UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN GÉNIE ÉLECTRIQUE

PAR

MOZIPO TCHOUPOU, Aurelien Landry

SYNTHÈSE D'ARCHITECTURES PARALLÈLES DÉDIÉES DU FILTRE DE KALMAN DANS L'ENVIRONNEMENT MMALPHA

JUIN 1999

Université du Québec à Trois-Rivières

Service de la bibliothèque

Avertissement

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

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

Dédicace

À mon fils Gabriel T. Mozipo,

À mon épouse Horthense Tamdem,

À toute ma Famille.

Remerciements

Je voudrais ici exprimer ma gratitude à tous ceux qui ont contribué à la réalisation de ce travail.

Mes remerciements vont à mon directeur de recherche Daniel Massicotte qui m'aura encadré et soutenu tout au long de ce projet. Sa contribution technique a indéniablement constitué un très grand apport dans les résultats de cette recherche.

J'adresse également mes sincères remerciements à mon codirecteur de recherche Patrice Quinton qui nous aura apporté une très grande contribution technique.

Je suis également reconnaissant envers Tanguy Risset avec qui la collaboration nous a apporté beaucoup de ressources.

J'exprime également ma plus profonde gratitude à mon Épouse Horthense Tamdem qui m'aura apporté tout le soutien moral nécessaire pour aller jusqu'au bout de cette recherche.

J'aimerais aussi exprimer ma sincère reconnaissance à Mourad Zakhama, Martin Vidal, Frédéric Morin et Sylvie Legendre ; la collaboration avec eux aura créé l'ambiance quotidienne nécessaire à mon épanouissement.

Que tous ceux qui n'ont pas été cités ne sentent nullement oublier, ils auront tous contribué à leur manière à la réalisation de ce travail.

Résumé

Dans ce projet, nous proposons l'utilisation de l'outil MMAlpha et son langage Alpha comme technique innovatrice pour obtenir de façon quasi automatique des architectures hautement parallèles du filtre de Kalman de covariance et du filtre de Kalman racine carrée de covariance. Le langage Alpha et son environnement MMAlpha ont été développés à l'IRISA dans le cadre du projet API (Architectures Parallèles Intégrées). Alpha est un langage fonctionnel développé pour la synthèse d'architectures régulières à partir des équations récurrentes affines. Alpha a été développé comme étant la base d'une méthodologie de conception des réseaux réguliers assistée par ordinateur. Ce langage est basé sur la relation fondamentale qui existe entre les réseaux réguliers et les systèmes d'équations récurrentes affines. MMAlpha est l'environnement supportant Alpha et comportant les commandes nécessaires à la dérivation des architectures parallèles et est fonctionnel sous Mathematica®, disponible sur plusieurs plates-formes : UNIX, Windows NT et Mac OS. La latence et la vitesse des algorithmes sont par conséquent améliorées de façon significative lorsqu'on dérive ainsi des architectures systoliques avec MMAlpha.

Le filtre de Kalman a été choisi comme cas d'étude de l'environnement MMAlpha dans le développement interactif d'algorithmes et d'architectures hautement parallèles. Le

filtre de Kalman est un estimateur linéaire optimal qui peut reconstruire l'état d'un système à partir des données mesurées et dans un environnement stochastique. Il a fait ses preuves dans plusieurs domaines et sur des applications variées notamment en commande avec par exemple la résolution du problème de positionnement global (GPS) en navigation aérienne et maritime ; en traitement du signal avec par exemple la reconstitution de signaux (ex: séismologie, optométrie, chromatographie, communication, etc.).

Sa structure matricielle montre clairement que $O(M^3)$ opérations doivent être effectuées pendant une période d'échantillonnage où M représente la dimension du système. Ces calculs ne sont pas réalisables en temps réel pour de nombreuses applications pratiques. C'est ce qui explique que le filtre de Kalman est peu utilisé dans ces applications et dans tous les autres domaines où la vitesse de calcul (débit) est un critère important, malgré sa versatilité.

Plusieurs architectures parallèles basées sur le filtre de Kalman ont été proposées, mais contrairement à cette étude, elles ne sont pas dérivées automatiquement par des outils. Dans le cas du filtre racine carrée de covariance, nous avons fait une étude comparative de ces architectures proposées dans la littérature à la nôtre et nous avons constaté qu'elle présente des avantages significatifs tant au niveau de la vitesse (nombre de cycles par échantillon) qu'au niveau de la surface (nombre de processeurs élémentaires).

En effet, l'efficacité de cette méthode nous a donné un réseau de M^2+1 processeurs élémentaires et 7M+6 cycles d'horloge par échantillon dans le cas du filtre de covariance, où M est la dimension de la matrice de covariance. Pour la version racine carrée,

l'architecture totale comprend (M+1)(M+2)/2 processeurs élémentaires et se déroule en 2M+5 cycles d'horloge par échantillon. Notons que cette architecture nécessite un vidage de M cycles. Mais après M/2 cycles, les opérations de l'étapes suivantes pourront démarrer.

Comme application, nous utilisons le filtre de Kalman pour la reconstitution des signaux et spécialement pour l'égalisation adaptative des canaux. Un canal est un milieu à travers lequel un signal est transmis, et récupéré à la sortie avec des modifications qui dépendent des caractéristiques du milieu (fonction de transfert).

Nous avons conçu deux architectures parallèles d'égalisation des canaux en vue de leur implantation dans une technologie VLSI. Nous avons choisi la plus performante basée sur des critères de comparaison comme la robustesse aux effets de quantification, le nombre de cycles d'horloge nécessaires pour filtrer un échantillon et l'efficacité de l'algorithme de reconstitution.

L'architecture proposée est validée par simulation en VHDL dans le cas d'un canal à réponse impulsionnelle variante. La modélisation a été faite dans Mentor Graphics® et le processeur global nommé **SRCKAL** comporte le réseau triangulaire de 10 processeurs élémentaires, un bloc multiplexage qui prend les données à la sortie du réseau de processeurs et les renvoie à ses entrées convenables pour l'étape suivante. Il comprend aussi un bloc de contrôle qui est une machine à états finies réalisées avec System Architect© de Mentor Graphics®. Enfin, il y a un bloc de normalisation/dénormalisation qui convertit les données entrant dans le processeur dans l'échelle de la dynamique interne de celui-ci, puis il reconvertit les résultats à leur grandeur normale avant de les mettre à la sortie du

processeur. Nous avons évalué les performances de cette architecture en prenant comme longueur des mots 20 bits et en utilisant une technologie 0.5 μm CMOS. La fréquence d'horloge est évaluée à 3 Mhz pour les processeurs chargés de générer les rotations de Givens, donc comprenant des divisions et des racines carrées et 40 Mhz pour les processeurs chargés d'appliquer les rotations. Le nombre total de transistors pour M=3 est d'environ 750 000 incluant tous les blocs du processeur.

Liste des sigles et abréviations

API : Architectures Parallèles Intégrées

BER : Taux d'erreur sur les bits (*Bit Error Rate*)

CMOS : Semiconducteur complémentaire à oxyde de métal (*Complementary Metal Oxide Semiconductor*)

DSP : Processeur de signaux numériques (*Digital Signal Processor*)

FPGA : Réseau de portes logiques programmables (*Field Programmable Gate Arrays*)

GPS : Système de positionnement global (*Global Positionning System*)

IRISA : Institut de Recherche sur les Systèmes Aléatoires

LAN : Réseau Local (*Local Area Network*)

LMS : Moindres carrés (*Least Mean Square*)

LQ : Quadratique linéaire (*Linear Quadratic*)

LU : Inférieur et Supérieur (*Lower and Upper*)

MAC : Multiplieur Acummulateur

PE : Processeur Élémentaire

PID : Proportionnel Dérivatif Intégrateur

RIF : (Filtres à) Réponse Impulsionnelle Finie

RII : (Filtres à) Réponse Impulsionnelle Infinie

RLS : Moindres carrés récursifs (*Recursive Least Square*)

SNR : Rapport signal sur bruit (*Signal to Noise Ratio*)

SRCKAL : Processeur basé sur le filtre de Kalman racine carrée de covariance (*Square Root Covariance Kalman Filter Processor*)

UQTR : Université du Québec à Trois-Rivières

USI : Unité du Système International

VHDL : Langage de description niveau matériel des circuits intégrés à haute vitesse. (*VHSIC (Very High Speed Integrated Circuit) Hardware Description Language*)

VLSI : Intégration à très grande échelle (*Very Large Scale Integration*)

API : Architectures Parallèles Intégrées

BER : Taux d'erreur sur les bits (*Bit Error Rate*)

CMOS : Semiconducteur complémentaire à oxyde de métal (*Complementary Metal Oxide Semiconductor*)

DSP : Processeur de signaux numériques (*Digital Signal Processor*)

FPGA : Réseau de portes logiques programmables (*Field Programmable Gate Arrays*)

GPS : Système de positionnement global (*Global Positionning System*)

IRISA : Institut de Recherche sur les Systèmes Aléatoires

LAN : Réseau Local (*Local Area Network*)

LMS : Moindres carrés (*Least Mean Square*)

LQ : Quadratique linéaire (*Linear Quadratic*)

LU : Inférieur et Supérieur (*Lower and Upper*)

MAC : Multiplieur Acummulateur

PE : Processeur Élémentaire

PID : Proportionnel Dérivatif Intégrateur

RIF : (Filtres à) Réponse Impulsionnelle Finie

RII : (Filtres à) Réponse Impulsionnelle Infinie

RLS : Moindres carrés récursifs (*Recursive Least Square*)

SNR : Rapport signal sur bruit (*Signal to Noise Ratio*)

SRCKAL : Processeur basé sur le filtre de Kalman racine carrée de covariance (*Square Root Covariance Kalman Filter Processor*)

UQTR : Université du Québec à Trois-Rivières

USI : Unité du Système International

VHDL : Langage de description niveau matériel des circuits intégrés à haute vitesse. (*VHSIC (Very High Speed Integrated Circuit) Hardware Description Language*)

VLSI : Intégration à très grande échelle (*Very Large Scale Integration*)

Liste des symboles

cov[.,.] : covariance

δ_{jk} : Symbole de Kronecker

$\tilde{d}_v$: Signal d_v entaché de bruit

$\hat{d}_v$: Valeur estimée de d_v

$\overset{\circ}{d}_v$: Valeur exacte de d_v

d_v^* : Valeur désirée de d_v (consigne sur d_v)

$\overset{\Delta}{=}$: Égale par définition

E[.] : Espérance mathématique

Φ^{-T} : Inverse de la transposée de Φ = Transposée de l'inverse de Φ

$\|.\|$: Norme 2

$\mathbf{A}^{1/2}$: Racine carrée de $\mathbf{A}$

â : Paramètre de syntonisation du filtre ($\hat{a} = \sigma_w^2 / \sigma_v^2$ pour le filtre de Kalman de covariance et $\hat{a} = \sigma_w / \sigma_v$ pour le filtre de Kalman racine carrée de covariance)

$\hat{a}_{RLS}$: Facteur d'initialisation du filtre RLS

ë : Facteur d'oubli du filtre RLS

tr(.) : trace

T[.] : Triangularisation

μ : Taux de convergence du filtre LMS

μm : micromètre

O(.) : De l'ordre de

σ_v^2 : Variance du signal aléatoire v

σ_w^2 : Variance du signal aléatoire w

σ_v : Écart type du signal aléatoire v

σ_w : Écart type du signal Aléatoire w

$\hat{\mathbf{x}}$: Valeur estimée de **x**

z^{-1} : Délai unitaire

cov[.,.] : covariance

δ_{jk} : Symbole de Kronecker

$\tilde{d}_v$: Signal d_v entaché de bruit

$\hat{d}_v$: Valeur estimée de d_v

$\mathring{d}_v$: Valeur exacte de d_v

d_v^* : Valeur désirée de d_v (consigne sur d_v)

$\stackrel{\Delta}{=}$: Égale par définition

$E[.]$: Espérance mathématique

Φ^{-T} : Inverse de la transposée de Φ = Transposée de l'inverse de Φ

$\|.\|$: Norme 2

$\mathbf{A}^{1/2}$: Racine carrée de **A**

â : Paramètre de syntonisation du filtre ($\hat{a} = ó_w^2 / ó_v^2$ pour le filtre de Kalman de covariance et $\hat{a} = ó_w / ó_v$ pour le filtre de Kalman racine carrée de covariance)

$\hat{a}_{RLS}$: Facteur d'initialisation du filtre RLS

ë : Facteur d'oubli du filtre RLS

tr(.) : trace

T[.] : Triangularisation

μ : Taux de convergence du filtre LMS

μm : micromètre

O(.) : De l'ordre de

σ_v^2 : Variance du signal aléatoire v

σ_w^2 : Variance du signal aléatoire w

σ_v : Écart type du signal aléatoire v

σ_w : Écart type du signal Aléatoire w

$\hat{\mathbf{x}}$: Valeur estimée de **x**

z^{-1} : Délai unitaire

Table des matières

DÉDICACE i

REMERCIEMENTS ii

RÉSUMÉ iv

LISTE DES SIGLES ET ABRÉVIATIONS viii

LISTE DES SYMBOLES xi

TABLE DES MATIÈRES xiv

LISTE DES TABLEAUX xviii

LISTE DES FIGURES xix

CHAPITRE 1

INTRODUCTION 1

1.1 Objectifs 3

1.2 Problématique 5

1.3 Méthodologie 7

1.4 État de la recherche sur les outils de synthèse automatique d'architectures parallèles. 8

1.5 Organisation du mémoire 10

CHAPITRE 2

ENVIRONNEMENT MMALPHA 13

2.1 Principe 13

2.2 Le langage Alpha et son environnement MMAlpha 17

2.3 Procédure de dérivation d'une architecture parallèle 19

2.4 Synthèse d'une architecture parallèle pour le produit matrice-vecteur 21

CHAPITRE 3

FILTRE DE KALMAN ET APPLICATIONS 23

3.1 Le filtre de Kalman 24

3.1.1 Principe de filtrage 25

3.1.2 Filtre de covariance 28

3.1.3 Filtre d'information 30

3.1.4 Filtre racine carrée de covariance 33

3.1.5 Filtre racine carrée de l'information 36

3.2 Applications du filtre de Kalman à l'égalisation des canaux 38

3.2.1 Égalisation des canaux par filtre de Kalman standard 44

3.2.2 Égalisation des canaux par filtre de covariance racine carrée 44

3.3 Résultats de simulation de l'égalisation des canaux par filtre de Kalman 46

3.3.1 Égalisation par Kalman Standard. 49

3.3.2 Égalisation par Kalman covariance racine carrée 50

3.3.3 Évaluation des performances 50

3.4 Application du filtre de Kalman à la commande 53

3.5 Justification de l'implantation du filtre de Kalman en technologie VLSI 60

CHAPITRE 4

SYNTHÈSES D'ARCHITECTURES PARALLÈLES AVEC MMALPHA 64

4.1 Filtre de covariance 65

4.1.1 Programmation en Alpha 65

4.1.2 Résultats d'ordonnancement de MMAlpha 67

4.1.3 Description de l'architecture systolique 68

4.2 Filtre racine carrée de covariance 76

4.2.1 Triangularisation de matrices denses dans MMAlpha 77

4.2.2 Programmation du filtre racine carrée de covariance en Alpha 83

4.2.3 Résultat d'ordonnancement de MMAlpha 85

4.2.4 Description de l'architecture systolique 86

4.3 Comparaison des performances 91

4.4 Résultats de simulation du programme Alpha 93

CHAPITRE 5

ARCHITECTURE ET SYNTHÈSE EN TECHNOLOGIE VLSI 94

5.1 Étude des effets de quantification 95

5.2 Choix de l'architecture et de la technologie VLSI 101

5.2.1 Choix de l'architecture 101

5.2.2 Choix de la technologie 103

5.3 Modélisation et résultats de simulation du VHDL du processeur 103

5.3.1 Modélisation VHDL du processeur 103

5.3.2 Résultats de correction du Processeur **SRCKAL** 107

5.4 Synthèse en technologie CMOS 0.5µm 109

CHAPITRE 6

CONCLUSION 111

6.1 Synthèse des résultats 111

6.2 Recommandations et suite des travaux 113

BIBLIOGRAPHIE 116

ANNEXES

I. ARTICLES PUBLIÉS AU COURS DE CETTE RECHERCHE 124

II. PROGRAMMES ALPHA 138

III. PROGRAMMES MATLAB® 167

IV. PROGRAMMES VHDL 213

Liste des tableaux

Tableau 4.1 : Résultat d'égalisation pour différents niveaux de bruit. 51

Tableau 4.1 : Cycles d'horloges dérivés du scheduling donnée par MMAlpha.................... 68

Tableau 4.2 : Ordonnacement du programme du filtre racine carrée de covariance 85

Tableau 4.3 : Comparaison à d'autres architectures du filtre de covariance......................... 92

Tableau 5.1 : Taille et vitesse des filtres de Kalman 102

Liste des Figures

Figure 2.1 : Additionneur avec retard 17

Figure 2.2 : Principales étapes de dérivation d'une architecture parallèle avec MMAlpha.. 22

Figure 3.1 : Diagramme bloc du filtre de Kalman........ 29

Figure 3.2 : Égalisation adaptative des canaux par filtrage adaptatif [HAY96]........ 39

Figure 3.3 : Diagramme bloc de l'égalisateur adaptatif des canaux par filtre de Kalman.... 40

Figure 3.4 : Exemple type de signaux générés synthétiquement : a) signal de test du canal a(k) et b) signal de sortie corrompu $\tilde{y}(k)$ avec $\sigma_v^2 = 0.6$ (SNR = 5dB)........ 41

Figure 3.5 : a) Réponse impulsionnelle invariante du canal et b) variations de l'amplitude des composantes de **H** dans le cas d'une réponse impulsionnele variante avec N = 500 points et P = 5 périodes 42

Figure 3.6 : Signal corrompu et entaché de bruit : SNR = 20dB et BER = 50% 47

Figure 3.7 : Résultats de simulation du filtre de Kalman standard pour SNR = 20dB,........ 49

Figure 3.8 : Résultats de simulation du filtre de Kalman racine carrée de covariance pour SNR = 20dB, ... 50

Figure 3.9 : BER pour égalisation adaptative des canaux par LMS, RLS et Kalman. SNR = 15dB 53

Figure 3.10 : Schéma général du système de commande 56

Figure 3.11 : Diagramme bloc du système 58

Figure 3.12 : Positionnement de la tige a) avec loi de commande sur retour d'état observé par le filtre de Kalman et b) un contrôleur PID 60

Figure 4.1 : Programme Alpha pour le filtre de covariance. a) Programme prinical b) Sous-programmes 66

Figure 4.2 : Architecture systolique à topologie carrée 69

Figure 4.3 : Architecture systolique et son flot de données pour un échantillon $\hat{x}_k$ 72

Figure 4.4 : Flot de données et cellule MAC pour l'exécution de l'équation (4.1) 73

Figure 4.5 : Les différents modes de fonctionnement 73

Figure 4.6. Architecture systolique à topologie triangulaire, appliquée à la triangularisation d'une matrice A avec dim(A)=M=3 79

Figure 4.7. Fonctionnement des cellules rondes pour la factorisation de Givens : a) Flot de données processeurs ronds, b) flot de données processeurs carrées, c) algorithme pour un processeur rond, d) algorithme pour un processeur carré. 80

Figure 4.8 : Programme général matlab de triangularisation de matrices 81

Figure 4.9 : Programme général Alpha de triangularisation de matrices 82

Figure 4.10 : Programme Alpha pour une étape du filtre de Kalman racine carrée de covariance 84

Figure 4.11 : Architecture globale 87

Figure 4.12. Mode d'opération pour les cellules carrées à l'étape 2. 89

Figure 4.13. Mode d'opération pour les cellules carrées du triangle inférieur à l'étape 3.... 89

Figure 4.14. Disposition des variables dans les registres après l'étape 3............................. 90

Figure 4.15. Décalage des éléments de $\mathbf{x}_+$ pour former $\mathbf{x}$. 90

Figure 5.1 : Erreur de filtrage en fonction du nombre de bits 100

Figure 5.2 : Erreur de quantification en fonction du nombre de bits................................ 100

Figure 5.3 : Diagramme bloc du processeur **SRCKAL** 104

Figure 5.4 : Signal corrompu à la sortie du canal ; SNR = 20dB, BER =50%.................. 108

Figure 5.5 : Résultats de simulation du modèle VHDL 20 : BER = 0 109

Chapitre 1

Introduction

Les processeurs spécialisés sont conçus pour des applications en temps réel et pour lesquelles le nombre d'opérations est extrêmement élevé. Ces applications ne peuvent fonctionner raisonnablement sur des microprocesseurs multi-usages qui ne possèdent pas d'unités de calcul performantes et qui nécessitent parfois plusieurs dizaines de cycles d'horloge pour effectuer une simple multiplication. Avec l'émergence des applications en temps réel et où les calculs sont très intenses, la conception des circuits spécialisés s'est considérablement développée tant au niveau de l'approche (méthode de conception) qu'au niveau de la réalisation (technologie utilisée pour fabriquer le circuit).

Les méthodologies de conception sont nombreuses et elles ont pour but de minimiser le temps de cycle, de maximiser le nombre d'opérations effectuées pendant un cycle d'horloge, d'optimiser la surface ou de minimiser la consommation, ou une combinaison quelconque des ces objectifs. Les premières techniques utilisées visaient à augmenter la fréquence d'horloge du système. Ensuite ont été introduites des méthodes visant tout simplement à accélérer les opérations en les réalisant en parallèle et non

séquentiellement comme c'était le cas jusqu'alors. Pour cela, plusieurs auteurs ont développé des architectures parallèles pour des algorithmes performants qui étaient encore non exploités à cause de la densité des calculs. Nous pensons ici au filtre de Kalman qui a fait ses preuves dans plusieurs domaines et sur des applications variées notamment en commande avec par exemple la résolution du problème de positionnement global (GPS) en navigation aérienne et maritime [IRW91] ; en traitement du signal avec par exemple la reconstitution de mesurandes [MAS95] et en chromatographie [MEI84]. Des tentatives d'approche ont été publiées pour des architectures parallèles du filtre de Kalman, parmi lesquels on cite entre autres [KUN91], [IRW91], [MAS92] et [FAY95]. Nous allons apporter notre contribution à cette recherche en utilisant une approche inédite qui cherche à bénéficier de l'avancée des recherches sur l'intégration à très grande échelle des circuits. Notre objectif est de concevoir des circuits très gourmands en silicium, mais dans lesquels le parallélisme est maximal et qui possèdent par conséquent des temps de réponse très courts. En effet, les technologies VLSI nous offrent les moyens de fabriquer des circuits possédant plus d'un million de transistors au millimètre carré.

Notre approche sera d'utiliser un logiciel de synthèse d'architectures parallèles pour essayer d'obtenir une architecture parallèle du Filtre de Kalman. Nous proposons l'application d'un ensemble d'outils de dérivation d'architectures parallèles, qui nous donneront automatiquement une architecture parallèle du filtre de Kalman standard et du filtre racine carrée de la covariance. Ces outils sont basés sur le formalisme des systèmes à équations affines récurrentes [QUI89b], [MOE96]. Ils sont intégrés dans l'environnement appelé MMAlpha et fonctionnel dans Mathematica®. Ces techniques avancées de calculs

parallèles dans les circuits VLSI ont été proposées dans [QUI89b], [VER91], [MOE96], [API97], [BAL98], [API98]. MMAlpha est un environnement de synthèse dédié à la dérivation d'architectures parallèles. L'un de ses principaux éléments est Alpha qui est un langage fonctionnel développé pour la synthèse d'architectures parallèles à partir du formalisme des systèmes à équations affines récurrentes. En plus d'être rapide, ce système nous conduit vers une architecture systolique de réseaux de processeurs où le parallélisme est maximal et par conséquent le temps de calcul entre deux échantillons consécutifs est minimal [MOZ98].

Nous réaliserons une étude comparative d'autres architectures proposées dans la littérature, d'une part pour évaluer les performances et d'autres part pour répondre aux objectifs que nous nous sommes fixés. L'exemple principal sur lequel nous allons appliquer notre architecture est l'égalisation adaptative des signaux.

1.1 Objectifs

Notre projet de recherche vise un objectif primordial : résoudre le problème de l'implantation en VLSI des algorithmes basés sur le filtre de Kalman à l'aide d'un outil de synthèse d'architecture parallèles.

Nous divisons notre tâche en deux parties :

i – Nous voulons utiliser le filtre de Kalman pour une application réelle pour laquelle il existe une solution qui nous permette de faire une comparaison ; nous avons choisi pour

cela l'égalisation adaptative des canaux pour laquelle des solutions par LMS et RLS ont été proposées dans [HAY96]. Étant donné que le filtre de Kalman est un estimateur linéaire optimal, il sera utilisé ici pour la reconstruction d'état dans un système dynamique non invariant. Dans notre recherche, nous devons appliquer le filtre de Kalman pour résoudre le problème de reconstitution de signaux ayant traversés un canal donné. En pratique, un canal peut être constitué par une réponse impulsionnelle caractérisant les canaux hertziens ou les câbles.

ii - Nous visons l'implantation de cet algorithme dans une technologie VLSI, avec une architecture dérivée par des outils modernes que nous voulons ici expérimenter. Ces outils sont les composantes de l'environnement MMAlpha dont le principal élément est le logiciel Alpha qui a été conçu pour introduire du parallélisme massif dans des applications. Nous allons donc utiliser ce logiciel pour dériver une architecture décrite au niveau matériel et obtenir un modèle de cette architecture.

Nous devons donc proposer une solution aux problèmes qui rendaient difficile l'implantation en VLSI des architectures du filtre de Kalman. L'émergence des techniques de calcul parallèle comme les réseaux systoliques [KUN82] nous donne les moyens de réaliser cet objectif. Les performances des architectures ainsi dérivées seront ensuite comparées à celles des architectures déjà publiées [IRW91], [YEH88].

1.2 Problématique

L'implantation en technologie VLSI des algorithmes basés sur le filtre de Kalman suscite un intérêt sans cesse croissant car ils peuvent être appliqués de manière très efficace dans plusieurs domaines. La structure matricielle du filtre de Kalman montre clairement que $O(M^3)$ opérations arithmétiques sont exécutées pendant chaque cycle d'horloge, M étant la dimension du système. Ces calculs ne sont pas réalisables en temps réel pour de nombreuses applications pratiques. C'est ce qui explique que le filtre de Kalman est peu utilisé dans les applications en temps réel et dans tous les autres domaines où la vitesse de calcul (débit) est un critère important, malgré sa versatilité.

En effet, le filtre de Kalman intervient en commande [KUN91], en traitements de signaux [MAS95] et en communication (ce que nous allons montrer dans ce rapport). L'estimation récursive du vecteur d'état d'un système par le filtre de Kalman est considérée comme optimale car le filtre de Kalman dérive des résultats optimisés par la minimisation de l'erreur d'estimation. Il est parmi les meilleurs algorithmes de filtre optimal qui existent dans la littérature, pour les systèmes linéaires. Mais, le nombre extrêmement élevé d'opérations qu'il y a dans l'algorithme du filtre de Kalman a jusqu'alors été un obstacle à son implantation dans un circuit dédié. D'autres algorithmes moins denses en calcul se trouvent facilement sur le marché ; par exemple l'algorithme LMS est disponible commercialement sur le DSP56200 de Motorola. Nous allons donc essayer de dériver une architecture où presque toutes les opérations indépendantes sont exécutées en parallèle, ne laissant en exécution séquentielle que les opérations dont l'exécution dépend des résultats

d'une opération en cours. Le logiciel MMAlpha est justement conçu pour réaliser des réseaux réguliers répondant à ces spécifications. Le filtre de Kalman est bien adapté pour ce logiciel de part sa structure régulière.

L'architecture du filtre de Kalman ainsi dérivée à l'aide de MMAlpha s'appliquera à un exemple pratique pour lequel des solutions par les méthodes LMS et RLS [HAY96] ont été proposées. Cet exemple est l'égalisation adaptative de canaux linéaires et variant.

Un canal possédant une réponse impulsionnelle donnée est traversé par un signal polaire {+1, -1}, le signal de sortie est corrompu par une séquence entachée de bruit aléatoire, de moyenne nulle et de variance donnée. Le rôle de l'égalisateur est de corriger les distorsions produites par le canal, et en présence de signal additif. Nous allons donc résoudre ce problème avec le filtre de Kalman, puis implanter l'architecture choisie dans une technologie VLSI. Le choix du filtre de basera sur celle qui présentera les meilleurs résultats de filtrage ainsi que des bonnes performances en ce qui concerne le nombre de cycles d'horloge nécessaires pour filtrer un échantillon. Les versions du filtre de Kalman utilisées seront le filtre de covariance standard (*Covariance Kalman Filter*) et le filtre racine carrée de la covariance (*Square Root Covariance Kalman Filter*) [KAM71]. Étant donné que la réponse impulsionnelle du canal est variable, les gains de Kalman le seront aussi : on aura donc un filtre de Kalman non stationnaire. Ainsi, on résout l'équation de Riccatti qui constitue un grand obstacle à l'implantation en VLSI des architectures du filtre de Kalman non stationnaire.

1.3 Méthodologie

La réalisation dans des délais raisonnables d'un circuit demande une approche qui définit et planifie toutes les étapes à réaliser pour obtenir le produit final. Une méthodologie efficace et couramment utilisée est la conception descendante (*top-down design*).

Elle stipule qu'une implantation, du concept au produit final, se fait en une séquence de trois tâches : la spécification comportementale, l'implantation ou la conception du circuit intégré, et la vérification [MAD95]. La spécification comportementale est la transcription de l'algorithme sous une forme intégrable dans un circuit et compréhensible par n'importe quel ingénieur de conception des circuits intégrés. Cette partie se fera à l'aide des outils de MMAlpha pour la dérivation de l'architecture systolique. La conception du circuit consistera en la modélisation niveau registre de transfert (*RTL*) en VHDL à l'aide des outils de CAO de Mentor Graphics®. Elle consistera également à la synthèse logique et à l'implantation dans une technologie VLSI (FPGA ou CMOS 0.5ì m) avec les outils de Synopsys®. Le choix de la technologie est fonction de la destination finale du produit. Enfin, la vérification et la validation qui se font au niveau du matériel ne seront pas étudiées dans le cadre de ce mémoire, mais elles seront étudiées dans la suite de ce travail au Laboratoire d'Algorithmes et d'Architectures Intégrées. C'est un aspect important qui sera considéré avant la fabrication finale du circuit.

Ces étapes décrivent donc celles que nous allons suivre tout au long de notre étude. Elles sont structurées comme indiquer dans la section suivante.

1.4 État de la recherche sur les outils de synthèse automatique d'architectures parallèles.

Plusieurs architectures ont été développées par des méthodes d'algèbres linéaires classiques, exploitant la régularité inhérente du filtre de Kalman [IRW91], [KUN91], [YEH88], [MAS95] et [FAY95]. Ces méthodes sont entre autres les rotations de Givens [QUI89a] et les algorithmes de Fadeev [YEH88]. D'autres auteurs ont préconisés des méthodes de transformation algébriques : par exemple les graphes de flot de signaux hiérarchiques (*Hierarchical Signal Flow Graphs : HSFG*), le cas de [BRO95] qui applique cette méthode à la conception d'un bloc systolique régularisé pour l'estimation de paramètres. D'autres méthodes de développement plus rapides peuvent être explorées aujourd'hui. Il s'agit de l'utilisation des outils informatiques pour dériver de façon automatique des architectures parallèles, non seulement pour le filtre de Kalman, mais aussi pour toutes les applications qui satisfont certains critères précis (régularité, algorithme définie sous la formation d'une série d'équations récurrentes affines, etc.).

En effet, plusieurs autres logiciels ont été développés pour produire des architectures parallèles des systèmes à partir de leur description mathématique. Le principe de fonctionnement de ces logiciels varie d'une équipe de recherche à l'autre. Les architectures ainsi dérivées peuvent être implantées dans des circuits FPGA, sur des réseaux de processeurs, sur des multiprocesseurs ou sur silicium (ASIC).

Les logiciels que nous avons rencontrés dans la littérature sont basés soit sur le formalisme des systèmes d'équations récurrentes , **MMAlpha** [BAL98], **OPERA** [LOE94], soit sur des méthodes classiques d'intelligence artificielle, le cas de **Transe** [DUR92]. Les graphes de flot données sont aussi utilisés dans [DUN92] pour développer **Hi-PASS**. Ces logiciels et d'autres essayent de donner des environnements de travail qui permettraient à l'avenir, à un utilisateur non familier avec les circuits intégrés de développer des architectures parallèles d'un algorithme donné et de l'implanter sur une plate forme bien spécifique. L'utilisateur pourra alors choisir d'implanter son architecture sur une technologie (DSP, FPGA, ASIC) appropriée à l'application. Dans le cas de l'implantation en VLSI, l'outil de synthèse génère automatique le code VHDL synthétisable, ensuite il pourrait passer à la synthèse dans sa technologie choisie. Certains outils avancés de synthèse automatique comme **Hi-PASS** proposent des étapes qui conduisent directement à une version de l'architecture propice à la génération automatique du layout.

Les outils énumérés ci-dessus peuvent produire des architectures de très haut niveau de parallélisme, de même qu'ils peuvent permettre de choisir le degrés de parallélisme que l'on désire obtenir dans l'architecture finale. Ce choix est fait en fonction de l'application dans laquelle l'architecture est utilisée.

Hi-PASS [DUN92] est un outil qui permet de dériver des architectures hautement parallèles, en étudiant le diagramme de flot de données. **Hi-PASS** permet également de générer une description de haut niveau très propice pour la génération automatique du Layout.

PRESAGE présenté dans [DON92], est un logiciel pour la dérivation d'architectures systoliques et périodiques. Il est basé sur le formalisme des systèmes d'équations récurrentes et des systèmes à équations quasi-affines récurrentes. C'est un prototype qui n'a pas été développé à un niveau avancé.

Approval est un environnement également basé sur le sur le formalisme des systèmes d'équations récurrentes [RAM95].

OPERA présenté dans [LOE94] est un autre logiciel basé sur le principe des systèmes d'équations récurrentes et lui également est une dérivée de **Alpha**.

Enfin, mentionons le logiciel **Transe** [DUR92], qui utilise les méthodes de l'intelligence artificielle pour dériver ses architectures systoliques.

Tout comme **MMAlpha** utilise le langage **Alpha** fonctionnel sous **Mathematica**®, **Transe** utilise le langage **Circuit-Lisp** implanté sous **Le Lisp**.

1.5 Organisation du mémoire

Nous avons présentons à la section précédente les résultats de la recherche bibliographique sur les différents outils de synthèse automatique d'architectures systoliques. Nous allons mettre un accent particulier sur l'environnement MMAlpha au chapitre 2. Nous présentons d'abord les outils de l'environnement puis nous montrons par sur exemple, les procédures de dérivation d'une architecture parallèle dans cet environnement.

Dans le chapitre 3 de ce travail, nous allons présenter les différentes versions du filtre de Kalman et quelques unes de ses applications dans le domaine du contrôle et du traitement du signal. Nous décrivons le principe de filtrage et montrons les équations des différentes versions du filtre de Kalman ainsi que ses applications. Aussi, nous donnons quelques raisons pour lesquelles nous avons choisi d'implanter une architecture du filtre de Kalman dans une technologie VLSI. Ensuite, nous présentons les résultats de simulation du langage Alpha pour l'égalisation adaptative des canaux à l'aide du filtre de covariance et du filtre racine carrée de covariance. Enfin, nous allons montrer une application du filtre de Kalman en commande.

La synthèse d'architectures parallèles à l'aide de MMAlpha est traitée dans le chapitre 4. D'une part nous synthétisons une architecture systolique pour le filtre de Kalman standard (filtre de covariance) et d'autre part une architecture systolique pour le filtre racine carrée de la covariance. Ensuite nous procédons à une étude comparative des architectures obtenues.

Dans le chapitre 5, nous choisissons une architecture parallèle que nous implantons dans une technologie VLSI. Ce chapitre est aussi consacré à l'étude des effets de quantification sur les différentes architectures étudiées. Nous faisons un choix en tenant compte des résultats d'étude de quantification, des résultats de simulation et de l'étude comparative. Nous faisons ensuite une modélisation tant architecturale que comportementale et nous présentons les résultats de simulation du modèle VHDL. Enfin

nous passons à la synthèse en VLSI de l'architecture modélisée et nous montrons les performances de l'architecture proposée.

Le chapitre 6 est la conclusion dans laquelle nous montrons notre contribution à cette recherche. Étant donné que nous avons été les précurseurs de l'approche qui consiste à utiliser l'environnement MMAlpha pour dériver des architectures parallèles du filtre de Kalman, nous faisons quelques recommandations pour la poursuite de la coopération entre l'IRISA de Rennes (France) et le Laboratoire d'Algorithmes et d'Architectures Intégrées de l'UQTR pour dériver des architectures parallèles dans d'autres applications en traitement des signaux et en commandes.

Chapitre 2

Environnement MMAlpha

Le langage Alpha et son environnement MMAlpha ont été développés à l'IRISA dans le cadre du projet API (Architectures Parallèles Intégrées) [API97]. Alpha a été développé comme étant la base d'une méthodologie de conception des réseaux réguliers assistée par ordinateur. Ce langage est basé sur la relation fondamentale qui existe entre les réseaux réguliers et les systèmes d'équations récurrentes affines [WIL95].

MMAlpha est un environnement supportant Alpha et comportant les commandes nécessaires à la dérivation des architectures parallèles et est fonctionnel sous Mathematica©, disponible sur plusieurs plates-formes : UNIX, Windows NT et Mac OS [API97].

2.1 Principe

Alpha est un langage fonctionnel développé pour la synthèse des architectures régulières à partir des équations récurrentes affines.

Les variables – et plus généralement les expressions - de Alpha sont définies par leurs valeurs (de type Booléennes, entières ou réelles) aux points de leur domaine de définition. Ce domaine P est l'ensemble des points de coordonnées entières, d'un polyèdre convexe de $\mathbb{Z}^n$ (on note $\mathbb{Z}$ l'ensemble des entiers relatifs, et $\mathbb{Z}^n$ le produits cartésien de dimension n de $\mathbb{Z}$).

En d'autres termes, le domaine P d'une expression est l'intersection d'une famille finie de demi espaces fermés et peut être spécifié par un système de contraintes [DOR94] :

$$P = \{z \in \mathbb{Z} \mid Az \geq b\}$$

Nous présentons ci-dessous une brève description de la syntaxe de Alpha. Des informations plus détaillées sur cette syntaxe peuvent être consultées dans [ALP97] et [ALP98].

Structure des variables :

Une variable dans Alpha est déclarée de la manière suivante :

<nom_var> : <domaine> `of` <type>

Le type peut être booléen, entier ou réel. Le domaine est un polyèdre convexe d'entiers.

Exemple 2.1 : Une matrice triangulaire **A** de taille $N \times N$ contenant des nombres réels sera définie comme suit en Alpha :

```
A : {i,j | 1<=i<=N; i<=j} of  real
```

Opérations point à point :

Alpha est un langage qui effectue des opérations point à point, c'est-à-dire les opérations sur une variable sont perçues comme une série d'opérations scalaires sur tout le domaine (ou plus exactement sur chacun des points du domaine) de la variable.

Par exemple soient a1 et a2 sont deux variables de domaines respectifs D1 et D2, et soit ℵ un opérateur, alors a1 ℵ a2 est une expression dont le domaine D est l'intersection des deux domaines D1 et D2 et dont la valeur à chaque point représente le résultat de l'opération entre un point de D1 et un point de D2.

Système :

Un programme Alpha a la structure suivante :

```
system <nom_systeme> (<declarations_des_entrees>)
       returns ( <declarations_des_sorties> ) ;
var <declarations_des_variables_locales>
let
  <equations>
tel;
```

Le système calcule les sorties en fonction des entrées et des variables locales. Dans un programme Alpha, au plus une équation définit une variable.

Exemple 2.2 :

Nous présentons ici un exemple simple et classique pour illustrer la syntaxe de Alpha. La multiplication matrice-vecteur de **A** par **v** est définie mathématiquement comme suit :

$$\mathbf{C = Av}$$

soit

$$C_{ij} = \sum_{k=1}^{N} A_{ik} v_k$$

En Alpha, un exemple de programme réalisant ce produit serait le système *prodVect* suivant :

```
  system prodVect : {N | N>1}
       (A : {i,j | 1<=i,j<=N} of real;
        V : {i | 1<=i<=N} of real)
  returns    (C : {i | 1<=i<=N} of real);
var
  c : {i,j | 1<=i<=N; 0<=j<=N} of real;
let
  c[i,j] = case
       {| j=0} : 0[];
       {| 1<=j<=N} : C[i,j-1] + A[i,j]*V[j];
        esac;
  C[i] = c[i,N];

tel;
```

Définition des delais

Alpha nous permet également de définir des délais sur les signaux. Pour cela on peut décrire un algorithme en interprétant les indices comme le temps.

Exemple 2.3 : Une addition avec délai comme définie à la Figure 2.1 pourrait être décrite par le système *adderDelayed* suivant :

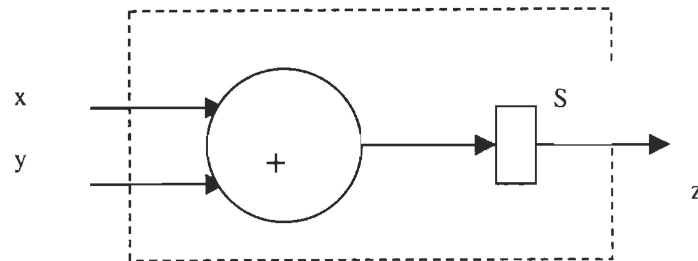


Figure 2.1 : Additionneur avec retard

```
system adderDelayed (x,y : {t|1<=t} of integer)
returns (z : {t|2<=t} of integer);
var S : {t|2<=t} of integer;
let
  S[t] = (x + y)[t-1];
z = S;
tel
```

2.2 Le langage Alpha et son environnement MMAlpha

L'objectif de Alpha est de donner aux concepteurs de circuits intégrés un outil de haut niveau pour la synthèse d'architectures parallèles VLSI. Mais Alpha peut également apporter la solution à des problèmes dans d'autres domaines comme : parallélisation, génération de code, théorie des polyèdres, réseaux systoliques, etc.

Dans Alpha, un algorithme est défini comme un ensemble d'équations sur des variables définies sur des domaines multidimensionnels. Chaque variable ou expression est en fait une fonction d'un ensemble de coordonnées entières satisfaisant des inégalités linéaires, vers un ensemble de valeurs. Le processus de synthèse consiste à appliquer une

série de transformations préservant la sémantique qui traduisent la spécification initiale de l'algorithme vers une architecture supportant son exécution. La description finale peut être traduite en VHDL pour ensuite générer une architecture VLSI. Les outils nécessaires pour faire ces transformations sont intégrés comme un ensemble package Mathematica® des librairies C dans l'environnement MMAlpha.

Le processus de conception commence par une description au niveau algorithmique de l'application. Cette description est une traduction directe des équations mathématiques dans le langage Alpha. Cette description peut être structurée hiérarchiquement comme on le ferait en programmation structurée classique, où les algorithmes d'algèbre linéaires classiques comme les multiplication matrice-matrice et matrice-vecteur sont d'abord décrites comme des systèmes indépendants utilisés dans l'application. Un programme C qui évalue cette description peut être généré de façon automatique pour vérifier par simulation l'exactitude des spécifications initiales. Ensuite cette description initiale subit une série de transformations donc certaines sont résumées à la section 2.3, pour délivrer une architecture abstraire. Parmi ces transformations, les plus importantes sont la localisation et l'ordonnancement. La localisation (aussi appelée `uniformisation` ou `pipeline` dans la littérature) remplace les opérations non-locales par des opérations locales. L'ordonnancement ordonne les opérations de telle sorte que l'évaluation d'une variable donnée soit faite avant celle de ses composants. L'ordonnancement résout un problème de programmation linéaire entière dont les inconnues sont les coefficients de la fonction affine qui définit le temps où chaque variable est évaluée [BAL98]. Par exemple, une variable V[i,j] sera évaluée à l'instant ai+bj+c, et les inconnues ici sont les coefficients a, b, et c.

L'ordonnancement donne également des informations intéressantes sur le temps total nécessaire à l'exécution de l'algorithme. Dès qu'un ordonnancement est trouvé, un changement de base est fait pour permettre à toutes les opérations d'être exprimées en terme de nouveaux indices donnant le temps d'évaluation et le numéro du processeur où le calcul est effectué.

A partir de cette architecture abstraite, la conception de l'architecture consiste à l'application d'une série de transformations de bas niveau qui modifie la description vers une description de type netlist appelée **AlpHard** [MOE96]. Ce processus de transformation est presque automatisé, l'environnement MMAlpha se comportant comme un compilateur qui traduit automatiquement un niveau de description vers le niveau suivant. Éventuellement, on peut obtenir le modèle VHDL synthétisable pour l'intégration dans une technologie VLSI (CMOS, FPGA) [MOE96].

2.3 Procédure de dérivation d'une architecture parallèle

La méthodologie de conception dans MMAlpha est appliquée à l'algorithme pour obtenir un ordonnancement des sorties et des variables internes, ainsi que la netlist du réseau de processeurs qui définit l'architecture. Les équations de l'algorithm sont écrites en Alpha et analysées pour éliminer certaines erreurs et pour vérifier les domaines des équations par analyse statique. Ensuite, on réduit tous les sous-systèmes [API97] : cette transformation revient à faire éclater toutes les expressions structurées de Alpha pour que le programme résultant puisse être traduit en C avec le traducteur `WriteC`. Le programme

C obtenu est exécuté et les résultats sont vérifiés pour s'assurer que l'algorithme effectue toujours la même fonction. Dans notre cas, nous avons comparé les résultats avec ceux obtenus lors des simulations sous Matlab®.

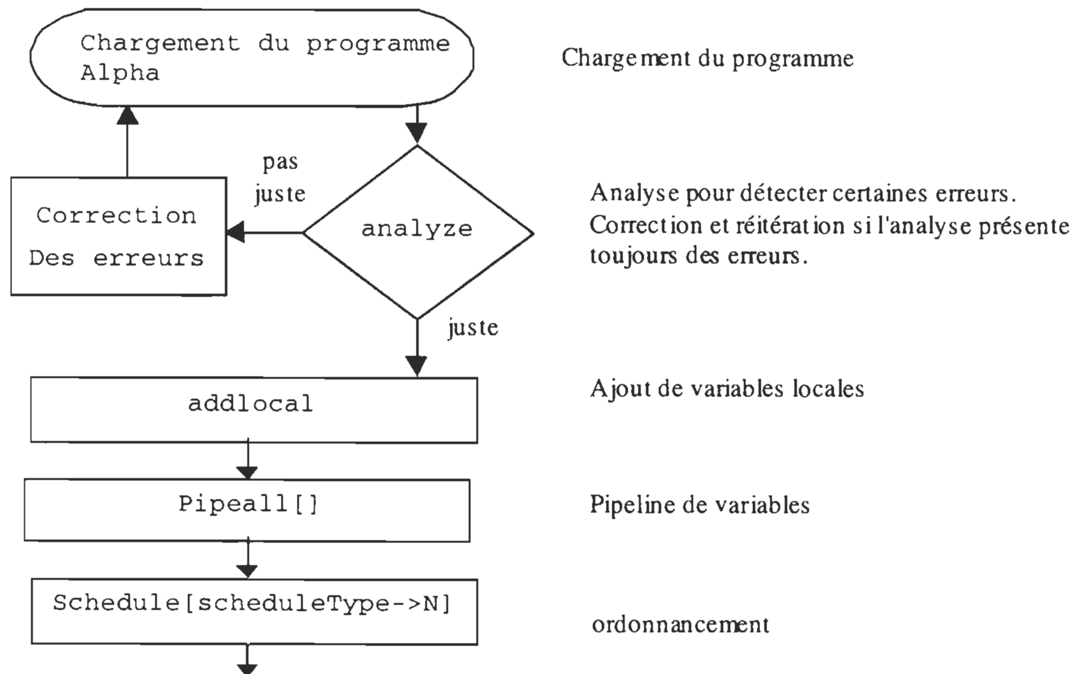
Après ces transformations préliminaires, la synthèse effective de l'architecture peut commencer. Premièrement, on pipeline certaines variables diffusées le long des directions données. Les opérations qui subissent ce processus de pipelinage sont les multiplications matrice-matrice, matrice-vecteur et vecteur-vecteur. Après le pipelinage, on peut rechercher un ordonnancement pour le programme Alpha. Le but de l'ordonnanceur est de trouver un ordre d'exécution des équations en respectant les dépendances entre les calculs. Le temps est considéré discret comme une horloge, c'est-à-dire constitué des unités et des sous-unités. L'idée globale du processus d'ordonnancement est de construire un problème de programmation linéaire et de le résoudre avec un logiciel particulier [API97]. L'ordonnancement se fait automatiquement et le résultat est rendu explicite en appliquant un réindixage sur le programme Alpha. Ensuite les signaux de contrôle sont générés et on dérive l'architecture au niveau registre de transfert en AlpHard. Enfin, on génère le code VHDL à l'aide de `AlphatoVHDL`.

Toutes les commandes citées ici sont présentées dans le cours exemple suivant qui montre comment on peut dériver une architecture systolique et le code VHDL pour la multiplication matrice-vecteur décrite à la section 2.1.

2.4 Synthèse d'une architecture parallèle pour le produit matrice-vecteur

Nous présentons à la Figure 2.2 les principales étapes de dérivation d'architectures systoliques avec MMAlpha. Ces étapes sont appliquées à un exemple de base qui est le produit matrice-vecteur et présentées en annexe II.

Le choix du produit matrice-vecteur est motivé par le fait que c'est la base de toutes les opérations élémentaires dans l'algorithme du filtre de Kalman en particulier et dans tous les algorithmes de traitement du signal en général. Ces étapes seront appliquées au chapitre 4 au filtre de Kalman. Elles figurent également dans [MOZ98] dont une copie est donnée en annexes I.



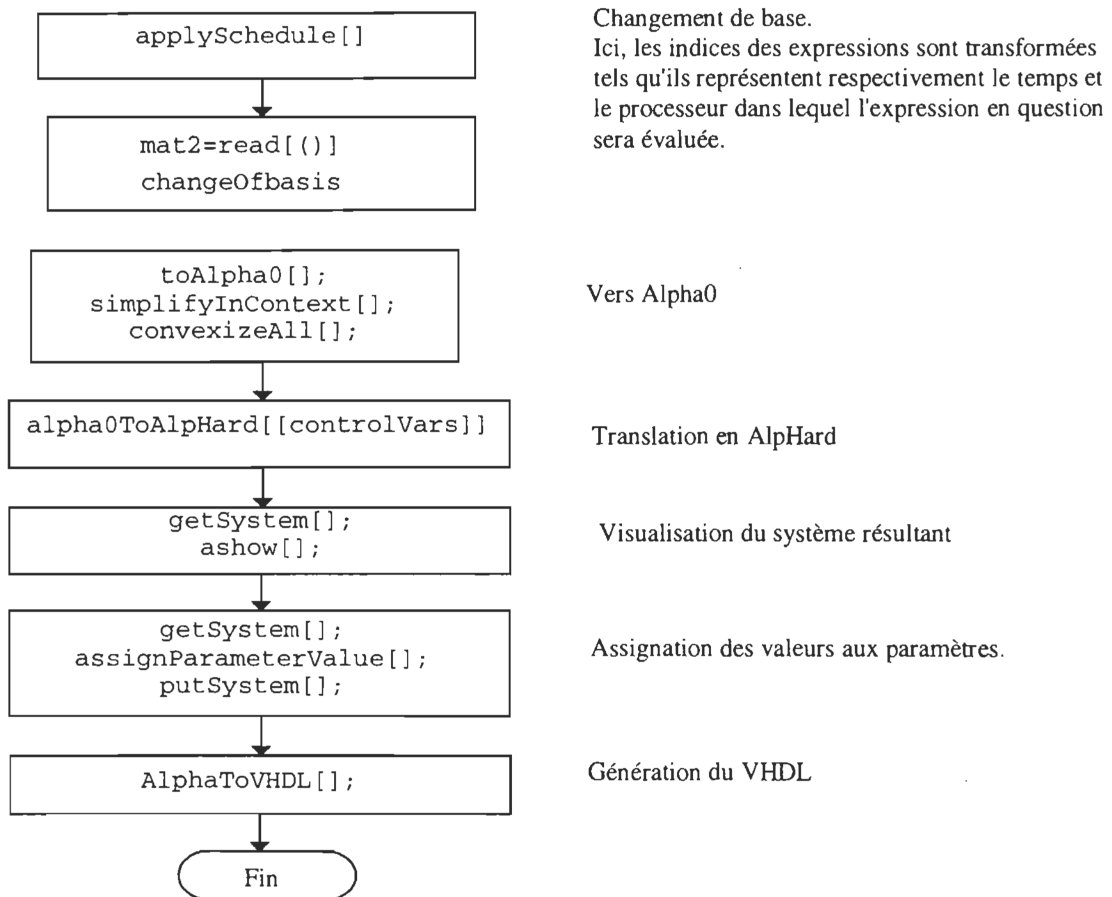


Figure 2.2 : Principales étapes de dérivation d'une architecture parallèle avec MMAlpha

Chapitre 3

Filtre de Kalman et Applications

Les domaines de la communication, du traitement du signal et de la commande ont en commun le besoin d'algorithmes de reconstitution performants pour estimer un signal inconnu altéré soit par les milieux dans lesquels ces signaux ont été propagés, par exemple les appareils de conversion composés d'instruments et de capteurs. Les signaux reçus à travers ces médiums sont corrompus selon une loi déterministe et également selon une loi stochastique. Des exemples particuliers peuvent être les communications par satellite ou par téléphone cellulaire où le médium de transmission ici est l'atmosphère, un câble de transmission dans une communication filaire ou un modem dans une communication par modem. Afin d'utiliser ces signaux, il faut les faire passer dans des algorithmes de reconstitution ou d'estimation, pour en extraire les valeurs réelles. Les nouvelles techniques de l'information qui engendrent les domaines principaux dans lesquels ces signaux sont utilisés sont de plus en plus exigeants quand à l'exactitude et la vitesse de fonctionnement des appareils utilisés. Pour cela, il faut développer des algorithmes de

reconstitution/estimation extrêmement rapides pour satisfaire aux exigences des circuits utilisateurs.

Les algorithmes de reconstitution et d'estimation sont choisies en fonction de l'application, des caractéristiques du canal et surtout de la qualité (taux d'erreur de reconstitution) de la réponse finale désirée. Ces algorithmes de reconstitution sont nombreux en traitement du signal et leurs performances dépendent du type de signal et bruit à filtrer. La méthode directe de déconvolution (qui est l'opération inverse de la convolution) a le très grand désavantage d'amplifier le bruit en hautes fréquences. On peut résoudre ce problème en utilisant une déconvolution avec un paramètre de régularisation. Mais, les filtres les plus performants dans ce genre d'opérations sont les filtres adaptatifs qui ont l'avantage sur les filtres non adaptatifs (filtres RIF, RII, filtres non linéaires, etc.) de s'adapter automatiquement aux paramètres du bruit et du canal variant. Le filtre de Kalman est une estimateur linéaire optimal qui peut reconstruire l'état d'un système à partir des données mesurées et dans un environnement stochastique [MOZ98], [MOZ99]. Il trouve son application dans presque tous les domaines.

3.1 Le filtre de Kalman

L'algorithme du filtre de Kalman a été développé en 1960 par R.E. Kalman [KAL60], et depuis lors, il est devenu un outil très puissant dans tous les domaines où le filtrage, le lissage et la prédiction des valeurs d'un signal sont nécessaires. En particulier, il est très utilisé en traitement du signal (filtrage numérique, reconstruction de mesurande,

traitement des signaux des radars), communication (égalisation des canaux) et en commande (suivi des trajectoires, la prédiction des cibles, estimation de l'état d'un système, navigation maritime GPS).

Le filtre de Kalman a été utilisé dans [MAS95] pour la correction des données spectrométriques pour améliorer la résolution d'un spectromètre. Le problème ici est d'éliminer les erreurs statiques introduites lors de la mesure par le spectromètre de même que les erreurs aléatoires (bruit) qui s'introduisent dans la valeur de la mesure. L'approche ici a été d'introduire une contrainte de positivité dans l'algorithme du filtre de Kalman stationnaire. Notre algorithme de filtrage utilisera la structure des matrices proposées ici, mais nous considérerons un système à réponse impulsionnelle variant.

Le filtre de Kalman a été également utilisé dans [MEI84] en chromatographie pour éliminer les erreurs de mesure dans les chromatogrammes. Nous présentons ci-dessous deux applications différentes que nous avons étudiées, à savoir l'égalisation des canaux et l'estimation d'état d'un système dynamique.

3.1.1 Principe de filtrage

Soit un système linéaire dynamique et variant défini par son équation d'état discrète suivante :

$$\mathbf{x}(k+1) = \mathbf{\Phi}(k)\mathbf{x}(k) + \mathbf{b}(k)\mathbf{w}(k), \ \ \mathbf{x}(0) = \mathbf{0} \tag{3.1}$$

$$\tilde{\mathbf{y}}(k) = \mathbf{Ç}(k)\mathbf{x}(k) + \mathbf{v}(k) \tag{3.2}$$

pour k=1,2,... et où $\mathbf{x}(k)$ est le vecteur d'état de dimension $(M \times 1)$, $\mathbf{y}(k)$ est le vecteur des mesures (entachées de bruit) de dimension $(m \times 1)$ $(m \leq M)$. $\mathbf{v}$ et $\mathbf{w}$ sont deux séquences non corrélées de bruit blanc et de matrices de covariances respectives $\mathbf{R}_v(k)$ et $\mathbf{R}_w(k)$.

Le filtre de Kalman calcule de façon récursive la prédiction du vecteur d'état $\mathbf{x}$ en fonction de chaque nouvelle séquence de la mesure entachée de bruit $\tilde{\mathbf{y}}(k)$ [IRW91]. Ce filtre est basé sur la minimisation de la valeur moyenne du carré de l'erreur d'estimation. Le filtre de Kalman agit donc comme un reconstructeur d'état en minimisant la covariance de l'erreur d'estimation [SIC97].

Les hypothèses suivantes doivent êtres faites sur la nature des bruits. Les variables $\mathbf{v}(k)$ et $\mathbf{w}(k)$ sont non corrélées et de distribution Gaussiennes ayant les propriétés suivantes :

$$E[\mathbf{w}(k)] = 0, \quad E[\mathbf{v}(k)] = 0 \tag{3.3}$$

$$\operatorname{cov}[\mathbf{w}(j), \mathbf{w}(k)] = E\left[\mathbf{w}(j)\mathbf{w}^T(k)\right] = \mathbf{R}_w \delta_{jk} \tag{3.4}$$

$$\operatorname{cov}[\mathbf{v}(j), \mathbf{v}(k)] = E\left[\mathbf{v}(j)\mathbf{v}^T(k)\right] = \mathbf{R}_v \delta_{ik} \tag{3.5}$$

E[.] est l'espérance mathématique, cov[.] est la covariance et δ_{jk} est le symbole de Kronecker défini par :

$$\delta_{jk} = \begin{cases} 0, & j \neq k \\ 1, & j = k \end{cases} \tag{3.6}$$

Dans ce cas, on dit que $\mathbf{w}(k)$ et $\mathbf{v}(k)$ sont des processus aléatoires gaussiens blancs. En plus, $\mathbf{R}_w(k)$ et $\mathbf{R}_v(k)$ doivent être symétriques et définies positives, ce qui a pour conséquence que leurs racines carrées définies par les facteurs de Choleski existent. Cette condition est nécessaire pour pouvoir définir les versions racines carrées du filtre de Kalman [KAM71].

Nous utiliserons dans la suite les notations suivantes :

Soit $\hat{\mathbf{x}}(i/j)$ l'état estimé à l'instant i tenant compte de la mesure et des autres informations connues à l'instant j. L'erreur d'estimation est définie par :

$$\mathbf{e}(k/k) = \mathbf{x}(k) - \hat{\mathbf{x}}(k/k) \tag{3.8}$$

Et la matrice de covariance de l'erreur d'estimation est définie par :

$$\mathbf{P}(k/k) = E\left[\mathbf{e}(k/k)\mathbf{e}^T(k/k)\right] = E\left[(\mathbf{x}(k) - \hat{\mathbf{x}}(k/k))(\mathbf{x}(k) - \hat{\mathbf{x}}(k/k))^T\right] \tag{3.8}$$

De la même manière, l'erreur de prédiction est

$$\mathbf{e}(k+1/k) = \mathbf{x}(k+1) - \hat{\mathbf{x}}(k+1/k) \tag{3.9}$$

La matrice de covariance de l'erreur de prédiction est définie par :

$$\mathbf{P}(k+1/k) = E\left[(\mathbf{x}(k+1) - \hat{\mathbf{x}}(k+1/k))(\mathbf{x}(k+1) - \hat{\mathbf{x}}(k+1/k))^T\right] \tag{3.10}$$

Compte tenu des propriétés (3.3) et (3.5), les éléments de la diagonale dans l'équation (3.8) sont donc la moyenne des carrées de l'erreur d'estimation tandis que les

éléments non diagonaux sont symétriques. Le Filtre de Kalman minimise la fonction de coût suivante [KAM71] :

$$J(k) = tr(\mathbf{P}(k/k)) = E\left[\mathbf{e}_1^2(k)\right] + E\left[\mathbf{e}_2^2(k)\right] + \cdots + E\left[\mathbf{e}_n^2(k)\right] \tag{3.11}$$

Le développement de cette équation conduit à quatre versions différentes du filtre de Kalman [KAM71] qui sont : filtre de covariance (encore appelé filtre de Kalman standard), filtre d'information, filtre racine carrée de covariance et filtre racine carrée d'information.

3.1.2 Filtre de covariance

Le filtre de Kalman conventionnel plus connu sous le nom de filtre de covariance (ou filtre de covariance standard) est donné par les équations suivantes [IRW91] :

$$\hat{\mathbf{x}}(k+1/k) = \mathbf{\ddot{O}}(k)\hat{\mathbf{x}}(k/k), \ \hat{\mathbf{x}}(0/0) = \mathbf{0} \tag{3.12}$$

$$\mathbf{P}(k+1/k) = \mathbf{\ddot{O}}(k)\mathbf{P}(k/k)\mathbf{\ddot{O}}^T(k) + \mathbf{B}(k)\mathbf{R}_w(k)\mathbf{B}^T(k), \quad \mathbf{P}(0/0) = \mathbf{I} \tag{3.13}$$

$$\mathbf{V}_e(k) = \mathbf{Ç}(k)\mathbf{\tilde{N}}(k+1/k)\mathbf{Ç}^T(k) + \mathbf{R}_v(k) \tag{3.14}$$

$$\mathbf{\hat{E}}(k+1) = \mathbf{\tilde{N}}(k+1/k)\mathbf{Ç}^T(k+1)\mathbf{V}_e^{-1}(k+1) \tag{3.15}$$

$$\hat{\mathbf{x}}(k+1/k+1) = \hat{\mathbf{x}}(k+1/k) + \mathbf{K}(k+1)\left[\tilde{\mathbf{y}}(k+1) - \mathbf{H}(k+1)\hat{\mathbf{x}}(k+1/k)\right] \tag{3.16}$$

$$P(k+1/k+1) = P(k+1/k) - K(k+1)H(k+1)P(k+1/k) \tag{3.17}$$

où $\hat{\mathbf{x}}(k+1/k+1)$ est l'estimé du vecteur d'état, $\hat{\mathbf{x}}(k+1/k)$ est la prédiction du vecteur d'état , $\mathbf{P}(k+1/k+1)$ est la matrice de covariance de l'erreur d'estimation de dimension $M \times M$, $\mathbf{P}(k+1/k)$ est la matrice de covariance de l'erreur de prédiction de dimension $M \times M$, $\mathbf{K}(k+1)$ sont les gains de Kalman de dimension $M \times 1$, et $\mathbf{V}_e(k)$ est une variable intermédiaire de dimension $m \times m$.

La Figure 3.1 schématise les équations de mise à jour de l'état (3.12) et de la mesure (3.15) précédentes ; le calcul des gains n'est pas montré sur cette figure.

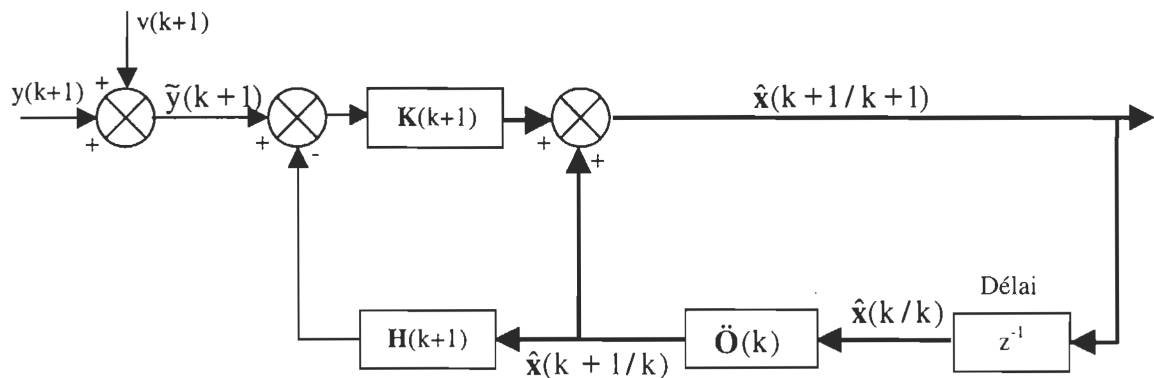


Figure 3.1 : Diagramme bloc du filtre de Kalman.

