on Reusable VHDL Digital Architectures and FPGA Implementation," **Proc. of Int. Conf. on Power Conversion & Intelligent Motion (PCIM'02)**, Nurnberg, Germany, May, 2002, pp. 199–202.

[7] Guillaume Melquiond, Florent de Dinechin, Christoph Lauter, **"Assisted verification of elementary functions using Gappa",** SAC'06 symposium, 2006, Dijon, France.

[8] Kuffel, R.; Giesbrecht, J.; Maguire, T.; Wierckx, R.P.; McLaren, P., "RTDS - a fully digital power system simulator operating in real time", **Energy Management and Power Delivery**, 1995. Proceedings of EMPD apos;95., 1995 International Conference on Volume 2, Issue , 21-23 Nov 1995 Page(s):498 - 503 vol.2.

[9] Dufour, Christian; Abourida, Simon; Belanger, Jean; Lapointe, Vincent, **Real-Time Simulation of Permanent Magnet Motor Drive on FPGA Chip for High-Bandwidth Controller Tests and Validation,** Page(s): 4581-4586, IECON.2006.

[10] Bin Lu; Monti, A.; Dougal, R.A., "Real-time hardware-in-the-loop testing during design of power electronics controls", Industrial Electronics Society, 2003. IECON apos;03. **The 29th Annual Conference of the IEEE**, Volume 2, Issue , 2-6 Nov. 2003 Page(s): 1840 - 1845 Vol.2 IECON.2003. Roanoke (VA)(USA).

[11] Monti A., E. Santi, R.Dougal, M. Riva, "Rapid Prototyping of Digital Controls for Power Electronics", **IEEE Trans. On Power Electronics**, May 2003 Issue.

[12] **Microchip, dsPICDEMTMMC1H 3-Phase High Voltage Power Module User's Guide**, Microchip Technology Inc., 2003.

[13] Pamela Bhatti, Blake Hannaford, **Single Chip Velocity Measurement System for Incremental Optical Encoders**, In Press, IEEE Transactions on Control Systems Technology, June 1997.

[14] **Stator Resistance Tuning in an Adaptive Direct Field-Orientation IM Drive at low speeds**, R. Beguenane, M. Ouhrouche, and Andrzej M. Tzynadlowski, Proceeding of the 30th annual Conference of the IEEE Industrila Electronics Society, IECON-04, Pusan, South Korea, November 2-6,2004.

[15] Mailloux, Jean-Gabriel; Simard, Stéphane; Beguenane, Rachid, **Rapid Testing of XSG-Based Induction Motor Vector Controller Using Free-Running Hardware Co-Simulation and SimPowerSystems**, The 4th International Conference on Cybernetics and Information Technologies, Systems and Applications CITSA 2007.

[16] S. Simard, J. G. Mailloux, R. Beguenane, **Optimal FPGA implementation of Unsigned Bit-serial division**, International Review on Computers and Software, September 2007 issue

[17] Mailloux, Jean-Gabriel; Simard, Stéphane; Beguenane, Rachid, **Rapid Testing of XSG-Based Induction Motor Vector Controller Using Free-Running Hardware Co-Simulation and SimPowerSystems**, The 4th International Conference on Cybernetics and Information Technologies, Systems and Applications CITSA 2007.

[18] Rachid Beguenane; Jean-Gabriel Mailloux; Stephane Simard; Arnaud Tisserand. **Towards the System-on-Chip Realization of a Sensorless Vector Controller with Microsecond-order Computation Time Electrical and Computer Engineering**, 2006. CCECE apos;06. Canadian Conference on Electrical and Computer Engineering, May 2006 Page(s):1073 - 1077.

ANNEXE A

Équations du contrôle vectoriel

Tiré de l'article *"Towards the System-on-Chip Realization of a Sensorless Vector Controller with Microsecond-order Computation Time"*

1) Speed PI Controller:

$$i_{sq}^{*} = k_{p_v}\epsilon_v + k_{i_v}\int \epsilon_v \, dt \; ; \quad \epsilon_v = \omega_r^{*} - \omega_r$$

2) Rotor Flux PI Controller:

$$i_{sd}^{*} = k_{p_f}\epsilon_f + k_{i_f}\int \epsilon_f \, dt \; ; \quad \epsilon_f = \Psi_r^{*} - \Psi_r$$

3) Rotor Flux Estimator Block:

$$\begin{aligned} \Psi_r &= \sqrt{\Psi_{r\alpha}^2 + \Psi_{r\beta}^2} \\ \cos\theta &= \frac{\Psi_{r\alpha}}{\Psi_r} \; ; \quad \sin\theta = \frac{\Psi_{r\beta}}{\Psi_r} \end{aligned}$$

with

$$\begin{aligned} \Psi_{r\alpha} &= \frac{L_r}{M}\left(\Psi_{s\alpha} - \sigma L_s i_{s\alpha}\right) \\ \Psi_{r\beta} &= \frac{L_r}{M}\left(\Psi_{s\beta} - \sigma L_s i_{s\beta}\right) \\ \Psi_{s\alpha} &= \int \left(u_{s\alpha} - R_s i_{s\alpha}\right) \\ \Psi_{s\beta} &= \int \left(u_{s\beta} - R_s i_{s\beta}\right) \end{aligned}$$

4) Current PI Controller:

$$\begin{aligned} v_{sd} &= k_{p_i}\epsilon_{i_{sd}} + k_{i_i}\int \epsilon_{i_{sd}} \, dt \; ; \quad \epsilon_{i_{sd}} = i_{sd}^{*} - i_{sd} \\ v_{sq} &= k_{p_i}\epsilon_{i_{sq}} + k_{i_i}\int \epsilon_{i_{sq}} \, dt \; ; \quad \epsilon_{i_{sq}} = i_{sq}^{*} - i_{sq} \end{aligned}$$

5) Decoupling:

$$u_{sd} = \sigma L_s v_{sd} + D_d \; ; \quad u_{sq} = \sigma L_s v_{sq} + D_q$$

with

$$\begin{aligned} D_d &= -\sigma L_s \omega i_{sq} + \frac{M}{L_r}\frac{\mathrm{d}}{\mathrm{d}t}\Psi_r \\ D_q &= +\sigma L_s \omega i_{sd} + \frac{M}{L_r}\omega \Psi_r \end{aligned}$$

6) ω Estimation:

$$\omega = P_p \omega_r + \frac{M\beta_r}{\Psi_r} i_{sq}$$

7) Clarke Transforms:

$$\begin{aligned} i_{s\alpha} &= i_{sa} \\ i_{s\beta} &= \frac{1}{\sqrt{3}} i_{sa} + \frac{2}{\sqrt{3}} i_{sb} \end{aligned}$$

and

$$\begin{aligned} u_{s\alpha} &= u_{sa} \\ u_{sb} &= -\frac{1}{2} u_{s\alpha} + \frac{\sqrt{3}}{2} u_{s\beta} \\ u_{sc} &= -\frac{1}{2} u_{s\alpha} - \frac{\sqrt{3}}{2} u_{s\beta} \end{aligned}$$

8) Park Transforms:

9) PT:

$$\begin{aligned} \begin{pmatrix} i_{sd} \\ i_{sq} \end{pmatrix} &= \mathrm{PT} \begin{pmatrix} i_{s\alpha} \\ i_{s\beta} \end{pmatrix} \\ \mathrm{PT} &= \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \end{aligned}$$

10) PT^{-1}:

$$\begin{aligned} \begin{pmatrix} u_{s\alpha} \\ u_{s\beta} \end{pmatrix} &= \mathrm{PT}^{-1} \begin{pmatrix} u_{sd} \\ u_{sq} \end{pmatrix} \\ \mathrm{PT}^{-1} &= \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \end{aligned}$$

u_{sd}, u_{sq}	Stator voltage of d-axis and q-axis
i_{sd}, i_{sq}	Stator current of d-axis and q-axis
Ψ_r	Rotor flux modulus
ω	Angular speed of the (d, q) reference frame
L_s, L_r	Stator and rotor inductances
M	Mutual inductance
R_s, R_r	Stator and rotor resistances
σ	Leakage coefficient of the motor
β_r	Constant: $\frac{R_r}{L_r}$
P_p	Number of pole pairs
ω_r	Rotor speed, or *angular frequency* (measured or estimated)
J	Inertial momentum
f	Friction coefficient
T_l	Torque load

ANNEXE B

Plateforme Amirix

6.1 Connectivité

6.1.1 AIO

C'est la carte AIO qui, sur la plateforme de travail, agit comme interface entre le FPGA de la carte Amirix et l'électronique de puissance du laboratoire. Les 8 entrées analogiques qui supportent un échantillonnage de 2 MSPS sont suffisamment rapides pour supporter la dynamique explorée à la section 2.1.2 où les détails quant à la vitesse d'échantillonnage des tensions et courants sont exposés.

Un ruban de 80 conducteurs lie la carte à une plaque d'interconnexion qui est ensuite reliée aux ports de l'électronique de puissance. Cette plaque a été conçue par un technicien selon les spécifications de notre équipe de recherche. Les 3 tensions et 3 courants du moteur sont, après avoir été filtrés via des filtres aussi produits par le technicien de département (spécification des filtres à la section 2.2.3), convertis par la carte AIO et ces valeurs rejoignent le FPGA via le PMC qui est rattaché au *Processor Local Bus* (PLB) via le Processor Bus Dual PCI Bridge (figure 36).

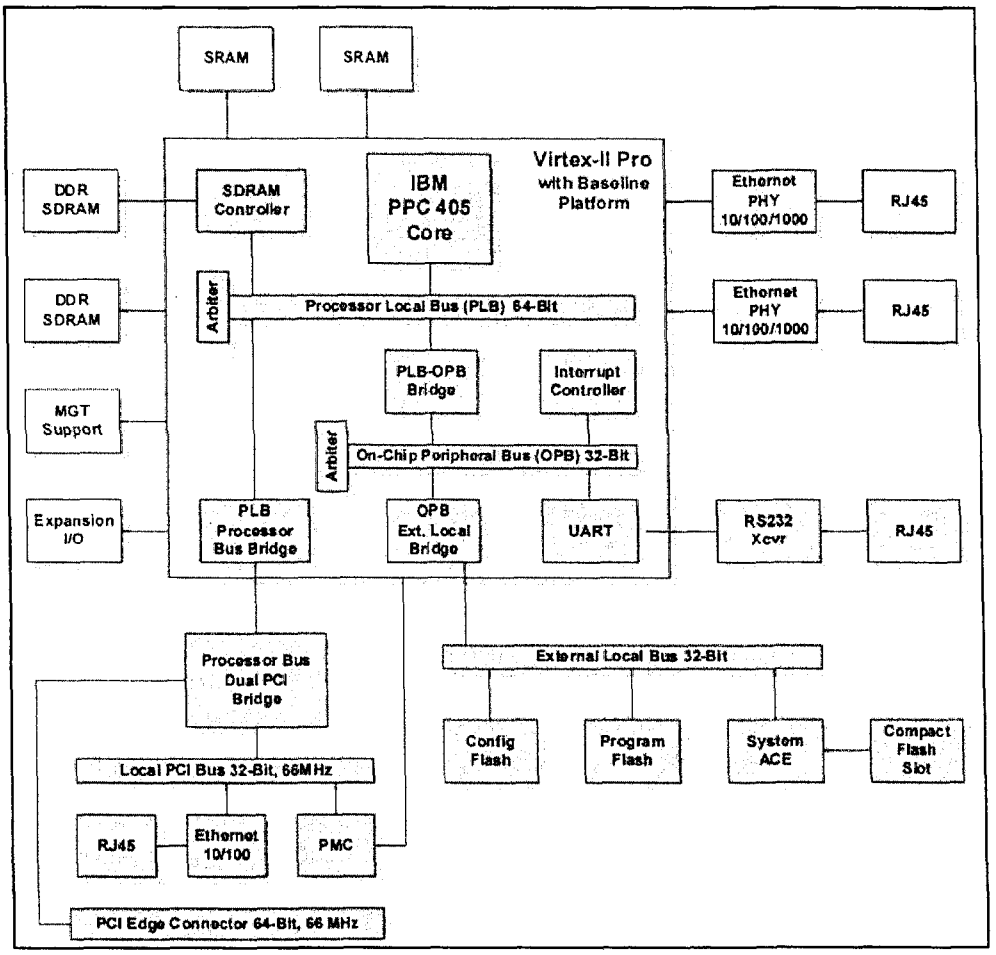


figure 36 - Plateforme Amirix AP1000

6.1.2 FPGA

Même si la carte AIO possède des entrées et des sorties numériques, ce ne sont pas ces dernières qui sont utilisées afin d'envoyer les signaux PWM au module de puissance et de recevoir les signaux de l'encodeur optique de vitesse du moteur. Il s'agit d'opérations où le nombre de pas d'horloge est critique. L'utilisation de la communication sur le PLB et le PMC ne peut garantir, même s'il y a peu de trafic sur le bus dans notre cas, le nombre de pas exact avant le transfert d'une valeur entre le FPGA et la carte AIO. De plus, le portillonnage et la lecture de vitesse doivent

s'exécuter en parallèle avec le contrôle vectoriel. Cela signifie qu'un seul de ces processus peut lire ou écrire à la fois sur la carte AIO (dans le mode de communication choisi pour les besoins de cette recherche).

Ces raisons font que le portillonnage et la lecture de vitesse seront gérés directement par le FPGA via les connecteurs d'expansions de la carte Amirix AP1000. Malheureusement, bien que très nombreux, les ports d'entrée et de sortie de la carte utilisent des connecteurs Samtec CON 0.8mm. Une carte d'adaptation sur mesure composée d'un circuit imprimé et des connecteurs correspondants est fabriquée par le technicien de département afin de convertir les connexions en broches standards qui facilitent l'interface avec l'équipement du laboratoire. Cela facilite aussi la tâche de débogage, car les signaux sont beaucoup plus faciles à observer sur un oscilloscope grâce à cet adaptateur de connexion.

6.2 Logique matérielle

6.2.1 Le Project Xilinx Platform Studio (XPS)

L'outil utilisé pour la gestion de la logique matérielle du FPGA de la carte Amirix est le Xilinx Platform Studio (XPS). CMC fournit un projet de base modifié (basé sur une source fournie par Amirix) qui contient tous les fichiers nécessaires pour la génération d'un fichier BIN à instancier sur le FPGA. Ce fichier BIN est transféré à la mémoire de configuration Flash de la carte pour ensuite reconfigurer le FPGA. Au cours de cette recherche, la programmation directe via JTAG n'est pas utilisée, car il fût découvert que cela rend le Linux embarqué inopérant. Ce point important n'étant nulle part documenté, l'usager de la méthode doit ici en prendre bonne note. Une fois programmée, la logique de l'usager attend la confirmation du logiciel d'initialisation de la carte PMC avant de commencer son traitement.

Dans le cadre du contrat CMC, le traitement effectué par le FPGA dans la phase 1 du contrat est un filtre FIR. La phase 2 du même contrat implique la création d'opérations plus compliquées, où l'usager peut sélectionner les signaux qu'il désire envoyer à la carte AIO à partir de générateurs

d'onde sinusoïdale CORDIC (crées pour le contrat). Le fichier VHDL est conçu de manière à simplifier l'implémentation de n'importe quelle opération sans avoir besoin de modifier le reste du code, ce qui est l'intérêt principal au sein de la méthode de prototypage rapide de cette recherche. Le projet contient un bloc *hw_math* (Hardware Math Operation) déjà instancié et connecté au PLB à l'adresse 0x2A001000 (figure 37). Les LEDS et commutateurs de la carte Amirix sont aussi connectés en tant que sorties et entrées, respectivement. Même le logiciel de l'utilisateur (qui fonctionne sur la plateforme Linux) peut accéder aux valeurs des commutateurs via le registre de statut et de contrôle (CSR illustré à la figure 40).

Filters: Bus Interface / Ports / Addresses | Generate Addresses

Instance	Name	Address	Base Address	High Address	Size	Lock	ICache	DCache	Bus Connection
plb_psb_bridge_i			0x30000000	0x3FFFFFFF	256M	☐			
opb_bus			0x4E000000	0x4EFFFFFF	16M	☐			
plb_psb_bridge_i		PLB_PSB_FPGA_REG	0x30002000		U	☐			
ppc405_i	MDCR	DSOCM_DCR	0b1000100000	0b1000100011	4	☐			dcr_bus
ppc405_i	MDCR	ISOCM_DCR	0b1000010000	0b1000010011	4	☐			dcr_bus
ppc405_ppcjtag_chain	MDCR	DSOCM_DCR			U	☐			No Connection
ppc405_ppcjtag_chain	MDCR	ISOCM_DCR			U	☐			No Connection
hw_math_0	MSPLB		0x2A001000	0x2A0011FF	512	☐			plb_bus
plb_bus	SDCR		0b0000000000	0b0011111111	256	☐			dcr_bus
plb2opb_bridge_i	SDCR	DCR	0b0100000000	0b0111111111	256	☐			dcr_bus
opb_uart16550_i	SOPB		0x4C000000	0x4CFFFFFF	16M	☐			opb_bus
opb_intc_i	SOPB		0x4D000000	0x4DFFFFFF	16M	☐			opb_bus
plb_ddr_controller_i	SPLB		0x00000000	0x1FFFFFFF	512M	☐			plb_bus
plb_bram_if_cntlr_i	SPLB	c_baseaddr:c_highaddr	0xFFFF0000	0xFFFFFFFF	64K	☐	☐	☐	plb_bus
plb2opb_bridge_i	SPLB	RNG0	0x20000000	0x27FFFFFF	128M	☐			plb_bus
plb2opb_bridge_i	SPLB	RNG1	0x28000000	0x29FFFFFF	32M	☐			plb_bus
plb2opb_bridge_i	SPLB	RNG2	0x4C000000	0x4FFFFFFF	64M	☐			plb_bus
plb2opb_bridge_i	SPLB	RNG3			U	☐			plb_bus

figure 37 - Instance de hw_math dans le Baseline Amirix

Le bloc hw_math est instancié sur le PLB. C'est un choix avantageux, car un bloc à cet endroit peut effectuer une communication de type master vers le bus PSB, ce qui serait impossible à partir du bus OPB (figure 38).

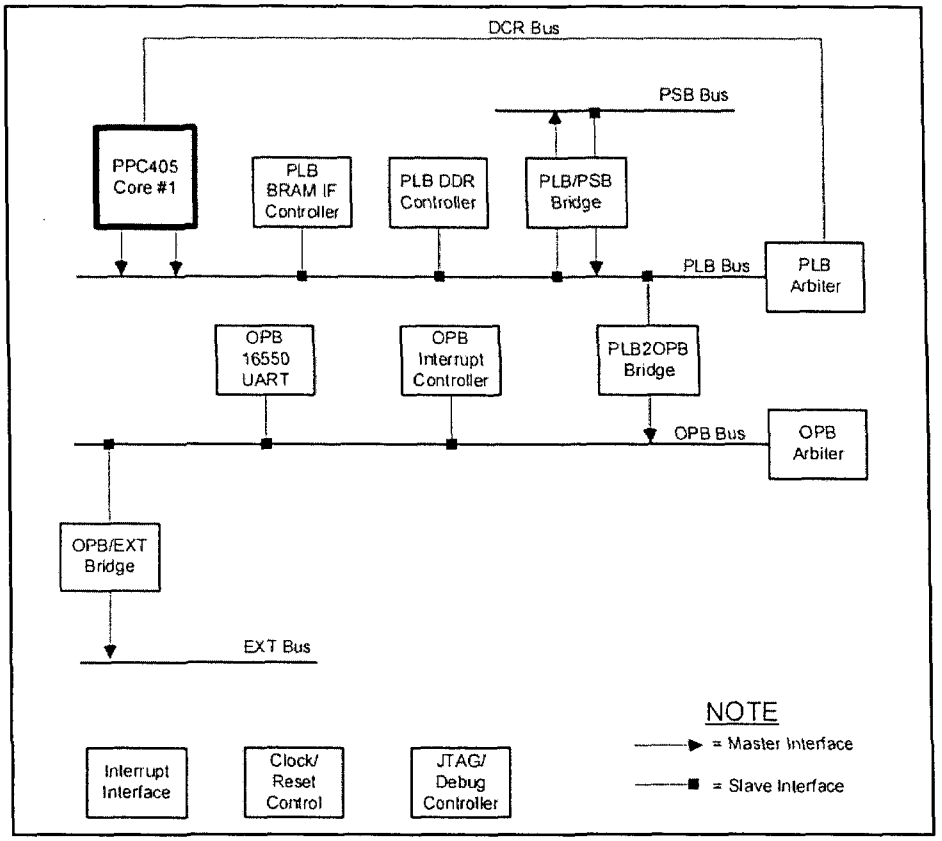


figure 38 - Configuration des bus à l'intérieur du FPGA (AP1000)

Pour accélérer la génération de logique, les options du *Mapper* et du *Place and Route* sont configurés à la valeur STD (standard) dans le fichier d'options d'implémentation (etc/fast_runtime.opt) qui se trouve dans *Project Files* (figure 39). Si l'usager désire un niveau d'effort plus agressif, ces options doivent être changées pour la valeur HIGH, ce qui demande beaucoup plus de temps. Cette attente supplémentaire se chiffre en heures selon les expériences conduites dans le cadre de cette recherche.

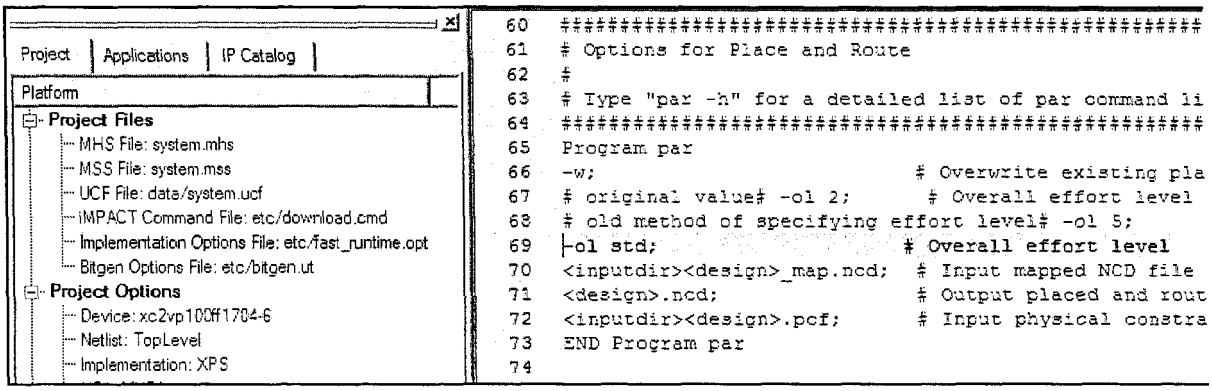


figure 39 - fichier d'options d'implémentation (etc/fast_runtime.opt)

6.2.2 Software / User Logic interaction

La logique est programmée de manière à lire les opérandes à une adresse sur le bus (addr_in). Cette donnée est ensuite traitée par une opération arbitraire et le résultat peut être renvoyé sur le bus à une adresse de sortie (addr_out). Les deux adresses doivent être fournies, ainsi que le nombre d'opérandes (op_cnt) à la logique par le logiciel utilisateur avant de commencer le traitement des données. La communication s'effectue selon les étapes suivantes :

1. Le logiciel écrit addr_in dans le registre slv_reg1, addr_out dans le registre slv_reg2 et le op_cnt dans slv_reg3
2. Le logiciel dit à la logique de démarrer son traitement en changeant le bit slv_reg0(8) à 1 (slv_reg étant le CSR)
3. Le traitement se complète dans la logique et le bit 9 de C_BASEADDR+0x0 est mis à 1
4. Le logiciel détecte ce changement et envoie un ACK en rechangeant le bit slv_reg0(8) à 0
5. La logique se remet en attente et le bit 9 de C_BASEADDR+0x0 est remis à 0

Il s'agit du fonctionnement du produit livrable conçu pour CMC et qui sert à la plateforme de développement établie dans cette recherche. Dans le cas de l'exemple d'application du contrôle

vectoriel, il est mentionné dans la section 3.2 que les sorties de la carte AIO ne seront pas utilisées. Le logiciel n'est donc pas ici obligé de fournir une adresse de sortie addr_out.

Le contrôle vectoriel étant une opération continue, il n'est pas non plus nécessaire de spécifier un nombre d'opérandes à traiter, ni de confirmer le traitement matériel à l'aide du bit 9 de C_BASEADDR+0x0. Le départ et l'arrêt du contrôle peuvent être gérés soit par le logiciel ou par un autre signal envoyé au FPGA. La **seule condition importante** est que le logiciel communique à la logique que la carte AIO est bien initialisée sur le bus PCI.

0-7: Dipswitches	8: Start HW	9: HW done	10-31: Réservé

figure 40 – C_BASEADDR+0x0 : Registre de contrôle et statut (CSR)

6.2.3 Le modèle esclave de lecture des registres

La communication sur le bus PLB et le bus OPB du FPGA de la carte Amirix supporte une architecture CoreConnect d'IBM. Ainsi, les modes de communication *master* et *slave* sont disponibles pour n'importe quelle logique instanciée dans le *Baseline* de la carte Amirix (pourvu que la communication des bus internes soit respectée selon la figure 38).

Dans le projet livrable, 4 registres esclaves sont disponibles aux autres périphériques du bus, soit les adresses C_BASEADDR + 0x0, C_BASEADDR + 0x4, C_BASEADDR + 0x8 et C_BASEADDR + 0xC. L'application de filtre de démonstration du produit livrable ne fait pas usage des registres 2 et 3. Le registre 1 est utilisé pour renvoyer la valeur calculée par la logique au logiciel usager, alors que le registre 0 devient un registre de contrôle et statut (CSR, illustré à la figure 40). Il est composé des signaux suivants :

0-7 : L'état des *dipswitches* de la carte Amirix AP1000

8 : Signal du logiciel qui déclenche le traitement de la logique

9 : Signal de la logique qui indique au logiciel que le traitement est complété

10-31: réservé

La figure 41 illustre comment ces différents signaux sont gérés par le multiplexeur du système de registres esclaves.

```
-- Implement slave model register read MUX
SLAVE_REG_READ_PROC : process( slv_reg_read_select, slv_reg0, slv_reg1, slv_reg2, slv_reg3 ) is
begin

  case slv_reg_read_select is
    when "1000" => slv_ip2bus_data <= dip_switch & slv_reg0(8) & hw_work_done & slv_reg0(10 to 31);
    when "0100" => slv_ip2bus_data <= MATH_RES;
    when "0010" => slv_ip2bus_data <= slv_reg1;
    when "0001" => slv_ip2bus_data <= slv_reg2;
    when others => slv_ip2bus_data <= (others => '0');
  end case;

end process SLAVE_REG_READ_PROC;
```

figure 41 - MUX du modèle esclave des registres (hw_math)

6.2.4 La machine d'état

La machine d'état qui se trouve dans le produit livrable pour CMC se comporte de la manière suivante :

- Détection d'un signal de démarrage sur slv_reg0(8) (voir CSR)
- Lecture d'une adresse de lecture des données d'entrée (addr_in), d'une adresse de sortie (addr_out) et d'un nombre d'opérandes à traiter sur slv_reg1, slv_reg2 et slv_reg3, respectivement.
- Vérification du compte d'opérande et préparation de requêtes de lecture (choisir l'adresse où la valeur lue sera inscrite, comme C_BASEADDR + 0xC par exemple). S'il n'y a plus d'opérandes, retour vers un état d'attente.
- Requête de lecture sur le bus et attente du ACK.

- Traitement de la donnée, nombre d'opérandes décrémenté et incrémentation de l'adresse de lecture (si c'est le cas).
- Écriture du résultat à l'adresse de sortie (requête et attente du ACK).
- Retour à l'état de vérification du compte d'opérandes.

Ce livrable est présenté dans la méthode de développement de cette recherche, car tous les éléments de communication de base y sont présents. L'application du contrôle vectoriel requiert beaucoup moins de généralité, et supprime certaines étapes de communication qui pourraient être pratiques dans une autre application. Le fait d'offrir un code générique dans cette méthode de développement permet à l'usager de rapidement porter son application XSG vers la plateforme Amirix en enlevant ce qui n'est pas nécessaire dans le Baseline, plutôt qu'en rajoutant des fonctions manquantes (ce qui implique du temps supplémentaire pour tester ces fonctions).

