UNIVERSITÉ DU QUÉBEC

MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INGÉNIERIE

PAR

JEAN-JACQUES CLAR

DÉVELOPPEMENT D'APPLICATIONS PARALLÈLES POUR UN SYSTÈME MULTIPROCESSEUR EXPÉRIMENTAL

Juillet 2002

Bibliothèque
Paul-Émile-Boulet

UQAC
Université du Québec à Chicoutimi

Mise en garde/Advice

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

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

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

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

Nous ne devrions pas croire une chose uniquement parce qu'elle a été dite,
ni croire aux traditions parce qu'elles ont été transmises depuis
l'Antiquité; ni aux "on dit" en tant que tels;
ni aux écrits des sages parce que se sont des sages qui les ont écrits;
ni aux imaginations que nous supposons avoir été inspirées par un être spirituel;
ni aux déductions tirées de quelque hypothèse hasardeuse que nous aurions pu faire;
ni à ce qui paraît être une nécessité analogique;
ni croire sur la simple autorité de nos instructeurs ou de nos maîtres.

Mais nous devons croire à un écrit,
à une doctrine ou
à une affirmation lorsque notre raison et notre expérience intime
les confirment.
C'est pourquoi je vous ai enseigné à ne pas croire simplement
d'après ce qui vous a été dit,
mais conformément à votre expérience personnelle,
et puis à agir en conséquence et généreusement.

Bouddha

Résumé

Un souci constant qui guide le développement de l'informatique est l'accélération des performances. Dans cette optique, une des solutions souvent utilisées réside en la mise en parallèle des traitements.

L'émergence, depuis quelques années, des applications multimédia et l'augmentation incessante de la complexité des systèmes pouvant être intégrés sur un circuit motive et rend possible l'apparition et le développement de DSP[1] de plus en plus complets, dédiés à certains traitements numériques intensifs. Les applications cibles traitent généralement une énorme quantité de données avec un nombre limité de fonctions. Ces traitements, souvent indépendants les uns des autres, peuvent être effectués en parallèle. Le but est d'exploiter le parallélisme dans les données de plusieurs algorithmes afin de les traiter nettement plus rapidement qu'avec un ordinateur séquentiel conventionnel.

Pour obtenir des applications à traitement parallèle performantes il faut effectuer une partition des algorithmes étudiés en assignant une partie du traitement à chacun des processeurs.

---

[1]DSP: **D**igital **S**ignal **P**rocesseurs : Processeurs de Traitement de Signaux

Le projet PULSE, issu d'un groupe de recherche de l'École Polytechnique de Montréal, travaille au développement d'une architecture multiprocesseur de type SIMD[2] dédiée au traitement numérique en temps réel.

Le circuit intégré cible possède quatre processeurs dans sa première version. La seconde version doit inclure seize processeurs à l'intérieur du même circuit intégré.

Durant ce projet, sept applications numériques connues ont étés écrites en langage assembleur et optimisées sur la première version de **PULSE**. Les applications sont : multiplications vecteurs-matrice, algorithmes de cryptage RSA et IDEA, algorithme de Bresenham, modèle continu (poisson) et transformation binaire d'images (inclut érosion et dilatation d'images). Pour certaines de ces applications - multiplications vecteurs-matrice, algorithme de Bresenham et transformation binaire d'images – les résultats obtenus se comparent avantageusement avec les librairies déjà développées sur des circuits intégrés concurrents.

En plus des applications numériques ce projet à permis d'améliorer les différents outils de **PULSE** – documentation et simulateur –, ainsi que le modèle VHDL[3].

---

[2]SIMD: **S**ingle **I**nstruction **M**ultiple **D**ata

[3]VHDL: **V**HSIC (Very High Speed Integrated Circuits) **H**ardware **D**escription **L**anguage

Remerciements

Bon cette page sert à remercier tous les gens m'ayant enrichi de leurs précieuses aides. Ce projet et le mémoire qui si rattache ont étés complétés en 2 parties.

La première partie s'échelonne sur 2 années passées à compléter les cours et le travail de recherche à l'UQAC pour tenter d'accéder au titre de maître en ingénierie. Il s'agit de membres du GRM : Yvon Savaria pour sa passion, ses connaissances et naturellement ses dollars, pour leurs indispensables aides Normand Bélanger, Nicolas Contandriopoulos, Ivan Kraljic , Paul Marriott et Claude Villeneuve.

La seconde partie couvre le temps de rédaction. C'est avec plaisir que je présente de profonds remerciements aux personnes que j'ai emmerdées par mon indisponibilité durant les trois dernières années pour rédiger ce mémoire. Tout d'abord cette merveilleuse famille qui m'entoure et me suivrait au bout de la galaxie si cela peut aider l'accomplissement de notre destin, Caterpillar et Brian Funke pour les nuits passées sur mon portable au lieu de partager une bonne bière dans le hot tub à Tucson, Novell et Brian Misbach pour l'opportunité d'utiliser et de pousser mes connaissances sur les systèmes multiprocesseurs. Sonia, je te remercie pour avoir corrigé ma première version et aussi pour m'avoir dit que mon français était récupérable, peut-être…

Daniel Audet conservera toujours une place spéciale pour m'avoir fait confiance en me proposant ce projet, pour son aide et sa non-ingérence dans mon travail. Pour la rédaction de ce mémoire faisait-il preuve d'une patience sans limite, avait-il lancé la serviette ou tout bonnement il était trop occupé à faire des choses plus importantes que de me taper sur les doigts ?

Table des Matières

Résumé .... *ii*

Remerciements .... *iv*

Table des Matières .... *vi*

Liste des encadrés .... *x*

Liste des tableaux .... *xi*

Liste des figures .... *xiii*

Liste des Annexes .... *xv*

Chapitre 1 – INTRODUCTION GÉNÉRALE *1*

1.1. Préambule **2**

1.1.1. Le Multimédia 2

1.2. Introduction au parallélisme **4**

1.2.1. La Classification des Différents Types de Machines 5

1.2.2. L'évolution de la quincaillerie 7

1.3. Projet de deuxième cycle **9**

1.4. Le projet PULSE **10**

1.4.1. Le Groupe 11

1.4.2. Interaction et Utilisation des Outils de Développement 12

1.4.3. Caractéristiques Générales de PULSE 14

1.4.4. V1 versus V2 25

1.5. Contribution du présent projet **26**

1.5.1. Méthodologie du Projet 26

1.6. Présentation de la thèse **27**

Chapitre 2 – REVUE DES SYSTÈMES EXISTANTS ***29***

2.1. Préambule **30**

2.2. Introduction **31**

2.2.1. Description générale 31

2.3. Architecture **35**

2.3.1. CNAPS 35

2.3.2. 'C80 37

2.3.3. SHARC 40

2.3.4. A236 42

2.4. Environnement de développement **44**

2.4.1. CNAPS 44

2.4.2. 'C80 45

2.4.3. SHARC 45

2.4.4. A236 45

2.5. Conclusion 47

Chapitre 3 – DÉVELOPPEMENT D'APPLICATIONS 49

3.1. Préambule 50

3.1.1. Objectifs de développement 50

3.1.2. Méthodologies adoptées 55

3.1.3. Environnement de développement 58

3.1.4. Évaluation de la performance, du débit et de l'efficacité 59

3.1.5. Goulot d'étranglement 60

3.2. Applications développées 62

3.2.1. Multiplication vecteurs - matrice 62

3.2.2. Algorithme de Bresenham 82

3.2.3. Algorithme RSA 89

3.2.4. Équations aux différences finies (Poisson) 93

3.2.5. Morphologie Binaire : Érosion et Dilatation 98

3.2.6. IDEA 104

3.2.7. Constat 112

Chapitre 4 – ÉVALUATION DE LA PERFORMANCE 113

4.1. Préambule 114

4.2. Présentation des résultats **115**

4.2.1. Paramètres et mesures de développement 115

4.2.2. Banc d'essai 115

4.2.3. Résultats 119

4.3. Conclusion **130**

Chapitre 5 - CONCLUSION ***131***

5.1. Dévelopement D'Applications **133**

5.1.1. Développement sur PULSE 134

5.1.2. Test, Évaluation et Recommandation 136

5.2. Conclusion **142**

Bibliographie ***143***

Annexe A - APPLICATIONS DÉVELOPPÉES ***147***

Annexe B – OPÉRATION MODULO RAPIDE ***197***

Liste des encadrés

Encadré 3.1 : Opérations : multiplication vecteur - matrice 64

Encadré 3.2 : Multiplication vecteurs-matrice, version de base 68

Encadré 3.3 : Multiplication vecteurs-matrice, données dans la mémoire interne 69

Encadré 3.4 : Multiplication vecteur-matrice, version pseudo-systolique 72

Encadré 3.5 : Multiplication vecteurs-matrice, version pseudo-systolique 79

Encadré 3.6 : Opérations – utilisation nouvelle combinaison instruction parallèle 80

Encadré 3.7 : Bresenham - Pseudo-code 84

Encadré 3.8 : Bresenham - Opérations – traitement d'un segment de droite 86

Encadré 3.9 : Bresenham - Opérations : calcul position pixel 87

Encadré 3.10 : Poisson - Opérations : équation aux différences finies 95

Encadré 3.11 : Morphologie - Opérations - morphologie binaire 102

Encadré 3.12 : Opérations de cryptage avec IDEA 105

Liste des tableaux

Tableau 1.1 : Classification de Flynn 5

Tableau 1.2 : Latence - Instructions d'exceptions 17

Tableau 3.1 : Vecteurs – Matrice - Organisation des données, mémoire d'entrée 66

Tableau 3.2 : Vecteurs – Matrice - Traitement des données dans chaque UT 67

Tableau 3.3 : Vecteurs – Matrice - Traitement des données dans chaque PE 70

Tableau 3.4 : Vecteurs – Matrice - Nombre de cycles par vecteur 74

Tableau 3.5 : Vecteurs – Matrice - Nombre de cycles par vecteur 81

Tableau 3.6 : Cryptage avec IDEA – Opérations récursives 109

Tableau 3.7 : Cryptage avec IDEA – Opérations finales 110

Tableau 4.1 : Présentation des Résultats 119

Tableau 4.2 : Goulots d'étranglements 120

Tableau 4.3 : Multiplication vecteurs – matrice – traitement de 200 vecteurs 122

Tableau 4.4 : Bresenham – traitement de 50 vecteurs de 10 pixels 123

Tableau 4.5 : Cryptage -Performance – traitement de 10^6 caractères 123

Tableau 4.6 : Poisson, simulation d'une maille de 32 x 32 = 1024 PEs 123

Tableau 4.7 : Morphologie Mathématique - Performance (128 x 128 image).................. 124

Tableau 4.8 : Morphologie Mathématique - Performance (256 x 256 image).................. 124

Tableau 4.9 : Relation entre calcul et efficacité 125

Tableau 4.10 : Facteur d'accélération Sp et Efficacité Ep 126

Tableau 4.11 : Multiplication vecteurs – matrice, nombre de vecteurs traités 127

Tableau 4.12 : Bresenham - Comparaison avec *UWICL* (50 segments de 10 pixels) 128

Tableau 4.13 : RSA – Comparaison (10^6 caractères de 8 bits) 129

Tableau 4.14 : Morphologie Binaire - Comparaison avec *UWICL* (512 x 512 images).... 129

Tableau 5.1 : Liste des recommandations et limitations rencontrées................................ 137

Liste des figures

Figure 1.1 : Modèle SIMD avec Mémoires Distribuées 7

Figure 1.2 : Membres du groupe PULSE 11

Figure 1.3 : PULSE - Environnement d'Opération - exemple 14

Figure 1.4 : Caractéristiques générales de l'architecture de PULSE 21

Figure 1.5 : PULSE - Entrées et sorties des données 22

Figure 1.6 : PULSE - Canaux de communications et traitement des données 23

Figure 1.7 : PULSE - Élément de calcul et étapes d'exécution 24

Figure 2.1 : Sous-système CNAPS 32

Figure 2.2 : CSC et CNAPS-10xx avec connexions 36

Figure 2.3 : Bloc diagramme du 'C80 38

Figure 2.4 : Bloc diagramme du SHARC 41

Figure 2.5 : Bloc diagramme du A236 43

Figure 3.1 : Pipeline de PULSE 53

Figure 3.2 : Décomposition d'un algorithme 57

Figure 3.3 : Vecteurs – Matrice - Mouvement de données et opérations 71

Figure 3.4 : Vecteurs – Matrice - Remplissage du pipeline et sortie des résultats 75

Figure 3.5 : Entrée des données, opérations et sortie des résultats 77

Figure 3.6 : Vecteurs – Matrice - Décomposition basée sur l'architecture 78

Figure 3.7 : Bresenham - Traçage de lignes.. 83

Figure 3.8 : RSA - Authentification et Cryptage de message 90

Figure 3.9 : Poisson - Voisinage nodal 94

Figure 3.10 : Poisson - Correspondance Maille – PE 96

Figure 3.11 : Morphologie - Correspondance Pixel – PE 99

Figure 3.12 : Morphologie - Colonne supplémentaire de 0 (zéro)....................................... 99

Figure 3.13 : Morphologie - Mouvement de données et calcul 101

Figure 3.14 : IDEA - Création des clés dérivées... 105

Liste des Annexes

Annexe A - APPLICATIONS DÉVELOPPÉES 147

Annexe B – OPÉRATION MODULO RAPIDE 197

CHAPITRE 1

INTRODUCTION GÉNÉRALE

1.1. PRÉAMBULE

1.1.1. Le Multimédia

À ce jour, le développement d'applications multimédias pour commercialisation présente de sérieuses barrières pour les concepteurs et les utilisateurs. Les CPUs[4] d'aujourd'hui (mais peut-être pas ceux de demain) font preuve de lacune au point de vue performance pour traiter de façon simultanée le graphisme, le son, la vidéo et les communications en temps réel. L'objectif étant de produire avec le plus de réalisme possible les applications multimédias.

Les applications multimédias de haute qualité, surtout les jeux, nécessitent une énorme puissance de traitement. Par exemple, un simple microprocesseur standard est incapable de façon simultanée d'interpréter des images en 3-D, de générer du son de qualité CD, de reconnaître la parole et de détecter les mouvements sur un affichage plein écran le tout de façon synchrone. Chacun de ces processus accapare une partie de la puissance de traitement du processeur maître, diminuant du même coup le pourcentage de la puissance totale disponible pour les autres tâches. Le tout se continue jusqu'à atteindre 100% de la puissance de traitement des composantes en places. Chacune des tâches initiées subséquemment gruge sur les tâches déjà en cours créant un ralentissement perceptible par l'utilisateur ; images saccadées, sorties sonores interrompues, décalage entre les différentes tâches, ...

---

[4] Central Processing Unit

Si toutes ces opérations peuvent être aujourd'hui, avec un niveau de performance raisonnable, traitées dans un seul système multimédia, c'est grâce à un support externe au processeur principal.

<u>Comment cela se fait-il ?</u>

Différents microprocesseurs, avec de multiples DSPs incorporés sur des cartes, s'occupent de mener à bien certaines des tâches de façon individuelle afin d'appuyer le processeur maître. Ces cartes sont conçues pour un groupe de tâches spécifiques souvent très coûteuses en temps processeur, ce qui résulte en une configuration optimale sans se soucier de généralisation comme le processeur principal. Le processeur maître peut déléguer une partie des tâches en cours et concentrer sa puissance sur les tâches restantes.

Plusieurs entreprises dont les noms sont aujourd'hui renommés, se partagent ce marché qui s'élève à plus de *1 milliard de dollars US* par année.

Un des critères principaux pour pouvoir pénétrer ce marché et y survivre, est le degré d'accélération des performances afin d'augmenter la rapidité de traitement des applications. Dans cette optique, une des solutions matérielles proposées réside dans le traitement parallèle des applications.

1.2. INTRODUCTION AU PARALLÉLISME

Les premiers ordinateurs furent tous des machines séquentielles. L'architecture de ces ordinateurs séquentiels s'inspire du modèle de von Neumann. Depuis l'avènement du premier ordinateur, les machines de type von Neumann ont vu leur puissance de calcul considérablement augmenter, sans pour autant pouvoir satisfaire la demande. L'apparition du parallélisme, dans les processeurs d'abord et ensuite dans les ordinateurs, est liée à deux principes.

1. **La puissance de calcul.** La course à la puissance de calcul est la première raison de l'apparition du parallélisme. Quelle que soit la vitesse à laquelle la puissance des processeurs augmente, il existe toujours une limite courante de puissance qui est le plus souvent d'ordre technologique, liée, par exemple, à la capacité d'intégration des circuits. Pour augmenter encore la puissance des machines, il y a deux solutions. Atteindre un progrès technologique qui permet d'accroître la complexité et la vitesse des circuits intégrés, ou bien introduire un nouveau concept : le parallélisme. En effet, même s'il n'est pas possible à cause de limitations technologiques d'effectuer une opération plus rapidement sur un processeur, rien ne nous empêche d'exécuter plusieurs opérations sur plusieurs processeurs simultanément.

2. **Le coût.** La seconde raison est d'ordre économique. Il est très important de pouvoir produire des machines ayant un excellent rapport coût/performance. Or, assez souvent, l'accroissement de la puissance d'un élément de calcul entraîne

une explosion des coûts et le prix de l'élément devient alors prohibitif. En revanche, il est possible d'atteindre de grandes puissances de calcul à des coûts compétitifs en faisant coopérer de nombreux éléments de calcul de moyenne ou faible puissance. La puissance des différents éléments étant compensée par leur nombre.

1.2.1. La Classification des Différents Types de Machines

Flynn [FLYN66] considère une machine de von Neumann comme étant une machine qui utilise un unique flot d'instructions et qui travaille sur un unique flot de données. Le terme anglais consacré est celui de **SISD** (Single Instruction, Single Data). Il est possible d'introduire le parallélisme en multipliant les flots d'instructions et/ou les flots de données. Nous obtenons alors les quatre classes de machines suivantes.

		Flot de données	
		Unique	**Multiple**
Flot d'instructions	**Unique**	SISD (von Neumann)	SIMD (tableau de processeurs)
	Multiple	MISD (pipeline)	MIMD (multiprocesseurs)

Tableau 1.1 : Classification de Flynn

Le sujet de recherche de cette thèse a été effectué sur une machine de type SIMD. Les informations qui suivent s'appliquent directement au type de modèle et d'architecture utilisés à l'intérieur du présent projet.

- **SIMD -** Dans une machine SIMD, il existe une seule unité de contrôle qui dirige tous les éléments de calcul, chaque élément de calcul travaillant sur une donnée qui lui est propre. Ce type de machine est souvent appelé tableau de processeurs (array processors). Les données traitées par chaque élément de calcul peuvent se trouver dans un espace de mémoire qui est global à toute la machine ou dans des espaces mémoire propres aux différents éléments de calcul. Dans le cas de **PULSE,** chaque élément de calcul opère sous le contrôle d'un séquenceur externe.

Il est maintenant possible d'introduire une classification des différents types de machines basée sur l'organisation de la mémoire. Les données peuvent se trouver dans une mémoire partagée ou distribuée.

La classe des machines SIMD avec une mémoire distribuée est caractérisée par un contrôle centralisé et des données distribuées. Les processeurs d'une machine de ce type sont souvent appelés *éléments de calcul* (processing elements), abrégé par *PE*. Les *PE*s ne disposent pas d'un séquenceur mais sont contrôlés par l'extérieur. La puissance de calcul de la machine parallèle est obtenue par le très grand nombre de PE utilisés. Chaque processeur possède une petite mémoire locale et reçoit ses instructions d'un unique séquenceur de sorte que tous les PE exécutent la même instruction de manière synchrone (figure 1.1).

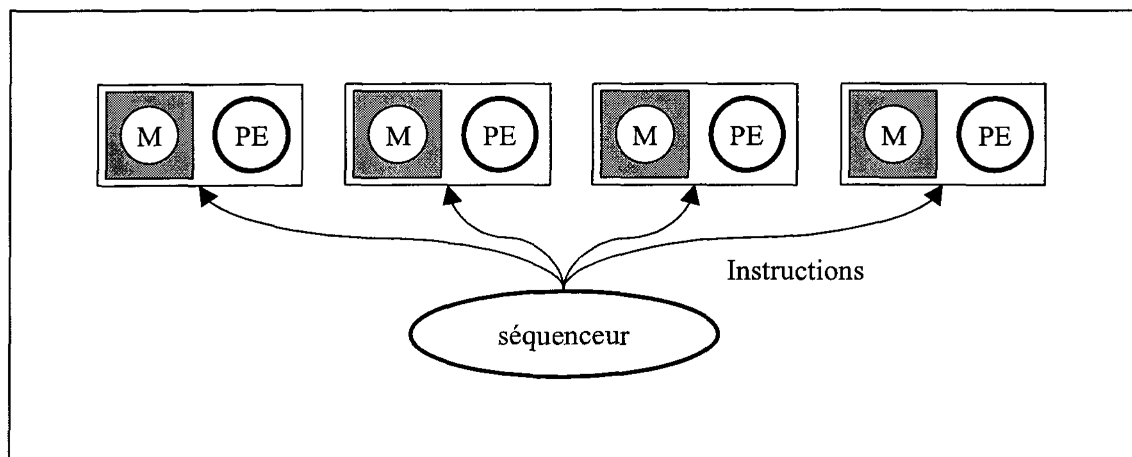


Figure 1.1 : Modèle SIMD avec Mémoires Distribuées

Un mécanisme de masquage permet de définir un sous-ensemble de processeurs, appelé ensemble de processeurs actifs, qui sont les seuls à exécuter l'instruction courante. Les processeurs inactifs ne font rien, en attendant que l'ensemble des processeurs actifs ait terminé l'instruction. Nous verrons plus tard en étudiant les différentes applications que c'est le cas lorsqu'une instruction conditionnelle est employée. Les communications entre les processeurs sont réalisées à l'aide d'instructions spécifiques via un réseau de communication.

1.2.2. L'évolution de la quincaillerie

L'augmentation incessante de la complexité des systèmes pouvant être intégrés sur un circuit rend possible le développement de DSPs de plus en plus complets. La loi de Moore[5],

[5] Gordon E. Moore, Intel Corporation co-fondateur

qui tient depuis 1965, prédit que le nombre de transistors à l'intérieur d'une puce double environ à chaque année. L'émergence, depuis quelques années, des applications multimédia motive et finance ce développement. Ces DSPs sont dédiés à certains traitements numériques intensifs. Les DSPs cibles traitent généralement une énorme quantité de données souvent avec un nombre limité de fonctions. Même Intel, avec sa famille de processeurs Pentium, a ajouté un jeu d'instructions limitées de type SIMD à compter de se technologie MMX au début de 1997. Par la suite, l'arrivée des processeurs Pentium III et 4 incluait de nouvelles technologies, SSE[6] et SSE2, avec pour but d'étendre l'utilisation de la technologie SIMD.

[6] Streaming SIMD Extensions

1.3. PROJET DE DEUXIÈME CYCLE

Ce projet de recherche a été effectué en collaboration avec le Groupe de Recherche en Micro-électronique (GRM) de l'École Polytechnique de Montréal. Le projet de deuxième cycle consistait à participer à l'évaluation matérielle et logicielle d'un système multiprocesseur expérimental développé par le GRM. Le projet du GRM a comme nom **PULSE**.

1.4. LE PROJET PULSE

L'acronyme **PULSE** vient de Parallel Ultra Large Scale Engine qui signifie un moteur de traitement utilisant des unités parallèles avec une intégration ULSI. Le circuit intégré développé par le Groupe est un processeur de type SIMD.

1.4.1. Le Groupe

Le Groupe **PULSE** comptait, à une certaine époque de son développement, près d'une quarantaine de membres des milieux universitaires et privés (voir figure 1.2). Notre participation s'est échelonnée de janvier 1997 à mai 1998.

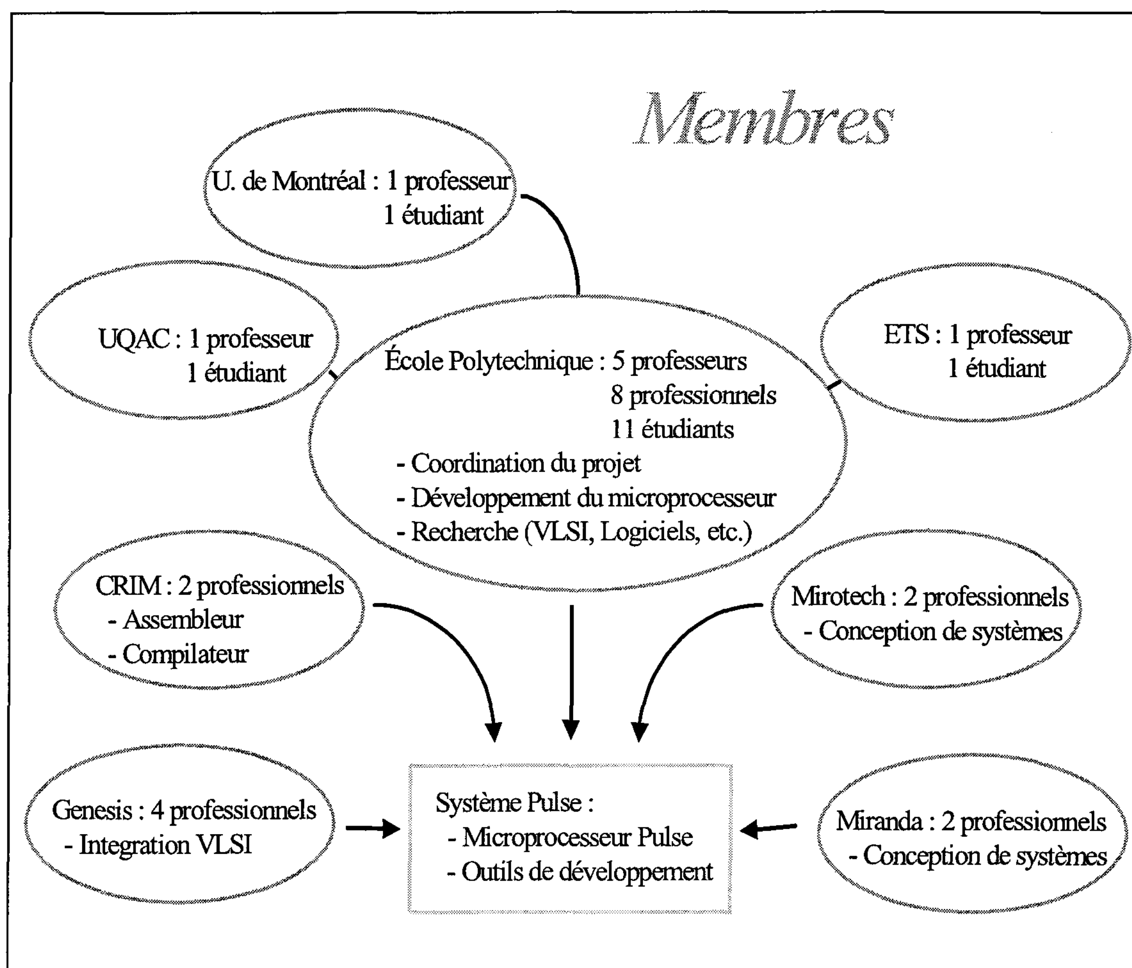


Figure 1.2 : Membres du groupe PULSE

1.4.2. Interaction et Utilisation des Outils de Développement

Le début de notre participation à l'intérieur du Groupe a coïncidé avec la disponibilité de la première version du simulateur logiciel de **PULSE**. Le simulateur a été l'outil principal de développement d'applications utilisé tout au long de ce projet. Le simulateur était basé sur le modèle **VHDL** de **PULSE**. Il n'était pas exportable, il a donc résidé sur un serveur à l'intérieur des locaux du Groupe à l'École Polytechnique de Montréal tout au long du projet. L'accès au simulateur s'est effectué via l'interface du fureteur Netscape.

Au mois d'octobre 1997 une première version portable du simulateur a été utilisée sur UNIX. Malgré l'absence d'interface graphique, ce simulateur a grandement aidé l'avancement des travaux grâce à sa rapidité d'exécution.

À partir de ce moment, les deux simulateurs ont été utilisés conjointement pour tester et développer les différentes applications. Voici leurs caractéristiques principales et les différents aspects de l'utilisation de l'un ou l'autre.

Modèle VHDL :

- Interface graphique : visualisation des résultats d'opérations dans les composantes principales du processeur, opérations pas à pas, production de résultats de performances.
- Modèle *assez* stable à partir de sa disponibilité, en fait, à part quelques exceptions les résultats d'exécutions étaient justes. Des défectuosités et des temps d'indisponibilités ont cependant été rencontrées en cours de développement.

- Exécution lente : un simulateur pour le groupe, temps d'obtention des résultats directement proportionnel au nombre d'utilisateurs et à la rapidité du réseau, alors qu'il est inversément proportionnel à la puissance du serveur. Mise à jour après plusieurs semaines d'utilisation d'un problème de configuration d'une composante de réseau du GRM ralentissant atrocement le transfert de données entre l'hôte du modèle **VHDL** et le monde extérieur.

Modèle exportable :

- Possibilité de produire des fichiers résultats
- Rapide d'exécution (dépend seulement de la performance du processeur hôte)
- Limite sur l'utilisation de certaines instructions et mode d'adressages.
- Simulateur expérimental, utilisé à partir de sa disponibilité à l'intérieur du groupe. Certaines anomalies se sont révélées durant son utilisation.

L'avantage majeur du simulateur exportable était sa rapidité d'exécution des programmes développés permettant ainsi de vérifier leur opération au fur et à mesure.

En résumé, le simulateur exportable a été utilisé pour tester, développer et déverminer de façon générale les applications. Alors que le modèle **VHDL** a été utilisé pour fins de validation, pour caractériser la performance des différentes applications et lorsque le déverminage demandait un travail plus en profondeur

Lors de la fin de mon association avec le Groupe **PULSE**, une version complètement remodelée du simulateur **VHDL** s'annonçait disponible dans les semaines suivantes.

1.4.3. Caractéristiques Générales de PULSE

PULSE V1 est la première mise en œuvre du circuit intégré **PULSE**. Ce chapitre donne une brève description des caractéristiques principales de l'architecture, de l'environnement de développement et du jeu d'instructions.

1.4.3.1. Environnement d'Opération

PULSE V1 est un processeur contenant 4 éléments de calcul de 16 bits à virgule fixe et un contrôleur. **PULSE V1** a été conçu comme support pour effectuer des opérations répétitives sur un nombre élevé de donnés. Sa conception découle des besoins définis par certains domaines d'application comme le traitement d'images et de vidéos. La figure 1.3 montre un exemple d'environnement pour **PULSE.**

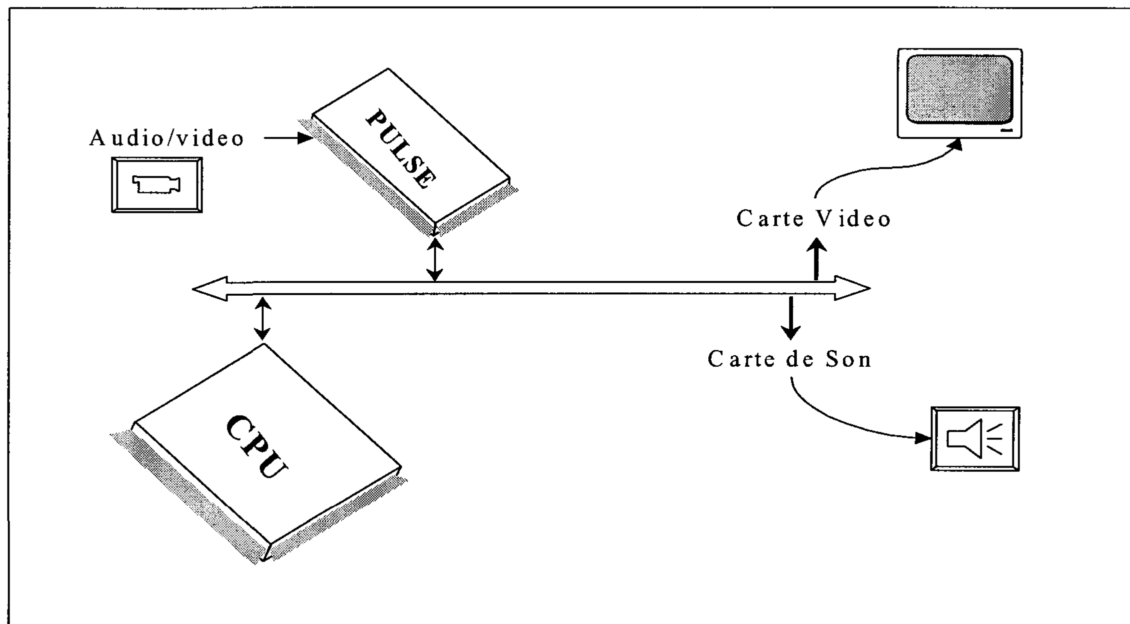


Figure 1.3 : PULSE - Environnement d'Opération - exemple

1.4.3.2. Caractéristiques du Jeu d'Instruction

Le jeu d'instruction de **PULSE** a été développé conjointement avec l'architecture du processeur. Il possède des caractéristiques directement liées aux applications cibles pour lesquelles il a été conçu. La mise en œuvre d'instructions non-linéaires spéciales (maximum, minimum, médian,..) en est un exemple.

De plus **PULSE** possède la possibilité de pouvoir employer des instructions parallèles. Les différentes instructions disponibles sont confidentielles au moment du dépôt de cette thèse. Pour en savoir plus sur le jeu d'instructions; contacter Yvon Savaria du Groupe de Recherche en Micro-électronique.

<u>**Mise en Parallèle des Instructions**</u>

Ces instructions permettent de maximiser à l'intérieur d'une seule instruction l'utilisation de diverses composantes de la puce avec 4 niveaux d'opérations parallèles [4] possibles :

traitement numérique||mouvement de données||communication||E/S contrôle

<u>**exemple:**</u>

macc *mcar(1), ra1 || fwd nport, *mcbw(2) || ssr || io *mccr%, *mcdw%

Cette ligne effectue les opérations suivantes : multiplication - addition, transfère une donnée du canal Nord à la mémoire B, fait un mouvement de données sur le port Sud et effectue une entrée/sortie de donnée sur les ports externes.

Ces instructions, en plus de multiplier le nombre d'instructions possibles par ligne de code, occupent moins d'espace dans la mémoire interne du processeur. Une variété limitée de combinaisons d'instructions est disponible dans le cas de **PULSE V1** alors que pour **V2** toutes les combinaisons possibles seront utilisables.

Mode d'Adressage

PULSE supporte cinq différents type de mode d'adressage. Ces cinq types d'adressage accordent un accès aux données provenant des registres, de la mémoire, des ports de communications et directement de l'instruction.

- Registre
- Direct
- Indirect
- Immédiat
- Adressage absolu

Une description des différents modes d'adressage, avec exemples, est disponible dans [PULS97].

Temps de Latence des Opérations : Normal et Exception

La majorité des instructions du langage **PULSE** nécessitent quatre cycles avant de produire un résultat utilisable par une autre instruction. Les exceptions se trouvent dans le tableau 1.2. Les figures des sections suivantes permettent de se faire une meilleure idée de l'architecture.

Les instructions d'exception sont les suivantes :

Instruction	Description Générale	Latence (Cycles)
fwd	Avance une donnée des registres Nord ou Sud dans la mémoire A ou B.	1
io	Communication d'Entré/Sortie utilisant les compteurs modulo externes C et/ou D.	1
nrr	Rotation des données des registres Nord vers la droite.	1
nsr	Décalage des données des registres Nord vers la droite.	1
nsrr	Rotation des données des registres Nord et Sud vers la droite.	1
n2ssr	Décalage des valeurs sur les ports de communication Nord et Sud simultanément avec les 2 ports agissant comme un seul port continu.	1
stc	Copie les données de mémoire A ou B sur le canal de communication Nord ou Sud.	2
srr	Rotation des données des registres Sud vers la droite.	1
ssr	Décalage des données des registres Sud vers la droite.	1
sla, sra	Décalage arithmétique gauche, droit	2

Tableau 1.2 : Latence - Instructions d'exceptions

L'instruction *ld* en est une particulièrement dérangeante. Cette instruction charge une valeur dans l'accumulateur, un port de communication ou un registre d'adresse. Intuitivement en comparant cette instruction avec le *fwd*, nous pouvions espérer effectuer cette opération en un cycle au lieu des quatre cycles actuels.

Branchement : Latence et pénalité

Instructions Conditionnelles

L'emploi d'instructions conditionnelles est extrêmement coûteux dans le cas d'une machine SIMD. Comme vu précédemment, cette machine ne peut qu'effectuer la même opération sur chacun de ses éléments de calcul à chaque cycle d'horloge. **PULSE** possède la possibilité de pouvoir désactiver les éléments de calcul ne remplissant pas la condition testée, pendant que les autres effectuent les opérations comprises dans la première partie du bloc conditionnel. Durant la seconde partie du bloc, le *else*, les éléments de calcul actifs sont désactivés pendant que les autres effectuent les opérations à l'intérieur du second bloc conditionnel.

Chaque bloc conditionnel nous ampute d'une partie de notre puissance de traitement en plus d'introduire le délai nécessaire à la résolution de l'instruction conditionnelle.

PULSE contient une pile servant à suivre l'activité des éléments de calcul dans le cas de l'utilisation d'instructions conditionnelles. À l'aide de cette pile, **PULSE** supporte les instructions conditionnelles imbriquées.

Selon [GOOR89] les branchements constituent entre 15 et 25 % de toutes les instructions exécutées.

Dans le cas de **PULSE,** toutes les parties d'un bloc d'instructions conditionnelles sont exécutées. Un avantage à cette situation est de pouvoir commencer à exécuter les instructions suivant la directive conditionnelle de façon préemtive sans savoir si l'élément de calcul est à bord ou non. La préemption ne fonctionne par contre qu'à moitié, le

compilateur se charge d'insérer deux (2) cycles vide (*NOP*) entre le *if* et l'instruction suivante au lieu des quatre (4) nécessaires à son exécution. L'explication en est que la quincaillerie doit utiliser ces deux (2) cycles afin de se positionner pour le traitement de ce type d'instruction.

Le fait d'effectuer chacun des blocs d'une structure de contrôle nous amène à poser la question suivante :

Et si aucun des éléments de calcul ne remplit les conditions d'exécution ?

