CHAPITRE 4

GAPPA et les outils de vérification XSG

MCours.com

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.

```
jg@dsasoc.uqac.ca: /home/jg/opt
[jg@dsasoc opt]$
[jg@dsasoc opt]$
[jg@dsasoc opt]$
[jg@dsasoc opt]$
[jg@dsasoc opt]$
[jg@dsasoc opt]$ gappa < clark.txt

Results for isa in [-100, 100] and isb in [-100, 100]:
ISBETA in [-4875289607221262 3b-48 {-173.205, -2^(7.43634)}, 48752896072212623b-48 {173.205, 2^(7.43634)}]
add_res in [-75b2 {-300, -2^(8.22882)}, 75b2 {300, 2^(8.22882)}]
clark_res in [-709425b-12 {-173.199, -2^(7.43629)}, 709425b-12 {173.199, 2^(7.43629)}]
clark_res - ISBETA in [-833650732113407b-57 {-0.00578461, -2^(-7.43356)}, 809619853221375b-57 {0.00561787, 2^(-7.47576)}]
[jg@dsasoc opt]$
```

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.

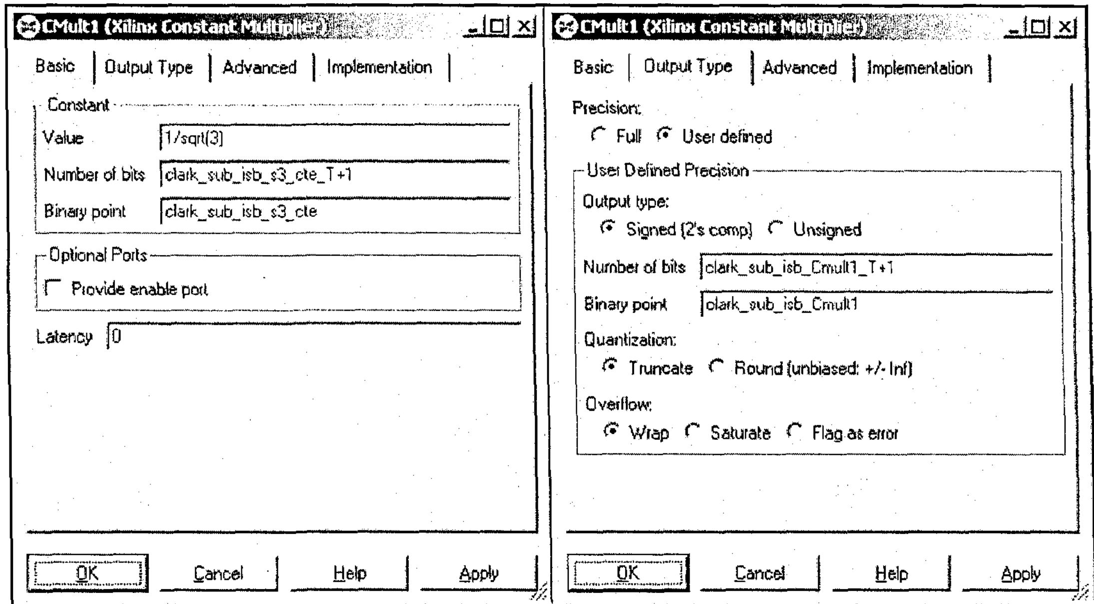

figure 21 - Précision variable dans un bloc XSG

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.