Pour l'application du contrôle vectoriel, une adresse de sortie n'est plus nécessaire, car les délais d'activation sont envoyés au module de puissance via les sorties du FPGA et non de la carte AIO. Il n'y a pas non plus de compte d'opérandes, car le contrôle s'effectue en continu. La machine se comporte de la manière suivante :

- Détection d'un signal de démarrage (soit envoyé par logiciel ou par un signal directement lié au FPGA)
- Lecture d'une adresse de lecture des données d'entrée (addr_in), correspondant à la source de données de la carte AIO (selon le mode de configuration de cette dernière).
- Attente de signal de démarrage d'acquisition (provenant du processus de portillonnage).
- Requête de lecture sur le bus et attente du ACK.
- Traitement des données et résultats enregistrés pour le processus de portillonnage qui s'exécute en parallèle.
- Retour à l'état d'attente de démarrage d'acquisition.

6.2.5 Intégration de fichiers NGC

Le modèle XSG généré par l'usager (voir section 4) donne comme résultat un fichier NGC qui est en réalité une boîte noire comprenant le code VHDL du système. Il n'est pas facile de trouver la documentation sur l'intégration d'un fichier NGC au sein d'un projet XPS, et c'est pourquoi la méthode est reproduite dans cette section :

1. Dans le fichier MPD, changer le paramètre OPTION STYLE pour la valeur MIX comme dans l'exemple suivant :

```
OPTION IPTYPE = PERIPHERAL
OPTION IMP_NETLIST = TRUE
OPTION HDL = VHDL
OPTION IP_GROUP = MICROBLAZE:PPC:USER
OPTION CORE_STATE = DEVELOPMENT
OPTION STYLE = MIX
```

2. Créer un dossier « netlist » dans le répertoire « pcore » qui va maintenant contenir 4 répertoires (data, devl, hdl et netlist)
3. Inclure un fichier BBD dans le dossier « data » qui va contenir le texte suivant :

```
FILES
blackbox1.ngc, blackbox2.ngc, blackbox3.edn
```

Et non la syntaxe suivante :

```
FILES blackbox1.ngc, blackbox2.ngc, blackbox3.edn
```

4. Mettre tous les fichiers NGC dans le dossier « netlist »

6.2.6 Génération de fichier BIN et configuration sur Amirix

Une fois le Baseline adapté aux besoins spécifiques d'une application, le projet doit être synthétisé et un fichier de reconfiguration pour la carte Amirix est créé. Cependant, ce fichier ne peut être transmis à cette dernière via le port JTAG car cette procédure empêche l'accès au système Linux embarqué. Il est donc nécessaire d'utiliser la mémoire flash de configuration. La carte Amirix AP1000 contient 2 *Intel Strataflash Flash memory devices* (tableau 4); un pour la configuration et un pour l'exécution de code (contenant U-Boot et ses variables de configuration, qu'il ne faut pas écraser).

Bank	Adresse	Size	Taille	Mode	Description
1	0x20000000	0x1000000 (16MB)	16	16	Program Flash
2	0x24000000	0x1000000 (16MB)	16	8	Configuration Flash

tableau 4 – Les 2 Intel Strataflash Flash memory devices

Région	Bank	Secteurs	Description
0	2	0 – 39	Configuration 0
1	2	40 – 79	Configuration 1
2	2	80 – 127	Configuration 2 (AMIRIX Default Platform)

tableau 5 - Configurations sur carte Amirix AP1000 (Configuration Flash)

La mémoire de configuration (tableau 5) se divise en trois régions de configuration. Il est recommandé de ne pas écraser la région 2, car il s'agit de la configuration par défaut d'Amirix.

Pour charger en mémoire le fichier BIN créé avec l'outil XPS, il faut configurer un serveur TFTP sur le PC hôte et utiliser le terminal afin de se brancher sur la carte Amirix et utiliser les commandes U-Boot de transfert TFTP (le tout est automatisé à l'annexe C grâce à AutoHotKey). Une fois le fichier sur la carte, une commande de reconfiguration (selon une région choisie) reconfigure et redémarre la carte Amirix AP1000.

6.3 Logiciel et pilotes

L'accès au système Linux embarqué sur la carte Amirix permet d'utiliser un pilote logiciel afin d'initialiser la carte AIO sur le bus PCI. L'initialisation ne s'effectuant qu'une seule fois avant le départ du traitement de la logique, il n'y a pas de pénalité dans l'efficacité et le temps de réponse du contrôle vectoriel (ou d'une autre application).

6.3.1 Opérations de préparation logicielle

Les pilotes et logiciels de l'usager sont copiés sur une carte *compactflash* qui est ensuite montée comme système de fichier sur l'environnement Linux embarqué. Les pilotes sont ensuite chargés avec les paramètres de la logique, soit l'adresse de base (ex. : 0x2A000000) et la taille de l'espace d'adressage.

Comme il s'agit d'une routine de commandes qui est souvent employée lors de tests en laboratoire, il est avantageux d'utiliser une solution automatisée. Le logiciel gratuit AutoHotKey (AHK) permet d'automatiquement entrer les différentes commandes nécessaires au terminal à la simple pression d'une combinaison de touches du clavier. Le script AHK se trouve à l'annexe C. Une fois la configuration effectuée, le logiciel usager peut être lancé afin de configurer la carte AIO et de signaler à la logique que son traitement peut être amorcé.

ANNEXE C

Code Embedded du portillonnage

```
function [out_pu, out_pv, out_pw, start] = Emfiregate(c_value,count, in_fu,
in_fv, in_fw)

    Teq = 100;
    Vdc = 400;

    % Outputs COMPLETE PULSE
    simout_pulse_u = 0;
    simout_pulse_v = 0;
    simout_pulse_w = 0;

    count_value = c_value;
    section_steps   = count_value+1;

    section_time    = Teq;

    count_ori = count;

    cur_zone = 1;
    % Zone picker
    if count > section_steps
        count = count - section_steps;
        cur_zone = 2;
    end

    start = 0;
    % Precision avec deux controles
    if count == section_steps-2
        start = 1;
    end

    % Precision avec un seul controle
    % if count_ori == (section_steps*2)-5
    %     start = 1;
    % end

                % Partial Outputs
                out_pu = 0;
                out_pv = 0;
                out_pw = 0;

    %            for i = 1:angles_size
                    % Restart Counter
                    %count = 1;

                    % Conversion des inputs
                    s_fu = in_fu/section_time * section_steps;
                    s_fv = in_fv/section_time * section_steps;
                    s_fw = in_fw/section_time * section_steps;

                    %for j = 1:section_steps
                        % ZONE 1
                        if cur_zone == 1

                            % Pour pulse fu
                            if count < s_fu
                                out_pu = 0;
                            else
```

```
            out_pu = 1;
        end

        % Pour pulse fv
        if  count < s_fv
            out_pv = 0;
        else
            out_pv = 1;
        end

        % Pour pulse fw
        if  count < s_fw
            out_pw = 0;
        else
            out_pw = 1;
        end

    % ZONE 2

    elseif cur_zone == 2


        % Pour pulse fu
        if count < (section_steps - s_fu)
            out_pu = 1;
        else
            out_pu = 0;
        end

        % Pour pulse fv
        if count < (section_steps - s_fv)
            out_pv = 1;
        else
            out_pv = 0;
        end

        % Pour pulse fw
        if count < (section_steps - s_fw)
            out_pw = 1;
        else
            out_pw = 0;
        end

    end

    % Increase count
    %count = count + 1;

% Accumulation des pulses
simout_pulse_u = out_pu;
simout_pulse_v = out_pv;
simout_pulse_w = out_pw;
```

ANNEXE D

Script Autohotkey pour Amirix

```
#4::
IfWinActive ap1000 - HyperTerminal
    Send insmod /mnt/hwlogic.o{Enter}
    Sleep 200
    Send insmod /mnt/aio.o{Enter}
    Sleep 200
    Send cd /dev{Enter}
    Send mknod hwlogic c 254 0{Enter}
    Sleep 200
    Send mknod aio c 253 0{Enter}
    Sleep 200
    Send /mnt/initaio

return

#3::
IfWinActive ap1000 - HyperTerminal
    Send insmod /mnt/hwlogic.o{Enter}
    Sleep 200
    Send insmod /mnt/gsc16aiss8a04.o 0{Enter}
    Sleep 200
    Send cd /dev{Enter}
    Send mknod hwlogic c 254 0{Enter}
    Sleep 200
    Send mknod gsc16aiss8a040 c 253 0{Enter}
    Sleep 200
    Send /mnt/testapp
    ;0x01000000 0x02000000 5
return

#,::
IfWinActive ap1000 - HyperTerminal
    Send rmmod hwlogic{Enter}
    Sleep 200
    Send rmmod gsc16aiss8a04{Enter}
    Sleep 200
    Send umount /mnt{Enter}
return


#2::
IfWinActive ap1000 - HyperTerminal
    Send insmod /mnt/logic2/logic.o base_addr=0x2A001000{Enter}
    Sleep 200
    Send insmod /mnt/aio.o{Enter}

    Sleep 200
```

```
    Send cd /dev{Enter}
    Send mknod logic c 254 0{Enter}
    Sleep 200
    Send mknod aio c 253 0{Enter}

    Sleep 200
    Send /mnt/init_aio{Enter}
return

#m::
IfWinActive ap1000 - HyperTerminal
    Send mount /dev/discs/disc0/part1 /mnt{Enter}
    Sleep 200
return

#1::
IfWinActive ap1000 - HyperTerminal

    Send setenv serverip 132.212.202.166{Enter}
    Sleep 200
    Send setenv ipaddr 132.212.201.223{Enter}
    Sleep 200
    Send erase 2:0-39{Enter}

;-----------------------------------------------------
    MsgBox, 4, , Tftp download.bin?
    IfMsgBox, No
            return
    WinActivate ap1000 - HyperTerminal
;-----------------------------------------------------

    Send tftp 00100000 download.bin{Enter}

;-----------------------------------------------------
    MsgBox, 4, , Copy to config flash 0?
    IfMsgBox, No
            return
    WinActivate ap1000 - HyperTerminal
;-----------------------------------------------------

    Send cp.b 00100000 24000000 00500000{Enter}

;-----------------------------------------------------
    MsgBox, 4, , Load config 0?
    IfMsgBox, No
            return
    WinActivate ap1000 - HyperTerminal
```

```
;-----------------------------------------------

    Send swrecon{Enter}

return
```

