HABILITATION À DIRIGER DES RECHERCHES

présentée devant

**L'Université de Rennes 1**
**Institut de Formation Supérieure**
**en Informatique et en Communication**

par

Tanguy Risset

# Contribution à la compilation de nids de boucle sur silicium

**soutenue le 22 Novembre 2000 devant le jury composé de**

| | | |
|---|---|---|
| M. | Paul Feautrier | Rapporteur |
| M. | Francky Catthoor | Rapporteur |
| M. | Eric Martin | Rapporteur |
| Mlle | Anne Mignotte | Rapporteur |
| M. | Patrice Quinton | Examinateur |
| M. | Yves Robert | Examinateur |

# Table des matières

**1 Introduction** **3**
1.1 Contexte des recherches . . . 4
1.2 Objectifs de la présentation . . . 7

**2 Concepts de base** **9**
2.1 Contexte scientifique . . . 9
2.2 Le langage Alpha . . . 13
2.2.1 Le langage Alpha originel . . . 14
2.2.2 Les extensions proposées . . . 16
2.2.3 Alpha0 et AlpHard . . . 20
2.3 Le système MMAlpha . . . 21

**3 Conception de circuits avec MMAlpha** **25**
3.1 Méthodologie standard avec MMAlpha . . . 25
3.1.1 Uniformisation . . . 26
3.1.2 Ordonnancement . . . 28
3.1.3 Génération RTL . . . 33
3.1.4 Analyse et simulation . . . 36
3.1.5 Après MMAlpha . . . 37
3.2 Traitements additionnels pour la conception . . . 40
3.2.1 les $\mathbb{Z}$-polyèdres . . . 40
3.2.2 Largeur en bits du chemin de donnée . . . 43
3.2.3 Interfaçage . . . 44
3.2.4 Compilation de code impératif . . . 46
3.2.5 Preuve algébrique . . . 48
3.2.6 Partitionnement et mémoire . . . 48
3.2.7 La conception hiérarchique . . . 52
3.3 Un exemple complet: Samba . . . 52
3.3.1 L'algorithme de comparaison de séquence . . . 52
3.3.2 Conception avec MMAlpha . . . 54
3.3.3 Résultats et performances . . . 65

**4 Synthèse et perspectives** **67**
4.1 Autres travaux autour de MMAlpha . . . . . . . . . . . . . . . . . . . . . . 67
4.1.1 Codage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
4.1.2 Estimation de mouvement . . . . . . . . . . . . . . . . . . . . . 69
4.1.3 Problème du chemin algébrique . . . . . . . . . . . . . . . . . . 69
4.1.4 Filtrage adaptatif . . . . . . . . . . . . . . . . . . . . . . . . . 70
4.2 Travaux et outils similaires à MMAlpha . . . . . . . . . . . . . . . . . . . 71
4.3 Synthèse et perspectives . . . . . . . . . . . . . . . . . . . . . . . . . . 73
4.3.1 Synthèse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
4.3.2 Perspectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75

**A Résultats de la conception de Samba présentée en section 3.3** **89**
A.1 Programme Alpha original . . . . . . . . . . . . . . . . . . . . . . . . . . 89
A.2 Programme Alpha0 dérivé du programme de la section A.1 . . . . . . . . . 90
A.3 Programme AlpHard dérivé du programme de la section A.2 . . . . . . . . 93
A.3.1 Interface du programme AlpHard . . . . . . . . . . . . . . . . . 93
A.3.2 Module du programme AlpHard . . . . . . . . . . . . . . . . . . 94
A.3.3 Cellule 1 du programme AlpHard . . . . . . . . . . . . . . . . . 96
A.3.4 Cellule 4 du programme AlpHard . . . . . . . . . . . . . . . . . 98
A.3.5 Controleur du programme AlpHard . . . . . . . . . . . . . . . . 98
A.4 Code VHDL généré par MMAlpha . . . . . . . . . . . . . . . . . . . . . . 99
A.4.1 Module . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
A.4.2 cellule 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
A.4.3 Controleur . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109

**B Liste de packages de MMAlpha** **113**

# Chapitre 1

# Introduction

Ce document traite de la compilation de circuits intégrés. Aujourd'hui les circuits intégrés sont décrits “à la main” en langage VHDL avant d'être implémentés. La complexité des circuits augmentant, il devient urgent de remplacer l'écriture manuelle par une compilation d'un langage plus “évolué” que VHDL. L'informatique a beau être une science jeune, il est difficile de changer une habitude ou un paradigme adoptés par toute une communauté. Le problème s'est par exemple déjà posé lorsqu'il s'est agi d'utiliser le calcul parallèle plutôt que le calcul séquentiel. Dans le cas de la production de circuits, l'introduction d'un mécanisme de compilation haut niveau implique beaucoup de changements : introduction de nouveaux formats pour la description de matériel et même mieux, de nouveaux langages, introduction de nouvelles méthodes de conception et de nouveaux outils. L'avantage d'une telle compilation est clair : on obtient plus de souplesse dans la conception. En effet, le concepteur ayant la possibilité de *régler* le parallélisme, il peut choisir entre une architecture à fort parallélisme qui sera rapide mais chère et une architecture à faible parallélisme qui sera plus lente mais moins chère. Le vrai défi n'est pas la compilation de circuits, mais la compilation/parallélisation, c'est à dire la réalisation d'un compilateur *intelligent* qui parallélise les applications de manière adéquate.

Aujourd'hui on sait compiler/paralléliser les opérations scalaires et les affectations, mais on ne sait pas compiler/paralléliser les boucles (`for`, `while`) et les indirections. Les indirections et les boucles `while` n'ont pas été traitées dans ce travail de recherche : cela reste du domaine des extensions envisageables lorsque les boucles `for` à contrôle statique auront été traitées correctement. Le sujet de ce document de synthèse est donc la compilation/parallélisation des circuits à partir d'algorithmes composés de nids de boucles à contrôle statique. Ce document synthétise les résultats des recherches que j'ai mené en tant que chargé de recherche INRIA au sein du projet Api (puis Cosi) après ma thèse (de 1994 à 2000).

L'approche présentée ici, basée sur le langage Alpha et le logiciel MMAlpha, n'est évidement pas la seule possible pour résoudre ce problème. Néanmoins c'est l'une des plus avancées dans le monde du fait, d'une part de la force de la communauté française dans ce domaine, d'autre part de l'éclectisme du projet Api/Cosi de l'Irisa. La composante “synthèse de haut niveau” du projet Api/Cosi a pris le parti de s'appuyer sur un langage solide (le langage Alpha) et de ne pas fixer la plate-forme visée par la compilation (génération possible sur plate-forme VLSI, FPGA,...). Nous [1] sommes aussi restés dans le domaine de la recherche fondamentale, c'est-à-dire que les expérimentations ont toujours été accompagnées de tentatives de formalisation et de généralisation des résultats obtenus. Nous avons cependant fourni un gros effort de développement à travers MMAlpha au risque d'obtenir un outil obsolète, courant sans cesse après les nouvelles technologies. Aujourd'hui, il me semble que ce risque est écarté. Les techniques utilisées, rassemblées sous le nom de *modèle polyédrique*, forment une base suffisamment solide pour que les outils correspondants s'adaptent facilement aux nouvelles technologies. Par exemple, l'adaptation aux architectures reconfigurables a été réalisée rapidement. De plus, l'intérêt industriel pour ce domaine de recherche de développe de jour en jour (il faut dire qu'on est parti de bien bas!).

La suite de ce chapitre refchapIntro précise le contexte et les motivations des recherches ainsi l'objectif du présent document. Le chapitre 2 présente les notions importantes utilisées, en particulier le langage Alpha et le système MMAlpha. Le chapitre 3 est le cœur du rapport : toutes les étapes de compilation/parallélisation de circuits utilisées dans la méthodologie de conception mise en place y sont exposées. Un exemple de conception complète y est présenté pour fixer les idées (section 3.3). Le chapitre 4 mentionne quelques travaux qui ne sont pas directement liés à la mise en place de la méthodologie de compilation/parallélisation de circuits. Il passe ensuite en revue d'autres travaux et outils en relation avec la parallélisation automatique puis fait une synthèse des recherches présentées dans ce document.

## 1.1 Contexte des recherches

Cette section précise le contexte dans lequel les recherches ont été effectuées. Les contextes environnemental, scientifique, et technologique influencent énormément l'approche scientifique ainsi que les résultats produits.

### Contexte environnemental

Le travail présenté dans ce document a bénéficié de deux environnements: la communauté française de parallélisation automatique et le projet INRIA Api/Cosi de l'Irisa. À l'époque de ma thèse (1991-1994), le groupement PRC $C^3$ (Programme de Recherche Coordonné Coopération, Communication, Concurrence) a créé des liens très forts entre les membres de la communauté française de parallélisation automatique et aujourd'hui encore cette communauté se réunit régulièrement pour des présentations informelles des travaux respectifs des différentes équipes. C'est aussi pendant cette période que j'ai participé aux projets européens Nana et Nana2 qui m'ont fait connaître la plupart des groupes de recherche européens travaillant dans le domaine de la parallélisation automatique. De ces échanges est né ce que nous appelons maintenant le *modèle polyédrique*, qui est omniprésent dans ce document. Lors de mon recrutement comme chargé de recherche INRIA dans le projet Api, j'ai eu l'occasion de découvrir la communauté française de synthèse de circuits, que rassemblait le PRC CAO (Conception Assistée par Ordinateur). J'ai pu aussi

1. Dans ce document le "nous" désignera la composante "synthèse de haut niveau" du projet Api/Cosi dont je fait parti. J'utiliserai le "je" lorsqu'il s'agit de spécifier un travail ou une opinion personnel

dialoguer avec des concepteurs de circuits et me familiariser avec leur langage. Le double environnement, compilation/parallélisation [2] et CAO/synthèse de circuits, faisait du projet Api (Architecture Parallèles Intégrées) un lieu unique. Ce projet, dont le successeur naturel est le projet Cosi (Conception de Système sur SIlicium), s'est révélé très productif du point de vue de la recherche, et les résultats présentés ici ne sont qu'un aspect de cette productivité.

### Contexte scientifique

Ce travail de recherche est l'héritier des travaux sur la synthèse automatique de réseaux systoliques et des travaux sur la parallélisation automatique de programmes. Le modèle systolique est un modèle d'architecture proposé pour implémenter efficacement les calculs réguliers effectués, par exemple, dans les programmes contenant des boucles imbriquées (la section 2.1 décrit plus précisément le modèle systolique). La simplicité du modèle est à la fois sa force et sa faiblesse. Il permet à "peu de frais" aux informaticiens et aux théoriciens d'avoir accès au monde des circuits intégrés, de proposer de nouvelles architectures et surtout de proposer des méthodes systématiques de génération d'architectures systoliques. Mais dans les faits le modèle se révèle trop simpliste, et peu d'implémentations effectives référencent les travaux mentionnés ci-dessus. Cette désaffection a conduit à un rejet du systolique à divers degrés. Néanmoins les résultats des recherches liés aux des architectures systoliques sont toujours utilisés.

Beaucoup des techniques développées autour du systolique servent à la parallélisation automatique. Ces techniques, centrées sur les manipulations de boucles imbriquées (ou nids de boucle), se revendiquent du modèle polyédrique. Grâce en particulier aux efforts de la communauté française, ce modèle est maintenant reconnu au niveau international et certaines applications apparaissent dans les paralléliseurs commerciaux.

Parallèlement, quelques groupes ont continué leurs efforts de recherche sur la synthèse de circuits et de véritables outils apparaissent aujourd'hui (voir section 4.2). C'est le cas du logiciel MMAlpha au sein du projet Cosi. Lors de mon arrivée dans le projet Api, le langage Alpha existait déjà depuis plusieurs années et des ébauches d'environnement de manipulation de programmes avaient été réalisées. Mon travail a consisté en l'étude et la réalisation d'un prototype d'environnement de compilation/parallélisation de circuits réguliers à partir du logiciel MMAlpha naissant.

### Contexte technologique

Il est important d'avoir conscience que l'évaluation de diverses solutions à un problème dépend très fortement de la technologie disponible (VLSI, FPGA, logiciel, etc...). Plus que des faits, ce sont des *évolutions* technologiques qui motivent l'approche présentée dans ce document.

Les concepteurs de circuits sont constamment confrontés au problème suivant : vaut-il mieux produire un circuit spécialisé dont le coût de construction est de 1 MF et dont la

2. Je désignerai tout au long du document le processus de compilation de circuits qui nous intéresse par *compilation/parallélisation*, ceci pour exprimer le fait qu'il s'agit bien d'un processus de compilation mais dans le lequel nous étudions très spécifiquement la parallélisation (en particulier, les optimisations classiques des compilateurs ne sont pas implémentées).

durée de conception est de un an ou vaut-il mieux attendre deux ans la sortie sur le marché d'un microprocesseur dix fois plus rapide que ceux du moment? Cette question illustre le problème très délicat du choix de la technologie cible d'une implémentation. Aujourd'hui il paraît clair que, si l'Europe a perdu la bataille des processeurs à usage général, elle n'a pas perdu celle des processeurs spécialisés. L'explosion de l'électronique à faible coût (*appliances*) est encore à venir. Ce développement s'accompagnera d'une augmentation de la complexité des systèmes implémentés en matériel (concept *System On Chip*). De ce fait, les notions de conception conjointe (*co-design*) et de propriété intellectuelle (*IP*) ont pris récemment beaucoup d'importance.

Du point de vue de la communauté développant le modèle polyédrique, la question importante est celle-ci: quel est l'élément décisif qui donnera à ces recherches un impact industriel alors que les techniques étudiées sont connues depuis plus dix ans? Le résultat du travail présenté ici est essentiellement une méthodologie de conception de circuits spécialisés. Sont détaillés ci-dessous trois aspects de cette conception qui sont à même de résoudre des problèmes succeptibles de devenir critiques pour les concepteurs de systèmes dans les années à venir.

– La conception proposée est une conception *très haut niveau*. Elle permet donc de prototyper rapidement diverses implémentations et d'explorer l'espace de conception beaucoup plus rapidement qu'avec une conception classique. En terme commercial, cela se traduit par un délai réduit pour l'apparition d'un circuit sur le marché. La conception d'un outil de prototypage rapide est certainement l'une des applications majeures des travaux présentés ici.
– La conception proposée est un conception sémantiquement correcte. L'aspect sémantique a toujours été une priorité. La conception de systèmes sans défaut (même si il s'agir d'une utopie) est l'un des défis majeurs des années à venir. Pour un même traitement, ces préoccupations passent de superflues à incontournables lorsque la taille des applications augmente, ce qui est le cas aujourd'hui.
– La conception proposée s'adapte naturellement aux architectures reconfigurables. Les architectures reconfigurables ou FPGA (Field Programmable Gate Array[3]) existent depuis longtemps, pourtant leur succès commercial ne fait que commencer. Sans faire de prospective sur les raisons d'un tel comportement commercial, on peut constater que ce marché a tiré les performances des FPGA vers le haut. C'est l'un des changements majeurs dans le domaine de la conception de circuits: les performances des FPGA augmentent plus vite que la loi de Moore (section 2.1). On peut dire aujourd'hui que les solutions à base de FPGA deviennent *de plus en plus* avantageuses. Actuellement les concepteurs de FPGA sont confrontés au même problème que les constructeurs de machines parallèles il y a 10 ans: la programmation efficace de telles machines est très difficile. D'où la nécessité d'un compilateur pour FPGA qui tirera efficacement profit du parallélisme offert par ces machines. Or les travaux présentés ici s'adaptent presque *canoniquement* aux architectures FPGA actuelles, du fait de l'architecture régulière et du très grand parallélisme potentiel de ces dernières.

3. Dans ce document, nous utilisons indifféremment les termes architectures reconfigurables ou FPGA, il se peut même que nous désignons par "les Xilinx" les architectures reconfigurables car nous avons uniquement travaillé sur des machines Xilinx. Même si les autres constructeurs ont des approches un peu différentes, les résultats que nous obtenons seront sans doute facilement transposables à leurs architectures.

C'est sans doute l'application des travaux la plus immédiatement réalisable.

**Motivations personnelles**

Les clés d'un succès ne sont pas seulement dans les indicateurs économiques. La motivation de chacun est un moteur bien plus puissant. Les travaux réalisés autour du modèle polyédrique allient le plaisir de la manipulation d'un outil mathématique non trivial avec la magie de l'obtention de résultats effectifs lors des implémentations. C'est pour cela que les chercheurs s'évertuent à démontrer que ces notions sont importantes. Comme tous ceux qui ont contracté le virus polyédrique, je suis en permanence étonné par la puissance du modèle et par le peu d'attention qu'il reçoit de la part de nombreux scientifiques. J'attache beaucoup d'importance à la reconnaissance naissante du modèle car elle récompense l'investissement et la confiance de toute la communauté. D'autre part, les technologies utilisées en synthèse de circuits imposent la manipulation de composants électroniques. Cet aspect "matériel" n'est que peu présent dans le monde de la recherche en informatique. L'orientation de mes recherches vers la synthèse de circuits doit beaucoup à cet aspect ludique.

## 1.2 Objectifs de la présentation

Un document d'habilitation à diriger des recherches sert avant tout à faire la synthèse de l'ensemble du travail de recherche réalisé par l'auteur. J'ai la chance de pouvoir articuler l'ensemble du document autour d'un thème unique : la compilation/parallélisation de circuits. J'aimerais que ce document remplisse aussi d'autres objectifs : donner une idée précise de la puissance de la compilation de circuits utilisant le modèle polyédrique aujourd'hui et défendre la démarche scientifique dont procède le langage Alpha et le logiciel MMAlpha.

**Évaluer la compilation/parallélisation de circuits**

J'ai essayé autant que possible d'avoir une approche pragmatique, c'est-à-dire d'appuyer tout résultat théorique par des implémentations et de fournir des *chiffres*. Les résultats des implémentations sont cruciaux pour les concepteurs car ce sont ces résultats qui vont les persuader d'utiliser les techniques de parallélisation automatique. Par exemple, des évaluations des circuits sont présentées en section 3.3.3 et 4.1, chiffrant le temps de compilation, la surface et le temps d'exécution. Ces chiffres aident à se faire une image mentale du développement de ces techniques.

Orthogonalement, j'ai essayé de montrer l'enchaînement des étapes de la compilation et de faire apparaître la façon dont elles dépendent l'une de l'autre. J'espère ainsi démontrer la nécessité d'un unique environnement de compilation (tel MMAlpha) pour arriver à compiler/paralléliser un circuit, environnement qui sera inévitablement de taille importante. J'ai aussi montré en section 4.1 quelles autres utilisations d'un tel environnement pouvaient être faites. En Bref, j'ai essayé de faire le point sur ce que doit proposer un environnement de compilation/parallélisation de circuits.

L'assemblage de ces deux réflexions devrait, je l'espère, permettre au lecteur averti de se forger une opinion sur l'applicabilité des techniques polyédriques au problème de la compilation/parallélisation de circuits.

### Défendre l'approche Alpha

les rivalités scientifiques existent, s'exprimant dans des querelles d'écoles scientifiques ou de personnes. Les critiques ou les désintérêts relatifs au langage Alpha et au logiciel MMAlpha sont nombreuses. Pourtant, même si le langage Alpha peut être jugé comme un "*horrible formalisme fonctionnel incompréhensible*", il n'empêche qu'il est incontournable! En d'autres termes, qu'elle soit officiellement appelée langage ou simplement format interne, la forme en assignation unique est nécessaire. L'obstination de Ed Depreterre à utiliser l'assignation unique, est exemplaire (voir section 4.2). De plus, beaucoup des caractéristiques syntaxiques non intuitives d'Alpha se justifient très bien lorsqu'on étudie en détail le processus de compilation/parallélisation. Même si Alpha doit être utilisé comme format interne, c'est un véritable langage de programmation.

D'autre part, en tant qu'"ingénieur en chef" du logiciel MMAlpha, je suis persuadé que nous ne serions pas arrivé à conserver la maîtrise du logiciel aussi longtemps si nous n'avions pas utilisé Mathematica. En pratique, le fait que l'accès au logiciel ne soit pas totalement gratuit n'est pas un problème pour le monde académique. En revanche, dans le cadre d'une utilisation industrielle, le fait de dépendre de Mathematica est beaucoup moins bien perçu.

Enfin, "l'approche scientifique Alpha", c'est aussi la capacité à se plonger dans divers domaines scientifiques pour communiquer et collaborer (voir par exemple les travaux mentionnés dans la section 4.1). J'ai pu rencontrer des gens exceptionnels provenant de beaucoup d'horizons scientifiques différents. Il me semble essentiel de conserver cette pluridisciplinarité, même si elle se fait au détriment de l'excellence dans un domaine particulier.

# Chapitre 2

# Concepts de base

Ce chapitre décrit plus précisément le contexte dans lequel se sont faites les recherches. On y trouve d'abord une description rapide des concepts scientifiques et technologiques utilisés, puis la description du langage Alpha et les extensions que nous y avons apportés, et enfin une présentation rapide de l'environnement MMAlpha qui a servi de plate-forme logicielle pour toutes les recherches présentées ici.

## 2.1 Contexte scientifique

Le contexte scientifique de ces recherches s'articule autour de trois concepts importants: les circuits intégrés, le modèle systolique, et la synthèse automatique de circuits.

### Les circuits intégrés

Un circuit intégré est un système permettant le contrôle des signaux électriques qui est implémenté avec une technologie permettant une très grande intégration (VLSI). Un circuit peut être digital ou analogique (pour l'interface avec des mécanismes de mesure physique par exemple). Il existe de nombreuses manières de représenter un circuit (voir figure 2.1). Dans la représentation que nous utiliserons dans ce rapport, un circuit digital est constitué de fils véhiculant les valeurs (booléenne, entière ou autres), d'opérateurs et de registres pouvant stocker une valeur entre deux tops d'une certaine *horloge* (c'est ce que par la suite j'appellerai une représentation au niveau *transfert de registre*, ou RTL: register transfer level). Un circuit digital est est dit synchrone si tous les registres sont cadencés par la même horloge. Dans ce rapport, on ne s'intéresse qu'aux circuits digitaux synchrones [1]. En plus de ces composants de base, on trouve aussi les bus et divers type de mémoire (S-RAM, D-RAM, ROM, etc...).

La conception des processeurs à usage général est un cas particulier (peu de compagnies en sont capables). Par contre, des circuits spécialisés sont conçus par beaucoup de

1. Notons que l'outil présenté ici pourrait probablement être utilisé dans le cadre d'une conception asynchrone, le modèle de calcul synchrone étant plus simple pour le concepteur.

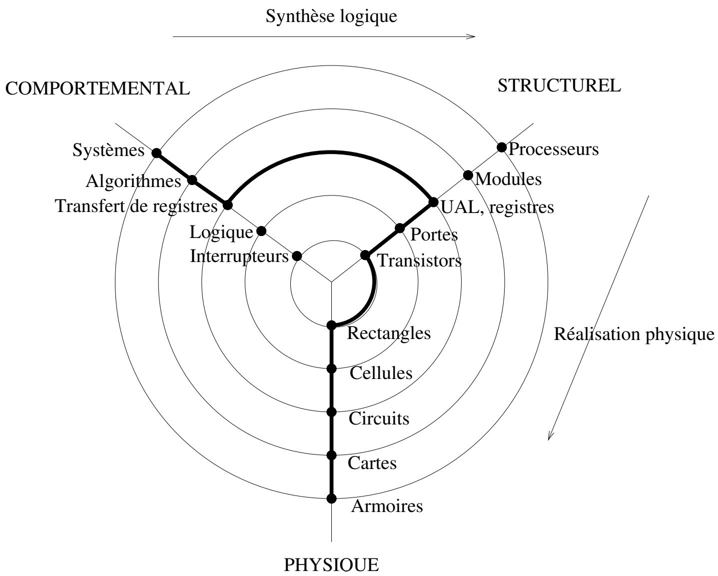


FIG. 2.1 – *Le fameux "diagramme de Gajski" représentant les trois axes de représentation des circuits: comportemental, structurel et physique. Le chemin marqué en gras est celui que nous suivons lors de la synthèse très haut niveau présentée ici. Plus précisément, la partie que nous étudions est le premier segment du chemin gras, en haut à gauche.*

petites et grosses compagnies (citons ST-micro, Alcatel, Comatlas, comme exemples français). Aujourd'hui, la synthèse industrielle de circuits se fait presque partout à partir d'une description RTL de l'architecture exprimée dans un langage de description d'architecture (VHDL ou Verilog). La *synthèse logique* est alors effectuée à l'aide de très gros outils de CAO (Cadence, Synopsis, Mentor Graphics sont les plus connus).

La conception de l'architecture à partir de l'algorithme à implémenter est donc faite "manuellement" par le concepteur, aidé d'un spécialiste du domaine dans lequel l'algorithme est utilisé. Cette méthodologie est évidement très coûteuse. Bien que ces chiffres dépendent énormément de l'application implémentée et évoluent très vite, on peut estimer à 10 à 12 mois le temps écoulé entre la prise de décision de la fabrication d'un circuit pour un traitement particulier et son arrivée sur le marché, et à au moins un millions de francs le coût d'une telle conception industrielle. Ces chiffres expliquent, d'une part la difficulté pour une équipe de recherche à rester compétitive dans ce domaine, d'autre part, la demande très forte pour des outils de conception de plus haut niveau que ceux qui existent actuellement.

La technologie utilisée pour le calcul reconfigurable permet de rapprocher la notion de circuit de celle de programme. Le circuit peut être télé-chargé sur le FPGA, puis exécuté. Pour d'obscures raisons technologico-économiques, cette technologie est longtemps restée dans l'ombre de la technologie VLSI. Aujourd'hui les fabricants de FPGA rattrapent leur retard: le Virtex XCV1000 permet d'émuler 1 millions de portes logiques (à titre de comparaison, le Pentium contient 3 millions de portes logiques). Les FPGA sont particulièrement intéressants pour nous pour plusieurs raisons. D'une part, ils sont constitués d'un assemblage régulier de structures simples (configurable logic blocs: CLB) sur laquelle s'implémentent très bien les circuits réguliers que nous allons produire. D'autre part, comme le montre la figure 2.2, l'intégration des FPGA augmente *plus vite* que la loi de Moore, ce qui signifie que la part de marché occupée par les FPGA va certainement augmenter. Enfin, qui dit programme dit compilateur, et ce compilateur devra pouvoir paralléliser des boucles pour tirer parti de la puissance de calcul disponible.

## Les réseaux systoliques

Un circuit systolique est un réseau composé d'un grand nombre de processeurs (la plupart identiques) simples. Les calculs se font sur des flots de donnés traversant le réseau, une horloge globale rythmant l'avancée des données à chaque top, comme l'impulsion cardiaque rythme l'avancée du sang dans le corps. Les processeurs (ou cellules) sont généralement connectés entre voisins et ont généralement un comportement que l'on peut voir comme "recevoir, calculer, envoyer", mais avec une granularité très fine (de l'ordre du mot de 16 bits), contrairement, par exemple, aux réseaux de processeurs à usage général.

Les réseaux systoliques furent originellement proposés par Kung [88, 89] pour des implémenter des algorithmes effectuant en série des calculs simples. En pipelinant les calculs, on obtient des architectures dont le temps de cycle est très petit. En introduisant le *retiming* [97], non seulement Leiserson proposait une modélisation d'une opération de conception de circuit qui allait avoir un énorme succès, mais en plus, il ouvrait la voie à un pan entier de recherche future: la conception automatique de réseaux systoliques.

Le modèle systolique eu un gros succès scientifique, d'innombrables publications de

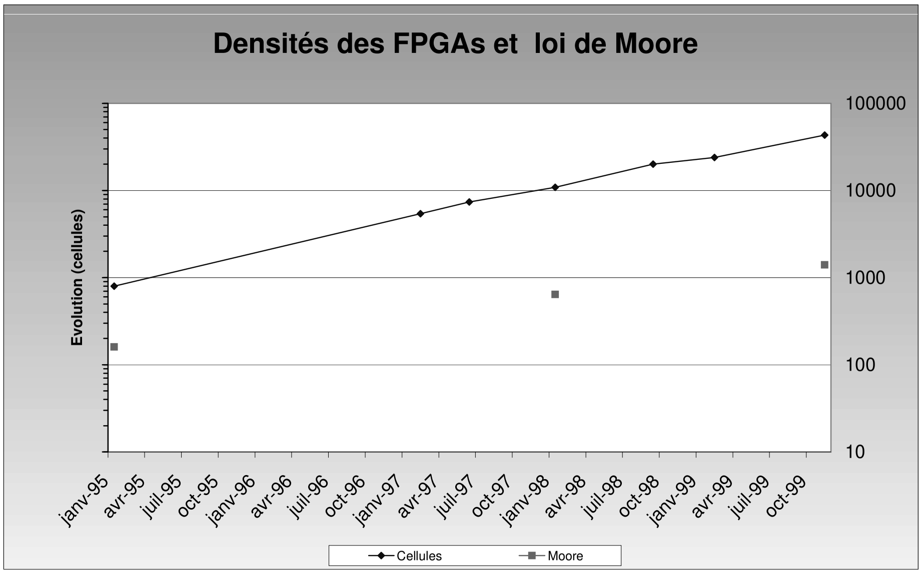


FIG. 2.2 – *Évolution de l'intégration des* FPGA *Xilinx (source: Xilinx)*

réseaux systoliques eurent lieux, en particulier pour les applications de traitement du signal et d'algèbre linéaire. beaucoup de ces réseaux restèrent sur le papier, car il reste un énorme travail à faire pour passer de la description RTL que l'on trouve dans ces papiers à une implémentation réelle. Aujourd'hui, Le modèle systolique à ses détracteurs et ses défenseurs, aussi virulents les uns que les autres. On peut toutefois constater que malgré leur relatif faible succès industriel, les architectures systoliques servent toujours de modèle à de nombreuses publications et implémentations et il semble que la technologie logicielle (compilateur/paralléliseur) et matérielle (architecture reconfigurable, plus de place sur le chip) nécessaire à son utilisation intensive soit en passe de devenir une réalité.