Lorsque le système est invariant, les matrices d'état ne dépendent pas de l'instant k et les gains $\mathbf{K}(k)$ peuvent être précalculés pour obtenir les gains de Kalman stationnaires, $\mathbf{K}_\infty$, avant d'être injectés dans l'algorithme. Ainsi, le système est adaptatif dès le premier échantillon. En utilisant les valeurs des matrices de covariances et des matrices d'autocorrelation, on peut calculer $\mathbf{K}_\infty$ comme suit :

$$\mathbf{K}_{\infty} = \lim_{i \to \infty} \mathbf{K}(i) \tag{3.18}$$

Le nombre d'itérations nécessaires pour atteindre la convergence est environ égal à n, la dimension du système. La convergence est donc très rapide lorsqu'on calcule à l'avance les gains de Kalman.

On obtient par conséquent les équations suivantes appelées filtre de Kalman stationnaire [MAS95] :

$$\hat{\mathbf{x}}(k+1/k) = \mathbf{\Phi}(k)\hat{\mathbf{x}}(k/k) \tag{3.19}$$

$$\hat{\mathbf{x}}(k+1/k+1) = \hat{\mathbf{x}}(k+1/k) + \mathbf{K}_{\infty}\left[\tilde{\mathbf{y}}(k+1) - \mathbf{H}(k+1)\hat{\mathbf{x}}(k+1/k)\right] \tag{3.20}$$

3.1.3 Filtre d'information

Dans la version standard du filtre de Kalman, la matrice de covariance est propagée d'un échantillon à l'autre. Le filtre conventionnel peut également être implanté pour propager l'inverse de la matrice de covariance $\mathbf{P}^{-1}$, appelée matrice d'information, accentuant ainsi la nature filtrage par moindres carrées récursives [KAM71]. Les équations suivantes décrivent une formulation du filtre d'information :

$$\mathbf{d}(k+1/k) = \left[\mathbf{I} - \mathbf{L}(k)\mathbf{B}^{\mathbf{T}}(k)\right]\mathbf{\Phi}^{-\mathbf{T}}(k)\mathbf{d}(k/k) \tag{3.21}$$

$$\mathbf{P}^{-1}(k+1/k) = \left[\mathbf{I} - \mathbf{L}(k)\mathbf{B}^{\mathbf{T}}(k)\right]\mathbf{F}(k) \tag{3.22}$$

$$\mathbf{F}(k) = \mathbf{\Phi}^{-\mathbf{T}}(k)\mathbf{P}^{-1}(k/k)\mathbf{\Phi}^{-1}(k) \tag{3.23}$$

$$\mathbf{L}(k) = \mathbf{F}(k)\mathbf{B}(k)\left[\mathbf{R}_{\mathbf{w}}^{-1}(k) + \mathbf{B}^{\mathbf{T}}(k)\mathbf{F}(k)\mathbf{B}(k)\right]^{-1} \tag{3.24}$$

$$\mathbf{d}(k+1/k+1) = \mathbf{d}(k+1/k) + \mathbf{H}^{T}(k+1)\mathbf{R}_{v}^{-1}(k+1)\tilde{\mathbf{y}}(k+1) \tag{3.25}$$

$$\mathbf{P}^{-1}(k+1/k+1) = \mathbf{P}^{-1}(k+1/k) + \mathbf{H}^{T}(k+1)\mathbf{R}_{v}^{-1}(k+1)\mathbf{H}(k+1) \tag{3.26}$$

où

$$\mathbf{d}(k+1/k) \stackrel{\Delta}{=} \mathbf{P}^{-1}(k+1/k)\hat{\mathbf{x}}(k+1/k) \tag{3.27}$$

$$\mathbf{d}(k+1/k+1) \stackrel{\Delta}{=} \mathbf{P}^{-1}(k+1/k+1)\hat{\mathbf{x}}(k+1/k+1) \tag{3.28}$$

La notation Φ^{-T} veut dire $(\Phi^{-1})^{T}$ qui est équivalent à $(\Phi^{T})^{-1}$.

Remarquons que, bien que $\mathbf{P}^{-1}$ apparaisse dans les équations, il n'est pas nécessaire d'inverser $\mathbf{P}$ chaque fois que qu'il apparaît. En effet, on calcule directement $\mathbf{P}^{-1}$ et on le propage dans tout l'algorithme. Pour obtenir $\hat{\mathbf{x}}$ à la fin de l'itération, on inverse $\mathbf{P}^{-1}$ et on utilise la formule (3.28).

L'avantage de ce filtre est qu'il permet d'estimer l'état sans connaître *à priori* les informations sur l'état initial du système, en posant tout simplement $\mathbf{P}^{-1}(0/0) = \mathbf{I}$ et $\mathbf{d}(0/0)=0$.

Une autre formulation du filtre d'information est présentée dans [FAY95]. Soient les vecteurs d'information $\mathbf{z}(i/j)$ définis comme suit :

$$\mathbf{z}(i/j) \overset{\Delta}{=} \mathbf{P}^{-1}(i/j)\hat{\mathbf{x}}(i/j) \tag{3.29}$$

Soit également la matrice d'information :

$$\mathbf{Z}(i/j) \overset{\Delta}{=} \mathbf{P}^{-1}(i/j). \tag{3.30}$$

Dans les deux cas précédents, si i=j=k+1, c'est l'estimation et si i=k+1 et j=k alors c'est la prédiction.

Soit la contribution des vecteurs d'information au vecteur de mesure :

$$\mathbf{i}_i(k) \overset{\Delta}{=} \mathbf{H}_i^T(k)(\mathbf{R}_v^{-1})_i(k)\mathbf{y}_i(k) \tag{3.31}$$

Soit la contribution de la matrice de covariance :

$$\mathbf{I}_i(k) \overset{\Delta}{=} \mathbf{H}_i^T(k)(\mathbf{R}_v^{-1})_i(k)\mathbf{H}_i(k) \tag{3.32}$$

Les équations du filtre de Kalman dans un système décentralisé sont définies comme suit :

Prédiction :

$$\mathbf{Z}(k+1/k) = \left[\mathbf{F}(k+1)\mathbf{Z}^{-1}(k/k)\mathbf{F}^T(k+1) + \mathbf{R}_w(k)\right]^{-1} \tag{3.33}$$

$$\mathbf{z}(k+1/k) = \mathbf{Z}(k+1/k)\mathbf{F}(k+1)\mathbf{Y}^{-1}(k/k)\mathbf{z}(k/k) \tag{3.34}$$

Estimation :

$$\mathbf{Z}(k+1/k+1) = \mathbf{Z}(k+1/k) + \sum_{i=1}^{m} \mathbf{I}_i(k) \quad (3.35)$$

$$\mathbf{z}(k+1/k+1) = \mathbf{z}(k+1/k) + \sum_{i=1}^{m} \mathbf{i}_i(k) \quad (3.36)$$

m représente le nombre de nœuds capteurs.

Cette version du filtre de Kalman permet la décentralisation des calculs vers les nœuds capteurs dans les systèmes où la mesure est multidimensionnelle tout en préservant la stabilité lorsque peu ou pas d'informations à priori sont connues [FAY95].

3.1.4 Filtre racine carrée de covariance

L'idée de base des filtres racine carrée (square root covariance Kalman filter et square root information Kalman filter) est le remplacement des matrices de covariance et d'information par leurs racines carrées respectives, pour améliorer la robustesse des filtres face aux effets de quantification.

Soient **S**, **U** et **V** des matrices définies par les expressions suivantes:

$$\mathbf{P} \overset{\Delta}{=} \mathbf{S}\mathbf{S}^T \quad (3.37)$$

$$\mathbf{R}_w = \mathbf{U}\mathbf{U}^T \quad (3.38)$$

$$\mathbf{R}_v = \mathbf{V}\mathbf{V}^T \quad (3.39)$$

Les matrices **S**, **U** et **V** représentent les racines carrées des matrices **P**, $\mathbf{R_w}$ et $\mathbf{R_v}$ respectivement et peuvent être obtenues par la décomposition de Choleski suivante [KAM71]:

$$\mathbf{A} = \mathbf{A}^{1/2}\mathbf{A}^{T/2} \tag{3.40}$$

où **A** est une matrice symétrique définie semi-positive. **S**, **U** et **V** sont donc des matrices triangulaires inférieures ; ce sont **S**(k+1/k+1) et **S**(k+1/k) qui seront propagées dans l'algorithme de filtrage.

Les équations du filtre racine carrée de covariance sont les suivantes :

$$\hat{\mathbf{x}}(k+1/k) = \Phi(k)\hat{\mathbf{x}}(k/k) \tag{3.41}$$

$$\begin{bmatrix} \mathbf{S}^T(k+1) \\ \hline \mathbf{0} \end{bmatrix} = \mathbf{T}\begin{bmatrix} \mathbf{S}_+^T(k)\Phi^T(k) \\ \hline \mathbf{U}^T(k)\mathbf{B}^T(k) \end{bmatrix} \tag{3.42}$$

$$\hat{\mathbf{x}}(k+1/k+1) = \hat{\mathbf{x}}(k+1/k) + K(k+1)\left[\tilde{y}(k+1) - H(k+1)\hat{\mathbf{x}}(k+1/k)\right] \tag{3.43}$$

$$\mathbf{S}(k+1/k+1) = \mathbf{S}(k+1/k) - \gamma(k+1)\mathbf{K}(k+1)\mathbf{F}^{\mathbf{T}}(k+1) \tag{3.44}$$

$$\mathbf{K}(k+1) = a(k+1)\mathbf{S}(k+1/k)\mathbf{F}(k+1) \tag{3.45}$$

$$\mathbf{F}(k+1) = \mathbf{S}^T(k+1/k)\mathbf{H}^T(k+1) \tag{3.46}$$

$$a(k) = \frac{1}{\mathbf{F}^T(k)\mathbf{F}(k) + \sigma_v^2(k)} \tag{3.47}$$

$$\gamma(k) = \frac{1}{1+\sqrt{a(k)\sigma_v^2(k)}} \tag{3.48}$$

Les équations (3.43) à (3.48) sont valables lorsque la mesure est un scalaire, ainsi σ_v^2 et σ_w^2 sont les variances des variables aléatoires v(k) et w(k) lorsque la mesure est monodimensionnelle. Lorsque le vecteur de mesure est multidimensionnel, ces étapes sont répétées m fois, m étant la dimension du vecteur de mesure $\tilde{\mathbf{y}}(k)$. Cette version améliore la stabilité numérique de la procédure.

Les équations suivantes donnent une autre formulation pour les équations (3.43) à (3.48) et sont bien appropriées pour une dérivation d'une architecture systolique [KUN82], [KAM71] :

$$\hat{\mathbf{x}}(k+1/k+1) = \hat{\mathbf{x}}(k+1/k) + \left(\mathbf{G}^T(k+1)/F(k+1)\right)\left[\tilde{\mathbf{y}}(k+1) - \mathbf{H}(k+1)\hat{\mathbf{x}}(k+1/k)\right] \tag{3.49}$$

$$\begin{bmatrix} F(k+1) & \mathbf{G}(k+1) \\ \mathbf{0} & \mathbf{S}^T(k+1/k+1) \end{bmatrix} = \mathbf{T}\begin{bmatrix} \mathbf{V}(k+1) & \mathbf{0} \\ \mathbf{S}^T(k+1/k)\mathbf{H}^T(k+1) & \mathbf{S}^T(k+1/k) \end{bmatrix} \tag{3.50}$$

La matrice $\mathbf{T}$ est une matrice de triangularisation linéaire orthogonale choisie telle que la matrice de gauche soit triangulaire supérieure. Le choix de l'algorithme de triangularisation est assez important dans la mesure où il est déterminant sur le degrés de parallélisme que l'on obtiendra au moment de la dérivation de l'architecture systolique.

Les principaux algorithmes de triangularisation rencontrés dans la littérature ont été développés par Gram-Schmidt, HouseHolder et Givens. Les rotations de Givens ont été très

utilisées pour obtenir des architectures systoliques basées sur les algorithmes de filtrage par LMS, RLS et aussi sur le filtre de Kalman. Dans le Chapitre 4, nous étudierons plus en détail l'algorithme de construction des matrices **T** par les rotations de Givens.

3.1.5 Filtre racine carrée de l'information

Le filtre racine carrée de l'information (*Square root information filter*) est la version la plus utilisée lorsque la matrice d'état ne présente aucune singularité. Il met plus d'accent sur le filtrage par moindres carrées. Les résultats numériques de l'approche par racine carrée font qu'elle est la meilleure approche pour résoudre le problème de filtrage par moindres carrées.

La fonction de coût à minimiser ici est la suivante tel que définit dans [KAM71]:

$$J(k) = \|\mathbf{x}(k) - \mathbf{x}(k/k-1)\|^2 \mathbf{P}^{-1}(k/k-1) + \|\tilde{y}(k) - \mathbf{H}(k)\mathbf{x}(k)\|^2 \mathbf{R}_v^{-1}(k) \qquad (3.51)$$

Le développement de l'équation (3.51) conduit aux formulations suivantes pour la version racine carrée de l'information:

$$\mathbf{b}(k+1/k) = \mathbf{b}(k/k) - a(k)\gamma(k)\mathbf{F}(k)\mathbf{F}^T(k)\mathbf{b}(k/k) \qquad (3.52)$$

$$\mathbf{S}^{-1}(k+1/k) = \mathbf{S}^{-1}(k/k)\mathbf{\Phi}(k) - \gamma(k)\mathbf{F}(k)\mathbf{L}(k) \qquad (3.53)$$

$$\mathbf{L}(k) = a\mathbf{F}^T(k)\mathbf{S}^{-1}(k/k)\mathbf{\Phi}^{-1}(k) \qquad (3.54)$$

$$\mathbf{F}(k) = \mathbf{S}^{-1}(k/k)\mathbf{\Phi}^{-1}(k)\mathbf{B}(k) \qquad (3.55)$$

$$a(k) = \frac{1}{\mathbf{F}^T(k)\mathbf{F}(k) + 1/\sigma_w^2(k)} \tag{3.56}$$

$$\gamma(k) = \frac{1}{1 + \sqrt{a(k)/\sigma_w^2(k)}} \tag{3.57}$$

σ_w^2 est la variance de la variable aléatoire w(k) et

$$\begin{bmatrix} \mathbf{S}^{-1}(k+1/k+1) \\ \mathbf{0} \end{bmatrix} = \mathbf{T} \begin{bmatrix} \mathbf{S}^{-1}(k+1/k) \\ \mathbf{V}^{-1}(k)\mathbf{H}(k) \end{bmatrix} \tag{3.58}$$

$$\begin{bmatrix} \mathbf{b}(k+1/k+1) \\ \mathbf{e}(k+1) \end{bmatrix} = \mathbf{T} \begin{bmatrix} \mathbf{b}(k+1/k) \\ \mathbf{V}^{-1}(k)\tilde{\mathbf{y}}(k) \end{bmatrix} \tag{3.59}$$

Les équations (3.52) à (3.57) peuvent être résumées par les deux équations suivantes:

$$\begin{bmatrix} F(k+1) & \mathbf{G}(k+1) \\ \mathbf{0} & \mathbf{S}^{-1}(k+1/k) \end{bmatrix} = \mathbf{T} \begin{bmatrix} 1/\mathbf{U}(k+1) & \mathbf{0} \\ \mathbf{S}^{-1}(k/k)\Phi^{-1}(k)\mathbf{B}(k) & \mathbf{S}^{-1}(k/k)\Phi^{-1}(k) \end{bmatrix} \tag{3.60}$$

$$\begin{bmatrix} \mathbf{a}(k+1) \\ \mathbf{b}(k+1/k) \end{bmatrix} = \mathbf{T} \begin{bmatrix} \mathbf{0} \\ \mathbf{b}(k/k) \end{bmatrix} \tag{3.61}$$

avec

$$\mathbf{b}(k/k) = \mathbf{S}^{-1}(k/k)\hat{\mathbf{x}}(k/k) \tag{3.62}$$

et

$$\mathbf{b}(k+1/k) = \mathbf{S}^{-1}(k+1/k)\hat{\mathbf{x}}(k+1/k) \tag{3.63}$$

Cette version plus compacte, est plus appropriée pour la dérivation d'une architecture systolique.

3.2 Applications du filtre de Kalman à l'égalisation des canaux

Un canal est un milieu à travers lequel un signal est transmis, et récupéré à la sortie avec des modifications qui dépendent des caractéristiques du milieu (réponse impulsionnelle), nous pouvons citer en exemple :

- L'atmosphère ; lors des communications par satellite et lors des communications par téléphone cellulaire, le signal original émis par le satellite transverse l'atmosphère et arrive au récepteur corrompu et entaché par un autre bruit indéterminé.

- Un modem constitue également un canal composé d'un système de communication numérique, de la modulation et de la démodulation d'un signal, tous introduisent un bruit aléatoire qui est difficile à séparer du signal original.

- Les fibres optiques et les câbles de communications produisent également les effets décrits précédemment sur les signaux qu'ils transmettent.

- Les câbles de communication dans un réseau local (*Local Area Network*) présentent une bande en fréquence limitée et des réflexions qui ont pour effet de détériorer la polarité des signaux numériques qu'ils communiquent.

Le modèle général d'un égalisateur adaptatif de canal est donné à la Figure 3.2. Ce schéma est celui d'un algorithme de filtrage adaptatif qui nécessite la connaissance du signal original comme les algorithmes LMS, RLS et les réseaux de neurones (lors de la phase d'adaptation, d'étalonnage ou d'apprentissage).

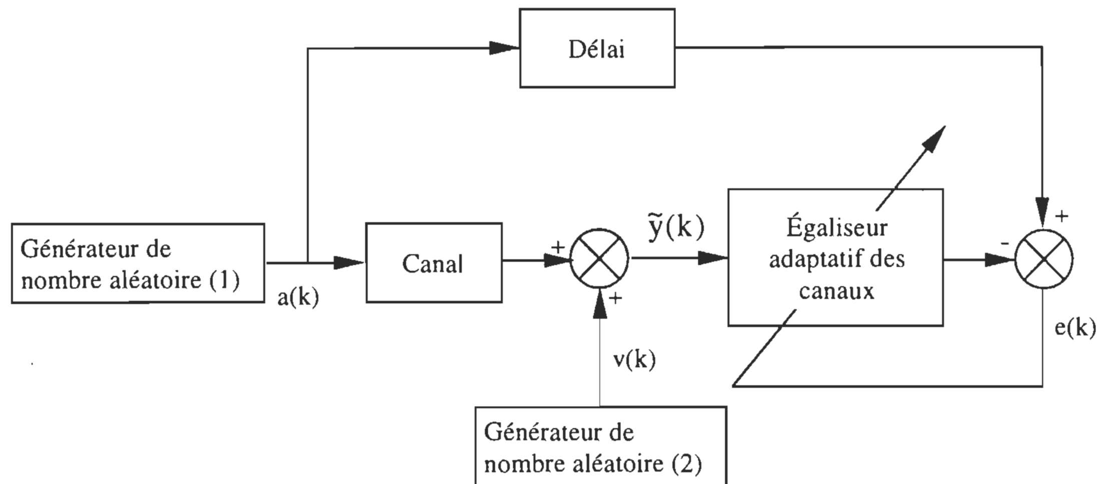


Figure 3.2 : Égalisation adaptative des canaux par filtrage adaptatif [HAY96].

La différence du filtre de Kalman par rapport aux méthodes basée sur la Figure 3.2 est qu'il ne nécessite pas la connaissance à priori des informations sur le signal transmis. Par contre, il nécessite la connaissance de la réponse impulsionnelle du canal. Dans notre projet, nous utilisons le filtre de Kalman pour l'égalisation adaptative d'un canal dispersif linéaire qui produit une distorsion déterministe (réponse impulsionnelle **H**(k)) en présence d'un bruit additif v(k). Le schéma du montage est celui de la Figure 3.3. Le générateur de nombres aléatoires (1) produit le signal test {a(k)} de moyenne nulle et de distribution normale. Cette séquence {a(k)} est le signal utilisé pour tester le canal. C'est une suite de valeurs +1 ou −1 générée de façon aléatoire. Le canal de réponse impulsionnelle **H**(k)

définie ci-dessous, produit un signal déformé à la sortie que l'on corrompt par une autre séquence aléatoire $\{v(k)\}$. Le générateur (2) est donc la source de bruit $\{v(k)\}$ qui perturbe la sortie du canal. $\{v(k)\}$ est un bruit blanc de variance σ_v^2 connue. Les signaux $\{a(k)\}$ et $\{v(k)\}$ sont considérés indépendants au sens des équations (3.4) à (3.6). Le but de l'égaliseur est de corriger les distorsions produites par le canal en présence du bruit additif $\{v(n)\}$.

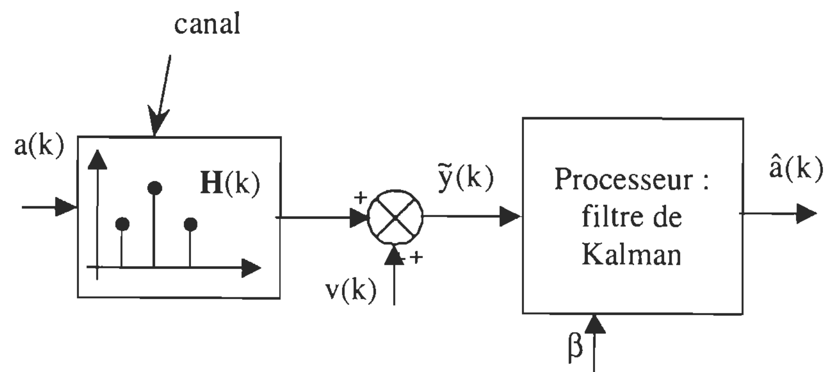


Figure 3.3 : Diagramme bloc de l'égalisateur adaptatif des canaux par filtre de Kalman.

Le signal $\{a(k)\}$ est illustré par la Figure 3.4a. La sortie de l'égalisateur devrait donc se rapprocher le plus possible de cette forme. Le signal à filtrer a la forme d'une somme de convolution corrompue par v(k) :

$$\tilde{y}(k) = \sum_{i=1}^{M} \mathbf{H}_i(k)\mathbf{a}_i + v(k) \quad \text{pour } k = 1,2,3,... \tag{3.64}$$

Il est schématisé à la Figure 3.4b pour la réponse impulsionnelle donnée par l'équation (3.66).

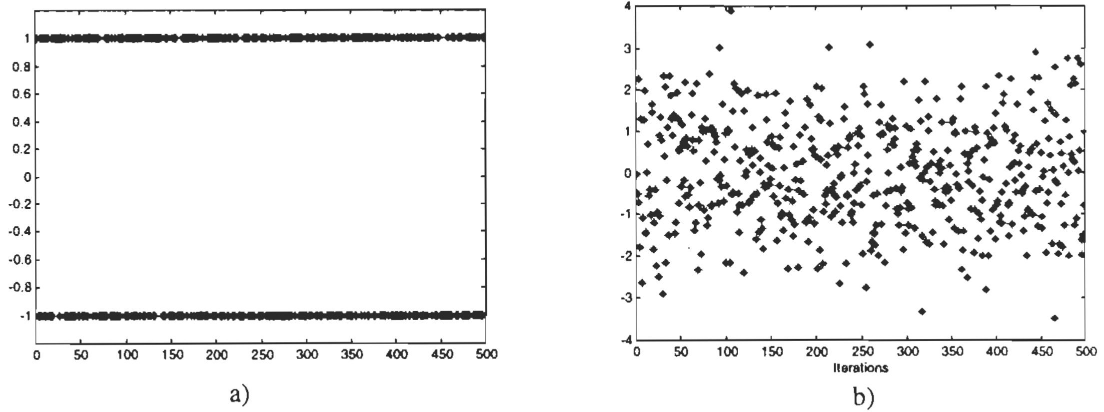


Figure 3.4 : Exemple type de signaux générés synthétiquement : a) signal de test du canal a(k) et b) signal de sortie corrompu $\tilde{y}(k)$ avec $\sigma_v^2 = 0.6$ (SNR = 5dB)

La réponse impulsionnelle invariante du canal est la suivante :

$$h_n = \begin{cases} \frac{1}{2}\left[1+\cos\left(\frac{2\pi}{W}(n-2)\right)\right] & n = 1,2,3 \\ 0 & \text{ailleurs} \end{cases} \tag{3.65}$$

où W contrôle la distorsion produite par le canal. Dans notre étude nous utiliserons la valeur de W qui produit les meilleurs résultats dans l'expérience réalisée dans [HAY96], à savoir W = 2.9. Cette réponse impulsionnelle est schématisée à la Figure 3.5a.

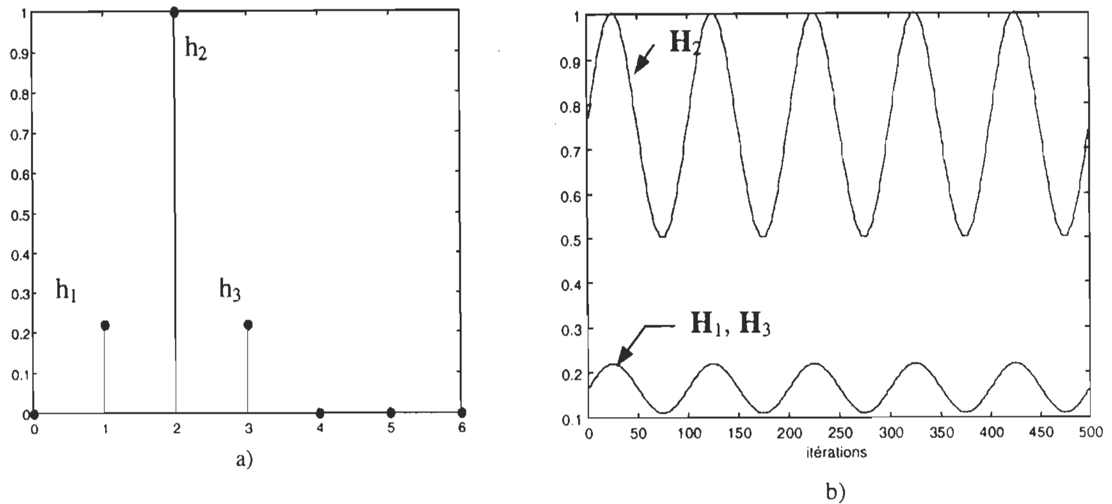


Figure 3.5 : a) Réponse impulsionnelle invariante du canal et b) variations de l'amplitude des composantes de **H** dans le cas d'une réponse impulsionnele variante avec N = 500 points et P = 5 périodes

Dans le cas d'une réponse impulsionnelle variante, nous choisissons les paramètres de variation qui permettent au canal de faire P périodes de variation sur N échantillons considérés ;

$$\mathbf{H}_n(k) = \mathbf{h}_n\left[0.75 + 0.25\sin(2\pi kP/N)\right] \quad n = 1,2,3 \quad \text{et} \quad k = 1,2,3,\ldots,N \tag{3.66}$$

La réponse impulsionnelle variante est schématisée à la Figure 3.5b.

En fait, dans la réalité il existe toujours des facteurs qui influencent les paramètres du canal de communication et par conséquent le vecteur **h** devient variant. Les gains du filtre de Kalman **K**(k) ne sont donc plus stationnaires. Il faudrait les calculer à chaque échantillon k pour adapter le filtre au canal. Le filtre de Kalman est en effet considéré comme l'un des meilleurs algorithmes de filtrage adaptatif qui existent dans la littérature.

Dans le cadre de ce mémoire, nous avons étudié la version filtre de covariance standard et la version racine carrée de covariance pour des raisons d'intégration en VLSI qui seront spécifiées au chapitre 4. Ensuite nous ferons une étude comparative des performances.

Les points les plus importants que nous allons tirer de [MAS95] sont les formes des matrices d'état et la méthode d'estimation mais nous allons considérer un système non invariant contrairement à ce qui a été fait par les auteurs.

Les valeurs des paramètres de l'équation d'état définie aux équations (3.1) et (3.2) sont définies comme suit [MAS95] :

$$\Phi = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 & 0 \\ . & . & \cdots & . & . & . \\ 0 & 0 & \cdots & 1 & 0 & 0 \\ 0 & 0 & \cdots & 0 & 1 & 0 \end{bmatrix} \quad \dim(\Phi) = M \times M \tag{3.67}$$

$$\mathbf{b} = [1 \quad 0 \quad 0 \quad \cdots \quad 0 \quad 0]^T \quad \dim(\mathbf{b}) = M \tag{3.68}$$

$$\mathbf{h} = [h_1 \quad h_2 \quad \cdots \quad h_{M-1} \quad h_M]^T \quad \dim(\mathbf{h}) = M \tag{3.69}$$

En fait, les échantillons sont de passage dans le vecteur et sont filtrés et décalés à chaque instant. Par conséquent plus on tarde pour lire la valeur d'un échantillon dans ce vecteur, plus on profite du lissage inhérent à la méthode [MAS95]. Le résultat de l'estimation $\hat{a}(k)$ est donné par :

$$\hat{a}(k) = \text{signe}(\hat{x}_M(k)) \quad (3.70)$$

3.2.1 Égalisation des canaux par filtre de Kalman standard

Les équations (3.12) à (3.17) peuvent être écrites simplement pour une mesure monodimensionnelle et en normalisant les matrices de covariance comme suit :

$$\hat{\mathbf{x}}(k+1/k) = \mathbf{\Phi}\hat{\mathbf{x}}(k/k) \quad (3.71)$$

$$\mathbf{Q}(k+1/k) = \mathbf{\Phi}\mathbf{Q}(k/k)\mathbf{\Phi}^T + \mathbf{b}\beta\mathbf{b}^T, \quad \mathbf{Q}(0/0) = \mathbf{I} \quad (3.72)$$

$$\mathbf{V}(k+1) = \mathbf{H}(k)\mathbf{Q}(k+1/k)\mathbf{H}^T(k) + 1 \quad (3.73)$$

$$\mathbf{K}(k+1) = \mathbf{Q}(k+1/k)\mathbf{H}^T(k+1)\mathbf{V}^{-1}(k+1) \quad (3.74)$$

$$\hat{\mathbf{x}}(k+1/k+1) = \hat{\mathbf{x}}(k+1/k) + \mathrm{K}(k+1)[\tilde{\mathbf{y}}(k+1) - \mathbf{H}(k+1)\hat{\mathbf{x}}(k+1/k)] \quad (3.75)$$

$$\mathbf{Q}(k+1/k+1) = \mathbf{Q}(k+1/k) - \mathbf{K}(k+1)\mathbf{H}(k+1)\mathbf{Q}(k+1/k) \quad (3.76)$$

avec

$$\beta = \frac{\sigma_w^2}{\sigma_v^2}, \ \mathbf{Q} = \mathbf{P}/\sigma_v^2 \ \text{et} \ \mathbf{V} = \mathbf{V}_e/\sigma_v^2 \quad (3.77)$$

3.2.2 Égalisation des canaux par filtre de covariance racine carrée

La deuxième version du filtre de Kalman utilisée ici est le filtre de covariance racine carrée. Cette version présente des caractéristiques qu'il est important d'étudier et ensuite

comparer à la version précédente. En effet, cette version présente une simplicité et une régularité qui peuvent être un avantage certain lors de la dérivation des architectures systoliques. En effet, il est affirmé par plusieurs auteurs que cette version présente une plus grande robustesse aux effets de quantifications que la version standard.

En appliquant également une normalisation des racines carrée des matrices de covariance, les équations (3.41), (3.42), (3.49) et (3.50) peuvent être réécrites comme suit :

$$\hat{\mathbf{x}}(k+1/k) = \mathbf{\Phi}(k)\hat{\mathbf{x}}(k/k) \tag{3.78}$$

$$\left[\frac{\mathbf{L}^{\mathbf{T}}(k+1/k)}{\mathbf{0}}\right] = \mathbf{T}\left[\frac{\mathbf{L}^{\mathbf{T}}(k/k)\mathbf{\Phi}^{\mathrm{T}}(k)}{\beta\mathbf{B}^{\mathbf{T}}(k)}\right] \tag{3.79}$$

$$\hat{\mathbf{x}}(k+1/k+1) = \hat{\mathbf{x}}(k+1/k) + \left(\mathbf{g}^{\mathrm{T}}(k+1)/F_1(k+1)\right)\left[\tilde{\mathbf{y}}(k+1) - \mathbf{H}(k+1)\hat{\mathbf{x}}(k+1/k)\right] \tag{3.80}$$

$$\begin{bmatrix} F_1(k+1) & \mathbf{g}(k+1) \\ \mathbf{0} & \mathbf{L}^{\mathrm{T}}(k+1/k+1) \end{bmatrix} = \mathbf{T}\begin{bmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{L}^{\mathrm{T}}(k+1/k)\mathbf{H}^{\mathrm{T}}(k+1) & \mathbf{L}^{\mathrm{T}}(k+1/k) \end{bmatrix} \tag{3.81}$$

avec

$$F_1 = \frac{F}{\sigma_v}, \quad \mathbf{L} = \mathbf{S}/\sigma_v, \quad \mathbf{g} = \mathbf{G}/\sigma_v \quad \text{et} \quad \beta = \sigma_w/\sigma_v \tag{3.82}$$

3.3 Résultats de simulation de l'égalisation des canaux par filtre de Kalman

Considérons le système dynamique variant modélisant le canal présenté au paragraphe 3.2, et rappelons les équations de la réponse impulsionnelle variante :

$$H_n(k) = h_n[0.75 + 0.25\sin(2\pi kP/N)] \quad n = 1,2,3 \quad \text{et} \quad k = 1,2,3,\ldots,N \tag{3.85}$$

$$h_n = \begin{cases} \frac{1}{2}\left[1+\cos\left(\frac{2\pi}{W}(n-2)\right)\right], & n = 1,2,3 \\ 0, & \text{ailleurs} \end{cases} \tag{3.86}$$

Le signal bruité présenté à l'entrée de l'égalisateur a la forme du signal de la Figure 3.6. Ce signal est en fait la convolution de la réponse impulsionnelle variante et de l'entrée a_n prenant les valeurs {-1,+1}, auquel on a ajouté un bruit blanc de variance σ_v^2 connue. Le rapport signal sur bruit SNR est donc donné par :

$$SNR = 10\log\frac{\|a(n)\|^2}{\|v(n)\|^2} \tag{3.87}$$

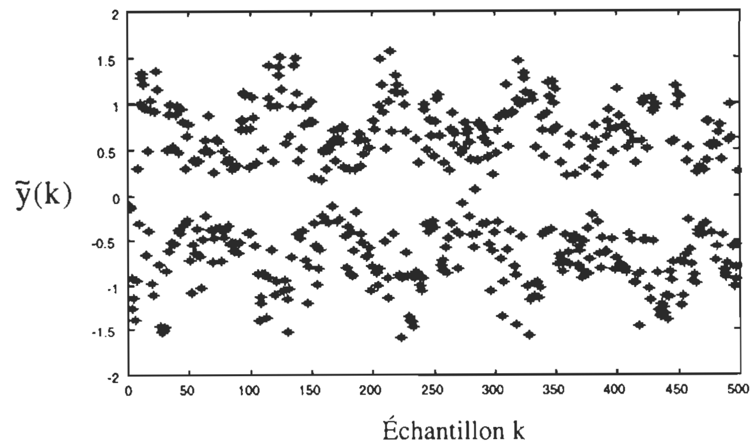


Figure 3.6 : Signal corrompu et entaché de bruit : SNR = 20dB et BER = 50%

Nous avons réalisé l'expérience pour plusieurs niveaux de bruit sur le signal et le filtrage à été fait par les deux algorithmes du filtre de Kalman.

La qualité de filtrage est évaluée par deux éléments : l'erreur quadratique moyenne de filtrage ε qui est définie par l'équation (3.88) et le BER (Bit Error Rate) qui est le rapport du nombre de bits erronés sur le nombre de bits transmis, équation (3.89).

$$\varepsilon = \frac{\|a(n) - \hat{x}(n)\|}{\|a(n)\|} \tag{3.88}$$

$$\text{BER} = \frac{\text{nbre de bits erronés}}{\text{nbre de bits transmis}} \tag{3.89}$$

En un mot, le BER représente le nombre de bits qui auront changé de polarité lorsqu'on aura fait un seuillage sur le résultat de filtrage obtenu. Un BER de 40% sur le signal bruité signifie que 40% de bits ont changé de polarité. Si on applique tout simplement un seuillage sur le signal bruité reçu sans le reconstituer, on aura 40% de bits

incorrects dans le signal. Un BER de 0 sur le signal reconstitué signifie que exactement tous les bits reçus corrompus on été reconstitués avec succès par l'algorithme du filtre de Kalman.

L'expérience a été réalisée sur N = 500 points mais le BER a été calculé sur une étendue de N = 10 000 points avec :

W = 2.9 : paramètre qui contrôle la distorsion du canal,

P = 5 : nombre de périodes de variation de la réponse impulsionnelle pendant la durée de l'expérience.

Le paramètre W a été défini dans [HAY96] qui a fait une étude pour trouver la valeur optimale dans le cas d'un filtrage adaptatif. Dans cette ouvrage, une expérience sur l'égalisation des canaux par LMS et RLS a été faite et les meilleurs résultats ont été obtenus pour la valeur W=2.9. Nous avons donc utilisé cette valeur pour pouvoir comparer la qualité de filtrage par Kalman avec les résultats donnés dans [HAY96]. La Figure 3.6 présente le signal bruité à filtrer. En la comparant à la Figure 3.4 et la Figure 3.5, on remarque que le signal à la sortie du canal suit légèrement les variations de sa réponse impulsionnelle. Mais, contrairement à ce qui est apparent sur cette figure, il n'y a pas de séparation entre les signaux de polarité originale positive et ceux de polarité originale négative. Il y a environ 50% d'échantillons qui ont changé de polarités après être passés dans le canal.

3.3.1 Égalisation par Kalman Standard.

La courbe de la Figure 3.7 montre les résultats de filtrage par Kalman standard. Si l'erreur quadratique est de 18.7%, le BER est identiquement nul pour ce niveau de bruit, qui peut être classé comme un bruit fort. Ceci veut dire Kalman stationnaire produirait une erreur de filtrage de 18.7%, mais après seuillage, on aurait reconstitué exactement le signal original. Mais, ceci sera fait à 3 échantillons près. En effet, le retard sur la sortie est égale à M, où M est le nombre de points de la réponse impulsionnelle. Donc les 3 premiers échantillons ne seront pas reconstitués. Néanmoins ce résultat est toujours meilleur que celui produit par les algorithmes basés sur LMS, RLS ou les réseaux de neurones. Ces algorithmes présentent une plus longue période d'adaptation ou d'apprentissage.

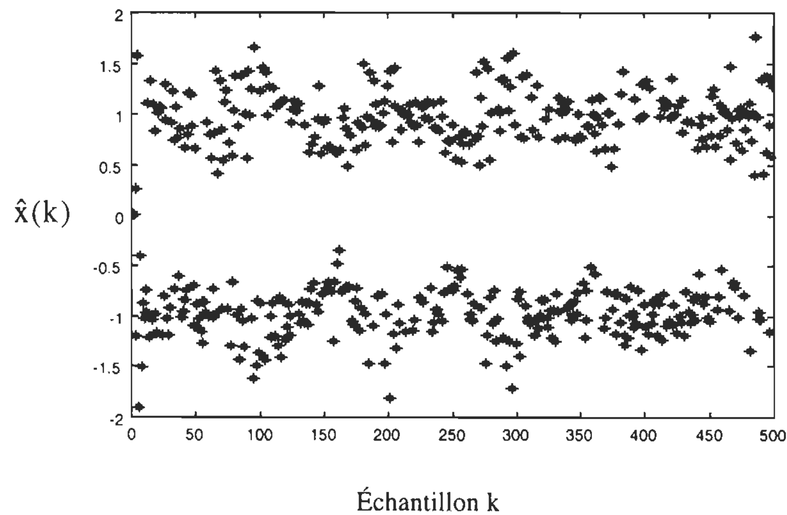


Figure 3.7 : Résultats de simulation du filtre de Kalman standard pour SNR = 20dB, Résultats : $\varepsilon = 18.6\%$ et BER = 0

3.3.2 Égalisation par Kalman covariance racine carrée.

Le filtrage par Kalman racine carrée de covariance, présente à peu près les mêmes résultats que le filtrage par Kalman standard.

À la Figure 3.8, on voit que le filtre de a bien filtrer le signal bruité et il se rapproche du signal original de la Figure 3.4. Ici également, le BER du signal obtenu est identiquement nul. Par conséquent, on peut dire que le filtre de Kalman réussi à reconstituer exactement le signal original pour un bruit de 20dB.

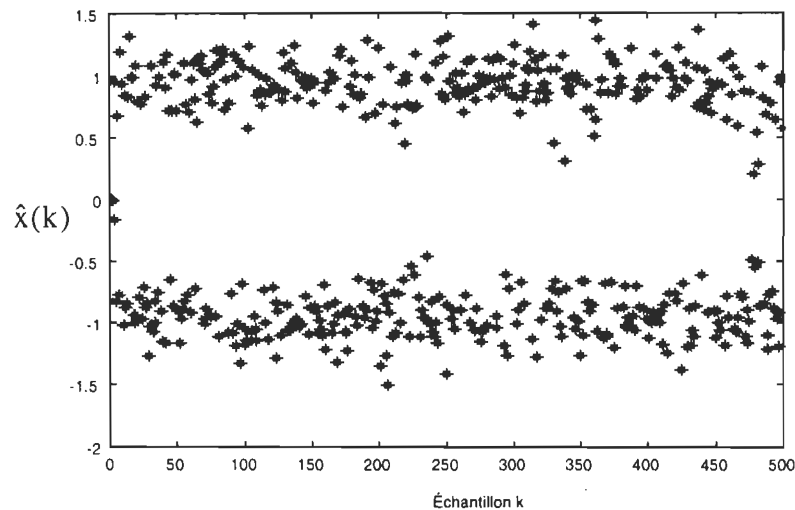


Figure 3.8 : Résultats de simulation du filtre de Kalman racine carrée de covariance pour SNR = 20dB, Résultats : $\varepsilon = 18.7\%$ et BER = 0

3.3.3 Évaluation des performances

De façon assez globale, nous avons effectué plusieurs expériences d'égalisation de canaux pour des niveaux de bruit différents. Le Tableau 4.1 présente les niveaux de bruit et les valeurs de BER tant sur le signal original que sur le signal filtré.

La réponse impulsionnelle choisie détruit la polarité d'environ 50% des points du signal et le bruit additionnel disperse davantage le signal sur le spectre. Le Tableau 4.1 montre par exemple que nous avons reconstitué un signal de SNR = 8dB avec 10% d'erreur. En plus, Kalman peut reconstituer un signal de SNR=15dB avec seulement 0.8% d'erreur.

Tableau 4.1 : Résultat d'égalisation pour différents niveaux de bruit.

Signal bruité			**Signal Filtré**			
SNR (dB)	Variance du Bruit	BER (%)	Kalman Standard		Kalman Racine Carrée Covariance	
			ε (%)	BER (%)	ε (%)	BER (%)
60	0.001	50.7	6.6	0	7.0	0
40	0.01	50.7	7.0	0	7.3	0
20	0.1	50	18.6	0	18.7	0
15	0.2	49.1	35.2	0.4	35.2	0.4
10	0.3	46.9	51.3	4.0	51.3	4.0
8	0.4	43.3	67.2	9.3	67.2	9.3
6	0.5	43	82.7	16.1	82.7	16.1
0.3	1	45	157.6	26.2	157.0	26.0

Les valeurs données dans ce tableau sont obtenues pour les valeurs de β presque optimales, c'est-à-dire celles qui nous donnent le plus de satisfaction. Mais, une amélioration pourrait être obtenue pour chaque valeur du bruit. En fait, il faudrait rechercher la valeur de β qui produirait l'erreur de filtrage minimale. Dans cet algorithme, β est donc utilisé comme paramètre de syntonisation, pour améliorer la qualité du filtrage.

Dans l'expérience qui suit, nous avons comparé nos résultats avec les autres méthodes d'égalisation de canaux proposées dans [HAY96], à savoir le filtrage par LMS et RLS.

Pour une intensité de bruit donnée, SNR = 15dB, on calcule le BER pour un signal de 10000 points. On choisit les paramètres des filtres RLS et LMS qui donnent les meilleurs résultats possibles à savoir :

Paramètres RLS : Nombre de poids M_{RLS} = 11, facteur d'oubli $\lambda = 1$, facteur d'initialisation de P(0) $\beta_{RLS} = 1$.

Paramètres LMS : Nombre de poids M_{LMS} = 11, taux de convergence μ=0.1.

La Figure 3.9 représente le BER pour les signaux LMS, RLS et Kalman de covariance racine carrée. Les résultats dans le cas de Kalman standard sont presque identiques à ceux de Kalman covariance racine carrée, et ne figurent donc pas ici.

Si les algorithmes RLS et Kalman présentent un net avantage sur LMS, le filtrage par Kalman est encore plus efficace car il ne nécessite aucune période d'apprentissage. On remarquera que dans ce cas, le BER est identiquement nul pour les 1500 premiers points de l'expérience.

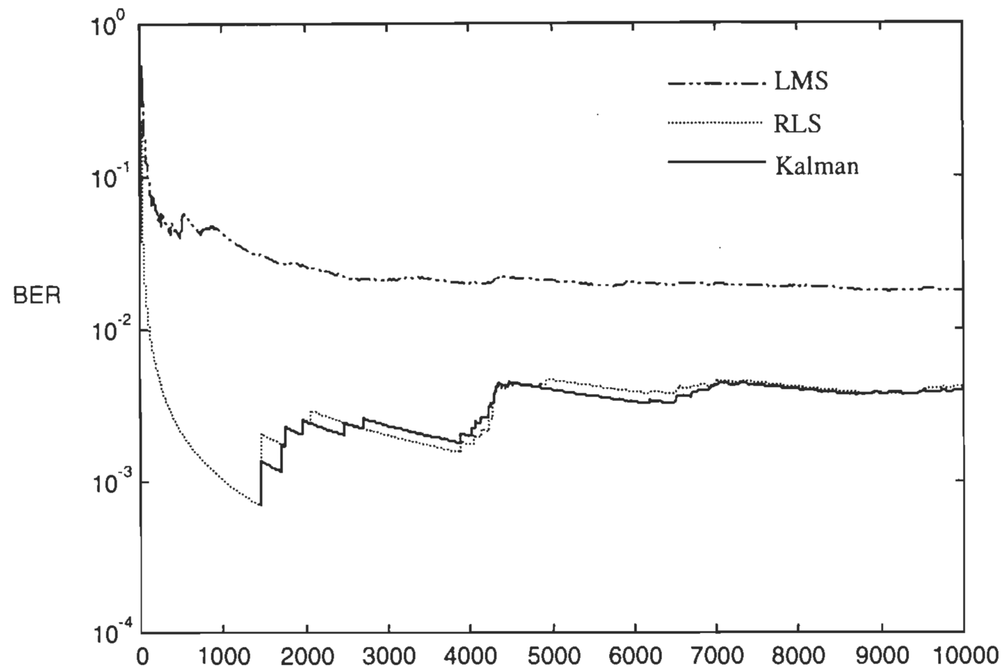


Figure 3.9 : BER pour égalisation adaptative des canaux par LMS, RLS et Kalman. SNR = 15dB

3.4 Application du filtre de Kalman à la commande

Le filtre de Kalman est un estimateur linéaire optimal et par conséquent trouve son application dans tout domaine où il est nécessaire de faire la prédiction et/ou du filtrage. Le filtre de Kalman est important pour l'estimation des paramètres dans une usine, pour les problèmes en commande adaptative à cause de la relation mathématique qu'il possède avec les moindres carrées. Il est aussi important pour la commande quadratique linéaire optimale (*LQ optimal control*). La demande sans cesse croissante des contrôleurs en temps réel a amené les ingénieurs à introduire le calcul parallèle dans les algorithmes de contrôle. Les calculs parallèles offrent la possibilité de concevoir des circuits qui, une fois implantés physiquement, donnent des performances très proches des performances obtenues lors des

simulations. Ils procurent également au filtre de Kalman une efficacité qui le rend toujours utile et performant malgré l'émergence des nouvelles techniques d'estimation de type non linéaire telles que les réseaux de neurones et les algorithmes génétiques.

La formulation de l'équation d'état d'un système en commande est en générale différente de celle donnée aux équations (3.1) et (3.2). En effet, les systèmes étudiés comprennent une entrée de commande déterministe qui permet de positionner le système en question dans un état désiré. Cette entrée notée u(k), est appelée la loi de commande du système. Ainsi, l'équation d'état du système devient :

$$\mathbf{x}(k+1) = \mathbf{A}(k)\mathbf{x}(k) + \mathbf{B}(k)\mathbf{u}(k) + \mathbf{B}_1(k)\mathbf{w}(k) \tag{3.90}$$

$$\mathbf{y}(k) = \mathbf{C}(k)x(k) + \mathbf{v}(k) \tag{3.91}$$

L'application du filtre de Kalman conventionnel à l'équation d'état du système s'écrit :

$$\hat{\mathbf{x}}(k+1/k) = \mathbf{A}(k)\hat{\mathbf{x}}(k/k) + \mathbf{B}(k)\mathbf{u}(k) \tag{3.92}$$

$$\hat{\mathbf{x}}(k+1/k+1) = \hat{\mathbf{x}}(k+1/k) + \mathbf{K}(k+1)\left[\tilde{\mathbf{y}}(k+1) - \mathbf{C}(k+1)\hat{\mathbf{x}}(k+1/k)\right] \tag{3.93}$$

$$\mathbf{K}(k+1) = \mathbf{P}(k+1/k)\mathbf{C}^T(k+1)\mathbf{V}_e^{-1}(k+1) \qquad (3.93)$$

$$\mathbf{P}(k+1/k) = \mathbf{A}(k)\mathbf{P}(k/k)\mathbf{A}^T(k) + \mathbf{B}_1(k)\mathbf{R}_w(k)\mathbf{B}_1^T(k) \tag{3.94}$$

$$\mathbf{V}_e(k) = \mathbf{C}(k)\mathbf{P}(k+1/k)\mathbf{C}^T(k) + \mathbf{R}_v(k) \tag{3.95}$$

$$\mathbf{P}(k+1/k+1) = \mathbf{P}(k+1/k) - \mathbf{K}(k+1)\mathbf{C}(k+1)\mathbf{P}(k+1/k) \qquad (3.96)$$

La structure du filtre de Kalman est très générale et permet grâce à des simplifications, d'obtenir d'autres algorithmes classiques d'observation comme l'observateur de Luenberger [SIC97].

Application

Nous présentons ici un exemple pratique où le filtre de Kalman à été intégré dans la conception d'un contrôleur pour un système dynamique invariant. Les matrices d'état ne dépendent donc pas du temps k. Le principe du contrôleur est une commande par retour d'état observé, l'observateur étant le filtre de Kalman. Il sera donc utilisé ici comme reconstructeur d'état dans un environnement stochastique. Le système utilisé est une vanne mécanique commandée par un convertisseur électropneumatique.

Le filtre de Kalman estime l'état du système et introduit les valeurs obtenues dans le contrôleur qui définit la loi de commande pour le système dynamique. Le système étudié est montré à la Figure 3.10. On désire positionner la vanne du système à une consigne donnée à l'aide d'une commande par retour d'état observé permettant de maintenir la vanne à une position constante d_v^*. Un capteur de position retourne la position de la vanne $\tilde{d}_v$. On est donc supposé connaître la position exacte de la vanne $\mathring{d}_v$. Mais cette valeur mesurée $\tilde{d}_v$ est entachée d'un bruit aléatoire qu'on considérera gaussien, de moyenne nulle et de variance connue σ_v^2.

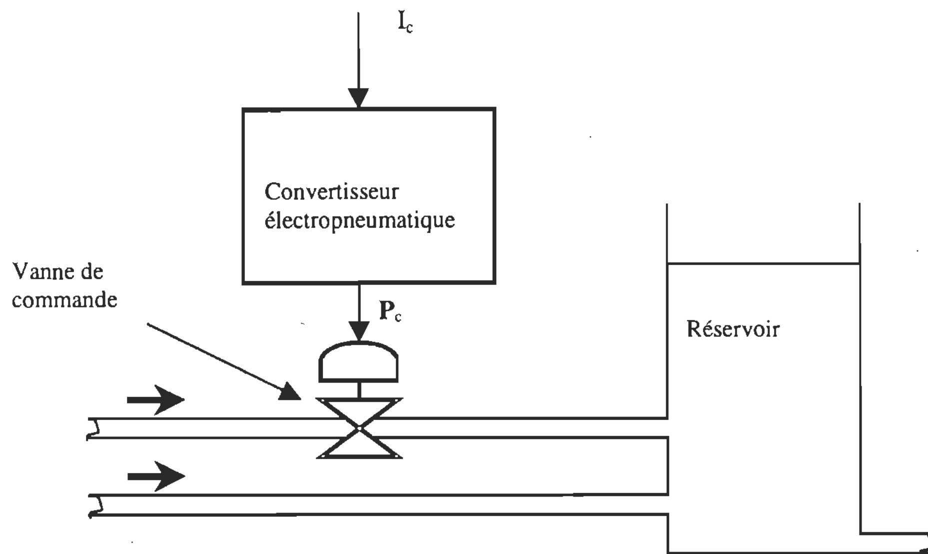


Figure 3.10 : Schéma général du système de commande

À cause de facteurs difficilement modélisables telles que les vibrations de la vanne pendant le mouvement ou l'échauffement de la tige, la position sera incorrectement estimée. Pour modéliser cette erreur, nous considérons un bruit global agissant sur l'état du système, de distribution gaussienne, de moyenne nulle et de variance σ_w^2 connue. Le Filtre de Kalman sera utilisé pour reconstruire l'état du système, c'est-à-dire retrouver une estimation possible de la valeur réelle de $\overset{\circ}{d}_v$ notée $\hat{d}_v$ dans cet environnement stochastique.

L'entrée de commande du système est un courant I_c qui en agissant sur le convertisseur électropneumatique produit une pression P_c sur la vanne.

L'équation de la position relative de la vanne d_v, linéarisée autour de son point d'équilibre est donnée par :

$$\ddot{d}_v = -\frac{k_v}{m_v} d_v - \frac{f_v}{m_v} \dot{d}_v + \frac{k_b \pi R_d^2}{m_v} I_c \tag{3.91}$$

où : m_v, k_v, f_v : Caractéristiques de la vanne,

k_b, R_d : Caractéristiques du convertisseur électropneumatique

En posant $x_1 = d_v$ et $x_2 = \dot{d}_v$, le vecteur d'état est donc $\mathbf{x}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ et $u = I_c$, les matrices d'état du modèle linéarisé sont donc les suivantes :

$$\mathbf{A}_l = \begin{bmatrix} 0 & 1 \\ -k_v / m_v & -f_v / m_v \end{bmatrix} \qquad \mathbf{B}_l = \begin{bmatrix} 0 \\ \dfrac{k_b \pi R_d^2}{m_v} \end{bmatrix} \tag{3.92}$$

$$\mathbf{C}_l = \begin{bmatrix} 1 & 0 \end{bmatrix} \qquad \mathbf{D}_l = 0 \tag{3.93}$$

La fonction *ctrb* de Matlab® nous a permis de vérifier la contrôlabilité du système. Le modèle discret est dérivé à l'aide de la fonction *c2d* également disponible sur Matlab®. On obtient donc les matrices d'état discrètes **A, B, C** et **D**. Rappelons que les matrices sont invariantes et que la mesure $\tilde{y}$ est un scalaire.

La Figure 3.11 présente le diagramme bloc du système global observé et commandé.

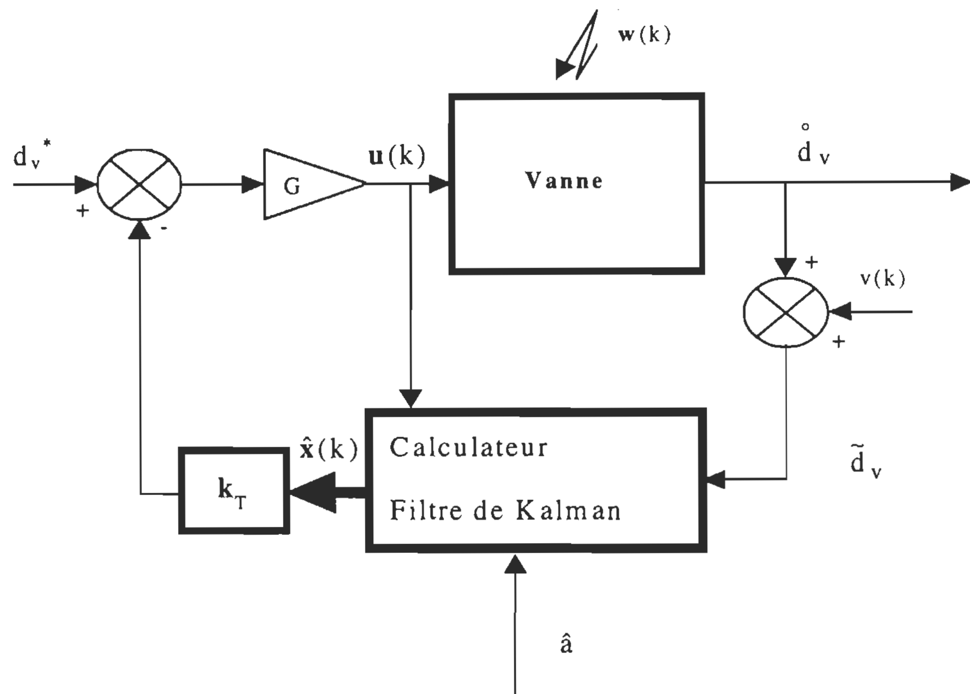


Figure 3.11 : Diagramme bloc du système

On désire obtenir les résultats suivants pour la vanne :

- une erreur stationnaire nulle,

- un dépassement nul

- une stabilisation à 2% de la consigne en 1s.

Pour cela, trouvons la valeur mathématique du gain G pour avoir une erreur nulle en régime permanent :

$$u(k) = G\left(-\mathbf{k}^T(k)\hat{\mathbf{x}}(k/k) + d_v^*\right) \tag{3.94}$$

$$\begin{aligned}\hat{\mathbf{x}}(k+1/k) &= \mathbf{Ax}(k/k) + \mathbf{B}u(k) \\ &= \left(\mathbf{A} - \mathbf{B}G\mathbf{k}^T(k)\right)\hat{\mathbf{x}}(k/k) + \mathbf{B}Gd_v^*\end{aligned} \tag{3.95}$$

$$y(k+1) = \mathbf{Cx}(k+1/k) \tag{3.96}$$

En prenant la transformée en z puis la transformée en z inverse, on obtient la réponse temporelle :

$$y(t) = y_\infty(1 - e^{-wt}) \tag{3.97}$$

En posant $y_\infty = d_v^*$ et $\mathbf{p} = G\mathbf{k}^T$, on obtient :

$$G = \frac{1}{\mathbf{C}[\mathbf{I} - (\mathbf{A} - \mathbf{Bp})]^{-1}\mathbf{B}} \tag{3.98}$$

L'exigence sur le temps de stabilisation nous donne la position des deux pôles du système. En utilisant la fonction *place* de Matlab®, on obtient la valeur du produit $\mathbf{p} = G\mathbf{k}^T$. On en déduit la valeur du vecteur gain $\mathbf{k}$ du contrôleur. Les valeurs numériques des constantes sont :

m_v=20Kg; k_v=2000N/m; f_v=40N.s/m; k_b=3*6895/4 USI; R_d = 0.036m, période d'échantillonnage : $T_s = 0.1s$, $ó_v^2 = 0.1, ó_w^2 = 0.008$ et $\mathbf{B}_1 = \begin{bmatrix}1\\1\end{bmatrix}$

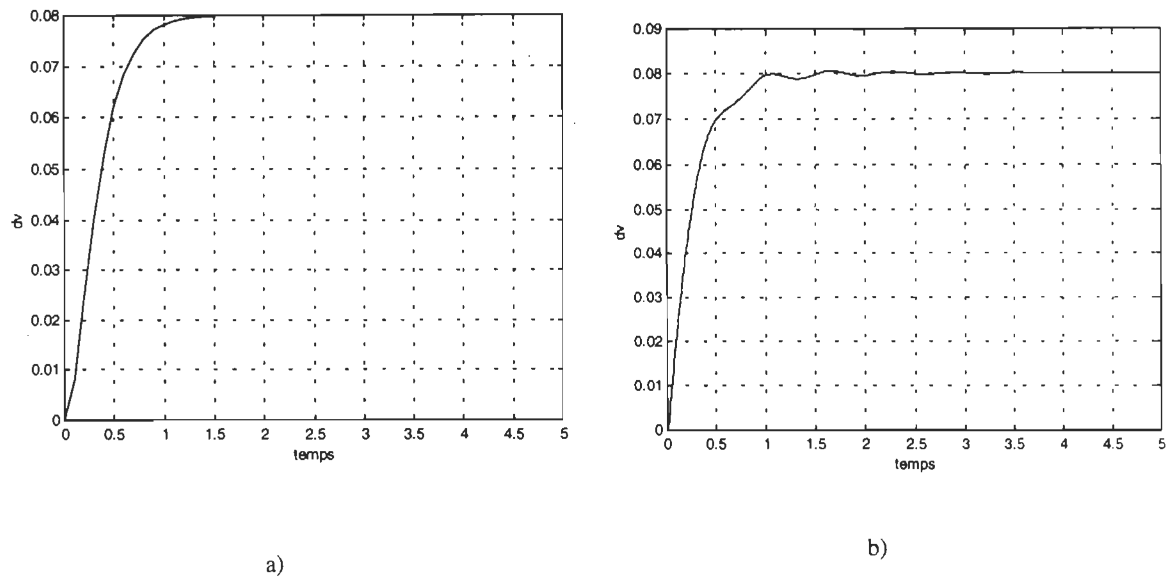


Figure 3.12 : Positionnement de la tige a) avec loi de commande sur retour d'état observé par le filtre de Kalman et b) un contrôleur PID

La Figure 3.12 montre le résultat obtenu pour une consigne $d_v^* = 0.08\text{m}$. Ce résultat est nettement meilleur que celui obtenu avec un contrôleur PID dans un environnement non stochastique [MOZ97].

3.5 Justification de l'implantation du filtre de Kalman en technologie VLSI

Les deux applications montrées à la section 3.2 montrent que le filtre de Kalman peut avoir de nombreux domaines d'utilisation possibles. En plus de cette versatilité, plusieurs autres motifs peuvent être énumérés comme raisons favorables à une implantation en technologie VLSI d'une architecture de reconstitution ou de filtrage basée sur le filtre de Kalman. Le plus important est celui qui a fait la popularité du filtre de Kalman, c'est l'efficacité qu'il procure dans les domaines où il est utilisé. En effet, il est l'un des

algorithmes de filtrage et de commande adaptative les plus efficaces qui existent dans la littérature. Le deuxième argument est que le filtre de Kalman est hautement régulier, donc rend possible une réalisation sur architecture hautement parallèle (systolique). En plus, l'émergence des outils de synthèse automatique favorise également cette implantation car on peut à l'aide de ces outils informatiques dériver de façon presque automatique et en un temps assez court, des architectures parallèles du filtre de Kalman.

Mais l'application de ce filtre dans les applications en temps est limitée par la complexité des calculs dans l'algorithme. En général, le filtrage en temps réel ne peut être effectué sur les problèmes de dimension très grande en utilisant les architectures monoprocesseurs. Ce phénomène a donc retardé la réalisation en VLSI d'architectures basées sur le filtre de Kalman. En effet, jusqu'alors, le filtre de Kalman était plus couramment utilisé dans les applications où le temps de réponse était élevé ou en simulation. Par conséquent il était implanté dans des ordinateurs multiprocesseurs [OHA88]. Le concept des architectures parallèles et plus particulièrement les architectures systoliques [KUN82] nous donne les moyens de réduire considérablement le temps de calcul dans le filtre de Kalman et par conséquent augmenté le débit de sortie [MOZ98], [MOZ99], [KUN91], [GAS88], [YEH88], [RAO91], [IRW91] et [MAS98].

