UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

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

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN ÉLECTRONIQUE INDUSTRIELLE

PAR
MOULAY BOUBKER ELOUAFAY

ÉTUDE DE L'APPLICATION DU PIPELINE PAR VAGUE SUR UNE ARCHITECTURE SYSTOLIQUE DÉDIÉE AU FILTRE DE KALMAN

DÉCEMBRE 1998

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.

RÉSUMÉ

La capacité de manipuler des données à une très grande vitesse est un paramètre très significatif pour les calculs à très haute débit comme les systèmes de communication. Actuellement une grande tendance s'oriente vers l'utilisation des techniques algorithmiques pour atteindre ces vitesses plutôt que d'effectuer un changement de technologie qui représente jusqu'à présent une solution très coûteuse. Parmi ces techniques, il existe la méthode de pipeline, qui consiste à répartir le problème en plusieurs étages travaillant en parallèle et permettant d'augmenter le flux de données à travers un système. Deux méthodes semblent très intéressantes dans ce type de technique, le pipeline conventionnel et le pipeline par vagues (*wave pipeline*). Les deux feront l'objet d'une étude dans le cadre de ce travail. Le choix de ces deux méthodes se base sur deux critères essentiels, le coût d'implantation et la performance. Ce travail suivant aura pour objectif spécifique l'étude des ces techniques avancées du pipeline et leur application à un processeur semi-systolique linéaire à structure en anneau appelé SYSKAL [MAS95b] [MAS98b]. Nous nous proposons d'étudier cette architecture dédiée à une équation récursive dérivée du filtre de Kalman dans le but d'augmenter ses performances. En effet, les simulations faites sur le processeur SYSKAL réalisées pour une technologie CMOS de 1.2µm, démontrent les limites en fréquence du fonctionnement à 8MHz. Cela ne satisfait pas aux exigences requises pour certaines applications en terme de vitesse de calcul. Nous avons ainsi jugé pertinent d'appliquer les différentes méthodes précédemment décrites sur les blocs les plus complexes du processeur.

Ainsi nous allons proposer trois architectures utilisant le pipeline conventionnel ($PIPEKAL_1$, $PIPEKAL_2$ et $PIPEKAL_3$) et une architecture utilisant le pipeline par vagues (WAVEKAL). Une modélisation en langage de programmation matériel (VHDL) de toutes ces architectures sera réalisée afin de valider leur fonctionnalité par des simulations. Finalement nous effectuons une étude comparative entre ces différentes architectures obtenues, d'une part pour savoir laquelle des deux techniques est la mieux adaptée à notre architecture de base, d'autre part pour fixer notre choix sur l'une d'eux qui est la mieux réalisable en technologie VLSI.

REMERCIMENTS

Je tiens à exprimer ma profonde gratitude et mes sincères remerciements :

- À mon directeur de recherche, le professeur Daniel MASSICOTTE, pour avoir fait l'honneur de diriger mon sujet de recherche et pour avoir su faire naître en moi la passion du travail bien fait.

  Sa clairvoyance, sa compétence, alliées à une grande disponibilité et un suivi pédagogique régulier, ont été un guide précieux au cours des nombreuses conversations scientifiques qui ont porté sur l'ensemble de ce sujet de recherche.
- À tous les membres du groupe de recherche en électronique industrielle plus particulièrement à l'équipe de micro-électronique.
- À ma famille pour sa patience durant mes longues années d'étude, son encouragement incessant, et son support moral.
- Enfin, à toute personne qui m'a aidé de près ou de loin et dont ma mémoire n'a pu restituer le nom.

TABLE DES MATIÈRES

LISTE DES FIGURES

Chapitre 5

Introduction

1.1 Problématique

Pendant longtemps, l'augmentation de la vitesse des circuits était telle qu'en suivant l'évolution des composants, il suffisait aux constructeurs de changer les composantes moins rapides par d'autres plus rapides pour pouvoir augmenter leur puissance de fonctionnement. Cependant l'augmentation des performances, aussi importante qu'elle soit, n'est plus suffisante aujourd'hui pour répondre aux besoins des utilisateurs. Ce qui a permis la naissance des circuits intégrés à applications spécifiques, connu sous le nom de ASIC (Application Specific Integrated Circuits).

Ce dernier sera donc la solution qui permettra d'atteindre plusieurs objectifs tels que la réduction du coût de production; la puissance consommée par la réduction des composantes (transistors) et réseau de communication; l'augmentation de la vitesse de

fonctionnement grâce à la réduction des délais dans la communication et la confidentialité des ASIC puisqu'ils permettent de garder en secret le contenu et l'architecture du concepteur.

La vitesse de fonctionnement, la surface d'intégration, la consommation en puissance et le coût, représentent actuellement les principales contraintes de conception en micro-électronique. Suivant le domaine d'application, l'optimisation de l'une de ces contraintes est souvent effectuée au détriment des autres. Tel est le cas des appareils portatifs où une consommation d'énergie peut compromettre la réussite du produit, on trouve donc comme architecture, les composants à consommation minimale. Aussi, on trouve plusieurs applications qui nécessitent un fonctionnement en temps réel comme la télécommunication par exemple ou le traitement de la parole et de l'image exigent une très grande rapidité. En conséquence les recherches dans ce domaine se sont orientées à la fois vers l'optimisation de la vitesse de fonctionnement et de la surface d'intégration. La recherche des systèmes digitaux performants nécessite donc une longue étude tant au niveau matériel que logiciel.

Une approche consiste à changer la dimension de technologie pour arriver à des fréquences plus élevées et surfaces réduits en passant d'une technologie CMOS de 1.5μm à 0.2 μm. Aussi, un changement de technologie peut être utile, comme par exemple, l'utilisation de l'arséniure de gallium, AsGa, ou les ECL à la place des CMOS. Le problème majeur avec cette approche est le coût. En effet, pour arriver à des technologies assez poussées, le coût sera parfois exorbitant, ce qui revient à abandonner cette idée. Une autre approche consiste à s'orienter vers des méthodes algorithmiques qui peuvent supporter les

exigences en terme de vitesse, on parle alors de technique de parallélisme. Cette approche peut être présentée comme une façon de décomposer un problème donné en sous problèmes pour pouvoir les exécuter en parallèle afin d'atteindre un temps d'exécution très rapide. Elle déterminera notamment l'architecture globale du système, ainsi on peut citer le parallélisme géométrique et le parallélisme algorithmique.

Dans le parallélisme géométrique, les tâches sont réparties géométriquement sur plusieurs blocs appelés processeurs. Chacun des processeurs effectue le même travail sur une partie différente des données. Le travail d'un processeur se déroule essentiellement de manière indépendante par rapport aux autres, les seules interactions serviront à résoudre les problèmes aux frontières.

Dans le parallélisme algorithmique, le travail effectué par les processeurs s'apparente à un travail à la chaîne où le processeur B reçoit les données du processeur A, il effectue le travail sur ces données et passe les résultats au processeur C pour qu'il effectue le sien. Pendant que C effectue son travail, B travaille sur de nouvelles données que vient de lui fournir A. Ce type de parallélisme prend généralement le nom du pipeline.

L'application de l'une ou de l'autre de ces techniques dépend alors du degré de liberté du problème. Si pour quelques algorithmes l'application de la première méthode est possible dans la majorité des cas, il semble très difficile de l'appliquer surtout dans le cas où il y a une dépendance récursive dans le calcul des données. Dans ce dernier cas la technique la plus appropriée est celle du pipeline.

1.2 Objectifs

Ce travail a pour objectif l'étude des techniques avancées du pipeline et son application à un processeur semi-systolique linéaire à structure en anneau appelé SYSKAL [MAS.95 b] [MAS.98 b]. Nous proposons d'étudier cette architecture dédiée à une équation récursive dérivée du filtre de Kalman dans le but d'augmenter ses performances. Deux méthodes semblent très intéressantes dans ce type de technique, le pipeline conventionnel [HEN.92] et le pipeline par vagues (*wave pipeline*) [GRA. 94], [WON. 91]. Les deux feront l'objet d'une étude dans le cadre de ce travail. Le choix de ces deux méthodes se base sur deux critères essentiels, le coût d'implantation et la performance. Le pipeline conventionnel est relativement simple, il ne demande en effet qu'une intégration des registres à l'intérieur du bloc à pipeliner. Sa performance reste limitée et rend fort complexe la synchronisation des registres. En effet la réalisation du réseau de distribution de l'horloge sur une surface d'intégration accrue de l'unité de contrôle présente un degré de difficulté qui limite l'utilisation de cette méthode. Le pipeline par vagues est une méthode qui revient à éliminer les registres en égalisant les délais. Il en résulte une très grande performance.

1.3 Méthodologie

La réalisation de ce projet n'a pu être établie sans la spécification exacte d'une méthodologie complète touchant aux différents axes du problème. Le lecteur remarquera

alors que ce travail peut prendre différents aspects suivant son champ d'intérêt: un aspect théorique, un aspect innovateur et un aspect pratique.

Dans l'aspect théorique on trouvera une étude plus ou moins approfondie sur la théorie du filtrage et de ses applications en particulier pour les systèmes de mesure. L'étude sera portée essentiellement sur le filtre de Kalman et l'architecture systolique du processeur numérique SYSKAL. En plus, une introduction à la théorie du pipeline en particulier le pipeline conventionnel et le pipeline par vagues qui feront l'objet de la dernière partie de cet aspect.

L'aspect innovateur, qui englobe l'idée générale de ce travail, consiste à pouvoir montrer la capacité de l'architecture à accepter la méthode du pipeline par vagues. En effet plusieurs simulations de l'architecture du processeur ont montré que les blocs combinatoires constitués du multiplieur, de l'additionneur et du soustracteur limitent sa fonctionnalité en terme de fréquence de fonctionnement. Dans cette partie nous allons donc introduire des blocs combinatoires pipelinés suivant deux méthodes; le pipeline conventionnel et une méthode plus récente qui est le pipeline par vagues. On peut aussi introduire ici l'analyse de performance des architectures tant au niveau surface que temps de calcul en fonction des caractéristiques de la latence et du dédit des architectures.

Finalement, l'aspect pratique donnera au projet une allure industrielle. Le lecteur remarquera le long de ce projet que les démarches suivies ressemblent exactement à celles utilisés pour la conception des ASIC dans le domaine industriel. Cela veut dire qu'on trouvera toutes les démarches de modélisation (Top-Down Design) qui sont :

1- Définir les spécifications de l'architecture.

2- Répartition d'architecture.

3- Écriture dans un langage de description matériel.

4- Vérification de la fonctionnalité des programmes.

5- Synthèse, optimisation et routage

6- Vérification de la fonctionnalité après synthèse.

1.4 Structure du mémoire

Afin de faciliter la compréhension de ce mémoire, la figure 1.1 montre la représentation schématique globale de sa structure. Pour atteindre tous les objectifs voulus nous avons partagé ce mémoire en six grands chapitres traitant chacun une partie du problème.

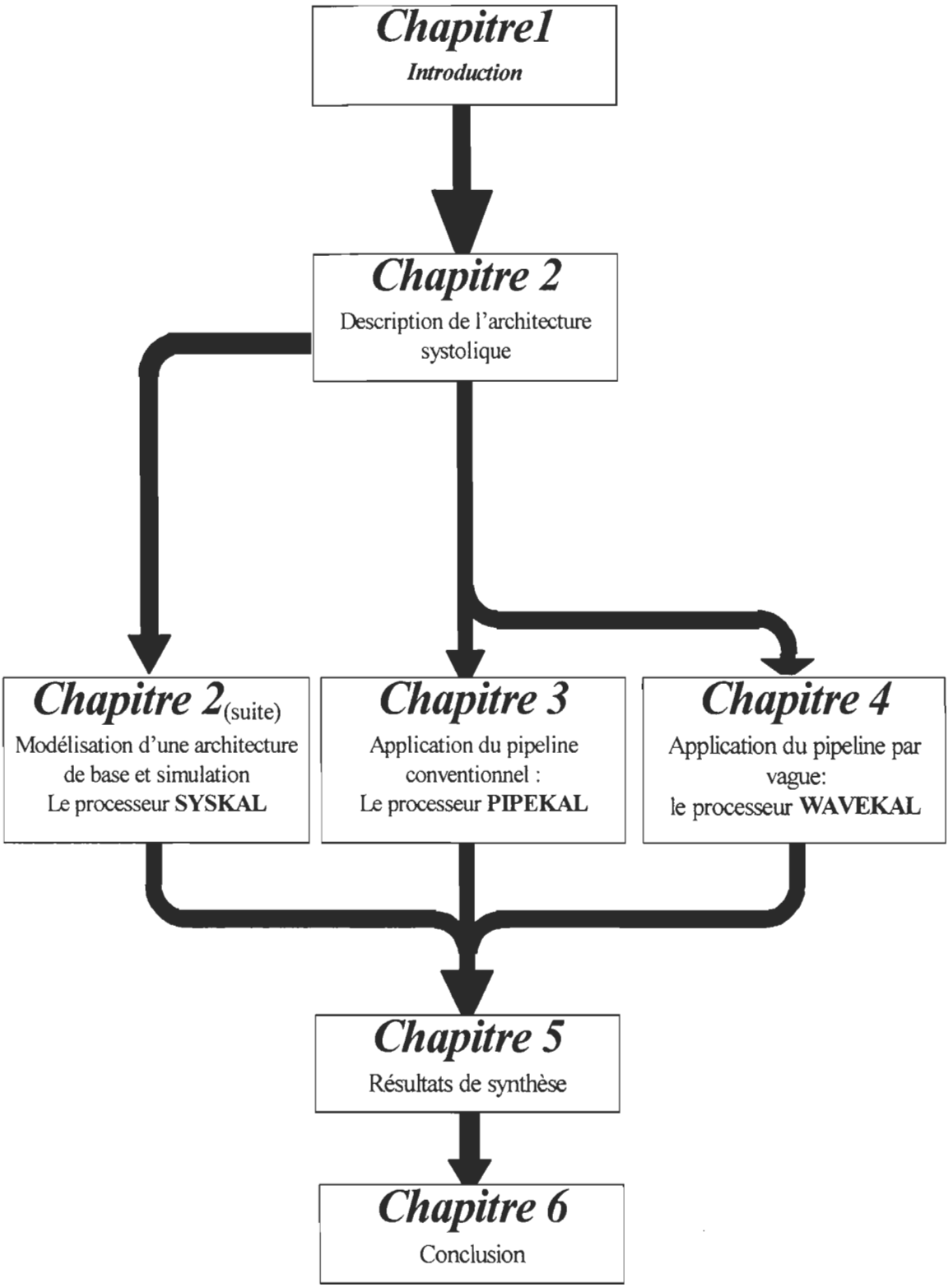


Figure 1.1 Structure du travail

2

Algorithme et architecture systolique SYSKAL

2.1 Systèmes de mesure

Le développement des moyens informatiques de traitement de signaux a accéléré énormément le progrès dans le domaine des méthodes et algorithmes de traitement numérique de signaux. En conséquence, de nombreuses publications scientifiques et plusieurs réalisations pratiques peuvent être observées ces dernières années. Une grande majorité de ces progrès touche plus particulièrement le développement de systèmes de mesure, élément essentiel dans la chaîne de traitement de l'information.

Un système de mesure est un système dont la fonction principale est de convertir l'information d'un objet physique ou biologique en signal électrique afin de lui appliquer un ensemble de transformations pour avoir à la sortie du système un signal facilement interprétable. La figure 2.1 présente la structure générale d'un système de mesure. La

variable x sur cette figure est une caractéristique empirique d'un objet de mesure, dont la détermination est le but de la mesure. Le signal $\hat{x}$ est le résultat final de mesure qui correspond à l'estimation de l'objet de mesure $\overset{o}{x}$. La transformation d'information de mesure $\overset{o}{x} \Rightarrow \hat{x}$ peut être logiquement décomposée en deux parties:

- La conversion aboutit à transmettre l'information de mesure dans le domaine de phénomènes physiques faciles à interpréter et à manipuler. Cette conversion représente en faite la chaîne de mesure de mise en forme du phénomène physique en signaux électriques, préférablement numériques ($\overset{o}{x} \Rightarrow \tilde{y}$).
- La reconstitution aboutit à interpréter le résultat de conversion. En effet elle consiste à estimer un signal $\overset{o}{x}$, qui n'est pas mesurable directement à partir des résultats de mesure d'un autre signal, qui est lié avec le premier de façon causative. Les méthodes de reconstitution sont généralement basées sur certaines suppositions concernant le modèle mathématique de la relation entre les signaux et l'information à priori accessible au signal reconstruit et aux bruits qui entachent les résultats de conversion ($\tilde{y} \Rightarrow \hat{x}$).

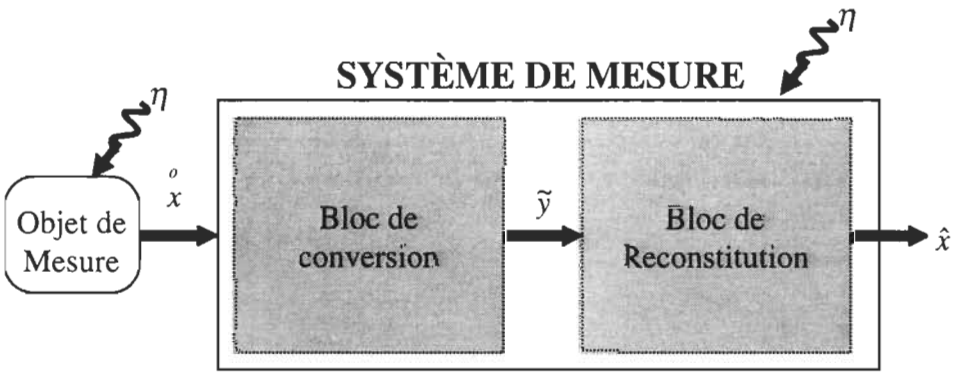


Figure 2.1 : Schéma général d'un système de mesure

Lors de la conception d'un système de mesure spécialisé, plusieurs critères rentrent dans la liste des paramètres à optimiser:

- la performance qui se traduit par le temps de calcul nécessaire pour réaliser une reconstitution pour tous les points d'échantillonnage,
- la facilité d'utilisation de l'architecture,
- la surface d'intégration,
- le coût de réalisation du système.

Idéalement et en absence de bruit, la reconstitution du signal $\overset{o}{x}$ à partir de $\tilde{y}$ ne poserait aucune difficulté puisqu'il s'agit d'une inversion simple. Alors que si on tient compte du bruit engendré par le milieu extérieur et aussi par le système de mesure lui-même, le problème devient plus difficile à résoudre, sa résolution devenant plus sensible aux erreurs de la modélisation. La modélisation mathématique de ce problème peut prendre la forme :

$$\tilde{y} = Q(x\,;\,\eta) \tag{2.1}$$

où η représente les sources de perturbation et Q est un opérateur dont l'inverse donnera la solution souhaitée qui est la reconstitution de la mesurande. Dans le cas d'un système lineaire, tel que présenté à la figure 2.2, le modèle représentant le lien entre le signal d'entrée $x(t)$ et la sortie de la conversion peut s'écrire sous la forme :

$$y(t) = \int_{-\infty}^{\infty} h(t,t')x(t')d(t') \tag{2.2}$$

alors qu'en présence de bruit il devient :

$$\tilde{y}(t) = \int_{-\infty}^{\infty} h(t,t')x(t')d(t') + \eta(t) \tag{2.3}$$

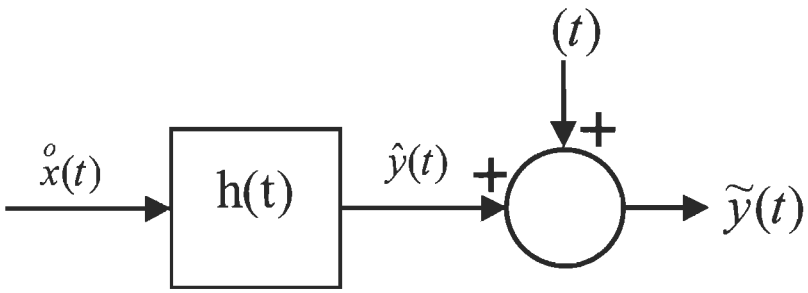


Figure 2.2 :Système linéaire monodimensionnel avec bruit à la sortie

Généralement pour résoudre ce type d'équation la seule méthode possible est de passé par la numérisation de l'équation (2.3) qui prend alors la forme suivante :

$$\tilde{y}_n = \sum_{i=0}^{M} h_{n-i} x_i + \eta_n \tag{2.4}$$

Pour résoudre ce type de problème, il n'y a pas de solution universelle, ce qui explique la diversité des méthodes qui existent et par la suite la difficulté du choix de la méthode qui sera la plus adaptée au type de problème envisagé. Parmi ces différentes méthodes on peut citer :

Méthodes directes, elles consistent à numériser de la relation liant l'entrée x(t) et la sortie $\tilde{y}(t)$. Le problème revient à résoudre une équation algébrique intégrale ou différentielle.

Méthode itérative, elle consiste à reconstituer le mesurande par approximations successives à partir du signal de sortie $\tilde{y}(t)$. Une des exemples de cette méthode est celle de Jansson [CRI.91]

Méthode stochastique, elle exploite le caractère aléatoire des erreurs de mesure. Un des exemples de cette méthode est celle du filtre de Kalman [MEN.77], [DEM.83], [DEM.85] [MAS.95 b].

Méthode variationnelle elle consiste à déterminer des approximations successives de la solution tel que la méthode spline [BEN.92].

Dans le reste de ce chapitre nous allons présenter le principe du filtre de Kalman puisqu'il constitue la base de l'architecture SYSKAL. Après nous décrirons le processeur semi-systolique SYSKAL, où un accent particulier sera porté sur la structure de ce processeur ainsi que son principe de fonctionnement.

2.2 Algorithme de reconstitution de signaux basé sur le filtre de Kalman

Le système linéaire de la figure 2.2 peut être modélisé par les équations suivantes :

$$\frac{d\mathbf{z}(t)}{dt} = \Phi\mathbf{z}(t) + \mathbf{b}u(t) \tag{2.5}$$

$$\tilde{y}(t) = \mathbf{h}(t)\mathbf{z}(t) + \eta(t) \tag{2.6}$$

Dans le cas discret ces équations deviennent :

$$\mathbf{z}_{n+1} = \Phi\mathbf{z}_n + \mathbf{b}u_n \tag{2.7}$$

$$\tilde{y}_n = \mathbf{h}_n^T\mathbf{z}_n + \eta_n \tag{2.8}$$

où $\mathbf{z}_n$ est le vecteur d'état du système à l'instant n, ϕ la matrice d'état du système, u_n le vecteur entrée du système et le vecteur **b** représente le lien entre l'entrée du système et les variables d'états. **h** représente le réponse impulsionnelle du système de conversion et permet de trouver l'état du système en observant les données $\tilde{y}_n$. Une classe d'algorithmes de reconstitution de signaux basés sur le filtre de Kalman a été proposé dans [MAS.95 b]. Cette classe d'algorithmes se définie par l'équation suivante:

$$\hat{\mathbf{z}}_{n+1} = \Phi\hat{\mathbf{z}}_n + \mathbf{k}_\infty [\tilde{y}_n - \mathbf{h}\Phi\mathbf{z}_n] \tag{2.9}$$

où k_∞. est le vecteur gain stationnaire de Kalman. Il peut être calculé d'avance dans le cas d'un système linéaire, **h** est le vecteur de la réponse impulsionnelle normalisée du système [MAS 98].

2.3 Description de l'architecture systolique SYSKAL

Une analyse de l'algorithme révèle qu'il est possible de distribuer l'exécution des calculs sur deux processus indépendants [MAS.95 b]. Cela est possible en calculant l'estimation du signal de sortie du système de mesure $\hat{y}_{n+1}$ au même moment que l'estimation du vecteur d'état. La figure 2.3 présente la séquence de fonctionnement en parallèle, où l'on obtient les équations récurrentes suivantes:

$$\hat{z}_{n+1/n,m-1} = \hat{z}_{n/n-1,m} + k_{\infty,m} I_n \tag{2.10}$$

$$\hat{y}_{n+1/m-1} = h_{m-1}\hat{z}_{n+1/n,m-1} + \hat{y}_{n=1/m-2} \tag{2.11}$$

$$I_n = \tilde{y}_n - \hat{y}_N \tag{2.12}$$

Où $\hat{Z}_{n+1/n,m-1}$ représente le m^iéme élément du vecteur estimé Z à l'instant d'échantillonnage n+1 tenant compte des n valeurs d'échantillonnages précédentes l'algorithme de reconstitution de signaux proposé dans [MAS 98] impose des conditions au limites au équation récurrentes suivante: a b

$$n = 1, 1 \le \mathrm{m} \le \mathrm{M} \quad \Rightarrow \quad \hat{z}_{n/n-1,m} = 0 \tag{2.13}$$

$$1 \le n \le N,\ \mathrm{m} = 1 \quad \Rightarrow \quad \hat{y}_{n+1,m-2} = 0 \text{ et } \hat{y}_{n+1,m-1} = 0 \tag{2.14}$$

$$1 \le n \le N,\ \mathrm{m} = 1 \quad \Rightarrow \quad \hat{x}_n = \hat{z}_{n+1/n,m-1} \tag{2.15}$$

$$1 \le n \le N,\ \mathrm{m} = 1 \quad \Rightarrow \quad h_{m-1} = 0 \tag{2.16}$$

$$1 \le n \le N,\ \mathrm{m} = \mathrm{M} \quad \Rightarrow \quad \hat{z}_{n+1/n,M} = \hat{z}_{n+1/n,M-1} \tag{2.17}$$

$$1 \le n \le N,\ \mathrm{m} = \mathrm{M} \quad \Rightarrow \quad I_{n+1} = \hat{y}_{n+1} - \hat{y}_{n+1} \tag{2.18}$$

Pour augmenter davantage la vitesse d'exécution de cet algorithme, une technique classique de type "divisé pour régner" a été utilisée [MAS.95 b] [MAS.98 b] et qui consiste à résoudre le problème en devisant celui-ci en L sous problèmes résolus séparément. Ceci a favorisé par la suite d'allouer le calcul à un réseau de processeurs qui communiquent localement et qui fonctionnent de façon régulière. Tous ces processeurs sont synchrones sur la même horloge et les données se déplacent au même rythme d'un processeur à l'autre via des registres internes. Cette architecture semi-systolique linéaire à structure en anneaux nommé SYSKAL est présentée à la figure 2.4 avec quatre processeurs élémentaires. La figure 2.5 présente la position du processeur SYSKAL dans un système de mesure.

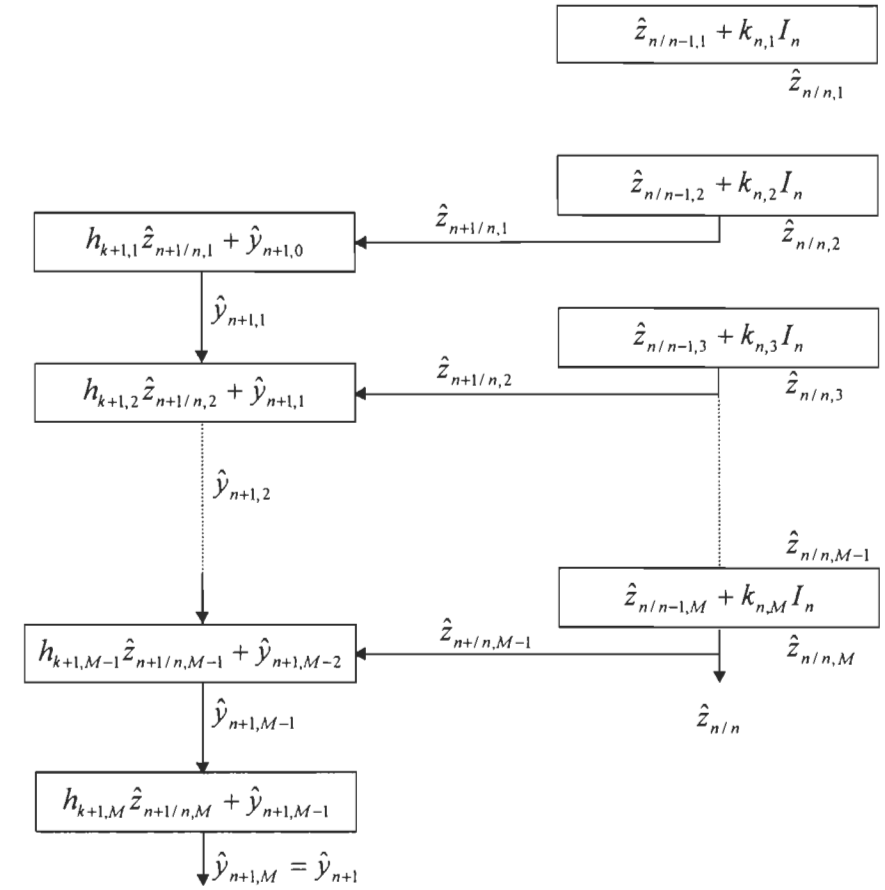


Figure 2.3 Séquences de fonctionnement de l'architecture en parallèles.

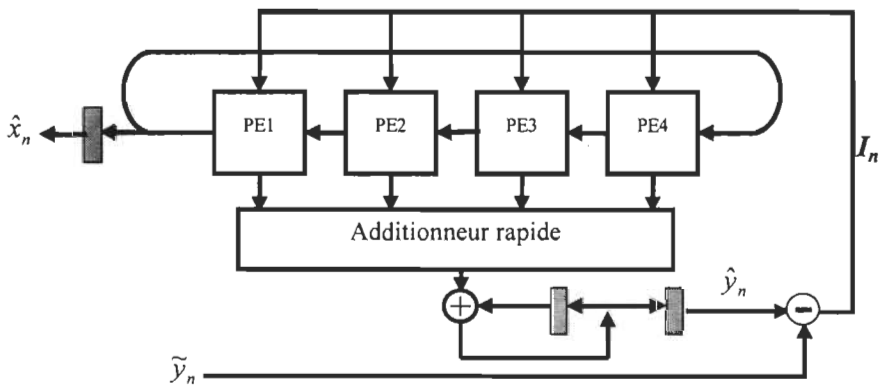


Figure 2.4 Architecture semi-systolique SYSKAL

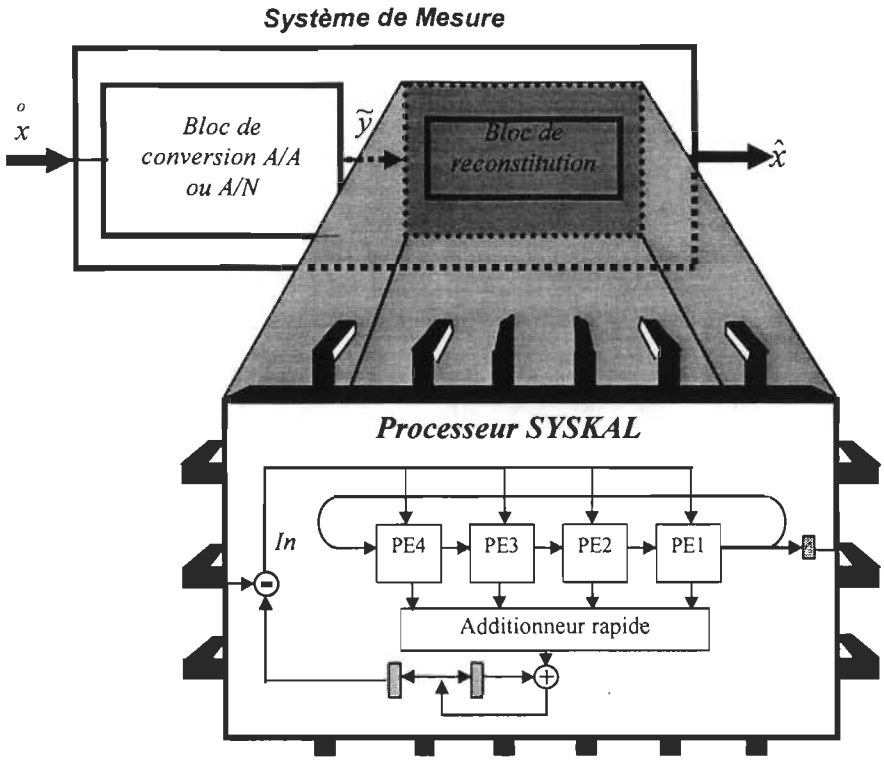


Figure 2.5 Processeur SYSKAL et système de mesure

La structure interne de tous les processeurs élémentaires (PEi) sont identiques, à l'exception du premier (PE_1) et du dernier (PE_L) qui doivent satisfaire les conditions limites (2.13) à (2.18). La figure 2.6 présente le diagramme bloc de la structure interne d'un processeur élémentaire. Ce dernier est constitué de deux cellules C_Z et C_y. Les cellules C_Z servent au calcul de $\hat{z}$ alors que les cellules C_y au calcul partiel de $\hat{y}$.

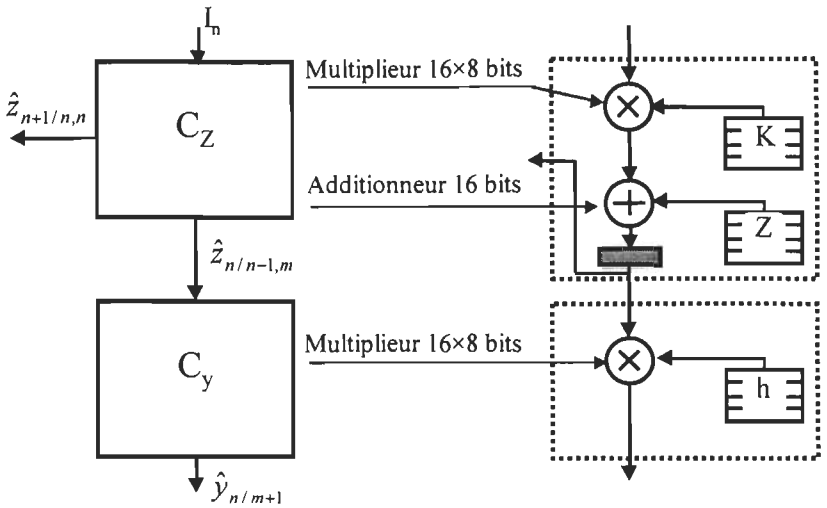


Figure 2.6 Digramme bloc de la structure interne des processeur élémentaires

La structure interne des différents processeurs élémentaires est présentée à la figure 2.7. On remarque alors que chaque processeur est formé de deux multiplieurs 16×8 bits, un additionneur 16×16 bits et de trois types de piles : une pour stocker le gain de Kalman $\mathbf{k}_\infty$, une pour stocker la réponse impulsionnelle du système **h** et le dernier pour stocker les résultats intermédiaires $\hat{y}$. Un décaleur est utilisé pour appliquer la contrainte de positivité selon [MAS.95 b]. Pour le premier processeur élémentaire (figure 2.7 a) un registre a été mis à la sortie pour rendre accessible le signal reconstitué $\hat{x}$. Un multiplexeur est ajouté dans le dernier processeur élémentaire (figure 2.7 c) pour satisfaire les conditions (2.13) et (2.15), alors que la figure 2.7 (b) présente la structure interne d'un processeur élémentaire intermédiaire.

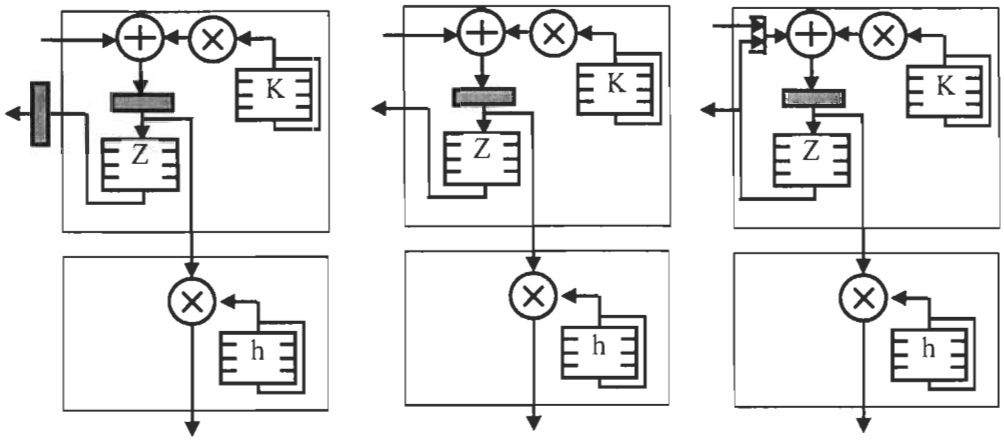


Figure 2.7 Architecture interne des processeurs élémentaires: a) Premier PE, b) PE intermédiaire et c) Dernier PE

La synthèse et la simulation du processeur SYSKAL ont été réalisées pour une technologie CMOS de 1.2µm [MAS.98 b]. Par la suite, une vérification des délais dans cette architecture a été réalisée grâce à l'outil Quick-Path® de Mentor-Graphics® [ELO.96]. Ceci nous a permis de trouver le chemin le plus lent qui limite la vitesse de fonctionnement. Le tableau 2.1 présente les résultats de simulation des différents blocs de logique combinatoire du processeur SYSKAL. Pour chercher la fréquence limite du processeur, il a fallu chercher le chemin le plus lent, soit le bloc C_y puisqu'il contient un multiplieur et deux additionneurs placer en série. La fréquence limite dans ce cas a été simulée à 8MHz, considérant aucun pipeline entre les cellules C_z et C_y.

Tableau 2.1 Résultats de synthèse du processeur SYSKAL

	Surface d'intégration u[1]	Fréquence [MHz]
Multiplieur	1339	25
Additionneur	792	32
Soustracteur	792	32
Pile de données (33)	2340	--
Décaleur gauche-droite	760	100

(1) u : unité de bloc élémentaire utilisé par Synopsys

2.4 Conclusion

Dans le cadre de ce projet le travail consiste à améliorer le bloc de reconstitution formé d'un processeur dédié à la reconstitution de signaux basé sur le filtre de Kalman de telle sorte à rendre le système de mesure plus rapide, en se basant sur les méthodes de pipeline afin d'augmenter le débit de calcul. Nous devons démontrer qu'il est possible d'appliquer à l'algorithme KALMAN+ défini selon l'équation (2.9) [MAS.95 b] et à son architecture SYSKAL [MAS.98 b] les techniques du pipeline conventionnel (chapitre 3) et du pipeline par vagues (chapitre 4). L'objectif de cette démonstration se divise en deux points:

1- L'utilisation de deux processeurs élémentaires pour une implémentations en VLSI permettrait de considérer des piles de données plus profondes afin d'appliquer les techniques avancées du pipeline;

2- Le débit très élevé que procure la technique du pipeline permettrait d'élargir le champ d'application du processeur au traitement à très haute vitesse.