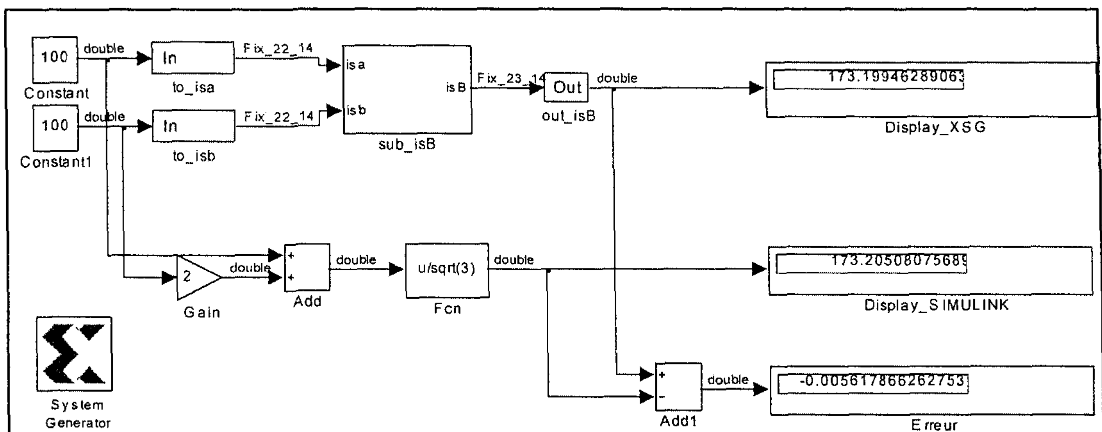

figure 22 - Résultat d'exécution XSG

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.

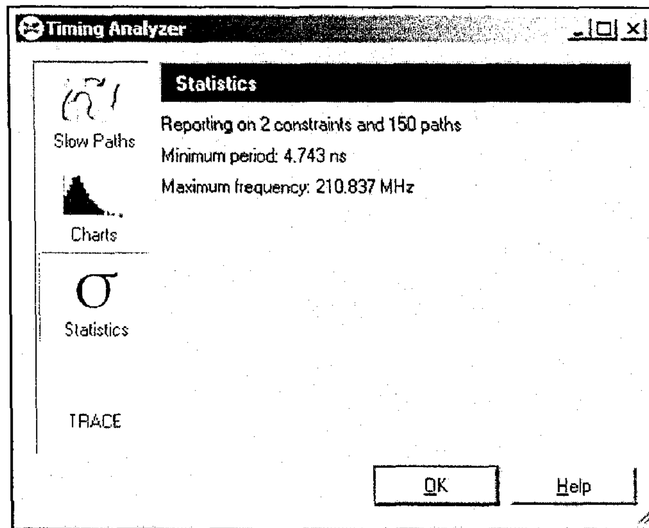

figure 23 - Analyse temporelle (vitesse maximum)

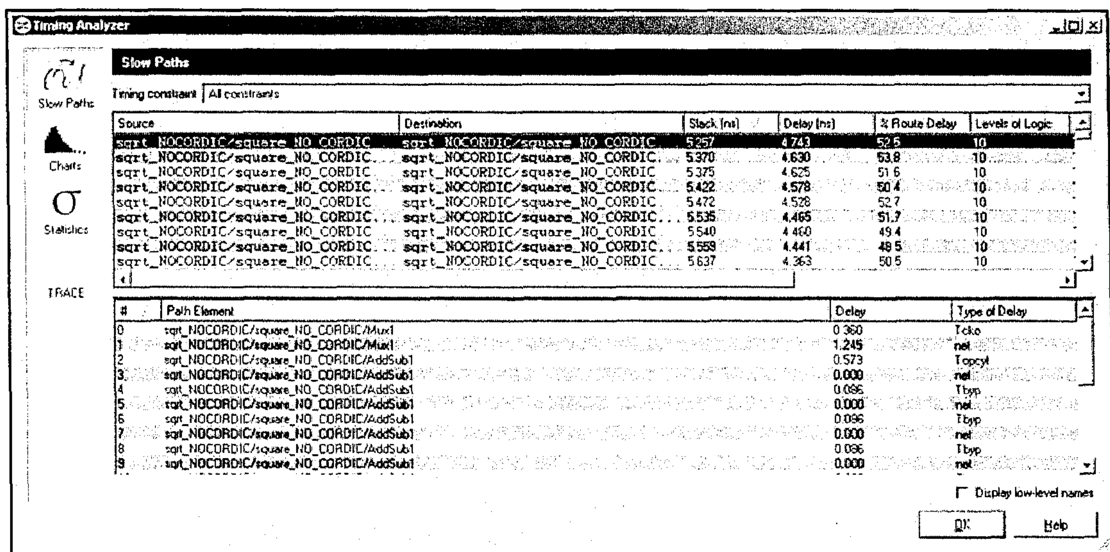

figure 24 - Analyse temporelle (détails des délais)