Dans ce cas le bloc est exécuté mais tourne à vide puisque tous les éléments de calcul sont désactivés. Le jeu d'instruction de **PULSE** possède une solution à ce problème de branchement : l'utilisation d'une instruction de branchement conditionnelle : *bnpa* (branch no PEs active). Il faut par contre payer un prix élevé ; quatre (4) cycles vide pour permettre au *if* de fournir sa réponse.

En résumé :

- Deux (2) *NOP*s si on utilise la préemption,
- Quatre (4) *NOP*s si on vérifie l'activité des éléments de calcul.

La morale de cette histoire :

Utiliser le moins possible les instructions conditionnelles.

Boucle

Chaque boucle est caractérisée en langage assembleur par une instruction conditionnelle de branchement (*dbr* : decrement & branch) à la fin du groupe d'instructions compris à l'intérieur de la boucle. Pour tenir le pipeline occupé, **PULSE** entame, à chaque tour de boucle, l'exécution de l'instruction suivant le *dbr*. Il est payant au niveau performance de prendre en considération et de tirer avantage de cette caractéristique en déplaçant une instruction du corps de boucle à la suite du *dbr*, si possible.

1.4.3.3. Caractéristiques de l'Architecture

Caractéristiques Générales

La figure 1.4 présente un bloc diagramme de haut niveau de l'architecture de **PULSE** ainsi que ses canaux de communications externes.

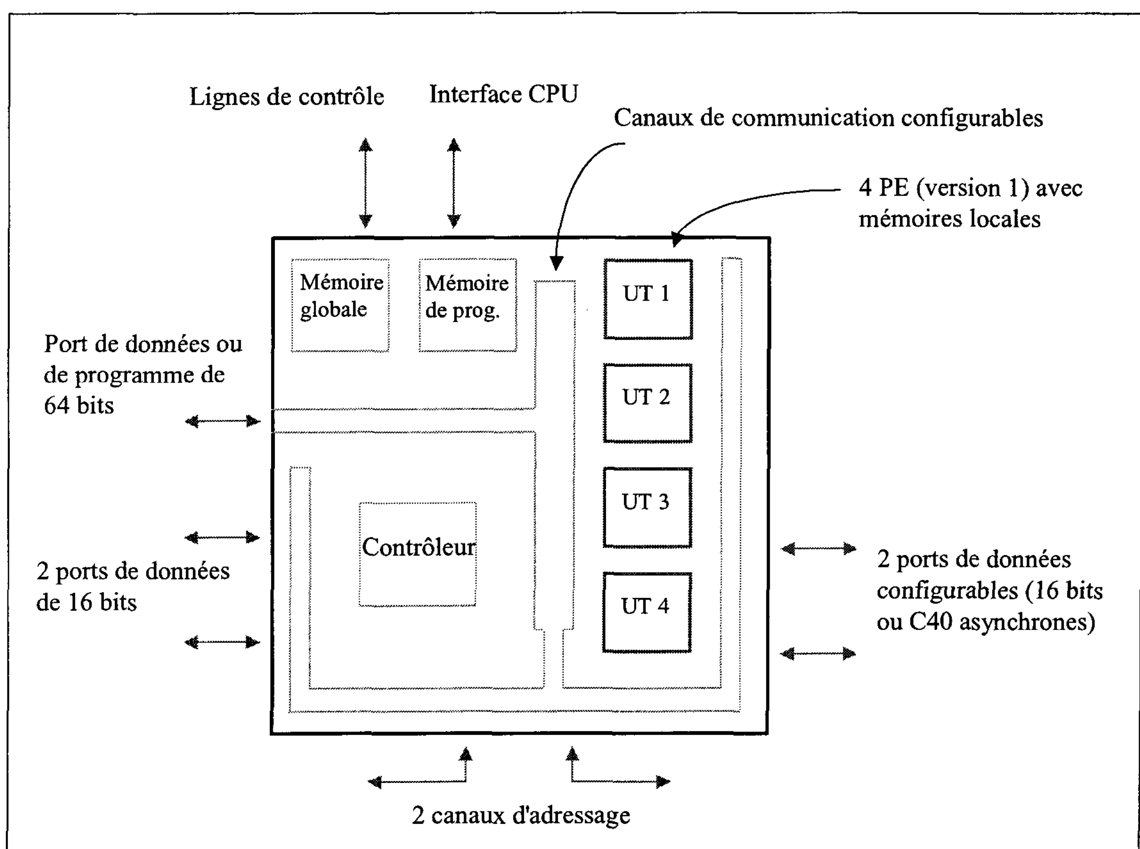


Figure 1.4 : Caractéristiques générales de l'architecture de PULSE

Entrées et Sorties des données

La figure 1.5 présente les canaux de communications pour la circulation des données entre les éléments de calculs, les canaux Nord et Sud et les mémoires source et destination.

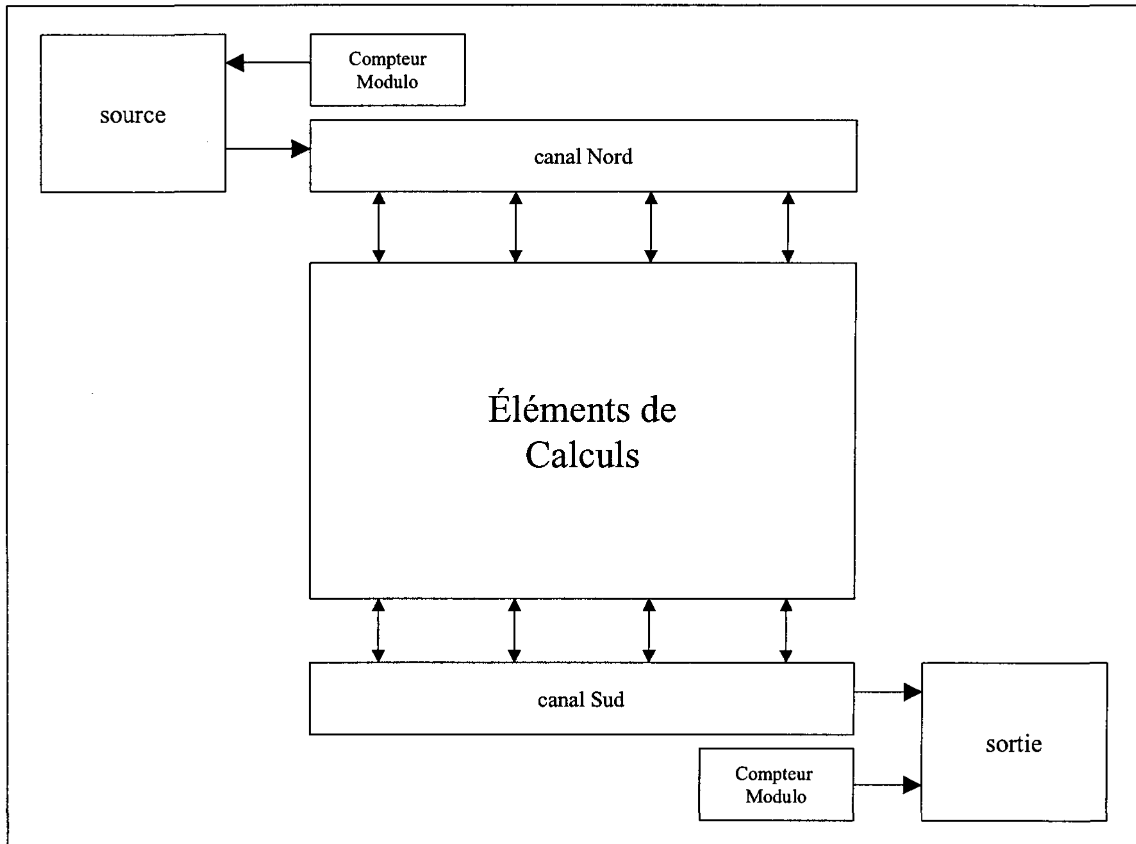


Figure 1.5 : PULSE - Entrées et sorties des données

Canaux de Communications et Traitement des Données

La figure 1.6 représente par un bloc diagramme les options de communications possibles entre les canaux Nord et Sud et les mémoires internes des éléments de calculs.

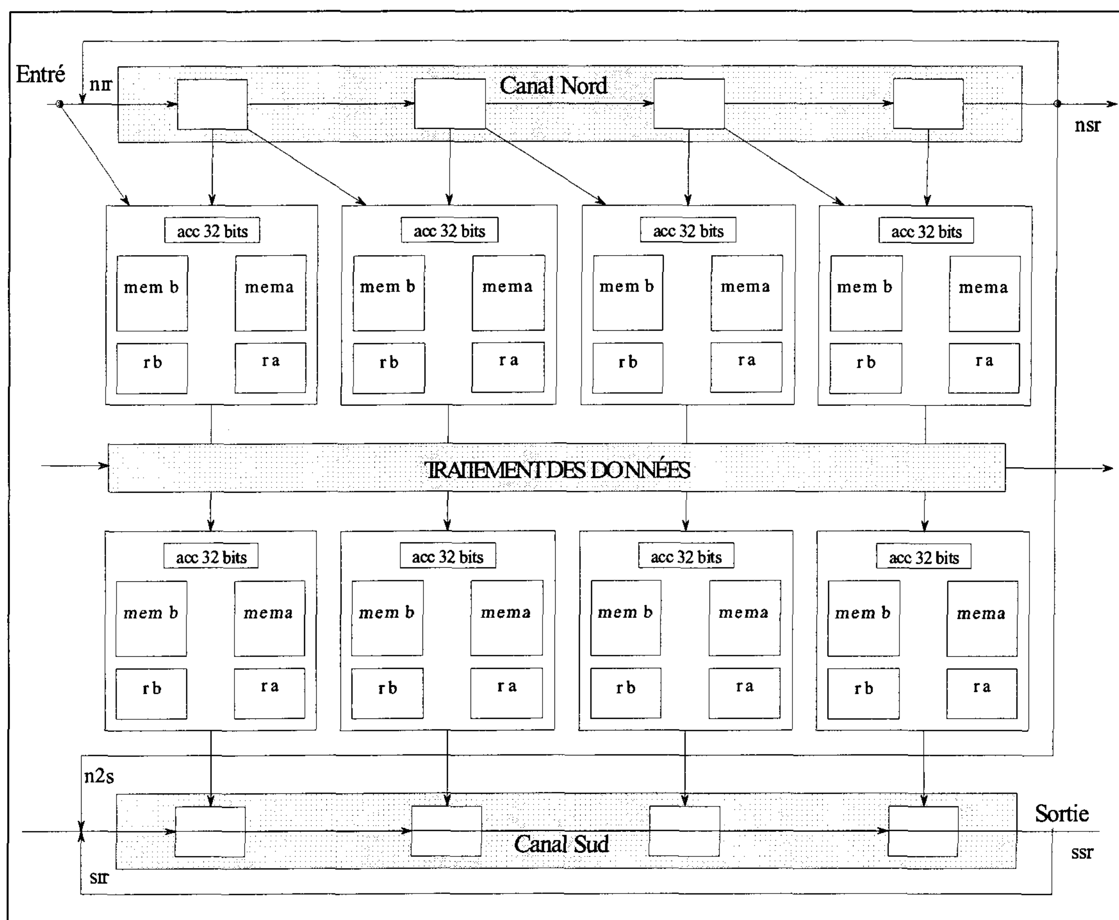


Figure 1.6 : PULSE - Canaux de communications et traitement des données

Élément de Calcul

La figure1.7 présente les différentes étapes du pipeline d'exécution de **PULSE**.

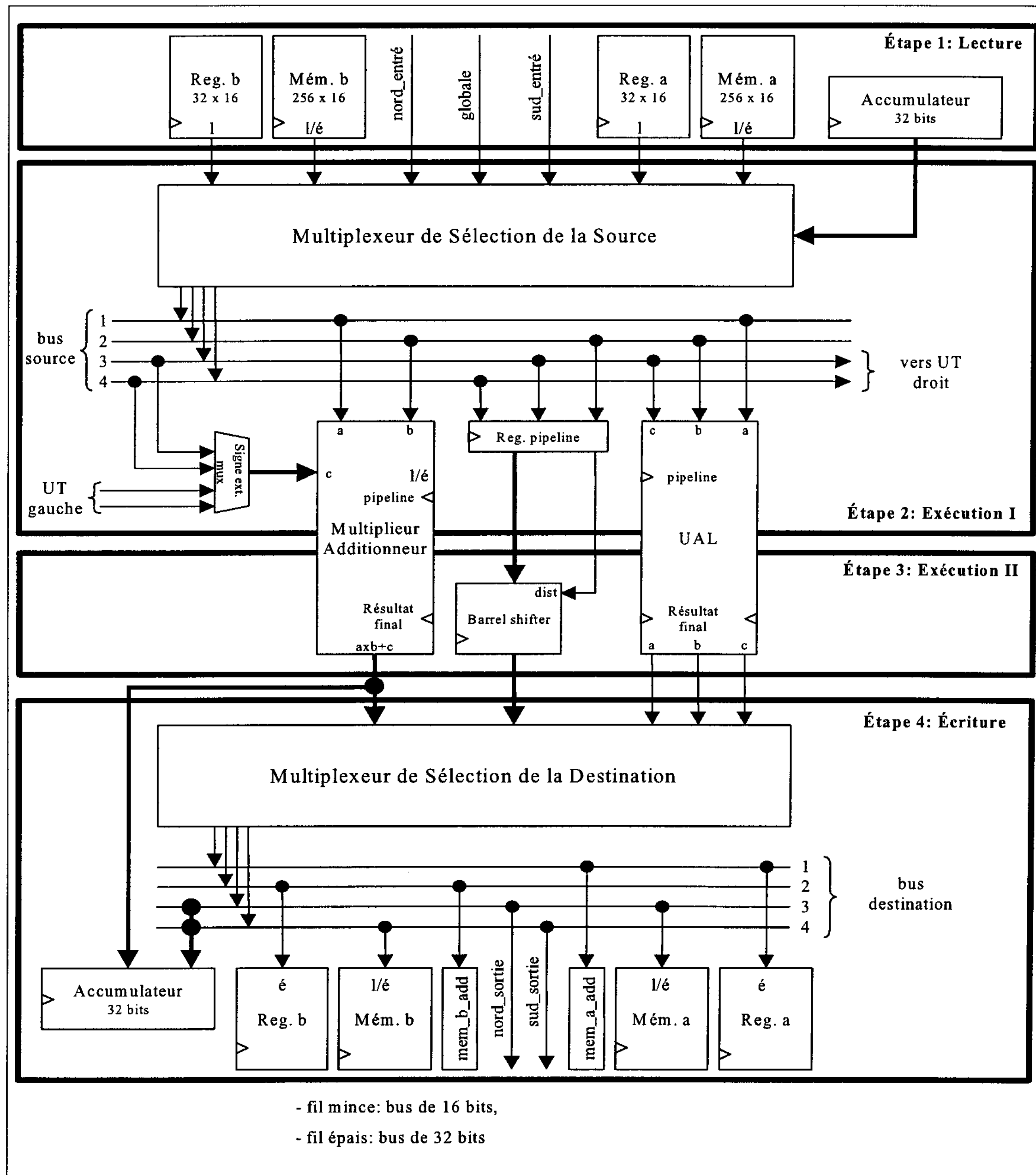


Figure 1.7 : PULSE - Élément de calcul et étapes d'exécution

1.4.4. V1 versus V2

De la création à l'aboutissement commercial d'un projet comme celui-ci, il existe plusieurs itérations.

Un second modèle de **PULSE** a vu le jour dans les derniers mois de 1997. **PULSE** V2 comporte 16 processeurs et améliore certaines limitations de V1.

Le développement et la validation du présent projet ont été faits entièrement sur **PULSE** V1. Les différentes applications développées ont été évaluées de façon théorique sur V2, dans certains cas, à partir des caractéristiques techniques fournies par le groupe de développement.

1.5. CONTRIBUTION DU PRÉSENT PROJET

Ce projet avait au départ deux objectifs principaux :

1. Développement d'applications et répartition de ces dernières sur **PULSE**,
2. Test et évaluation des outils en place et apporter, si possible, des suggestions pour améliorer le produit.

1.5.1. Méthodologie du Projet

Afin de pouvoir poursuivre ces deux objectifs, le projet s'est divisé en 5 étapes :

1. Étude de la documentation existante sur **PULSE**.
2. Étude des différents compétiteurs sur le marché.
3. Initiation au parallélisme.
4. Familiarisation avec le langage et l'architecture à l'aide d'une première application.
5. Recherche et développement de différentes applications.

Les trois premières étapes se sont chevauchées à l'intérieur des premiers mois du projet pour permettre d'aboutir avec une première application. La majeure partie du temps consacré au projet a porté sur la cinquième étape avec de multiples itérations sur la première étape.

1.6. PRÉSENTATION DE LA THÈSE

La présente thèse comporte deux parties distinctes :

1. Une étude succincte des principaux joueurs sur le marché visé, cette partie est couverte à l'intérieur du prochain chapitre.
2. Les applications développées dans le cadre de ce projet.
    - Le chapitre 3 décrit les applications et la façon dont leurs partitions se sont effectuées sur l'architecture de **PULSE**.
    - Le chapitre 4 montre les résultats (performance) obtenus pour chacune des applications développées. On y présentera aussi une comparaison des temps d'exécution de certaines applications avec d'autres processeurs.
    - Le chapitre 5 partage les «bonnes» et les «mauvaises» applications et présente la conclusion et les recommandations.

CHAPITRE 2

REVUE DES SYSTÈMES EXISTANTS

2.1. PRÉAMBULE

Quatre compétiteurs du processeur **PULSE** sont étudiés dans ce chapitre. Il s'agissait ainsi d'avoir une idée générale de ce que la compétition avait développé, de quelle façon elle attaquait le marché et quelles étaient les différences majeures avec **PULSE**.

Trois des compétiteurs sont : le **CNAPS**[7] de Adaptive Solutions, le **TMS320C80**('C80) de Texas Instruments et le **SHARC**[8] de Analog Devices. Tous étaient disponibles sur le marché lors de la rédaction de ce document. Le dernier, le **A236** de Oxford Micro Devices Inc. devait avoir un prototype disponible à la fin d'avril 1997. Le produit manufacturé devait être disponible vers le milieu de la même année.

De fait, mis à part le **CNAPS**, chacun à sa manière tend à avoir sur le même circuit intégré un système le plus complet possible. Il s'agit d'une combinaison d'éléments spécifiques orientés vers des applications requérant une grande quantité de traitements, comme **PULSE**. Ces traitements sont souvent indépendants les uns des autres, ils peuvent être effectués en parallèle. Le but est d'exploiter le parallélisme dans les données de plusieurs algorithmes afin de les traiter nettement plus rapidement qu'avec un ordinateur série. Tous ces DSPs sont censés être entièrement programmables, donc versatiles dans leur domaine d'application.

---

[7] « Co-processing Node Architecture for Parallel Systems »

[8] « Super Harvard Architecture Computer »

2.2. INTRODUCTION

Le but de cet exercice était d'avoir un premier contact avec les circuits intégrés prénommés et aussi d'essayer d'entrevoir la philosophie derrière chacune des conceptions. Je voulais aussi faire en partie le tour des informations contenues dans le tableau 1 du document ***PULSE VI** Overview*. Ce tableau compare les fonctionnalités de ces quatre compétiteurs avec **PULSE**. Un autre objectif était de situer chacun des processeurs dans son environnement, tant au niveau logiciel que matériel.

2.2.1. Description générale

2.2.1.1. CNAPS

Le **CNAPS** est le plus ancien des quatre compétiteurs, sa mise en marché remonte à la première moitié de 1994. Sa conception diffère des autres circuits intégrés en ce sens que le *CNAPS-10xx* qui est le cœur du système, n'est pas un système complet. ***Le CNAPS-10xx*** est fait de plusieurs processeurs mis en parallèle et interconnectés à l'intérieur du même circuit intégré. Il doit être couplé à d'autres éléments pour pouvoir être opérationnel. Un autre circuit intégré, le ***CNAPS Sequencer Chip*** (SCS), est nécessaire comme unité de contrôle pour gérer les instructions et les données. La figure 2.1 illustre de quelle façon sont reliés les différents éléments.

Le **CNAPS** est un système de 32 bits SIMD opérant à une fréquence de 20 MHz. Deux modèles du ***CNAPS-10xx*** sont manufacturés ; le 1016 (16 processeurs internes) et le 1064 (64 processeurs internes). Les processeurs sont tous à virgule fixe. Il est possible de mettre jusqu'à 8 ***CNAPS-10xx*** en parallèle.

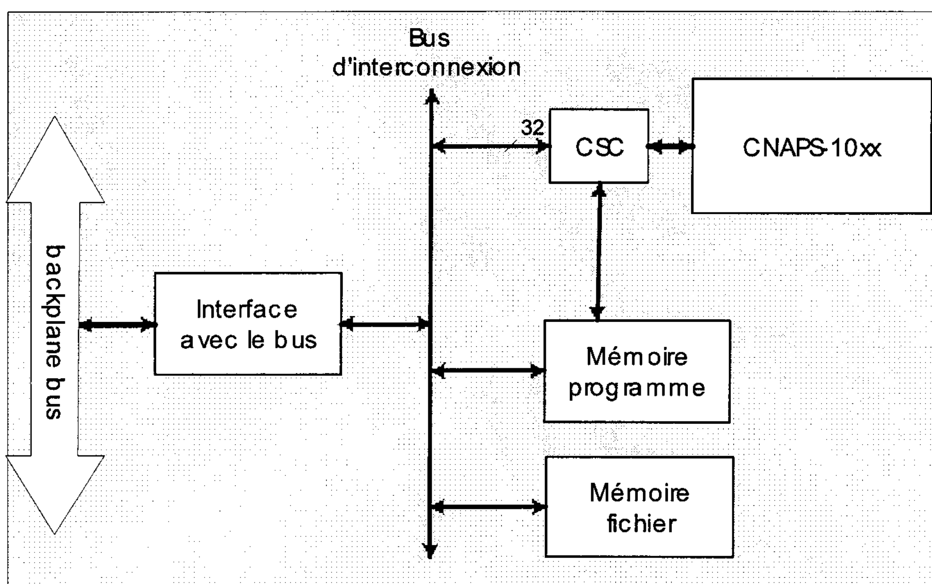


Figure 2.1 : Sous-système CNAPS

Le **CNAPS** peut être programmé en langage C ou en assembleur et il possède des outils de développement sur UNIX et sur PC.

2.2.1.2. 'C80

Le **'C80** est un processeur 32 bits MIMD de type RISC avec une horloge de 50 MHz. Le circuit intégré du 'C80 inclut cinq processeurs programmables, un contrôleur de transfert et un contrôleur pour la vidéo. Il possède une mémoire interne de 50 Kb.

Quatre des cinq processeurs sont identiques, il s'agit de DSP en configuration MIMD. Le cinquième processeur, le processeur maître, est un processeur 32 bits de type RISC. Il inclut une unité à virgule flottante. Les quatre autres processeurs sont à virgule fixe.

L'accès aux modules de mémoires internes se fait à l'aide d'un réseau d'interconnexion de type Cross-bar.

Les cinq processeurs peuvent être programmés en langage C et en langage d'assemblage. Le 'C80 possède des outils de développement sur station de travail seulement.

2.2.1.3. SHARC

Le **ADSP-2106x SHARC** est un processeur 32 bits à virgule flottante avec une architecture de type Harvard. Dans ce cas-ci également, deux modèles sont disponibles, l'***ADSP-2100*** et l'***ADSP-21062***. La seule différence entre les deux circuits est la quantité de mémoire interne. L'***ADSP-2100*** a une SRAM de 4 mégabits et l'***ADSP-21062*** à moitié moins, soit 2 mégabits.

Contrairement aux autres processeurs, les puces de la famille ADSP-210x intègrent un processeur unique. Le **SHARC** doit sa rapidité à l'organisation de sa mémoire. Le **SHARC** peut être programmé en C ou en assembleur. Ses outils de développement sont utilisables sur PC seulement. Pour avoir un système multiprocesseur, il faut mettre plusieurs SHARC en parallèle.

2.2.1.4. Oxford A236

Nommé aussi par la compagnie Oxford le **PDSP**[9], il est le dernier arrivé sur le marché de tous les DSP identifiés dans ce document.

Le **A236** est un processeur SIMD fonctionnant avec une vitesse d'horloge maximale de 50 MHz. Le circuit intégré du **A236** inclut quatre processeurs 16 bits à virgule fixe de type RISC avec un processeur scalaire de 24 bits qui sert de contrôleur.

Le A236 possède une mémoire interne de 2 K octets. Il peut être programmé en langage C et en assembleur mais, comme dans le cas du **SHARC**, il ne peut être utilisé que sur un PC.

[9] Parallel Video Signal Processor Chip

2.3. ARCHITECTURE

2.3.1. CNAPS

2.3.1.1. Le CSC

Le ***CSC*** est une unité externe chargée de contrôler les processeurs internes du ou des ***CNAPS-10xx***. Sa tâche principale est de gérer l'ordre d'exécution des instructions du programme. Il se sert du bus de commande (Op Bus) pour faire parvenir les instructions aux processeurs.

Le ***CSC*** a aussi comme fonction de contrôler les bus d'entrées et de sorties (In et Out Bus) des processeurs parallèles. Il est aussi responsable de toutes les connexions avec l'environnement externe.

Le ***CSC*** est composé de trois unités principales : une unité d'interface avec le bus d'interconnexion, une unité de décodage et d'exécution des instructions (circuit séquentiel) et une unité de contrôle et de synchronisation des E/S du ***CNAPS-10xx***. Une **UAL** et des registres internes de 32 bits sont utilisés conjointement par ces trois unités. La figure 2.2 présente un schéma des connexions reliant le ***CSC*** et multiple ***CNAPS-10xx***.

Le circuit intégré du CSC possède 240 broches.

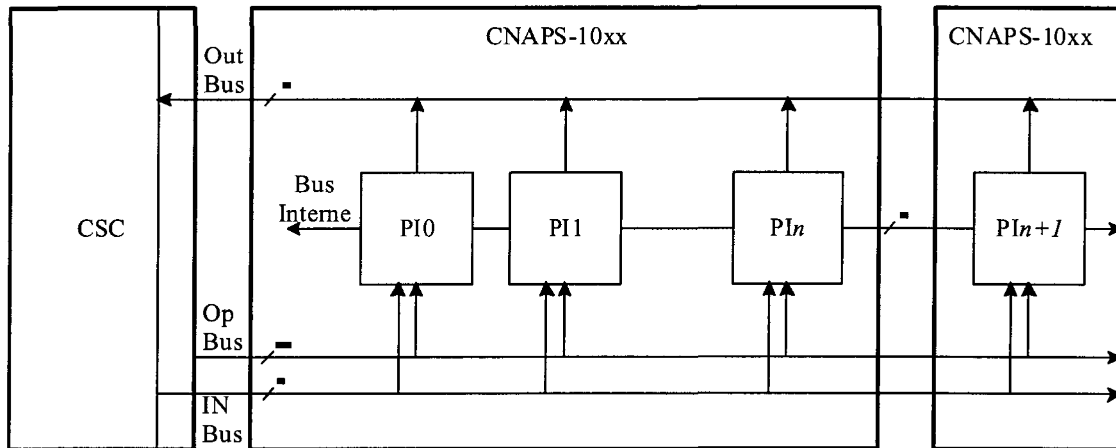


Figure 2.2 : CSC et CNAPS-10xx avec connexions[10]

2.3.1.2. Le CNAPS-10xx

Chaque processeur interne (PI) possède une mémoire locale de 4 K octets avec les composantes standards d'un processeur mathématique soit : un additionneur, un multiplicateur, une unité logique, 32 registres de 16 bits et des tampons d'E/S.

La communication inter-processeur se fait au moyen d'un bus de 4 bits (2 bits d'entrée, 2 bits de sortie). La dimension de ce bus indique clairement que son utilisation est restreinte à de rares cas.

Un système **CNAPS** peut être composé de un à huit circuits intégrés comportant chacun 16 ou 64 processeurs. Si on multiplie ces nombres, on arrive à un total possible de 512 processeurs parallèles, utilisant tous les mêmes bus d'E/S de 8 bits ! Pour faire face à

---

[10] Figure extraite de [ADAP94] PAGE 2-2

l'engorgement possible, chaque processeur a la possibilité de se déconnecter virtuellement du bus s'il ne traite pas de données. Il possède de plus un tampon de sortie.

Plusieurs types d'arbitrages (4) existent pour faire face aux conflits sur un des quatre bus.

Le circuit intégré du ***CNAPS-1016*** possède 85 broches alors que celui du 1064 en possède 105.

2.3.2. 'C80

Le **'C80** appelé aussi le **MVP**[11] est un circuit intégré développé tout particulièrement pour faire du traitement multimédia. Texas Instruments fait mention exclusivement d'applications multimédias pour son processeur parallèle dans sa documentation. Un contrôleur vidéo (CV) sert d'interface entre le circuit intégré et les images externes. Il contient deux tampons d'images, un pour l'affichage et un pour l'image suivante. La figure 2.3 présente le bloc diagramme du **'C80.**

[11] Multimedia Video Processor

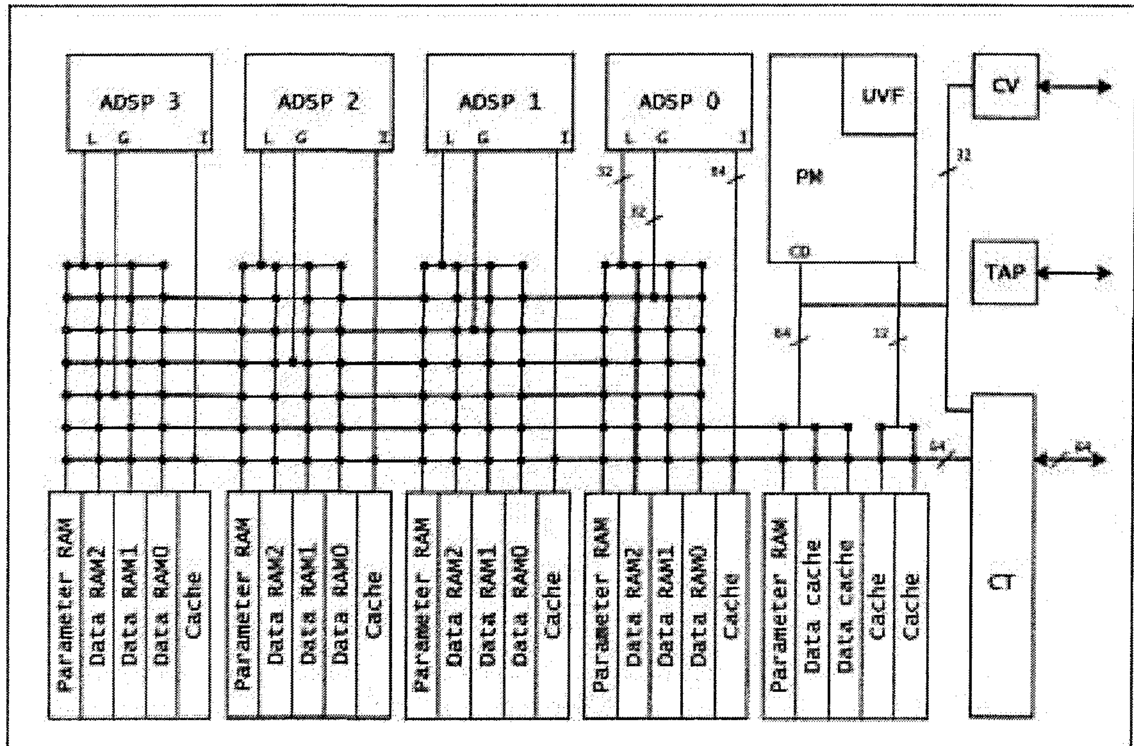


Figure 2.3 : Bloc diagramme du 'C80[12]

Le réseau d'interconnexion entre les processeurs et les modules de mémoire est de type Cross-bar (n x n). L'avantage de ce type de réseau est assez évident, il offre la possibilité de faire des transferts de données simultanément de la part de chaque module de mémoire. Ce type de réseau est une réponse matérielle aux problèmes d'engorgement auxquels d'autres processeurs, tel le **CNAPS**, doivent faire face. Le problème de ce réseau est naturellement son coût matériel à cause du nombre élevé et de la dimension des

[12] Figure extraite de [TI96] Chapitre 2

commutateurs sur le réseau. Le réseau possède une autre fonctionnalité, il est aussi utilisé pour établir une communication inter-processeur.

La mémoire interne de 50 Kb est divisée en 25 unités de dimensions égales, 16 sont partagées entre les quatre processeurs (ADSP) pour supporter les opérations parallèles (Data RAM et Antémémoire d'instruction). Les autres blocs (Parameter RAM) servent à emmagasiner les instructions et différentes adresses. Le contrôleur de transfert s'occupe, à travers deux bus de 64 bits, de gérer l'accès à la mémoire interne par les processeurs et d'effectuer les transferts de paquets avec l'extérieur.

Le circuit intégré du **'C80** possède 305 broches et, fait à noter, il dissipe jusqu'à 12 Watts et nécessite des ailettes de dissipation pour la chaleur. Il est le seul des quatre à avoir un boîtier en céramique.

2.3.2.2. Le processeur maître

Le processeur maître (PM) est un processeur 32 bits de type RISC avec une unité à virgule flottante (UVF) intégrée. Il a un ensemble d'opérations spéciales à virgule flottante pour supporter le graphisme à 2 ou 3 dimensions requérant une plus grande précision. Le PM s'occupe de gérer les interruptions externes. Le **PM** agit comme unité de contrôle pour l'exécution des instructions.

Le **PM** peut accéder simultanément par ses deux bus à la mémoire interne. Le bus de 64 bits sert à accéder l'antémémoire (C) ou les données (D). Le second bus (I) de 32 bits est utilisé pour accéder aux instructions de type RISC à partir de ses deux antémémoires d'instruction.

2.3.2.3. Les quatre processeurs parallèles (ADSP)

Chaque processeur est composé de trois unités principales ; l'unité de données, deux unités d'adressage (une locale et une globale) et l'unité de contrôle du flot d'instruction.

L'unité de données s'occupe d'effectuer les opérations sur les données. Les unités d'adressage s'occupent de charger ou de sauver en mémoire les adresses. Chaque processeur a deux bus d'accès à la mémoire de 32 bits, un local (L) et un global (G). Le troisième bus (I) de 64 bits est utilisé pour accéder aux instructions à partir de l'antémémoire d'instruction. Chaque **ADSP** peut donc accéder simultanément à chaque cycle d'horloge à une instruction et deux adresses. L'unité de contrôle du flot d'instruction s'occupe de contrôler le pipeline d'instruction, de décoder les instructions en plus des rapports avec le contrôleur de transfert (CT).

2.3.3. SHARC

L'architecture Harvard est caractérisée par deux blocs de mémoire indépendants permettant un accès simultané par le processeur aux instructions et aux données à l'aide de deux bus indépendants. Dans le cas du **SHARC**, le processeur peut aussi accéder simultanément à une antémémoire d'une dimension de 128 octets. Il peut donc charger au cours d'un cycle une instruction de l'antémémoire et accéder à une donnée de chaque bloc de mémoire comme le 'C80. Le processeur a quatre bus ; Program Memory Address (PMA), Data Memory Address (DMA), Program Memory Data (PMD), Data Memory Data (DMD).

Le **PMA** Bus et le **DMA** bus sont utilisés pour transférer l'adresse des instructions et des données. Ces adresses sont produites par les deux générateurs d'adresses (DAG 1 et 2) et le circuit séquentiel. Les **PMD** et **DMD** bus sont utilisés pour transférer des données et des instructions. Ils sont bidirectionnels. La figure 2.4 présente le bloc diagramme du processeur **SHARC**.

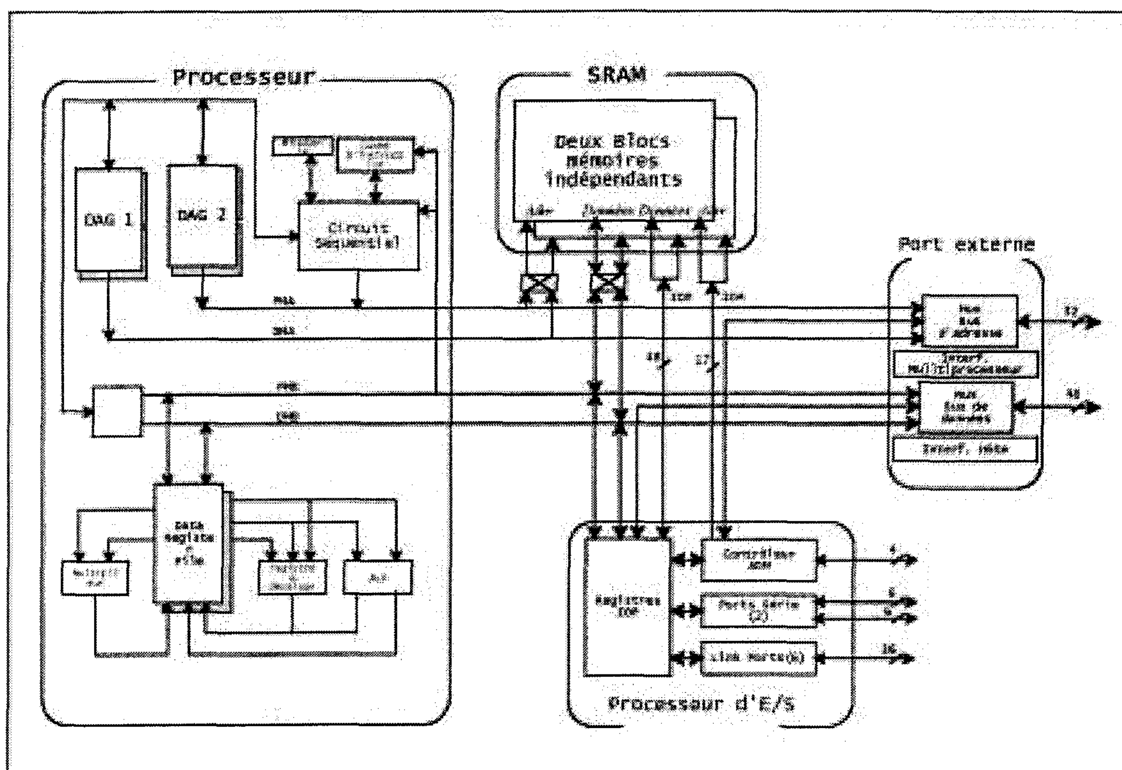


Figure 2.4 : Bloc diagramme du SHARC[13]

Les deux blocs de mémoires SRAM sont donc accessibles individuellement par le processeur principal et le processeur d'E/S ou le contrôleur d'**ADM**.