3

Pipeline conventionnel

L'évolution progressive dans le domaine de la télécommunication et l'informatique et la demande croissante de circuit miniaturisé ont influencé remarquablement l'évolution de la micro-électronique. Actuellement, la science de la micro-électronique dépend essentiellement de quatre paramètres principaux qui sont la fréquence de fonctionnement, la surface d'intégration, la consommation en puissance et finalement un facteur important qui est le coût de commercialisation.

Ce chapitre présente deux volets: le premier est consacré à une étude théorique qui permettra au lecteur une brève introduction à la théorie du pipeline conventionnel, et le deuxième volet présente l'application du pipeline conventionnel à l'architecture SYSKAL.

Dans ce paragraphe nous allons mettre l'accent sur un paramètre très important qui est la fréquence de fonctionnement pour savoir d'une part les bases des facteurs qui la définissent et d'autre part discuter de la théorie des différentes méthodes permettant son amélioration. Notons finalement que cette étude est partagée en deux sections. La section 3.1 est consacré à la base théorique du délai de propagation dans les circuits numériques. Elle nous permettra de savoir les fondements qui sont à la base des délais dans les circuits intégrés et d'autre part de connaître quelques solutions remédiant à ce problème. Nous allons discuter à la section 3.2 la théorie du pipeline conventionnel et ses limites d'application.

3.1 Notions de délais dans les circuits numériques

Aujourd'hui la technologie CMOS est la technologie la plus dominante dans le domaine des circuits intégrés. La raison de sa dominance est sa faible consommation en puissance, son faible coût et sa capacité d'intégration à très grande échelle. C'est pour cette raison que nous allons consacrer cette théorie en nous basant sur la technologie CMOS pour pouvoir augmenter les performances de l'architecture SYSKAL.

La théorie des délais de propagation dans les circuits numériques peut se décomposer généralement en deux parties: les délais de propagation dans une porte logique causé par les temps de commutations des transistors et les délais dans les interconnections des blocs combinatoires [SAV.88].

On note que les méthodes d'estimation des délais utilisées dans ce chapitre sont des méthodes simplifiées. Pour une bonne précision, il faudra tenir compte d'autres facteurs tels

que la tension d'alimentation et la température, cette dernière influence de façon directe la mobilité des porteurs majoritaires du canal, de type P ou de type N [SEV.96].

3.1.1 Délai dans un transistor

Commençant d'abord par un modèle simple du transistor MOS présenté à la figure 3.1, on peut expliquer son fonctionnement comme suit: quand la tension de la gâchette G dépasse une valeur qui est la tension seuil, un champ électrique s'établit entre la gâchette et le substrat ce qui attire les charges entre la source S et drain D à l'opposé de la grille. Ces charges permettent par la suite le passage d'un courant électrique qui s'établira entre la source S et le drain D.

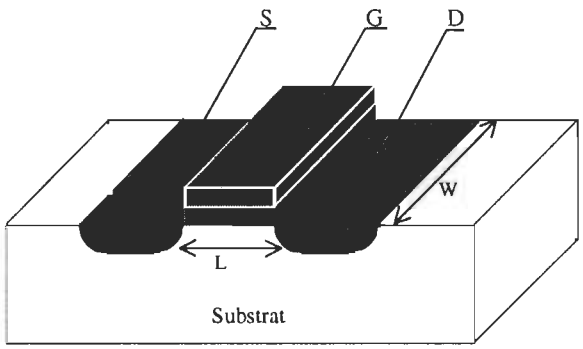


Figure 3.1 : Modèle d'un transistor CMOS

Idéalement, le passage d'un état à l'autre se fait instantanément dès que la tension de la gâchette change de polarité. Dans le cas réel la communication est ralentie par l'existence d'éléments parasites capacitifs et résistifs qui dépendent essentiellement de la largeur w et de la longueur l et aussi de la tension appliquée entre la source et le drain [WES.93].

Un modèle qui permet d'obtenir une première approximation du délai est celui en vertu duquel un transistor bloqué est remplacé par une résistance infinie et un transistor

actif par une résistance relativement faible. On note que pour un transistor de type N la résistance équivalente est R_s. Généralement, cette résistance est donnée par le fabriquant et correspond à la valeur d'un carré du canal [SAV.88]. L'expression du carré du canal est utilisée pour définir l'unité du travail. Le carré du canal est donc la plus petite dimension proposée par le fabricant. Par exemple dans une technologie de 1,2 μm le transistor de taille minimale est celui dont le canal occupe un carré de taille minimale soit donc 1,2 μm de côté.

La détermination des résistances dynamiques du transistor est basée sur la nature de la mobilité des porteurs majoritaires dans le canal utilisé. Ainsi, en comparant la mobilité du canal P et celle du canal N on remarque que celle de ce dernier est deux à trois fois plus élevée que celle du canal P. Cela nous permet de dire que la résistance d'un transistor de type P est de deux à trois fois plus grande que la résistance d'un transistor de type N, voir figure 3.2. Généralement on utilisera l'approximation suivante : $R_p = 2R_n$, avec $R_n = R_s(L/W)$ où W est la largeur de canal utilisé, L est la longueur du canal utilisé et R_s est la résistance unitaire équivalente.

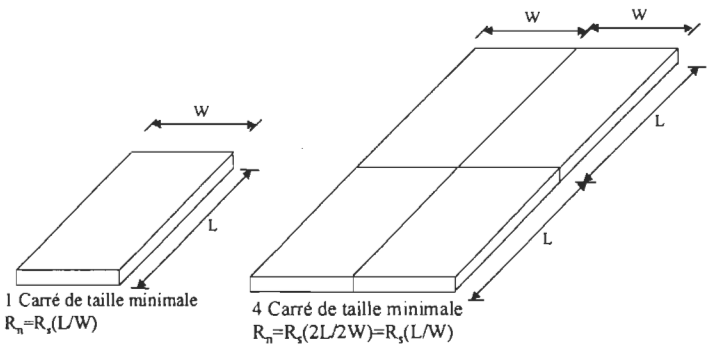


Figure 3.2 : Exemple de calcul de la résistance équivalente

Le schéma équivalent d'un transistor, présenté à la figure 3.3, montre qu'au niveau de chaque jonction apparaît une capacité dont la valeur est fonction de l'unité de travail. Ainsi la capacité équivalente d'un transistor noté C_g peut être estimée à partir de la capacité de grille dans les conditions de taille minimale. Le modèle d'estimation du délai, τ_d, pour un transistor sera donc le produit de la résistance R par la capacité équivalente de charge C:

$$\tau_d = R \times C_c \tag{3.1}$$

où

τ_d : délai à travers un transistor

R : Résistance équivalente du transistor

C_c : Capacité équivalente de charge

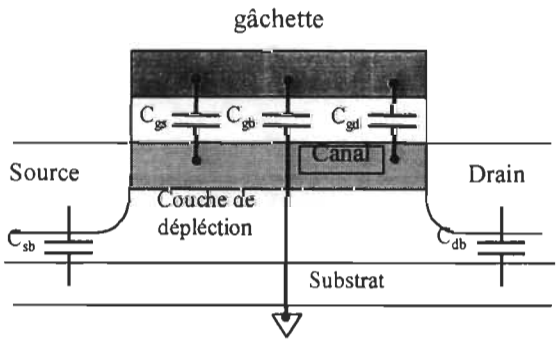


Figure 3.3 capacités parasites d'un transistor MOS

3.1.2 Délais des chemins de transmission

En plus de la méthode que nous venons d'utiliser, et qui détermine le délai à travers un transistor, une autre méthode permet la détermination du délai à travers le chemin des portes de transmission. Une porte de transmission est l'élément essentiel de la logique à

relais. Une caractéristique importante de la porte de transmission est son caractère bidirectionnelle grâce à l'utilisation de deux transistors dont l'un de type P et l'autre est de type N connectés en parallèle. Cette méthode se base sur la supposition que le chemin de portes de transmission est symétrique et que la largeur de transistor P est deux fois celle du transistor N. En plus on suppose que toutes les capacités parasites sont équivalentes à une capacité en série avec la résistance globale de la chaîne, on déduit donc que le délai à travers un chemin peut être majoré par le produit de la résistance équivalente avec la capacité totale. Nous aurons finalement [SAV.88].

$$\tau_{tr} = 3N^2 RC_c \tag{3.2}$$

N : Nombres de portes dans le chemin

R : résistance équivalente

C_c :Capacité de charge totale

Les résultats trouvés montrent que le délai à travers une chaîne de transmission est proportionnel au carré du nombre de portes de transmission et à la capacité totale de la chaîne (équation 3.2). Ainsi, pour une bonne amélioration du délai il faut d'abord réduire au minimum le nombre de portes, ensuite, il est recommandé d'insérer de façon périodique des amplificateurs qui permettent de régénérer le signal de sortie. Souvent utilisée comme interrupteur analogique, la porte de transmission est difficilement testable ce qui en fait un élément rarement rencontré dans les circuits numériques.

3.1.3 Délai dans un circuit à logique combinatoire

La connaissance des délais dans un circuit à logique combinatoire est très importante puisqu'il nous donne une idée directe sur la fréquence de fonctionnement de notre circuit et ses champs d'application.

Par définition, le délai est le temps que peut mettre un signal traversant un circuit logique depuis l'entrée jusqu'à la sortie. Ainsi, suivant la complexité du circuit et le nombre d'entrées et de sorties on peut avoir différents délais de propagation le long des différents chemins.

Pour une bonne illustration de ce concept, la représentation graphique de la figure 3.4 montre le flot de données à travers un bloc de logique combinatoire (BLC). Le système est composé d'un circuit de logique et de deux registres l'un à l'entrée et l'autre à la sortie. Ces registres contrôlent la circulation des données. Au début de chaque cycle d'horloge les données se propagent à travers le circuit combinatoire par le registre d'entrée et la différence des chemins à travers le circuit entraîne une zone (zone rayée) limitée par t_{max} et t_{min} pendant laquelle le circuit est fonctionnel. Le prochain renvoi de données à l'entrée ne peut avoir lieu avant que l'ancienne donnée soit complètement stable à la sortie du circuit combinatoire. De cela on peut déduire que le délai à travers ce circuit est égal à t_{max} ce qui explique la limite du fonctionnement du circuit. Pour plus de détails notons qu'il faut ajouter le temps de stabilisation des données dans les registres et le temps de stabilisation de l'horloge.

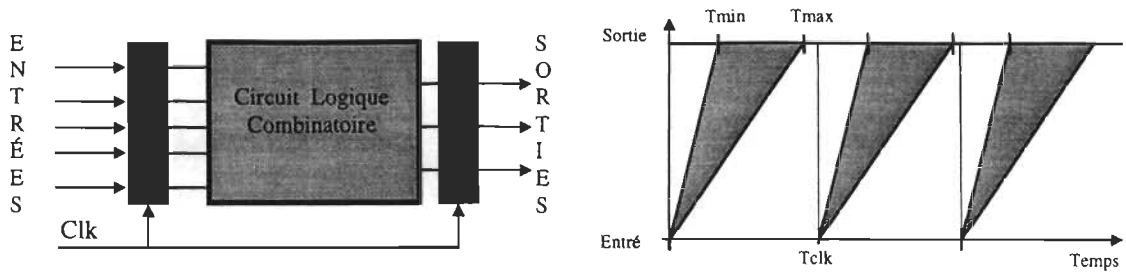


Figure 3.4 : Flot de données dans un BLC: a) Structure d'un BLC, b) Propagation de données dans un BLC

Ceci nous amène donc à discuter sur les différentes méthodes utilisées permettant de réduire la période de l'horloge et par la suite d'augmenter les performances des circuits à logique combinatoire en les rendants à logique séquentielle. Comme on l'a déjà mentionné au chapitre 1, on peut procéder suivant deux approches différentes, une matérielle et l'autre algorithmique [SEV.96].

Dans l'approche matérielle nous allons trouver généralement des modifications de type matériel telles que :

Réduction de l'entrance ''fanin'' et du sortance ''fanout'' : un courant d'entrée très élevé augmente généralement la charge capacitive des transistors ce qui augmente le délai de propagation de façon proportionnelle de même pour le courant de sortie.

Réduire la surface de l'architecture: l'utilisation des architectures très condensées réduit généralement les chemins de connexion entre les différentes portes et par la suite les délais de transmission peuvent être réduites.

Optimiser la largeur des transistors : connaissant que le délai dans un transistor dépend essentiellement de la dimension du canal, nous pouvons alors augmenter sa vitesse de fonctionnement en utilisant un canal de transistor de plus petite dimension telque 0,5 μm ou 0,2 μm au lieu de 1,5 μm .

Choix du type de transistor : pour les technologies MOS on préfère utiliser des transistors n-MOS que des p-MOS puisque les premiers sont plus rapides, on parle d'un rapport de deux.

Changement de technologie : actuellement la technologie MOS est la plus utilisée mais pour des raisons de rapidité on peut passer à des technologies plus rapides tel que ECL ou même AsGa si les coûts et la consommation sont acceptables.

Dans l'approche algorithmique, la contrainte de la vitesse de calcul de l'algorithme, dans sa mise en œuvre en VLSI, dépend des critères visés par ces applications. Le développement d'une architecture dédiée à un algorithme décrit sous la forme d'équations récurrentes, nous permet de prendre deux directions :

- soit une architecture parallèle à grains fins, (ex : systolique) offrant beaucoup de ressources (additionneur, soustracteur, multiplieur, etc.) afin d'augmenter la vitesse de calcul.

- soit une architecture sur laquelle les ressources existantes sont pipelinées afin d'augmenter le débit si la profondeur des récurrences de l'équation le permet.

Si pour la première approche l'augmentation de la performance est indépendante de l'algorithme utilisé dans le circuit, le coût reste énormément élevé puisqu'il s'agit de l'utilisation de nouvelles technologies. La deuxième approche reste donc la meilleure solution de performance pour les architectures qui acceptent le pipeline. Aussi on peut remarquer que même le parallélisme n'est pas toujours accessible puisqu'il dépend de la nature de l'architecture, ce qui n'est pas facile à appliquer. La seule méthode qui semble donc la plus adéquate si applicable est le pipeline. Cette technique du pipeline a été appliqué par le travail en chaîne dans les grandes usines où chaque ouvrier effectue une partie de la tâche de façon progressive. En micro-électronique le pipeline joue aussi le même rôle sauf qu'il peut prendre plusieurs formes, on parle alors du pipeline conventionnel, micropipeline et pipeline par vagues [GRA.92 a].

3.2 La technique du pipeline conventionnel

Un pipeline est une technique qui reproduit la division des opérations à effectuer en plusieurs étapes de durées égales et qui s'exécutent successivement. Il s'agit donc d'une forme élémentaire de parallélisme [HEN.92], [QUI.89], fondée sur la segmentation des instructions et des données et une anticipation de leur chargement. Par ailleurs, l'exploitation des structures pipelines apporte un gain important de performances sans pour autant nécessiter une modification profonde de la structure des algorithmiques.

L'application du pipeline peut être effectuée soit de façon matérielle ou logicielle [SEV.96]. Dans la première, il s'agit de l'application de cette technique sur des unités

matérielles telle que les blocs de logique combinatoire qui sont les multiplieurs, les additionneurs, etc. Une étude profonde de la structure interne de ces unités est donc obligatoire. L'autre type de pipeline concerne généralement le flot de données qui ne touche pas la structure matérielle de l'architecture. Il est souvent utilisé pour pipeliner le jeu d'instructions qui contrôle le fonctionnement d'une unité arithmétique et logique (UAL). Dans ce paragraphe nous présenterons d'abord la technique de base du pipeline, puis nous examinerons les problèmes introduits par cette technique ainsi que les performances qu'elle permet d'atteindre. Finalement, nous présenterons les avantages et les inconvénients de cette technique.

3.2.1 Les principes de base du pipeline conventionnel

Comme son nom l'indique, le pipeline conventionnel ou le pipeline synchrone propose une solution simple qui consiste à fractionner le bloc combinatoire en plusieurs blocs élémentaires dont chacun effectue une partie de la tâche globale assignée au circuit. Ensuite, on intercale ces sous blocs en insérant des registres qui doivent être tous sous le contrôle de la même horloge d'où le nom pipeline conventionnel.

Le rôle du pipeline est de faire propager les données sans qu'il y est une possibilité de chevauchement entre elles afin de ne pas nuire à la validité du résultat de sortie de l'unité logique pipeliné. La figure 3.5 présente les résultats de pipeline et la comparaison avec le flot de données dans le cas simple.

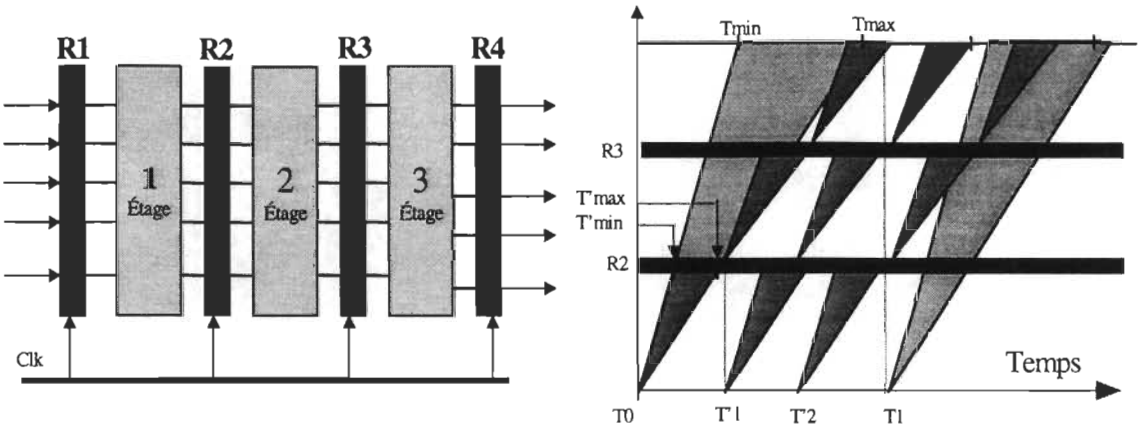


Figure 3.5: flot de données lors d'un pipeline conventionnel: a) BLC pipeliné, b) flot de données avec pipeline

Nous exécutons un ensemble d'opérations O_1, O_2,....O_n sur un opérateur pipeliné de k étages où chaque opération effectuée est la même pour des jeux de données différentes. Appelons τ le temps de traversée d'un étage, c'est à dire le temps de propagation des données dans un étage. Généralement τ est appelée temps de cycle machine. Les opérations sont chargées de manière successive l'une après l'autre avec un décalage de temps qui doit être supérieur ou égal à τ (à la limite) de telle sorte à avoir un enchaînement d'opérations. Autrement dit, lorsque l'étage i exécute l'étape i de l'opération I_j, l'étape i-1 exécute l'étape i-1 de I_{j-1}. Le premier résultat sera donc obtenu après k unités de temps appelées temps d'initialisation nécessaire au démarrage du pipeline; et les autres résultats se produiront ensuite avec un rythme d'unités de temps τ.

3.2.2 Les aléas

Comme le reste des techniques, le pipeline présente des situations où il est difficile de contrôler le bon fonctionnement de la structure pipeline, ces situations sont appelées aléas. Les aléas réduisent énormément la performance de pipeline. Il y a généralement trois classes d'aléas [HEN.92];

-*1 Les aléas structurels* interviennent lors de conflits de ressources, quand le matériel ne peut gérer toutes les combinaisons possibles de chevauchement d'instructions au moment de l'exécution.

-*2 Les aléas de données* interviennent quand une instruction dépend du résultat d'une instruction précédente, comme cela peut intervenir dans le recouvrement des instructions dans le pipeline.

-*3 Les aléas de contrôle* résultent du fonctionnement en pipeline des branchements et des instructions qui modifient le CP.

Ces aléas dans le pipeline peuvent obliger à suspendre le pipeline. La différence principale entre les suspensions d'une machine pipelinée et celles d'une machine non pipelinée est liée au fait qu'il y a plusieurs instructions qui s'exécutent en même temps. Une suspension dans une machine pipelinée nécessite souvent que des instructions puissent poursuivre leur exécution pendant que d'autres sont différées. Généralement, quand une instruction est suspendue, les instructions précédant l'instruction suspendue peuvent se poursuivre, mais aucune nouvelle instruction n'est lue durant la suspension.

3.2.3 Les performances

Les performances (vitesse de calcul, fréquence de fonctionnement) de pipeline est déterminé par la fréquence à laquelle les instructions sortent du pipeline. Comme les étages du pipeline sont connectés les uns aux autres, tous les étages doivent être active au même moment. Le temps nécessaire à une instruction pour passer d'un étage à l'autre est appelé *cycle machine*. La longueur d'un cycle machine est déterminée par l'étage le plus lent, puisque tous les étages travaillent en même temps. Le cycle machine vaut au plus un cycle d'horloge.

Notre objectif lors de la conception d'une structure pipeliné est de pouvoir équilibrer toutes les tailles des différents étages du pipeline. Si les étages sont parfaitement équilibrés, le temps par instruction dans une structure pipeliné devient

$$\text{temps par instruction} = \frac{\text{temps par instruction sur une machine non pipelinée}}{\text{nombre d'étages pipeline}} \qquad (3.3)$$

Dans des conditions idéales, l'accélération liée au pipeline est égale au nombre d'étages du pipeline.

3.2.4 Les limites du pipeline conventionnel

Le pipeline accroît le débit des données mais il ne réduit pas le temps d'exécution par unité de temps. En fait, il augmente légèrement le temps d'exécution d'une instruction à cause du surcoût lié au contrôle du pipeline. L'augmentation du débit d'instructions signifie

qu'un programme s'exécute plus vite et en un temps d'exécution total plus faible, même si aucune instruction ne va réellement plus vite grâce à une horloge plus rapide.

Le fait que le temps d'exécution de chaque instruction reste inchangé impose des limites à la profondeur réelle du pipeline. D'autres considérations de conception limitent la fréquence de fonctionnement qui peut être obtenue par un pipeline plus profond. L'élément le plus important est l'effet combiné des délais des bascules est des dispersions d'horloge. Les bascules qui sont nécessaires entre les étages pipelinés ajoutent un temps d'établissement et un temps de propagation à chaque période de l'horloge. Les dispersions d'horloge font de même dans une limite plus faible.

3.3 Application du pipeline conventionnel à l'architecture SYSKAL

Plusieurs de nos travaux ont présenté l'architecture du processeur SYSKAL formé de quatre processeurs élémentaires et un bloc pour le calcul d'innovation Eq (2.12) [MAS.95 b], [MAS.98 b]. Nous avons aussi montré les performances de cette architecture de point de vue temps de calcul et surface d'intégration [ELO.96]. Cette architecture pose une dualité entre la surface d'intégration et la vitesse de calcul du processeur. Pour rendre le processeur très rapide il faut augmenter le nombre de processeurs élémentaires ce qui augmente la surface d'intégration, donc un problème de coût. Une bonne implantation en VLSI demande une faible surface d'intégration donc un faible nombre de processeurs élémentaires. Une solution est d'augmenter la longueur des piles z, k et h, mais cela cause un ralentissement au niveau reconstitution du signal.

Nous serons donc devant un compromis entre la surface d'intégration, S et le temps de calcul t. En générale, le compromis revient à minimisé un critère d'optimisation comme St ou St^2.

Dans ce qui suit nous allons présenter une version de l'architecture SYSKAL constituée de deux processeurs élémentaires qui présentera une solution à la réduction de la surface du processeur. Nous passerons par la suite à l'application de la méthode du pipeline conventionnel sur cette version de SYSKAL.

3.3.1 Architecture SYSKAL à deux processeurs élémentaires

L'architecture du processeur SYSKAL est semi-systolique à structure en anneaux présentant quatre processeurs élémentaires, comme nous l'avons déjà remarqué ce nombre de processeurs nuit à la surface d'intégration, nous avons donc choisi d'utiliser seulement deux processeurs élémentaires d'une part pour réduire la surface d'intégration, et d'autre part pour obtenir une plus grande flexibilité des piles afin d'appliquer un pipeline profond. La figure 3.6 présente la structure interne de cette version de l'architecture sur laquelle sera appliquée le pipeline.

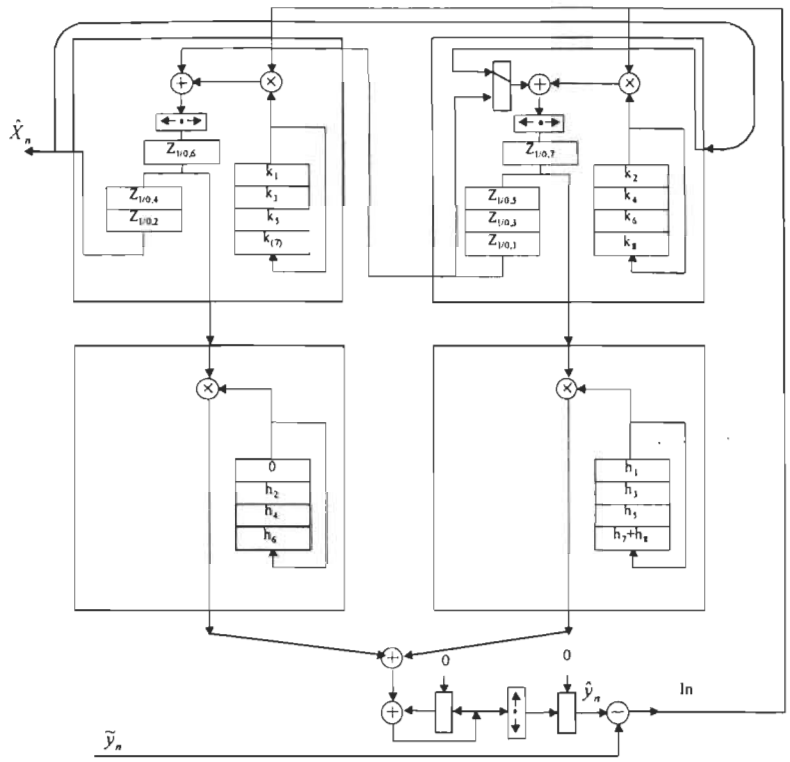


Figure 3.6 Structure interne du processeur réduit

Cette architecture présente les mêmes propriétés que celle présentée au chapitre 2 sauf qu'il y a absence de processeurs élémentaires intermédiaires. Les séquences de fonctionnement de cette architecture sont les mêmes que la précédente à la seule différence que la longueur des piles est passé d'une valeur de M/4 à une valeur de M/2. Cette version de l'architecture montre une diminution de la surface d'intégration par rapport à celle de quatre PE et une augmentation du temps de calcul.

Afin de saisir le fonctionnement complet de cette architecture nous avons présenté à la figure 3.7, le flot de données pour des piles de longueur 5 c'est-à-dire que M=10.

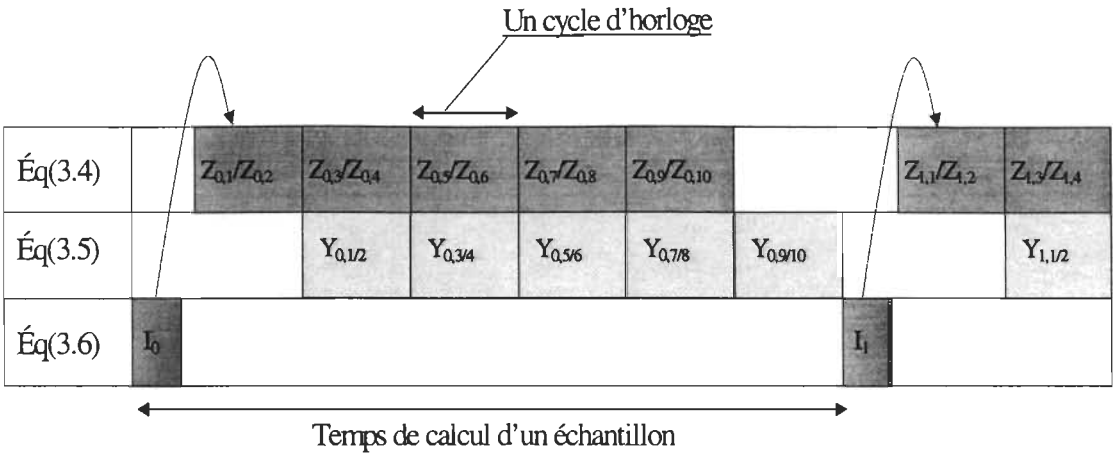


Figure 3.7 Séquences de fonctionnement du processeur SYSKAL

Nous allons utiliser cette nouvelle architecture comme architecture de base pour lui appliquer les différentes méthodes de pipeline. Pour cela nous l'avons modélisé en langage VHDL. Tous les programmes sont décrits à l'annexe A. Le programme principal est présenté à la figure 3.8 et la figure 3.9 présente les résultats de simulations de cette architecture. La figure 3.10 a) présente la structure interne alors que la figure 3.10 b) présente la structure en bloc de la même architecture. Sur cette architecture nous avons présenté les deux processeurs élémentaires A et B composé des piles z_A, k_A et h_A et z_B, k_B et h_B respectivement. Les séquences de fonctionnement de cette architecture peuvent être décrites en trois équations:

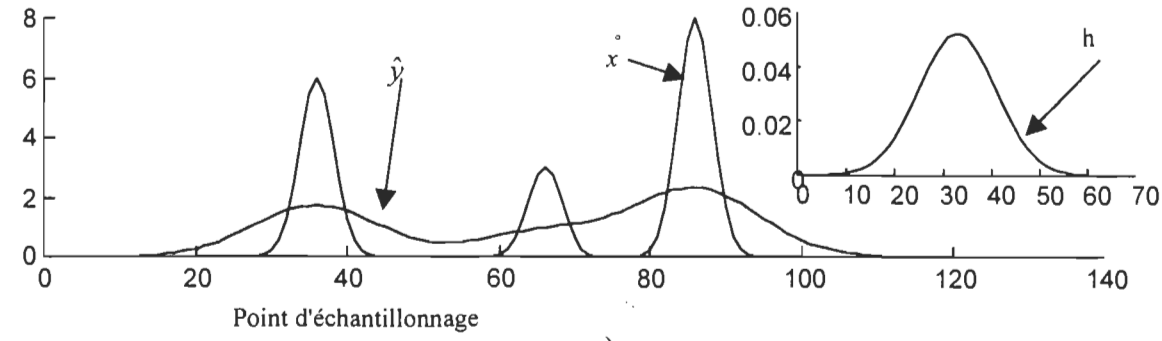
$$\hat{z}_{n+1/n,m-1} = \hat{z}_{n/n-1,m} + k_{\infty,m} I_n \tag{3.4}$$

$$\hat{y}_{n+1/m-1} = h_{m-1} \hat{z}_{n+1/n,m-1} + \hat{y}_{n+1/m-2} \tag{3.5}$$

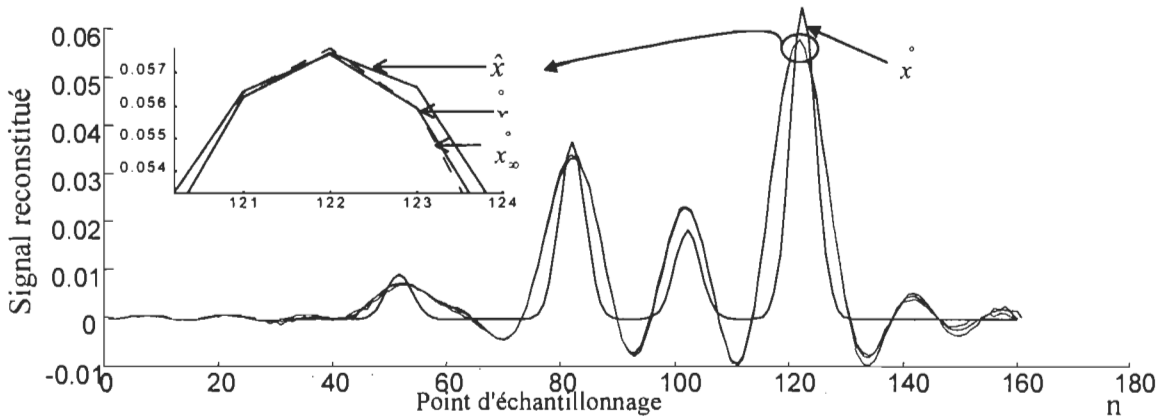
$$I_n = \tilde{y}_n - \hat{y}_n \tag{3.6}$$