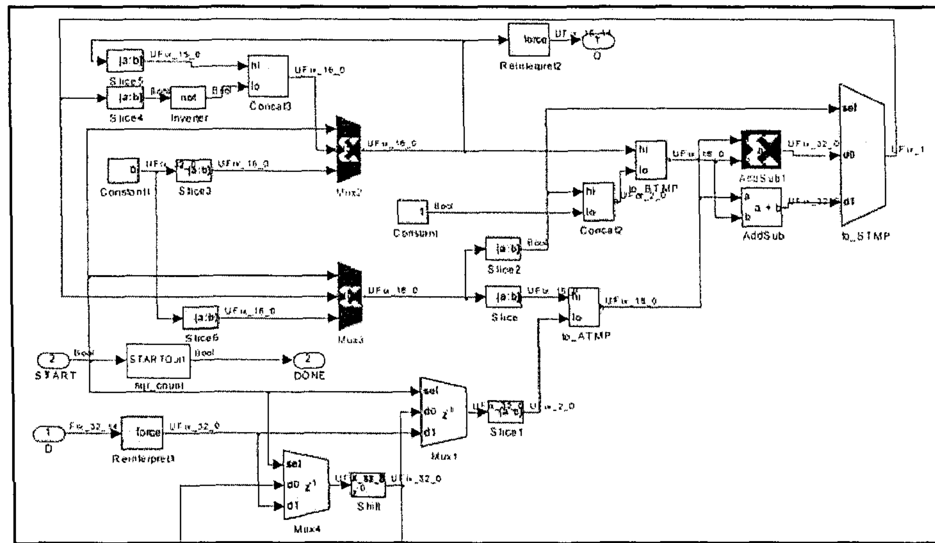

figure 25 - Analyse temporelle (identification des délais sur le modèle XSG)

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.