[13] Figure extraite de [ANAL95] Chapitre 1

Un registre de données est utilisé pour transférer des données entre les unités de calcul (ALU, multiplicateur et une unité à décalage). Ces trois unités de calcul sont connectées en parallèle. Le registre emmagasine aussi les résultats issus des trois unités de calcul. L'indépendance entre la génération d'adresses et l'unité de calcul permet à cette dernière d'occuper son temps exclusivement à travailler sur les données.

Le circuit intégré du **SHARC** possède 240 broches. Avec ses instructions de 1 cycle, il m'a semblé le plus simple de tous les processeurs.

2.3.4. A236

La mémoire interne est divisée en deux blocs de dimensions égales ; 1 Kb pour l'antémémoire d'instructions et 1 Kb pour l'antémémoire de données. La mémoire est accessible via un bus de 64 bits. Le réseau d'interconnexion entre les quatre processeurs vectoriels et l'antémémoire de données est de type Cross-bar. Le réseau connecte aussi le processeur scalaire et l'antémémoire de données. Les processeurs parallèles (vectoriels) peuvent se servir des commutateurs pour échanger des données entre eux et communiquer avec le processeur scalaire, rendant ainsi la communication inter-processeur possible.

Le processeur scalaire est utilisé comme unité de contrôle. Il sert aussi à générer simultanément les adresses mémoires à être utilisées par les quatre processeurs parallèles. Il possède 32 registres internes et une **UAL**. Il s'occupe aussi de contrôler les ports d'entrées et de sorties. Une unité spéciale s'occupe de gérer les Interruptions (Intr. Ctl.).

Chaque processeur parallèle possède 64 registres de 16 bits chacun. Les processeurs peuvent opérer normalement comme quatre processeurs de 16 bits ou comme 8 processeurs

de 8 bits dans certains cas spécifiques. Les processeurs parallèles sont utilisés comme unité arithmétique.

Un estimateur de mouvement (EM) est ajouté à l'intérieur de la puce. Sa fonction est d'accélérer la détection de mouvement lors de la compression vidéo.

Le circuit intégré du **A236** possède 208 broches. La figure 2.5 présente le bloc diagramme du **A236**.

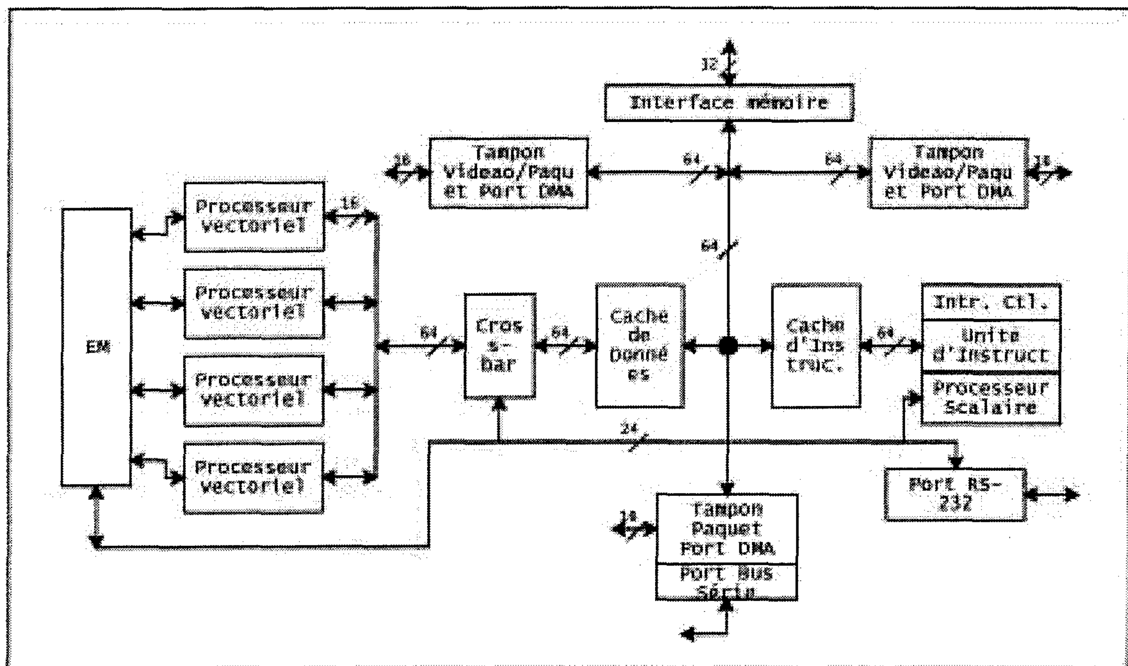


Figure 2.5 : Bloc diagramme du A236[14]

14 Figure extraite de [ANAL95]

2.4. ENVIRONNEMENT DE DÉVELOPPEMENT

2.4.1. CNAPS

Le développement avec **CNAPS** se fait via trois différents ensembles logiciels, un pour Windows, un pour UNIX et un spécialisé pour les réseaux de neurones sous UNIX. Les outils logiciels comprennent : compilateur C, assembleur, des librairies de fonctions, un simulateur et un dévermineur. Le langage utilisé par le compilateur est appelé ***le CNAPS-C***, il s'agit d'un dérivé du C ANSI spécialisé pour son architecture parallèle. Certaines extensions ont été ajoutées au langage, mais aussi certaines restrictions dans le but de supporter la programmation parallèle. Deux librairies sont disponibles de Adaptive Solutions. La première d'une centaine de fonctions (QuickLib) est surtout orientée pour traiter les données, elle est écrite en langage d'assemblage. La seconde (OEMlib) facilite la gestion programme. Elle est écrite en langage de haut niveau.

En plus des outils de développement standard pour construire un programme, l'ensemble logiciel pour Windows comprend une librairie dynamique pour modéliser les fonctions non-linéaires complexes (Back-Propagation Dynamic Link Library) comme les réseaux de neurones.

Basé sur les outils de développements, les librairies et les types de cartes disponibles, le système **CNAPS** oriente son utilisation sur le marché de façon différente par rapport aux autres concurrents étudiés. Le **CNAPS** semble assez présent dans le monde du traitement parallèle au niveau universitaire, de la recherche et commercial. Des applications et des environnements pour le système **CNAPS** ont aussi été développés à l'externe.

2.4.2. 'C80

Le développement logiciel avec le **'C80** se fait à partir d'une station SPARC. Des outils de développement sont disponibles, ils comprennent un simulateur, un compilateur C, un assembleur, un dévermineur et une librairie orientée multimédia.

Les fichiers sources peuvent être en C ANSI standard. La transition du langage pour le parallélisme se fait lors de l'assemblage. Deux assembleurs bâtissent le code respectif pour le processeur maître et pour les processeurs parallèles. Les deux fichiers assembleurs sont ensuite liés dans un fichier exécutable commun.

2.4.3. SHARC

Comme les autres puces, **SHARC** est supporté par un éventail d'outils de développement incluant un simulateur, un compilateur C, un assembleur, un dévermineur et une librairie.

L'ensemble des outils opère sur MS-DOS.

2.4.4. A236

L'environnement de développement du **A236** n'est utilisable que sur Windows. Un environnement de développement est disponible. Il inclut un simulateur, un compilateur C, un assembleur, un optimiseur et un dévermineur.

C'est le type de variable employée par le programmeur lors de leurs déclarations qui laissent savoir au compilateur si les données doivent être traitées par le processeur scalaire ou les processeurs parallèles. Ce type de structure est appelé par la compagnie Oxford «Quad». Cette approche semble à mi-chemin entre le **CNAPS** et le **'C80.** Elle demande au

programmeur de définir ses variables mais ne requiert pas de changement majeur en ce qui concerne le code. Elle ne demande pas au compilateur d'effectuer la partition des données par déduction pour un traitement parallèle.

2.5. CONCLUSION

Plusieurs architectures se côtoient au sein de ce groupe de DSPs. La possibilité de traiter rapidement (en temps réel) des données est l'objectif principal de chacun de ces processeurs. Il est intéressant de noter la diversité des approches employées par chacune des entreprises pour développer ces puces.

Bien entendu le parallélisme a aussi des répercussions sur la façon d'envisager les problèmes du point de vue algorithmique. Il est intéressant de voir les différentes solutions qui ont été employées afin de maximiser le traitement parallèle de certaines parties des logiciels. Une partie importante du travail doit être consacrée au partitionnement des algorithmes. Même avec une quincaillerie ultra performante, si un ou plusieurs processeurs tournent à vide, le problème demeure entier. La solution la plus élégante consiste à intégrer cette division lors de la compilation avec, pour avantage, de ne pas alourdir le travail des programmeurs et ainsi demeurer général. Est-il possible d'effectuer ces opérations de manière rentable, intelligente et applicable à différents algorithmes ?

Le développement et la mise en marché du MMX (Multimedia Extensions) prouvent hors de tout doute l'existence d'un marché en pleine expansion pour ces puces. L'énorme gain de rapidité, comparativement au Pentium standard, sur certains types d'opération confirme l'utilité et la réalité de l'accélération des performances lors de traitements parallèles.

CHAPITRE 3

DÉVELOPPEMENT D'APPLICATIONS

3.1. PRÉAMBULE

Ce chapitre comprend le corps de cette recherche. Il est divisé en deux parties. La première partie comprend une présentation des outils et des méthodes employées pour le développement d'applications. La seconde partie décrit les applications développées et les algorithmes employés durant ce projet. Cette description est accompagnée des caractéristiques particulières à chaque développement. La performance résultante de chaque application est présentée au chapitre suivant. Les différentes applications sur lesquelles cette recherche s'est développée sont : multiplication vecteurs-matrice, algorithmes de cryptage RSA et IDEA, algorithme de Bresenham, modèle continu (poisson) et transformation binaire d'images.

3.1.1. Objectifs de développement

Un des éléments essentiels d'une mise en marché réussie pour un produit de la catégorie de **PULSE** est de pouvoir posséder une bibliothèque de fonctions performantes. Par exemple, dans le cas du ***CNAPS*** d'*Adaptive Solutions,* une bibliothèque écrite en langage d'assemblage possédant une centaine de fonctions est disponible en plus d'une bibliothèque écrite en langage de haut niveau. Un des deux objectifs du projet était de tenter d'enrichir la bibliothèque de PULSE d'applications performantes.

Aucun critère spécifique et aucune direction de développement n'ont été soumis par le GRM, du moins à prime abord. Dans le cadre de ce projet, le mandat était d'explorer les champs d'applications susceptibles de contenir des applications intéressantes pour **PULSE** et de déterminer la complexité du développement sur cette plate-forme. Ces activités se

sont faites dans un environnement physique externe à celui du GRM, les échanges d'informations et les questions se faisant par courriel majoritairement.

La première étape à suivre est de prendre un algorithme séquentiel et d'essayer d'en augmenter la performance en effectuant une mise en parallèle du traitement. Une fois l'algorithme remodelé en application parallèle, la performance peut encore être améliorée sur **PULSE**. Il faut alors mettre en parallèle les instructions d'E/S et les instructions de traitement des données. Une seule ligne d'instruction peut ainsi contenir jusqu'à quatre instructions effectuées simultanément. Un problème associé à la dépendance de données peut alors se créer. Il y a deux solutions pour y remédier ;

1- Insérer des instructions vide (*NOP* - No OPeration),
2- Remplacer par d'autres opérations les cycles sans opération. Ces opérations doivent évidemment être faites en dehors du champ de dépendance de données.

3.1.1.1. Mise en Parallèle d'application

PULSE, dans sa version première, comprend quatre ***PE***. Chaque cycle d'horloge permet idéalement de traiter simultanément un nombre de données égal au nombre de ***PE***s. **PULSE** possède un jeu d'instructions parallèles tel que vu au chapitre 1 à la section 1.4.3.2.

3.1.1.2. Dépendance de données

Une dépendance de données se présente dans une mise en œuvre en pipeline lorsque une relation producteur – consommateur entre les instructions existent. La valeur de cette ressource partagée est accédée avant qu'une opération précédente n'ait produit son résultat

[GOOR89]. Cette dépendance peut être un raw (read after write), un war (write after read) ou un waw (write after write) [GOOR89].

La dépendance est reliée directement à la latence des différentes instructions. Tel que vu précédemment dans la section 1.4.3.2, à part certaines exceptions, le temps de latence est de 4 cycles pour chaque instruction.

Les instructions s'exécutent selon les étapes des figures 6 et 7 du chapitre 1.

1. Lecture des données

2 et 3. Exécution des opérations

4. Écritures des données et Accumulation

Ces quatre étapes sont exécutées en pipeline dans **PULSE** afin de pouvoir obtenir un résultat à chaque cycle d'horloge à compter du quatrième cycle, si le pipeline est plein (figure 3.1).

Une dépendance de données peut se présenter lorsque l'instruction $i + 2$ doit se servir du résultat de l'instruction $i + 1$. Ce résultat n'est disponible que trois cycles plus tard, la lecture se trouve donc fausse car la valeur en mémoire peut être n'importe quoi sauf la valeur désirée.

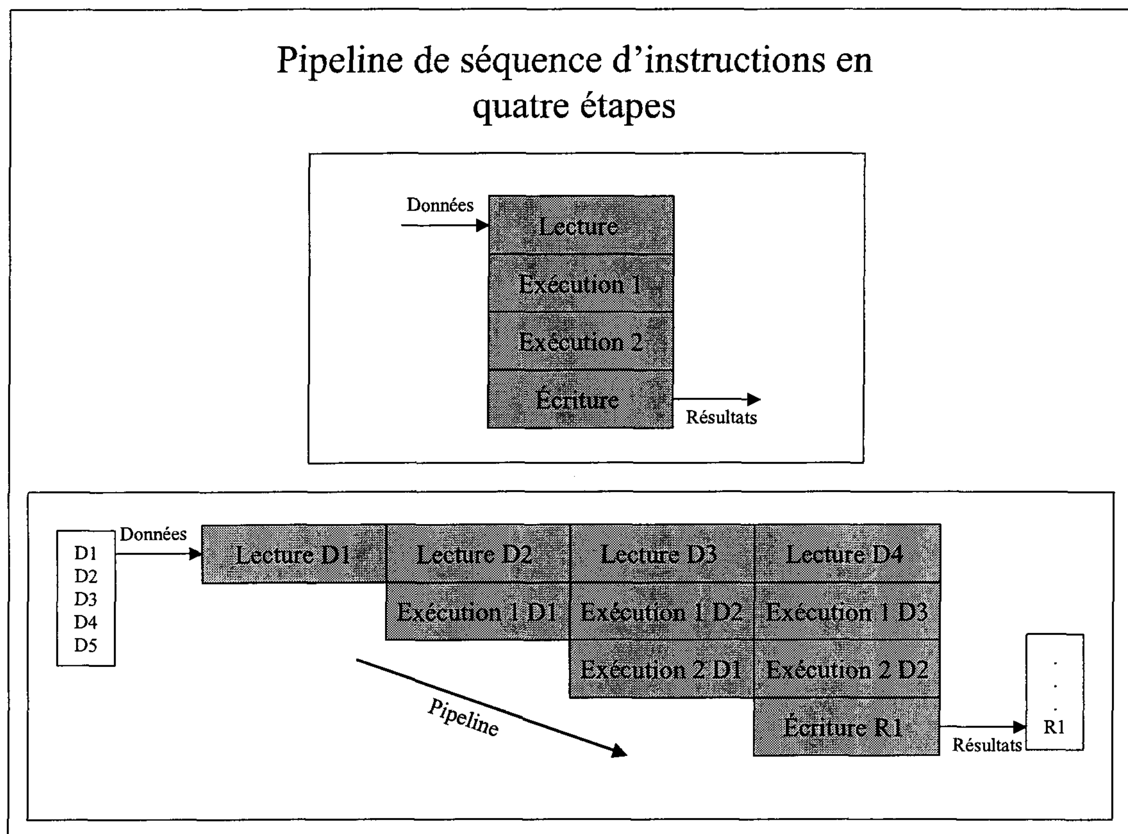


Figure 3.1 : Pipeline de PULSE

La prévention des aléas se fait par l'insertion de trois instructions entre deux instructions travaillant sur les mêmes emplacements mémoires.

3.1.1.3. Instruction vide NOPs

Lorsqu'un nombre insuffisant d'instructions utiles est disponible pour insertion entre deux instructions comportant une dépendance de données, il faut alors utiliser des instructions tampons (*NOP* - No OPeration). Le compilateur **PULSE** vérifie

automatiquement les dépendances de données entre les variables et instructions et insère par défaut le nombre de *NOP*s nécessaires pour éviter que ce problème se produise à l'exécution.

Un problème rencontré avec le compilateur est le fait qu'il ne discrimine pas les adresses lors d'accès aux mémoires a et b avec les compteurs modulo internes. Dans ce cas particulier, il faut encadrer les instructions cibles des étiquettes *.nodep* suivit de *.dep* pour signifier au compilateur de ne pas se soucier de la dépendance de données sur les instructions comprises entre ces deux étiquettes.

3.1.1.4. Efficacité et Utilisation

L'utilisation des instructions tampons nous mène à une définition de l'efficacité et de l'utilisation :

L'efficacité est la fraction de temps de traitement total qui est utilisée pour effectuer un travail utile. Avec les informations obtenues du simulateur de **PULSE**, le calcul de l'efficacité semble simple ; il suffit de soustraire le pourcentage de *NOP* du travail total. Cela n'est pas totalement vrai ; le résultat de cette soustraction nous donne le pourcentage d'**utilisation** de notre système. L'utilisation est le pourcentage de temps où les processeurs sont occupés à effectuer un travail. Le temps restant est du temps inexploité (idle time). Mathématiquement, l'utilisation est décrite comme le temps où les processeurs travaillent divisé par le temps total multiplié par le nombre de processeurs. Ce résultat est toujours une fraction comprise entre 0 et 1. Un haut pourcentage d'utilisation ne correspond pas nécessairement à une utilisation efficace.

L'efficacité est une mesure subjective et sa mesure est difficile. Il n'existe pas de «calculateur d'efficacité» pour quantifier cette variable. Dans le cas de **PULSE** et des applications étudiées dans ce mémoire, l'efficacité est le résultat de la rapidité de traitement de **PULSE** associée à la capacité d'optimisation des programmes par la personne qui écrit l'application.

En résumé :

- L'efficacité est la fraction du temps total qui est utilisée pour effectuer un travail utile.
- Une utilisation élevée n'implique pas le même résultat pour l'efficacité. Il n'y a pas de relation directe entre les deux.

3.1.2. Méthodologies adoptées

Cette section est tirée en grande partie d'un document produit par Ivan Kraljic membre du Groupe **PULSE** [KRAJ97].

3.1.2.1. Décomposition des algorithmes

L'objectif d'une excellente décomposition est d'occuper toutes les parties du circuit intégré avec des opérations concurrentielles effectuées à l'intérieur du même cycle.

Fondamentalement, le traitement d'un algorithme peut se décomposer en trois étapes :

1. entrer les données
2. traiter les données (calcul)
3. sortir les résultats

PULSE étant une machine de type **SIMD**, la première tâche associée à la répartition d'un algorithme est de décomposer l'algorithme sur les ***PE***s. Typiquement, deux techniques de décomposition peuvent être définies (figure 3.2) :

- **Décomposition basée sur les données (fig. 3.2a) :** Chaque processeur traite de façon complète son propre bloc de données. Il n'y a alors aucune interaction avec les voisins dans ce type de décomposition. Par exemple, dans le cas d'une image, chaque ***PE*** traitera 25% de l'image complète sans interaction avec les voisins.
- **Décomposition basée sur l'architecture (fig. 3.2b) :** Cette technique s'appuie sur le fait qu'une tâche de grande dimension peut être décomposée en de multiples tâches plus petites. Chaque élément de calcul produit un résultat partiel. Les résultats partiels peuvent ensuite être combinés ensemble pour produire le résultat final. Le flot de données et les résultats partiels doivent être communiqués entre les processeurs.

Pour la première application étudiée, **multiplication vecteurs-matrice**, l'algorithme a été décomposé selon les deux différentes façons dans un but d'apprentissage. Pour les algorithmes restant, la décomposition a été basée sur les données.

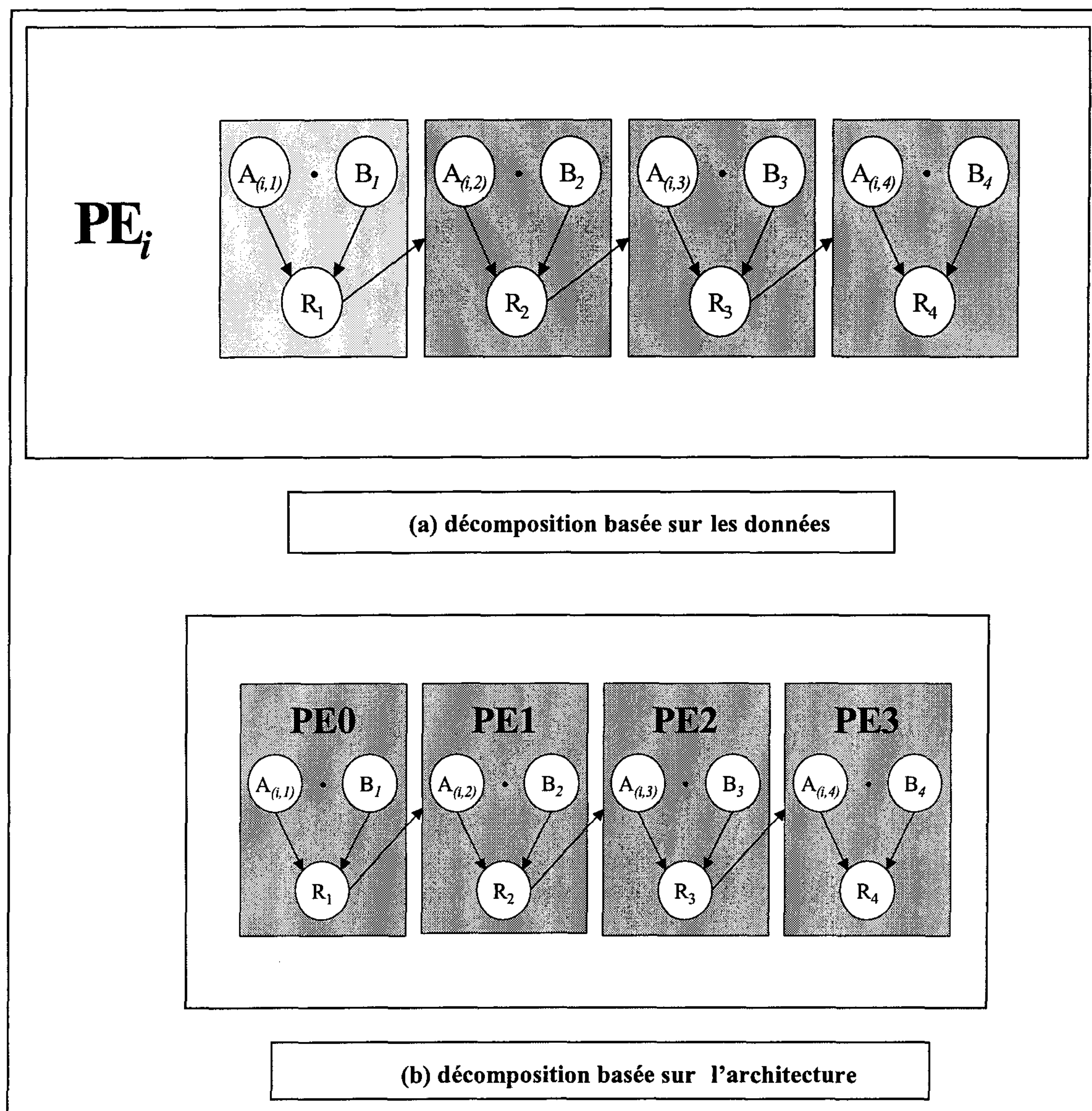


Figure 3.2 : Décomposition d'un algorithme

3.1.2.2. Opérations d'Entrées/Sorties(E/S)

Les opérations d'Entrées/Sorties sont responsables de la circulation des données et résultats entre les *PE* et les mémoires externes à travers des ports de communications. Ces

mémoires sont accessibles via des compteurs modulo. Les données en provenance de la mémoire externe peuvent être traitées directement à partir du canal d'entrée ("On the fly") ou emmagasinées dans la mémoire locale de chaque *PE* pour être ensuite traitées.

Le choix du flux de données pour les opérations d'Entrés/Sorties est une des tâches les plus importantes dans le processus de répartition d'un algorithme sur **PULSE**. La performance peut être grandement affectée par la décision relative au format du flux de données. Le fait qu'un simple load (*ld* src dst) ne puisse pas se faire en 1 cycle d'horloge est un facteur important.

3.1.2.3. Optimisation

L'optimisation ultime est de pouvoir faire travailler en tout temps les quatre processeurs tout en effectuant le maximum d'opérations en parallèle. L'objectif est d'utiliser un nombre de cycles d'horloge minimum pour exécuter chaque algorithme.

3.1.3. Environnement de développement

Le développement s'est fait avec trois outils :

- Au début de 1997 un compilateur était disponible. Il produit, à partir d'un fichier codé en assembleur, différents fichiers qui permettent de valider l'écriture du fichier source. Une version pour chacune des plates formes Unix et Windows NT est disponible.
- Simultanément, un simulateur basé sur le modèle **VHDL** de **PULSE V1** était accessible à l'École Polytechnique de Montréal via l'interface du fureteur Netscape®. Cet outil permet de vérifier la fonctionnalité des programmes développés et leurs performances. Ce simulateur produit un fichier résultant de la simulation qu'il est possible de

visualiser avec l'interface du fureteur. La validation se fait pas à pas en vérifiant le contenu des différents registres ainsi que les états dans lequel se trouve la machine.

- Le mois de décembre 1997 marque l'arrivée d'une version Bêta d'un simulateur exportable. Ce simulateur fonctionne sur UNIX et produit des fichiers texte résultat (.res) pour chacun des ***PE***.

3.1.4. Évaluation de la performance, du débit et de l'efficacité

La performance des applications est mesurée en nombre de cycles nécessaires pour traiter une donnée, un pixel ou un groupe de données (appelé donnée dans ce qui suit), dépendant de l'application décrite. Cette mesure est obtenue en divisant le nombre total de cycles nécessaires par le nombre de données traitées. Il s'agit là d'une mesure globale de la performance de **PULSE**, cette mesure nous permet d'estimer le temps nécessaire pour traiter les différentes quantités de données. Une performance de p cycles / donnée pour un ***n-PE*** **PULSE** correspond à une performance de p / n cycles / donnée.

Le débit mesure le nombre d'unité traitée par le système par unité de temps. L'unité est composée de l'objet source, soit par exemple des pixels dans le cas d'une image ou de vecteurs dans le cas de multiplication de matrices.

L'efficacité locale de **PULSE** peut être estimée en comparant sa performance avec deux limites :

1. **Limite indépendante de l'architecture** : le nombre de cycles d'exécution de l'algorithme est calculé sans dépendance sur l'architecture de **PULSE**. Il n'est pas possible d'obtenir une performance plus élevée que cette limite.
2. **Limite dépendante de l'architecture** : le nombre de cycles d'exécution de l'algorithme est calculé en tenant compte des limites architecturales de **PULSE**. Cette dépendance prend en considération la dépendance de données : une dépendance de données peut requérir jusqu'à 3 cycles additionnels avant de pouvoir effectuer une autre opération.

Les deux limites précédentes sont calculées en supposant que les donnés sont disponibles pour traitement. Les résultats de développement obtenus prennent en considération les instructions d'entrée et de sortie de données, si elles sont nécessaires.

L'efficacité de l'application sera aussi calculée dans le prochain chapitre en comparant nos résultats contre ceux obtenus sur d'autres circuits multiprocesseurs pour le même type d'applications.

Les résultats de performance, débit, utilisation et efficacité pour chacune des applications sont mises en tableau dans le prochain chapitre.

3.1.5. Goulot d'étranglement

Pour chacune des applications développées, la limite de performance atteinte peut dépendre de multiples facteurs; disponibilité des données, communication inter-processeur,

absences d'une combinaison particulière d'instruction parallèle, dimension de la mémoire, etc. Chacun des facteurs cités peut ralentir l'application dans son exécution, il s'agit de goulots d'étranglement. Un des objectifs de l'optimisation est de pouvoir ouvrir ces goulots aussi grand que possible. L'ouverture d'un goulot d'étranglement ne fait que repousser la limite de performance vers un autre qui devra, à son tour, être ouvert. Il en est ainsi jusqu'à l'atteinte d'une limite établie par différents facteurs : technologie, coût, temps, etc. Il existe toujours un goulot d'étranglement.

Pour chacune des applications, les goulots d'étranglement probables seront identifiés.

3.2. APPLICATIONS DÉVELOPPÉES

Le reste de ce chapitre décrit de façon générale les applications développées ainsi que les résultats de la décomposition pour chacun des algorithmes. La première application contient des parties de code développées placées en encadré. Pour les autres applications, une copie du code se retrouve en Annexe (Annexe A).

3.2.1. Multiplication vecteurs - matrice

Le but de cette première application était d'utiliser les différents éléments et outils de **PULSE** afin de se familiariser avec ces derniers, pour éventuellement développer des applications plus complexes. Un autre objectif était d'apprivoiser le fonctionnement de l'architecture interne de **PULSE** et finalement de tenter d'arriver avec une solution performante pour résoudre une application numérique simple et classique.

La matrice est de format carré, avec n = 4. Le choix est dans un but pratique, car chaque processeur **PULSE V1** possède quatre ***PE***. Le vecteur est évidemment de format 4 x 1. Ces vecteurs sont équivalents à un vecteur colonne utilisé pour spécifier un point dans un système de coordonnées homogènes 3D dans les applications graphiques.

A [4 x 4] • B [4 x 1] = C [4 x 1]

3.2.1.1. Répartition sur PULSE

Plus dans un but de connaissance personnelle que de recherche de performance ultime, une version pour chacun des deux types de décomposition vus précédemment a été développée. Il s'agit de la seule application décomposée sur l'architecture ayant été réalisée.

3.2.1.2. Limite de vitesse basée sur architecture

La limite indépendante de l'architecture est de 28 instructions / vecteur ; 16 multiplications et 12 additions. La limite est de 28 cycles / vecteur par ***PE***, soit 6.5 cycles / vecteur pour un circuit intégré à 4 ***PE***.

Matrice • **Vecteur** = **Sortie**

$$\begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{bmatrix} \quad \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} \quad \begin{bmatrix} w' \\ x' \\ y' \\ z' \end{bmatrix}$$

En tenant compte de la dépendance de données et de l'architecture de **PULSE,** un vecteur se traite de la façon suivante (les lettres majuscules représentent des résultats intermédiaires):

$a * w \rightarrow I_1$
$I_1 + e * x \rightarrow I_2$
$I_2 + i * y \rightarrow I_3$
$I_3 + m * z \rightarrow w'$

$b * w \rightarrow J_1$
$J_1 + f * x \rightarrow J_2$
$J_2 + j * y \rightarrow J_3$
$J_3 + n * z \rightarrow x'$

$c * w \rightarrow K_1$
$K_1 + g * x \rightarrow K_2$
$K_2 + k * y \rightarrow K_3$
$K_3 + o * z \rightarrow y'$

$d * w \rightarrow L_1$
$L_1 + h * x \rightarrow L_2$
$L_2 + l * y \rightarrow L_3$
$L_3 + p * z \rightarrow z'$

Encadré 3.1 : Opérations : multiplication vecteur - matrice

Pour chaque vecteur, la dépendance architecturale est égale à 16 cycles / vecteur pour un ***PE***. La dépendance est de 4 cycles / vecteur pour un circuit intégré à 4 ***PE***s.

3.2.1.3. Décomposition basée sur les Données

Chacun des quatre processeurs de **PULSE V1** traite un groupe de données et produit un résultat complet au terme de l'algorithme. Sur chaque processeur, il y a addition et multiplication des termes propres à chaque ***PE***.

$$\mathbf{PE_i:}\quad \boldsymbol{A_{i1} \bullet B_1} + \boldsymbol{A_{i2} \bullet B_2} + \boldsymbol{A_{i3} \bullet B_3} + \boldsymbol{A_{i4} \bullet B_4} = \mathbf{C_I}$$

La matrice A est divisée par ligne entre les quatre ***PE***s.

Des résultats pour trois exemples de traitement sont disponibles :

1. Un terme du vecteur est chargé et il est traité simultanément dans les quatre ***PE***.
2. Les vecteurs sont chargés dans la mémoire interne à l'initialisation (256 vecteurs).
3. Le vecteur complet est chargé et chaque ***PE*** traite un terme différent simultanément.

Organisation des données

Les données sont de format 16 bits et se trouvent dans une mémoire externe. Elles sont lues et emmagasinées dans l'ordre suivant pour le premier cas :

# ligne[15]	Description
0-15	matrice A, colonne après colonne (a, b, c,..., n, o, p)
16-∞	vecteurs B (w, x, y z)

Tableau 3.1 : Vecteurs – Matrice - Organisation des données, mémoire d'entrée

Après lecture des 16 premières données (matrice A), chacun des ***PE***s a une ligne de la matrice A dans sa mémoire A. Les différents vecteurs sont ensuite traités un après l'autre «on the fly» à partir des registres du port Nord.

Pour le deuxième cas, la matrice A doit se trouver entre les positions 1 et 16 de la mémoire externe et les vecteurs à partir de l'espace mémoire no 17. Le fait que les compteurs modulo[16] ne repassent jamais par le minimum rend ceci nécessaire. La position 0 de la mémoire externe n'est pas accédée.

---

[15] Le simulateur via interface Netscape commence à lire seulement à partir de la quatrième ligne du fichier de données, les trois premières lignes servent à définir la dimension d'un fichier image.

[16] Changé le 23 septembre 1997, les compteurs modulos repassent maintenant par le minimum.

Pour le troisième cas, les vecteurs sont situés en ordre dans la mémoire A de 0 à un maximum de 256.

Matrice [4 x 4] avec vecteur colonne [4 x 1] (première version)

L'algorithme est le suivant :

1) chargement de la matrice A dans la mémoire locale de **PULSE** (memA)
2) Pour chaque vecteur :
   a) pour chaque terme du vecteur :
      i) mettre la valeur dans les quatre registres du port Nord
      ii) multiplication de la paire de termes correspondants ($A_{ij} \bullet B_{j1}$) et addition au terme précédent
      iii) Sortie des résultats sur les canaux Nord (low16-bit) et Sud (high 16-bit)

Le flot des opérations à travers chaque ***PE*** et la position des valeurs traitées se trouvent dans le tableau 3.2.

flot →

	I_1		I_2		I_3		I_4		
PE	$A_{i1} \bullet B_1$		$A_{i2} \bullet B_2$		$A_{i3} \bullet B_3$		$A_{i4} \bullet B_4$		Résultats
3	a • w	+	e • x	+	i • y	+	m • z	=	w'
2	b • w	+	f • x	+	j • y	+	n • z	=	x'
1	c • w	+	g • x	+	k • y	+	o • z	=	y'
0	d • w	+	h • x	+	l • y	+	p • z	=	z'

Tableau 3.2 : Vecteurs – Matrice - Traitement des données dans chaque UT

Version de base

Le programme se trouve dans l'encadré 3.2 :

```
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ;         matrice A
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        push 4; nombre de lignes
LoadA:
        #4 nsr || io *mccr%  ; remplissage du canal Nord avec une ligne
        ld nport, *mcaw(1) ; matrice A charge la colonne i dans le PEi
        dbr LoadA

    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ;               traitement des vecteurs
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        #4 nsr ; remplissage du canal Nord : premier terme
        push Nb_vecteur ; nombre de vecteurs à traiter
Vect:
        push 4; chaque vecteur a quatre termes à traiter
V_Term:
        macc nport, *mcar(1) || io *mccr%; multiplication addition
        #3 nsr ; remplissage du canal Nord avec le vecteur Ai
        nsr
        dbr V_Term

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; sortie des résultats et entrée nouveau vecteur
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ld acc, nsport
        ld 0, acc                    ; réinitialise accumulateur
        #3 nsr || ssr || io *mcdr%
        nsr || ssr || io *mcdr%
        dbr Vect
```

Encadré 3.2 : Multiplication vecteurs-matrice, version de base

Boucles déroulées

Pour augmenter légèrement la performance, les boucles pour l'entrée en mémoire de la matrice A et pour le chargement du vecteur ont été déroulées. Le nombre de cycles nécessaires se trouve dans le tableau 3.4.

Version avec données dans les mémoires internes

Une version a été développée pour traiter un nombre de vecteurs inférieur ou égal à 256. Les données sont déjà en mémoire (mémoire A et B, 512 mots chacune). Les registres des canaux Nord et Sud sont inutilisés lors du traitement des données. Ils peuvent être utilisés pour sortir les résultats de la multiplication.

1) Pour chaque vecteur :

   a) pour chaque terme du vecteur :

      i) multiplication de la paire de termes correspondant ($A_{ij} \bullet B_{j1}$) et addition au terme précédent

      ii) Sortie des résultats sur les canaux Nord (low16-bit) et Sud (high 16-bit) simultanément

Le flot des opérations à travers chaque ***PE*** et la position des valeurs traitées sont les mêmes que précédemment.

```
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;                traitement des vecteurs
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        push Nb_vecteur; nombre de vecteurs à traiter
Vect:
        mult ra1, *mcar(1), acc ; init. accumulateur, 1er multiplication
        macc ra2, *mcbr(1)
        macc ra3, *mcar(1)
        macc ra4, *mcbr(1)|| io *mcdr% ;sortie première données
        #3 nsr || ssr || io *mcdr%     ;sortie autres résultats
        ld acc, nsport  ; chargement résultat dans registres
       dbr Vect
        io *mcdr% ;sortie première données
        #3 nsr || ssr || io *mcdr%     ;sortie dernier résultat
```

Encadré 3.3 : Multiplication vecteurs-matrice, données dans la mémoire interne

Matrice [4 x 4] avec vecteur colonne [4 x 1] (pseudo-systolique)

L'algorithme de traitement est similaire, la différence se situe dans le stockage des données dans la mémoire interne et dans la façon dont les données du vecteur sont traitées.