ANNEXE E

Blocs individuels du contrôle vectoriel XSG

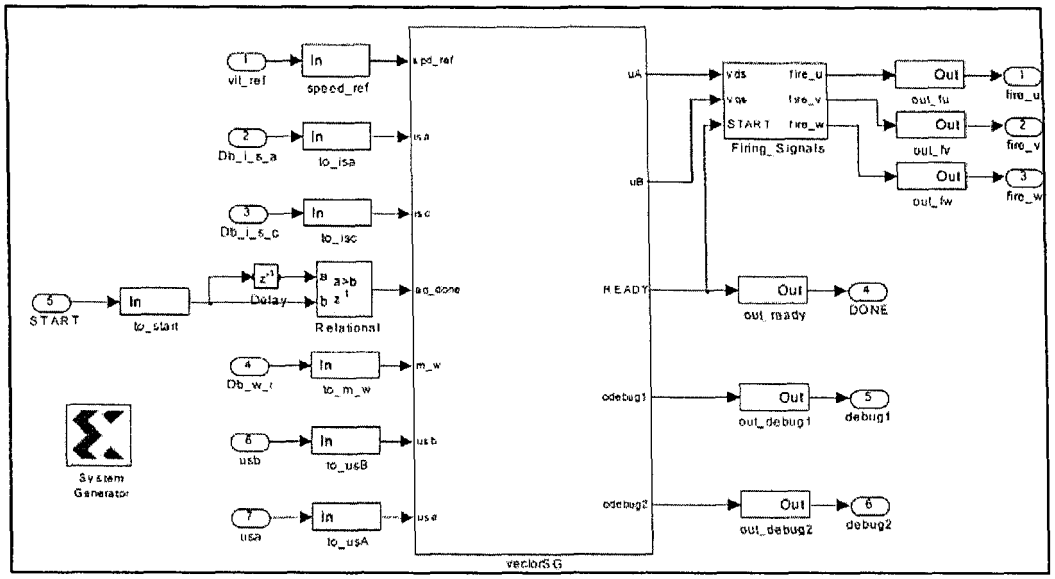


figure 42 – Contrôle vectoriel, vue extérieure

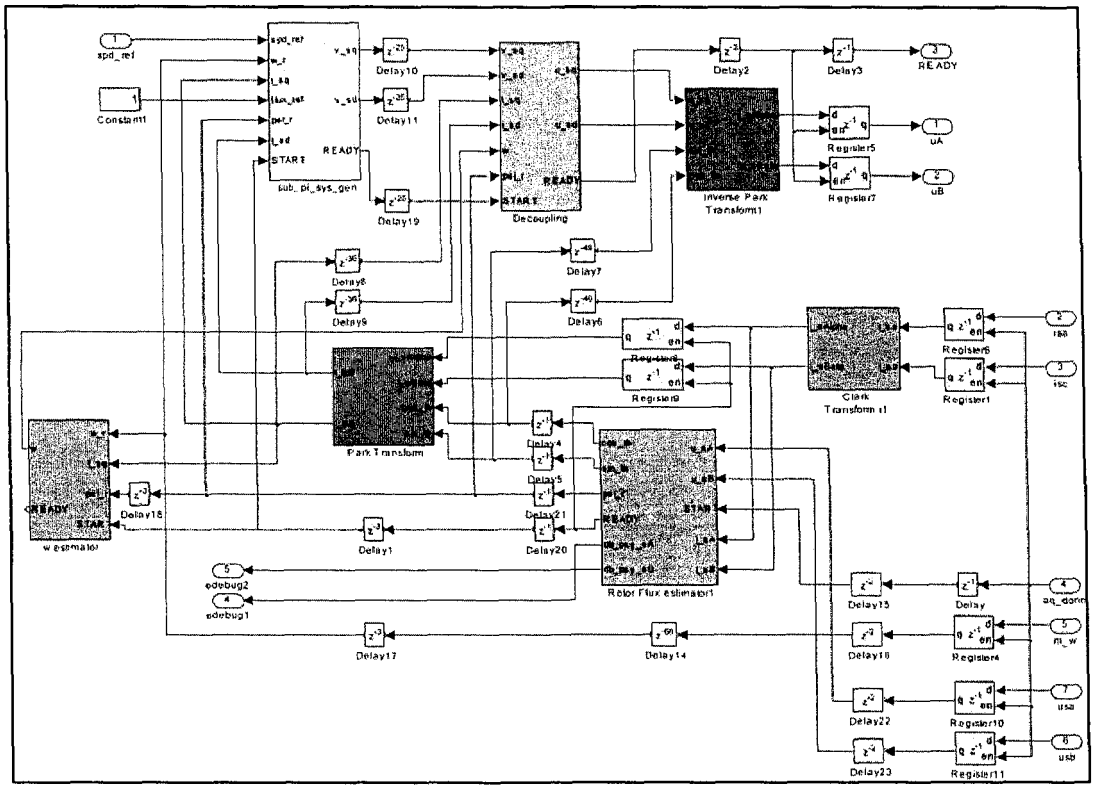


figure 43 – Contrôle vectoriel, vue intérieure

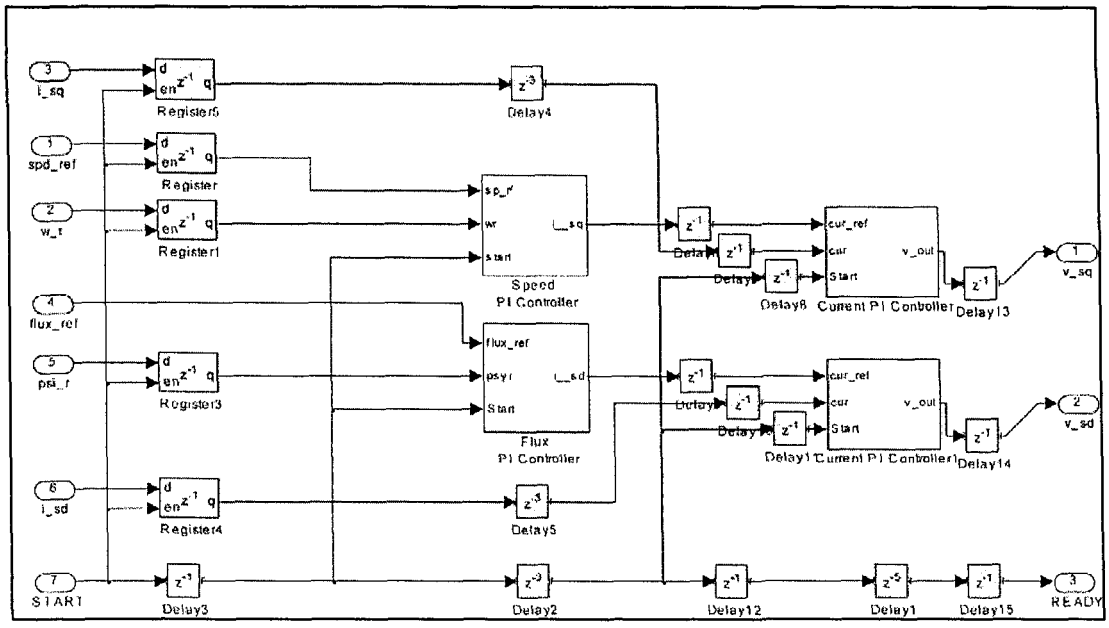


figure 44 – PI, vue d'ensemble

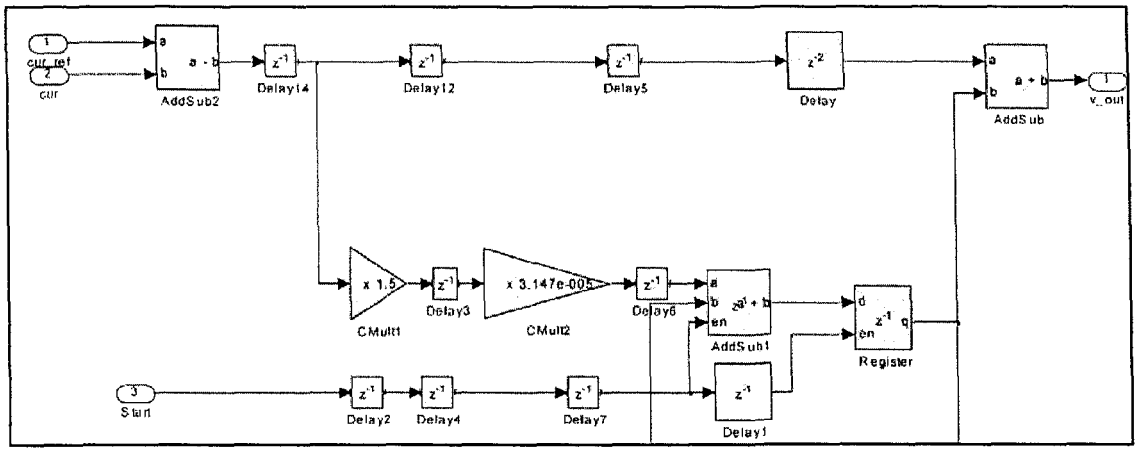


figure 45 – PI, vue intérieure

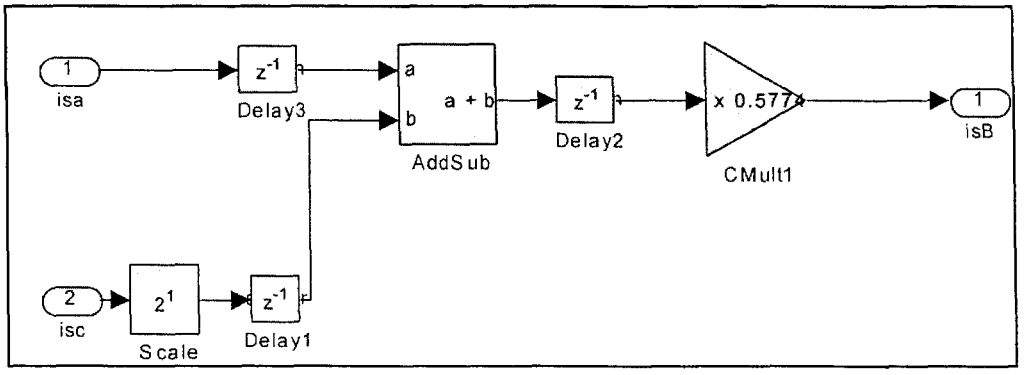


figure 46 – Transformée de Clark

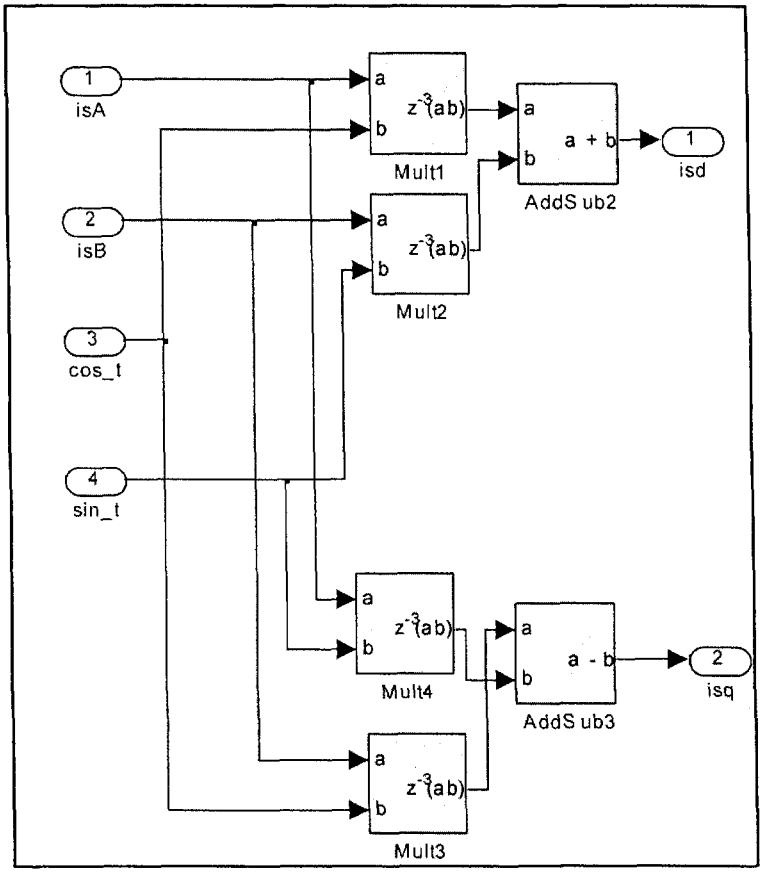


figure 47 – Transformée de Park

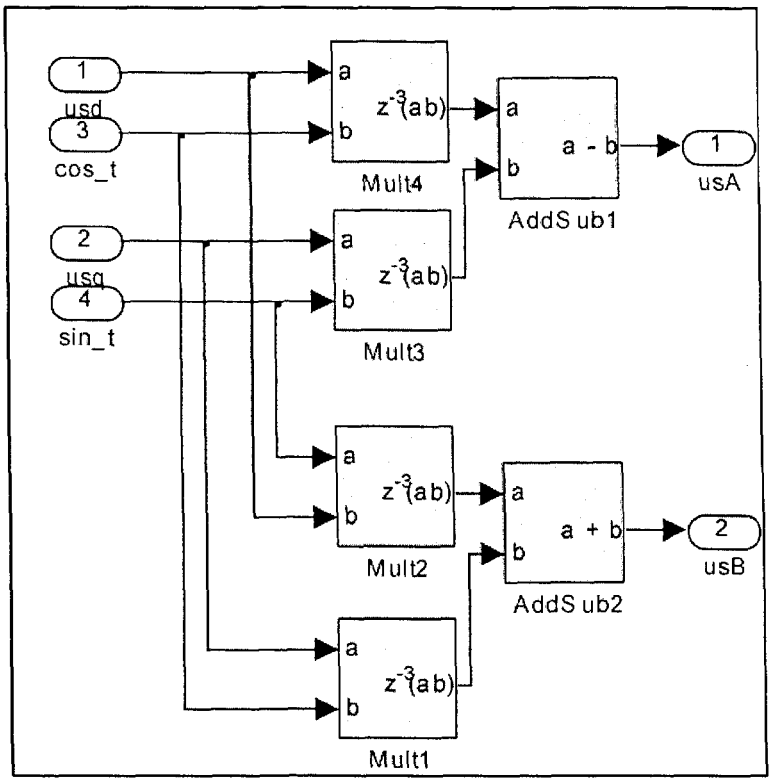


figure 48 – Transformée inverse de Park

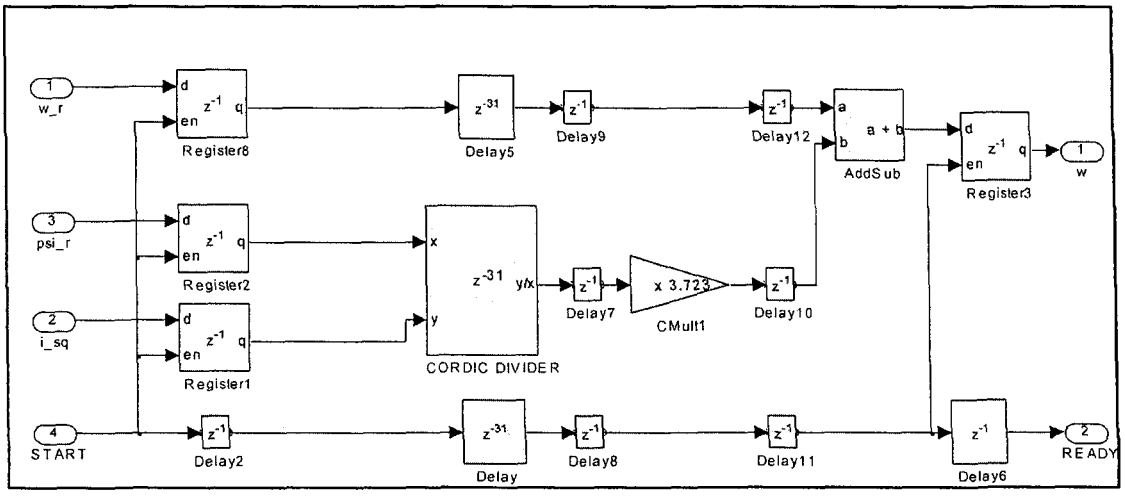


figure 49 – Estimateur de ω

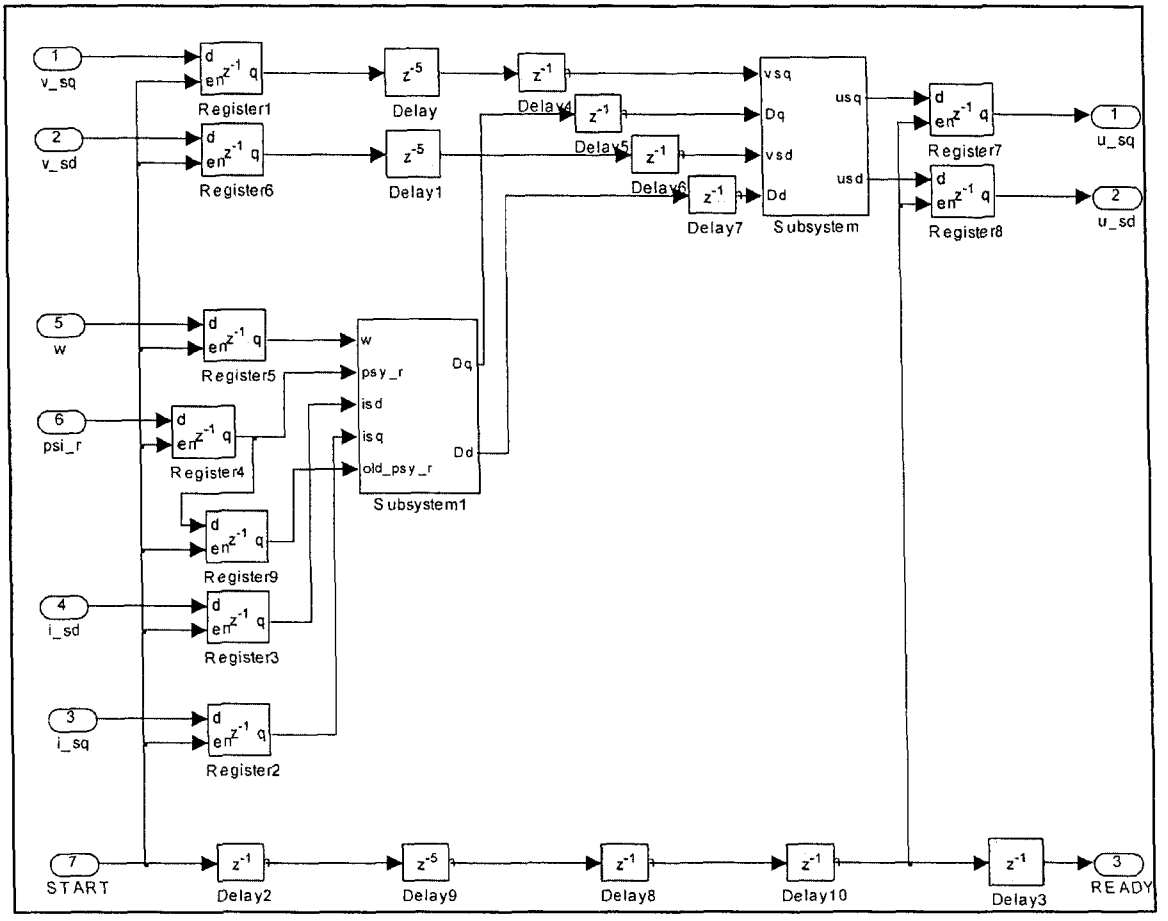


figure 50 – Bloc decoupling

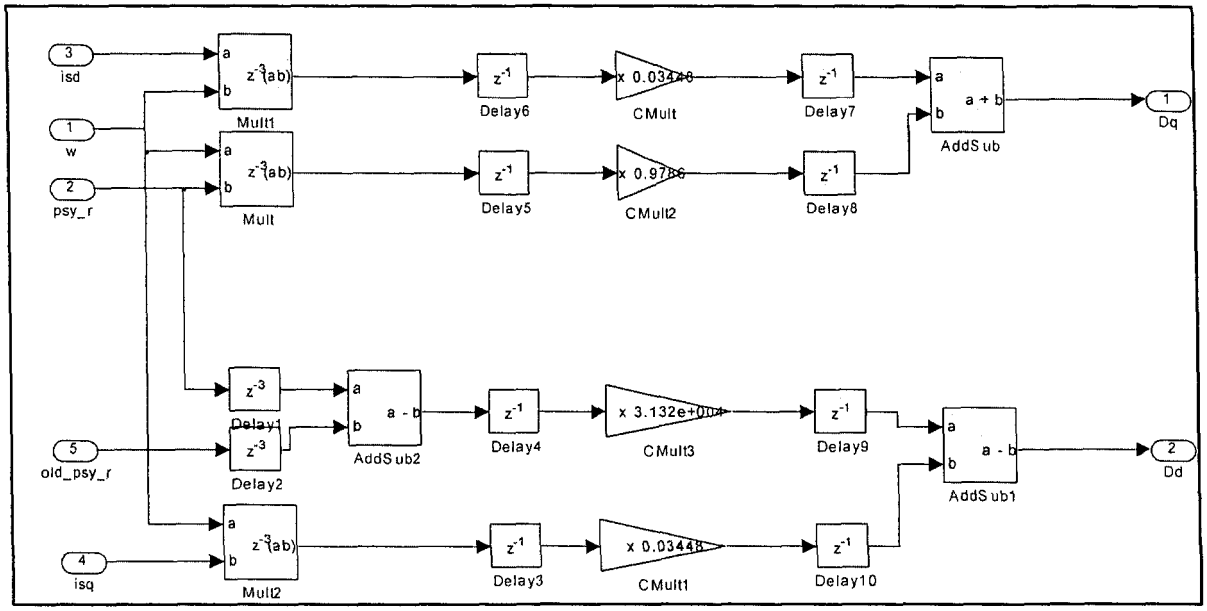


figure 51 – Calcul de Dq et Dd

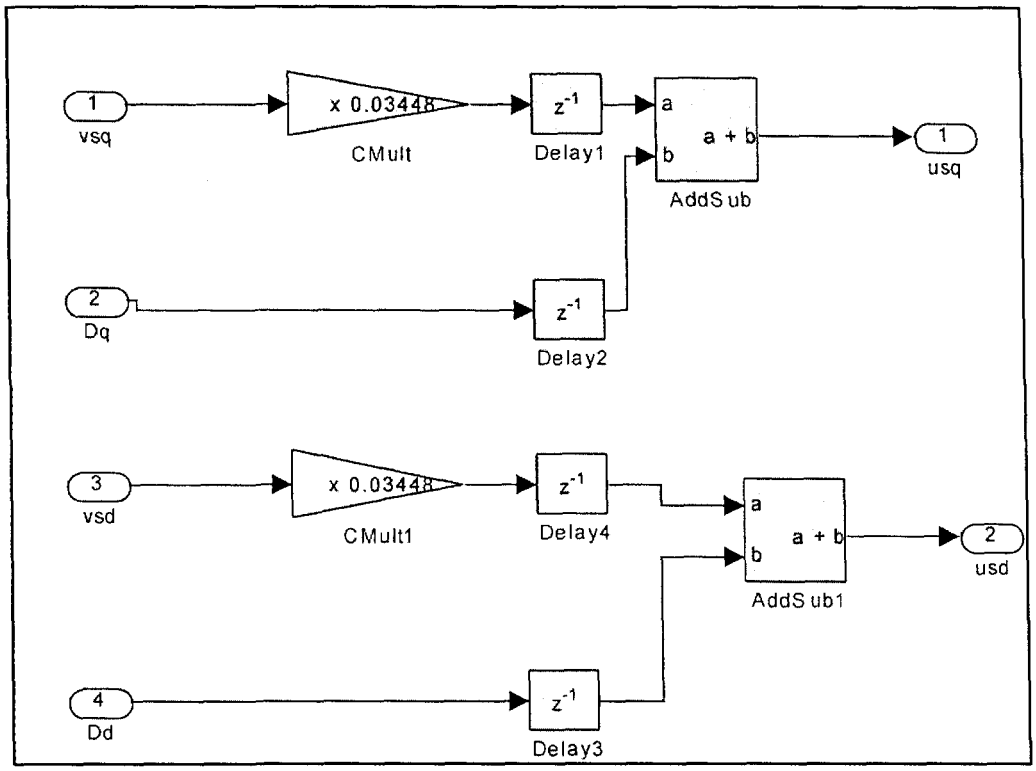


figure 52 – Calcul de usq et usd

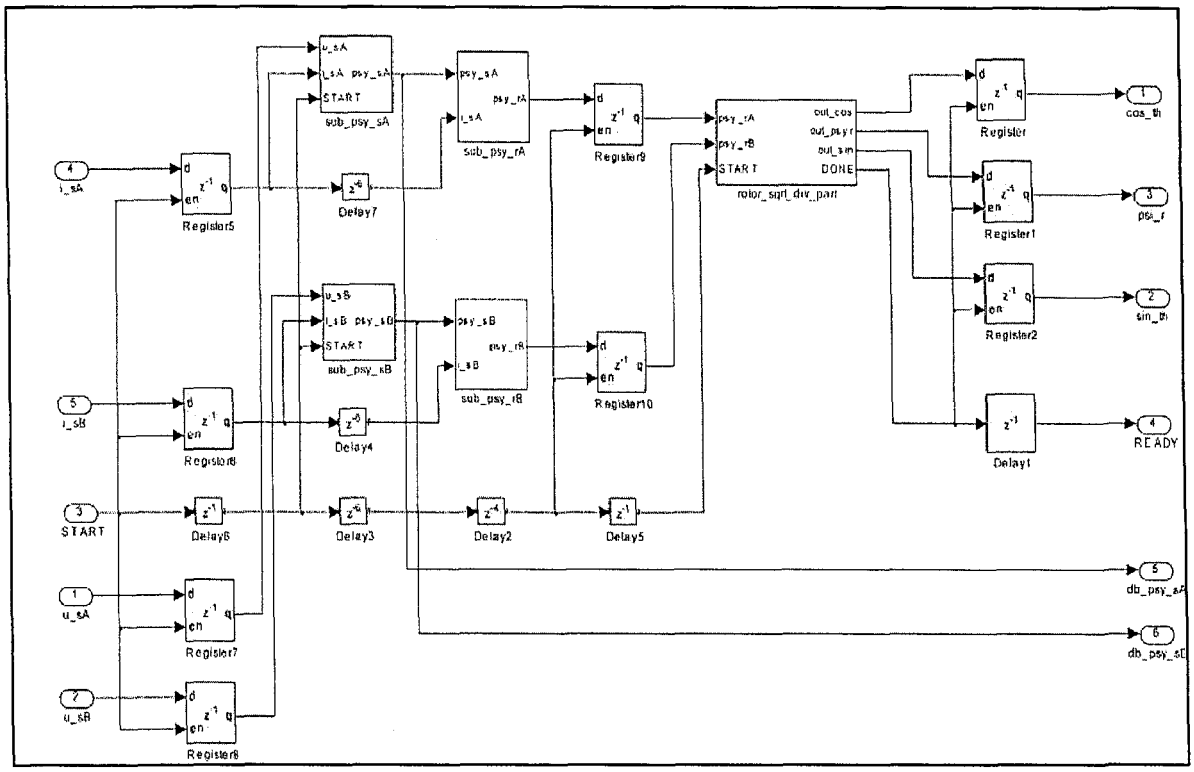


figure 53 – Estimateur de flux rotoriques

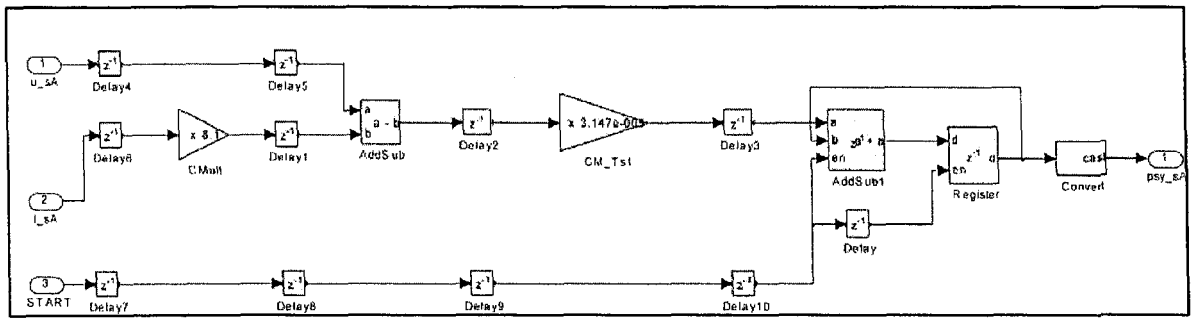


figure 54 – Calcul de flux statoriques

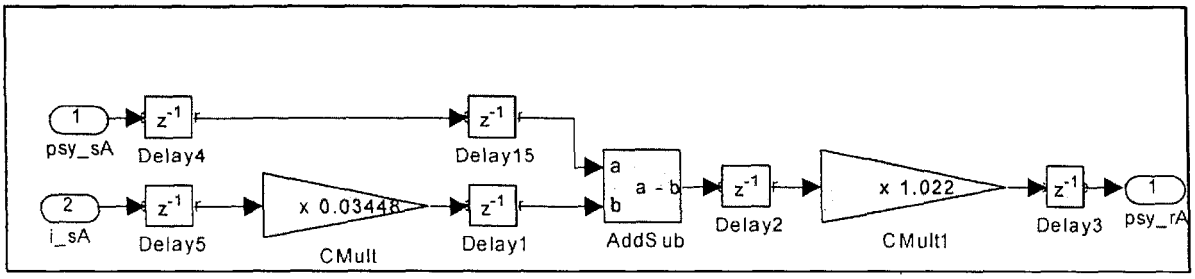


figure 55 – Calcul de flux rotoriques

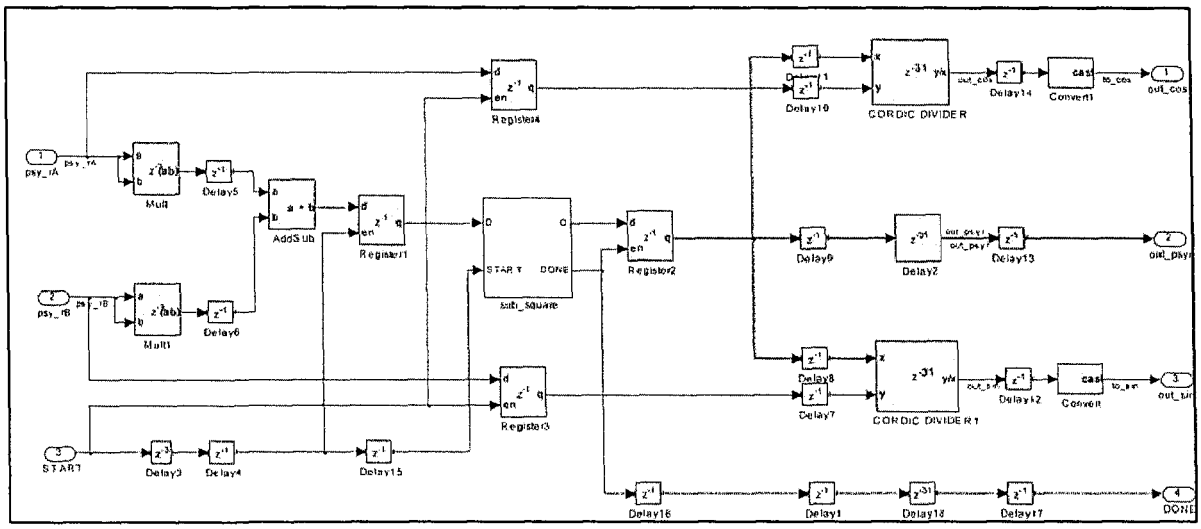


figure 56 – Racine carrée non restaurante dans l'estimateur de flux rotoriques

ANNEXE F

Schémas de racine carrée et division

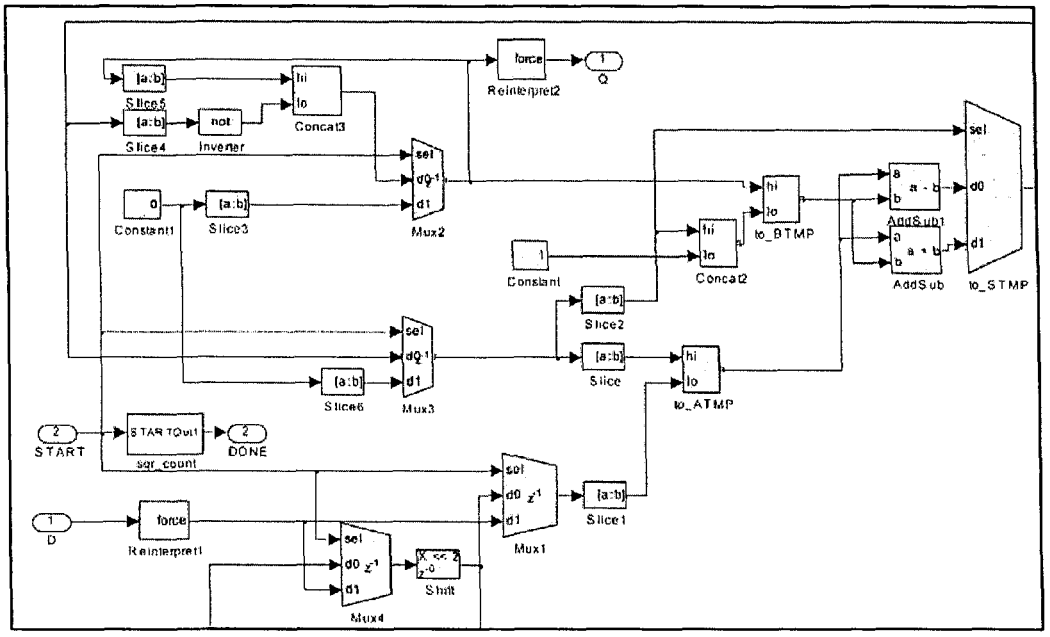


figure 57 – Racine carrée non restaurante

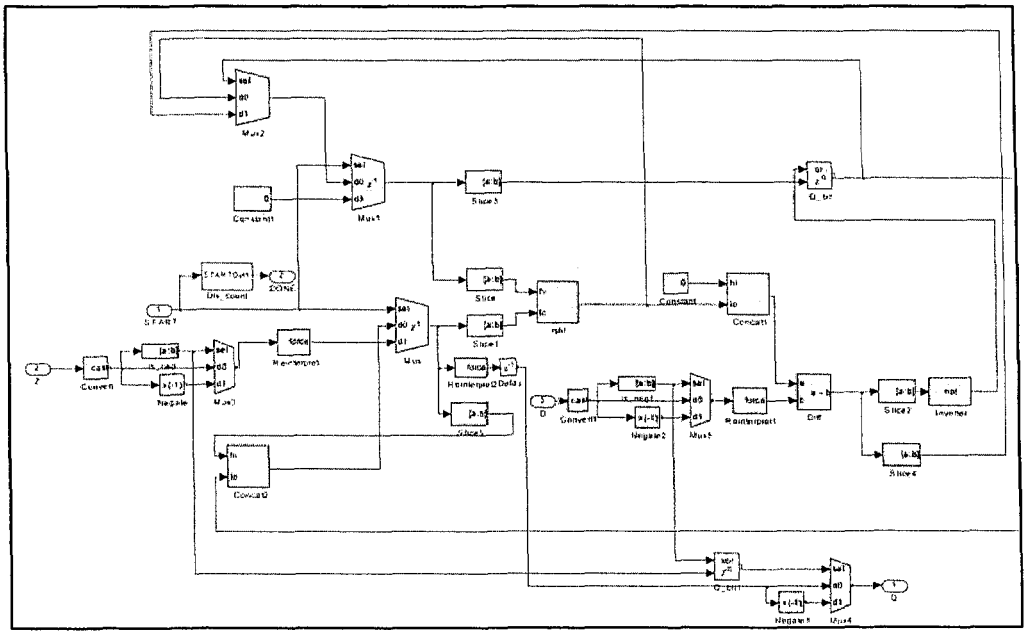


figure 58 – Division non restaurante

ANNEXE G

Scripts GAPPA

Optimisation de Clark

```
    # Operateurs

    @clark_sub_isb_Cmult1 = fixed<-14,dn>;   # Binary point de la multiplication
    @clark_sub_isb_AddSub = fixed<-14,dn>;   # Binary point de l'addition

    # Constantes et entrees

    @isa_bp = fixed<-14,dn>;
    @isb_bp = fixed<-14,dn>;
    @clark_sub_isb_s3_cte = fixed<-14,dn>;

    # Equations Theoriques

    s3    = 0.57735026918963;
    ISBETA = (isa + (2*isb)) * s3;

    # Conversion des entrees et cte

    gisa  = isa_bp(isa);
    gisb  = isb_bp(isb);
    gs3   = clark_sub_isb_s3_cte(s3);

    # Equation Pratiques

    add_res clark_sub_isb_AddSub   = gisa + (2*gisb);
    clark_res clark_sub_isb_Cmult1 = add_res*gs3;

    # Analyse

    {    isa in [-100,100] /\
       isb in [-100,100]
          ->
          ISBETA in ? /\
```

```
    clark_res in ? /\
    add_res in ? /\
    clark_res - ISBETA in ?
}
```

Optimisation de current_pi

```
@curpi_mulb = fixed<-16,dn>;  #Binary point des mult
@curpi_mulb1 = fixed<-20,dn>;  #Binary point des mult

# Theorique

kp_vit = 1;
Ts = 0.00005;
ki_vit = 1.5;
ISQ = (sp_ref*kp_vit) + ((sp_ref*Ts)+(sp_ref*Ts))*ki_vit;

# Pratique

a curpi_mulb= sp_ref*kp_vit;
b curpi_mulb1= sp_ref*Ts;
c curpi_mulb1= b+b;
d curpi_mulb1= c * ki_vit;
gISQ curpi_mulb1 = a+d;

{   sp_ref in [-170000,170000]
    ->
    a in ? /\
    b  in ? /\
    c in ? /\
    d in ? /\
    ISQ in ? /\
    gISQ in ? /\
    gISQ - ISQ in ?
}
```

Optimisation de decoupling 1

```
@decoup1_mul1 = fixed<-20,dn>; #Binary point des mult
@decoup1_add1 = fixed<-18,dn>; #Binary point des mult

# Theorique

LsSigma = 0.0037540626371672;
Lm_Lr = 0.88167962483031;
Lm = 0.015160569362457;
#-(Rr/Lr)*(Lm/Lr)
compl = -15.6337629210242;
mLsSigma = -0.0037540626371672;

Dq = ((isd*w)*LsSigma) + ((w*psy_r)*Lm_Lr);
Dd = (((isd*Lm)-psy_r)*compl) + ((w*isq)*mLsSigma);

# Pratique

a decoup1_mul1= isd*w;
b decoup1_mul1= a*LsSigma;
c decoup1_mul1= w*psy_r;
d decoup1_mul1= c*Lm_Lr;

gDq decoup1_add1= b + d;

e decoup1_mul1= isd*Lm;
f decoup1_add1= e-psy_r;
g decoup1_mul1= f*compl;
h decoup1_mul1= w*isq;
i decoup1_mul1= h*mLsSigma;

gDd decoup1_add1= g+i;
```

```
{   w in [-400,400] /\
    psy_r in [-4,4] /\
    isd in [-400,400] /\
    isq in [-400,400]
    ->
#   a in ? /\
#   b in ? /\
#   c in ? /\
#   e in ? /\
#   f in ? /\
    Dq in ? /\
    Dd in ? /\
    gDq - Dq in ? /\
    gDd - Dd in ?
}
```

Optimisation de decoupling 2

```
@decoup2_mul1 = fixed<-20,dn>; #Binary point des mult
@decoup2_add1 = fixed<-18,dn>; #Binary point des mult

# Theorique

LsSigma = 0.0037540626371672;

usq = (vsq*LsSigma) + Dq;
usd = (vsd*LsSigma) + Dd;

# Pratique

a decoup2_mul1= vsq*LsSigma;
b decoup2_mul1= vsd*LsSigma;

gusq decoup2_add1= a + Dq;
gusd decoup2_add1= b + Dd;
```

```
{   Dq in [-3000,3000] /\
    Dd in [-3000,3000] /\
    vsd in [-400,400] /\
    vsq in [-400,400]
    ->

    usq in ? /\
    usd in ? /\
    gusq - usq in ? /\
    gusd - usd in ?
}
```

Optimisation de Inverse de Park

```
@invpark_mulb = fixed<-16,dn>;  #Binary point des mult
@invpark_addb = fixed<-14,dn>;  #Binary point des add

# Theorique

USA = (usd*cost) - (usq*sint);
USB = (usd*sint) + (usq*cost);

# Pratique

a invpark_mulb= (usd*cost);
b invpark_mulb= (usq*sint);
c invpark_mulb= (usd*sint);
d invpark_mulb= (usq*cost);

gusa invpark_addb= a-b;
gusb invpark_addb= c+d;

{   usd in [0,400] /\
   usq in [0,400] /\
```

```
    cost in [-1,1] /\
    sint in [-1,1]
      ->
      a in ? /\
      b in ? /\
      c in ? /\
      d in ? /\
      USA in ? /\
      gusa - USA in ? /\
      USB in ? /\
      gusb - USB in ?
  }
```

Optimisation de Park Transform

```
  @park_in_isa = fixed<-8,dn>;
  @park_in_isb = fixed<-10,dn>;
  @park_in_cos = fixed<-14,dn>;
  @park_in_sin = fixed<-14,dn>;

  @park_mulb1 = fixed<-10,dn>;
  @park_mulb2 = fixed<-11,dn>;
  @park_mulb3 = fixed<-11,dn>;
  @park_mulb4 = fixed<-10,dn>;

  @park_addb = fixed<-11,dn>; #
  @park_subb = fixed<-11,dn>;

  # Theorique

  ISD = (isa*cost) + (isb*sint);
  ISQ = (isb*cost) - (isa*sint);

  # Pratique
```

```
a park_mulb1= (isa*cost);
b park_mulb2= (isb*sint);
c park_mulb3= (isb*cost);
d park_mulb4= (isa*sint);

gisd park_addb= a+b;
gisq park_subb= c-d;

{   isa in [-100,100] /\
   isb in [-174,+174] /\
   cost in [-1,1] /\
   sint in [-1,1]
     ->
     a in ? /\
     b in ? /\
     c in ? /\
     d in ? /\
     ISD in ? /\
     gisd - ISD in ? /\
     ISQ in ? /\
     gisq - ISQ in ?
}
```

Optimisation de psy_sB

```
@psysb_cmul = fixed<-10,dn>;
@psysb_sub = fixed<-10,dn>;
@psysb_imul = fixed<-10,dn>;

# Theorique
Rs = 0.3014;
Ts = 0.0001;

psysa = usA - (Rs*isA);
```

```
ia = Ts*psysa;

# Pratique
cmulres psysb_cmul= (Rs*isA);
gpsysa psysb_sub= usA - cmulres;
gia psysb_imul= Ts*gpsysa;

{   usA in [-500,500] /\
    isA in [-174,174]
    ->
    cmulres in ? /\
    gpsysa in ? /\
    gia in ? /\
    gpsysa - psysa in ? /\
    gia - ia in ?
}
```

Optimisation de psy_SA

```
@psysa_cmul = fixed<-10,dn>; #Binary point des mult
@psysa_sub = fixed<-10,dn>; #Binary point des mult
@psysa_imul = fixed<-10,dn>; #Binary point des mult

# Theorique
Rs = 0.3014;
Ts = 0.0001;

psysa = usA - (Rs*isA);
ia = Ts*psysa;

# Pratique

cmulres psysa_cmul= (Rs*isA);
gpsysa psysa_sub  = usA - cmulres;
```

```
gia imul= Ts*gpsysa;

{   usA in [-500,500] /\
    isA in [-100,100]
    ->
    cmulres in ? /\
    gpsysa in ? /\
    gia in ? /\
    gpsysa - psysa in ? /\
    gia - ia in ?
}
```

Optimisation de psy_RB

```
@psyrb_cmul  = fixed<-11,dn>;
@psyrb_sub   = fixed<-11,dn>;
@psyrb_cmul2 = fixed<-15,dn>;


# Theorique
sigmaLs = 0.0037540626371672;
LrM = 1.13419883122791;

addres = psy_sA-(isA*sigmaLs);
psyrA = addres*LrM;

# Pratique

gmulres psyrb_cmul= (isA*sigmaLs);
gaddres psyrb_sub = psy_sA-gmulres;
gpsyrA psyrb_cmul2= gaddres*LrM;

{   psy_sA in [-552,552] /\
    isA in [-174,174]
    ->
```

```
    gmulres in ? /\
    gaddres in ? /\
    gpsyrA in ? /\
    gpsyrA - psyrA in ? /\
    gaddres - addres in ?
}
```

Optimisation de psy_RA

```
@psyra_cmul = fixed<-11,dn>; #Binary point des mult
@psyra_sub = fixed<-11,dn>; #Binary point des mult
@psyra_cmul2 = fixed<-15,dn>; #Binary point des mult

# Theorique
sigmaLs = 0.0037540626371672;
LrM = 1.13419883122791;

addres = psy_sA-(isA*sigmaLs);
psyrA = addres*LrM;

# Pratique

gmulres psyra_cmul= (isA*sigmaLs);
gaddres psyra_sub= psy_sA-gmulres;
gpsyrA psyra_cmul2= gaddres*LrM;

{   psy_sA in [-552,552] /\
    isA in [-174,174]
    ->
    gmulres in ? /\
    gaddres in ? /\
    gpsyrA in ? /\
    gpsyrA - psyrA in ? /\
    gaddres - addres in ?
}
```