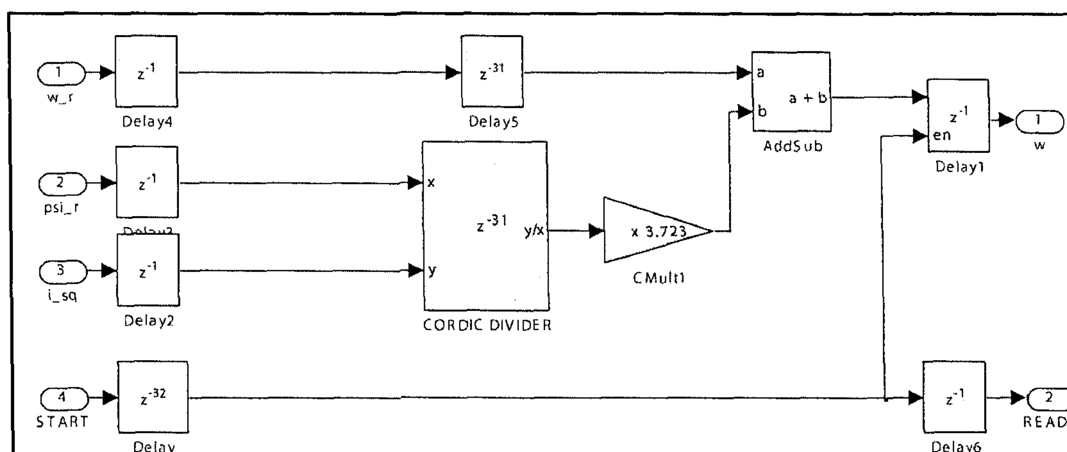

figure 26 - Bloc XSG avant analyse temporelle

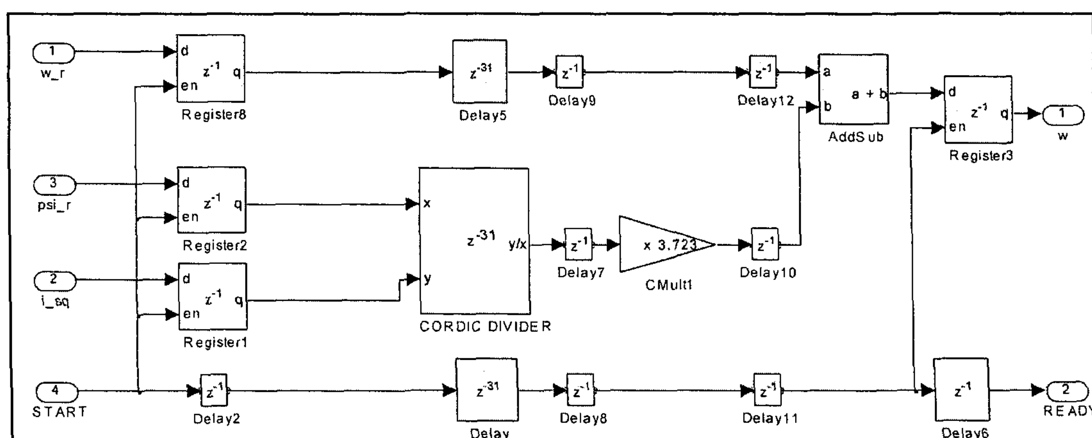

figure 27 - Bloc XSG après analyse temporelle

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.