1) chargement de la matrice A dans la mémoire locale de **PULSE** (memA) par pas de cinq
2) Pour chaque vecteur :
   a) chargement du vecteur dans les registres du port Nord
   b) faire quatre fois
      i) multiplication de la paire de termes correspondants (A_{ij}•B_{j1}) et addition au terme précédent
      ii) rotation des valeurs dans les registres du port Nord
   c) Sortie des résultats sur les canaux Nord (low 16-bit) et Sud (high 16-bit)

Le flot des opérations à travers chaque ***PE*** et la position des valeurs traitées se trouvent dans le tableau 3.3. La figure 3.3 montre avec lesquelles des valeurs s'effectuent les multiplications-additions dans chacun des ***PE***s.

flot →

	I_1		I_2		I_3		I_4		
PE	$A_{i1} \bullet B_1$		$A_{i2} \bullet B_2$		$A_{i3} \bullet B_3$		$A_{i4} \bullet B_4$		Résultats
3	a • w	+	e • x	+	i • y	+	m • z	=	w'
2	f • x	+	j • y	+	n • z	+	b • w	=	x'
1	k • y	+	o • z	+	c • w	+	g • x	=	y'
0	p • z	+	d • w	+	h • x	+	l • y	=	z'

Tableau 3.3 : Vecteurs – Matrice - Traitement des données dans chaque PE

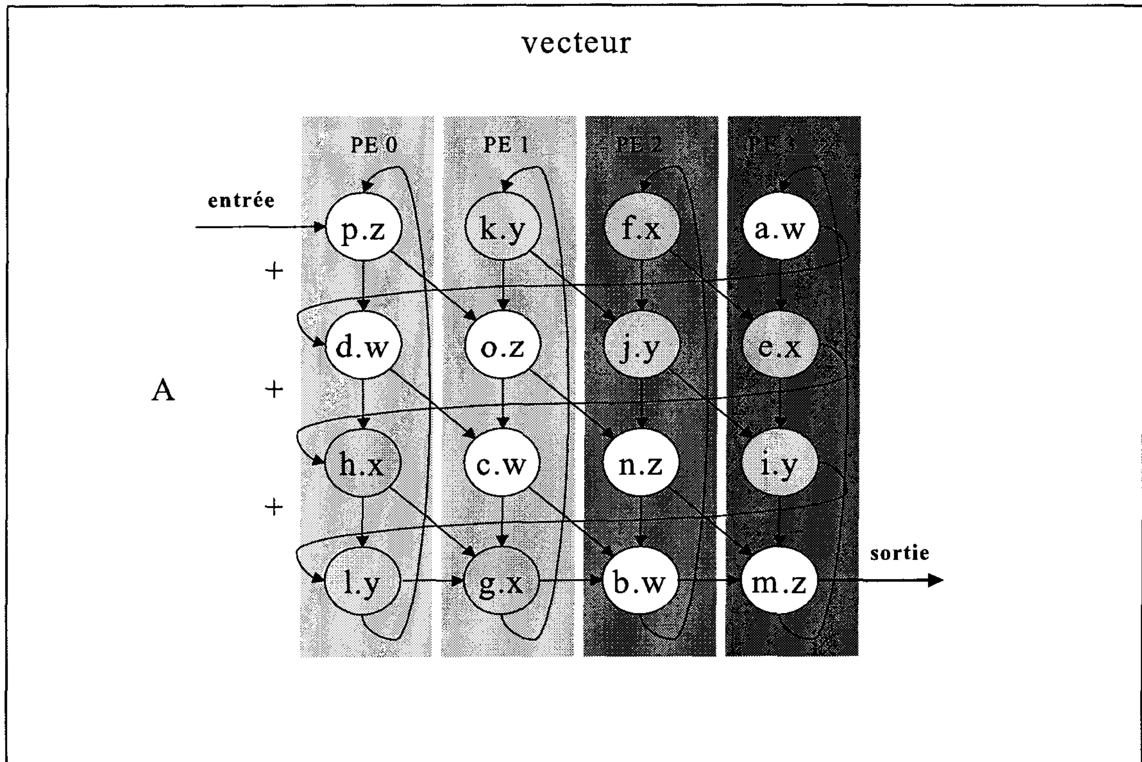


Figure 3.3 : Vecteurs – Matrice - Mouvement de données et opérations

Le compteur modulo c est initialisé une première fois pour la lecture de la matrice A (pas de 5), puis le pas est mis à 1 pour la lecture des vecteurs. La position 0 n'est pas utilisée. Le fait que les compteurs modulo ne repassent jamais par le minimum rend ceci nécessaire.

```
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;               traitement de la matrice A
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        push 4; nombre de lignes
LoadA:
        #4 nsr || io *mccr%  ; remplissage du canal Nord avec une ligne
        ld nport, *mcaw(1) ; matrice A charge la colonne i dans le PEi
        dbr LoadA

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;               traitement des vecteurs
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; réinitialisation compteur modulo c
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ldeamc 20000h, 0220000h, mc_max, mc_max;
        ldeamc 17, 0200000h, mc_start, mc_start;
        ldeamc 1, 1, mc_stride, mc_stride;
        ld 0, acc || io *mccr%, *mcdr%;toujours ajouter après chargement

        #3 nsr || io *mccr% ;premier vecteur
        macc nport, *mcar(1) || nsr || io *mccr%

        push Nb_vecteur; nombre de vecteurs à traiter
Vect:        ; multiplication addition vecteur
        nrr
        macc nport, *mcar(1)
        nrr
        macc nport, *mcar(1)
        nrr
        macc nport, *mcar(1)
  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ;               fin de traitement du vecteur
  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ldiamc 1, 1, 4, mcar;
  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ;               sortie des résultats
  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ld acc, nsport
        ld 0, acc
        #3 nsr || ssr || io *mcdr%, *mccr%
        macc nport, *mcar(1) || nsr || ssr || io *mcdr%, *mccr%
        dbr Vect
```

Encadré 3.4 : Multiplication vecteur-matrice, version pseudo-systolique

Une autre modification possible est l'insertion d'une multiplication en parallèle avec l'entrée/sortie d'une donnée sur la dernière ligne. Les opérations sur les registres prennent un cycle alors que le *macc* en prend quatre. Ceci rend possible la mise en parallèle d'une opération sur une donnée présente seulement le cycle suivant. Cette opération enlève un cycle pour chaque vecteur. Elle peut être mise en place dans la première structure étudiée dans ce document.

Nombre de cycles

Le tableau 3.4 présente le nombre de cycles nécessaire pour chacune des opérations lors de la décomposition de l'algorithme de multiplication vecteurs – matrice.

	opération	# de cycles totaux	# de NOP
Commun	initialisation	14	-
	push nb vect.	1	-
	fin programme	7	6
Total		22	6

Version	traitement	# de cycles totaux	# de NOP
de base	matrice A	25	-
	vecteur B	4 + (nb_vecteur • 25)	-
Déroulée	matrice A	20	-
	vecteur B	4 + (nb_vecteur • 16)	3
Systolique	vecteur B	9 + (nb_vecteur • 9)	3
avec macc ǁ nrr	vecteur B	9 + (nb_vecteur • 6)	3
Chaque vecteur	sortie données	8[17]	2
	branchement	1	-
Mémoire interne	calcul / io	nb_vecteur • 8	-
	branchement	1	-

Tableau 3.4 : Vecteurs – Matrice - Nombre de cycles par vecteur

[17] additionner au traitement du vecteur B

3.2.1.4. Décomposition basée sur l'architecture

Chaque ***PE*** de **PULSE** calcule un résultat partiel. Les résultats intermédiaires circulent entre les ***PE***s de la gauche vers la droite. Au terme de l'algorithme, le résultat final est produit par le ***PE*** situé à l'extrême droite. Chaque processeur fait une des quatre multiplications et les résultats sont mis en cascade et additionnés d'un ***PE*** à l'autre.

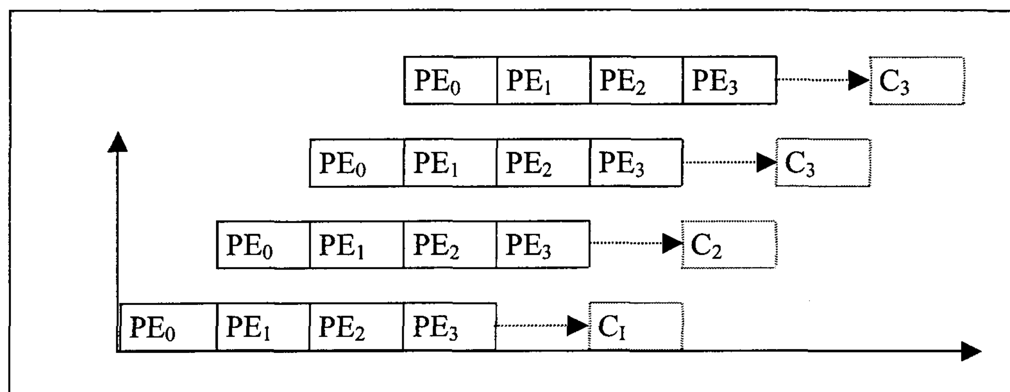


Figure 3.4 : Vecteurs – Matrice - Remplissage du pipeline et sortie des résultats

Après que le pipeline créé par le canal de communication entre les accumulateurs des ***PE***s est rempli, il y a production d'un résultat à chaque multiplication-addition (C_i).

La matrice A est divisée par colonnes entre les quatre ***PE***s.

Le traitement du vecteur B se fait comme suit :

1) Chaque ***PE*** traite un des quatre termes du vecteur.
2) Après la production du premier résultat à chaque multiplication subséquente, un terme du vecteur suivant doit être entré dans la chaîne de traitement (registres du port Nord) et positionné pour le bon ***PE***.

Données

Comme dans le cas précédent, la matrice A doit se trouver entre les positions 1 et 16 de la mémoire externe et les vecteurs sont emmagasinés à compter de l'espace mémoire no 17.

Version 3

L'algorithme de traitement est presque le même, la différence se situe dans le stockage des données dans la mémoire interne et dans la façon dont les données du vecteur sont traitées. La figure 3.6 illustre le mouvement des données et les opérations.

1) chargement de la matrice A dans la mémoire locale de **PULSE** (memA)
2) Pour le premier vecteur (remplissage du pipeline)
   a) charger les quatre valeurs dans les registres du port Nord
   b) faire quatre fois
      i) multiplication de la paire de termes correspondants ($A_{ij} \bullet B_{j1}$) et addition au terme produit par le ***PE*** de gauche avec la chaîne d'accumulation entre chaque ***PE***. (figure 3.6(a-d))
   c) sortie du résultat (figure 3.6(e))
3) Pour les autres vecteurs
   a) charger un terme dans le registre correspondant du port Nord (figure 3.6(e))

b) multiplication de la paire de termes correspondants ($A_{ij} \bullet B_{j1}$) et addition au terme produit par le ***PE*** de gauche à l'aide de la chaîne d'accumulation entre chaque ***PE***. (figure 3.6(e))

c) sortie des résultats (figure 3.6(e))

La figure 3.5 montre de quelle façon s'effectue la cascade des multiplications-additions dans les quatre ***PE***s. La figure montre aussi dans quels registres sont entrés les nouveaux termes du vecteur et dans quel ordre sont stockés les membres de la matrice A dans la mémoire interne de **PULSE V1**.

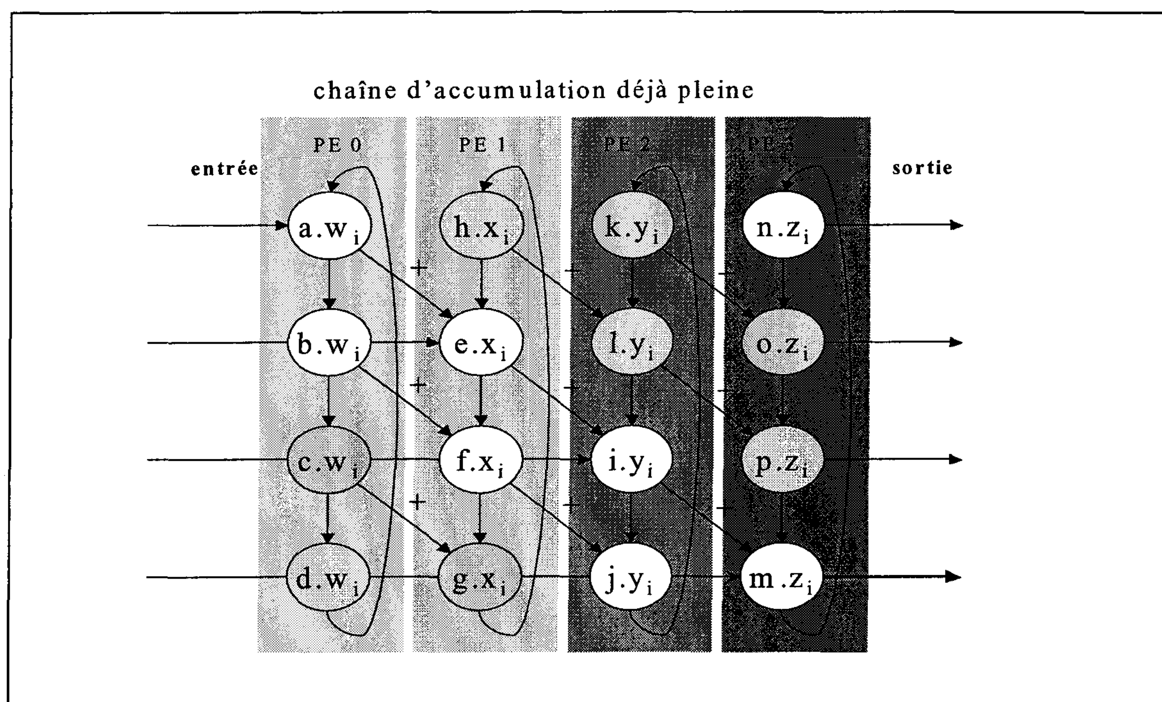
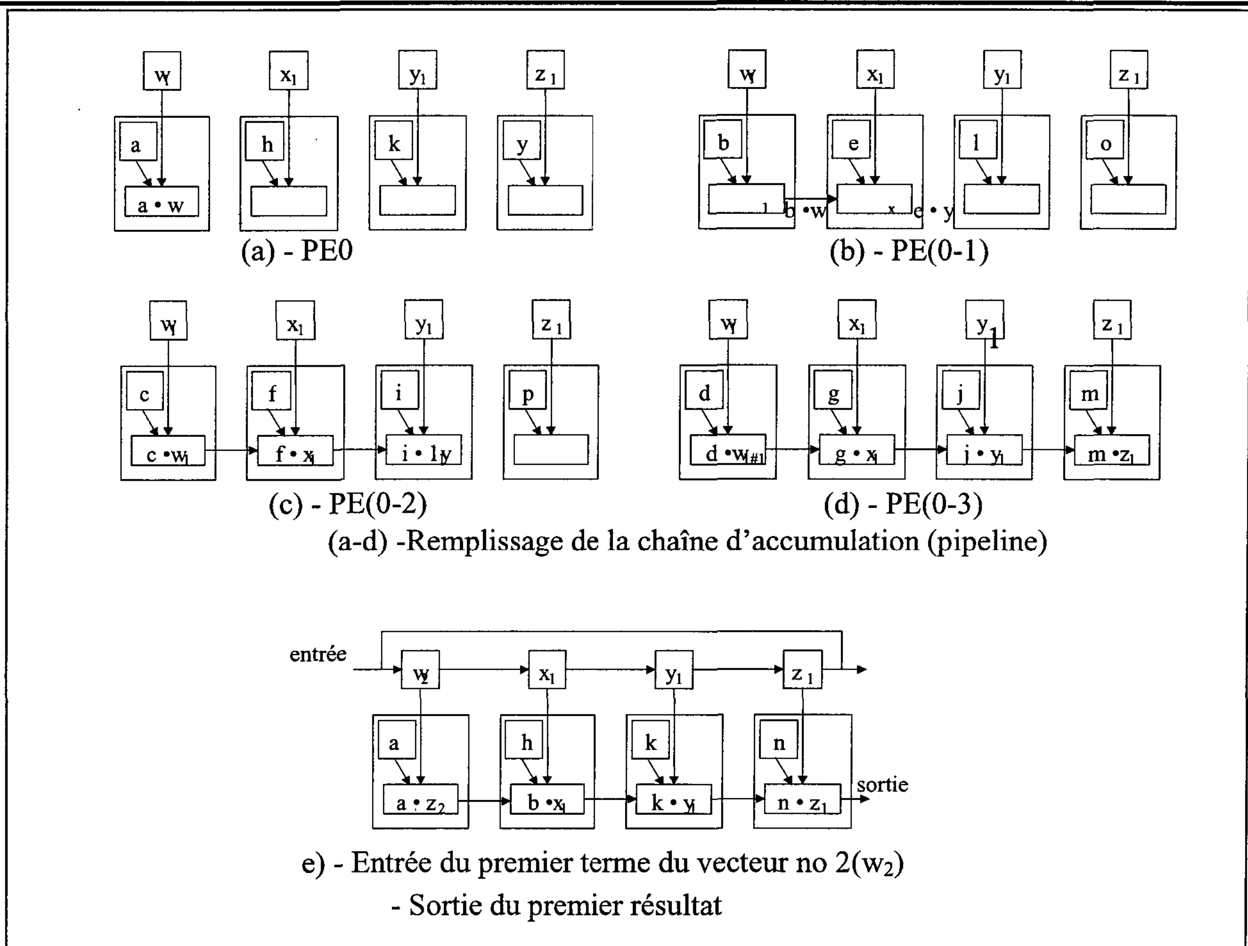


Figure 3.5 : Entrée des données, opérations et sortie des résultats



(a) - PE0

(b) - PE(0-1)

(c) - PE(0-2)

(d) - PE(0-3)

(a-d) -Remplissage de la chaîne d'accumulation (pipeline)

e) - Entrée du premier terme du vecteur no 2(w_2)

- Sortie du premier résultat

Figure 3.6 : Vecteurs – Matrice - Décomposition basée sur l'architecture

L'encadré 3.5 contient le programme résultant.

```
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;              remplissage du pipeline
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      mult nport, *mcar(1), acc
      maddl nport, *mcar(1), acc, acc

      maddl nport, *mcar(1), acc, acc
      maddl nport, *mcar(1), acc, acc  ; le premier résultat est prêt
                                       ; le w doit être changé
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;              traitement des vecteurs suivants
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      push Nb_vecteur; nombre de vecteurs à traiter
Vect:
      #2 nrr   ; positionnement de z dans le shift register du PE3
      nrr || io *mcdr%; sortie du premier résultat et placement de w
      maddl nport, *mcar(1), acc, acc || nsr ||io *mccr% ;entrer de wi
                                                         ;résultat a
sortir
                                                         ;le x doit
changer
                                                         ; le modulo xi
      #2 nrr   ; positionnement de x dans le shift register du PE3
      nsr || io *mccr%      ; entrer de xi
                            ; le modulo pointe sur yi
      nrr || io *mcdr%      ; sortie du deuxième résultat
                            ; retour en place
      maddl nport, *mcar(1), acc, acc  ;résultat à sortir
                                       ;le y doit être changé
      nrr      ; positionnement de y dans le registre de décalage PE3
      nsr || io *mccr%      ; entrer de yi
                            ; le modulo pointe sur zi
      nrr
      nrr || io *mcdr%      ; sortie du troisième résultat
                            ; retour en place
      maddl nport, *mcar(1), acc, acc  ;résultat à sortir
                                       ;le z doit être changé
      nsr || io *mccr%      ; entrer de zi
                            ; le modulo pointe sur wi
      #2 nrr   ; retour en place
      nrr || io *mcdr%      ; sortie du quatrième résultat
                            ; retour en place
      maddl nport, *mcar(1), acc, acc  ;résultat à sortir
                                       ;le w doit être changé
      dbr Vect
      end
      .end
```

Encadré 3.5 : Multiplication vecteurs-matrice, version pseudo-systolique

Accélération

Comme dans le cas précédent, l'ajout de l'instruction parallèle macc || nrr permettrait de retrancher 3 cycles à chaque vecteur traité. Dans ce cas, on arriverait à une solution optimale pour le traitement d'un vecteur de quatre termes.

```
multiplication-addition PE0
#3 NOP
multiplication-addition PE1
#3 NOP
multiplication-addition PE2
#3 NOP
multiplication-addition PE3 → sortie du résultat
#3 NOP
```

Encadré 3.6 : Opérations – utilisation nouvelle combinaison instruction parallèle

Les *NOP*s entre chaque instruction sont dus à une dépendance de données. Chaque ***PE*** doit attendre l'exécution complète de la multiplication du ***PE*** précédent afin de pouvoir ajouter le résultat à sa propre multiplication. Ces cycles morts sont utilisés pour faire entrer et circuler les termes des vecteurs sur le canal Nord.

Nombre de cycles

Le tableau 3.5 donne le nombre de cycles (real instructions) :

	opération	# de cycles totaux	# de NOP
Commun	initialisation	19	-
	push nb vect.	1	-
	fin programme	7	6
Total		27	6
Version	traitement	# de cycles totaux	# de NOP
de base	1er vecteur	14	9
	vecteurs	nb_vecteur • 19	-
avec macc || nrr	vecteurs	nb_vecteur • 16	-
Chaque vecteur	branchement	1	-

Tableau 3.5 : Vecteurs – Matrice - Nombre de cycles par vecteur

PULSE V2

L'emploi de 16 processeurs ne garantirait pas un facteur d'accélération de 4 par rapport à **PULSE V1**. La circulation à l'intérieur des canaux Nord et Sud avec 16 registres sur le même canal apporte une complexité accrue. De plus pour remplir ou vider un canal, il faut aussi quatre fois plus d'opérations. Il faudrait alors un minimum de 16 opérations de traitement pour pouvoir effectuer la circulation sur les canaux sans ajouter de temps mort entre le traitement de chaque groupe de vecteur.

3.2.2. Algorithme de Bresenham

3.2.2.1. Description de l'algorithme

En 1965, J.E. Bresenham [BREX65] présentait un algorithme qui augmentait de façon notable la rapidité avec laquelle une imprimante digitale pouvait tracer une ligne droite. Depuis sa publication, cet algorithme a été utilisé couramment lors de diverses opérations d'affichage à base de pixels.

La tâche de base d'un algorithme de traçage de lignes est de calculer les coordonnés du pixel qui est le plus près de la ligne réelle sur une grille à deux dimensions. L'algorithme de Bresenham gagne de l'intérêt dans le cas de **PULSE** du fait qu'il utilise exclusivement des opérations sur des valeurs entières.

Dans la figure 3.7, le point suivant le pixel tracé (représenté en noir plein) est d'un point de vue géométrique, dans le cas du premier quadrant, soit à droite, soit en diagonale en haut à droite. L'algorithme de Bresenham propose un critère de choix simple entre ces deux points. Il se base sur la distance verticale du point à la droite (d_1 et d_2). Cette distance peut être considérée comme une erreur d'approximation.

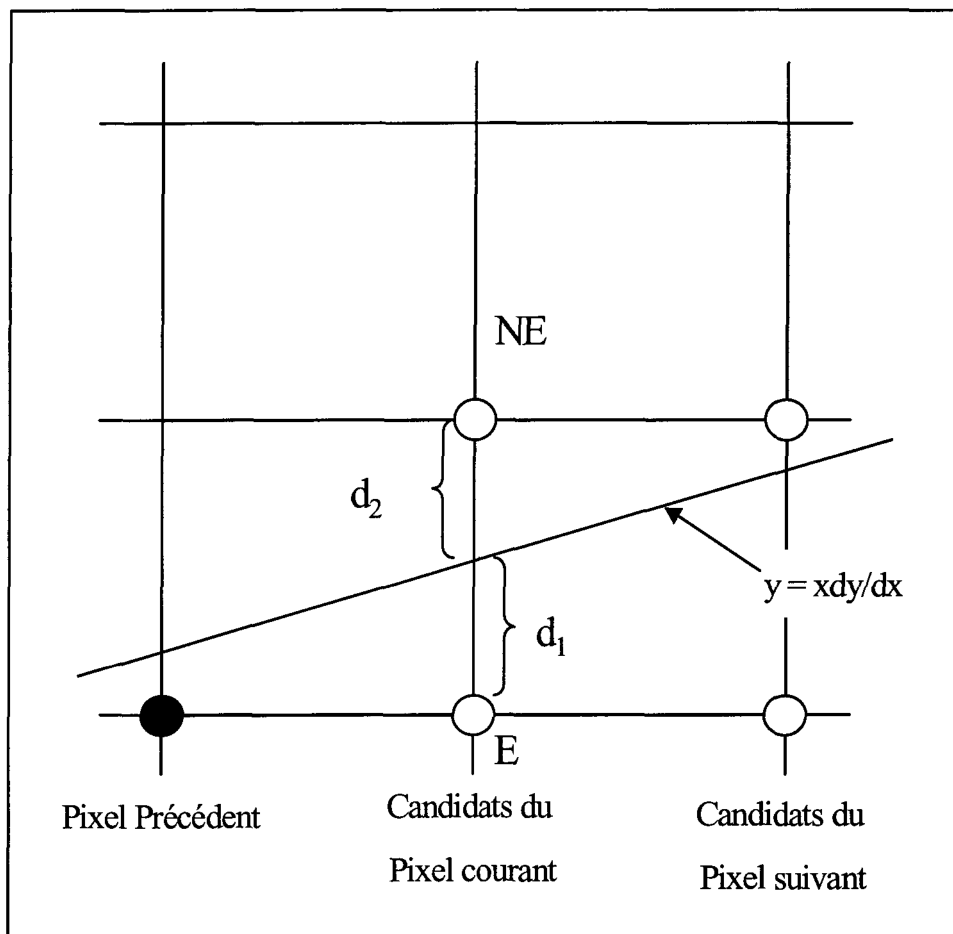


Figure 3.7 : Bresenham - Traçage de lignes

Nous assumons que la ligne se trouve dans le premier quadrant et que sa pente est comprise entre 0 et 1. L'algorithme utilise une variable de décision, **d**, qui entre chaque étape est proportionnelle à la différence entre $\mathbf{d_1}$ et $\mathbf{d_2}$. Si $\mathbf{d_1} < \mathbf{d_2}$, alors le point **E** est le plus proche de la droite, sinon le point **NE** devrait être le prochain pixel actif. De façon différente, **E** est choisi si $\mathbf{d_1} - \mathbf{d_2} < 0$, autrement **NE** est choisi. L'algorithme en pseudo-code est le suivant :

```
déterminer le point de départ de la droite, point avec la valeur de x la plus petite
calcul de dx
calcul de dy
calcul des constantes utilisées pour incrémenter la valeur du pixel précédent
  incr1 = 2 * dy
  incr2 = 2 * (dy- dx)
calcul de la valeur initiale de d = incr1 - dx
tant que x début < x dernier
  incrémenter x de 1
  si d < 0
          d = d + incr1
  sinon
          y = y + 1
          d = d + incr2
```

Encadré 3.7 : Bresenham - Pseudo-code

3.2.2.2. Répartition sur PULSE

1. Les limites de la droite sont entrées par les canaux Sud et Nord
2. Chaque ***PE*** s'occupe de traiter un segment de droite différent
3. Les coordonnés de la droite sont sorties par les canaux au fur et à mesure des traitements

Le nombre de segments de lignes est directement proportionnel aux nombres de ***PE***s disponibles dans **PULSE**.

3.2.2.3. Limite de vitesse basée sur architecture

Cet algorithme utilise une instruction conditionnelle afin de déterminer si la coordonnée en y doit être incrémentée ou non. Le problème lors de l'emploi d'instructions conditionnelles avec un système multiprocesseurs est rapidement évident. Les processeurs peuvent ne pas tous avoir des données qui remplissent les mêmes conditions. Ce qui fait que certains processeurs sont mis au repos pendant que les autres traitent une partie de la structure conditionnelle, pour être probablement réactivés par une autre partie de la structure. Le code associé à chacune des conditions doit donc être complété. Une instruction de **PULSE** permet de vérifier l'activité des processeurs. Elle permet de passer à la prochaine condition si aucun des processeurs n'est actif, ou simplement de poursuivre la boucle pour ensuite sauter à la fin de l'instruction conditionnelle si tous les processeurs se trouvent actifs. Le problème est la consommation d'instructions vides avant de pouvoir vérifier l'activité des processeurs : 2 cycles pour le *if* et 2 cycles pour le test sur l'activité. Le comportement et la latence des instructions conditionnelles ont déjà été expliqués plus en détails dans la section 1.4.3.2 – Caractéristiques du jeu d'instruction. Dans le cas présent, il y deux instructions maximum par boucle. Il est donc plus payant de passer à travers les deux cas peu importe l'activité des processeurs.

La limite indépendante de l'architecture est de 11 instructions/segment pour l'initialisation ; 3 soustractions, 2 valeurs absolues, 2 multiplications, 1 test, 2 égalisations. Pour chaque pixel du segment de droite, le nombre de cycles est de 3 pour une ligne horizontale ; 1 test, 2 additions et de 1 addition de plus par pixel dans le cas d'une droite avec une pente de 45 degrés. Donc, pour une droite de 10 pixels, le nombre de cycles total

est compris de 41 à 51 / cycles par segment. Pour un processeur à 4 ***PE***, la performance devrait être de 10.25 à 12.75 / cycles par segment.

En tenant compte de la dépendance de données, un segment de droite se traite de la façon suivante :

```
Cas(y1, y2) → @ 1, 32 bits sur mémoire A et B (dy = abs(y1 - y2) )
Cas(x1, x2) → @ 0, 32 bits sur mémoire A et B (dx = abs(x1 - x2) )
NOP ; deux cycles à cause d'une dépendance de données sur le résultat du Cas
NOP
A1 - B1 → ra3  (dy)
A0 - B0 → rb0  (dx)
NOP ; 2 cycles, dépendance de données sur ra3
NOP
Sla(r3, 1) → r1  (incr1: 2 * dy)
ra3 - rb0 → ra2  (incr2: dy - dx)
NOP ; 2 cycles, dépendance de données sur ra1
NOP
ra1 - rb0 → rb3  (d)
```

Encadré 3.8 : Bresenham - Opérations – traitement d'un segment de droite

L'initialisation des données des variables pour le calcul de la position des pixels prend donc 13 cycles par segment.

```
B0 + 1 → B0 ; x(1 + i)
if ( rb3 < 0 )
        NOP ; 2 cycles, dépendance sur le test
        NOP
        rb3 + ra1 → rb3 ; d = d + incr1
else
        NOP ; 2 cycles, dépendance sur le test
        NOP
        rb3 + ra2 → rb3 ; d = d + incr2
        B1 + 1 → B1 ; y(1 + i)
restore
NOP ; 2 cycles, dépendance pour réactiver tous les PEs
NOP
sortie des xi → canal Nord
sortie des yi → canal Sud
```

Encadré 3.9 : Bresenham - Opérations : calcul position pixel

Pour chacun des pixels, la dépendance architecturale est égale à 15 instructions / pixel, pour un total de 163 instructions / segment de 10 pixels. Avec 4 ***PE***s, la dépendance est de 40.75 instructions / segments.

3.2.2.4. Performance de l'application

La performance actuelle de l'algorithme de Bresenham sur **V1** est de 163 instructions / segments de 10 pixels. Avec 4 ***PE***, la dépendance est de 40.75 instructions / segments.

L'accélération perd presque tout son intérêt à cause de l'utilisation d'un test. Chaque instruction de test est suivie de deux cycles vides.

Dans ces cas, **PULSE** devrait pouvoir éliminer ces deux cycles en commençant l'opération sur tous les ***PE*** avec les instructions suivant le test. Dans le cas d'une réponse négative, vider le pipeline et atteindre le prochain test. Les conditions suivantes devraient tout de même dans certains cas contenir un ou deux cycles vides pour ne pas manipuler des données provenant de la boucle précédente.

PULSE V2

Dans le cas de **PULSE V2**, une multiplication par quatre du nombre de segments durant le même nombre de cycles est impossible par l'augmentation du nombre de cycles nécessaire pour faire voyager les données sur les canaux Nord et Sud. Neuf cycles supplémentaires sont requis pour s'acquitter de cette tâche. Sur **V2** la performance serait de (15 + 9) * 10 = 240 cycles / 10 pixels. Avec l'emploi de 16 *PE*s l'accélération serait de 2.72 par rapport à **V1** avec 4 *PE*s.

3.2.3. Algorithme RSA

3.2.3.1. Description de l'algorithme

RSA est un algorithme de cryptage à deux clés, une privée et une publique, créée par un groupe de trois professeurs du MIT en 1977. Un système de cryptage est un algorithme capable de convertir des données lisibles en un texte indéchiffrable, le chemin inverse est aussi possible. Le flot d'activités se fait selon la figure 3.8, un test clair et une clé de cryptage sont entrés dans la portion cryptage de l'algorithme. Pour décrypter, le texte crypté et une clé associée passent à travers la portion décryptage de l'algorithme. Les deux clés, publique et privée, sont obtenues à partir de nombres premiers et de manipulations mathématiques. Pour le traitement avec **PULSE**, il est assumé que les clés sont disponibles.

L'équation utilisée pour crypter chaque caractère (C) ASCII est $C = (C * P) \% N$. Chaque caractère est traité de façon récursive un nombre de fois déterminé par la clé. P et N sont des constantes et sont premiers entre eux.

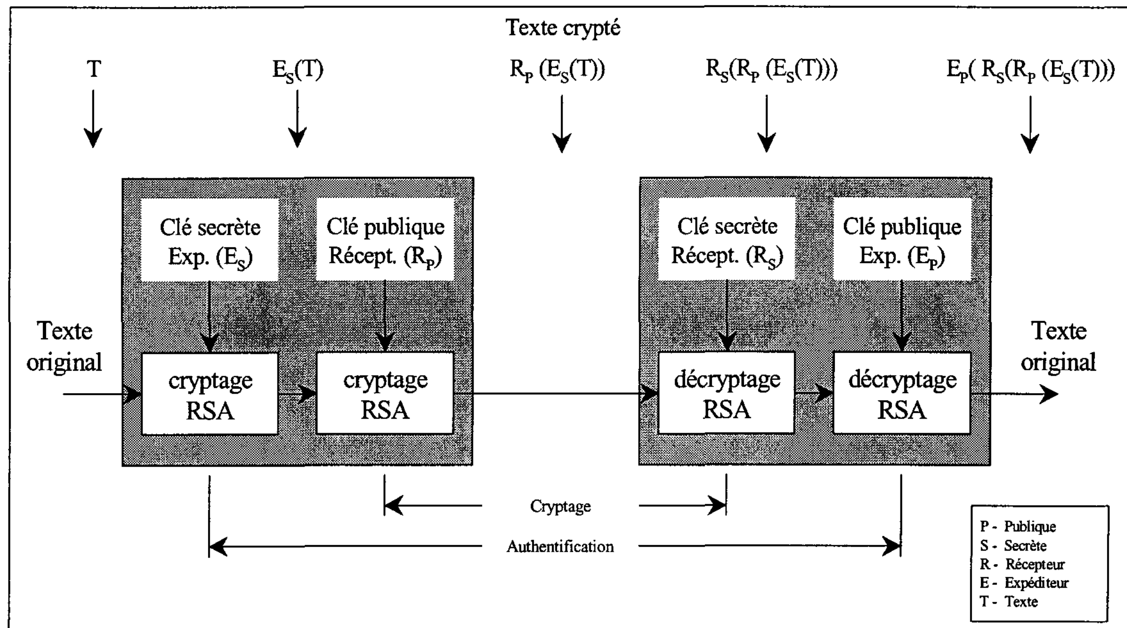


Figure 3.8 : RSA - Authentification et Cryptage de message

3.2.3.2. Répartition sur PULSE pour le cryptage

1. **PULSE** possède en mémoire la clé secrète de l'expéditeur et la clé publique du destinataire.
2. Le texte originel est entré par le canal Nord, le texte crypté trouve son chemin sur le canal Sud.
3. Chaque ***PE*** traite un caractère à la fois.
4. Chaque caractère est modifié *e* fois par la formule C = (P * C) % N

3.2.3.3. Limite de vitesse basée sur l'architecture

Une fois que les clés emmagasinées dans la mémoire de **PULSE**, la difficulté de cet algorithme repose sur l'opération modulo.

Une première version utilisant une instruction conditionnelle et une soustraction récursive (sur C de N) jusqu'à l'obtention du nombre cible a tout d'abord été testée. Le temps de traitement dépendant de la valeur du résultat de la multiplication et était nettement plus élevé que si une instruction modulo avait été disponible.

Deux algorithmes mathématiques ont été utilisés pour améliorer la performance.

- Dans le cas de l'opération récursive de multiplication l'algorithme est appelé *mise au carré et multiplication* (*square and multiply*) [KAYA94]. Il s'agit de minimiser le nombre de multiplications de C à partir de la décomposition binaire de son exposant originel. Cette méthode permet de réduire le temps de traitement de $O(n)$ à $O(\log n)$.
- Dans le cas de la recherche du modulo de (P * C), un algorithme soumis par Normand Bélanger [BELA97] du Groupe **PULSE** a été utilisé. Une copie détaillée de l'algorithme se trouve en Annexe (Annexe C). En résumé, il s'agit de trouver le premier bit de poids fort de $C * P$ qui a la valeur 1 et de multiplier N par la puissance de 2 la plus grande possible qui donne un nombre dont le premier bit de poids fort de valeur 1 soit situé au bit qui est à la position immédiatement à la droite de celui de $C * P$. Cette méthode a aussi permis de réduire le temps de traitement de $O(n)$ à $O(\log n)$.

Le problème est toujours l'utilisation d'instructions conditionnelles et d'instructions de branchement, afin de déterminer si l'opération modulo est bel et bien terminée.

La limite indépendante de l'architecture est de 2 instructions / caractère ; un OU exclusif et une division pour le modulo. Pour un circuit intégré à 4 ***PE*s**, la performance devrait être de 0.5 instructions / caractère / boucle. Pour pouvoir compter sur une

instruction modulo avec PULSE, il faudrait que le circuit intégré permette d'effectuer une opération de division. Cela n'est pas possible et n'entre pas dans la philosophie d'utilisation avec laquelle la puce a été développée.

En tenant compte de la dépendance de données, un caractère devrait prendre huit (8) instructions par boucle. Le nombre de boucles dépend de la valeur de E.