**La synthèse automatique de circuits**

En partant d'une description structurelle d'un circuit en VHDL, on peut obtenir un dessin de masque avec les outils commerciaux de CAO. Cette synthèse logique d'un circuit ne se fait pas automatiquement: le concepteur doit régler un certain nombre de paramètres pour la compilation, il prépare un “script”. En ce qui concerne la synthèse de plus haut niveau, de nombreux outils prétendent la faire (behavioral compiler de Synopsis, c2vhdl ESL), mais aucun des ces outils ne propose une véritable parallélisation. Nous touchons là un problème fondamental. La synthèse de haut niveau passe nécessairement par une phase de parallélisation et l'on sait que les constructeurs de machines parallèles se sont cassés les dents sur ce problème pendant des années (encore aujourd'hui, aucun compilateur commérciaux ne propose des parallélisation automatique efficace).

La communauté de synthèse automatique de circuits s'est retrouvée régulièrement à la conférence *Application Specific Array Processors* (ASAP). Aujourd'hui, en plus du travail présenté ici, ressortent du lot les outils développés par Imec, Philipps et Hewlett Packard (voir section 4.2). Mais il faut bien admettre que les techniques polyédriques ont plus de mal à pénétrer le monde des circuits que le monde de la compilation pour machines parallèles: cela est sans doute dû au fait que la formation des concepteurs de circuits a été effectuée dans une filière électronique alors que les gens travaillant sur le modèle polyédrique sont quasiment tous issus d'une filière informatique.

## 2.2 Le langage Alpha

Le langage Alpha est le socle grâce auquel toutes les expérimentations ont pu avoir lieu. C'est un langage fonctionnel à assignation unique qui a été conçu par Christophe Mauras [102] pour exprimer à la fois des algorithmes sous forme d'équations récurrentes et des représentations plus opérationnelles de ces algorithmes, comme une représentation espace/temps du réseau systolique exécutant l'algorithme ou une représentation RTL du matériel exécutant l'algorithme (sous ensemble Alphard).

Un système d'équations récurrentes affines ([82, 139]), est un système d'équations de la forme suivante:

$$z \in \mathcal{D} : U[z] = \mathcal{F}(\ldots, V[d(z)], \ldots) \tag{2.1}$$

$\mathcal{D}$ est l'ensemble des points entiers d'un *domaine* (union de polyèdres convexes dans le cas d'Alpha) de $\mathbb{Z}^n$ ($n$ est la dimension de $U$) et $d$ est une fonction affine de $z$. Ces équations définissent récursivement les valeurs des variables ($U$,$V$,...). Une telle équations est dite *uniforme* lorsque toutes les dépendance $d$ sont des translations.

Dans cette section nous proposons un rapide survol de la syntaxe du langage Alpha, nous présentons les extensions réalisés dans le cadre de ces recherches, puis nous insistons plus particulièrement sur les sous ensemble Alpha0 et AlpHard.

Le langage Alpha possède plusieurs caractéristiques importantes pour la compilation de circuits.

– Il possède une sémantique très bien définie par Christophe Mauras, ce qui permet de vérifier facilement que les transformations proposées sont correctes formellement. Cette sémantique solide permet aussi d'étendre le langage en toute sûreté.
– Chaque programme Alpha possède une forme normale simple. Ce processus de normalisation est très important car il permet de faire de nombreuses transformations source à source sans que la complexité du code augmente trop.
– Alpha est un langage à assignation unique. Cette propriété est primordiale pour la compilation/parallélisation car elle exhibe directement les vrais dépendances (comme on le verra plus loin, presque tous les outils apparentés à MMAlpha utilisent un format en assignation unique).
– Enfin, grâce aux différentes interprétations possibles mentionnées ci-dessus, il est possible de rester dans le même formalisme pour faire les différents raffinements sémantiques (notions d'opérateur physique, notion d'horloge, etc...) qui vont aboutir à la description d'une architecture et de son comportement. Le fait de rester dans le même formalisme permet de vérifier formellement que ces transformations sont correctes, alors que, si elles étaient faites lors d'une traduction entre deux formalismes, cette vérification serait quasiment impossible.

### 2.2.1 Le langage Alpha originel

Le langage Alpha a été conçu par Christophe Mauras pendant sa thèse [102]. Depuis, le langage noyau est resté le même, mais de nombreuses extensions ont été faites. Le langage Alpha est un langage *flot de données*, les variables qu'il manipule sont des tableaux de dimensions quelconques et qui ne sont pas nécessairement rectangulaires (en général le *domaine* de ces variables est un polyèdre convexe, mais ce peut être une union de polyèdres convexes, ou tout autre ensemble qui vérifie les propriétés énoncées par Mauras). Les variables sont déclarées de la manière suivante:

```
Y : {i,n | 1<=i<=10; 0<=n<=i}  of integer;
```

Ces variables sont utilisées dans des *expressions*. Ces expressions sont composées à partir de quelques opérateurs. Les opérateurs *point à point* (ou immobiles) s'appliquent sur chaque point du domaine, par exemple, l'addition: `Y+Z` additionne les valeurs de `Y` et de `Z` en chaque point du domaine résultant de l'intersection des domaines de `Y` et `Z`. Les opérateurs spatiaux agissent sur les domaines. Par exemple, la restriction `{i,n |i=1} : Y` restreint le domaine de l'expression (`Y`) au domaine (`{i,n | i=1}`). L'opérateur de dépendance (noté `.`) permet de *déplacer* les domaines, ainsi l'expression `Y.(i,n->i,n-1)` correspond à la variable `Y` que l'on aurait décalée suivant le vecteur $(0,1)$, le domaine de cette expression est donc

```
system fir      (x : {i | 1<=i<=100} of integer;
                 H : {n | 0<=n<=9} of integer)
       returns (res : {i | 10<=i<=100} of integer);
var
  Y : {i,n | 10<=i<=100; -1<=n<=9} of integer;
let
  Y = case
        {i,n | n=-1} : 0.(i,n->);
        {i,n | 0<=n} : Y.(i,n->i,n-1) +
                             H.(i,n->n) * x.(i,n->i-n);
      esac;
  res = Y.(i->i,9);
tel;
```

FIG. 2.3 – *Programme Alpha "originel" pour un filtre* FIR *en notation standard*

`{i,n | 1<=i<=10; 1<=n<=i+1}`. Enfin l'opérateur `case` permet de réunir des expressions ayant des domaines disjoints.

Les variables sont affectées par une seule équation du type

```
Y = expr
```

où `expr` est une expression Alpha. Une extension proposée par Doran Wilde est utilisée couramment: la notation tableau. Ainsi, l'équation `Z=Y.(i,n->i,n-1)` peut se noter de manière plus naturelle: `Z[i,n]=Y[i,n-1]` (on comprend mieux ainsi le calcul du domaine de l'expression `Y.(i,n->i,n-1)` ci dessus). Mais cette extension ne permet pas de réduire toutes les dépendances (elle peuvent être imbriquées, par exemple: `Y.(i,n->i-1,n).(i->i,0)` revient à `y.(i->i-1,0))` et surtout, on perd une propriété du langage: la localité référentielle. La propriété de localité référentielle peut être illustrée, par exemple, par le fait que les fonctions `(i,n->i-1,n)` et `(n,i->n-1,i)` sont les mêmes, les noms d'indices n'importent pas, seule leur position dans la liste d'indices compte. De fait, la notation tableau est une manière conviviale de visualiser des programmes Alpha, mais c'est bien un programme en notation standard qui est réellement manipulé. En figure 2.3 et 2.4, on peut voir un programme Alpha représentant le calcul d'une convolution (Finite Impulse Response filter: FIR) avec 10 coefficients `H` et 100 valeurs `x` entrantes, en notation standard et notation tableau.

La syntaxe du langage Alpha pose un réel problème, tant aux concepteurs de circuits qu'aux algorithmiciens des domaines applicatifs. De plus, l'expressivité du langage est restreinte, tous les algorithmes ne peuvent être traités. Cette expressivité est en passe d'augmenter (voir section 3.2.1), mais il est important de noter que ces restrictions correspondent aux restrictions de la méthodologie. On peut considérer grossièrement que l'on sait comment paralléliser automatiquement un algorithme sur un réseau régulier si et seulement si on peut l'exprimer en Alpha. D'autre part, on peut considérer Alpha comme un format interne pour la dérivation d'architecture (un peu comme le format Suif [57]). On peut en effet facilement afficher un programme Alpha normalisé avec une n'importe quelle autre syntaxe et on peut aussi traduire automatiquement un langage impératif (C, Fortran, Mat-

```
system fir      (x : {i | 1<=i<=100} of integer;
                 H : {n | 0<=n<=9} of integer)
       returns (res : {i | 10<=i<=100} of integer);
var
  Y : {i,n | 10<=i<=100; -1<=n<=9} of integer;
let
  Y[i,n] =
      case
        { | n=-1} : 0[];
        { | 0<=n} : Y[i,n-1] + H[n] * x[i-n];
      esac;
  res[i] = Y[i,9];
tel;
```

FIG. 2.4 – *Le programme de la figure 2.3 en notation tableau*

lab, voir section 3.2.4) en Alpha. Nous continuons de penser qu'un outil complet pour la compilation/parallélisation devra nécessairement utiliser un format interne très proche du langage Alpha pour le les transformations utilisant le modèle polyédrique.

### 2.2.2 Les extensions proposées

La syntaxe du langage Alpha a été présentée dans [163]. Nous présentons ici les modifications postérieures: la structuration du langage, l'extension aux domaines creux et d'autre extensions plus mineures. En même temps que la structuration du langage, Florent Dupont de Dinechin a formellement défini la manière d'utiliser les paramètres en Alpha. Cette extension, indispensable car elle permet de faire de la conception paramétrée et générique, avait été envisagée depuis longtemps ; elle est formellement décrite dans [49].

#### Structuration du langage

Ce travail a été réalisé en collaboration avec Florent Dupont de Dinechin et Patrice Quinton. Il s'agissait de proposer une structuration qui utilise toutes les possibilités du modèle polyédrique. La solution proposée, pour ce que l'on pourrait comparer à un simple appel procédural, encapsule un ensemble d'équations Alpha dans un *sous-système* (qui est en fait un simple système Alpha). L'appel de ce sous-système est sémantiquement équivalent aux équations le constituant. La syntaxe d'un tel appel dit *simple* est la suivant:

```
use  sousSyst.(N,M->M) (A,B) returns (C,D)
```

Dans l'appel ci-dessus, la fonction `(N,M->M)` correspond à l'*assignation des paramètres*. Dans ce cas précis le sous-système appelé a un seul paramètre alors que l'appelant en a deux. Ici, les variables `C` et `D` du système appelant doivent avoir exactement le même domaine que les variables de sortie du système `sousSyst`. Par contre en général, les entrées (ici `A` et `B`) peuvent être n'importe quelle expression Alpha pourvu que leur domaine englobe le domaine des arguments formels correspondants (variables d'entrée du système `sousSyst`). La notation tableau pour ce *use* est la suivant:

```
use  sousSyst[M] (A,B) returns (C,D)
```

Mais nous voudrions aussi pouvoir exprimer l'utilisation répétitive d'un sous-système. Par exemple, on peut décomposer un algorithme de Gauss sur une matrice $M \times M$ en $M$ étapes, toutes les étapes étant identiques, mais agissant sur une matrice de taille diminuant de $M \times M$ à $1 \times 1$. Pour cela on introduit la notion de domaine d'extension. Ce domaine peut être vu comme une indexation de l'instance du sous-système utilisé. Cette indexation peut se faire avec un entier (son numéro) qui parcourt un segment de $N$ mais aussi avec un vecteur qui parcourt un polyèdre en dimension quelconque.

L'exemple suivant est une implémentation possible de l'algorithme de Gauss. L'algorithme de la figure 2.5 consiste à appliquer `N` fois le sous-système `ZeroColumn` qui modifie une sous-matrice (`N-K`)×(`N-K`), `K` variant de `1` à `N`. La figure 2.6 représente la partie de la matrice calculée par le sous système `ZeroColumn` pour `K`=5 et `N`=9. On notera, dans le système `Gauss`, la manière dont on paramètre l'appel de `ZeroColumn` par l'indice d'extension `k`. On notera aussi la correspondance entre les indices des variables de l'appelant (`Ak[i,j,k]`: 3 indices), des variables de l'appelé (`Ak[i,j]`: 2 indices), et du domaine d'extension (`k`: 1 indice).

La structuration du langage est expliquée extensivement dans [49] et a été publiée dans [39]. Les extensions de MMAlpha nécessaires à la manipulation de programmes Alpha structurés ont été réalisées, jusqu'à un certain point car certaines se révèlent relativement complexes. Par exemple, ce n'est que récemment que l'on a implémenté l'ordonnancement structuré (voir section 3.1.2) et le problème de la génération de code structuré n'est toujours pas résolu. Les programmes structurés sont aujourd'hui couramment utilisés dans le système MMAlpha. En pratique, la structuration a grandement facilité l'écriture d'applications plus grosses comme la SVD [145] ou le filtre de Kalman [113], elle a aussi permis la définition d'AlpHard, et elle ouvre les portes à la conception structurée et à la conception conjointe.

**Les domaines creux**

Les domaines de définition des variables Alpha sont des polyèdres convexes. Ces polyèdres rationnels sont utilisés pour décrire l'ensemble des points entiers qu'ils contiennent (dans la suite le terme polyèdre désignera l'ensemble des points à coordonnées entières qu'il contient). On peut donc exprimer en Alpha des nids de boucles (pas nécessairement parfaits) dont le pas est 1. L'espace d'itération d'une boucle dont le pas est différent de 1 n'est pas un polyèdre mais un $\mathbb{Z}$-polyèdre (ou polyèdre creux, voir section 3.2.1 pour des exemples).

Les $\mathbb{Z}$-polyèdres ont été introduits par Corinne Ancourt [5] ; parallèlement Teich et Thiele introduisaient les LBL [156][2]. Il est apparu que les LBL ne pouvaient pas être utilisés comme domaines pour Alpha car certaines opérations ne pouvaient être faites. Les

2. La notation LBL n'est pas très heureuse car elle signifie *linearly bounded lattice*, c'est à dire exactement ce que sont les $\mathbb{Z}$-polyèdres: des treillis (ou réseaux) bornés linéairement. Dans la dénomination introduite par Thiele, un LBL est l'image d'un polyèdre par une fonction affine. La terminologie étant maintenant relativement figée, nous conservons la définition de Thiele pour les LBL.

```
system ZeroColumn: {N,K| 1<=K<N}
          (A : {i,j | 1<=i,j<=N; i<=j} |
                    {i,j | 1<=i,j<=N; j>=K} of real )
  returns (Ar: {i,j | 1<=i,j<=N; i<=j} |
                    {i,j | 1<=i,j<=N; j>K}  of real );
   let
      Ar[i,j] =
          case
           {| i<=K}       : A[i,j];
           {| i>K; j>K}   : A[i,j] - A[K,j]*A[i,K]/A[K,K];
          esac;
   tel;
```

```
system Gauss: {N | N>1} (A : {i,j | 1<=i,j<=N} of real)
                returns (T : {i,j | 1<=i<=j<=N} of real);
    var  Ak : {i,j,k | 1<=i,j,k<=N; i<=j} |
                 {i,j,k | 1<=i,j,k<=N; j>=k}  of real;
         Ak1: {i,j,k | 1<=i,j<=N; 1<=k<N; i<=j} |
                 {i,j,k | 1<=i,j,k<=N; j>k} of real;
    let
       use {k| 1<=k<N} ZeroColumn[N,k] (Ak) returns (Ak1);
       Ak[i,j,k] = case
                     {| k=1} : A[i,j];
                     {| k>1} : Ak1[i,j,k-1];
                   esac;
      T[i,j] = Ak[i,j,N];
    tel;
```

FIG. 2.5 – *Algorithme de Gauss exprimé en Alpha structuré (sans stratégie de îvot). La portion de la matrice* `A` *passé en paramètre au système* `ZeroColumn` *est indiquée en figure 2.6.*

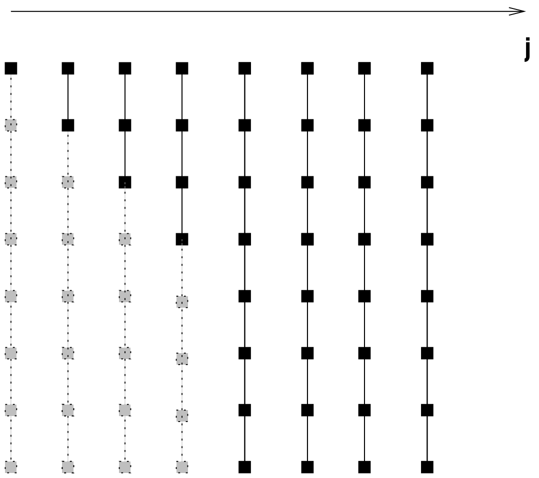


FIG. 2.6 – *portion de la matrice mis à jour par le système* `ZeroColumn` *de la figure 2.5 pour* `K=5` *et* `N=9`.

$\mathbb{Z}$-polyèdres, en revanche, peuvent être des domaines Alpha bien que leur manipulation soit un peu plus complexe que celle des polyèdres. Les $\mathbb{Z}$-polyèdres sont aussi indispensables pour exprimer qu'un signal booléen est à 1 un top sur 3, par exemple, ce qui est extrêmement utile pour contrôler l'exécution d'un calcul en matériel.

Nous avons étudié l'extension du langage Alpha aux $\mathbb{Z}$-polyèdres dans [130] dans le cadre de la manipulation de signaux multi-horloges. L'extension proposée autorise des équations de la forme suivante:

```
A = case
    {4i | 1<=2i<=N} : B.(i->i-4);
    {4i+2 | 0<=2i<=N-1} : B.(i->i-2)
  esac;
```

Dans cet exemple, la variable `A` est définie sur `{2i | 1<=i <=N}` et `B` sur, au moins, `{2i | 0<= i <=N-1}`. La notation tableau pour une telle équation est

```
A[j] = case
       {4i | 1<=2i<=N} : B[j-4];
       {4i+2 | 0<=2i<=N-1} : B[j-2];
    esac;
```

Une grosse partie du travail résidait dans la production d'une bibliothèque de calcul sur les $\mathbb{Z}$-polyèdres. Nous avons étudié le calcul sur les $\mathbb{Z}$-polyèdres dans [131] ; l'extension de la librairie polyédrique Polylib que nous avons réalisée [117] pour qu'elle manipule des $\mathbb{Z}$-polyèdres est détaillée en section 3.2.1. L'extension des parsers a été réalisée, sans avoir vraiment été testée car il reste un gros travail: celui de la mise à jour de toutes les fonctions de MMAlpha pour rendre opérationnelle l'utilisation des $\mathbb{Z}$-polyèdres dans MMAlpha.

**Autres modifications**

Sanjay Rajopadhye a proposé l'extension du langage aux expressions d'index, c'est à dire aux équations de la forme: `A[i,j]=i+j`. Cette extension a été intégrée aux parsers et à toutes les fonctions de base de MMAlpha (mais on ne sait pas comment réaliser le matériel correspondant). Une extension des types de Alpha (jusqu'alors limités aux types booléen, entier ou réel) est en cours d'étude pour pouvoir introduire la notion de précision finie et de représentation des nombres. L'idée est de remonter aussi haut que possible dans le processus de conception, le traitement qui consiste à déterminer les tailles en bits utilisées pour chaque signaux, ceci pour simplifier au maximum ce traitement. Nous espérons ainsi proposer une généralisation des techniques de passage en précision finie aux cas des programmes contenant des boucles imbriquées (voir section 3.2.2).

### 2.2.3 Alpha0 et AlpHard

Alpha0 et AlpHard sont des sous-ensembles de Alpha pour lesquels on peut interpréter le programme comme une description d'architecture au niveau RTL. Les programmes Alpha0 et AlpHard résultent du processus de dérivation décrit dans le chapitre suivant, ils ne sont généralement pas écrit à la main. La conception pour un système donné doit aboutir à un programme Alpha0. AlpHard correspond à la structuration de ce système, c'est à dire à une librairie qui distingue chaque processeurs de l'architecture comme un système indépendant. La définition de tels formats intermédiaires est très importante car elle permet d'avoir une traduction vers VHDL quasi immédiate, en tout cas sans changement sémantique important. Il s'agit, non seulement de définir la syntaxe des composant matériels qui vont constituer le circuit (registres, opérateurs, contrôleur, hiérarchie), mais aussi de définir précisément leur sémantique pour pouvoir les traduire correctement vers un autre format de description d'architecture.

Alpha0 fut d'abord défini par Catherine Dezan dans sa thèse [42]. Depuis de nombreuses évolutions ont amené à une redéfinition (en collaboration avec Patricia Le Moënner et Florent Dupont de Dinechin): Alpha0v2 (que nous appelons simplement Alpha0 dans la suite). Alpha0 décrit dans un seul système la topologie du réseau, la structure matérielle de chaque processeur et le fonctionnement global de l'architecture au cours du temps. Les indices des variables Alpha sont interprétés soit comme le temps (généralement noté `t`, on suppose que l'architecture résultante est cadencée par une horloge globale), soit comme un numéro de processeur dans une certaine dimension (les grilles, par exemple, ont deux indices d'espace). Les variables Alpha correspondent à la notion de signaux en VHDL, on peut les voir comme des fils en matériels. Par exemple, l'équation suivante:

```
A[t,p] = B[t-1,p]+C[t,p-1]
```

correspond à une addition dans un réseau linéaire (unidimensionnel). Dans un processeur `p`, l'addition a lieu entre le signal `B` retardé par un registre et le signal `C` généré dans le processeur précédent (`p-1`). Dans Alpha0, cette équation serait en réalité décomposée en trois équations, une pour le registre, une pour la connexion entre deux processeurs et une pour l'addition.

La structuration d'Alpha a permis la définition d'AlpHard. AlpHard a été défini par Patricia Le Moënner dans sa thèse [94], il a été présenté dans [93]. Un programme AlpHard correspond à la même description architecturale que le programme Alpha0 dont il provient,

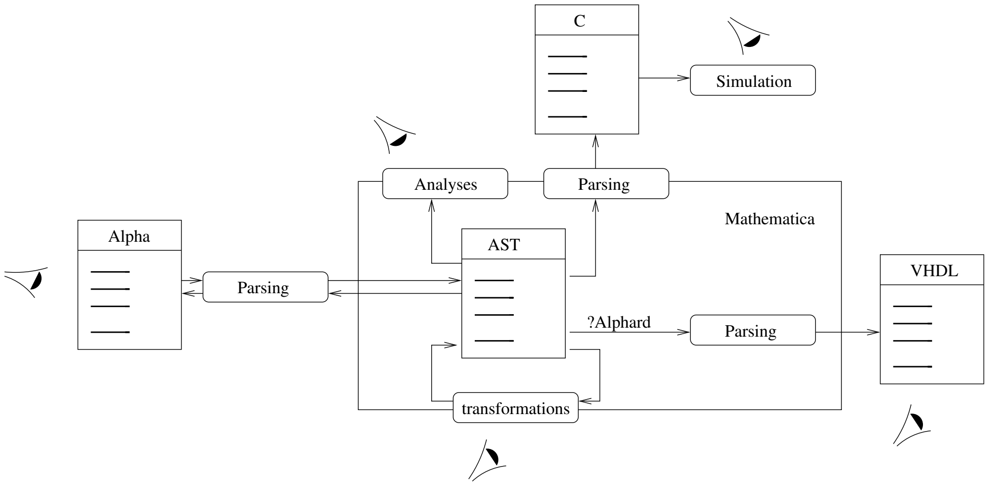


FIG. 2.7 – *le système* MMA*lpha vu par l'utilisateur, l'oeil symbolise l'action et/ou l'observation de l'utilisateur.*

mais la structure du circuit (cellules, réseau, contrôleur) apparaît à travers la structure du programme AlpHard. Ceci permet de faire une traduction directe en VHDL, en faisant correspondre à un système Alpha, une entité VHDL. En pratique, nous avons fixé simultanément le format Alpha0 le format AlpHard et la traduction vers VHDL. Ensuite nous avons avons écrit les fonctions permettant de générer du Alpha0 puis de le traduire automatiquement en AlpHard. Ces fonctions font maintenant partie intégrante de toute conception avec MMAlpha (c.f. chapitre 3). Des exemples de programmes Alpha0 et AlpHard sont proposés en annexe, section A.2 et A.3

## 2.3 Le système MMAlpha

Le système MMAlpha est l'environnement de manipulation des programmes Alpha. Historiquement, après la conception du langage, les environnements Diastol citeDiastol et Alpha du Centaur [43] ont vu le jour. L'environnement MMAlpha a émergé lorsque Patrice Quinton a décidé de reprendre l'implémentation sous Mathematica. Dans MMAlpha, les transformations de programmes Alpha se font par la manipulation symbolique de l'*arbre de syntaxe abstrait* (Abstract Syntax Tree: AST) qui est une structure symbolique interne à Mathematica. Les programmes de MMAlpha sont donc des paquetages Mathematica qui définissent les fonctions agissant sur l'AST.

La figure 2.7 montre l'architecture du système MMAlpha vue par l'utilisateur. Doran Wilde a mis en place tout un système de parser et de pretty-printer ainsi qu'un générateur de C qui permettait de compiler le code Alpha. Il a aussi écrit la librairie polyédrique dont le noyau était l'algorithme de Chernikova programmé par Hervé Leverge. Ces composants sont toujours essentiels dans MMAlpha. J'ai pris en main la maintenance du logiciel dans lequel ont été ajoutées petit à petit toutes les fonctions permettant la conception de circuits.

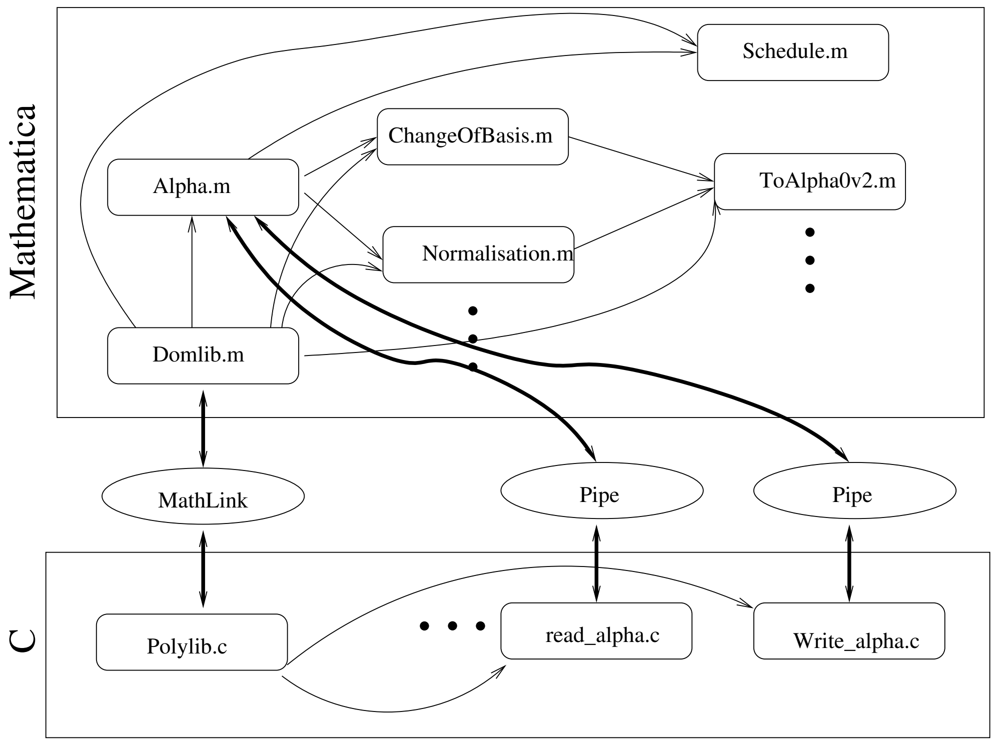


FIG. 2.8 – *Architecture logicielle du système* MMA*lpha. Les flèches minces symbolisent les dépendances entre les programmes (seuls quelques-uns ont été représentés, la liste complète peut être trouvé en annexe B), les flèches grasses représentent les communications de données entre les programmes écrits dans différents langages.*

La figure 2.8 montre l'architecture interne du logiciel. Polylib est les noyau sur lequel s'appuient les manipulations polyédriques. L'efficacité des transformations de MMAlpha provient en grande partie de l'implémentation optimisée de l'algorithme de Chernikova réalisée par Hervé Leverge [95]. L'implémentation de Doran Wilde [161] est actuellement utilisée dans plusieurs outils de parallélisation. Les extension de Strasbourg (Polynômes d'Ehrhart [28]) et de Rennes ($\mathbb{Z}$-polyèdres voir section 3.2.1) en font un outil très puissant.