Optimisation de fluxpi

```
@fluxpi_mulb = fixed<-14,dn>;
@fluxpi_mulb1 = fixed<-29,dn>;
@fluxpi_sp = fixed<-14,dn>;

# Theorique

kp_vit = 10000;
Ts = 0.00005;
ki_vit = 20500;

ISQ = (sp(sp_ref)*kp_vit) + ((sp(sp_ref)*Ts)+(sp(sp_ref)*Ts))*ki_vit;

# Pratique

a fluxpi_mulb= sp(sp_ref)*kp_vit;
b fluxpi_mulb1= sp(sp_ref)*Ts;
c fluxpi_mulb1= b+b;
d fluxpi_mulb1= c * ki_vit;
gISQ fluxpi_mulb1 = a+d;

{   sp_ref in [-4,4]
    ->
    a in ? /\
    b  in ? /\
    c in ? /\
    d in ? /\
    ISQ in ? /\
    gISQ in ? /\
    gISQ - ISQ in ?
}
```

Optimisation de w_est

```
@west_in_wr = fixed<-6,dn>;
@west_in_psir = fixed<-14,dn>;
@west_in_isq = fixed<-11,dn>;
@west_cmul = fixed<-10,dn>;
@west_add = fixed<-10,dn>;

# Theorique
LmRrLr = 0.268824117610761;

# Pratique

w = (2*w_r)+ (i_sur_p*LmRrLr);

gisq west_in_isq= i_sq;
gpsir west_in_psir= psi_r;
gwr west_in_wr= w_r;
divres = i_sq/psi_r;
cmulres cmul= i_sur_p*LmRrLr;
addres add= (2*w_r) + cmulres;

{   w_r in [-400,400] /\
    i_sur_p in [-274000,274000]
    #psi_r in [0.001,1] /\
    #i_sq in [0.001,274]
    ->
    cmulres in ? /\
    addres in ? /\
    addres - w in ?
}
```

Optimisation de Speed_pi

```
@spi_mulb = fixed<-14,dn>;
```

```
@spi_mulb1 = fixed<-26,dn>;
@spi_sp = fixed<-14,dn>;  #Binary point de sp_ref input

# Theorique

kp_vit = 325;
Ts = 0.00005;
ki_vit = 3250;

ISQ = (sp(sp_ref)*kp_vit) + ((sp(sp_ref)*Ts)+(sp(sp_ref)*Ts))*ki_vit;

# Pratique

a spi_mulb= sp(sp_ref)*kp_vit;
b spi_mulb1= sp(sp_ref)*Ts;
c spi_mulb1= b+b;
d spi_mulb1= c * ki_vit;
gISQ spi_mulb1 = a+d;

{   sp_ref in [-500,500]
    ->
    a in ? /\
    b  in ? /\
    c in ? /\
    d in ? /\
    ISQ in ? /\
    gISQ in ? /\
    gISQ - ISQ in ?
}
```

Optimisation de rotor1

```
@rotor1_mul1 = fixed<-17,dn>;
@rotor1_mul2 = fixed<-19,dn>;
@rotor1_mul3 = fixed<-17,dn>;
```

```
@rotor1_mul4 = fixed<-17,dn>;

@rotor1_add1 = fixed<-16,dn>;
@rotor1_add2 = fixed<-16,dn>;
@rotor1_add3 = fixed<-16,dn>;

# Theorique

Rs = 0.3014;
Ts = 0.00005;
sigmaLs = 0.0037540626371672;
LrLm = 1.13419883122791;

psy_sA = ((u_sA - (i_sA*Rs)) * Ts) + ((u_sA - (i_sA*Rs)) * Ts);
psy_rA = (psy_sA - (i_sA*sigmaLs))*LrLm;

# Pratique

a rotor1_mul1= i_sA*Rs;
b rotor1_add1= u_sA - a;
c rotor1_mul2= b*Ts;
d rotor1_add2= c+c;

e rotor1_mul3= i_sA*sigmaLs;
f rotor1_add3= d-e;
g rotor1_mul4= f*LrLm;

{   u_sA in [-400,400] /\
    i_sA in [-400,400]
    ->
#               a in ? /\
#   b in ? /\
#   c in ? /\
#   e in ? /\
#   f in ? /\
    psy_rA in ? /\
```

```
    psy_sA in ? /\
    d - psy_sA in ? /\
    g - psy_rA in ?
}
```

ANNEXE H

Module de puissance et Moteur SPS

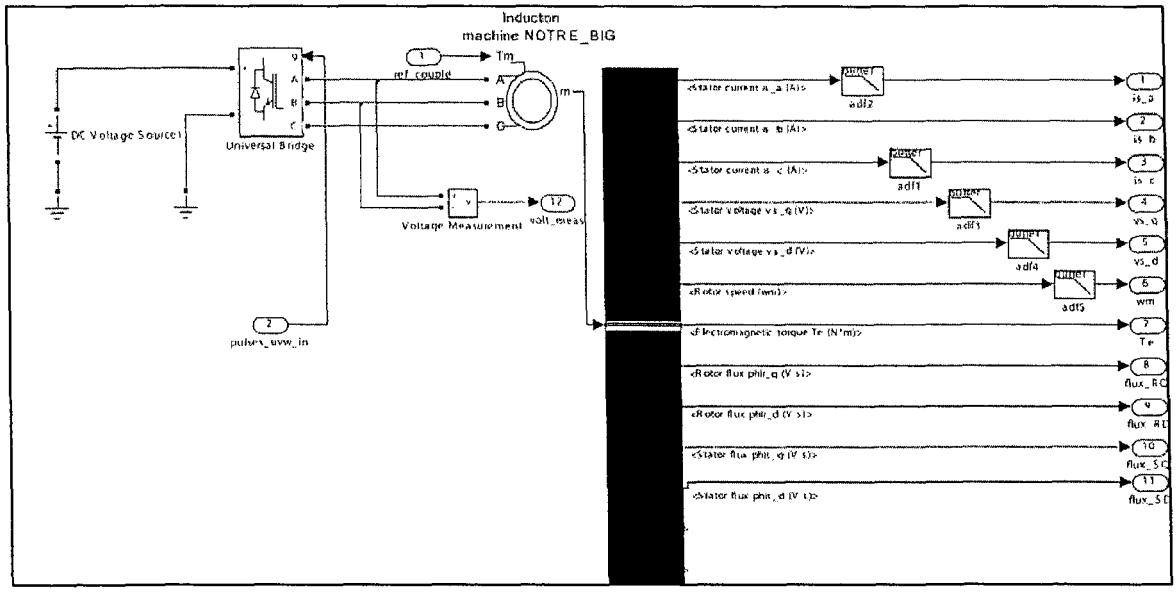


figure 59 – Bloc SPS

ANNEXE I

Articles acceptés

International Journal of Electrical and Power Engineering 1 (5): 524-529, 2007
ISSN: 1990-7958

Implementation of Division and Square Root Using XSG for FPGA-Based Vector Control Drives

Jean-Gabriel Mailloux, Stéphane Simard and Rachid Beguenane
Groupe ERMETIS, Département des Sciences appliquées, Université du Québec à Chicoutimi, Chicoutimi (QC), G7H 5B1, Canada

Abstract: The present study deals with hardware implementation of vector control for AC motors using XSG tool. Rotor flux estimation is done through a mathematical formula necessitating two divisions and one square root extraction. These are implemented using non restoring algorithms. FPGA implementations of such operators are considerably smaller compared to the division and the square root derived from XSG Core Library. More importantly, the corresponding modules exhibit a moderate operating frequency of more than 100MHz and present an infinite precision which matches with the idealized floating point Simulink model.

Key words: Vector control, rotor flux, non-restoring algorithm, division, square root, xilinx system generator

INTRODUCTION

To hardcode modern control laws of complex mechatronic systems, Hardware Description Languages (HDL), such as Verilog and VHDL, are not suitable for the designers who are unfamiliar with. A design and verification tool, such as Xilinx System Generator (XSG) for DSP design under Simulink is ideal to tackle the problems within the algorithm design stage by providing the designer an early proof-of-concept of the hard coded control algorithms. In fact XSG takes the graphical algorithmic approach and extend it to FPGA development by using dedicated Simulink Blockset (Xillinx, 2006). The knowledge of HDL, necessary to program FPGA devices, is then not required for DSP designers. This is because the XSG tool can automatically translate the algorithm into FPGA resource and loaded onto the FPGA device.

In the present study, XSG is used to prototype a complex control algorithm which is the vector control, a well known control strategy for AC drives. The use of XSG in electromechanical control systems has been subject of few papers. The authors of Ricci and Le-Huy (2003) have already modelled the widely used DTC algorithm, whereas study (Vasarhelyi *et al*., 2005) has presented the FPGA prototyping of a vector control system for a tandem converter fed induction motor. The divisions and square roots are often involved in many control schemes for electrical drives. For example, the vector control scheme contains a rotating rotor flux analyser. The computation of its modulus and its two axial components involve two divisions and one square root. To rapidly prototype the algorithm, the XSG based FPGA implementation of vector control is first done using IP cores for divisions and square root, leading to a huge logic utilization. Then modules for square root and division, based on non-restoring algorithms, are specifically designed for the estimation of rotor flux.

The hardware implementation of division and square root extraction, either for VLSI or FPGA, is hard given the complexity of the algorithms involved in their computation. This field has been the subject of many investigations in order to implement various algorithms, such as Newton Raphson, SRT-Redundant, Non Redundant, Restoring and Non-Restoring (Kabuo *et al*., 1994; Birman *et al*., 1990; Bannur and Verma, 1985; O'Leary, 1994; Harris *et al*., 1997; Sorkin, 2006; Deschamps, 2006). The Non-Restoring algorithms, for division and square root extraction, are considered as a better compromise in terms of complexity and precision (Li and Chu, 1997; Diromsopa *et al*., 2001; Marnane *et al*., 1997).

The present study is aiming to implement a fixed point division and square root modules using Xilinx System Generator (XSG) blocks. The designed modules are applied to the XSG-based vector control design replacing though the IP cores. A performance comparison, in terms of area and speed, between the two scenarios will be deeply discussed. In brief study reminds the

Corresponding Author: Rachid Beguenane, Groupe ERMETIS, Département des Sciences appliquées, Université du Québec à Chicoutimi, Chicoutimi (QC), G7H 5B1, Canada

theoretical background, respectively of vector control and non restoring algorithms for the division and square root operators, used for a fixed point FPGA implementation. The implementation results in the context of vector control for AC drives and a comparison in the case of using IP cores provided within XSG environment will be performed.

INDUCTION MOTOR VECTOR CONTROL SCHEME

The precise control of AC motors, such as the induction motor, needs an independent control of the stator current components that produce the required magnetizing flux and the electromechanical torque, in the same way as with the DC motor. This can be handled through sophisticated control algorithms, such as the vector control. This has several variants and rotor flux orientation control scheme, shown in Fig. 1, is one of them. This is derived from the electromechanical model of the induction motor in an adequate reference frame (d, q), known as Park domain. The mathematical description and the corresponding vector control algorithm can be found in many references (Beguenane *et al.*, 2006).

From Fig. 1 one can notice the vector control scheme presenting many blocks, each performing one particular task. For instance, the rotor flux estimator block evaluates the magnetizing flux using one of the exiting techniques such as Kalman Filter, MRAS, etc. The straightforward and less complicated method is the one derived from the electromechanical model of induction motor. In fact the modulus and the orientation of rotor flux can be computed as follows:

$$\Psi_r = \sqrt{\Psi_{r\alpha}^2 + \Psi_{r\beta}^2}$$

$$\cos\theta = \frac{\Psi_{r\alpha}}{\Psi_r};\ \sin\theta = \frac{\Psi_{r\beta}}{\Psi_r}$$

With

$$\Psi_{r\alpha} = \frac{L_r}{M}\left(\Psi_{s\alpha} - \sigma L_s i_{s\alpha}\right)$$

$$\Psi_{r\beta} = \frac{L_r}{M}\left(\Psi_{s\beta} - \sigma L_s i_{s\beta}\right)$$

$$\Psi_{s\alpha} = \int\left(u_{s\alpha} - R_s i_{s\alpha}\right)dt$$

$$\Psi_{s\beta} = \int\left(u_{s\beta} - R_s i_{s\beta}\right)dt$$

and

Ψ_r : Rotor flux modulus

$\Psi_{r\alpha}$, $\Psi_{r\beta}$: Components of rotor flux in (α, β) stationary frame

$\mu_{s\alpha}$, $\mu_{s\beta}$: Components of stator voltage in (α, β) stationary frame

$i_{s\alpha}$, $i_{s\beta}$: Components of stator current in (α, β) stationary frame

R_s, L_s : Stator resistance and inductance

M : Mutual inductance

σ : Leakage coefficient of the motor

Vector control scheme is fully implemented using XSG under Simulink environment as illustrated in Fig. 2. The computation of rotor flux necessitates two divisions and one square root. For a rapid validation of the algorithm, these operations have been implemented using a CORDIC IP cores provided within XSG interface. This leads to a massive logic in terms of LUTs and FFs. In order to reduce the consumption, the non-restoring division and square root algorithms have been coded in XSG.

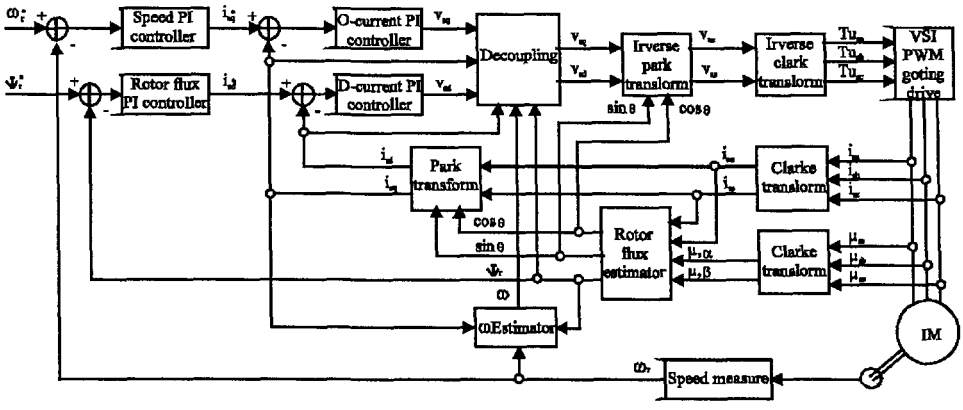


Fig. 1: Rotor flux oriented control scheme

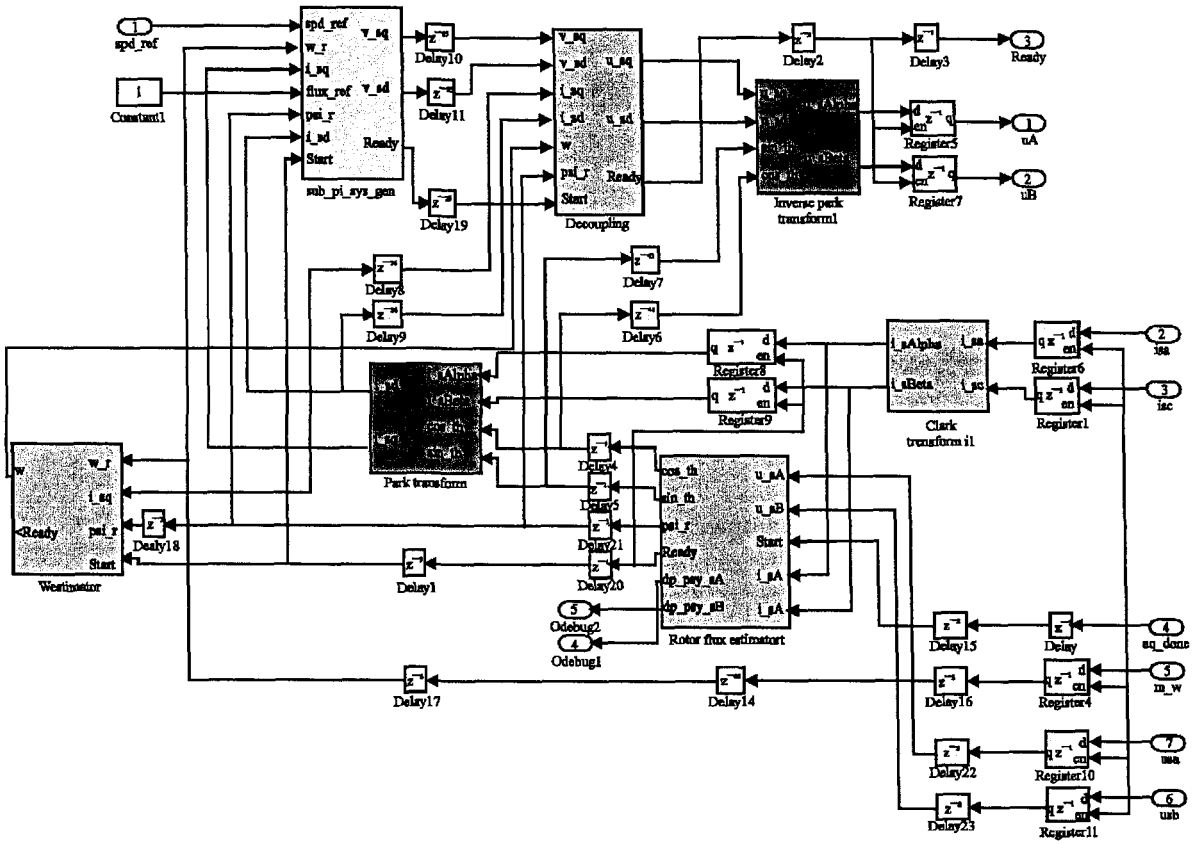


Fig. 2: Vector control coded in XSG

IMPLEMENTATION OF NON RESTORING DIVISION AND SQUARE ROOT USING XSG

Restoring and non-restoring algorithms are the basic digit recurrence schemes for division and square root extraction. The restoring algorithm uses an additional step to conditionally restore the partial remainder to its previous value when the difference is negative. Non-restoring algorithms always store the difference in the partial remainder register, eventually correcting the remainder, if necessary, by an addition in the next step. It is then the preferred algorithm to implement the division and square root operations in FPGA.

Non-restoring division: Binary division reduces to subtracting a set of numbers, each being 0 or a shifted version of the divisor Y, from the dividend X. The main equation for division is

$$X = (Y \cdot Q) + R$$

with Q and R being respectively the quotient and the remainder. This equation, along with the condition $R < Y$,

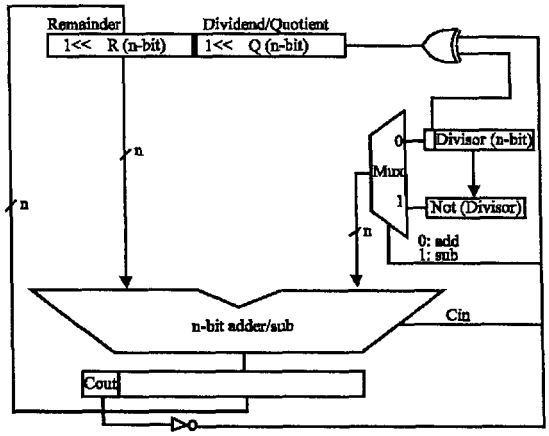


Fig. 3: Iterative non-restoring divider

completely defines unsigned integer division and the expressions for the quotient and the remainder can be derived as follows

$$Q = X/Y,\ R = X - (Y \cdot Q)$$

To obtain a fractional quotient, non restoring division can be performed iteratively by a sequence of additions and subtractions and shifts as illustrated in Fig. 3.

Non-restoring square root: Every pair of most significant bits of the unsigned 2n-bit radicand D = ($d_{2n-1}...d_0$), corresponds one bit of n-bit square root Q = ($q_{n-1}...q_0$). The square root algorithm starts by initializing the quotient Q = ($q_{n-1}...q_0$) to (0...0). Iteratively from j = n-1 down to 0, the remainder R is computed and shifted adequately as illustrated in the scheme of Fig. 4. In case of R≥0, the square root bit q_j is set to 1, otherwise it is reset to 0.

Implementation with XSG: The non-restoring division and square root algorithms were optimally and synchronously coded in XSG under Simulink environment using embedded blocs such as adders, muxes, inverters, etc. The implementations are fully parameterized with the widths declared as generic parameters of the XSG modules. This permits to select an arbitrary precision and word lengths for different operators of the division and square root. The mapping onto FPGA is done through ISE tool and the designs are analyzed in order to define their performances in terms of area usage and speed. Figure 5 and 6 illustrate the XSG codes of the division and square root for a particular case of 32-bits operators with 14 bits for binary point.

ANALYSIS AND IMPLEMENTATION RESULTS

To evaluate the precision of the proposed division and square root modules relatively to CORDIC IP Cores from XSG interface, Fig. 7 and 8 shows the computed rotor flux with respect to the idealized Simulink model for each case. The test is conducted under a complete closed loop where the motor and its power drive are simulated using SimPowerSystem. The rotor flux is controlled to have a reference value of 1Wb, under various speed profiles and loads.

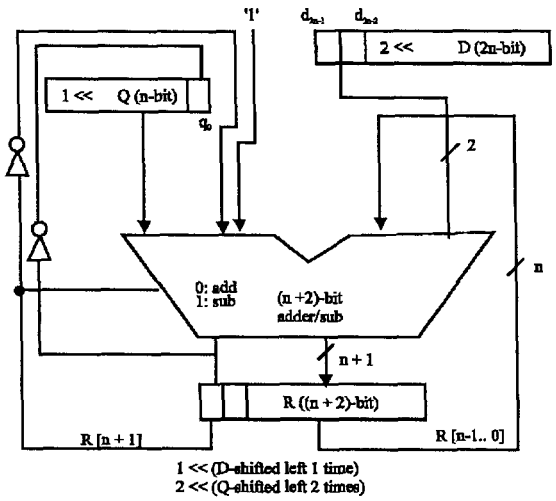


Fig. 4: Iterative non-restoring SQRT

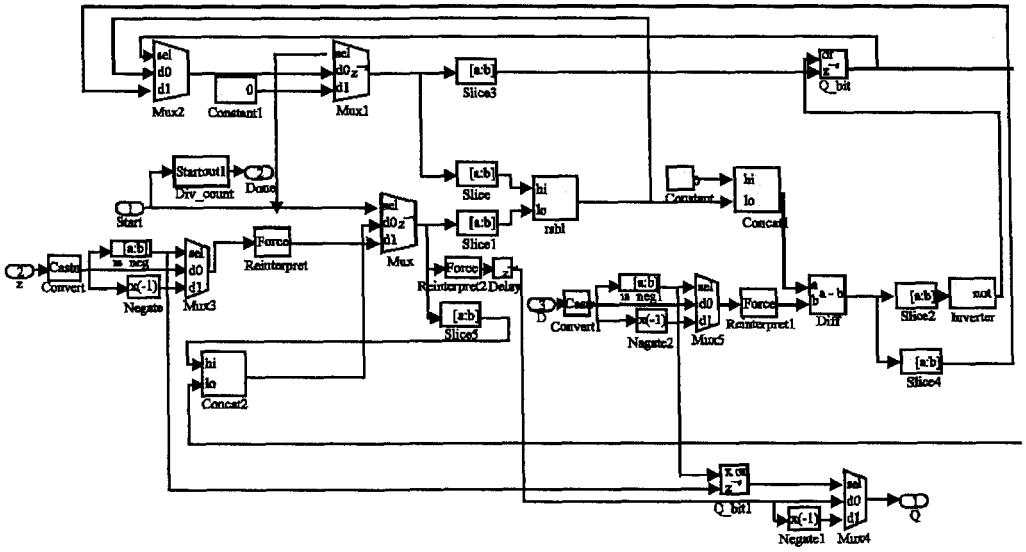


Fig. 5: XSG code for parameterized non restoring division

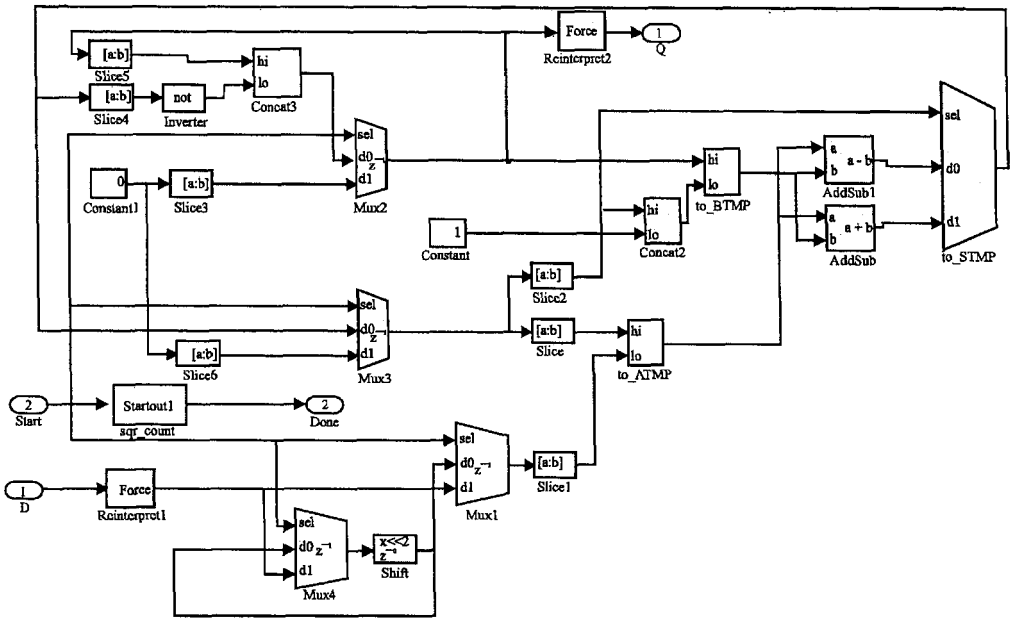


Fig. 6: XSG code for parameterized non restoring SQRT

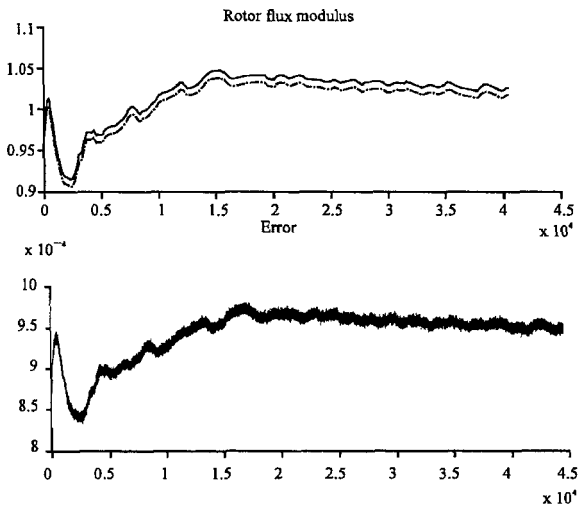


Fig. 7: Rotor Flux computed with CORDIC IP Cores vs. a floating point value from Simulink model

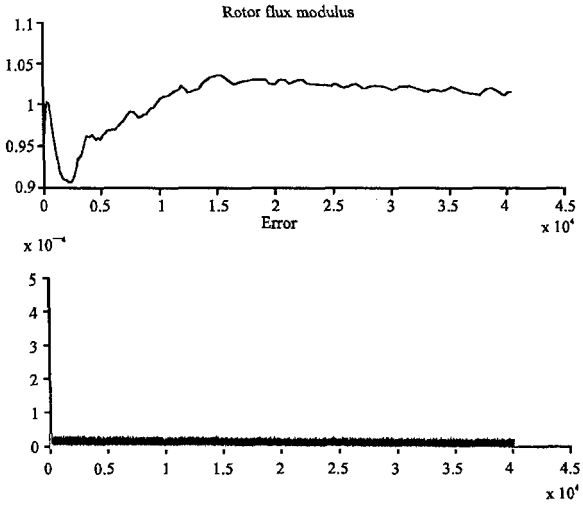


Fig. 8: Rotor Flux computed with non restoring operators vs. a floating point value from Simulink model

One can notice, for a particular precision of 32-bit, with the proposed modules the error is nil whereas it is about 1% when IP Cores are used.

Synthesis was targeted to evaluate the FPGA resources. The synthesis results are gathered in Table 1 for the division module and in Table 2 for the square root. In each case, the maximal operating frequency is given.