3.2.3.4. Performance de l'application

La version finale du programme se trouve en Annexe. En tenant compte de la dépendance de données, un caractère nécessite en moyenne 3002 cycles / caractères par ***PE***, soit une moyenne de 750.5 cycles / caractères pour **PULSE V1**. Ces nombres sont extraits d'un test effectué sur 10 millions de caractères avec un $E = 23$. En divisant la moyenne de cycles / caractères / ***PE*** (750.5) par le nombre de boucle (23) ; nous obtenons une moyenne de 32.6 cycles par cycles / caractères / ***PE*** / boucle.

Ce résultat est très loin de la limite calculée précédemment, le goulot d'étranglement pour cette application est définitivement l'absence de diviseur matériel avec **PULSE**.

3.2.4. Équations aux différences finies (Poisson)

3.2.4.1. Description de l'algorithme

Cet algorithme sert à modéliser un système composé de particules discrètes, où chaque particule est affectée par les autres particules. Chaque particule possède une dépendance directe avec les autres particules à l'intérieur du modèle.

Cette application a comme objectif de résoudre l'équation de Poisson pour un système à deux dimensions :

$$-\frac{\partial^2 U}{\partial x^2} - \frac{\partial^2 U}{\partial y^2} = f(x, y)$$

En utilisant l'itération de Jacobi ; c'est-à-dire en discriminant le domaine du problème et en appliquant l'opération suivante à tous les points intérieurs, jusqu'à convergence :

$$u_{(i,j)}^{(k+1)} = \frac{1}{4}(u_{(i-1,j)}^{k} + u_{(i+1,j)}^{k} + u_{(i,j-1)}^{k} + u_{(i,j+1)}^{k} + 4q_{(i,j)})$$

Où $q_{(i,j)}$ est la charge particulière à la coordonnée (i,j). Dans le cas présent, la charge et les conditions frontières ; demeurent des valeurs fixes durant le traitement. La nouvelle valeur associée à un point est égale à la moyenne de la valeur de ses quatre voisins plus un terme qui reflète, dans le cas d'un système électromagnétique, la charge locale à ce point.

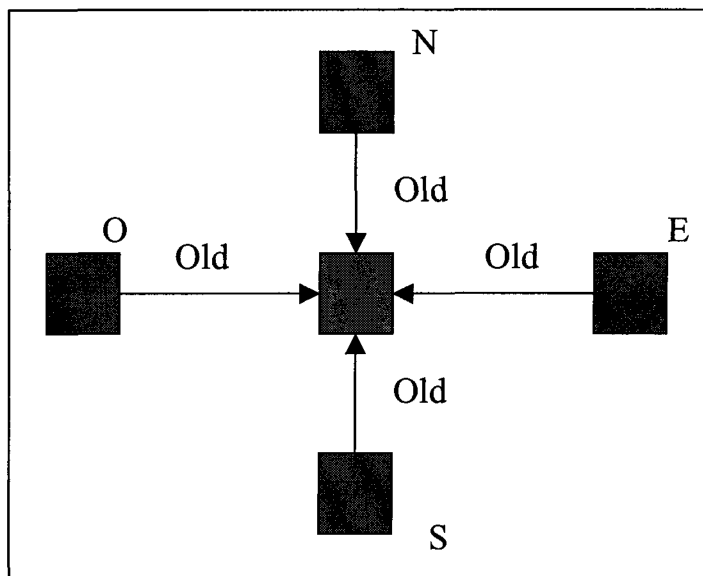


Figure 3.9 : Poisson - Voisinage nodal

3.2.4.2. Répartition sur PULSE

Le traitement parallèle de cet algorithme est normalement implanté sur un système multiprocesseur à architecture maillé. Le nombre de nœuds est égal aux nombres de processeurs disponibles sur le réseau. Dans le cas de **PULSE V1**, la puce ne comporte que 4 ***PE*s**. La grille a été modelée en attribuant à chacun des espaces mémoires A de chaque processeur l'identité d'un nœud. On se retrouve avec 1024 processeurs (256 x 4), l'équivalent d'une maille de dimension 32 X 32. La mémoire B est utilisée pour conserver la charge équivalente à l'adresse correspondante dans la mémoire A. Comme mentionné précédemment, des conditions frontières fixes sont utilisées pour le calcul.

3.2.4.3. Limite de vitesse basée sur architecture

La limite indépendante de l'architecture est de 9 instructions / par maille pour chaque tour de boucle ; 5 additions, 1 division, 1 soustraction, 1 valeur absolue et 1 instruction conditionnelle. Pour un circuit intégré à 4 ***PE***, la performance devrait être de 2.25 cycles / maille * 1024 mailles = 2304 cycles / boucle.

En tenant compte de la dépendance de données et de l'architecture de **PULSE,** une maille se traite de la façon suivante :

$U_{(i-1,j)} + U_{(i+1,j)}$ → Accumulateur
$U^k_{(i+1,j)}$ * (-4) + Accumulateur→ Accumulateur (soustraction valeur précédente)
$U_{(i,j-1)} + U_{(i,j+1)}$ + Accumulateur→ Accumulateur
$q_{(i,j)}$ * 4 + Accumulateur→ Accumulateur
3 NOPs; 3 cycles, dépendance de données avec accumulateur
Accumulateur / 4 → ($U^{(k+1)}_{(i+1,j)}$ - $U^k_{(i+1,j)}$) Registre 1
3 NOPs ; 3 cycles, dépendance de données avec nouvelle valeur nœud
Abs(Registre 1) → Registre 3
$U^{(k-1)}_{(i+1,j)}$ + Registre 1 → $U^k_{(i+1,j)}$
2 NOPs ; 2 cycles, dépendance de données Registre 3
Comparaison Registre et Valeur de Convergence
2 NOPs

Encadré 3.10 : Poisson - Opérations : équation aux différences finies

Pour chaque nœud, la dépendance architecturale est égale à 18 instructions / maille / boucle. Pour un circuit intégré à 4 ***PE***, la performance devrait être de 4.50 cycles / maille * 1024 mailles = 4608 cycles / boucle.

3.2.4.4. Performance de l'application et goulot d'étranglement

La division de la maille se fait selon la figure 3.10.

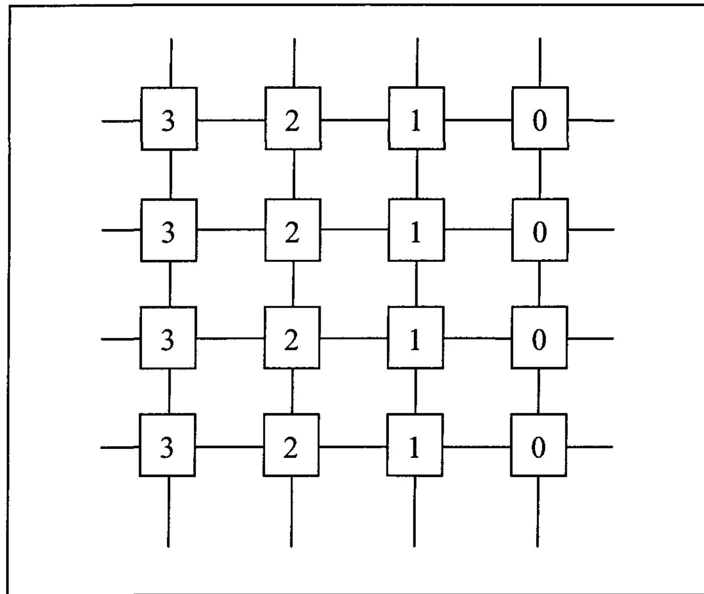


Figure 3.10 : Poisson - Correspondance Maille – PE

Les canaux Nord et Sud sont utilisés afin de rendre accessibles les données provenant des nœuds Est et Ouest. Afin de réduire le nombre de ***NOP*** dans la dernière partie de l'algorithme, la validation de la convergence n'est effectuée qu'une seule fois par boucle sur une valeur. Cette valeur est obtenue en faisant un *Cas* (Compare And Swap) des valeurs absolues de chaque étape de convergence pour chaque maille. Cette valeur sera égale au delta maximum durant une boucle.

Comme condition de départ, les nœuds possèdent leurs valeurs initiales en mémoire. La performance actuelle de l'algorithme sur **V1** est de 2574 / instructions / 1024 nœuds. Le gain de performance est attribuable à la façon dont est traitée l'étape de convergence.

PULSE V2

L'ajout de 12 *PE*s supplémentaires nous permet de diminuer le temps de convergence de l'architecture maillé utilisée avec **PULSE V1**. La communication doit se faire avec les voisins immédiats uniquement, il n'y a donc pas de temps de latence ajouté causé par le nombre de plus élevé *PE*s. Le gain de performance devrait être égal au facteur d'augmentation du nombre de *PE* : 4.

Une seconde approche est de garder le même temps de convergence mais d'augmenter la dimension de la structure maillée à 16 x 256 = 4096 nœuds, soit une maille de 64 x 64. En conclusion, cet algorithme devrait pleinement bénéficier d'une augmentation du nombre de *PE*.

3.2.5. Morphologie Binaire : Érosion et Dilatation

Morphologie veut dire "Étude de la forme et de la structure d'un objet" [PARK97]. La morphologie est en relation avec la silhouette, et la morphologie digitale est une façon de décrire et d'analyser la silhouette d'un objet digital.

L'idée derrière la morphologie digitale est que les images consistent en un ensemble de pixels rassemblées en différentes silhouettes à deux dimensions. Certaines opérations mathématiques peuvent être utilisées sur cet ensemble de pixels dans le but d'améliorer des aspects spécifiques de la silhouette, par exemple.

La morphologie binaire est définie pour les images à deux niveaux de couleurs ; pixels blanc ou noir seulement. Deux opérations ont été mises en application, l'érosion et la dilatation binaire. L'opération mathématique est effectuée sur tous les voisins immédiats du pixel cible, $p_{1,1}$, le traitement se fait sur une fenêtre de 3x3.

$p_{0,2}$	$p_{0,1}$	$p_{0,0}$
$p_{1,2}$	$p_{1,1}$	$p_{1,0}$
$p_{2,2}$	$p_{2,1}$	$p_{2,0}$

- Dilatation : Si le pixel central est égal à 1 ($p_{1,1}$), tous les voisins immédiats doivent être mis à 1.
- Érosion : Si le pixel central est égal à 0 ($p_{1,1}$), tous les voisins immédiats doivent être mis à 0.

3.2.5.1. Mouvement de données

Les pixels de l'image source sont entrés via le canal Nord et traités un à la suite de l'autre. **PULSE V1** traite simultanément quatre pixels successifs (*PE*0-3).

3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0
3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0
3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0

Figure 3.11 : Morphologie - Correspondance Pixel – PE

Les résultats sont mis sur le canal Sud avant d'être décalés dans la mémoire externe par le port #4. Le mouvement se fait aussi pixel par pixel.

3.2.5.2. Conditions

Pour chacune de ces deux applications, une colonne de zéro devra suivre la colonne de pixels à l'extrémité droite de l'image.

																0
																0
																0

Figure 3.12 : Morphologie - Colonne supplémentaire de 0 (zéro)

3.2.5.3. Dimensions

Avec la version 1 de **PULSE**, la dimension maximale de l'image source est de 512 x 512 pixels. Chaque mémoire, A et B, contient le résultat du traitement de deux lignes pour un pixel sur quatre.

$Image_{max} = (Espace\ mémoire \times Nombre\ de\ PEs) / 2$

3.2.5.4. Description de l'Algorithme

Dans le cas de la dilatation, la valeur finale du pixel cible correspond au OU logique des neuf pixels occupant la fenêtre courante (3x3). Si un seul de ces pixels est égal à 1, le résultat est égal à 1.

Pour l'opération d'érosion, on réalise le ET logique des neuf pixels compris dans la fenêtre d'opération. Si un seul des pixels est égal à 0, la valeur finale correspondant au pixel central est égale à 0.

3.2.5.5. Répartition sur PULSE

Le traitement s'effectue sur trois lignes subséquentes. Le résultat du OU logique de la ligne courante se retrouve dans une des deux mémoires internes (a ou b), de la position 0 à *NbPixelsLigne/NbPE-1*. La seconde partie de la mémoire est utilisée pour emmagasiner le résultat du OU logique des six premiers pixels de la fenêtre.

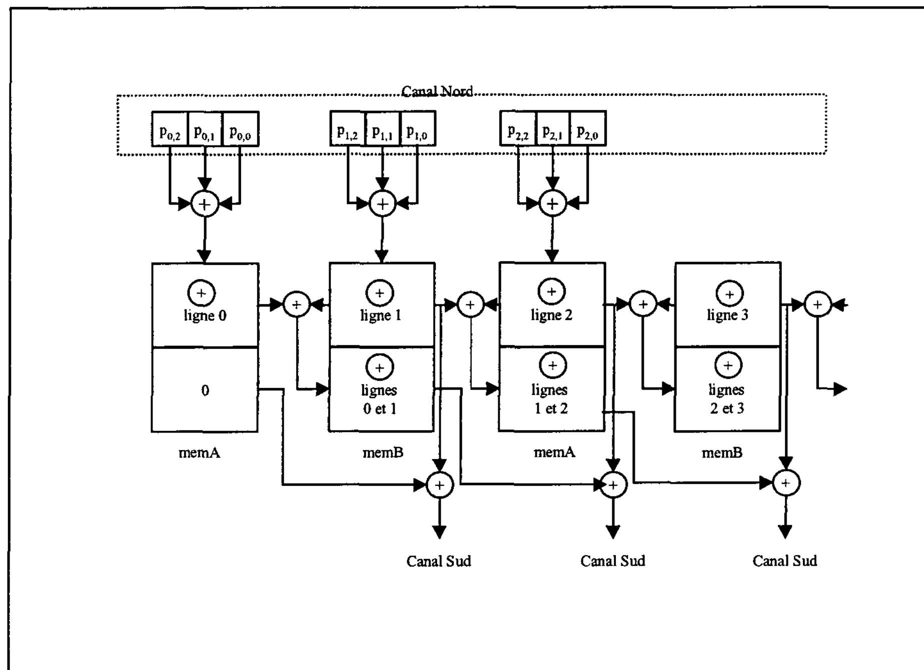


Figure 3.13 : Morphologie - Mouvement de données et calcul

3.2.5.6. Limite de vitesse basée sur l'architecture

La limite indépendante de l'architecture est de 8 instructions / pixel ; 8 ET logique dans le cas de l'érosion et 8 OU logique dans le cas de la dilatation. Pour un circuit intégré à 4 ***PE***, la performance devrait être de 2 cycles / pixel.

En tenant compte de la dépendance de données, un pixel se traite de la façon suivante (érosion dans ce cas-ci) :

And($p_{0,0}$, $p_{0,1}$) → Mémoire A (résultat 1), adressage indirect
And($p_{1,0}$, $p_{1,1}$) → Mémoire A (résultat 2), adressage indirect
And($p_{2,0}$, $p_{2,1}$) → Mémoire A (résultat 3), adressage indirect
And($p_{0,2}$, $p_{1,2}$) → Mémoire A (résultat 4), adressage indirect
And($p_{2,2}$, résultat 1) → Mémoire A (résultat 5), adressage indirect
NOP; 1 cycle dépendance sur résultat 3
And(résultat 2, résultat 3) → Mémoire A (résultat 6), adressage indirect
NOP; 1 cycle dépendance sur résultat 5
And(résultat 4, résultat 5) → Mémoire A (résultat 7), adressage indirect
3 NOPs; 3 cycles dépendance sur résultat 7
And(résultat 6, résultat 7) → Mémoire A (résultat 8), adressage indirect

Encadré 3.11 : Morphologie - Opérations - morphologie binaire

Pour chacun des pixels, la dépendance architecturale est égale à 13 instructions / pixel. Avec 4 ***PE***s, la dépendance est de 3.25 instructions / pixel.

3.2.5.7. Performance de l'application et goulot d'étranglement

Le traitement se fait de manière légèrement différente, le ET ou le OU de chaque groupe de trois pixels consécutifs sur une même ligne est emmagasiné dans la mémoire locale de **PULSE** (figure 3.13). Ce résultat subit ensuite une opération logique avec les trois pixels de même colonne de la ligne précédente associés aux même colonnes pour ensuite être réutilisé lors d'opérations sur la ligne suivante. Au total, le résultat emmagasiné du ET ou du OU est utilisé pour 6 pixels adjacents au lieu d'être recalculé.

La performance actuelle des algorithmes de morphologie binaire sur **V1** est de 10.00 instructions / pixel. Avec 4 ***PE*s**, la performance est de 2.25 instructions / pixel pour une image de 128 x 128 pixels, soit une performance supérieure à la limite de l'architecture. La réutilisation de résultats d'opérations précédents explique pourquoi la performance actuelle est supérieure à limite de l'architecture. Cette idée n'a pas été prise en considération lorsque le calcul de la limite a été effectué. La performance se dégrade bien sûr pour les images de très petites dimensions.

3.2.5.8. PULSE V2

Le traitement de l'image se fait pixel par pixel. L'augmentation du nombre de ***PE*s** est inversement proportionnelle au temps d'exécution pour traiter une image. On peut donc estimer que le temps de traitement avec 16 ***PE*s** sera quatre fois plus petit.

La dimension maximale de l'image source dépend du nombre de ***PE*s**, avec 16 ***PE*s** elle passe à 2048 x 2048 pixels.

3.2.6. IDEA

IDEA est un algorithme de cryptage classique. Il est utilisé dans PGP[18] pour le cryptage des données. Il agit sur des blocs de texte en clair de 64 bits à l'aide d'une clé de 128 bits. Il repose uniquement sur trois opérations algébriques : un XOR (Ou exclusif), une addition modulo 2^{16}, et une multiplication modulo $2^{16} + 1$. Le procédé de cryptage nécessite un groupe d'opérations identiques effectuées huit fois consécutives en plus d'une transformation finale.

Le bloc de texte clair de 64 bits est divisé en 4 blocs de 16 bits : *X1*, *X2*, *X3* et *X4*.

3.2.6.1. Description de l'algorithme

Création des clés dérivées

L'algorithme nécessite 52 clés dérivées (6 pour chacune des 8 rondes + 4 pour la transformation finale). La clé de 128 bits est d'abord divisée en 8 blocs de 16 bits qui forment les 8 premières clés : *Z1*, *Z2*, *Z3*, *Z4*, *Z5*, *Z6*, *Z7* et *Z8* parmi les 52. La clé de 128 bits est ensuite décalée circulairement vers la gauche de 25 bits (figure 3.14) et donnera 8 nouveaux blocs de 16 bits. Ainsi de suite jusqu'à avoir les 44 clés supplémentaires.

---

[18] Pretty Good Privacy

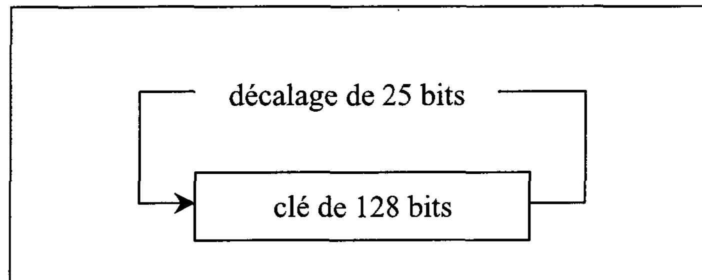


Figure 3.14 : IDEA - Création des clés dérivées

Opérations mathématiques

- Les 14 opérations mathématiques effectuées huit fois consécutives.

1. $X1 * Z1 \% (2^{16} + 1)$
2. $X2 + Z2 \% 2^{16}$
3. $X3 + Z3 \% 2^{16}$
4. $X4 * Z4 \% (2^{16} + 1)$
5. (Résultat Étape 1) XOR (Résultat Étape 3)
6. (Résultat Étape 2) XOR (Résultat Étape 4)
7. (Résultat Étape 2) * $Z5 \% (2^{16} + 1)$
8. (Résultat Étape 6) + (Résultat Étape 7) $\% 2^{16}$
9. (Résultat Étape 8) * $Z6 \% (2^{16} + 1)$
10. (Résultat Étape 6) + (Résultat Étape 7) $\% 2^{16}$
11. (Résultat Étape 1) XOR (Résultat Étape 9) $\Rightarrow$ $X1$ de la ronde suivante
12. (Résultat Étape 3) XOR (Résultat Étape 9) $\Rightarrow$ $X2$ de la ronde suivante
13. (Résultat Étape 2) XOR (Résultat Étape 10) $\Rightarrow$ $X3$ de la ronde suivante
14. (Résultat Étape 4) XOR (Résultat Étape 10) $\Rightarrow$ $X4$ de la ronde suivante

Encadré 3.12 : Opérations de cryptage avec IDEA

Transformation finale pour avoir le texte crypté :

- Quatre opérations mathématiques supplémentaires après les opérations dans l'encadré 3.12 sont requises pour avoir le texte crypté final.

1. $X\mathit{1} * Z\mathit{1} \% (2^{16} + 1)$
2. $X\mathit{2} + Z\mathit{2} \% 2^{16}$
3. $X\mathit{3} + Z\mathit{3} \% 2^{16}$
4. $X\mathit{4} * Z\mathit{4} \% (2^{16} + 1)$

3.2.6.2. Répartition sur PULSE

1. La clé de 128 bits est entrée par les canaux Sud et Nord et emmagasinée en mémoire (a et b).
2. Les 52 clés secondaires de 16 bits sont créées et emmagasinées en mémoire.
3. Entrée des premiers caractères originels.
4. Chaque caractère est modifié en suivant les opérations mathématiques vues dans la section précédente.
5. Sortie des caractères cryptées et entrée des caractères subséquents.
6. Chaque ***PE*** s'occupe de traiter un train de 64 bits à la fois.
7. Chaque ***PE*** possède une copie locale des 52 clés secondaires ce qui lui permet de travailler indépendamment des autres.

3.2.6.3. Limite de vitesse basée sur l'architecture

La limite indépendante de l'architecture est la suivante :

Étape 1 - Les opérations suivantes doivent être effectuées 8 fois :

8 modulo (divisions + OU exclusif) + 4 additions + 4 multiplications + 6 OU exclusif pour un total partiel de 30 instructions * 8 tours de boucle = 240 instructions.

Étape 2 - Les opérations suivantes sont effectuées sur le résultat de l'étape 1:

4 modulo (divisions + OU exclusif) + 2 multiplications + 2 additions = 12 instructions.

Le nombre d'instructions total est de 252 instructions.

Pour chaque train de 64 bits cryptés il faut effectuer ; 48 XOR, 36 additions modulo 2^{16} et 36 multiplications modulo $2^{16} + 1$. Pour un circuit intégré à 4 ***PE***s, la performance devrait être de 63 cycles / 64 bits / ***PE***.

En tenant compte de la dépendance de données, les 64 bits de caractères se traitent de la façon décrite à l'intérieur du tableau 3.6.

Étapes de chacune des rondes (faire 8 fois)
Étape 1
X*1* * Z*1* → Registre 1 NOP ; 3 cycles dépendance sur résultat dans registre 1 Modulo 2^{16} + 1 sur Registre 1→ Registre b1
Étape 2 et 3
X*2* + Z*2* → Registre b2 X*3* + Z*3* → Registre b3
Étape 4
X*4* * Z*4* → Registre 4 NOP ; 3 cycles dépendance sur résultat dans Registre 4 Modulo 2^{16} + 1 sur Registre 4→ Registre a4
Étape 4
(Registre b1) XOR (Registre b3) → Registre a5
Étape 6
NOP ; 2 cycles dépendance sur résultat dans Registre 4 (Registre b2) XOR (Registre a4) → Registre b6
Étape 7
(Résultat Étape b2) * Z*5*→ Registre 7 NOP ; 3 cycles dépendance sur résultat dans Registre 7 Modulo 2^{16} + 1 sur Registre 7→ Registre a7

Étape 8
NOP ; 3 cycles dépendance sur résultat dans Registre b7 (Registre b6) + (Registre a7) →Registre a8
Étape 9
NOP ; 3 cycles dépendance sur résultat dans Registre a8 (Registre a8) * Z*6* →Mémoire 60 NOP ; 3 cycles dépendance sur résultat dans Mémoire 60 Modulo $2^{16} + 1$ sur Mémoire 60→ Mémoire a60
Étape 10
NOP ; 3 cycles dépendance sur résultat dans Mémoire a60 (Registre a7) + (Mémoire a60) →Mémoire a61
Étape 11, 12, 13 et 14
(Résultat Étape 1) XOR (Résultat Étape 9) ⇒ X*1* de la ronde suivante (Résultat Étape 3) XOR (Résultat Étape 9) ⇒ X*2* de la ronde suivante NOP ; 1 cycle dépendance sur résultat dans Mémoire a61 – Étape 10 (Résultat Étape 2) XOR (Résultat Étape 10) ⇒ X*3* de la ronde suivante (Résultat Étape 4) XOR (Résultat Étape 10) ⇒ X*4* de la ronde suivant

Tableau 3.6 : Cryptage avec IDEA – Opérations récursives[19]

---

[19] Noter que les additions modulo 2^{16} résulte en une addition simple en utilisant les 16 bits de poids faible du résultat pour l'opération à suivre.

Étapes supplémentaires après la huitième ronde
Étape 1
X*1* * Z*1* → Résultat 1 NOP ; 3 cycles dépendance sur résultat 1 Modulo 2^{16} + 1 sur Résultat 1
Étape 2 et 3
X*2* + Z*2* → Résultat 2 X*3* + Z*3* → Résultat 3
Étape 4
X*4* * Z*4* → Résultat 4 NOP ; 3 cycles dépendance sur résultat dans Résultat 4 Modulo 2^{16} + 1 sur Résultat 4

Tableau 3.7 : Cryptage avec IDEA – Opérations finales

Pour chaque groupe de 64 bits, la dépendance architecturale est au minimum égale à 348 cycles / 64 bits / ***PE***.

3.2.6.4. Performance de l'application

Le problème avec cette application est la multiplication modulo $2^{16} + 1$, qui au terme de chaque multiplication requière une opération de soustraction récursive. L'utilisation d'un modulo, comme vu précédemment avec RSA, affecte les performances. Une solution plus rapide à cette opération serait de pouvoir compter sur une instruction conditionnelle de test (*if*) valide sur 32 bits. Cette instruction n'est pas disponible avec **PULSE V1**, mais fait

partie des spécifications techniques pour **V2**. La copie du programme en annexe utilise une multiplication modulo 2^{16} afin de pouvoir utiliser un *if* sur 16 bits.

La performance de cette application est de 471 cycles / 64 bits / PE. Cette application a été conservée pour montrer le travail ayant été effectué afin de transformer la clé de 128 bits en 56 clés de 16 bits.

3.2.7. Constat

Le but de ce chapitre était de présenter les différentes applications étudiées et développées. Les applications touchent plusieurs champs, tel que le traitement d'images ou le cryptage de données. Les premiers choix d'applications ont étés fait sans prendre en considérations les restrictions de l'architecture et du jeu d'instruction de **PULSE**. Ces tentatives de décomposition m'ont amené à rétrécir le champ des applications intéressantes pour cette machine.

En rétrospective, il est clair que les applications utilisant divisions et instructions conditionnelles doivent être évitées. Un circuit de type SIMD comme **PULSE** est mieux adapté à certains types d'applications où l'on peut tirer avantage du pouvoir de calcul des *PE*s à chaque cycle d'horloge. Ce n'est pas un hasard si **PULSE** appartient à la catégorie des circuits spécialisés.

Les résultats et recommandations sont présentés dans les deux prochains chapitres.

CHAPITRE 4

ÉVALUATION DE LA PERFORMANCE

4.1. PRÉAMBULE

Après avoir effectué de multiples itérations sur une application pour pouvoir maximiser l'utilisation de la bande passante des différents processeurs, il faut maintenant être capable d'en évaluer la performance.

Deux sources de données sont disponibles à partir des informations présentées au chapitre 3; les informations déduites et calculées après l'étude de chacun des programmes développés, et la compilation des résultats générés par le simulateur.

4.2. PRÉSENTATION DES RÉSULTATS

4.2.1. Paramètres et mesures de développement

Pour la majorité des applications vues dans le chapitre précédent, deux limites ont été évaluées ; la **limite indépendante de l'architecture** et la **limite dépendante de l'architecture**.

La version finale de chaque algorithme développé était comparée avec cette seconde limite et a permis d'évaluer l'écart (en cycles) entre la mise en œuvre et cette ultime limite. Cette limite peut être mise en relation directe avec un des objectifs de développement expliqués au début du chapitre 3 ; **l'efficacité**. L'efficacité, une mesure subjective, a été définie à la section 3.1.1.4 comme le résultat de la rapidité de traitement de **PULSE** associée au degré d'optimisation des programmes réalisés. Le parallèle entre la première partie de cette définition et la limite dépendante de l'architecture est direct.

4.2.2. Banc d'essai

Au début du chapitre précédent, deux méthodes permettant d'améliorer la performance d'une application sur **PULSE** ont été expliquées : réduction du nombre de *NOP* et mise en parallèle des opérations sur une même ligne d'instructions. Pour **PULSE**, la mise en parallèle des opérations est l'accélération par rapport à une même application traitée de façon séquentielle.

Ces informations sont disponibles à l'aide des résultats numériques obtenus grâce à l'utilisation d'un modèle VHDL de **PULSE V1**. Ce modèle avait, entre autres

caractéristiques, l'avantage de produire des fichiers comprenant différents résultats numériques :

- Facteur d'accélération **Sp** (speedup factor) défini par : Sp = Ts / Tp, où :
  - **Ts** : Nombre d'instructions nécessaires pour l'exécution de l'algorithme sur PULSE, sans mise en parallèle des opérations,
  - **Tp** : Nombre d'instructions nécessaires pour l'exécution du même algorithme avec utilisation du pipeline d'opérations en parallèle:

Le facteur d'accélération est l'amélioration relative des résultats numériques de performance de chacune des applications basées sur l'utilisation des instructions parallèles, sans tenir compte des instructions d'attente (NOP). Il s'agit du rapport entre l'utilisation d'instructions simples et parallèles. Sa valeur se situe entre 1 et 3, 3 étant le maximum possible. Les instructions parallèles permettent d'utiliser le pipeline de **PULSE** en utilisant les différentes couches de l'architecture durant le même cycle, comme vu à la section 1.4.3.2 « Caractéristique du jeu d'instructions »:

traitement numérique||mouvement de données||communication||E/S contrôle

La valeur maximale possible est 3, car la dernière partie de la ligne d'instruction, l'opération d'entrée et de sortie, ne fait pas partie du calcul du facteur d'accélération.

- Efficacité **Ep** (System efficiency factor) définie par : Ep = Sp / p est aussi disponible pour chaques applications évaluées. Sa valeur se situe entre 33 % et 100%, où 100% est le maximum possible.
  - **p** : nombre d'opérations pouvant être effectuées de façon concurrentielle, 3 dans notre cas; traitement numérique, communication et contrôle comme vu précédemment.
- Le Pourcentage de *NOP* et d'instructions dans chacun des trois groupes d'opérations suivants :
  1. Calcul,
  2. Communication,
  3. Contrôle.

  A partir de ces valeurs, l'**Utilisation** est calculée : il s'agit du travail total (%) duquel est soustrait le pourcentage de *NOP*.
- **Performance** : le nombre de cycles nécessaires pour traiter une donnée, un pixel ou un groupe de données. La performance est calculée en divisant le nombre total de cycles par le nombre de données traitées.
- **Comparable** : comparaison, dans certain cas, de résultats d'applications similaires traitées sur différents processeurs disponibles sur le marché, avec les résultats obtenus avec le simulateur.

Description des colonnes du tableau 4.1 :
LDC : **L**ignes **D**e **C**ode
LIA : **L**imite **I**ndépendante de l'**A**rchitecture
LDA : **L**imite **D**épendante de l'**A**rchitecture – résultat obtenu
Efficacité (%) : Limite dépendante architecture / résultat obtenu

Le tableau 4.2 est un résumé des goulots d'étranglements rencontrés durant l'optimisation et le développement de chacune des applications.

Application	**Goulot D'étranglement**
Matrice, décomposition sur les données	• Dimension de la mémoire interne, • Non-disponibilité d'une combinaison spécifique d'instructions parallèles.
Matrice, décomposition sur l'architecture	• Non-disponibilité d'une combinaison spécifique d'instructions parallèles.
Bresenham	• Utilisation d'instructions conditionnelles
RSA	• Absence de diviseur matériel, • Utilisation d'instructions conditionnelles
Poisson	• Dimension de la mémoire interne
Érosion / Dilatation	• Latence
IDEA	• Absence de diviseur matériel, • Utilisation d'instructions conditionnelles, • Opérations conditionnelles possibles sur 16 bits seulement.

Tableau 4.2 : Goulots d'étranglements

Pour chacune des applications étudiées, la **latence** est un goulot d'étranglement. La latence se réfère à l'incapacité pour PULSE de produire un nombre plus grand d'instructions ayant une latence de 1 cycle. Ce problème est encore plus évident dans le cas

de l'instruction *load* et dans certaines applications, comme Bresenham et Poisson, où les cycles vides ne peuvent être remplis avec d'autres instructions.

La colonne comportant les résultats d'efficacité du tableau 4.1 s'associe facilement avec un des goulots d'étranglement du tableau 4.2 : Utilisation d'instructions conditionnelles. Toutes les applications décomposées sur les données utilisant des instructions conditionnelles possèdent une efficacité égale ou inférieure à 100%. En plus de ne pas utiliser d'instructions conditionnelles, une caractéristique qui contribue à augmenter l'efficacité, est l'utilisation d'instructions spéciales de **PULSE**. Ces instructions ont été conçues pour permettre d'accélérer le traitement des applications ciblées par le groupe **PULSE**, par exemple les traitements d'images (érosion/dilatation).

Le tableau 4.1 montre aussi qu'une efficacité élevée est souvent, mais pas toujours, le résultat d'un pourcentage relativement faible d'instructions vides. Cette affirmation est particulièrement vraie dans le cas d'emploi d'instructions conditionnelles. Comme nous l'avons décrit précédemment dans le chapitre 1, chaque instruction conditionnelle force l'insertion d'un minimum de 2 instructions vides (4, si on utilise la préemption).

Il faut noter que le tableau 4.1, contrairement aux tableaux des pages suivantes, donne le pourcentage d'instructions vides par rapport au nombre total de lignes de code de l'application. Ce chiffre peut être trompeur si par exemple l'initialisation et la fin du programme contiennent un pourcentage élevé d'instructions vide tandis que le corps de l'application en comporte un pourcentage faible. Le pourcentage d'instructions vides résultant d'une simulation traitant un nombre élevé de données sera plus faible dans ce cas. Le contraire est aussi vrai; dans le cas d'un corps de programme avec un haut pourcentage

d'instructions vides, le pourcentage résultant d'une simulation sera plus élevé que celui du tableau 4.1.

Deux des goulots d'étranglements évoqués dans le tableau 4.2, non-disponibilité d'une combinaison spécifique d'instructions parallèles et opérations conditionnelles possibles sur 16 bits seulement, ont été réglés dans la seconde version de **PULSE**.

4.2.3.2. Résultats de simulations

Le tableau 4.3 présente les résultats obtenus par les trois différentes versions de la première application développée, multiplication vecteurs – matrice. Ces résultats sont extraits de simulations traitant 200 vecteurs. Dans les tableaux suivant, chaque colonne représente le pourcentage occupé par chaque groupe d'opérations. La dernière colonne égale le Total moins le pourcentage de cycle vide (NOP).

Application	NOP %	Calcul %	Comm. %	Contrôle %	Total %	Utilisation %
Matrice, décomposition sur les données	16.61	33.00	74.42	17.05	141.10	124.49
Vecteurs dans mem. Interne	0.44	54.95	65.93	11.65	133.00	132.56
Matrice, décomposition sur l'architecture	0.58	23.24	92.94	6.36	123.20	122.62

Tableau 4.3 : Multiplication vecteurs – matrice – traitement de 200 vecteurs

Le tableau 4.4 présente des résultats d'une simulation dans le cas de l'algorithme de Bresenham, celui-ci traitant 50 vecteurs de 10 pixels chacun.

Application	Temps (µs)	Perf. cyc/pix	NOP %	Calcul %	Comm. %	Contrôle %	Total %	Utilisation %
Bresenham	43.27	4.67	23.82	25.44	50.55	22.02	121.80	97.98

Tableau 4.4 : Bresenham – traitement de 50 vecteurs de 10 pixels

Le tableau 4.5 présente les résultats de cryptage sur 10^6 caractères de 8 bits avec les algorithmes RSA et IDEA.

Application	Temps (s)	Perf. cyl/car.	NOP %	Calcul %	Comm. %	Contrôle %	Total %	Utilisation %
RSA	73.05	395	59.30	15.73	0.957	24.49	100.50	41.20
IDEA	-	-	-	-	-	-	-	-

Tableau 4.5 : Cryptage -Performance – traitement de 10^6 caractères

Le tableau 4.6 présente les résultats obtenus après la simulation d'une itération de convergence avec l'algorithme de Poisson basé sur architecture maillée de dimension 32 x 32.

Application	Temps (ms)	Cycles	NOP %	Calcul %	Comm. %	Contrôle %	Total %	Utilisation %
1 itération	622.6	3362	23.5	60.98	38.13	7.763	130.40	106.90

Tableau 4.6 : Poisson, simulation d'une maille de 32 x 32 = 1024 PEs

Les deux tableaux suivants, 4.7 et 4.8, présentent les résultats de dilatation et d'érosion sur deux images de dimensions différentes : 128 x 128 et 256 X 256 pixels.

Application	Temps (μs)	Cycle / pixel	NOP %	Calcul %	Comm. %	Contrôle %	Total %	Utilisation %
Dilatation	690.7	2.27	11.12	43.94	154.10	11.63	220.80	209.68
Érosion	690.7	2.27	11.12	43.94	154.10	11.63	220.80	209.68

Tableau 4.7 : Morphologie Mathématique - Performance (128 x 128 image)

Application	Temps (μs)	Cycle / Pixel	NOP %	Calcul %	Comm. %	Contrôle %	Total %	Utilisation %
Dilatation	690.	2.27	0.36	44.00	165.3	11.30	221.00	220.64
Érosion	2.8	2.27	0.36	44.00	165.3	11.30	221.00	220.64

Tableau 4.8 : Morphologie Mathématique - Performance (256 x 256 image)

Les tableaux 4.3 à 4.8 montrent le résultat de simulations effectuées avec le simulateur **PULSE**. Le pourcentage d'utilisation, à défaut de prouver que l'application est efficace dans son traitement, montre à quel degré la bande passante de **PULSE** est utilisée. Rappelons qu'une utilisation élevée n'est pas directement une indication de la performance d'une application, mais souvent un bon indice. Cette information donne le pourcentage de travail effectué par les différentes composantes de **PULSE** durant le traitement d'un nombre élevé de données. Les pourcentages des tableaux précédents (4.3 à 4.8) démontrent les caractéristiques de construction du corps de chaque application.