Dans ce document, nous ne parlons pas des paquetages de “bas niveau” qui ont été programmés dans la jeunesse de MMAlpha. Ces paquetages contiennent les fonctions de base qui vont être utilisées par les fonctions “haut niveau”. Cela inclut, le paquetage `Alpha` contenant toutes les fonctions initiales de l'environnement, le paquetage `Domlib` effectuant l'interface avec `Polylib` pour les calculs sur les domaines, les paquetages contenant les fonctions de base de manipulation de l'AST (paquetages `Table`, `Matrix`, `ChangeOfBasis`, `Cut`, `Normalization`, `Substitution`, pour plus de détail concernant ces paquetages, on se reportera à citetutorial99-alpha)

Plutôt qu'un véritable compilateur/paralléliseur, MMAlpha est une boîte à outils pour la compilation/parallélisation de circuits, c'est-à-dire que la conception est guidée par l'utilisateur qui indique quelle transformation utiliser et de quelle manière. Même si la méthodologie globale de conception avec MMAlpha est relativement figée (voir chapitre 3), l'espace des conceptions possibles est si vaste qu'il nous semble illusoire de vouloir tout automatiser aujourd'hui. MMAlpha a servi de plate-forme logicielle pour les thèse de Florent Dupont de Dinechin [49], Patrice Le Moënner [94], Sophie Robert [145], Irina Smarandache citeIrina98, Fabien Quilleré (en cours d'écriture) et Anne-Claire Guillou (en cours) ainsi que pour tout le travail de recherche de la composante *synthèse de très haut niveau* de l'équipe Cosi. En annexe B, on trouvera la liste des paquetage de MMAlpha ainsi que la liste des personnes ayant contribué à leur développement.

# Chapitre 3

# Conception de circuits avec MMAlpha

Ce chapitre rassemble tous les travaux directement liés à la conception de circuits avec MMAlpha. Ces travaux sont regroupés en deux catégories: ceux liés aux transformations qui fonctionnent depuis assez longtemps pour être considérées comme étant *standard* dans la méthodologie, et ceux liés aux transformations qui ne sont pas complètement (ou pas du tout) implémentées, mais qui ont été étudiée dans le cadre d'une éventuelle intégration au processus de synthèse avec MMAlpha J'ai regroupé en section 3.1 et 3.2 l'ensemble des transformations qui constituerait un outil très complet pour la compilation/parallélisation de circuits. Un exemple de conception d'un circuit pour la comparaison de séquences génétiques est présenté en section 3.3, cette conception étant réalisée essentiellement de manière standard, c'est à dire avec la méthodologie décrite en section 3.1.

## 3.1 Méthodologie standard avec MMAlpha

Cette section décrit les grandes étapes de la compilation/parallélisation de circuits avec MMAlpha. Ces étapes correspondent à la méthodologie "classique" de conception de réseaux systoliques. L'application de cette méthodologie à un programme Alpha pour générer du code pour le placeur MadMacs [120] avait été proposé avant mon arrivée à Rennes par Doran Wilde et Oumarou Sié dans [162]. Les étapes présentées ici reprennent celles proposées dans [162], mais sont passées d'une conception adaptée à un algorithme à une méthodologie générale pour tout programme systolisable.

La méthodologie standard pour compiler/paralléliser un programme Alpha peut être décomposée en trois points : uniformisation (section 3.1.1), ordonnancement/allocation (section 3.1.2), génération RTL(section 3.1.3). En plus des ces transformations, il y a deux traitements transversaux: l'analyse et la simulation qui peuvent être réalisés à tout moment pendant la conception (section 3.1.4). Enfin, nous décrivons comment synthétiser le VHDL généré par MMAlpha et obtenir des circuits.

### 3.1.1 Uniformisation

Le langage Alpha permet d'exprimer des algorithmes sous la forme d'équations récurrentes affines. La méthodologie systolique permet essentiellement de produire des architectures à partir d'équations récurrentes uniformes. Bien qu'il soit aujourd'hui possible de considérer certaines dépendances affines dans le processus de conception (le fait d'avoir une diffusion peut être implémenté en matériel), la phase d'uniformisation reste une des étapes fondamentales du processus. Elle consiste en la transformation d'un programme Alpha quelconque en un programme Alpha uniforme (dont toutes les dépendances sont uniformes) équivalent.

L'uniformisation a beaucoup été étudiée du point de vue théorique car l'efficacité de l'architecture générée dépend grandement de la qualité de l'uniformisation. Citons Quinton et Van Dongen [48], Rao [139], Rajopadhye [134], Roychowdury [146], Megson [105] pour les plus connus. Cependant, peu d'implémentations réelles ont eu lieu du fait de la difficulté du traitement. Nous avons, en collaboration avec Manjunathaia, réalisé un paquetage pour l'automatisation du processus d'uniformisation (paquetage `Uniformisation`, [101]), mais le problème reste délicat car la procédure automatique choisit forcément un chemin dans l'espace de conception, et nous n'avons pas de garantie sur la qualité de l'architecture qui en résultera. Ceci pourra être amélioré par l'expérience pour essayer de détecter quelle est "intuitivement" la meilleure manière d'uniformiser un système.

La phase d'uniformisation peut être décomposée en deux étapes: une étape d'*alignement* (passage en forme complètement indexée) et une étape de localisation (généralement référencée elle-même comme l'étape d'uniformisation) où l'on rend effectivement toutes les dépendances uniformes.

#### Alignement

L'alignement consiste à plonger toutes le variables dans un espace commun, celui de la variable possédant le plus grand nombre d'indices. Si l'on dispose d'un programme Alpha manipulant un vecteur `V` et une matrice `M`, on aura un tableau à une dimension et un tableau à deux dimensions. supposons qu'on ait l'affectation:

```
M[i,j]=V[i]
```

L'alignement consistera alors a faire un changement de base généralisé sur `V` de manière à ce qu'il devienne une matrice "plate". Ainsi, effectuer le changement de base par la fonction `(i->i,0)` correspond à faire un alignement du vecteur `V` sur la colonne 0 de la matrice `M`. Le nouveau programme contiendra l'équation

```
M[i,j]= V[i,0]
```

Ce choix peux avoir une grosse influence sur la qualité du circuit résultant de la conception. Dans MMAlpha, cette transformation a reçu relativement peu d'attention car elle est généralement effectuée *en même temps* que le pipeline (voir plus loin). Cela ne posait pas de problème fondamental car le concepteur *voit* généralement très bien comment aligner les variables. Par contre, dans le cadre d'un processus de compilation, cette transformation devra être faite automatiquement. Ce problème est bien connu dans le domaine de la parallélisation automatique où il a beaucoup été étudié car il influence beaucoup les performances [100, 45]. Cette transformation a été aussi étudiée à Reading [105] pour la production de réseaux systoliques. Dans MMAlpha cette transformation peut être réalisée, dans les cas simples, par le paquetage `Uniformisation`. Le passage d'un programme Alpha en forme complètement indéxée n'est pas automatisé actuellement. En pratique la plupart des programmes qui doivent être synthétisés correspondent à cette utilisation simple, c'est à dire qu'il n'y a qu'un seul choix évident pour l'alignement des variables.

### Localisation

La phase de localisation est relativement complexe. On part d'un programme pouvant être ordonnancé et on traite séquentiellement toutes les dépendances devant être uniformisées. On peut distinguer trois manières d'uniformiser une dépendance : le pipeline, le routage et l'uniformisation contextuelle.

Le pipeline correspond à la transformation d'une diffusion en un pipeline des valeurs le long d'une direction particulière. Ainsi, dans l'exemple ci-dessus, la diffusion de `V[i,0]` à tous les points `(i,j)` du domaine de `M` peut être transformée en un pipeline de cette valeur aux points `(i,1)` puis `(i,2)` etc. Cette transformation peut être réalisée grâce à la fonction `pipeall` de MMAlpha si l'utilisateur désire indiquer les paramètres, ou par la fonction `uniformize` s'il préfère réaliser le traitement automatiquement. Les dépendances pouvant être uniformisées par cette transformation peuvent être détectées automatiquement [101], et l'ensemble des directions possibles pour le pipeline (c'est à dire qui conservent le fait que le programme a un ordonnancement) peuvent être obtenues automatiquement. Dans le cas où il y a plusieurs choix possibles, la fonction `uniformize` choisit au hasard (il n'y a pas de fonction coût implémentée).

Le routage correspond à un déplacement de valeurs à travers le domaine. Cela est possible si la *vraie* dimension de l'expression déplacée est inférieure à la dimension totale de l'espace ; cela correspond généralement à une l'uniformisation d'une dépendance qui n'est pas une diffusion. Par exemple, supposons que nous ayons une équation du type:

```
M[i,j]= case
        {| j=0}: V[i,N];
        {| j>0}: ...
      esac;
```

dans cet exemple, nous avons aligné le vecteur `V` avec la colonne `N` de la matrice `M`, il est possible d'amener les valeurs `V[i,N]` aux points `(i,0)` en passant par les points `(i,N-1)`, puis `(i,N-2)` etc. Notons qu'ici, nous aurions pu utiliser un changement de base, mais dans MMAlpha, il aurait fallu repasser par un programme dans lequel `V` aurait un seul indice (cela montre bien que ce changement de base correspond à une modification de l'alignement et non une localisation). Le routage, comme le pipeline, peut être effectué automatiquement (fonction `uniformize`) ou guidé par l'utilisateur (`pipeIO`). Il y a en général beaucoup de possibilités pour router une expression, la minimisation du nombre de registres résultant d'une telle transformation pourrait être une manière de départager les différentes possibilités. Le routage est aussi utilisé après l'ordonnancement pour *amener* les entrées/sorties du réseau systolique vers les processeurs du bord du réseau.

L'uniformisation contextuelle consiste simplement à repérer certaines dépendances uniformes cachées, du fait que dans un certain contexte, il peut y avoir plusieurs manières d'exprimer une même dépendance.

Dans le rapport [101], nous avons essayé de rassembler les travaux de plusieurs personnes (essentiellement les travaux de [48, 105, 135, 146]) dans un même formalisme, nous avons proposé et expérimenté l'uniformisation automatique sur quelques programmes (Gauss, le filtre ARMA). Nous ne traitons pas le problème de l'alignement (le programme traité doit être sous forme complètement indexée). Aujourd'hui, nous utilisons plus volontiers l'uniformisation dirigée par le concepteur, mais dans le cadre, par exemple, d'un compilateur pour FPGA, ce travail sera une base importante pour tester différentes stratégies d'uniformisation.

### 3.1.2 Ordonnancement

L'ordonnancement est sans aucun doute le processus central de la méthodologie présentée ici. C'est lui qui utilise le plus de "valeur ajoutée" (c'est à dire de travaux de recherche antérieurs), et de la qualité de son résultat dépend grandement la qualité de l'architecture finale. Nous avons passé beaucoup de temps à implémenter et à réfléchir sur ce traitement pour l'adapter au mieux aux besoins de la compilation de circuits.

Ordonnancer un programme Alpha consiste à trouver une date d'exécution pour chaque calcul (c'est à dire chaque variable puisque Alpha est en assignation unique). Comme les variables sont indexées par les indices des domaines, les fonctions de temps sont généralement des fonctions affines des indices (ou des vecteurs de fonctions affines pour l'ordonnancement multidimensionnel [1]). L'ordonnancement le plus général que l'on puisse théoriquement calculer est l'ordonnancement affine par morceaux : dans ce cas, la fonction affine dépend de la variable calculée *et* du point calculé. Ces ordonnancements sont inutilisables en Alpha ; nous avons donc permis de calculer des fonctions de temps *multidimensionnelles affines par variable*. Dans la méthodologie de conception, on va ordonnancer un programme uniforme, problème beaucoup plus simple en soi, et pour conserver la localité induite par l'uniformisation, on utilise un ordonnancement *affine par variable de partie linéaire unique* (toutes les fonctions de temps des variables ont une même partie linéaire ; ce type d'ordonnancement est aussi appelé *shifted linear schedule*). L'outil d'ordonnancement est aussi utilisé pour avoir un aperçu du parallélisme disponible dans les grosses applications.

Cette section devrait aussi traiter de l'allocation. En fait, dans l'état actuel de l'implémentation, la fonction d'allocation doit être une projection sur un espace de dimension `n-1` (si les variables ont `n` indices) qui doit en plus être telle que la matrice composée de la fonction de temps et de la fonction d'allocation soit unimodulaire. Pour l'instant, cette complétion est faite, généralement automatiquement, après le processus d'ordonnancement. La phase d'allocation pourra être plus précisément étudiée lorsque les changements de base non unimodulaires seront autorisés (grâce aux $\mathbb{Z}$-polyèdres, voir section 3.2.1) ou lorsque le partitionnement sera possible (voir section 3.2.6). Sur toutes les applications que nous traitons (applications qui *a priori* sont adaptées à une implémentation sur réseau systolique) l'allocation par projection était celle choisie par des concepteurs d'architectures (voir par exemple [154, 158, 116, **?**] pour le DLMS).

Historiquement, la communauté de réseaux systoliques savait comment ordonnancer

1. Dans le cas d'ordonnancement multidimensionnel, je parlerai d'*heures* et de *minutes* en ayant à l'esprit le fait que c'est un système dans lequel chaque heure n'a pas forcément le même nombre de minutes.

les équation récurrentes uniformes (avec par exemple, la fameuse méthode de l'hyperplan de Lamport [91]). Pour ordonnancer tous les systèmes d'équations récurrentes affines, trois méthodes émergèrent à peu près simultanément des travaux de Feautrier [64, 65], Quinton et al. [103] et Darte [33]. Ces travaux sont étudiés dans le contexte de la parallélisation automatique dans [38]. Nous avons ici une approche doublement orthogonale: d'une part il s'agit d'ordonnancement pour la synthèse de circuits, et d'autre part on s'intéresse à la production de résultats efficaces sur quelques algorithmes clés plutôt qu'à la puissance théorique des différents algorithmes d'ordonnancement. Toutes ces méthodes utilisent le théorème de la dualité [149], soit sous la forme du théorème fondamentale de la dualité (Feautrier et Darte), soit sous la forme de la décomposition des polyèdres en sommet, rayons et lignes (Quinton). Le principe général est de traduire les contraintes que doit respecter la fonction de temps en un programme linéaire qui peut être alors résolu avec un solveur classique.

Nous avons, en collaboration avec Stephane Balev, Patrice Quinton et Sanjay Rajopadhye, essayé de comparer les approches de Feautrier [64, 65] et Quinton [103]. Nous avons évalué (sur deux implémentations comparables réalisées par Stephane Balev) les temps de calcul et tailles des problèmes [7]. Il semble que la méthode proposée par Feautrier soit légèrement plus performante *en pratique* (c'est-à-dire sur les quelques gros systèmes d'équations récurrentes que nous avions à disposition). Les complexités effective de chaque algorithme sont difficiles à comparer car les deux méthodes s'appuient sur des représentations duales des polyèdres. Avant cela, nous avions choisi d'implémenter la méthode de Paul Feautrier, puis Patrice Quinton avait réalisé une autre implémentation se basant sur [103].

### Implémentation de l'ordonnancement cyclique

L'utilisation intensive des ces implémentations nous a appris beaucoup de choses. La plus importante est sans doute le fait que le processus d'ordonnancement doit être, lui aussi, guidé par l'utilisateur. Invariablement, la réaction de l'utilisateur lors d'un premier test de la fonction `schedule` sera: “Ce n'est pas l'ordonnancement que je veux”. Cette réaction ne signifie pas la fonction de coût n'est pas bonne, elle signifie simplement que la fonction de coût *est* l'utilisateur lui-même. Je suis intimement persuadé qu'aucune fonction de coût ne saura satisfaire l'ensemble des utilisateurs potentiels de MMAlpha. J'en veux pour preuve la foison d'ordonnanceurs apparus dans l'équipe: chaque utilisateur intensif d'ordonnancement a développé le sien (Patrice Quinton pour tester la méthode des sommets, Fabien Quilleré pour la génération de code efficace, Stefan Balev pour comparer les différentes méthodes, et moi pour avoir un outil suffisamment flexible). Cette constatation m'a amené à paramétrer tout ce qui pouvait l'être de manière à donner à l'utilisateur le maximum de contrôle sur l'ordonnancement à obtenir.

Les options qui me semblent indispensables pour paramétrer l'ordonnancement sont `durations` et `addConstraints`. L'option `durations` permet à l'utilisateur de préciser explicitement, pour chaque variable `A`, quel est la “durée” (c'est à dire le nombre de top d'horloge) nécessaire pour le calcul d'un élément de `A`. Dans la méthode de Feautrier destinée à la programmation de machines parallèles, chaque instruction dure 1 par défaut. Dans le cas de la synthèse de circuits, le problème est plus délicat. Schématiquement, une instruction dure 1 si son résultat est stocké dans un registre et 0 sinon. On peut aussi imaginer qu'une instruction dure plus que 1 (2, 3 etc...), ce qui générera plusieurs registres pour stocker la valeur, registres qui pourront être utilisés pour faire du retiming après coup. Le fait de préciser la durée des opérations revient donc à placer explicitement les registres dans l'architecture résultante. Ce processus de placement des registres doit essentiellement assurer la propriété mise en avant par Ipsen et Delosme [40] : le graphe de dépendance réduit valué par ces poids de durée ne doit pas contenir de circuit de poids nul. La fonction de coût proposée par ces derniers pour assurer cette propriété n'a pas été mise en oeuvre, c'est donc à l'utilisateur de repérer les cycles et de positionner les registres.

L'option `addConstraints` permet d'ajouter des contraintes (linéaires) sur les coefficients des fonctions de temps. Elle est utilisée pour imposer des contraintes sur les dates de disponibilité des variables d'entrée (donc dans le processus d'ordonnancement structuré, voir plus loin) mais aussi lorsque l'utilisateur ne parvient pas à obtenir exactement l'ordonnancement voulu de manière automatique. Le contrôle des solutions obtenues pour l'ordonnancement multidimensionnel est aussi extrêmement délicat. Le fait qu'une dépendance soit satisfaite à un niveau ou à un autre peut énormément changer les performances du programme résultant. L'approche de Paul Feautrier [64] assure uniquement la minimisation du nombre de dimensions de l'ordonnancement résultant. Pour aider un tel contrôle, le fait de pouvoir imposer des fonctions de temps se révèle extrêmement utile.

La puissance de logiciel MMAlpha a sensiblement augmenté lors de l'implémentation de l'ordonnancement multidimensionnel d'équations récurrentes affines. En effet, s'il est presque toujours possible de percevoir intuitivement si un programme admet un ordonnancement linéaire, l'existence d'un ordonnancement multidimensionnel ou quadratique est beaucoup moins facile à détecter. Les résultats donnés par l'ordonnancement multidimensionnel ont, en particulier, été utilisés dans le thèse de Sophie Robert pour l'algorithme de la SVD [145] et pour la conception d'une architecture systolique pour le filtrage de Kalman [114, 112]. Encore aujourd'hui, MMAlpha est un des seul outils permettant de prototyper rapidement le parallélisme potentiel d'une application.

#### Ordonnancement structuré

Avec la structuration du langage Alpha la question de l'ordonnancement de programmes structurés s'est posé. À l'époque, la seule possibilité était d'*aplatir* le programme structuré puis de l'ordonnancer (ordonnancement dit *à plat*). On arrive très vite à des programmes linéaires trop gros (par exemple [145, 7], 3000 variables et 4000 contraintes pour la SVD et plusieurs heures de calculs pour la résolution [2]).

Il s'agissait donc de définir ce que l'on appelait ordonnancement structuré, puis de proposer une méthode pour calculer efficacement un tel ordonnancement. Ce travail a été présenté dans [52, 53]. L'ordonnancement d'un programme Alpha `l1` utilisant le sous système

2. Il est d'ailleurs intéressant de constater que les programmes linéaires générés par la méthode de Feautrier sont résolus en nombres entiers, non seulement parce que c'était nécessaire (une solution rationnelle était théoriquement valide mais inutilisable en Alpha) mais aussi parce que la résolution en rationnel prenait sensiblement le même temps. Cela signifie que les programmes linéaires générés ont une structure particulière et que le polytope des contraintes est sans doute très proche de son enveloppe entière. On pourrait, sans doute démontrer cela dans des cas simples (dépendances uniformes avec coefficients +1, 0 ou -1), mais je ne vois pas comment le montrer dans tous les cas, or c'est bien ce qui arrive en pratique.

s1 donne les dates d'exécution des variables de l1. Cet ordonnancement est dit *structuré* s'il est possible de trouver un ordonnancement de s1 qui puisse être utilisé pour ordonnancer *toutes* les instances d'utilisation de s1 dans l1 de manière compatible avec l'ordonnancement de l1. La meilleur manière d'obtenir un tel ordonnancement serait sans doute de pouvoir garder l'ordonnancement de s1 paramétré par les dates des entrées et des sorties, en d'autres termes, de manipuler un polytope d'ordonnancements possibles pour s1 plutôt qu'un ordonnancement particulier. Cette approche n'a pas pu être testée car Polylib n'arrivait pas à manipuler les gros polytopes ainsi générés (bogue de gestion mémoire). D'autre part, on "triche" un peu car on ne perd pas d'information en faisant remonter l'information d'ordonnancement de s1 vers l1, et la structuration permet justement de simplifier les traitements en cachant des informations derrière un appel.

Nous avons naturellement généralisé la notion de graphe de dépendance (réduit) utilisé pour les équations récurrentes à la notion de graphe de dépendance structuré, dans lequel un appel à s1 est représenté par un arc particulier possédant éventuellement plusieurs sources et plusieurs buts (variables entrées et sorties de l'appel) et dont la valuation contient le domaine d'extension de l'appel ainsi que la fonction d'assignation des paramètres. L'ordonnancement d'un programme Alpha structuré pour lequel ce graphe ne possède pas de cycle contenant un appel ne pose pas de problème particulier. Lorsque ce graphe possède des cycles (dans presque tous les cas), nous ne pouvons définir un algorithme d'ordonnancement qui trouve un ordonnancement dès qu'il existe. Par exemple, la structuration en Alpha permet de définir un programme l1 appelant s1 de telle manière que l'exécution de s1 ne puisse pas avoir lieu d'un seul coup: les variables entrées à s1 dépendent des variables sorties de *la même instance* d'utilisation de s1.

Nous avons donc restreint le nombre de programmes considérés en introduisant la propriété de *dépendances séparées* pour un appel. Lors de l'appel à s1, on peut distinguer, pour chaque variable, les indices *de l'appelé* (utilisé dans le sous-système s1) et les indices *de l'appelant* (indices d'extension de l'appel). Cette propriété, illustrée en figure 3.1, permet d'assurer que les dépendances impliquées dans un cycle du graphe de dépendance structuré ne "mélange" pas les indices de l'appelant et de l'appelé. Cette propriété assure qu'une instance du sous-système s1 peut être exécutée d'un seul coup.

Pour les programmes structurés ayant cette propriété, on propose alors deux types d'ordonnancement de l'appel: un mode *unidimensionnel* où l'appel n'entraîne pas de dimension supplémentaire dans l'ordonnancement (l'ordonnancement reste de même dimension que celui qui serait obtenu si l'appel a s1 durait un temps constant) et un mode *bidimensionnel* où toutes les variables d'entrée/sortie de s1 sont ordonnancées la même heure, l'ordonnancement de s1 indiquant alors les minutes. Par exemple, sur la figure 3.1, supposons que l'ordonnancement de s1 soit[3]:

$$\forall \mathtt{i} = 1 \ldots n \;\; T^{\mathtt{s1}}_{\mathtt{ui}}\mathtt{[I]=I} \quad \forall \mathtt{j} = 1 \ldots m \;\; T^{\mathtt{s1}}_{\mathtt{vj}}\mathtt{[I]=I+1}$$

et supposons en plus que la dépendance entre V2 et U1 soit uniforme: (I,J->I,J-1). On peut alors définir un ordonnancement structuré linéaire (mode unidimensionnel) pour l1 compatible avec l'ordonnancement de s1 ci-dessus:

$$\forall \mathtt{i} = 1 \ldots n \;\; T^{\mathtt{l1}}_{\mathtt{Ui}}\mathtt{[I,J]=I+J} \quad \forall \mathtt{j} = 1 \ldots m \;\; T^{\mathtt{l1}}_{\mathtt{Vj}}\mathtt{[I,J]=I+J+1} \quad .$$

3. ui et vi sont les variables du système s1 qui correspondent respectivement aux variables Ui et Vi.

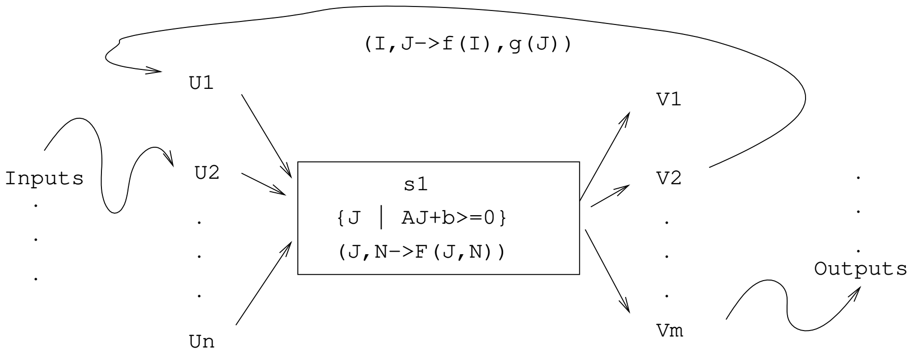


FIG. 3.1 – *Graphe de dépendance structuré générique (extrait) d'un programme* `l1` *appelant le système* `s1`*. Cet appel à la propriété des dépendances séparées,* `I` *représente les indices de l'appelé,* `J` *les indices de l'appelant.*

Mais si l'ordonnancement de `s1` avait été:

$$\forall \texttt{i} = 1 \ldots n \ T^{\texttt{s1}}_{\texttt{ui}}\texttt{[I]=I} \quad \forall \texttt{j} = 1 \ldots m \ T^{\texttt{s1}}_{\texttt{vj}}\texttt{[I]=I+N}$$

il ne serait plus possible de composer un ordonnancement structuré linéaire compatible avec cet ordonnancement. L'ordonnancement structuré trouvé serait alors (mode bidimensionnel):

$$\forall \texttt{i} = 1 \ldots n \ T^{\texttt{l1}}_{\texttt{Ui}}\texttt{[I,J]=}\begin{pmatrix} \texttt{J} \\ \texttt{I} \end{pmatrix} \quad \forall j = 1 \ldots n \ T^{\texttt{l1}}_{\texttt{Vj}}\texttt{[I,J]=}\begin{pmatrix} \texttt{J} \\ \texttt{I+N} \end{pmatrix} \quad .$$

Dans [101], nous avons identifié la propriété que doit respecter un appel avec dépendances séparées pour que l'on puisse l'ordonnancer en mode unidimensionnel. Nous avons aussi proposé une méthode générique pour la recherche d'ordonnancement structuré d'un programme structuré quelconque. Cet algorithme a récemment été implémenté dans MMAlpha, il permet d'avoir un approche remontante lors de l'ordonnancement d'une librairie (les sous-systèmes sont ordonnancés d'abord). En pratique, une approche descendante est aussi intéressante pour, par exemple, assurer le pipeline efficace d'une série d'utilisation du même sous-système. Donc on retrouve la situation de l'ordonnancement non structuré: une procédure automatique existe, mais la complexité du processus est telle qu'il est plus raisonnable de le faire contrôler par l'utilisateur.

Définir un *bon* ordonnancement structuré se révèle difficile car la structuration est un processus qui simplifie la représentation du programme en faisant disparaître des calculs derrière des appels. Du coup, l'optimisation globale qu'était l'ordonnancement ne peut avoir lieu : les résultats produits seront forcément moins bons que ceux de l'ordonnancement à plat. L'approche choisie ici n'est pas la seule possible : beaucoup de ces choix ont été guidés par la nécessité d'une implémentation rapide. On pourrai comme on l'a dit, proposer des ordonnancements paramétrés. On pourrait aussi imaginer de modifier la structuration du programme pour améliorer l'ordonnancement, c'est-à-dire fournir un outil d'aide à la structuration automatique de calculs, mais cela reste pour l'instant assez loin de nos préoccupations.