Table 1: Synthesis results for FPGA implementation of non-restoring division and CORDIC based division (32-bits, 14 bits for binary point)

Parameter	Non restoring division	CORDIC based division
Slices	124	1806
FFs	96	3132
LUTs	218	2981
Embedded multipliers	0	3
Steps for completion	48	47
Speed in MHz	173	77

Table 2: Synthesis results for FPGA implementation of non-restoring SQRT and CORDIC based SQRT (32-bits, 14 bits for binary point)

Parameter	Non restoring division	CORDIC based division
Slices	60	1377
FFs	82	2289
LUTs	103	2425
Embedded multipliers	0	6
Steps for completion	24	51
Speed in MHz	238	59

CONCLUSION

Some signal digital processing applications exhibit algorithms with one or more fixed-point square roots and/or divisions. In FPGA, integer square rooters and dividers are provided from an IP libraries provided by FPGA vendors. Besides their huge area utilization, they are limited in terms of precision and often the precision of the result don't exceed 32 bits. In the present study, the vector control algorithm for induction motor drives is implemented using XSG under Simulink. The rotor flux, which necessitates a couple of divisions and one square root extraction, is implemented based on a non restoring division and square root algorithms. The detailed study has shown the main advantage of such implementation which resides on its huge logic saving. Moreover, the proposed implementations of such operators lead to an impressive precision and are many times fast compared to the IP Cores.

ACKNOWLEDGEMENT

This research is funded by a grant from the National Sciences and Engineering Research Council of Canada (NSERC). Canadian Microelectronics Corporation (CMC) Microsystems provided development tools and support through the System-on-Chip Research Network (SOCRN) program.

REFERENCES

Bannur, J. and A. Varma, 1985. The VLSI Implementation of a square root. Algorithm. In: IEEE Symp. Computer Arithmetic, IEEE. Comput. Soc. Washington DC, pp: 159-165.

Beguenane, R., J.G. Mailloux, S. Simard and A. Tisserand, 2006. Towards the System-on-Chip Realization of a Sensorless Vector Controller with Microsecond-order Computation Time. In 19th IEEE Canadian Conference on Electrical and Computer Engineering, Ottawa, Canada.

Birman, M. *et al.*, 1990. Developing the WTL3170/3171 Sparc Floating-Point Coprocessors. IEEE. Micro, 10: 55-64.

Deschamps, J.P., G.J.A. Bioul and G.D. Sutter, 2006. Synthesis of Arithmetic Circuits. Copyright © John Wiley and Sons, Inc.

Harris, D.L., S.F. Oberman and M.A. Horowitz, 1997. SRT division architectures and implementations. In: Proc. 13th IEEE. Int. Symp. Comput. Arithmetic, pp: 18-25.

Kabuo, H. *et al.*, 1994. Accurate rounding scheme for the newton-raphson method using redundant binary representation. IEEE. Trans. Comput., 43: 43-51.

Li, Y. and W. Chu, 1997. Implementation of single precision floating point square root on FPGAs. In: 5th IEEE. Symp. FPGA-Based Custom Comput. Mach. (FCCM), Napa, California, USA., pp: 226-233.

Marnane, W.P., S.J. Bellis and P. Larsson-Edefors, 1997. Bit-serial interleaved high speed division. Elec. Lett., 33: 1124-1125.

O'Leary, J.W., M. Leeser, J. Hickey and M. Aagaard, 1994. Non-restoring integer square root: A Case Study in Design by Principled Optimization. In TPCD, 2nd Int. Conf. Theorem Provers in Circuit Design-Theory, Practice and Experience, Springer-Verlag, London, UK., pp: 52-71.

Piromsopa, K., C. Aporntewan and Chongstitvatana, 2001. An FPGA implementation of a fixed-point square root operation. In: International Symposium on communications and Information Technologies, ISCIT.

Ricci, F. and H. Le-Huy, 2003. Modeling and simulation of FPGA-based variable-speed drives using Simulink. Math. Comput. Simulation, 63: 183-195.

Sorokin, N., 2006. Implementation of high-speed fixed-point dividers on FPGA. JCS and T J., Vol. 6.

Vásárhelyi, J., M. Imecs, C. Szabó and I.I. Incze, 2005. FPGA Implementation vector control of tandem converter fed induction machine. In 6th International Symposium of Hungarian, Researchers on Computational Intelligence, Budapest, Magyar.

Xilinx Inc, 2006. System Generator for DSP. http://www.xilinx.com/ise/optional_prod/system_generator.htm

Optimal FPGA implementation of Unsigned Bit-serial division

Stéphane Simard, Jean-Gabriel Mailloux, Rachid Beguenane

Abstract – *In the present paper, we show that a slight modification to a well-known unsigned nonrestoring division algorithm leads to an optimal mapping of a bit-serial divider to FPGA hardware. Advantages of the proposed implementation are: minimal area occupancy, and no online delay (i.e. the MSB of the quotient is obtained right with the next clock cycle after input of the first bit of the operands). Synthesis results are presented for two different Xilinx families of FPGAs, and different operand widths up to 64 bits. These results show that the number of 4-input LUTs occupied by one such divider is at most equal to operand length, and that the maximum clocking frequency largely exceeds 100MHz in every case tested.*

Keywords: *FPGA, Unsigned division, Nonrestoring, Bit serial*

I. Introduction

By its very nature, division is an iterative process, where a single digit of the quotient is produced per iteration stage. In other words, the quotient is always produced with the most significant digits first (MSDF) in a serial fashion. In every stage during this process, all of the digits of the divisor are needed in parallel, but a single digit of the dividend is consumed.

Restoring and nonrestoring algorithms are the basic digit recurrence schemes for division. While the restoring algorithm uses an additional step per stage to conditionally restore the partial remainder to its previous value when the difference is negative, nonrestoring algorithms always store the difference in the partial remainder register, eventually correcting the remainder by an addition in the next step. Nonrestoring algorithms are therefore simpler, and faster. It is to be remarked that the fastest known division algorithm is nonrestoring SRT division.

Let N be the number of iterations needed with a divisor N digits wide. A divider module can be implemented either as a single stage that loops N times onto itself ("sequential divider"), or by unrolling the basic iteration to construct a large array of N stages ("array divider"), which involves duplicating the hardware for a stage as many times as the number N of iterations needed. Pipelining between stages is then usually needed in order to increase clocking frequency. An array divider is N times faster than a sequential one, but also occupies in the order of N times as much chip area.

Conventional implementations accept both operands in parallel, and, by parallelizing the quotient, give the result in the same fashion. They can either be sequential, or array dividers. Digit-serial implementations, on the other hand, are sequential in nature. They accept at least one operand (the dividend) serially, MSDF, and produce the result in the same way.

Sequential dividers, either parallel or serial, are often preferred in applications where economy of chip area is a concern, for example in some large digital signal processing and/or control applications. The simplest form of a sequential divider, with regards to the natural workings of division, is a digit-serial one, but conversion to parallel form is often needed in order to connect to other arithmetic operators, such as adders and multipliers.

Previous examples of serial dividers have been presented in [1]-[4]. In [5], a fully parallel FPGA implementation of division has shown the advantages of the non-restoring algorithm versus the restoring one in terms of speed and logic size. In [6], a fast radix 4 division algorithm using lookup table based FPGAs implementation has been presented. The quotient digits are determined by observing three most-significant radix 2 digits of the partial remainder. Implementations of bit-serial division which lead to very compact modules have been presented only for VLSI [1-4]. Marnane et al. [4] presented a two's complement serial–parallel divider based on the non-restoring algorithm. For n-bit divisor, the circuit provides a latency of n clock cycles before producing the first bit of the result.

The implementation proposed in this paper, which takes one operand in series, and the other in parallel, maps perfectly to the underlying FPGA hardware while having no online delay (i.e. the MSB of the output is available after the first clock cycle). This is made possible thanks to a slight modification to conventional unsigned nonrestoring division, and it leads to an implementation having roughly the same size as a carry-propagate adder. A divider with such characteristics

Manuscript received January 2007, revised January 2007

would be very adaptable to all kinds of interfaces with other arithmetic operators, as well parallel as serial, and could even serve as a basic building block for array dividers on FPGAs.

Minimizing the output delay is very useful in MSBF computations, for example when one is interested only in the most significant bits of the result. With signed nonrestoring division, an additional correction stage would be necessary, and the online delay could become *O(N)*. This will be addressed in a subsequent article.

This paper is organized as follows. We begin in section II with a brief review of the nonrestoring division algorithm, followed in section III by a detailed description of a novel FPGA structure for unsigned divisions. Its compactness and its minimal latency of only one clock cycle will be proved in Section IV which will discuss the implementation results before concluding in section V.

II. Non-Restoring Division Algorithm

The problem of binary division reduces to subtracting a set of numbers, each being 0 or a shifted version of the divisor Y, from the dividend X. The main equation for division is

$$X = (Y \cdot Q) + R \tag{1}$$

with Q and R being respectively the quotient and the remainder. This equation, along with the condition $R \prec Y$, completely defines unsigned integer division, and the expressions for the quotient and the remainder can be derived as follows

$$Q = X / R \tag{2}$$

$$R = X - (Y \cdot Q) \tag{3}$$

To obtain a fractional quotient, division can be performed by a sequence of subtractions and shifts. In each cycle or step j, the remainder is compared against the divisor. If the remainder is larger, then the quotient bit is set to1, otherwise it is set to 0. The comparison between the divisor and the remainder is the most critical operation. If this is done by subtracting Y from R, then whenever the result is negative the remainder must be restored to its previous value. This is a restoring division as it requires a subtraction followed by an addition to restore for each zero in the quotient. In non-restoring division, the decision whether to perform addition or subtraction in each step is based on the sign digit of the current partial remainder R. It only requires one operation, either addition or subtraction for each bit in quotient. Its algorithm can be found in many papers and text books, and it is summarized as follows

Algorithm 1 : Non-restoring Parallel Division

X, Y, Q and R are n-bits words

[*Initialisation*]
$R = 0$, $Q = X$

[*Recurrence*]
for $j = 0 \ldots n-1$ **do**
- step 1: $RQ = 2RQ$ (shift left both R and Q by one bit, the MSB is lost)
- step 2: **if** $R \geq 0$ **then** $R = R - Y$ **else** $R = R + Y$ **end if**
- step 3: **if** $R \geq 0$ **then** $q_0 = 1$ **else** $q_0 = 0$ **end if**

(R is computed from previous if statement and q_0 is the updated LSB of Q)

end for

[*Posterior correction for the remainder to be positive*]
if $R \prec 0$ **then** $R = R + Y$ **end if**

III. Optimal Mapping for FPGA

In order to derive an optimal mapping of bit-serial unsigned division to FPGA, a slight modification has been made to Algorithm 1, leading to Algorithm 2, which directly maps to the underlying hardware in a very simple way. Algorithm 2 is stated as follows:

Algorithm 2 : Non-restoring bit-serial Division

Y, is an n-bits parallel word, X is an n-bits loaded serially MSB first, i.e. x_i

R, is an n-bits parallel word, Q is an n-bits outputted serially MSB first, i.e. q_i

NOTE: x_{n-1}, and q_{n-1} are MSB bits, x_0, and q_0 are LSB bits

[*Initialisation*]
$R = 0$

[*Recurrence*]
for $j = n-1 \ldots 0$ **do**

$X = 2X$ (shift left X by one bit, MSB bit x_j is added/subtracted to/from Y giving a new R)

if $R \geq 0$ **then** $R = R - Y$ **else** $R = R + Y$ **end if**

if $R \geq 0$ **then** $q_j = 1$ **else** $q_j = 0$ **end if** (R is computed from previous **if** statement)

end for

[*Posterior correction is possible by adding a supplementary iteration*]

Fig. 1 illustrates the proposed mapping of Algorithm 2 to FPGA circuitry leveraging dedicated

carry logic. Dedicated carry logic is a fast circuit resource present in most modern FPGAs that take care of carry propagation outside LUTs, thus sparing the need for extra logic. The divider module of Fig. 1 consists of n *add-sub-register* cells, plus one additional full adder cell, for the operation to be unsigned. As the dividend is being loaded serially, MSBF, the quotient is produced in the same fashion with no online delay.

Each iteration, the addition/subtraction of the divisor Y to/from the partial remainder R when the previous value of R is negative/positive. The XOR gates along with the input carry implement conditional addition/subtraction. It can be noticed that non-restoring division should always start by a subtraction, hence the presence of an inverter to force the control signal AddSub to be '1' at startup.

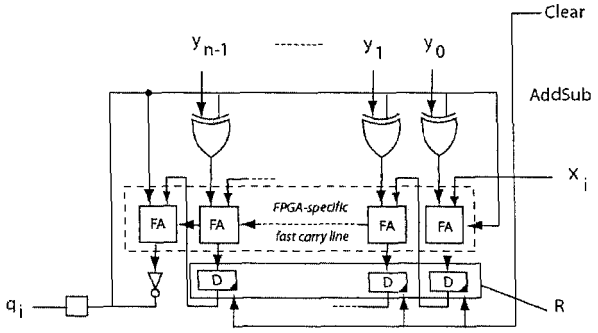
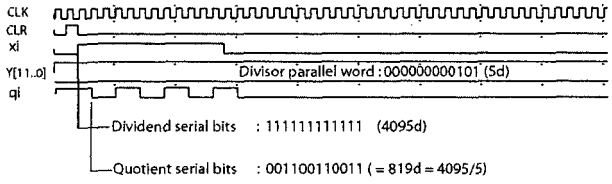


Fig. 1. Unsigned bit-serial non-restoring divider

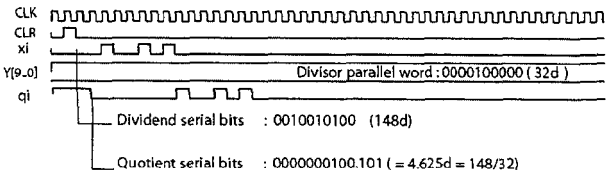
The circuit could be clocked at very high frequencies by pipelining the carry propagation, if it weren't for the presence of retroaction, necessary for the division algorithm to run correctly. The retroaction therefore constitutes the design's critical data path that limits maximum clocking frequency.

IV. Synthesis Results

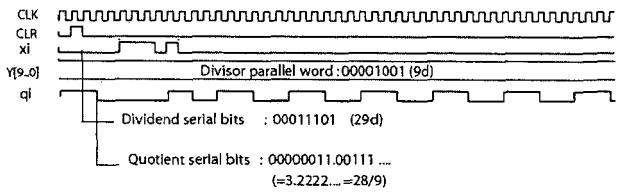
The proposed divider has been described in the VHDL hadware description language, and simulated. Care has been taken to ensure the insertion of a dedicated carry chain. The implementation has been synthesized for performance evaluation using the ISE toolkit from Xilinx, and ModelSim has been used for simulation in order to validate the proposed design. Fig. 2 shows some simulation examples for different operand widths. It can be observed that, in all cases, the MSB of Y is available right on the next clock cycle after input of the MSB of X, therefore no online delay.



(a) Example 1 (operands are 12-bit words): 4095 / 5 = 819



(b) Example 2 (operands are 10-bit words): 148 / 32 = 4.625



(c) Example 3 (operands are 8-bit words): 29 / 9 = 3.22222...

Fig. 2. Functional simulation of bit-serial non-restoring division with one clock cycle of latency

Currently, the CLR signal, which clears the partial remainder, is the only control signal needed to initialise the divider. Operator control has been left very simple for more flexibility, and the user may add posterior control signals if desired.

TABLE I
SYNTHESIS RESULTS FOR FPGA IMPLEMENTATION OF BIT-SERIAL NON-RESTORING DIVISION

N	Frequncy (MHz)		Area		
	Spartan 2 XC2s15-6	Virtex 2 XC2v40-6	Slices	FFs	LUTs
8	164	304	6	9	10
12	151	284	8	13	14
16	139	269	10	17	18
32	129	228	18	34	34
64	112	177	36	68	66

Synthesis results for different operand widths, targetting Spartan 2 and Virtex 2 FPGA families from Xilinx, are gathered in Table I. In each case, the results show that not only this architecture leads to a very low area usage, but it achieves a quite reasonable maximum clocking frequency of more than 100 MHz — even up to as much as 64 bits wide.

V. Conclusion

In the context of advanced DSP and control algorithm prototyping on FPGA, fixed-point dividers are often needed, which most of the time are instantiated from IP libraries provided by FPGA vendors. Besides being huge, vendor-supplied divider cores often have limited operand width and result precision.

The divider structure proposed in this paper has led to a very compact FPGA implementation thanks to a modified nonrestoring bit-serial unsigned division algorithm. As it can be coupled to almost any arithmetic network, either with parallel or serial operands, through very simple conversions, it can be used efficiently in a wide array of applications, especially those using a sign-and-absolute-value numerical representation. An adaptation for two's complement representation can be made simply by conditionally negating the quotient as a function of the signs of both operands.

Performances claimed for the proposed divider on FPGA have been confirmed by synthesis and simulation.

The expected timings have been confirmed by simulation, and the synthesis results have shown the implementation to consume a number of 4-input LUTs at most equal to operand width, and maximum clocking frequencies largely exceeding 100 MHz were obtained in every case tested. We can therefore conclude that the proposed structure is optimal for FPGA devices with dedicated carry logic. In addition, it is worthwhile to mention that an equivalent VLSI implementation can also be easily derived.

Acknowledgements

This research has been conducted in the context of a NSERC-Discovery grant titled "FPGA-based reconfigurable computing for digital motion controllers," from the National Sciences and Engineering Research council of Canada. CMC Microsystems gracefully provided prototyping stations and design automation software through their System-on-Chip Research Network (SOCRN) program.

References

[1] A. E. Bashagha, Pipelined area-efficient digit serial divider, *Signal Processing*, Vol. 83, No. 9, pp. 2011–2020, 2003.

[2] A. E. Bashagha, and M. K. Ibrahim, A new digit-serial divider architecture, *International Journal of Electronics*, Vol. 75, No. 1, pp. 133–140, 1993.

[3] S. J. Bellis, W. P. Marnane, and P. Larsson-Edefors, Bit-serial, MSB first processing units, *International Journal of Electronics*, Vol. 86, No. 6, 1999, pp. 723-738.

[4] W. P. Marnane, S. J. Bellis, and P. Larsson-Edefors, Bit-serial interleaved high speed division, *Electronics Letters*, Vol. 33, No. 13, pp. 1124–1125, 1997.

[5] Nikolay Sorokin, Implementation of high-speed fixed-point dividers on FPGA, *JCS&T*, Vol. 6, No. 1, April 2006

[6] A. I. Attif, A. E. Hamed, A. E. Salama, *FPGA Implementation of Fast Radix 4 Division Algorithm*, Proceedings of the 4th IEEE International Workshop on System-on-Chip for Real-Time Applications, Page 69, Year of Publication: 2004).

[7] Xilinx. 2003b. Using Look-Up Tables as Shift Registers (SRL16) in Spartan-3 Devices, *Application Note XAPP465*, Vol. 1.0, Xilinx, inc., San Jose, CA.

Authors' information

Stéphane Simard is a Ph.D student in computer engineering at the University of Quebec at Chicoutimi, Canada. He has a college degree in industrial informatics, and he received the B.Sc degree with a major in computer science, followed by a specialization diploma in applied informatics, and the M.Sc.A degree in computer engineering, from the same university. He works since 2004 as a research assistant within the ERMETIS research group in microelectronics and digital signal processing, in the department of applied sciences of his university. During his master's research he studied computer arithmetic implementation tradeoffs on field-programmable gate arrays, in the context of a commercial project to further the development of psC, a new, purely parallel, language for real-time and reconfigurable systems programming. He is currently working on a very-high-performance system-on-chip sensorless AC induction motor vector controller. His current professional interests include hardware-software codesign of embedded systems and controllers, computer arithmetic, digital design & VLSI, and industrial electronics.

Jean-Gabriel Mailloux was born in Fortierville, Canada, in 1982. He received the B.Ing degree in computer engineering from the University of Quebec at Chicoutimi, Canada, in 2005. He is currently working toward the M.Sc.A degree at the same university. In 2005, he joined the ERMETIS research group in microelectronics and digital signal processing, in the department of applied sciences of his university. He is currently working on reference designs for implementing sensorless AC induction motor vector control in field-programmable gate arrays. His current research interests include optimal approaches for testing control algorithms in FPGAs, and other control mplementation issues.

Rachid Beguenane received his B.Sc. degree in Electronic Engineering from Algiers University of Sciences and Technologies, Algeria, in 1988. He earned his D.E.A. and Ph.D. degrees, both in Electrical Engineering, from Conservatoire National des Arts et Mtiers, Paris, France, in 1991 and 1994 respectively. During this period, he conducted his Ph.D. research work at École des Mines de Douai, France. From 1995 to 1997, he holds a teaching and research position at the University Professional Institute of Amiens, France. During 1997/1998, he was a post-doctoral researcher at the School of Information Technology and Engineering of the University

of Ottawa, Canada. From 1998 to 2002, he has been employed as ASIC/FPGA designer by Telexis Inc. (Now March Networks), Nortel Networks, and Prova Scientific Corp. in Ottawa and Montreal, Canada. He mainly designed and verified IC chips for telecommunication industry. He is employed by the University of Quebec at Chicoutimi (UQAC), Canada, since August 2002, as an associate professor in electrical and computer engineering. He is currently the member representative of the Canadian Microelectronics Corporation at UQAC. His main research interests include realtime hardware design and implementation using µP/DSP//FPGA/ASIC for both microelectronic and macro-electronic systems, analysis and control of high performance electrical drives, hardware-software codesign of embedded controllers. Dr. Beguenane has authored and co-authored more than 30 technical papers, and he is member of the Ordre des Ingénieurs du Québec.

Optimized FPGA Mapping of a Bit-serial Square Root Operator With Minimum Output Delay

Stéphane Simard[1], Jean-Gabriel Mailloux[1], Rachid Beguenane[1]

Abstract – *We propose an FPGA implementation of a bit-serial square root operator based on the nonrestoring algorithm which optimally maps to the underlying hardware, and has minimal output delay. The most significant bit (MSB) of the square root is obtained right with the next clock cycle after input of the first bit of the radicand. Synthesis results for different radicand widths in the range of 8 to 64 bits targetting two different Xilinx families of FPGAs have shown this design to consume a number of 4-input LUTs at most twice the given width, and maximum clocking frequencies in excess of 100 MHz are possible in all cases. The proposed operator has been designed for use in advanced control loops, and has been found suitable for the intended purpose.*

Keywords: *Template, International Review on Computers and Software*

I. Introduction

This contribution has been motivated by the need for a compact, yet relatively fast square root operator for use in an advanced electric motor controller. The vendor-supplied square root included with some reputable design automation software was based on the general CORDIC algorithm, and turned out to be unsuitable. In any configuration, this operator had an inadequate 40-stage pipeline depth, consumed a huge amount of chip area, and still could not be clocked faster than about 58 MHz. The target application needed a system-wide clocking of around 100 MHz. The square root input had to be 32-bit precision fixed-point including 14 fractional bits, and the output had to be 16-bit with as much fractional precision. For some unexplained reason, the root produced by the vendor's operator turned out to have not more than one decimal digit accuracy after the point, even though it did in fact output an impressive 74 bits in total. As an indication, the following table (Tab. 1) lists some comparative figures between the vendor-supplied CORDIC implementation and our own proposal on a Virtex-4 SX35 FPGA (XC4VSX35-10).

TABLE I
INDICATIVE COMPARISON BETWEEN THE VENDOR-SUPPLIED SQUARE ROOT VS. THE PROPOSED DESIGN (32-BIT FIXED-POINT INPUT, 14-BIT FRACTIONAL), WITH THE XC4VSX35-10 AS TARGET

Square root implementation	Virtex Slices	FFs	4-LUTs	Max. freq.
Vendor:	1377	2289	2425	58.666
Proposed:	60	82	103	238.112

The square root operator proposed here has been designed for minimum delay, and minimum area usage on FPGA. It is therefore internally organized as a single looping stage, and has a bit-serial flow where the n-bit radicand and the resulting root are presented with their most significant bit first (MSBF). In order to obtain an implementation that maps to FPGA hardware as well as possible while having minimal bit output delay, a slightly modified version of a conventional nonrestoring algorithm has been used. This has led to the simplest and most area efficient FPGA mapping of a bit-serial square rooter with a bit output delay of only one clock cycle.

Digit-serial architectures, such as those proposed in [1] and [2], even though they are more complicated, might sometimes be considered when seeking a compromise between the speed of parallel array operators and the economy of bit-serial ones. While the former are mostly used in chip designs where a few critical arithmetic operators are shared and reused by different operations, for instance in microprocessors, the latter are often preferred in applications where chip area economy is a concern, such as custom circuits for some large digital signal processing and/or control applications. In application-specific circuits, as opposed to processors, even though individually slower serial operators might be used, great computations speeds can be attained thanks to the direct linking of operations in the form of arithmetic networks, with no need for such overhead as instruction decoding.

The MSBF mode is convenient when one is only interested in a certain number of the most significant digits of the result, as computations do not need to be allowed to progress further. On the opposite, it is worth knowing that serial dividers and square rooters are able

Manuscript received January 2007, revised January 2007

to provide arbitrary fractional precision if left running long enough with the same operands. For instance, a bit-serial square rooter such as the proposed one, provides one additional bit of precision per clock cycle it is left running.

Minimizing the bit output delay is useful in MSBF computations as it allows for the time-overlapping of cascaded operations, which improves the overall speed in an arithmetic network. For example, if the output of a bit-serial square root is fed as input to a bit-serial divider, the divider receives the MSB of its input and starts its computation only one clock cycle after the input the MSB of the square root operand. The two operators are thus practically computing simultaneously.

Several implementations of nonrestoring square root can be found in the litterature, as well for VLSI as for FPGA. Examples include [4]-[6] for VLSI, and [7], [8] for FPGA. In [7], two floating point implementations have been proposed, and in [8], it is a 32-bit fixed point one. However, although these implementations were synthesized using design automation software, and evaluated for FPGA, they are not bit-serial designs, as in our own proposal, and as their design goals were different, their mapping to FPGA hardware has not been shown to be optimized.

The paper is organized as follows. Section II describes the conventional nonrestoring square root algorithm versus the modified version. Section III presents the proposed FPGA mapping for fixed-point square root. Its compactness and its minimal delay of only one clock cycle will be shown in Section IV which will discuss the implementation results before concluding in section V.

II. The nonrestoring algorithm

Square root extraction is fundamentally an iterative process, where a single digit of the root is produced every step. Hardware square root operators are generally based either on restoring or nonrestoring algorithms, as it is the case for divider operators, and can be tailored to different needs by trading off area for speed or vice versa. Nonrestoring algorithms [3] are the fastest and most area-efficient ones for hardware implementation, as they do not use an extra step for restoring the reminder to its previous value in the cases where the current quotient digit turns out to be zero after trial. Depending on the design objectives, the flow of operands and results will be selected to be parallel or serial, and the internal architecture will be organized either as an array of successive stages when trading off for speed, or as a single looping stage when trading off for area.

Binary square root algorithms start by initializing the square root $Q=(q_{n-1}\cdots q_0)$ to $(0\ldots 0)$. The square root of a $2n$-bit radicand has n bits. To each pair of the most significant bits of the unsigned radicand $D=(d_{2n-1}\cdots d_0)$ correspond a single bit of the integer part of $Q=(q_{n-1}\cdots q_0)$. For $j=n-1$ down to 0, q_j is first assumed to be 1, and a trial remainder $R=D-Q\cdot Q=(r_n\cdots r_0)$ is computed. In the cases where this remainder turns out to be negative, this indicates a wrong guess, and the way this is handled differentiates between restoring and nonrestoring algorithms. A restoring algorithm takes an extra step to restore R to its previous value, and corrects q_j to be 0 instead of 1. In a nonrestoring square root, the remainder is computed differently depending on the sign of its previous value, and, as in the restoring version, q_j is corrected accordingly. In this case, however, everything is done on-the-fly, and there is not an extra step to slow down the computations. The nonrestoring square root procedure is formalized in Algorithm 1, and a corresponding circuit is shown in Fig. 1.