Les architectures systoliques sont un parallélisme à grain fin où les communications entre les processeurs sont locales (chaque processeur ne communique qu'avec ses voisins immédiats) et où les communications avec le milieu extérieur se fait par les processeurs périphériques. Les données se propagent de proche en proche d'un processeur à ses voisins

et le rythme de propagation est cadencé par l'horloge unique du système global. Avec ces méthodes, les données sont divisées et partagées aux différents processeurs élémentaires du réseau.

De plus, l'émergence des outils de parallélisation automatique (**MMAlpha** [QUI89b], [VER91] ; **HI-PASS DSP** [DUN92] ; **Approval** [RAM95] ; **OPERA** [LOE94]) qui est venue favoriser l'avancée des recherches sur l'implantation du filtre de Kalman dans une puce de silicium.

En effet, ces outils de parallélisation permettent de partir de l'équation de récurrence mathématique du filtre de Kalman (ou tout autre algorithme), de le traduire dans le langage utilisé par l'environnement et par transformations successives, de générer un ordre d'exécution des variables qui minimise le temps de calcul, de pipeliner certaines opérations lorsque nécessaire, de maximiser le parallélisme, de produire des descriptions niveaux architecturales et enfin de générer toujours de façon automatique le code VHDL pour l'architecture finale. Le parallélisme dans cette architecture finale est maximal et par conséquent le débit l'est aussi.

En particulier, nous avons choisi d'explorer l'implantation en VLSI d'une architecture systolique du filtre de Kalman parce que l'environnement MMAlpha nous permet de faire toutes ces étapes de parallélisation et d'aboutir à une architecture systolique.

En conclusion, les principales raisons qui militent en faveur de l'implantation en technologie VLSI ou FPGA des architectures basées sur le filtre de Kalman sont :

- Le filtre de Kalman est un outil très puissant pour le filtrage, la prédiction et le lissage en traitement du signal et la reconstruction d'état, la prédiction de trajectoires en commande. En bref, malgré la tendance moderne actuelle il demeure efficace en filtrage adaptatif et en commande adaptative.

- Les architectures parallèles permettent aujourd'hui de l'implanter pour les applications en temps réel. Mieux encore, les outils de parallélisation automatique tel que MMAlpha utilisé dans ce projet, permettent aujourd'hui de dériver quasi automatiquement des architectures parallèles pour ce filtre, ce qui nous donne un gain de temps substantiel dans la conception et un gain de vitesse énorme dans l'architecture finale obtenue.

Chapitre 4

Synthèses d'architectures parallèles avec MMAlpha

Le filtre de Kalman est un estimateur linéaire optimal qui peut reconstruire l'état d'un système à partir des données mesurées et dans un environnement stochastique. Le désavantage que présente ce filtre est la densité de calcul au sein de son algorithme. En effet, le nombre d'opérations arithmétiques qu'il faut effectuer dans le filtre de Kalman est $O(n^3)$ où n représente la dimension du système. Une exécution séquentielle de ces opérations serait non réaliste et non implantable dans un circuit intégré : il est donc nécessaire de trouver un moyen de réduire les temps de calcul en vue de son implantation en VLSI et pour satisfaire des applications en commande, en communication et en traitement numérique du signal. Plusieurs chercheurs ont proposé des architectures parallèles pour le filtre de Kalman dans le but de maximiser le nombres d'opérations réalisées dans un temps de cycle. Plusieurs architectures ont été développées par des méthodes d'algèbres linéaires classiques, exploitant la régularité inhérente du filtre de

Kalman [IRW91], [KUN91], [YEH88], [MAS95] et [FAY95]. Ces méthodes utilisent entre autres les rotations de Givens [QUI89a] et les algorithmes de Fadeev [YEH88].

Dans ce chapitre, nous allons utiliser MMAlpha comme outil pour dériver automatiquement deux architectures systoliques du filtre de Kalman. Une étude comparative des architectures obtenues avec celles proposées dans la litérature sera réalisée.

4.1 Filtre de covariance

Nous présentons ici l'architecture que nous avons obtenue après avoir appliqué les transformations de MMAlpha décrites au chapitre 2.

4.1.1 Programmation en Alpha

Dans une premier temps, il faut programmer les équations du filtre de covariance, (3.71) à (3.76), en Alpha. Pour raison de conformité, nous effectuons un changement de variable pour la matrice de covariance normalisée **Q** en **P**. Donc, dans la suite du travail **P** désignera la matrice de covariance nomaliseé par σ_v^2. Nous présentons à la Figure 4.1 le programme Alpha pour une étape de filtrage de Kalman.

La correspondance exacte entre les variables du programme Alpha et celles des équations est résumée dans le Tableau 4.1.

```
-- Une étape de Kalman de covariance
system OneStep :{M | 2<=M}
  (yb :  real;
  Ip : {m,i | 1<=m<=M; 1<=i<=M} of real;
  bbt : {m,i | 1<=m<=M; 1<=i<=M} of real;
  phi : {m,i | 1<=m<=M; 1<=i<=M} of real;
  phit :{m,i | 1<=m<=M; 1<=i<=M} of real;
  H : {m | 1<=m<=M} of real;
  Hz : {m | 1<=m<=M} of real;
  xchapz : {m | 1<=m<=M} of real;
  P : {m,i | 1<=m<=M; 1<=i<=M} of real);
returns (K : {m | 1<=m<=M} of real;
  xchap : {m | 1<=m<=M} of real;
  Pkk :{m,i | 1<=m<=M; 1<=i<=M} of real);
var
        xint : {m | 1<=m<=M} of real;
        Ychap :  real;
        I :  real;
        Veint :  real;
        invVe :  real;
        V2 : {m | 1<=m<=M} of real;
        V3 : {m | 1<=m<=M} of real;
       PP, PP1, V1 : {m,i | 1<=m<=M; 1<=i<=M}
of real;
        Pint : {m,i | 1<=m<=M; 1<=i<=M} of
real;
      let
        -- Équation (3.71)
        use  matvect[M] (phi, xchapz) returns
(xint) ;
        -- Équation (3.75)
        use  dotprod[M] ( xint, H) returns
(Ychap) ;
        I[] = yb[] - Ychap[];
        -- Équation (3.72)
        use  matmult[M] (P, phit) returns
(V1);
        use  matmult[M] (phi, V1) returns
(PP1);
        PP = PP1 + bbt;
        -- Équation (3.73)
        use  matvect[M] (PP, H) returns  (V2)
;
        use  dotprod[M] (Hz, V2) returns
(Veint) ;
        -- Équation (3.74)
        use  matvect[M] (PP, Hz) returns
(V3) ;
        invVe[] = 1 / (Veint[]+1[]);
        K[m] = V3[m] * invVe[];
        -- Équation (3.75)
        xchap[m] = xint[m] + K[m] * I[];
        -- Équation (3.76)
        Pint[m,i] = Ip[m,i] - K[m] * H[i];
        use  matmult[M] (Pint, PP) returns
(Pkk) ;
      tel;
```

a)

```
-- Multiplication matrice matrice
-- Inputs: a, b:square matrices of size M
-- Outputs: c: square matrix of size M
system matmult : {M |M>1}
   (a,b : {i,j | 1<=i,j<=M} of real)
returns
   (c : {i,j | 1<=i,j<=M } of real);
var
   C : {i,j,k | 1<=i,j<=M; 0<=k<=M} of real;
Let
        c[i,j] = C[i,j,M];
        C[i,j,k] = case
              {|k=0} : 0[];
              {|1<=k<=M} : C[i,j,k-
1]+a[i,k]*b[k,j];
          esac;
       tel;

       -- Multiplication matrice vecteur
       -- Input: a: a square matrix of size M
       --         v: a vector of size M
       -- Output: c: a vector of size M
       system matvect : {M | M>1}
                        (a : {i,j | 1<=i,j<=M}
of real;
                         v : {i | 1<=i<=M} of
real)
              returns  (c : {i | 1<=i<=M} of
real);
       var
              C : {i,j | 1<=i<=M; 0<=j<=M} of
real;
       Let
              C[i,j] = case
                        {| j=0} : 0[];
                        {| j>=1} : C[i,j-1] +
a[i,j]*v[j];
                         esac;
              c[i] = C[i,M];
       tel;

       -- produit Scalaire
       -- Input: v, w: two M vectors
       -- Output: s: a scalar
       system dotprod : {M | M>1}
                        ( v, w : {i | 1<=i<=M}
of real)
              returns  (s : real);
       var
              S : {i | 0<=i<=M} of real;
       Let
              S[i] = case
                       {| i=0} : 0[];
                       {| i>=1} : S[ i-1 ] +
v[i]*w[i];
                         esac;
              s[] = S[M];
tel;
```

b)

Figure 4.1 : Programme Alpha pour le filtre de covariance. a) Programme prinical b) Sous-programmes

Résultats d'ordonnancement de MMAlpha

Après avoir programmé et simulé les équations du filtre de Kalman en Alpha, on procède à la recherche d'un ordonancement des variables et expressions. Les résultats de l'ordonnancement sont donnés au Tableau 4.1.

La prémière colonne présente les équations à ordonnancer et dans la deuxième colonne est marquée le nom de la variable correspondante. Dans la troisième colonne indique l'opération à ordonnancer et les temps de calcul donnés par MMAlpha sont montrés dans la quatrième colonne. Le nombre total de cycles d'horloge est donné à la cinquième colonne deuxième partie (avec pipeline). En effet, les opérations de multiplication matrice-matrice et matrice-vecteur ont été pipelinées. La première partie de la cinquième colonne montre la durée totale d'exécution de ces variables dans le cas où elles n'auraient pas été pipelinées.

L'opération de pipelinage consiste ici à propager les variables à multiplier au lieu de le diffuser comme c'est le cas dans une système non pipeliné. L'avantage est évident. Notre architecture est plus rapide lorsqu'il y a pipelinage. Elle a besoin de 7M+6 cycles d'horloge pour filtrer un échantillon alors qu'elle aurait eu besoin de 16M le cas contraire. Cette opération de pipelinage est expliquée plus en détail à la section 4.1.3.

Tableau 4.1 : Cycles d'horloges dérivés du scheduling donnée par MMAlpha

<table>
<tr><th rowspan="2">Eq.</th><th rowspan="2">Variable</th><th rowspan="2">Opération</th><th rowspan="2">Ordonnancement donné par MMAlpha
i=1,2,…,M
m=1,2,…,M</th><th colspan="2">Nombre de cycles d'horloge</th></tr>
<tr><th>sans pipeline</th><th>avec pipeline</th></tr>
<tr><td>(3.71)</td><td>$\hat{x}^{int}$</td><td>$\Phi\hat{x}_{k/k}$</td><td>1+m+M</td><td>2M-1</td><td>M</td></tr>
<tr><td rowspan="2">(3.75)</td><td>$\hat{y}$</td><td>$\hat{x}^{int}h_{k+1}$</td><td>7+2M</td><td>M</td><td rowspan="3">M</td></tr>
<tr><td>δ_y</td><td>$\tilde{y}_{k+1}-\hat{y}$</td><td>8+2M</td><td>0</td></tr>
<tr><td rowspan="2">(3.72)</td><td>A_1</td><td>$P_{k/k}\,\Phi^T$</td><td>2+i+m+M</td><td>3M-1</td></tr>
<tr><td>$P_{k+1/k}$</td><td>$\Phi A_1 + b\beta b^T$</td><td>1+i+m+2M</td><td>3M-1</td><td>M</td></tr>
<tr><td>(3.73)</td><td>A_2</td><td>$P_{k+1/k}h_k$</td><td>1+m+3M</td><td rowspan="2">2M-1</td><td rowspan="2">M</td></tr>
<tr><td>(3.74)</td><td>A_3</td><td>$P_{k+1/k}h_{k+1}$</td><td>2+m+4M</td></tr>
<tr><td>(3.73)</td><td>V^{int}</td><td>$h_k A_2$</td><td>1+4M</td><td>M</td><td>M+1</td></tr>
<tr><td rowspan="2">(3.74)</td><td>V_{k+1}^{-1}</td><td>$1/(V^{int}+1)$</td><td>3+4M</td><td>1</td><td>1</td></tr>
<tr><td>K_{k+1}</td><td>$A_3 V_{k+1}^{-1}$</td><td>3+m+4M</td><td>2</td><td>2</td></tr>
<tr><td>(3.75)</td><td>$\hat{x}_{k+1/k+1}$</td><td>$\hat{x}^{int} + \delta_y K_{k+1}$</td><td>5+m+4M</td><td>2</td><td>2</td></tr>
<tr><td rowspan="2">(3.76)</td><td>P^{int}</td><td>$I - K_{k+1}h_{k+1}$</td><td>4+i+m+4M</td><td>M</td><td>M</td></tr>
<tr><td>$P_{k+1/k+1}$</td><td>$P^{int}P_{k+1/k}$</td><td>3+i+m+5M</td><td>3M-1</td><td>M</td></tr>
<tr><td colspan="4">Total</td><td>16M</td><td>7M+6</td></tr>
</table>

4.1.3 Description de l'architecture systolique

À partir de l'ordonnancement donné par MMAlpha (Tableau 4.1), on déduit l'architecture globale possible. À ce niveau, l'approche reste encore intuitive. Il faut analyser le temps de calcul obtenu et trouver la meilleure façon de représenter le calcul en question sur une architecture systolique. La forme de cette architecture est surtout donnée par le nombre maximal d'opérations à éxécuter simultanément, et dans une certaine mesure par la topologie du domaine des variables mises en jeu. Dans notre cas, le nombre maximal

d'opérations élémentaires à exécuter simultanémment est $O(M^3)$ qui correspond au produit matrice-matrice résultant à l'obtant des variables $\hat{\mathbf{x}}^{int}$, $\mathbf{A}_1$, $\mathbf{P}_{k+1/k}$, $\mathbf{A}_3$, $\mathbf{P}^{int}$, $\mathbf{P}_{k+1/k+1}$. Ces opérations peuvent être réduites à exactement M^2 si on combine les multiplications et additions dans une seule unité. On utilisera donc des multiplieurs accumulateurs dont les architectures donnent l'avantage d'être plus rapides que la somme d'une multiplication et d'une addition consécutives. On voit donc que l'on aura besoin de M^2 multiplieurs accumulateurs pour éxécuter un produit de matrices carrées. On peut donc conclure que l'on aura besoin d'une architecture systolique à topologie carrée. Cette architecture est présentée à la Figure 4.2 pour un problème de dimension M=3.

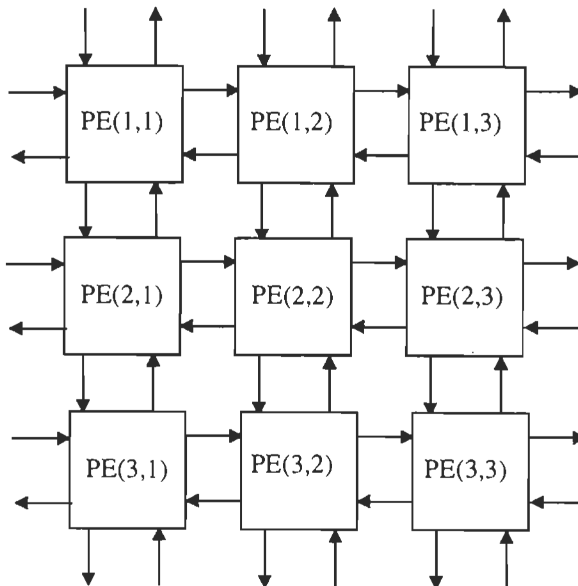


Figure 4.2 : Architecture systolique à topologie carrée

L'architecture globale est donc un réseau carré de MxM processeurs élémentaires. Elle est donnée à la Figure 4.3 pour M=3. Elle prendrait 16M cycles d'horloge par échantillon, mais en pipelinant les opérations matricielles, elle nécessite seulement 7M+6

cycles d'horloges par échantillon. Chaque processeur élémentaire est un multiplieur accumulateur plus des signaux de contrôle qui définissent le mode de fonctionnement du processeur. Les signaux de contrôles permettent de sélectionner les données d'entrée à mettre dans l'unité arithmétique, et sélectionne aussi les données à mettre sur la sortie du processeur élémentaire. Un diviseur est nécessaire pour calculer $1/(V^{int}+1)$. Par conséquent, l'architecture globale possède M^2+1 processeurs élémentaires disposés comme indiqué à la Figure 4.3. Un des principaux avantages que présente cette architecture est que plusieurs données intermédiaires sont utilisées immédiatement après qu'elles aient été calculées pour pipeliner l'ordonnancement. Par conséquent, elles restent dans le réseau pour l'étape suivante. Ces variables sont les suivantes : $\hat{\mathbf{x}}^{int}$, $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{A}_3$, $\mathbf{V}^{int}$, $\mathbf{K}_{k+1}$ et $\tilde{\mathbf{N}}^{int}$. Néanmoins, il est nécessaire de stocker les constantes $\mathbf{b}\beta\mathbf{b}^T$ et $\ddot{\mathbf{O}}$, ainsi que certaines variables intermédiaires telle que : $\hat{\mathbf{x}}_k$, $\mathbf{h}_k$, $\mathbf{h}_{k+1}$, $\mathbf{P}_k$, $\mathbf{P}_{k/k+1}$, $\mathbf{V}_{k+1}^{-1}$ et δ_y. La Figure 4.3 montre que l'on aura besoin de deux unités de stockage placées à l'ouest et au nord du réseau de processeurs ; chacune étant constituées de M mémoires. Étant donné que $\mathbf{P}_k$ et $\mathbf{P}_{k/k+1}$ sont des matrices symétriques, on aura besoin de stocker uniquement leurs parties triangulaires inférieures (ou supérieures) respectivement.

Chaque PE possède plusieurs modes (Figure 4.5) de fonctionnement qui seront sélectionner par des signaux de contrôle. Ces modes déterminent la direction du pipelinage (ouest, est, nord ou sud) et les variables qui seront introduites dans les unités arithmétiques.

La Figure 4.4 présente le fonctionnement de l'architecture sur le calcul de l'équation (3.72) rappelée ci-dessous, en tenant compte du changement de variable $\mathbf{P} = \mathbf{Q}$:

$$\mathbf{P}(k+1/k) = \Phi\mathbf{P}(k/k)\Phi^{T} + \mathbf{b}\beta\mathbf{b}^{T}, \quad \mathbf{P}(0/0) = \mathbf{I} \tag{4.1}$$

Sur cette figure, les ports d'entrées et de sorties sont omis par souci de simplicité. Chaque PE carré fonctionne en deux modes différents, mode 1 et mode 2. La première opération $\mathbf{P}_{k/k}\Phi^{T} = \mathbf{A}_{1}$ est effectuée en mode 1, le résultat $\mathbf{A}_{1}$ est stocké dans le réseau, ensuite $\ddot{\mathbf{O}}\mathbf{A}_{1}$ est calculé en mode 2 et le résultat est additionné à $\mathbf{b}\beta\mathbf{b}^{T}$. Ce calcul nécessite que tous les M^{2} PEs carrés puissent fonctionner dans les deux modes.

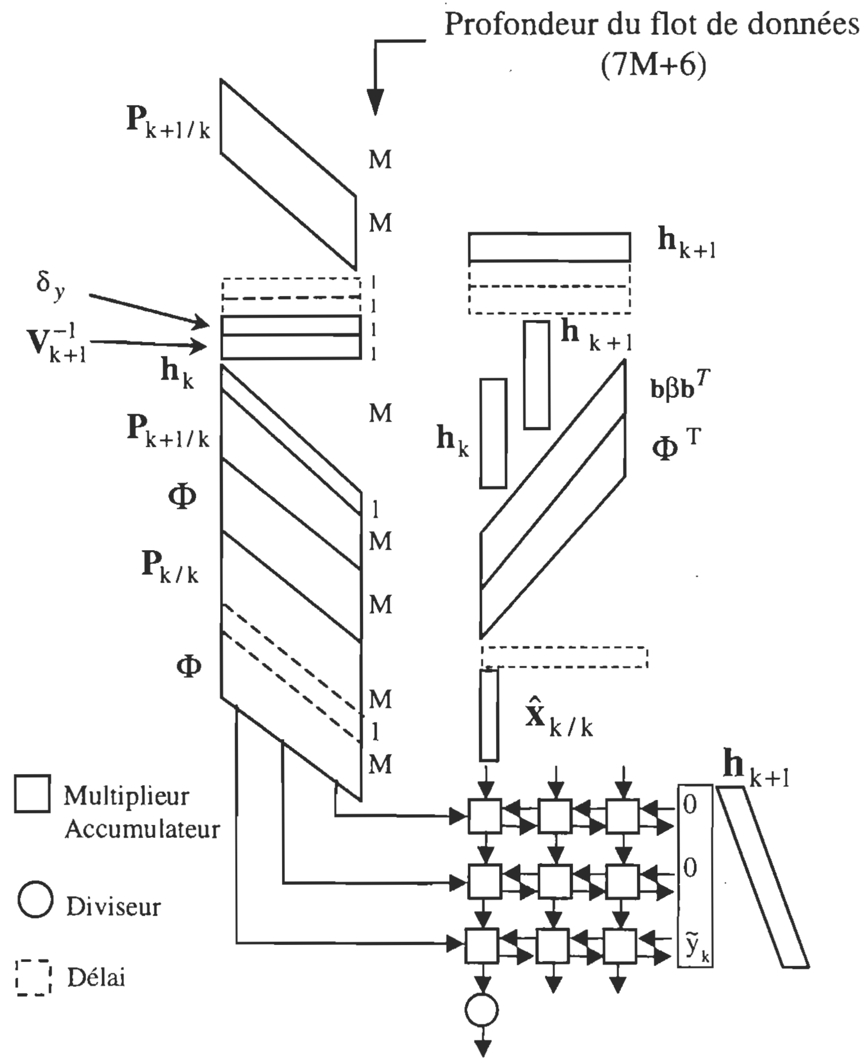


Figure 4.3 : Architecture systolique et son flot de données pour un échantillon $\hat{x}_k$.

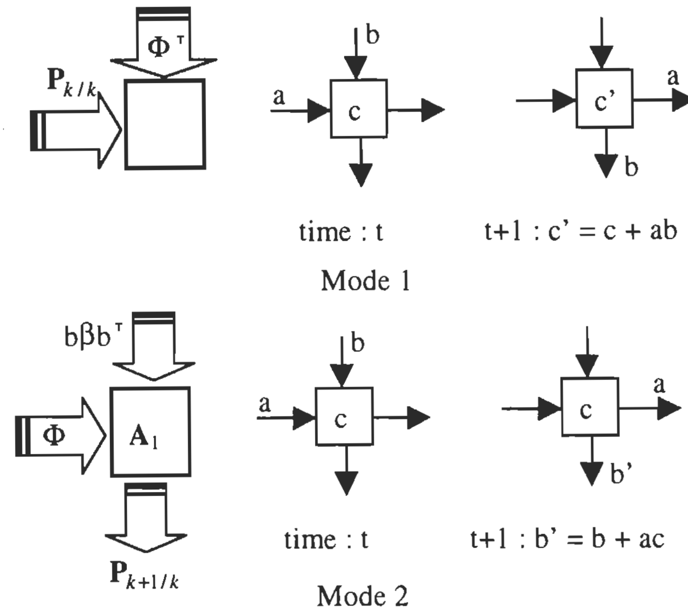


Figure 4.4 : Flot de données et cellule MAC pour l'exécution de l'équation (4.1)

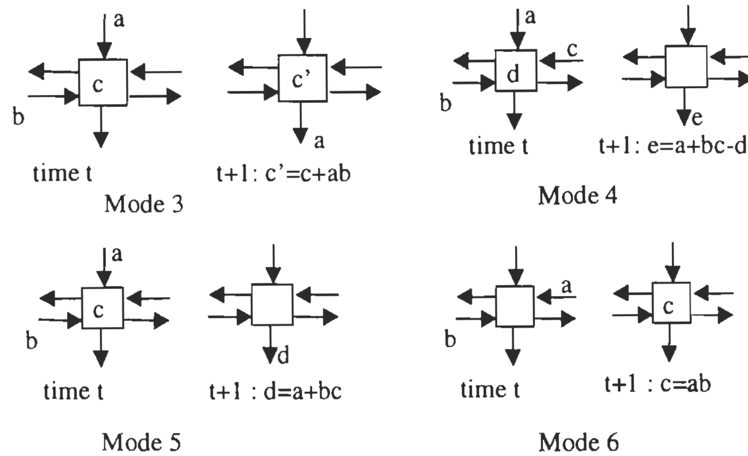


Figure 4.5 : Les différents modes de fonctionnement

Les calculs de l'architecture globale sont effectués en dix étapes :

Étape 1 : Φ et $\hat{x}_{k/k}$ sont introduits dans le réseau, $\tilde{y}_k$ est chargé dans PE(M,M), $\hat{x}^{int}$ est ensuite calculé dans la première colonne, les PEs fonctionnant en mode 3. Après M

cycles d'horloge, $\hat{\mathbf{x}}^{int}(1)$ est disponible dans PE(1,1) et est stocké dans le registre interne pour l'utilisation à l'étape suivante ; ensuite, il se déplace étape par étape vers le dernier processeur PE(1,M). Les autres éléments de ce vecteur font également la même chose immédiatement après qu'ils aient été calculés.

Étape 2 : $\hat{\mathbf{x}}^{int}$ calculé à l'étape 1 circule dans le réseau de l'ouest vers l'est et rencontre $\mathbf{h}_{k+1}$ dans la dernière colonne. $\hat{\mathbf{x}}^{int}$ et $\mathbf{h}_{k+1}$ sont multipliés élément par élément. δ_y est obtenu par accumulations successives vers le bas des résultats de multiplication précédents, comme montré à la Figure 4.4, mode 4. Notons qu'au début, $d=\tilde{y}_k$ pour PE(M,M) et d=0 pour PE(i,M), i=1,...,M. Un délai unitaire est observé avant que la donnée suivante ne soit introduite pour permettre à la multiplication de $\hat{\mathbf{x}}^{int}$ par $\mathbf{h}_{k+1}$ de continuer. Ensuite, $\mathbf{P}_{k/k}$ et $\mathbf{\Phi}^T$ sont introduits dans le réseau. $\mathbf{A}_1$ est calculé dans tous les PE du réseau carré, les processeurs fonctionnant en mode 1. Le résultat reste dans le réseau.

Étape 3 : $\mathbf{\Phi}$ et $\mathbf{b}\beta\mathbf{b}^T$ sont introduits dans le réseau. $\mathbf{P}_{k+1/k}$ est calculé, les processeurs fonctionnant au mode 2. Les résultats sortent du réseau par le sud.

Étape 4 : $P_{k+1/k}$ est introduit par l'ouest, $\mathbf{h}_k$ et $\mathbf{h}_{k+1}$ sont introduits dans les lignes 1 et 2 respectivement, avec un décalage entre les deux. $\mathbf{A}_2$ et $\mathbf{A}_3$ sont calculés par les PEs des deux premières lignes, les processeurs fonctionnant en mode 1. Les résultats sont stockés dans le réseau, aux endroits où ils ont été calculés.

Étape 5 : $\mathbf{h}_k$ est introduit par l'ouest du réseau à la suite de $P_{k+1/k}$. Il est multiplié élément par élément dans la première ligne, ensuite accumulé vers le bas pour donner $\mathbf{V}^{int}$ (mode de fonctionnement 5).

Étape 6 : Le résultat de l'étape 5 est envoyé dans le diviseur pour calculer $\mathbf{V}_{k+1}^{-1}$.

Étape 7 : $\mathbf{V}_{k+1}^{-1}$ est introduit par l'ouest dans toutes les lignes de la première colonne. Le contenu la deuxième colonne est retourné dans la première pour une multiplication élément par élément (mode 6). Le résultat $\mathbf{K}_{k+1}$ est stocké dans le réseau.

Étape 8 : δ_y est introduit par l'ouest dans toutes les lignes de la première colonne. $\hat{\mathbf{x}}^{int}$ est lu depuis les registres internes et $\hat{\mathbf{x}}_{k+1/k+1}$ est calculé. Le résultat $\hat{\mathbf{x}}_{k+1/k+1}$ est sorti et $\mathbf{K}_{k+1}$ est stocké dans les registres internes de la première colonne.

Étape 9 : $\mathbf{K}_{k+1}$ est lu depuis les registres internes, $\mathbf{h}$ est introduit par le nord du réseau. $\mathbf{K}_{k+1}$ circule de l'ouest vers l'est et rencontre $\mathbf{h}_{k+1}$ qui circule du nord vers le sud pour une multiplication élément par élément. La matrice obtenue est soustraite de la matrice identité $\mathbf{I}$. Le résultat P^{int} est stocké dans le réseau.

Étape 10 : $P_{k+1/k}$ est introduit par l'ouest, et $P_{k+1/k+1}$ est calculé. Le résultat sort du réseau par le sud. Tous les processeurs fonctionnent en mode 2.

Après avoir programmé le filtre de Kalman standard (équations (3.71) à (3.76)) en Alpha, nous avons obtenu les mêmes résultats que ceux générés par Matlab® après

simulation. Cette architecture a été publiée dans [MOZ98] dont une copie est donnée en Annexe I.

4.2 Filtre racine carrée de covariance

Les équations du filtre racine carrée de covariance décrites dans la sous-section 3.1.3 peuvent être implantées en utilisant un algorithme adéquat de triangularisation de matrices. Il en existe plusieurs et le choix dépend de la stabilité numérique de la méthode, de la précision des résultats et de la régularité. Un autre facteur est de s'assurer de la possibilité de la mise en parallèle de l'algorithme afin d'en déduire une architecture systolique. Les algorithmes de triangularisation de matrices couramment utilisés en traitement numérique de signaux et en commande sont : *Gram-Schmidt, Gram-Schimdt modifié, HouseHolder* [KAM71] *et Givens* [KUN91], [GAS89], [GAS88].

Les algorithmes de *Gram-Schimdt modifié* et de *Givens* nécessitent les plus grands nombres de racines carrées, multiplications et additions mais par contre, ils sont stables et précis. De plus, l'algorithme de Givens (basé sur les rotations de Givens) présente une structure très régulière qui le rend approprié pour en dériver une architecture systolique. L'architecture systolique basée sur les rotations de Givens la plus efficace et la plus utilisée est le réseau de Gentleman et Kung décrit dans [QUI89a], chapitre 4. Nous allons présenter ci-dessous ce réseau et expliquer son fonctionnement, et montré le programme Alpha qui l'implante. Ensuite nous l'utiliserons comme base dans notre processus de dérivation de l'architecture systolique du filtre racine carrée de covariance. Les procédures de

triangularisation par les algorithmes de Gram-Schmidt modifié et de HouseHolder peuvent être consultées dans [KAM71], mais ces deux algorithmes sont difficilement parallélisables car ils ne sont pas réguliers. D'autres procédures telles que la décomposition LU et les algorithmes de Gauss et de Jordan sont également expliquées dans [QUI89a].

4.2.1 Triangularisation de matrices denses dans MMAlpha

L'architecture ci-dessous basée sur les transformations de Givens et appelée couramment triangularisation QR a pour objectif de trouver une transformation orthogonale T telle que :

$$TA = \mathbf{W} \tag{4.2}$$

où **A** est la matrice dense à triangulariser et **W** est la matrice triangulaire supérieure obtenue. Pour cela, on utilise une série de matrices orthogonales $\mathbf{\Omega}_{ik}$ choisies de façon à annuler le coefficient en position (i,k) de la matrice **A** :

$$\begin{array}{l} pour\ k = 1\ à\ M-1 \\ pour\ i = k+1\ à\ M \\ \qquad \begin{pmatrix} ligne\ k \\ ligne\ i \end{pmatrix} = \mathbf{\Omega}_{ik} . \begin{pmatrix} a_{kk} \\ a_{ik} \end{pmatrix} \end{array} \tag{4.3}$$

où la matrice $\mathbf{\Omega}_{ik}$ est choisie de telle sorte que :

$$a_{kk} = \mathbf{\Omega}_{ik}.(a_{kk}) \tag{4.4}$$

En général, on utilise des matrices de factorisation orthogonales appelées matrices de Givens car elles ont pour avantage de conserver la régularité du réseau systolique implémentant l'algorithme :

$$\Omega_{ik} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \quad (4.5)$$

où

$$\theta = \arctan\left(\frac{a_{ik}}{a_{kk}}\right) \quad (4.6)$$

Le réseau de Kung et Gentleman est constitué de M(M+1)/2 + M processeurs connectés orthogonalement, sous forme triangulaire, où M est la dimension du système. Dans notre implémentation, nous n'avons pas besoin de manipuler le second membre de l'équation **Ax = b**, par conséquent, notre réseau sera constitué de M(M+1)/2 processeurs connectés comme indiqué à la Figure 4.6.

Le réseau comporte M lignes et chaque ligne k comprend M-k+1 processeurs numérotés de droite à gauche PE(k,1), ... , PE(k,M+1-k). La matrice **A** entre par le haut comme indiqué sur la Figure 4.6.

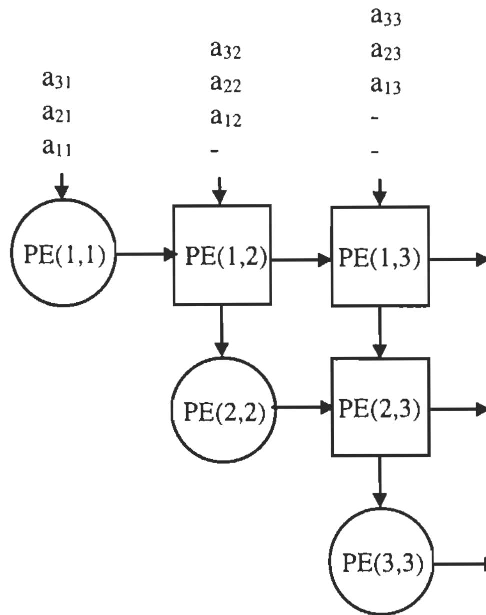
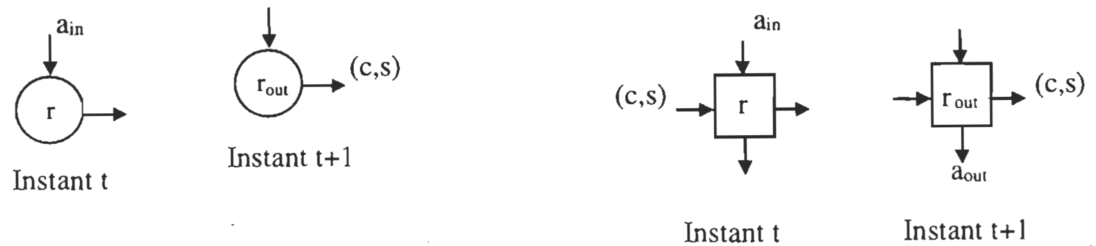


Figure 4.6. Architecture systolique à topologie triangulaire, appliquée à la triangularisation d'une matrice A avec dim(A)=M=3

Les cellules rondes (cellules diagonales) sont chargées de générer la rotation θ pour i>k, et les cellules carrées (cellules hors diagonales) sont chargées de l'appliquer. Ainsi, à chaque étape k, le coefficient en position (i,k) pour i>k est annulé. Le fonctionnement des cellules est expliqué dans la Figure 4.7 et le programme Alpha qui implante ces rotations de Givens est donné à la Figure 4.9.



a) b)

```
Si init alors
        {initialiser le registre interne}
        début
                r:=ain;
                init := faux;
        fin
sinon
        {générer une rotation}
        (c,s,rout)=GENERER(r,ain)

fonction (c,s,x)=GENERER(x,y)
{rotation de (x,y) pour annuler y}
si y=0 alors
        début
                c:=1;
                s:=0;
        fin
sinon
        si |y|>=|x| alors
                début
                        t:=x/y;
                        s:=1/sqrt(1+t*t);
                        c:=s*t;
                fin
        sinon
                début
                        t:=y/x;
                        c:= 1/sqrt(1+t*t);
                        s:=c*t;
                fin
x:=c*x + s*y;
```

c)

```
Si init alors
        {initialiser le registre interne}
        début
                r:=ain;
                init := faux;
        fin
sinon
        {appliquer la rotation}
        début
          (aout,rout,c,s)=APPLIQUER(r,ain,c,s)
        fin

Fonction (x,y,c,s) = APPLIQUER (x,y,c,s)
{appliquer la rotation (c,s) au couple (x,y)}
temp:=x;
x:=c*temp + s*y;
y:=-s*temp + c*y;
```

d)

Figure 4.7. Fonctionnement des cellules rondes pour la factorisation de Givens : a) Flot de données pour processeurs ronds, b) flot de données pour processeurs carrées, c) algorithme pour un processeur rond, d) algorithme pour un processeur carré [QUI89a].

```
function W = givens(A);

m = length(A(:,1));, n = length(A(1,:));, mdim = min(m,n);
aout_1(1,1) = 0; c(1,1) = 0; s(1,1) = 0; r(1,1) = A(1,1);, aout = [];
for i=1:m                    % PE initialization times
       for j=i:n
              if i==1, t_init(1,j) = j;
              else t_init(i,j) = t_init(i-1,j)+2;
              end
       end
end
nb_step = 2*min(m,n) + max(m,n) -1;
for j=1:n, input(j:j+m-1,j) = A(:,j);, end
for step = 1:nb_step
       for i=1:mdim
              for j=i:mdim
       active(i,j) = (t_init(i,j)<=step & step<=t_init(i,j)+m-i);
       init(i,j) = not(step>t_init(i,j));
              if active(i,j)==1
                     if i==1
                            ain(i,j) =  input(step,j);
                     else
                            ain(i,j) = aout_1(i-1,j);
                     end
                     if i==j
                            if step == t_init(i,j)
                                   r(i,j) = ain(i,j);
                                   init(i,j) = 0;
                            else
                                   out = gen_diag_cell(ain(i,j),r_1(i,j));
              r(i,j) = out(3);, c(i,j) = out(1);, s(i,j) = out(2);
                            end
                     else
                            if step == t_init(i,j)
                                   r(i,j) = ain(i,j);, init(i,j) = 0;
                            else
out = gen_of_diag_cell(ain(i,j),r_1(i,j),c_1(i,j-1),s_1(i,j-1));
r(i,j)=out(3);,     c(i,j)=out(1);, s(i,j)=out(2);, aout(i,j)=out(4);
                            end
                     end
              end
       end
end
aout_1 = aout;, c_1 = c;, s_1 = s;, r_1 = r;
end
      W = r;
```

Figure 4.8 : Programme général matlab de triangularisation de matrices

```
-- Givens factorisation, spécialisée pour dimension (M+1)*M
      system givensm1m : {M |M>1}
        (a : {i,j | 1<=i<=M+1; 1<=j<=M} of real)
      returns
        (givens : {i,j | 1<=i<=M+1; 1<=j<=M } of real);
      var
        A : {i,j,k | 0<=k<=M; k<i<=M+1; i>=1; k<=j<=M; j>=1} of real;
        Piv : {i,j,k | k<=i<=M+1; k<=j<=M;1<=k<=M+1} of real;
        C,S,T : {i,k | 1<=k<=M;k<i<=M+1} of real;
        Swap : {i,k | 1<=k<=M;k<i<=M+1} of boolean;
      let
        Swap[i,k] = Piv[i-1,k,k]>A[i,k,k-1];
```

```
        T[i,k] = if Swap[i,k] then Piv[i-1,k,k]/A[i,k,k-1] else A[i,k,k-1]/Piv[i-
1,k,k];

        C[i,k] = if (A[i,k,k-1]=0[]) then 1[] else
                    (if (Swap[i,k]) then 1[]/sqrt(1[]+T[i,k]*T[i,k])*T[i,k]
                                     else 1[]/sqrt(1[]+T[i,k]*T[i,k]));
        S[i,k] = if (A[i,k,k-1]=0[]) then 1[] else
                  (if (not Swap[i,k]) then 1[]/sqrt(1[]+T[i,k]*T[i,k])*T[i,k]
                                       else 1[]/sqrt(1[]+T[i,k]*T[i,k]));

        Piv[i,j,k]=case
        {|  i=k}: A[i,j,k-1];
        {|  i>k }:  C[i,k]*Piv[i-1,j,k]+S[i,k]*A[i,j,k-1];
        esac;

        A[i,j,k] =
          case
            {| k=0 }: a[i,j]; -- initialisation
            {| k>0; i>k; j>=k }: -S[i,k]*Piv[i-1,j,k]+C[i,k]*A[i,j,k-1];
          esac;
        givens[i,j] =
          case
            {| i>j}: 0[];
            {| i<=j}: Piv[M+1,j,i];
          esac;
tel;
```

Figure 4.9 : Programme général Alpha de triangularisation de matrices

La première donnée valide reçue par un processeur sert simplement à initialiser le registre interne du dit processeur : son fonctionnement est donc contrôlé par la variable interne *init* initialisée à *vrai* et prenant la valeur *faux* après l'entrée de la première donnée. Les autres données entrent de façon systolique dans les processeurs, sont transformées et transmises aux processeurs voisins, toujours de façon systolique.

L'opération totale de triangularisation se déroule en 3M-1 étapes car la dernière opération a lieu à l'instant 3M-1 dans le processeur PE(M,1) (processeur de la dernière ligne). Mais à partir de l'instant M, le processeur PE(1,1) est libre et peut être utilisé pour

l'opération suivante. Ainsi on peut dire que le temps nécessaire dans cette triangularisation est de M cycles.

4.2.2 Programmation du filtre racine carrée de covariance en Alpha

Pour raison de conformité, effectuons les changements de variables normalisées suivantes sur le filtre racine carrée de covariance :

$$\mathbf{S} = \mathbf{L}\ ,\ F=F_1,\ \beta = u \qquad (4.7)$$

Dans la suite du travail, la racine carrée de la matrice de covariance normalisée sera donc notée **S**.

Le programme du filtre de Kalman racine carrée de covariance diffère de celui de Kalman standard (Figure 4.1) car il intégre une procédure de triangularisation de matrices. Le programme général est donné à la Figure 4.10.

```
-- produit scalaire (Voir Figure 4.1)
system dotprod :{M | 2<=M}
                (v : {i | 1<=i<=M} of real;
                 w : {i | 1<=i<=M} of real;
returns (s :  real);
        ...
-- matlvect : retourne un vecteur. Entrée 'a' est une matrice triangulaire inférieure.
system matlvect : {M | M>1}
              (a : {i,j | 1<=i,j<=M; i<=j<=M} of real;
               v : {i | 1<=i<=M} of real)
returns      (c : {i | 1<=i<=M} of real);

-- Une étape de Kalman racine carrée de covariance
system sqrtcov : {M | 1<M}
              (yb   : real;
               H    : {m | 1<=m<=M } of real;
               xhatp1 : {m | 1<=m<=M } of real;
               Sp1 : {m,i | 1<=m<=M; 1<=i<=M } of real;
                          sigmav2, sigmaw2 :  real)
 returns (xe : real;
             Sp : {m,i | 1<=m<=M; 1<=i<=M } of real;
             xhatp : {m | 1<=m<=M } of real);
```

```
var
        xhat,xhatextra : {m | 1<=m<=M } of real;
        Hextra,Hextra1: {m | 1<=m<=M } of real;
        A,Aextra : {m,i | 1<=m<=M+1; 1<=i<=M } of real;
        B,C : {m,i | 1<=m<=M+1; 1<=i<=M+1 } of real;
        U,V,f : real;
        St,Stextra : {m,i | 1<=m<=M; m<=i<=M} of real;
        Sth,g : {m | 1<=m<=M} of real;
        ye : real;
let
        U = sqrt(sigmaw2[]);
        V = sqrt(sigmav2[]);
       -- Équation (3.79)
        A[m,i] =
         case
                    {| i=1; m<=M } : Sp1[1,m];
                    {| 2<=i<=M; m<=M } : Sp1[i-1,m];
                    {| m=M+1; i=1 } : U[];
                    {| m=M+1; i>1 } : 0[];
         esac;
        Aextra = A;
       -- Équation (3.79)
        use givensm1m[M] (Aextra) returns (St); -- Premier appel de Givens
       -- Équation (3.81)
        Stextra = St;
        Hextra1 = H;
        use matlvect[M] (Stextra,Hextra1) returns (Sth);
       B[m,i] =
         case
                    {| i=1; m=1 } : V[];
                    {| 2<=i<=M+1; m=1 } : 0[];
                    {| i=1; 2<=m<=M+1 } : Sth[m-1];
                    {| 2<=m<=M+1; 2<=i<=M+1 } : St[m-1,i-1];
                  {| i+1<=m<=M+1; 2<=i}: 0[];
         esac;
       -- Équation (3.81)
        use givensm1m1[M] (B) returns (C); -- Deuxième appel de Givens
        f[] = C[1,1];
        g[i] = C[1,i+1];
        Sp[m,i] = C[m+1,i+1];
       -- Time Update. Équation (3.78)
        xhat[m] =
         case
                    {| m=1 } : xhatp1[1];
                    {| 1<m<=M } : xhatp1[m-1];
         esac;
       -- Équation (3.80)
        Hextra = H;
        xhatextra = xhat;
        use dotprod[M] (Hextra,xhatextra) returns (ye);
       -- Equation (3.80)
       xhatp[m] = xhat[m] + (g[m]/f[])*(yb[] - ye[]);
       -- Échantillon filtré
       xe[] = xhatp[M];

tel;
```

Figure 4.10 : Programme Alpha pour une étape du filtre de Kalman racine carrée de covariance

4.2.3 *Résultat d'ordonnancement de MMAlpha*

L'ordonnancement du programme de la Figure 4.10 a donné les résultats du Tableau 4.2.

Tableau 4.2 : Ordonnacement du programme du filtre racine carrée de covariance

Eq.	**Nom de la variable Alpha** (Voir Figure 4.10)	**Opération**	**Ordonnancement donné par MMalpha** $i=1,2,\dots,M$ $m=1,2,\dots,M$	**Nombre d'itérations**
(7)	St	**TA**	$m+M$	$2M$
(10)	ye	$h^T_{k+1}\hat{x}_{k+1/k}$	$1+2M$	1
(8)	B		$1+m+2M$	M
	Sth	$s^T_{k+1}h_{k+1}$	$2+m+2M$	1
	f	$C(1,1)$	$3+3M$	1
	g	$C(1,i+1)$	$3+3M$	
(9)		$\hat{x}_{k+1/k+1}$	$4+3M$	1
(15)	xhatp	$\hat{s}_{k+1}$	**$4+3M$**	0
(8)	C	**TB**	$2+m+3M$	$M-1$
	Sp	$C(m+1,i+1)$	$3+m+3M$	$M-1$
Total			**$3+4M$**	

En observant ce tableau, on remarque que l'échantillon filtré $\text{xhatp} = \hat{\mathbf{x}}(k/k)(M)$ est disponible à l'instant 4+3M. Mais l'algorithme aura besoin de M-1 cycles d'horloge suplémentaires pour calculer la racine carrée de la covariance de l'erreur d'estimation S(k+1/k+1) nécessaire pour le cycle suivant. Donc les calculs sur l'échantillon en question seront achevés à l'instant 3+4M. Mais, aussitôt que le premier élément est disponible, le filtrage de l'échantillon suivant peut commencer. On en conclut donc que le débit de

l'architecture est de 4+3M et que la latence est de 3+4M. Par conséquent on aura un échantillon filtré tous les 4+3M cycles d'horloge.

Comme nous l'avons fait avec le filtre de Kalman standard, on déduit une architecture systolique du filtre racine carrée de covariance à partir de l'ordonnancement du Tableau 4.2. L'algorithme du filtre racine carrée de covariance étant constitué principalement de deux triangularisations de matrices, nous allons constituer une architecture systolique basée sur le réseau à topologie triangulaire présenté à la Figure 4.6.

Pour cela, nous aurons besoin d'un réseau triangulaire de M+1 lignes et M+1 colonnes. Le filtrage est réalisé par le passage consécutif des équations (3.78) à (3.81) dans le réseau.

4.2.4 Description de l'architecture systolique

L'architecture totale (présentée à la Figure 4.11 pour M=3), comprend (M+1)(M+2)/2 processeurs et se déroule en 2M+5 cycles d'horloge par échantillon, plus M-1 cycles pour le vidage des matrices **S**(k+1/k) et **S**(k+1/k+1), soit au total 3M+4 cycles. En fait, après M/2 cycles de vidage de chacune de ces matrices, les opérations de l'étape suivante sont effectuées.

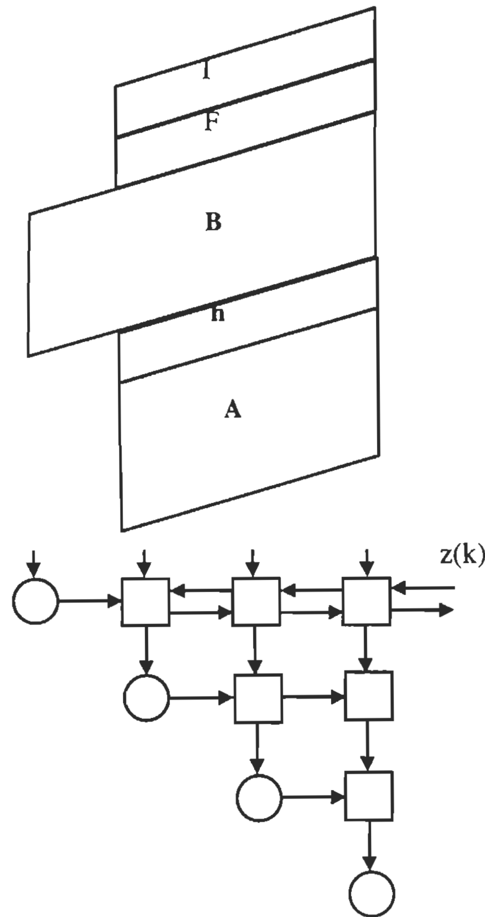


Figure 4.11 : Architecture globale

Avant de présenter l'architecture globale, rappellons les équations du filtre racine carrée de covariance, en tenant compte des changements de variables (4.7).

$$\hat{\mathbf{x}}(k+1/k) = \mathbf{\Phi}(k)\hat{\mathbf{x}}(k/k) \tag{4.8}$$

$$\left[\frac{\mathbf{S}^{\mathbf{T}}(k+1/k)}{\mathbf{0}}\right] = \mathbf{T}\left[\frac{\mathbf{S}^{\mathbf{T}}(k/k)\mathbf{\Phi}^{\mathbf{T}}(k)}{\beta\mathbf{B}^{\mathbf{T}}(k)}\right] \tag{4.9}$$

$$\begin{bmatrix} F(k+1) & \mathbf{g}(k+1) \\ \mathbf{0} & \mathbf{S}^{\mathbf{T}}(k+1/k+1) \end{bmatrix} = \mathbf{T}\begin{bmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{S}^{\mathbf{T}}(k+1/k)\mathbf{H}^{\mathbf{T}}(k+1) & \mathbf{S}^{\mathbf{T}}(k+1/k) \end{bmatrix} \tag{4.10}$$

$$\hat{\mathbf{x}}(k+1/k+1) = \hat{\mathbf{x}}(k+1/k) + \left(\mathbf{g}^T(k+1)/F(k+1)\right)\left[\tilde{y}(k+1) - \mathbf{H}(k+1)\hat{\mathbf{x}}(k+1/k)\right] \quad (4.11)$$

Les différentes étapes sont les suivantes :

Étape 1 : La matrice $\mathbf{A} = \left[\frac{\mathbf{S}^T(k/k)\Phi(k)}{\beta\mathbf{B}^T(k)}\right]$ (équation (4.9)) entre par le nord sur les M dernières colonnes. Les données traversent la première ligne sans être modifiées et entrent dans le triangle inférieur du réseau (M dernières lignes et M dernières colonnes) qui est utilisé pour réaliser la triangularisation de la matrice A de dimension (M+1) x M. Le résultat $\mathbf{S}^T(k+1/k)$ se trouve dans le réseau.

Étape 2 : Équation (4.11) : $\mathbf{H}(k)$ entre par le nord sur les M dernières colonnes du réseau ; il est multiplié élément par élément, puis accumulé, avec $\mathbf{x}(k)$ qui se trouve dans les registres internes de la première ligne (M dernières colonnes), chaque multiplication ayant lieu dans un processeur carré différent avec un retard. Au début du filtrage, les registres internes des processeurs carrés de la premières sont initialisés avec les éléments de $\hat{\mathbf{x}}(0/0)$. $\tilde{y}(k)$, entre dans le processeur PE(1,M) par l'est (La valeur de " a " est $\tilde{y}(k)$ pour le processeur PE(1,M) et 0 pour les autres processeurs). Le résultat final de cette opération est l'innovation I(k). Le mode de fonctionnement est celui indiqué à la Figure 4.12.

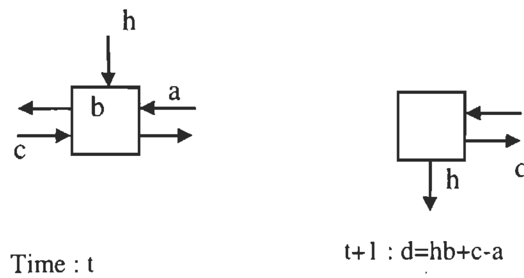


Figure 4.12. Mode d'opération pour les cellules carrées à l'étape 2.

Étape 3 : Équation (4.10) : **H**(k) entré à la suite de A dans les M dernières colonnes pour calculer I(k), continue la descente et rencontre $\mathbf{S}^T(k+1/k)$ dans le triangle inférieur. Il y a multiplication et accumulation selon le schéma de la Figure 4.13. Le résultat $\mathbf{S}^T(k+1/k)\mathbf{h}(k)$ est sorti par la l'est et aussitôt, est réintroduit dans le réseau par le nord pour effectuer la triangularisation de la matrice **B** à l'étape 4.

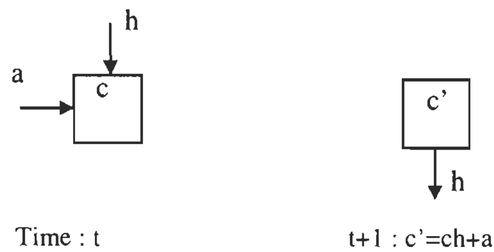
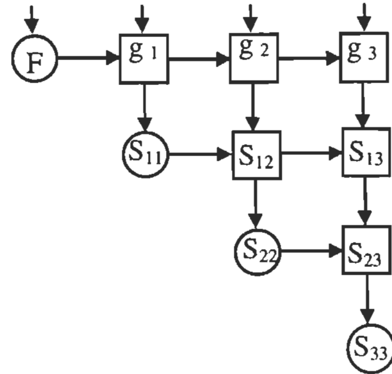


Figure 4.13. Mode d'opération pour les cellules carrées du triangle inférieur à l'étape 3.

Étape 4 : Équation (4.10) : $\mathbf{B} = \begin{bmatrix} 1 & \mathbf{0} \\ \mathbf{S}^T(k+1/k)\mathbf{H}^T(k) & \mathbf{S}^T(k+1/k) \end{bmatrix}$ entre par le nord sur tout le réseau pour triangularisation. Le résultat qui est un ensemble forme des variables F, **g** et $\mathbf{S}^T(k+1/k+1)$ est stocké dans les registres internes du réseau comme indique dans la Figure 4.14 :



$$S = S^T(k+1/k+1)$$

Figure 4.12. Disposition des variables dans les registres après l'étape 3.

Étape 5 : Équation (4.11) : **x**(k) est lu des registres internes de la première ligne (M derniers processeurs), **F** entre par le nord, l'opération **g/F** est effectuée dans les processeurs. Le résultat reste dans le réseau.

Étape 6 : Équation (4.11) : I entre par le nord ; $\mathbf{x}_+$ $\mathbf{x}_+ = \mathbf{x}$ - (**g**/F)I est calculé. Le vecteur $x = \Phi x_+$ est formé par décalage vers l'est des éléments de $\mathbf{x}_+$, Figure 4.13. Le résultat est stocké dans les registres correspondant. Parallèlement à cette opération de décalage, l'opération suivante commence dans le processeur PE(1,1).

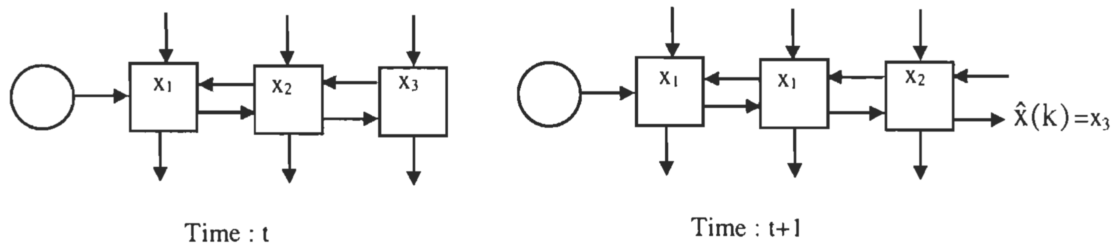


Figure 4.13. Décalage des éléments de $\mathbf{x}_+$ pour former **x**, et sortie de l'échantillon filtré

L'étude complète de cette architecture a également été publiée dans [MOZ99] dont une copie est présentée en annexe I.

4.3 Comparaison des performances

Il est intéressant de comparer des performances les architectures ainsi dérivées à celles qui ont été déjà été proposées dans la littérature.

L'étude comparative est basée sur le nombre de PE, le temps de calcul défini par le nombre de cycles d'horloge nécessaires pour filtrer un échantillon $\tilde{y}_k$, et le taux d'utilisation des PE. Le temps de calcul considéré sur la base du nombre de cycles d'horloge permet d'être indépendant de la technologie d,intégration (FPGA, CMOS, AsGa, etc.). Cette comparaison est aussi basée sur l'hypothèse selon laquelle la fréquence d'horloge est la même pour toutes les architectures, que la complexité des PE élémentaires est équivalente et que toutes les données nécessaires pour effectuer les opérations sont disponibles.

Nous introduisons ici la notion de taux d'utilisation des PE qui est le rapport du nombre moyen de processeurs actifs sur le nombre de processeurs. Le nombre moyen de processeurs actifs est défini comme étant le nombre total d'opérations arithmétiques divisé par le nombre de cycles d'horloge par itération k [KUN91] :

$$\text{Taux d'utilisation des PE} = \frac{\text{nombre d'opérations arithmétiques}}{\text{nombre cycles d'horloge} \times \text{nombre PE}} \qquad (4.12)$$

L'architecture systolique basée sur le filtre de Kalman racine carrée de covariance présentée au paragraphe précédent est difficilement comparable aux autres architectures du même genre car la nôtre est spécifiquement appliquée à l'égalisation des canaux, ce qui supprime le produit de matrice par la forme de notre matrice Φ. Toutefois, le Tableau 4.3 montre une étude comparative de l'architecture pour le filtre de Kalman de covariance avec certaines autres architectures basées également sur le filtre de Kalman de covariance. La première est le réseau trapézoïdal conçu par Irwin [IRW91]. Les deux autres architectures représentent deux schémas de la même architectures proposées par Yeh [YEH88].

Tableau 4.3 : Comparaison à d'autres architectures du filtre de covariance

Architecture	Nombre de PE	Nombre de cycles d'horloges par itération	Utilisation des PE	
			M=3	M>>1
Irwin [IRW91]	M(3M+1)/2	9M+9	41%	44%
Yeh [YEH88] Schéma A	$4M^2$	16M	13%	9%
Yeh [YEH88] Schéma B	$8M^2$	8M	13%	9%
Proposée	M^2+1	7M+6	82%	86%

À partir du Tableau 4.1, on obtient le nombre total d'opérations arithmétiques qui est : $6M^3+5M^2+5M+1$. Nous avons abouti à une architecture plus performante (en terme de nombre de cycles d'horloge par échantillon et de nombre de processeurs élémentaires) que ces architectures publiées dans la littérature. Le Tableau 4.3 montre que notre architecture présente le plus petit temps de calcul, le plus petit nombre de PE et le plus grand taux d'utilisation des processeurs.

4.4 Résultats de simulation du programme Alpha

Après avoir écrit notre programme en Alpha, nous allons le valider en effectuant une expérience de filtrage. La fonction *WriteC* de **MMAlpha** nous permet de générer le code C de notre programme. Les simulations du programme Alpha ont donné les mêmes résultats que ceux obtenus avec Matlab® et présentés à la section 3.3.

Chapitre 5

Architecture et Synthèse en Technologie VLSI

Nous avons conçu deux architectures parallèles d'égalisation des canaux en vue de leur implantation dans une technologie VLSI (CMOS ou FPGA). Nous allons choisir la plus performante basée sur des critères de comparaison comme la robustesse aux effets de quantification, le nombre de cycles d'horloge nécessaires pour filtrer un échantillon et l'efficacité de l'algorithme de reconstitution.

L'étude préalable des effets de quantification est une partie importante dans l'implantation des processeurs de signaux digitaux en ce sens qu'elle permet d'analyser le comportement du processeur face à la quantification, et par conséquent de voir si les résultats seront ceux escomptés. Les études de quantification incluent également les cycles limites qui sont un phénomène aléatoire dont il est nécessaire de tenir compte.

5.1 Étude des effets de quantification

Malgré la très grande versatilité du filtre de Kalman, sa structure matricielle montre que $O(M^3)$ opérations doivent être effectuées pendant une période d'échantillonnage où M représente la dimension du système [IRW91]. Les architectures basées sur le filtre de Kalman doivent être implantées en tenant compte de effets de la limitation du nombre de valeurs affichables des nombres sur les performances dynamiques du système commandé.

En effet, pour une représentation en virgule fixe, la limitation des valeurs affichables dans les systèmes numériques est introduite par le fait que les opérations internes sont faites en précision limitée : le processeur fonctionne donc avec un ensemble fini de valeurs représentables. Cette quantification a donc pour conséquence des erreurs dans les opérations numériques.

Il existe plusieurs lois de quantification pour la représentation de nombres réels en numérique. La méthode la plus naturelle et celle qui produit une erreur de quantification minimale est la quantification par arrondi [KUN91].

En effet, les nombres sont arrondis à la valeur représentable la plus proche (inférieur ou supérieure), qui est un multiple du pas de quantification choisi. Le pas de quantification étant la distance entre deux valeurs représentables consécutives. Les erreurs de quantification – les erreurs d'arrondi dans ce cas – sont bornées et pour une représentation en virgule fixe de pas q, elles seront toujours inférieures à q/2.

$$e(x) = |x - Q(x)| < q/2 \quad (5.1)$$

Q(x) est la représentation en virgule fixe sur un nombre de bits finis du nombre réel x.

Cette méthode de quantification n'est en réalité pas utilisée. Pour implanter cette loi de quantification, il faudrait en effet augmenter la longueur des mots d'un demi bit qui permettra de contrôler le sens de la quantification : arrondi vers le bas ou vers le haut.

En réalité, la quantification la plus simple qui est implantée dans les processeurs est la quantification pour troncature où les bits excédentaires sont tout simplement tronqués. Dans ce cas, l'erreur de quantification d'un nombre réel x est toujours bornée par le pas de quantification q :

$$e(x) = |x - Q(x)| < q \quad (5.2)$$

Avant d'effectuer la quantification proprement dite, il est nécessaire d'étudier la plage dynamique des signaux pour déterminer les valeurs maximales et minimales à représenter. Une simulation nous a permis de trouver X_{max} et X_{min} qui dépendent bien sûr du paramètre de syntonisation $\hat{a}$. Ainsi, pour harmoniser les opérations et pour pouvoir utiliser les mêmes ressources dans les différentes parties du processeur nous effectuons une normalisation pour ramener la dynamique interne dans l'intervalle [-1, +1]. Le facteur de normalisation est X_{norm} tel que :

$$X_{norm} > \max(|X_{max}|, |X_{min}|) \quad (5.3)$$

Ainsi, à l'entrée du processeur, toutes les données sont divisées par ce facteur de normalisation pour les mettre dans l'intervalle [-1, +1], et à la sortie, elles sont multipliées par ce facteur de normalisation pour les ramener à leur grandeur réelle. Lors de l'implantation, ce facteur de normalisation est choisi égal à une puissance de 2 : on évite ainsi l'utilisation d'un diviseur et d'un multiplieur en décalant simplement les données vers la gauche pour la normalisation et vers la droite pour la dénormalisation.

Une autre aspect de la loi de quantification choisie est le style de représentation des nombres négatifs. Nous avons opté pour un représentation en complément à deux parce qu'elle nous permet d'utiliser les additionneurs complets pour faire des soustractions, en mettant tout simplement à '1' la retenue d'entrée du bit de poids faible.

Nous avons effectué des simulations sous Matlab® des algorithmes du filtre de Kalman de covariance et du filtre de Kalman racine carrée de covariance en prenant la loi de quantification complète suivante :

- représentation des nombres en virgule fixe,

- nombre de bits variant de 8 à 32 avec un pas de 2,

- représentation des nombres négatifs en complément à deux,

- loi de quantification par troncature à l'intérieur de la dynamique,

- dynamique normalisée à [-1, +1],

- pas de quantification

- $q = 2/(2^n - 2)$ n : nombre de bits (5.4)

- loi de dépassement par saturation.

Les résultats de simulation sont présentés de deux façons différentes.