Il semble qu'actuellement le plus gros obstacle pour l'utilisateur de MMAlpha soit l'interprétation des résultats de l'ordonnanceur. En effet il n'existe pas actuellement de technique permettant de réaliser en matériel un programme dont l'ordonnancement est multidimensionnel. On sait générer du code [125] efficace pour une machine séquentielle ou parallèle, mais pas de matériel efficace. Dans le cas d'ordonnancement unidimensionnel, l'interprétation de l'indice `t` représentant le temps comme un *compteur* comptant les tops d'horloge était naturel, d'où l'interprétation des dépendances temporelles comme de simples registres. Si l'ordonnancement est multidimensionnel, l'interprétation des multiples indices temporels est beaucoup plus difficile : non seulement la conception des circuits multi horloge n'est pas bien maîtrisée, mais en plus les multi-horloges générées par MMAlpha risquent d'être particulièrement difficiles à implémenter (nombre variable de minutes par heure). Dans [52], nous avons formalisé deux interprétations du temps physique en fonction des indices de temps d'un programme ordonnancé mais sans résoudre le problème de l'implémentation effective pour le matériel. C'est une direction dans laquelle nous travaillons activement actuellement. À noter que récemment, Wilde et al. [15] ont proposé une méthode pour réaliser le contrôle quadratique dans une architecture systolique linéaire. Cette méthode pourrait être une alternative à l'approche multidimensionnelle que nous souhaitons réaliser.

### 3.1.3 Génération RTL

Après l'ordonnancement et l'allocation, on dispose d'une *spécification opérationnelle* du programme. Si on vise une implémentation logicielle, on sait "en gros" comment le programme va s'exécuter sur un réseau de processeurs. La phase suivante correspond à la génération de code. Comme on vise une implémentation matérielle, plutôt que du code, on va explicitement générer la description matérielle de chaque processeur. C'est une phase importante car on va introduire une "double" sémantique dans le programme Alpha: la sémantique induite par le langage lui-même et la sémantique induite par l'interprétation des différents composants. En effet, une fois traduits dans un langage de description de circuits (VHDL, Verilog, ...), tous les domaines Alpha auront disparus, il faut donc introduire des mécanismes de contrôle qui assurent que la spécification traduite en VHDL sera équivalente au programme AlpHard.

C'est la phase de parallélisation où les différents outils de compilation/parallélisation de circuits divergent. Même si, pour la plupart, ils vont au final générer du VHDL au niveau structurel, le modèle de machine cible qu'ils utilisent les distingue. À ce stade, par exemple, l'outil Compaan [86] génère un implémentation des processus de Kahn, l'outil Pico [56] génère une entrée pour le compilateur VLIW reciblable de Hewlett Packard. Pour MMAlpha, on génère une description RTL d'un circuit systolique. Cela se déroule en deux étapes: d'abord la mise en place de la description RTL proprement dite (génération d'un programme Alpha0), puis la structuration de ce programme (génération d'un programme AlpHard). La génération RTL avait été d'abord étudiée par Catherine Dezan [42] et les principales étapes de la génération avaient été mentionnées dans [162]. Actuellement, toute la phase de génération RTL est complètement automatisée grâce aux paquetages `ToAlpha0v2` et `AlpHard`. Nous avons mentionné ces transformations dans les publications [107, 50, 74, 93].

La génération de Alpha0 peut démarrer à partir du moment où chaque indice peut être interprété, soit comme un indice temporel (top d'horloge), soit comme un indice spatial (numéro de processeur). Actuellement, nous sommes capable d'effectuer cette transformation pour des programmes contenant un seul indice temporel et un ou plusieurs indices spatiaux. Le principe qui guide la transformation est le suivant: chaque variable Alpha sera interprétée comme un fil dans le matériel (à terme, elle sera traduite en un signal VHDL). La génération du programme Alpha0 peut être décomposée en deux parties : d'abord l'identification de composants matériels puis la génération et le pipeline des signaux de contrôle.

L'identification des composants matériels consiste à transformer le système Alpha de manière à ce que chaque équation puisse être interprétée comme un des composants suivants: registre, opérateur (addition, multiplication, opérateurs logique, multiplexeur), simple connexion ou initialisation d'un signal de contrôle. Par exemple, une dépendance temporelle entre deux variables:

```
A[t,p] = B[t-1,p]
```

est interprétée comme un registre entre les signaux `A` et `B` (sous réserve que `t` soit bien l'indice interprété comme le temps). La principale difficulté de cette phase réside dans la traduction des branches de `case`. Il convient de séparer les `case` *spatiaux* (les contraintes des domaines des restrictions du `case` portent uniquement sur les indices spatiaux) et les `case` *spatio-temporels* (les contraintes des domaines des restrictions du `case` portent sur des indices spatiaux et temporels). Les `case` spatiaux sont simplement interprétés comme modélisant différents comportements pour différents numéros de processeurs. Les `case` spatiaux-temporels correspondent à un changement de comportement d'un processeur au cours du temps et nécessitent l'adjonction d'un signal de contrôle qui contrôlera ce changement de comportement.

N'importe quel signal de contrôle peut être généré. Par exemple, l'équation suivante:

```
C[t,p] = case
          {| t=p }: C1[t,p];
          {| t>p }: C2[t,p];
      esac;
```

nécessite un signal de contrôle pour être réalisée en matériel. On génère donc le signal `C_ctl` et le contrôle “redondant” correspondant. La nouvelle équation de `C` ci-dessous est immédiatement interprétée comme un multiplexeur contrôlé par le signal `C_clt`

```
C_ctl[t,p] = case
               {| t=p }: True;
               {| t>p }: False;
             esac;

C[t,p] = if C_ctl[t,p] then
             case
               {| t=p }: C1[t,p];
               {| t>p }: 0[];
             esac else
             case
               {| t=p }: 0[];
               {| t>p }: C2[t,p];
```

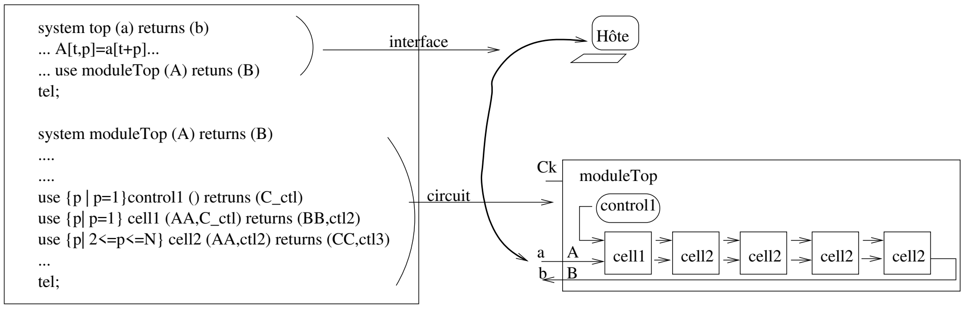


FIG. 3.2 – *Programme AlpHard générique pour un réseau linéaire. Le réseau contient* `N` *cellules, la première diffère des autres. Le contrôleur envoie ses signaux dans la première cellule, ces signaux sont pipelinés. Ici toutes les entrées se font dans la première cellule.*

```
            esac;
```

Pour le signal `C_ctl`, il sera réalisé avec un automate à états finis contenant un compteur sur l'horloge. A priori, cet automate est généré dans chaque processeur mais dans la plupart des cas (et c'est le cas ici), le signal de contrôle peut être pipeliné ou diffusé et un seul automate suffit pour tout le réseau.

En effet, sur l'exemple ci dessus on voit que la valeur `True` doit se trouver dans le processeur `p` au temps `t=p`. C'est un contrôle *linéaire* (la relation entre le temps et le processeur est linéaire) : on peut le réaliser avec un canal d'un bit qui traverse le réseau avec un registre par processeur. De même, un contrôle constant (condition du type `t=5`) peut être réalisé une seule fois et diffusé à tous les processeurs l'utilisant. Dans MMAlpha, la détection et le pipeline (ou la diffusion) de tels signaux se fait automatiquement grâce au paquetage `PipeControl`. La génération automatique de contrôle est une étape qui intéresse beaucoup les concepteurs car l'écriture manuelle du contrôle est très fastidieuse et augmente beaucoup le risque d'erreur.

Le passage de Alpha0 à AlpHard se fait automatiquement aussi. Grâce aux facilités de calcul sur les polyèdres, on identifie les régions de l'espace des processeurs sur lesquelles tous les processeurs ont le même fonctionnement. On génère alors une *cellule* qui représentera le matériel pour ce processeur (l'entité VHDL) et un `use` dont le domaine d'extension est cette région calculée. Le programme AlpHard est exactement équivalent au programme Alpha0, mais il est structuré et chaque sous-système est soit une *cellule*, soit un *contrôleur*, soit un *module* (circuit assemblant les cellules). Sur la figure 3.2 on a représenté un programme AlpHard générique pour un réseau linéaire. En plus des composants qui sont interprétables en matériel, l'*interface* (système `top` sur la figure 3.2) est le système qui encapsule le module pour avoir un système sémantiquement équivalent au programme de départ. Cette interface servira à interfacer le matériel généré avec le processeur hôte (voir le programme de la section A.3.1 en annexe).

Un programme AlpHard peut être directement traduit en VHDL dès que l'on affecte les paramètres de taille (jusqu'ici, toute la synthèse était paramétrée). Le VHDL généré est synthétisable par les outils les plus connus (il s'agit de VHDL comportemental de niveau transfert de registre), il est "synthétisable" au sens du groupe de travail "VHDL Synthesis

Interoperability Working Group of the Design Automation Standards Committee" (Swig) qui s'est attaché a définir un sous ensemble de VHDL synthétisable normalisé [1]. On peut voir un exemple de VHDL généré en annexe (section A.4). Le premier traducteur VHDL a été réalisé par Patricia Le Moenner [94]. Récemment une nouvelle version du traducteur a été réalisé par Patrice Quinton en utilisant le paquetage `Meta`. Des traductions vers d'autre formats ont été envisagées, en particulier vers le format SPRAY proposé par Dominique Lavenier et Erwan Fabiani pour implémenter efficacement les réseaux réguliers générés sur des FPGA.

### 3.1.4 Analyse et simulation

En plus des grandes lignes de la méthodologie de conception présentées dans les trois sections précédentes, on peut, à tout moment, analyser le système Alpha courant ou le simuler. L'analyse tient lieu de vérification formelle, la simulation tient lieu de test sur des données particulières.

L'outil d'analyse de MMAlpha (paquetage `Analyze` développé par Florent Dupont de Dinechin pour les systèmes simples et structurés [51]) est un des points forts du logiciel. Grâce à la sémantique très précise du langage et à l'assignation unique, beaucoup de propriétés peuvent être vérifiées statiquement. Par exemple, on peut vérifier facilement que chaque variable est affectée une et une seule fois sur tous les points de son domaine. Bien qu'elle soit simple à formuler, cette propriété est très difficile à vérifier à la main sur des programmes non triviaux possédant plus de trois indices. En traduisant en Alpha divers programmes tirés de librairies ou de publications, nous avons trouvé de très nombreuses erreurs. Grâce à cet outil, il devient difficile d'écrire des programmes incorrects du point de vue des dépendances ; un des seul moyens de le faire est d'introduire des cycles de dépendance (donnée dépendant d'elle même, Y. Saouter a montré [147], que la détection de tels cycles est un problème indécidable).

L'outil d'analyse se révèle aussi utile pour des utilisations externes à la compilation d'Alpha: en collaboration avec Sophie Robert et Florent Dupont de Dinechin, nous avons expliqué dans [59] l'utilisation de cet outil pour la spécialisation d'une implémentation ScaLaPack de l'algorithme de décomposition LU au cas des matrices sous forme de Hessenberg. Le résultat important sous-jacent est la mise en évidence de la puissance que peut apporter l'outil de manipulation de polyèdres Polylib pour la spécialisation de programmes contenant des boucles imbriquées. Pour être complet, le système MMAlpha devrait prochainement être interfacé au prouveur de théorème PVS, ce qui permettrait de prouver formellement beaucoup de nouvelles propriétés (voir section 3.2.5).

La simulation d'un programme Alpha se fait en générant du code `C` grâce aux programmes `write_c` écrit par Doran Wilde [129] ou `codeGen` écrit par Fabien Quilleré [125]. Le code `C` généré par `write_c` est un code d'évaluation des équations récurrentes à la demande. Il n'est pas très efficace, mais ne nécessite pas que le programme Alpha ait été ordonnancé auparavant. Le générateur de code ordonnancé `codeGen` est beaucoup plus optimisé. Pour un ordonnancement donné, ce générateur de code minimise la mémoire utilisée et simplifie au maximum le contrôle. Ce générateur de code peut être utilisé en dehors de l'environnement MMAlpha et semble être aujourd'hui très apprécié par la communauté de parallélisation de boucles. Le module `Meta` développé par Patrice Quinton permet maintenant de générer facilement des traducteurs d'Alpha en d'autres langages. Il pourrait permettre de résoudre le problème de la génération de code structuré qui n'a toujours pas été abordé.

Malgré la puissance de l'analyse statique, la simulation est toujours indispensable, au moins pour la spécification du programme Alpha initial ainsi que pour la vérification fonctionnelle du VHDL obtenu. Un autre domaine dans lequel les outils de simulation de révèlent utiles est la co-simulation. La conception avec MMAlpha sera très probablement intégrée à un processus de conception conjointe. Aujourd'hui, un système pouvant être intégré à une puce possède nécessairement des parties régulières (calculs) et des parties moins régulières (contrôle). Nous avons étudié dans le projet CAIRN, en collaboration avec Franck Bardoult, Patrice Quinton, Sanjay Rajopadhye et l'équipe Ep-Atr de l'Irisa, la conception conjointe avec les environnement Signal et Alpha [68, 58]. Dans ce type de coopération, la co-simulation joue un rôle primordial car la vérification formelle de l'ensemble est très difficile. Nous avons en particulier co-simulé un algorithme de reconstruction d'image écrit en Signal, le programme Signal appelant un programme Alpha pour les calculs intensifs. Cela a permis de valider expérimentalement le protocole d'interfaçage que nous avions proposé entre Signal et Alpha. De plus, le générateur de code Alpha était aussi utilisé dans le processus de co-synthèse pour remplacer, dans le programme Signal original, l'appel de la fonction initiale par des primitives de communications avec l'architecture générée à partir du programme Alpha. De par leur souplesse et leur simplicité, les générateurs de code de MMAlpha se révèlent donc très utiles pour la simulation ainsi que pour d'autres traitements.

La simulation de code Alpha se fait à partir de fichiers de données généralement fournis par un expert du domaine applicatif considéré. La simulation du VHDL résultant de la synthèse du programme Alpha se fait à partir de fichiers contenant les stimuli de tous les signaux d'entrée du programme VHDL. Nous avons réalisé un programme qui génère de tels fichiers de stimuli à partir des fichiers résultants de la simulation du programme Alpha. Ce générateur produit aussi le banc d'essai (*test-bench*) qui permet de simuler le VHDL produit et de vérifier que les résultats obtenus sont bien ceux prévus par l'exécution du programme Alpha. Cette vérification se fait non seulement sur les valeurs des résultats finaux des deux programmes mais aussi sur les dates d'arrivée des sorties (qui sont comparées avec les dates prévues par le programme AlpHard). La simulation fonctionnelle de VHDL étant extrêmement fastidieuse, un tel générateur facilite grandement le travail du concepteur.

### 3.1.5 Après MMAlpha

Le système MMAlpha permet donc de compiler/paralléliser semi-automatiquement des descriptions RTL de circuits à partir de spécifications haut niveau. Il est important de noter que la description RTL est exprimée en AlpHard (donc dans le langage Alpha). Le fait de générer une telle description ne constitue qu'une partie de la synthèse : reste le passage jusqu'à l'implémentation matérielle réelle réalisé par d'autres outils spécialisés. Cette étape est souvent sous-estimée par les gens travaillant "haut niveau". Or, même si des outils commerciaux existent, ils sont souvent délicats à maîtriser et cela peut ruiner les performances d'une architecture. C'est ici que la collaboration avec de véritables concepteurs de circuits est le plus indispensable.

Ce processus se décompose généralement en la génération d'une *liste d'interconnexion* (ou netlist, cette étape est appelé la synthèse logique), et le placement/routage pour obtenir un *dessin de masque* (layout) ou un fichier de configuration (bitstream) pour une architecture reconfigurable. Le passage par VHDL pour effectuer la synthèse logique semble obligé tant ce format est accepté universellement (au moins en Europe). Cependant, cette universalité existe au prix de performances moindres pour des traitements particuliers. Dans le cadre de MMAlpha, par exemple, nous avons eu de gros problèmes pour conserver, lors de la synthèse logique, les propriétés de régularité et de hiérarchie auxquelles nous tenions tant. Un exemple simple est représentatif des problèmes que l'on peut rencontrer: celui des `generate` imbriqués.

L'expression de la réutilisation d'un composant en VHDL (l'équivalent de la boucle `for`) se fait avec l'instruction `generate`. Le sous ensemble VHDL synthétisable définit par le groupe de travail Swip [1] interdit l'utilisation de `generate` imbriqués. Il en résulte qu'on ne peut pas définir une architecture composée d'une grille 2-D de processeurs, ce qui nous interdit de synthétiser la plupart des algorithmes systoliques pour l'algèbre linéaire par exemple. Pourtant, la méthodologie de conception est exactement identique que celle que l'on voit en section 3.3 jusqu'à la production d'AlpHard, mais on ne peut rien faire après. Ce que cette restriction met en lumière, c'est le fait qu'il est extrêmement inconfortable de dépendre d'outils que l'on ne maîtrise pas. C'est pourtant, à mon avis, obligatoirement le cas, car il me paraît impensable de réaliser ces traitements qui sont programmés par d'énormes équipes qui ont une expérience de plus de 10 ans. Il y a donc deux solutions: soit on expérimente notre méthodologie avec des algorithmes et des plates-formes cibles pour lesquelles les outils disponibles fonctionnent bien et sont contrôlables (c'est ce que nous avons fait avec les applications se synthétisant sur des réseaux systoliques linéaires [93, 74]), soit on continue de raffiner la description du matériel en passant d'une description structurelle à une description *topologique* et on essaye de s'interfacer avec des outils de plus bas niveau pour faire passer les informations topologiques (c'est ce que nous faisons avec la collaboration avec Dominique Lavenier et Erwan Fabiani pour utiliser arriver à placer efficacement des architectures régulières sur FPGA [61]).

Historiquement, lorsque la génération de programme RTL a été mise en place, nous avons tout de suite ciblé le VLSI en passant par les outils de synthèse que nous utilisions à l'époque. Les premiers résultats ont été présenté par Patricia Le Moenner dans sa thèse [94]. Pour donner une idée, l'architecture synthétisée pour un FIR avec 20 coefficients fonctionnait à une fréquence de 100 MHz et une surface de $4500 \times 7500\lambda^2$ (9 $mm^2$ en technologie 0.5 $\mu$). Les temps de placement/routage passaient de 2h à 20 minutes lorsque l'on réutilisait le placement d'un processeur (conservation de la hiérarchie).

Ces résultats ont été, en fait, relativement décevants pour nous. Nous pensions tirer partie de la régularité pour avoir un routage plus efficace et plus rapide. Si le processus de routage est effectivement plus rapide (d'un facteur variant entre 5 et 20 suivant les tailles de circuits générés) : les circuits obtenus en laissant l'outil (Compass) router *à plat* (c'est à dire sans respecter la hiérarchie ni la réutilisation de cellules identiques) sont moins gros en surface que ceux pour lesquels nous avons imposé de respecter la hiérarchie. C'est un gros désavantage car le silicium coûte cher pour une implémentation spécifique. Je pense que cette différence restera vraie très longtemps. Les outils de synthèse se doivent de traiter efficacement les circuits relativement gros, et la perte de surface en pavant le circuit en rectangle ne sera jamais compensée par le gain obtenu en optimisant mieux le placement routage à l'intérieur du rectangle. Il y avait un point positif : lorsque le circuit dépassait une certaine taille (50 processeurs pour le FIR par exemple), la synthèse à plat échouait, mais il est à mon avis peu probable que l'on utilise un compilateur/paralléliseur tel que MMAlpha pour synthétiser des circuits de cette taille.

Le problème se pose en termes différents pour les FPGA. Les ressources de routage sont figées et la complexité du processus de placement routage s'en trouve augmenté. Étant personnellement, depuis ces résultats, un peu sceptique quant à l'utilisation de notre méthodologie pour le VLSI, je décidai de me diriger vers les architectures reconfigurables. Comme expliqué en section 2.1, l'augmentation de complexité des FPGA augmente la demande d'outils pour de telles architectures. Les constructeurs fournissent des outils de placement/routage qui se révélèrent très mauvais pour les placement réguliers. Erwan Fabiani et Dominique Lavenier firent une étude assez poussée pour évaluer le besoin de travaux de recherche pour l'implémentation efficace d'applications régulières sur FPGA. Après avoir longuement tergiversé sur le format de transfert vers les outils Xilinx à utiliser pour transmettre les informations topologiques importantes, Dominique Lavenier proposait le format Spray (Specify and Place Regular fpgA arraY).

Le principe du traitement à faire après MMAlpha dans le cas d'une implémentation sur FPGA avec Spray est le suivant : on traduit le programme AlpHard en Spray puis on rajoute à la spécification Spray des informations de placement (informations dépendantes de la technologie cible: la granularité du FPGA, le nombre de CLBs, etc...). Ces informations de placement peuvent être obtenues en utilisant les outils de Xilinx pour placer une cellule par exemple. Une fois enrichie, la spécification Spray sera traduite vers, par exemple, une netlist au format EDIF enrichie de directives de placement pour chaque signal ou opérateur. Le routage est alors laissé aux soins de l'outil Xilinx. Cette partie de la méthodologie n'a pas été encore implémentée mais elle a été testée "manuellement" sur quelques exemples [60]. Une utilisation fréquente de ce chemin de conception demandera beaucoup de travail, notamment au niveau de l'implémentation de librairies pour les entiers (actuellement on travaille au niveau bit).

À côté de ces recherches concernant spécifiquement l'implémentation optimisée des réseaux réguliers, nous avons aussi exploré une voie plus traditionnelle pour produire du code pour FPGA avec les outils de Synopsis (`fpgaCompiler`) et de Xilinx (`Xilinx Design Manager`). Les expérimentations ont eu lieux avec la carte Spyder du FZI de Zurich. Les résultats se révèlent assez encourageants, c'est à dire que l'on arrive assez rapidement à produire un bit-stream. Comme nos architectures cibles sont cadrées dans un modèle précis, il va être possible de générer avec le code VHDL, un fichier de commande pour les outils à suivre (sorte de fichier `Makefile`) et de se rapprocher d'une génération d'IP (plus valorisée aujourd'hui que la simple production du code VHDL). À titre d'exemple, nous avons publié en collaboration avec Anne-Claire Guillou, Patrice Quinton et Daniel Massicote, dans [74], des chiffres concernant la synthèse du DLMS (FIR adaptatif) et montré que sur le Xilinx XCV800 dont nous disposions sur la carte Spyder on pouvait implémenter un filtre possédant au moins 64 cellules (64 poids dans la convolution, voir section 4.1.4). Comme prévu, la partie la plus difficile à optimiser est la génération de l'interface avec la machine hôte. Nous avons aussi, en collaboration avec Steven Derrien, étudié ce problème dans [140] : ce travail est plus détaillé en section 3.2.3.

Malgré les résultats en demi-teinte obtenu pour la synthèse VLSI, nous avons continué à croire à cette approche. Les courants récents semblent nous donner raison : la rapidité de la production de code VHDL correct semble un défi pour tout le monde. Lors d'une visite à l'Université du Quebec à Trois Rivières, nous avons regardé l'architecture présentée dans [**?**], il ne m'a fallu qu'une journée pour générer le code VHDL correspondant. Certains traitements demanderont nécessairement plus de temps, comme la simulation par exemple, mais le script de conception pouvait être utilisé pour explorer une allocation différente, qui dans ce cas précis pouvait mener à une implémentation intéressante. Pour ce type de prototypage rapide la méthodologie de conception présentée ici est certainement utile.

## 3.2 Traitements additionnels pour la conception

La section précédente décrivait ce que l'on peut considérer comme le squelette du système MMAlpha. Ces transformations ont été longuement expérimentées et validées. Cette section regroupe tous les traitements "complémentaires" qui ont été étudiés. Certains de ces traitements ont été étudiés, implémentés et testés, d'autres ont seulement été étudiés théoriquement. L'ensemble de ces traitements (accompagnés de ceux de la section précédente évidement) constitue, à mon avis, ce qui est nécessaire et suffisant pour avoir un prototype complet et opérationnel de compilateur/paralléliseur pour circuits.

Nous avons regroupé ces "extensions" en sept thèmes. Les $\mathbb{Z}$-polyèdres devrait bientôt apparaître dans le système et augmenter sensiblement le nombre d'algorithmes traités ainsi que la qualité des architectures générées. L'optimisation de la largeur en bit des chemins de donnée est un problème technique rencontré par tous les outils de synthèse. La génération d'une interface optimisée est une des clés pour la performance, en particulier sur FPGA. Le fait de démarrer la conception par une spécification impérative (type Matlab ou C) est une revendication permanente des concepteurs. La connexion avec un outil de preuve algébrique étendra à peu de frais la puissance des vérifications formelles effectuées. L'expression du partitionnement et de la mémoire permettra de ne pas être limité par les problèmes de contraintes de ressources et enfin la conception hiérarchisée ouvrera la porte aux très grosses conceptions (System On Chip).

### 3.2.1 les $\mathbb{Z}$-polyèdres

L'extension du langage Alpha aux $\mathbb{Z}$-polyèdres a été vue dans la section 2.2.2. Nous décrivons ici rapidement les solutions techniques que nous avons apportées pour implémenter l'extension de la Polylib aux $\mathbb{Z}$-polyèdres [130, 131, 117].

Lorsque je suis arrivé à Rennes (en 1994), nous avons longuement réfléchi aux différentes possibilités d'extension du langage, entre les $\mathbb{Z}$-polyèdres introduits par Corinne Ancourt, les LBL de Tiech et Thiele et même les formules de Presburger de Bill Pugh [123]. Les règles sémantiques du langage Alpha imposaient certaines propriétés que devait posséder l'ensemble des objets tenant lieu de domaine Alpha. Par exemple, on devait être capable de calculer l'intersection de deux domaines, ce qui est très difficile pour des LBLs. Le modèle des formules de Presburger était presque trop puissant pour nous et nous n'avions pas de garantie quant à la complexité des opérations dont nous avions besoin. Les $\mathbb{Z}$-polyèdres eux semblaient convenir parfaitement, ils permettaient de modéliser les nids de boucles

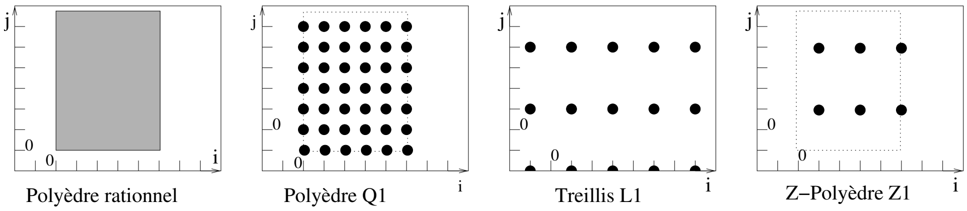


FIG. 3.3 – *Exemple montrant le polyèdre* `Q1` *le treillis* `L1` *correspondant à l'image de* $\mathbb{Z}^2$ *par la fonction* `(i,j->2i+1, 3j+5)` *et le* $\mathbb{Z}$*-polyèdre* `Z1=Q1∩L1`. *La courbe en pointillé indique la forme du polyèdre rationnel* `Q1`

dont le pas n'est pas 1, et ils avaient été pressentis pour la synthèse de circuits, en particulier dans les travaux de Van Dongen [47], Lenders et Rajopadhye [98]. Il manquait cependant les bases théoriques qui avait fait le succès de la librairie polyédrique: comment représenter et calculer sur les $\mathbb{Z}$-polyèdres.

La représentation des $\mathbb{Z}$-polyèdres fut relativement simple à choisir pour nous qui disposions déjà d'une expérience et d'une librairie sur les polyèdres. Un $\mathbb{Z}$-polyèdre est l'image d'un polyèdre par une fonction affine entière inversible. Par exemple, `Z1={2i+1, 3j+5 | -1<= 2i <= 4; -3<= 3j <= 2 }` est le $\mathbb{Z}$-polyèdre qui est l'image du polyèdre `Q2={i,j| -1<= 2i <= 4; -3 <= 3j <= 2 }` par la fonction `(i,j->2i+1, 3j+5)`. Ce $\mathbb{Z}$-polyèdre peut alternativement être défini comme l'intersection du domaine `Q1= {i,j | 0 <=i <= 5; 0 <=3j <= 20 }` avec le treillis obtenu en prenant l'image de $\mathbb{Z}^2$ par la fonction `(i,j->2i+1, 3j+5)`. La figure 3.3 montre les points appartenant au $\mathbb{Z}$-polyèdre `Z1`. La manipulation de $\mathbb{Z}$-polyèdres peut donc se ramener à la manipulation de polyèdres et de treillis entiers (ou réseaux).