Algorithm 1 : Nonrestoring Square Root

[*Initialisation*]:
For the algorithm to function correctly, the remainder is extended by an extra sign bit. Initially:
$R=(0r_n\ldots r_0)=(00\ldots 0)$ and $Q=(q_{n-1}\cdots q_0)=(0\ldots 0)$.

Note: $\vee, \neg$ are respectively the **OR** and **Concatenation** operators. $(\bullet \prec\prec m)$ means shift $(\bullet)$ left by m bits.

[*Recurrence*]:
for n - 1 to 0 **do**
if $R \geq 0$
then $R' = (R \prec\prec 2) \vee (0\ldots 0 d_{2j+1} d_{2j}) - (Q \neg 01)$
else $R' = (R \prec\prec 2) \vee (0\ldots 0 d_{2j+1} d_{2j}) + (Q \neg 11)$
end if

if $R' \geq 0$ **then** $Q = (Q \prec\prec 1) \vee (0\ldots 01)$ **else** $Q = (Q \prec\prec 1)$
end if

$R = R'$
end for

[*Correction*]:

if $R<0$ **then** $R = R + Q \neg 11$ **end if**

Algorithm 1 performs a set of substitutions or additions depending on the sign of the remainder computed during the previous iteration. The radicand is

Manuscript received January 2007, revised January 2007

the one and only operand. It can be remarked that every step a pair of bits of the radicand is processed simultaneouly, starting with the two most significant bits, and shifting left by two positions to fetch the next ones.

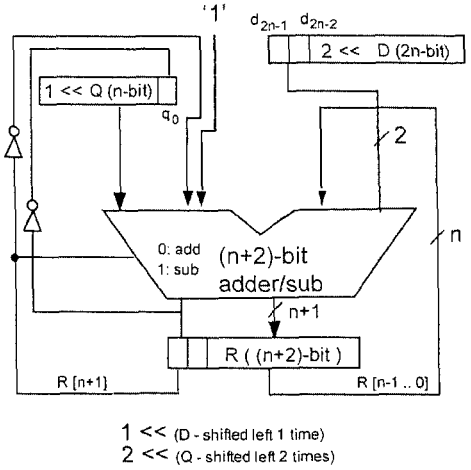


Fig. 1. Circuit Implementing Algorithm 1

As for the fonts and the sizes of the headings, this manuscript in itself constitutes a good example.

III. Optimized FPGA Mapping

In order to meet the objectives stated in introduction, an optimized mapping of Algorithm 1 to FPGA hardware has been devised, as illustrated in Fig. 2. This constitute proof by construction. The core of the design, composed of a conditional add/subtract cell made up of an XOR gate, a full adder, and a register, is very simple, and tightly fit within a 4-input LUT with dedicated carry logic on FPGA. The result is a very compact operator which can either be used as is in a serial arithmetic network, or be adapted to a parallel data flow by serializing the radicand, and parallelizing the root.

The Clear signal is the only control signal needed to initialize the partial remainder and the root. Control management has been left to the minimum for more flexibility. Besides the compactness and economy of this design, its bit output delay is indeed of only one clock cycle, and is therefore minimal.

IV. Simulation and Synthesis Results

The proposed design has been described in VHDL (VHSIC Hardware Description Language), and tested by simulation. Care has been taken in the HDL description to ensure that a dedicated carry chain would be inferred at the hardware synthesis step, in order to obtain a proper performance evaluation. The software tools used were ModelSim for simulation, and Xilinx XST for synthesis. Fig. 3 shows simulation examples for different radicand widths. It can be observed that the MSB of the root is indeed provided on the next clock cycle just after the Clear signal.

Performance evaluation for different radicand widths has been done by hardware synthesis targetting two different Xilinx FPGA families: the Spartan 2 and the Virtex 2. The results for maximum frequency and estimated area consumption are gathered in Table II.

TABLE II

SYNTHESIS RESULTS FOR MAXIMUM FREQUENCY AND AREA CONSUMPTION

N	Freq. (MHz)		Area		
	Spartan 2 XC2s15-6	Virtex 2 XC2v40-6	Slices	FFs	4LUTs
8	133	251	13	23	16
12	129	244	18	33	22
16	125	237	24	43	28
20	121	231	30	53	34
24	117	225	35	63	40
32	110	215	46	83	52
64	102	188	92	164	100

V. Conclusion

Some algorithms used in digital signal processing and control applications include one or more fixed-point square roots. IP libraries provided by design automation vendors can be used, provided that they meet the application's requirements. Unfortunately, the libraries from prominent vendors often have important drawbacks, such as limited clocking speeds, and important area occupancy as a function of precision.

We have proposed a very simple mapping of a bit-serial nonrestoring square root to FPGA hardware that leads to a very compact, and relatively fast operator. Our discussion has shown that the main advantages of this operator are: minimal bit delay (of only one clock cycle) and optimal FPGA mapping using dedicated carry logic, as it consumes a number of slices at most equal to two times the radicand width.

The implementation has been tested for two Xilinx families with different radicand widths up to 64 bits. In every tested case, simulation and synthesis results corroborate the claim of very low area usage, and show that maximum clocking frequencies of more than 100 MHz can be achieved, which is considered fast on FPGA.

The proposed bit-serial square root operator can be coupled to any arithmetic network, either with a parallel

Manuscript received January 2007, revised January 2007

or a serial data flow, through simple conversions. Equivalent VLSI implementations can be easily derived for use in ASIC design.

Acknowledgements

This work is part of a research funded by a grant from the National Sciences and Engineering Research Council of Canada (NSERC) on the use of FPGAs for the control of electric machines. CMC Microsystems provided prototyping stations and design automation software through their System-on-Chip Research Network (SOCRN) program.

References

[1] M. K. Ibrahim, A. E. Bashagha, Area-time efficient two's complement square root, *International Journal of Electronics*, Vol. 86, Issue 2, pp. 127-140, 1999.

[2] A. E. Bashagha, M. K. Ibrahim, Radix digit-serial pipelined divider/square-root architecture, Computers and Digital Techniques, *IEE Proceedings*, Vol. 141, Issue 6, pp. : 375 – 380, Nov. 1994.

[3] O'Leary, J., Leeser, M., Hickey, J. and Aagaard, M., *Nonrestoring Integer Square Root: A Case Study in Design by Principled Optimization Source,* Proceedings of the Second International Conference on Theorem Provers in Circuit Design - Theory, Practice and Experience (Pages: 52 – 71, Year of production: 1994).

[4] Bannur, J., and Varma, A., *A VLSI implementation of a square root algorithm*, In IEEE Symposium on Computer Arithmetic, IEEE Comp. Soc. Press (Pages: 159-165, Washington D.C., Year of Publication: 1985).

[5] Takagi, N., Takagi, K., *A VLSI Algorithm for Integer Square-Rooting*, International Symposium on Intelligent Signal Processing and Communications, ISPACS '06 (Pages: 626-629, Year of Production: 2006).

[6] Yamin, L., Wanming, C., *A New Non-Restoring Square Root Algorithm and its VLSI Implementation*, Proceedings of the International Conference on Computer Design, VLSI in Computers and Processors (Pages. 538 – 544, Year of Production : 1996).

[7] Yamin, L., Wanming, C., *Implementation of single precision floating point square root on FPGAs*, 5th IEEE Symposium on FPGA-Based Custom Computing Machines (Pages: 226-232, Year of Production: 1997).

[8] K. Piromsopa, K., Aporntewan, C., Chongstitvatana, P., *An FPGA implementation of a fixed-point square root operation*, International Symposium on Communications and Information Technology, (Pages 587-589, Thailand, Year of Publication: 2001).

Authors' information

[1] Université du Québec à Chicoutimi, Quebec, Canada.

Stéphane Simard was born in the province of Quebec, Canada, in 1971. He specializes in system-on-chip design, digital control, computer architecture and arithmetic, and high-performance computing, and is concerned with social implications of technology. Currently a doctorandus in engineering (electrical & computer) at Université du Québec à Chicoutimi (UQAC), he previously earned a M.Sc.A degree in engineering, a specialization degree in applied informatics, and a B.Sc degree with a major in computer science from the same university. He works since 2004 as a research assistant within the ERMETIS research group in microelectronics and digital signal processing, at UQAC, where he is doing research on system-on-chip integration of advanced electric motor controllers.

Jean-Gabriel Mailloux was born in Fortierville, Canada, in 1982. He received the B.Ing degree in computer engineering from the University of Quebec at Chicoutimi, Canada, in 2005. He is currently working toward the M.Sc.A degree at the same university. In 2005, he joined the ERMETIS research group in microelectronics and digital signal processing, in the department of applied sciences of his university. He is currently working on reference designs for implementing sensorless AC induction motor vector control in field-programmable gate arrays. His current research interests include optimal approaches for testing control algorithms in FPGAs, and other control implementation issues.

Rachid Beguenane received his B.Sc. degree in Electronic Engineering from Algiers University of Sciences and Technologies, Algeria, in 1988. He earned his D.E.A. and Ph.D. degrees, both in Electrical Engineering, from Conservatoire National des Arts et Mtiers, Paris, France, in 1991 and 1994 respectively. During this period, he conducted his Ph.D. research work at ' Ecole des Mines de Douai, France. From 1995 to 1997, he hold a teaching and research position at the University Professional Institute of Amiens, France. During 1997/1998, he was a post-doctoral researcher at the School of Information Technology and Engineering of the University of Ottawa, Canada. From 1998 to 2002, he has been employed as ASIC/FPGA designer by Telexis Inc. (Now

Manuscript received January 2007, revised January 2007

March Networks), Nortel Networks, and Prova Scientific Corp. in Ottawa and Montreal, Canada. He mainly designed and verified IC chips for telecommunication industry.
He is employed by the University of Quebec at Chicoutimi (UQAC), Canada, since August 2002, as an associate professor in electrical and computer engineering. He is currently the member representative of the Canadian Microelectronics Corporation at UQAC. His main research interests include real-time hardware design and implementation using μP/DSP//FPGA/ASIC for both microelectronic and macro-electronic systems, analysis and control of high performance electrical drives, hardware-software codesign of embedded controllers. Dr. Beguenane has authored and co-authored more than 30 technical papers, and he is member of the Ordre des Ingénieurs du Québec.

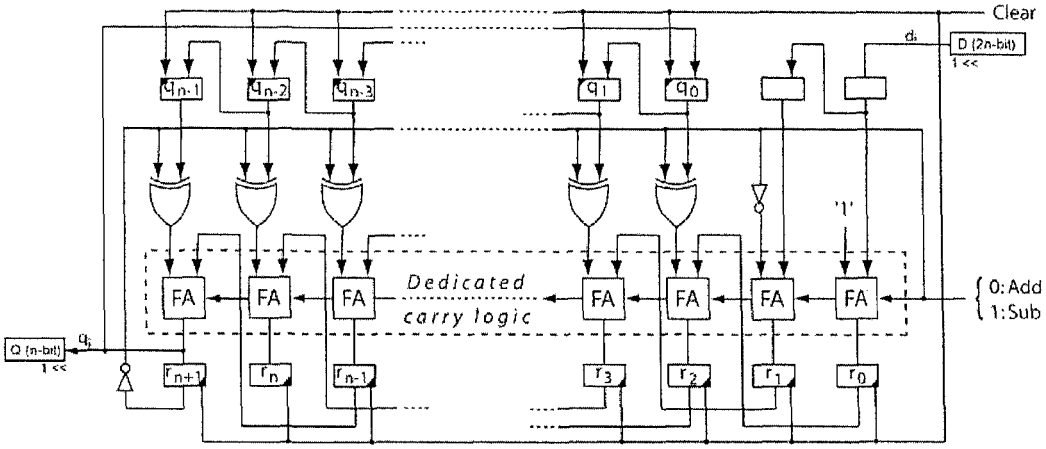
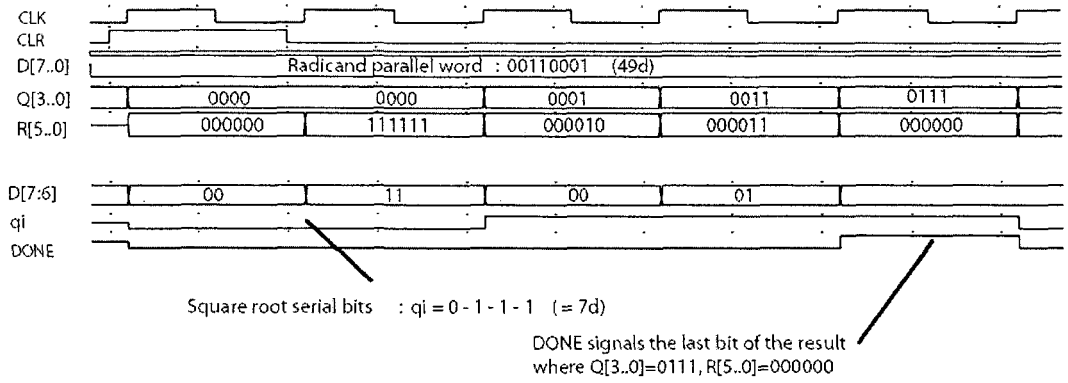
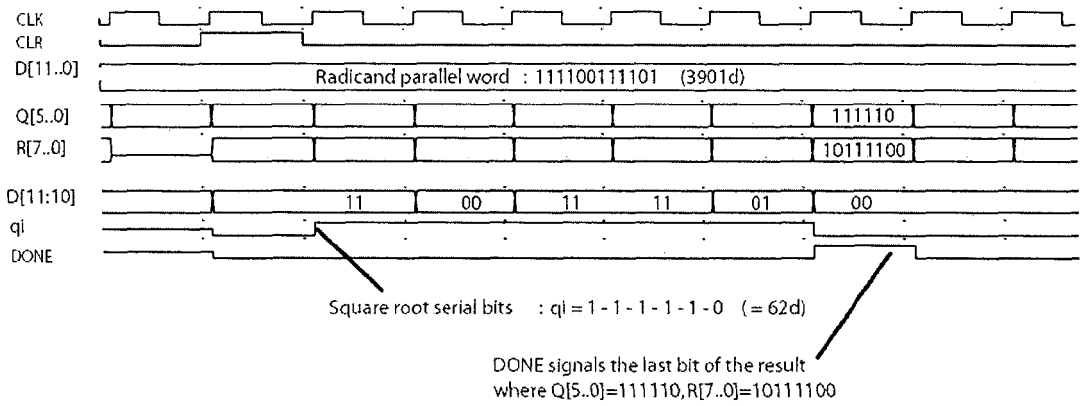


Fig. 2: Optimized FPGA mapping of bit-serial nonrestoring square root

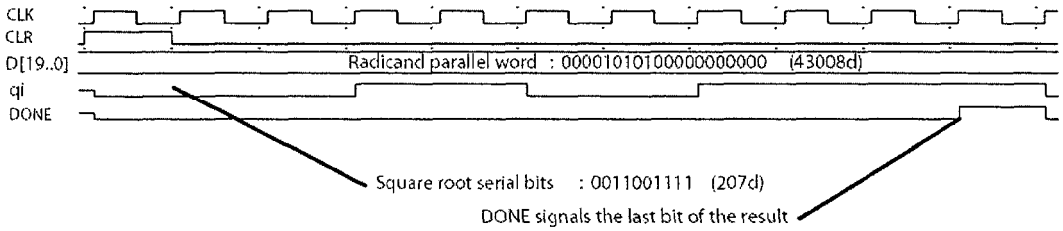


(a) Example for $n = 8 : \sqrt{00110001} = 011 \left(\sqrt{49} = 7\right)$.

Manuscript received January 2007, revised January 2007



(b) Example for $n = 12 : \sqrt{111100001111101} = 111110 \ \left(\sqrt{3901} = 62\right)$.



(c) Example for $n = 20 : \sqrt{00001010100000000000} = 0011001111$.

Note: Depending on the placement of the fixed point, this corresponds to
$\sqrt{43008} = 207, \sqrt{10752} = 103.5, \sqrt{2688} = 51.75, \sqrt{672} = 25.875, \sqrt{168} = 12.9375, \sqrt{42} = 6.46875,$
$\sqrt{10.5} = 3.23875, \sqrt{2.625} = 1.616875$ or $\sqrt{0.65625} = 0.8046975$.

Fig. 3: Functional Simulation Examples for Different Radicand Widths

Manuscript received January 2007, revised January 2007

FPGA implementation of Induction Motor Vector Control using Xilinx System Generator

Jean-Gabriel Mailloux, Stéphane Simard, Rachid Beguenane
Groupe ERMETIS, Département des Sciences appliquées
Université du Québec à Chicoutimi,
555, boulevard de l'université, G7H 5B1 (QC), Canada
rbeguena@uqac.ca

Abstract: - Mechatronic systems are complex interdisciplinary products where many parts are present that overlap each other; electronic control board, electromechanical machines, and power drives. With FPGAs, their control can run faster as multiple operations are executed in parallel. A simulation tool that facilitates the direct translation into hardware of mechatronic control algorithms is desirable. The Xilinx System Generator (XSG), a high-level tool for designing high-performance DSP systems under Simulink, has emerged as an excellent tool for such purpose. In this paper such tool is exercised to implement the conventional vector control algorithm for AC drives, which exhibit a good level of complexity.

Key-Words: - FPGA, System Generator, Induction Motor, Vector Control

1 Introduction

The control of mechatronic systems, a complex interdisciplinary product, can run faster with FPGAs as multiple operations are executed in parallel. It is then highly desirable to have a simulation tool that can easily make the direct translation into hardware of control algorithms with no-knowledge of any Hardware Description Language (HDL). The Xilinx System Generator (XSG), a high-level tool for designing high-performance DSP systems under Simulink environment, has emerged as an excellent tool advisable for such purpose. In fact the XSG takes the graphical algorithmic approach and extends it to FPGA development by using dedicated Simulink Blockset [1]. The knowledge of HDL, necessary to program FPGA devices, is then not required for DSP designers who are not familiar with. This is because the XSG tool can automatically translate the algorithm into FPGA resource and loaded onto the FPGA device. The XSG tool has been used in some areas, such as to cite few; developing hardware-based computer vision algorithms from a system level approach [2], designing a reconfigurable video encryption [3], implementing a wide variety of neural models with the performance of custom analogue circuits or computer clusters [4].

The present work is aiming to show the usefulness of using XSG to prototype complex control algorithms such as the vector control, a well known control strategy for AC drives, particularly those based on induction motors [5]. The authors of paper [6] have already modelled the widely used DTC algorithm where they have used two FPGA boards with the control algorithm implemented on the master board, while the other board controls the power inverter.

The authors of [7] have presented a number of papers on FPGA prototyping of a vector control system using XSG. They mainly developed new IP modules specifically dedicated for vector control design. Moreover, no experimental results were reported to illustrate the profiles of the controlled signals as the design has been tested only in open loop without coupling the vector control design with a virtual induction motor drive.

In this paper the vector control, considered as a special field of digital signal processing exhibiting a complex modularity, is designed using XSG version 8.1. The design will be tested in closed loop with the electromechanical drive being modelled with SimPowerSystems blockset. Section 2 presents briefly the vector control algorithm in order to derive the

corresponding scheme from which the algorithm is implemented using XSG, which is the focal point of section 3. Section 4 discusses the implementation analysis and provides some simulation results while section 5 sketches few conclusions.

2 Vector Control Principle

Vector control is the first method which makes it possible to artificially give certain linearity to the torque control of the induction motor. Precise induction motor control requires the independent control of the stator current components producing the required magnetizing flux and the electromechanical torque, in the same way as with the DC motor.

There are several variants of vector control, among them the rotor flux orientation control scheme as shown in Fig. 1. This is derived from the electromechanical model of the induction motor in the Park reference frame (d, q), known as Park domain. The mathematical description and the corresponding vector control expressions can be found in [8].

3 Vector Control Modeling with XSG

Initially, an algorithm is designed and simulated at the system level with the floating-point Simulink blocksets. A hardware representation for FPGA implementation is then derived using XSG, a plug-in tool to the Simulink modelling software. The later provides a bit-accurate model of FPGA circuits and automatically generates a synthesizable VHDL code for implementation in Xilinx FPGAs. It contains common parameterizable blocks that enable bit-true and cycle-true modelling. For vector control modelling, the blocks used are mostly multipliers, adders, MACs, etc. Some complex arithmetic operators, such as square-root and division used in rotor flux estimation, are embedded from the Xilinx IP core generator. The important feature of the XSG tool is a quick evaluation of the algorithm response when modifying data width and coefficient width of the used variables and parameters. The design and simulation process of the vector control algorithm consists of the following steps:

1. Start by coding separately each module of the vector control algorithm using Xilinx blocks, shown in Fig. 1. An example of PI controller, duplicated 4 times, is illustrated in Fig. 2. This particular instance of vector control requires an external start signal to execute, meaning that the MACs used in the four PIs are also driven by an enable signal so as not to overflow.

2. Once completed, use the Xilinx Gateway blocks to automatically convert floating-point numbers from the Simulink environment into the fixed-point numbers for the XSG environment. There are Gateway-in and Gateway-out blocks to define respectively the inputs and outputs of the vector control fully designed with Xilinx blocks (Fig. 3).

3. Import pre-existing hardware implementations that may not exist in the base XSG blockset. For example, the CORDIC square-root function from the Xilinx IP core generator is useful for rotor flux estimation (Fig. 4). The recent versions of XSG offer these math functions in their Xilinx Reference Blockset.

4. Depending on the bit precision desired for each module and for the system as a whole, the XSG blocks are optimized through the setup of their number of bits and binary point position. Any method may be used to deduce the optimal values, but a tool such as GAPPA [9] proves very helpful in pinpointing the best numbers.

5. Interface the XSG design with any other Simulink resource for simulation, analysis or other purposes. For example, to simulate the vector control in closed loop, Embedded MATLAB functions are used to emulate the PWM firing and gating signals generation while the SimPowerSystems blockset is used to model the power drive and induction motor. The simulated induction motor used in this paper has the following parameters nominal power of 2238 VA, 220 V^{rms}, 5.87 A^{rms}, 2 pairs of poles and 60 Hz frequency.

6. Verify the results using stimuli inputs, for reference speed and load torque, and scopes or

other visual outputs to evaluate the algorithm response.

4 Analysis and Simulation Results

In order to validate the XSG vector control algorithm, an example of one particular test is depicted. Under rotor flux rated value and a slight load, the modelled drive starts from standstill, and then smoothly accelerates to an arbitrary speed of 286 rpm, slows down, reverses, accelerates to −286 rpm, slows down again to standstill and goes back up to 573 rpm. After the first 7 seconds, a very similar speed profile is conducted for another 7 seconds for nominal load. Speed response of the vector control algorithm is shown in Fig. 4, and the corresponding rotor flux profile is illustrated in Fig. 5. As we can see, the speed is well controlled while the rotor flux is kept regulated to its rated value of 1 Wb, which is the objective of vector control.

Once the vector control is tested, hardware blocks can be synthesized and downloaded into a desirable FPGA device. Table 1 shows the implementation results for each developed module when targeting the Virtex-4 device and using an overall precision of 16 bits. We can notice the rotor flux estimator module is the most resource intensive as it requires the use of one square-root and two division operations, all of which are implemented with the CORDIC Xilinx IP core.

5 Conclusion

An FPGA implementation of the conventional vector control algorithm for AC drives has been presented using the XSG tool under Simulink environment. The XSG is very practical since knowledge of any HDL is unnecessary; and it renders algorithm prototyping more accessible and less laborious as it permits an automatic extraction of bit-file targeting commercial FPGA devices.

An equivalent ASIC design could then be easily derived for mass production. The design has been validated by integrating the vector control designed solely with XSG blocks into a closed loop where the PWM firing and gating signal generation, power drive, and induction motor are modelled using standard Simulink blocks and the SimPowerSystems blockset.

Table 1: Vector Control Modules Characteristics (16 bits precision)

Module Name	Resources		
	Slices	*FFs*	*LUTs*
Decoupling	965	995	1759
PIs (All 4 PIs)	703	463	1143
Park Transform	666	1124	1268
Inverse Park	624	1064	1206
Clarke Transform	102	67	148
Rotor Flux Estimator	3066	4763	4505
ω estimator	299	267	459
Total Estimated Ressources	6425	8743	10488

A test has been conducted when the complete AC drive runs under a particular speed profile and physical resources for the FPGA have been estimated.

The massive available number of XSG blocks and dedicated libraries for implementing advanced control algorithms makes it a highly suitable environment for designing and simulating the control systems in modern FPGAs with the advantage of being close to real hardware.

Acknowledgment

This research is funded by a grant from the National Sciences and Engineering Research Council of Canada (NSERC). CMC Microsystems provided development tools and support through the System-on-Chip Research Network (SOCRN) program.

References:

[1] "System Generator for DSP," *Xilinx Inc. 2006*, 10, February 2006. http://www.xilinx.com/ise/optional_prod/system_generator.htm

[2] Ana Toledo Moreo, et al, Experiences on developing computer vision hardware algorithms using Xilinx system generator, *Microprocessors and Microsystems*, vol. 29, Issues 8-9, 1 November 2005, pp. 411-419.

[3] Daniel Denning, et al, Using System Generator To Design A Reconfigurable Video Encryption System, *13th International Conference on Field Programmable Logic and Applications*, Sept.1-3, 2003, Lisbon – Portugal.

[4] Randall K Weinstein and Robert H Lee, Architectures for high-performance FPGA implementations of neural models, *Journal of Neural Engineering*, 3 (2006) 21–34.

[5] Ramon Blasco Giménez., High performance Sensorless Vector Control Induction Motor Drives, *PhD Thesis*, University of Nottingham, UK, December 1995.

[6] Francesco Ricci and Hoang Le-Huy, Modeling and simulation of FPGA-based variable-speed drives using Simulink, *Mathematics and Computers in Simulation*, vol. 63, Issues 3-5, 17 Nov. 2003, pp. 183-195.

[7] József Vásárhelyi, et al, FPGAImplementation Vector Control of Tandem Converter Fed Induction Machine, *6th International Symposium of Hungarian, Researchers on Computational Intelligence*, Nov. 18-19, 2005, Budapest, Magyar.

[8] Rachid Beguenane, et al, Towards the System-on-Chip Realization of a Sensorless Vector Controller with Microsecond-order Computation Time, *19th IEEE Canadian Conference on Electrical and Computer Engineering*, Ottawa, Canada, May 10, 2006.

[9] Florent de Dinechin and Christoph Lauter, Assisted verification of elementary functions using Gappa, *SAC'06*, 2006, Dijon, France.