Le tableau suivant, 4.9, établit une relation entre l'efficacité d'un programme et le pourcentage de calculs effectués durant la simulation. Effectuer un calcul produit un résultat qui éventuellement est la valeur numérique recherchée. Les deux autres types d'opérations effectuées, communication et contrôle, ne produisent pas de résultats

numériques mais servent plutôt à vérifier et déplacer les données lors du déroulement de l'application. Ce tableau montre, que dans le cas des applications développées, il y a une relation directe entre le pourcentage de calcul effectué et l'efficacité. Cette constatation comporte elle aussi une zone grise dans la mesure où trouver une façon rapide, avec un nombre restreint de calculs, de traiter un algorithme peut diminuer d'autant le pourcentage de calcul.

Application	Calcul (%)	Efficacité (%)
Mémoire interne	33.00	178
Matrice, décomposition sur les données	54.95	178
Matrice, décomposition sur l'architecture	23.24	80
Bresenham	25.44	100
RSA	15.73	Variable
Poisson	60.98	180
Érosion / Dilatation	44.0	144
IDEA	-	74

Tableau 4.9 : Relation entre calcul et efficacité

À l'intérieur du tableau 4.10 se trouvent les résultats obtenus pour deux facteurs discutés précédemment : le facteur d'accélération Sp et l'efficacité Ep. Ces valeurs sont évaluées par le simulateur lors de chacune des simulations.

Application	Sp	Ep (%)
Mémoire interne	1.330	44.32
Matrice, décomposition sur les données	1.274	42.46
Matrice, décomposition sur l'architecture	1.232	41.08
Bresenham	1.192	39.73
RSA	1.003	33.44
Poisson	1.304	43.46
Érosion / Dilatation	1.660	55.34
IDEA	-	-

Tableau 4.10 : Facteur d'accélération Sp et Efficacité Ep

Nous avons vu précédemment que le facteur d'accélération est le rapport entre les instructions utilisant le parallélisme à l'intérieur d'un même cycle d'horloge et les instructions simples. Sa valeur se situe entre 1 et 3, 3 étant le maximum possible. Une valeur de trois indique qu'à l'intérieur d'un même cycle trois opérations sont effectuées; traitement numérique, mouvement de données et communications. Une valeur de 1 indique qu'aucune instruction parallèle n'est utilisée à l'intérieur de l'application. Les résultats obtenus varient entre 1.003 et 1.660. Une valeur de 1.660 indique une accélération de 66% de l'application sur une possibilité de 200%.

Tous les programmes développés pour cette recherche l'ont été par la même personne, ce qui donne une uniformité dans la qualité des applications. Ce facteur donne un indice du

degré d'optimisation atteint pour les différentes applications. **Ep** revient à exprimer **Sp** en pourcentage, avec 100% équivalent à une valeur de 3 pour **Sp**.

La disponibilité de toutes les combinaisons d'instructions parallèles avec PULSE V2 devrait théoriquement faire grimper ce pourcentage.

Comparaisons avec d'autres architectures

Les comparaisons suivantes sont extraites de différentes bibliothèques construites autour de processeurs compétiteurs disponibles commercialement.

Ces comparaisons sont probablement le test ultime d'une application: peu de zone grise ici, seulement des chiffres nets montrant si les applications développées peuvent dépasser en performance la compétition.

Le tableau 4.11 donne le nombre de millions de vecteurs traités dans un intervalle de une seconde avec les données accessibles des mémoires interne et externe. Le gain par rapport au **C80** de Texas Instrument est de plus de 50% dans les deux cas. Le **C80** possède, à 50 MHz, une horloge légèrement moins rapide que celle de **PULSE** avec ses 54 MHz. Cette différence ne représente que 7%. Le C80 est une puce qui comporte 5 processors, un des processors est utilisé comme unité de contrôle ou pour effectuer les opérations en virgule flottante.

données internes		**données externes**	
C80 (10^6 vecteurs/sec)	PULSEV1 (10^6 vecteurs/sec)	C80 (10^6 vecteurs/sec)	PULSEV1 (10^6 vecteurs/sec)
3.1	5.56	1.6	2.78

Tableau 4.11 : Multiplication vecteurs – matrice, nombre de vecteurs traités

Le tableau 4.12 montre le nombre de cycles et l'intervalle de temps requis (μs) pour traiter 50 vecteurs d'une longueur de 10 pixels chacun avec **PULSE** et le **C80**. Dans ce cas, l'avantage peut être d'un côté comme de l'autre à cause du nombre variable d'instructions nécessaires au **C80** pour traiter les vecteurs. En assumant une vitesse d'horloge égale et une distribution linéaire des temps de traitements pour le **C80**, **PULSE** sera plus rapide dans 68% des cas.

	Processeurs	
	C80	**PULSEV1**
Cycles	2000-3650	2337
temps (μs)	40.0 - 73.0	43.27

Tableau 4.12 : Bresenham - Comparaison avec *UWICL* (50 segments de 10 pixels)

Le tableau 4.13 montre le temps d'exécution en secondes pour le cryptage d'un million de caractères de 8 bits avec l'algorithme **RSA** sur **PULSE** et sur une station de travail avec un processeur de 133 MHz. Le test effectué sur le Pentium 133 a été fait à partir d'un algorithme écrit en language C compilé avec gcc. Le temps d'exécution a été calculé avec la fonction time().

Bien que cette application comporte des problèmes d'optimisation à cause de l'emploi d'instructions conditionnelles, les résultats obtenus démontrent que si une instruction modulo rapide était disponible, **PULSE** pourrait jouer un rôle en ce qui concerne le cryptage de données.

Temps d'exécution (secondes)	
Pentium 133	PULSEV1
90	73.05

Tableau 4.13 : RSA – Comparaison (10^6 caractères de 8 bits)

Le dernier tableau, 4.14, compare encore une fois **PULSE** avec le **C80**. Les applications cibles sont dans ce cas-ci, la dilatation et l'érosion binaire. La différence de performance entre les deux puces disparaît presque entièrement dans le cas de **V1** si les horloges sont ramenées au même niveau : moins de 4 % (3.5%). Dans le cas de **V2** par contre, le nombre de processeurs permet un traitement plus rapide de grandes images et la différence est nettement plus importante (+75% avec même vitesse d'horloge).

Temps d'exécution			
	C80 (ms)	PULSEV1 54 MHz (ms)	PULSEV2 80 MHz (ms)
Dilatation	12.3	11.0	1.9
Érosion	12.3	11.0	1.9

Tableau 4.14 : Morphologie Binaire - Comparaison avec *UWICL* (512 x 512 images)

4.3. CONCLUSION

Les tableaux montrés dans ce chapitre donnent une représentation numérique des différents résultats obtenus au cours de ce projet. Les applications qui se démarquent en ce qui concerne la performance répondent généralement à trois critères comme soulignés dans ce chapitre :

1. Utilisation des instructions spéciales.
2. Applications qui ne nécessitent pas l'utilisation d'instructions conditionnelles.
3. Corps de programme qui comporte assez d'opérations pour pouvoir effectuer une mise en parallèle des instructions en déjouant les dépendances de données.

Les algorithmes de morphologie binaire et de Poisson répondent à ces critères et présentent à notre avis les résultats les plus intéressants de cette recherche.

L'optimisation de chacune des applications est un travail itératif où l'expérience du programmeur est un facteur excessivement important. Les applications dans ce projet ont été revisitées à maintes reprises. La qualité des programmes est donc généralement constante d'une application à l'autre.

Le développement des applications ne représente par contre qu'une partie du travail accompli depuis le début. Dans le prochain chapitre, le dernier de ce mémoire, un résumé basé sur les objectifs établis au début de cette recherche est présenté. On peut voir les différents impacts sur l'architecture de deuxième génération de PULSE générés lors du développement de cette recherche.

CHAPITRE 5

CONCLUSION

Comme présenté au chapitre 1, ce projet avait deux objectifs principaux :

1. Le développement d'applications diverses et évaluation de leurs performances sur **PULSE**.
2. Le test et évaluation des outils en place, et apporter si possible des suggestions pour améliorer le produit.

5.1. DÉVELOPEMENT D'APPLICATIONS

Durant ce projet huit applications différentes ont été étudiées. De ces huit, une a été abandonnée avant d'avoir atteint le stade de l'écriture de code, il s'agit de la transformée de Hough. Cette transformée est un algorithme qui s'applique sur une image et qui à prime abord, comportait des éléments intéressants pour **PULSE** : un traitement intensif, de multiples opérations à effectuer sur les mêmes pixels et la possibilité de réutilisé des algorithmes déjà écrits sur **PULSE** :

- Calcul du niveau de gris (Sobel),
- Détection de contours (threshold),
- Calcul du gradient (direction),
- Calcul de la position dans un espace spécial nommé espace de Hough,
- Production d'un histogramme,

Deux facteurs ont mené à l'abandon de cet algorithme, soit le calcul de la position dans l'espace de Hough et le calcul du gradient qui demandait l'emploi d'une division.

Des sept applications restantes, six ont permis d'effectuer de multiples simulations et itérations sur les algorithmes.

La septième, l'algorithme de chiffrage IDEA a été développé en entier. La non-disponibilité d'une instruction conditionnelle sur 32 bits afin d'effectuer la multiplication modulo $2^{16} + 1$ de façon rapide a rendu inutile toute optimisation.

Les autres applications ; multiplication vecteurs – matrice, algorithme de Bresenham, algorithme RSA, Équation aux différences finies, Morphologie binaire avec l'Érosion et la

Dilatation d'images, présentent une variété d'applications numériques. Une des requêtes du groupe **PULSE** était d'évaluer avec quelle facilité, ou difficulté, différents types d'algorithmes pouvaient être portés sur **PULSE** et d'essayer de sortir des traits caractéristiques qui fonctionnaient bien sur **PULSE**.

5.1.1. Développement sur PULSE

Basé sur les résultats des comparables vus au chapitre précédent, les chiffres obtenus lors de ce projet se comparent avantageusement à ce qui se fait à l'extérieur. On note cependant une lacune évidente. En effet, le travail de cette recherche s'est étendu sur environ 16 mois, incluant une session passée à temps plein sur l'avancement des travaux de recherches. En mettant en relation le temps investi dans ce projet et les quelques centaines de lignes de code écrites, il y a un déséquilibre important entre les deux.

Ce déséquilibre s'explique par plusieurs facteurs qui font partie des difficultés rencontrées durant ce projet.

1- **Développement d'applications performantes :** il s'agit du critère majeur de développement qui a dirigé les actions et les changements effectués durant la phase de développement. Les multiples itérations et simulations n'avaient qu'un seul but : traiter plus de données en un même laps de temps. La recherche de vitesse pour chacune des applications se joue au niveau des instructions parallèles et par le retrait des instructions vides (*NOP*). Cette recherche nécessite souvent un réarrangement de la façon donc se déroule le programme, de multiples changements mineurs et des tests de performances entre chaque changement.

2- **Diversité des applications :** Au lieu d'exploiter un filon d'application similaire et de développer mes connaissances dans ce domaine spécifique, j'ai appliqué les directives du groupe en touchant à différents domaines.

3- **Courbe d'apprentissage :** Hormis une introduction au logiciel PVM sur UNIX, à l'occasion d'un cours sur les réseau d'ordinateurs, **PULSE** aura été une première expérience de programmation sur un système parallèle. Le fait de prendre plusieurs semaines en début de projet pour étudier les grands principes du parallélisme et les quatre compétiteurs de PULSE aura eu un effet d'ouverture sur la suite de ce projet. La plus grande difficulté d'apprentissage s'est trouvée dans l'environnement de développement de **PULSE**. Une architecture et un langage de programmation qui rivalisent de complexité l'un avec l'autre. Il est certain que le fait de se retrouver à 500 kilomètres des autres membres du groupe a contribué aux difficultés d'apprentissage durant le déroulement de ce projet. Je crois pouvoir affirmer avec expérience que l'ère des communications améliore grandement la rapidité et les possibilités de développement en différent lieu physique, une fois la courbe d'apprentissage gravie.

4- **Produit non totalement stable :** une partie du projet était de pouvoir améliorer l'environnement de développement, l'architecture et de pouvoir définir les limitations courantes de **PULSE**. Le déroulement du projet s'est donc effectué sur un produit en développement. Ainsi, plusieurs problèmes de conception ont été mis à jour, tant dans le circuit que dans ses outils de développement. C'est un privilège

de pouvoir participer à la validation d'un produit mais cela comporte aussi des contraintes dans le développement et la validation d'applications.

5- **Documentation :** la documentation présente possédait des erreurs, ce qui est normal pour un produit en phase de développement.

5.1.2. Test, Évaluation et Recommandation

Cette partie est faite d'un document présenté au groupe au mois d'avril 1998 (section suivante) et d'une liste de recommandations faite pour le simulateur de deuxième génération.

5.1.2.1. Limites de l'architecture de PULSE V1

Un document intitulé *Limites de l'architecture PULSE V1* a été présenté au groupe. Ce rapport présentait de façon chronologique les différentes limitations de PULSE V1 rencontrées depuis le début de ce projet.

Voici un sommaire des principales limitations identifiées :

Caractéristique	*PULSE V2*
1. Opération modulo	
2. Boucle variable	✓
3. Combinaison d'instruction	✓
4. Sortie d'une instruction conditionnelle	
5. Insertion d'instruction utile avec un if	
6. Instructions conditionnelles imbriquées	✓
7. Load de 4 cycles	
8. Chaîne d'accumulation	
9. Opération en virgule flottante	✓
10. Mouvement à gauche sur les canaux Nord et Sud	✓
11. Canaux Nord et Sud à configuration modifiable	
12. Décalage sur 16 bits	✓
13. Compteur modulo interne sur 32 bits	✓
14. If sur une donnée de 32 bits	✓

✓ :implanté dans V2

Tableau 5.1 : Liste des recommandations et limitations rencontrées

1. Opération Modulo

La seule instruction modulo disponible est effectuée par l'assembleur, **PULSE** ne fait que traiter le résultat de l'opération. Dans le cas des deux algorithmes de cryptage étudiés (RSA et IDEA) une opération modulo aurait été très efficace.

2. Boucle variable

Possibilité d'avoir des boucles de dimensions variables. Il faudrait pouvoir employer l'instruction *push* avec, comme opérande, une adresse mémoire.

3. Combinaison d'instructions

La version 1 de **PULSE** possède un nombre limité de combinaisons d'instructions parallèles.

4. Mécanisme de sortie d'instruction conditionnelle

Lors de l'exécution d'une séquence conditionnelle, même si aucun des processeurs n'est activé, les instructions sont effectuées à vide. La solution trouvée réside dans l'utilisation de l'instruction *bnpa* (branch if no active PEs). Il y a insertion de quatre *NOPs* entre le *if* et le *bnpa* si le *bnpa* est suivi d'une instruction ou insertion de deux *NOPs* si le *bnpa* est suivi d'un *else* ou d'un *restore.*

5. Insertion d'instructions utiles avec un if

L'instruction *if* comporte quatre modes d'opération qui permettent de conserver ou non la fonctionnalité des canaux de communications même si les PEs sont inactifs. Il devrait être possible de remplacer les *NOPs* par des instructions utiles (*nsr, ssr,* ...) combinées avec *nodep.*

.nodep

if xxxx

instruction utile 1(*n2ssr*)

instruction utile 2(*n2ssr*)

instruction du *if* (*abs* ra2, rb5)

.dep

L'assembleur génère deux *NOPs* après les instructions utiles 1 et 2 avec l'utilisation d'une instruction de branchement conditionnel.

.nodep

if xxxx

instruction utile 1(*n2ssr*)

instruction utile 2(*n2ssr*)

#2 *NOP*

b*cond*(*bnpa* coco)

instruction du *if* (*abs* ra2, rb5)

.dep

6. Instructions conditionnelles imbriquées

Lors de l'utilisation de d'instructions conditionnelles imbriquées (*if*), les conditions doivent se propager à travers les différentes conditions successives rencontrées. Ce qui n'était pas le cas.

7. *Load* de 4 cycles

Le *load* devrait pouvoir se faire en 1 cycle comme le *fwd*.

8. Chaîne d'accumulation

La chaîne d'accumulation s'arrête avec le dernier PE de la puce. L'idée était de pouvoir utiliser la donnée comprise dans l'accumulateur du PE3 comme source pour le PE0 avec l'instruction *maddl*.

maddl $src1_{16}$, $src2_{16}$, acc_{PE3}, acc_{PE0}

Une autre suggestion était de se servir des canaux Nord et Sud au lieu de l'accumulateur pour avoir une chaîne d'accumulation de 32 bits entre le PE3 et le PE0.

maddl $src1_{16}$, $src2_{16}$, nsport, nsport

9. Opération en virgule flottante

Ajout d'une unité de traitement des nombres en virgule flottante.

10. Mouvement à gauche sur les canaux Nord et Sud

Utilisation des canaux Nord et Sud pour pouvoir effectuer des rotations de données vers la gauche.

11. Ports Sud et Nord à configuration modifiable

Possibilité de modifier la configuration des canaux de communication afin de pouvoir passer directement une donnée du registre mémoire du canal au registre mémoire de n'importe quel autre PE.

12. Décalage sur 16 bits

Possibilité de faire des opérations de décalage avec source et destination de 16 bits.

13. Compteur modulo (32 bits) interne

L'objectif est de pouvoir utiliser les mémoires internes a et b comme une mémoire de 32 bits et de pouvoir incrémenter et décrémenter les adresses mémoires avec une seule opération.

14. Instruction conditionnelle sur 32 bits

Possibilité d'utiliser l'instruction *if* avec une source et une destination de 32 bits.

5.1.2.2. Recommandations pour le simulateur de deuxième génération.

Durant les derniers mois de 1997, Nicolas Contandriopoulos a mis en chantier le simulateur de deuxième génération, une liste de recommandations comportant les points suivants lui a été soumise :

1- Production de fichiers résultats avec de l'informations supplémentaires.

2- Possibilité de sauvegarder les fichiers résultats automatiquement.

3- Recommandation sur l'implémentation de la nouvelle interface.

4- Interface usager configurable.

5- Lors de l'exécution pas à pas, mettre en évidence les valeurs et registres modifiées.

6- Possibilité d'insérer des points d'arrêt logiciels et matériels dans le code.

5.2. CONCLUSION

Durant ce projet de maîtrise, une approche ouverte, sans connaissance préalable des applications qui «fonctionnent » bien sur une puce de type SIMD, a été adoptée.

Les applications étudiées proviennent de différents domaines. Nous avons pu voir que PULSE performe bien pour les algorithmes basés sur les opérations mathématiques de base, que les instructions conditionnelles sont à oublier, et que l'emploi des instructions spéciales, dans le cas des algorithmes de Poisson et de Morphologie Mathématique, permet d'augmenter la performance de façon considérable.

Je suis convaincu que la contribution de ce projet a aidé les gens de **PULSE** à faire avancer et améliorer leur produit. Les différents documents et présentations produites en cours de projet en plus des recommandations apportées sur l'architecture et le simulateur en sont le résultat.

Bibliographie

[ADAP94] Adaptive Soulution, Inc., Getting Acquainted with CNAPS, décembre 1994

[ANAL95] Analog Devices, Inc., ADSP-2106x User's Manual, mars 1995

[BELA97] Normand Bélanger, Opération Modulo Rapide en l'Absence de Diviseur Matériel, PULSE, 1997

[BJOR00] P. Björstad, F. Manne, T. Sörevik, M. Vajtersic, Efficient Matrix Multiplication on SIMD Computers, Institutt for Informatikk, University of Bergen, Bergen, Norvège

[BRES65] Bresenham, J. E., Algorithm for Computer Control of a Digital Plotter, IBM System Journal, Vol. 4, No. 1, 1965, pp 25–30

[CAMB88] Cambridge Parallel Processing, Dap Series: Image Processing Library, 1993, Cambridge Parallel Processing Ltd., UK

[DELL99] Dell Computing, Vectors; IA Architecture and Itanium, Dell Computer Corporation, 1999

[FLYN66] M.J. Flynn, Very High Speed Computing Systems, Proceeding IEEE, 1966

[FOLE83] J. D. Foley, A van Dam, Fundamentals of Interactive Computer Graphics, Addison-Wesley Publishing Co., 1983, pp 432–439.

[GENG96] M. Gengler, S. Ubéda, F. Desprez, Initiation au parallélisme: concepts, architectures et algorithmes, Masson, Paris, 1996

[GERB02] Richard Gerber, The Software Optimization Cookbook, Intel Press, USA, 2002

[GOOR89] A. J. van de Goor, Computer Architecture and Design, Addison Wesley, USA, 1989

[HALS94] F. Halsall, Data Communications, Computer Networks and Open Systems, Thrid Edition, Addison-Wesley, USA, 1994,

[HUSS91] Z. Hussain, Digital Image Processing : Practical Applications of Parallel Processing Techniques, Ellis Horwood, Chichester, Englang, 1991

[INTL00] Intel, Intel IA-64 – Architecture Software Developer's Manual, Intel Corporation, Revision 1.1, Juillet 2000

[INTLXX] Intel, ASC Performance Terminology Concepts, Intel Corporation,

[JAJA92] J. Jájá, An Introduction to Parallel Algorithms, Addison-Wesley, USA, 1992

[KAYA94] Çetin Kaya Koç, High-Speed RSA Implementation, RSA Laboratories, 1994, p. 10-11, Redwood City, USA

[KRAJ97] Ivan Kraljic, Mapping Image Processing Algorithms on PULSE, PULSE, 1997

[OXFR96] Steven G. Morton, A236 Parallel DSP Chip, Oxford Computer, Inc., octobre 1996

[PARK97] J. R. Parker, Algorithms for Image Processing and Computer Vision, John Wiley & Sons, USA, 1997

[PITA93] Ioannis Pitas, Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks, John Wiley & Sons, England, 1993

[POLA96] S. Poland, TMS320C8X (DSP), Fundamental Graphic Algorithms, Texas Instruments, 1996, literature number SPRA069, pp 1–32

[PULS97] Tahar Ali Yahia, Christophe Bonello, PULSE Instruction Set V1, Revision 1.0, Février 97

[PULS96] PULSE Development Team, PULSE V1 Overview, Revision 1.0, Juin 1996

[STON93] Harold S. Stone, High-Performance Computer Architecture, Addison-Wesley, USA, 1993

[TI96] Texas Instruments, TMS320C80 Technical Reference, 1996

[VELD94] Eric F. Van de Velde, Concurrent Scientific Computing, Springer-Verlag, New-York, 1994

ANNEXE A

APPLICATIONS DÉVELOPPÉES

Multiplication vecteurs – matrice

Version de base

```
;*************************************************************/
;*
;*      PROGRAMME           : basic.asm      , 200 vecteur
;*      date                : avril 97
;*      crée par            : Jean-Jacques Clar
;*
;*         Utilisation de 2 mémoires externes: Entrée : A et B
;*                                              Sortie: C
;*        Entrée :     port 1 to north channel (Matrice A et Matrice B)
;*        Sortie :     port 3 to north channel (accumulator lsw)
;*                     port 4 to south channel (accumulator msw)
;*
;*   Description : multiplication de deux matrices
;*        - la multiplication se fait en traitant la matrice B comme quatre
;*      vecteurs différents de dimension [1 x 4]
;*        - la colonne i de la matrice A est chargée dans le PEi
;*        - la ligne   i de la matrice B est chargée dans le port nord
;*      ( la même valeur dans les 4 registres )
;*        - les multiplications-additions sont mises en pipeline dans chaque PE
;*
;*************************************************************/

        .global matrix
        .data

        .title "********** listing du fichier matrix.asm **********"

        .text

matrix:
```

```
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; configuration des ports d'E/S
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; ldcr: Load Configuration Register at address dst
ldcr 2, port1config  ; (90 2)  ; port 1 as input, synchronoeus mode
ldcr 1, port3config  ; (92 3)  ; port 3 as output, synchroneous mode
ldcr 3, port4config  ; (92 3)  ; port 4 as output, synchroneous mode

; configuration de la connexion des ports d'E/S
ldcr 1, port1incon   ; (A0 0)  ; port 1 as connect to north channel
ldcr 1, port3outcon  ; (B2 0)  ; port 3 as connect to north channel
ldcr 2, port4outcon  ; (B3 1)  ; port 4 as connect to south channel

; initialisation du processeur
; 3 -> boot from internal memory
;    address port 1 use modulo counter
ldcr 3, bootcontrol  ; Use internal program memory and MCC for address 1

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; initialisation des compteurs modulo externes mcc, mcd
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ldeamc 0, 0200000h, mc_min, mc_min;
ldeamc 040000h, 0240000h, mc_max, mc_max;
ldeamc 0, 0200000h, mc_start, mc_start;
ldeamc 1, 1, mc_stride, mc_stride;

io *mccr%; toujours ajouter après chargement des compteurs
io *mcdr%; toujours ajouter après chargement des compteurs

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;          matrice 4 x 4    A x B =C
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   ; initialisation des compteurs modulo interne mca, mcb
   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   ldiamc 1, 0, 4, mcar;

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;          traitement du premier vecteur
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   ; chargement de la matrice A ligne par ligne
   ; chaque PE a la colonne PEi en mémoire
   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
    push 4; nombre de lignes
LoadA:
    #4 nsr || io *mccr%  ; remplissage du canal nord avec une ligne
    ld nport, *mcaw(1) ; matrice A charge la colonne i dans le PEi

    dbr LoadA
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; A est complètement chargée dans les 4 PEs
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 #4 nsr
    push 200; nbs vecteurs a traiter
Vect:
 push 4; chaque vecteur a quatre termes a traiter
V_Term:
    macc nport, *mcar(1) || io *mccr%; multiplication addition
 #3 nsr
 nsr
    dbr V_Term

 ld acc, nsport
    ld 0, acc

    #3 nsr || ssr || io *mcdr%  ; sortie des résultats dans mémoire externe

    nsr || ssr || io *mcdr%  ; sortie des résultats dans mémoire externe
    dbr Vect

    end
    .end
```

Version utilisant mémoire interne

```
;*************************************************************/
;*              données à l'intérieur de la mémoire
;*    PROGRAMME          : sys.asm , 10 vecteur
;*    date               : août 97
;*    crée par           : Jean-Jacques Clar
;*
;*  Entrée :    la matrice A est dans le registre 1
;*              les vecteurs sont dans la mémoire A
;*  Sortie :    les résultats sont envoyés par les canaux nord
;*              et sud
;*
;* Description : multiplication de deux matrices
;*
;*              a e i m          w           w'
;*              b f j n          x           x'
;*              c g k o     .    y     =     y'
;*              d h l p          z           z'
;*
;*************************************************************/

    .include "data.asm"
    .global matrix
    .title "********** listing du fichier sys.asm **********"

    .text

matrix:
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; configuration des ports d'E/S
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; ldcr: Load Configuration Register at address dst
    ldcr 3, port3config  ; (92 3)  ; port 3 as output, synchroneous mode
    ldcr 1, port4config  ; (92 3)  ; port 4 as output, synchroneous mode

    ; configuration de la connexion des ports d'E/S
    ldcr 1, port3outcon  ; (B2 0)  ; port 3 as connect to north channel
    ldcr 2, port4outcon  ; (B3 1)  ; port 4 as connect to south channel

    ; initialisation du processeur
```

```
    ; 3 -> boot from internal memory
    ;     address port 1 use modulo counter
    ldcr 3, bootcontrol  ; Use internal program memory and MCC for address 1

    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; initialisation des compteurs modulo externes mcc, mcd
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ldeamc 0, 0200000h, mc_min, mc_min;
    ldeamc 0, 0220000h, mc_max, mc_max;
    ldeamc 1, 0200000h, mc_start, mc_start;
    ldeamc 1, 1, mc_stride, mc_stride;

    io *mccr%; a toujours ajouter après chargement des compteurs
    io *mcdr%; a toujours ajouter après chargement des compteurs

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;           matrice 4 x 4     A x B =C
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; initialisation des variables
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    Nb_vecteurs .set 200

    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; initialisation des compteurs modulo interne mca, mcb
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ldiamc 0, 0, 255, mcar;

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;          traitement des vecteurs
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    push Nb_vecteurs; nombre de vecteurs a traiter
Vect:
    mult ra1, *mcar(1), acc
    macc ra2, *mcar(1)
    macc ra3, *mcar(1)
    macc ra4, *mcar(1)|| io *mcdr%
    #3 nsr || ssr || io *mcdr%
    ld acc, nsport
    dbr Vect

    end
    .end
```

Pseudo-systolique

```
;**************************************************************/
;*              architecture pseudo-systolique
;*      PROGRAMME           : sys.asm  , 1 vecteur
;*      date                : avril 97
;*      crée par            : Jean-Jacques Clar
;*
;*        Utilisation de 2 mémoires externes:          Entrée : A
;*                                                   Sortie: C
;*       Entrée : port 1 canal Nord (Matrice A et Matrice B)
;*       Sortie  : port 3 canal Sud (accumulateur lsw)
;*                     port 4 canal Sud (accumulateur msw)
;*
;*   Description : multiplication de deux matrices
;*      - la multiplication se fait en traitant la matrice B comme quatre
;*        vecteurs différents de dimension [1 x 4]
;*      - la colonne i de la matrice A est chargée dans le PEi
;*          - le vecteur b est charger dans le canal Nord et il subit ensuite une rotation
;*            pour que chaque PE puissent utiliser chaque terme
;*      - les multiplications-additions sont mises en pipeline dans chaque PE
;*
;*              a e i m            w            w'
;*              b f j n            x            x'
;*              c g k o      .     y      =     y'
;*              d h l p            z            z'
;*
;**************************************************************/

    .global matrix
    .data

    .title "********** listing du fichier sys.asm **********"

    .text

matrix:
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; configuration des ports d'E/S
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; ldcr: Load Configuration Register at address dst
```

```
ldcr 2, port1config  ; (90 2)  ; port 1 as input, synchroneous mode
ldcr 0, port2config  ; (91 0)  ; port 2 as input
ldcr 1, port3config  ; (92 3)  ; port 3 as output
ldcr 3, port4config  ; (92 3)  ; port 4 as output, synchroneous mode

; configuration de la connexion des ports d'E/S
ldcr 1, port1incon   ; (A0 0)  ; port 1 as connect to north channel
ldcr 2, port2incon   ; (A1 2)  ; port 2 as connect to south channel
ldcr 1, port3outcon  ; (B2 0)  ; port 3 as connect to north channel
ldcr 2, port4outcon  ; (B3 1)  ; port 4 as connect to south channel

; initialisation du processeur
; 3 -> boot from internal memory
;     address port 1 use modulo counter
ldcr 3, bootcontrol  ; Use internal program memory and MCC for address 1

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; initialisation des compteurs modulo externes mcc, mcd
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ldeamc 0, 0200000h, mc_min, mc_min;
ldeamc 16, 0200400h, mc_max, mc_max;
ldeamc 1, 0200000h, mc_start, mc_start;
ldeamc 5, 1, mc_stride, mc_stride;

io *mccr%; a toujours ajouter après chargement des compteurs
io *mcdr%; a toujours ajouter après chargement des compteurs

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;           matrice 4 x 4     A x B =C
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; initialisation des compteurs modulo interne mca, mcb
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ldiamc 1, 0, 4, mcaw;

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;          traitement de la matrice A
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    push 4; nombre de lignes
LoadA:
    #4 nsr || io *mccr%  ; remplissage du canal nord avec une ligne
    ld nport, *mcaw(1) ; matrice A charge la colonne i dans le PEi

    dbr LoadA
```

```
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;          traitement des vecteurs
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ; réinitialisation compteur modulo c
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ldeamc 2097152, 0240000h, mc_max, mc_max;
     ldeamc 17, 0200000h, mc_start, mc_start;
     ldeamc 1, 1, mc_stride, mc_stride;
     ld 0, acc || io *mccr%, *mcdr%; a toujours ajouter après chargement des
compteurs

     #3 nsr || io *mccr%
     macc nport, *mcar(1) || nsr || io *mccr%

     push 1; nombre de vecteurs a traiter
Vect:
.nodep
     ; multiplication addition vecteur
     nrr
     macc nport, *mcar(1)
     nrr
     macc nport, *mcar(1)
     nrr
     macc nport, *mcar(1)

 ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 ;          fin de traitement du vecteur
 ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 ;          sortie des résultats
 ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ld acc, nsport
     srl acc, 1, r1
     sll acc, 2. ra2
     sll acc, 2, rb2
     sll acc, 2, r3
     ld 0, acc
     nop
 io *mcdr%
     #3 nsr || ssr || io *mcdr%, *mccr%
     macc nport, *mcar(1) || nsr || ssr || io *mccr%
.dep
```

```
dbr Vect

end
.end
```

Décomposition basée sur l'architecture

```
;*****************************************************************/
;*              décomposition basée sur l'architecture
;*               architecture pseudo-systolique
;*     PROGRAMME           : sys.asm ,
;*     date                : avril 97
;*     crée par            : Jean-Jacques Clar
;*
;*
;*****************************************************************/

    .global matrix
.data
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; chargement des mémoires des PEs, matrice A et premier vecteur
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ;PE0
    .init 0
    .ma1 30,32,34,36 ;m,n,o,p
    .mb0 44 ;z

    ;PE1
    .init 1
    .ma1 28,22,24,26 ;l,i,j,k
    .mb0 42 ;y

    ;PE2
    .init 2
    .ma1 18,20,14,16 ;g,h,e,f
    .mb0 40 ;x

    ;PE3
    .init 3
    .ma1 8,10,12,6 ;d,c,b,a
    .mb0 38 ;w

    .title "********** listing du fichier sys.asm **********"

    .text
```

```
matrix:
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; configuration des ports d'E/S
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; ldcr: Load Configuration Register at address dst
        ldcr 2, port1config  ; (90 2)  ; port 1 as input, synchroneous mode
        ldcr 1, port3config  ; (92 3)  ; port 3 as output, synchroneous mode
        ldcr 3, port4config  ; (92 3)  ; port 4 as output, synchroneous mode

        ; configuration de la connexion des ports d'E/S
        ldcr 1, port1incon   ; (A0 0)  ; port 1 as connect to north channel
        ldcr 1, port3outcon  ; (B2 0)  ; port 3 as connect to north channel
        ldcr 2, port4outcon  ; (B3 1)  ; port 4 as connect to south channel

        ; initialisation du processeur
        ; 3 -> boot from internal memory
        ;     address port 1 use modulo counter
        ldcr 3, bootcontrol  ; Use internal program memory and MCC for address 1

        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; initialisation des compteurs modulo externes mcc, mcd
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ldeamc 20, 0200000h, mc_min, mc_min;
        ldeamc 250000, 0220000h, mc_max, mc_max;
        ldeamc 20, 0200000h, mc_start, mc_start;
        ldeamc 1, 1, mc_stride, mc_stride;

        io *mccr%; a toujours ajouter après chargement des compteurs
        io *mcdr%; a toujours ajouter après chargement des compteurs

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;           matrice 4 x 4     A x B =C
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; initialisation des compteurs modulo interne mca, mcb
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ldiamc 1, 1, 4, mcar;
        ldiamc 5, 5, 255, mcaw;

        ldiamc 0, 0, 255, mcbr;
        ldiamc 1, 1, 255, mcbw;

        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
        ; initialisation des variables
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        Vecteurs .set 200

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;           traitement des vecteurs
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;       #3 nsr || io *mccr%
;       fwd nport, *mcbw(1) || io *mccr%

        push Vecteurs; nombre de vecteurs a traiter
Vect:
        mult *mcbr, *mcar(1), nsport
        nrr || ssr

        madd *mcbr, *mcar(1), nsport, nsport
        nrr || ssr

        madd *mcbr, *mcar(1), nsport, nsport
        nrr || ssr

        madd *mcbr(1), *mcar(1), nsport, nsport
  nop
        ldiamc 1, 1, 4, mcar
        io *mcdr%
        #3 nsr || ssr || io *mccr%,*mcdr%
        fwd nport, *mcbw(1) || io *mccr%
        dbr Vect

        end
        .end
```