Notre première préoccupation a été de vérifier que les $\mathbb{Z}$-polyèdres constituaient bien des candidats valides pour implémenter les domaines Alpha. Dans [130] nous avons montré, en collaboration avec Patrice Quinton et Sanjay Rajopadhye, que les $\mathbb{Z}$-polyèdres pouvaient être utilisés comme domaines et que de plus, dans ce cadre, on pourrait alors utiliser des dépendances rationnelles. Nous avons aussi introduit la notion de *pseudo unimodularité* d'une fonction rationnelle par rapport à un domaine, qui permet d'identifier les cas ou un changement de base par une fonction rationnelle carré inversible est valide.

Ce travail s'appuyait sur l'étude théorique de manipulation des $\mathbb{Z}$-polyèdres que nous avions réalisée avec les mêmes personnes [131]. Le principal problème que nous avions rencontré pour implémenter les transformations dont nous avions besoin était celui de la représentation canonique des $\mathbb{Z}$-polyèdres: comment tester l'égalité de deux $\mathbb{Z}$-polyèdres? En effet, nous savions tester l'égalité (rationnelle) de deux polyèdres, ainsi que celle de deux treillis (grâce à la décomposition de Hermite), mais plusieurs couples polyèdre-treillis pouvaient représenter le même $\mathbb{Z}$-polyèdre (dans le cas de polyèdres "plats"). Cette situation est illustrée en figure 3.4. Nous avons donc proposé une représentation canonique pour les $\mathbb{Z}$-polyèdres. Nous avons aussi, pour l'occasion, proposé une forme affine de la décomposition de Hermite. De plus, le fait de tester si un polyèdre contenait ou non un point entier était depuis toujours remplacé par le test de vacuité rationnel dans Polylib. Cela ne posait pas de problème en pratique, mais nous ne savions pas si cela serait toujours le cas pour

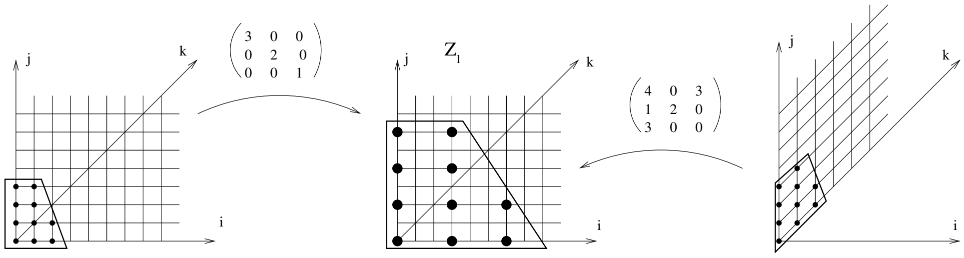


FIG. 3.4 – *Le $\mathbb{Z}$-polyèdre du milieu peut être vu comme l'image (par les fonctions indiquées sur les flèches) des deux polyèdres des bords. La représentation utilisant le polyèdre de gauche est en forme canonique (matrice en forme de Hermite et polyèdre dans l'espace des deux premiers indices). Le test d'égalité de ces deux représentations peut se faire grâce à la représentation canonique proposée dans [131].*

```
system even:{N | 3<=N }
          (B : {2i | 0<= i <=N-1} of integer)
       returns
          (A : {2i | 1<= i <=N} of integer);
let
  A = case
    {4i | 1<=2i<=N} : B.(j->j-4);
    {4i+2 | 0<=2i<=N-1} : B.(j->j-2);
   esac;
tel;
```

FIG. 3.5 – *programme Alpha contenant des $\mathbb{Z}$-polyèdres*

les $\mathbb{Z}$-polyèdres. Nous avons donc implémenté une interface avec l'outil Omega [123] et pu constater que, si nous en avions besoin, le coût de ce calcul n'était pas prohibitif.

L'étape d'implémentation aurait dû suivre immédiatement. Par manque de main d'oeuvre, elle a dû attendre l'été 1999 pour être réalisée par Sunder Nookala [117]. La réalisation, comme toujours, a révélé quelques petits problèmes qui n'avait pas été résolus. Dans [117], nous avons détaillé les algorithmes utilisés pour chacune des transformations. Nous avons en particulier proposé un algorithme original pour le calcul de la différence entre deux treillis entiers. L'implémentation réalisée est opérationnelle, toutes les fonctions usuelles sont disponibles pour les $\mathbb{Z}$-polyèdres et les treillis entiers. Nous avons intégré à la librairie les codes de calcul des formes normales de Hermite et de Smith développés par Alain Darte. Nous avons aussi généralisé ces transformations aux fonctions affines et montré comment réaliser cela efficacement dans la représentation homogènes des espaces affines (représentation utilisée dans Polylib [161]).

Comme mentionné précédemment, cela ne signifie pas que les $\mathbb{Z}$-polyèdres sont accessibles dans MMAlpha. Si on peut actuellement écrire des programmes Alpha contenant des $\mathbb{Z}$-polyèdres (par exemple, un tel programme est présenté en figure 3.5) et manipuler des $\mathbb{Z}$-polyèdres depuis Mathematica, il reste encore à modifier toutes les fonctions de MMAlpha qui accèdent aux domaines directement par l'AST (sans passer par les fonctions de Polylib) – c'est un problème de programmation uniquement– mais aussi, ce qui est plus délicat, il faut réfléchir à l'extension de traitements dans lesquels les polyèdres jouent un rôle important, comme l'ordonnancement ou la génération de code par exemple.

Même si ils ne sont pas réellement accessibles dans le système MMAlpha actuel, je suis persuadé que les $\mathbb{Z}$-polyèdres doivent impérativement pouvoir être manipulés dans un outil de ce type. Il permettent une expressivité du langage raisonnable (expressions des algorithmes pair-impair en imagerie, des réseaux multi-horloge [98]), et autorisent l'implémentation de transformations importantes dans la conception de circuits systolique comme le partage d'opérateurs entre plusieurs signaux (voir [158] pour l'utilisation de cette technique pour le DLMS) ou le partitionnement (voir section 3.2.6). Les $\mathbb{Z}$-polyèdres seront aussi certainement utiles pour d'autres outils de compilations/parallélisation. Par exemple, les nouveaux paramètres générés par Pip pourrait être manipulés plus facilement car il correspondent souvent à une décomposition en unions de $\mathbb{Z}$-polyèdres (voir section 3.2.4).

### 3.2.2 Largeur en bits du chemin de donnée

Le passage en précision finie, c'est à dire la mise au point des tailles (largeur en nombre de bits) des chemins de donnée, est un problème important et difficile. Important car le coût de l'architecture est directement proportionnel à ces tailles, difficile car ce raffinement ne peut, en général, pas se faire manière formellement correcte. Par exemple, la preuve de convergence d'un algorithme implémenté en précision finie n'est pratiquement jamais faite: on prouve la validité de l'algorithme théoriquement (précision infinie) ou l'on simule extensivement l'algorithme (en précision flottante sur un processeur) puis on simule l'algorithme en précision finie (en virgule fixe car actuellement le coût des opérateurs flottants est trop important) pour valider le choix des tailles des chemins de donnée.

Ce traitement reste un problème technique qui est géré de manière plus ou moins manuelle par les outils de conception. On peut distinguer deux approches, une approche "formelle" tentant de formaliser les relations entre les largeur en bits des différentes données en fonction de leurs relations dans l'algorithme pour aider le concepteur à la mise au point de ces largeurs en bits (une telle approche est utilisée dans Gaut [150] et a été expliqué dans [159]) – et une approche "pratique" basée sur la simulation (utilisé dans Ptolemy, DSP station par exemple [87, 153]). Il semble que les deux approches soient complémentaires: fournir à l'utilisateur des outils pour choisir les largeur en bits va accélérer le processus, mais la simulation sera toujours nécessaire pour valider les choix (de plus certain choix ne peuvent être faits par des méthodes formelles, il faudrait des outils plus puissants qui soient capables de montrer des propriétés complexes comme la convergence par exemple). Parmis les approches citées ici, aucune ne traite convenablement les boucles (les boucles sont déroulées pour être parallélisées). L'environnement MMAlpha et le modèle polyédrique peuvent apporter ici quelque chose de nouveau.

J'ai tout d'abord été confronté au problème lors des travaux menés en collaboration avec Étienne Mémin sur l'estimation de mouvement (voir section 4.1.2). Avec Guillaume Giacobi, nous avions modifié le générateur de code C pour pouvoir simuler le fonctionnement d'un algorithme en précision finie. J'ai passé des semaines à simuler l'estimateur avec différentes largeur en bits pour les signaux. La validation d'un choix passait obligatoirement par l'appréciation subjective du résultat global, d'où la lenteur du processus. Ensuite, en collaboration avec David Cachera, nous avons essayé de réfléchir plus profondément au problème en essayant de relier cela aux techniques d'analyse statique (typage, interprétation abstraite).

Ce travail est en cours. Lors du stage de DEA de Djamel Zegaoui, nous avons généralisé les détections automatiques de largeur en bits (proposées dans [159]) aux algorithmes contenant des boucles (programmes Alpha). Nous avons aussi indiqué comment les équations régissant ces quantités pour un algorithme particulier pouvaient être générées automatiquement à partir du programme Alpha décrivant cet algorithme. Ces traitements seront complétés par une extension du système de typage du langage Alpha (voir figure 3.10 page 54) pour intégrer le processus de détermination de la taille des chemins de donnée aussi tôt que possible dans le processus de conception, c'est à dire juste après le processus d'ordonnancement/allocation (plus on attend, plus cette détermination devient complexe, mais on ne peut déterminer les largeurs en bits sans avoir une interprétation architecturale du programme). Ce traitement peut se superposer de manière complètement transparente aux autres transformations de la méthodologie de conception avec MMAlpha (voir section 3.3). Il sera aussi complété par la simulation en précision finie de l'algorithme ainsi adapté, le but principal étant d'aider le concepteur lors de ces choix.

### 3.2.3 Interfaçage

Dans le cadre de ce travail, on appelle interface la partie qui permet de relier l'architecture générée (le réseau systolique) à la partie qui va la contrôler (en général un processeur hôte). L'interfaçage est un problème difficile à plusieurs niveaux. D'une part, il fait intervenir des langages différents et donc des sémantiques différentes, il est donc difficile de générer une interface correcte par construction (fiable). D'autre part, les protocoles d'interfaces "matérielles" sont souvent lourds et peu efficaces, il est donc difficile de réaliser une interface qui soit efficace (rapide). Enfin, c'est un processus qui est très sensible au changement de technologie, il demande donc beaucoup d'investissement pour maîtriser finement les générations successives de matériels sans que cela soit considéré comme de la *vraie* recherche (l'investissement est difficilement rentabilisé par des publications). Nous avons étudié l'interfaçage de Alpha et Signal dans le projet CAIRN [68] en insistant sur l'aspect validité de la génération d'interface. Plus récemment, dans [140], nous avons étudié les interfaçages possible pour les architectures générées par MMAlpha placées sur FPGA en essayant d'obtenir de bonnes performances (il ne s'agissait pas, bien sûr, de générer des interface non correctes! Le cadre de ces travaux était beaucoup plus restreint que celui des travaux de CAIRN).

Le Langage Signal est un langage *temps réel* particulièrement adapté à la conception de systèmes réactifs dans lesquels le contrôle est complexe. Le projet EP-ATR travaille beaucoup sur Signal et sur l'environnement de manipulation de programmes Signal. Le but de la collaboration avec le projet Api était d'intégrer les deux environnements en un environnement de conception conjointe dans lequel les parties de contrôle auraient pu être traitées par Signal et les parties de calcul intensif par Alpha. Un des intérêts d'une telle collaboration était la capacité naissante de Signal à manipuler différentes horloges ayant des rapports de sur- ou sous-échantillonage entre elles. Or le processus de conception de MMAlpha effectue justement un changement de granularité dans la précision de la description de l'algorithme. Prenons par exemple un programme signal décrivant une suite (potentiellement infinie) de produits de matrice (chaque produit ayant lieu au top de l'horloge `h1`). Lors de la conception d'un réseau systolique pour le produit de matrice avec MMAlpha, cette horloge `h1` est subdivisée grâce à une sous-horloge `h2` qui correspond précisément à l'indice `t` introduit lors du processus d'ordonnancement (voir section 3.1.2). Dans le projet CAIRN [68], nous avons essayé de formaliser ce processus en définissant une interface Signal-Alpha au niveau *système* (correspondant, par exemple au produit de matrice) et une interface au niveau *élément* (correspondant alors aux données envoyées à chaque top dans le circuit de produit de matrice). Ces travaux se sont intégrés à la thèse d'Irina Smarandache [151] qui s'est attaché à définir les opérateurs dont Signal avait besoin pour pouvoir exprimer ces programmes *multi-horloge*.

Définir les différents niveaux d'interface autorise de faire, avec MMAlpha, la synthèse d'une fonction que l'on estime critique pour le temps d'exécution dans d'un programme Signal. Il faut bien sûr s'assurer que les appels de la fonction initiale et du circuit final sont équivalents. L'environnement MMAlpha s'est encore une fois montré utile pour cela: en utilisant, d'une part les informations retenues par le système interface du programme AlpHard (voir figure reffigSamba3, page pagereffigSamba3) et, d'autre part, les possibilités des générateurs de code de MMAlpha, on a pu générer, en même temps que le circuit AlpHard le code Signal remplaçant, dans le programme Signal de départ, l'appel de la fonction synthétisée. Ce processus ainsi que la preuve de sa validité et le protocole en résultant (interface entre l'hôte, implémentant le programme Signal, et l'architecture), se trouve détaillé dans [58].

Dans le cadre de cette collaboration, nous avons pu réaliser la co-simulation, au niveau système et élément, mais pas de véritable conception (en partie à cause du fait que le processus de conception avec MMAlpha n'était pas très stable). Lorsque nous avons voulu implémenter les architectures générées par MMAlpha, sur FPGA, nous avons été confronté au problème de l'efficacité de l'interface. L'interfacage au niveau élément était utile pour prouver la validité du système global, il se révèle catastrophique au niveau des performances. Les données transitant dans l'interface passent généralement par un bus. Or les bus sont gérés par des protocoles complexes, peuvent parfois être interrompus par des taches systèmes et leurs performances augmentent dans le temps encore beaucoup moins vite que celle des mémoires (dont on sait déjà que les performances augmentent beaucoup moins vite que celles des processeurs).

Prenons, par exemple, le bus PCI (32 bits, 33 MHz) alimentant la carte Spyder. Supposons que l'on doive implémenter une architectures systolique ayant 16 bits en entrée, 16 bits en sortie. En supposant que l'on sache gérer tout le reste de manière optimale, on ne pourra activer l'architecture systolique plus vite que 33 MHz, or les architectures que l'on génère peuvent souvent fonctionner beaucoup plus vite. Que faire? La première chose est d'implémenter une interface efficace. Dans [140], en collaboration avec Steven Derrien, nous avons étudié la génération automatique d'une telle interface pour FPGA dans le cadre restreint des architectures linéraires générées par MMAlpha aujourd'hui. Nous avons d'abord introduit des concepts importants qui permettent d'avoir une architecture systolique tournant plus vite que le bus (concept d'*horloge virtuelle* pour contrôler chaque pas de l'exécution du réseau, mécanisme de registre à décalage pour les entrées/sorties du réseau) et nous avons montré que l'on devait utiliser le mode de transfert par bloc ("burst mode") pour tirer le maximum de performance du bus. Les expérimentations ont été menées sur la carte Labomat3 préttée par le LSL de l'EPFL de lausanne citeLabomat3 et sur la carte spyder du FZI de Zurich citeSPYDER. D'autre part, il faudra, lorsque cela sera possible avec MMAlpha, trouver le bon compromis entre le partitionnement et le parallélisme pour optimiser la bande passante disponible du bus. C'est sur ce sujet que travaille Steven Derrien.

Actuellement l'interfacage n'a pas encore été physiquement réalisé, il devrait prochainement permettre de donner des chiffres bruts concernant l'accélération que l'on peut obtenir avec, à sa disposition, une carte Spyder et un logiciel MMAlpha.

### 3.2.4 Compilation de code impératif

Comme je l'ai déjà indiqué, le langage Alpha est un format rébarbatif pour ceux qui sont succeptibles de l'utiliser. Partir de code impératif est un des objectif que nous nous sommes fixé depuis longtemps (lors de mon recrutement sur le poste INRIA par exemple). Il me semble pourtant indispensable de conserver comme format interne un langage en assignation unique. Si l'on veut partir de et arriver à un code impératif, il faut d'une part pouvoir traduire automatiquement de l'impératif en assignation unique, d'autre part savoir générer un code impératif dans lequel la mémoire utilisée n'est pas trop importante.

Le second problème a été résolu par les travaux sur l'optimisation de la taille mémoire [126, 96, 23], c'est à dire que l'on sait comment générer du code utilisant peu de mémoire et on a des outils qui implémentent ces techniques. Le problème de la traduction de code impératif avait été étudié par Paul Feautrier [66]. La solution que propose Feautrier consiste à renommer les tableaux en partie gauche d'affectation par leur numéro d'instruction et à changer les lectures en partie droite grâce à une analyse de dépendance exacte (analyse data-flow). Il fallait donc adapter cette analyse pour traduire du C ou du Fortran en Alpha. Le problème était difficile et présentait peu d'intérêt au niveau recherche, nous avons donc essayé de récupérer des outils existants. Cette analyse avait d'abord été réalisée en LeLisp pour le paralléliseur Paf [119], puis une deuxième version avait été réalisé par Arnaud Leservot pour le CEA. Denis Barthou avait aussi programmé un tel outil en Caml. J'ai finalement opté pour la version développée par Peter Held pendant sa thèse à TUDelft [76].

D'une part l'outil HiPars [76] été bien documenté, accessible facilement, d'autre part il partait de Matlab ce qui est sensiblement plus intéressant que Fortran dans le domaine de la conception de systèmes dédiés. Grâce à HiPars, nous avions un traducteur de Matlab en Matlab assignation unique: j'ai réalisé un traducteur de Matlab assignation unique en Alpha. Cependant plusieurs problèmes rendent l'utilisation de ce chemin peu praticable. Le Matlab autorisé par HiPars ne contient aucun opérateur, seulement des appels de fonctions. Le programme Alpha en résultant contient une succession de `use` (la plupart correspondant en fait à des opérateurs) et les noms de beaucoup de variables avaient été générés par HiPars, ce qui rendait le programme peu lisible. Cela combiné avec le fait que pas mal de bogues subsistaient et que nous n'avions finalement pas, en interne, de besoin urgent d'un tel outil fait que nous avons laissé dormir cela pendant quelques années.

```
%parameter X 1 100;
%parameter Y 1 100;

%% Samba algorithm

for j=1 : 1 : Y,
     [ M(0,j) ] = const0();
end
for i=1 : 1 : Y,
     [ M(i,0) ] = const0();
end

for i= 1 : 1 : X,
  for j= 1 : 1 : Y,
     [ EqualQ ] = FuncEqual(QS(i),DB(j));
     [ MatchQ ] = FuncIf(EqualQ(),const15(),const12());
     [ Above ] = FuncMinus(M(i-1,j),const8());
     [ Left ] = FuncMinus(M(i,j-1),const8());
     [ Diag ] = FuncPlus(M(i-1,j-1),MatchQ());
     [ M(i,j) ]= FuncMax(const02(),Above(),Left(),Diag());
  end
end

for j = 1:1:Y,
     [ res(j) ] = M(X,j);
end
```

FIG. 3.6 – *Programme Matlab exécutant l'algorithme de Samba étudié en section 3.3 fourni à MatParser pour traduction en assignation unique. Les constantes ainsi que les opérateurs, non autorisés par la syntaxe restreinte de MatParser, ont été remplacés par des appels de fonctions*

Récemment, le travail de Peter Held a été repris par Bart Kienhuis, qui a reprogrammé cela de manière beaucoup plus robuste lors de son passage dans l'équipe d'Ed Lee à Berkeley. Le nouvel outil, MatParser [85], est maintenant intégré à MMAlpha, Le paquetage `imperativeToAlpha` permet de simplifier le programme après passage dans la moulinette MatParser. Il reste un problème pour généraliser ce processus, le processus de traduction en assignation unique peut faire intervenir des “nouveau paramètres” (hérités des nouveaux paramètres générés par les appels à Pip). En général, ces nouveaux paramètres correspondent à la manipulation de $\mathbb{Z}$-polyèdres. Nous réfléchissons actuellement à la manière de gérer cela le plus simplement possible. Il est possible, en effet, que l'utilisation de la $\mathbb{Z}$-Polylib dans MatParser puisse grandement simplifier la complexité des codes en assignation unique obtenus (1300 lignes pour la SVD en sortie de MatParser).

Voici, par exemple, ce qu'il est maintenant possible de faire depuis MMAlpha. En figure 3.6 se trouve un programme représentant le code Matlab correspondant à l'application Samba traités dans la section 3.3. En figure 3.7 se trouve le programme Alpha qui en est tiré (27 appel a Pip et 8 secondes pour la traduction en assignation unique). On a d'abord utilisé la fonction `readMatlab` pour produire automatiquement un code Alpha puis on a retravaillé ce code Alpha (l'automatisation de ces simplifications après MatParser n'est pas encore au point). J'ai volontairement laissé une erreur que j'avais faite en écrivant le programme Matlab pour illustrer la puissance du formalisme en assignation unique (voir le commentaire de la figure 3.7. On pourra comparer le programme de la figure 3.7 avec le programme écrit manuellement (figure 3.9), les programmes sont différents mais aboutiraient probablement à des circuits très similaires.

La possibilité d'utiliser Matlab comme langage de départ et éventuellement de représentation intermédiaire est une des clés de la diffusion de MMAlpha. Nous allons probablement développer cet aspect appuyé par la collaboration avec l'Université du Québec à Trois Rivières dont toutes les conceptions sont faites à partir des outils Matlab.

### 3.2.5 Preuve algébrique

Le développement des outils de vérification formelle permet aujourd'hui d'augmenter la puissance d'autres logiciels. Avec les outils d'analyse de MMAlpha, il est impossible de simplifier, par exemple, l'expression `0[]+MatchQ_1[i,j]` du programme de la figure 3.7. De manière générale, nous avons de plus en plus de problèmes pour montrer l'équivalence de deux expressions (car nos programmes sont de plus en plus gros). David Cachera a réalisé une interface avec le prouveur de theorème PVS [32], ce qui ouvre véritablement de nouvelles voies pour les traitements pouvant être réalisés avec MMAlpha. En plus des outils d'analyse et de transformation de polyèdres, nous aurons par exemple accès aux transformations utilisant les propriétés algébriques des opérateurs.

Dans [21], nous avons montré comment utiliser PVS pour prouver formellement qu'une fonction de temps est valide, qu'une fonction d'allocation est compatible avec cette fonction de temps et enfin que la manière dont on gérait le mémoire dans chaque processeur était valide. C'est à dire que nous avons réellement prouvé la conception complète! Même si ce travail reste aujourd'hui un peu fastidieux, il est beaucoup plus sûr que la vérification humaine (nous avons du changer plusieurs fois notre fonction mémoire pour en obtenir une correcte). Les possibilités offertes restent liées à la puissance de PVS, et nécessitent aussi quelqu'un qui soit familier avec l'utilisation de PVS.

### 3.2.6 Partitionnement et mémoire

Une des plus grosses lacunes du système MMAlpha aujourd'hui est qu'il ne permet pas d'exprimer (et donc de réaliser) le partitionnement des réseaux systoliques synthétisés. Le partitionnement a toujours été étudié par les concepteurs de réseaux systoliques [111] dans l'optique de pouvoir s'adapter aux contraintes de ressources en terme de matériel disponible. Ce qui semble encore plus critique aujourd'hui est l'adaptation à la bande passante disponible pour les entrées/sorties.

Le partitionnement consiste à exécuter un réseau systolique *virtuel* sur une architecture physique plus petite. On distingue deux types de partitionnement, LPGS (pour Localement

```
system samba : {X,Y | 2<=X<=Y}
               (QS : {I0 | 1<=I0<=X} of real;
                DB : {I0 | 1<=I0<=Y} of real;
                M : {I0,I1 | I0=0; I1=0} of real)
       returns (res_1 : {j | 1<=j<=Y} of real);
var
  MatchQ_1 : {i,j | 1<=i<=X; 1<=j<=Y} of real;
  Above_1 : {i,j | 1<=i<=X; 1<=j<=Y} of real;
  Left_1 : {i,j | 1<=i<=X; 1<=j<=Y} of real;
  Diag_1 : {i,j | 1<=i<=X; 1<=j<=Y} of real;
  M_3 : {i,j | 1<=i<=X; 1<=j<=Y} of real;
let
  MatchQ_1[i,j] = if (QS[i] = DB[j]) then 15[] else -12[];
  Above_1[i,j] =
      case
        { | i=1} : 0[] - 8[];
        { | 2<=i} : M_3[i-1,j] - 8[];
      esac;
  Left_1[i,j] =
      case
        { | j=1} : 0[] - 8[];
        { | 2<=j} : M_3[i,j-1] - 8[];
      esac;
  Diag_1[i,j] =
      case
        { | i=1; 2<=j} | { | 2<=i; j=1} : 0[] + MatchQ_1[i,j];
        { | i=1; j=1} : M[i-1,j-1] + MatchQ_1[i,j];
        { | 2<=i; 2<=j} : M_3[i-1,j-1] + MatchQ_1[i,j];
      esac;
  M_3[i,j] = Max4(0[], Above_1[i,j], Left_1[i,j], Diag_1[i,j]);
  res_1[j] = M_3[X,j];
tel;
```

FIG. 3.7 – *Programme Alpha obtenu après traduction en assignation unique du programme de la figure 3.6, quelques transformations manuelles (changement du domaine des paramètres par exemple) et des transformations correctes de* MMA*lpha (remplacement des appels de sous systèmes par les opérateurs correspondant, élimination des variables intermédiaires introduite par MatParser). On notera qu'il apparaît ici très clairement que la variable* `M` *n'a pas été correctement initialisée dans le programme Matlab de la figure 3.6*

Parallèle Globalement Séquentiel) et LSGP (pour Localement Séquentiel Globalement Parallèle). Le partitionnement LPGS utilise, pour résoudre un problème de taille N, N/P passes d'un réseau pour un problème de taille P. L'architecture du réseau ne change presque pas (on ajoute juste un multiplexeur en début de réseau) et le débit des données reste égal à celui du réseau systolique virtuel. Le partitionnement LSGP fait exécuter à un processeur du réseau physique de taille P le travail de N/P processeurs du réseau virtuel de taille N. L'architecture interne de chaque processeur est donc modifiée, et le débit des données est divisé par N/P. Le partitionnement utilisé en pratique sera une combinaison des deux pour trouver le bon compromis entre la diminution de la bande passante, la diminution du parallélisme et les ressources disponibles. Steven Derrien travaille actuellement à la formulation du problème pour la programmation d'une carte de type Spyder, contenant un gros FPGA et de la mémoire.

La raison pour laquelle le partitionnement n'a pas été implémenté est intrinsèque au problème: le partitionnement fait intervenir des transformations non linéaires (divisions entière et modulo) qu'il est impossible d'exprimer en Alpha. Le problème peut être contourné en partitionnant de manière LPGS un réseau dont tous les paramètres sont affectés (voir figure 3.8). Le résultat n'est pas très satisfaisant car on voudrait garder une conception paramétrée et surtout la taille du code dépend du nombre de partition. D'autre part on ne peut pas exprimer de la même manière le partitionnement LSGP car il faudrait avoir les $\mathbb{Z}$-polyèdres pour exprimer le contrôle d'un processeur du réseau partitionné (après le travaux de Darte et Delosme sur le choix du partitionnement [34], la génération d'un tel contrôle été étudiée par Darte et Schreiber [35] pour Pico).

L'implémentation du partitionnement est donc, pour l'instant, suspendu à la possibilité de manipuler les $\mathbb{Z}$-polyèdres en Alpha, mais il faut aussi être capable de décrire une mémoire en Alpha. Ce problème a été brièvement étudié par Patricia Le Moenner [94]. Nous avons vu en section 2.2 comment représenter des registres en Alpha. Les mémoires auxquelles on accède par une adresse étant généralement synthétisées par des outils spéciaux, il suffit donc de pouvoir exprimer la *fonction* d'une mémoire en Alpha pour considérer que l'on sait utiliser des mémoires dans un programme Alpha (voir [94]). La *synthèse* de mémoire, c'est-à-dire la détection automatique des portions de matériels nécessitant une telle mémoire est beaucoup plus délicate et n'a pas été abordée. Il faut noter que dans le cadre des applications traitées par MMAlpha, les mémoires utilisées sont très spécifiques, il s'agit, soit de tables de look-up (que l'on charge une fois pour toute au début du traitement), soit de files d'attente (pour le partitionnement). La synthèse de files pourra d'ailleurs rejoindre les recherches sur la synthèse de réseau à temps multidimensionnel (voir section 3.1.2).