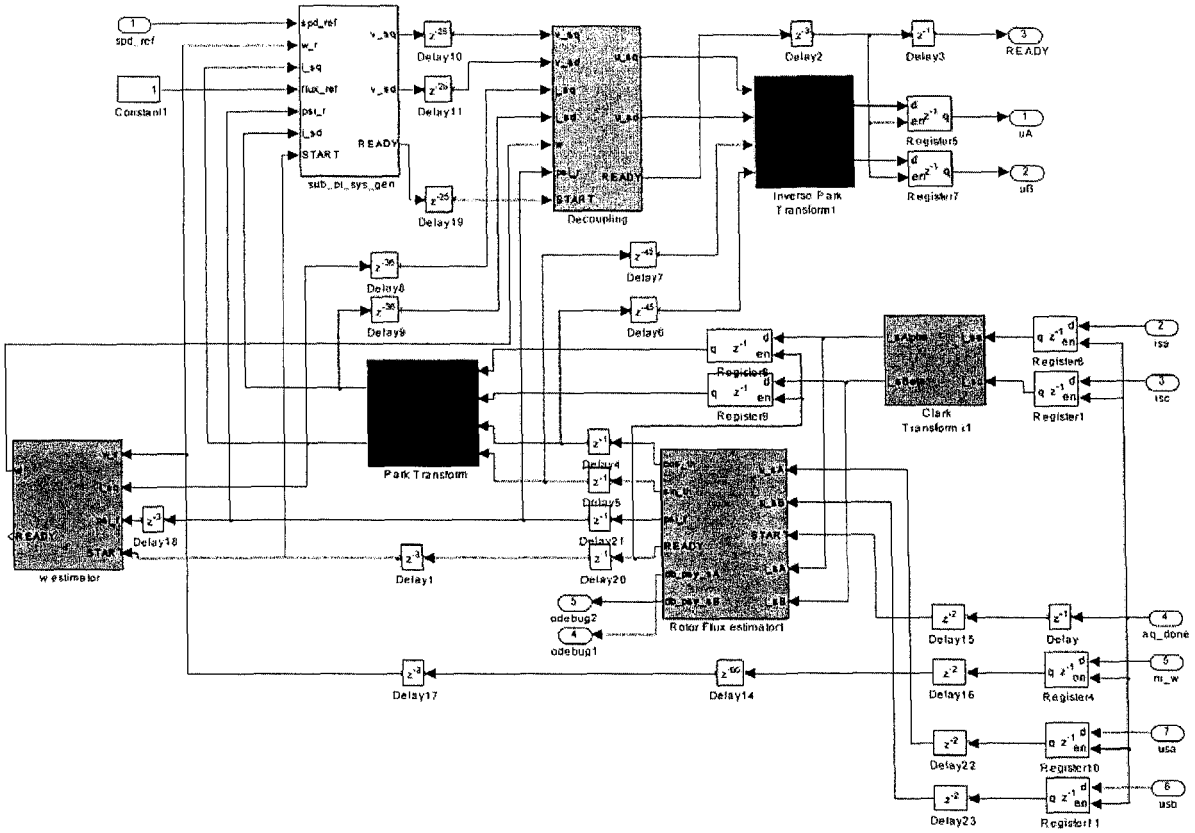


Figure 1. Vector control XSG modules

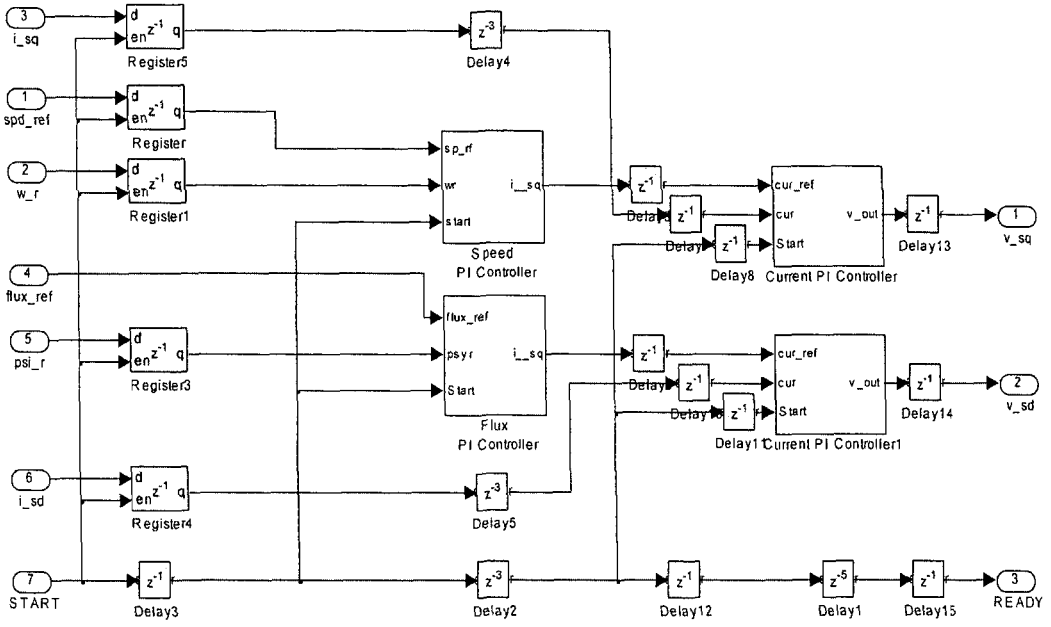


Figure 2. Four PI instances using XSG blocks and a start signal

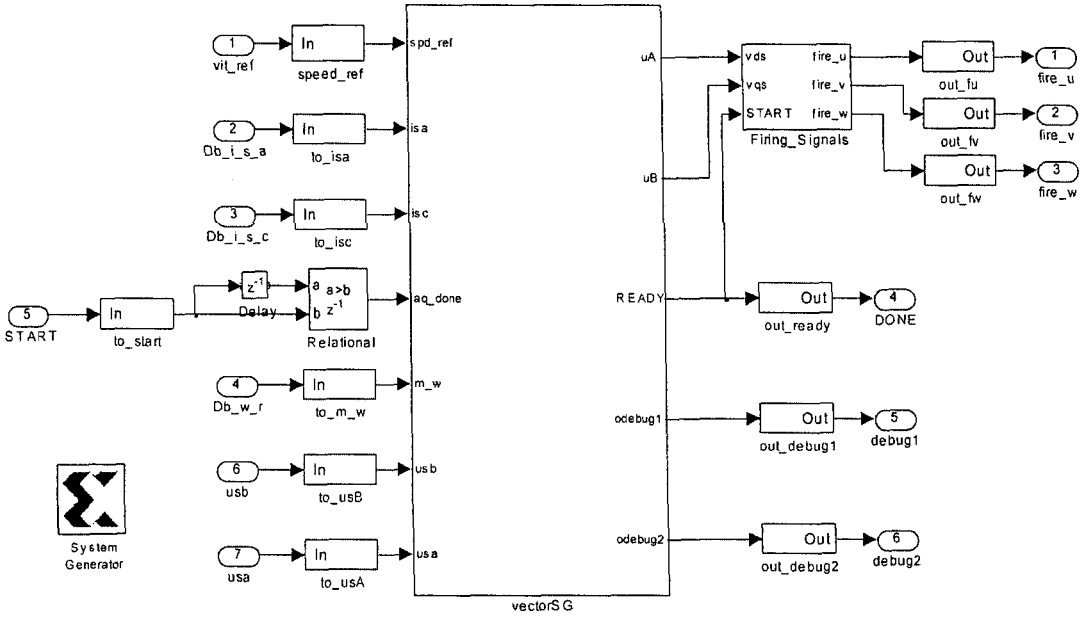


Figure 3. An interface of gateway-in and gateway-out blocks

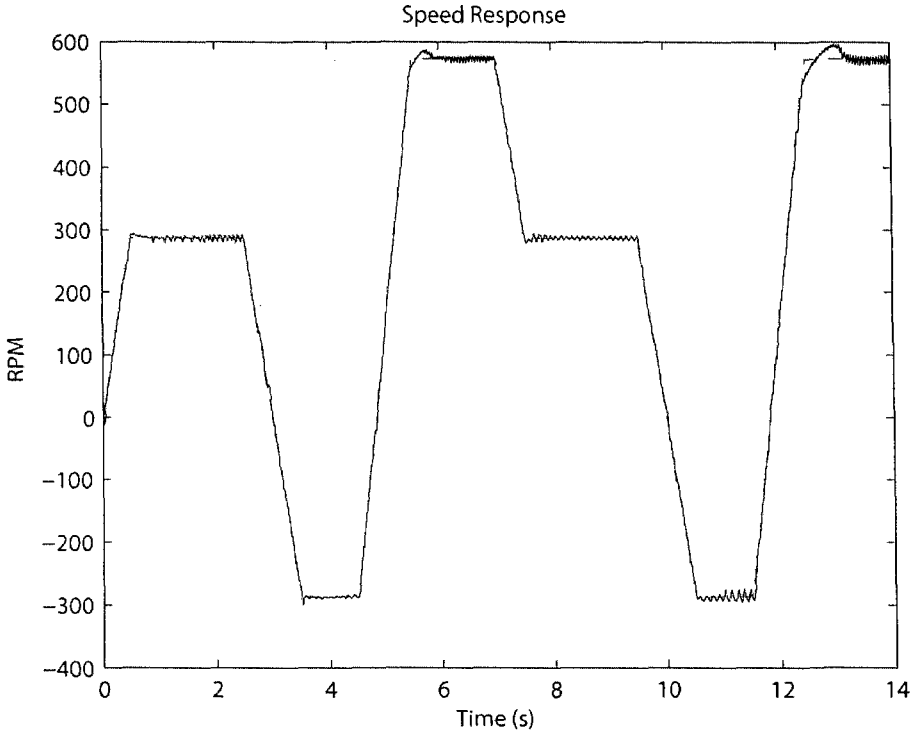


Figure 4. Speed response of the vector control in closed-loop

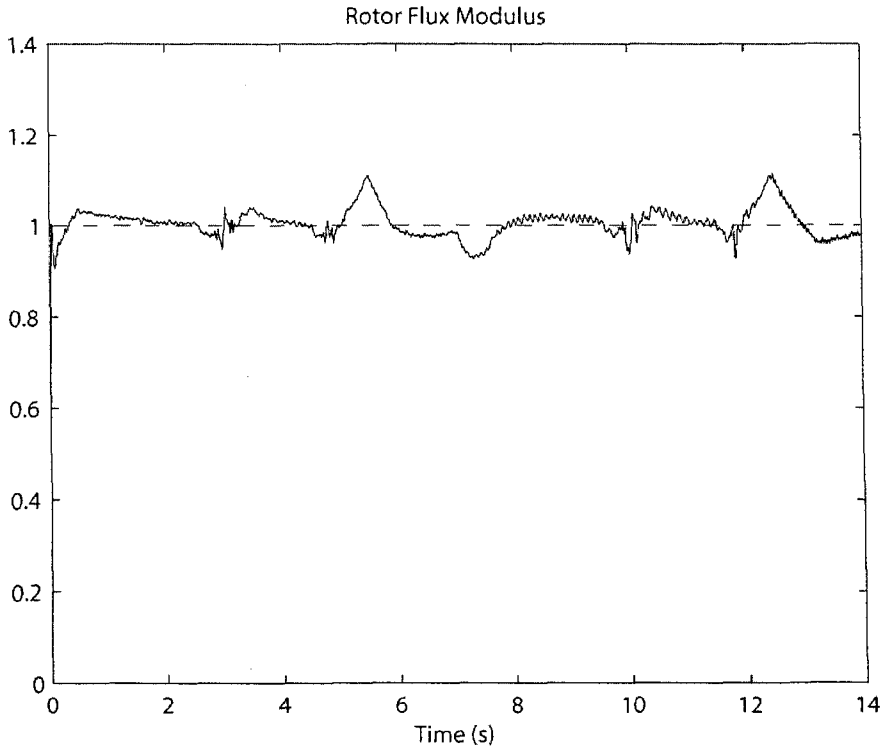


Figure 5. Rotor flux modulus in closed-loop

Rapid Testing of XSG-based Induction Motor Vector Controller using Free-Running Hardware Co-Simulation and SimPowerSystems.

Jean-Gabriel MAILLOUX, Stéphane SIMARD, Rachid BEGUENANE
Groupe ERMETIS, Département des Sciences appliquées
Université du Québec à Chicoutimi
Chicoutimi (QC), G7H 5B1, CANADA
(418) 545-5011 ext. 2268, jean-gabriel_mailloux@uqac.ca

ABSTRACT

The Xilinx System Generator (XSG) Simulink blockset allows for rapid design of control algorithms to be implemented on a FPGA with no VHDL coding and yet much control over the hardware implementation. As a blockset, it benefits from easily linking to other tools offered in the Simulink environment, like the SimPowerSystems (SPS) blockset. When the complexity of a control algorithm increases, the single-step simulation approach of XSG requires a very small fixed-step size in Simulink if used in conjunction with SPS, so that the output of the motor may be processed by the control before the next PWM signal is generated. Using free-running co-simulation, the complications of setting the Simulink simulation parameters for both blocksets are avoided, and only SPS needs to be considered. This paper shows how free-running co-simulation is used to simulate the closed loop vector control system of an asynchronous induction machine, where the control is implemented in XSG and the induction machine and power electronics in SPS. The design is tested and co-simulated using a Gigabit Ethernet connection between the host PC and target FPGA.

Keywords: Xilinx System Generator (XSG), field-programmable gate array (FPGA), free-running co-simulation, vector control, induction motor.

INTRODUCTION

Mechatronic systems are complex interdisciplinary products where many parts are present and overlap each other. With FPGAs, their control can run faster as multiple operations are executed in parallel [1]. However, their development is not easy and an integrated software solution is essential to complete all the steps related to their design, simulation, and optimization.

For a control algorithm to be implemented on a prototyping platform, it is necessary to have a simulation tool that can represent the target hardware with an automatic generation of the necessary code from the algorithm model. This is to avoid the overwhelming task of rewriting the control algorithm for implementation in software languages, like C, or hardware description languages (HDL), like VHDL. The latter offers a speed advantage over its sequential counterpart, but designers are not always familiar with HDL. The concepts of Hardware in the Loop (HIL) and co-simulation using the existing software and hardware tools present an alternative solution. These enable a cost efficient design and verification of modern control laws for future mechatronic systems. Among these tools, the well established the Mathworks Matlab/Simulink and XSG, a dedicated blockset working in the Matlab/Simulink environment, are an ideal mixture to tackle the problems within the algorithm design stage by providing the designer an early proof-of-concept of the hard coded control algorithms. The XSG blockset provides the user with a library of blocks, similar to the traditional Simulink ones, but they represent functions which can be implemented in a FPGA. In order to run advanced and computing-intensive control algorithms, a FPGA-based prototyping platform is then possible with XSG under Simulink environment. This can be used to simulate accurately the hardware and to automatically generate the corresponding VHDL code, with no need to create two different models, one for the simulation and one for the implementation.

Proprietary solutions for HIL simulation have been proposed by some industrial companies. Among these, one can mention the Canadian RTDS Technologies and OPAL-RT Technologies. The first one provides real time simulation tool mainly to test electrical power systems [2], and the second provides HIL simulation services and products from small rapid control prototyping systems to large-scale distributed real-time simulators [3]. In order to compensate for these expensive commercial systems, the authors of [4, 5] propose a cost effective solution called VTB-RT (Virtual Test Bed Real Time) that combines open-source software and low-cost off-the-shelf hardware for power electronics controls. The proposed solution still needs a complex configuration, requiring the setup of additional off-the-shelf hardware and Linux as the OS for VTB-RT. Oftentimes, budget, time or financial constraints do not allow for the implementation of such platforms for using solutions such as VTB-RT.

The focus of the current paper is to present an alternative rapid prototyping platform based only on the well established educational and cost-effective XSG/Simulink environment. To show the feasibility of this approach, a real world design example is used; the conventional vector control drive for induction motors, which exhibits a sufficient degree of complexity and heterogeneity. For safety purposes and to reduce unnecessary turn-around cycles with the real plant to perform hardware in-the-loop simulation, SPS blockset under

Simulink is used to model the electromechanical parts. In our case these parts are the induction machine and its power supply. A graphical environment based on a fixed point XSG dataflow is then used to capture suitable algorithms that are the vector control and PWM gating signal generation. An academic precision evaluation tool called Gappa [6] is used to optimize the silicon logic for cost-efficient FPGA devices. To assess the performances, the results are compared with the same algorithm, previously simulated under Simulink and considered as a reference ideal floating-point algorithm.

Similar works have been done in the past. In their paper [7], the authors have modeled the widely used DTC algorithm. Given the FPGA state of the art at the time when the authors carried out their research, they used two FPGA boards (possibly to target a cost effective FPGA device) with the control algorithm implemented on the master board, while the other board controlled the power inverter and digitized the measures. The authors of [8] have presented a number of papers on FPGA prototyping of a vector control system using XSG. They mainly developed new IP modules specifically dedicated for vector control design without using the co-simulation concept. Moreover, besides estimating hardware resources and execution times for each module block, no experimental results were reported to illustrate the profiles of the controlled signals as the design seems to have only been tested in open loop.

This paper presents a FPGA-based rapid prototyping platform using the free-running co-simulation feature of the XSG/Simulink environment for the design and verification of modular high performance algorithms for mechatronic systems. The complete solution offers an easy-to-use programming interface, and shorter turn-around cycles that make it a realistic rapid prototyping platform, ideal for education and academic research.

In the following section, the reasoning behind the choice of vector control as an application example is explained. Next the design procedure using the proposed method is presented to illustrate the important steps, notably the timing analysis and hardware testing phases. The issues related to timing in both domains; floating-point Simulink and fixed-point XSG, as well as the interaction between them will be fully discussed. Finally an application example using a Vector Control induction drive is presented and the co-simulation results are discussed.

CHOICE OF VECTOR CONTROL

The vector control scheme was used because its complexity provides a good terrain for simulation time comparisons. Many variants of vector control schemes exist, and abundant literature is readily available concerning them [9]. The rotor flux orientation version in its simplest form is used as an application example for the purpose of this paper. The mathematical description and the corresponding vector control expressions can be found in [10]. The presence of the square root operation, used in rotor flux estimation, is especially useful in demonstrating the flexibility of testing different design approaches through free running co-simulation. Said equation for the rotor flux estimator is

$$\Psi_r = \sqrt{\Psi_{r\alpha}^2 + \Psi_{r\beta}^2}$$

where Ψ_r is the rotor flux modulus [10].

To rapidly ensure that the rotor flux estimator equations are well implemented, a rough simulation using the CORDIC square root XSG implementation is run. The advantage of first using this version of square root is that it accepts input even if it has not completed the previous calculation. This means that new data acquisition from the motor can occur even if the vector control is not done processing the previous values. The user can send an array of values to be processed, without waiting for additional steps in between each input.

In the case of the vector control design, it requires 122 steps for completion. The XSG square root operation requires 51 steps before generating a result. If the latter did not accept new values before completion, the user would have to wait at least 51 steps before sending each new value through the vector control algorithm. The simulation time would therefore be 51 times longer. Once it's been established that the equations are correctly represented in the design with the XSG square root block, two problems arise.

First, the XSG CORDIC square root block runs too slowly when setup to obtain a precision great enough for the rotor flux estimation to be accurate. Free-running co-simulation requires a speed of 100 MHz whereas the XSG square root will only run at 58.779 MHz. This slows down the vector control system, which for 122 steps (excluding firing signals generation) would then complete in 2.07 µs. While the XSG CORDIC square root block could run at this speed in actual conditions (because it still makes for a very fast control speed), it cannot be co-simulated using the XSG free-running mode given the 100 MHz requirement. Secondly, the high precision required of the square root operation means that the CORDIC XSG version will reach an impressive size. These two issues render the XSG CORDIC square root approach impractical as opposed to the use of a smaller, more precise square root operation.

The square root operation used in this paper is an iterative implementation of a conventional nonrestoring square root algorithm, tailored to use fast carry logic in FPGAs. While very small in size and precise enough for good rotor flux estimation in 27 steps, it has one disadvantage; it has to complete before accepting a new value. But since vector control will be run at 100 MHz, those 27 steps will complete in 270 ns. This is considerably slower than the sampling rate of our overall closed loop system and thus the vector control will always complete before new data is sent to it. Given that the square root operation is the only part of the vector control system which cannot accept more than one value at a time, this implementation of vector control can support a sampling rate of 3.7 MHz. Using this square root scheme is therefore a perfectly viable option.

This new constraint is what forces the use of different simulation methods. If the user wants to simulate 15 minutes (using CORDIC square root) worth of data with this new square root version, the simulation will last at least 27 times longer, meaning almost 7 hours. At this stage in the design process of the vector control design, this is not acceptable.

CO-SIMULATION FOR DESIGNS WITH MANY STEPS

Single-step co-simulation can improve simulation time when replacing one part of a bigger system. This is especially true when replacing blocks that cannot be natively accelerated by Simulink, like embedded Matlab functions. Replacing said block with an XSG co-simulated design will shift the burden from Matlab to the FPGA and the block no longer remains the simulation's bottleneck.

In a closed loop setting, this performance benefit holds true, but only as long as the replaced block does not require a lot of steps for completion. If the XSG design requires more steps to process the data it is sent than is necessary for the next data to be ready for processing, a costly (time wise) adjustment has to be made. The Simulink period for one simulated FPGA clock (one XSG design step) must be reduced, while the rest of the Simulink system runs at the same speed as before. In a fixed step Simulink simulation environment, this means that the fixed step size must be reduced enough so that the XSG system has ample time to complete between two data acquisition.

In the vector control design chosen for this paper, completion requires 122 steps. For a sampling rate of 32 kHz, the generation of a PWM signal is divided into 100 sections (two zones divided by 50) which is a good compromise between precision and simulation time. The simulation fixed-step size is then 625 ns which is already small enough to hinder the performance of simulating the SPS model. Since the PWM signals generation is divided in two zones, for every 50 steps of Simulink operations (PWM signal generation and SPS model simulation), the 122 steps of vector control must complete. The XSG Simulink System Period must be adjusted for the XSG model to run 2.44 times faster than the other Simulink components. The simulation fixed-step size becomes 2.56 ns, thus prolonging simulation time, later shown in table 1. In other words, since the SPS model and PWM signals generation take little time (in terms of steps) to complete whereas the vector control scheme requires numerous steps, the coupling of the two forces the use of a very small simulation fixed–step size.

Free-running co-simulation means that the FPGA will always be running at full speed. Simulink will no longer be dictating the speed of a XSG step as was the case in single-step co-simulation. With the Virtex-4 ML402 SX XtremeDSP Evaluation Platform, that step will now always be 10 ns. Therefore, even a very complicated system requiring many steps for completion should have ample time to process its data before the rest of the Simulink system does its work. Nevertheless, a synchronization mechanism should always be used for linking the free-running co-simulation block with the rest of the design. In this paper, a simple yet robust delay method, shown in figure 1, has been used to ensure an exterior start signal will not be mistakenly interpreted as more than one start pulse.

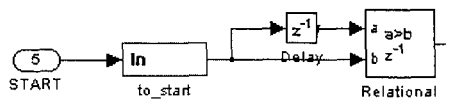


Figure 1 – Delay method of synchronization

TIMING ANALYSIS

No matter then chosen hardware co-simulation method (free-running or single-step), the user must go through two steps that are always necessary when moving the design from Simulink to an actual FPGA: timing analysis and hardware testing.

The XSG co-simulation feature allows the user to run a design on the FPGA found on a certain platform. While it is possible for the user to create new targets for testing, this is not the subject of this paper. Therefore, the target used for the example application herein is one of two preprogrammed targets in XSG, the Virtex-4 ML402 SX XtremeDSP Evaluation Platform illustrated in figure 2. Before actually generating a BIT file to reconfigure the XC4VSX35-FF668-10C which is found on that board, it must meet a certain speed requirement. Whether the co-simulation is done through JTAG or ETHERNET, the design must be able to run at 10ns (100 MHz).

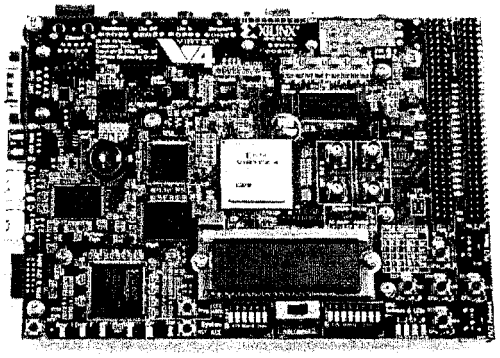


Figure 2 – Virtex-4 ML402 SX XtremeDSP Evaluation Platform

As long as the design is running inside Simulink, there are never any issues with meeting timing requirements for the XSG model. The idea often is to get a prototype working as quickly as possible, and thus the user is mostly concerned with the algorithm. At this point in the design, numerous adjustments and changes will likely occur, and having a lot of delay blocks in the way will only serve to slow down development. Once completed, the design will need be simulated on the FPGA, which means that the synthesis will occur. If the user launches the co-simulation block generation, the timing errors will be mentioned quite far into the process. This means that, after waiting for a relatively long amount of time (sometimes 20-30 minutes depending on the complexity of a design and the speed of the host computer), the user notices the failure to meet timing requirements with no extra information to quickly identify the problem. This is why the timing analysis tool must always be run *prior* to co-simulation. While it might seem a bit time-consuming, this tool will not simply tell you that your design does not meet requirements, but it will tell you where the longest delays are found. It will even mark in red the paths in Simulink where constraints are not met. With the help of this tool, the user can add delays and adjust latencies until a speed of 100 MHz is achieved so that the hardware testing may begin.

HARDWARE TESTING

The co-simulation option provided by XSG also ensures that our design will respond correctly once implemented in hardware. Once a prototype has been sufficiently tested in the Simulink environment and is ported to and FPGA platform, a tool such a chipscope is often used to test and debug the design. It allows the user to inspect the signals inside the FPGA to identify problems or to measure the accuracy of the implementation. While this process is very useful for fine tuning a prototype, it is somewhat wasteful time wise when eradicating glitches and bugs in design phases.

Because co-simulation loads the actual design onto the FPGA, the behavior of the design on the chip and running at full speed is immediately tested. If a glitch appears that was not present when doing Simulink simulations, the user needs only add more outputs to pinpoint the source of the problem (instead of using chipscope or similar tools). Should the user need analyze a great many output signals for debugging purposes or otherwise, it is faster to choose the ETHERNET method of co-simulation since it offers a larger bandwidth [11]. For the duration of the design period, it is a good idea to keep a few debug output signals to be used when trying to identify the source of a problem. Having them there means that the designer needs only change the internal routing of signals within the XSG design, whereas the the interface with other subsystems inside Simulink is left unchanged (figure 3). In the case of the vector control implementation of this paper, this technique allowed for the discovery of a glitch concerning overflow handling of the XSG subtractor block. While there were no errors when running the simulation in Simulink, the co-simulated block proved erratic when choosing the saturation method instead of wrap(i.e., to discard bits to the left of the most significant representable bit). Once all major glitches have been fixed, the results of closed loop simulation can be analyzed.

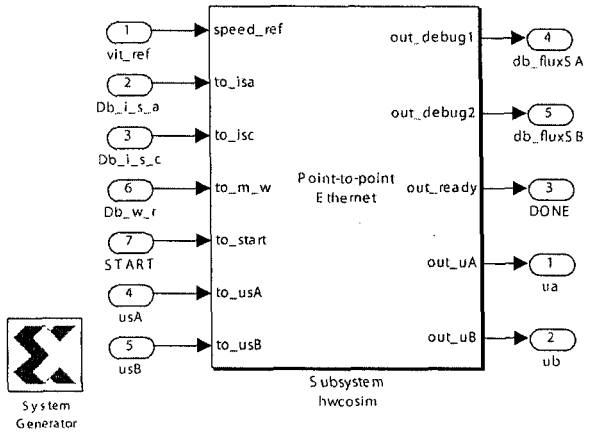


Figure 3 – Debug outputs for co-simulation

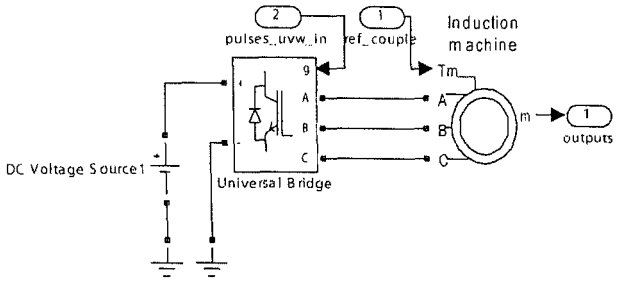


Figure 4 – SPS plant

TESTING IN CLOSED LOOP

When testing and debugging a control system made with XSG, one of the challenges is to test the design in a closed loop setting. While an open loop simulation using a theoretical model (be it in Simulink or Matlab) can provide an array of input and output data to compare against the XSG design's results, that method has two disadvantages. It does not fully represent how well the system will react in a closed loop environment, and it doesn't change the fact that closed-loop testing will still have to be conducted later on.

Testing of the control algorithm in closed loop is facilitated by the use of the SPS blockset which simulates both the motor and power electronics (figure 4). PWM signal generation is done through the use of two embedded Matlab function blocks to show the flexibility of coupling XSG co-simulated blocks with various Simulink tools (figure 6).