D'abord, on présente les erreurs de filtrage pour les deux versions du filtre pour des simulation en virgule fixe, en fonction du nombre de bits, et on les compare aux erreurs de filtrage obtenues dans le cas idéal d'une simulation en virgule flottante. L'erreur de filtrage est l'erreur quadratique moyenne $å(a, \hat{x}_q)$ donnée par :

$$\varepsilon(a, \hat{x}_q) = \frac{\|a(k) - \hat{x}_q(k)\|}{\|a(k)\|} \tag{5.5}$$

où a(k) est le signal original à l'entrée du canal et $\hat{x}_q$ est le signal filtré avec une quantification. L'erreur de filtrage dans le cas idéal d'une simulation en virgule flottante a été étudiée au paragraphe 3.3. Elle est donnée par :

$$\varepsilon(a, \hat{x}_\infty) = \frac{\|a(k) - \hat{x}_\infty(k)\|}{\|a(k)\|} \tag{5.6}$$

On présente ensuite les erreurs de quantification en fonction du nombre de bits. Les erreurs de quantifications ici représentent l'erreur quadratique moyenne entre le filtrage en virgule flottante et le filtrage en virgule fixe :

$$\varepsilon(\hat{x}_{\infty}, \hat{x}_q) = \frac{\left\| \hat{x}_{\infty}(k) - \hat{x}_q(k) \right\|}{\left\| \hat{x}_{\infty}(k) \right\|} \tag{5.7}$$

Notons que par souci d'économie de surface lors de l'implantation en VLSI, les constantes on été représentées sur un nombre de bits inférieur de 4 au nombre de bits des variables.

Nous avons retrouvé les paramètres de normalisation selon l'équation (5.3) suivants lors de la simulation en virgule flottante pour $\sigma_v^2 = 0.1$ et $\sigma_w^2 = 100$:

Filtre de covariance : $X_{max} = 1630$, $X_{min} = -959$, d'où $X_{norm} = 2048$.

Filtre de covariance racine carrée : $X_{max} = 44.1$, $X_{min} = -31$, d'où $X_{norm} = 64$.

Si on Considère le canal variant dont la réponse impulsionnelle est définie par les équations (3.65) et (3.66), les figures suivantes donnent les erreurs de quantification pour un signal de rapport signal sur bruit SNR = 20dB.

Les courbes des erreurs de quantification des Figure 5.1 et Figure 5.2 montrent clairement que le filtre de covariance standard est très affecté par les effets de quantification.

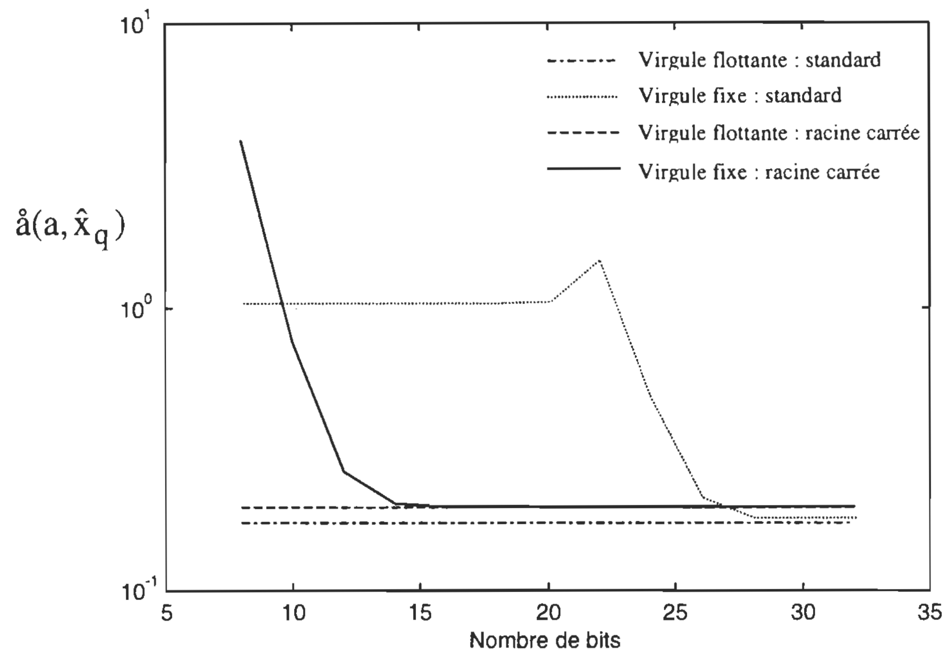


Figure 5.1 : Erreur de filtrage en fonction du nombre de bits

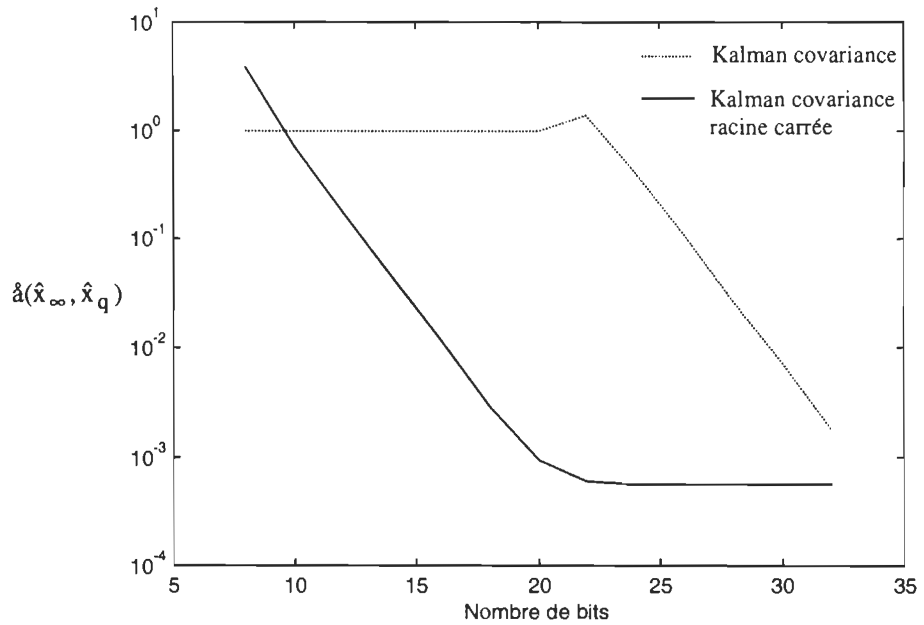


Figure 5.2 : Erreur de quantification en fonction du nombre de bits

En effet, la quantification en dessous de 20 bits est très inefficace car elle produit des valeurs nulles pour la variable V_{k+1} qui doit être ensuite inversée : dans ces conditions, cet algorithme provoque une division par zéro, et la sortie du filtre est complètement erronée. Ceci est dû au fait que la plage dynamique des matrices de covariance est très large. La version filtre racine carrée élimine cette anomalie car elle propage plutôt la racine carrée de la matrice de covariance, ce qui diminue la plage dynamique des signaux, permettant ainsi d'avoir une double précision par rapport à la version standard. Les plages de variation des signaux attestent la nature racine carrée de ce filtrage car dans ce dernier cas la plage de variation est [-31, 44.1] qui est approximativement la racine carrée de [-959, 1630]. Ainsi, si on représente un nombre sur 2n bits dans le filtre de Kalman racine carrée, ce même nombre ne pourra être représenté que sur n bits seulement dans la version standard, d'où la nature double précision de filtre racine carrée.

5.2 Choix de l'architecture et de la technologie VLSI

5.2.1 Choix de l'architecture

Malgré la complexité accrue des calculs dans le filtre racine carrée de covariance, cette version est plus propice à une implantation dans un processeur numérique que la version standard. L'étude précédente nous permet non seulement de choisir le bon filtre à implanter, mais également le nombre de bits de quantification des variables et des constantes dans le filtre choisi.

En observant la Figure 5.1 et la Figure 5.2, on remarque que le filtre racine carrée pourrait probablement fonctionner sur 16 bits (12 bits pour les constantes) mais il persiste une incertitude au niveau des résultats. Pour plus de sécurité, on prend 20 bits (16 bits pour les constantes). Notons néanmoins que la valeur du nombre de bits est un paramètre que l'on peut modifier à un endroit unique dans les programmes VHDL que nous avons écrits. Cette remarque est également valable pour toutes les constantes du programme qui sont paramétrables et donc on peut modifier à un endroit unique dans le programme VHDL.

Un autre argument de taille milite en faveur du choix de la version racine carrée est qu'elle nécessite un nombre d'itérations par échantillon inférieur à celui de la version standard. Le Tableau 5.1 montre la comparaison de ces nombres d'itérations pour les deux architectures.

Tableau 5.1 : Taille et vitesse des filtres de Kalman

Filtre	Nombre de PE	Nombre de cycles d'horloge
Standard	M^2+1	7M+6
Racine carrée	(M+1)(M+2)/2	2M+5

En consultant ce tableau, les avantages du filtre racine carrée sont évidents. La surface est deux fois plus petite et le nombre de cycle d'horloge par itération est encore plus petite : 2M+5 contre 7M+6.

Notons en passant que la forme filtre de racine carrée de covariance a été préférée au filtre racine carrée d'information à cause de la singularité de la matrice d'état. En effet, la version filtre d'information nécessite l'inversion de la matrice Φ que nous ne pouvons faire ici.

5.2.2 Choix de la technologie

Les technologies VLSI dont nous disposons au Laboratoire d'Algorithmes et Architectures Intégrés pour réaliser l'implantation sont : CMOS 1.5ìm de Mitel (Mitel15), CMOS 0.5ìm de HP (CMOSIS5) et une carte FPGA Xilinx 4036.

Le premier choix qui est venu à l'esprit a été la technologie CMOS 0.5 ìm . Tout d'abord parce qu'elle est plus récente, mais aussi parce que la surface de notre processeur une fois synthétisée sera plus petite. De plus, la vitesse du processeur sera plus élevée avec la technologie 0.5ìm comparée à la technologie 1.5ìm . La carte FPGA (Xilinx 4036) ne peut être utilisée dans le cadre de ce projet car elle est de dimension insuffisante.

5.3 Modélisation et résultats de simulation du VHDL du processeur

5.3.1 Modélisation VHDL du processeur

La modélisation de cette architecture a été faite dans Mentor Graphics® et la structure globale du processeur, nommé **SRCKAL,** est montrée à la Figure 5.3.

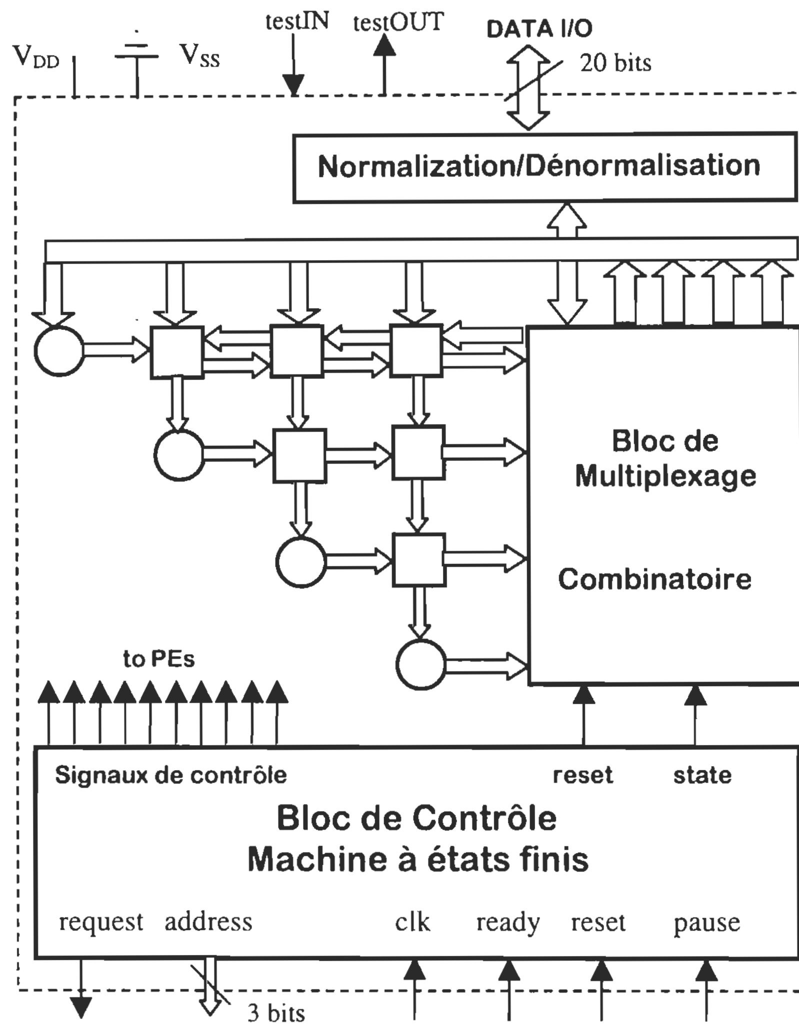


Figure 5.3 : Diagramme bloc du processeur **SRCKAL**

Le processeur SRCKAL comprend le réseau de 10 processeurs élémentaires, donc 4 diagonaux (ronds) et 6 non diagonaux (carrés). Les processeurs ronds qui sont chargés de générer des rotations comprennent une diviseur et une racine carrée.

L'algorithme de division est basé sur la méthode de Newton-Raphson [HEN90] et se fait en trois itérations. Le point de départ est un choix sur dix valeurs différentes

uniformément réparties sur l'intervalle [1, 2[comme le veut l'algorithme d'inversion. Ainsi, un diviseur comporte six multiplications et trois additions.

L'algorithme de racine carrée est une approximation polynomiale d'ordre 3 sur l'intervalle des valeurs possibles qui est [1, 2]. Étant donné que ces valeurs sont connues à l'avance, nous avons jugé intéressant de faire cette approximation qui calcule la racine carrée sur 20 bits avec une erreur relative de 0.1%. Mais ces réalisations pourraient être améliorées à l'avenir en adaptant les multiplieurs et diviseurs disponibles dans la librairie **Syn-AdvMath©** de **Synopsys®** pour faire des opérations en virgule fixe. Des architectures systoliques pour résoudre à la fois des l'opération de division et de racine carrée sont proposées dans [MCQ94], celles-ci peuvent également être utilisées pour augmenter les performances générales de cette architecture.

Les processeurs carrés sont chargés d'appliquer les rotations. Ils sont formés de trois multiplieurs et deux additionneurs. Mais les processeurs de la frontière nord ont aussi besoin d'un diviseur pour réaliser l'opération **g**/f. Les signaux de contrôle qui pilotent ces processeurs sont générés par une machine à états finis.

Le processeur comprend également un bloc de normalisation/dénormalisation qui convertit les données dans le plage de fonctionnement à savoir [-1, +1] en les divisant par le facteur de normalisation X_{norm}, avant de les introduire dans le réseau de processeurs. Après les calculs, il reconvertit les résultats dans leurs grandeurs normales en les multipliant par le facteur de normalisation. Le facteur de normalisation est une diadique (2^n), pour éviter

l'utilisation d'un diviseur et d'un multiplieur dans ce bloc. En effet, il faudra simplement décaler les bits à gauche pour la normalisation, et à droite pour la dénormalisation.

L'architecture comprend également un bloc de multiplexage qui est un bloc combinatoire et qui a pour rôle de prendre les données sortant du réseau de processeurs et de les renvoyer sur les entrées convenables du réseau. Ces données sont donc utilisées dans le cycle d'horloge suivant. Par conséquent, cette architecture ne nécessite aucun stockage de données. Le bloc de multiplexage est contrôlé par le signal *state* qui représente l'état du système. Ce signal est généré par le bloc de contrôle.

Le bloc de contrôle est une machine à états finis réalisée avec System Architect® de Mentor Graphics® . Elle représente le module d'interface du processeur avec le milieu extérieur. Sa tâche principale est de générer les signaux de contrôle nécessaires aux séquences de fonctionnement des PEs et le signal *state* qui définit le fonctionnement du bloc de multiplexage. Les signaux que l'utilisateur manipule pour commander le système entrent dans ce bloc de contrôle. Ces signaux sont les suivants :

clk : horloge système

ready : signal envoyé par le milieu extérieur pour signaler qu'une donnée demandée par *request,* au bus d'adresse *address,* est prête sur le bus de données.

reset pour remettre le système à zéro et recommencer une nouvelle séquence de filtrage

hold pour marquer une pause dans le processus de filtrage. Lorsque *hold* est activé (logique '1') pendant le processus de filtrage, le processeur se met en attente. Après la pause (*hold* remis à '0'), le processeur continue le filtrage.

Les signaux envoyés vers l'extérieur sont :

Request pour demander une donnée : $\tilde{y}(k)$, $\mathbf{h}(k)(1)$, $\mathbf{h}(k)(2)$ ou $\mathbf{h}(k)(3)$

Address la donnée demandée par *request* est spécifiée sur le mini-bus d'adresses *address*

address = '001' : demande de $\tilde{y}(k)$

address = '010' : demande de $\mathbf{h}(k)(1)$ –Premier élément de **h** à l'instant k

address = '011' : demande de $\mathbf{h}(k)(2)$ –Deuxième élément de **h** à l'instant k

address = '100' : demande de $\mathbf{h}(k)(3)$ –Troisième élément de **h** à l'instant k

DATA I/O est le port bidirectionnel d'entrée et de sortie des données

TestIn et testOUT sont les ports d'entrée et de sortie prévus pour l'entrée et la sortie du signal de test.

5.3.2 *Résultats de correction du Processeur* ***SRCKAL***

Nous avons effectué une simulation du modèle VHDL du processeur **SRCKAL** dans l'environnement Mentor Graphics® avec les paramètres suivants :

$\sigma_v^2 = 0.1$, $\sigma_w^2 = 10$ d'où SNR = 20dB et X_{norm} = 32. Le nombre de bits des variables a été fixée à 20 selon les résultats de l'étude des effets de quantification faite à la section 5.2.1. Les constantes seront donc réprésentées sur 16 bits.

Le signal original à filtrer qui est le même que celui de la Figure 2.1 est rappelé ici : (Figure 5.4)

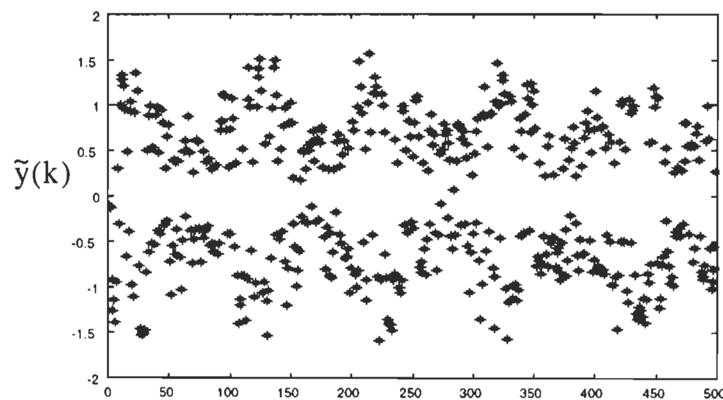


Figure 5.4 : Signal corrompu à la sortie du canal ; SNR = 20dB, BER =50%.

Les résultats d'égalisation du VHDL montrant les échantillons reconstitués à la sortie du processeur sont présentés à la Figure 5.5. La qualité de reconstitution est exactement ce qui avait été trouvé lors de la simulation, section 3.3.3 Figure 3.8, à savoir BER = 0. L'erreur quadratique de filtrage est aussi identique aux résultats de simulation, c'est-à-dire $\varepsilon(a, \hat{x}_{20}) = 18.7\%$.

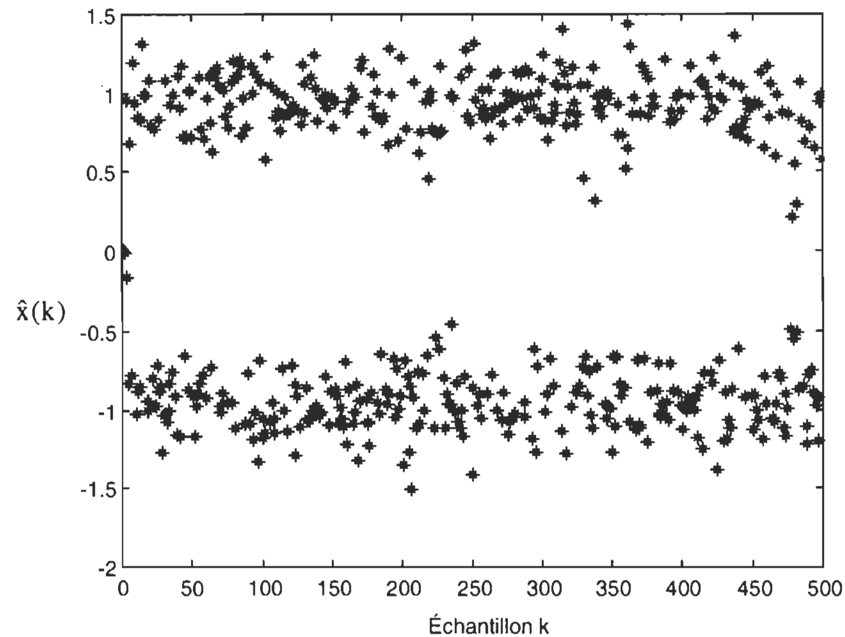


Figure 5.5 : Résultats de simulation du modèle VHDL 20 : BER = 0

5.4 Synthèse en technologie CMOS 0.5μm

Nous avons procédé à la synthèse dans la technologie 0.5 μm CMOS des processeurs élémentaires : un processeur rond contenant l'algorithme de division et de racine carrée, et un processeur carré contenant simplement trois multiplieurs et 2 additionneurs.

L'outil de synthèse utilisé est **Design Compiler©** de **Synopsys®** et la librairie cible est la librairie h-cell. Les performances du processeur complet sont évaluées en terme de surface et de vitesse. La vitesse globale du circuit sera dictée par la vitesse du processeur le plus lent du réseau.

La synthèse du processeur rond faite sans aucune contrainte a donné une surface de 69945 cellules. Notons que la cellule élémentaire est de la taille d'un inverseur de taille minimale dans cette technologie. Au total on aura environ 140 000 transistors dans un processeurs élémentaires rond. Le chemin critique est de 340.27ns, soit une fréquence de fonctionnement d'environ 3MHz.

Le processeur carré (ne contenant pas de diviseur) est évidemment moins gourmand que les processeurs ronds. Sa synthèse a donne une surface de 14 723 cellules élémentaires, soit environ 29 500 transistors. Le temps d'arrivée des données dans ce cas est de 25 ns, soit une fréquence de 40 MHz.

Le bloc de contrôle qui est une machine à état n'occupe que 425 cellules, qui est négligeable devant la taille des autres processeurs élémentaires. Le bloc de multiplexage a également une taille très négligeable par rapport aux processeurs élémentaires.

Une rapide estimation de la taille du processeur complet nous donne environs 350 000 cellules élémentaires, soit environ 700 000 transistors. La vitesse du processeur sera sensiblement de 3MHz. Ces paramètres pourront être modifiés en optimisant davantage les architectures. Par exemple, il faudrait optimiser les processeurs ronds pour la vitesse car ce sont eux qui représentent le goulot d'étranglement de l'architecture totale. Ensuite il faudra optimiser les autres processeurs pour la surface car leur vitesse sera imposée par celle des processeurs ronds. Ainsi, on réduirait au minimum la surface des processeurs carrés.

Chapitre 6

Conclusion

6.1 Synthèse des résultats

Dans ce mémoire, nous avons appliquée une approche innovatrice dans le domaine de l'implantation en technologie VLSI d'algorithmes parallèles décrites par des équations récurrentes par l'utilisation des outils MMAlpha. MMAlpha permet de dériver automatiquement des architectures systoliques. Le filtre de Kalman a été utilisé à titre de cas d'étude afin d'étudier MMAlpha pour la synthèse d'architectures parallèles.

Le filtre de Kalman a toujours fait et continue de faire l'objet de recherches dans le domaine du traitement des signaux et de son intégration en circuit VLSI. Nous voulons apporter notre contribution à cette recherche en proposant un circuit intégré implémentant l'architecture systolique du filtre de Kalman à covariance et à racine carrée de covariance. Dans ce travail, les algorithmes basés sur ce filtre ont été appliqués avec succès à la

résolution du problème de l'égalisation adaptative de canaux de communication numérique. En plus des domaines de la commande, du traitement numérique du signal et de la télécommunication que nous avons explorés tout au long de ce projet, les applications de ces architectures systoliques sont également possible dans des domaines variés comme la métrologie (la reconstitution de mesurande), l'ingénierie biomédicale, la séismologie et la spectrométrie.

L'outil MMAlpha pour résoudre le problème de programmation linéaire qui apparaît entre les variables a conduit par conséquent vers des architectures dans lesquelles le parallélisme est maximal. Ceci a permis d'atteindre des performances architecturales intéressantes et dans certain cas supérieures à ce qui est proposées dans la littérature. Deux architectures systoliques ont été dérivé par l'assistance des outils MMAlpha d'une part, pour le filtre de Kalman de covariance [MOZ98] et d'autre part, pour le filtre de Kalman à racine carré de covariance [MOZ99]. Dans les deux cas nous avons ciblé l'égalisation adaptative de canaux comme cas d'application.

L'architecture que nous avons proposée pour le filtre de Kalman à covariance réalise un taux d'utilisation des processeurs élémentaires de plus de 82%, avec 7M+6 cycles pour chaque itérations, sur un réseau carré de M^2+1 processeurs élémentaires. L'architecture pour le filtre de Kalman à racine carré de covariance fonctionne en 3M+5 cycles d'horloges et nécessite un réseau triangulaire de (M+1)(M+2)/2 processeurs élémentaires. De plus, nous avons démontré que ce dernier présente une plus grande robustesse à l'effet de

quantification et a été retenu comme architecture pour proposer un processeur dédié à l'égalisation de canaux.

Ce processeur a été complètement modélisé et simulé en VHDL. Il est donc fonctionnel sous n'importe quel simulateur de VHDL. Les deux types de processeurs élémentaires de cette architecture ont été synthétisés sur une technologie CMOS 0.5 µm à l'aide des outils logiciels de Synopsys. Les performances du processeur en terme de surface d'intégration et de vitesse de calcul ont alors été déduites. Par conséquent on a estimé sa taille à environ 800 000 transistors et sa vitesse de fonctionnement à 40 MHz. On pourra donc faire les prévisions de sa fabrication en fonction de la surface ainsi obtenue. L'avantage que présente le modèle du processeur que nous avons obtenu est qu'il est indépendant de la technologie de fabrication. Il sera donc possible de l'intégrer dans une autre technologie (ex: CMOS 0.25 µm ou 0.35 µm) lorsque le Laboratoire d'Algorithmes et d'Architectures Intégrés acquerra de nouvelles librairies de technologie.

Finalement, l'architecture proposée dans le cadre de cette étude sera d'ailleurs utilisée en commande pour l'estimation de paramètres pour la commande adaptative de joint flexible dans le cadre du projet de maîtrise de M. Sébastien Lesueur [LES99].

6.2 Recommandations et suite des travaux

Nous pensons qu'il serait profitable de synthétiser l'architecture du processeur développée pour le filtre de Kalman à racine carré de covariance dans la technologie 0.25 µm CMOS afin de déterminer les gains de surface et de vitesse que l'on pourra faire avant

de décider la technologie cible finale. Évidemment, les performances évaluées ci-dessus seront améliorées dans cette nouvelle technologie, mais il faudrait réexaminer les rapports coût de fabrication sur vitesse et surface et voir si cela vaut la peine de faire la fabrication finale en 0.5 µm, 0.35 µm ou 0.25 µm CMOS.

Par rapport à l'environnement MMAlpha, nous recommandons la poursuite très active de la collaboration entre le Laboratoire d'Algorithmes et Architectures Intégrées du Département de Génie Électrique à l'Université du Québec à Trois-Rivières et l'Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA de Rennes) sur la génération d'architectures parallèles des algorithmes de traitement de signaux.

L'environnement MMAlpha ne nous a pas seulement permis de développer des architectures pour le filtre de Kalman, mais il nous a également permis de développer une architecture pour l'égalisation de canaux basée sur la logique floue [ZAK99a], [ZAK99b]. Nous avons également entamé des études sur l'utilisation de MMAlpha pour la dérivation d'architectures systoliques basées sur des réseaux de neurones. Les résultats de ces travaux seront présentés dans les mémoires de maîtrise à venir [ZAK99c], [VID99]. Il est logique que ces travaux soient poursuivis en collaboration avec l'IRISA. En effet, l'environnement MMAlpha est très bien adapté pour la synthèse d'architecture systoliques à partir des équations récurrentes, et les algorithmes basées sur la logique floue et les réseaux de neurones en sont un bon exemple.

Toujours par rapport à l'environnement MMAlpha, nous recommandons que les deux parties (Laboratoire d'Algorithmes et Architectures Intégrées de l'UQTR et IRISA de

Rennes) explorent les moyens de faire un ordonnancement piloté par le concepteur du circuit. En effet, un ordonnancement dirigé donnerait la possibilité à l'utilisateur de spécifier l'instant de calcul de ses variables et aussi de spécifier les ressources sur lesquelles elles seront calculées. Ceci suppose que l'utilisateur a la possibilité de spécifier les ressources disponibles. Un exemple concret serait le suivant : lors de la synthèse logique du processeur carré de notre architecture, Synopsys® nous a généré automatiquement autant de multiplieurs qu'il y a de multiplications dans l'algorithme. Ceci mène certes vers une architecture rapide mais la surface est trop importante. En plus nous n'avons pas besoin de vitesse dans le processeur carré car notre vitesse ici est limitée par le processeur rond. On pourrait donc demander à MMAlpha de nous programmer les variables de telle sorte qu'elles soient toutes calculées sur un seul et unique multiplieur, les variables intermédiaires étant stockées dans des registres si nécessaires : on gagnerait ainsi en surface. Cet ordonnancement réalisé au niveau algorithmique serait très avantageux par rapport aux autres logiciels qui réalisent ceci mais au niveau synthèse logique. En effet, l'utilisateur sera moins contraint aux exigences au niveau silicium pour réaliser son architecture.

Finalement, suite à ce travail, nous pouvons confirmer que l'assistance des outils MMAlpha pour la synthèse d'architectures parallèles d'algorithmes basés sur des équations récurrentes présente un gain important dans le temps de conception.

Bibliographie

[ALB84] D. Alba et G. R. Meira, "Inverse optimal filtering method for the instrumental spreading correction in size chromatography", J. of Liquid Chromatography, vol. 7, n°14, 2833-2862, 1984.

[API97] Getting Started with Alpha, API-COSI, Internal report, IRISA, Rennes, Sept. 97. http://www.irisa.fr/api/ALPHA/welcome.html, June 1998.

[AZI91] M. R. Azimi-Sadjadi, T. Lu, et E. M. Nebot, "Parallel and Sequential Block Kalman Filtering and Their Implementation Using Systolic Arrays", IEEE - Transactions on Signal Processing, vol. 39, n° 1, Janvier 1991, pp. 137-147.

[BAK94] K. R Baker, A.D Brown et A.J. Currie, "Optimisation Efficiency in Behavioral Synthesis", IEE proc.-Circuits Devices Syst., Vol. 141, n° 5, Octobre 1994, pp. 399-406.

[BAL98] S. Balev, P. Quinton, S. Rajopadhye, et T. Risset, "Linear Programming Models for Scheduling Systems of Affine Recurrence Equations - a Comparative Study", SPAA98, Puerto Vallarta, Mexico, Juin 1998, pp 250-258.

[BRO95] D.W.Brown et F.M.F Gaston, "The systolic Design of a Block Regularised Parameter Estimator using Hierarchical Signal Flow Graphs", IEEE Int'l Conf. On Application-Specific Array Processors, 1995, pp.141-144.

[DON92] Vincent Van Dongen, "From systolic to Periodic Design." Algorithms and Parallel Architectures II , P. Quinton et Y, Robert, Elsevier Science Publishers B.V, 1992, pp.151-162.

[DUR92] Guy Durrieu, Kamel Kessaci et Michel Lemaitre, "Transe : An Experimental Design Tool." Algorithms and Parallel Architectures II, P. Quinton and Y, Robert, Elsevier Science Publishers B.V, 1992, pp.298-303.

[FAY95] Christian J.B. Fayoumi, Mohamad Sawan et Saad Bennis «Parallel VLSI Implementation of a new simplified aechitecture of Kalman filter », 1995 Canadian Conference on Computer and Electrical Engineering (CCECE'95).

[HAY96] S. Haykin, "Adaptive Filter Theory", Prentice Hall, 1996, Chap. 9.

[HEN92] J. Hennessy and D.A. Patterson, "Computer Architecture: A quantitative Approach", McGraw-Hill, 1992

[GAS88] FMF Gaston, G W IRWIN, "A systolic square root information Kalman Filter", IEEE International Conference on Systolic Arrays, 1988, pp .643-652

[GAS89] F. M. F. Gaston et G. W. Irwin, "Systolic approach to square root information Kalman filtering", International Journal of Control, vol. 50, no. 1, 1989, pp. 225-248.

[GAS89] F. Gaston et G. Irwin, "VLSI architectures for square root covariance Kalman filtering", Proc. SPIE, vol.1152, 1989, pp. 44-55.

[HEN92] J. Hennessy and D.A. Patterson, "Computer Architecture: A quantitative Approach", McGraw-Hill, 1992.

[IRI98] "A langage for synthesis of regular architectures", http://www.irisa.fr/api/ALPHA/welcome.html, June 1998, June 1998

[IRW91] G. W. Irwin, "Architectures for Control", Chap. 9 de Algorithms and parallel VLSI architectures, Elsevier Science, 1991, pp. 431-443.

[JOV86] J.M Jover and T. Kailath, "A parallel Architecture for Kalman Filter Measurement Upadate and Parameter Estimation", Automatica, 1986, vol.22, n°1, pp 43-57.

[KAM71] P. G. Kaminski, A. E. Bryson Jr., and S. F. Schmidt, "Discrete Square Root Filtering : A survey of Current Techniques", Reprinted from IEEE Trans. Automat. Contr., Dec. 1971, vol. AC-16, pp. 727-735.

[KIA86] S. Kiaei et U. B. Desai, "Independent Data Flow Wavefront Array Processors for Recursive Equations", VLSI signal processing II, IEEE press, NY, 1986, pp. 152-164.

[KUN82] H. T. Kung, "Why systolic architectures", IEEE Computer, Jan. 1982, Vol. 15, pp. 37-46.

[KUN91] S. Y. Kung and J. N. Hwang, "Systolic Array Designs for Kalman Filtering", IEEE - Transactions in Signal Processing, vol. 39, N° 1, Janvier 1991, pp. 171-182.

[LEV91] H. Le Verge, C. Mauras et P. Quinton, "The ALPHA language and its use for the design of systolic arrays", Journal of VLSI Signal Processing, Vol.3, 1991, pp. 173-182.

[LIN88] R. A. Lincoln et K. Yao, "Efficient Systolic Kalman Filtering Design by Dependence Graph Mapping", VLSI Signal Processing III, Edited by R. W. Brodersen et H.S. Moscovitz, IEEE Press, 1988, pp.396-407.

[MAD95] V.K. Madisetti, "VLSI Digital Signal Processors: An introduction to Rapid Prototyping and Design Synthesis", IEEE Press, 1995.

[MAS95] D. Massicotte, R. Z. Morawski, et A. Barwicz, "Incorporation of a Positivity Constraint Into a Kalman-Filter-Based Algorithm for Correction of

Spectrometric Data", IEEE Trans. Instr. and Meas., Vol. 44, No 1, February 1995, pp. 2-7.

[MAS98] D. Massicotte, "A Systolic VLSI Implementation of Kalman-Filter-Based Algorithms for Signal Reconstruction", IEEE Int. Conf. Acoustics, Speech, and Signal Processing, Seattle, 12-15 May 1998, pp. 3029-3032.

[MCQ94] S.E. McQuillan et J.V. McCanny, "Fast VLSI Algorithms for Division and square root", Journal of VLSI Signal Processing 8, 1994, pp.151-168, Kluver Academec Publishers, Boston.

[MEG91] G.M. Megson, "Fast Multi-layer Systolic Arrays for Kalman Filtering", Algorithms and Parallel VLSI Architectures, Vol.B: Proceedings, E.F. Depretter and A. Deprettere and A.-J. van der Veen (eds.), Elsevier Science Publishers B.V., 1991, Chap. 15, pp. 145-154.

[MOE96] P. Le Moenner et al., "Generating Regular Arithmetic Circuits with ALPHARD", MPCS'96, Ischia, Italy, 6-9 May 1996.

[MOO94] Marc Moonen, "Implememtation of a Square-root Information Kalman Filter on a Jacobi-Type Systolic Array", Journal of VLSI Signal Processing 8, 1994, pp.283-291 , Kluver Academec Publishers, Boston.

[MOZ97] Aurelien L. T. Mozipo, Étude de l'effet de quantification sur le comportement dynamique d'un système estimé par le filtre de Kalman, Rapport de projet dans

le cours GEI6026 : Théorie des systèmes asservis échantillonnés et non linéaires. Automne 97, Université du Québec à Trois-Rivières, Département de Génie Électrique.

[MOZ98] Aurelien. L. T. Mozipo, D. Massicotte, P. Quinton et T. Risset, "Automatic Synthesis of a Parallel Architecture for Kalman Filtering using MMAlpha", 1998' International Conference on Parallel Computing in Electrical Engineering (PARELEC'98), Bialystok, Poland, Sept. 2-5, 1998, pp. 201-206.

[MOZ99] Aurelien. L. T. Mozipo, Daniel Massicotte, Patrice Quinton et Tanguy Risset, "A Parallel Architecture for Adaptive Channel Equalization based on Kalman Filter using MMAlpha", 1999' IEEE Canadian Conference on Electrical and Computer Engineering, Edmonton, Alberta, Canada, 12-15 Mai 1999.

[MYE76] Kenneth A. myers et Byron D. Tapley, "Adaptive Sequential Estimator with Unknown noise Statistics", IEEE trans. On aut. Cont. , Août 1976, pp. 520-523

[PAI77] C.C. Paige et M.A. Saunders, "Least Square Estimation of Discrete Linear Dynamic Systems Using orthogonal Transformations", SIAM J. NUMER. ANAL., vol. 14, n°2, April 1977, pp180-193

[PAR91] Raffaele Parisi, Elio D. Di Claudio, Gianni Orlandi et Bhaskar D. Rao, " Fast Adaptive Digital Equalization by Recurrent Neural Networks ", IEEE Trans. On Signal Processing, Vol 45 n° 11, Nov. 1997, pp.2731-2739.

[QUI89a] P. Quinton et Y. Robert, Algorithmes et architectures systoliques, Masson 1989, Paris

[QUI89b] P. Quinton et V. Van Dongen, "The mapping of linear recurrence equations on regular arrays", Journal of VLSI Signal Processing, Vol. 1, No 2, October 1989, pp. 95-113.

[RAM95] G. Ramstein, O. Deforges et P. Bakowski, " A design Tool for Specification and simulation of Array Processors Architectures. Applications to image Processing : the extraction of regions of Interests.", IEEE Int. Conf. On Application-Specific Array Processors, 1995, pp.322-329.

[RAO91] P. Rao et M. Bayoumi, "An Algorithm Specific VLSI Parallel Architecture for Kalman Filter", IEEE Press: VLSI Signal Processing IV, 1991, pp. 264-273.

[SIC97] P. Sicard et J. Dubé, Théorie des systèmes asservis échantillonnés et non linéaires. Notes de cours GEI6026, Automne 97, Université du Québec à Trois-Rivières, Département de Génie Électrique.

[SOR85] H. W. Sorenson, Kalman Filtering: Theory and Application, IEEE Press, 1985.

[VID99] Martin Vidal, "Développement d'une architecture systolique pour l'égalisation de canaux de communication non linéaires basée sur les réseaux de neurones", Mémoire de Maîtrise en Génie Électrique, UQTR, Août 1999.

[WIL95] Doran Wilde et Sanjay Rajopadhye, "The naive Execution of Affine recurrence Equations", IEEE, Int'l Conf. On Application-Specific Array Processors, 1995, pp.1-12.

[WIS94] Wilde,D. K., Oumarou, Sié, "Regular Array synthesis Using Alpha", Publication interne INRIA, Mai 1994.

[YEH88] H. G. Yeh, "Systolic Implementation on Kalman Filters", IEEE Trans. on acoustics, speech, and signal processing, Vol.36, No 9, pp.1514-1517, 1988

[ZAK99a] Mourad Zakhama, Aurelien T. Mozipo et Daniel Massicotte, "Synthèse Automatique avec MMAlpha d'une Architecture Parallèle pour l'Égalisation de Canaux basée sur la Logique Floue", ACFAS, Ottawa, Canada, Mai 1999.

[ZAK99b] Mourad Zakhama et Daniel Massicotte, "A Systolic Architecture for Channel Equalization based on fuzzy Logic Algorithm", 1999' IEEE Canadian Conference on Electrical and Computer Engineering, Edmonton, Alberta, Canada, (Coming up in) May 1999.

[ZAK99c] Mourad Zakhama, "Implantation dans une technologie ITGE d'un filtre adaptatif basé sur la logique floue pour l'égalisation des canaux non linéaires.", Mémoire de Maîtrise en Génie Électrique, UQTR, Août 1999.

Annexes I

Articles publiés au cours de cette recherche

1998' INTERNATIONAL CONFERENCE ON PARALLEL COMPUTING IN ELECTRICAL ENGINEERING
(PARELEC98)
BIALYSTOK, POLAND, SEPTEMBER 2-5, 1998

Automatic Synthesis of a Parallel Architecture for Kalman Filtering using MMAlpha

Aurelien L. T. MOZIPO, Daniel MASSICOTTE, Patrice QUINTON*, and Tanguy RISSET*

Université du Québec à Trois-Rivières, Electrical Engineering Department
Research Group on Industrial Electronics
C.P. 500, Trois-Rivières, Québec, Canada, G9A 5H7, Tel.: +1-(819)-376-5071, Fax: +1-(819)-376-5219
E-mail: Aurelien_Landry_Mozipo_Tchoupou@uqtr.uquebec.ca, Daniel_Massicotte@uqtr.uquebec.ca
*INRIA Rennes, University of Rennes 1 and IRISA of Rennes
Parallel VLSI Architectures Team

*Campus de Beaulieu, 35042 Rennes Cedex, Rennes France, Tel.: +33-2.99.84.71.85, Fax: +33-2.99.84.71.00
E-mail: Patrice.Quinton@irisa.fr, Tanguy.Risset@irisa.fr

Abstract - **The intensive computations involved in the Kalman filtering are not feasible for many practical applications. Despite the great versatility of this method, this phenomenon largely reduces the use of Kalman filtering in real time applications and in all other fields where throughput is an important criterion. In this paper, we propose the use the MMAlpha tool as an innovative technique which gives automatically a parallel architecture of the covariance Kalman filter. Alpha is a functional language developed for the synthesis of regular architectures from recurrence equations. Many other Kalman-filter-based parallel architectures have been proposed, but, unlike ours, they are not derived automatically by means of a tool. In addition to having the advantage of being fast, this method leads to a systolic architecture of an array of M^2+1 elementary processors and a timesteps per iteration of 7M+6, where M is the dimension of the covariance matrix. A comparative study is done with other architectures proposed in the literature. As an application, we use Kalman filtering for signal reconstruction and specifically for adaptive channel equalization.**

I. INTRODUCTION

With the emergence of real time applications with intensive demanding computations, the design of dedicated processors has been considerably developed, either in the methodology or in means to accelerate the computation rate within the processor. Modern VLSI design techniques are numerous and all have the same objectives: to minimize the number of timesteps between updates and to maximize the number of operations carried out during one clock cycle. Early techniques developed aimed at executing operations as quickly as possible, but with the emergence of the parallel computing techniques in VLSI circuits like systolic arrays [15], a new approach consisting of performing many operations during the same cycle quickly spread among the integrated circuits designers. Consequently, several authors used this approach to develop parallel architectures based on information and covariance Kalman filter in various application fields (e.g. [2]-[4]). These architectures are derived by linear algebraic transformations like Faddeev algorithm and Givens rotations (e.g. [2], [3]), or manually by computing in advance the Kalman gains before feeding them into the process [16], [19]. Obviously, this last method does not integrate the resolution of the Riccatti equation, which however constitutes the biggest obstacle to the integration of the Kalman filter on a silicon chip. Many other Kalman filter based parallel architectures have been proposed (e.g. [5]-[10]).

We propose the application of a set of tools that give automatically a parallel architecture of a Kalman filter, based on the formalism of systems of affine recurrence equations [11]-[13]. The tools are integrated within the environment called MMAlpha and functional in Mathematica®. These advanced techniques for parallel computing in very large-scale integration circuits were proposed in [11-13], [18], [20], [21]. In the final architecture derived with these tools, parallelism is maximal and therefore, the time between two consecutive output samples is minimized.

In this work, the MMAlpha environment is applied to the Kalman filter to obtain a parallel architecture. Since the Kalman filter is a linear optimal estimator, it can also be used for state reconstruction of a time varying dynamic system. In this paper we apply the Kalman filter to solve the ill-posed problem in signal reconstruction and specifically for adaptive equalization [14].

In section II, we discuss the covariance Kalman filtering and its application to adaptive channel equalization. An extended summary of the MMAlpha environment is given in section III, and the application of this tool to the Kalman filter to obtain a systolic architecture is described in section IV. Section V presents the systolic architecture obtained. Finally, in section VI, we compare the performance of our design with some previously published architectures.

II. THE KALMAN FILTERING AND SIGNAL RECONSTRUCTION

The implementation in a VLSI technology of Kalman filter based algorithms for specific applications is required in various fields

This work was supported by Natural Sciences and Engineering Research Council of Canada, Fonds FCAR Québec, and Association of Universities and Colleges of Canada.

such as signal processing, communications, as well as control systems. This interest is explained by the fact that the Kalman filtering is a very powerful tool in real time estimation process [1]. This filter is based on the principle of estimating the state of a system based on noisy measurements in a stochastic environment, by minimizing the estimated mean squared error. Consider a discrete time-varying dynamic system defined by the following state equation:

$$\mathbf{x}_{k+1} = \mathbf{\Phi}_k \mathbf{x}_k + \mathbf{B}_k \mathbf{u}_k + \mathbf{w}_k, \quad \mathbf{x}_0 = 0 \tag{1}$$

$$\tilde{\mathbf{y}}_k = \mathbf{H}_k \mathbf{x}_k + \mathbf{v}_k \tag{2}$$

for k=1,2,... and where $\mathbf{x}_k$ is the (M×1) state vector, $\tilde{\mathbf{y}}_k$ is the (1xN) measurement vector and $\mathbf{u}(k)$ is the (P×1) control vector. $\mathbf{\Phi}_k$, $\mathbf{B}_k$ and $\mathbf{H}_k$ are known matrices. Also, $\mathbf{v}$ and $\mathbf{w}$ are two sequences of non-correlated white noise with known covariance matrix $\mathbf{R}_k^v$ and $\mathbf{R}_k^w$ respectively. The conventional Kalman filtering equations can be summarized in the following covariance form [2]:

$$\hat{\mathbf{x}}_{k+1/k} = \mathbf{\Phi}_k \hat{\mathbf{x}}_{k/k} + \mathbf{B}_k \mathbf{u}_k \tag{3}$$

$$\mathbf{P}_{k+1/k} = \mathbf{\Phi}_k \mathbf{P}_{k/k} \mathbf{\Phi}_k^T + \mathbf{R}_k^w, \quad \mathbf{P}_{0/0} = \mathbf{I} \tag{4}$$

$$\mathbf{V}_{k+1} = \mathbf{H}_k \mathbf{P}_{k+1/k} \mathbf{H}_k^T + \mathbf{R}_k^v \tag{5}$$

$$\mathbf{K}_{k+1} = \mathbf{P}_{k+1/k} \mathbf{H}_{k+1}^T \mathbf{V}_{k+1}^{-1} \tag{6}$$

$$\hat{\mathbf{x}}_{k+1/k+1} = \hat{\mathbf{x}}_{k+1/k} + \mathbf{K}_{k+1}\left[\tilde{\mathbf{y}}_{k+1} - \mathbf{H}_{k+1}\hat{\mathbf{x}}_{k+1/k}\right] \tag{7}$$

$$\mathbf{P}_{k+1/k+1} = \mathbf{P}_{k+1/k} - \mathbf{K}_{k+1}\mathbf{H}_{k+1}\mathbf{P}_{k+1/k} \tag{8}$$

where $\hat{\mathbf{x}}_{k+1/k}$ is the prediction of state $\mathbf{x}$ at time k+1 given measurements and information up to and including time k; $\hat{\mathbf{x}}_{k+1/k+1}$ is the estimation of state $\mathbf{x}$ at time k+1 given measurements and information up to and including time k+1; $\mathbf{P}_{k+1/k+1}$ is the covariance of estimation error; $\mathbf{P}_{k+1/k}$ is the covariance of prediction error; $\mathbf{K}_{k+1}$ is the Kalman gain.

We apply the Kalman filter to adaptive channel equalization problem. This problem is defined by the convolution operation of a signal a_k crossing a channel (e.g. conversion system in a measurement system, modem, satellite communication, cellular phone), represented by a impulse response function vector $\mathbf{h}_k$, where the output $\tilde{y}_k$ is a scalar corrupted with additive noise v_k. So, the dimensions of the matrices and vectors in the equations (1)-(8) are N=1 and P=1. The discrete form of the convolution equation is:

$$\tilde{y}_k = \sum_{m=1}^{M} h_{k-m} a_m + v_k \quad \text{for } k = 1,2,3,... \tag{9}$$

The signal which is applied to the channel, defined as the measurand signal, can be estimated by numerical methods on the basis of an a priori knowledge of the impulse response of the conversion system and the measured samples of the output. This operation is a numerically ill-conditioned rule.

The solutions given by LMS and RLS algorithms have been proposed in [14] and by Kalman filter in [17], [22]. Here, we used the model gives by the equations (1) and (2) without the control input $\mathbf{u}_k$ and we considered the matrix $\mathbf{\Phi}$ as invariant. By normalizing the covariance matrices, we obtain the following form for equations (3)-(8) respectively [22]:

$$\hat{\mathbf{x}}_{k+1/k} = \mathbf{\Phi}\hat{\mathbf{x}}_{k/k} \tag{10}$$

$$\mathbf{P}_{k+1/k} = \mathbf{\Phi}\mathbf{P}_{k/k}\mathbf{\Phi}^T + \mathbf{b}\beta\mathbf{b}^T, \quad \mathbf{P}_{0/0} = \mathbf{I} \tag{11}$$

$$\mathbf{V}_{k+1} = \mathbf{h}_k \mathbf{P}_{k+1/k} \mathbf{h}_k^T + 1 \tag{12}$$

$$\mathbf{K}_{k+1} = \mathbf{P}_{k+1/k} \mathbf{h}_{k+1}^T \mathbf{V}_{k+1}^{-1} \tag{13}$$

$$\hat{\mathbf{x}}_{k+1/k+1} = \hat{\mathbf{x}}_{k+1/k} + \mathbf{K}_{k+1}\left[\tilde{\mathbf{y}}_{k+1} - \mathbf{h}_{k+1}\hat{\mathbf{x}}_{k+1/k}\right] \tag{14}$$

$$\mathbf{P}_{k+1/k+1} = \mathbf{P}_{k+1/k} - \mathbf{K}_{k+1} h_{k+1} \mathbf{P}_{k+1/k} \tag{15}$$

Moreover, the measurement noise is taken stronger than the value in [14] to appreciate the accuracy of data correction with the Kalman Filter. Our experiment has shown that the Kalman filter gives better results than the LMS and RLS algorithms.

The next step of the work is to translate equations (10)-(15) in the Alpha language and then to derive a parallel architecture with MMAlpha tool.

III. SUMMARY OF THE MMALPHA ENVIRONMENT

The tool we use to derive these parallel architectures is the Alpha language and its development environment called MMAlpha. Alpha is a functional language developed for the synthesis of regular architectures from recurrence equations [12]. In Alpha, an algorithm is described as a set of equations on variables defined on multi-dimensional domains. Each variable or expression of the language is actually a function from a set of integer coordinate points satisfying linear inequalities, to a set of values. The synthesis process consists of applying a sequence of semantic preserving transformations which map the initial specification of the algorithm to an architecture which supports its execution. The final description can be translated into VHDL in order to generate a VLSI architecture.

All tools needed to perform these transformations are implemented in the MMAlpha environment as a set of Mathematica® packages together with C libraries.

The design process starts from an algorithmic level description of the application which can be readily obtained from the equations of the process such as those of Part II. This description can be organized as a hierarchy where basic linear algebra algorithms such as matrix vector multiplication, matrix multiplication, etc. are first described and then used in the application. A C program which evaluates this description can be automatically generated in order to check the correctness of the initial specification by simulation. Then the initial description undergoes a series of transformations which deliver an abstract architecture. Among these transformations, localization and scheduling are the most important ones. Localization (also called uniformization or

pipelining in the literature) replaces non-local calculations by local ones. Scheduling orders the calculations in such a way that the evaluation of a given variable can be performed after that of its components. Scheduling amounts to solving an integer linear programming problem whose unknowns are the coefficients of the affine function which defines the time at which each variable may be evaluated [20]. For example, a variable V[i,j] is scheduled at time ai+bj+c, and the coefficients a, b, c are unknowns. Scheduling also provides interesting information on the total time needed to execute the algorithm. Once a schedule has been found, a change of basis allows all calculations to be expressed in a new index space giving the evaluation time and the processor number where a calculation is to be performed.

From this abstract architecture, the design of an actual architecture consists of applying a sequence of low-level transformations which brings the description to a net-list format called AlpHard. The transformation process is almost automated, the MMAlpha environment behaving as a compiler which automatically maps one description level to the next one. Eventually, one obtains a VHDL model to implement in a VLSI technology (e.g. FPGA, CMOS) [13].

IV. TRANSFORMATION PROCESS

The design methodology within the MMAlpha environment explained above is applied to equations (10) to (15) to obtain a scheduling for all output and internal variables and a netlist of a processors array which defines the architecture. Equations (10) to (15) are rewritten into the Alpha language and are analyze to remove syntax error and check the domains of the equations by static analysis. Then we inline all subsystems [18]: this transformation flattens all structured Alpha expressions such that the resulting program can be translated in C with the *WriteC* translator. The C program obtained is executed and results are compared to those obtained with Matlab®.

After these preliminary transformations, the effective architecture derivation can begin. First, we pipeline some broadcasted variables along given directions. Computations involved in the pipeline process are matrix-matrix, matrix-vector and vector-vector multiplications. After the pipelining, we can look for a scheduling for the Alpha program. The goal of the *scheduler* is to find a valid execution order with respect to a particular criterion. The time is considered as a discrete single clock. The overall idea of the scheduling process is to build a linear programming problem and solve it with a software tool [18].

The scheduling for the computation of all variables of the Alpha program is summarized in Table I. Table I shows the order in which each variable is computed, the total time needed to compute it, and the number of timesteps. The number of timesteps is defined both without pipelined and with pipelined on the scheduling. All these information are provided automatically by the scheduler of MMAlpha. This schedule is made explicit by applying a time space reindexing to the Alpha program. Then the control signals are generated and we derive the AlpHard model of the architecture.

V. ARCHITECTURE

We show here the architecture that one obtains after applying the transformations of MMAlpha. This architecture is made of a

Table I : Timesteps derivation from the scheduling given by MMAlpha.

Eq.	Variable	Operation	Scheduling time given by MMAlpha i=1,2,...,M m=1,2,...,M	Number of timesteps	
				without pipeline	with pipeline
(10)	$\hat{\mathbf{x}}^{int}$	$\Phi\hat{\mathbf{x}}_{k/k}$	1+m+M	2M-1	M
(14)	$\hat{\mathbf{y}}$	$\hat{\mathbf{x}}^{int}\mathbf{h}_{k+1}$	7+2M	M	M
	δ_y	$\tilde{\mathbf{y}}_{k+1}-\hat{\mathbf{y}}$	8+2M	0	
(11)	$\mathbf{A}_1$	$\mathbf{P}_{k/k}\Phi^T$	2+i+m+M	3M-1	
	$\mathbf{P}_{k+1/k}$	$\Phi\mathbf{A}_1+\mathbf{b}\beta\mathbf{b}^T$	1+i+m+2M	3M-1	M

M×M processing elements (PE) square array; each of which is a multiplier-accumulator (MAC), with a control input which defines the operating mode, i.e. it selects appropriate input data to feed into the arithmetic unit, and the data to be output. One divider is needed to compute 1/(Vint+1). Therefore, the overall architecture has M2+1 PEs arranged as shown later in this section. One advantage presented by this architecture is that many intermediate variables and data are used immediately after their computations to pipeline the scheduling. Hence, they stay in the array for the next timestep. However, we need to store the constant data $\mathbf{b}\beta\mathbf{b}^T$ and Φ, and some variables $\hat{\mathbf{x}}_k$, $\mathbf{h}_k$, $\mathbf{h}_{k+1}$, $\mathbf{P}_k$, $\mathbf{P}_{k/k+1}$, $\mathbf{V}_{k+1}^{-1}$ and δ_y. Figure 3 shows two storage units; each can be constituted of M memories. $\mathbf{P}_k$ and $\mathbf{P}_{k/k+1}$ matrices are symmetric, we need to store only their lower (or upper) triangular part respectively.

We explain how the architecture works by showing in figure 1, as an example, the computation of equation (11). In this figure, the control signals and other I/O pads are omitted for the sake of simplicity. Here, each processing cell uses two operating modes when computing equation (11). The first operation is $\mathbf{P}_{k/k}\Phi^T=\mathbf{A}_1$ in mode 1, the result $\mathbf{A}_1$ is stored in the array,

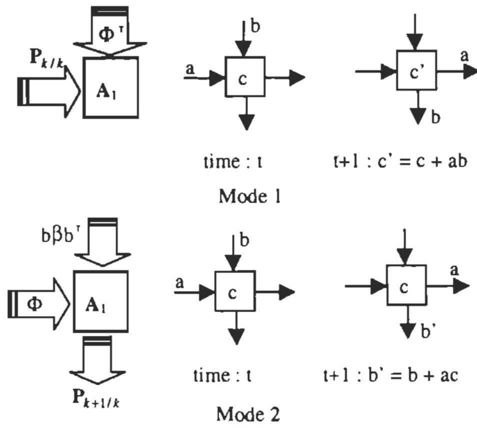


Figure 1: Data flow and MAC cell for operating mode 1 and 2.

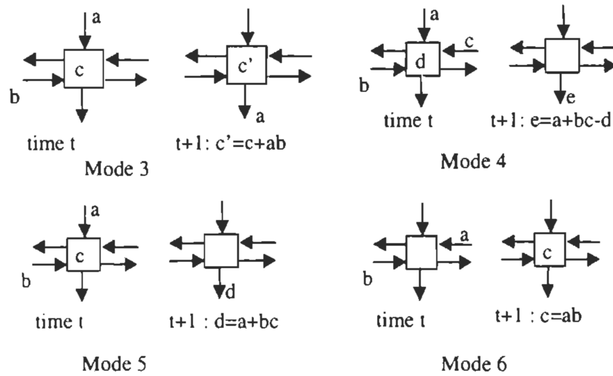


Figure 2: Operating modes

then ΦA_1 is computed in mode 2 and the result is added to $b\beta b^T$. This computation involves two operating modes for all the M2 PEs of the array. Each PE has several operating modes determined by variables scheduled to be computed within it, and also according to the direction (left, right, top and bottom) throw which these variables are pipelined into the PE.

The overall systolic architecture is given in figure 3 for M=3. The operation of the array processors is realized in 10 steps as follows:

Step 1: Φ and $\hat{x}_{k/k}$ are fetched into the array, $\tilde{y}_k$ is loaded to PE(M,M), $\hat{x}^{int}$ is then computed in the first column, with the PEs operating in mode 3. After M timesteps $\hat{x}_{k/k}(1)$ is available in PE(1,1) and it is stored in the internal register for the next step; then it moves step by step toward the last PE of the row, PE(1,M). And so do all other elements of the vector immediately after they are computed.

Step 2: $\hat{x}^{int}$ computed in step 1 circulates in the array and meets h_{k+1} in the last column. $\hat{x}^{int}$ and h_{k+1} are multiplied element by

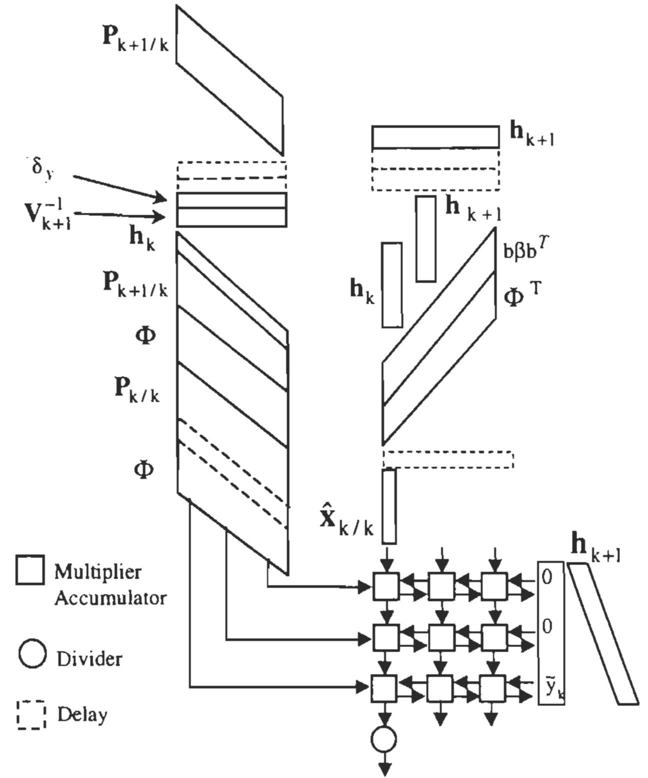


Figure 3: Overall architecture and data arrangements for one update.

element. The scalar δ_y is obtained by successive downward accumulations of the obtained elements, as shown in figure 2 in mode 4. Note that d= $\tilde{y}_k$ for PE(M,M) and d=0 in PE(i,M) for all i=1,..,M-1, at the beginning. A unit delay is observed before the next data input, to allow the multiplication of $\hat{x}^{int}$ by h_{k+1} to take place. Then $P_{k/k}$ and Φ^T are fetched into the array. A_1 is computed in all the PEs of the array, with operating mode 1; the result stays in the array.

Step3: Φ and $b\beta b^T$ are fetched into the array. $P_{k+1/k}$ is computed with operating mode 2 shown above. The results leave the array by the bottom.

Step 4: $P_{k+1/k}$ is fetched in from the left, h_k and h_{k+1} are fetched in row 1 and 2 respectively, with a unit delay between them. A2 and A3 are computed by PEs of the first two rows operating in mode 1. Results are stored in corresponding rows of the array.

Step 5: h_k is fetched in from the left of the array after $P_{k+1/k}$. It is multiplied element by element in the first row, then accumulated downward to give V^{int} (operating mode 5).

Table II: Comparison to other covariance Kalman filter based designs

Architecture	Number of PE	Number of Timesteps per iteration	PE Utilization	
			M=3	M>>1

Step 6: The result of step 5 is sent to the divider to compute $\mathbf{V}_{k+1}^{-1}$.

Step 7: $\mathbf{V}_{k+1}^{-1}$ is then fetched from the left in all rows of the first column. The content of the second column is fed back to the first column for an element by element multiplication (operating mode 6). The result, $\mathbf{K}_{k+1}$, is stored in the array.

Step 8: δ_y is fetched from the left to all the rows of the first column. $\hat{\mathbf{x}}^{int}$ is retrieved from the internal register and $\hat{\mathbf{x}}_{k+1/k+1}$ is computed. The result $\hat{\mathbf{x}}_{k+1/k+1}$ is output and $\mathbf{K}_{k+1}$ is stored in the internal register of the first column.

Step 9: $\mathbf{K}_{k+1}$ is retrieved from the internal register, h is fetched in from the top of the array. $\mathbf{K}_{k+1}$ circulates from left to right and meets $\mathbf{h}_{k+1}$ which circulates from top to bottom for element by element multiplication. The obtained matrix is subtracted from identity matrix I. The result P^{int} is stored in the array.

Step 10: $P_{k+1/k}$ is fetched in from the left, and $P_{k+1/k+1}$ is computed. The result moves out by the bottom. All PEs operate in mode 2.

VI. COMPARISON OF PERFORMANCES

Table II shows a comparative study with some architectures based on the Kalman filter defined by the Eq. (10)-(15) such as the trapezoidal array designed by Irwin [2] and both architectures scheme proposed by Yeh [3]. The comparison is based on number of PEs, the computation time defined by the number of cycles to execute the computation applied to one sample $\tilde{y}_k$, and the PE utilization. The comparison was based on the assumption that the clock frequency is the same for all architectures, that the complexity of PEs is equivalent, and that all data needed to execute all operations are available.