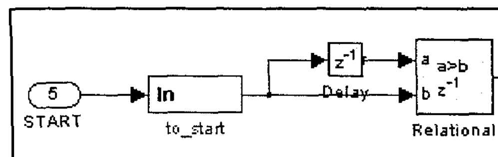

figure 28 - Simple synchronisation d'un bloc XSG co-simulé

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

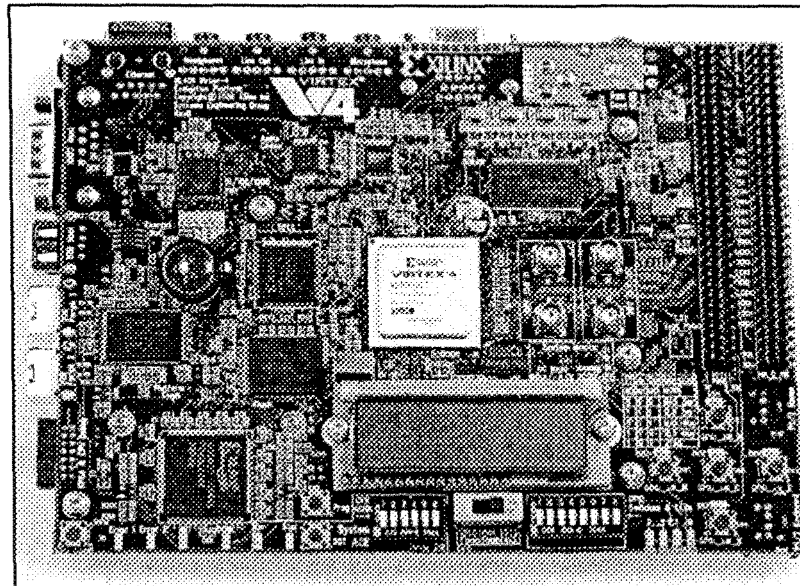

figure 29 - Carte de co-simulation Xilinx ML402

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

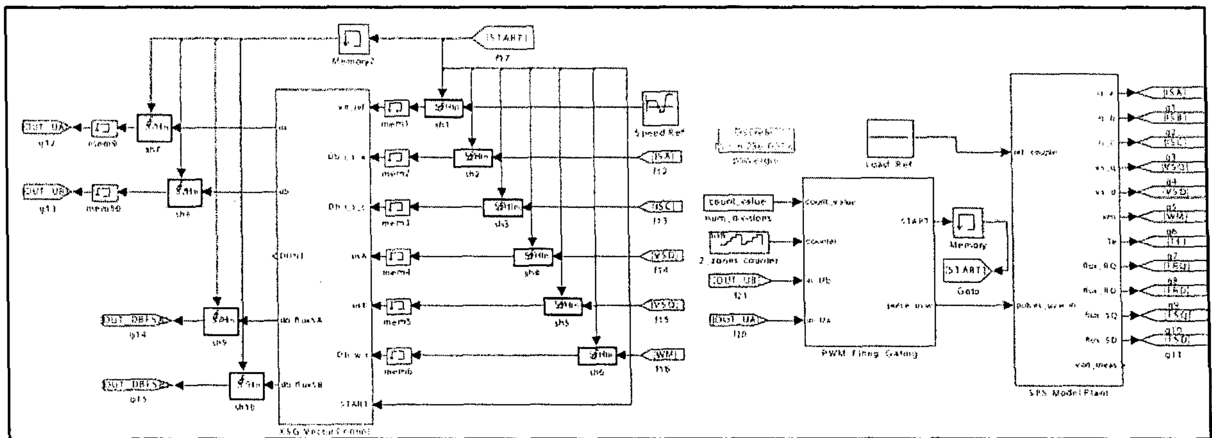

figure 30 - Co-simulation avec portillonnage, moteur et module de puissance

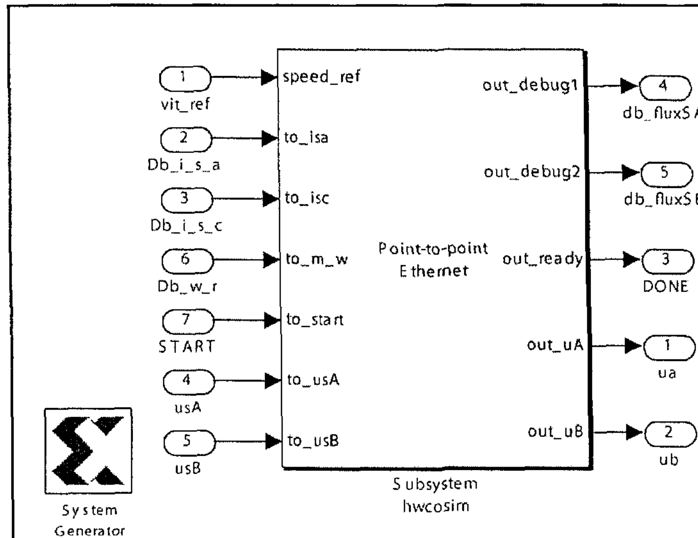

figure 31 - Interface du bloc XSG co-simulé

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.

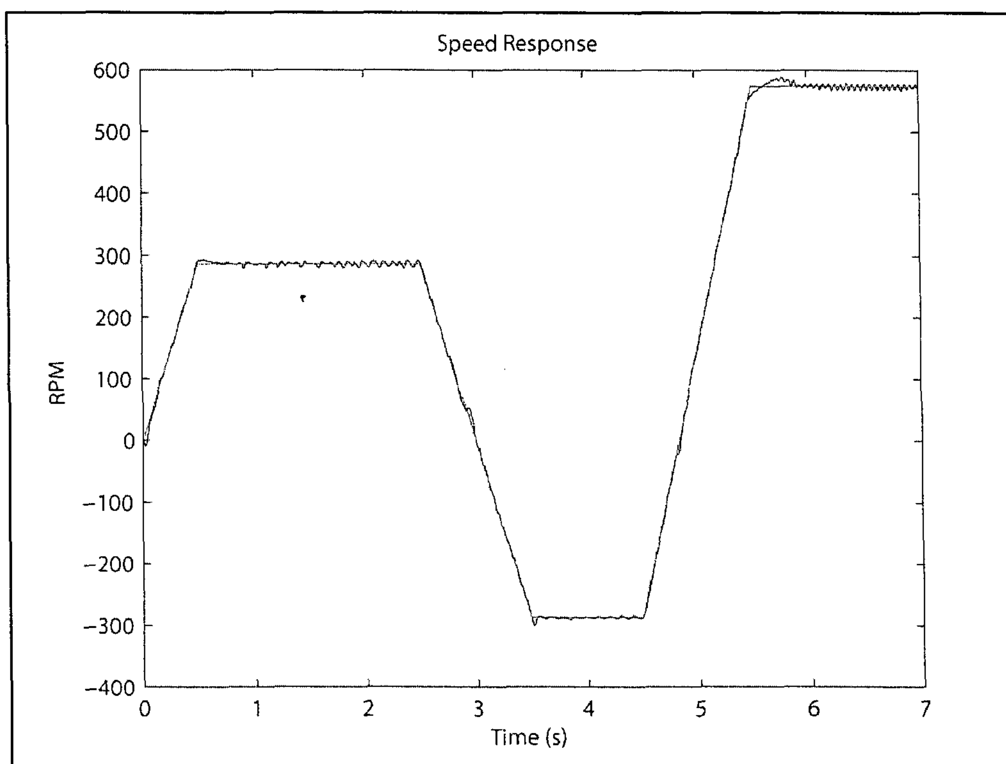

figure 32 - Réponse du système XSG co-simulé (rouge, lisse = profil, bleu = réponse)

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.