Le partitionnement est indispensable dans une environnement de compilation/parallélisation tel que MMAlpha. Il existe bien des manière de réaliser le partitionnement. Il ne me semble pas qu'un processus automatique réalisé après la conception du réseau virtuel soit adapté, on pourra plutôt partir d'une spécification initiale déjà partitionnée. D'autre part, les capacités d'intégration des nouveaux matériels sont tels qu'il faudra certainement considérer le partitionnement non comme un moyen de s'adapter aux contraintes de ressources mais comme une manière de trouver les bon compromis entre la bande passante, la parallélisme, la granularité, les différentes ressources etc...

```
 system prodVect (a : {i,j | 1<=i<=120; 1<=j<=10} of integer;
                  b : {i | 1<=i<=10} of integer)
       returns  (c : {i | 1<=i<=120} of integer);
var
  b_mirr1 : {t,p | 1<=t<=10; p=1} of integer;
  b_mirr1par : {t,p,pass | 1<=t<=10; p=1; 1 <= pass <= 12 } of integer;
  b_mirr1Outpar : {t,p,pass | 10<=t<=19; p=10;1 <= pass <= 12 } of integer;
  A : {t,p | p<=t<=p+9; 1<=p<=120} of integer;
  Apar : {t,p,pass | p<=t<=p+9; 1<=p<=10;1 <= pass <= 12 } of integer;
  C : {t,p | p<=t<=p+9; 1<=p<=120} of integer;
  Cpar : {t,p,pass | p<=t<=p+9; 1<=p<=10;  1 <= pass <= 12} of integer;
let
  b_mirr1[t,p] = b[t];
  A[t,p] = a[p,t-p+1];
  Apar[t,p,pass]=A[t+10pass-10,p+10pass-10];


  b_mirr1par[t,p,pass]=
        case
        {| pass=1 }: b_mirr1[t,p];
        {| pass>1 }: b_mirr1Outpar[t+9,10,pass-1];
        esac;

  use  {pass |  1 <= pass <= 12 } prodVectModule[10] (b_mirr1par, Apar)
                                  returns  (Cpar, b_mirr1Outpar) ;

-  C[t,p] = Cpar[t-10(pass-1),p mod 10(pass-1),pass];
  C[t,p]= case
        {| 1<=p<=10}: Cpar[t,p,1];
        {| 11<=p<=20}: Cpar[t-10,p-10,2];
        {| 21<=p<=30}: Cpar[t-20,p-20,3];
        {| 31<=p<=40}: Cpar[t-30,p-30,4];
        {| 41<=p<=50}: Cpar[t-40,p-40,5];
        {| 51<=p<=60}: Cpar[t-50,p-50,6];
        {| 61<=p<=70}: Cpar[t-60,p-60,7];
        {| 71<=p<=80}: Cpar[t-70,p-70,8];
        {| 81<=p<=90}: Cpar[t-80,p-80,9];
        {| 91<=p<=100}: Cpar[t-90,p-90,10];
        {| 101<=p<=110}: Cpar[t-100,p-100,11];
        {| 111<=p<=120}: Cpar[t-110,p-110,12];
        esac;

  c[i] = C[i+9,i];

tel;
```

FIG. 3.8 – *Partitionnement* LPGS *d'un réseau virtuel de 120 cellules en 12 passes d'un réseau physique de 10 cellules. L'équation en commentaire est celle qui remplacerait les 12 branches de* `case` *si les divisions entières étaient autorisées.*

### 3.2.7 La conception hiérarchique

La conception hiérarchique permet au système MMAlpha de traiter de grosses applications en les structurant, mais elle permet surtout de traiter des *morceaux* d'applications ou de *ne pas* traiter des morceaux d'applications et donc de s'intégrer dans un processus de conception conjointe. Ce n'est pas réellement un sujet sur lequel on décide de porter son effort, c'est plutôt une capacité qui se développe grâce au développement de tous les autres mentionnés dans cette section.

Encore une fois, le processus d'ordonnancement est crucial pour la conception structurée, c'est lui qui va représenter ce que peut apporter de plus l'environnement MMAlpha. Par exemple, dans [112], en collaboration avec Aurélien Mauzipo, Daniel Massicote et Patrice Quinton, nous avons utilisé l'ordonnancement pour trouver une architecture pour le filtre de Kalman. L'environnement MMAlpha ne permettait pas de réaliser la synthèse pour une conception aussi complexe, mais l'outil d'ordonnancement a aidé à concevoir l'architecture pipelinée [114].

Dans [74], en collaboration avec Anne-Claire guillou, Daniel Massicote et Patrice Quinton, nous avons montré comment une architecture complexe comme celle de [83] (version améliorée du DLMS) peut être réalisé avec MMAlpha, en laissant au concepteur, le soin d'écrire des parties de code VHDL à la main (ici, il s'agissait d'un arbre complet d'additionneur qui ne pouvait être synthétisé avec MMAlpha). Ce travail permet de prototyper rapidement le coût réel des architecures publiées (on a montré que le matériel additionnel présenté dans [83] représentait un augmentation d'environ 150% en surface par rapport au DLMS).

## 3.3 Un exemple complet: Samba

Cette section illustre la méthodologie de synthèse avec MMAlpha sur un exemple particulier: un algorithme de comparaison de séquences utilisé en particulier pour la comparaison de séquences génomiques. Cet algorithme a été beaucoup étudié par Dominique Lavenier qui a réalisé une architecture prototype nommé Samba [92] (Systolic Accelerator for Molecular Biological Applications). La synthèse présentée ici part du programme Alpha représenté en figure 3.9 et abouti à un code VHDL dont une partie est présentée annexe A.4. Les différentes étapes de la compilation/parallélisation sont présentées dans la section 3.3.2. La synthèse logique et les performances du circuit sont présentées en section 3.3.3.

### 3.3.1 L'algorithme de comparaison de séquence

L'algorithme le plus utilisé dans les logiciels de comparaison de séquences biologiques (les logiciels Blast, FASTA et SSearch) est l'algorithme de Smith et Waterman [152] qui est un algorithme de programmation dynamique. Cet algorithme compare deux séquences de caractères (acides aminés où nucléotide), `DB` (pour Data Base) et `QS` (pour Query Sequence), et tente de retrouver les segments similaires dans les deux séquences contenant éventuellement des omissions, des ajouts ou des mutations. La séquence `QS` est de taille `X` et la séquence `DB` est de taille `Y` avec $Y \gg X$. L'algorithme complet est présenté présenté

```
system sequence: {X,Y| Y>X>2}
         (QS : {i| 1<= i <= X} of integer;
          DB : {j| 1<= j <= Y} of integer)
         returns
         (res : {j| 1<= j <= Y} of integer);
var
M: {i,j|  0<= i <= X; 0<= j <= Y} of integer;
MatchQ: {i,j|  1<= i <= X; 1<= j <= Y} of integer;
let
M[i,j]=case
         {|i=0}: 0[];
         {|i>0; j=0}: 0[];
         {|i>0; j>0}: Max4(0[],M[i,j-1]-8[],
                        M[i-1,j]-8[],M[i-1,j-1]+MatchQ[i,j]);
        esac;
MatchQ[i,j]=if (QS[i]=DB[j]) then 15[] else -12[];
res[j]=M[X,j];
tel;
```

FIG. 3.9 – *Programme Alpha pour l'algorithme de comparaison de séquence Samba étudié dans [73].*

dans [92]. Cet algorithme permet de résoudre plusieurs problèmes, nous nous sommes intéressé à un problème particulier (recherche de similarités entre des séquences d'ADN), cas particulier qui avait aussi été étudié spécifiquement par Dominique Lavenier dans [72]. L'algorithme est représenté sous forme d'équations récurrentes dans les équations (3.1–3.3), le résultat étant la dernière ligne de la matrice `M`. Le programme Alpha (dans lequel le gap `g` a été mis à 8) peut en être immédiatement déduit, il est représenté en figure 3.9.

$$\{i,j \mid i=0\} \cup \{i,j \mid j=0\} \quad : \quad M(i,j) = 0 \tag{3.1}$$

$$\{i,j \mid 1 \leq i \leq X; 1 \leq j \leq Y\} \quad : \quad M(i,j) = \text{Max} \begin{cases} 0 \\ M(i,j-1) - g \\ M(i-1,j) - g \\ M(i-1,j-1) + MatchQ(i,j) \end{cases} \tag{3.2}$$

$$\{i,j \mid 1 \leq i \leq X; 1 \leq j \leq Y\} \quad : \quad MatchQ(i,j) = \begin{array}{l} \text{If } QS[i] = DB[j] \\ \text{Then } 15 \\ \text{Else } -12 \end{array} \tag{3.3}$$

Comme on l'a vu dans la section 3.2.4, on peut aujourd'hui envisager de partir d'une spécification impérative, comme par exemple celle représentée en Matlab dans la figure 3.6. Le programme Alpha sur lequel cette méthodologie serait appliquée serait celui de la figure 3.7, dont la formulation est légèrement plus complexe que celle de la figure 3.9, mais qui représente rigoureusement les mêmes calculs. Nous avons aussi mentionné en section 3.2.2 l'extension du langage nécessaire à la mise en place d'information concernant la

```
system sequence: {X,Y| Y>X>2}
        (QS : {i| 1<= i <= X} of integer[U,2];
         DB : {j| 1<= j <= Y} of integer[U,2])
        returns
        (res : {j| 1<= j <= Y} of integer[S,12]);
var
M: {i,j|  0<= i <= X; 0<= j <= Y} of integer[S,12];
MatchQ: {i,j|  1<= i <= X; 1<= j <= Y} of integer[S,12];
let
M[i,j]=case
        {|i=0}: 0[];
        {|i>0; j=0}: 0[];
        {|i>0; j>0}: Max4(0[],M[i,j-1]-8[],
                     M[i-1,j]-8[],M[i-1,j-1]+MatchQ[i,j]);
       esac;
MatchQ[i,j]=if (QS[i]=DB[j]) then 15[] else -12[];
res[j]=M[X,j];
tel;
```

FIG. 3.10 – *Programme Alpha correspondant à celui de la figure 3.9 mais dans lequel les tailles de bits de chaque signaux ont été indiqués ainsi que le système de codage (*`U` *pour unsigned et* `S pour signed`*)*.

largeur en bit des données. Le travail d'analyse des tailles de bits des signaux ayant déjà été fait par Dominique Lavenier, on aura des variables sur 2 bits non signés (les caractères) et des variables entières sur 12 bits signés (les poids). Le programme sur lequel la conception a lieu réellement est donc le programme représenté en figure 3.10. Le parseur n'ayant pas encore été mis à jour pour afficher les types exacts, nous avons illustré les étapes sur le programme de la figure 3.9.

### 3.3.2 Conception avec MMAlpha

Le programme de la figure 3.9 se trouvant dans le fichier `samba.alpha` du répertoire de travail, il est chargé dans MMAlpha par la commande:

```
In[1]:= load["samba.alpha"];
[sequence]
 Library Loaded
```

On peut, à tout moment, le visualiser en notation tableau comme sur la figure 3.9 avec la commande `ashow` ou en notation standard. L'analyse mentionnée en section 3.1.4 peut être effectuée à tout moment lors de la dérivation:

```
In[2]:= analyze[]

Static Analysis of system sequence
-Checking declaration of variables.
-Checking single assignment rule.
-Checking definitions of output/local variables.
-Checking definition of input variables.
-Checking that input/local variables are used.
-Checking type and domain consistency in the equations:
---equation defining M
---equation defining MatchQ
---equation defining res

Analysis Successful...

Out[2]= True
```

On peut aussi générer du code C pour simuler ce programme pour des valeurs particulières des paramètres. Comme ce programme n'est pas encore ordonnancé, le code C fonctionne sur le principe de l'évaluation à la demande [129]:

```
In[3]:= writeC["samba.c","-p 10 100"];
```

### Uniformisation

FIG. 3.11 – *Graphe de dépendance développé du programme* `samba.alpha`*, avant et après uniformisation*

Ici, l'uniformisation consiste à pipeliner les deux diffusions des entrées `QS[i]` et `DB[j]`. `QS[i]` étant diffusé à tous les points `(i,j)`, on doit le pipeliner suivant la direction `j`. On choisit ici de le pipeliner dans la direction des `j` croissant. `DB[j]` est lui pipeliné dans la direction des `i` croissant. Le graphe de dépendance développé avant et après le pipeline est représenté sur la figure 3.11

```
In[5]:=pipeall["MatchQ","QS.(i,j->i)","QQS.(i,j->i,j+1)"];

In[6]:=pipeall["MatchQ","DB.(i,j->j)","DDB.(i,j->i+1,j)"];
```

Le résultat est le programme ci dessous (la fonction `simplifySystem` effectue une normalisation et tente de rendre convexe les unions de polyèdres). On aurait pu utiliser l'uniformisation automatique (voir section 3.1.1) qui aurait fourni exactement le même résultat à condition que l'on ait aligné `QS[i]` avec la première colonne et `DB[j]` avec la première ligne.

```
In[6]:= simplifySystem[];ashow[]

system sequence :{X,Y | 3<=X<=Y-1}
                 (QS : {i | 1<=i<=X} of integer;
                  DB : {j | 1<=j<=Y} of integer)
       returns   (res : {j | 1<=j<=Y} of integer);
var
  DDB : {i,j | 1<=i<=X; 1<=j<=Y} of integer;
  QQS : {i,j | 1<=i<=X; 1<=j<=Y} of integer;
  M : {i,j | 0<=i<=X; 0<=j<=Y} of integer;
  MatchQ : {i,j | 1<=i<=X; 1<=j<=Y} of integer;
let
  DDB[i,j] =
      case
        { | i=1} : DB[i+j-1];
        { | 2<=i} : DDB[i-1,j];
      esac;
  QQS[i,j] =
      case
        { | j=1} : QS[i];
        { | 2<=j} : QQS[i,j-1];
      esac;
  M[i,j] =
      case
        { | i=0} : 0[];
        { | 1<=i; j=0} : 0[];
        { | 1<=i; 1<=j} : Max4(0[], M[i,j-1] - 8[],
                               M[i-1,j] - 8[],
                               M[i-1,j-1] + MatchQ[i,j]);
      esac;
  MatchQ[i,j] = if (QQS = DDB) then 15[] else -12[];
  res[j] = M[X,j];
tel;
```

**Ordonnancement et allocation**

L'ordonnancement est effectué par la commande `schedule` à laquelle il faut indiquer quelle durée on associe à chaque variable (voir section 3.1.2):

```
In[7]:= schedule[scheduleType->sameLinearPart,
                 durations->{0,0,0,1,1,0,0}];
Checking options...
Dependence analysis...
Duration not null for :{DDB, QQS}
Building LP...
LP: 179      variables, 204 Constraints
Writing file for PIP....
Writing line 100
Writing line 200
Solving the LP...
Version D.1
cross : 3535402, alloc : 1, compa : 0
n 1 u 143"' s 1"'
Shift coef: 0
Total execution Time: 2 Y
T_QS{i, X, Y} = 0
T_DB{j, X, Y} = 0
T_res{j, X, Y} = j + X
T_DDB{i, j, X, Y} = i + j
T_QQS{i, j, X, Y} = i + j
T_M{i, j, X, Y} = i + j
T_MatchQ{i, j, X, Y} = i + j
```

La phase d'allocation est réalisée automatiquement par complétion unimodulaire de la fonction de temps. Après renommage des indices, le programme indique clairement où et sur quel processeur chaque calcul est effectué. On voit en particulier, que des registres ont effectivement été placés sur les chemins de données de `QS` et `DB`.

```
In[8]:= applySchedule[];renameIndices[{"t","p"}];
```

```
In[9]:= simplifySystem[];ashow[]

system sequence :{X,Y | 3<=X<=Y-1}
                (QS : {i | 1<=i<=X} of integer;
                 DB : {j | 1<=j<=Y} of integer)
```

```
       returns  (res : {j | 1<=j<=Y} of integer);
var
  DDB : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
  QQS : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
  M : {t,p | p<=t<=p+Y; 0<=p<=X} of integer;
  MatchQ : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
let
  DDB[t,p] =
      case
        { | p=1} : DB[t-1];
        { | 2<=p} : DDB[t-1,p-1];
      esac;
  QQS[t,p] =
      case
        { | t=p+1} : QS[t-1];
        { | p+2<=t} : QQS[t-1,p];
      esac;
  M[t,p] =
      case
        { | p=0} : 0[];
        { | t=p; 1<=p} : 0[];
        { | p+1<=t; 1<=p} : Max4(0[], M[t-1,p] - 8[],
                                  M[t-1,p-1] - 8[],
                                  M[t-2,p-1] + MatchQ[t,p]);
      esac;
  MatchQ[t,p] = if (QQS[t,p] = DDB[t,p]) then 15[] else -12[];
  res[j] = M[j+X,X];
tel;
```

**Pipeline des entrées**

On voit aussi sur le programme précédent que les données de DB sont toutes entrées dans le premier processeur (p=1) alors que les données de QS sont entrées dans tous les processeurs. Si l'on décide que toutes les entrées doivent être faites dans le même processeur, on doit pipeliner les entrées. Cela peut être fait en routant l'expression concernée (QS[t-1]) jusqu'au “bord” du réseau, c'est à dire jusqu'à p=1.

```
In[10]:=pipeIO["QQS","QS.(t,p->t-1)",
               "QQS_In.(t,p->t+2,p+1)",
               "{t,p| p>=1}"];
```

Le chargement des données de `QS` se fait maintenant en deux temps, d'abord un pipeline dans le réseau avec deux registres, puis, une fois arrivé dans le bon processeur (`t=p+1`), le coefficient est stocké dans un registre (la commande `asketch` permet de n'afficher que certaines variables du programme Alpha).

```
In[11]:= simplifySystem[];asketch[{"QQS","QQS_In"}];

system sequenceOnlyQQS :{X,Y | 3<=X<=Y-1}
                       ( )
       returns         ( );
var
  QQS : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
  MatchQ : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
  QQS_In : {t,p | 2p-X+1<=t<=p+1; 1<=p} of integer;
let
  QQS[t,p] =
      case
        { | t=p+1} : QQS_In;
        { | p+2<=t} : QQS[t-1,p];
      esac;
  MatchQ[t,p] = if (QQS[t,p] = DDB[t,p]) then 15[] else -12[];
  QQS_In[t,p] =
      case
        { | p=1} : QS[-t+3];
        { | 2<=p} : QQS_In[t-2,p-1];
      esac;
tel;
```

**Dérivation RTL**

La phase suivante est complètement automatique. On génère d'abord du Alpha0. On notera que deux signaux de contrôle ont été créés, un pour le chargement de `QQS` (lorsque `t=p+1`), l'autre pour le début du calcul de `M` (`t>=p+1`). Ces signaux ont ensuite été pipelinés jusqu'au premier processeur du réseau.

```
In[12]:= toAlpha0v2[];
Time index: {1}  space indices: {2}
Calling spaceTimeDecomposition[];
Calling makeAllMuxControl[];
  Equation of QQS_In...
  Equation of DDB...
  Equation of QQS...
```

```
      is in ST form
       Adding multiplexer.
     Equation of M...
      is in ST form
       Adding multiplexer.
     Equation of MatchQ...
     Equation of res...
   Calling pipeAllControl[];
     Pipelining control for: QQS_ctl1
        From dimension 2 To dimension 1
        Control generated in cell: {p | p=1; 3<=X<=Y-1}
     Pipelining control for: M_ctl2
        From dimension 2 To dimension 1
        Control generated in cell: {p | p=1; 3<=X<=Y-1}
   Calling decomposeSTdeps[];
     In equation of M_ctl2P, adding a local variable: M_ctl2P_reg1
     In equation of QQS_ctl1P, adding a local variable: QQS_ctl1P_reg2
     In equation of QQS_In, adding a local variable: QQS_In_reg3
     In equation of DDB, adding a local variable: DDB_reg4
     In equation of QQS, adding a local variable: QQS_reg5
     In equation of M, adding a local variable: M_reg6
     In equation of M, adding a local variable: M_reg7
     In equation of M, adding a local variable: M_reg8
    Decomposing the space/time dependencies
     Space/time dependency to decompose in M_reg8
       Adding local variable M_reg8loc
     Space/time dependency to decompose in M_reg7
       Adding local variable M_reg7loc
     Space/time dependency to decompose in DDB_reg4
       Adding local variable DDB_reg4loc
     Space/time dependency to decompose in QQS_In_reg3
       Adding local variable QQS_In_reg3loc
     Space/time dependency to decompose in QQS_ctl1P_reg2
       Adding local variable QQS_ctl1P_reg2loc
     Space/time dependency to decompose in M_ctl2P_reg1
       Adding local variable M_ctl2P_reg1loc
   Calling makeSimpleExpr[];
   TSep1 generated ...
   TSep2 generated ...
   Calling makeInputMirrorEqus[];
     Adding mirror equation for input QS
     Adding mirror equation for input DB
```

Le programme Alpha0 peut être vu en annexe A.2. Il est ensuite structuré en un programme AlpHard: les différentes *régions* sont identifiées (une région est un ensemble de processeurs qui ont exactement le même fonctionnement), puis le programme est structuré. La cellule 4 (p=X+1) est supprimée car elle ne comporte que des entrées, elle ne produit aucun résultat (il s'agit de données qui ne sont plus utilisées, en fin de pipeline). Le `module` (qui correspond au circuit) est isolé de l'`interface`.

```
In[13]:= alpha0ToAlphard[];
-building Interface
checking form of output equation
-structuring Module sequenceModule
--Determining the different cell types...
   Please wait...
   34 region(s) found  before    simplification...
   Simplify identical regions ...
   4 type(s) of cell have been    identified, their domains are:
{p,X,Y | p=1; 3<=X<=Y-1}
{p,X,Y | 2<=p<=X; 3<=X<=Y-1}
{p,X,Y | p=X+1; 3<=X<=Y-1}
{p,X,Y | p=0; 3<=X<=Y-1}
--Generating Controler ...
--Building cell 1 present on :
{p,X,Y | p=1; 3<=X<=Y-1}
   Parameters of Cell :cellsequenceModule1
   input: {DB_mirr1, QS_mirr1, M_reg8loc, M_reg7loc, M_ctl2P_Init_In,
  QQS_ctl1P_Init_In}
   output: {M1, DDB1, QQS_In1, QQS_ctl1P1, M_ctl2P1}
--Building cell 2 present on :
{p,X,Y | 2<=p<=X; 3<=X<=Y-1}
   Parameters of Cell :cellsequenceModule2
   input: {M_reg8loc, M_reg7loc, DDB_reg4loc, QQS_In_reg3loc,
  QQS_ctl1P_reg2loc, M_ctl2P_reg1loc}
   output: {M2, DDB2, QQS_In2, QQS_ctl1P2, M_ctl2P2}
--Building cell 3 present on :
{p,X,Y | p=X+1; 3<=X<=Y-1}
   Parameters of Cell :cellsequenceModule3
   input: {M_reg8loc, M_reg7loc, DDB_reg4loc, QQS_In_reg3loc,
  QQS_ctl1P_reg2loc, M_ctl2P_reg1loc}
   output: {}

alpha0ToAlphard::emptyCell:
    Warning: cell cellsequenceModule3
     has no output hence this   cell is not generated
--Building cell 4 present on :
{p,X,Y | p=0; 3<=X<=Y-1}
   Parameters of Cell :cellsequenceModule4
   input: {}
```

```
system sequence :{X,Y | 3<=X<=Y-1}
                (QS : {i | 1<=i<=X} of integer;
                 DB : {j | 1<=j<=Y} of integer)
       returns  (res : {j | 1<=j<=Y} of integer);
var
  DB_mirr1 : {t,p | 2<=t<=Y+1; p=1} of integer;
  QS_mirr1 : {t,p | -X+3<=t<=2; p=1} of integer;
  M : {t,p | p<=t<=p+Y; 0<=p<=X} of integer;
let
  DB_mirr1[t,p] = DB[t-1];
  QS_mirr1[t,p] = QS[-t+3];
  res[j] = M[j+X,X];
  use  sequenceModule[X,Y] (DB_mirr1, QS_mirr1) returns  (M) ;
tel;
```

FIG. 3.12 – *Interface du programme AlpHard obtenu à partir du programme de la figure 3.9 après conception. Le programme AlpHard complet est en annexe A.3.*

```
   output: {M4}
adding local variables in library for isolating output pins
adding local variable: Mloc1
adding local variable: DDBloc2
adding local variable: QQS_Inloc3
adding local variable: QQS_ctl1Ploc4
adding local variable: M_ctl2Ploc5
adding local variable: Mloc1
adding local variable: DDBloc2
adding local variable: QQS_Inloc3
adding local variable: QQS_ctl1Ploc4
adding local variable: M_ctl2Ploc5
sequence removed from $library
sequence removed from $result
ControlsequenceModule added to library.
cellsequenceModule1 added to library.
cellsequenceModule2 added to library.
cellsequenceModule4 added to library.
sequenceModule added to library.
sequence added to library.
```

L'`interface` contient uniquement les équations permettant de faire la connexion entre la fonction de départ du programme de la figure 3.9 et le module dont les entrées sont ordonnancées et placées. L'`interface` est représentée sur la figure 3.12

Le programme AlpHard présenté dans l'annexe (section A.3) peut être directement traduit en VHDL une fois que l'on a affecté des valeurs aux paramètres et que l'on a supprimé l'interface (qui ne peut être synthétisée).

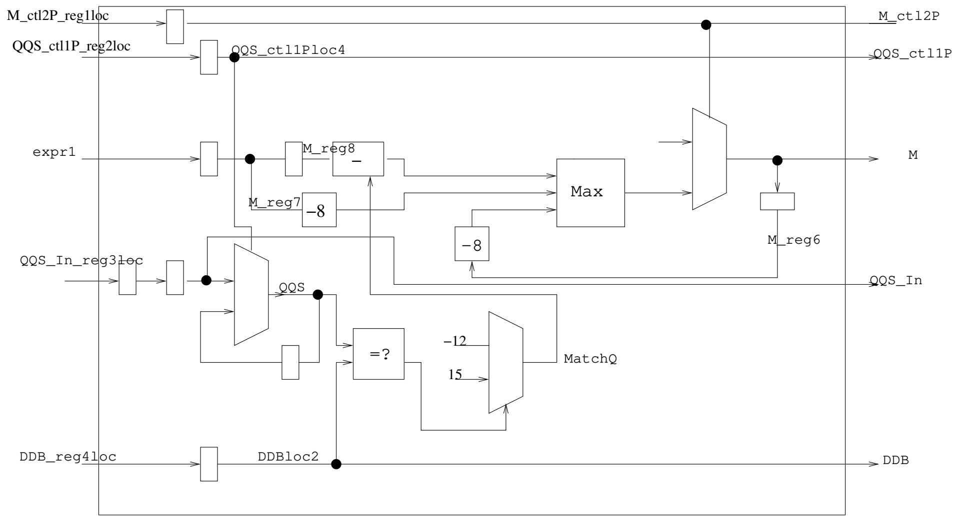


FIG. 3.13 – *Schema d'une cellule du réseau systolique obtenu pour Samba*

```
In[12]:=
fixParameter["X",10];
fixParameter["Y",100];
getSystem["sequenceModule"];
alphaToVhdl[];

Please wait...

ControlsequenceModule was recognized as a Controller.
0.75 Second needed to translate ControlsequenceModule
cellsequenceModule1 was recognized as a Cell.
2.34 Second needed to translate cellsequenceModule1
cellsequenceModule2 was recognized as a Cell.
1.99 Second needed to translate cellsequenceModule2
cellsequenceModule4 was recognized as a Cell.
0.13 Second needed to translate cellsequenceModule4
sequenceModule was recognized as a Module.
4.19 Second needed to translate sequenceModule
```

Le code VHDL est présenté en annexe, section A.4. L'architecture obtenue pour une cellule est représentée en figure 3.13

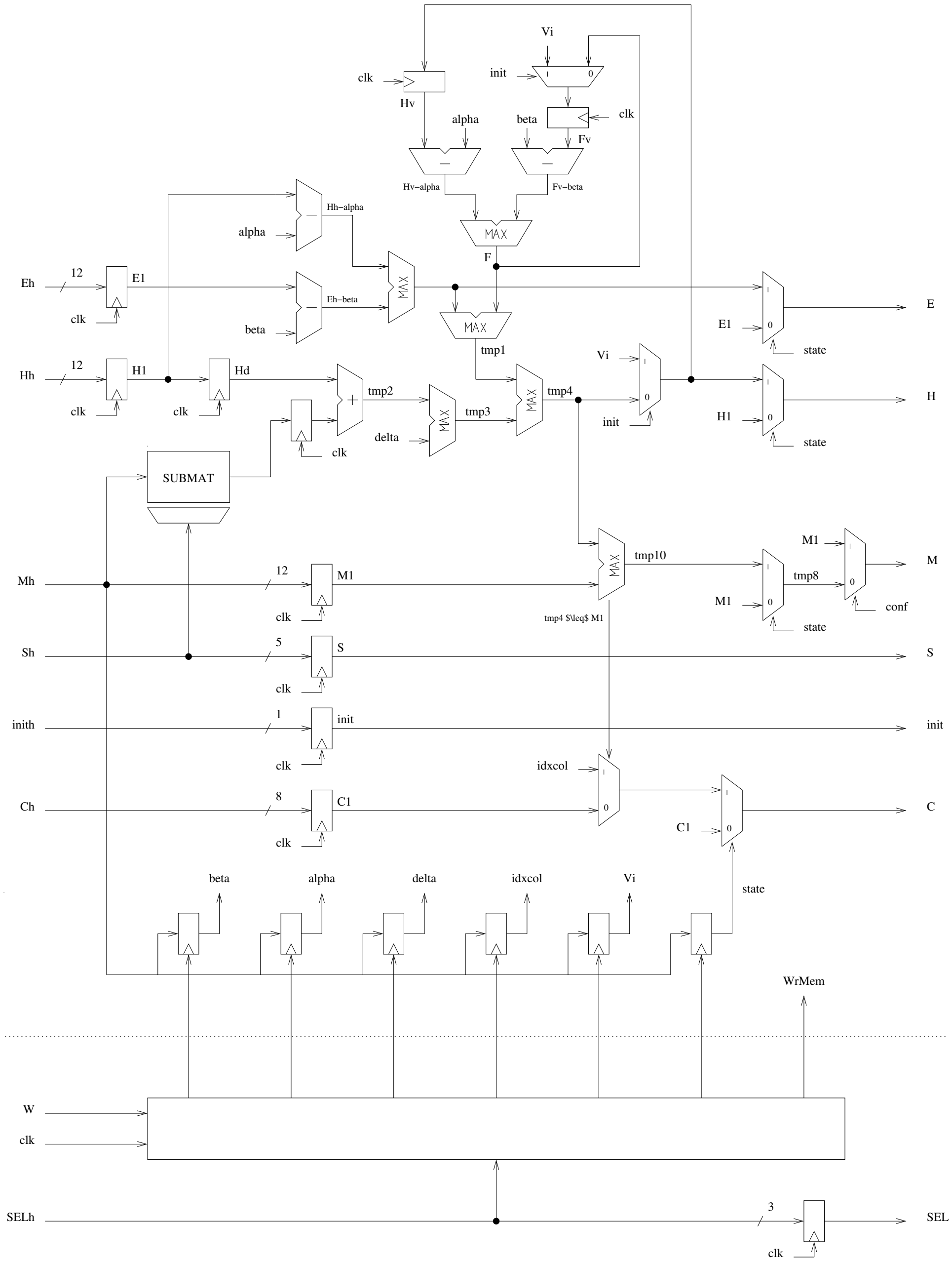


FIG. 3.14 – *Schéma d'une cellule de la machine Samba réalisée par Dominique Lavenier [92]. On pourra comparer cette architecture que l'on a obtenue, représentée en figure 3.13, sachant que la cellule ici présente est plus complexe car plus générale. les paramètres (*`alpha, beta`*, ... ) servant à différencier parmi plusieurs traitements.*

### 3.3.3 Résultats et performances

Le script composé de toutes les commandes décrites dans la section 3.3.2 peut être exécuté d'un seul coup. Le temps d'exécution du script un fois mis au point est de 2mn sur une station Sun Sparc 20. Le temps de mise au point peut varier entre un demi-heure et quelques jours suivant pour un concepteur connaissant bien MMAlpha. La conception présentée ici n'est bien sur pas la meilleure possible ; on pourrait encore optimiser la conception en passant plus de temps (on pourrait par exemple, avancer légèrement le chargement de `QQS` et utiliser le canal de `QQS_in` pour charger `DDB`, en effet, on voit sur les programmes affichés page 58 que ces deux signaux ont des domaines dont l'intersection se limite a `{t,p | t=P+1}`, `QQS_in` vivant avant et `DDB` vivant après).