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE work.pack_processeur_syskal.all;
   ENTITY processeur_syskal IS
PORT(pil_ink1,pil_ink2          :IN   std_logic_vector(7 downto 0);
     pil_inh1,pil_inh2          :IN   std_logic_vector(7 downto 0);
     clk,clr_Z,chain, s         :IN   std_logic;
     en_Z,en_K,en_s,en_h        :IN   std_logic;
     y_mes                      :IN   std_logic_vector(15 downto 0);
     clr_acc,en_acc,clr_s       :IN   std_logic;
     distanc                    :IN   std_logic_vector(5 downto 0);
     en_x , clr_x               :IN   std_logic;
     clr_cal, en_cal            :IN   std_logic;
     xout                       :out  std_logic_vector(15 downto 0));
   END processeur_syskal ;
   ARCHITECTURE comportement OF processeur_syskal IS
          signal mult1,mult2,add_out1,add_out2     :std_logic_vector(15 downto 0);
          signal mux_out, Z_out1,Z_out2            :std_logic_vector(15 downto 0);
          signal pil_outk1 ,pil_outk2              :std_logic_vector(7 downto 0);
          signal pil_outh1 ,pil_outh2              :std_logic_vector(7 downto 0);
          signal sortie1 ,sortie2                  :std_logic_vector(15 downto 0);
          signal yout1,yout2, yout                 :std_logic_vector(15 downto 0);
          signal yacc_in, yacc_out                 :std_logic_vector(15 downto 0);
          signal y_cal_in, y_cal_out               :std_logic_vector(15 downto 0);
          signal innovation                        :std_logic_vector(15 downto 0);
   begin
          soustracteur   :soustract1              port map (y_mes , y_cal_out , inovation);
          MULT_1         :multiplieur_16_8        port map ( inovation,pil_outk1,mult1 );
          MULT_2         :multiplieur_16_8        port map ( inovation,pil_outk2,mult2 );
          ADD_1          :adder1                  port map (mult1,mux_out, add_out1);
          ADD_2          :adder1                  port map (mult2,Z_out1 , add_out2);
          Reg1           :reg16                   port map (add_out1,clk,en_s,clr_s,sortie1);
          Reg2           :reg16                   port map (add_out2,clk,en_s,clr_s,sortie2);
          BLOC_Z1        :bloc_stock_donnees_33   port map (clk,en_Z,clr_Z,sortie1,Z_out1);
          BLOC_Z2        :bloc_stock_donnees_32   port map (clk,en_Z,clr_Z,sortie2,Z_out2);
          Multiplex      :mux2a1                  port map (Z_out1,Z_out2,s,mux_out);
          BLOC_K1        :PILE_34_8               port map (clk, en_K, chain, pil_ink1, pil_outk1);
          BLOC_K2        :PILE_34_8               port map (clk, en_K, chain, pil_ink2, pil_outk2);
          MULT_3         :multiplieur_16_8        port map ( sortie1,pil_outh1,yout1 );
          MULT_4         :multiplieur_16_8        port map ( sortie2,pil_outh2,yout2 );
          BLOC_h1        :PILE_34_8               port map (clk, en_h, chain, pil_inh1, pil_outh1);
          BLOC_h2        :PILE_34_8               port map (clk, en_h, chain, pil_inh2, pil_outh2);
          ADD_3          :adder1                  port map (yout1,yout2 , yout);
          ADD_4          :adder1                  port map (yout,yacc_in , yacc_out);
          Reg3           :reg16                   port map (yacc_out,clk,en_acc,clr_acc,yacc_in);
          Decaleur       :decaleur_gauch_droit    port map (distanc,yacc_in,y_cal_in);
          Reg4           :reg16                   port map (y_cal_in,clk,en_cal,clr_cal,y_cal_out);
          Reg6           :reg16                   port map (z_out2 ,clk,en_x,clr_x,xout);
   END comportement ;
```

Figure 3.8 Programme principale du processeur SYSKAL



a)



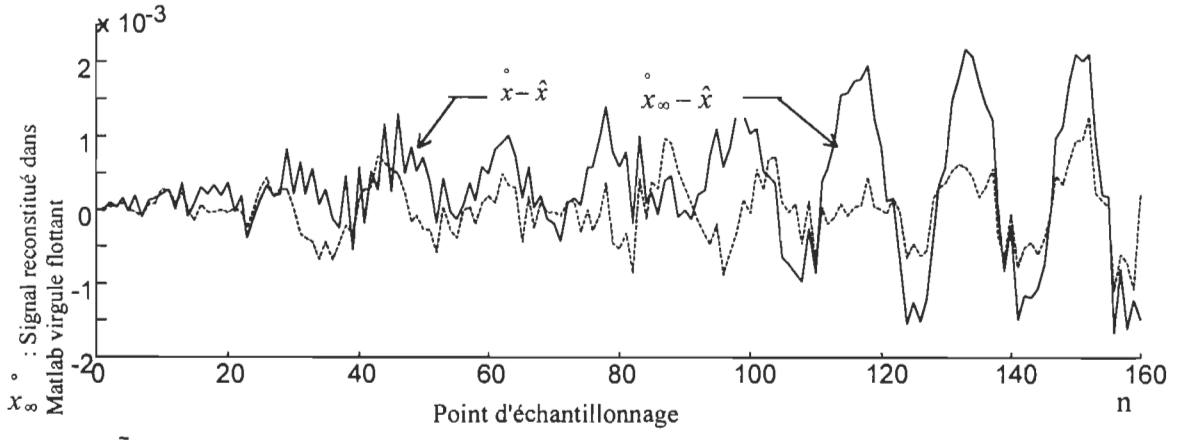
$\hat{x}_{sys}$: Signal reconstitué dans SYSKAL

$\overset{\circ}{x}_{\infty}$: Signal reconstitué dans Matlab virgule flottant

$\overset{\circ}{x}_{q}$: Signal reconstitué dans Matlab virgule fixe

$\overset{\circ}{x}$: Signal idéal

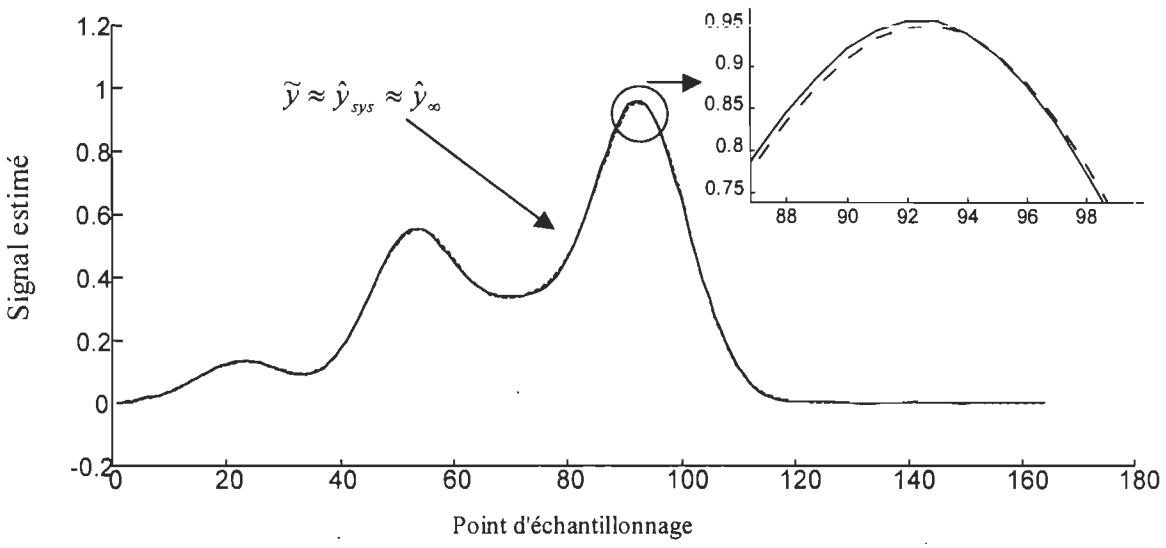
b)



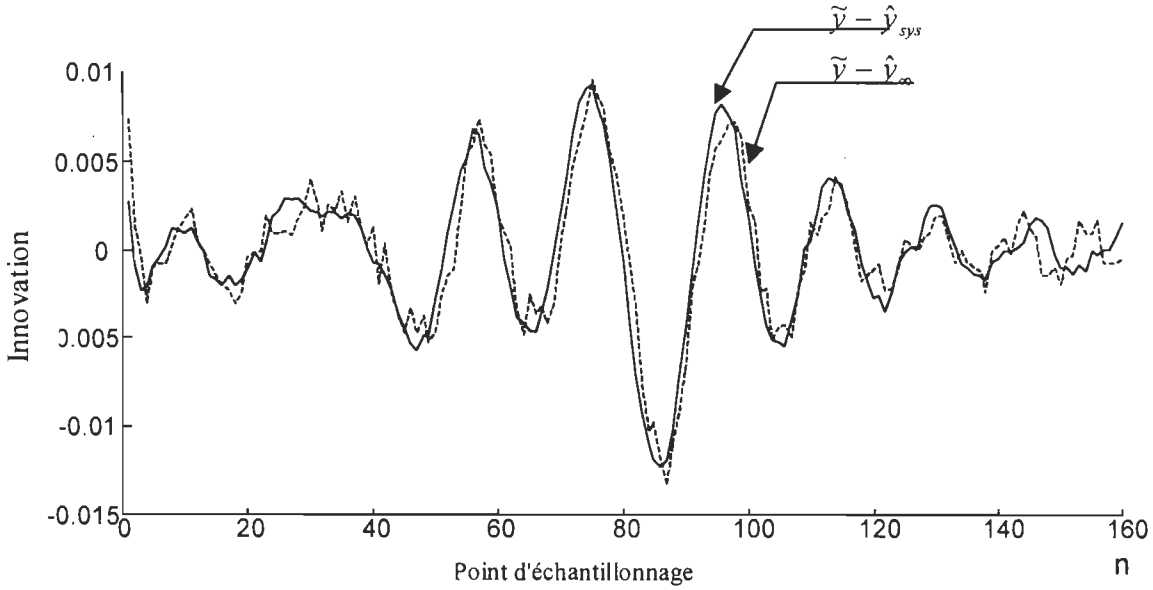
c)

Figure 3.9 : Résultats de simulation de SYSKAL a) signal idéal

b) signal reconstitué $\hat{x}_{sys}$, $\overset{\circ}{x}_{q}$ et $\overset{\circ}{x}_{\infty}$ c) erreur de reconstitution



d)



e)

Figure 3.9 (suite) : Résultats de simulation de SYSKAL d) signal estimé e) innovation

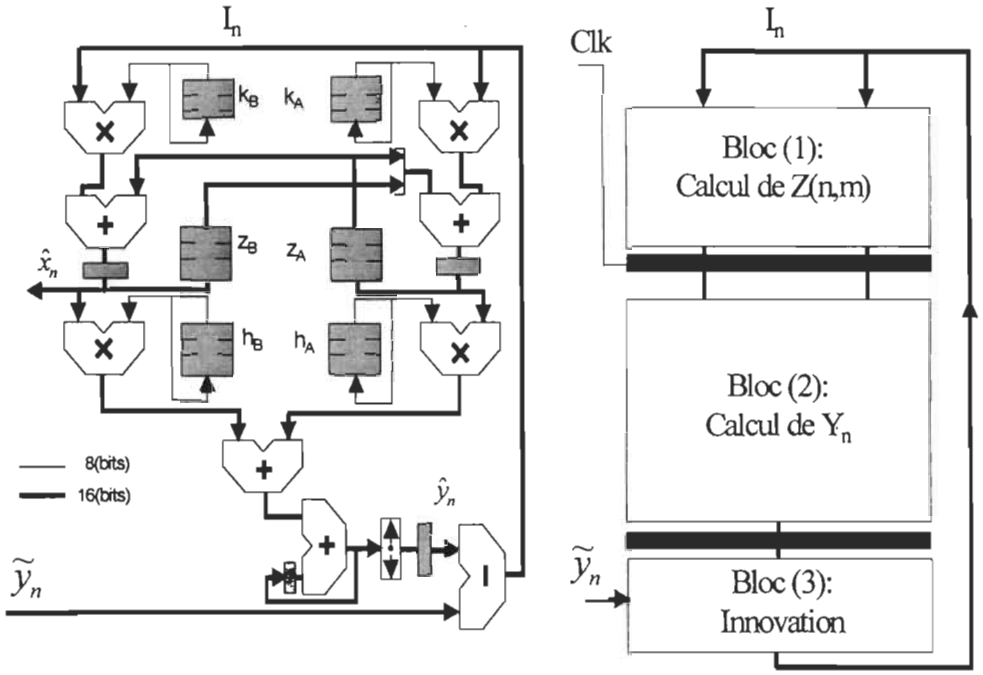


figure 3.10 Structure de l'architecture du processeur SYSKAL: a) Structure interne de l'architecture et b) Structure en bloc de l'architecture

3.3.2 Flot de données pipeliné: PIPEKAL$_I$

L'amélioration de l'architecture de point de vu temps de calcul peut être résumée à une diminution des délais de propagation le long des trois principaux blocs qui sont: bloc(1) pour le calcul de Z, bloc(2) pour le calcul de Y et le bloc(3) pour le calcul de l'innovation. Le temps de calcul d'un point d'échantillonnage peut être décrit selon la forme suivante:

$$t_{\hat{x}} = [\frac{M}{2}\max(d_{b1}, d_{b2})] + d_{b3} \qquad (3.7)$$

où $t_{\hat{x}}$ est le temps de reconstitution d'un point d'échantillonage, d_{b1} est le délai du bloc(1) et d_{b2} est le délai du bloc(2) et d_{b3} est le délai du bloc(3).

De l'équation (3.7) on peut dire que l'amélioration de $t_{\hat{x}}$ peut s'effectuer en diminuant d_{b1} et d_{b2} puisque M/2 égale 32. Ce qui signifie que le bloc(3) ne fonctionne qu'une fois sur les (M/2 +1) coups d'horloge ce qui nous permet de négliger le délai d_{b3} devant les autres délais. Une première approche pour augmenter le débit de données est l'application de la méthode de pipeline sur l'architecture. Ceci consiste à séparer tous les principaux blocs de logique combinatoire (BLC) comme les multiplieurs et les additionneurs par des registres. Ce type de pipeline nous l'avons appelé pipeline externe puisqu'il ne touche qu'à l'architecture mais pas au BLC. L'architecture obtenue appelée $PIPEKAL_1$ est présentée à la figure 3.11 où on voit la structure interne et le schéma blocs. Les séquences de fonctionnement de cette architecture sont :

1) $M^A_{z\,n+1/n,2m} = k_{\infty,2m} I_n$ et $M^B_{z\,n+1/n,2m-1} = k_{\infty,2m-1} I_n$ (3.8)

2) $S^A_z(Z^A_{n+1/n,m}) = Z^B{}_{n/n-1,m-1} + M^A_{z\,n+1/n,m-1}$

$S^A_z(Z^B_{n+1/n,m}) = Z^A{}_{n/n-1,m-1} + M^B_{z\,n+1/n,m-1}$ (3..9)

3) $M^A_{h\,n+1/m-1} = h_{m-1} Z^A{}_{n+1/n,m-1}$

$M^B_{h\,n+1/m-1} = h_{m-1} Z^B{}_{n+1/n,m-1}$ (3.10)

4) $S_{hn+1/m} = M^A_{h\,n+1,m} + M^B_{h\,n+1,m}$ (3.11)

5) $\hat{y}_{n+1} = S_{hn+1,m} + \hat{y}_{n+1,m-1}$ (3.12)

6) $I_n = \tilde{y}_n - \hat{y}_n$ (3.13)

Où M^A et M^B représentent l'opérateur de multiplication pour les processeurs élémentaires A et B respectivement, l'opérateur de sommation pour les processeur A et B sont représentés par S^A et S^B.

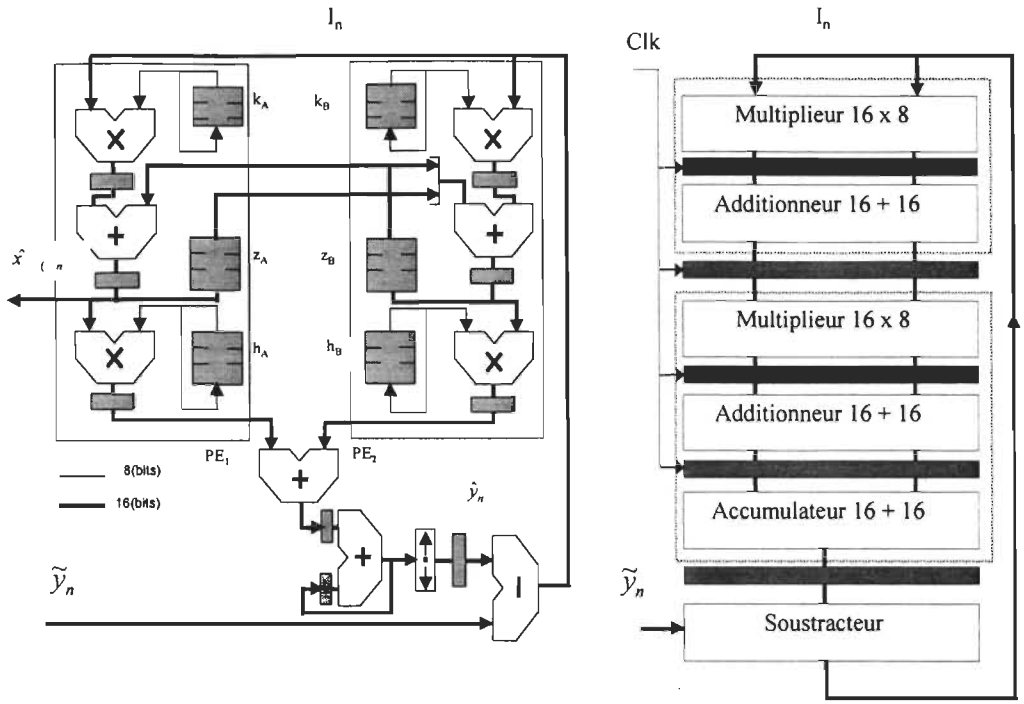


Figure 3.11 Architecture Interne de PIPEKAL$_1$: a) Structure interne, b) Structure en blocs

Nous avons donc augmenté le nombre d'étages, on a passé de trois à six étapes, ce qui a pour effet d'augmenter la latence. Par contre grâce à la réduction des délais pour réaliser une étape, nous avons réduit le temps de calcul d'un échantillon $\hat{x}_n$. La fréquence de fonctionnement ne sera limitée dans ce cas que par les BLC et surtout par le multiplieur 16×8 bits. La figure 3.12 montre le flot de données de l'architecture PIPEKAL$_1$ pour une longueur de piles de 10 soit M=20. Les performances pour réaliser la reconstitution d'un échantillon de cette architecture évaluée en terme de débit et de la latence sont respectivement :

$$Débit = [(\frac{M}{2} + 5)t_{d\max} + t_{sous}]^{-1} \quad (3.14)$$

$$Latence = (\frac{M}{2} + 5)t_{d\max} + t_{sous} \quad (3.15)$$

où t_{dmax} est le délai maximal du bloc le plus lent et t_{sous} est le délai du soustracteur.

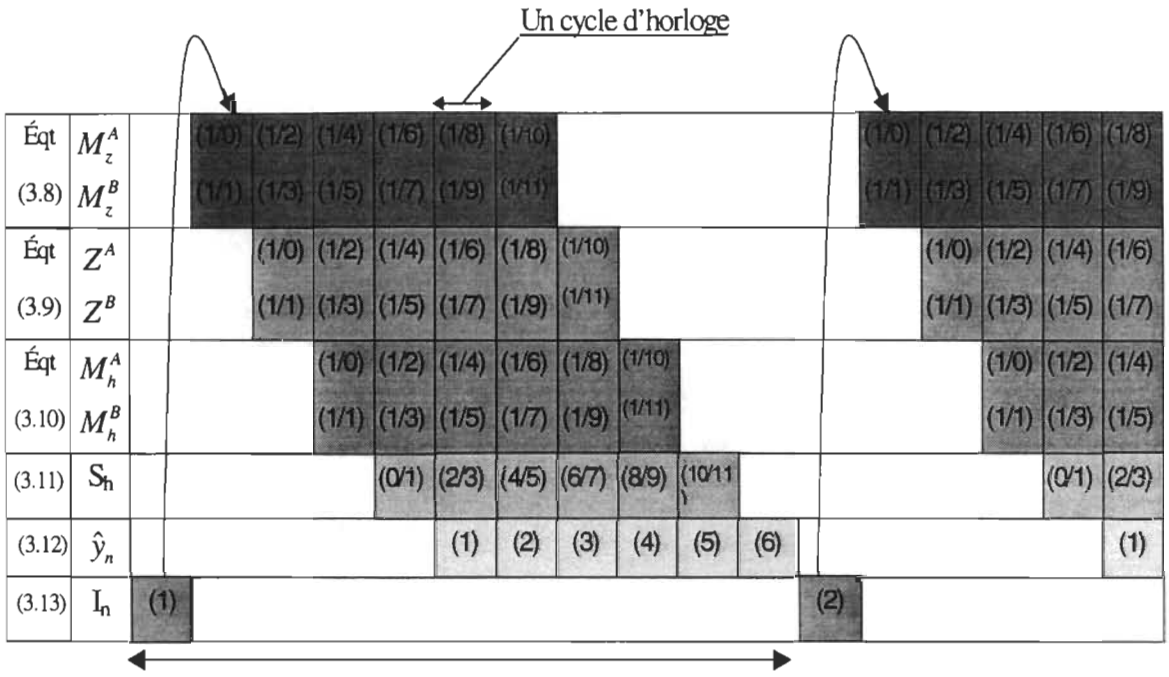


Figure 3.12 Flot de données de $PIPEKAL_1$

Une comparaison de $PIPEKAL_1$ avec l'architecture SYSKAL montre une augmentation du nombre de cycle d'horloge pour reconstituer un échantillon du signal. Par contre, dû à la présence du pipeline le temps du cycle d'horloge est passé de 125 ns à 40 ns puisque le chemin critique est maintenant celui d'un multiplieur.

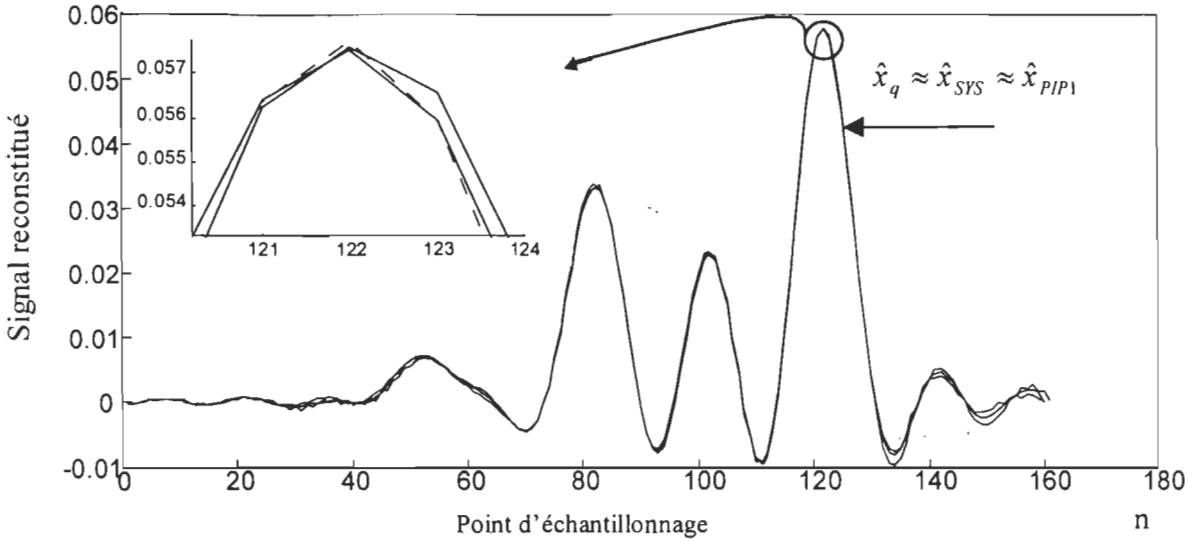
La performance de point de vue latence et débit est :

$$\text{Temps de calcul}(SYSKAL) = (\frac{M}{2} + 2).f_{\max} \qquad (3.16)$$

$$\text{Temps de calcul}(PIPEKAL_1) = (\frac{M}{2} + 5).f_{\max} \qquad (3.17)$$

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE work.pack_processeur_pipeakl1.all;
   ENTITY processeur_PIPKAL1 IS
PORT( clk,clr_Z ,chain, s, en_Z,en_K                :IN   std_logic;
      clr_acc,en_acc,clr_s, en_x , en_s             :IN   std_logic;
      clr_cal, en_cal, clr_reg, clr_x,  en_h        :IN   std_logic;
      pil_ink1,pil_ink2, pil_inh1,pil_inh2          :IN   std_logic_vector(7 downto 0);
      y_mes, distanc                                :IN   std_logic_vector(15 downto 0);
      xout                                          :out  std_logic_vector(15 downto 0));
   END processeur_PIPKAL1 ;
   ARCHITECTURE comportement OF processeur_PIPKAL1 IS
            signal mult1,mult2,add_out1,add_out2     :std_logic_vector(15 downto 0);
            signal mux_out, Z_out1,Z_out2            :std_logic_vector(15 downto 0);
            signal pil_outk1 ,pil_outk2              :std_logic_vector(7 downto 0);
            signal pil_outh1 ,pil_outh2              :std_logic_vector(7 downto 0);
            signal sortie1 ,sortie2                  :std_logic_vector(15 downto 0);
            signal yout1,yout2, yout, yacc_in, yacc_out :std_logic_vector(15 downto 0);
            signal y_cal_in, y_cal_out, innovation   :std_logic_vector(15 downto 0);
begin
            soustracteur  :soustract1          port map (y_mes , y_cal_out , inovation);
            MULT_1        :multiplieur_16_8    port map ( inovation,pil_outk1,mult1 );
            MULT_2        :multiplieur_16_8    port map ( inovation,pil_outk2,mult2 );
            Reg_1         :Register_16         port map(mult1, clk, mult11);
            Reg_2         :Register_16         port map(mult2, clk, mult22);
            ADD_1         :adder1              port map (mult1,mux_out, add_out1);
            ADD_2         :adder1              port map (mult2,Z_out1 , add_out2);
            Reg_3         :reg16               port map (add_out1,clk,en_s,clr_s,sortie1);
            Reg_4         :reg16               port map (add_out2,clk,en_s,clr_s,sortie2);
            BLOC_Z1       :bloc_stock_donnees_33 port map (clk,en_Z,clr_Z,sortie1,Z_out1);
            BLOC_Z2       :bloc_stock_donnees_32 port map (clk,en_Z,clr_Z,sortie2,Z_out2);
            Multiplex     :mux2a1              port map (Z_out1,Z_out2,s,mux_out);
            BLOC_K1       :PILE_34_8           port map (clk, en_K, chain, pil_ink1, pil_outk1);
            BLOC_K2       :PILE_34_8           port map (clk, en_K, chain, pil_ink2, pil_outk2);
            MULT_3        :multiplieur_16_8    port map ( sortie1,pil_outh1,yout1 );
            MULT_4        :multiplieur_16_8    port map ( sortie2,pil_outh2,yout2 );
            Reg_5         :reg16               port map (yout1,clk,yout11);
            Reg_6         :reg16               port map (yout2,clk,yout22);
            BLOC_h1       :PILE_34_8           port map (clk, en_h, chain, pil_inh1, pil_outh1);
            BLOC_h2       :PILE_34_8           port map (clk, en_h, chain, pil_inh2, pil_outh2);
            ADD_3         :adder1              port map (yout11,yout22 , yout);
            Reg_7         :reg16               port map (yout,clk,yout5);
            ADD_4         :adder1              port map (yout5,yacc_in , yacc_out);
            Reg_8         :reg16               port map (yacc_out,clk,yacc_out1);
            Reg3          :reg16               port map (yacc_out1,clk,en_acc,clr_acc,yacc_in);
            Decaleur      :decaleur_gauch_droit port map (distanc,yacc_in,y_cal_in);
            Reg4          :reg16               port map (y_cal_in,clk,en_cal,clr_cal,y_cal_out);
            Reg6          :reg16               port map (z_out2 ,clk,en_x,clr_x,xout);
   END comportement ;
```

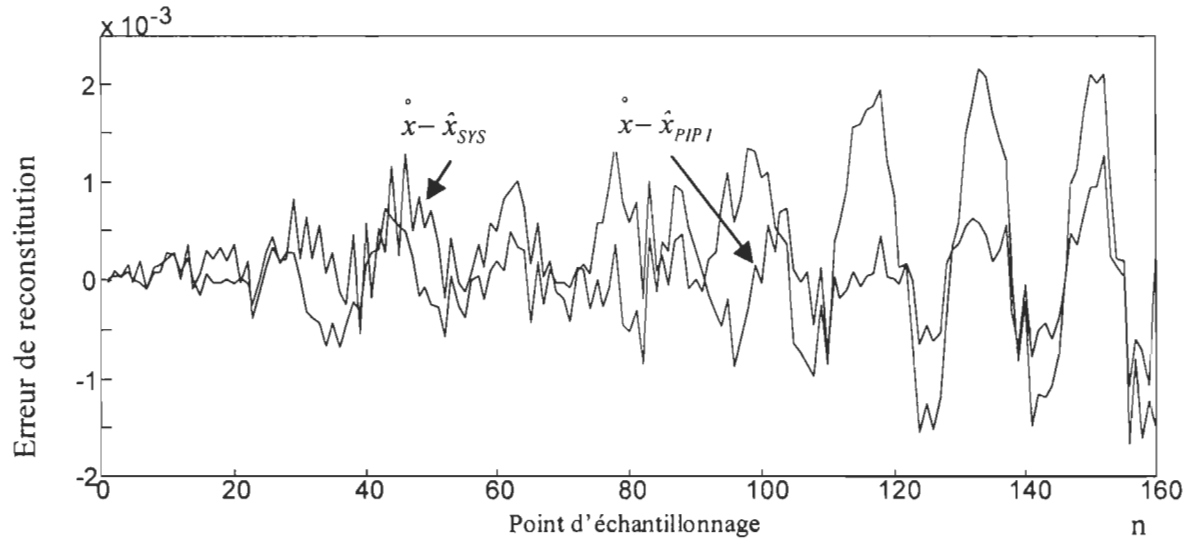
Figure 3.13 Programme principal du processeur PIPEKAL$_1$



$\hat{x}_{SYS}$: Signal reconstitué par SYSKAL

$\hat{x}_{PIP1}$: Signale reconstitué par PIPEKAL$_1$

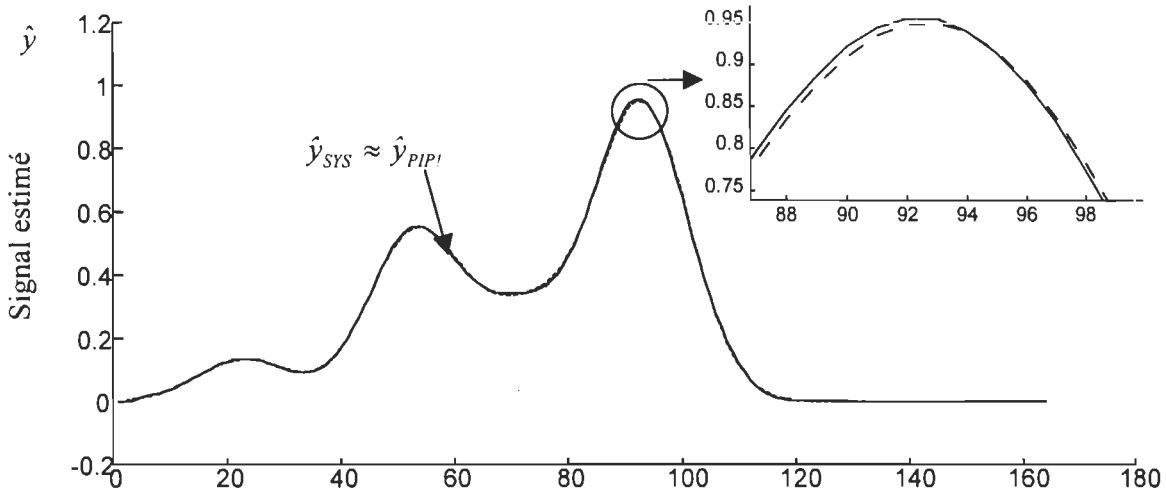
a)



b)

Figure 3.14 : Résultats de simulation de PIPEKAL$_1$ a) Signale reconstitué

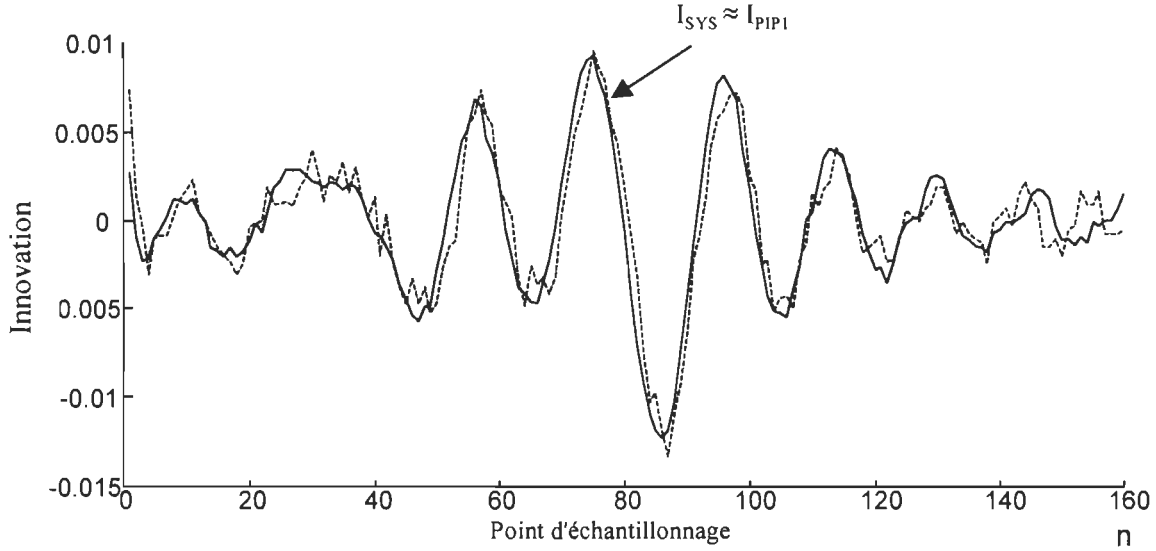
b) Erreur de reconstitution dans SYSKAL et PIPEKAL$_1$



$\hat{y}_{SYS}$: Signale estimé par SYSKAL

$\hat{y}_{PIP1}$: Signal estimé par PIPEKAL$_1$

c)



I_{SYS} : Innovation calculé dans SYSKAL

I_{PIP1} : Innovation calculé dans PIPEKAL$_1$

d)

Figure 3.14 : Résultats de simulation de PIPEKAL$_1$ (suite) a) Signal estimé b) Innovation

3.3.3 Blocs à logique combinatoire pipelinés: PIPEKAL$_2$

Les équations (3.14) et (3.15) montrent bien que le temps de calcul du processeur dépend essentiellement de la fréquence de fonctionnement et du nombre d'étages pipelinés. Alors qu'avec PIPEKAL$_1$ il semble bien difficile d'atteindre des fréquences de fonctionnement très élevés, puisque chaque bloc est caractérisé par sa propre fréquence de fonctionnement. Dans ce paragraphe nous allons adopter un autre type de pipeline qui nous permettra d'approfondir le pipeline à l'intérieur de chaque BLC.

Appliquer ce type de pipeline demande des connaissances approfondies de la construction interne de chaque BLC, cela dit nous allons subdiviser les BLC de façon matérielle on tenant compte des égalités des délais entre les différents sous-blocs.

Nous avons vu dans le chapitre précédent que le multiplieur et l'additionneur présentent les blocs les plus lents de l'architecture puisqu'ils utilisent des vecteurs d'entrées de 16×8 bits pour le multiplieur et 16×16 bits pour l'additionneur. Dans ce qui suit, nous allons appliquer le pipeline conventionnel aux additionneurs et aux multiplieurs. Le choix de cet ordre vient du fait que l'additionneur fait une partie intégrale du multiplieur. Par la suite, nous allons intégrer ces différents BLC pipelinés dans l'architecture PIPEKAL$_2$.

3.3.3.1 Application du pipeline conventionnel sur l'additionneur

Un additionneur 16 bits est un circuit combinatoire permettant d'effectuer l'addition sur deux variables A et B de 16 bits et donnant une sortie C de 16 bits. Le problème majeur dans l'addition est la dépendance entre les résultats intermédiaires. En effet cela peut se voir clairement lors du calcul du bit le plus significatif exprimé comme suit:

$$S_{15} = a_{15} + b_{15} + c_{14}$$

$$c_{14} = f(a_0 \ldots a_{15}, b_0 \ldots b_{15}, c_0 \ldots c_{13})$$

avec S_{15} est le résultat du 16^{iem} bits

a_i, b_i et c_i : les entrées et le retenus

La figure 3.15 montre le cas d'un additionneur 4×4 bits. Dans cette figure on remarque que le chemin le plus lent est celui du retenu du 4^{ieme} bit qui traverse les quatres blocs successivement alors que le reste des bits ne traversent qu'un seul à la fois. Sachant que chaque bloc peut avoir un délai de propagation noté d_b. Le délai le plus lent dans notre exemple est donc de l'ordre de $4d_b$. Cela montre que le délai de propagation le long des additionneurs est proportionnel au nombre de bits d'entrées, se qui pose de sérieux problème lors de l'utilisation des additionneurs très larges.

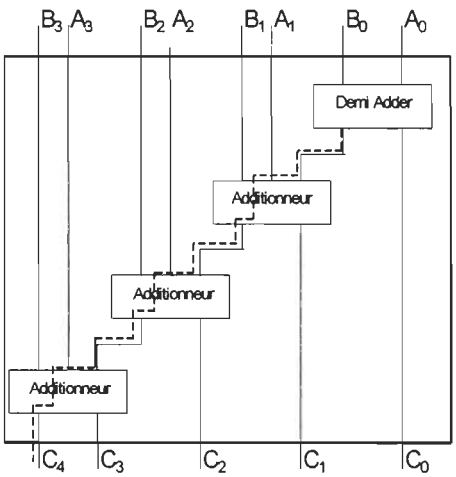


Figure 3.15 Additionneur 4 × 4 bits.

Plusieurs travaux ont été effectués pour réduire le temps de réponse des additionneurs [HEN.92]. Parmi les plus intéressants on peut citer les ***additionneurs à retenu anticipé*** (Carry Loockahed Adder) et les ***additionneurs à saut de retenu***. Une comparaison entre

ces différentes méthodes est présentée au tableau 3.1. Le choix de l'additionneur le plus performant pour nous applications se base sur le critère St^2. Nous remarquons alors que seuls l'additionneur à retenu anticipé présentent un temps de réponse court. C'est pourquoi notre choix du type d'additionneur c'est arrêté sur celui ci.

Tableau 3.1 Comparaison de différents types d'additionneurs [HEN.92]

	Temps	*surface*
Propagation de retenu	***f(n)***	***f(n)***
Retenu anticipé	***f(log n)***	***f(n log n)***
Saut de retenu	$f(\sqrt{n})$	***f(n)***
Sélection de retenu	$f(\sqrt{n})$	***f(n)***

Pour pipeliner un bloc de logique, il faut toujours passer par les trois pricipales suivantes:

1) Spécification du problème : dans notre cas il s'agit d'un additionneur 16×16 bits avec complément à 1. Nous avons utilisé le complément à un pour simplifier le problème et aussi parce que l'erreur introduite est négligeable dans le cas de 16 bits. Dans le cas normal, l'additionneur 16 bits a un délai de propagation de 33 ns soit une fréquence de 30 MHz.
2) Partager l'additionneur : pour partager un bloc de logique combinatoire, il faut tout d'abord avoir accès à sa structure interne. Pour cela, nous l'avons modélisé en langage VHDL, par la suite nous l'avons partagé en trois blocs où chaque a bloc à un délai de propagation de 11 ns dans une technologie CMOS de 1,5µm. Le premier et le dernier blocs sont formés d'un bloc complément à un et un

additionneur complet 4×4 bits, alors que le deuxième est formé par un additionneur 8×8 bits.

3) Insertion des registres : une fois que les simulations sont concluantes nous allons insérer par la suite les registres qui permettront de synchroniser la propagation des données. La figure 3.16 montre la structure globale de l'additionneur pipeliné.

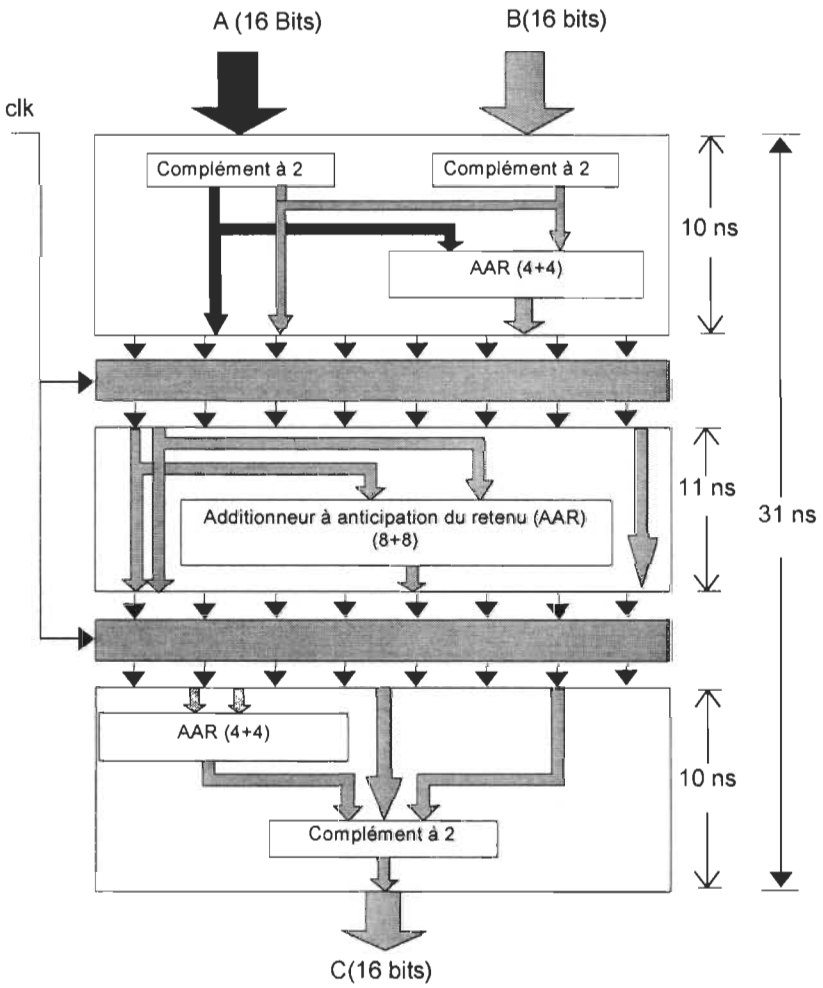


Figure 3.16 Structure interne de l'additionneur pipeliné

3.3.3.2 Résultats de simulation de l'additionneur pipeliné

Les résultats de simulation de l'additionneur ont été faits sur MENTOR GRAPHICS. Nous avons fait l'addition de deux signaux générés par les fonctions sin(x) avec un sin(2x). Nous avons comparé les résultats avec ceux obtenus de MATLAB. Ces résultats sont présentés à la figure 3.17.