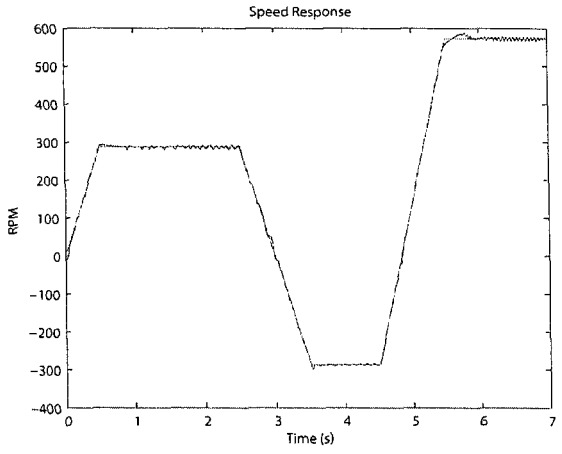


Figure 5 – Speed Response of the complete system

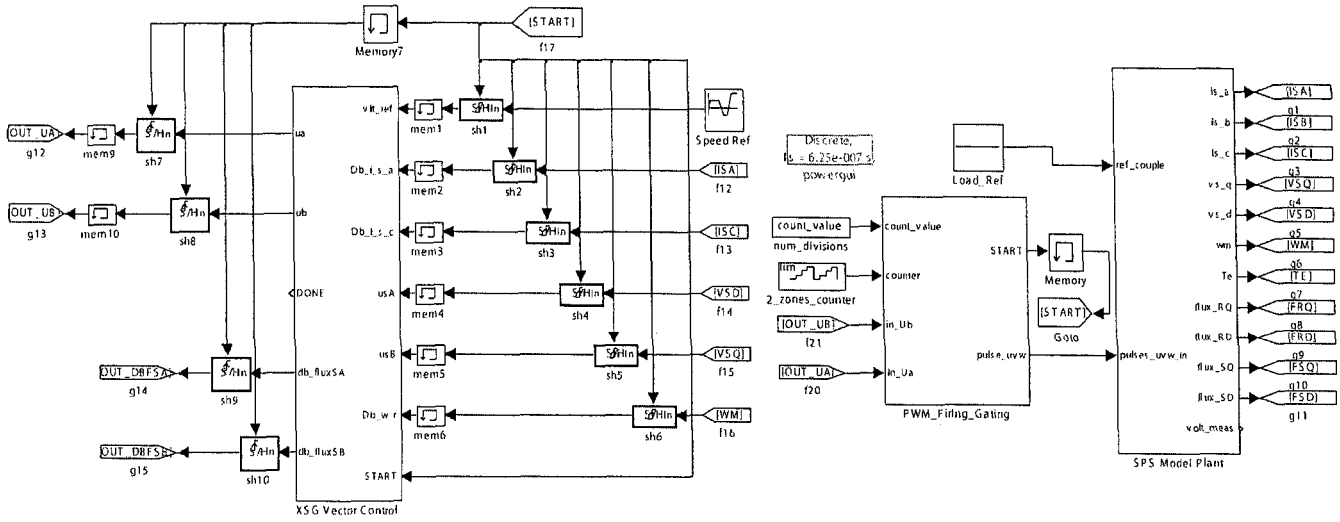


Figure 6 – Complete System with co-simulated vector control, PWM signal generation and SPS plant

The results shown in figure 5 represent the system response when using the co-simulated vector control block in response to six quick changes in speed reference under slight load.

Type of simulation	Simulation Time
Free-running co-simulation	1734 s
Single Step co-simulation	174610 s (48 hours)
Simulink Vector Control	763 s

Table 1 - Simulation Times and Methods

Table 1 shows the decrease of simulation time afforded by the free-running vector control. The tests were conducted on a standard personal computer with an Intel P4 3.40Ghz CPU. For the same precision in results (50 divisions per zone) and the same amount of data to be simulated (speed variations over a period of 7 seconds), a single-step approach would require 100.7 times longer to complete, thus being an ineffective approach.

A third test is conducted where the PWM signal generation block is connected to the SPS model but the vector control is done entirely in Simulink blocks. The increase in simulation time using the XSG model as opposed to the non real-time Simulink model is only 2.27 times.

CONCLUSION

This paper explains how free-running co-simulation in XSG can be used to test and debug a XSG model in order to have a working prototype with reasonable performance which can then be fine tuned and further developed. While commercial tools like RT-LAB grant the means for real-time analysis of control designs and greatly facilitate fine-tuning of these systems, implementing such solutions during the design phase is not always practical.

Before testing a design against real physical constraints, like connecting it to a real motor, it is simulated so as not to damage the equipment (often a costly ordeal). But moving from a XSG-SPS design to a real-time simulation platform involves a lot of time for synthesis, conversions and other miscellaneous processes. This time is irrelevant when compared to the time saved in running long simulations of a working prototype. But when the user is still debugging a design, a lot of modifications are required to fix the various bugs and glitches, get the desired response, etc. What this means practically is that there is a lot of back and forth between design and simulation. Therefore, if it is possible to reduce the time involved in getting ready for a simulation, the user can quickly get a working prototype out of a XSG design that can be tested in real conditions. With the help of SPS to create a model of the plant in a system, free-running co-simulation using the same host computer used for development of the model and a target FPGA allows for rapid testing of a control algorithm without investing time and resources in setting up third party solutions for simulation acceleration.

ACKNOWLEDGMENTS

This research is funded by a grant from the National Sciences and Engineering Research Council of Canada (NSERC). CMC Microsystems provided development tools and support through the System-on-Chip Research Network (SOCRN) program.

REFERENCES

[1] E. Monmasson and Y. A. Chapuis, "Contributions of FPGAs to the Control of Electrical Systems - a Review," **IEEE Electronics Society Newsletter**, Vol. 49, No. 4, 2002.

[2] Kuffel, R.; Giesbrecht, J.; Maguire, T.; Wierckx, R.P.; McLaren, P., "RTDS - a fully digital power system simulator operating in real time", **Energy Management and Power Delivery**, 1995. Proceedings of EMPD apos;95., 1995 International Conference on Volume 2, Issue , 21-23 Nov 1995 Page(s):498 - 503 vol.2.

[3] Dufour, Christian; Abourida, Simon; Belanger, Jean; Lapointe, Vincent, **Real-Time Simulation of Permanent Magnet Motor Drive on FPGA Chip for High-Bandwidth Controller Tests and Validation,** Page(s): 4581-4586, IECON.2006.

[4] Bin Lu; Monti, A.; Dougal, R.A., "Real-time hardware-in-the-loop testing during design of power electronics controls", Industrial Electronics Society, 2003. IECON apos;03. **The 29th Annual Conference of the IEEE,** Volume 2, Issue , 2-6 Nov. 2003 Page(s): 1840 - 1845 Vol.2 IECON.2003. Roanoke (VA)(USA).

[5] Monti A., E. Santi, R.Dougal, M. Riva, "Rapid Prototyping of Digital Controls for Power Electronics", **IEEE Trans. On Power Electronics**, May 2003 Issue.

[6] **Assisted verification of elementary functions using Gappa**, Florent de Dinechin, Christoph Lauter, SAC'06 2006, Dijon, France.

[7] **Modeling and simulation of FPGA-based variable-speed drives using Simulink Mathematics and Computers in Simulation**, Volume 63, Issues 3-5, 17 November 2003, Pages 183-195 Francesco Ricci and Hoang Le-Huy.

[8] József Vásárhelyi, Mária Imecs, Csaba Szabó, Ioan Iov Incze, "FPGA Implementation Vector Control of Tandem Converter Fed Induction Machine," **6th International Symposium of Hungarian Researchers on Computational Intelligence**, Magyar Kutat'ok 6, Nemzetk¨ozi Szimp'oziuma, November 18-19, 2005, Budapest.

[9] **High performance Sensorless Vector Control Induction Motor Drives**, PhD Thesis, University of Nottingham, England, December 1995 by Ramon Blasco Giménez.

[10] Rachid Beguenane, Jean-Gabriel Mailloux, St'ephane Simard, and Arnaud Tisserand, "Towards the System-on-Chip Realization of a Sensorless Vector Controller with Microsecond-order Computation Time", **19th IEEE Canadian Conference on Electrical and Computer Engineering**, Ottawa, Canada, May 10, 2006

[11] Ben Chan, Nabeel Shirazi, Jonathan Ballagh, **"Achieving High-Bandwidth DSP Simulations Using Ethernet Hardware-in-the-Loop"**, DSP magazine, May 2006

Towards the System-on-Chip Realization of a Sensorless Vector Controller with Microsecond-order Computation Time

Rachid Beguenane*, Jean-Gabriel Mailloux*, Stéphane Simard*, and Arnaud Tisserand†
*Groupe ERMETIS, Département des Sciences appliquées
Université du Québec à Chicoutimi
Chicoutimi (QC), G7H 2B1, CANADA
rbeguena@uqac.ca, jean-gabriel_mailloux@uqac.ca, s.simard@ieee.org
† LIRMM, CNRS-Univ. Montpellier II
161 rue Ada. F-34392 Montpellier, FRANCE
arnaud.tisserand@lirmm.fr

Abstract— **The aim of this research is to implement sensorless vector control algorithms on a single, eventually reconfigurable, chip, with a computation timing constraint of, at most, 1-6 microseconds, and a concern for implementation cost. In this article, we discuss the implementation problems and tradeoffs involved in meeting these goals on Field-Programmable Gate Arrays (FPGAs). To be able to fit a complete induction motor vector controller on a single, inexpensive FPGA chip, we estimate the area/time requirements of each module involved in sensorless vector control. We discuss, in particular, the tradeoffs of implementing the key modules, the speed and flux observers and the Clarke and Park transformations. The speed and flux observers here under consideration are extended Kalman filter-based.**

I. Introduction

Technological progress depends more and more on miniaturization and increasingly fast and powerful computing systems, and we witness a true revolution in microelectronics. In almost every field, significant progress now greatly depends on advances in nanotechnologies and their applications. The design of lighter, less cumbersome, and more economic application-specific computing processors becomes required in order to reach increasingly demanding performances.

The very fast evolution of CMOS integrated circuit fabrication technologies already makes it possible to design complete digital systems integrated on the same chip.

The reconfigurable chips known as FPGAs (Field-Programmable Gate Arrays) currently on the market are manufactured at a level of integration of around 90 nanometers or less, and usually comprise several millions of gates on the same chip. They offer considerable advantages for accelerating the time to market, and reduce the development and production costs.

In field of electric motor control, with which we are concerned in this research, the real-time computing capacity of traditional approaches using PC computers and off-the-shelf digital signal processors (DSPs), is largely superseded. One must now turn to nanotechnologies in order to obtain adequate hardware acceleration. FPGAs, light and relatively inexpensive, are usually more powerful than traditional devices, and appear therefore ideal for implementing real-time control systems without involving the considerable costs traditionally related to the design and fabrication of application-specific integrated circuits (ASIC).

The semiconductor industry is trying to design digital signal controllers (DSCs) having a computing time of only a few microseconds for the precise and robust control of electric motors. It would actually be possible of increasing the current operating efficiency of electric motors, now in the order of approximately 40-60%, up to 90%.

The high cost and complexity of the required electronics always constituted a significant impediment to the implementation of complex algorithms within a dynamic of only a few microseconds. The DSC technology, traditionally composed of DSPs coupled to a microcontroller unit or a microprocessor, is presently reaching its physical limits, with a minimal computing time about 6 microseconds for the most minimalistic implementation of vector control using a speed sensor. The challenge to which we attack ourselves here is to realize on a single chip, without using a speed sensor, and in an even shorter lapse of time, the most sophisticated vector control algorithms, where the speed and the flux will not be measured, but estimated by hardware. We will see the complexity of these estimates in the following discussion.

II. Induction Motor Vector Contol

The characteristics of the induction motor are basically nonlinear. Vector control, also called *flux directed control*, is the first method which makes it possible to artificially give a certain linearity to the torque control of the induction motor. Speed sensors are however necessited, in general, for the implementation of vector control. This does not pose any problem as long as the induction motor is used for regular motion control using a position or speed encoder, but whenever

1-4244-0038-4 2006 IEEE
CCECE/CCGEI, Ottawa, May 2006

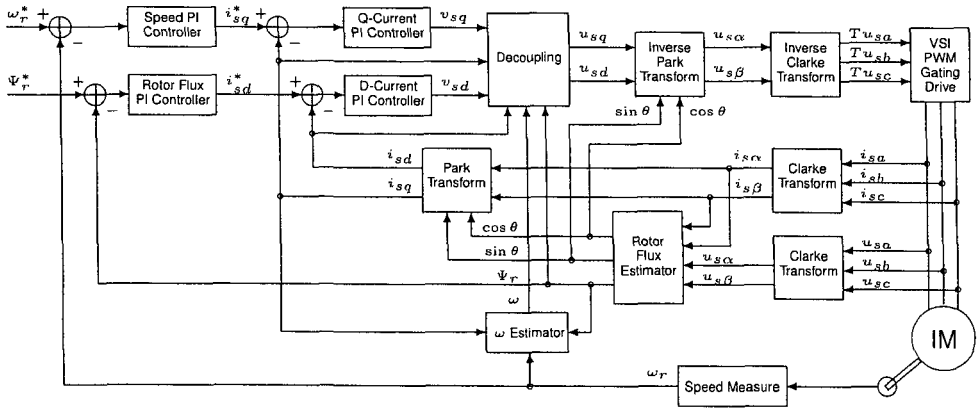


Fig. 1. IM Vector Control Scheme

it is impossible to connect a sensor to the motor shaft, the implementation proves difficult. It is then necessary to carry out vector control without using a speed sensor, with all the difficulties that that poses.

Precise induction motor control requires the independent control of the components of its input current producing the field and the couple, as it is the case with the DC motor. The only theoretical solution which makes it possible to realize such an independence of control consists in breaking up the stator current into its components in a domain or a suitable frame of reference. By taking a synchronously revolving frame of reference, with the space vector of rotor flux as phasor of reference, it is mathematically possible to separate the stator current in two independent components i_{sd} and i_{sq}, respectively controlling the field and the torque of the motor. Decoupling between the effects of the components of the current then makes it possible to simplify the control of the mechanical variables of the drive, and to impose on the motor fast variations of the rotor flux at starting and at constant power.

A. Induction Motor Model in Park Domain

The electromechanical model of the induction motor in the Park reference frame (d, q), known as Park domain, synchronously revolving at speed ω, is stated as follows:

$$u_{sd} = R_s i_{sd} + \sigma L_s \frac{\mathrm{d}}{\mathrm{d}t} i_{sd} \underbrace{-\sigma L_s \omega i_{sq} + \frac{M}{L_r} \frac{\mathrm{d}}{\mathrm{d}t} \Psi_r}_{D_d} \quad (1)$$

$$u_{sq} = R_s i_{sq} + \sigma L_s \frac{\mathrm{d}}{\mathrm{d}t} i_{sq} \underbrace{+\sigma L_s \omega i_{sd} + \frac{M}{L_r} \omega \Psi_r}_{D_q} \quad (2)$$

$$\frac{\mathrm{d}}{\mathrm{d}t} \Psi_r = -\beta_r \Psi_r + M \beta_r i_{sd} \; ; \quad \beta_r = \frac{R_r}{L_r} \quad (3)$$

$$\omega = P_p \omega_r + \frac{M \beta_r}{\Psi_r} i_{sq} \quad (4)$$

$$\frac{\mathrm{d}\omega_r}{\mathrm{d}t} = \frac{3}{2} P_p \frac{M}{J L_r} \Psi_r i_{sq} - \frac{f}{J} \omega_r - \frac{T_l}{J} \quad (5)$$

with

u_{sd}, u_{sq}	Stator voltage of d-axis and q-axis
i_{sd}, i_{sq}	Stator current of d-axis and q-axis
Ψ_r	Rotor flux modulus
ω	Angular speed of the (d, q) reference frame
L_s, L_r	Stator and rotor inductances
M	Mutual inductance
R_s, R_r	Stator and rotor resistances
σ	Leakage coefficient of the motor
β_r	Constant: L_r / R_r
P_p	Number of pole pairs
ω_r	Rotor speed, or *angular frequency* (measured or estimated)
J	Inertial momentum
f	Friction coefficient
T_l	Torque load

B. System Block Diagram

In the following, we develop the mathematical expressions for the blocs in Fig. 1. The starred (*) variables are the input references for the PI controllers. In general, ϵ denotes the error signal, and k_p, k_i, the PI controller parameters. The α and β subscripts denote the components of the corresponding variables in the stationnary (α, β) reference frame. The a, b, and c subscripts denote the components of the corresponding variables in the stationnary (a, b, c) reference frame. θ is the angular position of the rotor flux vector, with $\theta = \int \omega \, \mathrm{d}t$.

a) Speed PI Controller:

$$i_{sq}^* = k_{p_v} \epsilon_v + k_{i_v} \int \epsilon_v \, \mathrm{d}t \; ; \quad \epsilon_v = \omega_r^* - \omega_r$$

b) Rotor Flux PI Controller:

$$i_{sd}^* = k_{p_f} \epsilon_f + k_{i_f} \int \epsilon_f \, \mathrm{d}t \; ; \quad \epsilon_f = \Psi_r^* - \Psi_r$$

c) Rotor Flux Estimator:

$$\Psi_r = \sqrt{\Psi_{r\alpha}^2 + \Psi_{r\beta}^2}$$

$$\cos\theta = \frac{\Psi_{r\alpha}}{\Psi_r} \; ; \quad \sin\theta = \frac{\Psi_{r\beta}}{\Psi_r}$$

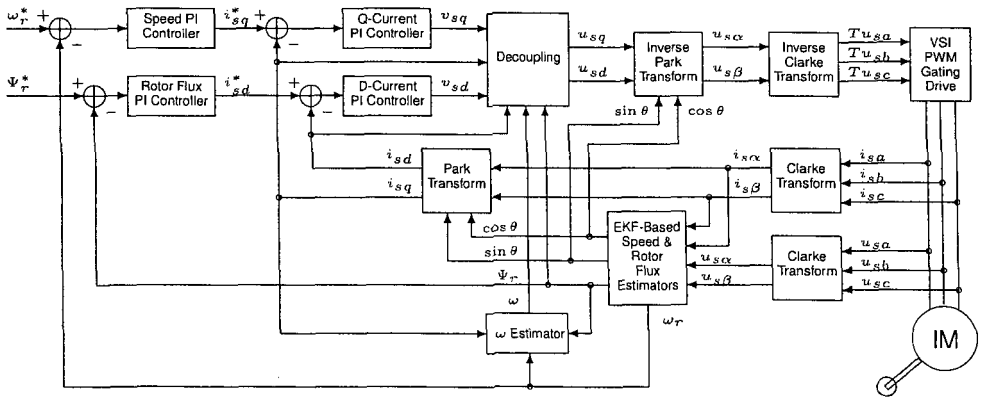


Fig. 2. EKF-based Speed-sensorless IM Vector Control Scheme

with

$$\Psi_{r\alpha} = \frac{L_r}{M}\left(\Psi_{s\alpha} - \sigma L_s i_{s\alpha}\right)$$

$$\Psi_{r\beta} = \frac{L_r}{M}\left(\Psi_{s\beta} - \sigma L_s i_{s\beta}\right)$$

$$\Psi_{s\alpha} = \int \left(u_{s\alpha} - R_s i_{s\alpha}\right)$$

$$\Psi_{s\beta} = \int \left(u_{s\beta} - R_s i_{s\beta}\right)$$

d) Current PI Controller:

$$v_{sd} = k_{p_i}\epsilon_{i_{sd}} + k_{i_i}\int \epsilon_{i_{sd}}\ \mathrm{d}t; \quad \epsilon_{i_{sd}} = 1^*_{sd} - i_{sd}$$

$$v_{sq} = k_{p_i}\epsilon_{i_{sq}} + k_{i_i}\int \epsilon_{i_{sq}}\ \mathrm{d}t; \quad \epsilon_{i_{sq}} = 1^*_{sq} - i_{sq}$$

e) Decoupling:

$$u_{sd} = \sigma L_s v_{sd} + D_d\ ; \quad u_{sq} = \sigma L_s v_{sq} + D_q$$

with

$$D_d = -\sigma L_s \omega i_{sq} + \frac{M}{L_r}\frac{\mathrm{d}}{\mathrm{d}t}\Psi_r$$

$$D_q = +\sigma L_s \omega i_{sd} + \frac{M}{L_r}\omega\Psi_r$$

f) Omega (ω) Estimator:

$$\omega = P_p\omega_r + \frac{M\beta_r}{\Psi_r} i_{sq}$$

g) Clarke Transformations:

$$i_{s\alpha} = i_{sa}$$

$$i_{s\beta} = \frac{1}{\sqrt{3}} i_{sa} + \frac{2}{\sqrt{3}} i_{sb}$$

and

$$u_{sa} = u_{s\alpha}$$

$$u_{sb} = -\frac{1}{2}u_{s\alpha} + \frac{\sqrt{3}}{2}u_{s\beta}$$

$$u_{sc} = -\frac{1}{2}u_{s\alpha} - \frac{\sqrt{3}}{2}u_{s\beta}$$

h) Park Transformation:

$$\begin{pmatrix} i_{s_d} \\ i_{s_q} \end{pmatrix} = PT \begin{pmatrix} i_{s_\alpha} \\ i_{s_\beta} \end{pmatrix}$$

with

$$PT = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

i) Inverse Park Transformation:

$$\begin{pmatrix} u_{s_\alpha} \\ u_{s_\beta} \end{pmatrix} = \mathrm{PT}^{-1} \begin{pmatrix} u_{s_d} \\ u_{s_q} \end{pmatrix}$$

with

$$\mathrm{PT}^{-1} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

When it is necessary to carry out vector control without using a speed sensor, the speed can be calculated from the values of the current and voltage of an AC motor. Open-loop solutions give a certain speed estimate, but inherently bear a large error. For better results, it is necessary to design an estimator or a filter. The Kalman filter has a good dynamic behavior, a good resistance to perturbations, and it can function at stand still. Designing a filter for an AC motor remains however a very complex problem which requires the calculation of the motor model in real time. Moreover, it is necessary to calculate the filter equations, which normally implies several matrix multiplications as well as one matrix inversion. These requirements can nevertheless be satisfied by a high-performance computing processor. Fig. 2 shows the speed-sensorless induction motor vector control scheme including the speed and rotor flux estimator based on an extended kalman filter (EKF). The tradeoffs involved in the implementation of this estimator constitute the core of the present research and require in-depth analysis and careful considerations which will make the object of a subsequent study.

III. Implementation Analysis

The basic IM vector control scheme (Fig.1) comprises 24 multiplications (including multiplication by a constant and squaring), 3 divisions, and only one square root operation. We further observe that the division and square root operators are localized in close mutual coupling inside the rotor flux and ω estimators. Because of the fact that $\cos\theta = \Psi_{r\alpha}/\Psi_r$ and $\sin\theta = \Psi_{r\beta}/\Psi_r$, no actual sine or cosine computation is involved.

Most modern FPGAs embed several tens, even up to a couple hundreds, of small, ultra-fast, 18x18 VLSI multipliers, which can readily be used in signal processing applications. All multiplications involved in an implementation of the IM vector control scheme can efficiently be implemented using these embedded multipliers.

The division and square root operators, thanks to their small number and the modest operand width required, can efficiently be implemented by traditional hardware modules without being overcostly in area. An extensive comparative study of divider implementations on FPGAs, including all kinds of restoring, non-restoring, and SRT dividers has been presented in [1]. It is even possible to implement dividers based on the small embedded 18x18 multiplier blocks [3], [4].

The matrix operations involved in implementing the EKF-based estimator are outside the scope of the present article.

We analysed the system of Fig.1 using the Xilinx blockset in Simulink. This analysis revealed that the required internal precision would be of at least 32 bits. We will show in the next section the results of a straightforward FPGA implementation of this design using System Generator.

IV. Area and Time Estimates

We did a worst case analysis of the system implementation as a network of on-line modules following the methodology proposed in [2]. Table II shows our area cost estimates for a 16-bit on-line implementation. The arithmetic modules implementation data presented in Table I have been taken from [2] for the on-line (ol-XXX) modules and SRT-DIV, while NR-SQRT is our home implementation of a non-restoring parallel-sequential square root. We see from Table III that the on-line delay on the bottleneck path of the system is about 116 clock cycles. One lap of the whole control loop therefore takes about 164 clock cycles to complete, taking into account the required conversions back and forth between standard binary and redundant number representations. Estimating that this on-line design could be clocked at 100 MHz on a Virtex-II FPGA, its total computation time would therefore be of around 1.6 microseconds.

For a crude, but rather convincing, comparision with a 32-bit parallel arithmetic implementation, the synthesis results of the VHDL code generated using System Generator, targetting

TABLE I

16-bit Arithmetic Modules Implementation Results (mostly from [2])

Module	δ	CLB	LUT	FF	Freq. (MHz)
ol-ADD	2	3	4	5	-
ol-cMUL	1	14	24	18	89
ol-cMAC	2	16	27	19	86
ol-MUL	4	93	149	139	79
ol-Div	5	115	187	122	57
SRT-DIV	N/A	36	58	34	78
NR-SQRT	N/A	25	28	43	–

TABLE II

16-bit On-line Implementation Area Estimates

	LUT	FF
4 Error Differences	4	5
Speed PI Controller	55	65
Rotor Flux PI Controller	55	65
Q-Current PI Controller	55	65
D-Current PI Controller	55	65
Decoupling	584	540
Park Transform	604	566
Inverse Park Transform	604	566
Clarke Transform 1	53	42
Clarke Transform 2	53	42
Inverse Clarke Transform	58	48
Rotor Flux Estimator	1039	791
ω Estimator	241	165
TOTAL	3460	3025

TABLE III

Estimated On-line Delay of the Bottleneck Path

Module	ol-delay
Clarke Transformation	19
Rotor Flux Estimator	38
Park Transformation	23
ω Estimator	36
TOTAL	116

TABLE IV

System Generator Synthesis Results for Virtex II xc2v2000-4

Slices	6484
Flip-flops	4700
LUTs	11086
MULT18x18	50

TABLE V

Number of Clock Cycles on the Bottleneck Path for the System Generator Design

Module	N Cycles Variant A	N Cycles Variant B
Clarke Transformation	3	3
Rotor Flux Estimator	77	15
Park Transformation	6	6
ω Estimator	34	3
TOTAL	120	27

a Virtex II xc2c2000-4 FPGA, are presented in Table IV. The dividers and square root operators used in this design are all sequential, based on the non-restoring algorithm. On lap of the complete control loop of an implementation of this design using the minimum possible number of registers takes 131 clock cycles at a maximum frequency of 50 MHz. The total computation time is therefore of about 2,5 microseconds.

In addition, two variants of this design have been evaluated where each arithmetic module outputs are registered. The first, Variant A, uses sequential dividers and square root operators, while the second, Variant B, uses a 1-clock version of the same operators. Variant A can be clocked up to around 100 MHz and takes 120 clock cycles to complete one lap of the whole control loop. The maximum frequency of Variant B is about 8 MHz, and one lap takes 27 clock cycles. From this data, a simple calculation tells us that the computation time of Variant A is about 1.2 microseconds, while that of Variant B is about 3.4 microseconds.

V. Conclusion

For the basic IM vector control scheme using a speed sensor, our analyses and estimations showed that, thanks to the absence of actual sine and cosine functions computations, and to the localized, small number of dividers (3 of them) and square root operators (only 1) involved, an FPGA implementation in parallel arithmetic using the embedded multipliers present in modern FPGAs has the potential to outperform one in on-line arithmetic, while not being overcostly in area. Even at twice the number of bits of precision than used in the on-line design, the parallel designs compare advantageously with the on-line one, both in area cost and computation time.

On-line arithmetic might reveal superior characteristics, however, when implementing the sensorless, EKF-based, vector control scheme, because of the intrinsic complexity of the matrix operations involved and the consequently increased system size. This will make the object of a further study.

Acknowledgments

This research is funded by a grant from the National Sciences and Engineering Research Council of Canada (NSERC).

CMC Microsystems provided development tools and support through the System-on-Chip Research Network (SOCRN) program.

References

[1] G. Sutter, G. Bioul, and J.-P. Deschamps, "Comparative Study of SRT-Dividers in FPGA," FPL 2004, LNCS 3203, pp. 209–220, 2004.

[2] R. Galli and A. Tenca, "A Design Methodology for Networks of Online Modules and Its Application to the Levinson–Durbin Algorithm," *IEEE Trans. on VLSI,* Vol. 12, No. 1, Jan. 2004.

[3] B.R. Lee and N. Burgess, "Improved Small Multiplier Based Multiplication, Squaring and Division," *Proc. 11th Annual Symposium on Field-Programmable Custom Computing Machines* (FCCM'03), 2003.

[4] J.-L. Beuchat and A. Tisserand, "Small Multiplier-Based Multiplication and Division Operators for Virtex-II Devices," FPL 2002, LNCS 2438, pp. 513–522, 2002.