The PE utilization introduced here is computed as the ratio between the average number of active processors and the number of processors. The average number of active processors is defined as the total number of arithmetic operations divided by the number of timesteps per iteration or per sample k [8]:

$$\text{PE Utilization} = \frac{\text{nb of arithmetic operations}}{\text{nb timestep} \times \text{nb PE}} \qquad (16)$$

From the Table I we obtain $6M^3+5M^2+5M+1$ arithmetic operations. Table II shows that our architecture presents the smallest computation time, the smallest number of PEs, and the highest processor utilization.

VII. CONCLUSION

In this paper we have applied the MMAlpha environment to obtain automatically a systolic architecture for the covariance Kalman filter. This architecture achieves a processor utilization of more than 82% with a speed of 7M+6 timesteps per iteration. These performances are due to the fact that we have used a particular software tool, the MMAlpha environment, to solve the linear programming problem that appear among variables. This leads us therefore to an architecture in which the parallelism is maximal. The algorithm used has been applied successfully to solve the adaptive channel equalization problem. Applications of these systolic architectures are possible in a wide variety of fields such as control, telecommunications, metrology, biomedical engineering, seismology and spectrometry. The next step of this work will be to implement the proposed architecture in a VLSI technology (0.5µm CMOS or FPGA).

REFERENCES

[1] H. W. Sorenson, "Kalman Filtering: Theory and Application", IEEE Press, 1985.

[2] G. W. Irwin, "Architectures for Control", Chap. 9 of Algorithms and parallel VLSI architectures, Elsevier Science, 1991, pp. 431-443.

[3] H. G. Yeh, "Systolic Implementation on Kalman Filters", IEEE Trans. on acoustics, speech, and signal processing, Vol.36, No 9, pp.1514-1517, 1988.

[4] R. A. Lincoln and K. Yao, "Efficient Systolic Kalman Filtering Design by Dependence Graph Mapping", VLSI Signal Processing III, Edited by R. W. Brodersen et H.S. Moscovitz, IEEE Press, 1988, pp.396-407.

[5] F. M. F. Gaston and G. W. Irwin, "Systolic approach to square root information Kalman filtering", International Journal of Control, vol. 50, no. 1, 1989, pp. 225-248.

[6] F. Gaston and G. Irwin, "VLSI architectures for square root covariance Kalman filtering", Proc. SPIE, vol.1152, 1989, pp. 44-55.

[7] M. R. Azimi-Sadjadi, T. Lu, and E. M. Nebot, "Parallel and Sequential Block Kalman Filtering and Their Implementation Using Systolic Arrays", IEEE - Transactions on Signal Processing, vol. 39, N° 1, January 1991, pp. 137-147.

[8] S. Y. Kung and J. N. Hwang, "Systolic Array Designs for Kalman Filtering", IEEE - Transactions in Signal Processing, vol. 39, N° 1, January 1991, pp. 171-182.

[9] G.M. Megson, "Fast Multi-layer Systolic Arrays for Kalman Filtering", Algorithms and Parallel VLSI Architectures, Vol.B: Proceedings, E.F. Depretter and A. Deprettere and A.-J. van der Veen (eds.), Elsevier Science Publishers B.V., 1991, Chap. 15, pp. 145-154.

[10] P. Rao and M. Bayoumi, "An Algorithm Specific VLSI Parallel Architecture for Kalman Filter", IEEE Press: VLSI Signal Processing, IV, 1991, pp. 264-273.

[11] P. Quinton and V. Van Dongen, "The mapping of linear recurrence equations on regular arrays", Journal of VLSI Signal Processing, Vol. 1, No 2, October 1989, pp. 95-113.

[12] H. Le Verge, C. Mauras, and P. Quinton, "The ALPHA language and its use for the design of systolic arrays", Journal of VLSI Signal Processing, Vol.3, 1991, pp. 173-182.

[13] P. Le Moenner et al., "Generating Regular Arithmetic Circuits with ALPHARD", MPCS'96, Ischia, Italy, 6-9 May 1996.

[14] S. Haykin, "Adaptive Filter Theory", Prentice Hall, 1996, Chap. 9.

[15] H. T. Kung, "Why systolic architectures", IEEE Computer, Vol. 15, pp. 37-46, Jan. 1982

[16] D. Massicotte, "A Systolic VLSI Implementation of Kalman-Filter-Based Algorithms for Signal Reconstruction", IEEE Int. Conf. Acoustics, Speech, and Signal Processing, Seattle, 12-15 May 1998, pp. 3029-3032.

[17] D. Alba, G. R. Meira, "inverse optimal filtering method for the instrumental spreading correction in size chromatography", J. of Liquid Chromatography, 7(14), 2833-2862 (1984), by Marcel dekker, Inc.

[18] Getting Started with Alpha, API-COSI, Internal report, IRISA, Rennes, Sept. 97.

[19] S. Kiaei, U. B. Desai, "Independent Data Flow Wavefront Array Processors for Recursive Equations", in proc. VLSI signal processing II, IEEE press, NY, 1986, pp. 152-164.

[20] S. Balev, P. Quinton, S. Rajopadhye, and T. Risset, "Linear Programming Models for Scheduling Systems of Affine Recurrence Equations - a Comparative Study", SPAA?98, Puerto Vallarta, Mexico, pp 250-258, June 1998.

[21] "A langage for synthesis of regular architectures", http://www.irisa.fr/api/ALPHA/welcome.html, June 1998.

[22] D. Massicotte, R. Z. Morawski, and A. Barwicz, "Incorporation of a Positivity Constraint Into a Kalman-Filter-Based Algorithm for Correction of Spectrometric Data", IEEE Trans. Instr. and Meas., Vol. 44, No 1, February 1995, pp. 2-7.

APPENDIX

Alpha program describing one step of Kalman filtering

The following is the Alpha program for one step of the Kalman filter. This program contains the definition of four subsystems. System OneStep is the main program, and corresponds, up to the renaming of some variables, to the equations given in Table I. This system calls three other subsystems, called matmult (for matrix multiplication), matvect (matrix vector multiplication) and dotprod (for dot product). Each subsystem contains a list of input variables, a list of output variables (preceded by the keyword returns), a list of local variables, and a list of equations. In system OneStep, most of the equation are so-called use statements, and behave (roughly speaking) much as subroutine calls in a conventional language. Notice that all systems are parameterized by the size parameter M.

```
-- OneStep of Kalman
system OneStep :{M | 2<=M}
        (yb : real;
         Ip : {m,i | 1<=m<=M; 1<=i<=M} of real;
         bbt : {m,i | 1<=m<=M; 1<=i<=M} of real;
         phi : {m,i | 1<=m<=M; 1<=i<=M} of real;
         phit : {m,i | 1<=m<=M; 1<=i<=M} of real;
         H : {m | 1<=m<=M} of real;
         Hz : {m | 1<=m<=M} of real;
         xchapz : {m | 1<=m<=M} of real;
 P : {m,i | 1<=m<=M; 1<=i<=M} of real)
    returns (K : {m | 1<=m<=M} of real;
         xchap : {m | 1<=m<=M} of real;
         Pkk : {m,i | 1<=m<=M; 1<=i<=M} of real);
var
 xint : {m | 1<=m<=M} of real;
 Ychap : real;
 I : real;
 Veint : real;
 invVe : real;
 V2 : {m | 1<=m<=M} of real;
 V3 : {m | 1<=m<=M} of real;
 PP, PP1, V1 : {m,i | 1<=m<=M; 1<=i<=M} of real;
 Pint : {m,i | 1<=m<=M; 1<=i<=M} of real;

let
 -- Equation 9
 use matvect[M] (phi, xchapz) returns (xint) ;
 -- Equation 7
 use dotprod[M] ( xint, H) returns (Ychap) ;
 I[] = yb[] - Ychap[];
 -- Equation 10
 use matmult[M] (P, phit) returns (V1);
 use matmult[M] (phi, V1) returns (PP1);
 PP = PP1 + bbt;
 -- Equation 11
 use matvect[M] (PP, H) returns (V2) ;
 use dotprod[M] (Hz, V2) returns (Veint) ;
 -- Equation 6
 use matvect[M] (PP, Hz) returns (V3) ;
 invVe[] = 1 / (Veint[]+1[]);
 K[m] = V3[m] * invVe[];
 -- Equation 7
 xchap[m] = xint[m] + K[m] * I[];
 -- Equation 8
 Pint[m,i] = Ip[m,i] - K[m] * H[i];
 use matmult[M] (Pint, PP) returns (Pkk) ;
tel;
```

```
-- Matrix matrix multiplication
-- Inputs: a, b: square matrices of size M
-- Outputs: c: square matrix of size M
system matmult : {M |M>1}
 (a,b : {i,j | 1<=i,j<=M} of real)
returns
 (c : {i,j | 1<=i,j<=M } of real);
var
 C : {i,j,k | 1<=i,j<=M; 0<=k<=M} of real;
let
 c[i,j] = C[i,j,M];
 C[i,j,k] = case
     {|k=0} : 0[];
     {|1<=k<=M} : C[i,j,k-1]+a[i,k]*b[k,j];
 esac;
tel;

-- Matrix vector multiplication
-- Input: a: a square matrix of size M
--        v: a vector of size M
-- Output: c: a vector of size M
system matvect : {M | M>1}
        (a : {i,j | 1<=i,j<=M} of real;
         v : {i | 1<=i<=M} of real)
    returns (c : {i | 1<=i<=M} of real);
var
    C : {i,j | 1<=i<=M; 0<=j<=M} of real;
let
    C[i,j] = case
```

```
            {| j=0} : 0[];
            {| j>=1} : C[i,j-1] + a[i,j]*v[j];
            esac;
      c[i] = C[i,M];
tel;

-- Dot product
-- Input: v, w: two M vectors
-- Output: s: a scalar
system dotprod : {M | M>1}
            ( v, w : {i | 1<=i<=M} of real)
      returns (s : real);
var
      S : {i | 0<=i<=M} of real;
let
      S[i] = case
            {| i=0} : 0[];
            {| i>=1} : S[ i-1 ] + v[i]*w[i];
            esac;
      s[] = S[M];
tel;
```

A Parallel Architecture for Adaptive Channel Equalization Based on Kalman Filter Using MMAlpha

Aurelien L. T. Mozipo, Daniel Massicotte, Patrice Quinton and Tanguy Risset**

Electrical Engineering Department
Université du Québec à Trois-Rivières
Research Group on Industrial Electronics
C.P. 500, Trois-Rivières, Québec, Canada, G9A 5H7
Tel.: 1-(819)-376-5071, Fax : 1-(819)-376-5219
E-mail: {mozipotc, Daniel_Massicotte}@uqtr.uquebec.ca

*INRIA Rennes, University of Rennes 1
IRISA of Rennes
Parallel VLSI Architectures Team
Campus de Beaulieu, 35042 Rennes Cedex, Rennes France,
Tel.: +33-2.99.84.71.85, Fax: +33-2.99.84.71.00
E-mail: {Patrice.Quinton, Tanguy.Risset}@irisa.fr

ABSTRACT

In this paper we apply the square-root covariance Kalman filter to solve the ill-posed problem of signal reconstruction specifically for adaptive channel equalization. The computation latency and the throughput of this algorithm are significantly improved with the derivation of a systolic architecture using MMAlpha which is a tool dedicated to automatic synthesis of systolic architectures. The proposed architecture is validated by a VHDL simulation in the case of a time varying channel impulse response. The performance evaluation is based on a 20-bits wordlength and is synthesized in a 0.5 μm CMOS technology.

1. INTRODUCTION

The fields of communication, signal processing and control systems have this in common that they all need powerful reconstruction algorithms to regenerate an unknown signal corrupted either by the medium in which it is propagated, or by the conversion equipment, or by instruments used to sense them. The signal received through this medium is corrupted according to a deterministic law and distorted according to a stochastic law. Particular examples of can be the satellite communications or cellular telephone communications where the transmission medium here is the atmosphere, a transmission cable in wired communications (for example a LAN) or a modem in a communication by modem [18], [19]. Before using these signals, it is necessary to use reconstitution algorithms to extract the noise free real transmitted signal.

The fast growing of information technologies requires more and more high-speed devices with very high throughput. Therefore, we need to develop extremely fast reconstruction algorithms to satisfy the requirements of the circuits users. These applications are in general very demanding in term of speed and require consequently that one maximizes the parallelism in architectures used.

A popular estimation algorithm used in adaptive filtering is the Kalman filter, which is a powerful tool in real time estimation process [11]. This filter is based on the principle of estimating the state of a system based on noisy measurements in a stochastic environment, by minimizing the estimation mean squared error [1] and [2]. The intensive computations involved in this filter make inefficient to implement it sequentially in a VLSI technology. But by taking into account its regularity, we can map it onto a systolic architecture to increase the throughput. Several systolic architectures for the Kalman filtering have been done manually by means of linear algebraic transformations [4]-[6]. Here, we use a particular software, called MMAlpha, to perform the same task.

Alpha is a functional language developed in an environment MMAlpha and functional in Mathematica® [8], [13]. It is based on the formalism of systems of affine recurrence equations [9]. Alpha is a functional language for the expression and the synthesis of regular architectures. MMAlpha has been used in [1] to derive a systolic architecture for the covariance Kalman filter.

In section 2, we describe the channel equalization problem. The square root covariance Kalman filter (SRCKF) and its application to adaptive channel equalization are given in Section 3. The application of the MMAlpha tool to derive a systolic architecture to the Kalman filter is described in Section 4. Section 5 presents the simulation results and a performance evaluation of the architecture derived with MMAlpha. The conclusion is given in Section 6.

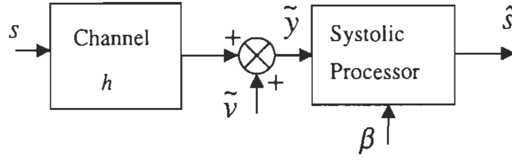
Throughout the paper, the notation of vectors and matrices is printed in bold type for clarity, and the $v_{i/j}(m)$ is the mth element of the vector $\mathbf{v}$ at time i given the data available at time j.

2. THE CHANNEL EQUALIZATION PROBLEM

This problem is defined by the convolution operation of a signal (or symbol) s_k crossing a channel (e.g. conversion system in a measurement system, modem, satellite communication, cellular phone), represented by a non invariant impulse response function

This work was supported by Natural Sciences and Engineering Research Council of Canada and Le Fonds FCAR Québec.

vector h_k, where the output $\tilde{y}$ is a scalar corrupted with



additive white noise $\tilde{v}$, Figure 1 [18]. The discrete form of the convolution equation is

$$\tilde{y}_k = \sum_{m=1}^{M} h_{k-m} s_m + \tilde{v}_k \quad \text{for } k=1,2,3,... \tag{1}$$

The signal which is applied to the channel, defined as the measurand signal, can be estimated by numerical methods on the basis of an a priori knowledge of the impulse response of the conversion system and the measured samples of the output. This operation is a numerically ill-conditioned rule.

The solutions given by LMS and RLS algorithms have been proposed in [11], by Kalman filter in [1], [19], and by neural networks for nonlinear channel in [10], [14]. We propose the resolution of this problem for a non invariant channel by the SRCKF to implement in fixe point VLSI processor.

3. SQUARE ROOT COVARIANCE KALMAN FILTERING TO ADAPTIVE CHANNEL EQUALIZATION

In [1], we proposed to use the MMAlpha tool as an innovative technique which gives automatically a parallel architecture of the covariance Kalman filter. However, this version of Kalman filter is affected by the quantification effects. In this paper, we propose an architecture more robust to the quantification effects and based on the SRCKF equations by using the method proposed in [1].

Considering a discrete time-varying dynamic system defined by its following state equation

$$\boldsymbol{x}_{k+1} = \boldsymbol{\Phi x}_k + \boldsymbol{b} w_k, \; x_0 = 0 \tag{2}$$

$$\tilde{y}_k = \boldsymbol{h}_k^T \boldsymbol{x}_k + v_k \tag{3}$$

for k=1,2,..., N-1 and where $\mathbf{x}_k$ is the $(n \times 1)$ state vector, $\tilde{y}_k$ is the measurement data. The matrix $\boldsymbol{\Phi}$ and vectors $\mathbf{b}$ and $\mathbf{h}_k$ are known [12]

$$\Phi(i,j) = \begin{cases} 1 & \text{for } i=j=1; i+1=j \\ 0 & \text{elsewhere} \end{cases}, \; dim(\Phi) = M \times M \tag{4}$$

$$\boldsymbol{b} = [1\,0\,0 \cdots 0]^T \;\; dim(\boldsymbol{b}) = M \tag{5}$$

Also v and w are two sequences of non correlated white noise with known invariant covariance σ_v^2 and σ_w^2 respectively. The covariance form of the Kalman filter has been preferred to the square root information Kalman filter form because of the singularity of the state matrix $\boldsymbol{\Phi}$. The SRCKF equations can be summarized as follows [2]:

Time Update:

$$\hat{\boldsymbol{x}}_{k+1/k} = \boldsymbol{\Phi} \hat{\boldsymbol{x}}_{k/k}, \qquad \hat{\boldsymbol{x}}_{0/0} = \boldsymbol{0} \tag{6}$$

$$\begin{bmatrix} \boldsymbol{S}^T{}_{k+1/k} \\ \boldsymbol{0} \end{bmatrix} = \boldsymbol{T} \begin{bmatrix} \boldsymbol{S}_{k/k}^T \boldsymbol{\Phi} \\ \boldsymbol{b}^T \end{bmatrix} \tag{7}$$

Measurement update:

$$\begin{bmatrix} \mathrm{f}_{k+1} & \boldsymbol{g}_{k+1}^T \\ 0 & \boldsymbol{S}_{k+1/k+1}^T \end{bmatrix} = \boldsymbol{T} \begin{bmatrix} \beta & \boldsymbol{0}^T \\ \boldsymbol{S}_{k+1/k}^T \boldsymbol{h}_{k+1} & \boldsymbol{S}_{k+1/k}^T \end{bmatrix} \tag{8}$$

$$\hat{\boldsymbol{x}}_{k+1/k+1} = \hat{\boldsymbol{x}}_{k+1/k} + \frac{\boldsymbol{g}_{k+1}}{\mathrm{f}_{k+1}} I_{k+1} \tag{9}$$

$$I_{k+1} = \tilde{y}_{k+1} - \boldsymbol{h}_{k+1}^T \hat{\boldsymbol{x}}_{k+1/k} \tag{10}$$

where $\beta = \sigma_v / \sigma_w$ is a parameter optimized empirically; $\hat{\boldsymbol{x}}_{k+1/k}$ is the prediction of state $\mathbf{x}$ at time k+1 given measurements and information up to and including time k; $\hat{\boldsymbol{x}}_{k+1/k+1}$ is the estimation of state $\mathbf{x}$ at time k+1 given measurements and information up to and including time k+1; $\boldsymbol{S}_{k+1/k+1}$ is the normalized covariance square-root of the estimation error; $\boldsymbol{S}_{k+1/k}$ is the normalized covariance square-root of the prediction error. $\mathbf{T}$ is an orthogonal transformation matrix. In this case, $\mathbf{T}$ is a QR triangularization matrix implicitly computed by MMAlpha using the Given's rotations [15]. The matrices $\boldsymbol{S}_{k+1/k+1}$, $\boldsymbol{S}_{k+1/k}$, $\mathbf{W}_k$ and $\mathbf{V}_k$ are defined by the Cholesky decomposition [2]

$$\boldsymbol{P}_{k+1/k+1} = \boldsymbol{S}_{k+1/k+1} \boldsymbol{S}_{k+1/k+1}^T \tag{11}$$

$$\boldsymbol{P}_{k+1/k} = \boldsymbol{S}_{k+1/k} \boldsymbol{S}_{k+1/k}^T \tag{12}$$

$$\boldsymbol{R}_{wk} = \boldsymbol{W}_k \boldsymbol{W}_k^T \tag{13}$$

$$\mathbf{R}_{\mathrm{v}k+1/k} = \mathbf{V}_{k+1/k} \mathbf{V}^{\mathrm{T}}_{k+1/k} \tag{14}$$

In these equations, the matrices $\boldsymbol{P}_{k+1/k+1}$ and $\boldsymbol{P}_{k+1/k}$ are covariance matrices defined in [1]. The replacement of covariance matrices by their square-root improves the numerical behavior of the filter, leading us to a system more robust to quantification effects. With this formulation, the reconstructed sample $\hat{s}_k$ corresponds to

Table I: Timesteps derivation from the scheduling given by MMAlpha.

Eq.	Alpha Variable Name (see Appendix)	Operation	Scheduling time given by MMAlpha i=1,2,..., M m=1,2,..., M	Number of timesteps
(7)	St	TA	$m+M$	$2M$
(10)	ye	$h^T_{k+1}\hat{x}_{k+1/k}$	$1+2M$	1
(8)	B		$1+m+2M$	M
	Sth	$S^T_{k+1}h_{k+1}$	$2+m+2M$	1
	f	$C(1,1)$	$3+3M$	1
	g	$C(1,i+1)$	$3+3M$	
(9)		$\hat{x}_{k+1/k+1}$	$4+3M$	1
(15)	xhatp	$\hat{s}_{k+1}$	$4+3M$	0
(8)	C	TB	$2+m+3M$	$M-1$
	Sp	$C(m+1,i+1)$	$3+m+3M$	$M-1$
Total			$3+4M$	

the sign detection of the extraction of the Mth element in the estimation state vector $\hat{x}_{k/k}$

$$\hat{s}_k = sign(\hat{x}_{k/k}(M)) \tag{15}$$

4. SYSTOLIC DESIGN WITH MMALPHA

In this particular algorithm dedicated to adaptive channel equalization, Φ is a sparse matrix, therefore, its multiplication by $\hat{x}_{k+1/k+1}$ and $S^T_{k+1/k}$ is obtained just by shifting the appropriate elements in matrix $S^T_{k+1/k}$.

Hence, operations who need to be scheduled in MMAlpha are the triangularization of matrices **A** (Eq. (7)) and **B** (Eq. (8)), plus the computation of Eq. (9). The triangularization of matrix **B** produces the matrix **C**. Given's Rotations and Eqs (6) to (9) are programmed in Alpha language and the scheduling is obtained in the MMAlpha environment with the schedule command [13]. The scheduling result given by MMAlpha is shown in Table I where the matrices **A, B** and **C** are defined as follows :

$$A = \begin{bmatrix} S^T_{k/k}\Phi \\ b^T \end{bmatrix} \tag{16}$$

$$B = \begin{bmatrix} \beta & 0^T \\ S^T_{k+1/k}h_{k+1} & S^T_{k+1/k} \end{bmatrix} \tag{17}$$

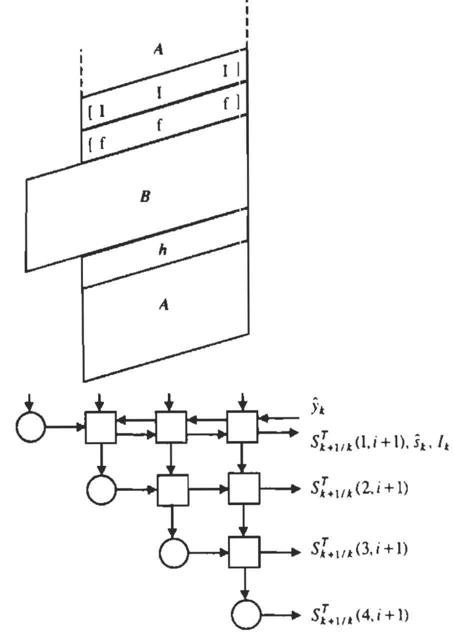


Figure 2: Overall systolic architecture

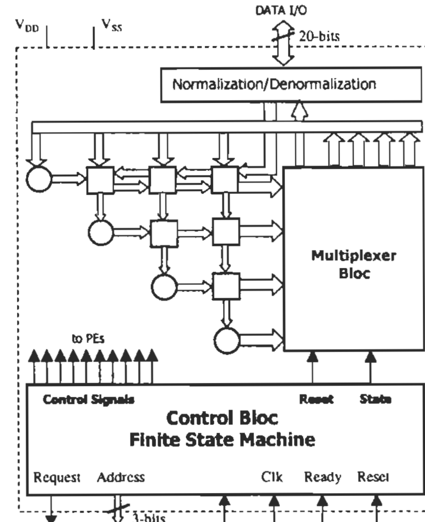


Figure 3: SRCKAL Processor bloc diagram

$$C = \begin{bmatrix} f_{k+1} & g^T_{k+1} \\ 0 & S^T_{k+1/k+1} \end{bmatrix} \tag{18}$$

We observe from this scheduling that the overall architecture will work in 3+4M timesteps per iterations. But the time sample $\hat{s}_k$ will be available at time 4+3M. Therefore, the throughput is (4+3M)f_c and the latency is (3+4M)/f_c where f_c is the clock frequency of the processor. The extra time is devoted to complete the computation of $S^T_{k+1/k+1}$ needed for the next iteration. The systolic architecture that we derived from this scheduling is shown

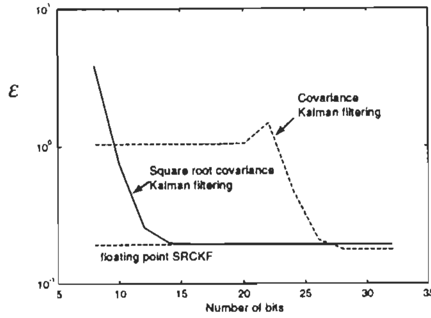


Figure 4: Relative mean square errors $\varepsilon(s, \hat{x}(M))$ for different number of bit in channel equalization with SNR=20dB.

in Figure 2. The circular and square processing elements (PE) of the architecture are described in [15]. More information on how to derive systolic architectures, particularly for the Kalman filter, with MMAlpha can be found in [1] and details on the MMAlpha environment are given in [7]-[9], [13].

The proposed processor structure named SRCKAL, for equalizer based on SRCKF is shown in Figure 3. It comprises the processing array shown above, a normalization/denormalization bloc which converts data into the suitable scale [-1,1] by dividing them by the normalization factor before feeding them into the processing array. After the computation, it converts the results back to their normal scale by multiplying them by the normalization factor. This normalization factor is taken equal to a power of two, which allows us to right-shift for normalization and left-shift for denormalization, instead of using a divider and a multiplier. The multiplexer is a combinatorial bloc who takes data output by the array and feed them to the appropriate input for the next timesteps. Therefore no data storage is needed in this architecture. The control bloc is a finite state machine designed with System Architect® of Mentor Graphics®. It interfaces with the external world and its main task is to generate the necessary control signals for each PE and the multiplexer bloc.

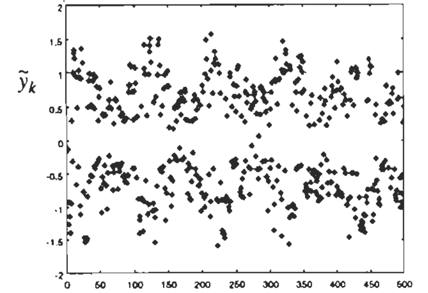
5. SIMULATION RESULTS AND PERFORMANCE EVALUATION

To study the proposed architecture derived from MMAlpha, we have experimented an adaptive channel equalization problem. Let us consider a time varying linear channel with the following three point impulse response:

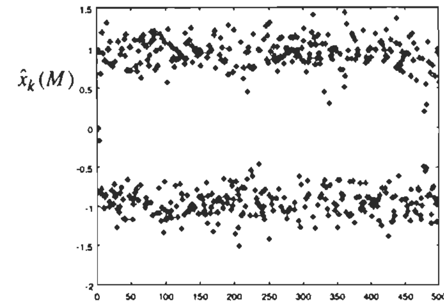
$$h_k(m) = h^{\circ}(m)[0.75 + 0.25\sin(2\pi kP/N)] \tag{19}$$

$$h^{\circ}(m) = \begin{cases} \frac{1}{2}[1 + \cos(2(m-2)/W)] & \text{for } m = 1,2,3 \\ 0 & \text{elsewhere} \end{cases} \tag{20}$$

for k=1,2,...,N and m=1,2,3 with W=2.9 which controls the dispersion of the channel and P=5 is the number of variation periods during the experiment.



a)



b)

Figure 5 : Test of correction for a time varying channel with SNR=20 dB and β=1: a) the output signal $\tilde{y}_k$ (BER=50%) and b) the correction $\hat{x}_k(M)$ result with VHDL simulation (BER=0).

The dynamic behavior of the covariance Kalman filter and the SRCKF is shown in Figure 4. From this figure, we can conclude that a 16-bits wordlength is a good tradeoff between the quality of the adaptive equalizer and the integration area of the proposed systolic processor. To assure the quality of correction, the processor has them been synthesized with a wordlength of 20-bits.

The performance evaluation is carried out in term of the latency and throughput. The design was made by means of standard CAD tools available from the Canadian Microelectronics Corporation (CMC). The structural model of the proposed architecture was made in VHDL using Mentor-Graphics CAD tools for register transfer level (RTL) modeling and simulation. The simulation results showing the reconstructed data at the processor output with signal noise ratio (SNR) of 20 dB are displayed in Figure 5. The reconstruction quality is assessed using the relative mean square error and the Bit Error Rate (BER) which is defined as the ratio of the number erroneous bit over the number of bits transmitted. The relative mean square error is equal to 17%. The BER is typically null for this noise level when we extent the simulation to 10 000 samples. More over, this BER is less than 0.4% for SNR to 10dB. One of the most advantages of this filter is that, it needs only M iterations to be fully adaptive, and the BER is practically constant

regardless of the number of samples.

A low-effort synthesis optimization was made with Synopsys tools with the Hewlett-Packard 0.5-µm CMOS technology available from MOSIS through CMC. The integration area is about 140 000 transistors and 30 000 transistors for the circle PE and square PE respectively. The evaluation of clock frequency is 40 MHz and 3 MHz for the circle PE and square PE respectively. The total number of transistors for M=3 is evaluated at 750 000 including all bloc of the processor shown in Fig. 3. The clock frequency, f_c, of the architecture is evaluated to 3 MHz and is limited by the Newton Raphson divider [16] in the circle PE. This frequency can be increase and the area decrease in using the systolic for division and square root proposed in [17].

6. CONCLUSION

In this paper, we have applied the square-root covariance Kalman filter to solve the adaptive channel equalization problem. The proposed systolic processor array has been derived automatically using MMAlpha tools. A study of the wordlength effect has shown that we can use 16-bits arithmetic units to conserve the same quality of reconstruction of that obtained with floating point arithmetic units. We have obtained a throughput of $(4+3M)f_c$ where M and f_c are the dimension of the channel and the clock frequency of the processor respectively. The next step of this design will consist in applying a sequence of low-level transformations which brings the description to a net-list format called AlpHard [7]. The transformation process is almost automated, the MMAlpha environment behaving as a compiler which automatically maps one description level to the next one. Eventually, one obtains a VHDL model to implement in a VLSI technology (e.g. FPGA, CMOS).

REFERENCES

[1] A. L. T. Mozipo, D. Massicotte, P. Quinton and T. Risset, "Automatic Synthesis of a Parallel Architecture for Kalman Filtering using MMAlpha", Int. Conf. on Parallel Computing in Electrical Engineering, Bialystok, Poland, Sept. 2-5, 1998, pp. 201-206.

[2] P. G. Kaminski, A. E. Bryson Jr., and S. F. Schmidt, "Discrete Square Root Filtering: A survey of Current Techniques", IEEE Trans. Automat. Contr., vol. AC-16, 1971, pp727-735.

[3] K. R Baker, A.D Brown, A.J. Currie, "Optimisation Efficiency in Behavioral Synthesis", IEE Proc.-Circuits Devices Syst., Vol. 141, No. 5, Oct. 1994, pp. 399-406.

[4] W. G. Irwin, "Architectures for Control", Algorithms and parallel VLSI architectures, Chap. 9, Elsevier Science, 1991, pp. 431-443.

[5] F. Gaston and G. Irwin, "VLSI architectures for square root covariance Kalman filtering", Proc. SPIE, vol.1152, 1989, pp. 44-55.

[6] S. Y. Kung and J. N. Hwang, "Systolic Array Designs for Kalman Filtering", IEEE Transactions in Signal Processing, vol. 39, N° 1, Jan. 1991, pp. 171-182.

[7] P. Le Moenner et al., "Generating Regular Arithmetic Circuits with ALPHARD", Massively Parallel Computing Systems (MPCS'96), Ischia, Italy, 6-9 May 1996.

[8] H. Le Verge, C. Mauras, and P. Quinton, "The ALPHA language and its use for the design of systolic arrays", Journal of VLSI Signal Processing,Vol.3, 1991, pp.173-182.

[9] C. Mauras, "ALPHA: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones", Thesis of Université de Rennes 1, Dec. 1989.

[10] R. Parisi, E.D. Di Claudio, G. Orlandi and B.D. Rao, "Fast Adaptive Digital Equalization by Recurrent Neural Networks", IEEE Trans. On Signal Processing, Vol 45, No 11, Nov. 1997, pp.2731-2739.

[11] S. Haykin, Adaptive Filter Theory, Prentice Hall, 1996.

[12] D. Massicotte, R. Z. Morawski, and A. Barwicz, "Incorporation of a Positivity Constraint Into a Kalman-Filter-Based Algorithm for Correction of Spectrometric Data", IEEE Trans. Instr. and Meas., Vol. 44, No 1, February 1995, pp. 2-7.

[13] "A langage for synthesis of regular architectures", http://www.irisa.fr/api/ALPHA/welcome.html, June 1998.

[14] M. Vidal, D. Massicotte, "A Parallel Architecture of a Piecewise Linear Neural Network for Nonlinear Channel Equalization", Instr&Meas. Tech. Conf., Venice, May 1999.

[15] P. Quinton and Y. Robert, Systolic Algorithms and Architectures, Prentice Hall, 1991.

[16] J. Hennessy and D.A. Patterson, "Computer Architecture: A quantitative Approach", McGraw-Hill, 1992.

[17] S.E. McQuillan and J.V. McCanny, "Fast VLSI Algorithms for Division and Square Root", J. of VLSI Signal processing, Vol. 8, 1994, pp. 151-168.

[18] J.G. Proakis, "Digital Communications", 3rd Ed., McGraw-Hill, 1995.

[19] R. Prasad, "Universal Wireless Personal Communications", Artech House, 1998.

APPENDIX

Alpha program describing one step of Kalman filtering

The following is the Alpha program for one step of the square root covariance Kalman filter. This program contains the definition of five subsystems. System sqrtcov is the main program, and corresponds, up to the renaming of some variables, to the equations given in Table I. This system calls four other subsystems, called givensm1m, givensm1m1 (two instanciations of the givens algorithm), matlvect (matrix vector multiplication) and dotproduct (for dot product). Each subsystem contains a list of input variables, a list of output variables (preceded by the keyword returns), a list of local variables, and a list of equations. In system sqrtcov, most of the equations are so-called use statements, and behave (roughly speaking) much as subroutine calls in a conventional language. Notice that all systems are parameterized by the size parameter M.

```
-- Dot product
system dotprod :{M | 2<=M}
        (v : {i | 1<=i<=M} of real;
            w : {i | 1<=i<=M} of real)
```

```
    returns (s : real);
...
-- matlvect : Returns a vector. Input 'a' is a lower triangular matrix.
system matlvect : {M | M>1}
                  (a : {i,j | 1<=i,j<=M; i<=j<=M} of real;
                   v : {i | 1<=i<=M} of real)
        returns   (c : {i | 1<=i<=M} of real);
...
-- Givens factorisation, specialized to the case (M+1)*M
system givensm1m : {M |M>1}
 (a : {i,j | 1<=i<=M+1; 1<=j<=M} of real)
returns
 (givens : {i,j | 1<=i<=M+1; 1<=j<=M } of real);
var
 A : {i,j,k | 0<=k<=M; k<i<=M+1; i>=1; k<=j<=M; j>=1} of real;
 Piv : {i,j,k | k<=i<=M+1; k<=j<=M;1<=k<=M+1} of real;
 C,S,T : {i,k | 1<=k<=M;k<i<=M+1} of real;
 Swap : {i,k | 1<=k<=M;k<i<=M+1} of boolean;
let
 Swap[i,k] = Piv[i-1,k,k]>A[i,k,k-1];

 T[i,k] = if Swap[i,k] then Piv[i-1,k,k]/A[i,k,k-1] else A[i,k,k-1]/Piv[i-1,k,k];

 C[i,k] = if (A[i,k,k-1]=0[]) then 1[] else
        (if (Swap[i,k]) then 1[]/sqrt(1[]+T[i,k]*T[i,k])*T[i,k]
                else 1[]/sqrt(1[]+T[i,k]*T[i,k]));
 S[i,k] = if (A[i,k,k-1]=0[]) then 1[] else
       (if (not Swap[i,k]) then 1[]/sqrt(1[]+T[i,k]*T[i,k])*T[i,k]
                 else 1[]/sqrt(1[]+T[i,k]*T[i,k]));

 Piv[i,j,k]=case
         {| i=k}: A[i,j,k-1];
         {| i>k }: C[i,k]*Piv[i-1,j,k]+S[i,k]*A[i,j,k-1];
         esac;

 A[i,j,k] =
  case
   {| k=0 }: a[i,j]; -- initialisation
   {| k>0; i>k; j>=k }: -S[i,k]*Piv[i-1,j,k]+C[i,k]*A[i,j,k-1];
  esac;
 givens[i,j] =
  case
   {| i>j}: 0[];
   {| i<=j}: Piv[M+1,j,i];
  esac;
tel;
-- Givens factorisation, specialized to the case (M+1)*(M+1)
system givensm1m1 : {M |M>1}
 (a : {i,j | 1<=i<=M+1; 1<=j<=M+1} of real)
returns
 (givens : {i,j | 1<=i<=M+1; 1<=j<=M+1} of real);
...
-- One step of the covariance method
system sqrtcov : {M | 1<M}
        (yb  : real;
                  H   : {m | 1<=m<=M } of real;
                  xhatp1 : {m | 1<=m<=M } of real;
                  Sp1 : {m,i | 1<=m<=M; 1<=i<=M } of real;
                  sigmav2, sigmaw2 : real)
    returns (xe : real;
                  Sp : {m,i | 1<=m<=M; 1<=i<=M } of real;
                  xhatp : {m | 1<=m<=M } of real);
var

 xhat,xhatextra : {m | 1<=m<=M } of real;
 Hextra,Hextra1: {m | 1<=m<=M } of real;
 A,Aextra : {m,i | 1<=m<=M+1; 1<=i<=M } of real;
 B,C : {m,i | 1<=m<=M+1; 1<=i<=M+1 } of real;
 U,V,f : real;

--  S : {m,i | 1<=m<=M; 1<=i<=m} of real;
 St,Stextra : {m,i | 1<=m<=M; m<=i<=M} of real;
 Sth,g : {m | 1<=m<=M} of real;
 ye : real;

let
 U = sqrt(sigmaw2[]);
 V = sqrt(sigmav2[]);

-- Equations (16) and (7)
 A[m,i] =
        case
                {| i=1; m<=M } : Sp1[1,m];
                {| 2<=i<=M; m<=M } : Sp1[i-1,m];
                {| m=M+1; i=1 } : U[];
                {| m=M+1; i>1 } : 0[];
        esac;
 Aextra = A;
 use givensm1m[M] (Aextra) returns (St); -- First call of Givens

-- Equations (17) and (8)
 Stextra = St;
 Hextra1 = H;
 use matlvect[M] (Stextra,Hextra1) returns (Sth);

 B[m,i] =
        case
                {| i=1; m=1 } : V[];
                {| 2<=i<=M+1; m=1 } : 0[];
                {| i=1; 2<=m<=M+1 } : Sth[m-1];
                {| 2<=m<=M+1; 2<=i<=M+1 } : St[m-1,i-1];
       {| i+1<=m<=M+1; 2<=i}: 0[];
        esac;

 use givensm1m1[M] (B) returns (C); -- Second call of Givens

 f[] = C[1,1];
 g[i] = C[1,i+1];
 Sp[m,i] = C[m+1,i+1];

-- Time Update
 xhat[m] =
        case
                {| m=1 } : xhatp1[1];
                {| 1<m<=M } : xhatp1[m-1];
        esac;

-- Equation (10)
 Hextra = H;
 xhatextra = xhat;
 use dotprod[M] (Hextra,xhatextra) returns (ye);

-- Equation (9)
xhatp[m] = xhat[m] + (g[m]/f[])*(yb[] - ye[]);

xe[] = xhatp[M];
tel;
```

Annexes II

Programmes Alpha

I. Bloc note des étapes de transformation pour la multiplication matrice-vecteur

Les expressions représentées en *Italique* sont celles les réponses données par MMAlpha aux commandes, représentées en `Normal`. Cet ordonnancement est présenté en détail en annexe 1.

Les phrases commençant par -- sont des commentaires.

```
-- Chargement du programme

load["prodVect.alpha"];ashow[];

system prodVect :{N | 3<=N}
   (a : {i,j | 1<=i<=N; 1<=j<=N} of integer;
    b : {i | 1<=i<=N} of integer)
   returns (c : {i | 1<=i<=N} of integer);
var  C : {i,j | 1<=i<=N; 0<=j<=N} of integer;
let  C[i,j] =
   case
     {| j=0} : 0[];
     {| 1<=j} : C[i,j-1] + a[i,j] * b[j];
   esac;
```

```
   c[i] = C[i,N];
tel;
```

-- Analyse

```
-- L'analyse doit être faite pendant tout le processus,
-- car certaines transformations peuvent modifier le
programme.

analyze[];

Static Analysis of system prodVect
--Checking declaration of variables.
--Checking single assignment rule.
   --Checking definitions of output/local variables.
--Checking definition of input variables.
--Checking that input/local variables are used.
--Checking type and domain consistency in the equations:
----equation defining C
----equation defining c
Analysis Successful...
True
```

-- **Ajout d'une variable locale pour a**

```
addlocal["A=a"]; ashow[];

system prodVect :{N | 3<=N}
   (a : {i,j | 1<=i<=N; 1<=j<=N} of integer;
    b : {i | 1<=i<=N} of integer)
        returns
   (c : {i | 1<=i<=N} of integer);
var
   A : {i,j | 1<=i<=N; 1<=j<=N; 3<=N} of integer;
   C : {i,j | 1<=i<=N; 0<=j<=N} of integer;
let
   A[i,j] = a;
   C[i,j] =
     case
          {| j=0} : 0[];
          {| 1<=j} : C[i,j-1] + A[i,j] * b[j];
     esac;
   c[i] = C[i,N];
tel;
```

-- Pipeline

```
-- Pipeline de la variable b[j]
-- L'expression b[j] est diffusée à tous les (i,j),
```

```
-- nous pipelinons cette diffusion le long de j

pipeall["C","b.(i,j->j)","B1.(i,j->i+1,j)"]; ashow[];

system prodVect :{N | 3<=N}
     (a : {i,j | 1<=i<=N; 1<=j<=N} of integer;
      b : {i | 1<=i<=N} of integer)
    returns
     (c : {i | 1<=i<=N} of integer);
var
     B1 : {i,j | 1<=i<=N; 1<=j<=N; 3<=N} of integer;
     A : {i,j | 1<=i<=N; 1<=j<=N; 3<=N} of integer;
     C : {i,j | 1<=i<=N; 0<=j<=N} of integer;
let
   B1[i,j] =
     case
          {| i=1; 1<=j<=N; 3<=N} : b[j];
          {| 2<=i<=N; 1<=j<=N; 3<=N} : B1[i-1,j];
     esac;
   A[i,j] = a;
   C[i,j] =
     case
          {| j=0} : 0[];
          {| 1<=j} : C[i,j-1] + A[i,j] * B1;
     esac;
   c[i] = C[i,N];
tel;

-- Ordonnancement (Scheduling)
-- Nous avons un programme uniforme et nous allons chercher -
- un ordonnancement avec la partie linéaire pour toutes
-- les variables du programme.

schedule[scheduleType->2];

Total execution Time: 1+2N
   T_a :{i,j,N} = 0
   T_b :{i,N} = 0
   T_c :{i,N} = 1+i+N
   T_B1 :{i,j,N} = -1+i+j
   T_A :{i,j,N} = -1+i+j
   T_C :{i,j,N} = i+j

-- Changement de base (mapping)
-- Nous appliquons le changement de base correspondant.

applySchedule[]; ashow[];
```

```
system prodVect :{N | 3<=N}
        (a : {i,j | 1<=i<=N; 1<=j<=N} of integer;
         b : {i | 1<=i<=N} of integer)
   returns
         (c : {i | 1<=i<=N} of integer);
var
   B1 : {t,j | j<=t<=j+N-1; 1<=j<=N; 3<=N} of integer;
   A : {t,j | j<=t<=j+N-1; 1<=j<=N; 3<=N} of integer;
   C : {t,j | j<=t<=j+N; 1<=j<=N} of integer;
let
   B1[t,j] =
     case
      {| 1<=t<=N; j=1; 3<=N} : b[t-j+1];
           {| j<=t<=j+N-1; 2<=j<=N; 3<=N} : B1[t-1,j-1];
     esac;
   A[t,j] = a[j,t-j+1];
   C[t,j] =
     case
           {| t=j} : 0[];
           {| j+1<=t} : C[t-1,j] + A[t-1,j] * B1[t-1,j];
     esac;
   c[i] = C[i+N,i];
tel;
```

-- Renommons les indices t,j -> t,p

```
mat2=readMat[(t,p,N->t,p,N)];
changeOfBasis[C,mat2];
changeOfBasis[A,mat2];
changeOfBasis[B1,mat2]; ashow[];
```

```
system prodVect :{N | 3<=N}
       (a : {i,j | 1<=i<=N; 1<=j<=N} of integer;
        b : {i | 1<=i<=N} of integer)
   returns
       (c : {i | 1<=i<=N} of integer);
var
   B1 : {t,p | p<=t<=p+N-1; 1<=p<=N; 3<=N} of integer;
   A : {t,p | p<=t<=p+N-1; 1<=p<=N; 3<=N} of integer;
   C: {t,p | p<=t<=p+N; 1<=p<=N} of integer;
let
   B1[t,p] =
     case
      {| 1<=t<=N; p=1; 3<=N} : b[t-p+1];
      {| p<=t<=p+N-1; 2<=p<=N; 3<=N} : B1[t-1,p-1];
     esac;
```

```
   A[t,p] = a[p,t-p+1];
   C[t,p] =
     case
          {| t=p} : 0[];
          {| p+1<=t} : C[t-1,p] + A[t-1,p] * B1[t-1,p];
     esac;
   c[i] = C[i+N,i];
tel;
```

-- **Downto Alpha0**
-- Alpha0 est un sous-ensemble de Alpha qui décrit l'architecture de façon non structurée. Ici, on présente la dérivation automatique de Alpha0 à partir du programme ordonnancé. La fonction toAlpha0v2 suivante permet de la génération des signaux de contrôle, le pipelinage des signaux de contrôle, décomposition en expressions simples.
--

```
toAlpha0v2[];
simplifyInContext[];
convexizeAll[];

Time index: {1} space indices: {2}
   Calling spaceTimeDecomposition[];
   Calling makeAllMuxControl[];
     Equation of B1...
     Equation of A...
     Equation of C...
      is in ST form
       Adding multiplexer.
     Equation of c...
Calling pipeAllControl[];
     Pipelining control for: C_ctl1
        From dimension 2
      To dimension 1
        Control generated in cell: {p | p=1; 3<=N}
Calling decomposeSTdeps[];
In equation of C_ctl1P, adding a local variable: C_ctl1P_reg1
In equation of B1, adding a local variable: B1_reg2
In equation of C, adding a local variable: C_reg3
In equation of C, adding a local variable: A_reg4
In equation of C, adding a local variable: B1_reg5
   Decomposing the space/time dependencies
    Space/time dependency to decompose in B1_reg2
      Adding local variable B1_reg2loc
    Space/time dependency to decompose in C_ctl1P_reg1
      Adding local variable C_ctl1P_reg1loc
```

```
Calling makeInputMirrorEqus[];
    Adding mirror equation for input b
```

-- Le programme résultant Alpha0 devrait être optimisé (en particulier pour la réutilisation des sous-exprssions communes). Le programme obtenu est le suivant.
--

```
   ashow[];

system prodVect :{N | 3<=N}
   (a : {i,j | 1<=i<=N; 1<=j<=N} of integer;
    b : {i | 1<=i<=N} of integer)
     returns
   (c : {i | 1<=i<=N} of integer);
var
   b_mirr1 : {t,p | 1<=t<=N; p=1; 3<=N} of integer;
   C_ctl1P_reg1loc : {t,p | p-1<=t<=p+N-1; 2<=p<=N+1; 3<=N}
                        of boolean;
   B1_reg2loc : {t,p | p-1<=t<=p+N-2; 2<=p<=N+1; 3<=N} of
                        integer;
   B1_reg5 : {t,p | p+1<=t<=p+N; 1<=p<=N; 3<=N} of
                        integer;
   A_reg4 : {t,p | p+1<=t<=p+N; 1<=p<=N; 3<=N} of integer;
   C_reg3 : {t,p | p+1<=t<=p+N; 1<=p<=N; 3<=N} of integer;
   B1_reg2 : {t,p | p<=t<=p+N-1; 2<=p<=N; 3<=N} of integer;
   C_ctl1P_reg1 : {t,p | p<=t<=p+N; 2<=p<=N; 3<=N} of
                        boolean;
   C_ctl1P_Init_In : {t,p | 1<=t<=N+1; p=1; 3<=N} of
                        boolean;
   C_ctl1P_Init : {t | 1<=t<=N+1; 3<=N} of boolean;
   C_ctl1P : {t,p | p<=t<=p+N; 1<=p<=N; 3<=N} of boolean;
   B1 : {t,p | p<=t<=p+N-1; 1<=p<=N; 3<=N} of integer;
   A : {t,p | p<=t<=p+N-1; 1<=p<=N; 3<=N} of integer;
   C : {t,p | p<=t<=p+N; 1<=p<=N} of integer;
   C_ctl1 : {t,p | p<=t<=p+N; 1<=p<=N; 3<=N} of boolean;
let
   b_mirr1[t,p] = b[t];n  B1_reg2loc[t,p] = B1[t,p-1];
   B1_reg2[t,p] = B1_reg2loc[t-1,p];
   C_ctl1P_reg1loc[t,p] = C_ctl1P[t,p-1];
   C_ctl1P_reg1[t,p] = C_ctl1P_reg1loc[t-1,p];
   B1_reg5[t,p] = B1[t-1,p];
   A_reg4[t,p] = A[t-1,p];
   C_reg3[t,p] = C[t-1,p];
   C_ctl1P_Init_In[t,p] = C_ctl1P_Init[t];
   C_ctl1P_Init[t] =
     case
```

```
          {| 1>=} :
               case
                    {| t=1} : True[];
                    {| 2<=t} : False[];
               esac;
     esac;
   B1[t,p] =
     case
          {| p=1} : b_mirr1;
          {| 2<=p} : B1_reg2;
     esac;
   A[t,p] = a[p,t-p+1];
   C_ctl1P[t,p] =
     case
          {| p=1} : C_ctl1P_Init_In[t,p];
          {| 2<=p} : C_ctl1P_reg1;
     esac;
   C_ctl1[t,p] = C_ctl1P;n  C[t,p] =
     case
          {| 1>=} : if (C_ctl1) then
               case
                    {| t=p} : 0[];
                    {| p+1<=t} : 0[];
               esac else
               case
                    {| t=p} : 0[];
                    {| p+1<=t} : C_reg3 + A_reg4 * B1_reg5;
               esac;
     esac;
   c[i] = C[i+N,i];
tel;
```

-- Translation en AlpHard

-- La traduction vers AlpHard nous permet d'obtenir une version structurée de la description architecturale. Cette traduction est faite avec la fonction alpha0toAplHard.
--

```
alpha0ToAlphard[controlVars[]];

-building Interface
checking form of output equation
-structuring Module prodVectModule
   ---Determining the different cell types...
   Please wait...
   16 region(s) found  before  simplification...
```

```
Simplify identical regions ...
    3 type(s) of cell have been  identified, their domains
   are:
   {p,N | p=1; 3<=N}
   {p,N | 2<=p<=N; 3<=N}
   {p,N | p=N+1; 3<=N}
   ---Generating Controler ...
   ---Building cell 1 present on :
   {p,N | p=1; 3<=N}
      Parameters of Cell :cellprodVectModule1
      input: {b_mirr1,C_ctl1P_Init_In,A}
      output: {B11,C_ctl1P1,C1}
   ---Building cell 2 present on :
   {p,N | 2<=p<=N; 3<=N}
      Parameters of Cell :cellprodVectModule2
      input: {B1_reg2loc,C_ctl1P_reg1loc,A}
      output: {B12,C_ctl1P2,C2}
   ---Building cell 3 present on :
   {p,N | p=N+1; 3<=N}
      Parameters of Cell :cellprodVectModule3
      input: {B1_reg2loc,C_ctl1P_reg1loc}
      output: {}
   alpha0ToAlphard::emptyCell:
        Warning: cell !(cellprodVectModule3) has no output
hence this
   cell is not generated
   adding local variables in library for isolating output
pins
   adding local variable: B1loc1
   adding local variable: C_ctl1Ploc2
   adding local variable: Cloc3
   adding local variable: B1loc1
   adding local variable: C_ctl1Ploc2
   adding local variable: Cloc3
   prodVect removed from $library
   prodVect removed from $result
   ControlprodVectModule added to library.
   cellprodVectModule1 added to library.
   cellprodVectModule2 added to library.
   prodVectModule added to library.
   prodVect added to library.

-- The new program in $result is

ashow[];

system prodVect :{N | 3<=N}
```

```
   (a : {i,j | 1<=i<=N; 1<=j<=N} of integer;
    b : {i | 1<=i<=N} of integer)
   returns
   (c : {i | 1<=i<=N} of integer);
var
   b_mirr1 : {t,p | 1<=t<=N; p=1; 3<=N} of integer;
   A : {t,p | p<=t<=p+N-1; 1<=p<=N; 3<=N} of integer;
   C : {t,p | p<=t<=p+N; 1<=p<=N} of integer;
let
   b_mirr1[t,p] = b[t];n  A[t,p] = a[p,t-p+1];n  c[i] =
                     C[i+N,i];
   use  prodVectModule[N] (b_mirr1, A)
   returns  (C) ;
tel;

-- Pour visualiser firModule (qui décrit l'architecture).
-- Cell 1 uniquement

getSystem[prodVectModule]; ashow[];
system prodVectModule :{N | 3<=N}
    (b_mirr1In : {t,p | 1<=t<=N; p=1; 3<=N} of integer;
     AIn : {t,p | p<=t<=p+N-1; 1<=p<=N; 3<=N} of integer)
      returns
    (COut : {t,p | p<=t<=p+N; 1<=p<=N} of integer);
var
   C_ctl1P_Init : {t | 1<=t<=N+1; 3<=N} of boolean;
   b_mirr1  : {t,p | 1<=t<=N; p=1; 3<=N} of integer;n  B11 :
{t,p | 1<=t<=N; p=1; 3<=N} of integer;
   B1_reg2loc : {t,p | p-1<=t<=p+N-2; 2<=p<=N+1; 3<=N} of
                               integer;
   C_ctl1P1 : {t,p | 1<=t<=N+1; p=1; 3<=N} of boolean;
   C_ctl1P_reg1loc : {t,p | p-1<=t<=p+N-1; 2<=p<=N+1; 3<=N}
                               of boolean;
   C_ctl1P_Init_In : {t,p | 1<=t<=N+1; p=1; 3<=N} of
                               boolean;
   A : {t,p | p<=t<=p+N-1; 1<=p<=N; 3<=N} of integer;
   C : {t,p | p<=t<=p+N; 1<=p<=N} of integer;
   C1 : {t,p | 1<=t<=N+1; p=1; 3<=N} of integer;
   B12 : {t,p | p<=t<=p+N-1; 2<=p<=N; 3<=N} of integer;
   C_ctl1P2 : {t,p | p<=t<=p+N; 2<=p<=N; 3<=N} of boolean;
   C2 : {t,p | p<=t<=p+N; 2<=p<=N; 3<=N} of integer;
let
   A[t,p] = AIn[t,p];n  B1_reg2loc[t,p] =
     case
        {| 1<=t<=N; p=2; 3<=N} : B11[t,p-1];
        {| p-1<=t<=p+N-2; 3<=p<=N+1; 3<=N} : B12[t,p-1];
     esac;
```

```
    b_mirr1[t,p] = b_mirr1In[t,p];
    C[t,p] =
         case
              {| 1<=t<=N+1; p=1; 3<=N} : C1[t,p];
              {| p<=t<=p+N; 2<=p<=N; 3<=N} : C2[t,p];
         esac;
    C_ctl1P_Init_In[t,p] = C_ctl1P_Init[t];
    C_ctl1P_reg1loc[t,p] =
         case
              {| 1<=t<=N+1; p=2; 3<=N} : C_ctl1P1[t,p-1];
              {| p-1<=t<=p+N-1; 3<=p<=N+1; 3<=N} :
                   C_ctl1P2[t,p-1];
         esac;
    COut[t,p] = C[t,p];
    use  ControlprodVectModule[N]
         ()
    returns  (C_ctl1P_Init) ;
    use {p | p=1; 3<=N} cellprodVectModule1[p,N] (b_mirr1,
              C_ctl1P_Init_In, A)
    returns  (B11, C_ctl1P1, C1) ;
    use {p | 2<=p<=N; 3<=N} cellprodVectModule2[p,N]
              (B1_reg2loc, C_ctl1P_reg1loc, A)
    returns  (B12, C_ctl1P2, C2) ;
tel;

-- La première cellule est spéciale, les N-1 autres sont
-- identiques.
--

getSystem[cellprodVectModule1];ashow[];

system cellprodVectModule1 :{p,N | p=1; 3<=N}
   (b_mirr1 : {t | 1<=t<=N; p=1; 3<=N} of integer;
    C_ctl1P_Init_In : {t | 1<=t<=N+1; p=1; 3<=N} of
              boolean;
    A : {t | 1<=t<=N; p=1; 3<=N} of integer)
    returns
   (B1 : {t | 1<=t<=N; p=1; 3<=N} of integer;
    C_ctl1P : {t | 1<=t<=N+1; p=1; 3<=N} of boolean;
    C : {t | 1<=t<=N+1; p=1; 3<=N} of integer);
var
   Cloc3 : {t | 1<=t<=N+1; p=1; 3<=N} of integer;
   C_ctl1Ploc2 : {t | 1<=t<=N+1; p=1; 3<=N} of boolean;
   B1loc1 : {t | 1<=t<=N; p=1; 3<=N} of integer;
   A_reg4 : {t | 2<=t<=N+1; p=1; 3<=N} of integer;
   B1_reg5 : {t | 2<=t<=N+1; p=1; 3<=N} of integer;
   C_ctl1 : {t | 1<=t<=N+1; p=1; 3<=N} of boolean;
```

```
   C_reg3 : {t | 2<=t<=N+1; p=1; 3<=N} of integer;
let
   C[t] = Cloc3[t];n  C_ctl1P[t] = C_ctl1Ploc2[t];
   B1[t] = B1loc1[t];
   B1_reg5[t] = B1loc1[t-1];
   A_reg4[t] = A[t-1];
   C_reg3[t] = Cloc3[t-1];
   B1loc1[t] = b_mirr1[t];
   C_ctl1Ploc2[t] = C_ctl1P_Init_In[t];
   C_ctl1[t] = C_ctl1Ploc2[t];
   Cloc3[t] =
     case
     {| t=1; p=1; 3<=N} : if (C_ctl1[t]) then 0[] else 0[];
     {| 2<=t; p=1; 3<=N} : if (C_ctl1[t]) then 0[] else
               C_reg3[t] + A_reg4[t] * B1_reg5[t];
     esac;
tel;

getSystem[prodVect];
asaveLib[/u/hping/memoire/alpha/demos/prodVectHard.alpha];

-- Generation du VHDL
-- Pour la génération du VHDL on affecte des valeurs aux
-- paramètres.

   getSystem[prodVectModule];
   assignParameterValue[N,10];
   putSystem[];
   getSystem[cellprodVectModule1];
   assignParameterValue[N,10];
   putSystem[];
   getSystem[cellprodVectModule2];
   assignParameterValue[N,10];
   putSystem[];
   getSystem[ControlprodVectModule];
   assignParameterValue[N,10];
   putSystem[];
   getSystem[prodVect];
   assignParameterValue[N,10];
   putSystem[];

prodVectModule replaced in library.
N suppressed in use of prodVectModule in $library
prodVectModule replaced in library.
cellprodVectModule1 replaced in library.
N suppressed in use of
cellprodVectModule1 in $library
```

```
cellprodVectModule1 replaced in library.
cellprodVectModule2 replaced in library.
N suppressed in use of
cellprodVectModule2 in $library
cellprodVectModule2 replaced in library.
ControlprodVectModule replaced in library.
N suppressed in use of
ControlprodVectModule in $library
ControlprodVectModule replaced in library.
prodVect replaced in library.
N suppressed in use of prodVectin $library
prodVect replaced in library.

asaveLib[/u/hping/memoire/alpha/demos/prodVectHard10.alpha];

-- À ce niveau il y a encore une transformation manuelle
-- pour obtenir le modèle alpHard définitif.

load[prodVectHard10man.alpha];
    Library Loaded

ashow[];

   system prodVectModule (b_mirr1In : {t,p | 1<=t<=10; p=1}
of integer;
   AIn : {t,p | p<=t<=p+9; 1<=p<=10} of integer)
   returns
   (COut : {t,p | p<=t<=p+10; 1<=p<=10} of integer);
var
   C_ctl1_Init : {t | 1<=t<=11} of boolean;
   b_mirr1 : {t,p | 1<=t<=10; p=1} of integer;
   B11 : {t,p | 1<=t<=10; p=1} of integer;
   B1_reg2loc : {t,p | p-1<=t<=p+8; 2<=p<=11} of integer;
   C_ctl11 : {t,p | 1<=t<=11; p=1} of boolean;
   C_ctl1_reg1loc : {t,p | p-1<=t<=p+9; 2<=p<=11} of
                    boolean;
   C_ctl1_Init_In : {t,p | 1<=t<=11; p=1} of boolean;
   A : {t,p | p<=t<=p+9; 1<=p<=10} of integer;
   C : {t,p | p<=t<=p+10; 1<=p<=10} of integer;
   C1 : {t,p | 1<=t<=11; p=1} of integer;
   B12 : {t,p | p<=t<=p+9; 2<=p<=10} of integer;
   C_ctl12 : {t,p | p<=t<=p+10; 2<=p<=10} of boolean;
   C2 : {t,p | p<=t<=p+10; 2<=p<=10} of integer;
let
   A[t,p] = AIn[t,p];
   B1_reg2loc[t,p] =
   case
```

```
      {| 1<=t<=10; p=2} : B11[t,p-1];
      {| p-1<=t<=p+8; 3<=p<=11} : B12[t,p-1];
   esac;
   b_mirr1[t,p] = b_mirr1In[t,p];n  C[t,p] =
   case
      {| 1<=t<=11; p=1} : C1[t,p];
      {| p<=t<=p+10; 2<=p<=10} : C2[t,p];
   esac;
   C_ctl1_Init_In[t,p] = C_ctl1_Init[t];
   C_ctl1_reg1loc[t,p] =
   case
      {| 1<=t<=11; p=2} : C_ctl11[t,p-1];
      {| p-1<=t<=p+9; 3<=p<=11} : C_ctl12[t,p-1];
   esac;
   COut[t,p] = C[t,p];
   use ControlprodVectModule[] () returns  (C_ctl1_Init) ;
   use {p | p=1} cellprodVectModule1[p] (b_mirr1,
     C_ctl1_Init_In, A) returns  (B11, C_ctl11, C1) ;
   use {p | 2<=p<=10} cellprodVectModule2[p] (B1_reg2loc,
   C_ctl1_reg1loc, A) returns  (B12, C_ctl12, C2) ;
tel;

-- Pour générer les fichiers VHDL files dans /tmp, taper:

dir=Directory[];
SetDirectory[/tmp];
alphaToVHDL[];

Tinit fixe a 0
Please wait...
Alpha`Vhdl`Private`alUnitairebis::ok:
   Fichier !(ControlprodVectModule).vhd Ok
Alpha`Vhdl`Private`trtCaseCellule::oldMux:
   ? Forme peu sure d'expression du mux. Verifier la
traduction !
Alpha`Vhdl`Private`alUnitairebis::ok:
       Fichier !(cellprodVectModule1).vhd Ok
Alpha`Vhdl`Private`trtCaseCellule::oldMux:
   ? Forme peu sure d'expression du mux. Verifier la
traduction !
Alpha`Vhdl`Private`alUnitairebis::ok:
    Fichier !(cellprodVectModule2).vhd Ok
Alpha`Vhdl`Private`alUnitaire::inter:
    Systeme !(prodVectModule) traduit comme si c'etait un
module.
Verifier que c'est bien le cas
Alpha`Vhdl`Private`genArchitecture::warning:
```

```
    ? WARNING : le systeme traduit n'est peut-etre pas un
module
Alpha`Vhdl`Private`alUnitairebis::ok:
    Fichier !(prodVectModule).vhd Ok
```

II – Listing du code VHDL généré par MMAlpha pour la multiplication matrice-vecteur précédente.