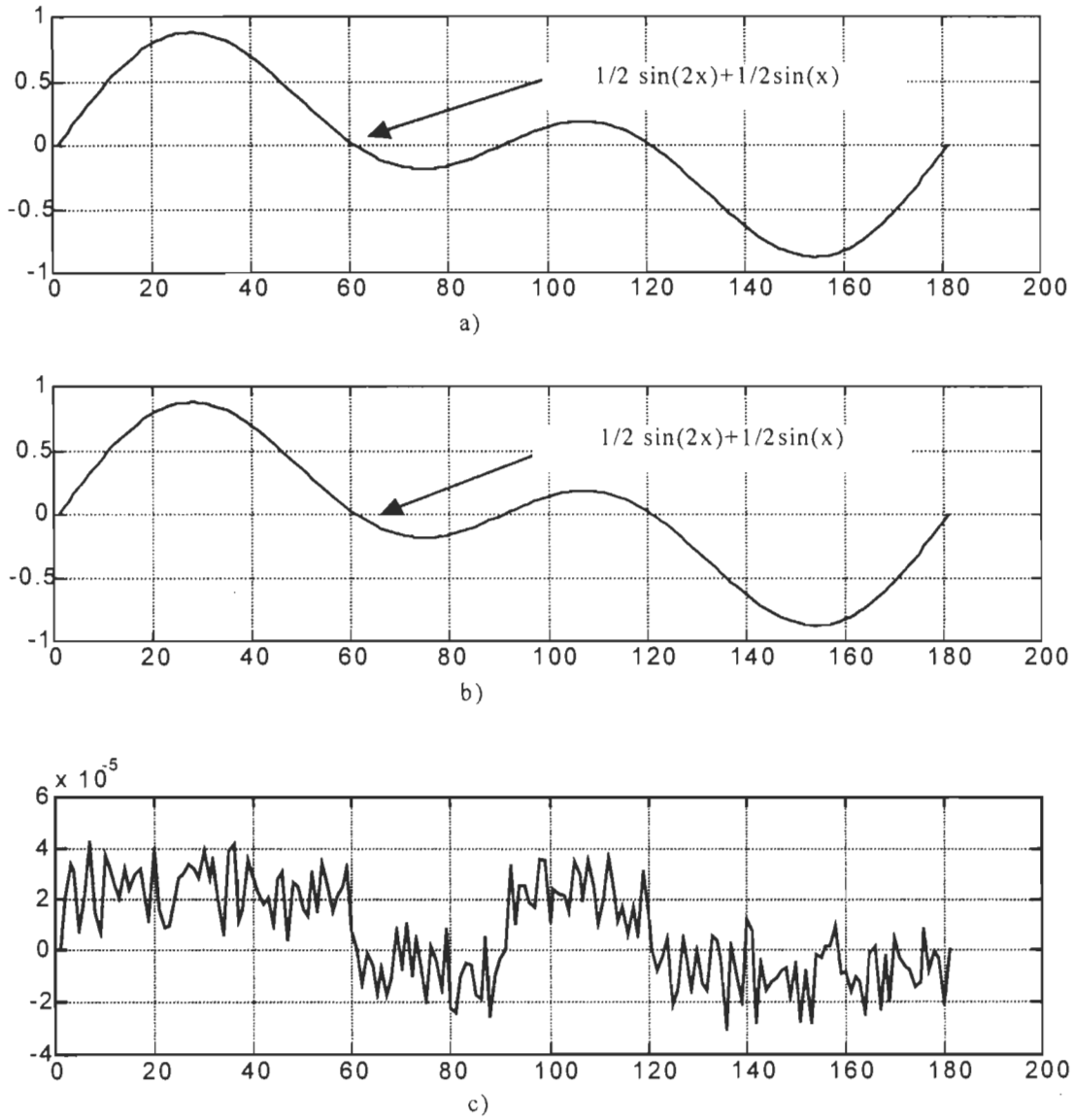


Figure 3.17 Résultats de simulation de l'additionneur pipeliné

3.3.3.3 Application du pipeline conventionnel sur le multiplieur

De la même façon que pour l'additionneur, nous allons procéder pour l'application du pipeline synchrone sur le multiplieur

1) Spécification du problème : le multiplieur utilisé dans le processeur PIPEKAL$_1$ est un multiplieur 16 × 8 bits donnant un résultat sur 16 bits. Le chemin a un délai de propagation de 36,42ns soit donc une fréquence de 27 MHz pour une technologie de 1,5µm.
2) Partager le multiplieur :comme nous l'avons déjà mentionné, c'est l'étape la plus difficile. Nous avons choisi d'utiliser des multiplieurs de base de dimension 4 × 4 bits, ceci a fixé le délai de propagation pour le premier bloc du pipeline à 11,2 ns, cela explique aussi le choix dans l'additionneur. En effet pour maîtriser cette partie nous allons présenter un exemple qui montrera en détails cette étape pour un multiplieur 16×8 bits.

Généralement on peut décrire une multiplication 16 × 8 bits comme une somme partielle des différents produits intermédiaires, tels que montré à la figure 3.18 selon que:

$A = (a_{15},a_{14}, a_{13}, a_{12}, a_{11}, a_{10}, a_9, a_8, a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$

$B = (b_7,b_6,b_5,b_4,b_3, b_2,b_1,b_0)$

On peut donc sectionner le produit de 16×8 bits en une série de produits s'effectuant de façon parallèle en 4×4 bits, puis les résultats seront traites par la méthode de l'arbre de WALLACE qui permettra de donner deux vecteurs: le vecteur somme et le vecteur retenu. Ces deux derniers seront additionnés l'un à l'autre pour donner finalement le résultat du produit. L'additionneur final est un additionneur à retenu anticipé, nous avons

donc utilisés le même additionneur 16×16 bits déjà étudié. Pour plus de détail sur le fonctionnement de l'arbre de WALLACE nous inviterons les lecteurs à prendre connaissance du référence suivant [HEN 92].

Pour répondre aux exigences de l'architecture nous avons tronqué les bits les moins significatifs de tel sorte d'avoir une sortie de 16 bits. Nous avons donc sectionné le multiplieur en quatre étages, le premier permet d'effectuer le 8 multiplications partielles 4×4 bits, le deuxième bloc effectue l'addition par la méthode de Wallace et une partie de l'addition à retenu anticipé 4×4 bits et les deux derniers blocs terminent l'addition avec des additionneurs de 8×8 bits. La figure 3.18 montre en détails les différents blocs du multiplieur déjà partagé

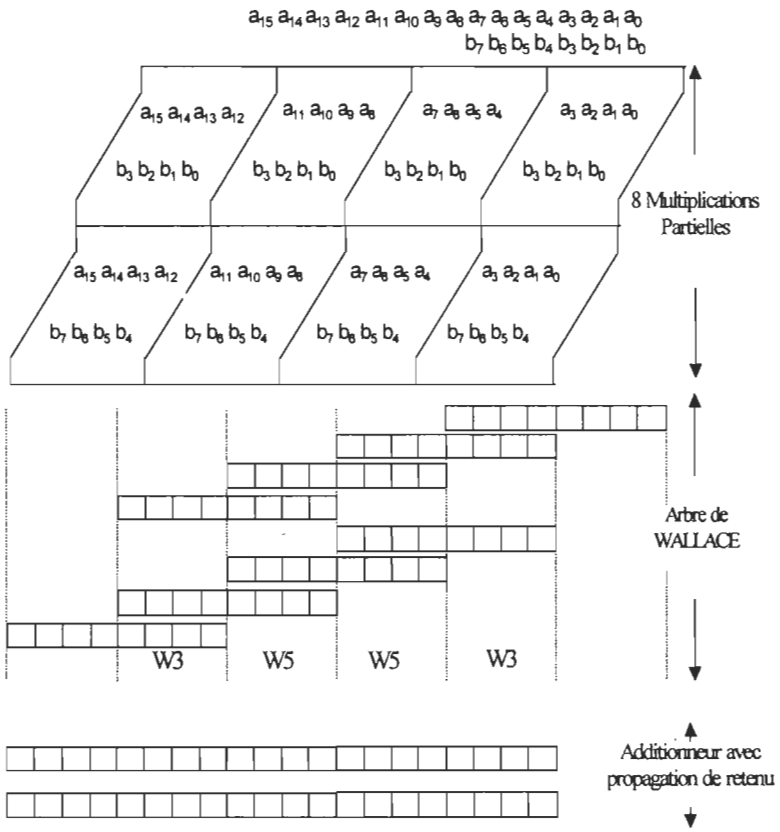


Figure 3.18 : différents bloc d'un multiplieur 16 × 8

3) Insertion des registres

De la même façon que l'additionneur dans cette étape nous allons juste introduire les registres dans le multiplieur de telle sorte que le flot de données soit complètement synchrone (figure 3.19).

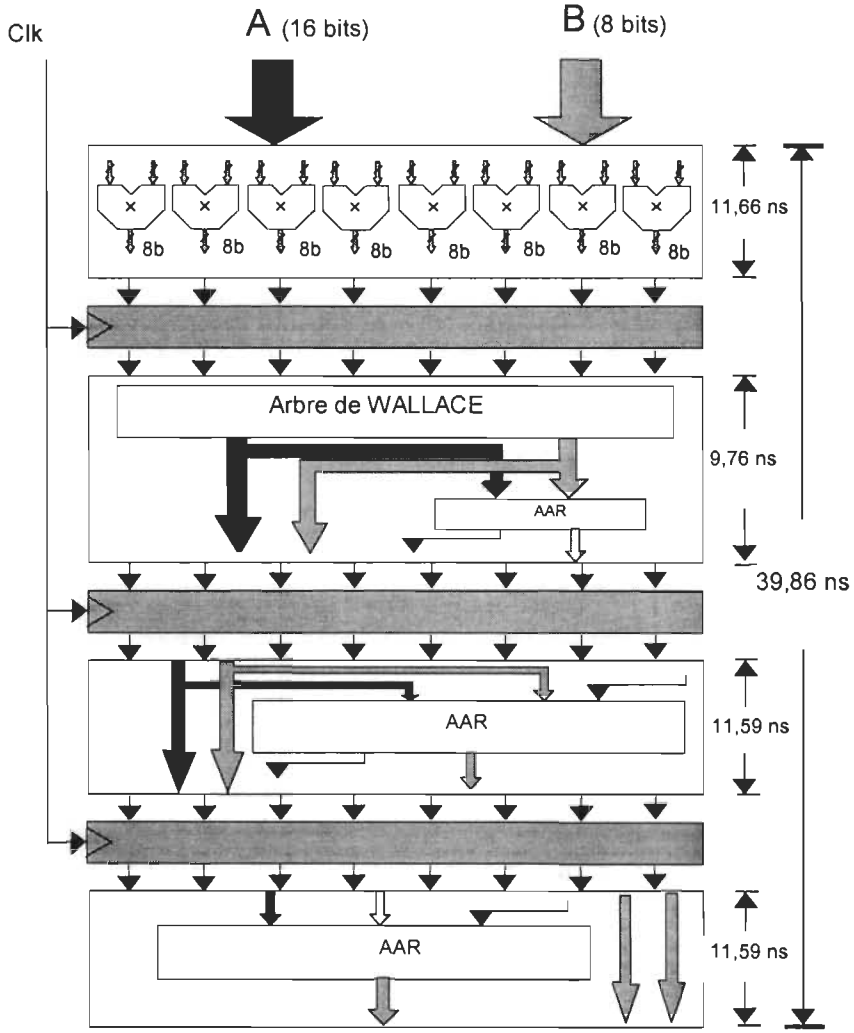


Figure 3.19 Architecture interne du multiplieur pipeliné

3.3.3.4 Résultats de simulations du multiplieur pipeliné

Les résultats de simulation du multiplieur ont été faits dans MENTOR GRAPHICS. Nous avons fait la multiplication de (½)sin(x) avec un (½)sin(2x) pour pouvoir la comparer avec les résultats trouvé avec MATLAB. Les résultats trouvés sont présentés à la figure 3.20, L'erreur est de l'ordre de 10^{-5} ce qui justifie donc le bon fonctionnement du multiplieur.

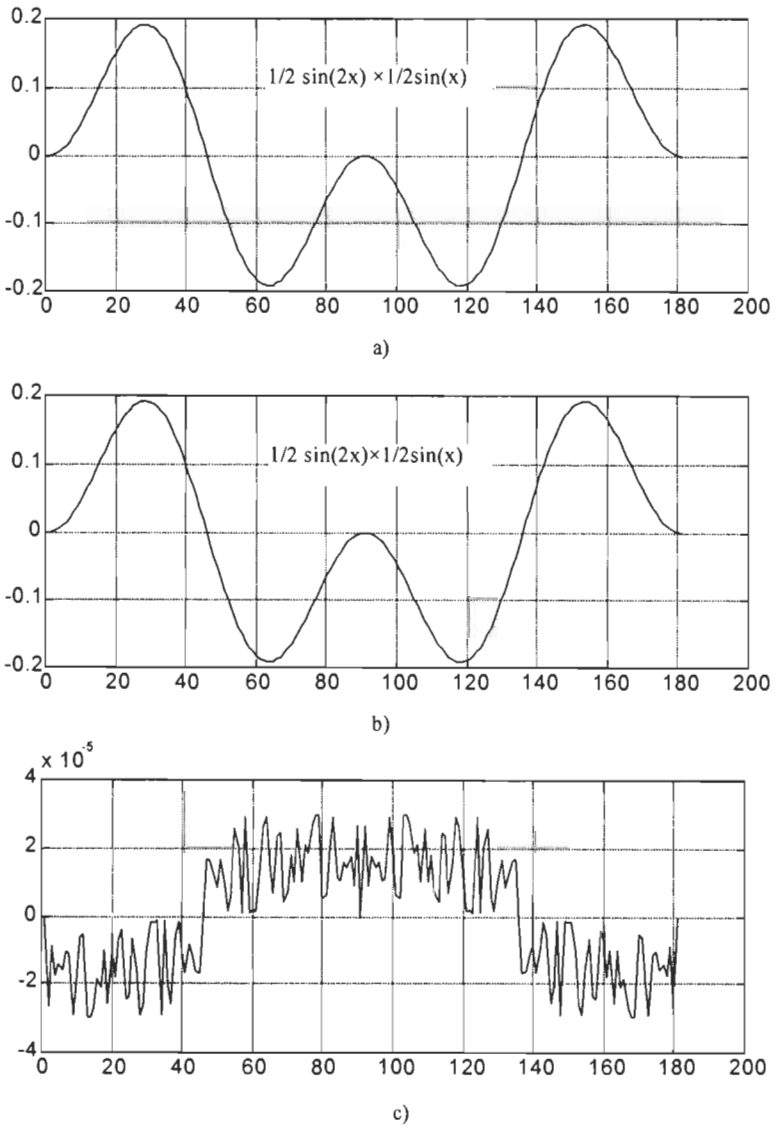


Figure 3.20 : Résultats de simulation du multiplieur pipeliné

Pour les séquences de fonctionnement de PIPEKAL$_2$, ils restent les mêmes que pour PIPEKAL$_2$ juste que pour la multiplication il faut tenir compte des quatres sous étapes qui constituent le pipeline interne, aussi des trois sous étapes de l'additionneur. La fréquence de fonctionnement sera donc la fréquence d'un étage pipeline. La figure 3.21 présente la structure globale de PIPEKAL$_2$.

Pour valider les résultats, nous avons passés à la modélisation de l'architecture en langage VHDL, la figure 3.22 montre le fichier principal du programme, les résultats de simulation de cette architecture sont montrés à la figure 3.23. La synthèse de cette architecture dans une technologie CMOS de 1.5 um de Mitel nous donne donc la structure finale de PIPEKAL$_2$ avec une fréquence de fonctionnement qui dépasse 88MHz.

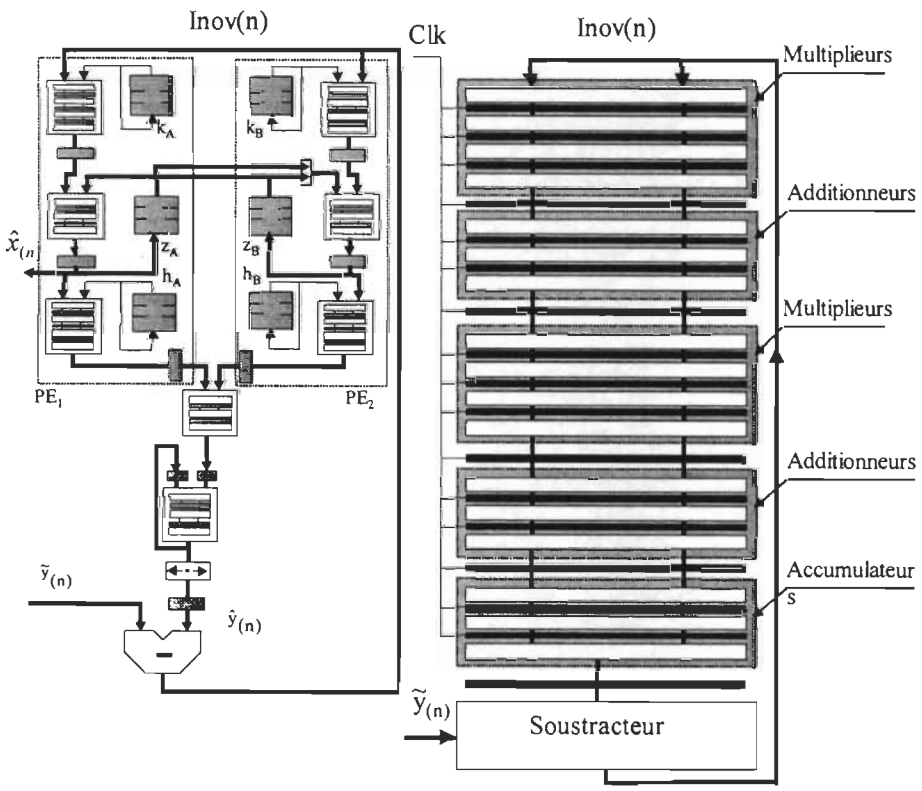
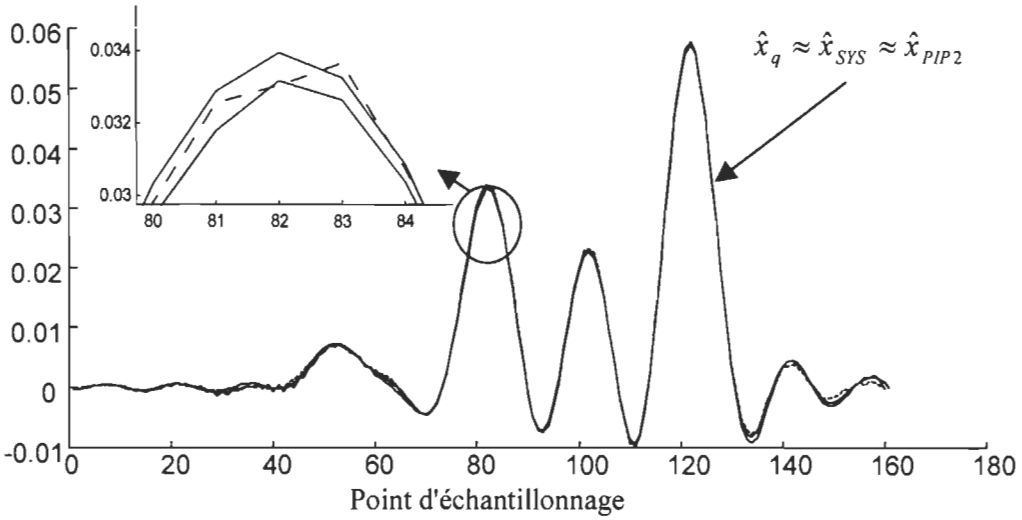


Figure 3.21 Architecture du processeur PIPEKAL$_2$

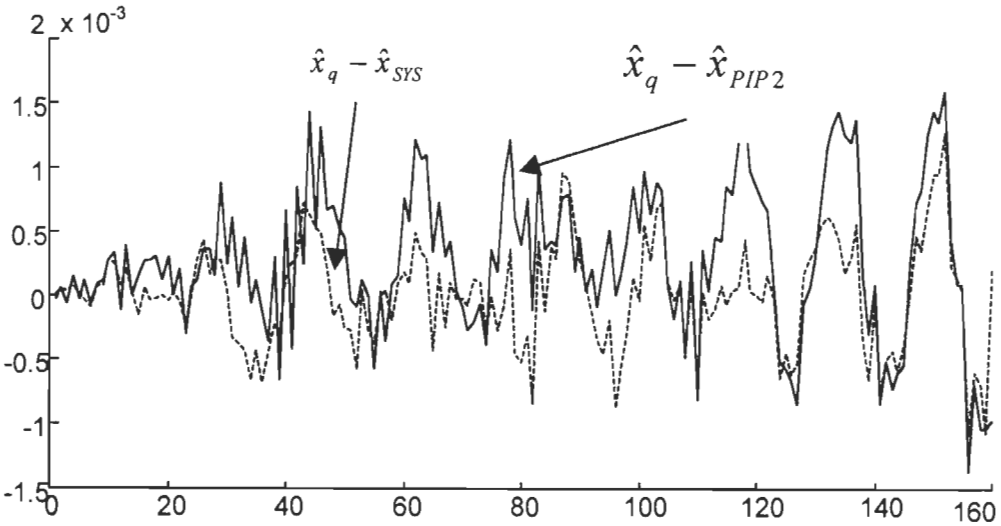
```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE work.pack_processeur_pipekal2.all;
   ENTITY processeur_PIPKAL2 IS
PORT( clk,clr_Z,chain, s, en_Z,en_K                    :IN   std_logic;
      clr_acc,en_acc,clr_s, en_x , en_s                :IN   std_logic;
      clr_cal, en_cal, clr_reg, clr_x,  en_h           :IN   std_logic;
      pil_ink1,pil_ink2, pil_inh1,pil_inh2             :IN   std_logic_vector(7 downto 0);
      y_mes, distanc                                   :IN   std_logic_vector(15 downto 0);
      xout                                             :out  std_logic_vector(15 downto 0));
   END processeur_PIPKAL2 ;
   ARCHITECTURE comportement OF processeur_PIPKAL2 IS
            signal mult1,mult2,add_out1,add_out2       :std_logic_vector(15 downto 0);
            signal mux_out, Z_out1,Z_out2              :std_logic_vector(15 downto 0);
            signal pil_outk1 ,pil_outk2                :std_logic_vector(7 downto 0);
            signal pil_outh1 ,pil_outh2                :std_logic_vector(7 downto 0);
            signal sortie1 ,sortie2                    :std_logic_vector(15 downto 0);
            signal yout1,yout2, yout, yacc_in, yacc_out :std_logic_vector(15 downto 0);
            signal y_cal_in, y_cal_out, innovation     :std_logic_vector(15 downto 0);
begin
            soustracteur  :soustract1                  port map (y_mes , y_cal_out , inovation);
            MULT_1        :multiplieur_pipeliner_16_8  port map ( inovation,pil_outk1,mult1 );
            MULT_2        :multiplieur_pipeliner_16_8  port map ( inovation,pil_outk2,mult2 );
            Reg_1         :Register_16                 port map(mult1, clk, mult11);
            Reg_2         :Register_16                 port map(mult2, clk, mult22);
            ADD_1         :adder_pipeliner             port map (mult1,mux_out, add_out1);
            ADD_2         :adder_pipeliner             port map (mult2,Z_out1 , add_out2);
            Reg_3         :reg16                       port map (add_out1,clk,en_s,clr_s,sortie1);
            Reg_4         :reg16                       port map (add_out2,clk,en_s,clr_s,sortie2);
            BLOC_Z1       :bloc_stock_donnees_33       port map (clk,en_Z,clr_Z,sortie1,Z_out1);
            BLOC_Z2       :bloc_stock_donnees_32       port map (clk,en_Z,clr_Z,sortie2,Z_out2);
            Multiplex     :mux2a1                      port map (Z_out1,Z_out2,s,mux_out);
            BLOC_K1       :PILE_34_8                   port map (clk, en_K, chain, pil_ink1, pil_outk1);
            BLOC_K2       :PILE_34_8                   port map (clk, en_K, chain, pil_ink2, pil_outk2);
            MULT_3        :multiplieur_pipeliner_16_8  port map ( sortie1,pil_outh1,yout1 );
            MULT_4        :multiplieur_pipeliner_16_8  port map ( sortie2,pil_outh2,yout2 );
            Reg_5         :reg16                       port map (yout1,clk,yout11);
            Reg_6         :reg16                       port map (yout2,clk,yout22);
            BLOC_h1       :PILE_34_8                   port map (clk, en_h, chain, pil_inh1, pil_outh1);
            BLOC_h2       :PILE_34_8                   port map (clk, en_h, chain, pil_inh2, pil_outh2);
            ADD_3         :adder_pipeliner             port map (yout11,yout22 , yout);
            Reg_7         :reg16                       port map (yout,clk,yout5);
            ADD_4         :adder_pipeliner             port map (yout5,yacc_in , yacc_out);
            Reg_8         :reg16                       port map (yacc_out,clk,yacc_out1);
            Reg3          :reg16                       port map (yacc_out1,clk,en_acc,clr_acc,yacc_in);
            Decaleur      :decaleur_gauch_droit        port map (distanc,yacc_in,y_cal_in);
            Reg4          :reg16                       port map (y_cal_in,clk,en_cal,clr_cal,y_cal_out);
            Reg6          :reg16                       port map (z_out2 ,clk,en_x,clr_x,xout);
   END comportement ;
```

Figure 3.22 Programme principal du processeur PIKALE$_2$

3.3.3.5 Résultats de simulation et description physique de PIPELKA$_2$



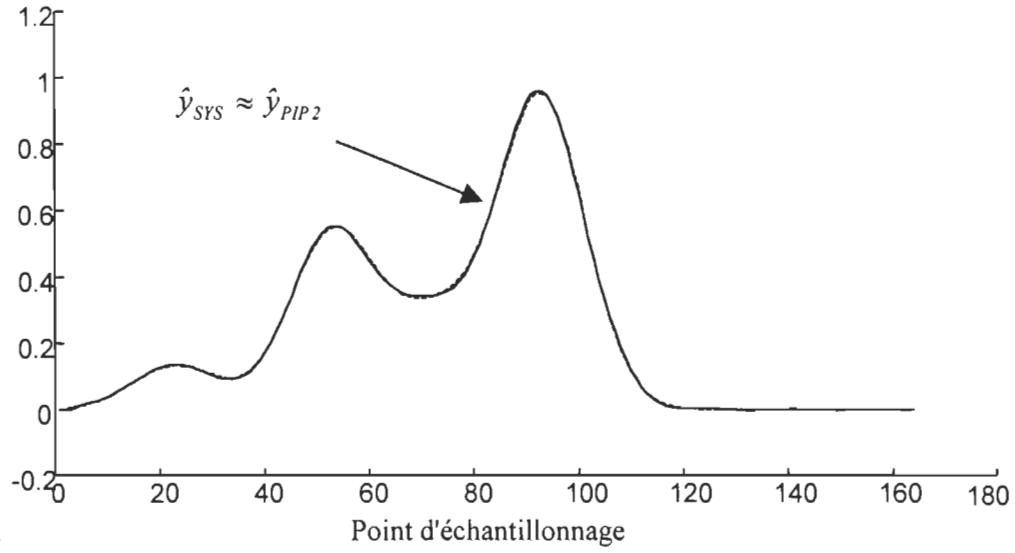
a)



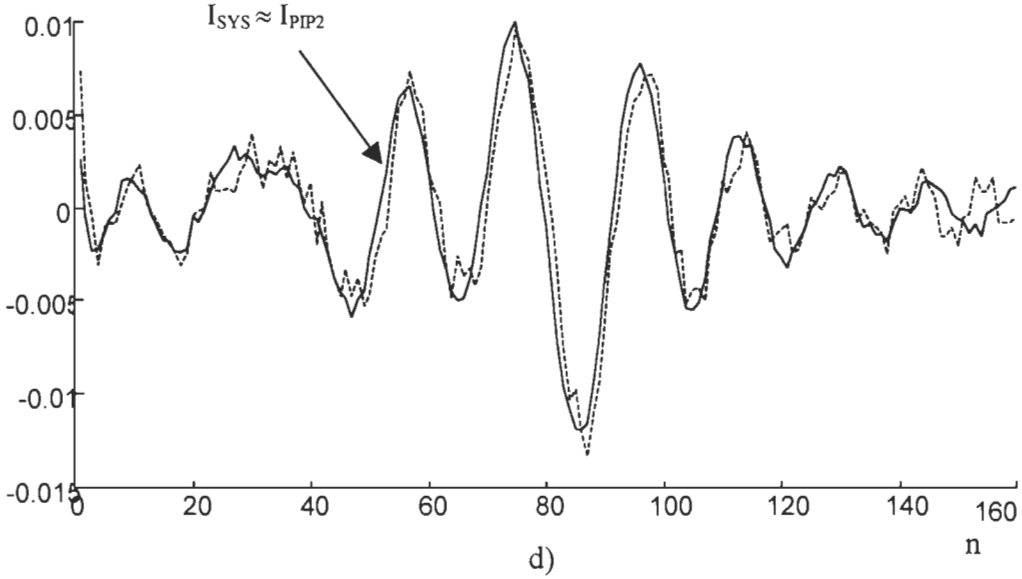
b)

Figure 3.23 : Résultats de simulation de PIPEKAL$_2$ a) signal reconstitué

b) erreur entre les signaux reconstitués



c)



d)

Figure 3.23 : Résultats de simulation de PIPEKAL$_2$ (suite) c) signal estimé d) innovation

3.3.3.6 Description physique

Concernant l'implantation physique de cette architecture, nous avons réalisé le routage de ces différents blocs à logique combinatoire du processeur dans une technologie CMOS de 1,5µm avec les outils de CADENCE. Ces blocs sont le multiplieur 16×8 bits, l'additionneur 16 bits, le soustracteur 16 bits, le décaleur 16 bits et les piles contenant les données Z, k_∞ et h voire (figure 3.24). La description physique de l'architecture PIPEKAL$_2$ montre une structure intéressante au niveau symétrie chose qui nous a encouragé à réaliser sa fabrication.

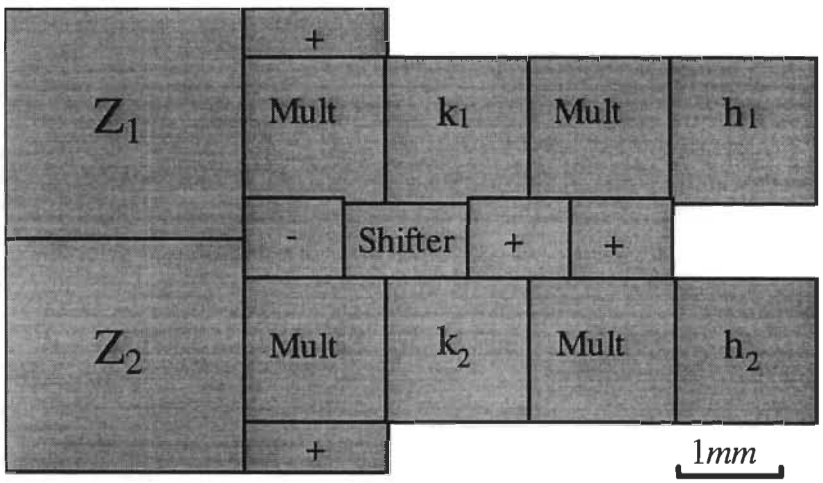


Figure 3.24 Description physique du processeur PIPEKAL$_2$.

3.3.4 Réduction du nombre de multiplieurs: PIPEKAL$_3$

Dans cette section nous allons proposer une nouvelle architecture qui présente les mêmes caractéristiques que PIPEKAL$_2$ sauf que cette dernière contient moins de BLC. En fait, cette architecture que nous avons appelé PIPEKAL$_3$ n'est que la version réduite de PIPEKAL$_2$. Pour diminuer la surface d'intégration, nous avons exploité la symétrie de

chaque processeur élémentaire en éliminant un multiplieur. Ce qui a pour effet de diminuer d'avantage la surface d'intégration sans effet marqué sur la performance.

La figure 3.25 présente la structure interne de PIPEKAL$_3$. L'apparition des multiplexeurs à l'entrée des multiplieurs assure le synchronisme des données à travers toute l'architectures, le même principe a été utilisé dans [THO.96] et [CHE.95]. PIPEKAL$_3$ fonctionne à chaque front montant de l'horloge. À la différence des autres architectures précédentes qui fonctionnaient à l'état de l'horloge.

Nous avons modélisé cette architecture en langage VHDL pour pouvoir vérifier sa fonctionnalité et aussi pour chercher ces caractéristiques de point de vue temps de calcul et surface d'intégration. La figure 3.26 et 3.27 présentent respectivement le modèle VHDL de cette architecture et les résultats de simulation effectués dans Mentor Graphics.

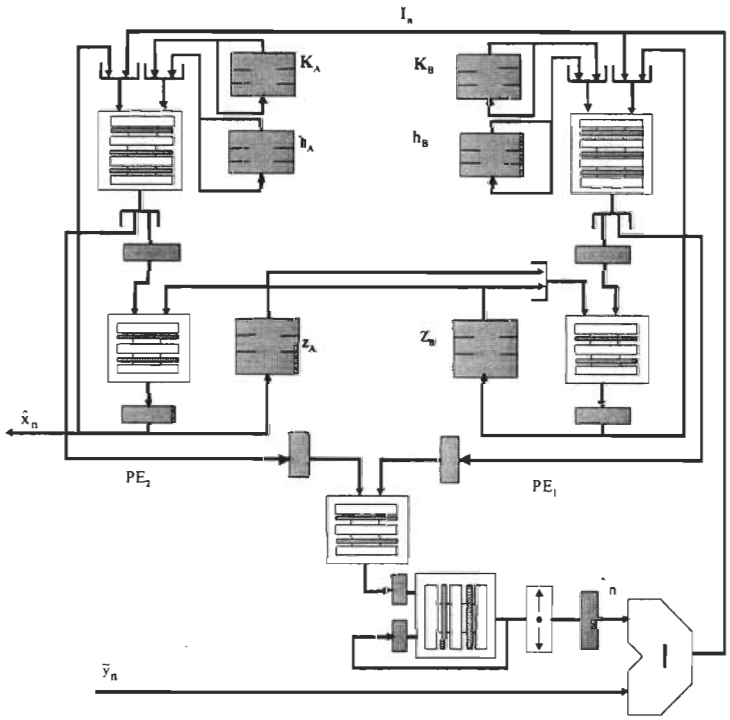
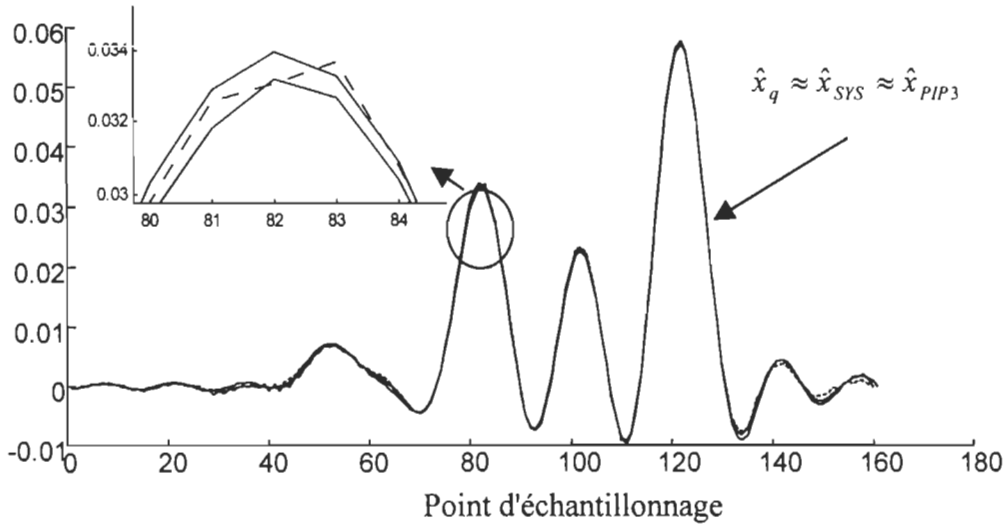


Figure 3.25 Structure globale de l'architecture PIPEKAL$_3$

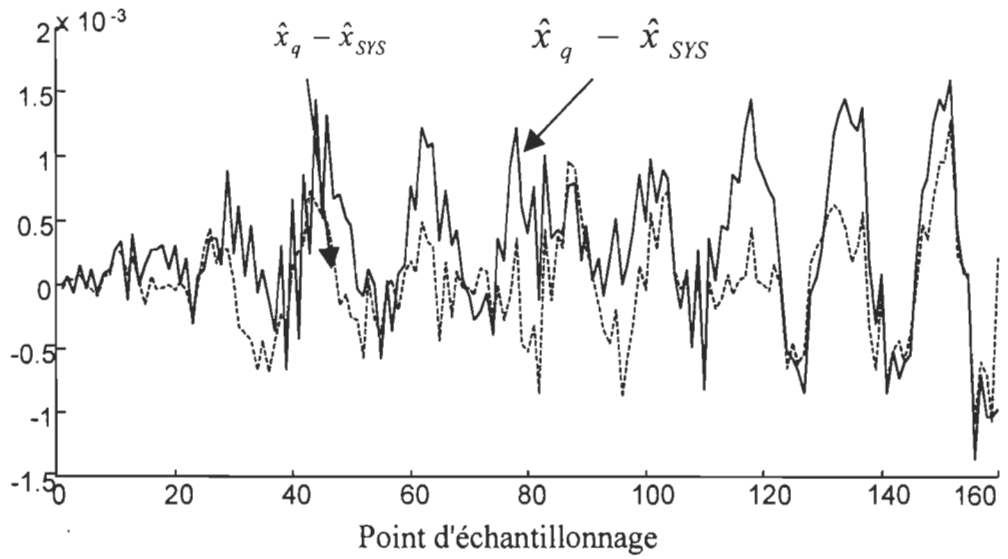
```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE work.pack_processeur_pipekal3.all;
   ENTITY processeur_PIPKAL3 IS
PORT( clk,clr_Z ,chain, s, en_Z,en_K, select1        :IN   std_logic;
      clr_acc,en_acc,clr_s, en_x , en_s              :IN   std_logic;
      clr_cal, en_cal, clr_reg, clr_x,  en_h         :IN   std_logic;
      pil_ink1,pil_ink2, pil_inh1,pil_inh2           :IN   std_logic_vector(7  downto 0);
      y_mes, distanc                                 :IN   std_logic_vector(15 downto 0);
      xout                                           :out  std_logic_vector(15 downto 0));
   END processeur_PIPKAL3 ;
   ARCHITECTURE comportement OF processeur_PIPKAL3 IS
            signal mult1,mult2,add_out1,add_out2      :std_logic_vector(15 downto 0);
            signal mux_out, Z_out1,Z_out2             :std_logic_vector(15 downto 0);
            signal pil_outk1 ,pil_outk2               :std_logic_vector(7 downto 0);
            signal pil_outh1 ,pil_outh2               :std_logic_vector(7 downto 0);
            signal sortie1 ,sortie2,sorin1, sorin2, kh1,kh2 :std_logic_vector(15 downto 0);
            signal yout1,yout2, yout, yacc_in, yacc_out :std_logic_vector(15 downto 0);
            signal y_cal_in, y_cal_out, innovation    :std_logic_vector(15 downto 0);
            signal mult1yout1, mult1yout2             :std_logic_vector(15 downto 0);
begin
            soustracteur  :soustract1                 port map (y_mes , y_cal_out , inovation);
            Mux1          : multiplexeur_16           port map(inovatio, soti1,select1, sorin1);
            Mux2          : multiplexeur_16           port map(inovatio, soti2,select1, sorin2);
            Mux3          : multiplexeur_8            port map(pil_outk1,pil_outh1,select1, kh1);
            Mux4          : multiplexeur_8            port map(pil_outk2,pil_outh2,select1, kh2);
            MULT_1        :multiplieur_pipeliner_16_8 port map ( sorin1,kh1,mult1yout1 );
            MULT_2        :multiplieur_pipeliner_16_8 port map ( sorin2,kh2,mult2yout2 );
            Demux1        :demultiplexeur_16          port map (mult1yout1, select1, mult1,yout1);
            Demux2        :demultiplexeur_16          port map (mult2yout2, select1, mult2,yout2);
            Reg_1         :Register_16                port map(mult1, clk, mult11);
            Reg_2         :Register_16                port map(mult2, clk, mult22);
            ADD_1         :adder_pipeliner            port map (mult1,mux_out, add_out1);
            ADD_2         :adder_pipeliner            port map (mult2,Z_out1 , add_out2);
            Reg_3         :reg16                      port map (add_out1,clk,en_s,clr_s,sortie1);
            Reg_4         :reg16                      port map (add_out2,clk,en_s,clr_s,sortie2);
            BLOC_Z1       :bloc_stock_donnees_33      port map (clk,en_Z,clr_Z,sortie1,Z_out1);
            BLOC_Z2       :bloc_stock_donnees_32      port map (clk,en_Z,clr_Z,sortie2,Z_out2);
            Multiplex     :mux2a1                     port map (Z_out1,Z_out2,s,mux_out);
            BLOC_K1       :PILE_34_8                  port map (clk, en_K, chain, pil_ink1, pil_outk1);
            BLOC_K2       :PILE_34_8                  port map (clk, en_K, chain, pil_ink2, pil_outk2);
            MULT_3        :multiplieur_pipeliner_16_8 port map ( sortie1,pil_outh1,yout1 );
            MULT_4        :multiplieur_pipeliner_16_8 port map ( sortie2,pil_outh2,yout2 );
            Reg_5         :reg16                      port map (yout1,clk,yout11);
            Reg_6         :reg16                      port map (yout2,clk,yout22);
            BLOC_h1       :PILE_34_8                  port map (clk, en_h, chain, pil_inh1, pil_outh1);
            BLOC_h2       :PILE_34_8                  port map (clk, en_h, chain, pil_inh2, pil_outh2);
            ADD_3         :adder_pipeliner            port map (yout11,yout22 , yout);
            Reg_7         :reg16                      port map (yout,clk,yout5);
            ADD_4         :adder_pipeliner            port map (yout5,yacc_in , yacc_out);
            Reg_8         :reg16                      port map (yacc_out,clk,yacc_out1);
            Reg3          :reg16                      port map (yacc_out1,clk,en_acc,clr_acc,yacc_in);
            Decaleur      :decaleur_gauch_droit       port map (distanc,yacc_in,y_cal_in);
            Reg4          :reg16                      port map (y_cal_in,clk,en_cal,clr_cal,y_cal_out);
            Reg6          :reg16                      port map (z_out2 ,clk,en_x,clr_x,xout);
   END comportement ;
```