Algorithme de Bresenham

```
;*************************************************************/
;*      PROGRAMME           : bre.asm
;*      date                : janvier 98
;*      crée par            : Jean-Jacques Clar
;*
;*        traitement de 100 vecteurs
;*
;*        validation résultats avec simulateur .bin  9/2/98
;*
;*************************************************************/
.global bres

    .title "********** listing du fichier bres.asm **********"
    .text

bres:
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; configuration des ports d'E/S
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; ldcr: Load Configuration Register at address dst
    ldcr 2, port1config  ; (90 2)  ; port 1 as input, synchroneous mode
    ldcr 0, port2config  ; (91 0)  ; port 2 as input
    ldcr 1, port3config  ; (92 3)  ; port 3 as output
    ldcr 3, port4config  ; (92 3)  ; port 4 as output, synchroneous mode

    ; configuration de la connexion des ports d'E/S
    ldcr 1, port1incon   ; (A0 0)  ; port 1 as connect to north channel
    ldcr 2, port2incon   ; (A1 1)  ; port 1 as connect to south channel
    ldcr 1, port3outcon  ; (B2 0)  ; port 3 as connect to north channel
    ldcr 2, port4outcon  ; (B3 1)  ; port 4 as connect to south channel
```

```
        ; initialisation du processeur
        ldcr 3, bootcontrol  ; Use internal program memory and MCC for address 1

        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; initialisation des compteurs modulo externes mcc, mcd
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ldeamc 0, 0, mc_min, mc_min;
        ldeamc 23, 2097152, mc_max, mc_max;
        ldeamc 0, 0, mc_start, mc_start;
        ldeamc 1, 1, mc_stride, mc_stride;

        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; initialisation des compteurs modulo internes
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ldiamc 2, 2, 3, mcar;
        ldiamc 2, 2, 3, mcaw;
        ldiamc 2, 2, 3, mcbr;
        ldiamc 2, 2, 3, mcbw;

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;               début du programme
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; début de traitement
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; entrée des coordonnées des quatre premières lignes
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ;  (x1, y1), (x2, y2), value
        #3 nsr || ssr || io *mccr%; (x1, y1)
        fwd sport, *mcbw(1) ;y1
        fwd nport, *mcaw(1) ;x1

        #4 nsr || ssr || io *mccr%; (x2, y2)

        push 25 ; nb de lignes a traiter/4
Newline:
.nodep
        ;dy = ABS(y2-y1)
        cas *mcbr(1),sport ,@1 || nsr || ssr || io *mccr%    ; ma1->lsw , y1
                                                             ; mb1->msw , y2
        cas nport, *mcar(1),@0 ; ma1->lsw , x1
                               ; mb1->msw , x2
```

```
	#2 nsr || ssr || io *mccr%; (x2, y2)

	sub @a1, @b1, ra3	; ra3 -> dy
	sub @a0, @b0, rb0	; rb0 -> dx
	nsr || ssr || io *mccr%; (x2, y2)
	sla r3, 1, r1	; ra1 -> incr1 = 2 * dy
	sub ra3, rb0, ra2 ; ra2 -> incr2 = dy - dx
	fwd sport, *mcbw(1) ;prochain y1
	fwd nport, *mcaw(1) ;prochain x1

	;d  = incr1 - dx
	sub ra1, rb0, rb3 ; rb3 -> d
.dep
	;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
	; traitement des lignes
	;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
	;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
	; traitement du premier pixel
	;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
	inc @b0, @b0		; x1
	iflt rb3, 0, 1;c_actived  ;d<0
	 add rb3, ra1, rb3	; d = d + incr1
	else
	 add rb3, ra2, rb3	; d = d + incr2
	 inc @b1, @b1		; y1
	restore
	stc @b0, nport
	stc @b1, sport

	;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
	; traitement des autres pixels
	;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
	push 9; nombre de pixel max dans un écran
Pixel:
.nodep
	inc @b0, @b0 || io *mcdr%	;sortie pe3 et incr x1
	iflt rb3, 0, 1  ;c_actived  ;d<0
	 #2 nsr || ssr || io *mcdr%	;sortie pe2-pe1
	 add rb3, ra1, rb3	; d = d + incr1
.dep
	else
	 nsr || ssr || io *mcdr% ; sortie pe0
	 add rb3, ra2, rb3	; d = d + incr2
	 inc @b1, @b1		; y1
```

```
restore
stc @b0, nport
stc @b1, sport
dbr Pixel

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; sortie résultats dernier pixel
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
io *mccr%, *mcdr%        ;sortie pe3
mult 0, 0, r3 || nsr || ssr || io *mccr%, *mcdr% ; (x2, y2)
, sortie pe2
nsr || ssr || io *mccr%, *mcdr%; (x2, y2), sortie pe1
nsr || ssr || io *mccr%, *mcdr%; (x2, y2), sortie pe0
dbr Newline

end
.end
```

RSA

```
;*************************************************************/
;*     PROGRAMME            : rsa_bin.asm
;*     date                 : janvier 98
;*     crée par             : Jean-Jacques Clar
;*
;*       Utilisation de 2 mémoire externe :  Entrée : A
;*                                           Sortie: C
;*   Entrée      : port 1 canal Nord (plaintext)
;*   Sortie      : port 4 canal Sud (ciphertext)
;*
;*   description: modulo avec recherche binaire
;*
;*
;*************************************************************/
.data
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; définition des variables
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ;var   def                  mémoire
    ;C  : texte crypté            @a1
    ;P  : plaintext             nport(16 bits), @b0
    ;p  : exposant de P            @a12
    ;N  : partie de la clé         @a11
    ;n  : exposant de N            @a13
    ;a  : borne inférieure exposant ra1
    ;b  : borne supérieure exposant rb1
    ;I  : Intervalle médian            ra5
```

```
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; chargement des mémoires des PEs
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.init
.ma1 1      ; C = 1
.ra0 1      ; reinitialisation de C = 1(madd 0, 0, 1, @1: C=0x0+1)
.ra10 253   ; N
.ma10 253   ; N
.ra9  8     ; n, exposant de N
.ra11 13    ; n + I
.mb11 23    ; E
.mb12 5     ; e, exposant de E
.ma13 8     ; p, exposant de P
.ra1  0     ; a,
.rb1  10    ; b, p + 1
.ma100      1,1,1,1,1,0,1,0 ;230 en binaire k = 8

.title "********** listing du fichier rsa.asm **********"
.text

rsa:
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; configuration du chips
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ldcr 2, port1config  ; (90 2)  ; port 1 as input, synchroneous mode
ldcr 3, port4config  ; (92 3)  ; port 4 as output, synchroneous mode
ldcr 1, port1incon   ; (A0 0)  ; port 1 as connect to north channel
ldcr 2, port4outcon  ; (B3 1)  ; port 4 as connect to south channel

ldcr 3, bootcontrol  ; Use internal program memory and MCC for address 1

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; initialisation des compteurs modulo externes mcc, mcd
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ldeamc 0, 1, mc_min, mc_min
ldeamc 160, 250, mc_max, mc_max
ldeamc 0, 1, mc_start, mc_start
ldeamc 1, 1, mc_stride, mc_stride

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; initialisation des compteurs modulo interne mca, mcb
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ldiamc 100, 100, 107, mcar;
```

```
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;           début du programme
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     #3 nsr ;|| io *mccr% ; 4 premiers caractères

     push 1; nombre de caractères a crypter / nb PEs
Cipher:
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ; calcul le ciphertext C du plaintext P
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ; premier tour N > P
     fwd nport, @b0; P->b0

     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ; square and multiply algorithm (méthode binaire)
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ; premier bit, le résultat est certainement + < N, pas de modulo
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ifne *mcar(1), 1, 0
        ld 1, @a1       ; C = 1
     else
        fwd nport, @a1   ; C = P
     restore
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ; bits suivant
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
         push 7 ;for (i=1; i=e-1; i++)
exposant de E
modci:
.nodep
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ; toujours, bit = 1 ou 0
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ;C = mod(P*C, N)

     ;a) C = C * C (mod n)
     mult @a1 , @a1, @1
     call coco

     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ; seulement si bit = 1
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
      ;b) C = C * M (mod n)
      ifeq *mcar(1), 1, 0
         bnpa nob    ; si le bit est égal a zéro
      mult @b0 , @a1, @1 ; c*p
      call coco
nob:
      restore
      stc @a1, sport
      dbr modci
.dep
      ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      ; sortie du texte crypter
      ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      madd  0, rb0, ra0, @1 ;|| io *mcdr%, *mccr%
      ; ld 1, C
      #2 nsr || ssr ;|| io *mcdr%, *mccr%
      nsr || ssr ;|| io *mcdr%, *mccr%

      dbr Cipher

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;     fonction modulo
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
coco:
      ld 5, ra3     ; I milieu, Intervalle, remplace un nop
.nodep
      sra @1, ra11, @3 ; n + I
      sra @1, ra9, @2 ; rot sur C de n, pour avoir msb < p + 1(b)
.dep
      ifgt @a1, ra10, 0   ;si (C*P)>N, premier test pour diminuer intervalle
        bnpa finmod                ;sortie de boucle si les 4 PEs inactif

      restore
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; début de la recherche binaire
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; premier passage I = (a + b)/2, dans ra3
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
       ifeq @a3, 0, 0 ;vérifié si a droite ou a gauche de la médiane
          ld 0, ra1     ; a
          ld ra3, rb1     ; b = rot
       else
          ld ra3, ra1     ; a = rot
```

```
        ld 10, rb1        ; b
      restore
   push 3
bin:
      add ra1, rb1, ra2 ;a + b
      sra r2, 1, r3    ;(a + b) / 2
      sra @2, ra3, @3   ; rotation sur les msb de C
      ifeq @a3, 0, 0 ;vérifié si a droite ou a gauche de la médiane
        ld ra3, rb1     ; b = rot
      else
        ld ra3, ra1     ; a = rot
      restore
          dbr bin

; dernier tour de boucle
   ifne @a3, 0, 0
        add 1, ra3, ra3
   restore

; la position du bit de poids fort est dans ra3
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ; début de la soustraction
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      sla @10, ra3, r5  ; augmente N par le multiplicateur trouve
      iflt ra5, @a1, 0
  ; vérification si N augmenter > C
        sub @a1, ra5, @a1 ; soustrait N augmente a C
      restore
      sra r5, 1, r5                ;divise N augmente par 2

subi:
   iflt 252, ra5, 0          ;
si N < N augmenter continuer
        bnpa finsub              ; N augmenter < N pour les 4 PEs donc toute sub
effectuer
      iflt ra5, @a1, 0 ;si (C*P)>N, premier test pour diminuer intervalle
        sub @a1, ra5, @a1

        restore          ;pop the activity stack
        restore
        sra r5, 1, r5     ;diminue
        bu subi          ;branchement inconditionnelle
```

```
subr:
        ifgt @a1, 253, 0 ;si (C*P)>N
         bnpa fins        ;sortie de boucle si les 4 PEs inactif
         sub @a1, 253, @a1
         restore
         bu subr          ;branchement inconditionnelle
fins:                     ;modulo calculer sur 4 PEs
      restore
  bu finmod

finsub:                    ;modulo calculer sur 4 PEs
      restore

finmod:                     ; N > C, pas besoin de calculer
      restore
  ret

      end
      .end
```

Équations aux différences finies (Poisson)

```
;**************************************************************/
;*
;*     PROGRAMME            : Continuum Model,  Stone chapitre 4
;*     date                 : mars 98
;*     crée par             : Jean-Jacques Clar
;*
;*       avec maille de 34x34, 1024 processeurs + les bordures
;*
;*
;*   Input : les données de base sont toutes stockées en mémoire
;*   convergence
;*
;**************************************************************/
; résultat OK C++

     .global continuum

     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ; initialisation des variables
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
         ;bound_N           .set 20 ; condition frontière nord
         ;delta             .set 1  ; condition de convergence
         ;bound_E           .set 40 ; condition frontière est
         ;bound_S           .set 30 ; condition frontière sud
     ;Nb_lines  .set 1
         ; nombre de lignes - 2
     ;Itérations  .set 25 ; nombre itération
     ;last              .set 4  ; the last one
     ;bf_last           .set 3  ; the one before the last
```

```
.data
      ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      ; chargement des valeurs
      ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      ; up: ajouter chiffre dans mem
      ;PE0
.init 0
.ma0 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26
.ma27 27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49
.ma50 50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72
.ma73 73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95
.ma96 96,97,98,99,100,101,102,103,104,105,106,107,108,109,110,111,112,113,114
.ma115 115,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130,131
.ma132 132,133,134,135,136,137,138,139,140,141,142,143,144,145,146,147,148
.ma149 149,150,151,152,153,154,155,156,157,158,159,160,161,162,163,164,165
.ma166 166,167,168,169,170,171,172,173,174,175,176,177,178,179,180,181,182
.ma183 183,184,185,186,187,188,189,190,191,192,193,194,195,196,197,198,199
.ma200 200,201,202,203,204,205,206,207,208,209,210,211,212,213,214,215,216
.ma217 217,218,219,220,221,222,223,224,225,226,227,228,229,230,231,232,233
.ma234 234,235,236,237,238,239,240,241,242,243,244,245,246,247,248,249,250
.ma251 251,252,253,254,255

      ;PE1
.init 1
.ma0 256,254,252,250,248,246,244,242,240,238,236,234,232,230,228,226,224,222
.ma18 220,218,216,214,212,210,208,206,204,202,200,198,196,194,192,190,188,186
.ma36 184,182,180,178,176,174,172,170,168,166,164,162,160,158,156,154,152,150
.ma54 148,146,144,142,140,138,136,134,132,130,128,126,124,122,120,118,116,114
.ma72 112,110,108,106,104,102,100,98,96,94,92,90,88,86,84,82,80,78,76,74,72
.ma93 70,68,66,64,62,60,58,56,54,52,50,48,46,44,42,40,38,36,34,32,30,28,26
.ma116 24,22,20,18,16,14,12,10,8,6,4,2,0,-2,-4,-6,-8,-10,-12,-14,-16,-18,-20
.ma139 -22,-24,-26,-28,-30,-32,-34,-36,-38,-40,-42,-44,-46,-48,-50,-52,-54
.ma156 -56,-58,-60,-62,-64,-66,-68,-70,-72,-74,-76,-78,-80,-82,-84,-86,-88
.ma173 -90,-92,-94,-96,-98,-100,-102,-104,-106,-108,-110,-112,-114,-116,-118
.ma188 -120,-122,-124,-126,-128,-130,-132,-134,-136,-138,-140,-142,-144,-146
.ma202 -148,-150,-152,-154,-156,-158,-160,-162,-164,-166,-168,-170,-172,-174
.ma216 -176,-178,-180,-182,-184,-186,-188,-190,-192,-194,-196,-198,-200,-202
.ma230 -204,-206,-208,-210,-212,-214,-216,-218,-220,-222,-224,-226,-228,-230
.ma244 -232,-234,-236,-238,-240,-242,-244,-246,-248,-250,-252,-254

      ;PE2
.init 2
.ma0 0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48
.ma25 50,52,54,56,58,60,62,64,66,68,70,72,74,76,78,80,82,84,86,88,90,92,94
```

```
.ma48 96,98,100,102,104,106,108,110,112,114,116,118,120,122,124,126,128,130
.ma66 132,134,136,138,140,142,144,146,148,150,152,154,156,158,160,162,164,166
.ma84 168,170,172,174,176,178,180,182,184,186,188,190,192,194,196,198,200,202
.ma102 204,206,208,210,212,214,216,218,220,222,224,226,228,230,232,234,236
.ma119 238,240,242,244,246,248,250,252,254,256,258,260,262,264,266,268,270
.ma136 272,274,276,278,280,282,284,286,288,290,292,294,296,298,300,302,304
.ma153 306,308,310,312,314,316,318,320,322,324,326,328,330,332,334,336,338
.ma170 340,342,344,346,348,350,352,354,356,358,360,362,364,366,368,370,372
.ma187 374,376,378,380,382,384,386,388,390,392,394,396,398,400,402,404,406
.ma204 408,410,412,414,416,418,420,422,424,426,428,430,432,434,436,438,440
.ma221 442,444,446,448,450,452,454,456,458,460,462,464,466,468,470,472,474
.ma238 476,478,480,482,484,486,488,490,492,494,496,498,500,502,504,506,508
.ma255 510

    ;PE3
.init 3
.ma0 512,508,504,500,496,492,488,484,480,476,472,468,464,460,456,452,448,444
.ma18 440,436,432,428,424,420,416,412,408,404,400,396,392,388,384,380,376,372
.ma36 368,364,360,356,352,348,344,340,336,332,328,324,320,316,312,308,304,300
.ma54 296,292,288,284,280,276,272,268,264,260,256,252,248,244,240,236,232,228
.ma72 224,220,216,212,208,204,200,196,192,188,184,180,176,172,168,164,160,156
.ma90 152,148,144,140,136,132,128,124,120,116,112,108,104,100,96,92,88,84,80
.ma109 76,72,68,64,60,56,52,48,44,40,36,32,28,24,20,16,12,8,4,0,-4,-8,-12,-16
.ma133 -20,-24,-28,-32,-36,-40,-44,-48,-52,-56,-60,-64,-68,-72,-76,-80,-84
.ma150 -88,-92,-96,-100,-104,-108,-112,-116,-120,-124,-128,-132,-136,-140,-144
.ma165 -148,-152,-156,-160,-164,-168,-172,-176,-180,-184,-188,-192,-196,-200
.ma179 -204,-208,-212,-216,-220,-224,-228,-232,-236,-240,-244,-248,-252,-256
.ma193 -260,-264,-268,-272,-276,-280,-284,-288,-292,-296,-300,-304,-308,-312
.ma207 -316,-320,-324,-328,-332,-336,-340,-344,-348,-352,-356,-360,-364,-368
.ma221 -372,-376,-380,-384,-388,-392,-396,-400,-404,-408,-412,-416,-420,-424
.ma235 -428,-432,-436,-440,-444,-448,-452,-456,-460,-464,-468,-472,-476,-480
.ma249 -484,-488,-492,-496,-500,-504,-508

    .init
    ;.ma0 bound_N
    ;.ma5 bound_S
    ;.mb0 40; bound_E
    .ra5 0
    .ra16 0
    ; up: ajouter  charges
.mb0 0,4,8,12,16,20,24,28,32,36,40,44,48,52,56,60,64,68,72,76,80,84,88,92,96
.mb25 100,104,108,112,116,120,124,128,132,136,140,144,148,152,156,160,164,168
.mb43 172,176,180,184,188,192,196,200,204,208,212,216,220,224,228,232,236,240
.mb61 244,248,252,256,260,264,268,272,276,280,284,288,292,296,300,304,308,312
```

```
.mb79 316,320,324,328,332,336,340,344,348,352,356,360,364,368,372,376,380,384
.mb97 388,392,396,400,404,408,412,416,420,424,428,432,436,440,444,448,452,456
.mb115 460,464,468,472,476,480,484,488,492,496,500,504,508,512,516,520,524
.mb132 528,532,536,540,544,548,552,556,560,564,568,572,576,580,584,588,592
.mb149 596,600,604,608,612,616,620,624,628,632,636,640,644,648,652,656,660
.mb166 664,668,672,676,680,684,688,692,696,700,704,708,712,716,720,724,728
.mb183 732,736,740,744,748,752,756,760,764,768,772,776,780,784,788,792,796
.mb200 800,804,808,812,816,820,824,828,832,836,840,844,848,852,856,860,864
.mb217 868,872,876,880,884,888,892,896,900,904,908,912,916,920,924,928,932
.mb234 936,940,944,948,952,956,960,964,968,972,976,980,984,988,992,996,1000
.mb251 1004,1008,1012,1016,1020

    .title "********** listing du fichier poisson.asm **********"

    .text

continuum:
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; configuration du chip
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ldcr 2, port1config  ; (92 3)  ; port 1 as input, synchroneous m
    ldcr 1, port3config  ; (92 3)  ; port 3 as output synchroneous m

    ldcr 1, port1incon  ; (A0 0)  ; port 1 as connect to north channel
    ldcr 1, port3outcon ; (A0 0)  ; port 3 as connect to north channel

    ldcr 3, bootcontrol  ; Use internal program memory and MCC for address 1

    ; up: ajouter nombre
       sub @a16, @b16, ra3 ; pour passer le premier if
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; initialisation des compteurs modulo externes mcc, mcd
    ldeamc 0, 0, mc_min, mc_min ;OK entre 100 ouest value
    ldeamc 10, 1000000, mc_max, mc_max;
    ldeamc 1, 0, mc_start, mc_start;
    ldeamc 1, 1, mc_stride, mc_stride;

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ;sla r3, 2, acc ;|| io *mccr%
    ld 0, acc

    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
    ; initialisation des compteurs modulo interne mca, mcb
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; up: ajuster compter modulo
    ldiamc 0, 0, 255, mcar;
    ldiamc 255, 0, 255, mcaw;
    ldiamc 0, 0, 255, mcbr;


;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;          tableau de processeurs
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.nodep
    stc @b10, sport
    stc *mcar, nport        ; local value to north port
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;  avec 4 x q -> acc
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; résultat : u2 = (4q - 4u1 + N + W + E + S)/4 +4u1
; additionne -4u1 et 4q pour avoir le delta lors de la division par 4
; Est value : sport
; Ouest value : nport
push 1
Iter:
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; traitement de la première ligne
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    inc ra16, ra16
    ld 0, ra17
    sra acc,2,r1                ; division par quatre du résultat
    io *mcdr%
    mult *mcbr(1), 4, acc||n2ssr     ; charge dans accumulateur
    io *mcdr%
    macc *mcar(1), -4            ; soustraction value-1
    add nport, 20, acc+        ; add in north and west value, bound_N
    abs ra1, rb2 || n2ssr         ;add, n2ssr ,add indissociable
                                  ; abs de la différence entre nouvelle et vielle value
    io *mcdr%
    n2ssr
    io *mcdr%
    add sport, *mcar, acc+;||n2ssr  ; add in east and south value
    stc *mcar(-2), nport              ; local value to north port
    add ra1,*mcar(2), *mcaw(1)        ; chargement nouvelle valeur dans a
    cas rb2, ra3, r3||stc @b10, sport ; +> change dans ra3
  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
; traitement de la deuxième ligne
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    sra acc,2,r1
    io *mcdr%
    mult *mcbr(1), 4, acc||n2ssr
    io *mcdr%
    macc *mcar(-1), -4
    add nport, *mcar(2), acc+

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.dep   ; test de convergence
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ifgt ra3, 1, 0       ;delta
    bnpa dernier
    restore
    #2 nop

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; test de convergence
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ;n2ssr
    .nodep  ;n2ssr
    ld 0, ra3
    abs ra1, ra3 || n2ssr
    io *mcdr%
    n2ssr
    io *mcdr%
    add sport, *mcar(-2), acc+;||n2ssr
    stc *mcar, nport
    add ra1,*mcar(2), *mcaw(1);||stc @b0, sport
    stc *mcar, nport
    stc @b10, sport

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; traitement autres lignes
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; up: ajuster le push
    push 253 ;Nb_lines; nombre de lignes a traiter - 3
Lignes:
    cas rb2, ra3, r3||stc @b0, sport
    inc ra17, ra17
    sra acc,2,r1
    io *mcdr%
    mult *mcbr(1), 4, acc||n2ssr
```

```
        io *mcdr%
        macc *mcar(-1), -4
        madd *mcar(-1), -4, acc, acc            ; soustraction valeur-1
        add nport, *mcar(2), acc+
        abs ra1, rb2 || n2ssr
        io *mcdr%
        add sport, *mcar(-2), acc+||n2ssr
        io *mcdr%
        add ra1,*mcar(2), *mcaw(1);||
        stc *mcar, nport
        cas rb2, ra3, r3||stc @b10, sport
        dbr Lignes

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; traitement de la dernière ligne
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        cas rb2, ra3, r3||stc @b0, sport
        sra acc,2,r1
        io *mcdr%
        mult *mcbr(1), 4, acc||n2ssr
        io *mcdr%
        madd *mcar(-1), -4, acc, acc            ; soustraction value-1
        macc *mcar(-1), -4
        add nport, *mcar, acc+
        abs ra1, rb2 || n2ssr
        io *mcdr%
        add sport, 30, acc+||n2ssr
; bound_S
        io *mcdr%
        add ra1,*mcar(2), *mcaw(1);||
        stc *mcar, nport
        cas rb2, ra3, r3||stc @b10, sport
        dbr Iter
        bu fin

 ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 ; résultat du dernier traitement
 ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
dernier:
 restore
fin:
 ld 0, nport
 #3 nop
 io *mcdr%
```

```
  io *mcdr%
  io *mcdr%
  ld ra3, nport
  #3 nop
      io *mcdr%
      #3 nsr || io *mcdr%
  ld 0, nport
  #3 nop
  io *mcdr%
  io *mcdr%

  ldiamc 151, 151, 166, mcar;

     push 16
result:
      stc *mcar(1), nport
      io *mcdr%
      nsr||io *mcdr%
      nsr||io *mcdr%
      nsr||io *mcdr%
      dbr result

      ld ra16, nport
      io *mcdr%

;fin du traitement convergence
.dep
      end
      .end
```

Morphologie binaire

Érosion

```
;*************************************************************/
;*  version générale
;*
;*      PROGRAMME        : shrink.asm, morphologie mathématique
;*      date             : mars 98
;*      crée par         : Jean-Jacques Clar
;*
;*************************************************************/
;* 3x3 template
;*
;* Pixel E/S répartition
;* Port d'entrée Sud connecte au port de donnée 1
;* Port d'entrée Nord connecte au port de donnée 3
;* Port d'adresse 1 associé au port de donnée 1 (Externe)
;* Port d'adresse 2 associé au port de donnée 3 (Externe)
;*
;* I. Kraljic  - 26 mai 1997
;*
;* image jusqu'à 512 x 512, 14 mai 98
;*
;*************************************************************/

    .global shrink

    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; initialisation des variables
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Nb_line         .set X      ; nombre de lignes image
Nb_pix          .set X      ; nombre pixels par ligne
```

```
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 pix_q          .set Nb_pix / 4
 mem            .set (Nb_pix * Nb_line) - 1
 mci_max                .set Nb_pix/2 - 1

    .data
    .init
    .mb0 0,0,0,0,0,0,0,0,0,0
    .ma0 0,0,0,0,0,0,0,0,0,0
        .rb0 0

    .text
shrink:
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; configuration du chip
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ldcr 2, port1config  ; (90 2)  ; port 1 as input, synchronous mode
    ldcr 1, port3config  ; (92 3)  ; port 3 as output
    ldcr 1, port1incon   ; (A0 1)  ; port 1 connect to north channel(input)
    ldcr 2, port3outcon  ; (B2 1)  ; port 3 for south output (synch.)

    ldcr 3, bootcontrol; address port1 for modulo counter

    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; initialisation des compteurs modulo externes mcc, mcd
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ldeamc 0,0, mc_min,mc_min
    ldeamc mem+Nb_line, mem, mc_max,mc_max;
    ldeamc 0,mem-Nb_pix-4, mc_start,mc_start;
    ldeamc 1,1,mc_stride,mc_stride;

 ld 0, nport
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; initialisation des compteurs modulo interne mca, mcb
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ldiamc 0,0,mci_max, mcar
    ldiamc 0,0,mci_max, mcaw
    ldiamc 0,0,mci_max, mcbr
    ldiamc 0,0,mci_max, mcbw

    nsr || io *mccr%
    nsr || io *mccr%

.nodep
```

```
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; préparation 1 ième ligne, première fenêtre
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; mcar=0, mcaw=0,
fwd nport, *mcaw          ;pixel 0 de la fenêtre
nsr || io *mccr%
and nport, *mcar, *mcaw || nsr || io *mccr%     ;and pixel 1 et 0
fwd nport, @bmci_max
#2 nsr || io *mccr%
and @bmci_max, *mcar, *mcaw(1)                  ;and pixel 2 et (0,1)
; mcbr=0, mcbw=0,
; mcar=0, mcaw=1,

push Nb_line/2 ; (nombre de lignes/2)
image:
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; lignes impaires
; seconde fenêtre et suivante pour couvrir les lignes
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
push pix_q-1 ;quart des pixels -1
linedu:
fwd nport, *mcaw(pix_q-1)       ;pixel 0 de la fenêtre courante
io *mcdr%;
;ET ligne précédente, ligne courante(fenêtre précédente)
;stocker dans la partie inférieure de la mémoire
and *mcar(1), *mcbr(-pix_q), *mcaw(1-pix_q) || nsr || ssr || io *mccr%, *mcdr%
;ET pixel 1 et 0
and nport, *mcar(-1), *mcaw || nsr || ssr || io *mccr%, *mcdr%
fwd nport, @b0                  ;pixel 2 de la fenêtre courante
nsr || ssr || io *mccr%, *mcdr%
;ET deux lignes précédente(memb), ligne courante(fenêtre précédente)
and *mcar(1), *mcbr(1-pix_q), sport || nsr || io *mccr%
and *mcar, @b0, *mcaw(1)        ;ET pixel 2 et (0,1)
; mcar = mcar + 1
; mcaw = mcaw + 1
; mcbr = mcbr + 1
; mcbw = mcbw
dbr linedu

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; re-positionnement compteur modulo
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ldiamc 0,0,mci_max, mcbw
```

```
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; préparation lignes paires, première fenêtre
    ; + fin lignes impaires
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    nsr || io *mccr%, *mcdr%
    fwd nport, *mcbw
    ;ET ligne précédente, ligne courante(dernière fenêtre de la ligne)
    ;stocker dans ra0, pour laisser libre la dernière adresse mémoire(512 x 512)
    and *mcbr(1-pix_q), *mcar, ra0 || nsr || ssr || io *mccr%, *mcdr%
    and nport, *mcbr, *mcbw || nsr || ssr || io *mccr%, *mcdr%
    fwd nport, @amci_max
    nsr || ssr || io *mccr%, *mcdr%
  ;ET deux lignes précédente(rb0), ligne courante(dernière fenêtre)
    and rb0, *mcar(1-pix_q), sport || nsr || io *mccr%
    and *mcbr, @amci_max, *mcbw(1)

   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   ; lignes paires
   ; seconde fenêtre et suivante pour couvrir les lignes
   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    push pix_q - 1
linetr:
    fwd nport, *mcbw(pix_q-1)
    io *mcdr%
    and *mcbr(1), *mcar(-pix_q), *mcbw(1-pix_q) || nsr || ssr || io *mccr%, *mcdr%
    and nport, *mcbr(-1), *mcbw || nsr || ssr || io *mccr%, *mcdr%
    fwd nport, @a0
    nsr || ssr || io *mccr%, *mcdr%
    and *mcbr(1), *mcar(1-pix_q), sport || nsr || io *mccr%
    and *mcbr, @a0, *mcbw(1)
    ; mcar = mcar + 1
    ; mcaw = mcaw
    ; mcbr = mcbr + 1
    ; mcbw = mcbw + 1
dbr linetr

    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; re-positionnement compteur modulo
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ldiamc 0,0,mci_max, mcaw

    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; restant ligne précédente + début ligne impaire suivante
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
        nsr || io *mccr%, *mcdr%
        fwd nport, *mcaw
        and *mcbr, *mcar(1-pix_q), rb0 || nsr || ssr || io *mccr%, *mcdr%
        and nport, *mcar, *mcaw || nsr || ssr || io *mccr%, *mcdr%
        fwd nport, @bmci_max
        nsr || ssr || io *mccr%, *mcdr%
        and ra0, *mcbr(1-pix_q), sport || nsr || io *mccr%
        and *mcar, @bmci_max, *mcaw(1)
        ; mcbr=0, mcbw=0,
        ; mcar=0, mcaw=1,
dbr image

        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; reste 4 caractères de l'avant dernière ligne a sortir, déjà sur le canal sud
        ; + une ligne de 0 pour la dernière ligne
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        io *mcdr%
  and 0, @b1, @b1 || ssr || io *mcdr%
  ssr || io *mcdr%
  ssr || io *mcdr%
  stc @b1, sport

        push pix_q
last:
        #4 io *mcdr%
dbr last

fin:
end
        .end
```

Dilatation

```
;************************************************************/
;*  version générale
;*
;*      PROGRAMME           : expand.asm, morphologie mathématique
;*      date                : mars 98
;*      crée par            : Jean-Jacques Clar
;*
;* 3x3 template
;*
;* Pixel E/S répartition
;* Port d'entrée Sud connecte au port de donnée 1
;* Port d'entrée Nord connecte au port de donnée 3
;* Port d'adresse 1 associé au port de donnée 1 (Externe)
;* Port d'adresse 2 associé au port de donnée 3 (Externe)
;*
;* I. Kraljic  - 26 mai 1997
;*
;* image jusqu'à 128 x 128, 14 mai 98
;*
;************************************************************/

    .global expand

    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; initialisation des variables
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 Nb_line            .set 128        ; nombre de lignes image
 Nb_pix             .set 128        ; nombre pixels par ligne
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 pix_q        .set Nb_pix / 4
 mem          .set (Nb_pix * Nb_line) - 1
 mci_max            .set Nb_pix/2 - 1

    .data
    .init
        .rb0 0

    .text
expand:
```

```
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; configuration du chip
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ldcr 2, port1config   ; (90 2)  ; port 1 as input, synchronous mode
ldcr 1, port3config   ; (92 3)  ; port 3 as output
ldcr 1, port1incon    ; (A0 1)  ; port 1 connect to north channel(input)
ldcr 2, port3outcon   ; (B2 1)  ; port 3 for south output (synch.)

ldcr 3, bootcontrol; address port1 for modulo counter

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; initialisation des compteurs modulo externes mcc, mcd
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ldeamc 0,0, mc_min,mc_min
ldeamc mem+Nb_line, mem, mc_max,mc_max;
ldeamc 0,mem-Nb_pix-4, mc_start,mc_start;
ldeamc 1,1,mc_stride,mc_stride;

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; initialisation des compteurs modulo interne mca, mcb
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ldiamc 0,0,mci_max, mcar
ldiamc 0,0,mci_max, mcaw
ldiamc 0,0,mci_max, mcbr
ldiamc 0,0,mci_max, mcbw

nsr || io *mccr%
nsr || io *mccr%

.nodep
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; préparation 1ième ligne
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; mcar=0, mcaw=0,
fwd nport, *mcaw
nsr || io *mccr%
or nport, *mcar, *mcaw || nsr || io *mccr%
fwd nport, @bmci_max
nsr || io *mccr%
or @bmci_max, *mcar, *mcaw(1) || nsr || io *mccr%
; mcbr=0, mcbw=0,
; mcar=0, mcaw=1,

push Nb_line/2 ; (nombre de lignes/2)
```

```
image:
   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   ; lignes impaires
   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      push pix_q-1 ;quart des pixels -1
linedu:
      fwd nport, *mcaw(pix_q-1)
      io *mcdr%;
      or *mcar(1), *mcbr(-pix_q), *mcaw(1-pix_q) || nsr || ssr || io *mccr%, *mcdr%
      or nport, *mcar(-1), *mcaw || nsr || ssr || io *mccr%, *mcdr%
      fwd nport, @b0
      nsr || ssr || io *mccr%, *mcdr%
      or *mcar(1), *mcbr(1-pix_q), sport || nsr || io *mccr%
      or *mcar, @b0, *mcaw(1)
      ; mcar = mcar + 1
      ; mcaw = mcaw + 1
      ; mcbr = mcbr + 1
      ; mcbw = mcbw
dbr linedu

      ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      ; re-positionnement compteur modulo
      ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      ldiamc 0,0,mci_max, mcbw

      ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      ; préparation lignes paires + fin lignes impaires
      ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      nsr || io *mccr%, *mcdr%
      fwd nport, *mcbw
      or *mcbr(1-pix_q), *mcar, ra0 || nsr || ssr || io *mccr%, *mcdr%
      or nport, *mcbr, *mcbw || nsr || ssr || io *mccr%, *mcdr%
      fwd nport, @amci_max
      nsr || ssr || io *mccr%, *mcdr%
      or rb0, *mcar(1-pix_q), sport || nsr || io *mccr%
      or *mcbr, @amci_max, *mcbw(1)

   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   ; lignes paires
   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      push pix_q - 1
linetr:
      fwd nport, *mcbw(pix_q-1)
      io *mcdr%
```

```
        or *mcbr(1), *mcar(-pix_q), *mcbw(1-pix_q) || nsr || ssr || io *mccr%, *mcdr%
        or nport, *mcbr(-1), *mcbw || nsr || ssr || io *mccr%, *mcdr%
        fwd nport, @a0
        nsr || ssr || io *mccr%, *mcdr%
        or *mcbr(1), *mcar(1-pix_q), sport || nsr || io *mccr%
        or *mcbr, @a0, *mcbw(1)
        ; mcar = mcar + 1
        ; mcaw = mcaw
        ; mcbr = mcbr + 1
        ; mcbw = mcbw + 1
dbr linetr

        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; re-positionnement compteur modulo
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ldiamc 0,0,mci_max, mcaw

        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; restant ligne précédente + début ligne impaire
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        nsr || io *mccr%, *mcdr%
        fwd nport, *mcaw
        or *mcbr, *mcar(1-pix_q), rb0 || nsr || ssr || io *mccr%, *mcdr%
        or nport, *mcar, *mcaw || nsr || ssr || io *mccr%, *mcdr%
        fwd nport, @bmci_max
        nsr || ssr || io *mccr%, *mcdr%
        or ra0, *mcbr(1-pix_q), sport || nsr || io *mccr%
        or *mcar, @bmci_max, *mcaw(1)
        ; mcbr=0, mcbw=0,
        ; mcar=0, mcaw=1,
dbr image

        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; re-positionnement compteur modulo sortie or ligne 31 et 32, mb4-7
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ldiamc pix_q,0,mci_max, mcbr
        ld rb0, @bmci_max

        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; dernière ligne
        ; reste 4 caractères de l'avant dernière ligne a sortir, déjà sur le canal sud
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        push 1+pix_q
last:
```

```
        io *mcdr%
  ssr || io *mcdr%
  ssr || io *mcdr%
  ssr || io *mcdr%
        stc *mcbr(1), sport
dbr last

fin:
end
        .end
```