■ Fichier definition.vhd

```
library IEEE;
 use IEEE.std_logic_1164.all;

 Package definition is

type b_mirr1InType is array (1 to 1) of Integer range -255 to 255;
type AInType is array (1 to 10) of Integer range -255 to 255;
type COutType is array (1 to 10) of Integer range -255 to 255;
type b_mirr1Type is array (1 to 1) of Integer range -255 to 255;
type B11Type is array (1 to 1) of Integer range -255 to 255;
type B1_reg2locType is array (2 to 11) of Integer range -255 to 255;
type AType is array (1 to 10) of Integer range -255 to 255;
type CType is array (1 to 10) of Integer range -255 to 255;
type C1Type is array (1 to 1) of Integer range -255 to 255;
type B12Type is array (2 to 10) of Integer range -255 to 255;
type C2Type is array (2 to 10) of Integer range -255 to 255;
end definition;
```

■ Fichier prodVectModule.vhd

```
-- VHDL Model Created for "system prodVectModule"
 --  28/5/1998 16:34:45

library IEEE;
use IEEE.std_logic_1164.all;
library WORK;
use WORK.definition.all;


entity prodVectModule is
      Port ( Ck : In std_logic;
      Rst : In std_logic;
      b_mirr1In : In b_mirr1InType;
      AIn : In AInType;
      COut : Out COutType );
end prodVectModule;


architecture Behavioral of prodVectModule is

  signal C_ctl1_Init : std_logic;
  signal b_mirr1 : b_mirr1Type;
  signal B11 : B11Type;
  signal B1_reg2loc : B1_reg2locType;
  signal C_ctl11 : std_logic_vector(1 to 1);
  signal C_ctl1_reg1loc : std_logic_vector(2 to 11);
```

```
  signal C_ctl1_Init_In : std_logic_vector(1 to 1);
  signal A : AType;
  signal C : CType;
  signal C1 : C1Type;
  signal B12 : B12Type;
  signal C_ctl12 : std_logic_vector(2 to 10);
  signal C2 : C2Type;

Component cellprodVectModule1
      Port ( Ck : In std_logic;
      b_mirr1 : In Integer range 0 to 255;
      C_ctl1_Init_In : In std_logic;
      A : In Integer range 0 to 255;
      B1 : Out Integer range 0 to 255;
      C_ctl1 : Out std_logic;
      C : Out Integer range 0 to 255 );
end Component;


Component cellprodVectModule2
      Port ( Ck : In std_logic;
      B1_reg2loc : In Integer range 0 to 255;
      C_ctl1_reg1loc : In std_logic;
      A : In Integer range 0 to 255;
      B1 : Out Integer range 0 to 255;
      C_ctl1 : Out std_logic;
      C : Out Integer range 0 to 255 );
end Component;


Component ControlprodVectModule
      Port ( Ck : In std_logic;
      Rst : In std_logic;
      C_ctl1_Init : Out std_logic );
end Component;


begin

ETIQUETTE1 : FOR i IN 1 to 10 GENERATE
   A(i) <= AIn(i);
END GENERATE ETIQUETTE1;

ETIQUETTE2 : FOR i IN 2 to 2 GENERATE
   B1_reg2loc(i) <= B11(i-1);
END GENERATE ETIQUETTE2;

ETIQUETTE3 : FOR i IN 3 to 11 GENERATE
   B1_reg2loc(i) <= B12(i-1);
END GENERATE ETIQUETTE3;

ETIQUETTE4 : FOR i IN 1 to 1 GENERATE
   b_mirr1(i) <= b_mirr1In(i);
END GENERATE ETIQUETTE4;

ETIQUETTE5 : FOR i IN 1 to 1 GENERATE
   C(i) <= C1(i);
END GENERATE ETIQUETTE5;
```

```
ETIQUETTE6 : FOR i IN 2 to 10 GENERATE
   C(i) <= C2(i);
END GENERATE ETIQUETTE6;

ETIQUETTE7 : FOR i IN 1 to 1 GENERATE
   C_ctl1_Init_In(i) <= C_ctl1_Init;
END GENERATE ETIQUETTE7;

ETIQUETTE8 : FOR i IN 2 to 2 GENERATE
   C_ctl1_reg1loc(i) <= C_ctl11(i-1);
END GENERATE ETIQUETTE8;

ETIQUETTE9 : FOR i IN 3 to 11 GENERATE
   C_ctl1_reg1loc(i) <= C_ctl12(i-1);
END GENERATE ETIQUETTE9;

ETIQUETTE10 : FOR i IN 1 to 10 GENERATE
   COut(i) <= C(i);
END GENERATE ETIQUETTE10;

ETIQUETTE11: ControlprodVectModule Port Map(Ck,Rst,C_ctl1_Init);

ETIQUETTE12 : FOR i IN 1 to 1 GENERATE
   ETIQUETTE13: cellprodVectModule1 PORT MAP(
Ck,b_mirr1(i),C_ctl1_Init_In(i),A(i),B11(i),C_ctl11(i),C1(i));
 END GENERATE  ETIQUETTE12;

ETIQUETTE14 : FOR i IN 2 to 10 GENERATE
   ETIQUETTE15: cellprodVectModule2 PORT MAP(
Ck,B1_reg2loc(i),C_ctl1_reg1loc(i),A(i),B12(i),C_ctl12(i),C2(i));
 END GENERATE  ETIQUETTE14;

end Behavioral;
```

■ Fichier ControlprodVectModule.vhd

```
-- VHDL Model Created for "system ControlprodVectModule"
 --  28/5/1998 16:34:42

library IEEE;
use IEEE.std_logic_1164.all;

entity ControlprodVectModule is
      Port ( Ck : In std_logic;
      Rst : In std_logic;
      C_ctl1_Init : Out std_logic );
end ControlprodVectModule;


architecture state_machine of ControlprodVectModule is
  signal cpt : integer;
  type states is (E0,E0bis,E1,E2);
  signal currentState,nextState : states;

begin
reset_smreset_sm : PROCESS
```

```
 begin
-- compass stateMachine adj currentState

  WAIT UNTIL (Ck = '1' AND Ck'event);

  IF Rst ='1' THEN
   cpt <= -1;
   currentState <= E0;
  ELSE
   cpt <= cpt + 1;
   currentState <= nextState;
  END IF;
 END PROCESS;

evolution_sm : PROCESS(cpt,currentState)
 begin
 CASE currentState IS
  WHEN E0 => IF( cpt < 0) THEN nextState <= E0;
    ELSIF( cpt = 0) THEN  nextState <= E1; END IF;
  WHEN E1 =>  IF (cpt = 1) THEN nextState <= E2; END IF;
  WHEN E2 => IF ((cpt >= 2) AND (cpt < 11 )) THEN nextState <= E2;  END
IF;
   IF (cpt = 11) THEN nextState <= E0bis; END IF;-- remise a zero de la
SM
  WHEN OTHERS => -- erreurs et hors service
   nextState <= E0bis ;
 END CASE;
END PROCESS;

output_sm : PROCESS(currentState)
 begin
 CASE currentState IS
  WHEN E1=>C_ctl1_Init <= '1';
  WHEN E2=>C_ctl1_Init <= '0';
  WHEN OTHERS =>
   C_ctl1_Init <= 'X';
 END CASE;
END PROCESS;

END state_machine;


Fichier cellprodVectModule1.vhd
-- VHDL Model Created for "system cellprodVectModule1"
 --  28/5/1998 16:34:43

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

-- library COMPASS_LIB;
--  use COMPASS_LIB.STDCOMP.all;
-- library COMPASS_LIB;
-- use COMPASS_LIB.COMPASS.all;

entity cellprodVectModule1 is
      Port ( Ck : In std_logic;
      b_mirr1 : In Integer range 0 to 255;
```

```
      C_ctl1_Init_In : In std_logic;
      A : In Integer range 0 to 255;
      B1 : Out Integer range 0 to 255;
      C_ctl1 : Out std_logic;
      C : Out Integer range 0 to 255 );
end cellprodVectModule1;


architecture Behavioral of cellprodVectModule1 is

  signal Cloc3 : Integer range 0 to 255;
  signal C_ctl1loc2 : std_logic;
  signal B1loc1 : Integer range 0 to 255;
  signal A_reg4 : Integer range 0 to 255;
  signal B1_reg5 : Integer range 0 to 255;
  signal C_reg3 : Integer range 0 to 255;

begin


  C <= Cloc3;

  C_ctl1 <= C_ctl1loc2;

  B1 <= B1loc1;

  process(ck)
  begin
    if (ck='1' AND ck'event) then
        B1_reg5 <= B1loc1;
    end if;
  end process;

  process(ck)
  begin
    if (ck='1' AND ck'event) then
        A_reg4 <= A;
    end if;
  end process;

  process(ck)
  begin
    if (ck='1' AND ck'event) then
        C_reg3 <= Cloc3;
    end if;
  end process;

  B1loc1 <= b_mirr1;

  C_ctl1loc2 <= C_ctl1_Init_In;

  Cloc3 <=
    0 when C_ctl1loc2 = '1'  else
    0 when C_ctl1loc2 = '1'  else (C_reg3 + (A_reg4 * B1_reg5));


end Behavioral;
```

■ Fichier cellprodVectModule2

```
-- VHDL Model Created for "system cellprodVectModule2"
 --  28/5/1998 16:34:44

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

-- library COMPASS_LIB;
--  use COMPASS_LIB.STDCOMP.all;
-- library COMPASS_LIB;
-- use COMPASS_LIB.COMPASS.all;

entity cellprodVectModule2 is
      Port ( Ck : In std_logic;
      B1_reg2loc : In Integer range 0 to 255;
      C_ctl1_reg1loc : In std_logic;
      A : In Integer range 0 to 255;
      B1 : Out Integer range 0 to 255;
      C_ctl1 : Out std_logic;
      C : Out Integer range 0 to 255 );
end cellprodVectModule2;


architecture Behavioral of cellprodVectModule2 is

  signal Cloc3 : Integer range 0 to 255;
  signal C_ctl1loc2 : std_logic;
  signal B1loc1 : Integer range 0 to 255;
  signal A_reg4 : Integer range 0 to 255;
  signal B1_reg2 : Integer range 0 to 255;
  signal B1_reg5 : Integer range 0 to 255;
  signal C_ctl1_reg1 : std_logic;
  signal C_reg3 : Integer range 0 to 255;

begin


  C <= Cloc3;

  C_ctl1 <= C_ctl1loc2;

  B1 <= B1loc1;

  process(ck)
  begin
    if (ck='1' AND ck'event) then
        B1_reg2 <= B1_reg2loc;
    end if;
  end process;

  process(ck)
  begin
    if (ck='1' AND ck'event) then
```

```
        C_ctl1_reg1 <= C_ctl1_reg1loc;
    end if;
  end process;

  process(ck)
  begin
    if (ck='1' AND ck'event) then
        B1_reg5 <= B1loc1;
    end if;
  end process;

  process(ck)
  begin
    if (ck='1' AND ck'event) then
        A_reg4 <= A;
    end if;
  end process;

  process(ck)
  begin
    if (ck='1' AND ck'event) then
        C_reg3 <= Cloc3;
    end if;
  end process;

  B1loc1 <= B1_reg2;

  C_ctl1loc2 <= C_ctl1_reg1;

  Cloc3 <=
    0 when C_ctl1loc2 = '1'  else
    0 when C_ctl1loc2 = '1'  else (C_reg3 + (A_reg4 * B1_reg5));


end Behavioral;
```

III. Programme Alpha du filtre de Kalman standard

Cette version est celle qui a été simulée par comparer aux résultats obtenus avec Matlab® Pour faire l'ordonnancement à ce stade d'évolution de MMAlpha, il faut écrire le programme pour un seul échantillon de la mesure. Donc il faut supprimer l'indice de temps qui est celui variant de 1 (0 pour les variables à initialiser) à N. Le programme correspondant est donné à la fin de l'article intitulé **"Automatic Synthesis of a Parallel Architecture for Kalman Filtering using MMAlpha"** de l'Annexe I.

```
-- *****************************************************************
-- *
-- * Filtre de Kalman de covariance,
-- * pour egalisation adaptative des canaux
-- *                                                             -- *
Par :                                                   *
-- *                                                            +
-- * Aurelien T. Mozipo
-- * Le 21 Mars 1999                                            *
-- *
-- *****************************************************************
```

```
--  matmult : returns a full matrix. Inputs 'a' end 'b' are both
-- full matrix

system matmult : {M |M>1}
  (a,b : {i,j | 1<=i,j<=M} of real)
returns
  (c : {i,j | 1<=i,j<=M } of real);
var
  C : {i,j,k | 1<=i,j<=M; 0<=k<=M} of real;
let
  c[i,j] = C[i,j,M];
  C[i,j,k] = case
        {|k=0} : 0[];
        {|1<=k<=M} : C[i,j,k-1]+a[i,k]*b[k,j];
  esac;
tel;
```

```
-- matmultsym : Returns a lower triangular matrix. Inputs 'a' end
-- 'b' are both full matrix

system matmultsym : {M |M>1}
  (a,b : {i,j | 1<=i,j<=M} of real)
returns
  (c : {i,j | 1<=i<=M; 1<=j<=i } of real);
var
  C : {i,j,k | 1<=i<=M; 1<=j<=i; 0<=k<=M} of real;
let
  c[i,j] = C[i,j,M];
  C[i,j,k] = case
        {|k=0} : 0[];
        {|1<=k<=M} : C[i,j,k-1]+a[i,k]*b[k,j];
  esac;
tel;
```

```
-- matmultsym2 : Returns a lower triangular matrix. Input 'b' is a
-- lower triangular matrix

system matmultsym2 : {M |M>1}
  (a : {i,j | 1<=i,j<=M} of real;
   b : {i,j | 1<=i<=M; 1<=j<=i} of real)
returns
  (c : {i,j | 1<=i<=M; 1<=j<=i } of real);
var
  C : {i,j,k | 1<=i<=M; 1<=j<=i; 0<=k<=M} of real;
let
  c[i,j] = C[i,j,M];
  C[i,j,k] = case
        {|k=0} : 0[];
        {|1<=k<=M; j<=k} : C[i,j,k-1]+a[i,k]*b[k,j];
        {|1<=k<=M; j>k} : C[i,j,k-1]+a[i,k]*b[j,k];
  esac;
tel;
```

```
-- matsymmult : Returns a full matrix. Input 'b' is a lower
-- triangular matrix

system matsymmult : {M |M>1}
  (a : {i,j | 1<=i,j<=M} of real;
   b : {i,j | 1<=i<=M; 1<=j<=i} of real)
returns
  (c : {i,j | 1<=i,j<=M } of real);
var
  C : {i,j,k | 1<=i,j<=M; 0<=k<=M} of real;
let
  c[i,j] = C[i,j,M];
  C[i,j,k] = case
        {|k=0} : 0[];
        {|1<=k<=M; j<=k} : C[i,j,k-1]+a[i,k]*b[k,j];
        {|1<=k<=M; j>k} : C[i,j,k-1]+a[i,k]*b[j,k];
  esac;
tel;



-- matmult3sym : Returns a lower triangular matrix. Input 'b' is a
-- lower triangular matrix.

system matmult3sym :  {M |M>1}
  (a :  {i,j | 1<=i,j<=M} of real;
   b :  {i,j | 1<=i<=M; 1<=j<=i} of real;
   c :  {i,j | 1<=i,j<=M} of real)
returns
  (d :  {i,j | 1<=i<=M; 1<=j<=i } of real);
var
  tmp: {i,j | 1<=i,j<=M } of real;
let
  use matsymmult[M]
      (a,b)
      returns
      (tmp);
  use matmultsym[M]
      (tmp,c)
      returns
      (d);
tel;



system transp : {M | M>1}
  (a : {i,j | 1<=i,j<=M} of real)
returns
  (at : {i,j | 1<=i,j<=M} of real);
let
      at[i,j] = a[j,i];
tel;
```

```
-- matvect : Returns a vector. Input 'a' is a full matrix.

system matvect : {M | M>1}
          (a : {i,j | 1<=i,j<=M} of real;
           v : {i | 1<=i<=M} of real)
     returns      (c : {i | 1<=i<=M} of real);
var
     C : {i,j | 1<=i<=M; 0<=j<=M} of real;
let
     C[i,j] = case
          {| j=0} : 0[];
          {| 1<=j<=M} : C[i,j-1] + a[i,j]*v[j];
           esac;
     c[i] = C[i,M];

tel;



-- matsymvect : Returns a vector. Input 'a' is a lower triangular
-- matrix.

system matsymvect : {M | M>1}
          (a : {i,j | 1<=i<=M; 1<=j<=i} of real;
           v : {i | 1<=i<=M} of real)
     returns      (c : {i | 1<=i<=M} of real);
var
     C : {i,j | 1<=i<=M; 0<=j<=M} of real;
let
     C[i,j] = case
          {| j=0} : 0[];
          {| j>=1; j<=i} : C[i,j-1] + a[i,j]*v[j];
          {| j>=1; j>i} : C[i,j-1] + a[j,i]*v[j];
           esac;
     c[i] = C[i,M];
tel;



-- ----------------------------------------------------------------------
--
-- KalNStat : covariance kalman filtering for channel equalization
--
-- ----------------------------------------------------------------------

system KalNStat : {N,M | 1<=N; 1<M}
               (yb   : {n | 1<=n<=N} of real;
          phi  : {m,i | 1<=m,i<=M} of real;
          b    : {m | 1<=m<=M} of real;
          H    : {m,n | 1<=m<=M; 0<=n<=N} of real;
          beta :  real)
      returns (Kkal : {m,n | 1<=m<=M; 1<=n<=N} of real;
          xe : {n | 0<=n<=N} of real);
var
  Z : {m,n | 0<=n<=N; 1<=m<=M} of real;
```

```
  Zint1 : {m,n | 1<=m<=M; 0<=n<=N} of real;
  Zint : {m,n | 1<=m<=M; 1<=n<=N} of real;
  Ve,Veint,invVe : {n | 1<=n<=N} of real;
  I : {n | 1<=n<=N} of real;
  Ye : {n | 1<=n<=N} of real;
  h : {m,n | 1<=m<=M;  0<=n<=N} of real;
  h_1,PPh_1,PPh : {m,n | 1<=m<=M;  1<=n<=N} of real;
  P, FPFt : {m,i,n | 1<=m<=M; 1<=i<=m; 0<=n<=N} of real;
  Pint1 : {m,i,n | 1<=m,i<=M; 1<=n<=N} of real;
  Pint2,PP : {m,i,n | 1<=m<=M; 1<=i<=m; 1<=n<=N} of real;

  Ip,bbt : {m,i | 1<=m,i<=M} of real;
  F,Ft : {m,i,n | 1<=m,i<=M; 0<=n<=N} of real;
let

      F[m,i,n] = phi[m,i];

      h[m,n] = H[m,n];

      h_1[m,n] = h[m,n-1];


use {n|0<=n<=N} transp[M]
      (F)
returns (Ft);

use {n|0<=n<=N} matmult3sym[M]
      (F,P,Ft)
returns (FPFt);

use {n|1<=n<=N} matsymvect[M]
      (PP,h_1)
returns (PPh_1);

use {n|1<=n<=N} matsymvect[M]
      (PP,h)
returns (PPh);

use {n|1<=n<=N} matmultsym2[M]
      (Pint1,PP)
returns (Pint2);

      Ip[m,i] =
            case
                  {| m=i} : 1[];
                  {| m<=i-1} | {| m>=i+1} : 0[];
            esac;

      bbt[m,i] = b[m]*beta[]*b[i];
      PP[m,i,n] = FPFt[m,i,n-1] + bbt[m,i];

      Veint[n] = reduce(+,(m,n->n), h_1.(m,n->m,n) * PPh_1.(m,n-
>m,n))[n];
      Ve[n] = Veint[n] + 1[];
      invVe[n] = 1[]/Ve[n];
      Kkal[m,n] = PPh[m,n]*invVe[n];
      Pint1[m,i,n] = Ip[m,i] - Kkal[m,n]*h[i,n];
      P[m,i,n] =
```

```
                case
                        {| n=0} : Ip[m,i];
                        {| 1<=n<=N} : Pint2[m,i,n];
                esac;
        Z[m,n] =
                case
                        {| 1<=m<=M; n=0} : 0[];
                        {| 1<=m<=M; 1<=n<=N} : Zint[m,n] + Kkal[m,n] * I[n];
                        esac;
        Ye[n] = reduce(+, (m,n->n), h.(m,n->m,n) * Zint.(m,n->m,n))[n];
        I[n] = yb[n] - Ye[n];

use {n|0<=n<=N} matvect[M]
        (F,Z)
returns (Zint1);

        Zint[m,n] = Zint1[m,n-1];

-- Filtered sample

        xe[n] =
                case
                        {| 0<=n<=N-1} : Z[M,n];
                        {| n=N} : Z[1,N];
                esac;
tel;
```

IV. Programme Alpha du filtre de Kalman racine carrée de covariance

Les remarques faites sur le filtre de Kalman standard précédent sont également valables ici. Le programme pour une étape est donné à la fin de l'article intitulé A **"Parallel Architecture for Adaptive Channel Equalization Based on Kalman Filter Using MMAlpha".**

```
-- *****************************************************************
-- *                                                               +
-- * Filtre de Kalman racine carree de covariance, pour egalisation
-- * adaptative des canaux  *
-- *                                                               *
-- * Par :                                                         *
-- *                                                               +
-- * Aurelien T. Mozipo
-- * Le 21 Mars 1999                                               *
-- *
-- *****************************************************************


--
-- transp : a is an upper triangular matrix

system transp : {M | M>1}
  (a : {i,j | 1<=i<=M; i<=j<=M } of real)
returns
  (at : {i,j | 1<=i<=M; 1<=j<=i } of real);
```

```
let
      at[i,j] = a[j,i];
tel;


-- matlvect : Returns a vector. Input 'a' is a lower triangular matrix.

system matlvect : {M | M>1}
            (a : {i,j | 1<=i<=M; 1<=j<=i} of real;
             v : {i | 1<=i<=M} of real)
      returns      (c : {i | 1<=i<=M} of real);
var
      C : {i,j | 1<=i<=M; 0<=j<=i} of real;
let
      C[i,j] = case
                  {| j=0 } : 0[];
                  {| j>0 } : C[i,j-1] + a[i,j]*v[j];
            esac;
      c[i] = C[i,i];
tel;


-- matuvect : Returns a vector. Input 'a' is an upper triangular matrix.

system matuvect : {M | M>1}
            (a : {i,j | 1<=i<=M; i<=j<=M} of real;
             v : {i | 1<=i<=M} of real)
      returns      (c : {i | 1<=i<=M} of real);
var
      C : {i,j | 1<=i<=M; i-1<=j<=M} of real;
let
      C[i,j] = case
                  {| j=i-1 } : 0[];
                  {| j>i-1 } : C[i,j-1] + a[i,j]*v[j];
             esac;
      c[i] = C[i,M];
tel;


-- Givens : Givens rotations for triangularization
--    Dimension of input C     : MxN
--    Dimension of output W    : NxN
--    Triangularization is done on LxL square array,
-- with L = min(M,N) = N

% VOIR ANNEXE 1
system givens : {M,N |M>1; N<=M}
      (C : {i,j | 1<=i<=M; i<=j<=N} of real)

returns (W : {i,j | 1<=i<=N; i<=j<=N} of real);

let

      W[i,j] = C[i,j];

tel;
```

```
-- --------------------------------------------------
-- kalman : square root covariance kalman filtering
-- --------------------------------------------------

system kalman : {N,M | 1<=N; 1<M}
               (yb   : {n | 1<=n<=N} of real;
            h    : {m,n | 1<=m<=M; 1<=n<=N} of real;
            sigmav2, sigmaw2 :  real)
       returns (xe : {n | 0<=n<=N} of real);
var
  x_hat_p : {m,n | 1<=m<=M; 0<=n<=N} of real;
  x_hat, S_t_h : {m,n | 1<=m<=M; 1<=n<=N} of real;
  A : {m,i,n | 1<=m<=M+1; 1<=i<=M; 1<=n<=N} of real;
  B : {m,i,n | 1<=m<=M+1; 1<=i<=M+1; 1<=n<=N} of real;
  LHS : {m,i,n | 1<=m<=M+1; m<=i<=M+1; 1<=n<=N } of real;
  U,V : real;
--  S_p : {m,i,n | 1<=m<=M; 1<=i<=m; 0<=n<=N} of real;
  S_p_t : {m,i,n | 1<=m<=M; m<=i<=M; 0<=n<=N} of real;
--  S : {m,i,n | 1<=m<=M; 1<=i<=m; 1<=n<=N} of real;
  S_t : {m,i,n | 1<=m<=M; m<=i<=M; 1<=n<=N} of real;
  ye, f: {n | 1<=n<=N} of real;
  g :{m,n | 1<=m<=M; 1<=n<=N} of real;
  Ip : {m,i | 1<=m,i<=M} of real;


let

  U = sqrt(sigmaw2[]);
  V = sqrt(sigmav2[]);

  Ip[m,i] =                                      -- Identity matrix
            case
                  {| m=i} : 1[];
                  {| m<=i-1} | {| m>=i+1} : 0[];
            esac;


--    Time Update

  x_hat[m,n] =                                   -- Equation (4)
                                                 -- phi  = [1  0  0
      case                                       --         1  0  0
            {| m=1 } : x_hat_p[1,n-1];           --         0  1  0]
            {| 1<m<=M } : x_hat_p[m-1,n-1];
      esac;

  A[m,i,n] =                    -- Right Hand Side of equation (5)
      case
            {| i=1; m=1 } : S_p_t[i,m,n-1];
            {| i=1; m>i ; m<=M } : 0[];

            {| 2<=i<=M; m<=i-1; m<=M } : S_p_t[m,i-1,n-1];
            {| 2<=i<=M; m>i-1; m<=M } : 0[];

            {| m=M+1; i=1 } : U[];

            {| m=M+1; i>1 } : 0[];
```

```
    esac;

  use {n|1<=n<=N} givens[M+1,M]                      -- Equation (5)
      (A)
  returns (S_t);

--  use {n|1<=n<=N} transp[M]
--    (S_t)
--  returns (S);

--  use {n|0<=n<=N} transp[M]
--    (S_p_t)
--  returns (S_p);



--    Measurement Update

  use {n|1<=n<=N} matuvect[M]               -- In Equation (6) : S_t*h
      (S_t,h)
  returns (S_t_h);

  B[m,i,n] =                            -- Right hand Side of Eq. (6)
      case
            {| m=1 ; i=1} : V[];
            {| m=1 ; i>1} : 0[];
            {| m>1 ; i=1} : S_t_h[m-1,n];
            {| m>1 ; i>1 ; m<=i} : S_t[m-1,i-1,n];                 -
- S_t is a upper triangular array not defined on the lower part
            {| m>1 ; i>1 ; m>i} : 0[];
      esac;

  use {n|1<=n<=N} givens[M+1,M+1] -- Eq. (6) : Triangularization of
                                 -- B a square array of dimension M+1
      (B)                               -- to produce LHS of the same
      (C)                               -- dimension
  returns (LHS);


  f[n] = LHS[1,1,n];                        -- In Eq. (6)
  g[m,n] = LHS[1,m+1,n];
  S_p_t[m,i,n] =
      case
            {|n=0} : Ip[m,i];
            {|n>=1} : LHS[m+1,i+1,n];       -- S_p_t = S_p transpose
      esac;

  ye[n] = reduce(+, (m,n->n), h.(m,n->m,n) * x_hat.(m,n->m,n))[n];
      -- In Eq. (7) : h*x_hat

  x_hat_p[m,n] =                            -- Eq. (7)
      case
            {| n=0 } : 0[];
            {| 1<=n<=N } : x_hat[m,n] + (g[m,n]/f[n])*(yb[n] - ye[n]);
      esac;
```

```
--    Filtering

  xe[n] =
      case
             {| 0<=n<N } : x_hat_p[M,n];
             {| n=N } : x_hat_p[1,N];
      esac;

tel;
```

Annexes III

Programmes Matlab®

```
% -----------------------------------------------------------------------
-------------
%
% call.m
% General function
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% By :
% Aurelien T. Mozipo
% July 14, 1998
%
%-----------------------------------------------------------------------
--------------

clear;
close all;

global nb_per nb_trial nb_pt dim;
global X_max X_min X_nor;
% global max_f min_f max_g min_g;
global nb_bit;
global nb_bit_min step_bit nb_bit_max;
global sigmav2 sigmaw2;
global h;
global p_float p_nor;
global flag_overflow abs_min abs_max;

nb_bit_min = 8;
nb_bit_max = 32;
step_bit = 2;

dim = 3;
W = 2.9;

nb_pt = 500;
```

```
nb_trial = 1;
nb_per = 4;

p_float(1) = 1.868449253821409e-01;
p_float(2) = 1.561215531933085e+00;
p_float(3) = -1.424927599171505e+00;
p_float(4) = 7.676057100024836e-01;
p_nor = p_float(2);                          % p_float(2)

sigmav2 = 0.1;
sigmaw2 = 1e1;
B = zeros([dim 1]);
B(1) = 1;

fig = 1;

% Initialization of maximum numbers
% X_max = -1e256;
% X_min = 1e256;

% max_f = -1e256;
% min_f = 1e256;
% max_g = -1e256;
% min_g = 1e256;

flag_overflow = 0;

n = [1:dim];
centre = n((length(n)+1)/2);

h = (1 + cos(2*pi*(n - centre)/W))/2;
% figure(fig); fig = fig+1;
% stem(n,h,'filled'); grid;
% title('Reponse impulsionnelle du canal - Cas Invariant');

phi = zeros(dim);
phi(1,1) = 1;
for i=2:dim
      phi(i,i-1) = 1;
end;

%      [1 0 0
%         1 0 0
%         0 1 0];


%%% Cas invariant

% [xe,yb,e,e_ens] =  trial_s(phi,B,h,sigmav2,sigmaw2,nb_pt,nb_trial);

% figure(fig); fig = fig+1;
% plot(xe,'*'); hold on;
% plot(yb,'r*');


% figure(fig); fig = fig+1;
% semilogy(e_ens);
```

```
% figure(fig); fig = fig+1;
% plot(e);
% mean(e)


%%% Signal and noise

[a,v] = entree(nb_pt,sigmav2);
[yb,H] = signal_ns(h,a,v);
SNR = 10*log10(norm(a)^2/norm(v)^2)

% [xe,e,e_ens] =  trial_ns(phi,B,a,yb,v,H);

% figure(fig); fig = fig+1;
% plot(xe,'*'); hold on;
% plot(yb,'r*');

% figure(fig); fig = fig+1;
% plot(e);
% disp(' Norme Erreur / Norme Signal (Sans quant.) = '); mean(e)

% figure(fig); fig = fig+1;
% semilogy(e_ens);

%
************************************************************************
************
%
% Generation of data to transfer to the Alpha program file
%
%
************************************************************************
************

% Transfer2Alpha(yb',H,xe');

%
************************************************************************
************
%
% Simulations for covariance Kalman filter and Square root covariance
Kalman filter
%
%
************************************************************************
************

X_max =  1;        % 44.3;
X_min = -1;        % -31.0;           % -45 pour nb_bit = 20
X_nor = 50;

[xe,xe_q] = simul(phi,B,a,yb,v,H);

X_max = 1; % 1.6300e+03;
X_min = -1; % -658.4937;
X_nor = 2000;

[xe_std,xe_std_q] = simul_std(phi,B,a,yb,v,H);
```

```
% *****************************************************************
%
% Generation of the force file for simulation in QuickHDL
%
% *****************************************************************

force(yb,H,xe);

% **********************
%
% Quantization errors
%
% **********************


[f_e,f_e_q] = f_err(a,xe,xe_q)
[e] = q_err(xe,xe_q)

[f_e_std,f_e_std_q] = f_err(a,xe_std,xe_std_q)
[e_std] = q_err(xe_std,xe_std_q)


axex =[nb_bit_min:step_bit:nb_bit_max];

figure(fig); fig = fig+1;
semilogy(axex,f_e_q);hold on;
semilogy(axex,f_e*ones([1 length(axex)]), '.'); hold on;

semilogy(axex,f_e_std_q, ':');hold on;
semilogy(axex,f_e_std*ones([1 length(axex)]), '--');
xlabel('Nbre of bit');  ylabel('Filtering errors');


figure(fig); fig = fig+1;
semilogy(axex,e); hold on;
semilogy(axex,e_std, ':');
xlabel('Nbre of bit'); ylabel('Quantization errors');

figure(fig); fig = fig+1;
plot(xe, '*');
xlabel('Iterations'); ylabel('Reconstructed sample');

figure(fig); fig = fig+1;
ber_yb = compute_ber(a,yb,dim)
ber = compute_ber(a,xe,dim)
ber_std = compute_ber(a,xe_std,dim)

semilogy(ber);
xlabel('Iterations'); ylabel('BER');
```

```
function ber = compute_ber(a,xe,dim);

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

```
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
% compute_ber.m
% Computation of BER for the  reconstructed  signal
%
% Input :
%     a     : ideal signal
%     xe    : reconstructed signal
% Output :
%     ber   : BER
%
% By Aurelien T. Mozipo
% February 22th, 1999
%
%--------------------------------------------------------------------------
--------------

nb_sample = length(a);

for i=dim+1:nb_sample
      xe_i = xe(1:i);
      xe_seuil = sign(xe_i);
      a_i = a(1:i);
%     comp = a(1:nb_sample-dim) - xe_seuil(dim+1:nb_sample);
      comp = a_i(1:i-dim) - xe_seuil(dim+1:i);
      not_null = find(comp);
%     ber = length(not_null)/(nb_sample-dim)
      ber(i) = length(not_null)/(i-dim);
end;
```

■■

```
function y = conv2bin(x, nb_bit);

% -------------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% conv2bin.m
%
% Conversion of a decimal number (< 1) to a binary number
% Input     x     : number to convert
%           nb_bit      : number of bits of the converted number
%
% Output    y     : result
%
%
% By :
% Aurelien T. Mozipo
% July 14, 1998
%
%--------------------------------------------------------------------------
--------------
```

```
if (x<-1 | x>1)
      disp(' Conversion en Binaire : x doit etre compris entre [-1,1]');
      return;
end;

temp = floor(2^(nb_bit-1) * abs(x));
y = dec2bin(temp,nb_bit-1);

if x<0

      for i=1:nb_bit-1
            if y(i) == '1'
                  y(i) = '0';
            else
                  y(i) = '1';
            end
      end
      y = dec2bin(bin2dec(y) + 1, nb_bit-1);
      y = strcat('1', y);
else
      y = strcat('0', y);
end
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

function y = conv2dec(x);

% ---------------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
% conv2dec.m
%
% Conversion of a binary number (< 1) to a decimal number
% Input      x      : binary number to convert
%            x is in 2'complement mode
%
% Output     y      : result
%
%
% By :
%
% Aurelien T. Mozipo
% July 14, 1998
%
%----------------------------------------------------------------------------
--------------

nb_bit = length(x);
sign = x(1);

temp = x(2:nb_bit);

if sign == '1'
      temp = dec2bin(bin2dec(temp) - 1, nb_bit-1);
      for i=1:nb_bit-1
            if temp(i) == '1'
```

```
                temp(i) = '0';
            else
                temp(i) = '1';
            end
      end
end

y = 0;
for i=1:nb_bit-1
      y = y + str2num(temp(i))*(2^-i);
end;

if sign == '1'
      y = -y;
end;
```

```
%  ***************************
%  º   PROGRAMME FAIT PAR     º
%  º   DANIEL MASSICOTTE      º
%  º   LE 23 octobre 1991
% ****************************
%  Modifie Par
%     Aurelien Mozipo
%       Le 26 Avril 1998
%
% ******************************************************
%
% Dernier Modification :
% Quantification des donnees et operations.
%
%  Le 16 Juil. 1998
%  Par Aurelien T. Mozipo
%
%  ******************************************************
%
%  ****************************
%  ³  RECONTITUTION DE          ³
%  ³  MESURANDE A L'AIDE        ³
%  ³  DU FILTRE DE KALMAN       ³
%  ****************************
%
%            AVEC QUANTIFICATION DES OPERATIONS ET DONNEES
%
%  Fonction pour le filtre de Kalman
%
% [xr] = cov_std_q(yb,H,C,Beta);
%

function [xr,K] = cov_std_q(yb,H,C,Beta);

global X_nor;

retard=1;
ordre = 1;
Nb_pt_y =length(yb);
Nb_pt_x = Nb_pt_y;
Nb_pt_h = length(H(:,1));
```

```
%
%  Definition du systeme:
%                          x(k+1) = F x(k) + b w(k)
%                          y(k) = H' x(k) + v(k)
%
F(2:Nb_pt_h,1:Nb_pt_h-1) = eye(Nb_pt_h-1);
F(1:Nb_pt_h,Nb_pt_h) = zeros(Nb_pt_h,1);
F(1,1) = ordre;
F = q( F );

B=zeros(Nb_pt_h,1);
B(1)=1;
B = q( B );


%
%  Conditions initiales
%
z = zeros(Nb_pt_h,1);

P = q( eye(Nb_pt_h)/X_nor );
Ip = q( eye(Nb_pt_h)/X_nor );
Beta = q( Beta/X_nor );
yb = q(yb/X_nor);

%
%  Reconstitution de l'entree ...a l'aide du filtre de Kalman
%
%

for k=1:Nb_pt_y

      if length(H(1,:))>1
            h=q( H(:,k) );                % H(:,k) correspond a H a
l'instant k
      else
            h=q( H );
      end

%    F_z = [ z(1) ; z(1:Nb_pt_h-1) ];

     F_z = matvect(F,z);

%    ye(k) = H'*F*z;
     ye(k) = vectvect(h',F_z);

   i(k) = q( yb(k) - ye(k) );

%
% Calcul du gain de Kalman
%


     P = q( matmult(F,matmult(P,F')) + q( B*q( Beta*B' ) ) );
     P_h = matvect(P,h);

% PP = P
```

```
    hPh_1 = vectvect(H(:,k)',matvect(P,H(:,k)));   % H(:,k) correspond a
H a l'instant k

    K(:,k) = q( P_h/(hPh_1 + q( 1/X_nor ) ) );
    K(:,k) = K(:,k)/X_nor;

    P = q( matmult(q( Ip - q( K(:,k)*h' ) ),P)/X_nor );

% P

%   z = F*z + K*i(k);
    z = q( q( F_z*X_nor ) + q( K(:,k)*i(k) ) );

    z = q( z/X_nor );

   x_fkal(k) = z(Nb_pt_h);

end

x_fkal = x_fkal';
i = i';
ye = ye';

xr = x_fkal*X_nor;
```

```
function quot = division(num, den);

% ------------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% division.m
%
% Division of two numbers with the Newton Iterative method
%
%
% Input     num : numerator
%           den : denominator
%
% Output    quot : quotient
%
%
% By :
% Aurelien T. Mozipo
% July 14, 1998
%
%------------------------------------------------------------------------
--------------
```

```
% Reduction of den so that 1<=den<2
a = 1;
signe = sign(den);
den = abs(den);

while ~(den >= 1 & den<2)
      a = 2*a;
      den = 2*den;
end;

% Seeking an approche value of x0 = 1/den

i = 0;
x(i+1) = table_div(den);

% Iterations

e = 1;
nb_bit = 2;
inv_float = 1/den;

% while e >= 1e-4
for i = 1:3
      %i = i+1;
      nb_bit = 2*nb_bit;
      x(i+1) = quant(x(i)*(quant(2 - quant(x(i)*den, nb_bit), nb_bit)),
nb_bit);
      % e = abs((x(i+1) - x(i))/x(i));
      e = abs((x(i+1) - inv_float)/inv_float);

end;

% Inverse reduction

if signe == 0
      disp('Division par zero !');
else
      quot = signe*quant(a*quant(num*x(i+1), nb_bit), nb_bit);
end;
 erreur = e;
 nb_iterations = i;
 nb_bit = nb_bit;function [a,v] = entree(nb_pt,sigmav2);
```

■■

```
% ------------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% entree.m
%
%  Generation of the random sequence {a(n)} = {-1,+1}
%  and the noise v
%
```

```
% Input       : Nb of iterations
%             : sigmav2 : noise variance
%
% Output      a : test signal
%             v : noise
%
% By :
% Aurelien T. Mozipo
% July 14, 1998
%
%----------------------------------------------------------------------
--------------

a = round(rand([1 nb_pt]));
for n = 1:nb_pt
      if a(n) == 0
            a(n) = -1;
      end
end

%%%%%%%%% Generation of {v(n)}

v = randn([1 nb_pt]);
v = v*sigmav2/(std(v)^2);
sigmav2
norm(v)function sortie = f(p,x);

% ---------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% f.m
%
% Approximation of the square root by a third order polynomial
%
%
% Input       p      : polyne gave by the function "curvefit"
%             x      : value to compute
%
% Output      sortie       : result = sqrt(x)
%
%
% By :
% Aurelien T. Mozipo
% July 14, 1998
%
%----------------------------------------------------------------------
--------------

sortie = p(1)+p(2)*x+p(3)*x.^2 + p(4)*x.^3;


% Syntaxe de la function CURVEFIT
% p = curvefit('f',[1 1 1],x,y)
```

```
function force(yb,H,xr);

% -----------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% force.m
% Generation du fichier des forces de yb, H, xr pour transfert a QUICKHDL
% pour simulation
%
% Inputs :  yb    : input signal (with noise)
%            H    : time varying impulse response
%           xr    : reconstructed signal
%
% By :
%
%******************************************
%     Aurelien T. Mozipo
%     Fev., 4, 1999
%
%******************************************
%
%-----------------------------------------------------------------------
------------
global X_nor;

nb_pt = length(yb);
dim = length(H(1,:));
nb_bit = 20;

yb_nor = yb/X_nor;
xr_nor = xr/X_nor;


%*********************************************************************
********
%%%    Generation du fichier unique "results.hdt" pour Kalman non
stationnaire
%*********************************************************************
********

disp('Patienter. creation de fichier pour Kalman non Stationnaire en
cours... ');



inpns(:,1) = yb_nor';

for i=1:dim
      inpns(:,i+1) = H(:,i);
end
```

```
dlmwrite('force_dec.hdt',inpns,' ');

disp('Fichier de donnees "data_sqrt.hdt" cree. Ordre de lecture :');
disp(' 1ere colonne : yb_nor; 2eme col.: H(1); 3eme col. : H(2); 4eme
col.: H(3).');

dlmwrite('xr.hdt',xr_nor',' ');


disp('Fichier de resultats "xr.hdt" cree.');

% Création du fichier résultats en binaire

fid1 = fopen('xr.bin', 'wt+');
for i=1:nb_pt
   xr_bin = conv2bin(xr_nor(i), nb_bit);
   x = strcat(xr_bin,'\n');
      fprintf(fid1, x);
end;


%%% *******************************************************************
%                                                                     *
%  Creation du force file covkala.stim pour QUICKHDL
*
%                                                                     *
%%% *******************************************************************

disp('Creation du force file en cours... ');

fid = fopen('/u/hping/mozipo/memoire/covkal/design/covkala.stim', 'wt+');

for i=1:nb_pt

   % state0

      if i==1
            fprintf(fid, 'forc rst ''1''\n');
      else
            fprintf(fid, 'forc rst ''0''\n');
      end;
      fprintf(fid, 'forc initpe22 ''1''\n');
      fprintf(fid, 'forc modepe11 "000"\n');
      fprintf(fid, 'forc modepe12 "001"\n');
      if i==1
            fprintf(fid, 'forc modepe13 "000"\n');
            fprintf(fid, 'forc modepe14 "000"\n');
      else
            fprintf(fid, 'forc modepe13 "110"\n');
            fprintf(fid, 'forc modepe14 "101"\n');
      end;
      fprintf(fid, 'forc modepe22 "101"\n');
      fprintf(fid, 'forc modepe23 "011"\n');
      fprintf(fid, 'forc modepe24 "000"\n');
      fprintf(fid, 'forc modepe33 "000"\n');
      if i==1
            fprintf(fid, 'forc modepe34 "000"\n');
      else
```

```
        fprintf(fid, 'forc modepe34 "001"\n');
    end;
    fprintf(fid, 'forc modepe44 "000"\n');
    fprintf(fid, 'forc datain "WWWWWWWWWWWWWWWWWWWWWW"\n');
    fprintf(fid, 'forc clk ''1''\n');
fprintf(fid, 'run 5\n\n');

fprintf(fid, 'forc clk ''0''\n');
fprintf(fid, 'run 5\n\n');

% state1

    fprintf(fid, 'forc rst ''0''\n');
    fprintf(fid, 'forc initpe22 ''0''\n');
    fprintf(fid, 'forc modepe11 "000"\n');
    fprintf(fid, 'forc modepe12 "001"\n');
fprintf(fid, 'forc modepe13 "001"\n');
if i==1
        fprintf(fid, 'forc modepe14 "000"\n')
    else
        fprintf(fid, 'forc modepe14 "110"\n');
end;
fprintf(fid, 'forc modepe22 "001"\n');
    fprintf(fid, 'forc modepe23 "000"\n');
    fprintf(fid, 'forc modepe24 "101"\n');
fprintf(fid, 'forc modepe33 "000"\n');
fprintf(fid, 'forc modepe34 "000"\n');
    if i==1
        fprintf(fid, 'forc modepe44 "000"\n');
    else
        fprintf(fid, 'forc modepe44 "001"\n');
    end
    fprintf(fid, 'forc datain "WWWWWWWWWWWWWWWWWWWWWW"\n');
    fprintf(fid, 'forc clk ''1''\n');
fprintf(fid, 'run 5\n\n');

fprintf(fid, 'forc clk ''0''\n');
fprintf(fid, 'run 5\n\n');

% state2

fprintf(fid, 'forc rst ''0''\n');
    fprintf(fid, 'forc initpe22 ''0''\n');
    fprintf(fid, 'forc modepe11 "000"\n');
    fprintf(fid, 'forc modepe12 "001"\n');
fprintf(fid, 'forc modepe13 "001"\n');
    fprintf(fid, 'forc modepe14 "001"\n');
fprintf(fid, 'forc modepe22 "001"\n');
    fprintf(fid, 'forc modepe23 "001"\n');
    fprintf(fid, 'forc modepe24 "000"\n');
fprintf(fid, 'forc modepe33 "101"\n');
fprintf(fid, 'forc modepe34 "000"\n');
fprintf(fid, 'forc modepe44 "000"\n');
datain = conv2bin(yb_nor(i), nb_bit);
t = strcat('forc datain ',' "',datain,'"\n');
    fprintf(fid, t);
    fprintf(fid, 'forc clk ''1''\n');
fprintf(fid, 'run 5\n\n');
```

```
   fprintf(fid, 'forc clk ''0''\n');
fprintf(fid, 'run 5\n\n');

% state3

fprintf(fid, 'forc rst ''0''\n');
   fprintf(fid, 'forc initpe22 ''0''\n');
   fprintf(fid, 'forc modepe11 "010"\n');
   fprintf(fid, 'forc modepe12 "001"\n');
   fprintf(fid, 'forc modepe13 "001"\n');
   fprintf(fid, 'forc modepe14 "001"\n');
   fprintf(fid, 'forc modepe22 "001"\n');
   fprintf(fid, 'forc modepe23 "001"\n');
   fprintf(fid, 'forc modepe24 "001"\n');
   fprintf(fid, 'forc modepe33 "000"\n');
   fprintf(fid, 'forc modepe34 "000"\n');
fprintf(fid, 'forc modepe44 "000"\n');
datain = conv2bin(H(i,1), nb_bit);
t = strcat('forc datain ',' "',datain,'"\n');
   fprintf(fid, t);
   fprintf(fid, 'forc clk ''1''\n');
   fprintf(fid, 'run 5\n\n');

   fprintf(fid, 'forc clk ''0''\n');
fprintf(fid, 'run 5\n\n');

% state4

fprintf(fid, 'forc rst ''0''\n');
   fprintf(fid, 'forc initpe22 ''0''\n');
   fprintf(fid, 'forc modepe11 "000"\n');
   fprintf(fid, 'forc modepe12 "010"\n');
   fprintf(fid, 'forc modepe13 "001"\n');
   fprintf(fid, 'forc modepe14 "001"\n');
   fprintf(fid, 'forc modepe22 "001"\n');
   fprintf(fid, 'forc modepe23 "001"\n');
   fprintf(fid, 'forc modepe24 "001"\n');
   fprintf(fid, 'forc modepe33 "001"\n');
   fprintf(fid, 'forc modepe34 "000"\n');
fprintf(fid, 'forc modepe44 "000"\n');
datain = conv2bin(H(i,2), nb_bit);
t = strcat('forc datain ',' "',datain,'"\n');
   fprintf(fid, t);
   fprintf(fid, 'forc clk ''1''\n');
   fprintf(fid, 'run 5\n\n');

   fprintf(fid, 'forc clk ''0''\n');
   fprintf(fid, 'run 5\n\n');

% state5

   fprintf(fid, 'forc rst ''0''\n');
   fprintf(fid, 'forc initpe22 ''0''\n');
   fprintf(fid, 'forc modepe11 "000"\n');
   fprintf(fid, 'forc modepe12 "000"\n');
   fprintf(fid, 'forc modepe13 "010"\n');
   fprintf(fid, 'forc modepe14 "001"\n');
```

```
   fprintf(fid, 'forc modepe22 "010"\n');
   fprintf(fid, 'forc modepe23 "001"\n');
   fprintf(fid, 'forc modepe24 "001"\n');
   fprintf(fid, 'forc modepe33 "001"\n');
   fprintf(fid, 'forc modepe34 "001"\n');
fprintf(fid, 'forc modepe44 "000"\n');
datain = conv2bin(H(i,3), nb_bit);
t = strcat('forc datain ',' "',datain,'"\n');
   fprintf(fid, t);
   fprintf(fid, 'forc clk ''1''\n');
   fprintf(fid, 'run 5\n\n');

   fprintf(fid, 'forc clk ''0''\n');
   fprintf(fid, 'run 5\n\n');

% state6

   fprintf(fid, 'forc rst ''0''\n');
   fprintf(fid, 'forc initpe22 ''0''\n');
   fprintf(fid, 'forc modepe11 "000"\n');
   fprintf(fid, 'forc modepe12 "000"\n');
   fprintf(fid, 'forc modepe13 "000"\n');
   fprintf(fid, 'forc modepe14 "010"\n');
   fprintf(fid, 'forc modepe22 "000"\n');
   fprintf(fid, 'forc modepe23 "010"\n');
   fprintf(fid, 'forc modepe24 "001"\n');
   fprintf(fid, 'forc modepe33 "001"\n');
   fprintf(fid, 'forc modepe34 "001"\n');
   fprintf(fid, 'forc modepe44 "000"\n');
   fprintf(fid, 'forc datain "WWWWWWWWWWWWWWWWWWWWWW"\n');
   fprintf(fid, 'forc clk ''1''\n');
   fprintf(fid, 'run 5\n\n');

   fprintf(fid, 'forc clk ''0''\n');
   fprintf(fid, 'run 5\n\n');

% state7

   fprintf(fid, 'forc rst ''0''\n');
   fprintf(fid, 'forc initpe11 ''1''\n');
   fprintf(fid, 'forc initpe22 ''0''\n');
   fprintf(fid, 'forc modepe11 "001"\n');
   fprintf(fid, 'forc modepe12 "000"\n');
   fprintf(fid, 'forc modepe13 "000"\n');
   fprintf(fid, 'forc modepe14 "000"\n');
   fprintf(fid, 'forc modepe22 "000"\n');
   fprintf(fid, 'forc modepe23 "000"\n');
   fprintf(fid, 'forc modepe24 "010"\n');
   fprintf(fid, 'forc modepe33 "010"\n');
   fprintf(fid, 'forc modepe34 "001"\n');
   fprintf(fid, 'forc modepe44 "001"\n');
   fprintf(fid, 'forc clk ''1''\n');
   fprintf(fid, 'run 5\n\n');

   fprintf(fid, 'forc clk ''0''\n');
   fprintf(fid, 'run 5\n\n');

% state8
```

```
    fprintf(fid, 'forc rst ''0''\n');
    fprintf(fid, 'forc initpe11 ''0''\n');
    fprintf(fid, 'forc initpe22 ''0''\n');
    fprintf(fid, 'forc modepe11 "001"\n');
    fprintf(fid, 'forc modepe12 "011"\n');
    fprintf(fid, 'forc modepe13 "000"\n');
    fprintf(fid, 'forc modepe14 "000"\n');
    fprintf(fid, 'forc modepe22 "101"\n');
    fprintf(fid, 'forc modepe23 "011"\n');
    fprintf(fid, 'forc modepe24 "000"\n');
    fprintf(fid, 'forc modepe33 "000"\n');
    fprintf(fid, 'forc modepe34 "010"\n');
    fprintf(fid, 'forc modepe44 "001"\n');
    fprintf(fid, 'forc clk ''1''\n');
    fprintf(fid, 'run 5\n\n');

    fprintf(fid, 'forc clk ''0''\n');
    fprintf(fid, 'run 5\n\n');

% state9

    fprintf(fid, 'forc rst ''0''\n');
    fprintf(fid, 'forc initpe11 ''0''\n');
    fprintf(fid, 'forc initpe22 ''0''\n');
    fprintf(fid, 'forc modepe11 "001"\n');
    fprintf(fid, 'forc modepe12 "011"\n');
    fprintf(fid, 'forc modepe13 "011"\n');
    fprintf(fid, 'forc modepe14 "000"\n');
    fprintf(fid, 'forc modepe22 "000"\n');
    fprintf(fid, 'forc modepe23 "000"\n');
    fprintf(fid, 'forc modepe24 "101"\n');
    fprintf(fid, 'forc modepe33 "000"\n');
    fprintf(fid, 'forc modepe34 "000"\n');
    fprintf(fid, 'forc modepe44 "010"\n');
    fprintf(fid, 'forc clk ''1''\n');
    fprintf(fid, 'run 5\n\n');

    fprintf(fid, 'forc clk ''0''\n');
    fprintf(fid, 'run 5\n\n');

% state10

    fprintf(fid, 'forc rst ''0''\n');
    fprintf(fid, 'forc initpe11 ''0''\n');
    fprintf(fid, 'forc initpe22 ''0''\n');
    fprintf(fid, 'forc modepe11 "001"\n');
    fprintf(fid, 'forc modepe12 "011"\n');
    fprintf(fid, 'forc modepe13 "011"\n');
    fprintf(fid, 'forc modepe14 "011"\n');
    fprintf(fid, 'forc modepe22 "001"\n');
    fprintf(fid, 'forc modepe23 "000"\n');
    fprintf(fid, 'forc modepe24 "011"\n');
    fprintf(fid, 'forc modepe33 "101"\n');
    fprintf(fid, 'forc modepe34 "000"\n');
    fprintf(fid, 'forc modepe44 "000"\n');
    fprintf(fid, 'forc clk ''1''\n');
    fprintf(fid, 'run 5\n\n');
```

```
    fprintf(fid, 'forc clk ''0''\n');
    fprintf(fid, 'run 5\n\n');

% state11

    fprintf(fid, 'forc rst ''0''\n');
    fprintf(fid, 'forc initpe11 ''0''\n');
    fprintf(fid, 'forc initpe22 ''0''\n');
    fprintf(fid, 'forc modepe11 "101"\n');
    fprintf(fid, 'forc modepe12 "011"\n');
    fprintf(fid, 'forc modepe13 "011"\n');
    fprintf(fid, 'forc modepe14 "011"\n');
    fprintf(fid, 'forc modepe22 "001"\n');
    fprintf(fid, 'forc modepe23 "001"\n');
    fprintf(fid, 'forc modepe24 "000"\n');
    fprintf(fid, 'forc modepe33 "000"\n');
    fprintf(fid, 'forc modepe34 "011"\n');
    fprintf(fid, 'forc modepe44 "000"\n');
    fprintf(fid, 'forc clk ''1''\n');
    fprintf(fid, 'run 5\n\n');

    fprintf(fid, 'forc clk ''0''\n');
    fprintf(fid, 'run 5\n\n');

% state12

    fprintf(fid, 'forc rst ''0''\n');
    fprintf(fid, 'forc initpe11 ''0''\n');
    fprintf(fid, 'forc initpe22 ''0''\n');
    fprintf(fid, 'forc modepe11 "000"\n');
    fprintf(fid, 'forc modepe12 "100"\n');
    fprintf(fid, 'forc modepe13 "011"\n');
    fprintf(fid, 'forc modepe14 "011"\n');
    fprintf(fid, 'forc modepe22 "001"\n');
    fprintf(fid, 'forc modepe23 "001"\n');
    fprintf(fid, 'forc modepe24 "001"\n');
    fprintf(fid, 'forc modepe33 "000"\n');
    fprintf(fid, 'forc modepe34 "000"\n');
    fprintf(fid, 'forc modepe44 "011"\n');
    fprintf(fid, 'forc clk ''1''\n');
    fprintf(fid, 'run 5\n\n');

    fprintf(fid, 'forc clk ''0''\n');
    fprintf(fid, 'run 5\n\n');

% state13

    fprintf(fid, 'forc rst ''0''\n');
    fprintf(fid, 'forc initpe11 ''0''\n');
    fprintf(fid, 'forc initpe22 ''0''\n');
    fprintf(fid, 'forc modepe11 "000"\n');
    fprintf(fid, 'forc modepe12 "101"\n');
    fprintf(fid, 'forc modepe13 "100"\n');
    fprintf(fid, 'forc modepe14 "011"\n');
    fprintf(fid, 'forc modepe22 "011"\n');
    fprintf(fid, 'forc modepe23 "001"\n');
    fprintf(fid, 'forc modepe24 "001"\n');
```

```
        fprintf(fid, 'forc modepe33 "001"\n');
        fprintf(fid, 'forc modepe34 "000"\n');
        fprintf(fid, 'forc modepe44 "000"\n');
        fprintf(fid, 'forc clk ''1''\n');
        fprintf(fid, 'run 5\n\n');

        fprintf(fid, 'forc clk ''0''\n');
        fprintf(fid, 'run 5\n\n');

    % state14

        fprintf(fid, 'forc rst ''0''\n');
        fprintf(fid, 'forc initpe11 ''0''\n');
        fprintf(fid, 'forc initpe22 ''0''\n');
        fprintf(fid, 'forc modepe11 "000"\n');
        fprintf(fid, 'forc modepe12 "111"\n');
        fprintf(fid, 'forc modepe13 "101"\n');
        fprintf(fid, 'forc modepe14 "100"\n');
        fprintf(fid, 'forc modepe22 "101"\n');
        fprintf(fid, 'forc modepe23 "000"\n');
        fprintf(fid, 'forc modepe24 "001"\n');
        fprintf(fid, 'forc modepe33 "001"\n');
        fprintf(fid, 'forc modepe34 "001"\n');
        fprintf(fid, 'forc modepe44 "000"\n');
        fprintf(fid, 'forc clk ''1''\n');
        fprintf(fid, 'run 5\n\n');

        fprintf(fid, 'forc clk ''0''\n');
        fprintf(fid, 'run 5\n\n');
end;

% state0

fprintf(fid, '-- Sortie du dernier échantillon\n\n');

fprintf(fid, 'forc rst ''0''\n');
fprintf(fid, 'forc initpe22 ''1''\n');
fprintf(fid, 'forc modepe11 "000"\n');
fprintf(fid, 'forc modepe12 "001"\n');
fprintf(fid, 'forc modepe13 "110"\n');
fprintf(fid, 'forc modepe14 "101"\n');
fprintf(fid, 'forc modepe22 "000"\n');
fprintf(fid, 'forc modepe23 "011"\n');
fprintf(fid, 'forc modepe24 "000"\n');
fprintf(fid, 'forc modepe33 "000"\n');
fprintf(fid, 'forc modepe34 "001"\n');
fprintf(fid, 'forc modepe44 "000"\n');
fprintf(fid, 'forc datain "WWWWWWWWWWWWWWWWWWWWWW"\n');
fprintf(fid, 'forc clk ''1''\n');
fprintf(fid, 'run 5\n\n');

fprintf(fid, 'forc clk ''0''\n');
fprintf(fid, 'run 5\n\n');

% state1

fprintf(fid, 'forc rst ''0''\n');
fprintf(fid, 'forc initpe22 ''0''\n');
```

```
fprintf(fid, 'forc modepe11 "000"\n');
fprintf(fid, 'forc modepe12 "001"\n');
fprintf(fid, 'forc modepe13 "001"\n');
fprintf(fid, 'forc modepe14 "110"\n');
fprintf(fid, 'forc modepe22 "001"\n');
fprintf(fid, 'forc modepe23 "000"\n');
fprintf(fid, 'forc modepe24 "101"\n');
fprintf(fid, 'forc modepe33 "000"\n');
fprintf(fid, 'forc modepe34 "000"\n');
fprintf(fid, 'forc modepe44 "001"\n');
fprintf(fid, 'forc datain "WWWWWWWWWWWWWWWWWWWWWW"\n');
fprintf(fid, 'forc clk ''1''\n');
fprintf(fid, 'run 5\n\n');

fprintf(fid, 'forc clk ''0''\n');
fprintf(fid, 'run 5\n\n');

fclose('all');
```

```
function [e,e_q] = f_err(a,xe,xe_q);

% ---------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% f_err.m
% Computation of filtering errors
%
% Inputs     a     : channel test signal
%            xe    : estimated signalcomputed in floating point (filtered
value)
%            xe_q  : Estimated signal computed with quantization
%
% Output     e     : Filtering error with a floating point algorithm (xe)
%            e_q   : Filtering error withquantization (xe_q)
%
% Aurelien T. Mozipo
% July, 16, 1998
%
%
%---------------------------------------------------------------------
--------------

global nb_pt ;
global dim;
global nb_bit_min nb_bit_max step_bit;
dec = 1;


% ------------------------------------------
% Filtering errors : without quantization
% ------------------------------------------
      % err = a(1:nb_pt-dim)-xe(1+dim:nb_pt);
      e = norm(a(1:nb_pt-dim)-xe(1+dim:nb_pt))/norm(a(dec:nb_pt-dim));
```

```
i = 0;
for n = nb_bit_min:step_bit:nb_bit_max
      i = i+1;
      e_q(i) = norm(a(1:nb_pt-dim)-xe_q(i,1+dim:nb_pt))/norm(a(dec:nb_pt-
dim));


end;

return;
```

■■

```
% -----------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% gen_diag_cell.m
%     Diagonal cells behavior
%     Generation of a rotation
%     Rotation of (r,ain) to nullify ain
%
% Inputs    ain   : input data
%           r     : value of the internal register
%
% Outputs   out = [c,s,rout] with
%           c     : cosine
%           s     : sine
%           rout  : internal register new value
%
%
% By :
% Aurelien T. Mozipo
% July 14, 1998
%
%------------------------------------------------------------------------
--------------

if ain == 0
      c = 1;
      s = 0;
else

      if abs(ain)>=abs(r)
            t = r/ain;
            % s_int = sqrt(1+t*t)
            s = 1/sqrt(1+t*t);
            c = s*t;
      else
            t = ain/r;
            % c_int = sqrt(1+t*t)
            c = 1/sqrt(1+t*t);
            s = c*t;
      end
```

```
end

rout = c*r + s*ain;

out = [c,s,rout];
```

```
function out = gen_diag_cell_q(ain,r);

% -----------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% gen_diag_cell_q.m
%     Diagonal cells behavior
%     With quantization
%     Generation of a rotation
%     Rotation of (r,ain) to nullify ain
%
% Inputs    ain   : input data
%           r     : value of the internal register
%
% Outputs   out = [c,s,rout] with
%           c     : cosine
%           s     : sine
%           rout  : internal register new value
%
%
% By :
% Aurelien T. Mozipo
% July 14, 1998
%
%-----------------------------------------------------------------------
----------------

% global p;
% global X_max;


if ain == 0
      c = q( 1 );
      s = 0;
else
      if abs(ain)>=abs(r)
            t = division( r,ain );
            s = division( 0.5,sqrt1(0.25 + q( 0.25*q( t*t )) ));
            c = q( s*t );
      else
            t = division( ain,r );
            c = division( 0.5,sqrt1(0.25 + q( 0.25*q( t*t )) ));
            s = q( c*t );
      end
end

rout = q( q( c*r ) + q( s*ain ) );
```

```
out = [c,s,rout];
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

function out = gen_of_diag_cell(ain,r,c,s);

% -----------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% gen_of_diag_cell.m
%      Of Diagonal cells behavior
%     Without quantization
%
%
% Inputs    ain   : input data
%           r     : value of the internal register
%           c     : cosine
%           s     : sine
%
% Outputs   out = [c,s,rout,aout] with
%           c     : cosine
%           s     : sine
%           rout  : internal register new value
%           aout  : output data value
%
%
% By :
% Aurelien T. Mozipo
% July 14, 1998
%
%-----------------------------------------------------------------------
----------------

% Application of a rotation

% temp = r;
rout = c*r + s*ain;
aout = -s*r + c*ain;

out = [c,s,rout,aout];

■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

function out = gen_of_diag_cell_q(ain,r,c,s);
% -----------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% gen_of_diag_cell_q.m
%      Of Diagonal cells behavior
%      With quantization
```