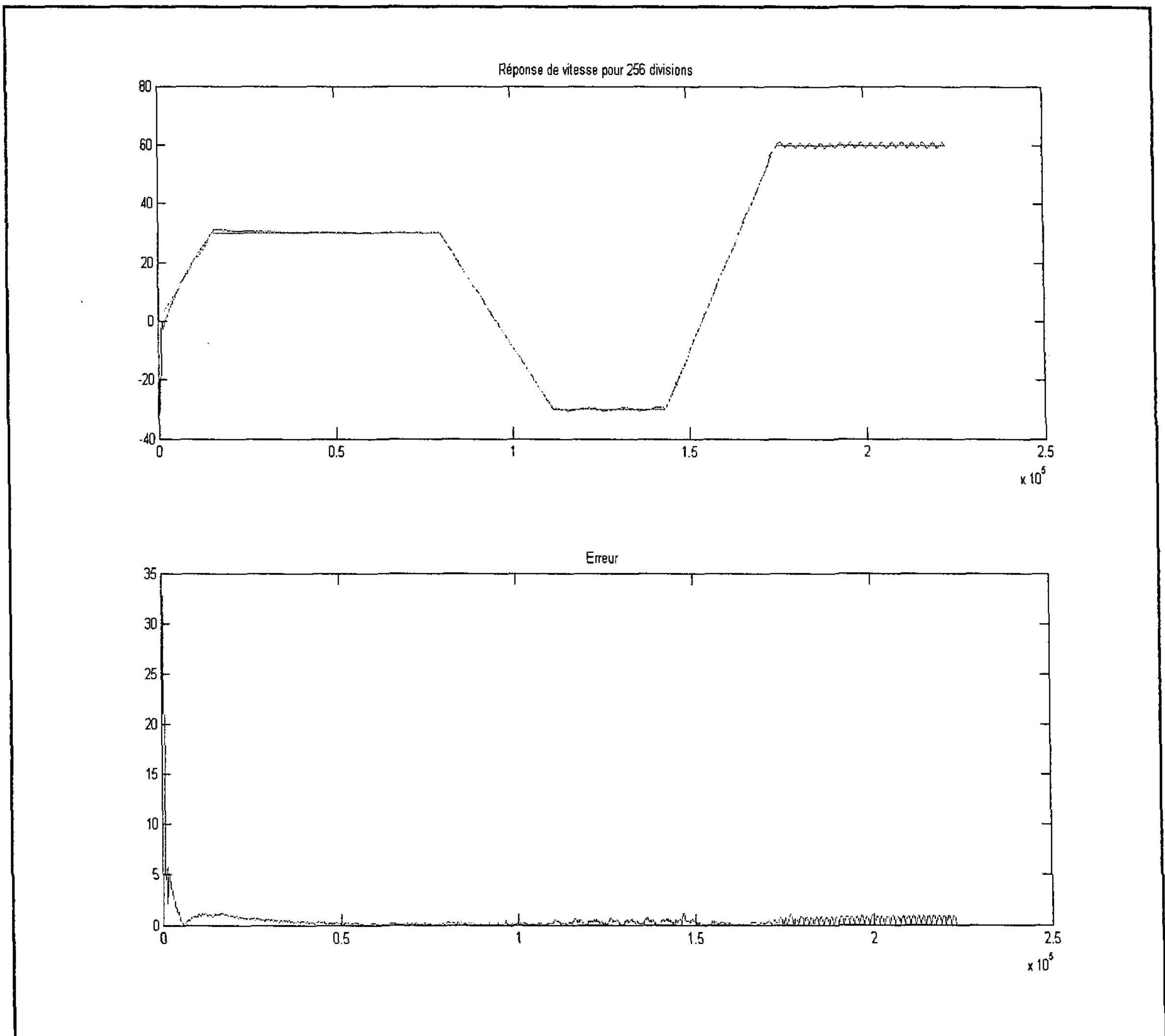

figure 33 - Réponse de vitesse - 256 divisions - Filtre 5000 Hz

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.