IDEA

```
***********************************************************/
;*     PROGRAMME  : idea.asm
;*     date       : janvier 98
;*     crée par   : Jean-Jacques Clar
;*
;*        Utilisation de 2 mémoire externe :  Entrée : A
;*                                            Sortie: C
;*   Entrée : port 1 au cannal Nord (plaintext)
;*   Sortie: canal Sud au port 4 (ciphertext)
;*
;*   description: algorithme IDEA, utilise dans PGP
;*
;***********************************************************/
.data
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ; définition des variables
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ;var   def                    mémoire
     ;C  : texte crypté             @a1
     ;P  : plaintext               nport(16 bits), @b0
     ;p  : exposant de P             @a12
     ;N  : partie de la clé         @a11
     ;n  : exposant de N             @a13
     ;a  : borne inférieure exposant ra1
     ;b  : borne supérieure exposant rb1

     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ; chargement des mémoires des PEs
     ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     .init

     .title "********** listing du fichier idea.asm **********"
```

```
	.text

idea:
	;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
	; configuration du chips
	;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
	ldcr 2, port1config  ; (90 2)  ; port 1 as input, synchroneous mode
	ldcr 0, port2config  ; (91 0)  ; port 2 as input
	ldcr 1, port3config  ; (92 3)  ; port 3 as output
	ldcr 3, port4config  ; (92 3)  ; port 4 as output, synchroneous mode

	; configuration de la connexion des ports d'E/S
	ldcr 1, port1incon   ; (A0 0)  ; port 1 as connect to north channel
	ldcr 2, port2incon   ; (A1 1)  ; port 1 as connect to south channel
	ldcr 1, port3outcon  ; (B2 0)  ; port 3 as connect to north channel
	ldcr 2, port4outcon  ; (B3 1)  ; port 4 as connect to south channel


	ldcr 3, bootcontrol  ; Use internal program memory and MCC for address 1

	;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
	; initialisation des compteurs modulo externes mcc, mcd
	;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
	ldeamc 0, 0, mc_min, mc_min
	    ldeamc 4, 250, mc_max, mc_max
	    ldeamc 0, 0, mc_start, mc_start
	    ldeamc 1, 1, mc_stride, mc_stride

	;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
	; initialisation des compteurs modulo interne mca, mcb
	;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
	ldiamc 0, 0, 51, mcaw;
	ldiamc 0, 0, 51, mcar;
	ldiamc 0, 0, 7, mcbw;
	ldiamc 0, 0, 7, mcbr;

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;		début du programme
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
	;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
	; clé de 128 bit, ma-b0,ma-b3
	;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.nodep
	push 4
```

```
clé:
        #3 nsr || ssr
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; le a ou le b en premier ?
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        fwd nport, *mcaw(1);
        fwd sport, *mcaw(1);
        fwd sport, *mcbw(2) || io *mccr%;
        nsr || ssr
        dbr clé
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; 8 première clés de 16 bits,  ma0-7
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; vérification des clés
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;       srl @0, 0, @0       ; pour clé 9 et 10
;       srl @1, 0, @1       ; pour clé 11 et 12
;       srl @2, 0, @2       ; pour clé 13 et 14
;       srl @3, 0, @3       ; pour clé 15 et 16
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; calcul des clés secondaires
        ; décalage de 25 bits vers la gauche
        ; obtenir 52 clés
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; clé de 9-16
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; décalage de 7 bits, droite, 25 bits de poids faible positionnes
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        srl @0, 7, @10 ; pour clé 11 et 12
        srl @2, 7, @12 ; pour clé 13 et 14
        srl @4, 7, @14 ; pour clé 15 et 16
        srl @6, 7, @8 ; pour clé 9 et 10

        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; décalage de 25 bits, gauche, 7 bits de poids fort positionnes
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        sll @0, 25, r0        ; pour clé 9 et 10
        sll @2, 25, r1  ; pour clé 11 et 12
        sll @4, 25, r2  ; pour clé 13 et 14
        sll @6, 25, r3  ; pour clé 15 et 16
```

```
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; un or des 7 msb(rb#) et des 9 lsb(9msb des 25 bits de poids faible, @b#)
; et déplacement des clés pair vers la mémoire A
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; doit être fait a cause des sll, srl des clés suivantes
or rb0, @b8, @b8
or rb1, @b10, @b10
or rb2, @b12, @b12
or rb3, @b14, @b14

; et déplacement des clés pair vers la mémoire A
; le or ne corrige pas les valeurs négatives de b ?
srl @8, 16, @9
srl @10, 16, @11
srl @12, 16, @13
srl @14, 16, @15

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; OK, valeur valide le 17/02/98
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; clé de 17-24 + vérification clés 9-16
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
srl @8, 7, @18 ; pour clé 19 et 20
srl @10, 7, @20  ; pour clé 21 et 22
srl @12, 7, @22  ; pour clé 23 et 24
srl @14, 7, @16 ; pour clé 17 et 18

sll @8, 25, r0
sll @10, 25, r1
sll @12, 25, r2
sll @14, 25, r3

or rb0, @b16, @b16
or rb1, @b18, @b18
or rb2, @b20, @b20
or rb3, @b22, @b22

srl @16, 16, @17
srl @18, 16, @19
srl @20, 16, @21
srl @22, 16, @23

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
; clés de 25-32 + vérification des clés
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
srl @16, 7, @26  ; pour clé 27 et 28
srl @18, 7, @28  ; pour clé 29 et 30
srl @20, 7, @30 ; pour clé 31 et 32
srl @22, 7, @24 ; pour clé 25 et 26

sll @16, 25, r0
sll @18, 25, r1
sll @20, 25, r2
sll @22, 25, r3

or rb0, @b24, @b24
or rb1, @b26, @b26
or rb2, @b28, @b28
or rb3, @b30, @b30

srl @24, 16, @25
srl @26, 16, @27
srl @28, 16, @29
srl @30, 16, @31
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; clé de 33-40
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
srl @24, 7, @34 ; pour clé 35 et 36
srl @26, 7, @36 ; pour clé 37 et 38
srl @28, 7, @38 ; pour clé 39 et 40
srl @30, 7, @32 ; pour clé 33 et 34

sll @24, 25, r0
sll @26, 25, r1
sll @28, 25, r2
sll @30, 25, r3

or rb0, @b32, @b32
or rb1, @b34, @b34
or rb2, @b36, @b36
or rb3, @b38, @b38

; et déplacement des clés pair vers la mémoire A
srl @32, 16, @33
srl @34, 16, @35
srl @36, 16, @37
srl @38, 16, @39
```

```
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; clé de 41-48
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
srl @32, 7, @42 ; pour clé 43 et 44
srl @34, 7, @44 ; pour clé 45 et 46
srl @36, 7, @46 ; pour clé 47 et 48
srl @38, 7, @40 ; pour clé 41 et 42

sll @32, 25, r0
sll @34, 25, r1
sll @36, 25, r2
sll @38, 25, r3

or rb0, @b40, @b40
or rb1, @b42, @b42
or rb2, @b44, @b44
or rb3, @b46, @b46


; et déplacement des clés pair vers la mémoire A
srl @40, 16, @41
srl @42, 16, @43
srl @44, 16, @45
srl @46, 16, @47

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; clé de 49-52
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
srl @40, 7, @50 ; pour clé 43 et 44
srl @46, 7, @48 ; pour clé 41 et 42

sll @40, 25, r0
sll @42, 25, r1
nop
nop
or rb0, @b48, @b48
or rb1, @b50, @b50
nop
nop
; et déplacement des clés pair vers la mémoire A
srl @48, 16, @49
srl @50, 16, @51
```

```
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; vérification
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ldiamc 0, 0, 51, mcaw;
        ldiamc 0, 0, 51, mcar;
; ld 0, ra4
;       push 52
valid:
;       inc ra4, ra4
;       add *mcar(1), 0, ra0

;           dbr valid

.dep
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; entrer des quatre premier blocs de 64 bits de texte
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        push 2; 2 * 32 bits
texte:
        #3 nsr || ssr || io *mccr%;
        fwd sport, *mcbw(1);mb0-2
        fwd nport, *mcbw(1);mb1-3
        nsr || ssr || io *mccr%;
        dbr texte
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; texte dans mb0-3
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        push 1; nombre de caractères a crypter / nb PEs
Cipher:
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ; compute the ciphertext C of plaintext P
        ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


  push 8; faire de 1 a 8
ronde:
        ;e1 = x1*z1
        mult *mcbr(1), *mcar(1), r1
  ifgt 0, rb1,0
    and 0, rb1, rb1
  restore
        ;ifgt r1, modulo, 0
        ;    bnpa pas
```

```
    ;   call modulo
pas:
 ;restore
    ;e2 = x2+z2
    add *mcbr(1), *mcar(1), rb2
    ;e3 = x3+z3
    add *mcbr(1), *mcar(1), rb3
    ;e4 = x4*z4
    mult *mcbr(1), *mcar(1), r4
    ;e1 validation du résultat, multiplication modulo 2^16 + 1 (65537)
 ifgt 0, rb4,0
  and 0, rb4, rb4
 restore

    ;e5 = e1 XOR e3
    xor ra1, rb3, ra5
    ;e6 = e2 XOR e4
    xor rb2, ra4, rb6
    ;e7 = e2*z5
    mult rb2, *mcar(1), r7
 ifgt 0, rb7,0
  and 0, rb7, rb7
 restore
    ;e8 = e6+e7
    add rb6, ra7, ra8
    ;e9 = e8*z6
    mult ra8, *mcar(1), @60
 iflt 32768, @b60,0
  not @b60, @b60
 restore
    ;e10 = e7+e9
    add ra7, @a60, @a61
    ;e11 = e1 XOR e9
    xor ra1, @a60, *mcbw(1)
    ;e12 = e3 XOR e9
    xor rb3, @a60, *mcbw(1)
    ;e13 = e2 XOR e10
    xor rb2, @a61, *mcbw(1)
    ;e14 = e4 XOR e10
    xor ra4, @a61, *mcbw(1)

    dbr ronde
    ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ; après les huit rondes
```

```
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;res1 = x1*z1
mult *mcbr(1), @a0, nsport
;res2 = x2+z2
add *mcbr(1), @a1, sport
;res3 = x3+z3
add *mcbr(1), @a2, @a200
;res4 = x4*z4
mult *mcbr(1), @a3, @201

;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; sortie du texte crypté et entrée du prochain texte
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

io *mcdr%
#3 nsr || ssr || io *mccr%, *mcdr%;
fwd sport, *mcbw(1);mb0-2
fwd nport, *mcbw(1) || stc @a201, sport;mb1-3
stc @a200, nport;
io *mcdr%
#3 nsr || ssr || io *mccr%, *mcdr%;
fwd sport, *mcbw(1);mb0-2
fwd nport, *mcbw(1);mb1-3

dbr Cipher

fin:
    end
    .end
```

ANNEXE B

OPÉRATION MODULO RAPIDE EN L'ABSENCE DE DIVISEUR MATÉRIEL

Opération modulo rapide en l'absence de diviseur matériel

Normand Bélanger

19 Octobre 1997

Résumé

On veut trouver une méthode pour calculer rapidement le reste de la division entre deux nombres entiers pour supporter l'algorithme d'encryptage RSA. On cherche un algorithme qui minimise, à l'implantation, le nombre d'instructions de branchement parce que ces instructions sont très couteuses (en temps) sur **Pulse**.

1 Introduction

L'équation ciblée est $C = (C \times P)\%N$ où P et N sont des constantes et sont premiers entre eux. La méthode proposée consiste à trouver le premier bit de poids fort de $C \times P$ qui a la valeur 1 et à multiplier N par la puissance de 2 la plus grande possible qui donne un nombre dont le premier bit de poids fort de valeur 1 soit situé au bit qui est à la position immédiatement à la droite de celui de $C \times P$. On obtient alors un nombre qui est certainement plus petit que $C \times P$ mais qui requiert de lui soustraire N au maximum deux fois pour obtenir le reste de la division. On doit faire deux soustraction lorsque $C \times P$ est un peu plus grand que $2 \times N$. Pour ce qui est de la multiplication, comme elle implique un nombre qui est une puissance de 2, elle est, en fait, un décalage vers la gauche de $C \times P$ (ce qui est une opération très rapide sur **Pulse**).

2 Méthode

Sachant que N est une constante, considérons que $2^n \leq N < 2^{n+1}$, donc le premier bit de poids fort de N qui a la valeur 1 est le bit à la position n (la position 0 étant celle du bit de poids faible). De façon similaire, on défini p de sorte que $2^p \leq P < 2^{p+1}$.

Dans ce contexte, le bit premier bit de poids fort de $C \times P$ doit être situé à une position c donnée par l'inéquation $p + n \leq c \leq p$ parce que le premier bit de poids fort de C qui est à 1 doit être situé dans l'intervalle $[0, n]$ à cause de l'opérateur %.

La méthode la plus rapide (et qui implique un minimum de décision) pour trouver le premier bit de poids fort qui est à 1 consiste à effectuer une recherche binaire dans l'intervalle $[p, p + n]$. Cette recherche consiste à décaler $C \times P$ vers la droite de $\lfloor \frac{p+(p+n)}{2} \rfloor$

positions et de tester si le nombre résultant est 0; si c'est le cas, alors le premier bit de poids fort qui soit à 1 est situé dans l'intervalle $[p, \lfloor \frac{p+(p+n)}{2} \rfloor]$ si non, il se trouve dans l'intervalle $]\lfloor \frac{p+(p+n)}{2} \rfloor, p+n]$. On divise ensuite celui de ces intervalles qui contient le bit cherché en deux et on recommence jusqu'à ce que l'intervalle où se trouve le bit en question ait une taille de 1 (i.e. il ne contient que le bit cherché). Le nombre maximum d'opérations de division de l'intervalle qui devront être effectuées est de $\lceil \log_2 n \rceil$.

3 Remarques

Puisque N et P sont des constantes, on connait n et p au moment d'écrire le programme, donc le nombre d'étapes à parcourir et les intervalles impliqués sont connus d'avance ce qui implique que l'algorithme peut être construit sous forme de structures de contrôle conditionnelles ("IF ... THEN ... ELSE") plutôt que sous forme de boucle ("WHILE").

Dans ce cadre, il faut faire attention à un cas particulier: lorsque l'intervalle ne contient que deux nombres. Un algorithme mal conçu pourrait "manquer" un des deux éléments et produire un résultat incorrect.

Ceci est une page de garde

CHAPITRE 5

CONCLUSION

Comme présenté au chapitre 1, ce projet avait deux objectifs principaux :

1. Le développement d'applications diverses et évaluation de leurs performances sur **PULSE**.
2. Le test et évaluation des outils en place, et apporter si possible des suggestions pour améliorer le produit.

5.1. DÉVELOPEMENT D'APPLICATIONS

Durant ce projet huit applications différentes ont été étudiées. De ces huit, une a été abandonnée avant d'avoir atteint le stade de l'écriture de code, il s'agit de la transformée de Hough. Cette transformée est un algorithme qui s'applique sur une image et qui à prime abord, comportait des éléments intéressants pour **PULSE** : un traitement intensif, de multiples opérations à effectuer sur les mêmes pixels et la possibilité de réutilisé des algorithmes déjà écrits sur **PULSE** :

- Calcul du niveau de gris (Sobel),
- Détection de contours (threshold),
- Calcul du gradient (direction),
- Calcul de la position dans un espace spécial nommé espace de Hough,
- Production d'un histogramme,

Deux facteurs ont mené à l'abandon de cet algorithme, soit le calcul de la position dans l'espace de Hough et le calcul du gradient qui demandait l'emploi d'une division.

Des sept applications restantes, six ont permis d'effectuer de multiples simulations et itérations sur les algorithmes.

La septième, l'algorithme de chiffrage IDEA a été développé en entier. La non-disponibilité d'une instruction conditionnelle sur 32 bits afin d'effectuer la multiplication modulo $2^{16} + 1$ de façon rapide a rendu inutile toute optimisation.

Les autres applications ; multiplication vecteurs – matrice, algorithme de Bresenham, algorithme RSA, Équation aux différences finies, Morphologie binaire avec l'Érosion et la

Dilatation d'images, présentent une variété d'applications numériques. Une des requêtes du groupe **PULSE** était d'évaluer avec quelle facilité, ou difficulté, différents types d'algorithmes pouvaient être portés sur **PULSE** et d'essayer de sortir des traits caractéristiques qui fonctionnaient bien sur **PULSE**.

5.1.1. Développement sur PULSE

Basé sur les résultats des comparables vus au chapitre précédent, les chiffres obtenus lors de ce projet se comparent avantageusement à ce qui se fait à l'extérieur. On note cependant une lacune évidente. En effet, le travail de cette recherche s'est étendu sur environ 16 mois, incluant une session passée à temps plein sur l'avancement des travaux de recherches. En mettant en relation le temps investi dans ce projet et les quelques centaines de lignes de code écrites, il y a un déséquilibre important entre les deux.

Ce déséquilibre s'explique par plusieurs facteurs qui font partie des difficultés rencontrées durant ce projet.

1- **Développement d'applications performantes :** il s'agit du critère majeur de développement qui a dirigé les actions et les changements effectués durant la phase de développement. Les multiples itérations et simulations n'avaient qu'un seul but : traiter plus de données en un même laps de temps. La recherche de vitesse pour chacune des applications se joue au niveau des instructions parallèles et par le retrait des instructions vides (*NOP*). Cette recherche nécessite souvent un réarrangement de la façon donc se déroule le programme, de multiples changements mineurs et des tests de performances entre chaque changement.

2- **Diversité des applications :** Au lieu d'exploiter un filon d'application similaire et de développer mes connaissances dans ce domaine spécifique, j'ai appliqué les directives du groupe en touchant à différents domaines.

3- **Courbe d'apprentissage :** Hormis une introduction au logiciel PVM sur UNIX, à l'occasion d'un cours sur les réseau d'ordinateurs, **PULSE** aura été une première expérience de programmation sur un système parallèle. Le fait de prendre plusieurs semaines en début de projet pour étudier les grands principes du parallélisme et les quatre compétiteurs de PULSE aura eu un effet d'ouverture sur la suite de ce projet. La plus grande difficulté d'apprentissage s'est trouvée dans l'environnement de développement de **PULSE**. Une architecture et un langage de programmation qui rivalisent de complexité l'un avec l'autre. Il est certain que le fait de se retrouver à 500 kilomètres des autres membres du groupe a contribué aux difficultés d'apprentissage durant le déroulement de ce projet. Je crois pouvoir affirmer avec expérience que l'ère des communications améliore grandement la rapidité et les possibilités de développement en différent lieu physique, une fois la courbe d'apprentissage gravie.

4- **Produit non totalement stable :** une partie du projet était de pouvoir améliorer l'environnement de développement, l'architecture et de pouvoir définir les limitations courantes de **PULSE**. Le déroulement du projet s'est donc effectué sur un produit en développement. Ainsi, plusieurs problèmes de conception ont été mis à jour, tant dans le circuit que dans ses outils de développement. C'est un privilège

de pouvoir participer à la validation d'un produit mais cela comporte aussi des contraintes dans le développement et la validation d'applications.

5- **Documentation :** la documentation présente possédait des erreurs, ce qui est normal pour un produit en phase de développement.

5.1.2. Test, Évaluation et Recommandation

Cette partie est faite d'un document présenté au groupe au mois d'avril 1998 (section suivante) et d'une liste de recommandations faite pour le simulateur de deuxième génération.

5.1.2.1. Limites de l'architecture de PULSE V1

Un document intitulé *Limites de l'architecture PULSE V1* a été présenté au groupe. Ce rapport présentait de façon chronologique les différentes limitations de PULSE V1 rencontrées depuis le début de ce projet.

CHAPITRE 5

CONCLUSION

Comme présenté au chapitre 1, ce projet avait deux objectifs principaux :

1. Le développement d'applications diverses et évaluation de leurs performances sur **PULSE**.
2. Le test et évaluation des outils en place, et apporter si possible des suggestions pour améliorer le produit.

5.1. DÉVELOPEMENT D'APPLICATIONS

Durant ce projet huit applications différentes ont été étudiées. De ces huit, une a été abandonnée avant d'avoir atteint le stade de l'écriture de code, il s'agit de la transformée de Hough. Cette transformée est un algorithme qui s'applique sur une image et qui à prime abord, comportait des éléments intéressants pour **PULSE** : un traitement intensif, de multiples opérations à effectuer sur les mêmes pixels et la possibilité de réutilisé des algorithmes déjà écrits sur **PULSE** :

- Calcul du niveau de gris (Sobel),
- Détection de contours (threshold),
- Calcul du gradient (direction),
- Calcul de la position dans un espace spécial nommé espace de Hough,
- Production d'un histogramme,

Deux facteurs ont mené à l'abandon de cet algorithme, soit le calcul de la position dans l'espace de Hough et le calcul du gradient qui demandait l'emploi d'une division.

Des sept applications restantes, six ont permis d'effectuer de multiples simulations et itérations sur les algorithmes.

La septième, l'algorithme de chiffrage IDEA a été développé en entier. La non-disponibilité d'une instruction conditionnelle sur 32 bits afin d'effectuer la multiplication modulo $2^{16} + 1$ de façon rapide a rendu inutile toute optimisation.

Les autres applications ; multiplication vecteurs – matrice, algorithme de Bresenham, algorithme RSA, Équation aux différences finies, Morphologie binaire avec l'Érosion et la

Dilatation d'images, présentent une variété d'applications numériques. Une des requêtes du groupe **PULSE** était d'évaluer avec quelle facilité, ou difficulté, différents types d'algorithmes pouvaient être portés sur **PULSE** et d'essayer de sortir des traits caractéristiques qui fonctionnaient bien sur **PULSE**.

5.1.1. Développement sur PULSE

Basé sur les résultats des comparables vus au chapitre précédent, les chiffres obtenus lors de ce projet se comparent avantageusement à ce qui se fait à l'extérieur. On note cependant une lacune évidente. En effet, le travail de cette recherche s'est étendu sur environ 16 mois, incluant une session passée à temps plein sur l'avancement des travaux de recherches. En mettant en relation le temps investi dans ce projet et les quelques centaines de lignes de code écrites, il y a un déséquilibre important entre les deux.

Ce déséquilibre s'explique par plusieurs facteurs qui font partie des difficultés rencontrées durant ce projet.

1- **Développement d'applications performantes :** il s'agit du critère majeur de développement qui a dirigé les actions et les changements effectués durant la phase de développement. Les multiples itérations et simulations n'avaient qu'un seul but : traiter plus de données en un même laps de temps. La recherche de vitesse pour chacune des applications se joue au niveau des instructions parallèles et par le retrait des instructions vides (*NOP*). Cette recherche nécessite souvent un réarrangement de la façon donc se déroule le programme, de multiples changements mineurs et des tests de performances entre chaque changement.

2- **Diversité des applications :** Au lieu d'exploiter un filon d'application similaire et de développer mes connaissances dans ce domaine spécifique, j'ai appliqué les directives du groupe en touchant à différents domaines.

3- **Courbe d'apprentissage :** Hormis une introduction au logiciel PVM sur UNIX, à l'occasion d'un cours sur les réseau d'ordinateurs, **PULSE** aura été une première expérience de programmation sur un système parallèle. Le fait de prendre plusieurs semaines en début de projet pour étudier les grands principes du parallélisme et les quatre compétiteurs de PULSE aura eu un effet d'ouverture sur la suite de ce projet. La plus grande difficulté d'apprentissage s'est trouvée dans l'environnement de développement de **PULSE**. Une architecture et un langage de programmation qui rivalisent de complexité l'un avec l'autre. Il est certain que le fait de se retrouver à 500 kilomètres des autres membres du groupe a contribué aux difficultés d'apprentissage durant le déroulement de ce projet. Je crois pouvoir affirmer avec expérience que l'ère des communications améliore grandement la rapidité et les possibilités de développement en différent lieu physique, une fois la courbe d'apprentissage gravie.

4- **Produit non totalement stable :** une partie du projet était de pouvoir améliorer l'environnement de développement, l'architecture et de pouvoir définir les limitations courantes de **PULSE**. Le déroulement du projet s'est donc effectué sur un produit en développement. Ainsi, plusieurs problèmes de conception ont été mis à jour, tant dans le circuit que dans ses outils de développement. C'est un privilège

de pouvoir participer à la validation d'un produit mais cela comporte aussi des contraintes dans le développement et la validation d'applications.

5- **Documentation :** la documentation présente possédait des erreurs, ce qui est normal pour un produit en phase de développement.

5.1.2. Test, Évaluation et Recommandation

Cette partie est faite d'un document présenté au groupe au mois d'avril 1998 (section suivante) et d'une liste de recommandations faite pour le simulateur de deuxième génération.

5.1.2.1. Limites de l'architecture de PULSE V1

Un document intitulé *Limites de l'architecture PULSE V1* a été présenté au groupe. Ce rapport présentait de façon chronologique les différentes limitations de PULSE V1 rencontrées depuis le début de ce projet.

CHAPITRE 5

CONCLUSION

Comme présenté au chapitre 1, ce projet avait deux objectifs principaux :

1. Le développement d'applications diverses et évaluation de leurs performances sur **PULSE**.
2. Le test et évaluation des outils en place, et apporter si possible des suggestions pour améliorer le produit.

5.1. DÉVELOPEMENT D'APPLICATIONS

Durant ce projet huit applications différentes ont été étudiées. De ces huit, une a été abandonnée avant d'avoir atteint le stade de l'écriture de code, il s'agit de la transformée de Hough. Cette transformée est un algorithme qui s'applique sur une image et qui à prime abord, comportait des éléments intéressants pour **PULSE** : un traitement intensif, de multiples opérations à effectuer sur les mêmes pixels et la possibilité de réutilisé des algorithmes déjà écrits sur **PULSE** :

- Calcul du niveau de gris (Sobel),
- Détection de contours (threshold),
- Calcul du gradient (direction),
- Calcul de la position dans un espace spécial nommé espace de Hough,
- Production d'un histogramme,

Deux facteurs ont mené à l'abandon de cet algorithme, soit le calcul de la position dans l'espace de Hough et le calcul du gradient qui demandait l'emploi d'une division.

Des sept applications restantes, six ont permis d'effectuer de multiples simulations et itérations sur les algorithmes.

La septième, l'algorithme de chiffrage IDEA a été développé en entier. La non-disponibilité d'une instruction conditionnelle sur 32 bits afin d'effectuer la multiplication modulo $2^{16} + 1$ de façon rapide a rendu inutile toute optimisation.

Les autres applications ; multiplication vecteurs – matrice, algorithme de Bresenham, algorithme RSA, Équation aux différences finies, Morphologie binaire avec l'Érosion et la

Dilatation d'images, présentent une variété d'applications numériques. Une des requêtes du groupe **PULSE** était d'évaluer avec quelle facilité, ou difficulté, différents types d'algorithmes pouvaient être portés sur **PULSE** et d'essayer de sortir des traits caractéristiques qui fonctionnaient bien sur **PULSE**.

5.1.1. Développement sur PULSE

Basé sur les résultats des comparables vus au chapitre précédent, les chiffres obtenus lors de ce projet se comparent avantageusement à ce qui se fait à l'extérieur. On note cependant une lacune évidente. En effet, le travail de cette recherche s'est étendu sur environ 16 mois, incluant une session passée à temps plein sur l'avancement des travaux de recherches. En mettant en relation le temps investi dans ce projet et les quelques centaines de lignes de code écrites, il y a un déséquilibre important entre les deux.

Ce déséquilibre s'explique par plusieurs facteurs qui font partie des difficultés rencontrées durant ce projet.

1- **Développement d'applications performantes :** il s'agit du critère majeur de développement qui a dirigé les actions et les changements effectués durant la phase de développement. Les multiples itérations et simulations n'avaient qu'un seul but : traiter plus de données en un même laps de temps. La recherche de vitesse pour chacune des applications se joue au niveau des instructions parallèles et par le retrait des instructions vides (*NOP*). Cette recherche nécessite souvent un réarrangement de la façon donc se déroule le programme, de multiples changements mineurs et des tests de performances entre chaque changement.

2- **Diversité des applications :** Au lieu d'exploiter un filon d'application similaire et de développer mes connaissances dans ce domaine spécifique, j'ai appliqué les directives du groupe en touchant à différents domaines.

3- **Courbe d'apprentissage :** Hormis une introduction au logiciel PVM sur UNIX, à l'occasion d'un cours sur les réseau d'ordinateurs, **PULSE** aura été une première expérience de programmation sur un système parallèle. Le fait de prendre plusieurs semaines en début de projet pour étudier les grands principes du parallélisme et les quatre compétiteurs de PULSE aura eu un effet d'ouverture sur la suite de ce projet. La plus grande difficulté d'apprentissage s'est trouvée dans l'environnement de développement de **PULSE**. Une architecture et un langage de programmation qui rivalisent de complexité l'un avec l'autre. Il est certain que le fait de se retrouver à 500 kilomètres des autres membres du groupe a contribué aux difficultés d'apprentissage durant le déroulement de ce projet. Je crois pouvoir affirmer avec expérience que l'ère des communications améliore grandement la rapidité et les possibilités de développement en différent lieu physique, une fois la courbe d'apprentissage gravie.

4- **Produit non totalement stable :** une partie du projet était de pouvoir améliorer l'environnement de développement, l'architecture et de pouvoir définir les limitations courantes de **PULSE**. Le déroulement du projet s'est donc effectué sur un produit en développement. Ainsi, plusieurs problèmes de conception ont été mis à jour, tant dans le circuit que dans ses outils de développement. C'est un privilège

de pouvoir participer à la validation d'un produit mais cela comporte aussi des contraintes dans le développement et la validation d'applications.

5- **Documentation :** la documentation présente possédait des erreurs, ce qui est normal pour un produit en phase de développement.

5.1.2. Test, Évaluation et Recommandation

Cette partie est faite d'un document présenté au groupe au mois d'avril 1998 (section suivante) et d'une liste de recommandations faite pour le simulateur de deuxième génération.

5.1.2.1. Limites de l'architecture de PULSE V1

Un document intitulé *Limites de l'architecture PULSE V1* a été présenté au groupe. Ce rapport présentait de façon chronologique les différentes limitations de PULSE V1 rencontrées depuis le début de ce projet.

CHAPITRE 5

CONCLUSION

Comme présenté au chapitre 1, ce projet avait deux objectifs principaux :

1. Le développement d'applications diverses et évaluation de leurs performances sur **PULSE**.
2. Le test et évaluation des outils en place, et apporter si possible des suggestions pour améliorer le produit.

5.1. DÉVELOPEMENT D'APPLICATIONS

Durant ce projet huit applications différentes ont été étudiées. De ces huit, une a été abandonnée avant d'avoir atteint le stade de l'écriture de code, il s'agit de la transformée de Hough. Cette transformée est un algorithme qui s'applique sur une image et qui à prime abord, comportait des éléments intéressants pour **PULSE** : un traitement intensif, de multiples opérations à effectuer sur les mêmes pixels et la possibilité de réutilisé des algorithmes déjà écrits sur **PULSE** :

- Calcul du niveau de gris (Sobel),
- Détection de contours (threshold),
- Calcul du gradient (direction),
- Calcul de la position dans un espace spécial nommé espace de Hough,
- Production d'un histogramme,

Deux facteurs ont mené à l'abandon de cet algorithme, soit le calcul de la position dans l'espace de Hough et le calcul du gradient qui demandait l'emploi d'une division.

Des sept applications restantes, six ont permis d'effectuer de multiples simulations et itérations sur les algorithmes.

La septième, l'algorithme de chiffrage IDEA a été développé en entier. La non-disponibilité d'une instruction conditionnelle sur 32 bits afin d'effectuer la multiplication modulo $2^{16} + 1$ de façon rapide a rendu inutile toute optimisation.

Les autres applications ; multiplication vecteurs – matrice, algorithme de Bresenham, algorithme RSA, Équation aux différences finies, Morphologie binaire avec l'Érosion et la

Dilatation d'images, présentent une variété d'applications numériques. Une des requêtes du groupe **PULSE** était d'évaluer avec quelle facilité, ou difficulté, différents types d'algorithmes pouvaient être portés sur **PULSE** et d'essayer de sortir des traits caractéristiques qui fonctionnaient bien sur **PULSE**.

5.1.1. Développement sur PULSE

Basé sur les résultats des comparables vus au chapitre précédent, les chiffres obtenus lors de ce projet se comparent avantageusement à ce qui se fait à l'extérieur. On note cependant une lacune évidente. En effet, le travail de cette recherche s'est étendu sur environ 16 mois, incluant une session passée à temps plein sur l'avancement des travaux de recherches. En mettant en relation le temps investi dans ce projet et les quelques centaines de lignes de code écrites, il y a un déséquilibre important entre les deux.

Ce déséquilibre s'explique par plusieurs facteurs qui font partie des difficultés rencontrées durant ce projet.

1- **Développement d'applications performantes :** il s'agit du critère majeur de développement qui a dirigé les actions et les changements effectués durant la phase de développement. Les multiples itérations et simulations n'avaient qu'un seul but : traiter plus de données en un même laps de temps. La recherche de vitesse pour chacune des applications se joue au niveau des instructions parallèles et par le retrait des instructions vides (*NOP*). Cette recherche nécessite souvent un réarrangement de la façon donc se déroule le programme, de multiples changements mineurs et des tests de performances entre chaque changement.

2- **Diversité des applications :** Au lieu d'exploiter un filon d'application similaire et de développer mes connaissances dans ce domaine spécifique, j'ai appliqué les directives du groupe en touchant à différents domaines.

3- **Courbe d'apprentissage :** Hormis une introduction au logiciel PVM sur UNIX, à l'occasion d'un cours sur les réseau d'ordinateurs, **PULSE** aura été une première expérience de programmation sur un système parallèle. Le fait de prendre plusieurs semaines en début de projet pour étudier les grands principes du parallélisme et les quatre compétiteurs de PULSE aura eu un effet d'ouverture sur la suite de ce projet. La plus grande difficulté d'apprentissage s'est trouvée dans l'environnement de développement de **PULSE**. Une architecture et un langage de programmation qui rivalisent de complexité l'un avec l'autre. Il est certain que le fait de se retrouver à 500 kilomètres des autres membres du groupe a contribué aux difficultés d'apprentissage durant le déroulement de ce projet. Je crois pouvoir affirmer avec expérience que l'ère des communications améliore grandement la rapidité et les possibilités de développement en différent lieu physique, une fois la courbe d'apprentissage gravie.

4- **Produit non totalement stable :** une partie du projet était de pouvoir améliorer l'environnement de développement, l'architecture et de pouvoir définir les limitations courantes de **PULSE**. Le déroulement du projet s'est donc effectué sur un produit en développement. Ainsi, plusieurs problèmes de conception ont été mis à jour, tant dans le circuit que dans ses outils de développement. C'est un privilège

de pouvoir participer à la validation d'un produit mais cela comporte aussi des contraintes dans le développement et la validation d'applications.

5- **Documentation :** la documentation présente possédait des erreurs, ce qui est normal pour un produit en phase de développement.

5.1.2. Test, Évaluation et Recommandation

Cette partie est faite d'un document présenté au groupe au mois d'avril 1998 (section suivante) et d'une liste de recommandations faite pour le simulateur de deuxième génération.

5.1.2.1. Limites de l'architecture de PULSE V1

Un document intitulé *Limites de l'architecture PULSE V1* a été présenté au groupe. Ce rapport présentait de façon chronologique les différentes limitations de PULSE V1 rencontrées depuis le début de ce projet.

CHAPITRE 5

CONCLUSION

Comme présenté au chapitre 1, ce projet avait deux objectifs principaux :

1. Le développement d'applications diverses et évaluation de leurs performances sur **PULSE**.
2. Le test et évaluation des outils en place, et apporter si possible des suggestions pour améliorer le produit.

5.1. DÉVELOPEMENT D'APPLICATIONS

Durant ce projet huit applications différentes ont été étudiées. De ces huit, une a été abandonnée avant d'avoir atteint le stade de l'écriture de code, il s'agit de la transformée de Hough. Cette transformée est un algorithme qui s'applique sur une image et qui à prime abord, comportait des éléments intéressants pour **PULSE** : un traitement intensif, de multiples opérations à effectuer sur les mêmes pixels et la possibilité de réutilisé des algorithmes déjà écrits sur **PULSE** :

- Calcul du niveau de gris (Sobel),
- Détection de contours (threshold),
- Calcul du gradient (direction),
- Calcul de la position dans un espace spécial nommé espace de Hough,
- Production d'un histogramme,

Deux facteurs ont mené à l'abandon de cet algorithme, soit le calcul de la position dans l'espace de Hough et le calcul du gradient qui demandait l'emploi d'une division.

Des sept applications restantes, six ont permis d'effectuer de multiples simulations et itérations sur les algorithmes.

La septième, l'algorithme de chiffrage IDEA a été développé en entier. La non-disponibilité d'une instruction conditionnelle sur 32 bits afin d'effectuer la multiplication modulo $2^{16} + 1$ de façon rapide a rendu inutile toute optimisation.

Les autres applications ; multiplication vecteurs – matrice, algorithme de Bresenham, algorithme RSA, Équation aux différences finies, Morphologie binaire avec l'Érosion et la

Dilatation d'images, présentent une variété d'applications numériques. Une des requêtes du groupe **PULSE** était d'évaluer avec quelle facilité, ou difficulté, différents types d'algorithmes pouvaient être portés sur **PULSE** et d'essayer de sortir des traits caractéristiques qui fonctionnaient bien sur **PULSE**.

5.1.1. Développement sur PULSE

Basé sur les résultats des comparables vus au chapitre précédent, les chiffres obtenus lors de ce projet se comparent avantageusement à ce qui se fait à l'extérieur. On note cependant une lacune évidente. En effet, le travail de cette recherche s'est étendu sur environ 16 mois, incluant une session passée à temps plein sur l'avancement des travaux de recherches. En mettant en relation le temps investi dans ce projet et les quelques centaines de lignes de code écrites, il y a un déséquilibre important entre les deux.

Ce déséquilibre s'explique par plusieurs facteurs qui font partie des difficultés rencontrées durant ce projet.

1- **Développement d'applications performantes :** il s'agit du critère majeur de développement qui a dirigé les actions et les changements effectués durant la phase de développement. Les multiples itérations et simulations n'avaient qu'un seul but : traiter plus de données en un même laps de temps. La recherche de vitesse pour chacune des applications se joue au niveau des instructions parallèles et par le retrait des instructions vides (*NOP*). Cette recherche nécessite souvent un réarrangement de la façon donc se déroule le programme, de multiples changements mineurs et des tests de performances entre chaque changement.

2- **Diversité des applications :** Au lieu d'exploiter un filon d'application similaire et de développer mes connaissances dans ce domaine spécifique, j'ai appliqué les directives du groupe en touchant à différents domaines.

3- **Courbe d'apprentissage :** Hormis une introduction au logiciel PVM sur UNIX, à l'occasion d'un cours sur les réseau d'ordinateurs, **PULSE** aura été une première expérience de programmation sur un système parallèle. Le fait de prendre plusieurs semaines en début de projet pour étudier les grands principes du parallélisme et les quatre compétiteurs de PULSE aura eu un effet d'ouverture sur la suite de ce projet. La plus grande difficulté d'apprentissage s'est trouvée dans l'environnement de développement de **PULSE**. Une architecture et un langage de programmation qui rivalisent de complexité l'un avec l'autre. Il est certain que le fait de se retrouver à 500 kilomètres des autres membres du groupe a contribué aux difficultés d'apprentissage durant le déroulement de ce projet. Je crois pouvoir affirmer avec expérience que l'ère des communications améliore grandement la rapidité et les possibilités de développement en différent lieu physique, une fois la courbe d'apprentissage gravie.

4- **Produit non totalement stable :** une partie du projet était de pouvoir améliorer l'environnement de développement, l'architecture et de pouvoir définir les limitations courantes de **PULSE**. Le déroulement du projet s'est donc effectué sur un produit en développement. Ainsi, plusieurs problèmes de conception ont été mis à jour, tant dans le circuit que dans ses outils de développement. C'est un privilège

de pouvoir participer à la validation d'un produit mais cela comporte aussi des contraintes dans le développement et la validation d'applications.

5- **Documentation :** la documentation présente possédait des erreurs, ce qui est normal pour un produit en phase de développement.

5.1.2. Test, Évaluation et Recommandation

Cette partie est faite d'un document présenté au groupe au mois d'avril 1998 (section suivante) et d'une liste de recommandations faite pour le simulateur de deuxième génération.

5.1.2.1. Limites de l'architecture de PULSE V1

Un document intitulé *Limites de l'architecture PULSE V1* a été présenté au groupe. Ce rapport présentait de façon chronologique les différentes limitations de PULSE V1 rencontrées depuis le début de ce projet.