La synthèse du VHDL généré a été réalisée avec Synopsys (en 1mn). Le temps de cycle estimé d'après la netlist obtenue est de 28ns (36 Mhz). La netlist a ensuite été transmise aux outils Xilinx (XDM) qui, en 22 minutes, a généré un bitstream utilisant 1500 slices (16% du Virtex800), soit un équivalent de 25500 portes. Selon cette estimation, on pourrait implémenter jusqu'à 60 cellules sur le virtex (à titre de comparaison, 4 processeurs de la machine samba pouvait être implémentés sur une puce à l'époque de sa réalisation). Le temps d'une ré-exécution de la conception complète avec changement d'un paramètre peut être estimé a 26 minutes dont 5 % pour la partie haute avec MMAlpha.

# Chapitre 4

# Synthèse et perspectives

Ce chapitre présente une synthèse des résultats obtenus avec le logiciel MMAlpha et ouvre quelques perspectives (section 4.3). On y trouve aussi les autres travaux réalisés autour de MMAlpha (section 4.1), et une revue rapide des travaux et outils proches de MMAlpha (section 4.2).

## 4.1 Autres travaux autour de MMAlpha

Cette section décrit brièvement des travaux plus spécifiquement dirigés vers divers domaines applicatifs (codage, analyse d'image, traitement du signal). Ces travaux ont été menés en collaboration avec des spécialistes de ces domaines, ils proposent de nouvelles solutions architecturales en utilisant les techniques systoliques ou explicitement la méthodologie de synthèse avec MMAlpha.

### 4.1.1 Codage

Avec le développement d'Internet, le codage et le cryptage sont des domaines en constant développement. En collaboration avec Yannick Saouter, nous avons étudié le calcul dans les anneaux de polynômes sur les corps de Galois et puis nous avons proposé une amélioration du réseau de Kung pour le calcul du PGCD.

#### Bases de Gröbner

Les bases de Gröbner ont été développées en 1965 par Bruno Buchberger [18] dans le cadre des calculs de classes de résidus d'anneaux polynomiaux. Ce concept permet de nombreuses applications en matière de calcul symbolique, mais aussi de géométrie algorithmique non linéaire, de preuve automatique en géométrie et de théorie de la modélisation. Les bases de Gröbner sont des bases qui permettent de décomposer des polynômes dont les coefficients sont dans un corps de Galois. Au cours des dernières années, plusieurs applications des bases de Gröbner sont apparues dans le domaine du décodage des codes correcteurs d'erreurs (codes de Reed-Solomon [81], BCH et codes à résidus quadratiques, codes abéliens [24], codes cycliques en général). Dans [143, 142], nous proposons une architecture systolique pour le calcul des bases de Gröbner en temps réel. L'architecture a été décrite en AlpHard, puis traduite en VHDL, simulée et synthétisée (la figure 4.1 présente le réseau pour 5 cellules). Nous avons aussi évalué le débit qu'un tel réseau autoriserait pour une application particulière (le décodage des codes de Jensen par la méthode de Chabanne) et il est apparu que ce débit était faible avec la technologie disponible en 1998 ; cependant pour l'application que nous avons étudiée, c'est la seule solution envisageable en pratique du fait de la complexité du processus.

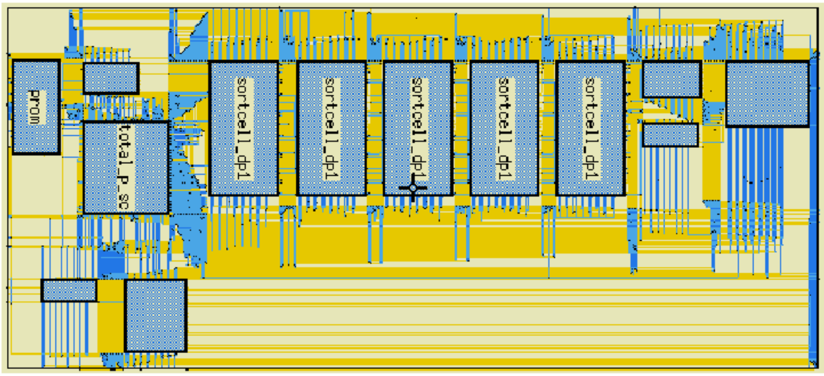


FIG. 4.1 – *Plan de masque du circuit pour les bases de Gröbner, synthèse structurée à 5 cellules, placement automatique des blocs (surface : 1,6 mm2 ).*

**PGCD**

L'algorithme d'Euclide pour calculer le PGCD de deux nombres est toujours employé, en particulier pour calculer le PGCD de polynômes. Brent et Kung avait proposé une architecture systolique pour ce problème [17]. Le PGCD de polynôme est souvent utilisé dans les algorithme de codage et de décodage, en particulier dans le décodage des codes de Reed-Solomon qui est utilisé, par exemple dans certain niveau de Mpeg. Nous avons proposé pour ce problème une architecture qui améliore les performance de [17] et [19]. Cette architecture a fait l'objet d'une tentative de dépôt de brevet. L'organisme consulté par le CNRS (le Fist) a déconseillé le dépôt en recommandant de démontrer l'intêret industriel auparavant.

### 4.1.2 Estimation de mouvement

En collaboration avec Étienne Mémin, nous avons étudié l'utilisation de MMAlpha pour l'estimation de mouvement sur une séquence d'image par un algorithme de minimisation globale d'énergie. Les algorithmes utilisé actuellement pour ce type de traitement sont basés sur des techniques de block-matching et ne permettent pas d'avoir une approche globale de l'image. Les techniques à base de minimisation d'énergie sont beaucoup plus efficaces [12, 27, 69] mais beaucoup plus coûteuse en temps de calcul. Lors du stage de DEA de Guillaume Giacobi, nous avons effectué la dérivation de code AlpHard à partir de spécifications haut niveau pour les algorithmes d'estimation de mouvement du type IRLS (Iterative Reweighted Least Square [69]) consistant en une série de minimisation itératives du type Jacobi ou Gauss-Seidel, entre lesquelles sont remis à jour les les paramètres grâce à des estimateurs robustes [12, 78] qui permettent de conserver les irrégularité (atténuer le "flou" introduit par les méthodes de minimisation itératives). Ces architectures étaient relativement naïves (un pixel par processeurs) mais leur évaluation a permis de chiffrer réellement leur taille. En pratique, il était impossible de les réaliser pour des tailles d'images supérieures à $16 \times 16$ pixels [108].

Le résultat plus intéressant de cette évaluation architecturale est qu'elle nous ont amenée à proposer un nouvel algorithme (baptisé FAJM pour Full Alternate Jacobi Method) dans lequel les poids sont remis à jour lors de chaque itération de Jacobi. La convergence de ce nouvel algorithme a été prouvée et ses performances se révèlent bien meilleures, tant en qualité qu'en rapidité, que toutes les méthodes précédentes. Ces travaux ont été publié d'abord à la conférence Iwipipa99 [106], puis une version article est apparue dans IJPRAI [107]. Les évaluations de performance des algorithmes et des architectures associées ont était faites pour l'estimation de mouvement et la restauration d'images bruitées, mais la méthode FAJM s'applique en fait à tous les algorithmes qui tentent de minimiser une énergie par des méthodes itératives. Nous travaillons actuellement, en collaboration avec Patrick Perez et Étienne Mémin à une comparaison plus systématique de cette méthode avec celles existant actuellement.

Le fait que cette méthode ait été découverte en "tâtonnant" pour essayer d'adapter un algorithme au matériel est intéressant. En s'ouvrant sur la frontière entre deux domaines de recherche, on s'expose à n'être reconnu par aucun, mais c'est certainement de là que viennent les résultats les plus efficaces.

### 4.1.3 Problème du chemin algébrique

Le problème du chemin algébrique (Algrebraic Path Problem: APP) est résolu par un algorithme qui unifie plusieurs problèmes provenant de divers domaines: l'inversion de matrice, la recherche du plus court chemin entre toutes les paires de sommets dans un graphe valué ou la clôture transitive d'une relation. Depuis longtemps le problème du chemin algébrique sert de d'algorithme "test" pour les différentes méthodologies de synthèse systolique. La liste des publications proposant des réseaux pour cet algorithme est impressionnante. Nous avons étudié l'implémentation de cet algorithme sur réseau systolique bidimensionnel [155, 46] puis sur réseau unidimensionnel [133].

En collaboration avec Clementin Tayou Djamegni, Patrice Quinton, et Sanjay Rajopadhye, nous avons montré dans [155, 46] que l'on pouvait unifier les contributions concernant les réseaux systoliques bidimensionnels pour ce problème, en les exprimant sous forme de transformations d'équations récurrentes. Nous avons proposé une nouvelle uniformisation qui nous a mené à un ordonnancement affine par morceaux dont le temps total est de $4n$ en incluant les temps de chargement et de déchargement du réseau ($n$ étant la taille du problème). Nous avons aussi proposé, grâce à une méthode d'allocation relativement complexe, une architecture comprenant $n^2/3$ processeurs, ce qui est une amélioration des bornes connues jusqu'alors.

Nous avons ensuite étudié cet algorithme avec Claude Tadonki et Sanjay Rajoapdhye dans [133]. Ici, nous avons cherché une implémentation sur un réseau linéaire, ce qui implique que la taille de la mémoire disponible sur chaque processeur dépend de la taille du problème. Nous avons proposé un algorithme original pour la réalisation de l'APP sur réseau linéaire ($n$ processeurs et $3n^2$ tops pour un problème de taille $n$) qui possède des propriétés intéressantes pour la réalisation en matériel: son contrôle est simple et toutes les mémoires peuvent êtres implémentées à base de files d'attente. Ce réseau peut être partitionné très facilement, et nous avons montré que le partitionnement *augmentait* l'efficacité du réseau. Enfin, c'est sur cet algorithme que nous avons commencé à expérimenter les possibilités du prouveur PVS en collaboration avec David Cachera. Dans [21], nous avons retravaillé l'implémentation en décrivant très précisément le fonctionnement du réseau grâce à une fonction décrivant la mémoire. Nous avons alors prouvé formellement la validité de cette implémentation générant les demande de preuves PVS à partir du programme Alpha de l'APP.

### 4.1.4 Filtrage adaptatif

En collaboration avec le Laboratoire de Génie Électrique de l'université du Québec à Trois Rivière, nous avons étudié l'implémentation matérielle d'algorithmes de filtrage adaptatifs pour le traitement du signal. Ces algorithmes sont très fréquemment utilisés dans les applications de télécommunication pour reconstruire des signaux bruités lorsqu'ils traversent des canaux. Nous avons étudié un algorithme utilisant le filtrage de Kalman en utilisant l'ordonnancement de MMAlpha [115, 112] puis nous avons proposé des synthèses de réseaux systoliques pour l'algorithme DLMS [74].

Le filtrage de Kalman est largement utilisé pour l'adaptation de canaux. Dans [115], en collaboration avec Aurélien Mauzipo, Daniel Massicote, et Patrice Quinton nous avons proposé une architecture pour cet algorithme. Cette architecture avait été conçue en utilisant le logiciel MMAlpha pour la phase d'ordonnancement. Dans [112], nous avons adapté cette architecture à une variante du filtrage de Kalman qui est plus robuste face aux effets de la quantification (Square Root Covariance Kalman Filtering). Cette architecture a ensuite été décrite en VHDL, simulée et synthétisée. Pour une taille de matrice de $M = 3$, le nombre de transistors était estimé à 750 000 et la fréquence était estimée à 3Mhz, limitée en grande partie par le diviseur de Newton Raphson.

Plus récemment, nous avons étudié une méthode plus classique pour résoudre le même problème qui utilise un calcul pipeliné des moindres carrés: l'algorithme DLMS (Delayed

| valeur de N | vitesse d'horloge | place du circuit | temps de placement et routage |
|---|---|---|---|
| 8 | 72.8 MHz | 10% | 10 mn 40 s |
| 10 | 71.5 MHz | 14% | 14 mn 41 s |
| 16 | 73.1 MHz | 23% | 28 mn 13 s |
| 32 | 72.0 MHz | 44% | 68 mn 26 s |

TAB. 4.1 – *Résultats de la conception du* DLMS *obtenue avec* MMA*lpha pour diverse valeurs de $N$. On voit bien les caractéristiques architecturales des réseaux systoliques: lorsque la taille du réseau augmente, le temps de cycle ne change pas, la taille du réseau augmente linéairement.*

Least Mean Square). Le logiciel MMAlpha étant en état de faire une synthèse complète, nous avons pu faire semi-automatiquement la dérivation de réseaux publiés dans la littérature ([116]) ainsi que l'évaluation des performances des architectures obtenues grâce à la synthèse du VHDL généré, ces résultats sont présentés en table 4.1. Nous avons également proposé une évaluation pour un algorithme plus récent [83] proposant une optimisation du DLMS en réduisant le délais entre les résultats et leur utilisation pour l'adaptation des poids. Cette évaluation a permis de chiffrer à 150% l'augmentation de surface liée à cette optimisation (le temps de cycle restant inchangé).

## 4.2 Travaux et outils similaires à MMAlpha

Cette section fait un rapide survol bibliographique des travaux qui ont rendu possible le développement de MMAlpha et de la méthodologie qui y est associée. Nous présentons aussi les outils qui ont été développés pour la compilation/parallélisation. Nous avons classé ces travaux en trois groupes : les fondations, la parallélisation automatique et la compilation de circuits.

#### Les fondations

S'il existe un article fondateur, c'est sans aucun doute celui de Karp, Miller et Winograd, intitulé "The Organization of Computations for Uniform Recurrence Equations" [82]. Cet article formalise le concept d'équation récurrente que nous utilisons toujours avec le langage Alpha et propose pour la première fois une technique d'ordonnancement cyclique. Ces travaux sont constamment référencés, aussi bien dans le domaine de la parallélisation automatique que dans celui de la synthèse de circuits. En ce qui concerne le VLSI, les travaux de Kung et Leiserson [89, 88, 97] sont aussi extrêmement importants car ils ont ouvert la porte aux conceptions régulières automatisées. Citons aussi les travaux de Lamport [91] qui ont joué un peu ce rôle de déclencheur dans le domaine de la parallélisation automatique.

La "génération" suivante vit apparaître des méthodologies pour la synthèse de réseaux systoliques à partir d'équations récurrentes. Les travaux de Quinton [127, 128] et Rao [139] sont particulièrement connus, mais on peut aussi citer Rajopadhye [134] et Moldovan [109].

### Parallélisation automatique

Le monde de la parallélisation automatique a été très influencé par les travaux des partenaires du PRC $C^3$. L'équipe de Paul Feautrier a proposé beaucoup d'articles fondamentaux [63, 62, 66, 65, 64]. L'équipe des Mines de Fontainebleau apporta les expériences du projet Pips [80, 79, 6, 4, 30]. Guy René Perrin a encadré les activités de Strasbourg se situant, un peu comme à l'Irisa, entre la parallélisation de boucles et la synthèse systolique [29, 121, 28]. Le Lip a apporté des résultats plus théoriques sur la modélisation et la résolution de plusieurs problèmes spécifiques [34, 44, 33, 31, 37, 13, 144, 36, 38] et l'Irisa travaillait aussi sur ce sujet [26, 25].

L'activité américaine a été très importante aussi, mais avec une optique plus pragmatique. Pendant longtemps, peu de travaux se sont intéressés au modèle polyédrique. Parmi les travaux importants pour nous, citons ceux de Bill Pugh [123, 124, 84], Monica Lam [164, 90, 3], Mikael Wolfe [165, 166], Rob Schreiber [148], Keshav Pingali [122], J. Ramanujam [136, 137, 138] Uptal Bannerjee [9, 10], David Padua [11]. Ailleurs dans le monde, on retiendra les travaux de Lenguaer [99], de Thomas Brandes [14, 16]

De ces travaux ont émergé de nombreux prototypes de paralléliseurs. L'un des plus puissants étant sans doute Pips [79, 30] grâce à son analyse polyédrique interprocédurale. Le paralléliseur Paf [119] implémente les travaux de Feautrier. Loopo [71] sert de plateforme pour tester différents algorithmes pour chaque étape de la parallélisation. Adaptor [16] a beaucoup été utilisé comme "back end" par des outils de parallélisation. Beaucoup de compilateurs existent aux États-Unis, (Polaris [118], Paradigm [8], Promis [55], Nascent [167]), mais peu utilisent les techniques polyédriques. Suif [57] est peut-être l'outils le plus puissant actuellement. Le travail de Saman Amarasinghe [2] est sans doute celui qui se raproche le plus de la communauté polyédrique.

### Synthèse de circuits

Il est très difficile de faire une revue de tous les travaux sur la synthèse de circuits, le domaine est beaucoup trop vaste. Nous nous contentons ici de mentionner les travaux liés au modèle polyédrique. Le livre de Patrice Quinton et Yves Robert [132] fait le tour de la question en 1989. Depuis, de nombreux outils sont apparus: Diastol [67], Advis [110], Decomposer [77], Arrest [20], Compar [157], Decomp [160]. La plupart de ces outils n'existe plus aujourd'hui.

L'activité de recherche concernant la compilation/parallélisation de circuits a continué dans quelques groupes (en plus du projet Api/Cosi bien sur). À l'Université de Reading, Graham Megson a continué à étudier les techniques d'uniformisation avec l'outil Sade [70] et la parallélisation automatique [104]. Lothar Thiele a travaillé sur les systèmes embarqués et a constamment utilisé le modèle polyédrique [54, 75]

Les outils les plus importants comparablesx à MMAlpha résultent des projets menés par Francky Catthoor, Ed Depretterre et Rob Schreiber. L'Imec, sous la houlette de Francky Catthoor, produit Cathedral, Atomium [23] puis Acropolis [22]. Cette méthodologie est avant tout axée sur la mémoire [23] (pour la consommation en particulier) : ils ont breveté une méthodologie de conception permettant l'optimisation de la mémoire. Du fait de son statut intermédiaire Université/compagnie et grâce à l'obstination de Francky Catthoor, l'Imec a pu produire de gros efforts. Ces recherches sont certainement parmi les plus avancées dans ce domaine, mais les résultats sont souvent confidentiels.

Ed Deprettere a supervisé les projets Hifi [41] puis Compaan [86, 85] dans lesquels beaucoup de techniques et d'outils polyédriques sont utilisés (Pip,Polylib en particulier). Ces outils sont dédiés aux systèmes embarqués et sont développés en collaboration avec les laboratoires de recherche de Phillips. Une des particularités intéressantes est qu'ils utilisent tous un passage en assignation unique avant de faire la parallélisation, ce qui les rend très proches du système MMAlpha.

Les laboratoires de Hewlett Packard, grâce en particulier à Rob Schreiber, se sont lancé dans le développement de Pico citePico. La classe d'algorithmes visé est assez restreinte (algorithmes utilisés en interne chez HP) ainsi que la classe d'architecture cible, mais cet outil a été rapidement développé et a produit automatiquement des conceptions qui ne sont que 10% moins bonnes que celles faites à la main [56]. Pico intègre les techniques de partitionnement et d'allocation développées grâce au modèle systolique [35].

## 4.3 Synthèse et perspectives

Cette section fait une synthèse des travaux qui correspondent aux années de recherches que j'ai effectuées à l'Irisa après ma thèse, puis ouvre des perspectives pour la suite des recherches en compilation/parallélisation de circuits.

### 4.3.1 Synthèse

On peut faire une synthèse des résultats à trois niveaux. D'abord en ce qui concerne les résultats des conceptions effectuées avec MMAlpha, ensuite en ce qui concerne les résultats plus fondamentaux obtenus avec le développement de la méthodologie associée à MMAlpha et enfin on peut faire un bilan de l'expérience logicielle menée pendant ces recherches. J'insisterai un peu plus sur ce dernier point ici car il n'a pas été évoqué dans le reste du document.

#### Résultats des circuits synthétisés

Les résultats et performances des circuits synthétisés sont prometteurs mais il certaines traitements demandent encore à être évvalués (en particulier, la conception de l'interface pour exécuter les circuits résultants sur un FPGA et la conception partitionnée). Cependant, dans le cas où le partitionnement n'est pas nécessaire, on peut constater plusieurs résultats positifs:

– Le temps de conception avec MMAlpha est faible, il faut quelques heures pour générer du code VHDL alors que l'écriture à la main demanderai probablement beaucoup plus de temps.
– La conception est fiable, les simulations faites sur le VHDL généré se révèlent toujours cohérentes avec les simulations fonctionnelles effectuées sur le code Alpha.
– Les architectures générées sont relativement efficaces. Bien qu'il soit difficile de les comparer avec des architectures générées à la main (car il faudrait prendre en compte

les compétences du concepteur), on peut remarquer qu'en général le temps de cycle est relativement faible et, tant que l'on reste en virgule fixe, la taille des FPGA actuels permet d'implémenter des circuits non triviaux.

Ces trois remarques me poussent à penser que l'environnement MMAlpha peut aujourd'hui être utilisé comme générateur de code VHDL et aussi comme plate-forme de prototypage rapide. Cependant, comme je l'ai plusieurs fois rappelé, il reste du travail pour avoir un outil complet.

**Résultats fondamentaux**

Les avancées scientifiques que nous (le projet Api/Cosi) avons produits constituent un points positifs de ces recherches, comme en témoignent les nombreuses publications que nous avons faites autour de la compilation/parallélisation de circuits avec le modèle polyédrique. Nous avons directement étendu le modèle polyédrique, avec la manipulation des $\mathbb{Z}$-polyèdres, et nous avons répandu son utilisation en le faisant connaître à travers diverses collaborations (Reading, Yaoundé, Trois Rivières, Calcutta, ...). Nous avons aussi solidifié la méthodologie de conception qui avait été envisagée par Leiserson puis Quinton. Nous avons introduit les concepts de conception hiérarchique, d'ordonnancement structuré et de conception conjointe dans la compilation/parallélisation de circuits à l'aide du modèle polyédrique. Nous avons aussi montré que cette méthodologie s'adaptait rapidement aux nouvelles technologies (FPGA par exemple) et que les principes dégagés par la méthodologie restaient en place (même si la technologie changeait rapidement). Enfin, nous avons, par des études approfondies dans divers domaines applicatifs, introduit de nouvelles architectures et de nouveaux algorithmes. Tous ces résultats ont été produits dans une atmosphère de collaboration efficace avec la communauté française de parallélisation ainsi que la communauté bretonne de CAO et les liens scientifiques et amicaux restent très forts aujourd'hui.

**Expérience logicielle**

Il faut rappeler que ces recherches ont été menées grâce à la plate-forme logicielle MMAlpha qui a été développée par toutes les personnes citées en annexe B et essentiellement administrée par moi-même durant la période où j'étais à Rennes. J'estime grossièrement avoir utilisé un tiers de mon temps de travail au développement et à l'administration de MMAlpha. Le travail des thésards a bien sûr été essentiel, mais nous avons toujours essayé de les dégager des implémentations qui n'étaient pas directement utiles à leur thèse. En terme de bilan, on peut faire les remarques suivantes:

– Le bilan pédagogique interne à l'équipe est positif car MMAlpha a servi d'infrastructure logicielle à de nombreuses thèses, en particulier depuis deux ou trois ans lorsque la méthodologie décrite en section 3.3.2 a été solidifiée.
– L'utilisation extérieure à l'Irisa est en train de se mettre en place et est en phase de test ; il est trop tôt pour dire si cette utilisation perdurera. Nous avons fourni de gros efforts pour documenter le logiciel mais il semble que nous ayons un peu surévalué les connaissances polyédriques des utilisateurs potentiels.

– L'expérience logicielle elle-même a été menée avec les moyens du bord. Mathematica a, à mon avis, grandement facilité le développement et surtout le portage sur d'autres plates-formes (Linux, Windows) mais semble freiner la diffusion du logiciel, surtout en milieu industriel. Le problème du choix de la plate-forme de programmation pour un tel logiciel est toujours difficile. Je pense que l'utilisation de techniques plus robustes comme celles utilisées pour la conception de Ptolemy par exemple, aurait entrainé un surcoût trop important car nous ne savions pas exactement dans quelle direction aller. Le prototypage rapide grâce (en particulier) aux fonctions de calcul symbolique de Mathematica s'est révélé très utile.

Je pense qu'aujourd'hui les grosses plates-formes logicielles telles que MMAlpha sont nécessaires à la recherche en informatique (on en trouve d'ailleurs dans presque tous les projets de l'Irisa), mais leur développement nécessite une évolution importante des mentalités et des règles de fonctionnement des laboratoires. Le fait de passer du temps au développement et à la maintenance de logiciel est très difficile à rentabiliser en France. J'ai eu la chance de pouvoir profiter d'un environnement de recherche dynamique et de publier fréquemment, mais il est certain que le travail de développement que j'ai effectué peut difficilement être réalisé par quelqu'un ayant des charges d'enseignement plus importantes. Le problème vient du fait qu'en France, les publications sont sur-évaluées par rapport aux développements logiciels. Selon les propres paroles d'Ed Lee (concepteur de Ptolemy à l'université de Berkeley), un code qui est utilisé vaut beaucoup plus qu'une publication car en général personne ne lit la publication. Aux États-Unis, la manière dont sont intégrés le développement et les techniques de génie logiciel à la recherche explique aussi la meilleure considération qu'ont les industriels pour les thésards.

### 4.3.2 Perspectives

Il est toujours difficile de faire des perspectives pour la recherche qui ne soient pas utopiques. Je distingue ci-dessous les perspectives pour la recherche en compilation/parallélisation de circuits, les perspectives pour MMAlpha puis je dirai un mots sur les formats communs.