Figure 3.26 Programme principal du processeur PIPEKAL$_3$



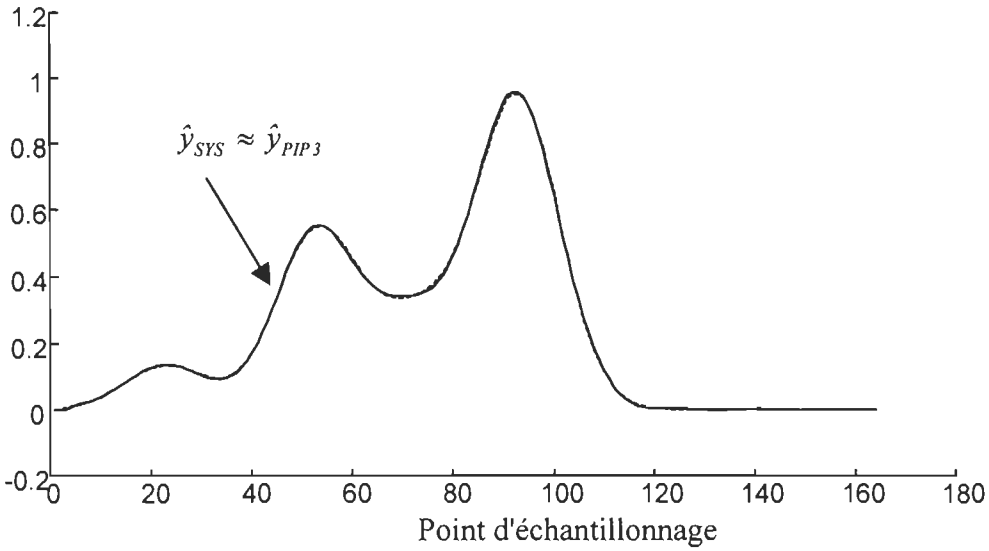
$\hat{x}_{PIP3}$: Signal reconstitué par PIPEKAL$_3$

a)

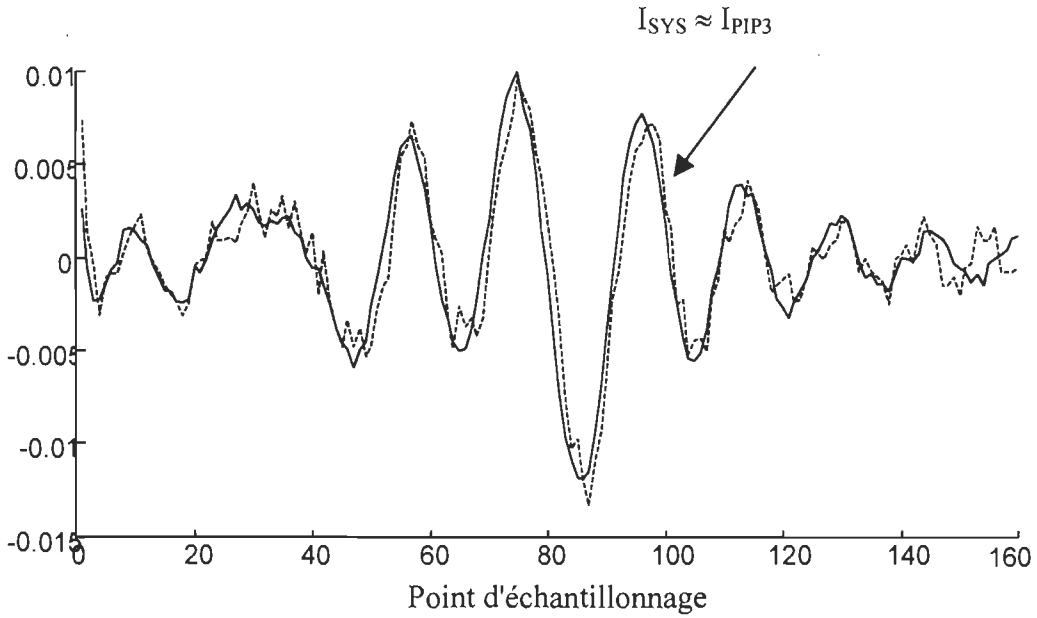


b)

Figure 3.27 : Résultats de simulation de PIPEKAL$_3$ a) résultat de reconstitution
b) erreur de reconstitution



c)



d)

Figure 3.27 : Résultats de simulation de $PIPEKAL_3$ (suite) c) signal estimé calculé par SYSKAL et $PIPEKAL_3$ d) Innovation calculé par SYSKAL et $PIPEKAL_3$

3.4 Conclusion

Dans ce chapitre, nous avons appliqué la première technique du pipeline qui est le pipeline conventionnel. Cela nous a permis d'atteindre des fréquences moyennement élevés ainsi que des surfaces relativement raisonnable pour une intégration à très grande échelle. Aussi nous avons proposé trois architectures présentent une différence au niveau du degré de pipeline et de surface occupée. Dans l'architecture $PIPEKAL_1$ le degré du pipeline est peu profond et ne touche pas au bloc à logique combinatoire. Ceci nous a permis d'atteindre une fréquence de 25 MHz dans une technologie CMOS 1,5µm de Mitel, soit 3 fois celle de SYSKAL avec une surface presque identique. Dans l'architecture $PIPEKAL_2$, nous avons approfondi davantage le pipeline pour arriver à des fréquences au voisinage de 80 MHz sans ajout substantiel de la surface. Finalement nous avons présenté l'architecture $PIPEKAL_3$ qui utilise le même degré de pipeline que $PIPEKAL_2$ mais avec deux fois moins de multiplieurs.

4

Pipeline par vagues

Dans le chapitre précédent nous avons appliqué la méthode du pipeline conventionnel pour augmenter la fréquence de fonctionnement de l'architecture SYSKAL. Cela nous a conduit à une fréquence de 80 MHz ce qui est acceptable lors des traitements à des vitesses moyennes, mais qui reste toujours faible pour qu'elle soit applicable à des cas haute vitesse tel que rencontré en télécommunication. Pour cela, nous avons trois possibilités: approfondir le pipeline déjà utilisé en augmentant le nombre d'étages, appliquer une nouvelle méthode appelée pipeline par vagues, ou changer de technologie.

Théoriquement on peut penser à appliquer la première méthode qui consiste à répartir le problème en plusieurs étages travaillant de façon continue et permettant par la suite d'augmenter le flot de données à travers le l'architecture. Cette façon de partager le circuit en plusieurs sous blocs peut théoriquement continuer jusqu'à des niveaux très élevés pour des délais très faibles. Mais malheureusement, il y a deux facteurs qui s'opposent à

l'applicabilité de cette approche. Le premier facteur est dû à l'effet des registres, plus on augmente le nombre de blocs (on diminue par la suite le délai le long du bloc) plus il faut des registres de synchronisation. Par conséquent, ces derniers augmentent à leur tour les délais de propagation dans le système d'une part, et d'autre part leurs paramètres de temporels du préparation et du maintien (setup, hold) ne diminuent pas en fonction du nombre de registres, ceci à pour effet de limiter la période de l'horloge. Le second facteur est l'effet de dispersion de l'horloge. Plus on augmente le nombre de registres plus le réseau de distribution d'horloge augmente ce qui, par la suit, augmentera le nombre d'amplificateur dans la distribution. Par conséquent, ceci entraîne une dispersion d'horloge, une augmentation de la surface et une consommation plus élevée de puissance.

On peut remarquer aussi que cette méthode se base aussi sur le délai de propagation des données le long des sous blocs, chose qui nuit à la rapidité du système. Une autre approche consiste à tenir seulement de l'écartement entre le délai maximal et minimal. C'est la méthode du *pipeline par vagues* ou aussi *pipeline asynchrone* [DEM.96] [COT.69]. Dans ce qui suit nous allons introduire quelques notions de base du pipeline par vagues pour démontrer son efficacité lorsqu'elle est applicable à l'architecture SYSKAL. Cette introduction donnera les notions qui seront utiles pour que le lecteur puisse comprendre la suite du projet. Pour ceux qui veulent plus d'informations nous suggérons de consulter les références suivantes [EKR.87], [GRA.92 a], [GRA.94] et [WON.93].

4.1 La technique du pipeline par vagues

4.1.1 Historique

L'idée du pipeline par vagues ou le pipeline à débit maximal a été formalisée pour la première fois par Cotton en 1969. Bien qu'elle était utilisée aussi dans quelques produits de IBM, système 1360 modèle 91 à virgule flottante, récemment plusieurs recherches se sont intéressées à cette approche puisque les limites en technologie et en fréquence sont presque atteintes ou très coûteuses à réaliser. Actuellement, le travail dans ce domaine de pipeline par vagues a touché plusieurs parties du problème que ce soit au niveau théorique, au niveau technique concernant l'équilibre de propagation des données le long des chemins ou au niveau conception utilisant cette technique.

Parmi les travaux les plus récents dans ce domaine, citons ceux de Ekroot [EKR.87] et Wong [WON.91]. Ekroot a développé toute une étude théorique sur les contraintes permettant le bon fonctionnement de la méthode, Wong a proposé plusieurs méthodes pour équilibrer les délais le long des circuits à logiques combinatoires, ses travaux se sont basés sur l'insertion des ports logiques constituant des délais qui permettent le ralentissement des chemins les plus rapides. Une équipe de recherche constituée de T. Gray, W. Lieu et Cavin ont développé une méthode de placement des délais, de l'optimisation de l'horloge et aussi quelques théories sur les contraintes de temps.

La théorie du pipeline par vagues se base sur trois axes principaux: les bases théoriques, les outils d'aide à la conception CAO (conception assistée par ordinateur) et les

techniques particulières de design. Dans l'étude théorique, le travail se concentrera sur: la recherche des liens entre la fréquence limite et les différents paramètres du temps; les méthodes permettant d'équilibrer la propagation des données dans chaque étage; l'étude de la possibilité d'unir les deux types de pipeline (conventionnel et par vagues); et finalement l'étude des cas particuliers où il y a existence de boucles (feed-back).

Dans le cas pratique la validation de la théorie du pipeline par vagues ne peut se faire que par la recherche des délais le long des circuits. Dans l'application de cette technique, cela exige la connaissance pratique des outils de simulation permettant de donner toutes les caractéristiques du temps du circuit pipeliné à la différence du pipeline conventionnel où la recherche des contraintes de temps consiste à trouver tout simplement le délai maximal du chemin le plus long qui nous permettra de déduire la fréquence limite du circuit.

Le pipeline par vagues est une technique très sensible au changement des délais, on peut le remarquer par exemple lors de l'utilisation de la technologie MOS. Dans cette technologie le délai de propagation dépend du type de transistor activé (n-MOS ou p-MOS). Puisqu'il y a un facteur de deux au niveau de la résistance dynamique entre les transistor de type N et P, ce rapport se répercute au niveau délai de propagation tant au front montant qu'au front descendent des portes logiques. La connaissance des techniques modernes permettant le choix des architectures donnant une certaine homogénéité des délais de propagation est un facteur essentiel dans l'application du pipeline par vagues.

4.1.2 Principe de base

Le pipeline par vagues est une méthode de temporisation qui peut être utilisée pour pipeliner les circuits de logique combinatoire sans l'utilisation de registres de séparations des blocs tels qu'utilisées dans le pipeline conventionnel comme il est montré à figure 4.1. Cette technique se base donc sur un réarrangement des portes logiques dans les différents chemins logiques de ces derniers. Plusieurs délais sont introduits pour assurer des délais de propagation identiques le long des différents chemins logiques. Ce qui permettra par la suite la possibilité de propager des données sans se soucier du chevauchement des données dans le circuit logique.

La figure 4.2 montre le principe général de cette approche en comparaison avec celui du pipeline conventionnel.

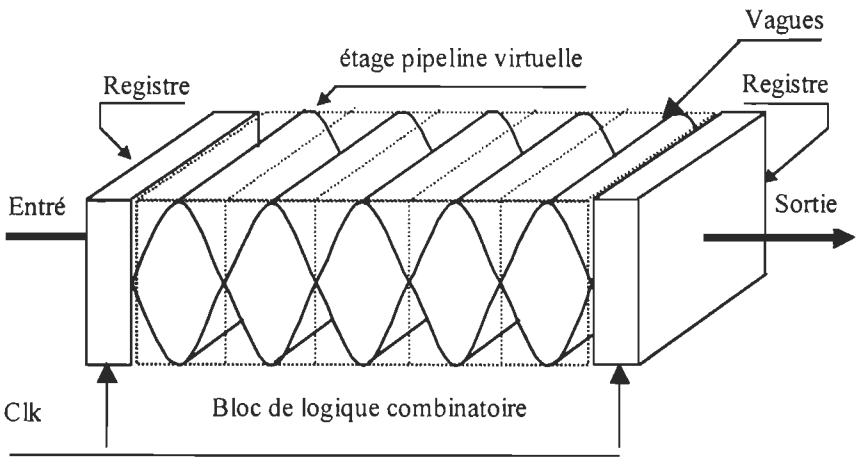


Figure 4.1: Bloc utilisant pipeline par vagues

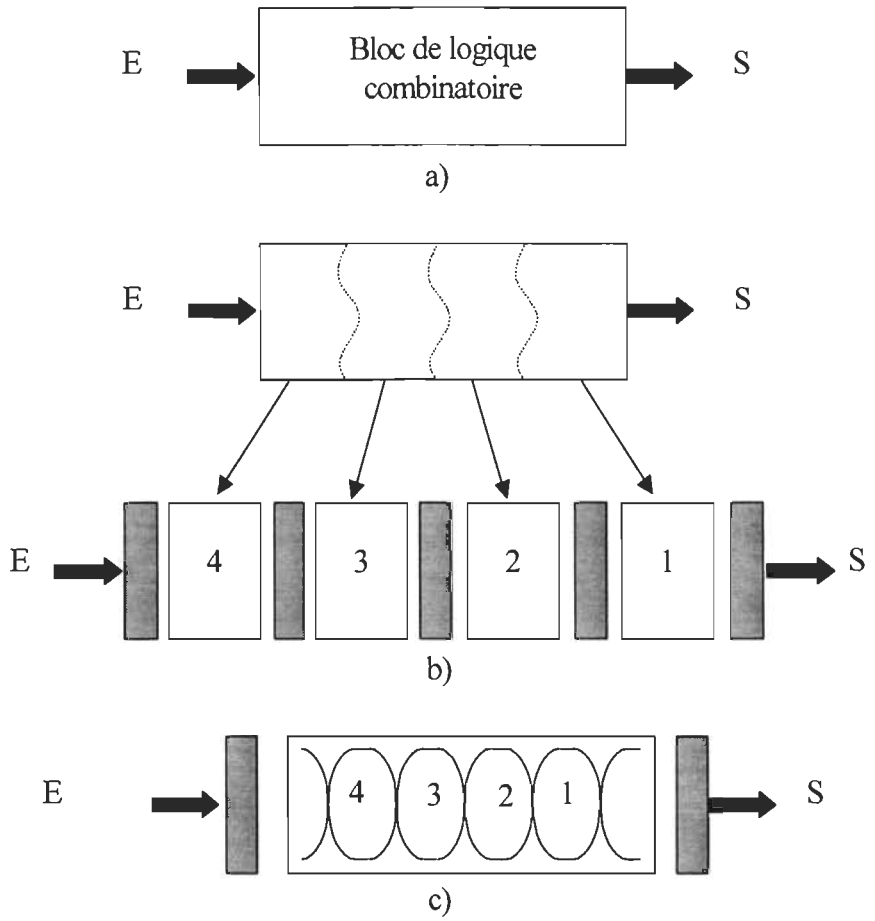


Figure 4.2 Principe du pipeline par vagues en comparaison avec le pipeline conventionnel

a) Sans pipeline b) Avec pipeline conventionnel c) Avec pipeline par vagues

D'après la figure 3.4, on remarque que la fréquence d'horloge ne dépend pas des délais de propagation dans un circuit logique mais elle dépend essentiellement de la différence entre le délai maximal et le délai minimal le long du circuit. L'idée consiste donc à réduire cette différence pour pouvoir propager les données telles que montré à la figure 4.3. Ainsi le délai entre deux tâches successive est théoriquement réduit à cette différence.

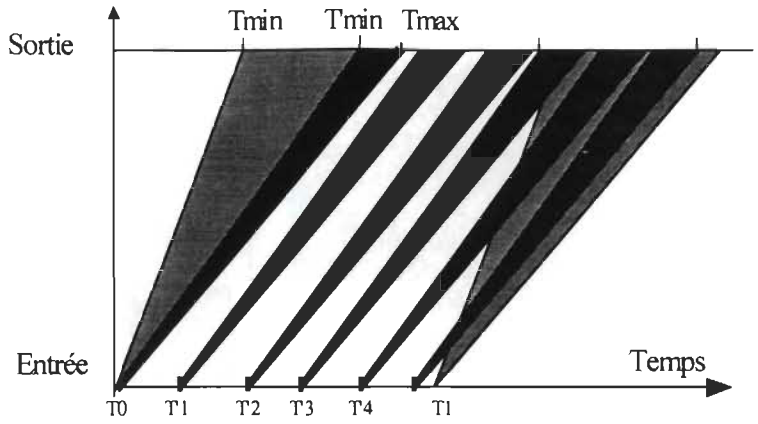


Figure 4.3 Flot de données dans le cas du pipeline par vagues

Il est bien évident qu'une temporisation correcte du système serait nécessaire pour coordonner l'horloge et les données à l'entrée de chaque registre et aussi une coordination des différentes données à travers le circuit combinatoire.

4.1.3 Contraintes et paramètres

La section suivante décrit donc en bref les contraintes et les paramètres interagissant dans la temporisation du circuit, et aussi la méthode idéale pour chercher les limites maximales du débit des opérations. Une étude détaillée sur ce sujet a été développé dans [GRA.92 a], [GRA.92 b] et [THO.92].

Le bloc à logique combinatoire (BLC) qui sera utilisé comme exemple est celui de la figure 4.4 et dont le flot de donnée est présenté à la figure 4.3. On suppose que toutes les entrées et sorties du circuit sont synchronisées par les registres d'entrée et sortie fonctionnant sur la même horloge. Tous les nœuds du BLC sont numérotés afin de les identifier dans la suite de l'exemple. La figure 4.4, montre que plusieurs chemins logiques

sont possibles selon la donnée d'entrée qui se propage dans le circuit. Par conséquent ces chemins donnent naissance à différents délais.

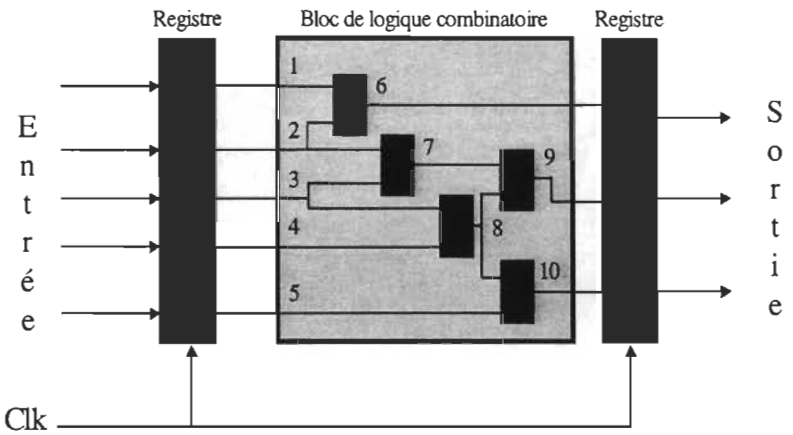


Figure 4.4 : Exemple de la structure interne d'un BLC

En général, ces chemins peuvent être classé en quatre ensembles:

N: ensemble des chemins liant une sortie à une entrée successive;

C: ensemble des connections possibles entre l'entrée et la sortie des différentes portes logiques;

G: ensemble des sorties des portes logiques.

O: ensemble des sorties du bloc combinatoire;

Dans l'exemple de la figure 4.4, nous obtenons les ensembles suivants :

N = [1,2,3,4,5,6,7,8,9,10];

C = [(1,6),(2,6),(2,7),(3,7),(3,8),(4,8),(5,10),(8,10),(8,9),(7,9)];

G = [6,7,8,9,10];

O = [6,9,10].

Un ensemble de délais sera nécessaire dans cette étude. On peut donc diviser les paramètres du temps en trois catégories: les paramètres du temps du circuit, les paramètres du temps des registres et les paramètres du temps de l'horloge. Tous ces paramètres sont présenté à la figure 4.5.

Les paramètres du temps du circuit sont:

t_{max}: délai de propagation maximal du BLC;

t_{min}: délai de propagation minimal du BLC;

$t_{max}(i)$: délai maximal entre la sortie du registre d'entrée et la sortie de la porte logique (i);

$t_{min}(i)$: délai minimal entre la sortie du registre d'entrée et la sortie de la porte logique (i);

$t_{stable}(i)$: le temps minimal pendant lequel un signal doit rester à l'entrée pour avoir un résultat stable à la sortie de la porte logique (i).

Les paramètres d'horloge sont:

tclk: période d'horloge;

Δ_i: délai ajouté à l'entrée d'horloge dû à l'erreur de synchronisation (*skew*);

Δ_o: délai ajouté à la sortie d'horloge;

Δ : $\Delta_o - \Delta_i$;

Δt_e^n: le temps d'avance de l'horloge à l'entrée;

Δt_l^n : le temps du retard de l'horloge à l'entrée;

Δt_e^o : le temps d'avance de l'horloge à la sortie;

Δt_l^o : le temps de retard de l'horloge à la sortie.

Les paramètres du temps des registres sont:

t_d : délai dans un registre ;

t_{setup} :le temps de stabilisation des données à l'entrée du registre avant l'arrivée du signal d'horloge ;

t_{hold} : le temps de stabilisation des données à l'entrée des registres après l'arrivée de l'horloge.

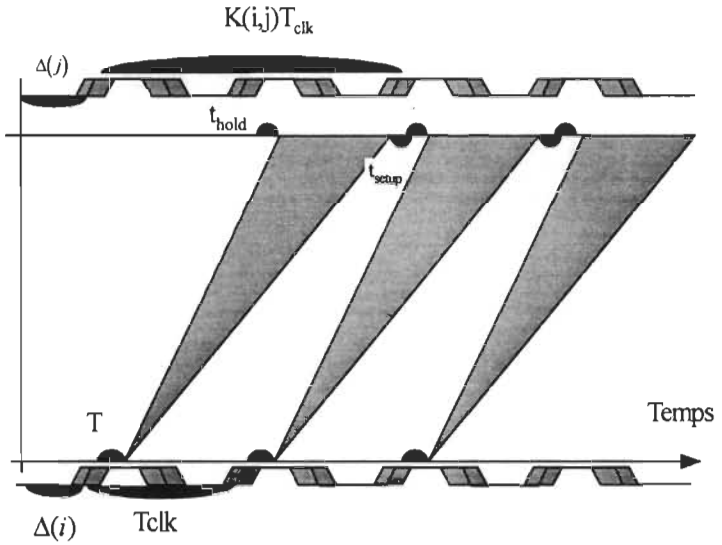


Figure 4.5 : Représentation des contraintes du temporelles

Un ensemble de contraintes tant sur les registres que sur les noeuds est proposé par [THO.92] et sera utilisé comme référence pour la détermination de la fréquence limite du circuit. Une relation est donc proposée qui est de la forme:

$$T_{clk} \geq (t_{\max} - t_{\min}) + t_{setup} + t_{hold} + \Delta t_e^n + \Delta t_e^o + \Delta t_l^n + \Delta t_l^o \quad (4.1)$$

D'après cette relation, il est bien clair que la fréquence d'horloge est inversement proportionnelle à la différence entre $t_{\max}$ et $t_{\min}$, si on suppose que les autres paramètres sont constants pour une technologie donnée et à une fréquence donnée. Plusieurs travaux convergent donc vers la méthode qui consiste à réduire au maximum cette différence entre $t_{\max}$ et $t_{\min}$ pour chercher T_{clk} limite.

4.2 Application du pipeline par vagues à l'architecture SYSKAL

4.2.1 Blocs de logique combinatoire pipelinés

Dans la section précédente on remarque bien que pour atteindre des performances maximales, la propagation de délais à travers les différents chemins du circuit de logique combinatoire doit être équilibrée. L'équilibrage du délai à travers les chemins peut être atteint par deux méthodes [SJU.92] [KLA.92] [EKR.87]: la première méthode appelée "rough tuning" consiste à insérer des délais le long du chemin le plus court de façon à augmenter son délai de propagation total; la deuxième méthode appelée "fine tuning" ou réglage fin, consiste à ajuster les délais de propagation de toutes les portes logiques de

façon à équilibrer les délais de propagation le long de chaque étage. Les deux méthodes peuvent être combinées ensemble pour atteindre des résultats optimaux. Notons aussi que ces techniques ont été utilisées dans plusieurs architectures et implantées dans différentes technologies telles que ECL, CMOS etc [KLA.90].

Les contraintes du temps pour le pipeline par vagues sont différentes de celles du pipeline conventionnel [THO.92]. La période minimale de l'horloge sur laquelle le pipeline par vagues peut fonctionner est limitée par la différence maximale entre le chemin le plus long et le chemin le plus court. D'autres facteurs limitent la fréquence de l'horloge l'erreur de recouvrement de transition des données et de l'horloge d'un registre (setup/hold) le flot de données dans le circuit est aussi limité par le délai des portes (RC),et la variation de température. Sachant que la différence des délais entre les différents chemins peut tendre vers une valeur très faible dans le cas où les chemins sont parfaitement équilibrés, dans ce cas on peut dire que les performances du pipeline par vagues s'approchent des limites physiques déterminées par la technologie utilisée. Le lecteur intéressé peut consulter la référence [THO.92] pour une étude plus détaillée sur les limites de fonctionnement du pipeline par vagues.

Dans ce travail nous avons fixé notre technologie qui sera utilisée, soit la technologie CMOS 1.5 μm de Mitel. La raison de ce choix technologique est la disponibilité des librairies via la SCM, la possibilité de fabrication via SCM, la possibilité de commercialisation et le coût.

La nature imprévisible des paramètres dans la technologie CMOS [KLA.90] peut apparaître comme un sérieux obstacle concernant l'implantation du pipeline par vagues.

Comme il a été expliqué, l'un des plus importants facteurs de la technique du pipeline par vagues est la minimisation de la différence entre le délai maximal et minimal. Généralement les sources de variation peuvent se résumer comme suit [THO.92]:

- la différence dans la longueur des chemins
- l'effet de la puissance dissipée,
- l'effet des capacités parasites,
- le procédé de fabrication,
- la variation de température.
- les différents types de transistors

Si pour le premier facteur, l'introduction de délais le long du chemin le plus court peut être une bonne solution, le reste des facteurs créent des problèmes peut contrôlables et auxquels il faut donner beaucoup d'importance lors de la conception. Cependant, le facteur le plus important concerne le type de transistors. La figure 4.6 montre par exemple l'effet de l'état des entrées sur la variation du délai de propagation des signaux d'entrées à la sortie de différentes portes.

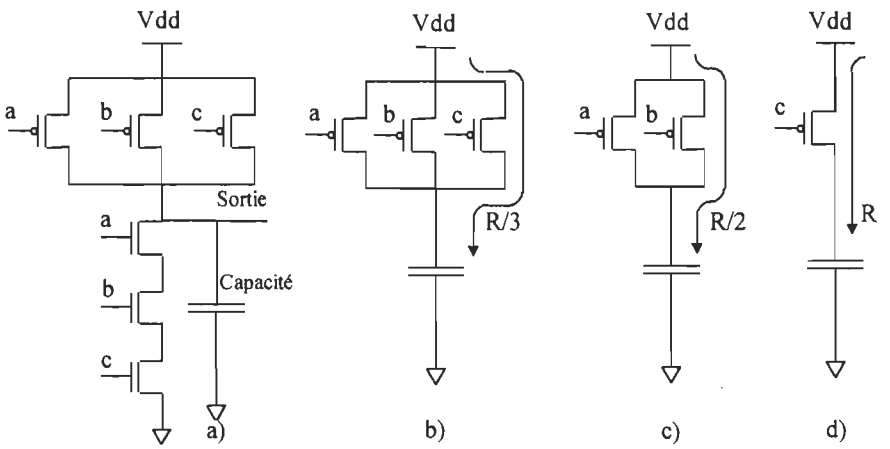


Figure 4.6 Variation de la résistance équivalente suivant les états des bits d'entrées

Dans cet exemple on remarque bien que la résistance équivalente pour la propagation d'un front montant peut varier suivant l'état du bit d'entrée. Dans (b) toutes les entrées sont à l'état `1` et donc la résistance équivalente est de R/3, dans (c) une des entrées est à l'état `0`, et la résistance équivalente est de R/2. Et finalement dans (d) deux des entrées sont à l'état `0` et une résistance équivalente est de R. Les délais de propagation le long de cette porte vont varier respectivement entre RC/3, RC/2 et RC suivant les états des bits d'entrée. Une méthode pour remédier à ce type de problèmes est proposée dans [KLA.94], [LIU.94]et [NGU.93]. Dans ces travaux le pipeline par vagues a été modélisé avec une cellule de base constituée d'une porte NAND et d'un inverseur pour réaliser toutes les fonctions logiques utilisées pour construire un multiplieur de 16×16 bits. L'approche utilisée pour équilibrer les délais est la suivante: un équilibrage de délai le long des cellules de base a été effectué pour passer par la suite à un équilibrage de l'architecture globale réalisée par cellules de base déjà équilibrées.

La complexité de cette méthode, vu qu'elle utilise juste deux fonctions de base, a rendu la tâche trop difficile. Cela a conduit par la suite à la recherche d'autres architectures qui peuvent réaliser plusieurs fonctions de base avec des délais de propagation constantes. Ainsi dans [GHO.95] une nouvelle structure d'une cellule de base former par quatre transistors de types pMOS a été présentée. Les caractéristiques remarquables de cette cellule est sa capacité de réaliser la majorité des fonctions de base avec des délais de propagation constant. La figure 4.7 montre cette cellule de base. Toutes les fonctions de bases de cette cellule sont présentées au tableau 4.1

Tableau 4.1 : Les différentes fonctions réalisées par la cellule de base

Fonction	A_i	A_j	B_i	Q
ET	A	B	B	AB
OU	A	B	Not(B)	A+B
OUEXCLUSIF	A	Not(A)	Not(B)	A⊕ B
RETENU	C	B	A⊕ B	Retenu
SOMME	C	Not(C)	A⊕ B	A⊕ B⊕ C
DELAI	B(t)	B(t)	B(t)	B(t+1)

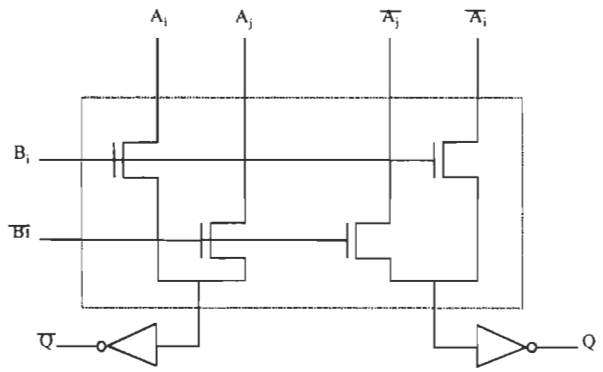


Figure 4.7: Structure interne de la cellule de base

Comme il est présenté au tableau 4.1, cette cellule peut réaliser différentes fonctions suivant la façon dont elle a été configurée par ses signaux d'entrées. Elle peut aussi être utiliser comme délai pour équilibrer les différents chemins du circuit. Dans le travail qui suit nous avons utilisé cette cellule comme unité de base pour réaliser notre propre bloc de logique combinatoire. Les justifications de l'utilisation de cette cellule pour notre design sont les suivantes: sa simplicité et sa structure symétrique qui la rend très adaptée à ce type de pipeline, ses caractéristiques de point de vue surface d'intégration, sa fréquence de

fonctionnement et sa consommation en puissance. Nous pouvons ainsi estimer facilement la surface d'intégration et la fréquence de fonctionnement de notre architecture sans avoir recours aux différentes étapes de simulation et de modélisation lors de l'intégration à très grande échelle (full custom).

La conception d'un circuit de base du pipeline par vagues demande un synchronisme rigoureux. Cela nous a conduit à choisir une méthode de conception basée sur des étapes [GHO.95] bien définies d'équilibrage des délais. Ces différentes étapes présentés à la figure 4.8 peuvent se résumer en quatre étapes suivantes:

1^er étape: La fonction de base du circuit à pipeliner est subdivisée en plusieurs sous blocs dont chacun peut être réalisé à partir d'une combinaison donnée de la cellule de base.

2^éme étape: Décomposition de tous les blocs obtenus à partir de la première étape en plusieurs blocs formés par les cellules de base.

3^éme étape: Recherche du chemin critique qui revient à trouver le chemin qui contient le nombre le plus élevé de cellules de base.

4^éme étape: Équilibrer tous les chemins en introduisant des délais le long des chemins les plus courts. Ces délais sont constitué par les mêmes cellules de base.

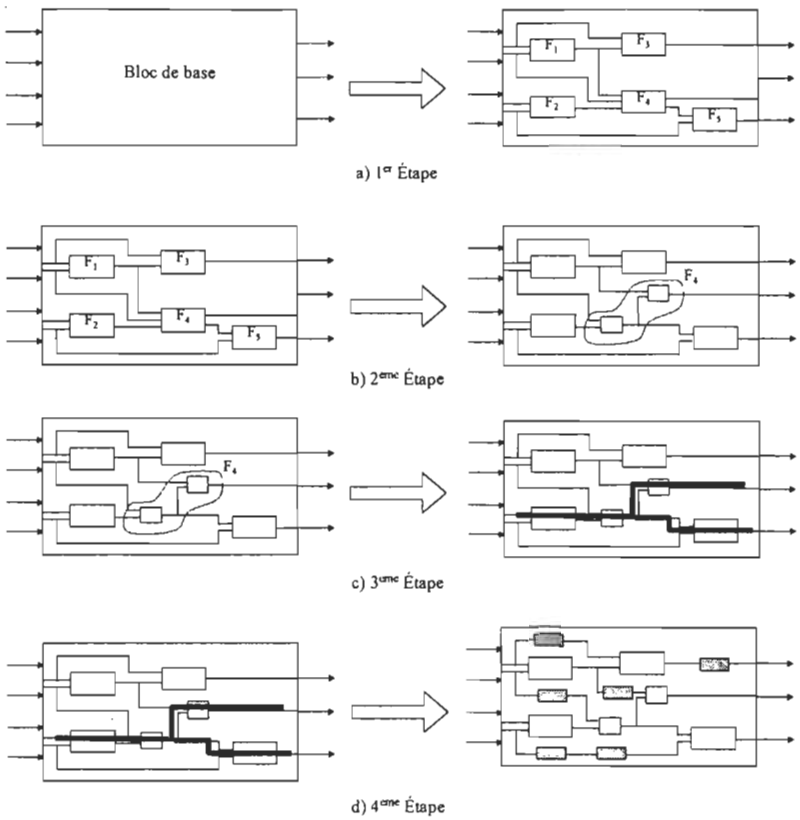


Figure 4.8 : différents étapes d'équilibrage des délais dans un BLC

4.2.2 Multiplieur et additionneur pipelinés

4.2.2.1 Réalisation d'un multiplieur 16 × 16 bits

Pour réaliser le multiplieur, nous avons mis en évidence toutes les fonctions de base qui vont conduire à la réalisation de la fonction de multiplication. Mais avant cette étape, il a fallu fixer la méthode algorithmique du fonctionnement de la multiplication. Cette méthode a été présentée dans [GHO.95]. Nous allons donc appliquer cette méthode pour réaliser notre multiplieur 16×16 bits.

L'architecture du multiplieur est semi-systolique. Elle peut être partagée en deux parties: une première partie qui permet le calcul du produit partiel et une deuxième partie effectuant la somme finale sous forme d'un additionneur à propagation de retenue. Afin de synchroniser la propagation des données nous avons partagé la partie effectuant les produits partielles en 8 pseudo-étages dont chacun effectue le produit du vecteur multiplicateur par un bit du multiplicande. Les résultats du produit sont ainsi additionnés aux résultats de l'étage précédent. La figure 4.9 présente l'architecture en bloc du multiplieur. Pour le premier et le deuxième pseudo-étage nous n'avons utilisé que des portes ***ET*** et des demi-additionneurs alors que pour les autres étages nous avons utilisé des blocs de base réalisant les fonctions suivantes:

$$S_i = (C_{i-1} \oplus S_{i-1})\overline{x_i y_i} + \overline{(C_{i-1} \oplus S_{i-1})} x_i y_i \tag{4.1}$$

$$C_i = (\overline{C_{i-1} \oplus S_{i-1}})C_{i-1} + (C_{i-1} \oplus S_{i-1})x_i y_i \tag{4.2}$$

La figure 4.10 a) montre la structure interne du bloc utilisant la cellule de base déjà décrite dans ce chapitre. Pour rendre ce bloc synchrone nous avons procédé de la même façon décrite préalablement en ajoutant des délais le long des différents chemins du bloc. La figure 4.10 b) montre la structure finale du bloc de base. Aussi nous avons ajouté des délais le long du chemin qui propage la multiplicande pour la rendre ainsi synchrone avec le reste du circuit.