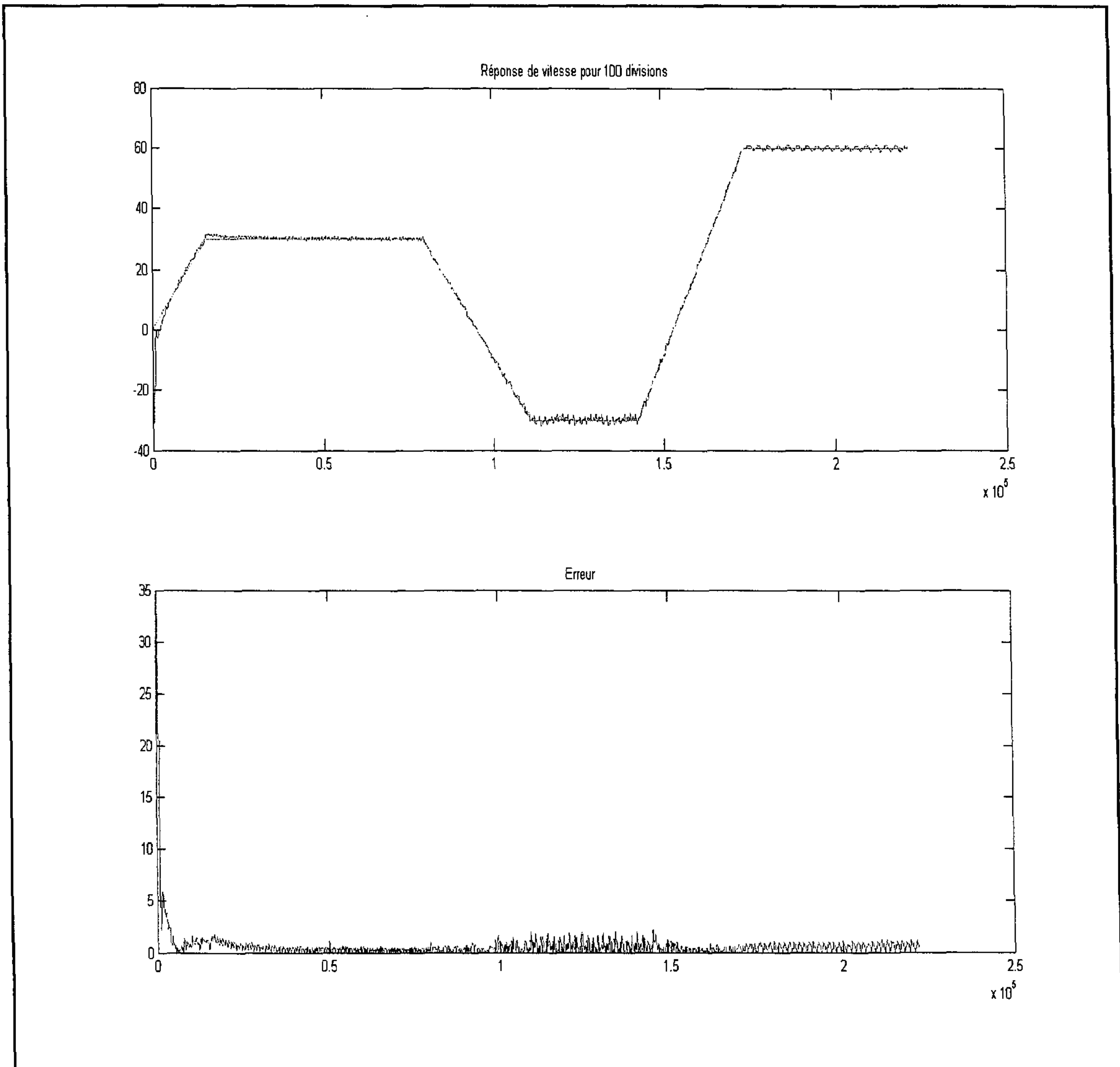

figure 34 - Réponse de vitesse - 100 divisions - Filtre 5000 Hz

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.