```
%
%
% Inputs    ain   : input data
%           r     : value of the internal register
%           c     : cosine
%           s     : sine
%
% Outputs   out = [c,s,rout,aout] with
%           c     : cosine
%           s     : sine
%           rout  : internal register new value
%           aout  : output data value
%
%
% By :
% Aurelien T. Mozipo
% July 14, 1998
%
%-----------------------------------------------------------------------
---------------

% temp = r;
rout = q( q( c*r ) + q( s*ain ));
aout = q( q( -s*r ) + q( c*ain ));

out = [c,s,rout,aout];
```

```
function W = givens(A);

% ----------------------------------------------------------------------
------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% givens.m
% Matrix triangularization by Givens rotations
% Find : an orthogonal transformation T such that
%
%        TA = [W
%              0]
%
% Input :   A : matrix of dimension m x n
% Output    W : upper triangular matrix dimensoin m x n
%
% Aurelien T. Mozipo
% July 27, 1998
%
%-----------------------------------------------------------------------
-------------

m = length(A(:,1));
n = length(A(1,:));

mdim = min(m,n);
```

```
%     Inputs

aout_1(1,1) = 0;

c(1,1) = 0;
s(1,1) = 0;
r(1,1) = A(1,1);
aout = [];

% PE initialization times

for i=1:m

      for j=i:n
            if i==1
                  t_init(1,j) = j;
            else
                  t_init(i,j) = t_init(i-1,j)+2;
            end
      end
end

nb_step = 2*min(m,n) + max(m,n) -1;

for j=1:n
      input(j:j+m-1,j) = A(:,j);
end

for step = 1:nb_step

      for i=1:mdim
            for j=i:mdim

                  % active(i,j) = (t_init(i,j)<=step &
step<=t_init(i,j)+m-1);
                  active(i,j) = (t_init(i,j)<=step & step<=t_init(i,j)+m-
i);

                  init(i,j) = not(step>t_init(i,j));

                  if active(i,j)==1
                        if i==1
                              ain(i,j) =  input(step,j);

                        else
                              ain(i,j) = aout_1(i-1,j);
                        end

                        if i==j
                              if step == t_init(i,j)
                                    r(i,j) = ain(i,j);
                                    init(i,j) = 0;
                              else
                                    out =
gen_diag_cell(ain(i,j),r_1(i,j));

                                    r(i,j) = out(3);
                                    c(i,j) = out(1);
```

```
                                        s(i,j) = out(2);
                                end
                        else
                                if step == t_init(i,j)
                                        r(i,j) = ain(i,j);
                                        init(i,j) = 0;
                                else
                                        out =
gen_of_diag_cell(ain(i,j),r_1(i,j),c_1(i,j-1),s_1(i,j-1));

                                        r(i,j) = out(3);
                                        c(i,j) = out(1);
                                        s(i,j) = out(2);
                                        aout(i,j) = out(4);

                                end
                        end
                end
        end
    end

    aout_1 = aout;
    c_1 = c;
    s_1 = s;
    r_1 = r;
    % pause;
end

W = r;
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

function W = givens_q(A);

% -------------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% givens_q.m
% Matrix triangularization by Givens rotations
% With quantization
% Find : an orthogonal transformation T such that
%
%         TA = [W
%               0]
%
% Input :    A : matrix of dimension m x n
% Output     W : upper triangular matrix dimensoin m x n
%
% Aurelien T. Mozipo
% July 27, 1998
%
%-------------------------------------------------------------------------
--------------

m = length(A(:,1));
```

```
n = length(A(1,:));

mdim = min(m,n);

%       Inputs

aout_1(1,1) = 0;

c(1,1) = 0;
s(1,1) = 0;
r(1,1) = A(1,1);
aout = [];

% PE initialization times

for i=1:mdim

      for j=i:mdim
            if i==1
                  t_init(1,j) = j;
            else
                  t_init(i,j) = t_init(i-1,j)+2;
            end
      end
end

nb_step = 2*min(m,n) + max(m,n) -1;

for j=1:n
      input(j:j+m-1,j) = A(:,j);
end

for step = 1:nb_step*2

      for i=1:mdim
            for j=i:mdim

                  % active(i,j) = (t_init(i,j)<=step &
step<=t_init(i,j)+m-1);
                  active(i,j) = (t_init(i,j)<=step & step<=t_init(i,j)+m-
i);

                  init(i,j) = not(step>t_init(i,j));

                  if active(i,j)==1
                        if i==1

                              ain(i,j) =  input(step,j);

                        else
                              ain(i,j) = aout_1(i-1,j);
                        end

                        if i==j
                              if step == t_init(i,j)
                                    r(i,j) = ain(i,j);
                                    init(i,j) = 0;
                              else
```

```
                                    out =
gen_diag_cell_q(ain(i,j),r_1(i,j));

                                    r(i,j) = out(3);
                                    c(i,j) = out(1);
                                    s(i,j) = out(2);
                              end
                        else
                              if step == t_init(i,j)
                                    r(i,j) = ain(i,j);
                                    init(i,j) = 0;
                              else
                                    out =
gen_of_diag_cell_q(ain(i,j),r_1(i,j),c_1(i,j-1),s_1(i,j-1));

                                    r(i,j) = out(3);
                                    c(i,j) = out(1);
                                    s(i,j) = out(2);
                                    aout(i,j) = out(4);

                              end
                        end
                  end
            end
      end

      aout_1 = aout;
      c_1 = c;
      s_1 = s;
      r_1 = r;
      % pause;
end

W = r;
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

function quot = inversion(den);

% ----------------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% inversion.m
%
% Inversion of a number by the Newton-Raphson iteration method
%
%
% Input     num : numerator
%
%
% Output    quot : quotient
%
%
% By :
% Aurelien T. Mozipo
```

```
% July 14, 1998
%
%-----------------------------------------------------------------------
--------------

signe = sign(den);
den = abs(den);

% Seeking an approche value of x0 = 1/den

i = 0;
x(i+1) = table_div(den);

% Iterations

e = 1;
nb_bit = 2;
inv_float = 1/den;

while e >= 1e-4
      i = i+1;
      nb_bit = 2*nb_bit
      x(i+1) = quant(x(i)*(quant(2 - quant(x(i)*den, 2*nb_bit),
2*nb_bit)), 2*nb_bit)
      % e = abs((x(i+1) - x(i))/x(i));
      e = abs((x(i+1) - inv_float)/inv_float);

end;


if signe == 0
      disp('Division par zero !');
else
      quot = signe*x(i+1);
end;
 erreur = e
 nb_iterations = i
 nb_bit = nb_bit%
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

%  **************************
%  º   PROGRAMME FAIT PAR    º
%  º   DANIEL MASSICOTTE     º
%  º   LE 23 octobre 1991
%  Modifie par Mozipo le   º
%  **************************
%  Modifie Par
%     Aurelien Mozipo
%       Le 26 Avril 1998
%
% *****************************
%
%  ****************************
%  ³  RECONTITUTION DE         ³
%  ³  MESURANDE A L'AIDE       ³
%  ³  DU FILTRE DE KALMAN      ³
%  ****************************
%
%  Fonction pour le filtre de Kalman
```

```
%
% [xr] = kalmanK(yb,H,C,Beta);
%

function [xr,K] = kalmanK(yb,H,C,Beta);

retard=1;
ordre = 1;
Nb_pt_y =length(yb);
Nb_pt_x = Nb_pt_y;
Nb_pt_h = length(H(:,1));


%
%  Definition du systeme:
%                          x(k+1) = F x(k) + b w(k)
%                          y(k) = H' x(k) + v(k)
%
F(2:Nb_pt_h,1:Nb_pt_h-1) = eye(Nb_pt_h-1);
F(1:Nb_pt_h,Nb_pt_h) = zeros(Nb_pt_h,1);
F(1,1) = ordre;

B=zeros(Nb_pt_h,1);
B(1)=1;

%
%  Conditions initiales
%
z = zeros(Nb_pt_h,1);

P = eye(Nb_pt_h);
Ip = eye(Nb_pt_h);


%
%  Reconstitution de l'entree …a l'aide du filtre de Kalman
%
%

for k=1:Nb_pt_y

      if length(H(1,:))>1

            h=H(:,k);
                              % H(:,k) correspond a H a l'instant k
      else
            h=H;
      end

%    F_z = [ z(1) ; z(1:Nb_pt_h-1) ];

     F_z=F*z;

%    ye(k) = H'*F*z;
     ye(k) = h'*F_z;

   i(k) = yb(k) - ye(k);
```

```
    P = F*P*F' + B*Beta*B';
    P_h = P*h;

% PP = P

%%%%
%%%% Conditions initiales
    % if k == 1
    % hPh_1 = (H(:,1))'*P*H(:,1);
    % else
    hPh_1 = (H(:,k))'*P*H(:,k);   % H(:,k) correspond a H a l'instant k

    % end

    K(:,k) = P_h*inv( hPh_1 + 1 );
%%%%

    P = ( Ip - K(:,k)*h' )*P;

% P

%   z = F*z + K*i(k);
    z = F_z + K(:,k)*i(k);


   x_fkal(k) = z(Nb_pt_h);

end

x_fkal = x_fkal'; % /dt;
z = z; % /dt;
i = i';
ye = ye';

% for k = 1:retard
%    x_fkal(Nb_pt_x+k-1) = z(retard-k+1);
% end

xr = x_fkal;
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

%  ***************************
%  º   PROGRAMME FAIT PAR    º
%  º   DANIEL MASSICOTTE     º
%  º   LE 23 octobre 1991    º
%  ***************************
%
% *****************************
%  Modifie Par
%     Aurelien Mozipo
%       Le 26 Avril 1998
%
% *****************************
%
%  *****************************
%  ³  RECONTITUTION DE          ³
%  ³  MESURANDE A L'AIDE        ³
%  ³  DU FILTRE DE KALMAN       ³
```

```
%  ****************************
%
%  Fonction pour le filtre de Kalman
%
% [xr] = kalman_c(yb,h,K,dt,retard,C);
%
% Valeurs par defauts:
%
%     retard = max(size(h));       pour h non causal et centre
%     retard = max(size(h))/2;     pour h causal
%

function [xr] = kalman_c(yb,h,K,dt,retard,C);

ordre = 1;
Nb_pt_y =length(yb);
Nb_pt_x = Nb_pt_y;
Nb_pt_h = max(size(h));


%
%  D,finition du systŠme:
%                        x(k+1) = F x(k) + b w(k)
%                        y(k) = h' x(k) + v(k)
%
%F(2:Nb_pt_h,1:Nb_pt_h-1) = eye(Nb_pt_h-1);
%F(1:Nb_pt_h,Nb_pt_h) = zeros(Nb_pt_h,1);
%F(1,1) = ordre;

%
%  Conditions initiales
%
z = zeros(Nb_pt_h,1);

%
%  Reconstitution de l'entr,e … l'aide du filtre de Kalman
%
%

for k=1:Nb_pt_y

   F_z = [ z(1) ; z(1:Nb_pt_h-1) ];

%    ye(k) = h'*F*z;
    ye(k) = h'*F_z;

   i(k) = yb(k) - ye(k);
%    z = F*z + K*i(k);
    z = F_z + K*i(k);

%
%  Contrainte de positivit, sur x_kal
%
   for n = 1:Nb_pt_h
      if z(n) < 0
         z(n) = z(n)*C;
%         z(n) = z(n)*C;
      end
```

```
   end

%   x_fkal(k) = z(retard);
   x_fkal(k) = z(Nb_pt_h);    % apres verification le 26 janvier 1994

end

x_fkal = x_fkal'; % /dt;
z = z; % /dt;
i = i';
ye = ye';

for k = 1:retard
    x_fkal(Nb_pt_x+k-1) = z(retard-k+1);
end
%xr = x_fkal;
xr = x_fkal(retard : Nb_pt_x + retard-1);
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

function xe = kal_ns_q(phi, B, H, z, sigmav2, sigmaw2);

% -----------------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
% kal_ns_q.m
% Square Root Covariance Kalman Filter
% Dedicated to signal reconstruction
%
%     WITH QUANTIZATION
%
%     State Equations : x(k+1) = phi*x(k) + B*w
%                     : z(k) = H(k)*x(k) + v
% v, w : sequences of non correlated white noise of variance sigmav2 and
sigmaw2 respectively
%
% Output :  xe (estimate)
%
% Aurelien T. Mozipo
% July 14, 1998
%
%-----------------------------------------------------------------------------
--------------

%%
%     Initialisation
%
global nb_pt dim;
global X_nor;

% Covariance of measurement
teta = sigmav2;
dseta = sigmaw2;
U = sqrt(dseta);
V = sqrt(teta);
```

```
beta =  q((U/V)/X_nor) ;

S_p = q( eye(dim)/X_nor );
x_hat_p = zeros([dim 1]);           %  /X_nor

for k=1:nb_pt

      C = q(H(k,:));

      % Time Update

      A =  [S_p(1,:)' S_p(1:dim-1,:)'];
      A(dim+1,1) = beta;

      S_t = givens_q(A);
      S = S_t';


      % x_hat = phi*x_hat_p;
      x_hat = [x_hat_p(1)
                  x_hat_p(1:dim-1)];



      %-------------
      %     Measurement update
      %     Implementation II
      %-----------

      % A = [V zeros([1 dim])
      %     S'*(C') S'];
      A = [q( 1/X_nor ) zeros([1 dim])
          matvect(S',(C')) S'];


            LHS = givens_q(A);
            % LHS = MGS_q(A);

            F = LHS(1,1);
            G = LHS(1,2:dim+1);
            S_p = LHS(2:dim+1,2:dim+1)';


            % Quantization of G'/F

            X_nor1 = 4;
            % G'/F
            g_f = q( G'/(F*X_nor1) );
            x_hat1 = q( x_hat/X_nor1 );

            x_hat_p1 = q( x_hat1 + q( (g_f)*(q( z(k) - vectvect(C,x_hat)
)) ) );
            x_hat_p = q( x_hat_p1*X_nor1);

      % Filtering
       xe(k) = x_hat_p(dim);
```

```
end

% Correction of the last sample
xe(nb_pt) = x_hat_p(1);

xe = xe*X_nor;
```

```
function xe = kal_ns_sqrt_cov(phi, B, H, z, sigmav2, sigmaw2 );

% ------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
% kal_ns_sqrt_cov.m
% Square Root Covariance Kalman Filter
% Dedicated to signal reconstruction
%
%     Without quantization
%
%     State Equations : x(k+1) = phi*x(k) + B*w
%                     : z(k) = C(k)*x(k) + v
% v, w : sequences of non correlated white noise of variance sigmav2 and
sigmaw2 respectively
%
% Output :  xe (estimate)
%
%     Aurelien T. Mozipo
%     July 14, 1998
%
%------------------------------------------------------------------
--------------

%%
%     Initialisation
%

global nb_pt dim ;

% Covariance of measurement
teta = sigmav2;
dseta = sigmaw2;
U = sqrt(dseta)
V = sqrt(teta)
beta = U/V

S_p = eye(dim);
x_hat_p = zeros([dim 1]);

for k=1:nb_pt

      C = H(k,:);

      % Time Update
```

```
        % A = [S_p'*(phi')
        %      beta*(B')]
                                  % U' = U since U is a scalar
        A =  [S_p(1,:)' S_p(1:dim-1,:)'];
        A(dim+1,1) = beta;

        % S_t = MGS(A);
        S_t = givens(A);

        S = S_t';

        % x_hat = phi*x_ hat_p;
        x_hat = [x_hat_p(1)
                 x_hat_p(1:dim-1)];


        % Measurement update
        % Implementation II

              A = [1 zeros([1 dim])
                   S'*(C') S'];

              % A = [V zeros([1 dim])
              %      S'*(C') S'];

              % LHS = MGS(A);
              LHS = givens(A);


              F = LHS(1,1);
              G = LHS(1,2:dim+1);
              S_p = LHS(2:dim+1,2:dim+1)';

              % Maximum and minimum F and G
              % [max_f,min_f,max_g,min_g] = findmax_f_g(F,G);

              x_hat_p = x_hat + (G'/F)*(z(k) - C*x_hat);

        % Filtering
         xe(k) = x_hat_p(dim);

        % pause;

end

% Correction of the last sample
xe(nb_pt) = x_hat_p(1);
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
function c = matmult(a,b);

% -------------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
```

```
%
%
% matmult.m
% Matrix-Matrix multiplication for quantization
% Input Matrices are considered quantizised
%
% Input     a,b
%
% Output    c = a*b
%
% By :
% Aurelien T. Mozipo
% July 14, 1998
%
%------------------------------------------------------------------------
--------------

nb_lg_a = length(a(:,1));
nb_col_a = length(a(1,:));
nb_col_b = length(b(1,:));

% insert here test on correspondances betwen nuber of colunm and rows in
a and b

c = zeros([nb_lg_a nb_col_b]);

for i = 1:nb_lg_a
      for j =1:nb_col_b
            for k=1:nb_col_a
                  c(i,j) = q( c(i,j) + q( a(i,k)*b(k,j) ) );
            end
      end
endfunction c = matvect(a,b);

% ----------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% matvect.m
% Matrix-vector multiplication for quantization
% Input Matrix and vector are considered quantizised
%
% Input     a,b
%
% Output    c = a*b
%
% By :
% Aurelien T. Mozipo
% July 14, 1998
%
%------------------------------------------------------------------------
--------------

nb_lg_a = length(a(:,1));
```

```
nb_col_a = length(a(1,:));

% insert here test on correspondances betwen nuber of colunm and rows in
a and b

c = zeros([nb_lg_a 1]);

for i = 1:nb_lg_a
      for k=1:nb_col_a
                  c(i) = q( c(i) + q( a(i,k)*b(k) ) );
      end
end;
```

■■

```
function sortie = q(entree)

% -----------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% q.m
% return a quantized number
% quantization by truncation, sign and modulus, 2's complement
% Division of two numbers with the Newton Iterative method
%
%
% Input     entree : input data
%
% Output    sortie : quantizised output data
%
%
% By :
% Aurelien T. Mozipo
% July 14, 1998
%
%------------------------------------------------------------------------
--------------

global nb_bit;
global X_max X_min;
global flag_overflow abs_min abs_max;

%     Seeking the maximum
%     Determination of the maximum value

% maximum = max(max(entree));
% X_max = max(X_max,maximum);


%     Seeking the minimum
%     Determination of the minimum value

% minimum = min(min(entree));
% X_min = min(X_min,minimum);
```

```
%--------------------
% Absolute max. value

% X_max = max(X_max,abs(X_min));

% ---------------------
%
%     Quantization
%
% ----------------------

pas = (X_max - X_min)/(2^(nb_bit-1));     % -1);

sortie = floor(entree/pas)*pas;

sortie = min(max(sortie, X_min), X_max);
```

■■■

```
function [e_q] = q_err(xe,xe_q);

% ------------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% q_err.m
% Computation of quantization errors
%
% Inputs    a     : channel test signal
%           xe    : estimated signal computed in floating point (filtered
value)
%           xe_q  : Estimated signal computed with quantization
%
% Output    e_q   : quantization error withquantization (xe_q)
%
%
% Aurelien T. Mozipo
% July, 16, 1998
%
%------------------------------------------------------------------------
--------------

global nb_bit_min step_bit nb_bit_max;

i = 0;
for n = nb_bit_min:step_bit:nb_bit_max
      i = i+1;
      e_q(i) = norm(xe - xe_q(i,:))/norm(xe);


end;

return;
```

■■■

```
function xr_da = result(xr);
```

```
% ---------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% result.m
% Plots the simulation results given by QuickHDL
%
%
% Input     xr    : QuickHDL results (Binary)
%
%
% Output    xr_da : xr converted in decimal
%
% By :
%
%*******************************************
%     Aurelien T. Mozipo
%     Feb.,16, 1999
%
%----------------------------------------------------------------------
-------------


global X_nor;

nb_bit = 20;
nb_pt = length(xr);

for i=1:nb_pt
      if xr(i)>=2^19
            temp = dec2bin(xr(i)-2^(nb_bit-1)-1, nb_bit-1);
            for j = 1:nb_bit-1
                  if temp(j) == '1'
                        temp(j) = '0';
                  else
                        temp(j) = '1';
                  end;
            end;
            x = -bin2dec(temp)*2^-(nb_bit-1);
      else
            x = xr(i)*2^-(nb_bit-1);
      end;
      xr_da(i) = x;
end;
```

```
function y = signal(h,a,v)

% Entree :   h : reponse impulsionnelle invariante
%          a : Signal d'entree du canal
%          v : Bruit de mesure ( a la sortie du canal)
% NbrePeriodes : Nbre de periodes de variation de la rep. imp. sur la
duree de l'experience
% Sortie :   y : Signal corrompu a la sortie du canal
```

```
%
%
%
% GEI6033 - Techniques avancees de traitement numeriques de signaux
%
%     Aurelien Mozipo
%     Le 26 Avril 1998
%
% Dernieres Modifications
%     Le 24 Juin 1998

% Generation du signal bruite pour reconstitution

NbreIterations = length(a);

% Initialisation de y a v
y = v;

for n=1:NbreIterations
      for k=1:length(h)
            if (n-k <= 0)
                  ank = 0;
            else
                  ank = a(n-k);
            end
            y(n) = y(n) + h(k)*ank;
      end
end
```

■■■

```
function [xe,xe_q] = simul(phi,B,a,yb,v,H);


% -----------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% simul.m
% Computation of the estimation with and without quantizations
% Square root covariance Kalman filtering
%
% Inputs     phi, B, H    : state matrices
%            a            : Channel test signal (original signal)
%            yb           : noisy corrupted signal
%            v            : noise
%
% Output     xe           : estimated signal without quantization
%            xe_q         : estimated signal with quantization
%
% Aurelien T. Mozipo
% July, 16, 1998
%
%
```

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

global sigmav2 sigmaw2;
global nb_pt nb_trial;
global nb_bit_min nb_bit_max step_bit;
global nb_bit;
global h;
global p p_float p_nor;
global X_nor;

dim = length(h);

      % [a,v] = entree(nb_pt,sigmav2) ;
      % [yb,H] = signal_ns(h,a,v);


      % -------------------------------
      % Filtering without quantization
      %--------------------------------

xe = kal_ns_sqrt_cov(phi,B,H,yb,sigmav2,sigmaw2);

i = 0;
for n = nb_bit_min:step_bit:nb_bit_max
      i = i+1;
      nb_bit = n;
      B_q = q( B );
      yb_q = q( yb/X_nor );
      H_q = q( H );
      p = q(p_float/p_nor);

      xe_q(i,:) = kal_ns_q(phi,B_q,H_q,yb_q,sigmav2,sigmaw2);

end;

return;
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

function [xe,xe_q] = simul_std(phi,B,a,yb,v,H);


% ----------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% simul_std.m
% Computation of the estimation with and without quantizations
% Standard Covariance Kalman filtering
%
% Inputs    phi, B, H   : state matrices
%           a           : Channel test signal (original signal)
%           yb          : noisy corrupted signal
%           v           : noise
```

```
%
% Output    xe          : estimated signal without quantization
%           xe_q        : estimated signal with quantization
%
% Aurelien T. Mozipo
% July, 16, 1998
%
%
%-----------------------------------------------------------------------
--------------

global sigmav2 sigmaw2;
global nb_pt nb_trial;
global nb_bit;
global nb_bit_min step_bit nb_bit_max;
global h;
global X_nor;

dim = length(h);
sigmav20 = sigmav2 ;
sigmaw20 = sigmaw2 ;
% ***************************************************************
%
%           Filtering without quantization                   *
%
% ***************************************************************

beta = sigmaw2/sigmav2;
xe = kalmanK(yb,H',1,beta);
xe = xe';

% ***************************************************************
%
%           Filtering with quantization                *
%
% ***************************************************************

i = 0;
for  n = nb_bit_min:step_bit:nb_bit_max
      i = i + 1;
      nb_bit = n;
      % sigmav2 = q( sigmav20 );
      % sigmaw2 = q( sigmaw20 );
      % B_q = q( B );
      yb_q = q( yb );
      H_q =  H ;
      beta_q = q( beta );

      xe_q(:,i) = cov_std_q(yb_q,H_q',1,beta_q);
end;

xe_q = xe_q';
```

```
function sortie = sqrt1(x);

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

```
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% sqrt1.m
% Estimation of SQRT by a 3rd ordre polynomial.

% Input      x      : input data
%
% Output     sortie       : output data = sqrt(x)
%
%
% By :
% Aurelien T. Mozipo
% July 14, 1998
%
%------------------------------------------------------------------------
--------------

global p p_float p_nor;

% global X_max;


% p = q(p);
x = q(x);

temp_sortie = q( p(1) + p(2)*x + q( p(3)*q( x.^2 ) ) + q( q( p(4)*q(
x.^2) ).*x) );
sortie = q( p_nor*temp_sortie );
```

```
function y = table_div(x);

% ----------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% table_div.m
% retunrs an approximation of 1/x with 1<=x<2
%
%
%
% Input      x      : input data ; 1<=x<2
%
% Output     y      : output data = 1/x
%
%
% By :
% Aurelien T. Mozipo
% July 14, 1998
%
%------------------------------------------------------------------------
--------------
```

```
if (x>=1 & x<1.1)
      y = 0.9524;
elseif (x>=1.1 & x<1.2)
      y = 0.8696;
elseif (x>=1.2 & x<1.3)
      y = 0.8;
elseif (x>=1.3 & x<1.4)
      y = 0.7407;
elseif (x>=1.4 & x<1.5)
      y = 0.6897;
elseif (x>=1.5 & x<1.6)
      y = 0.6452;
elseif (x>=1.6 & x<1.7)
      y = 0.6061;
elseif (x>=1.7 & x<1.8)
      y = 0.5714;
elseif (x>=1.8 & x<1.9)
      y = 0.5405;
elseif (x>=1.9 & x<2)
      y = 0.5128;
end;
```

```
function c = vectmat(a,b);

% --------------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% vectmat.m
% Vector-matrix multiplication for quantization
% Input Matrix and vector are considered quantizised
%
% Input     a,b
%
% Output    c = a*b
%
% By :
% Aurelien T. Mozipo
% July 14, 1998
%
%---------------------------------------------------------------------------
--------------

nb_lg_b = length(b(:,1));
nb_col_b = length(b(1,:));

% insert here test on correspondances betwen number of colunm and rows in
a and b

c = zeros([1 nb_col_b]);

for i = 1:nb_col_b
```

```
      for k=1:nb_lg_b
            c(i) = q( c(i) + q( a(k)*b(k,i) ) );
      end
end;
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

function c = vectvect(a,b);

% --------------------------------------------------------------------------
-------------
%
% Modelisation and simulation of a Klaman filtering based systolic array
processors
% for adaptive channel equalization using MMAlpha
%
%
% vectvect.m
% Vector-vector multiplication for quantization
% Input vectors are considered quantizised
%
% Input     a,b
%
% Output    c = a*b
%
% By :
% Aurelien T. Mozipo
% July 14, 1998
%
%---------------------------------------------------------------------------
--------------

nb_lg_b = length(b);
nb_col_a = length(a);

% insert here test on correspondances betwen nuber of colunm and rows in
a and b

c = 0;

for k = 1:nb_col_a
      c = q( c + q( a(k)*b(k) ) );
end
```

Annexes IV

Programmes VHDL

```
-- ---------------------------------------------------------------------------
-------
--
-- File : constants.vhd
--          Constants definition package
--
-- Modelisation of an array processors for Adaptive Channel Equalization
-- based on square root covariance Kalman Filtering using Givens
rotations.
--
-- Author :
--               Aurelien T. Mozipo
--               March 30, 1999
--
-- ---------------------------------------------------------------------------
--------


LIBRARY IEEE;
 USE IEEE.std_logic_1164.all;
 USE IEEE.std_logic_arith.all;
 USE IEEE.std_logic_signed.all;


PACKAGE constants IS

--
-- Constants real values
--
--    CONSTANT p_float1 : REAL := 1.868449253821409e-01;
--    CONSTANT p_float2 : REAL := 1.561215531933085e+00;
--    CONSTANT p_float3 : REAL :=  -1.424927599171505e+00;
--    CONSTANT p_float4 : REAL :=  7.676057100024836e-01;
--    CONSTANT p_nor_float : REAL := p_float2;
```

```
-- CONSTANT beta : REAL := 3.162277660168379e+01;
-- CONSTANT X_nor : REAL := 50;
-- CONSTANT beta_nor : REAL := 6.324555320336758e-01;



-- Number of bits for constants and variables respectively
--
      CONSTANT nb_const : integer := 16;
      CONSTANT nb_var : integer := 20;


--

-- Others constants
--
      CONSTANT c0 : SIGNED(0 DOWNTO 0) := "0";
                                -- 0
      CONSTANT c1 : SIGNED(nb_const-1 DOWNTO 0) := "0111111111111111";
      -- 1
      CONSTANT c05 : SIGNED(nb_const-1 DOWNTO 0) := "0100000000000000";
      -- 0.5
      CONSTANT c025 : SIGNED(nb_const-1 DOWNTO 0) := "0010000000000000";
      -- 0.25


--    -----------------------------------------------------------------------
----------------------------------
--    Constant for division and inversion
-- Shape : MSB = integer part, others = decimal part. No sign bit. All
theses constants assumes to be > 0
--    -----------------------------------------------------------------------
----------------------------------

      CONSTANT nb_init        : NATURAL := 5;
            -- initial number of bits
      CONSTANT nb_iter        : NATURAL := 3;
            -- Number of iterations for the division algorithm
      CONSTANT c1p0 : UNSIGNED := "10000000000000000000";
      CONSTANT c1p1 : UNSIGNED := "10001100110011001100";
      CONSTANT c1p2 : UNSIGNED := "10011001100110011001";
      CONSTANT c1p3 : UNSIGNED := "10100110011001100110";
      CONSTANT c1p4 : UNSIGNED := "10110011001100110011";
      CONSTANT c1p5 : UNSIGNED := "11000000000000000000";
      CONSTANT c1p6 : UNSIGNED := "11001100110011001100";
      CONSTANT c1p7 : UNSIGNED := "11011001100110011001";
      CONSTANT c1p8 : UNSIGNED := "11100110011001100110";
      CONSTANT c1p9 : UNSIGNED := "11110011001100110011";
      CONSTANT c2p0m : UNSIGNED := "11111111111111111111";

--    ----------------------------------------

--  Constantes X_nor, one_nor, beta_nor

      CONSTANT X_nor          : UNSIGNED(5 DOWNTO 0) := "110010";
                                          -- 50 : No decimal part, No
sign bit                  -- "111111"; -- 63
```

```
      CONSTANT one_nor   : SIGNED(nb_const-1 DOWNTO 0) :=
"0000001010001111";
                                                    -- "0000001000001000";
      CONSTANT beta_nor       : SIGNED(nb_const-1 DOWNTO 0) :=
"0101000011110100";
                                                    --
"0100000000111111""0100000000111111"
      CONSTANT fgnor : INTEGER := 2;



--  Constantes p1 p2 p3 p4 et p_nor

      CONSTANT p1 : SIGNED(nb_const-1 DOWNTO 0) := "0000111101010001";
      -- p_float1/p_nor;
      CONSTANT p2 : SIGNED(nb_const-1 DOWNTO 0) := "0111111111111111";
      -- p_float2/p_nor;
      CONSTANT p3 : SIGNED(nb_const-1 DOWNTO 0) := "1000101100101110";
      -- p_float3/p_nor;
      CONSTANT p4 : SIGNED(nb_const-1 DOWNTO 0) := "0011111011101111";
      -- p_float4/p_nor;
      CONSTANT p_nor : UNSIGNED(nb_const-1 DOWNTO 0) :=
"1100011111010101";  -- p_nor = 1.561215531933085

                          -- MSB = integer part

                          -- Others = decimal part

                          -- No sign bit

-- ----------------------------------------------------
-- Elementary Processor operating modes
-----------------------------------------------------

      CONSTANT mode0 : STD_LOGIC_VECTOR(2 DOWNTO 0) := "000";
      CONSTANT mode1 : STD_LOGIC_VECTOR(2 DOWNTO 0) := "001";
      CONSTANT mode2 : STD_LOGIC_VECTOR(2 DOWNTO 0) := "010";
      CONSTANT mode3 : STD_LOGIC_VECTOR(2 DOWNTO 0) := "011";
      CONSTANT mode4 : STD_LOGIC_VECTOR(2 DOWNTO 0) := "100";
      CONSTANT mode5 : STD_LOGIC_VECTOR(2 DOWNTO 0) := "101";
      CONSTANT mode6 : STD_LOGIC_VECTOR(2 DOWNTO 0) := "110";
      CONSTANT mode7 : STD_LOGIC_VECTOR(2 DOWNTO 0) := "111";

END constants;

■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
-- --------------------------------------------------------------------------
-------
--
-- File : type_def.vhd
--          Type definition package
--
-- Modelisation of an array processors for Adaptive Channel Equalization
-- based on square root covariance Kalman Filtering using Givens
rotations.
--
-- Author :
--                Aurelien T. Mozipo
```

```
--                March 30, 1999
--
-- ------------------------------------------------------------------------
--------


LIBRARY IEEE;
 USE IEEE.std_logic_1164.all;
 USE IEEE.std_logic_arith.all;
 USE IEEE.std_logic_signed.all;
 --use ieee.numeric_std.all;

LIBRARY lib;
 USE lib.constants.all;


PACKAGE type_def IS


      SUBTYPE dataType IS SIGNED( nb_var-1 DOWNTO 0 );
      SUBTYPE datainType IS SIGNED ( nb_var-1 DOWNTO 0 );
      SUBTYPE dataoutType IS SIGNED ( nb_var-1 DOWNTO 0 );
      SUBTYPE consType IS SIGNED (nb_const-1 DOWNTO 0);

      SUBTYPE clkType IS STD_LOGIC ;
      SUBTYPE rstType IS STD_LOGIC ;
      SUBTYPE initType IS STD_LOGIC;


      SUBTYPE modeType1 IS STD_LOGIC_VECTOR (2 DOWNTO 0);
      SUBTYPE modeType2 IS STD_LOGIC_VECTOR(1 DOWNTO 0);
      SUBTYPE modeType IS STD_LOGIC_VECTOR(2 DOWNTO 0);


-- ---------------------------------------------------
-- Types for State Machine and muxbloc
-----------------------------------------------------

      TYPE stateType IS (state14, state0, state1, state0_1, state1_1,
state2, state3, state4, state5, state6,
                                    state7, state8, state9,
state10, state11, state12, state13);
      TYPE stateType0 IS (state14, state0, state1, state2, state3,
state4, state5, state6,
                                    state7, state8, state9,
state10, state11, state12, state13);
      SUBTYPE addrtype IS STD_LOGIC_VECTOR(2 DOWNTO 0);


END type_def;
```

■■

```
-- ---------------------------------------------------------------------------
-------
--
-- File : components.vhd
--          Components definition package
--
-- Modelisation of a systolic array processors for Adaptive Channel
Equalization
-- based on square root covariance Kalman Filtering using Givens
rotations.
--
-- Author :
--                Aurelien T. Mozipo
--                January 30, 1999
--
-- ---------------------------------------------------------------------------
--------


library IEEE, lib;
      use IEEE.std_logic_1164.all;
      use IEEE.std_logic_arith.all;

      use lib.constants.all;
      use lib.type_def.all;


package components is

------------------------------------------------------------------------------
-------
--              Declarations from : DW01_add.vhd
------------------------------------------------------------------------------
-------

  component DW01_add
    generic(width : NATURAL);
    port(A,B : in SIGNED(width-1 downto 0);
         CI : in std_logic;
         SUM : out SIGNED(width-1 downto 0);
         CO : out std_logic);

  end component;


------------------------------------------------------------------------------
-------
--              Declarations from : add.vhd
------------------------------------------------------------------------------
-------

  component add
    generic(width : NATURAL);
    port(A,B : in SIGNED(width-1 downto 0);
         -- CI : in std_logic;
         SUM : out SIGNED(width-1 downto 0);
         CO : out std_logic);
```

```
  end component;

------------------------------------------------------------------------
-------
--              Declarations from : DW01_csa.vhd
------------------------------------------------------------------------
-------

  component DW01_csa
    generic (
      width : INTEGER
    );
    port (
      a     : in SIGNED(width-1 downto 0);
      b     : in SIGNED(width-1 downto 0);
      c     : in SIGNED(width-1 downto 0);
      ci    : in std_logic;
      carry : out SIGNED(width-1 downto 0);
      sum   : out SIGNED(width-1 downto 0);
      co    : out std_logic
    );

  end component;



------------------------------------------------------------------------
-------
--              Declarations from : csa.vhd
------------------------------------------------------------------------
-------

  component csa
    generic (
      width : INTEGER
    );
    port (
      a     : in SIGNED(width-1 downto 0);
      b     : in SIGNED(width-1 downto 0);
      c     : in SIGNED(width-1 downto 0);
      -- ci     : in std_logic;
      carry : out SIGNED(width downto 0);
      sum   : out SIGNED(width downto 0)
      -- co     : out std_logic
    );

  end component;



------------------------------------------------------------------------
-------
--              Declarations from : /csa/csa.vhd
------------------------------------------------------------------------
-------

  component csa_op
    generic (
```

```
      width : INTEGER
    );
    port (
      a           : in SIGNED(width-4 downto 0);
      b           : in SIGNED(width-3 downto 0);
      c           : in SIGNED(width-3 downto 0);
      c_1msb      : in STD_LOGIC;
      carry       : out SIGNED(width-3 downto 0);
      sum         : out SIGNED(width-3 downto 0)

    );

  end component;

  ------------------------------

component mux2x1
    port (
      in1         : in std_logic;
      in2         : in std_logic;
      sel         : in std_logic;

      s     : out std_logic

    );

  end component;

 -----------------------------------------------------------------------
--------
--            Declarations from : cell1.vhd
-----------------------------------------------------------------------
------


component cell1
      PORT(din, zin_p, zin_m, tin, restorein, asin          : IN
STD_LOGIC;
              asout, restoreout, dout, zout_p, zout_m, tout : OUT
STD_LOGIC);
end component;


 -----------------------------------------------------------------------
--------
--            Declarations from : cell2.vhd
-----------------------------------------------------------------------
------

component cell2
      PORT(z2in_p, z2in_m, tin, restorein, asin, compress    : IN
STD_LOGIC;
              asout, restoreout, z1out_p, z1out_m           : OUT
STD_LOGIC);
end component;
```

```
 -----------------------------------------------------------------------
--------
--             Declarations from : cellS.vhd
------------------------------------------------------------------------
------

component cellS
      PORT(                  -- clk
                                   : IN STD_ULOGIC;
              z1in_p, z1in_m, z2in_p, z2in_m                  : IN
STD_LOGIC;
           asout, compress, restoreout, q_p, q_m        : OUT STD_LOGIC);
end component;


 -----------------------------------------------------------------------
--------
--             Declarations from : one_ligne.vhd
------------------------------------------------------------------------
------

component one_line
--    GENERIC(nb_var : integer := nb_var);      -- Input numerator word
length
--                 nb_var_den : integer);             -- Input
denominator word length
      PORT(clk                          : IN STD_ULOGIC;
              n_p                        : IN STD_LOGIC_VECTOR(nb_var-1
downto 2);
              n_m                        : IN STD_LOGIC_VECTOR(nb_var-1
downto 2);
           d                             : IN STD_LOGIC_VECTOR(nb_var-2
downto 3);
           nout_m                  : OUT STD_LOGIC_VECTOR(nb_var-1
downto 2);
              nout_p                     : OUT STD_LOGIC_VECTOR(nb_var-1
downto 2);
           d_out                   : OUT STD_LOGIC_VECTOR(nb_var-2
downto 3);
              qout_p, qout_m   : OUT STD_LOGIC
          );

end component;



 -----------------------------------------------------------------------
--------
--             Declarations from : modSRT.vhd
------------------------------------------------------------------------
------

COMPONENT modSRT

      --GENERIC (length_num   : integer := nb_var;
      --                 length_den  : integer := nb_var;
      --                 length_quot : integer := nb_var);
      PORT (clk                           : IN STD_ULOGIC;
```

```
                  num_p                          : IN
STD_LOGIC_VECTOR(nb_var-1 downto 2);
               num_m                       : IN STD_LOGIC_VECTOR(nb_var-1
downto 2);
            den                            : IN STD_LOGIC_VECTOR(nb_var-2
downto 3);
               quot_p, quot_m        : OUT STD_LOGIC_VECTOR(nb_var downto
1));

END COMPONENT;
-------------------------------------------------------------------------
-------
--              Declarations from : diag.vhd
-------------------------------------------------------------------------
------

COMPONENT diag
      PORT(clk                                         : IN clkType;
              Rst                                      : IN rstType;
              mode                                     : IN modeType;
              initin                                   : IN initType ;
              datain                                   : IN dataType ;
              cout, sout                         : OUT dataType ;
              initout                            : OUT initType;
              dataout_dn                         : OUT dataType
             );
END COMPONENT;


-------------------------------------------------------------------------
-------
--              Declarations from : ofdiag.vhd
-------------------------------------------------------------------------
------

-- Component for PE(1,2:3)

COMPONENT ofdiag1
      PORT(clk                                               : IN clkType;
              Rst                                            : IN rstType;
              mode                                           : IN
modeType;
              initin                                         : IN initType
;
              datain                                         : IN dataType
;
              c, s                                           : IN dataType
;
              sout                                           : OUT
dataType;
              cout                                           : OUT
dataType;
              initout_rt, initout_dn       : OUT initType;
              dataout_dn                                 : OUT dataType
             );
END COMPONENT;
```

```
-- Component for PE(1,4)

COMPONENT ofdiag14
      PORT(clk                                                  : IN clkType;
              Rst                                               : IN rstType;
              mode                                              : IN
modeType;
              initin                                            : IN initType
;
              datain                                            : IN dataType
;
              c, s                                              : IN dataType
;
              z                                                 : IN
dataType;
              sout                                              : OUT
dataType;
              cout                                              : OUT
dataType;
              initout_rt, initout_dn          : OUT initType;
              dataout_dn                                : OUT dataType
             );
END COMPONENT;


-- Component for PE(2:3,:)

COMPONENT ofdiagX
      PORT(clk                                                  : IN clkType;
              Rst                                               : IN rstType;
              mode                                              : IN
modeType;
              initin                                            : IN initType
;
              datain                                            : IN dataType
;
              c, s                                              : IN dataType
;
              cout, sout                                : OUT dataType;
              initout_rt, initout_dn          : OUT initType;
              dataout_dn                                : OUT dataType
             );
END COMPONENT;


------------------------------------------------------------------------
-------
--             Declarations from : array_proc.vhd
------------------------------------------------------------------------
------

COMPONENT array_proc
      PORT( clk                                                         :
IN clkType;
                   Rst
      : IN rstType;
```

```
                  initpe11
        : IN initType;
                  initpe22
        : IN initType;
                  modepe11, modepe12, modepe13,
                  modepe14, modepe22, modepe23,
                  modepe24, modepe33, modepe34,
                  modepe44
        : IN modeType;
                  datainpe11, datainpe12,
                  datainpe13, datainpe14                  : IN
dataType;
                  zin
        : IN dataType;
                  dataout_dnpe11, dataout_dnpe22,
                  dataout_dnpe33, dataout_dnpe44,
                  dataout_rtpe14, dataout_rtpe24,
                  dataout_rtpe34, dataout_rtpe44      : OUT dataType
              );
END COMPONENT;


-------------------------------------------------------------------------
-------
--              Declarations from : file.vhd
-------------------------------------------------------------------------
------

-- Component for file1

COMPONENT file1
      PORT( clk               : IN clkType;
--                Rst               : IN rstType;
                  datain            : IN dataType;
                  dataout           : OUT dataType
              );
END COMPONENT;


-- Component for file2, file3

COMPONENT file23
      PORT( clk               : IN clkType;
                  Rst               : IN rstType;
                  datain            : IN dataType;
                  dataout           : OUT dataType
              );
END COMPONENT;


-- Component for file4

COMPONENT file4
      PORT( clk               : IN clkType;
                  Rst               : IN rstType;
                  dir               : IN STD_LOGIC;
```

```
                  datain              : IN dataType;
                  dataout             : OUT dataType
              );
END COMPONENT;

--------------------------------------------------------------------------
-------
--             Declarations from : muxbloc.vhd
--------------------------------------------------------------------------
------


COMPONENT muxbloc
      PORT( Rst
                        : IN rstType;
                  state
                              : IN stateType0;
                  datain
                              : IN dataType;
                  frompe11, frompe22, frompe33, frompe44,
                                          -- pe44 : cout => frompe44
                  frompe14, frompe24, frompe34
      : IN dataType;
                  tope11, tope12, tope13, tope14_up, tope14_rt    : OUT
dataType;
                  x_hat
                              : OUT dataType);
END COMPONENT ;


--------------------------------------------------------------------------
-------
--             Declarations from : sm.vhd
--------------------------------------------------------------------------
------

COMPONENT sm
      PORT (clk : IN clkType;
                  Rst : IN rstType;
                  stateout : OUT stateType0);
END COMPONENT ;

--------------------------------------------------------------------------
-------
--             Declarations from : $DESIGNS/control/entity.vhd
--------------------------------------------------------------------------
------
COMPONENT control
   PORT (
      clk : IN std_logic;
      pause : IN std_logic;
      ready : IN std_logic;
      reset : IN std_logic;
      address : OUT addrtype;
      clk_to_array : OUT std_logic;
      initpe11 : OUT std_logic;
      initpe22 : OUT std_logic;
      modepe11 : OUT modetype;
```

```
      modepe12 : OUT modetype;
      modepe13 : OUT modetype;
      modepe14 : OUT modetype;
      modepe22 : OUT modetype;
      modepe23 : OUT modetype;
      modepe24 : OUT modetype;
      modepe33 : OUT modetype;
      modepe34 : OUT modetype;
      modepe44 : OUT modetype;
      request : OUT std_logic;
      rst_to_array : OUT std_logic;
      state_to_mux : OUT stateType
   );
END COMPONENT ;

end components;
```

■■

```
--
-- Component : type_def
--
-- Generated by System Architect version v8.5_3.3 by mozipo on Feb 19, 99
--

LIBRARY IEEE;
 USE IEEE.std_logic_1164.all;
 USE IEEE.std_logic_arith.all;
 USE IEEE.std_logic_signed.all;

--LIBRARY DESIGNS;
--     USE DESIGNS.constants.all;


PACKAGE type_def IS

      SUBTYPE modeType1 IS STD_LOGIC_VECTOR (2 DOWNTO 0);
      SUBTYPE modeType2 IS STD_LOGIC_VECTOR(1 DOWNTO 0);
      SUBTYPE modeType IS STD_LOGIC_VECTOR (2 DOWNTO 0);


      SUBTYPE addrType IS STD_LOGIC_VECTOR (2 DOWNTO 0);

      -- For State Machine

      TYPE stateType IS (state14, state0, state1, state0_1, state1_1,
state2, state3, state4, state5, state6,
                                           state7, state8, state9,
state10, state11, state12, state13);


END type_def;
```

■■

```
-- -----------------------------------------------------------------------
-------
--
-- File : package_body.vhd
--          Functions and procedures definition package
--
-- Modelisation of an array processors for Adaptive Channel Equalization
-- based on square root covariance Kalman Filtering using Givens
rotations.
--
-- Author :
--                  Aurelien T. Mozipo
--                  March 30, 1999
--
-- -----------------------------------------------------------------------
--------


Library IEEE;
  USE IEEE.STD_LOGIC_1164.ALL;
  USE IEEE.STD_LOGIC_ARITH.ALL;
  USE IEEE.STD_LOGIC_SIGNED.ALL;
--  USE ieee.numeric_std.ALL;
```

■■

```
-- -----------------------------------------------------------------------
-------
--
-- File : package_head.vhd
--          Functions and procedures declaration package
--
-- Modelisation of an array processors for Adaptive Channel Equalization
-- based on square root covariance Kalman Filtering using Givens
rotations.
--
-- Author :
--                  Aurelien T. Mozipo
--                  March 30, 1999
--
-- -----------------------------------------------------------------------
--------


Library ieee, lib;
      USE ieee.std_logic_1164.all;
      USE ieee.std_logic_arith.all;
      USE ieee.std_logic_signed.all;

--    USE ieee.numeric_std.ALL;

      USE lib.constants.all;
      USE lib.type_def.all;


PACKAGE packages IS
```

```
        ATTRIBUTE dont_unroll : BOOLEAN;

        FUNCTION adder(a, b, cin : IN STD_LOGIC) RETURN STD_LOGIC_VECTOR;

        FUNCTION and_wise(a : IN STD_LOGIC;
                          b : IN SIGNED) RETURN SIGNED;

        FUNCTION not_wise(a : IN UNSIGNED) RETURN UNSIGNED;

        FUNCTION MUX2x1(in1, in2, sel : IN STD_LOGIC) RETURN STD_LOGIC;

--      FUNCTION fun_MUX2x1(Input0, Input1, Sel: UX01) RETURN UX01;

        FUNCTION quantif(a : SIGNED) RETURN dataType;

        FUNCTION quant(a : SIGNED) RETURN dataType;

        FUNCTION sqrt1(x : UNSIGNED) RETURN UNSIGNED;

        FUNCTION table_div(x : UNSIGNED) RETURN UNSIGNED;

        FUNCTION inversion(x : UNSIGNED) RETURN UNSIGNED;

        FUNCTION division(num,den : SIGNED) RETURN SIGNED;

        FUNCTION reduce(x : UNSIGNED; n : INTEGER) RETURN UNSIGNED;


-- Function r_extend : Extend of the number of bits of the variable to n
bits
-- adding zeroes to the right

        FUNCTION r_extend(x : UNSIGNED; n : INTEGER) RETURN UNSIGNED;

-- Function l_extend : Extend of the number of bits of a variable to n
bits
-- adding zeroes to the left

        FUNCTION l_extend(x : UNSIGNED; n : INTEGER) RETURN UNSIGNED;

        FUNCTION xsll(arg : STD_LOGIC_VECTOR; count : NATURAL) RETURN
STD_LOGIC_VECTOR;

        FUNCTION xsrl(arg : STD_LOGIC_VECTOR; count : NATURAL) RETURN
STD_LOGIC_VECTOR;

        FUNCTION shift_left (arg : UNSIGNED; count : NATURAL) RETURN
UNSIGNED;

        FUNCTION shift_right (arg : UNSIGNED; count : NATURAL) RETURN
UNSIGNED;

        FUNCTION mshift_left (arg : SIGNED; count: NATURAL) RETURN SIGNED;

        FUNCTION mshift_right (arg : SIGNED; count: NATURAL) RETURN SIGNED;

        PROCEDURE gen_of_diag(ain, r, cin, sin          : IN SIGNED;
                    cout, sout, rout, aout  : OUT SIGNED);
```

```
      PROCEDURE gen_diag(ain, r      : IN SIGNED;
               c, s, rout     : OUT SIGNED);

      FUNCTION mac1(x,y,z : SIGNED) RETURN SIGNED;

      FUNCTION mac2(w,x,y,z : SIGNED) RETURN SIGNED;


END packages;
```

■■■

```
PACKAGE BODY packages IS

-- null range array constants

--  constant NAU: UNSIGNED(0 downto 1) := (others => '0');
--  constant NAS: SIGNED(0 downto 1) := (others => '0');

      constant NAU: UNSIGNED(0 downto 0) := (others => 'X');          -
- !!
   constant NAS: SIGNED(0 downto 0) := (others => 'X');               -
- !!

-- implementation controls

  constant NO_WARNING: BOOLEAN := FALSE;  -- default to emit warnings


      FUNCTION adder (a, b, cin : IN STD_LOGIC) RETURN STD_LOGIC_VECTOR
IS

      VARIABLE g,p : STD_LOGIC;
      VARIABLE s   : STD_LOGIC_VECTOR (1 DOWNTO 0);
      BEGIN
            p := a XOR b;
            g := a AND b;
            s(0) := p XOR cin;            -- sum
            s(1) := (cin AND p) OR g;     -- carry out
      RETURN s;
      END adder;


      FUNCTION and_wise(a : IN STD_LOGIC;
                        b : IN SIGNED) RETURN SIGNED IS
      CONSTANT  b_length : integer := b'LENGTH;
      VARIABLE r : SIGNED (b_length-1 DOWNTO 0);
      BEGIN
            FOR i IN 0 TO b_length-1 LOOP
                  r(i) := a AND b(i);
            END LOOP;
      RETURN r;
      END and_wise;
```

```
      FUNCTION not_wise(a : IN UNSIGNED) RETURN UNSIGNED IS
      CONSTANT  a_left : integer := a'LEFT;
      CONSTANT  a_right : integer := a'RIGHT;
      VARIABLE r : UNSIGNED (a_left DOWNTO a_right);

      BEGIN
            FOR i IN a_left DOWNTO a_right LOOP
                  r(i) := NOT(a(i));
            END LOOP;
            RETURN r;
      END not_wise;



      FUNCTION MUX2x1 (in1, in2, sel : IN STD_LOGIC) RETURN STD_LOGIC IS
      BEGIN
            IF (sel = '0' or sel = 'L') THEN
                  RETURN in1;
            ELSIF (sel = '1' or sel = 'H') THEN
                  RETURN in2;
            ELSE
                  RETURN 'X';    --- !!!!!!!!!!!!!!!!!!!!
            END IF;
      END MUX2x1;


--    FUNCTION fun_MUX2x1(Input0, Input1, Sel: UX01) RETURN UX01 IS
--    TYPE MUX_TABLE IS ARRAY (UX01, UX01, UX01) of UX01;

      -- truth table for "MUX2x1" function
--    CONSTANT tbl_MUX2x1: MUX_TABLE :=
      -----------------------------------------
   --| In0  'U'  'X'  '0'  '1'      | Sel In1 |
      -----------------------------------------
--          ((('U', 'U', 'U', 'U'),   --| 'U' 'U' |
--            ('U', 'U', 'U', 'U'),   --| 'X' 'U' |
--                ('U', 'X', '0', '1'),   --| '0'  'U' |
--                ('U', 'U', 'U', 'U')),  --| '1'  'U' |
--           (('U', 'X', 'U', 'U'),   --| 'U' 'X' |
--            ('U', 'X', 'X', 'X'),   --| 'X' 'X' |
--                ('U', 'X', '0', '1'),   --| '0'  'X' |
--                ('X', 'X', 'X', 'X')),  --| '1'  'X' |
--           (('U', 'U', '0', 'U'),   --| 'U' '0' |
--            ('U', 'X', '0', 'X'),   --| 'X' '0' |
--                ('U', 'X', '0', '1'),   --| '0'  '0' |
--                ('0', '0', '0', '0')),  --| '1'  '0' |
--           (('U', 'U', 'U', '1'),   --| 'U' '1' |
--            ('U', 'X', 'X', '1'),   --| 'X' '1' |
--                ('U', 'X', '0', '1'),   --| '0'  '1' |
--                ('1', '1', '1', '1')));--| '1'  '1' |
--
--   BEGIN
--          RETURN tbl_MUX2x1(Input1, Sel, Input0);
--   END fun_MUX2x1;
```

```
-- Reduction of number of bits for signed number multiplication result
variables
-- From argument'length to nb_var (length of vaiables in the model)

FUNCTION quant(a : SIGNED) RETURN dataType IS
VARIABLE s : dataType;
BEGIN

      -- Bonne fonction optimisee !! A verifier

      s(nb_var-2 downto 0) := a(a'LEFT-2 downto a'LEFT-nb_var);

      RETURN s;

END quant;


-- Reduction of number of bits for signed number multiplication result
variables
-- From argument'length to nb_var (length of vaiables in the model)

FUNCTION quantif(a : SIGNED) RETURN dataType IS
CONSTANT length_a : INTEGER := a'LENGTH;
VARIABLE s : dataType;
VARIABLE temp1 : UNSIGNED(length_a-2 downto 0);
VARIABLE temp2 : UNSIGNED(nb_var-2 downto 0);
BEGIN
      s(nb_var-1) := a(a'LEFT);


-- Bonne fonction optimisee !! A verifier
--
-- s(nb_var-2 downto 0) := a(a'LEFT-2 downto a'LEFT-nb_var);

--  RETURN s;

      IF a(a'LEFT) = '0' OR a(a'LEFT) = 'L' THEN

                  -- Positive value
            s(nb_var-2 downto 0) := a(length_a-3 downto length_a-1-
nb_var);
            RETURN s;

      ELSIF a(a'LEFT) = '1' OR a(a'LEFT) = 'H' THEN

            -- Negative Value
            temp1 := UNSIGNED(a(length_a-2 downto 0));
            temp1 := not_wise(temp1 - "1");
            temp2 := temp1(length_a-3 downto length_a-1-nb_var);
            temp2 := not_wise(temp2) + "1";
            s(nb_var-2 downto 0) := SIGNED(temp2);

--          If the modulus part of the number is "00...00" then put the
sign bit to '0'
--          to avoid having the overflowing number "100...00"
--
```

```
            IF temp2 = UNSIGNED(c0) THEN
                  s(nb_var-1) := '0';
            END IF;

            RETURN s;
      END IF;

END quantif;



--
-- Function sqrt1 : Approximation of SQRT by a 3rd order polynomial
--
-- Input x is between [1,2[
-- x assumes to be in the shape "1----...---"
-- denoting that the MSB is the interger part
-- and the others bits form the decimal part
-- The output is in the same shape

      FUNCTION sqrt1(x : UNSIGNED) RETURN UNSIGNED IS
      VARIABLE tempx, x2, temp1, tempr1, extp1 : dataType;
      VARIABLE tempr2, r                              : UNSIGNED(nb_var-
1 DOWNTO 0);
      VARIABLE tempr3                                 : UNSIGNED(nb_var
DOWNTO 0);

      BEGIN

--          r := p1 + p2*x + p3*x*x + p4*x*x*x;

            tempx := SIGNED(shift_right(x,2));
                                    -- Scaling : divide x by 4. The
leading zero becomes the sign bit

                                                                        -
- Cofficients pi are computed for the interval [1/4 1/2[
            x2 := quant(tempx*tempx);
            temp1 := quant(p4*tempx);

                  -- Extension p1'length to nb_var
                  -- !! Warning This concatenation is done only because
p1 is positive
            extp1 := (OTHERS => '0');
            extp1(extp1'LEFT DOWNTO extp1'LEFT-p1'LEFT) := p1;
            ---
                                                            -- p2 = 1
enlever dans multiplication
            tempr1 := extp1 + quant(p2*tempx) + quant(p3*x2) +
quant(temp1*x2);
            tempr2 := shift_left(UNSIGNED(tempr1),1);
                              -- Inverse Scaling : Multiply r by 2
            tempr3 := reduce(tempr2*p_nor, nb_var+1);
                              -- Multiplication of the result by p_nor
(The normalization factor)
```

```
            r := tempr3(tempr3'LEFT-1 DOWNTO 0);
                                         -- The MSB is supposed to always be
'0'. r is output without it

            RETURN r;

      END sqrt1;


-- Table for Approximation of the inversion
-- Returns an approximation of 1/x with 1<=x<2

      FUNCTION table_div(x : UNSIGNED) RETURN UNSIGNED IS
      VARIABLE r : UNSIGNED(nb_init-1 DOWNTO 0);

      BEGIN

                                                -- Structure of constant x:
            IF x>=c1p0 AND x<c1p1 THEN
                  -- MSB = integer part
                  r := "11110";
                  RETURN r    ;
      -- 1/1.05         -- Others = decimal part

            ELSIF x>=c1p1 AND x<c1p2 THEN
            -- Therefore : "10001" = 1.0001
                  r := "11011";
                  RETURN r;                                                   -
- 1/1.15          --  Structure of returned constant:
            ELSIF x>=c1p2 AND x<c1p3 THEN
            --     All bits are decimal part.
                  r := "11001";
                  RETURN r;                                                   -
- 1/1.25          --  Therefore "10001" = .10001
            ELSIF x>=c1p3 AND x<c1p4 THEN
                  r := "10111";
                  RETURN r;                                                   -
- 1/1.35
            ELSIF x>=c1p4 AND x<c1p5 THEN
                  r := "10110";
                  RETURN r;                                                   -
- 1/1.45
            ELSIF x>=c1p5 AND x<c1p6 THEN
                  r := "10100";
                  RETURN r;                                                   -
- 1/1.55
            ELSIF x>=c1p6 AND x<c1p7 THEN
                  r := "10011";
                  RETURN r;                                                   -
- 1/1.65
            ELSIF x>=c1p7 AND x<c1p8 THEN
                  r := "10010";
                  RETURN r;                                                   -
- 1/1.75
            ELSIF x>=c1p8 AND x<c1p9 THEN
                  r := "10001";
                  RETURN r;                                                   -
- 1/1.85
```

```
            ELSIF x>=c1p9 AND x<=c2p0m THEN
                  r := "10000";
                  RETURN r;                                              -
- 1/1.95
            ELSE                                                      --
!!!!!!!!!!!
                  r := "XXXXX";
                  RETURN r;
            END IF;

      END table_div;


-- Approximation of the inversion
-- x is assumed to be 1<=x<2
--

      FUNCTION inversion(x : UNSIGNED) RETURN UNSIGNED IS
      VARIABLE l, nb_bit      : NATURAL;
      VARIABLE r                    : UNSIGNED(nb_init*2**nb_iter-1
DOWNTO 0);

      ATTRIBUTE dont_unroll OF iterations : LABEL IS true;

      BEGIN

            -- synopsys synthesis_off

            ASSERT x(x'LEFT) = '1' OR x(x'LEFT) = 'H'
                  REPORT "inversion : input argument not in the interval
[1, 2[. The MSB (the integer part) must be '1' !"
            SEVERITY ERROR;

            -- synopsys synthesis_on

            nb_bit := nb_init;

            r(nb_bit-1 DOWNTO 0) := table_div(x);
                                    -- Returns an approximation of 1/x in
the form

                                                                         -
- "10110" = 0.10110
            l := nb_bit;
            r(nb_bit) := '0';

            iterations : FOR j IN 1 TO nb_iter LOOP
                  r(2*nb_bit-1 DOWNTO 0) :=
reduce(r_extend(shift_left(r(l DOWNTO 0),1),2*l+x'LENGTH)

            - r(l-1 DOWNTO 0)*r(l-1 DOWNTO 0)*x,

            2*nb_bit);
                  nb_bit := nb_bit*2;
                  l := nb_bit-1;

            END LOOP iterations;
```

```
            RETURN r(r'LENGTH-2 DOWNTO 0);
                                          -- Return r without the leading
zero

                                                                        -
- The output is in the form "1-----" = .1-------
      END inversion;



--
-- Division of two signed numbers
-- This function assumes that -1 < num,den < 1
-- This function also assumes that num<=den so that the result would
always be <= 1
--

      FUNCTION division(num,den : SIGNED) RETURN SIGNED IS

      CONSTANT N : NATURAL := 6;
      VARIABLE tempden0, tempnum0, quot : dataType;
      VARIABLE tempden, tempnum, tempden1 : UNSIGNED(nb_var-1 DOWNTO 0);
      VARIABLE invden : UNSIGNED(38 DOWNTO 0);
      VARIABLE tempr1 : UNSIGNED(num'LENGTH-1+invden'LENGTH-1 DOWNTO 0);
      VARIABLE tempr2 : UNSIGNED(nb_var+N-1 DOWNTO 0);
      VARIABLE tempr3 : UNSIGNED(tempr2'LEFT DOWNTO N+1);
      VARIABLE i : NATURAL;
      VARIABLE sign : STD_LOGIC;
      VARIABLE res0 : UNSIGNED(2*nb_var+N-2 DOWNTO 0);
      VARIABLE res : SIGNED(2*nb_var+N-1 DOWNTO 0);

      ATTRIBUTE dont_unroll OF reduction : LABEL IS true;

      BEGIN


            IF num = den THEN
                                    -- If the num the numerator end the
denominator are equal
                  quot := (OTHERS => '1');
                  quot(quot'LEFT) := '0';
--                quot := "0111111111111111111";
                        -- the result is 1.