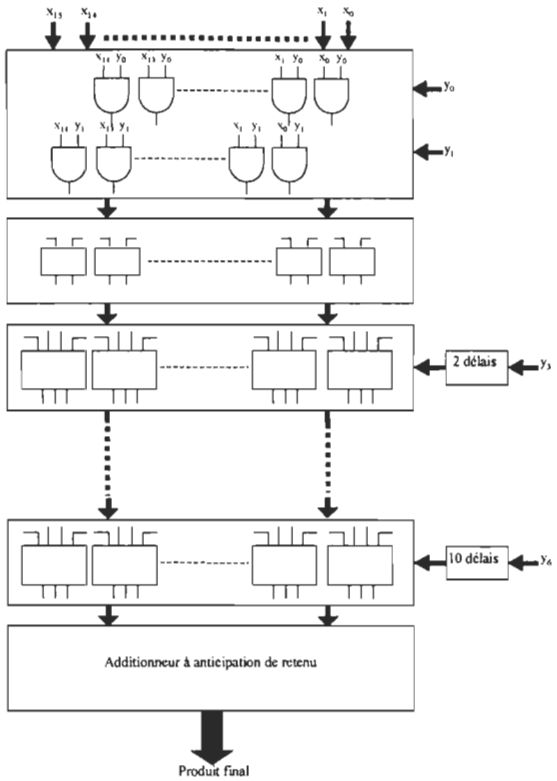


Figure 4.9 Architecture en bloc du multiplieur

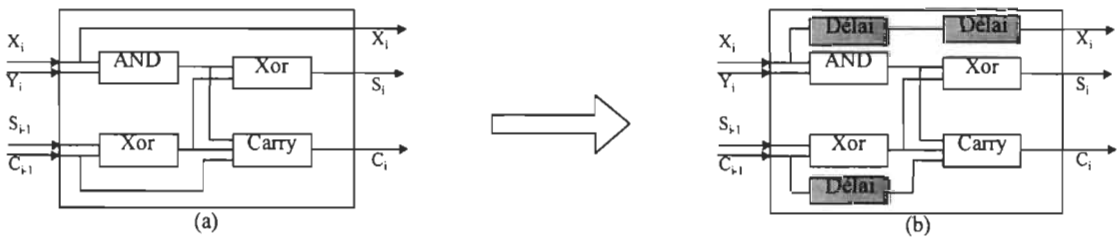


Figure 4.10 Structure interne du bloc de base

Concernant la réalisation de l'additionneur à propagation de retenue, nous avons procédé de la même façon que pour la première partie. Nous avons réalisé le bloc de base qui est un additionneur simple à partir de la cellule de base. Une partie de l'architecture interne de l'additionneur au complet est présenté à la figure 4.11 où on voit clairement la succession des pseudo-étages complètement synchrones.

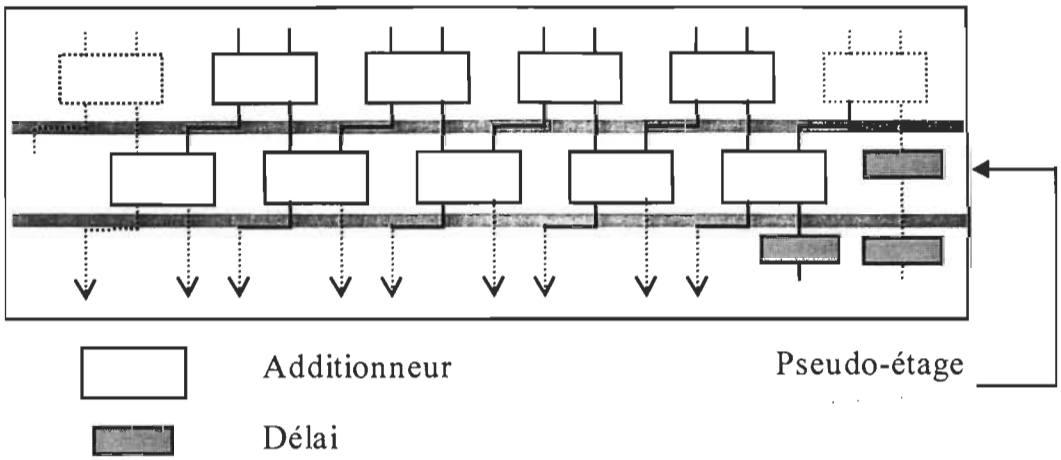
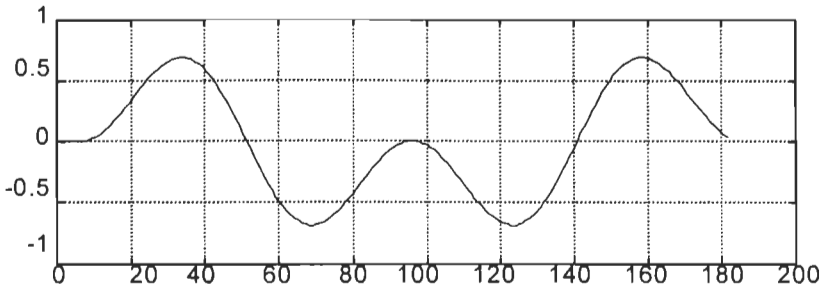
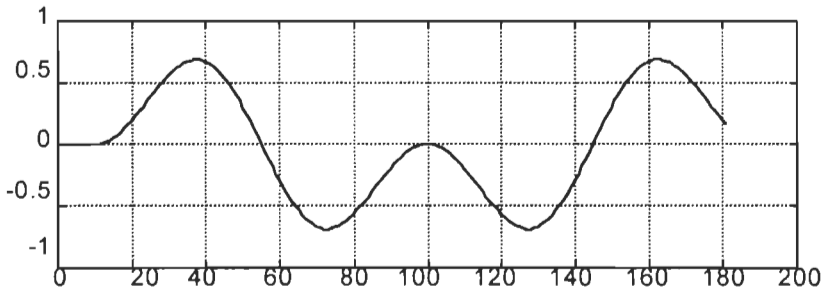


Figure 4.11 Structure de l'additionneur à propagation de retenu

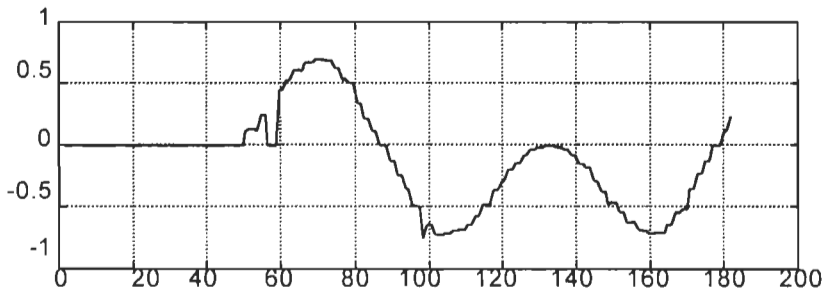
Pour vérifier le fonctionnement du multiplieur nous l'avons modélisé en langage de description matériel VHDL. Toutes les caractéristiques du fonctionnement logique de la cellule de base (figure 4.7) ont été tirées de la référence [GHO.95] et disponible au tableau 4.1: Les résultats de simulation du code VHDL sont présentés à la figure 4.12. À la figure 4.12 a) nous avons présenté les résultats obtenu dans MATLAB de la multiplication de deux fonctions sin(x) et sin(2x) effectuée dans MATLAB. Aux figures 4.12 b, c et d nous avons présenté les résultats de simulation de notre multiplieur selon les fréquences d'utilisation 400 MHz, 500MHz et 1 GHz respectivement. Ainsi nous remarquons que pour une fréquence limite de 400 MHz, les résultats de simulation sont identiques à ceux trouvés par MATLAB, alors que pour des fréquences supérieures, le multiplieur n'est plus capable de suivre la fréquence ce qui explique bien la théorie du pipeline par vagues.



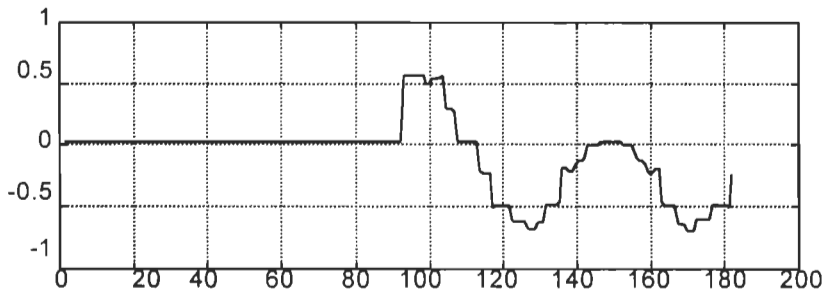
(a) Résultat du produit dans MATLAB



(b) Résultat du multiplieur pipeliné pour une fréquence de 400 Mhz



(c) Résultats du multiplieur pipeliné pour une fréquence de 500 Mhz



(d) Résultats du multiplieur pipeliné pour une fréquence de 1 Ghz

Figure 4.12 Résultats du produit de deux fonctions sin(x) et sin(2x),

a) Produit dans MATLAB, b),c) et d) Produit du multiplieur pipeliné à 400MHz 500MHz et 1GHz respectivement

4.2.2.2 Réalisation d'un additionneur à 16 bits

Pour la réalisation de l'additionneur 16×16 bits nous avons utilisé la même procédure que pour le multiplieur de la section 4.2.2.1 en se basant sur la même cellule de base de la figure 4.7. En effet, l'additionneur est identique à celui utilisé dans le multiplieur sauf qu'ici nous avons ajouté un bloc pour la conversion en complément à 2 à l'entrée et à la sortie de l'additionneur.

L'additionneur a été modélisé en langage VHDL. Les résultats de simulation sont présentés à la figure 4.13 dans laquelle nous avons présenté les résultats de la somme de deux fonction sin(x) et sin(2x). Nous remarquons aussi que pour des fréquences inférieures à 400 MHz, les résultats de l'additionneur pipeliné sont identiques à ceux trouvés par MATLAB. Alors que pour une fréquence supérieure à la fréquence seuil 400 MHz, l'additionneur pipeliné présente des résultats erronés

4.2.2.3 Réalisation d'un multiplieur/'additionneur 16 × 16 bits

Le nombre d'étages pipelines (ou d'étages virtuelles) pour le pipeline par vagues ont un rôle essentiel dans l'évaluation des performances de l'architecture en termes de temps de calcul. En fait plus on augmente le nombre d'étages plus la latence de l'architecture est élevée et peut engendrer un temps de calcul très long. Nous avons donc pensé à réduire la latence de l'architecture en utilisant des multiplieur/additionneurs puisque l'architecture du processeur accepte bien cette configuration. La structure globale du multiplieur/additionneur est présentée à la figure 4.14.

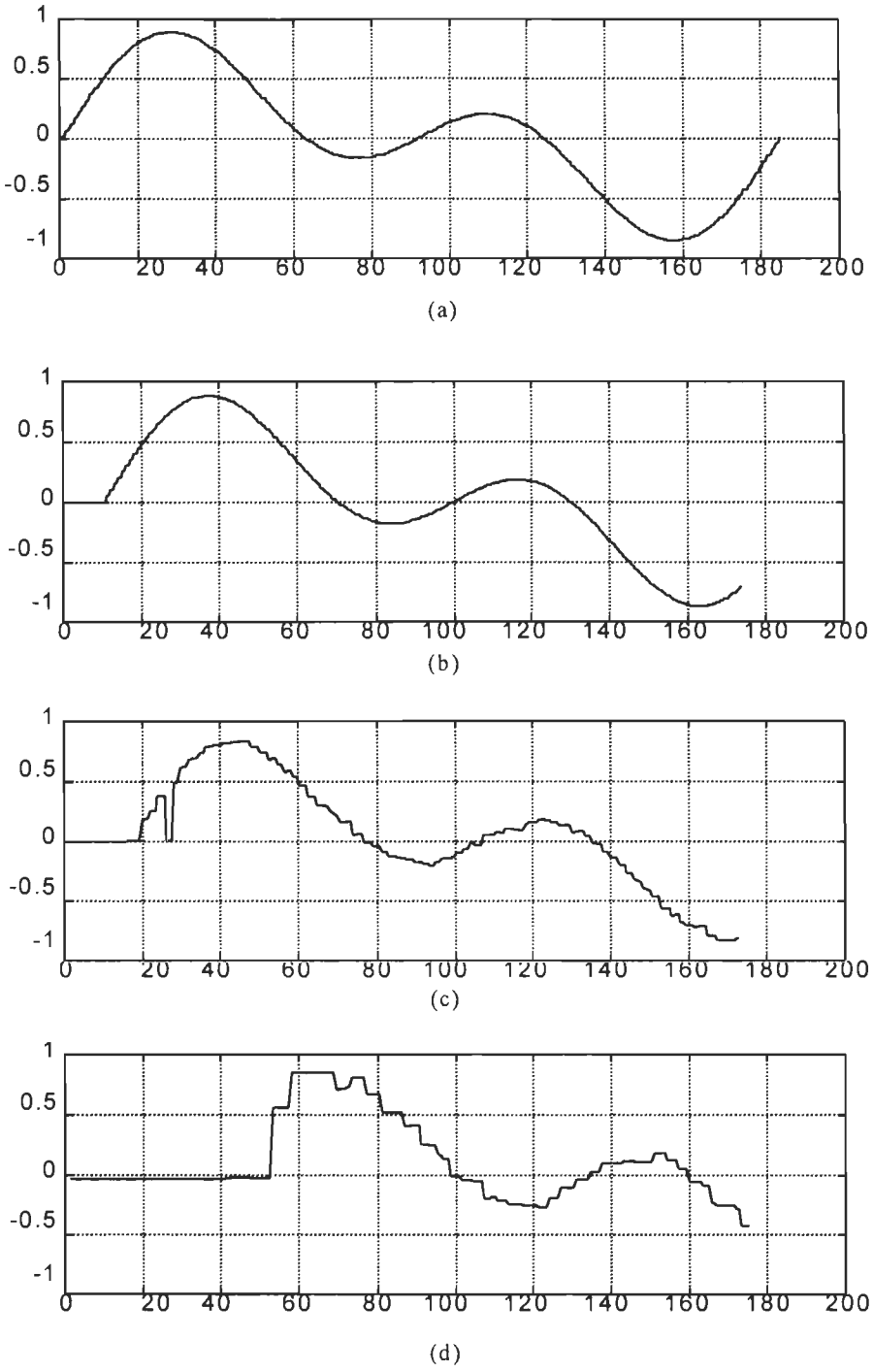


Figure 4.13 Résultats de la somme du sin(x) et sin(2x), a) résultat effectué dans MATLAB b),c) et d) résultats obtenue par l'additionneur pipeliné pour des fréquences de 400MHz,500MHz et 1GHz respectivement

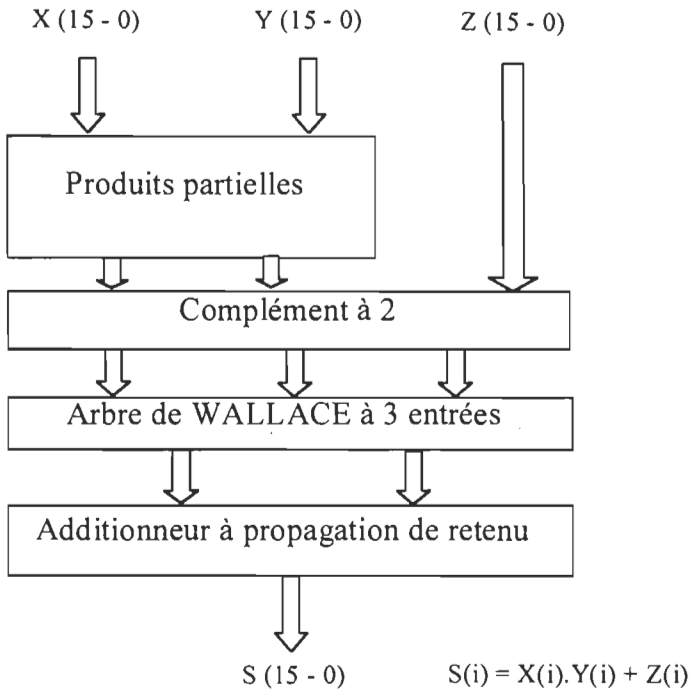


Figure 4.14 Structure globale du multiplieur/additionneur

4.2.3 Proposition et simulation de l'architecture à pipeline par vagues: WAVEKAL

Les blocs combinatoires pipelinés, seront intégrer dans l'architecture globale. La figure 4.15 présente l'architecture avec pipeline par vagues que nous nommons WAVEKAL. Cela nous a amené à effectuer des changements dans le programme principal utilisé dans le chapitre 3. Nous avons donc changé le multiplieur et l'additionneur utilisant le pipeline conventionnel par le multiplieur et additionneur à pipeline par vagues.

La structure globale du programme principal du VHDL est présentée à la figure 4.16, Les résultats de simulation sont présentés à la figure 4.17. Ces résultats sont identiques à ceux obtenus par la méthode conventionnelle, ce qui confirme bien la réussite de l'application du pipeline par vagues et aussi l'aptitude de l'architecture à accepter une telle théorie.

Nous allons donc effectuer l'étude des performances de la nouvelle architecture en termes de temps de calcul et de fréquence de fonctionnement. Par la suite une étude comparative sera complétée entre les deux méthodes de pipeline conventionnel, PIPEKAL, et la méthode de pipeline par vagues, WAVEKAL.

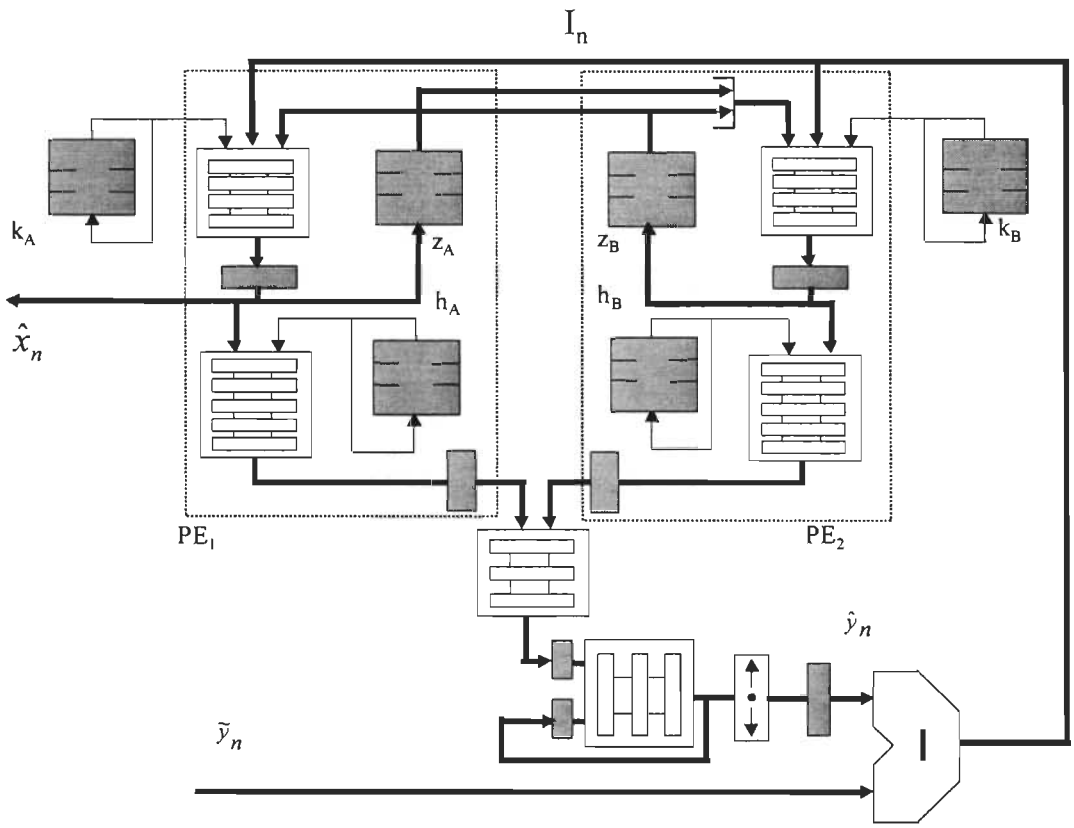
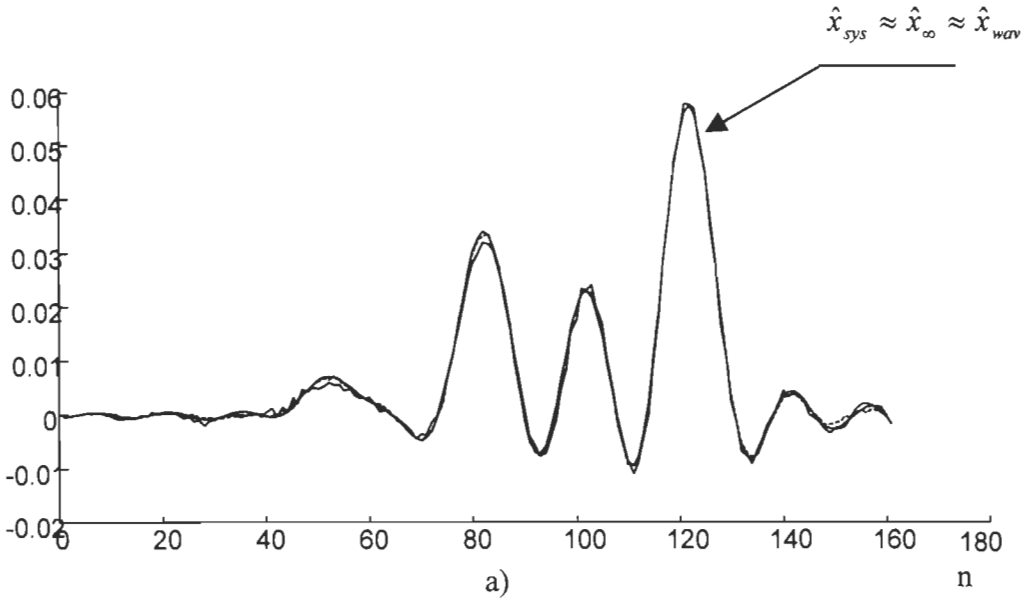