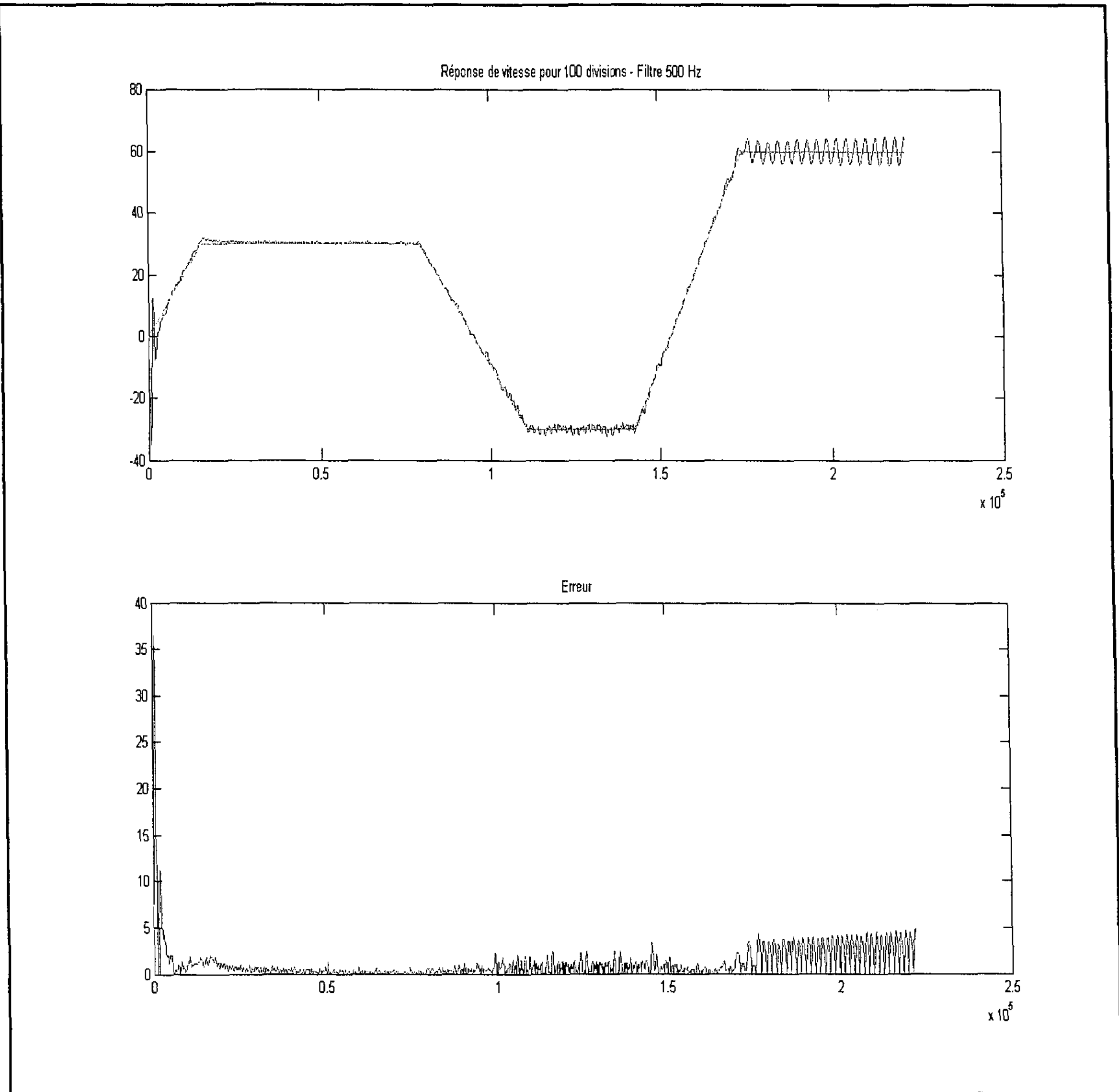

figure 35 - Réponse de vitesse - 100 divisions - Filtre 500 Hz