            ELSIF num = -den THEN
                              -- If their are opposite, the result is -1
                  quot := (OTHERS => '0');
                  quot(quot'LEFT) := '1';
                  quot(quot'RIGHT) := '1';
--                quot := "1000000000000000001";

            ELSE

                  tempden0 := ABS(den);
                  tempnum0 := ABS(num);
```

```
                  -- synopsys synthesis_off

                  ASSERT tempnum0 <= tempden0
                        REPORT "Division : ABS(num) > ABS(den) ! This
function works for ABS(num) <= ABS(den) and returns q in ]-1, 1[ !"
                        SEVERITY ERROR;

                  ASSERT den /= c0
                        REPORT "Division : WARNING ! Division by zero"
                        SEVERITY WARNING;

                  -- synopsys synthesis_on

                  tempden := UNSIGNED(tempden0);
                                                            -- With this
conversion, the sign bit becomes the integer
                  tempnum := UNSIGNED(tempnum0);
                                                            -- part of
the variable tempden.
                  tempden1 := tempden;
                                                                        -
- Therefore "010010" is taken to be 0.10010

--     Reduction of tempden so that 1 <= tempden < 2
-- Equivalent to multiply by 2^i

                  i := 0;

                  reduction : WHILE NOT(tempden(tempden'LEFT)='1' OR
tempden(tempden'LEFT)='H') LOOP
                        i := i + 1;
                        tempden := SHIFT_LEFT(tempden,1);
                  END LOOP reduction;

                  invden := inversion(tempden);
                  tempr1 := tempnum(num'LENGTH-2 DOWNTO 0)*invden;
                                          -- num is taken without the
leading zero

--
-- Inverse reduction
-- The result is left shifted i times
-- Equivalent to multiply by 2^i
--

                  IF i /= 0 THEN
                        tempr1 := SHIFT_LEFT(tempr1,i);
--                      tempr1 := SHIFT_LEFT(tempr1,3);
                  END IF;

--
-- q is taken with N more bits of precesion
--
                  tempr2 := tempr1(tempr1'LENGTH-1 DOWNTO tempr1'LENGTH-
nb_var-N);

--
-- Comparison of q*den to num
```

```
--
                    res0 := tempr2*tempden1(den'LENGTH-2 DOWNTO 0);
                    res := SIGNED('0' & res0) -
SIGNED(r_extend(tempnum,2*nb_var+N));

                    IF  res(res'LEFT) = '1' OR res(res'LEFT) = 'H' THEN
                          -- Negative value
                          tempr2 := tempr2 + "1";
                    ELSIF res(res'LEFT) = '0' OR res(res'LEFT) = 'L' THEN
                          -- Positive value
                          tempr2 := tempr2 - "1";
                    END IF;

                    tempr3 := tempr2(tempr2'LEFT DOWNTO N+1);
                    sign := den(den'LEFT) XOR num(num'LEFT);
                    IF sign = '0' OR sign = 'L' THEN
                          quot := SIGNED('0' & tempr3);
                    ELSIF sign = '1' OR sign = 'H' THEN
                          tempr3 := not_wise(tempr3) + "1";
                          IF tempr3 = c0 THEN
                                                          -- If the absolute
value of the negative number is null, put the sign to '0'
                                sign := '0';
                                                                -- to avoid
the overflowing number "100...00"
                          END IF;
                          quot := SIGNED(sign & tempr3);
                                                -- Convert to 2's complement
                    END IF;
              END IF;

              RETURN quot;

        END division;



-- Reduction of the number of bits in a variable to n bits

        FUNCTION reduce(x : UNSIGNED; n : INTEGER) RETURN UNSIGNED IS
        BEGIN
              RETURN x(x'LEFT DOWNTO x'LEFT-n+1);
        END reduce;




--      FUNCTION reduce(x : SIGNED; n : INTEGER) RETURN SIGNED IS
--      BEGIN
--            RETURN x(x'LEFT DOWNTO x'LEFT-n+1);
--      END reduce;



-- Extend of the number of bits in a variable to n bits
-- adding zeroes to the right
```

```
      FUNCTION r_extend(x : UNSIGNED; n : INTEGER) RETURN UNSIGNED IS
      VARIABLE r : UNSIGNED(n-1 DOWNTO 0);
      BEGIN
            r := (OTHERS => '0');
            r(n-1 DOWNTO n-x'LENGTH) := x;
            RETURN r;
      END r_extend;


-- Extend of the number of bits in a variable to n bits
-- adding zeroes to the left

      FUNCTION l_extend(x : UNSIGNED; n : INTEGER) RETURN UNSIGNED IS
      VARIABLE r : UNSIGNED(n-1 DOWNTO 0);
      BEGIN
            r := (OTHERS => '0');
            r(x'LENGTH-1 DOWNTO 0) := x;
            RETURN r;
      END l_extend;


      FUNCTION XSLL(arg:STD_LOGIC_VECTOR; count : NATURAL) RETURN
STD_LOGIC_VECTOR IS
    CONSTANT arg_l:INTEGER := arg'LENGTH-1;
    VARIABLE result : STD_LOGIC_VECTOR(arg_l DOWNTO 0); -- := (others =>
'0');
  BEGIN

       result := (others => '0');

    IF count <= arg_l THEN
            result(arg_l DOWNTO count) := arg(arg_l-count DOWNTO 0);
    END IF;
    RETURN result;
  END XSLL;

  FUNCTION XSRL(arg : STD_LOGIC_VECTOR; COUNT : NATURAL) RETURN
STD_LOGIC_VECTOR IS
    CONSTANT ARG_L : INTEGER := arg'LENGTH-1;
    VARIABLE result : STD_LOGIC_VECTOR(arg_l DOWNTO 0); -- := (others =>
'0');

  BEGIN
       result := (others => '0');

    IF count <= arg_l THEN
            result(arg_l-count DOWNTO 0) := arg(arg_l DOWNTO count);
    END IF;
    RETURN result;
  END XSRL;



  FUNCTION shift_left (arg : UNSIGNED; count : NATURAL) RETURN UNSIGNED
IS
  BEGIN
    IF (arg'LENGTH < 1) THEN RETURN NAU;
    END IF;
```

```
    RETURN UNSIGNED(XSLL(STD_LOGIC_VECTOR(arg), count));
  END shift_left;


  FUNCTION shift_right (arg: UNSIGNED; count: NATURAL) RETURN UNSIGNED IS
  BEGIN
    IF (arg'LENGTH < 1) THEN RETURN NAU;
    END IF;
    RETURN UNSIGNED(XSRL(STD_LOGIC_VECTOR(arg), count));
  END shift_right;


-- Multiplication of a signed number by A =  2^count.
-- arg assume to be < 1. Exp: "00111111" = .0111111
-- The MSB is the sign bit

      FUNCTION mshift_left (arg : SIGNED; count : NATURAL) RETURN SIGNED
IS
      VARIABLE r1 : UNSIGNED(arg'LENGTH-2 DOWNTO 0);
      VARIABLE r : SIGNED(arg'LENGTH-1 DOWNTO 0);
      BEGIN
    IF arg(arg'LEFT)='0' OR arg(arg'LEFT) = 'L' THEN
                                         -- Positive value
            r1 := UNSIGNED(arg(arg'LEFT-1 DOWNTO 0));
            r1 := shift_left(r1, count);
            r := SIGNED('0' & r1);
       ELSIF arg(arg'LEFT)='1' OR arg(arg'LEFT) = 'H' THEN
            r1 := UNSIGNED(arg(arg'LEFT-1 DOWNTO 0));
            r1 := not_wise(r1 - "1");
                                                                       -
- Convert to sign and modulus
            r1 := shift_left(r1,count);
            r1 := not_wise(r1) + "1";
                                                                       -
- Convert to 2's complement
            r := SIGNED('1' & r1);
       END IF;
       RETURN r;
  END mshift_left;


-- Division of a signed number by A =  2^count.
-- arg assume to be < 1. Exp: "00111111" = .0111111
-- The MSB is the sign bit

      FUNCTION mshift_right (arg : SIGNED; count : NATURAL) RETURN SIGNED
IS
      VARIABLE r1 : UNSIGNED(arg'LENGTH-2 DOWNTO 0);
      VARIABLE r : SIGNED(arg'LENGTH-1 DOWNTO 0);
      BEGIN
    IF arg(arg'LEFT)='0' OR arg(arg'LEFT)='L' THEN
                                               -- Positive value
            r1 := UNSIGNED(arg(arg'LEFT-1 DOWNTO 0));
            r1 := shift_right(r1, count);
            r := SIGNED('0' & r1);
       ELSIF arg(arg'LEFT)='1' OR arg(arg'LEFT)='H' THEN
            r1 := UNSIGNED(arg(arg'LEFT-1 DOWNTO 0));
```

```
            r1 := not_wise(r1 - "1");

- Convert to sign and modulus
            r1 := shift_right(r1,count);
            r1 := not_wise(r1) + "1";

- Convert to 2's complement
            IF r1 = UNSIGNED(c0) THEN

- If the moduls part is "00...00" then put the sign to '0'
                  r(r'LEFT) := '0';

      -- to avoid having the overflowing number "1000...000"
            ELSE
                  r(r'LEFT) := '1';
            END IF;
            r(r'LEFT-1 DOWNTO 0) := SIGNED(r1);
       END IF;
       RETURN r;
  END mshift_right;



-- Apply rotation on of-diagonal cells

      PROCEDURE gen_of_diag(ain, r, cin, sin          : IN SIGNED;
                        cout, sout, rout, aout   : OUT SIGNED) IS
      BEGIN

-- Modifier ici pour augmenter precision

            rout := quant(cin*r) + quant(sin*ain);
            -- rout := quant(cin*r + sin*ain);

            aout := quant(-sin*r) + quant(cin*ain);
      --aout := quant(-sin*r + cin*ain);



            cout := cin;
            sout := sin;

      END gen_of_diag;


-- Generate rotation in diagonal cells

      PROCEDURE gen_diag(ain, r      : IN SIGNED;
                               c, s, rout       : OUT SIGNED) IS
      VARIABLE t, extc1, extc0, abs_a, abs_r, temps, tempc : dataType;
      BEGIN

            abs_a := ABS(ain);
            abs_r := ABS(r);

            -- Augmentation du nombre de bits des constantes c1 etv c0
            extc1 := (OTHERS => '1');
```

```
            extc1(nb_var-1) := '0';
            extc0 := (OTHERS => '0');

            IF ain = c0 THEN

                  tempc := extc1;
                  c := tempc;
                                                            -- c = 1

                  temps := extc0;
                                                      -- s = 0
                  s := temps;

            ELSIF abs_a >= abs_r THEN
                  t := division(r,ain);

                  temps := SIGNED('0' & reduce(inversion(sqrt1(c1p0 +
UNSIGNED(quant(t*t)))), nb_var-1));       -- With this transformation,
the leading '0'
                  tempc := quant(temps*t);

                                                                              --
becomes the interger part

                  s := temps;
                  c := tempc;
            ELSE
                  t := division(ain,r);

                  -- Constants c025 and c05 are added normalization
                  -- The result 'c' remainds the same.
--                temp_c := temp_c05/sqrt1(c025 +
quant(quant(c025*t)*t));

            tempc := SIGNED('0' & reduce(inversion(sqrt1(c1p0 +
UNSIGNED(quant(t*t)))), nb_var-1));
                  temps := quant(tempc*t);

                  s := temps;
                  c := tempc;
            END IF;

-- Modifier ici pour augmenter precision

            rout := quant(tempc*r) + quant(temps*ain);

      END gen_diag;


      -- MAC
      FUNCTION mac1(x,y,z : SIGNED) RETURN SIGNED IS
      VARIABLE m : dataType;
      BEGIN
            m := quant(x*y) + z;
            RETURN m;
      END mac1;
```

```
        -- MAC and sub
       FUNCTION mac2(w,x,y,z : SIGNED) RETURN SIGNED IS
       VARIABLE m : dataType;
       BEGIN
             m := quant(w*x) + y - z;
             RETURN m;
       END mac2;


END packages;
-- -------------------------------------------------------------------------
-------
--
-- File : diag.vhd
--          Entity and architecture for the circle PEs (diagonal PEs)
--
--
-- Modelisation of an array processors for Adaptive Channel Equalization
-- based on square root covariance Kalman Filtering using Givens
rotations.
--
-- Author :
--               Aurelien T. Mozipo
--               March 30, 1999
--
-- -------------------------------------------------------------------------
--------

LIBRARY IEEE, lib;
      use ieee.std_logic_1164.all;
      use ieee.std_logic_arith.all;
      use ieee.std_logic_signed.all;

      use lib.constants.all;
      use lib.type_def.all;
      use lib.packages.all;


ENTITY diag IS
      PORT(clk                                  : IN clkType;
              Rst                               : IN rstType;
              mode                              : IN modeType;
              initin                            : IN initType ;
              datain                            : IN dataType ;
              cout, sout                      : OUT dataType ;
              initout                         : OUT initType;
              dataout_dn                      : OUT dataType
             );
END diag;


-- Architecture for PE(1,1) only

ARCHITECTURE behav_line1 OF diag IS
BEGIN
PROCESS
```

```
    VARIABLE reg, tempreg, tempc, temps : dataType;

    BEGIN

          WAIT UNTIL (clk'EVENT AND clk = '1');

          initout <= initin;

          CASE mode IS

                WHEN mode2 =>
                            -- O is output to the right (Step 2)

                      cout <= (OTHERS => '0');

                WHEN mode5 =>
                      dataout_dn <= reg;

                WHEN OTHERS =>

                      IF (mode = mode1) OR (mode = mode4) THEN
          -- Triangularization    (Step4)

                            IF (initin = '1') OR (initin = 'H') THEN
                                  reg := datain;
                            ELSIF (initin = '0') OR (initin = 'L') THEN
                                        tempreg := reg;
                                        gen_diag(ain => datain, r =>
tempreg,
                            c => tempc, s => temps, rout => reg);
                                        cout <= tempc;
                                        sout <= temps;
                            END IF;
                            IF mode = mode4 THEN
                                  -- Data output to the buttom (end of
Step 5)

                                  dataout_dn <= reg;
                            END IF;
                      ELSE
--                          cout <= (OTHERS => 'W');
--                          sout <= (OTHERS => 'W');
                            initout <= '0';
--                          dataout_dn <= (OTHERS =>'W');
                      END IF;

          END CASE;

END PROCESS;

END behav_line1;


-- Architecture for PE(2,2,) PE(3,3) and PE(4,4)

ARCHITECTURE behav_lineX OF diag IS

BEGIN
```

```
PROCESS
      VARIABLE tempreg, reg  : dataType;
      VARIABLE tempc  : dataType;
--    VARIABLE tempdatain : dataType;
      VARIABLE temps : dataType;

      BEGIN

            WAIT UNTIL (clk'EVENT AND clk = '1');

--          tempdatain := datain;
            initout <= initin;

            IF Rst = '1' THEN
                        -- For PE22, PE33 Initialization of the internal
register reg to the
                  reg := (OTHERS => '0');
                  -- initial values of S+(1,1), S+(2,2) respectively
                  reg(nb_var-1 DOWNTO nb_var-nb_const) := one_nor;
            END IF;
                                    -- This can be done at the very first
step


            CASE mode IS

                  WHEN mode3 =>
                              -- Data output to the right (Step5)
                        cout <= reg;

                  WHEN mode2 =>
                              -- Multiplication and accumulation (Step3)
                        cout <= quant(datain*reg);
            -- + 0 -- Left input is null for  thess processors

                                                -- The value is output on
the right (bus 'cout') -- (Step3)
                  WHEN mode5 =>
                        dataout_dn <= reg;

                  WHEN OTHERS =>

                        IF (mode = mode1) OR (mode = mode4) THEN
      -- Triangularization          (Step1)

                              IF (initin = '1') OR (initin = 'H') THEN
                                    reg := datain;
                              ELSIF (initin = '0') OR (initin = 'L') THEN
                                    tempreg := reg;
                                    gen_diag(ain => datain, r => tempreg,
                                          c => tempc, s => temps, rout =>
reg);
                                    cout <= tempc;
                                    sout <= temps;

                                    IF mode = mode4 THEN
                                          dataout_dn <= reg;
```

```
                                        END IF;
                                END IF;
                        ELSE
--                              cout <= (OTHERS => 'W');

--                              sout <= (OTHERS => 'W');
                                initout <= '0';
--                              dataout_dn <= (OTHERS => 'W');
                        END IF;

        END CASE;

END PROCESS;

END behav_lineX;
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■



-- -------------------------------------------------------------------------
-------
--
-- File : ofdiag.vhd
--              Entities an architectures for squared PEs (of diagonal PEs)
--
-- Modelisation of systolic array processors for Adaptive Channel
Equalization
-- based on square root covariance Kalman Filtering using Givens
rotations.
--
-- Author :
--                      Aurelien T. Mozipo
--                      March 30, 1999
--
-- -------------------------------------------------------------------------
--------


LIBRARY IEEE, lib;
        use ieee.std_logic_1164.all;
        use ieee.std_logic_arith.all;
        use ieee.std_logic_signed.all;

        use lib.constants.all;
        use lib.type_def.all;
        use lib.packages.all;


-- Entity for PE(1,2:3)

ENTITY ofdiag1 IS
        PORT(clk                                                : IN clkType;
                Rst                                             : IN rstType;
                mode                                            : IN
modeType;
```

```
                initin                                          : IN initType
;
                datain                                          : IN dataType
;
                c, s                                            : IN dataType
;
                sout                                            : OUT
dataType;
                cout                                            : OUT
dataType;
                initout_rt, initout_dn      : OUT initType;
                dataout_dn                              : OUT dataType
               );
END ofdiag1;


LIBRARY IEEE, lib;
      use ieee.std_logic_1164.all;
      use ieee.std_logic_arith.all;
      use ieee.std_logic_signed.all;

      use lib.constants.all;
      use lib.type_def.all;
      use lib.packages.all;


-- Entity for PE(1,4)

ENTITY ofdiag14 IS
      PORT(clk                                                  : IN clkType;
                Rst                                             : IN rstType;
                mode                                            : IN
modeType;
                initin                                          : IN initType
;
                datain                                          : IN dataType
;
                c, s                                            : IN dataType
;
                z                                               : IN
dataType;
                sout                                            : OUT
dataType;
                cout                                            : OUT
dataType;
                initout_rt, initout_dn      : OUT initType;
                dataout_dn                              : OUT dataType
               );
END ofdiag14;


LIBRARY IEEE, lib;
      use ieee.std_logic_1164.all;
      use ieee.std_logic_arith.all;
      use ieee.std_logic_signed.all;

      use lib.constants.all;
      use lib.type_def.all;
```

```
      use lib.packages.all;



-- Entity for PE(2:3,:)

ENTITY ofdiagX IS
      PORT(clk                                             : IN clkType;
              Rst                                     : IN rstType;
              mode                                         : IN
modeType;
              initin                                       : IN initType
;
              datain                                       : IN dataType
;
              c, s                                         : IN dataType
;
              cout, sout                              : OUT dataType;
              initout_rt, initout_dn      : OUT initType;
              dataout_dn                              : OUT dataType
             );
END ofdiagX;


-- Architecture for PE(1,2) and PE(1,3)

ARCHITECTURE behav_ofdiag12 OF ofdiag1 IS

BEGIN

main : PROCESS
      VARIABLE tempdatain, temps, tempc, temp1, temp2, reg1, tempreg2,
reg2, tempcout, tempsout, tempdataout_dn  : dataType;

      VARIABLE tempinitin : initType;
      VARIABLE i : NATURAL;

      BEGIN

            WAIT UNTIL (clk'EVENT AND clk ='1');

            tempdatain := datain;
            tempc := c;
            temps := s;

            IF Rst = '1' THEN
                                -- Initialization of the internal register
reg1 to the
                  reg1 := (OTHERS => '0');
                        -- value of x(1) = phi * x+(0) = (0,0,0)'
            END IF;
                                                    -- This can be done at the very
first step when mode = mode21

            CASE mode IS
```

```
                    WHEN mode1 =>
                                        -- Data pass through (Step 1)
                          IF Rst = '1' THEN
                                dataout_dn <= (OTHERS => '0');
                                dataout_dn(nb_var-1 DOWNTO nb_var-nb_const)
<= one_nor;         -- S+(1,1)init
                          ELSE
                                dataout_dn <= datain;
                                      -- S+(1,1)
                          END IF;

                          initout_dn <= initin;
                          -- Send initin for triangularization in PE22



                    WHEN mode2 =>
                                        -- MAC (Step 2)

-- Possibilite de mModifier cette etape pour PE12
-- Pour diminuer un cycle d'horloge
-- tempc = 0.
-- cout <= quant(tempdatain*reg1);
-- (1)

                          cout <= mac1(tempdatain, reg1, tempc);
            -- Data are fed in from the right on "c" bus
                          dataout_dn <= tempdatain;
                          --

                    WHEN mode3 =>
                                        -- Triangularization  (Step 4)
                          IF initin = '1' OR initin = 'H' THEN
                                reg2 := datain;
                                tempinitin := '0';
                          ELSIF initin = '0' OR initin = 'L' THEN
                                tempreg2 := reg2;
                                gen_of_diag(ain => datain, r => tempreg2,
cin => tempc, sin => temps,
                                                          cout => tempcout,
sout => tempsout, rout => reg2, aout => tempdataout_dn);
                                cout <= tempcout;
                                sout <= tempsout;
                                dataout_dn <= tempdataout_dn;
                          END IF;
                          initout_rt <= initin;
                          -- Output of '1' to initout_rt with speed 1

                          initout_dn <= tempinitin;
                          -- Output of '1' to initout_dn with a speed 1/2
                          tempinitin := initin;

                    WHEN mode4 =>
                                        -- F in fed into the proc. front the
up (Step 5)
```

```
                                                -- and the containt of
register is divide by F
                        temp1 := ABS(reg2);
                        temp2 := ABS(tempdatain);
                        i := 0;
                        WHILE temp1>temp2 LOOP
                              i := i+1;
                              reg2 := mshift_right(reg2,1);
                  -- Normalization of reg2 before the division to ensure
that reg2<=datain : division by 4
                              temp1 := ABS(reg2);
                        END LOOP;
                        reg2 := division(reg2,tempdatain);

                  WHEN mode5 =>
                                    -- I is fed, then computation of x+

--                      Also divide reg1 by 2^i
                        reg1 := mshift_right(reg1,i);

                        reg1 := mac1(-reg2, tempdatain, reg1);
            -- x+ = x - (G/F)*I , result into reg1 (x is overriden)

--                      Inverse normalization : multiply reg1 by 2^fgnor
                        reg1 := mshift_left(reg1,i);

                  WHEN mode6 =>
                                    -- x+ is shifted on the right to form
x (Step 6)

                                                -- If start state (Mode
reset), no shift needed
                        cout <= reg1;
                        reg1 := c;

                  WHEN mode7 =>
                                    -- Valid only for PE(1,2)  : x1 is
shifted
                        cout <= reg1;
                                    -- to the right but remainds is reg1
(Step 6)

                  WHEN OTHERS =>
--                      sout <= (OTHERS => 'W');
--                      cout <= (OTHERS => 'W');
                        initout_rt <= '0';
                        initout_dn <= '0';
--                      dataout_dn <= (OTHERS => 'W');
            END CASE;

      END PROCESS main;

END behav_ofdiag12;


-- Architecture For PE(1,4) only
```

```
ARCHITECTURE behav_ofdiag14 OF ofdiag14 IS
BEGIN

main : PROCESS
      VARIABLE tempdatain, temp1, temp2, temps, tempc, reg1, tempreg2,
reg2, tempz, tempcout, tempsout, tempdataout_dn       : dataType;
      VARIABLE tempinitin : initType;
      VARIABLE i : NATURAL;
      BEGIN

            WAIT UNTIL (clk'EVENT AND clk ='1');

            tempdatain := datain;
            tempc := c;
            temps := s;
            tempz := z;

            IF Rst = '1' THEN
                                    -- Initialization of the internal
register reg1 to the
                  reg1 := (OTHERS => '0');
                              -- value of x(1) = phi * x+(0) = (0,0,0)'
                  cout <= (OTHERS => 'Z');
                              -- x_hat is put at hi-Z at initial
            END IF;
                                                -- This can be done at
the very first step when mode = mode1

            CASE mode IS
                  WHEN mode1 =>
                                          -- Data pass through (Step 1)
                        dataout_dn <= datain;

                  WHEN mode2 =>
                                          -- MAC and SUB (Step 2)
                        cout <= mac2(tempdatain, reg1, tempc, tempz);
      -- Data are fed in from the right on "c" bus
                        dataout_dn <= tempdatain;

                  WHEN mode3 =>
                                          -- Triangularization  (Step 4)
                        IF initin = '1' OR initin = 'H' THEN
                              reg2 := datain;
                              tempinitin := '0';
                        ELSIF initin = '0' OR initin = 'L' THEN
                              tempreg2 := reg2;
                              gen_of_diag(ain => datain, r => tempreg2,
cin => tempc, sin => temps,
                                                      cout => tempcout,
sout => tempsout, rout => reg2, aout => tempdataout_dn);
--                            cout <= tempcout;
--                            sout <= tempsout;
                              dataout_dn <= tempdataout_dn;
                        END IF;

                        initout_rt <= initin;
                              -- Output of '1' to initout_rt with speed 1
```

```
                        initout_dn <= tempinitin;
                              -- Output of '1' to initout_dn with speed
1/2
                        tempinitin := initin;

                  WHEN mode4 =>
                                          -- F in fed into the proc.
front the up (Step 5)

                                                      -- and the containt
of register is divide by F
                        temp1 := ABS(reg2);
                        temp2 := ABS(datain);
                        i := 0;
                        WHILE temp1>temp2 LOOP
                              i := i+1;
                              reg2 := mshift_right(reg2,1);
                              -- Normalization of reg2 before the
division to ensure that reg2<=datain : division by 4
                              temp1 := ABS(reg2);
                        END LOOP;

                        reg2 := division(reg2,datain);


                  WHEN mode5 =>
                                          -- I is fed, then computation
of x+
--                      Also divide reg1 by 2^i
                        reg1 := mshift_right(reg1,i);

                        reg1 := mac1(-reg2, tempdatain, reg1);
                  -- x+ = x - (G/F)*I , result into reg1 (x is overriden)

--                      Inverse normalization : multiply reg1 by 2^i
                        reg1 := mshift_left(reg1,i);

                  WHEN mode6 =>
                                          -- x+ is shifted on the right
to form x (Step 6)
                        cout <= reg1;
                                          -- x_hat is output here

                                                      -- If start state,
no shift needed
                        reg1 := c;


--                WHEN mode7 =>
                                          -- Valid only for PE(1,4)   :
tempcout = mac2() is output
--                      cout <= reg3;
                                          -- to the right  (Step 2)

                  WHEN OTHERS =>
--                      sout <= (OTHERS => 'W');
--                      cout <= (OTHERS => 'W');
```

```
--                      initout_rt <= 'W';
--                      initout_dn <= 'W';
--                      dataout_dn <= (OTHERS => 'W');
            END CASE;

      END PROCESS main;
END behav_ofdiag14;




-- Architecture for PE(2,3), PE(2,4), PE(3,4)

ARCHITECTURE behav_ofdiagX OF ofdiagX IS
BEGIN

main : PROCESS
      VARIABLE tempdatain, temps, tempc, reg, tempcout, tempsout,
tempdataout_dn    : dataType;
      VARIABLE tempinitin : initType;
      BEGIN

            WAIT UNTIL (clk'EVENT AND clk ='1');

            tempdatain := datain;
            tempc := c;
            temps := s;

            IF Rst = '1' THEN
                                    -- For PE23, Initialization of the
internal register reg to the
                  reg := (OTHERS => '0');
                              -- initial value of S+(1,2)
            END IF;
                                                -- This can be done at
the very first step

            CASE mode IS
                  WHEN mode1 =>
                                          -- Triangularization (Step 1
and Step 4)
                        IF initin = '1' OR initin = 'H' THEN
                              reg := tempdatain;
                              tempinitin := '0';
                        ELSIF initin = '0' OR initin = 'L' THEN
                              gen_of_diag(ain => tempdatain, r => reg,
cin => tempc, sin => temps,
                                                cout => tempcout, sout =>
tempsout, rout => reg, aout => tempdataout_dn);
                              cout <= tempcout;
                              sout <= tempsout;
                              dataout_dn <= tempdataout_dn;
                        END IF;
                        initout_rt <= initin;
                              -- Output of '1' to initout_rt with speed 1
```

```
                        initout_dn <= tempinitin;
                              -- Output of '1' to initout_dn with a speed
1/2
                        tempinitin := initin;

                  WHEN mode2 =>
                                          -- MAC Formation of S'*h (Step
3)
                        cout <= mac1(reg, tempdatain, tempc);
                  -- Data are fed in from the up on "datain" bus
                        dataout_dn <= tempdatain;


                  WHEN mode3 =>
                                          -- PE(2,4) and PE(3,4) (Step5)
                        cout <= reg;
                                          -- PE(2,3) output of S  (Step
2)

                  WHEN mode4 =>
                                          -- Formation of S*phi (Step5)
                        cout <= reg + tempc;
                                    -- PE(2,3) S12 + S11

                  WHEN mode5 =>
                                          -- PE(2,4)
                        cout <= tempc;

                  WHEN OTHERS =>
--                      cout <= (OTHERS => 'W');
--                      sout <= (OTHERS => 'W');
                        initout_rt <= '0';
                        initout_dn <= '0';
--                      dataout_dn <= (OTHERS => 'W');

            END CASE;

      END PROCESS main;
END behav_ofdiagX;
```

■■■

```
-- -----------------------------------------------------------------
-------
--
-- File : covkal.vhd
--          Entity and architecture for the global processor
--
-- Modelisation of an array processors for Adaptive Channel Equalization
-- based on square root covariance Kalman Filtering using Givens
rotations.
--
-- Author :
--                Aurelien T. Mozipo
--                March 30, 1999
--
-- -----------------------------------------------------------------
--------
```

```
LIBRARY IEEE, lib;
      use ieee.std_logic_1164.all;
--    use ieee.std_logic_arith.all;
--    use ieee.std_logic_signed.all;

--    use lib.constants.all;
      use lib.type_def.all;
      use lib.components.all;

ENTITY covkal IS
      PORT (clk                                        : IN clkType;
                  reset                                      : IN rstType;
                  pause,
            ready                                      : IN STD_LOGIC;
            address                                    : OUT addrtype;
                  request                                    : OUT STD_LOGIC;
                  data                                       : INOUT dataType
               );
END covkal ;


ARCHITECTURE struct_covkal OF covkal IS
      SIGNAL datainpe11, datainpe12, datainpe13, datainpe14,
                  dataout_dnpe11, dataout_dnpe22,

      dataout_dnpe33, dataout_dnpe44,

      dataout_rtpe14, dataout_rtpe24,

      dataout_rtpe34, dataout_rtpe44, z   : dataType;
      SIGNAL state
                                                 : stateType;
      SIGNAL initpe11, initpe22
                            : initType;
      SIGNAL modepe11, modepe12, modepe13,
                  modepe14, modepe22, modepe23,
                  modepe24, modepe33, modepe34,
                  modepe44
                                                 : modeType;
      SIGNAL rst_to_array, clk_to_array
                         : STD_LOGIC;
BEGIN


      proc : array_proc PORT MAP(clk => clk, --clk_to_array,
                                                          rst => reset,
--rst_to_array,
                                                          initpe11 =>
initpe11,
                                                          initpe22 =>
initpe22,
                                                          modepe11 =>
modepe11, modepe12 => modepe12, modepe13 => modepe13,
                                                          modepe14 =>
modepe14, modepe22 => modepe22, modepe23 => modepe23,
```

```
                                                                modepe24 =>
modepe24, modepe33 => modepe33, modepe34 => modepe34,
                                                                modepe44 =>
modepe44,
                                                                datainpe11 =>
datainpe11, datainpe12 => datainpe12,
                                                                datainpe13 =>
datainpe13, datainpe14 => datainpe14,
                                                                zin => z,


     dataout_dnpe11 => dataout_dnpe11, dataout_dnpe22 => dataout_dnpe22,

     dataout_dnpe33 => dataout_dnpe33, dataout_dnpe44 => dataout_dnpe44,

     dataout_rtpe14 => dataout_rtpe14, dataout_rtpe24 => dataout_rtpe24,

     dataout_rtpe34 => dataout_rtpe34, dataout_rtpe44 => dataout_rtpe44
                                                                );

     mux : muxbloc PORT MAP( rst => reset,
                                                           state => state,
                                                           datain => data,

                                                           frompe11 =>
dataout_dnpe11, frompe22 => dataout_dnpe22, frompe33 => dataout_dnpe33,
frompe44 => dataout_rtpe44,         -- frompe11, frompe22, frompe33,
frompe44,
                                                           frompe14 =>
dataout_rtpe14, frompe24 => dataout_rtpe24, frompe34 => dataout_rtpe34,
                                                                      ---
frompe14, frompe24, frompe34
                                                           tope11 =>
datainpe11, tope12 => datainpe12, tope13 => datainpe13, tope14_up =>
datainpe14, tope14_rt => z,                     -- tope11, tope12,
tope13, tope14_up, tope14_rt
                                                           x_hat => data


                                                                        -
- x_hat
                                                           );

     csm : control PORT MAP( clk => clk,
                                                      pause => pause,
                                                      ready => ready,
                                                      reset => reset,
                                                      address => address,
                                                      clk_to_array =>
clk_to_array,
                                                      initpe11 => initpe11,
                                                      initpe22 => initpe22,
                                                      modepe11 => modepe11 ,
                                                      modepe12 => modepe12,
                                                      modepe13 => modepe13,
                                                      modepe14 => modepe14,
                                                      modepe22 => modepe22,
```

```
                                                            modepe23 => modepe23,
                                                            modepe24 => modepe24,
                                                            modepe33 => modepe33,
                                                            modepe34 => modepe34,
                                                            modepe44 => modepe44,
                                                            request => request,
                                                            rst_to_array =>
rst_to_array,
                                                            state_to_mux => state
   );

END struct_covkal ;


-- synopsys synthesis_off;


-- Configuration for simulation


CONFIGURATION conf_covkal OF covkal IS
      FOR struct_covkal
            FOR proc : array_proc USE CONFIGURATION lib.conf_array_proc;
            END FOR;

            FOR mux : muxbloc USE ENTITY lib.muxbloc(behav_muxbloc);
            END FOR;

            FOR csm : control USE ENTITY lib.control(data_flow);
            END FOR;
      END FOR;

END conf_covkal;

-- synopsys synthesis_on;
```

■■

```
-- -----------------------------------------------------------------------
-------
--
-- File : array_proc.vhd
--          Entity and architecture for the triangular array processor
--
-- Modelisation of an array processors for Adaptive Channel Equalization
-- based on square root covariance Kalman Filtering using Givens
rotations.
--
-- Author :
--               Aurelien T. Mozipo
--               March 30, 1999
--
-- -----------------------------------------------------------------------
--------


LIBRARY IEEE, lib;
```

```
      use ieee.std_logic_1164.all;

      use lib.type_def.all;
      use lib.components.all;


ENTITY array_proc IS
      PORT( clk                                                               :
IN clkType;
                  rst
      : IN rstType;
                  initpe11
      : IN initType;
                  initpe22
      : IN initType;
                  modepe11, modepe12, modepe13,
                  modepe14, modepe22, modepe23,
                  modepe24, modepe33, modepe34,
                  modepe44
      : IN modeType;
                  datainpe11, datainpe12,
                  datainpe13, datainpe14                          : IN
dataType;
                  zin
      : IN dataType;
                  dataout_dnpe11, dataout_dnpe22,
                  dataout_dnpe33, dataout_dnpe44,
                  dataout_rtpe14, dataout_rtpe24,
                  dataout_rtpe34, dataout_rtpe44       : OUT dataType
            );
END array_proc;


ARCHITECTURE struct_array_proc OF array_proc IS

TYPE data_type1 IS ARRAY(1 TO 3) OF dataType;
TYPE data_type2 IS ARRAY(1 TO 2) OF dataType;
TYPE init_type1 IS ARRAY(1 TO 3) OF initType;
TYPE init_type2 IS ARRAY(1 TO 2) OF initType;

SIGNAL s_sig1, c_sig1, data_sig1                          : data_type1;
SIGNAL s_sig2, c_sig2, data_sig2                          : data_type2;
SIGNAL s_sig3, c_sig3, data_sig3                          : dataType;
SIGNAL init_sig_rt1, init_sig_dn                          : init_type1;
SIGNAL init_sig_rt2
      : init_type2;
SIGNAL init_sig_rt3
      : initType;
SIGNAL tempinitpe22
      : initType;


BEGIN

      pe11 : diag PORT MAP(clk => clk, Rst => Rst, mode => modepe11,
initin => initpe11, datain => datainpe11,
```

```
                                                  cout => c_sig1(1), sout
=> s_sig1(1), initout => init_sig_rt1(1),
                                                  dataout_dn =>
dataout_dnpe11);

      pe12 : ofdiag1 PORT MAP(clk => clk, Rst => Rst, mode => modepe12,
initin => init_sig_rt1(1),
                                                        datain =>
datainpe12, c => c_sig1(1), s => s_sig1(1),
                                                        sout => s_sig1(2),
cout => c_sig1(2), initout_rt => init_sig_rt1(2),
                                                        initout_dn =>
init_sig_dn(1), dataout_dn => data_sig1(1));

      pe13 : ofdiag1 PORT MAP(clk => clk, Rst => Rst, mode => modepe13,
initin => init_sig_rt1(2),
                                                        datain =>
datainpe13, c => c_sig1(2), s => s_sig1(2),
                                                        sout => s_sig1(3),
cout => c_sig1(3), initout_rt => init_sig_rt1(3),
                                                        initout_dn => OPEN,
dataout_dn => data_sig1(2));

      pe14 : ofdiag14 PORT MAP(clk => clk, Rst => Rst, mode => modepe14,
initin => init_sig_rt1(3),
                                                         datain =>
datainpe14, c => c_sig1(3), s => s_sig1(3), z => zin,
                                                         sout => OPEN, cout
=> dataout_rtpe14, initout_rt => OPEN,
                                                         initout_dn =>
OPEN, dataout_dn => data_sig1(3));

                  tempinitpe22 <= initpe22 OR init_sig_dn(1);
      pe22 : diag PORT MAP(clk => clk, Rst => Rst, mode => modepe22,
initin => tempinitpe22, datain => data_sig1(1),
                                                  cout => c_sig2(1), sout
=> s_sig2(1), initout => init_sig_rt2(1),
                                                  dataout_dn =>
dataout_dnpe22);

      pe23 : ofdiagX PORT MAP(clk => clk, Rst => Rst, mode => modepe23,
initin => init_sig_rt2(1),
                                                        datain =>
data_sig1(2), c => c_sig2(1), s => s_sig2(1),
                                                        cout => c_sig2(2),
sout => s_sig2(2), initout_rt => init_sig_rt2(2),
                                                        initout_dn =>
init_sig_dn(2), dataout_dn => data_sig2(1));

      pe24 : ofdiagX PORT MAP(clk => clk, Rst => Rst, mode => modepe24,
initin => init_sig_rt2(2),
                                                        datain =>
data_sig1(3), c => c_sig2(2), s => s_sig2(2),
                                                        cout =>
dataout_rtpe24, sout => OPEN, initout_rt => OPEN,
                                                        initout_dn => OPEN,
dataout_dn => data_sig2(2));
```

```
      pe33 : diag PORT MAP(clk => clk, Rst => Rst, mode => modepe33,
initin => init_sig_dn(2), datain => data_sig2(1),
                                                cout => c_sig3, sout =>
s_sig3, initout => init_sig_rt3,
                                                dataout_dn =>
dataout_dnpe33);

      pe34 : ofdiagX PORT MAP(clk => clk, Rst => Rst, mode => modepe34,
initin => init_sig_rt3,
                                                    datain =>
data_sig2(2), c => c_sig3, s => s_sig3,
                                                    cout =>
dataout_rtpe34, sout => OPEN, initout_rt => OPEN,
                                                    initout_dn =>
init_sig_dn(3), dataout_dn => data_sig3);

      pe44 : diag PORT MAP(clk => clk, Rst => Rst, mode => modepe44,
initin => init_sig_dn(3), datain => data_sig3,
                                                cout => dataout_rtpe44,
sout => OPEN, initout => OPEN,
                                                dataout_dn =>
dataout_dnpe44);

END struct_array_proc;


-- Configuration for simulation

CONFIGURATION conf_array_proc OF array_proc IS
      FOR struct_array_proc

            FOR pe11 : diag USE ENTITY lib.diag(behav_line1);
            END FOR;

            FOR pe12 : ofdiag1 USE ENTITY lib.ofdiag1(behav_ofdiag12);
            END FOR;

            FOR pe13 : ofdiag1 USE ENTITY lib.ofdiag1(behav_ofdiag12);
            END FOR;

            FOR pe14 : ofdiag14 USE ENTITY lib.ofdiag14(behav_ofdiag14);
            END FOR;

            FOR pe22 : diag USE ENTITY lib.diag(behav_lineX);
            END FOR;

            FOR pe23 : ofdiagX USE ENTITY lib.ofdiagX(behav_ofdiagX);
            END FOR;

            FOR pe24 : ofdiagX USE ENTITY lib.ofdiagX(behav_ofdiagX);
            END FOR;

            FOR pe33 : diag USE ENTITY lib.diag(behav_lineX);
            END FOR;

            FOR pe34 : ofdiagX USE ENTITY lib.ofdiagX(behav_ofdiagX);
            END FOR;
```

```
            FOR pe44 : diag USE ENTITY lib.diag(behav_lineX);
            END FOR;

      END FOR;
END conf_array_proc;

■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

-- --------------------------------------------------------------------------
-------
--
-- File : muxbloc.vhd
--          Multiplixer bloc
--
-- Modelisation of an array processors for Adaptive Channel Equalization
-- based on square root covariance Kalman Filtering using Givens
rotations.
--
-- Author :
--               Aurelien T. Mozipo
--               March 30, 1999
--
-- --------------------------------------------------------------------------
--------


LIBRARY IEEE, lib;
      use ieee.std_logic_1164.all;
--    use ieee.std_logic_arith.all;
--    use ieee.std_logic_signed.all;

      use lib.constants.all;
      use lib.type_def.all;
--    use lib.packages.all;



ENTITY muxbloc IS
      PORT( Rst
                        : IN rstType;
                  state
                              : IN stateType0;
                  datain
                              : IN dataType;
                  frompe11, frompe22, frompe33, frompe44,
                                          -- pe44 : cout => frompe44
                  frompe14, frompe24, frompe34
      : IN dataType;
                  tope11, tope12, tope13, tope14_up, tope14_rt    : OUT
dataType;
                  x_hat
                              : OUT dataType
            );
END muxbloc ;
```

```
ARCHITECTURE behav_muxbloc OF muxbloc IS
      SIGNAL z                      :dataType;
BEGIN

      read_z : PROCESS (state, datain)
      BEGIN
            IF state = state2 THEN
                  z <= datain;
                  -- Read sample z
            END IF;
      END PROCESS read_z;


      main : PROCESS (Rst, state, datain, frompe11, frompe22, frompe33,
frompe44,
                                 frompe14, frompe24, frompe34,z)

--          VARIABLE x_hattemp       : dataType;
      BEGIN

            CASE state IS
                  WHEN state0 =>
                        tope12 <= (OTHERS => '0');

                        tope13 <= frompe22;                             -
- S+(1,1) Le signal generer a l'etape 14 par pe22 reste sur le driver
pour les deux etapes 0 et 1

--                      tope11 <= (OTHERS => 'W');
--                      tope14_up <= (OTHERS => 'W');
--                      tope14_rt <= (OTHERS => 'W');

                  WHEN state1 =>
                        tope12 <= (OTHERS => '0');
                        tope13 <= (OTHERS => '0');
                        tope14_up <= frompe24;                   --
S+(1,2)

                        x_hat <= frompe14;
      -- estimated value of z


--                      tope11 <= (OTHERS => 'W');

                  WHEN state0_1 =>
                        tope12 <= (OTHERS => '0');

                        tope13 <= frompe22;                             -
- S+(1,1) Remarquer que le signal generer a l'etape 14 par pe22 reste sur
le driver pour les deux etapes 0 et 1

--                      tope11 <= (OTHERS => 'W');
--                      tope14_up <= (OTHERS => 'W');
--                      tope14_rt <= (OTHERS => 'W');
```

```
                  WHEN state1_1 =>
                        tope12 <= (OTHERS => '0');
                        tope13 <= (OTHERS => '0');
                        tope14_up <= frompe24;
                  -- S+(1,2)
                        x_hat <= frompe14;
                        -- estimated value of z


--                      tope11 <= (OTHERS => 'W');

                  WHEN state2 =>
--                      x_hat <= (OTHERS => 'Z');
                  -- To avoid multiple signal drivers on data bus
                        tope12 <= (OTHERS => '0');
                        tope12(nb_var-1 DOWNTO nb_var-nb_const) <=
beta_nor;
                        tope13 <= (OTHERS => '0');
                        tope14_up <= frompe33;
                  -- S+(2,2)

--                      tope11 <= (OTHERS => 'W');

                  WHEN state3 =>
                        tope12 <= datain;
                        -- h1
                        tope13 <= (OTHERS => '0');
                  --
                        tope14_up <= (OTHERS => '0');

--                      tope11 <= (OTHERS => 'W');

                  WHEN state4 =>
                        tope13 <= datain;
                        -- h2
                        tope14_up <= (OTHERS => '0');

--                      tope11 <= (OTHERS => 'W');
--                      tope12 <= (OTHERS => 'W');

                  WHEN state5 =>
                        tope14_up <= datain;
                        -- h3
                        tope14_rt <= z;
                        -- Sample z

--                      tope11 <= (OTHERS => 'W');
--                      tope12 <= (OTHERS => 'W');
--                      tope13 <= (OTHERS => 'W');

                  WHEN state6 =>
                        tope11 <= (OTHERS => '0');
                        tope11(nb_var-1 DOWNTO nb_var-nb_const) <=
one_nor;                            -- V

--                      x_hat <= x_hattemp;

--                      tope12 <= (OTHERS => 'W');
```

```
--                          tope13 <= (OTHERS => 'W');
--                          tope14_up <= (OTHERS => 'W');
--                          tope14_rt <= (OTHERS => 'W');

                    WHEN state7 =>
                          tope11 <= frompe24;
            -- sh1
                          tope12 <= (OTHERS => '0');

--                          tope13 <= (OTHERS => 'W');
--                          tope14_up <= (OTHERS => 'W');
--                          tope14_rt <= (OTHERS => 'W');

                    WHEN state8 =>
                          tope11 <= frompe34;
            -- sh2
                          tope12 <= frompe22;
            -- S(1,1)
                          tope13 <= (OTHERS => '0');

--                          tope14_up <= (OTHERS => 'W');
--                          tope14_rt <= (OTHERS => 'W');

                    WHEN state9 =>
                          tope11 <= frompe44;
            -- sh3
                          tope12 <= (OTHERS => '0');
                          tope13 <= frompe24;
            -- S(1,2)
                          tope14_up <= (OTHERS => '0');

                    WHEN state10 =>
                          tope12 <= (OTHERS => '0');
                          tope13 <= frompe33;
            -- S(2,2)
                          tope14_up <= frompe24;
      -- S(1,3)

--                          tope11 <= (OTHERS => 'W');

                    WHEN state11 =>
                          tope12 <= frompe11;
            -- F
                          tope13 <= (OTHERS => '0');
                          tope14_up <= frompe34;
      -- S(2,3)

--                          tope11 <= (OTHERS => 'W');

                    WHEN state12 =>
                          tope12 <= frompe14;
            -- I
                          tope13 <= frompe11;
            -- F
                          tope14_up <= frompe44;
      -- S(3,3)

--                          tope11 <= (OTHERS => 'W');
```

```
                    WHEN state13 =>
                          tope13 <= frompe14;
              -- I
                          tope14_up <= frompe11;
        -- F

--                        tope11 <= (OTHERS => 'W');
--                        tope12 <= (OTHERS => 'W');

                    WHEN state14 =>
--                        S11_F := frompe22;
                          tope12 <=  frompe22;                         -
-- S+(1,1);
                          tope14_up <= frompe14;                       -
- I
                                                                      --
L'architecture de pe14 est telle que pendant la triangulation ( modepe14
= 011)
                                                                 -- cout
n'est pas modifie (pas beson de sortir cout vers la droite) ainsi I est
actif sur cout pendant que pe14 triangularise
                                                                  -- et
est utilise par pe12, pe13 et p14 aux etapes 13, 14 et 0

--                        tope13 <= (OTHERS => 'W');
--                        tope11 <= (OTHERS => 'W');

              END CASE ;

        END PROCESS main;

END behav_muxbloc;

■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

--
-- Component : control
--
-- Generated by System Architect version v8.5_3.3 by mozipo on Feb 26, 99
--
-- Source views :-
-- $DESIGNS/control/type_def/types
--
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
LIBRARY designs_control_sdslocal ;
USE designs_control_sdslocal.type_def.all;

ENTITY control IS
   PORT (
      clk : IN std_logic;
      pause : IN std_logic;
      ready : IN std_logic;
      reset : IN std_logic;
      address : OUT addrtype;
      clk_to_array : OUT std_logic;
      initpe11 : OUT std_logic;
      initpe22 : OUT std_logic;
```

```
      modepe11 : OUT modetype;
      modepe12 : OUT modetype;
      modepe13 : OUT modetype;
      modepe14 : OUT modetype;
      modepe22 : OUT modetype;
      modepe23 : OUT modetype;
      modepe24 : OUT modetype;
      modepe33 : OUT modetype;
      modepe34 : OUT modetype;
      modepe44 : OUT modetype;
      request : OUT std_logic;
      rst_to_array : OUT std_logic;
      state_to_mux : OUT stateType
   );
END control ;
```

■■

```
--
-- Component : control
--
-- Generated by System Architect version v8.5_3.3 by mozipo on Feb 26, 99
--
-- compatible :: Synopsys
-- Source views :-
-- $DESIGNS/control/data_flow
--

ARCHITECTURE data_flow OF control IS
   COMPONENT clk_sequence
      PORT (
         clk : IN std_logic;
         pause : IN std_logic;
         state_to_mux : IN stateType;
         ck : OUT std_logic;
         clk_to_array : OUT std_logic
      );
   END COMPONENT ;

   COMPONENT state_sequence
      PORT (
         ck : IN std_logic;
         clk : IN std_logic;
         ready : IN std_logic;
         reset : IN std_logic;
         address : OUT addrtype;
         initpe11 : OUT std_logic;
         initpe22 : OUT std_logic;
         modepe11 : OUT modetype;
         modepe12 : OUT modetype;
         modepe13 : OUT modetype;
         modepe14 : OUT modetype;
         modepe22 : OUT modetype;
         modepe23 : OUT modetype;
         modepe24 : OUT modetype;
         modepe33 : OUT modetype;
         modepe34 : OUT modetype;
         modepe44 : OUT modetype;
         request : OUT std_logic;
```

```
        rst_to_array : OUT std_logic;
        state_to_mux : OUT stateType
     );
  END COMPONENT ;

--synopsys translate_off
  FOR ALL : clk_sequence USE ENTITY
designs_control_sdslocal.clk_sequence ;
  FOR ALL : state_sequence USE ENTITY
designs_control_sdslocal.state_sequence ;
--synopsys translate_on

  -- Internal Signals
  SIGNAL ck : std_logic ;

  -- Internal Buffered Signals
  SIGNAL state_to_mux_internal : stateType ;
BEGIN

  instance_clk_sequence : clk_sequence
     PORT MAP (
        clk,
        pause,
        state_to_mux_internal,
        ck,
        clk_to_array
     );

  instance_state_sequence : state_sequence
     PORT MAP (
        ck,
        clk,
        ready,
        reset,
        address,
        initpe11,
        initpe22,
        modepe11,
        modepe12,
        modepe13,
        modepe14,
        modepe22,
        modepe23,
        modepe24,
        modepe33,
        modepe34,
        modepe44,
        request,
        rst_to_array,
        state_to_mux_internal
     );

  -- Internal Buffered Signal Mappings
  state_to_mux <= state_to_mux_internal ;

END data_flow ;
```

```
--
-- Component : clk_sequence
--
-- Generated by System Architect version v8.5_3.3 by mozipo on Feb 26, 99
--
-- Source views :-
-- $DESIGNS/control/type_def/types
--
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
LIBRARY designs_control_sdslocal ;
USE designs_control_sdslocal.type_def.all;

ENTITY clk_sequence IS
   PORT (
      clk : IN std_logic;
      pause : IN std_logic;
      state_to_mux : IN stateType;
      ck : OUT std_logic;
      clk_to_array : OUT std_logic
   );
END clk_sequence ;
```

■■

```
--
-- Component : clk_sequence
--
-- Generated by System Architect version v8.5_3.3 by mozipo on Feb 24, 99
--
-- sensitivity_attr ::
-- Source views :-
-- $DESIGNS/control/type_def/types
--



ARCHITECTURE behav_clk_sequence OF clk_sequence IS
SIGNAL s : BOOLEAN;
BEGIN

   -------------------------------------------------------------------
   vhdl_clk_sequence_sm : PROCESS (pause, clk)

   -------------------------------------------------------------------
      VARIABLE prop_delay : TIME := 1 ns;
   BEGIN

         IF pause /= '1' THEN
               ck <= clk;
         END IF;

   END PROCESS vhdl_clk_sequence_sm ;

  -------------------------------------------------------------------
   vhdl_clk_sequence_array : PROCESS (state_to_mux, clk)

  -------------------------------------------------------------------
       VARIABLE prop_delay : TIME := 1 ns;
```

```
   BEGIN
                IF state_to_mux = state0 THEN
                      clk_to_array <= clk;
                ELSIF state_to_mux'EVENT THEN
                      clk_to_array <= '1';
                ELSE
                      clk_to_array <= '0';
                END IF;

   END PROCESS vhdl_clk_sequence_array ;

END behav_clk_sequence ;
```

■■

```
--
-- Component : state_sequence
--
-- Generated by System Architect version v8.5_3.3 by mozipo on Feb 26, 99
--
-- clock :: clk
-- reset :: reset
-- Source views :-
-- $DESIGNS/control/type_def/types
--
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
LIBRARY designs_control_sdslocal ;
USE designs_control_sdslocal.type_def.all;

ENTITY state_sequence IS
   PORT (
      ck : IN std_logic;
      clk : IN std_logic;
      ready : IN std_logic;
      reset : IN std_logic;
      address : OUT addrtype;
      initpe11 : OUT std_logic;
      initpe22 : OUT std_logic;
      modepe11 : OUT modetype;
      modepe12 : OUT modetype;
      modepe13 : OUT modetype;
      modepe14 : OUT modetype;
      modepe22 : OUT modetype;
      modepe23 : OUT modetype;
      modepe24 : OUT modetype;
      modepe33 : OUT modetype;
      modepe34 : OUT modetype;
      modepe44 : OUT modetype;
      request : OUT std_logic;
      rst_to_array : OUT std_logic;
      state_to_mux : OUT stateType
   );
END state_sequence ;
```

■■

```
--
-- Component : state_sequence
--
```

```
-- Generated by System Architect version v8.5_3.3 by mozipo on Feb 26, 99
--
-- Clock outputs
-- clock :: clk rising
-- reset :: reset active_high synchronous_reset
-- animation_mode :: noanimate
-- compatible :: Synopsys
-- Source views :-
-- $DESIGNS/control/state_sequence/state_machine
-- $DESIGNS/control/type_def/types
--

ARCHITECTURE state_machine OF state_sequence IS

   -- SDS Defined State Signals
   SIGNAL current_state : statetype ;
   SIGNAL next_state : statetype ;
   -- State Variable attribute declaration for Synopsys.
   attribute STATE_VECTOR : string;
   attribute STATE_VECTOR of state_machine : architecture is
"current_state";
BEGIN

   -----------------------------------------------------------------
   clocked : PROCESS (
      clk
   )
   -----------------------------------------------------------------
   BEGIN
      IF ( clk'EVENT AND clk = '1' ) THEN
         IF ( reset = '1' ) THEN
            current_state <= state0;
            -- Start State Actions
            request<='0';
            address<=(OTHERS => '0');
            initpe11<='0';
            initpe22<='1';
            rst_to_array<=reset;
            modepe11<=(OTHERS => '0');
            modepe12<="001";
            modepe13<=(OTHERS => '0');
            modepe14<=(OTHERS => '0');
            modepe22<="101";
            modepe23<="011";
            modepe24<=(OTHERS => '0');
            modepe33<=(OTHERS => '0');
            modepe34<=(OTHERS => '0');
            modepe44<=(OTHERS => '0');
         ELSE
            current_state <= next_state;

            -- State Actions
            CASE next_state IS
            WHEN state0 =>
               request<='0';
               address<=(OTHERS => '0');
               initpe11<='0';
               initpe22<='1';
```

```
      rst_to_array<=reset;
      modepe11<=(OTHERS => '0');
      modepe12<="001";
      modepe13<=(OTHERS => '0');
      modepe14<=(OTHERS => '0');
      modepe22<="101";
      modepe23<="011";
      modepe24<=(OTHERS => '0');
      modepe33<=(OTHERS => '0');
      modepe34<=(OTHERS => '0');
      modepe44<=(OTHERS => '0');
   WHEN state1 =>
      request<='1';
      address<="001";
      initpe11<='0';
      initpe22<='0';
      rst_to_array<=reset;
      modepe11<=(OTHERS => '0');
      modepe12<="001";
      modepe13<="001";
      modepe14<=(OTHERS => '0');
      modepe22<="001";
      modepe23<="000";
      modepe24<="101";
      modepe33<=(OTHERS => '0');
      modepe34<=(OTHERS => '0');
      modepe44<=(OTHERS => '0');
   WHEN state2 =>
      request<='1';
      address<="010";
      initpe11<='0';
      initpe22<='0';
      rst_to_array<=reset;
      modepe11<=(OTHERS => '0');
      modepe12<="001";
      modepe13<="001";
      modepe14<="001";
      modepe22<="001";
      modepe23<="001";
      modepe24<="000";
      modepe33<="101";
      modepe34<=(OTHERS => '0');
      modepe44<=(OTHERS => '0');
   WHEN state3 =>
      request<='1';
      address<="011";
      initpe11<='0';
      initpe22<='0';
      rst_to_array<=reset;
      modepe11<="010";
      modepe12<="001";
      modepe13<="001";
      modepe14<="001";
      modepe22<="001";
      modepe23<="001";
      modepe24<="001";
      modepe33<="000";
      modepe34<=(OTHERS => '0');
```

```
        modepe44<=(OTHERS => '0');
    WHEN state4 =>
        request<='1';
        address<="100";
        initpe11<='0';
        initpe22<='0';
        rst_to_array<=reset;
        modepe11<="000";
        modepe12<="010";
        modepe13<="001";
        modepe14<="001";
        modepe22<="001";
        modepe23<="001";
        modepe24<="001";
        modepe33<="001";
        modepe34<=(OTHERS => '0');
        modepe44<=(OTHERS => '0');
    WHEN state5 =>
        request<='1';
        address<="101";
        initpe11<='0';
        initpe22<='0';
        rst_to_array<=reset;
        modepe11<="000";
        modepe12<="000";
        modepe13<="010";
        modepe14<="001";
        modepe22<="010";
        modepe23<="001";
        modepe24<="001";
        modepe33<="001";
        modepe34<="001";
        modepe44<=(OTHERS => '0');
    WHEN state6 =>
        request<='0';
        address<="000";
        initpe11<='0';
        initpe22<='0';
        rst_to_array<=reset;
        modepe11<="000";
        modepe12<="000";
        modepe13<="000";
        modepe14<="010";
        modepe22<="000";
        modepe23<="010";
        modepe24<="001";
        modepe33<="001";
        modepe34<="001";
        modepe44<=(OTHERS => '0');
    WHEN state7 =>
        request<='0';
        address<="000";
        initpe11<='1';
        initpe22<='0';
        rst_to_array<=reset;
        modepe11<="001";
        modepe12<="000";
        modepe13<="000";
```

```
      modepe14<="000";
      modepe22<="000";
      modepe23<="000";
      modepe24<="010";
      modepe33<="010";
      modepe34<="001";
      modepe44<="001";
   WHEN state8 =>
      request<='0';
      address<="000";
      initpe11<='0';
      initpe22<='0';
      rst_to_array<=reset;
      modepe11<="001";
      modepe12<="011";
      modepe13<="000";
      modepe14<="000";
      modepe22<="101";
      modepe23<="011";
      modepe24<="000";
      modepe33<="000";
      modepe34<="010";
      modepe44<="001";
   WHEN state9 =>
      request<='0';
      address<="000";
      initpe11<='0';
      initpe22<='0';
      rst_to_array<=reset;
      modepe11<="001";
      modepe12<="011";
      modepe13<="011";
      modepe14<="000";
      modepe22<="000";
      modepe23<="000";
      modepe24<="101";
      modepe33<="000";
      modepe34<="000";
      modepe44<="010";
   WHEN state10 =>
      request<='0';
      address<="000";
      initpe11<='0';
      initpe22<='0';
      rst_to_array<=reset;
      modepe11<="001";
      modepe12<="011";
      modepe13<="011";
      modepe14<="011";
      modepe22<="001";
      modepe23<="000";
      modepe24<="011";
      modepe33<="101";
      modepe34<="000";
      modepe44<="000";
   WHEN state11 =>
      request<='0';
      address<="000";
```

```
      initpe11<='0';
      initpe22<='0';
      rst_to_array<=reset;
      modepe11<="101";
      modepe12<="011";
      modepe13<="011";
      modepe14<="011";
      modepe22<="001";
      modepe23<="001";
      modepe24<="000";
      modepe33<="000";
      modepe34<="011";
      modepe44<="000";
   WHEN state12 =>
      request<='0';
      address<="000";
      initpe11<='0';
      initpe22<='0';
      rst_to_array<=reset;
      modepe11<="000";
      modepe12<="100";
      modepe13<="011";
      modepe14<="011";
      modepe22<="001";
      modepe23<="001";
      modepe24<="001";
      modepe33<="000";
      modepe34<="000";
      modepe44<="011";
   WHEN state13 =>
      request<='0';
      address<="000";
      initpe11<='0';
      initpe22<='0';
      rst_to_array<=reset;
      modepe11<="000";
      modepe12<="101";
      modepe13<="100";
      modepe14<="011";
      modepe22<="011";
      modepe23<="001";
      modepe24<="001";
      modepe33<="001";
      modepe34<="000";
      modepe44<="000";
   WHEN state14 =>
      request<='0';
      address<="000";
      initpe11<='0';
      initpe22<='0';
      rst_to_array<=reset;
      modepe11<="000";
      modepe12<="111";
      modepe13<="101";
      modepe14<="100";
      modepe22<="101";
      modepe23<="000";
      modepe24<="001";
```

```
            modepe33<="001";
            modepe34<="001";
            modepe44<="000";
         WHEN state0_1 =>
            request<='0';
            address<=(OTHERS => '0');
            initpe11<='0';
            initpe22<='1';
            rst_to_array<=reset;
            modepe11<=(OTHERS => '0');
            modepe12<="001";
            modepe13<="110";
            modepe14<="101";
            modepe22<="101";
            modepe23<="011";
            modepe24<=(OTHERS => '0');
            modepe33<=(OTHERS => '0');
            modepe34<="001";
            modepe44<=(OTHERS => '0');
         WHEN state1_1 =>
            request<='1';
            address<="001";
            initpe11<='0';
            initpe22<='0';
            rst_to_array<=reset;
            modepe11<=(OTHERS => '0');
            modepe12<="001";
            modepe13<="001";
            modepe14<="110";
            modepe22<="001";
            modepe23<="000";
            modepe24<="101";
            modepe33<=(OTHERS => '0');
            modepe34<=(OTHERS => '0');
            modepe44<="001";
         WHEN OTHERS =>
            NULL;
         END CASE;

      END IF;

   END IF;

END PROCESS clocked ;

-------------------------------------------------------------------
set_next_state : PROCESS (
   current_state,
   ck,
   clk,
   ready,
   reset
)
-------------------------------------------------------------------
BEGIN
   next_state <= current_state;
   CASE current_state IS
   WHEN state0 =>
```

```
        IF ( TRUE ) THEN
           next_state <= state1;
        END IF;

     WHEN state1 =>
        IF ( ready = '1' ) THEN
           next_state <= state2;
        END IF;

     WHEN state2 =>
        IF ( ready = '1' ) THEN
           next_state <= state3;
        END IF;

     WHEN state3 =>
        IF ( ready = '1' ) THEN
           next_state <= state4;
        END IF;

     WHEN state4 =>
        IF ( ready = '1' ) THEN
           next_state <= state5;
        END IF;

     WHEN state5 =>
        IF ( ready = '1' ) THEN
           next_state <= state6;
        END IF;

     WHEN state6 =>
        IF ( TRUE ) THEN
           next_state <= state7;
        END IF;

     WHEN state7 =>
        IF ( TRUE ) THEN
           next_state <= state8;
        END IF;

     WHEN state8 =>
        IF ( TRUE ) THEN
           next_state <= state9;
        END IF;

     WHEN state9 =>
        IF ( TRUE ) THEN
           next_state <= state10;
        END IF;

     WHEN state10 =>
        IF ( TRUE ) THEN
           next_state <= state11;
        END IF;

     WHEN state11 =>
        IF ( TRUE ) THEN
           next_state <= state12;
        END IF;
```

```
      WHEN state12 =>
         IF ( TRUE ) THEN
            next_state <= state13;
         END IF;

      WHEN state13 =>
         IF ( TRUE ) THEN
            next_state <= state14;
         END IF;

      WHEN state14 =>
         IF ( TRUE ) THEN
            next_state <= state0_1;
         END IF;

      WHEN state0_1 =>
         IF ( TRUE ) THEN
            next_state <= state1_1;
         END IF;

      WHEN state1_1 =>
         IF ( ready = '1' ) THEN
            next_state <= state2;
         END IF;

      WHEN OTHERS =>
         NULL;
      END CASE;

   END PROCESS set_next_state ;
   -- Current State Signal Assignment
   state_to_mux <= current_state;
END state_machine ;
```