Figure 4.15 Structure globale de l'architecture du processeur WAVEKAL

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE work.pack_processeur_pipekal2.all;
   ENTITY processeur_WAVEKAL IS
PORT( clk,clr_Z,chain, s, en_Z,en_K                    :IN   std_logic;
      clr_acc,en_acc,clr_s, en_x , en_s                :IN   std_logic;
      clr_cal, en_cal, clr_reg, clr_x,  en_h           :IN   std_logic;
      pil_ink1,pil_ink2, pil_inh1,pil_inh2             :IN   std_logic_vector(7 downto 0);
      y_mes, distanc                                   :IN   std_logic_vector(15 downto 0);
      xout                                             :out  std_logic_vector(15 downto 0));
   END processeur_WAVEKAL ;
   ARCHITECTURE comportement OF processeur_WAVEKAL IS
            signal add_out1,add_out2                   :std_logic_vector(15 downto 0);
            signal mux_out, Z_out1,Z_out2              :std_logic_vector(15 downto 0);
            signal pil_outk1 ,pil_outk2                :std_logic_vector(7 downto 0);
            signal pil_outh1 ,pil_outh2                :std_logic_vector(7 downto 0);
            signal sortie1 ,sortie2                    :std_logic_vector(15 downto 0);
            signal yout1,yout2, yout, yacc_in, yacc_out :std_logic_vector(15 downto 0);
            signal y_cal_in, y_cal_out, innovation     :std_logic_vector(15 downto 0);
begin
            soustracteur  :soustract1                  port map (y_mes , y_cal_out , inovation);
            MULT_1        :mult/add_pipe_vagues_16_8   port map ( inovation,pil_outk1, mux_out, add_out1
      );
            MULT_2        :mult/add_pipe_vagues_16_8   port map ( inovation, pil_outk2,Z_out1 , add_out2 );
            Reg_3         :reg16                       port map (add_out1,clk,en_s,clr_s,sortie1);
            Reg_4         :reg16                       port map (add_out2,clk,en_s,clr_s,sortie2);
            BLOC_Z1       :bloc_stock_donnees_33       port map (clk,en_Z,clr_Z,sortie1,Z_out1);
            BLOC_Z2       :bloc_stock_donnees_32       port map (clk,en_Z,clr_Z,sortie2,Z_out2);
            Multiplex     :mux2a1                      port map (Z_out1,Z_out2,s,mux_out);
            BLOC_K1       :PILE_34_8                   port map (clk, en_K, chain, pil_ink1, pil_outk1);
            BLOC_K2       :PILE_34_8                   port map (clk, en_K, chain, pil_ink2, pil_outk2);
            MULT_3        :multip_pipe_vagues_16_8     port map ( sortie1,pil_outh1,yout1 );
            MULT_4        :multip_pipe_vagues_16_8     port map ( sortie2,pil_outh2,yout2 );
            Reg_5         :reg16                       port map (yout1,clk,yout11);
            Reg_6         :reg16                       port map (yout2,clk,yout22);
            BLOC_h1       :PILE_34_8                   port map (clk, en_h, chain, pil_inh1, pil_outh1);
            BLOC_h2       :PILE_34_8                   port map (clk, en_h, chain, pil_inh2, pil_outh2);
            ADD_3         :adder_pipe_vagues                 port map (yout11,yout22 , yout);
            Reg_7         :reg16                       port map (yout,clk,yout5);
            ADD_4         :adder_pipe_vagues                 port map (yout5,yacc_in , yacc_out);
            Reg_8         :reg16                       port map (yacc_out,clk,yacc_out1);
            Reg3          :reg16                       port map (yacc_out1,clk,en_acc,clr_acc,yacc_in);
            Decaleur      :decaleur_gauch_droit        port map (distanc,yacc_in,y_cal_in);
            Reg4          :reg16                       port map (y_cal_in,clk,en_cal,clr_cal,y_cal_out);
            Reg6          :reg16                       port map (z_out2 ,clk,en_x,clr_x,xout);
   END comportement ;
```

Figure 4.16 Code VHDL du programme source du processeur WAVEKAL



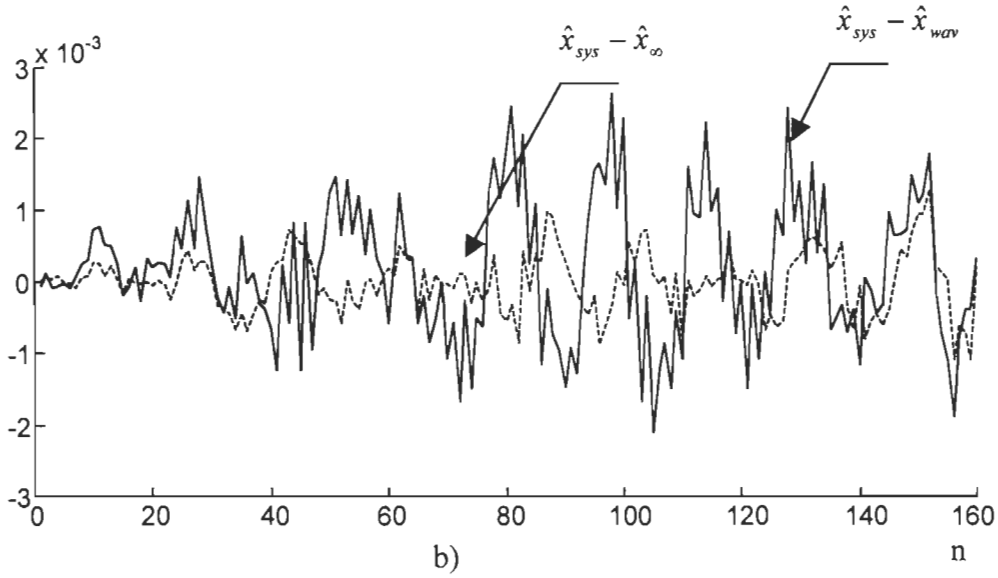
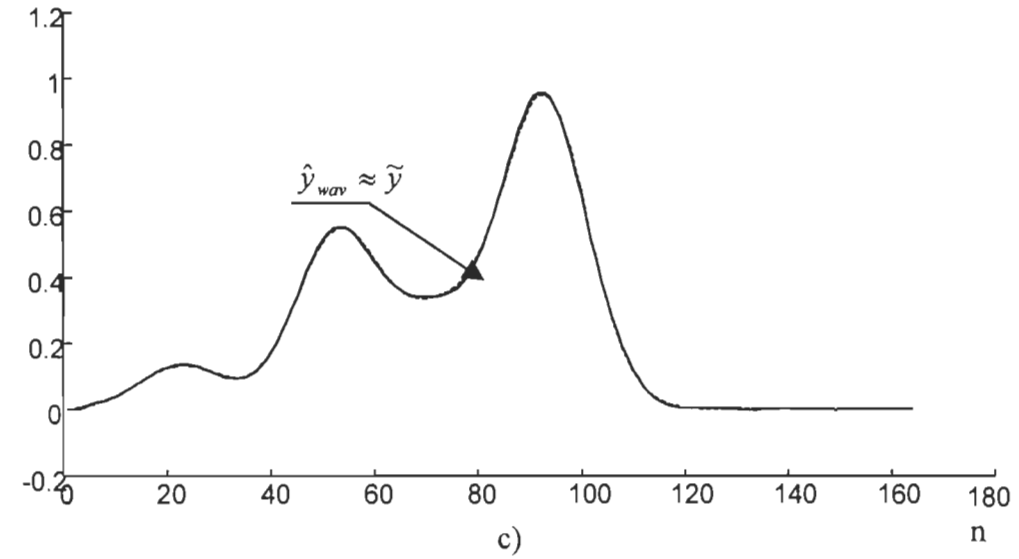
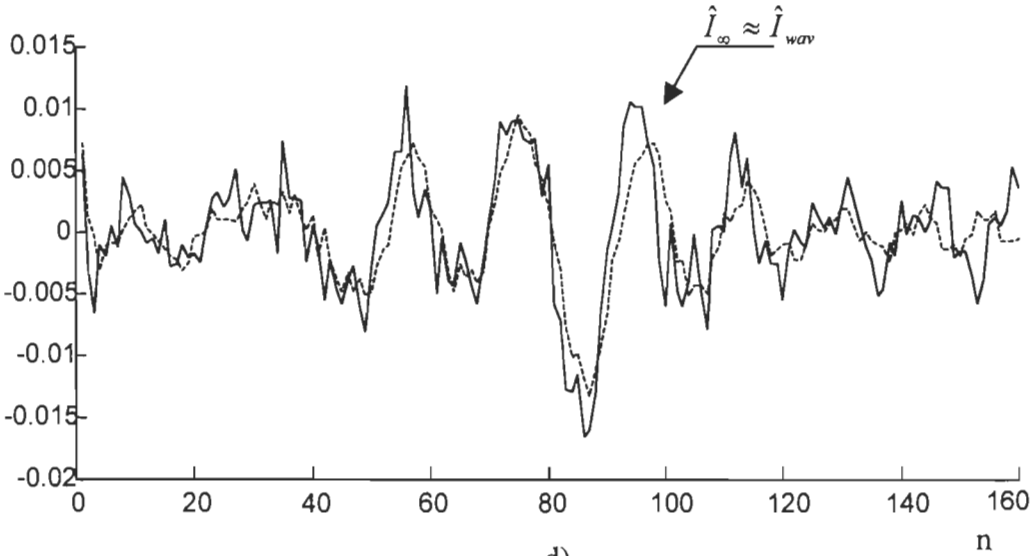


Figure 4.17 Résultats de simulation du processeur WAVEKAL a) Signal reconstitué

b) erreur du signal reconstitué



c)



d)

Figure 4.17 Résultats de simulation du processeur WAVEKAL (suite) c) Signal estimé

d) Innovation

4.3 Conclusion

Dans ce chapitre, la méthode du pipeline par vagues a été appliquée pour améliorer une architecture systolique dédiée à la reconstruction de signaux. Cette architecture SYSKAL est l'architecture de base utilisée dans ce travail. La réussite de cette technique est d'une part d'obtenir un degré très élevé de parallélisme afin d'augmenter de façon substantielle le débit des résultats et d'autre part de prouver que l'algorithme et l'architecture se prête à l'utilisation de cette méthode de pipeline. Les résultats obtenus prouvent que l'algorithme KALMAN+ [MAS.95 b] et l'architecture associée SYSKAL [MAS.98 b] acceptent cette méthode grâce à la profondeur importante des piles. De plus, les résultats obtenus sont très intéressants en terme d'amélioration du temps de calcul. Néanmoins, on doit prêter une grande attention en utilisant cette technique dans la conception du réseau de distribution de l'horloge. De préférence, l'horloge doit être localisé dans le dessin de masque ce qui est possible avec la structure de notre architecture.

5

Résultats de synthèse

Nous avons exploré jusqu'à maintenant l'application des deux méthodes du pipeline qui sont l'objet de ce travail: le pipeline conventionnel et le pipeline par vagues. Nous avons aussi montré par simulation l'efficacité et l'applicabilité de ces deux méthodes pour accélérer le processus de la multiplication et de l'addition. Dans cette section nous présentons les critères de performance pour pouvoir effectuer une étude comparative des deux méthodes en question. Les critères d'évaluation d'une méthode donnée sont multiples et variés, on cite en particulier le temps de calcul, la latence, la surface d'intégration et la consommation en termes de puissance. La satisfaction de ces différents critères dans une même architecture n'est pas généralement réalisable sur le plan pratique. La sélection de la meilleure méthode sera faite en se basant sur celle qui répond aux meilleures compromis entre ces différents critères. Dans notre cas, le choix a été fixé sur deux critères qui sont le temps de calcul et la surface d'intégration.

5.1 Évaluation du temps de calcul

Le temps de calcul est déterminé par :

$$t_c = (\frac{M}{S} + N_{EP}) \times f^{-1} \quad (5.1)$$

Avec

M : longueur du vecteur de la réponse impulsionelle,

S : nombre de processeurs élémentaires,

t_c : Temps de calcul pour reconstitué un échantillon x_n

N_{EP} : nombre d'étages pipeline ou d'étages pipeline virtuelles

f : fréquence de fonctionnement de l'horloge.

5.2 Évaluation de la surface d'intégration

Concernant la surface d'intégration elle peut être exprimée comme suit :

$$S_{total} = \sum S_i = S_{mut} + S_{add} + S_{mut/add} + S_{sous} + S_{pil} + S_{decal} \quad (5.2)$$

Les S_i sont les surfaces des différents blocs utilisés dans l'architecture à savoir le multiplieur, l'additionneur, le multiplieur/additionneur, le décaleur, le soustracteur et les piles pour sauvegarder les données. Pour les deux architectures le pipeline n'a été appliqué que pour les blocs critiques (blocs de logique combinatoire), ainsi l'équation (5.2) peut être réduite à

$$S_{total} = S_{cst} + S_{blc} \quad (5.3)$$

où

S_{cst} : Surface des différents blocs non pipelinés

S_{blc} : Surface des différents blocs pipelinés

En général pour un bloc donné la surface d'intégration peut être exprimée en fonction d'une unité usuelle appelée cellule élémentaire. Pour un pipeline conventionnel, l'insertion d'un registre de n bits augmente la surface d'intégration initiale de 5 n cellules élémentaires. Cela veut dire que pour m étages pipelinés la surface augmente de 4mn cellules élémentaires.

Pour un pipeline conventionnel la surface totale devient alors :

$$Spip_{total} = S_{cst} + S_{blc} + 8nm \tag{5.4}$$

Pour le pipeline par vagues, il est très difficile d'estimer la surface d'intégration des différents blocs vu que les cellules élémentaires utilisées doivent être fabriquées. Cependant on peut l'estimer à 2 fois la surface de la cellule élémentaire utilisée pour le pipeline (si on suppose qu'une cellule élémentaire est formée de deux transistors).Ceci dit

$$Spvag_{total} = S_{cst} + 2S_{blc} \tag{5.5}$$

5.3 Étude comparative

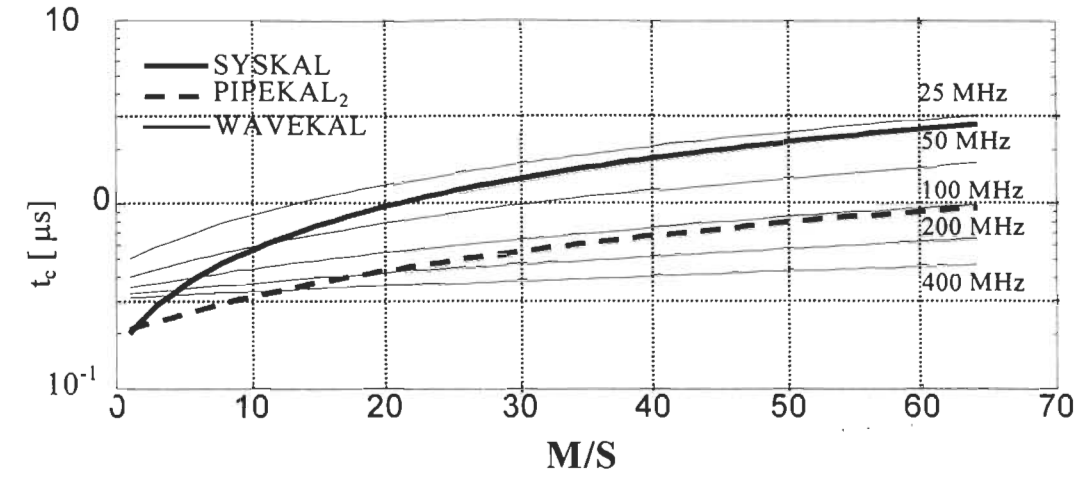
Le tableau 5.1 présente la synthèse des résultats du mémoire obtenus pour les différentes architectures proposées. Le critère temps de calcul et surface d'intégration (At) présentent la partie de synthèse des résultats du tableau. De ce critère on peut dire que le gain de performance (performance sans pipeline/performance avec pipeline) varie entre 2.8 et 9.7, suivant le degré et le type de pipeline. La dépendance entre la sortie et l'entrée des données dans cette architecture limite notre pouvoir d'exploiter au maximum les performances du pipeline. Cela vient du fait qu'il faut vider le pipeline à la fin du calcul de chaque échantillon pour pouvoir établir l'innovation qui sera utiliser pour l'échantillon suivant.

Tableau 5.1 : Évaluation des performances entre les différentes architectures pour une technologie CMOS de 1.5 μm de Mitel ($A=31mm^2$).

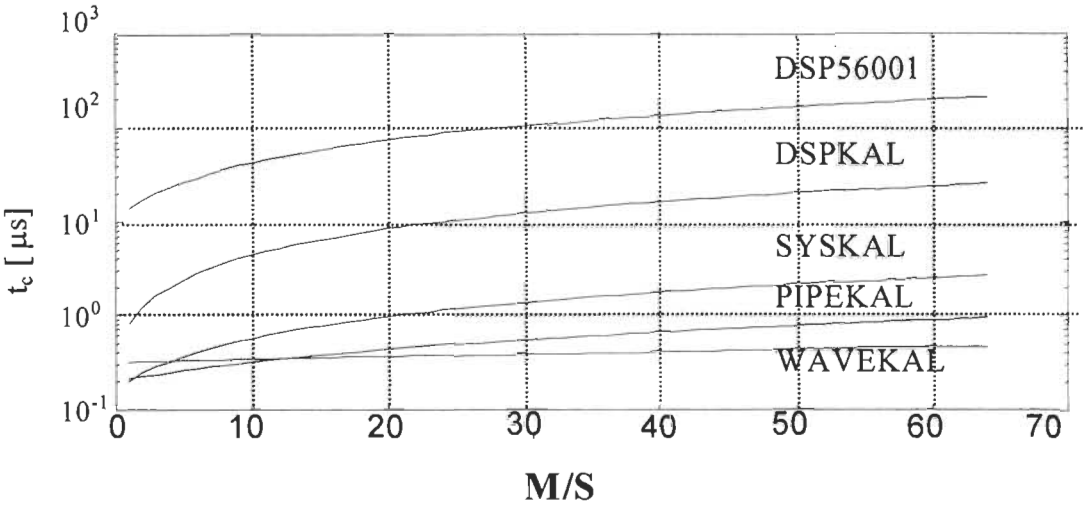
Architectures	Latence [ns]		Nombre de ports	Fréquence d'horloge [MHz]	Nombre de cycles	t_c [μs]	Surface	At
	Mult.	Add						
DSP56001 [8]	94.6	-	50 000	20	1 386	138.60	-	-
SYSKAL*	23.22	22.69	25 500	40	18	0.45	-	-
SYSKAL	39.75	36.75	15 823	8	35	4.38	A	4.38A
$PIPEKAL_1$	39.86	36.75	16 223	25	38	1.52	A	1.52A
$PIPEKAL_2$	50.0	49.0	17 983	80	52	0.65	1.25A	0.81A
$PIPEKAL_3$	50.0	49.0	16 384	40	45	1.12	1.03A	1.15A
WAVEKAL	110	80	19 380	400	120	0.3	1.5A	0.45A

* présente SYSKAL proposé dans [MAS95] avec S=4, M=64, dans une technologie CMOS de 1.2 μm et utilisant le pipeline tel $PIPEKAL_1$

Pour pouvoir effectuer une étude comparative entre les deux techniques du pipeline nous avons choisis de varier un autre le M/S qui a un impact important sur le rendement du pipeline. La figure 4.18 (a) montre l'effet de la variation de la fréquence de fonctionnement de WAVEKAL et du paramètre M/S sur le temps de calcul, t_c, des l'architectures. On voit donc que pour des fréquences faibles, inférieure à 100 MHz le pipeline conventionnel est plus performant puisqu'il présente toujours un temps de calcul latence inférieure à celle du pipeline par vagues. Alors que pour des fréquences très élevées le choix de l'architecture la plus performante dépend de la valeur de M/S. La figure 4.18 (b) montre par la suite le temps de calcul des différentes architectures en fonction de la valeur de M/S pour des fréquences supérieures à 200 MHz. Dans cette figure nous remarquons que pour des valeurs de M/S inférieures à 10 les architectures utilisant le pipeline conventionnel restent toujours les plus performantes, alors que pour des valeurs supérieures à 10, l'architecture WAVEKAL est la plus performante.



c)



b)

Figure 4.18 Effet de la longueur des piles dans les PE sur le temps de calcul, t_c, latence des architectures a) pour différentes fréquences de fonctionnement de WAVEKAL et b) pour différentes architectures avec une fréquence supérieure à 200 MHz

6
Conclusion

Le travail présenté propose une séries d'architecture dérivée du processeur SYSKAL dédié à la reconstitution de signaux basée sur le filtre stationnaire de Kalman [MAS.95 b] avec l'objectif principal d'augmenter les performances. Tel que proposé au chapitre 1, nous avons appliqué pour atteindre cet objectif, la technique du pipeline conventionnelle et la méthode du pipeline par vagues. Plus précisément, ce projet décrit les différentes étapes qui ont permis d'une part, d'atteindre des vitesses de fonctionnement très élevées et d'autre part, une surface d'intégration raisonnable en technologie VLSI. Tous les objectifs ont été atteints suivant la méthodologie de recherche définie au chapitre 1.Les détails du développement des architectures ont été présentés aux chapitres 3 et 4 pour l'application du pipeline conventionnel et par vagues respectivement.

Au cours de ce travail, nous avons pu appliquer deux méthodes avancées du pipeline qui sont le pipeline conventionnel et le pipeline par vague sur un processeur semi-systolique linéaire à structure en anneaux appelé SYSKAL. Dans un souci de réduire la surface d'intégration, nous avons procédé à la réduction de la surface du processeur en utilisant seulement deux processeurs élémentaires à la place de quatre. La surface d'intégration a alors été réduite d'un rapport de 63%. En contre partie, la profondeur des piles contenant les données a augmenté, augmentant ainsi le temps de calcul de 50% mais permettra d'exploiter avantages du pipeline. Sur cette architecture SYSKAL réduite nous avons proposé les architectures suivantes [ELO.97], [ELO.98] et [MAS.98 a]:

- $PIPEKAL_1$: Cette architecture est la première version pipelinée de SYSKAL, elle a été obtenue par l'application du pipeline séparant les différents blocs à logique combinatoire. Nous avons ainsi obtenu une vitesse de fonctionnement de 25MHz et un temps de calcul de 1.25µs par échantillon.
- $PIPEKAL_2$: afin d'améliorer les performances en terme de vitesse de calcul. Nous avons augmenter la profondeur du pipeline de l'architecture. Dans cette architecture l'accent a été mis sur les blocs a logique combinatoire du processeur pour lesquels le temps est une variable critique en locurrence les multiplieurs et les additionneurs. La fréquence de fonctionnement atteinte est de l'ordre de 80 MHz avec un temps de calcul d'un échantillon de 0.65 µs.
- $PIPEKAL_3$: Cette dernière version utilisant le pipeline conventionnel a été proposée pour réduire davantage la surface d'intégration on profitant de la symétrie de l'architecture. Un seul multiplieur est ainsi utilisé par processeur élémentaire ,

permettant un temps de calcul d'un échantillon de 1.25 μs avec une réduction de surface de 20% par apport à $PIPEKAL_2$.

- WAVEKAL: Pour les trois architectures précédentes, la vitesse de fonctionnement ne dépasse pas 100MHz. Nous avons donc proposé l'application d'une nouvelle technique de pipeline, appelé pipeline par vagues, cette technique à permis d'augmenter la fréquence de fonctionnement et de proposer une solution au problème de distribution de l'horloge. Nous avons donc démontrer la capacité de l'architecture à accepter ce type de pipeline et à assurer théoriquement une vitesse de fonctionnement de 400 MHz, en se basant sur les caractéristiques d'un principalement développée pour ce type de pipeline [GHO.95].

Ces architectures ont été modélisées en langage de programmation matériel VHDL, ce qui a permis, lieu, de valider leur fonctionnalité. Les résultats de simulation concordent bien avec ceux effectués sur le logiciel MATLAB. De plus, cela nous a permis de faire la synthèse dans l'environnement de Synopsys avec la technologie CMOS de la compagnie Mitel 1.5 μm via la société canadienne de la micro-électronique (SCM). Ceci nous a donné tous les paramètres de ces architectures, que ce soit de point de vue fréquence de fonctionnement ou surface d'intégration. Ces caractéristiques nous ont permis par la suite de faire une comparaison entres ces différentes architectures. Nous avons conclu que l'architecture $PIPEKAL_2$ est la plus efficace des architectures utilisant le pipeline conventionnel, suivant le critère, temps de calcul. Alors que $PIPEKAL_3$ est plus intéressante selon le critère, surface d'intégration. Concernant les deux méthodes de

pipeline, conventionnel et par vagues, le choix dépend essentiellement de la longueur des piles contenant les vecteurs de données $\hat{z}$,h et k_∞ d'un processeur élémentaire (M/S). Si elle dépasse une longueur de 20, le pipeline par vagues sera le plus performant, dans le cas contraire le pipeline conventionnel est le plus intéressante..

Durant ce travail, nous avons pu participer à trois conférences où les trois publications présentées, forment la majeure partie de ce travail. Dans [ELO.97] nous avons présenté la possibilité d'appliquer le pipeline conventionnel au processeur aussi que les différentes étapes de pipeline. Les résultats obtenus avec le pipeline conventionnel et l'étude comparative entre les trois architectures obtenues ont été publiés dans [ELO.98]. Finalement, dans le travail [MAS.98 a] nous avons proposé les différentes étapes d'application du pipeline par vagues ainsi qu'une étude comparative entre les deux techniques utilisées.

Les travaux futurs consisteront d'une part à fabriquer les trois architectures utilisant le pipeline conventionnel, d'autre part à faire le dessin de masques des cellules de base utilisées par le pipeline par vagues pour chercher leurs caractéristiques réelles et pouvoir ainsi fabriquer le processeur WAVEKAL.

Finalement, notre travail présente une contribution dans le domaine de l'implantation d'algorithmes en technologie CMOS. Dédiée au filtre de Kalman, les résultats de ce travail peuvent être appliqués dans des domaines variés tels que le contrôle, la prédiction, l'identification, l'adaptation, le traitement d'images et le traitement de la parole.

BIBLIOGRAPHIE

[BEN92] M. Ben Slima, R. Z. Morawaski et A. Barwicz, " Splined_based Variational Method with Constraints for Spectrophotometric Data Correction'', IEEE Trans. Instrumentation and Measurement, vol.41, N^0 6, Décembre 1992, pp.786-790.

[CHE95] L.G. Chen, Y.S. Jehng, and T.D Chiueh, ''Pipeline Interleaving Design for FIR, IIR, and FFT Array Processors'', *Journal of VLSI Signal Processing*, Vol. 10, 1995, pp. 275-293.

[COT69] L.W. Cotten. ''Maximum-rate pipeline systems'', In 1996 AFIPS Proc. Spring Joint Computer Conference, pages 581-586, 1969.

[CRI91] P. B. Crilly, ''A Quantitative Evaluation of Various Iterative Deconvolution Algorithms'', IEEE- Trans on Instrumentation & Measurement, vol. 40,N^0 3, Juin 1991, pp. 558-562.

[DEM83] G. Demoment et A. Segalen, ''Adéle, Afast Suboptimal Estimator for Real-Time Deconveolution'', Electronics Letters, vol. 19, N^0 3, 1983, pp. 86-88.

[DEM85] G. Demoment et R. Reynaud, '' Fast Minimum Variance Deconvolution'', IEEE Transactions on Acoustics, Speech, Processing, vol. 33, N^0 4, 1985, pp. 1324-1326.

[DEM96] T A DeMasse, Z. Ciccone ''Digital Integrated Circuits'', JON WILEY & SONS 1996.

[EKR87] B. Ekroot, ''Optimization of pipelined Processors by insertion of combinational logic dely'', Ph.D. thesis, Stanford University, Stanford, CA, 1987.

[ELO96] M. B. Elouafay, ''Estimation des délais d'un processeur dans l'environnement de MENTOR-GRAPHICS'', projet de fin d'étude à UQTR, Hiver 1996.

[ELO97] M. B. Elouafay, D. Massicotte ''Application de la méthode du pipeline par vagues sur un processeur systolique'', Congrès de l'Association Canadienne-Française pour l'Avancement des Sciences (ACFAS'97), Québec., Trois-Riviéres Août 1997.

[ELO98] M. B. Elouafay, D. Massicotte "A Pipelined Systolic Architecture for A KALMAN-Filter-Based Signal Reconstruction Algorithm", IEEE Canadian Conference on Electronic and Computer Engineering (CCECE'98), Waterloo May 1998.

[GHO95] D. Ghosh and S.K. Nandy, "Design and Realization of High-Performance Wave-Pipelined 8×8 Multiplier in CMOS Technology", *IEEE Trans. on VLSI*, Vol. 3, No. 1, March 1995, pp. 36-48.

[GRA92 a] T. Gray. T. Hughes. S. Arora. W. Liu. And R. Cavin. "Theortical and practical issues in CMOS wave-pipelining", in IFIP Transactions A:Computer Science and Technology-VLSI '91, pp. 397-409, Edinburgh, Scotland. 1992.

[GRA92 b] C. T. Gray, W. Liu, and R. K. Cavin III. "Timing Constraints for wave Pipelined Systems", Technical Report NCSU_VLSI-92-06, Nort Carolina State University, December 1992.

[GRA94] C. T. Gray. W. Liu. And R. K. Cavin. Wave-pipelining: Theory and CMOC Implementation. Kluwer Academic Publishers. 1994.

[HEN92] J L. Hennessy, D A Patterson "Architecture des ordinateurs une approche quantitative", Mc GRAW-HILL 1992.

[KLA90] F. Klass and J. Mulder. "CMOS implementation of wave pipelining", Tech. Rep. 1-68340-44(1990)02. Delft University of Technol., Dec. 1990.

[KLA92] F. Klass, "Balacing Circuits for Wave pipelining", Technical Report, Stanford University, 1992, CSL-TR-92-549.

[KLA94] F. Klass and M.J. Flynn. "Fast Multiplication in VLSI Using Wave Pipelining Techniques", Journal of VLSI Signal Processing, 7, 233-248 (1994).

[KUN82] H.T. Kung, "Why systolic architecture", *IEEE Computer Magazin*, vol.15, 1982, pp. 37-46.

[LIU94] W. Liu and D. Fan "A 250-MHz Wave Pipelined Adder in 2-∝m CMOS", IEEE J. Solid-State Sircuits, Vol. 29. No. 9. pp. 1117-1128, Sept 1994

[MAS95 a] D. Massicotte, "Une approche à l'implantation en technologie VLSI d'une classe d'algorithmes de reconstitution de signaux", Ph.D. Thesis, École Polytechnique of Montreal, 1995.

[MAS95 b] D. Massicotte, R. Z. Morawski, A. Barwicz, "Incorporation of a Positivity Constraint Into a Kalman-Filter-Based Algorithm for Correction of Spectrometric Data", *IEEE Trans. Instrum. Meas*, Vol. 44, No 1, Feb. 1995, pp. 2-7

[MAS98 a] D. Massicotte, M. B. Elouafay "A Systolic Architecture for A KALMAN-Filter-Based Signal Using the Wave Pipelines Method" International Conference on Electronics, Circuits and Systems (ICECS'98), Lisbon Mars 1998.

[MAS98 b] D. Massicotte, "A Systolic VLSI Implementation of Kalman-Filter-Based Algorithms for Signal Reconstruction", *IEEE ICASSP*, Seattle, 12-15 May 1998.

[MEN77] J. Mendel, "White noise estiomators for seismic data processing", IEEE Transactions on Automatic Control, vol Ac-22, N^0 5, 1977, pp.694-706.

[NGU93] V. Nguyen. W. Liu . C. T. Gray. And R. K. Cavin. "A CMOS multiplier using wave-pipelining", in Proc. Custom Integrated Circuits Conf., May 1993. San Diego. CA.

[QUI89] P. Quinton, Y. Robert, "Algorithmes et Architectures systoliques" MASSON 1989.

[SEV96] "Low-Power HF Microelectronics a unified approach", Gerson A. S. Machado 1996.

[SJU92] S. T. Ju and C.W. Jen. "A high speed multiplier design using wave-pipelining technique", in Proc. IEEE APCCAS. 1992. Australia. Pp. 502-506.

[THO92] C. Thomas Gray, andR. K. Cavin. "Timing Constrants for Wave-Pipelined Systems", Technical Report, North Carolina State University, December 1992, NCSU-VLSI-92-06.

[THO96] J. Thomas, "Pipelined Systolic Architectures for DLMS Adaptive Filtering", *Journal of VLSI Signal Processing*, Vol. 12, 1996, pp. 223-246.

[WES93] N.Weste, K.Eshraghian "Principles of CMOS VLSI Design A Systems Prespective", Addison-Wesley 1993.

[WON91] D.Wong, "Techniques for designing high-performance digital circuits using wave pipelining", Ph.D. Stanford University. Stanford. CA. Sept. 1991.

[WON93] D. C. Wong, G. De Micheli. And M. J. Flynn, ``Designing high-performance digital circuits using wave-pipelining: Algorithms and practical experiences``. IEEE Trans. Computer-Aided Design. Pp. 25-46. Jan. 1993.

[SAV88] Yvon Savaria ''Conception et vérification des circuits VLSI'', polytechnique, 1988.

ANNEXE A

Programmes sources des architectures :
SYSKAL, PIPEKAL$_1$, PIPEKAL$_2$, PIPEKAL$_3$ et
WAVEKAL

****
****
****
****
****
****
****
****
****
****

Programme principal pour l'architecture SYSKAL

Réalisé par Elouafay My Boubker

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;

PACKAGE pack_processeur_syskal IS


-------------------------------------------------------------------
-------------------------------------------------------------------

 Component multe_4_4
   port(A  : in   std_logic_vector(3 downto 0)  ;
        B  : in   std_logic_vector(3 downto 0)  ;
        S  :out  std_logic_vector(7 downto 0)) ;
 end component;
-------------------------------------------------------------------
 Component multe_4_3
   port(A  : IN  std_logic_vector(3 downto 0  );
        B  : IN  std_logic_vector(3 downto 0  );
        s1 : OUT std_logic_vector(5 downto 0);
        s2 : OUT std_logic                    );
 end component;
-------------------------------------------------------------------
Component csfa_wallac_3

   port (e1, e2 ,e3 : IN  std_logic ;
```

```
          s1 , s2    : OUT std_logic);
 end component;
-----------------------------------------------------------------
```

Component ful_adder

```
   port  (e1 , e2    : IN std_logic   ;
          s1 , c     : OUT std_logic  );
 end component;
-----------------------------------------------------------------
```

Component ful_adder1

```
   port  (e1 , e2 ,ci   : IN  std_logic  ;
          s1 , c        : OUT std_logic );
 end component;
-----------------------------------------------------------------
```

Compenent multiplieur_16_8

```
        port ( A      : in  std_logic_vector (7  DOWNTO 0)   ;
               B      : in  std_logic_vector (15  DOWNTO 0)  ;
              Sort    : out std_logic_vector (15  DOWNTO 0)  );
end component ;
-----------------------------------------------------------------
```

Compenet adder16

```
        PORT( A  : IN  std_logic_vector (15 DOWNTO 0);
              B  : IN  std_logic_vector (15 DOWNTO 0);
             Sor: OUT std_logic_vector(15 DOWNTO 0));
end component ;
-----------------------------------------------------------------
```

Compent soustract16

```
        PORT( A  : IN  std_logic_vector (15 DOWNTO 0);
              B  : IN  std_logic_vector (15 DOWNTO 0);
             Sor: OUT std_logic_vector(15 DOWNTO 0));
        END component ;
-----------------------------------------------------------------
```

Compentent PILE_34_16

```
GENERIC (larg     : INTEGER :=15);
PORT   (clk       : IN std_logic;
          en      : IN std_logic;
        selector  : IN std_logic;
         pile_in  : IN std_logic_vector(larg DOWNTO 0);
        pile_out  : BUFFER std_logic_vector(larg DOWNTO 0));
end COMPONENT;
```

```
-------------------------------------------------------------

Compenet mux2a1_16
PORT(IN0      : in  std_logic_vector(15 downto 0);
     IN1      : in  std_logic_vector(15 downto 0);
     S        : in  std_logic;
  OUT_MUX   : out std_logic_vector(15 downto 0));
end COMPONENT;
-------------------------------------------------------------
Compenent reg16
   PORT( A          : IN std_logic_vector (15 DOWNTO 0);
         clk, en, clr : IN std_logic;
           Sor        : OUT std_logic_vector (15 DOWNTO 0));
end COMPONENT;
-------------------------------------------------------------
Compenent bloc_stock_donnees_33
   PORT( clk    : IN  std_logic;
         en     : IN  std_logic;
         clr    : IN  std_logic;
        pile_in : IN  std_logic_vector(15 DOWNTO 0);
        pile_out: OUT std_logic_vector(15 DOWNTO 0));

end COMPONENT;
-------------------------------------------------------------
 Compenent bloc_stock_donnees_32
   PORT( clk    : IN  std_logic;
         en     : IN  std_logic;
         clr    : IN  std_logic;
        pile_in : IN  std_logic_vector(15 DOWNTO 0);
        pile_out: OUT std_logic_vector(15 DOWNTO 0));

END COMPONENT;
-------------------------------------------------------------
Compenent decaleur_gauch_droit
PORT(distance: IN  std_logic_vector (5 DOWNTO 0);
     input   : IN  std_logic_vector (15 DOWNTO 0);
     output  : OUT std_logic_vector (15 DOWNTO 0));

END COMPONENT;
-------------------------------------------------------------

END pack_processeur_syskal ;

-------------------------------------------------------------
```

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


ENTITY multe_4_4 IS
   port ( A  : in  std_logic_vector (3 DOWNTO 0)  ;
          B  : in  std_logic_vector (3 DOWNTO 0)  ;
          s  : out std_logic_vector (7 DOWNTO 0)  );
END multe_4_4;

ARCHITECTURE behav OF multe_4_4 IS
 BEGIN
 s <= unsigned(A)*unsigned(B);
END behav;
-------------------------------------------------------------------

LIBRARY IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_ARITH.all;


ENTITY multe_4_3 IS
  port(A  : IN  std_logic_vector (3 DOWNTO 0)  ;
       B  : IN  std_logic_vector (3 DOWNTO 0)  ;
       s1 : OUT std_logic_vector (5 DOWNTO 0)  ;
       s2 : OUT std_logic      );

END  multe_4_3 ;

ARCHITECTURE behav OF multe_4_3 IS
BEGIN

        PROCESS(A,B)
        VARIABLE A1, B1 :std_logic_vector (2 DOWNTO 0);
        VARIABLE A4, B4 :std_logic      ;

        BEGIN
        A1:= (A(2),A(1),A(0));
        B1:= (B(2),B(1),B(0));
        A4:=  A(3);
        B4:=  B(3);
```

```
        s1 <= unsigned(A1)*unsigned(B1);
        s2 <= A4 xor B4;

        END PROCESS;

END behav;
-------------------------------------------------------------------

LIBRARY IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_ARITH.all;


ENTITY csfa_wallac_3 IS
    port (e1, e2 ,e3 : IN  std_logic ;
          s1 , s2    : OUT std_logic);
end csfa_wallac_3;

ARCHITECTURE behave OF csfa_wallac_3 IS
 BEGIN
 s1 <= (e1 xor e2)xor e3;
 s2 <= ((e1 xor e2)and e3)or (e1 and e2);
end behave;
-------------------------------------------------------------------
LIBRARY IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_ARITH.all;


ENTITY ful_adder IS
    port (e1 , e2 : IN  std_logic ;
          s1 , c : OUT std_logic);
end ful_adder;

ARCHITECTURE behave OF ful_adder IS
 BEGIN
 s1 <= e1 xor e2;
 c <= e1 and e2;
end behave;

-------------------------------------------------------------------

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

```
use ieee.std_logic_ARITH.all;


ENTITY ful_adder1 IS
    port (e1 , e2 , ci : IN  std_logic ;
          s1 , c       : OUT std_logic);
end ful_adder1;

ARCHITECTURE behave OF ful_adder1 IS
 BEGIN
 s1 <= e1 xor e2 xor ci;
 c <= ((e1 or e2) and ci) or (e1 and e2);
end behave;

-----------------------------------------------------------------
LIBRARY IEEE ;

USE ieee.std_logic_1164.all  ;
USE ieee.std_logic_ARITH.all ;



ENTITY multiplieur_16_16 IS
  port ( A      : in  std_logic_vector (15  DOWNTO 0)   ;
         B      : in  std_logic_vector (15  DOWNTO 0)   ;
         Sort   : out std_logic_vector (15  DOWNTO 0)  );

END multiplieur_16_16;

ARCHITECTURE behav OF multiplieur_16_16 IS
BEGIN
PROCESS(A,B)
variable aa : std_logic_vector (14 DOWNTO 0):="000000000000000" ;
variable bb : std_logic_vector (14 DOWNTO 0):="000000000000000" ;
variable sig: std_logic;
variable S  : std_logic_vector(29 downto 0);


begin
        aa  := A(14 downto 0);
        bb  := B(14 downto 0);
        S   := unsigned(aa)*unsigned(bb);

        sig := A(15) xor B(15);
```

```
        Sort <= sig & S(29 downto 15);

end PROCESS;
END behav;

-------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE Ieee.std_logic_arith.all;



ENTITY adder16 IS
        PORT( A: IN std_logic_vector (15 DOWNTO 0);
            B: IN std_logic_vector (15 DOWNTO 0);
            Sor: OUT std_logic_vector(15 DOWNTO 0));
        END adder16;

ARCHITECTURE behave OF adder16 IS
BEGIN
PROCESS(A,B)
   VARIABLE e1,e2 ,s : std_logic_vector (15 DOWNTO 0);
   BEGIN
      e1:= A(15)&A(14 downto 0);
                e2:= B(15)&B(14 downto 0);


                IF (e1(15)= '0')then
          e1:= e1;
        elsif (e1="1000000000000000")then
            e1(15):= NOT(e1(15));
        else
            e1:= e1(15)&NOT(e1(14 downto 0));
        END IF;

        IF (e2(15)= '0')then
          e2:= e2;
        elsif (e2="1000000000000000")then
            e2(15):= NOT(e2(15));
        else
            e2:= e2(15)&NOT(e2(14 downto 0));
        END IF;

        s:= unsigned(e1) + unsigned(e2) ;
```

```
        IF (s(15)= '0')then
            Sor<= s(15 downto 0);
        elsif (s ="1000000000000000")then
           Sor<= NOT(s(15))&s(14 downto 0);
         else
            Sor<= s(15)&NOT(s(14 downto 0));
        END IF;
END PROCESS;

END behave;


-----------------------------------------------------------------


LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;



ENTITY  soustract16  IS
        PORT( A  : IN  std_logic_vector (15 DOWNTO 0);
              B  : IN  std_logic_vector (15 DOWNTO 0);
              Sor: OUT std_logic_vector(15 DOWNTO 0));
        END soustract16  ;

ARCHITECTURE behave OF soustract16  IS

BEGIN
PROCESS(A,B)
   VARIABLE e1,e2 ,s : std_logic_vector (15 DOWNTO 0);
   BEGIN
      e1:= A(15)&A(14 downto 0);
                e2:= B(15)&B(14 downto 0);


         IF (e1(15)= '0')then
        e1:= e1;
        elsif (e1="1000000000000000")then
        e1(15):= NOT(e1(15));
        else
           e1:= e1(15)&NOT(e1(14 downto 0));
        END IF;
```

```
        IF (e2(15)= '0')then
           e2:= Not(e2);
        elsif (e2="1000000000000000")then
           e2(15):= NOT(e2(15));
        else
           e2:= Not(e2(15))&e2(14 downto 0);
        END IF;

        s:= unsigned(e1) + unsigned(e2) ;

        IF (s(15)= '0')then
            Sor<= s(15 downto 0);
        elsif (s ="1000000000000000")then
           Sor<= NOT(s(15))&s(14 downto 0);
         else
            Sor<= s(15)&NOT(s(14 downto 0));



        END IF;
END PROCESS;

END behave;

-------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;



ENTITY PILE_34_16 IS
  GENERIC (larg     : INTEGER :=15);
    PORT  (clk      : IN std_logic;
             en     : IN std_logic;
             selector : IN std_logic;
        pile_in  : IN std_logic_vector(larg DOWNTO 0);
        pile_out : BUFFER std_logic_vector(larg DOWNTO 0));
END PILE_34_16;

ARCHITECTURE behav OF PILE_34_16 IS
CONSTANT stack_depth: INTEGER := 31;
TYPE pile_vect IS ARRAY (stack_depth DOWNTO 0) OF
```

```
                    std_logic_vector(larg DOWNTO 0);
SIGNAL a: std_logic_vector(larg DOWNTO 0);
SIGNAL gated_clk : std_logic;

BEGIN
        gated_clk <= clk AND en;
        mux: PROCESS (selector,pile_in,pile_out)
    VARIABLE aa : std_logic_vector(larg DOWNTO 0);
        BEGIN
                CASE selector IS
                    WHEN   '1' => aa := pile_in;
                WHEN   '0' => aa := pile_out;
                    WHEN OTHERS => aa := (OTHERS => 'X');
                END CASE;
            a <= aa;
        END PROCESS mux;

    reg_ar: PROCESS (gated_clk)
        VARIABLE temp_pile: pile_vect;
    BEGIN
IF ((gated_clk = '1') AND (gated_clk'LAST_VALUE = '0') AND
    (gated_clk'EVENT)) THEN pile_out <= temp_pile(0);
      FOR i IN 0 TO ((stack_depth)-1) LOOP
            temp_pile(i) := temp_pile(i+1);
      END LOOP;
  temp_pile(stack_depth) := a;
END IF;
END PROCESS reg_ar;
END behav;

-------------------------------------------------------------------


LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;

ENTITY mux2a1_16 IS
      PORT(IN0     : in std_logic_vector(15 downto 0);
      IN1     : in std_logic_vector(15 downto 0);
      S       : in std_logic;
      OUT_MUX  : out std_logic_vector(15 downto 0));
END mux2a1_16;

ARCHITECTURE behv OF mux2a1_16 IS
```

```
BEGIN
  mux2a1_process: PROCESS(S,IN0,IN1)
  BEGIN
      CASE S IS
          WHEN '0' => OUT_MUX <= IN0;
          WHEN '1' => OUT_MUX <= IN1;
          WHEN OTHERS => NULL;
      END CASE;
  END PROCESS mux2a1_process;
END behv;

-----------------------------------------------------------------


LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;

-- D flip-flop avec clear et enable
--     type = D
--     W = 12 bits
--     clock_edge = RISING
--     load_enable = MUX_GATE
--     clear = SYNC
--     set = NONE
--     test = NONE

ENTITY reg16 IS
        PORT( A           : IN  std_logic_vector (15 DOWNTO 0);
                clk, en, clr  : IN  std_logic;
                Sor         : OUT std_logic_vector (15 DOWNTO 0));
END reg16;

ARCHITECTURE behv OF reg16 IS
BEGIN
        PROCESS(clk)
        BEGIN
                IF ((clk='1') and (clk'event) and (clk'last_value ='0')) THEN
                        IF clr= '1' THEN
                   Sor <= "0000000000000000";
                        ELSIF en = '1' THEN
                        Sor <= A;

                        END IF;
                END IF;
```

```
        END PROCESS;
END behv;
-------------------------------------------------------------



LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;



ENTITY  bloc_stock_donnees_33 IS
  PORT(clk : IN  std_logic;
            en : IN  std_logic;
      clr : IN  std_logic;
   pile_in : IN  std_logic_vector(15 DOWNTO 0);
  pile_out : OUT std_logic_vector(15 DOWNTO 0));
END bloc_stock_donnees_33;

ARCHITECTURE behv OF  bloc_stock_donnees_33 IS
CONSTANT stack_depth: INTEGER := 31;
TYPE pile_vect IS ARRAY ((stack_depth) DOWNTO 0) OF
                        std_logic_vector(15 DOWNTO 0);
SIGNAL gated_clk :std_logic;
BEGIN
        gated_clk <= clk AND en;
        shift_stack15_process: PROCESS (gated_clk, clr)
        VARIABLE temp_pile: pile_vect;
BEGIN
        IF      ( clr = '1' ) THEN
        pile_out <= "0000000000000000";
        temp_pile := ( OTHERS => "0000000000000000");
        ELSIF   ((gated_clk = '1') AND (gated_clk'LAST_VALUE = '0')
                        AND (gated_clk'EVENT)) THEN
        pile_out <= temp_pile(0);
FOR i IN 0 TO ((stack_depth)-1) LOOP
        temp_pile(i) := temp_pile(i+1);
END LOOP;
    temp_pile(stack_depth) := pile_in;
END IF;
END PROCESS shift_stack15_process;
END behv;
```

```
------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;



ENTITY  bloc_stock_donnees_32 IS
  PORT(clk : IN  std_logic;
           en : IN  std_logic;
      clr : IN  std_logic;
   pile_in : IN  std_logic_vector(15 DOWNTO 0);
  pile_out : OUT std_logic_vector(15 DOWNTO 0));
END bloc_stock_donnees_32;

ARCHITECTURE behv OF  bloc_stock_donnees_32 IS
CONSTANT stack_depth: INTEGER := 30;
TYPE pile_vect IS ARRAY ((stack_depth) DOWNTO 0) OF
                                 std_logic_vector(15 DOWNTO 0);
SIGNAL gated_clk :std_logic;
BEGIN
        gated_clk <= clk AND en;
        shift_stack15_process: PROCESS (gated_clk, clr)
        VARIABLE temp_pile: pile_vect;
  BEGIN
                IF      ( clr = '1' ) THEN
                          pile_out <= "0000000000000000";
                          temp_pile := ( OTHERS => "0000000000000000");
                ELSIF   ((gated_clk = '1') AND (gated_clk'LAST_VALUE =
                      '0') AND (gated_clk'EVENT)) THEN
                                  pile_out <= temp_pile(0);
                                  FOR i IN 0 TO ((stack_depth)-1) LOOP
                                                  temp_pile(i) := temp_pile(i+1);
                                  END LOOP;
                          temp_pile(stack_depth) := pile_in;
                END IF;
END PROCESS shift_stack15_process;
END behv;
```

```
-------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;

ENTITY  decaleur_gauch_droit IS
        PORT(distance : IN  std_logic_vector (5 DOWNTO 0);
                        input    : IN  std_logic_vector (15 DOWNTO 0);
                        output   : OUT std_logic_vector (15 DOWNTO 0));
END decaleur_gauch_droit ;

ARCHITECTURE behv OF decaleur_gauch_droit IS
BEGIN
        decaleur_gauch_droit_process:
        PROCESS (distance,input)
BEGIN

                        CASE distance IS

                                -- decalage vers la droite MSB(distance) = '0' (division)


WHEN "000000" => output <= input;
WHEN "000001" => output <= input(15)& '0' & input(14 DOWNTO 1);
WHEN "000010" => output <= input(15) &"00" & input(14 DOWNTO 2);
WHEN "000011" => output <= input(15) &"000" & input(14 DOWNTO 3);
WHEN "000100" => output <= input(15) &"0000" & input(14 DOWNTO 4);
WHEN "000101" => output <= input(15) &"00000" & input(14 DOWNTO 5);
WHEN "000110" => output <= input(15) &"000000" & input(14 DOWNTO 6);
WHEN "000111" => output <= input(15) &"0000000" & input(14 DOWNTO 7);
WHEN "001000" => output <= input(15) &"00000000" & input(14 DOWNTO 8);
WHEN "001001" => output <= input(15) &"000000000" & input(14 DOWNTO 9);
WHEN "001010" => output <= input(15) &"0000000000" & input(14 DOWNTO 10);
WHEN "001011" => output <= input(15) &"00000000000" & input(14 DOWNTO 11);
WHEN "001100" => output <= input(15) &"000000000000" & input(14 DOWNTO 12);
WHEN "001101" => output <= input(15) &"0000000000000" & input(14 DOWNTO 13);
WHEN "001110" => output <= input(15) &"00000000000000" & input(14 DOWNTO 14);
WHEN "001111" => output <= input(15) &"000000000000000" & input(14);
WHEN "010000" => output <= "0000000000000000";

                                -- decalage vers la gauche MSB(distance) = '1' (multiplication)


WHEN "100000" => output <= input;
WHEN "100001" => output <= input(15) & input(15-2 DOWNTO 0) & '0';
```

```
WHEN "100010" => output <= input(15) & input(15-3 DOWNTO 0) & "00";
WHEN "100011" => output <= input(15) & input(15-4 DOWNTO 0) & "000";
WHEN "100100" => output <= input(15) & input(15-5 DOWNTO 0) & "0000";
WHEN "100101" => output <= input(15) & input(15-6 DOWNTO 0) & "00000";
WHEN "100110" => output <= input(15) & input(15-7 DOWNTO 0) & "000000";
WHEN "100111" => output <= input(15) & input(15-8 DOWNTO 0) & "0000000";
WHEN "101000" => output <= input(15) & input(15-9 DOWNTO 0) & "00000000";
WHEN "101001" => output <= input(15) & input(15-10 DOWNTO 0) & "000000000";
WHEN "101010" => output <= input(15) & input(15-11 DOWNTO 0) & "0000000000";
WHEN "101011" => output <= input(15) & input(15-12 DOWNTO 0) & "00000000000";
WHEN "101100" => output <= input(15) & input(15-13 DOWNTO 0) & "000000000000";
WHEN "101101" => output <= input(15) & input(15-14 DOWNTO 0) & "0000000000000";
WHEN "101110" => output <= input(15) & input(15-15) & "00000000000000";
WHEN "101111" => output <= input(15) & "000000000000000";
WHEN "110000" => output <= "0000000000000000";
WHEN OTHERS => NULL;
END CASE;
END PROCESS decaleur_gauch_droit_process;
END behv;
--------------------------------------------------------------------
```

```
*************************************************************************
************************                    ************************
 ************ MULTIPLIEUR PIPELINÉ ************
************************      Pipeline simple      ************************
*************************************************************************

LIBRARY IEEE ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all ;
USE WORK.pack0.all;
USE work. pack_processeur_syskal .all;

PACKAGE pack22 IS
-------------------------------------------------------------------------
component  multip1_4_4
PORT ( E1        : IN  slv_15_0 ;
     E2        : IN  slv_7_0  ;
     s1, s2, s3 : OUT slv_7_0  ;
     s4, s5, s6 : OUT slv_7_0  ;
     s7, s8     : OUT slv_7_0  ;
     sig       : OUT sl      );
end component;
-------------------------------------------------------------------------
component wallace_tree
PORT ( s1,s2,s3  : IN slv_7_0   ;
     s4,s5,s6  : IN slv_7_0   ;
     s7, s8    : IN slv_7_0   ;
     sig       : IN  sl       ;
     r1        : OUT slv_23_0 ;
     r2        : OUT slv_16_0 ;
     sign      : OUT sl      );
end component ;
-------------------------------------------------------------------------
component  bloc_adit4_4
PORT(A : IN  slv_3_0 ;
    B : IN  slv_3_0;
    s : OUT slv_3_0 ;
    c : OUT sl     );
END COMPONENT ;
-------------------------------------------------------------------------
```

```
component bloc_adit8_8
PORT(A  : IN  slv_7_0 ;
   B  : IN  slv_7_0 ;
   ci : IN  sl      ;
   s  : OUT slv_7_0 ;
   co : OUT sl      );
END COMPONENT ;
--------------------------------------------------------------------------------

END pack22 ;



--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
LIBRARY IEEE ;
use ieee.std_logic_1164.all;
use ieee.std_logic_ARITH.all;
USE WORK.pack0.all         ;
USE work. pack_processeur_syskal.all;

ENTITY multip1_4_4  is
PORT ( E1   :  IN    std_logic_vector (15 downto 0);
          E2   :  IN    std_logic_vector ( 7  downto 0);
 s1, s2, s3,s4 : OUT std_logic_vector (  7 downto 0);
 s5, s6, s7, s8: OUT std_logic_vector (  7 downto 0);
          sig   : OUT std_logic);

END  multip1_4_4;

ARCHITECTURE  behave OF multip1_4_4 IS

signal
  a1, a2 ,a3, a4,  b1, b2 :std_logic_vector(3 downto 0);


BEGIN
a1 <= ( E1(2) , E1(1) ,  E1(0),'0') ;
a2 <= ( E1(6) , E1(5) ,  E1(4),E1(3)) ;
a3 <= ( E1(10), E1(9) ,  E1(8), E1(7)) ;
a4 <= ( E1(14), E1(13),  E1(12),E1(11));
b1 <= ( E2(2) , E2(1) ,  E2(0),'0') ;
b2 <= ( E2(6) , E2(5) ,  E2(4),E2(3)) ;

M1 :multe_4_4  port map (a1 , b1 ,s1);
```

```
M2 :multe_4_4  port map (a2 , b1 ,s2);
M3 :multe_4_4  port map (a3 , b1 ,s3);
M4 :multe_4_4  port map (a4 , b1 ,s4);
M5 :multe_4_4  port map (a1 , b2 ,s5);
M6 :multe_4_4  port map (a2 , b2 ,s6);
M7 :multe_4_4  port map (a3 , b2 ,s7);
M8 :multe_4_4  port map (a4 , b2 ,s8);
sig <= E1(15)XOR E2(7);

end behave;
--------------------------------------------------------------------------
--------------------------------------------------------------------------


LIBRARY IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_ARITH.all;
USE WORK.pack0.all        ;
USE work. pack_processeur_syskal.all;

 ENTITY wallace_tree IS

PORT ( s1, s2, s3, s4, s5, s6, s7, s8 : IN    std_logic_vector(7 downto 0);
         sig                          : IN    std_logic;
         r1                           : Out   std_logic_vector (23 downto 0);
         r2                           : OUT std_logic_vector (16 downto 0);
         sign                         : OUT std_logic);

end  wallace_tree;


ARCHITECTURE behave of wallace_tree IS


signal T0, T1, T2, T3, T8, T9, T10, T11 : std_logic_vector(3  downto 0);
signal T4, T5, T6, T7                   : std_logic_vector(7  downto 0);

signal w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15,w16,w17 :std_logic;
signal c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15, c16, c17 , c18 :std_logic;
signal g5, g6, g7, g8, g9, g10, g11, g12, g13, g14, g15, g16 : std_logic;
signal h6,h7, h8, h9, h10, h11, h12, h13, h14, h15, h16, h17 : std_logic;


BEGIN
```

```
T0 <= (s1(3), s1(2), s1(1), s1(0));
T1 <= (s1(7), s1(6), s1(5), s1(4));
T2 <= (s2(3), s2(2), s2(1), s2(0));
T3 <= (s5(3), s5(2), s5(1), s5(0));
T4 <=  s3;
T5 <= (s4(3), s4(2), s4(1), s4(0), s2(7), s2(6), s2(5), s2(4));
T6 <= (s7(3), s7(2), s7(1), s7(0), s5(7), s5(6), s5(5), s5(4));
T7 <=  s6;
T8 <= (s4(7), s4(6), s4(5), s4(4));
T9 <= (s7(7), s7(6), s7(5), s7(4));
T10<= (s8(3), s8(2), s8(1), s8(0));
T11<= (s8(7), s8(6), s8(5), s8(4));

Wal1  :csfa_wallac_3 port map  ( T1(0), T2(0),T3(0), w1,  c2  );
Wal2  :csfa_wallac_3 port map  ( T1(1), T2(1),T3(1), w2,  c3  );
Wal3  :csfa_wallac_3 port map  ( T1(2), T2(2),T3(2), w3,  c4  );
Wal4  :csfa_wallac_3 port map  ( T1(3), T2(3),T3(3), w4,  c5  );
Wal5  :csfa_wallac_3 port map  ( T7(0), T5(0),T6(0), g5,  h6  );
Wal6  :csfa_wallac_3 port map  ( T7(1), T5(1),T6(1), g6,  h7  );
Wal7  :csfa_wallac_3 port map  ( T7(2), T5(2),T6(2), g7,  h8  );
Wal8  :csfa_wallac_3 port map  ( T7(3), T5(3),T6(3), g8,  h9  );
Wal9  :csfa_wallac_3 port map  ( T7(4), T5(4),T6(4), g9,  h10 );
Wal10 :csfa_wallac_3 port map  ( T7(5), T5(5),T6(5), g10, h11 );
Wal11 :csfa_wallac_3 port map  ( T7(6), T5(6),T6(6), g11, h12 );
Wal12 :csfa_wallac_3 port map  ( T7(7), T5(7),T6(7), g12, h13 );
ad1     :ful_adder       port map  ( T4(0), g5,   w5 ,  c6 );
Wal13 :csfa_wallac_3 port map  ( T4(1), g6,   h6,   w6,  c7 );
Wal14 :csfa_wallac_3 port map  ( T4(2), g7,   h7,   w7,  c8 );
Wal15 :csfa_wallac_3 port map  ( T4(3), g8,   h8,   w8,  c9 );
Wal16 :csfa_wallac_3 port map  ( T4(4), g9,   h9,   w9,  c10 );
Wal17 :csfa_wallac_3 port map  ( T4(5), g10,  h10,  w10, c11 );
Wal18 :csfa_wallac_3 port map  ( T4(6), g11,  h11,  w11, c12 );
Wal19 :csfa_wallac_3 port map  ( T4(7), g12,  h12,  w12, c13 );
Wal20 :csfa_wallac_3 port map  ( T8(0), T9(0),T10(0),g13, h14 );
Wal21 :csfa_wallac_3 port map  ( T8(1), T9(1),T10(1),g14, h15 );
Wal22 :csfa_wallac_3 port map  ( T8(2), T9(2),T10(2),g15, h16 );
Wal23 :csfa_wallac_3 port map  ( T8(3), T9(3),T10(3),g16, h17 );
ad2 :ful_adder      port map  ( g13 , h13 , w13 , c14);
ad3 :ful_adder      port map  ( g14 , h14 , w14 , c15);
ad4 :ful_adder      port map  ( g15 , h15 , w15 , c16);
ad5 :ful_adder      port map  ( g16 , h16 , w16 , c17);
ad6 :ful_adder      port map  ( h17 , T11(0), w17, c18);
```

```
r1 <= (T11(3), T11(2), T11(1), w17, w16, w15, w14, w13, w12, w11, w10, w9,  w8, w7, w6,
w5, w4, w3, w2, w1, T0(3), T0(2), T0(1), T0(0));
r2 <= ( c18, c17, c16, c15, c14, c13, c12, c11, c10, c9, c8, c7, c6, c5, c4, c3, c2);
sign <= sig;
END behave;
```

```
LIBRARY IEEE    ;
USE IEEE.std_logic_1164.all     ;
USE IEEE.std_logic_arith.all    ;
USE WORK.pack0.all        ;
USE work.pack_processeur_syskal.all;


entity bloc_adit4_4 Is

PORT(A : IN  slv_3_0 ;
       B : IN  slv_3_0 ;
        s : OUT slv_3_0 ;
        c : OUT sl    );
end bloc_adit4_4;

ARCHITECTURE behave OF bloc_adit4_4 IS

begin
       process(A,B)
              variable c1,c2,c3 : sl;
       begin
              c1:=  A(0) and B(0);
              c2:= (A(1) and B(1)) or ((A(1) or B(1)) and c1 );
              c3:= (A(2) and B(2)) or ((A(2) or B(2)) and c2 );
              s(0)<= A(0) xor B(0) ;
              s(1)<= A(1) xor B(1) xor c1;
```

```
                s(2)<= A(2) xor B(2) xor c2;
                s(3)<= A(3) xor B(3) xor c3;
                c   <= (A(3) and B(3)) or ((A(3) or B(3)) and c3 );
        end process;
end behave;

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
LIBRARY IEEE    ;
USE IEEE.std_logic_1164.all     ;
USE IEEE.std_logic_arith.all    ;
USE WORK.pack0.all          ;
USE work.pack_processeur_syskal.all;


entity bloc_adit8_8 Is

PORT(A  : IN  slv_7_0 ;
        B  : IN  slv_7_0 ;
        Ci  : IN  sl     ;
        S   : OUT slv_7_0 ;
        Co : OUT sl     );
end bloc_adit8_8;

ARCHITECTURE behave OF bloc_adit8_8 IS

begin
        process(A,B, ci)
                variable c1,c2,c3,c4,c5,c6,c7,c8 : sl;
        begin
                c1:= (A(0) and b(0)) or ((a(0) or b(0)) and ci );
                c2:= (a(1) and b(1)) or ((a(1) or b(1)) and c1 );
                c3:= (a(2) and b(2)) or ((a(2) or b(2)) and c2 );
                c4:= (a(3) and b(3)) or ((a(3) or b(3)) and c3 );
                c5:= (a(4) and b(4)) or ((a(4) or b(4)) and c4 );
                c6:= (a(5) and b(5)) or ((a(5) or b(5)) and c5 );
                c7:= (a(6) and b(6)) or ((a(6) or b(6)) and c6 );

                s(0)<= a(0) xor b(0) xor ci;
                s(1)<= a(1) xor b(1) xor c1;
                s(2)<= a(2) xor b(2) xor c2;
                s(3)<= a(3) xor b(3) xor c3;
                s(4)<= a(4) xor b(4) xor c4;
                s(5)<= a(5) xor b(5) xor c5;
                s(6)<= a(6) xor b(6) xor c6;
```

```
                s(7)<= a(7) xor b(7) xor c7;
                co  <= (a(7) and b(7)) or ((a(7) or b(7)) and c7 );
        end process;
end behave;
```

```
LIBRARY IEEE                    ;
USE ieee.std_logic_1164.all     ;
USE ieee.std_logic_arith.all    ;
USE work.pack0.all              ;
USE work.pack_processeur_syskal.all             ;
USE work.pack22.all             ;


entity mult_pip is
PORT(A1         : IN  slv_15_0 ;
        A2      : IN  slv_7_0  ;
         clk    : in sl;
         Sout   : out slv_15_0    );
end mult_pip ;


ARCHITECTURE fonc OF mult_pip is
signal  s1,s2,s3,s4,s5,s6,s7,s8  : slv_7_0;
signal  s11,s22,s33,s44,s55,s66,s77,s88  : slv_7_0;
signal  sig1,sig11, sig2,sig22,sig3,sig4 : sl;
signal  r1 :slv_23_0;
signal  r2 :slv_16_0;
signal  r11 :slv_23_0;
signal  r22 :slv_16_0;
signal  r3    :slv_14_0;
signal  r4    :slv_12_0;
signal  r31   :slv_7_0:="00000000";
signal  r41   :slv_7_0:="00000000";
signal  sor1:slv_3_0;
signal  sor2,sor22,sor3 :slv_7_0;
```

```
signal  c1,c11,c2,c22,c3 :sl;
signal  z1:sl:='0';
begin

bloc1 : multip1_4_4   port map (A1, A2, s1,s2,s3,s4,s5,s6,s7,s8,sig1);
bloc2 : wallace_tree port map (s11,s22,s33,s44,s55,s66,s77,s88,sig11,r1,r2,sig2);
bloc3 : bloc_adit4_4   port map (r11(8 downto 5),r22(3 downto 0),sor1,c1);
bloc4 : bloc_adit8_8   port map (r3(7 downto 0),r4(7 downto 0),c11,sor2,c2);
bloc5 : bloc_adit8_8   port map (r31,r41,c22,sor3,c3);
Sout <=  sig4 & sor3(6 downto 0) & sor22(7 downto 0);
process(clk,s1,s2,s3,s4,s5,s6,s7,s8,sig1,r1,r2,sig2,r11,r22,c1,sig22,r3,r4,c2,sig3,sor2,sig4)
begin
if clk = '1' then

s11<=s1;s22<=s2;s33<=s3;s44<=s4;s55<=s5;s66<=s6;s77<=s7;s88<=s8;sig11<=sig1;
r11<=r1; r22<=r2;sig22<=sig2;
r3<=r11(23 downto 9);r4<=r22(16 downto 4);c11<=c1;sig3<=sig22;
r31(6 downto 0)<=r3(14 downto 8); r41(4 downto 0)<=r4(12 downto 8);
c22<=c2;sig4<=sig3;sor22<=sor2;
end if;
end process;

end fonc;
```

```
**************************************************************************
************************                    ************************
**********                                            **********
********** ADDITIONNEUR PIPELINÉ **********
**********                                            **********
************************                    ************************
**************************************************************************
```

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;
USE WORK.pack0.all;

PACKAGE pack_additionneur IS

--------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------

COMPONENT bloc_complement_1

Port (A : IN  slv_15_0;
      S : OUT slv_15_0);
END COMPONENT;

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
COMPONENT bloc_addit_4_4

PORT(a : IN   slv_3_0 ;
       b : IN   slv_3_0 ;
        s : OUT  slv_3_0 ;
        c : OUT  sl    );
end COMPONENT;
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
COMPONENT bloc_addit_4_4_comp
PORT(a  : IN   slv_3_0;
       b  : IN   slv_3_0;
       ci : IN   sl    ;
       s  : OUT  slv_3_0;
      co : OUT  sl    );
end COMPONENT;
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
COMPONENT bloc_addit_8_8_comp
PORT(a  : IN   slv_7_0 ;
   b  : IN   slv_7_0 ;
   ci : IN   sl     ;
   s  : OUT  slv_7_0 ;
   co : OUT  sl    );
end COMPONENT;
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

 END pack_additionneur;
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;
```

```
USE WORK.pack0.all        ;
ENTITY bloc_complement_1 IS

PORT (A : IN  slv_15_0;
    S : OUT slv_15_0);
END bloc_complement_1;

ARCHITECTURE behave_blc_complement_1 OF bloc_complement_1 IS

BEGIN
PROCESS(A)
      BEGIN
       IF (A(15)= '0')then
          S<= A;
       elsif (A="1000000000000000")then
          S<= NOT(A(15))&A(14 downto 0);
       else
          S<= A(15)&NOT(A(14 downto 0));
       END IF;

 END PROCESS;
 END behave_blc_complement_1;

-------------------------------------------------------------------------------
-------------------------------------------------------------------------------

LIBRARY IEEE    ;

USE IEEE.std_logic_1164.all     ;
USE IEEE.std_logic_arith.all    ;
USE WORK.pack0.all              ;



ENTITY bloc_addit_4_4 IS


PORT(a : IN  slv_3_0 ;
   b : IN  slv_3_0 ;
   s : OUT slv_3_0 ;
   c : OUT sl     );
end bloc_addit_4_4;

ARCHITECTURE behave OF bloc_addit_4_4 IS
```

```
begin
        process(a, b)
                variable c1,c2,c3 : sl;
        begin
                c1:=  a(0) and b(0);
                c2:= (a(1) and b(1)) or ((a(1) or b(1)) and c1 );
                c3:= (a(2) and b(2)) or ((a(2) or b(2)) and c2 );
                s(0)<= a(0) xor b(0) ;
                s(1)<= a(1) xor b(1) xor c1;
                s(2)<= a(2) xor b(2) xor c2;
                s(3)<= a(3) xor b(3) xor c3;
                c   <= (a(3) and b(3)) or ((a(3) or b(3)) and c3 );
        end process;
end behave;

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

LIBRARY IEEE    ;

USE IEEE.std_logic_1164.all     ;
USE IEEE.std_logic_arith.all    ;
USE WORK.pack0.all        ;



entity bloc_addit_8_8_comp Is

PORT(a  : IN  slv_7_0 ;
   b  : IN  slv_7_0 ;
   ci : IN  sl     ;
   s  : OUT slv_7_0 ;
   co : OUT sl     );
end bloc_addit_8_8_comp;

ARCHITECTURE behave OF bloc_addit_8_8_comp IS

begin
        process(a, b, ci)
                variable c1,c2,c3,c4,c5,c6,c7,c8 : sl;
        begin
                c1:= (a(0) and b(0)) or ((a(0) or b(0)) and ci );
                c2:= (a(1) and b(1)) or ((a(1) or b(1)) and c1 );
                c3:= (a(2) and b(2)) or ((a(2) or b(2)) and c2 );
```

```
                c4:= (a(3) and b(3)) or ((a(3) or b(3)) and c3 );
                c5:= (a(4) and b(4)) or ((a(4) or b(4)) and c4 );
                c6:= (a(5) and b(5)) or ((a(5) or b(5)) and c5 );
                c7:= (a(6) and b(6)) or ((a(6) or b(6)) and c6 );

                s(0)<= a(0) xor b(0) xor ci;
                s(1)<= a(1) xor b(1) xor c1;
                s(2)<= a(2) xor b(2) xor c2;
                s(3)<= a(3) xor b(3) xor c3;
                s(4)<= a(4) xor b(4) xor c4;
                s(5)<= a(5) xor b(5) xor c5;
                s(6)<= a(6) xor b(6) xor c6;
                s(7)<= a(7) xor b(7) xor c7;
                co  <= (a(7) and b(7)) or ((a(7) or b(7)) and c7 );
        end process;
end behave;

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------


LIBRARY IEEE    ;

USE IEEE.std_logic_1164.all     ;
USE IEEE.std_logic_arith.all    ;
USE WORK.pack0.all              ;



ENTITY bloc_addit_4_4_comp IS


PORT(a  : IN  slv_3_0 ;
   b  : IN  slv_3_0 ;
   ci : IN  sl     ;
   s  : OUT slv_3_0 ;
   co  : OUT sl    );
end bloc_addit_4_4_comp;

ARCHITECTURE behave OF bloc_addit_4_4_comp IS

begin
        process(a, b, ci)
                variable c1,c2,c3 : sl;
        begin
```

```
        c1:= (a(0) and b(0)) or ((a(0) or b(0)) and ci );
        c2:= (a(1) and b(1)) or ((a(1) or b(1)) and c1 );
        c3:= (a(2) and b(2)) or ((a(2) or b(2)) and c2 );



        s(0)<= a(0) xor b(0) xor ci;
        s(1)<= a(1) xor b(1) xor c1;
        s(2)<= a(2) xor b(2) xor c2;
        s(3)<= a(3) xor b(3) xor c3;

        co  <= (a(3) and b(3)) or ((a(3) or b(3)) and c3 );
    end process;
end behave;
```

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;
USE work.pack0.all;
USE work.pack_additionneur.all;
```

ENTITY additionneur_pipeliner_16par16 IS

```
    PORT (A   : IN  slv_15_0;
          B   : IN  slv_15_0;
          clk : IN sl      ;
          S   : OUT slv_15_0);
END  additionneur_pipeliner_16par16;
```

```
ARCHITECTURE behave OF additionneur_pipeliner_16par16  IS


 SIGNAL A1,B1     : slv_15_0;
 SIGNAL A11,B11      : slv_11_0;
 SIGNAL S1,S11 : slv_3_0;
 SIGNAL S2 :slv_7_0;
 SIGNAL S3  : slv_15_0;
 SIGNAL A111,B111:slv_3_0;

 SIGNAL C1,C11,C2,C22,C3  : sl;

  BEGIN
       bloc1 : bloc_complement_1   port map (A,A1);
       bloc2 : bloc_complement_1   port map (B,B1);
       bloc3 : bloc_addit_4_4      port map (A1(3 downto 0), B1(3 downto 0),S1,C1);

       bloc4 : bloc_addit_8_8_comp port map (A11(7 downto 0),B11(7 downto
0),C11,S2,C2);

       bloc5 : bloc_addit_4_4_comp port map (A111,B111,C22,S3(15 downto 12),C3);
       bloc6 : bloc_complement_1   port map (S3, S);

   process(clk,A,B,A1,A11,A111,B1,B11,B111,C11,C22)
       begin
              if clk'event and clk ='1' then
                     A11<=A1(15 downto 4);
                     B11<=B1(15 downto 4);
                     S11<=S1;
                     C11<=C1;

                     A111<=A11(11 downto 8);
                     B111<=B11(11 downto 8);
                     C22<=C2;
                     S3(3 downto 0)<=S11;
                     S3(11 downto 4)<=S2;

              end if;
     end process;
 END behave;
```

```
**************************************************************************
**********                                                  **********
**********                                                  **********

********** MULTIPLIEUR AVEC PIPELINE PAR VAGUE *********
**********                                                  **********

**********                                                  **********
**************************************************************************


LIBRARY IEEE ;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;


PACKAGE pack1_port_elemet IS



-------------------------------------------------------
-------------------------------------------------------
COMPONENT  porte_and
        port (          A       : in  std_logic  ;
          B       : in  std_logic  ;
          S       : out std_logic );
END component ;
-------------------------------------------------------
-------------------------------------------------------
COMPONENT  porte_xor

        port (          A       : in  std_logic  ;
          B       : in  std_logic  ;
          S       : out std_logic );
END component ;
-------------------------------------------------------
-------------------------------------------------------
COMPONENT  porte_carry
        port (          A       : in  std_logic  ;
          B       : in  std_logic  ;
          C       : in  std_logic  ;
                                  S       : out std_logic );
```

```
END component ;
-------------------------------------------------------
-------------------------------------------------------
COMPONENT  porte_sum
        port (          A       : in  std_logic  ;
          B       : in  std_logic  ;
          C       : in  std_logic  ;
                                          S       : out std_logic );
END component ;
-------------------------------------------------------
-------------------------------------------------------
COMPONENT  porte_delai
        port (          A       : in  std_logic  ;
          S       : out std_logic );
END component ;
-------------------------------------------------------
-------------------------------------------------------
COMPONENT  porte_delai2

        port (          A       : in  std_logic  ;
          S       : out std_logic );
END component ;
-------------------------------------------------------
-------------------------------------------------------

end pack1_port_elemet;

-------------------------------------------------------
-------------------------------------------------------

LIBRARY IEEE;
USE ieee.std_logic_1164.all;

ENTITY  porte_and IS
        port (          A       : in  std_logic  ;
          B       : in  std_logic  ;
          S       : out std_logic );
END porte_and ;

ARCHITECTURE behave of porte_and is
begin
S<= A and B after 2.5 ns;
end behave;
-------------------------------------------------------
-------------------------------------------------------
```

```
LIBRARY IEEE;
USE ieee.std_logic_1164.all;


ENTITY  porte_xor IS
        port (          A       : in  std_logic  ;
          B       : in  std_logic  ;
          S       : out std_logic );
END porte_xor ;

ARCHITECTURE behave of porte_xor is
begin
S <= A xor B after 2.5 ns;
end behave;
-------------------------------------------------------------
-------------------------------------------------------------

LIBRARY IEEE;
USE ieee.std_logic_1164.all;


ENTITY  porte_carry IS
        port (          A       : in  std_logic  ;
          B       : in  std_logic  ;
          C       : in  std_logic  ;
                                        S       : out std_logic );
END porte_carry;

ARCHITECTURE behave of porte_carry is
begin
S <= ((not(A) and C) or (A and B))   after 2.5 ns;
end behave;
-------------------------------------------------------------
-------------------------------------------------------------

LIBRARY IEEE;
USE ieee.std_logic_1164.all;


ENTITY  porte_sum IS
        port (          A       : in  std_logic  ;
          B       : in  std_logic  ;
          C       : in  std_logic  ;
                                        S       : out std_logic );
```

```
END porte_sum;

ARCHITECTURE behave of porte_sum is
begin
S <= (A xor B)xor C    after 2.5 ns;
end behave;

--------------------------------------------------------------------
--------------------------------------------------------------------

LIBRARY IEEE;
USE ieee.std_logic_1164.all;


ENTITY  porte_delai IS
        port (          A      : in  std_logic  ;
          S      : out std_logic );
END porte_delai;

ARCHITECTURE behave of porte_delai is
begin
S <= A    after 2.5 ns;
end behave;
--------------------------------------------------------------------
--------------------------------------------------------------------
LIBRARY IEEE;
USE ieee.std_logic_1164.all;


ENTITY  porte_delai2 IS
        port (          A      : in  std_logic  ;
          S      : out std_logic );
END porte_delai2;

ARCHITECTURE behave of porte_delai2 is
signal C :std_logic;
begin
S <= A    after 2*2.5 ns;
end behave;
--------------------------------------------------------------------
--------------------------------------------------------------------


LIBRARY IEEE ;
USE IEEE.std_logic_1164.all;
```

```
USE work.pack1_port_elemet.all;

PACKAGE pack2_port_elemet IS

------------------------------------------------------
------------------------------------------------------

COMPONENT  porte_half_adder

        port (          A      : in  std_logic  ;
          B      : in  std_logic  ;
          S      : out std_logic  ;
                                    C      : out std_logic );


END component ;
------------------------------------------------------
------------------------------------------------------
COMPONENT  porte_Proc_elem

        port (          A      : in  std_logic  ;
          B      : in  std_logic  ;
          C      : in  std_logic  ;
                                    D      : in  std_logic  ;
                                    X      : out std_logic  ;
                                    S      : out std_logic  ;
                                    R      : out std_logic );


END component ;

------------------------------------------------------
------------------------------------------------------
        COMPONENT complem_1
        PORT (Ain  : IN  std_logic_vector (15 downto 0);
                     Aout : OUT std_logic_vector (15 downto 0));
        END COMPONENT ;
------------------------------------------------------
------------------------------------------------------

END pack2_port_elemet;
------------------------------------------------------
```

```
----------------------------------------------------------

LIBRARY IEEE ;
USE IEEE.std_logic_1164.all;
USE work.pack1_port_elemet.all;

ENTITY  porte_half_adder IS
port (          A      : in  std_logic  ;
        B      : in  std_logic  ;
        S      : out std_logic  ;
                                    C      : out std_logic );
END porte_half_adder ;

ARCHITECTURE behave of porte_half_adder is

begin

bloc1 : porte_xor  port map (A, B, S);
bloc2 : porte_and  port map (A, B, C);

end behave;



----------------------------------------------------------
----------------------------------------------------------

LIBRARY IEEE ;
USE IEEE.std_logic_1164.all;
USE work.pack1_port_elemet.all;

ENTITY  porte_Proc_elem IS
port (          A      : in  std_logic  ;
        B      : in  std_logic  ;
        C      : in  std_logic  ;
                                    D      : in  std_logic  ;
                                    X      : out std_logic  ;
                                    S      : out std_logic  ;
                                    R      : out std_logic );
END porte_Proc_elem ;


ARCHITECTURE behave of porte_Proc_elem IS

signal a1, ab, cxord, d1 : std_logic ;
```

```
begin

bloc1  : porte_delai port map (A, a1);
bloc2  : porte_delai port map (a1, X);
bloc3  : porte_and   port map (A, B, ab);
bloc4  : porte_xor   port map (C, D, cxord);
bloc5  : porte_xor   port map (cxord, ab, S);
bloc6  : porte_delai port map (D,d1);
bloc7  : porte_carry port map (cxord ,ab, d1,R);

end behave;

-- ==========================================================

        LIBRARY IEEE;
        USE ieee.std_logic_1164.all;
        USE work.pack1_port_elemet.all;

        ENTITY complem_1 IS
        PORT (Ain  : IN   std_logic_vector (15 downto 0);
                    Aout : OUT  std_logic_vector (15 downto 0));
        END  complem_1 ;

        ARCHITECTURE fonct of complem_1 IS

        BEGIN

        label1 : for i in 0 to 14 generate
                bloc1 : porte_xor   port map (Ain(15),Ain(i),Aout(i));
        END generate;
                bloc2 : porte_delai port map ( Ain(15), Aout(15));

        END fonct;



**********************************************************

LIBRARY IEEE ;
USE IEEE.std_logic_1164.all;
USE work.pack1_port_elemet.all;

PACKAGE pack_mult_pip_vague IS
```

```
-- ================================================
        COMPONENT multi_pip_vague
        PORT(e1 : IN  std_logic_vector (15 downto 0);
           e2 : IN  std_logic_vector (7  downto 0);
           sort : OUT std_logic_vector (15 downto 0));
        END COMPONENT;
-- ================================================


END pack_mult_pip_vague;


-- ================================================
        LIBRARY IEEE ;
        USE IEEE.std_logic_1164.all;
        USE work.pack1_port_elemet.all;
  USE work.pack2_port_elemet.all;
        USE work.pack1_mult.all;

        ENTITY multi_pip_vague IS
        PORT(e1   : IN  std_logic_vector (15 downto 0);
                  e2   : IN  std_logic_vector (7  downto 0);
                  sort : OUT std_logic_vector (15 downto 0));
        END multi_pip_vague;

        ARCHITECTURE fonct of  multi_pip_vague IS

        SIGNAL Aout1,Aout2,Aout3       :std_logic_vector(14 downto 0);
        SIGNAL Aout4,Aout5,Aout6,Aout7  :std_logic_vector(14 downto 0);
        SIGNAL S3, S1,S2,C1,C2,C3,C4,C5 :std_logic_vector(14 downto 0);
        SIGNAL S4,R1,R2,R3,R4,R5       :std_logic_vector(13 downto 0);
        SIGNAL eint                    :std_logic_vector(6 downto 0) ;

        begin
        bloc0 : bloc_delai_multiplieur  port map          (e2(6 downto 0), eint);
        bloc1 : bloc1_and port map (e1(14 downto 0),eint(1 downto 0),Aout1,S1,S2);
        bloc2 : bloc2_halfe_ader port map (Aout1,S1(14 downto 1), S2,Aout2,S3,S4);
        bloc3 : bloci_PE port map  (Aout2,eint(2),S3(14 downto 1),S4,Aout3,C1,R1);
        bloc4 : bloci_PE port map  (Aout3,eint(3),C1(14 downto 1),R1,Aout4,C2,R2);
        bloc5 : bloci_PE port map  (Aout4,eint(4),C2(14 downto 1),R2,Aout5,C3,R3);
        bloc6 : bloci_PE port map  (Aout5,eint(5),C3(14 downto 1),R3,Aout6,C4,R4);
        bloc7 : bloci_PE port map  (Aout6,eint(6),C4(14 downto 1),R4,Aout7,C5,R5);
        bloc8 : bloc_additioneur_final  port map       (C5,R5,sort(14 downto 0));
        bloc9 : bloc_sign1              port map         (e1(15),e2(7),sort(15));
```

```
        END fonct;


**********************************************************************
**********************                            **********************
***************                                          ***************

************ ADDITIONNEUR AVEC PIPELINE PAR VAGUE ********

***************                                           **************

************************                        ************************
**********************************************************************


LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

PACKAGE  addit_pip_vague IS

--
=======================================================================
==      COMPONENT addi_vague
        PORT (A  : IN  std_logic_vector (15 downto 0);
                       B  : IN  std_logic_vector (15 downto 0);
                       Sout : OUT std_logic_vector (15 downto 0));
        END COMPONENT ;
--
=======================================================================
==
END addit_pip_vague;

--
=======================================================================
==

        LIBRARY IEEE;
        USE ieee.std_logic_1164.all;
        USE work.pack1_port_elemet.all;
        USE work.pack2_port_elemet.all;
        USE work.pack1_mult.all;
```

```
        ENTITY  addi_vague IS
        PORT (A  : IN   std_logic_vector (15 downto 0);
                        B  : IN   std_logic_vector (15 downto 0);
                        Sout : OUT  std_logic_vector (15 downto 0));
        END  addi_vague  ;

        ARCHITECTURE fonct of  addi_vague IS

        Signal Aout,Bout: std_logic_vector (15 downto 0);
  Signal S : std_logic_vector (16 downto 0);

        BEGIN


                bloc1 : complem_1 port map ( A , Aout );
                bloc2 : complem_1 port map ( B , Bout );
                bloc3 : bloc_additioneur_final port map (Aout(15 downto 1),Bout(15 downto
2),S(16 downto 2) );
                bloc4 :  complem_1 port map (S(15 downto 0), Sout);

        END fonct;
--
==========================================================================
==
```

ANNEXE B

Programmes sources de l'architectures : SYSKAL dans MATLAB

```
%==========================================================================
%==========================================================================
%
        Ce fichier est le fichier source pour la préparation des données utiliser pour simuler
        le processeur SYSKAL, il faut noter que ce programme fait la reconstitution en
        virgule flottante et aussi en virgule fixe.
%
%==========================================================================
%==========================================================================

clear

tic

%============================
%  Definition des variables
%============================


Nb_pt_x = 128;              % Nombre de points de x
Nb_pt_h = Nb_pt_x/2;        % Nombre de points du transoperateur
sigma_b = 0;                % Deviation standard du bruit
poid = 1;                   %
Nb_pt = Nb_pt_h;            % Nombre max d'iterations pour le calcul du gain
Q_R = 1e1;                  % Q_R si h multiplie par 1
ordre = 1;                  % F(1,1)
LB=2;                       %
%retard = 1;                % Element 1 du vecteur d'état á extraire
retard = 1; %Nb_pt_h/2;     % Dernier element du vecteur d'état á extraire
echelonnement_h = 'oui';    % Indique si oui ou non le vecteur h
                            % sera re-echelonneé
echelonnement_K = 'oui';    % Indique si oui ou non le vecteur K
                            % sera re-echelonneé
ampli_mesure = 'oui';       % Indique si le signal de mesure sera reechelonne
C0 = 1;                     % Parametre de la contrainte de positivite
Nb_bit_z = 15;              %
Nb_bit_h = 15;              %
Nb_bit_k = Nb_bit_h;        %
Nb_bit_y = Nb_bit_z;        %
```

```
figure(1)

%==================================
%       Creation des signaux
%==================================

dt=1;

[x,h,hh,y,yb,tx,SNR] = rkb2bfct(dt,LB,sigma_b,poid);       %dt = tx(2)-tx(1);

yb(length(yb)+1:length(yb)+length(h)/2)=zeros(length(h)/2,1);

if ampli_mesure == 'oui'
        gain_m = 0.95/max(yb);
        yb_A = yb * gain_m;
  else
        gain_m = 1;
end

disp('Convolution terminee et bruit ajoute')

%yr = zeros(Nb_pt_x,1);



%==================================
% echelonnement du transoperateur
%==================================


if echelonnement_h == 'oui'
        disp('Echelonnement du transoperateur')
        gain_h =  0.95/ max(h);
        h_A = h * gain_h;
  else
        gain_h = 1;
end



%================
% calcul du gain
%================
```

```
disp('Calcul du gain en cours')
K_kal = gain_k(tx,h_A,Nb_pt,Q_R);



%==============================
% echelonnement du transoperateur
%==============================


if echelonnement_K == 'oui'
        disp('Reechelonnement du vecteur K')
        %  Reechelonnement du gain K

        gain_K = 1 / max(K_kal);
      gain_K = 2^(floor(log10(1/max(K_kal))/log10(2)));

        K_kal_A = K_kal * gain_K;
  else
        gain_K = 1;
end


%================================
%  Deconvolution point-flottant
%================================

disp('Deconvolution point-flottant')

        %xr_A = kalbin2_c(yb_A,h_A,K_kal_A,retard,1,inf,gain_K);
        %xr = xr_A*gain_h/(gain_m*gain_K*dt);
        %Exr = erelmoy(x, xr);

xrc_A = kalbin2_c(yb_A,h_A,K_kal_A,retard,C0,inf,gain_K);
xrc = xrc_A*gain_h/(gain_m*gain_K*dt);
Exrc = erelmoy(x, xrc);

%plot(tx,x,tx,xrc,tx,yb)


%============================
%  Deconvolution point-fixe
%============================

disp('Deconvolution point-fixe')
```

```
En(length(Nb_bit_z),1) = 0;
Ecn(length(Nb_bit_z),1) = 0;

for nh = 1:length(Nb_bit_h)

  for nz = 1:length(Nb_bit_z)

    C0n(nz) = digital(C0, Nb_bit_z(nz), 1);

    %==============================
    % Numerisation des vecteurs yb
    %==============================

    ybn_A = digital(yb_A, Nb_bit_y(nz), 1);

    %====================================
    % Numerisation des vecteurs k, et h
    %====================================

        hn_A = digital(h_A, Nb_bit_h(nh), 1);
        K_kaln_A = digital(K_kal_A, Nb_bit_k(nh), 1);

        %xrn_A = kalbin2_c(ybn_A,hn_A,K_kaln_A,retard,1,Nb_bit_z(nz),gain_K);
        %xrn = xrn_A*gain_h/(gain_m*gain_K*dt);
        %EXrn(nz,nh) = erelmoy(x, xrn);
        %En(nz,nh) = erelmoy(xr, xrn);

        [xrcn_A,yen,in] = kalbin2_c(ybn_A,hn_A,K_kaln_A,retard,C0n(nz),Nb_bit_z(nz),gain_K);
        xrcn = xrcn_A*gain_h/(gain_m*gain_K*dt);
        EXrcn(nz,nh) = erelmoy(x, xrcn);
        Ecn(nz,nh) = erelmoy(xrc, xrcn);

        disp(' ')
        %       [ SNR Nb_bit_h(nh) Nb_bit_z(nz) ]
        %       [ Exr Exrc EXrn(nz,nh) EXrcn(nz,nh) En(nz,nh) Ecn(nz,nh) ]



%============================
%   Tracer les resultats
%============================
```

```
%plot(tx,x,tx,xr,tx,xrn)
%figure(2)
plot(tx,x,tx,xrc,tx,xrcn)

semilogy(Nb_bit_z,En)
semilogy(Nb_bit_z,Ecn)


end

end

Tcal = toc



%================================================
%  Sauver les resultats qui seront utiliser lors
%  des preparation des fichiers des simulation d
%  u processeur SYSKAL et PIPEKAL2
%================================================

vectnul_10 = [0 0 0 0 0 0 0 0 0 0];
vectnul_34 = [vectnul_10';vectnul_10';vectnul_10';0;0;0;0];
ybn_A = [ybn_A;vectnul_34];


%save donne_kal3.mat  K_kaln_A   gain_K    hn_A     xrcn_A    ybn_A  yen  in



%================================================================
%================================================================
%
%                          FIN DU PROGRAMME
%
%================================================================
%================================================================
```