#### Perspectives pour la recherche

La recherche fondamentale en compilation/parallélisation de circuits continuera à produire des résultats, en particulier en France car la culture scientifique dans ce domaine est forte. Cependant, la taille des compilateurs actuels ainsi que la rapidité d'évolution des technologies me poussent à penser que l'on obtiendra des résultats efficaces en associant les efforts de plusieurs équipes de recherche et de plusieurs industriels. Nous essayons de réaliser cela en fédérant les équipes de l'Ouest travaillant autour de la CAO pour faciliter le dialoque avec les industriels.

Plus ponctuellement, les problèmes de génération automatique d'interface efficace et du partitionnement optimal restent encore des problèmes de recherche critiques. Nous y travaillons activement.

### Perspective pour MMAlpha

La pérennisation du logiciel MMAlpha passera nécessairement par un phase de solidification du noyau qu'est Polylib. Je pense qu'aujourd'hui, il est extrêmement important de profiter du rayonnement de Polylib pour produire un outil solide qui intègre tous les outils utilisés par les méthodes polyédriques (essentiellement Polylib et Pip) et qui pourra être utilisé par les compilateurs/paralléliseurs de codes ou de circuits. Cette réalisation est une double clef ; d'abord elle scellera les efforts de la communauté française de parallélisation et les rendra visibles ; ensuite elle offrira un plate-forme commune à plusieurs outils.

En ce qui concerne MMAlpha, il y a deux voies possibles : la consolidation de la partie décrite en section 3.3.2 pour une diffusion académique du logiciel ou une évaluation des possibilités commerciales offertes par le logiciel, soit en terme de connaissances accumulées dans l'équipe Cosi, soit en terme de résultats produits par le logiciel lui-même (IP). Nous avons commencé à étudier les réaction industrielles face à MMAlpha (présentation à HP Labs, à DATE2000 [141]).

### Ahhhh les formats communs....

Aujourd'hui, le succès d'un logiciel libre passe par l'utilisation de formats reconnus. L'utilisation de VHDL synthétisable a grandement aidé la diffusion du logiciel. D'autres tentatives de format communs (comme le Common Nana Format) se revelèrent des échecs. Il me semble indispensable de choisir des formats communs aux niveaux suivants: code impératif de haut niveau, code en assignation unique, représentation intermédiaire, niveau RTL, liste d'interconnection, liste d'interconnection placée. Le sous ensemble VHDL synthétisable [1] semble être le meilleur candidat pour le niveau RTL bien qu'il impose beaucoup de restrictions. Le format Suif est utilisé comme représentation intermédiaire dans l'outils Pico. L'outil MarParser propose un format de Matlab en assignation unique qui pourrait être une base pour un formalisme en assignation unique. Les formats plus bas niveau sont plus délicats à choisir ; EDIF semble presque universellement utilisé, mais les directives de placement sont pour l'instant directement liées au constructeur. Il semble que l'évolution constante des architectures reconfigurables va pousser à la réalisation d'une abstraction pour les différents niveaux topologiques ; La proposition et le succès d'un format à ce niveau accélèreraient la diffusion de logiciels académiques tels que MMAlpha.

# Bibliographie

[1] VHDL Synthesis Interoperability Working Group IEEE PAR 1076.6. Ieee standard for vhdl register transfer level synthesis level 1, ieee p1076.6/d1.12. Draft report, 1999. available at http://www.vhdl.org/vhdlsynth/siwg/.

[2] S. P. Amarasinghe. *Paralelizing Compiler Techniques Based on Linear Inequalities*. Phd thesis, Stanford University, Computer Systems Laboratory, January 1997.

[3] S.P. Amarasinghe and M.S. Lam. Communication optimization and code generation for distributed memory machines. In Sigplan, editor, *ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, volume 28, 1993.

[4] C. Ancourt. Code generation for data movements in hierarchical memory machines. In *Int. Workshop on Compilers for Parallel Computers*, pages 81–102. Ecole des Mines de Paris, 1990.

[5] C. Ancourt. *Génération automatique de codes de transfert pour multiprocesseurs à mémoires locales*. PhD thesis, Université de Paris VI, 1991.

[6] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proc. ACM SIGPLAN '91*, pages 39–50, June 1991.

[7] S. Balev, P. Quinton, S. V. Rajpoadhye, and T. Risset. Linear programming models for scheduling systems of affine recurrence equations – a comparative study –. In *10th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1998.

[8] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[9] Utpal Banerjee. An introduction to a formal theory of dependence analysis. *The Journal of Supercomputing*, 2:133–149, 1988.

[10] Utpal Banerjee. A theory of loop permutations. In Gelernter, Nicolau, and Padua, editors, *Languages and Compilers for Parallel Computing*. The MIT Press, Cambridge, Massachusetts, 1990.

[11] Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors. *Languages and Compilers for Parallel Computing*. Number 589 in Lecture Notes in Computer Science. Springer-Verlag, 1992.

[12] M. Black and A. Rangarajan. On the unification of line processes, outlier rejection, and robust statistics with applications in early vision. *Int. J. Computer Vision*, 19(1):75–104, 1996.

[13] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling. In *Scalable High-Performance Computing Conference*, Mai 23-25 1994.

[14] Pierre Boulet and Thomas Brandes. Evaluation of automatic parallelization strategies for HPF compilers. In H. Liddell, A. Colbrook, B. Hertzberger, and P. Sloot, editors, *High-Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 778–783, Brussels, Belgium, April 1996. Springer Verlag.

[15] Scott Bowden, Doran Wilde, and Sanjay Rajopadhye. Quadratic control in linear systolic arrays. In *IEEE International Conference on Application-specific Systems, Architectures and Processors*, jul 2000.

[16] T. Brandes. Adaptor - a transformation tool for HPF programs. In *Programming Environments for Massively Parallel Distributed Systems*, pages 91–96. Birkhäuser Verlag, April 1994.

[17] R. P. Brent and H.T. Kung. Systolic VLSI arrays for polynomial GCD computation. *IEEE Trans. Comp.*, 33(8):731–736, 1984.

[18] B. Buchberger. *An algorithm for finding a basis for the residue class ring of a zero-dimensional polynomial ideal*. PhD thesis, Math. Inst. University of Innsbruck, Austria, 1965.

[19] W. Burleson and Y. Jeong. VLSI array synthesis for polynomial GCD computation and application to finite field division. *IEEE Trans. on Circuit and Systems*, 41(12):891–897, 1994.

[20] W. Burleson and B. Jung. Systolic filter for fast DNA similarity search. In *IEEE International Conference on Application-specific Systems, Architectures and Processors*, 1992.

[21] D. Cachera, S. Rajopadhye, T. Risset, and C. Tadonki. Parallelization of the algebraic path problem on linear simd/SPMD arrays. Technical Report 1346, Irisa, 2000.

[22] F. Catthoor, K. Danckaert, C. Kulkarni, and T. Omnes. Data transfer and storage architecture issues and exploration in multimedia processors. In *Programmable Digital Signal Processors: Architecture, Programming, and Applications*. Marcel Dekker, Inc, New York, 2000.

[23] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology*. Kluwer Academic Publishers, 1998.

[24] H. Chabanne. Permutation decoding of abelian codes. *IEEE Transactions on Information Theory*, 38(6):1826–1829, November 1992.

[25] Z. Chamski. *Environnement logiciel de programmation d'un accélérateur de calcul parallèle*. PhD thesis, Ifsic, Rennes I, February 1993.

[26] Z. Chamski. Fast and efficient generation of loop bounds. In D. Trystram, editor, *ParCo '93*, Grenoble, France, September 1993. Elsevier Science Publishers (North Holland). À paraître/to appear.

[27] P. Charbonnier, L. Blanc-Féraud, G. Aubert, and M. Barlaud. Deterministic edge-preserving regularization in computed imaging. *IEEE Trans. Image Processing*, 6(2):298–311, 1997.

[28] P. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces, extended version. *Journal of VLSI Signal Processing*, 19(2):179–194, 1998.

[29] P. Clauss, C. Mongenet, and G.-R. Perrin. Synthesis of size-optimal toroidal arrays for the algebraic path problem : A new contribution. *Parallel Computing*, 18:185–194, 1992.

[30] F. Coelho. *Contributions to High Performance Fortran Compilation*. PhD thesis, École des mines de Paris, 1996.

[31] J.F. Collard, P. Feautrier, and T. Risset. Construction of DO loops from systems of affine constraints. *Parallel Processing Letters*, 5:421–436, 1995.

[32] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995.

[33] A. Darte. *Techniques de parallélisation automatique de nids de boucles*. PhD thesis, LIP ENS-Lyon, 1993.

[34] A. Darte and J.M. Delosme. Partitioning for array processors. Technical Report 90-23, LIP-IMAG, Ecole Normale Supérieure de Lyon, France, 1991.

[35] A. Darte and B. Rau et F. Vivien R. Schreiber. A constructive solution to juggling problem in systolic array synthesis. Technical Report 1999-15, Laboratoire de L'informatique du parallélisme, 1999.

[36] A. Darte and F. Vivien. Revisiting the decomposition of Karp, Miller and Winograd. In *Application Specific Array Processor*, pages 13–25, 1997.

[37] Alain Darte and Yves Robert. Constructive methods for scheduling uniform loop nests. *IEEE Transactions on Parallel Distributed Systems*, 5.8:814–822, 1994.

[38] Alain Darte, Yves Robert, and Frédéric Vivien. *Scheduling and Automatic Parallelization*. Birkhauser, 2000.

[39] F. D. De Dinechin, P. Quinton, and T. Risset. Structuration of the Alpha language. In W.K Giloi, S. Jahnichen, and B.D. Shriver, editors, *Massively Parallel Programming Models*, pages 18–24. IEEE Computer Society Press, 1995.

[40] J.M. Delosme and I.C.F. Ipsen. An illustration of a methodology for the construction of efficient systolic architectures in VLSI. In *International Symposium on VLSI Technology, Systems and Applications*, pages 268–273, Taipei, Taiwan, 1985.

[41] P.M. Dewilde, A.A.J de Lange, A.J van der Hoeven, and F. Deprettere. Hifi: An object oriented system for the structural synthesis of signal processing algorithms and the VLSI compilation of signal-flow graphs. In *IFIP Int. Workshop on Applied Formal Methods for Correct VLSI Design*. IEEE Press, 1989.

[42] C. Dezan. *Génération automatique de circuits avec* ALPHA DU CENTAUR. Thèse de doctorat, Ifsic, Université de Rennes 1, February 1993.

[43] C. Dezan, H. Le Verge, P. Quinton, and Y. Saouter. The Alpha du Centaur environment. In P. Quinton and Y. Robert, editors, *International Workshop Algorithms and Parallel VLSI Architectures II*, Bonas, France, June 1991. North-Holland.

[44] M. Dion, T. Risset, and Y. Robert. Resource-constrained scheduling of partitioned algorithms on processor arrays. In *EuroMicro Workshop on Parallel and Distributed Processing*, pages 571–580. IEEE Computer Society Press, 1995.

[45] Michele Dion, Cyril Randriamaro, and Yves Robert. Compiling affine nested loops: how to optimize the residual communications after the alignment phase? *J. Parallel and Distributed Computing*, 38(2):176–187, 1996.

[46] C. Tayou Djamegni, P. Quinton, S. Rajopadhye, and T. Risset. Derivation Of Systolic Algorithms For The Algebraic Path Problem By Recurrence Transformations. In *4ème Colloque Africain sur la Recherche Informatique*, pages 551,564. Presse Universitaire de Dakar, October 1998.

[47] V. Van Dongen. *From systolic to periodic array design*. PhD thesis, Université Catholique de Louvain, january 1991.

[48] V. Van Dongen and P. Quinton. Uniformization of linear recurrence equations: a step towards the automatic synthesis of systolic array. In K. Bromley et al. eds., editor, *International Conference on Systolic Arrays*, pages 473–482. IEEE Computer Society Press, 1988.

[49] F. Dupont de Dinechin. *Systèmes structurés d'équations récurrentes : mise en œuvre dans le langage Alpha et applications*. Thèse de doctorat, université de Rennes I, January 1997.

[50] F. Dupont de Dinechin, P. Quinton, S. Rajopadhye, and T. Risset. First steps in Alpha. Technical Report 1244, Irisa, 1999.

[51] F. Dupont de Dinechin and S. Robert. Hierarchical static analysis of structured systems of affine recurrence equations. In *Application Specific Array Processors*. IEEE Cmputer Society Press, August 1996.

[52] F. Dupont De Dinechin, S. Robert, and T. Risset. Structured scheduling of recurrence equations. Technical Report 1140, Irisa, Rennes, France, 1997.

[53] F. Dupont De Dinechin, S. Robert, and T. Risset. Introduction to structured scheduling of recurrence equations. *J. Parallel and Distributed Computing*, 1999. submitted.

[54] R. Esser, J. Teich, and L. Thiele. Codesign: An embedded system design environment. *J. IEE Proceedings Computers and Digital Techniques*, 145(3):171–180, 1998.

[55] H. Saito et al. The design of the promis compiler. In *Proceedings of the International Conference on Compiler Construction (CC), LNCS 1575*. Springer Verlag, 1999.

[56] R. Schreiber et al. High-level synthesis of nonprogrammable hardware accelerators. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2000)*, Boston, july 2000.

[57] S. Amarasinghe et al. Suif: An Infrastructure for Research on Paralleli zing and Optimizing Compilers. Technical report, Stanford University, May 1994.

[58] F. Bardoult and P. Quinton and S. Rajopadhye and Tanguy Risset. Synthesis of data-flow interfaces for regular parallel programs. Technical Report 1260, Irisa, September 1999.

[59] T. Risset F. Dupont de Dinechin and S. Robert. Hierarchical static analysis for improving the complexity of linear algebra algorithms. In *Parallel computing (PARCO)*, 1997.

[60] E. Fabiani and D. Lavenier. Placement of linear arrays. In H. Gruenbacher and R. Hartenstein, editors, *Field-programmable Logic: The Roadmap to Reconfigurable Systems*. Springer Verlag, 2000.

[61] E. Fabiani, D. Lavenier, and L. Perraudeau. Loop parallelization on a reconfigurable coprocessor. In *WDTA'98 : Workshop on Design, Test and Applications*, June 1998.

[62] P. Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing, St Malo*, 1988.

[63] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[64] P. Feautrier. Some efficient solution to the affine scheduling problem, part II, multi-dimensional time. *Int. J. of Parallel Programming*, 21(6), December 1992.

[65] P. Feautrier. Some efficient solutions to the affine scheduling problem, part I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[66] Paul Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–51, 1991.

[67] P. Frison, P. Gachet, and P. Quinton. Designing Systolic Arrays with DIASTOL. In S.-Y. Kung, R. E. Owen, and J. G. Nash, editors, *VLSI Signal Processing II*, pages 93–105. IEEE Press, November 1986.

[68] T. Gautier, P. Le Guernic, P. Quinton, S. Rajopadhye, T. Risset, and I. Smarandache. Le projet CAIRN : vers la conception d'architectures à partir de Signal et Alpha. In *Collection Technique et scientifique des Télécommunications*. CNET, 1997.

[69] D. Geman and G. Reynolds. Constrained restoration and the recovery of discontinuities. *IEEE Trans. Pattern Anal. Machine Intell.*, 14(3):367–383, 1992.

[70] G.M. Megson and D. Comish. Systolic algorithm design environments (SADEs). In *Transformational Approaches to Systolic Design*, chapter 9. London : Chapman and Hall, 1994.

[71] M. Griebl and C.s Lengauer. The loop parallelizer LooPo. In Michael Gerndt, editor, *Proc. Sixth Workshop on Compilers for Parallel Computers*, volume 21 of *Konferenzen des Forschungszentrums Jülich*, pages 311–320. Forschungszentrum Jülich, 1996.

[72] P. Guerdoux-jamet and D. Lavenier. Systolic filter for fast DNA similarity search. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 1995)*, 1995.

[73] P. Guerdoux-Jamet and D. Lavenier. Systolic filter for fast DNA similarity search. In *IEEE International Conference on Application-specific Systems, Architectures and Processors*, Strasbourg, july 1995.

[74] A. Guillou, P. Quinton, and T. Risset. Automatic design of VLSI pipelined lms architectures. In *2000 IEEE Canadian Conference on Electrical & Computer Engineering*, Trois rivière, Canada, August 2000.

[75] C. Heckler and L. Thiele. Computing linear data dependencies in nested loop programs. *Parallel Processing Letters*, 4(1):193–204, 1994.

[76] P.C. Held. Hipars: a tool for automatic conversion of nested loop programs into single assignment programs. Technical Report Dept. Electrical Engineering, Delft University of Technology, 1994.

[77] P.P. Hou, R.M. Owens, and M.J. Irwin. Decomposer: a synthesizer for systolic systems. In *Proceedings of the International Design automation conference*, pages 650–653, 1988.

[78] P. Huber. *Robust Statistics*. John Wiley & Sons, 1981.

[79] F. Irigoin, P.Jouvelot, and R.Triolet. Overview of the pips project. In *Procs of the Int. Workshop on Compiler for Parallel Computers, Paris*, pages 199–212, December 1990.

[80] F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319–329, San Diego, CA, January 1988.

[81] S.M. Jennings. A Gröbner basis view of the Welch-Berlekamp algorithm for Reed-Solomon codes. *IEE Proceedings Communications*, 142(6):349–351, December 1995.

[82] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[83] M. Katsushige, N. Kiyoshi, and K. Hitoshi. Pipelined LMS Adaptative Filter Using a New Look-Ahead Transformation. *IEEE Transactions on Circuits and Systems*, 46:51–55, January 1999.

[84] W. Kelly and W. Pugh. A unifying framework for iteration reordering transformations. In *International Conference on Algorithms And Architectures for Parallel Processing*, pages 153–162. IEEE Computer Society Press, 1995.

[85] B. Kienhuis. Matparser: An array dataflow analysis compiler. Technical Report M00/9, EECS, University of Berkeley, 2000.

[86] B. Kienhuis, E. Rijpkema, and E.F. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *8th International Workshop on Hardware/Software Codesign (CODES'2000)*, 2000.

[87] S. Kim and E. A. Lee. Infrastructure for numeric precision control in the ptolemy environment. In *Proceedings of the 40th Midwest Symposium on Circuits and Systems*, August 1997.

[88] H.T. Kung. Why systolic architectures? *Computer*, 15(1):37–46, 1982.

[89] H.T. Kung and C.E. Leiserson. Systolic arrays for VLSI. In C.A. Mead and L.A. Conway, editors, *in Introduction to VLSI systems*. Addison-Wesley, 1980.

[90] M.S. Lam and M. E. Wolf. Automatic blocking by a compiler. In J. Dongarra, K. Kennedy, P. Messina, D. C. Sorensen, and R. G. Voigt, editors, *Proc. of the fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 537–542, 1992.

[91] L. Lamport. The parallel execution of do loops. *Communications of The ACM*, 17(2):83–93, February 1974.

[92] D. Lavenier. SAMBA: Systolic Accelerator for Molecular Biological Application. Technical Report 988, Irisa, March 1996.

[93] P. Le Moenner, L. Perraudeau, S. Rajopadhye, T. Risset, and P. Quinton. Generating regular arithmetic circuits with AlpHard. In *Massively Parallel Computing Systems (MPCS'96)*, May 1996.

[94] Patricia Le Moënner. *Contribution à la réalisation d'une chaîne complète pour la synthèse de circuits réguliers*. Thèse de doctorat, université de Rennes I, November 1997.

[95] H. Le Verge. A note on Chernikova's algorithm. RR 635, Irisa, Irisa, Campus de Beaulieu, 35042 Rennes Cedex, France, February 1992.

[96] V. Lefebvre and P. Feautrier. Storage management in parallel programs. In *5th Euromicro Workshop on Parallel and Distributed Processing*, pages 181–188, London, January 1997. IEEE Computer Society Press.

[97] C.E. Leiserson. *Area-efficient VLSI computation*. PhD thesis, Carnegie Mellon University, U.S.A., 1981.

[98] P. Lenders and S. Rajopadhye. Multirate VLSI arrays and their synthesis. *IEEE Transactions on Computers*, may 1996.

[99] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[100] J. Li and MS. Chen. The data alignent phase in compiling programs for distributed memory machines. *Journal Parallel Distributed Computing*, 13:213–221, 1991.

[101] M. Manjunathaiah, S. Rajoapdhye, and T. Risset. Automatic uniformization. Technical Report 1350, Irisa, Rennes, 2000.

[102] C. Mauras. *Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. Thèse de doctorat, Ifsic, Université de Rennes 1, December 1989.

[103] C. Mauras, P. Quinton, S. Rajopadhye, and Y. Saouter. Scheduling affine parameterized recurrences by means of variable dependent timing functions. In S.Y Kung, Jr. E.E. Swartzlander, J.A.B. Fortes, and K.W. Przytula, editors, *Application Specific Array Processors*, pages 100–110. IEEE Computer Society Press, September 1990.

[104] G.M. Megson and X. Chen. *Automatic Parallelization for a class of regular computations*. World Scientific, 1997.

[105] G.M. Megson and L. Rapanotti. Uniformization techniques for reducible integral recurrence equations. In *Algorithms and Parallel VLSI Architectures III*, pages 283–296. Elsevier Science Publishers B.V., 1995.

[106] E. Mémin and T. Risset. Full alternate Jacobi minimization and VLSI derivation of hardware for motion estimation. In *Int. Workshop on Parallel Image Processing and Analysis, IWPIPA'99*, Madras, India, Jan. 1999.

[107] E. Mémin and T. Risset. On the study of VLSI derivation for optical flow estimation. *International Journal of pattern recognition and Artificial Intelligence (IJPRAI)*, 2000. in press.

[108] E. Mémin and T. Risset. VLSI design methodology for edge-preserving image reconstruction. *Real-Time Imaging, Special issue on Fast Energy Minimization-Based Imaging and Vision Techniques*, accepted for publication.

[109] D.I. Moldovan. On the analysis and synthesis of VLSI systolic arrays. *IEEE Transactions on Computers*, 31:1121–1126, 1982.

[110] D.I. Moldovan. ADVIS: a software package for the design of systolic arrays. *IEEE Transaction on Computer Aided Design of Intergrated Circuit and Systems*, CAD-6(1):33–40, 1996.

[111] D.I. Moldovan and J.A.B. Fortes. Partitioning and mapping algorithms into fixed-size systolic arrays. *IEEE Transactions on Computers*, 35(1):1–12, jan 1986.

[112] A. Mozipo, D. Massicote, P. Quinton, and T. Risset. A Parallel Architecture for Adaptative Channel Equalization Based On Kalman Filter Using MMAlpha. In *1999 IEEE Canadian Conference on Electrical & Computer Engineering*, Calgary, Canada, May 1999.

[113] A. Mozipo, D. Massicote, P. Quinton, and T. Risset. Automatic Synthesis of a Parallel Architecture for Kalman Filtering using MMAlpha. In *1999 IEEE Canadian Conference on Electrical & Computer Engineering*, Edmonton, Canada, May 1999.

[114] A. Mozipo, D. Massicote, P. Quinton, and T. Risset. Structured design of parallel VLSI architectures for Kalman filters using a declarative language. *IEEE transaction on VLSI and Signal Processing*, 2000. submitted.

[115] A. Mozipo, D. Massicotte, P. Quinton, , and T. Risset. Automatic synthesis of a parallel architecture for kalman filtering using mmalpha. In *International conference on parallel computing in electrical engineering (PARELEC 98)*, pages 201–206, Bialystok, Poland, September 1998.

[116] R. Shanbhag Naresh and K. Parhi Keshab. Pipelined Adaptative DFE Architecture Using Relaxed Look-Ahead. *IEEE Transactions on Signal Processing*, 43(6):1368–1385, June 1995.

[117] S. P. K. Nookala and T. Risset. A library for z-polyhedral operations. Technical Report 1330, Irisa, Rennes, 2000.

[118] D. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin. Polaris: A new-generation parallelizing compiler for MPP's. Technical report, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev, June 1993.

[119] Manuel de référence de paf. Groupe "Calcul Parallèle" du MASI, January 1990. available on request from Paul Feautrier.

[120] L. Perraudeau. *Madmacs : un outil pour le dessin des masques de réseaux réguliers intégrés*. Thèse de doctorat, Ifsic, université de Rennes 1, September 1993.

[121] G.-R. Perrin, S. Genaud, and E. Violard. PEI: a theoretical framework for data-parallel programming. Technical report, ICPS, Strasbourg, 1994.

[122] K. Pingali and A. Rogers. Compiling for locality. In *Proceedings of the International conference on parallel Processing.*, pages 142–146, 1990.

[123] W. Pugh. The omega test: a fast and practical integer programming algorithm for depedence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[124] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):27–47, August 1992.

[125] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469,498, 2000.

[126] F. Quilleré, S. Rajopadhye, and D. Wilde. Gestion de la mémoire dans un compilateur d'équations récurrentes. *Technique et Science Informatique*, 19(6), 2000.

[127] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *The 11th Annual International Symposium on Computer Architecture*, Ann Arbor, Michigan, June 1984. IEEE Computer Society Press.

[128] P. Quinton. The systematic design of systolic arrays. In Françoise Fogelman Soulie, Yves Robert, and Maurice Tchuente, editors, *Automata Networks in Computer Science*, chapter 9, pages 229–260. Manchester University Press, 1987.

[129] P. Quinton, S. Rajopadhye, and D. Wilde. The naive execution of affine recurrence equations. Technical Report 904, Irisa, Rennes, France, Jan 1995.

[130] P. Quinton, S. V. Rajopadhye, and T. Risset. Extension of the Alpha language to recurrences on sparse periodic domains. In *Int. Conf. on Application Specific Array Processors*, 1996.

[131] P. Quinton, S. V. Rajopadhye, and T. Risset. On manipulating z-polyhedra using a canonical representation. *Parallel Processing Letters*, 7(2):181–194, June 1997.

[132] P. Quinton and Y. Robert. *Systolic Algorithms and Architectures*. Prentice Hall and Masson, 1989.

[133] S. Rajopadhye, T. Risset, and C. Tadonki. The algebraic path problem revisited. In *Fifth International Euro-Par Conference*, pages 698–707, Toulouse, France, August 1999.

[134] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *Proceedings, Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 488–503, New Delhi, India, December 1986. Springer Verlag, LNCS 241.

[135] Sanjay V. Rajopadhye and Richard M. Fujimoto. Synthesizing systolic arrays from recurrence equations. In *Parallel Computing*, volume 14, pages 163–189. Elsevier Science Publishers B.V., North Holland, 1990.

[136] J. Ramanujam. A linear algebraic view of loop transformations and their interaction. In J. Dongarra, K. Kennedy, P. Messina, D. C. Sorensen, and R. G. Voigt, editors, *Proc. of the fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 543–548, 1992.

[137] J. Ramanujam. Non-unimodular transformations of nested loops. In *Proc. Supercomputing'92*, pages 214–223. IEEE Computer Society Press, November 1992.

[138] J. Ramanujam and P. Sadayappan. Tiling of iteration spaces for multicomputers. In *Proc. Internal Conference on Parallel Processing*, volume 2, pages 179–186, August 1990.

[139] Sailesh K. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays*. PhD thesis, Stanford University, October 1985.

[140] T. Risset and S. Derrien. Interfacing compiled FPGA programs: the MMAlpha approach. In *PDPTA2000: Second International Workshop on Engineering of Reconfigurable Hardware/Software Objects*. CSREA Press, june 2000.

[141] T. Risset and P. Quinton. MMAlpha: a toolbox for silicon compilation. In *Design, Automation and Test in Europe (DATE2000)*, March 2000. University booth stand.

[142] T. Risset and Y. Saouter. A linear systolic array for the computation of gröbner basis. Technical Report 1069, Irisa, Rennes, France, 1996.

[143] T. Risset and Y. Saouter. Synthèse de haut niveau d'un co-processeur pour le calcul des bases de grobner. In *5eme symposium en architecture nouvelles de machines (Sympa'5)*, Rennes, Jun 1999.

[144] Tanguy Risset. *Parallélisation automatique: du modèle systolique à la compilation de nids de boucles*. Thèse de doctorat, ENS-Lyon, February 1994.

[145] Sophie Robert. *Parallélisation d'algorithm de portaits spectraux. Utilisation d'un environnement de transformations de programmes*. Thèse de doctorat, Ifsic, université de Rennes 1, November 1997.

[146] V.P. Roychowdhury. *Derivation, Extensions and Parallel Implementation of Regular Iterative Algorithms*. PhD thesis, Stanford University, Department of Electrical Engineering, Stanford, CA, December 1988.

[147] Y. Saouter. *A propos de systèmes d'équations récurrentes*. Thèse de doctorat, Ifsic, Université de Rennes 1, oct 1992.

[148] R. Schreiber and J.J. Dongarra. Automatic blocking of nested loops. Technical Report 90.38, RIACS, August 1990.

[149] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, New York, 1986.

[150] O. Sentieys, J.P. Diguet, and J.L. Philippe. Gaut: a high level synthesis tool dedicated to real time signal processing application. In *EURO-DAC*, September 2000. University booth stand.

[151] Irina Madalina Smarandache. *Transformations affines d'horloges: application au codesign de systèmes temps-réel en utilisant les langages Signal et Alpha*. Thèse de doctorat, Ifsic, Université de Rennes 1, 1998.

[152] T.F. Smith and J.F. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.

[153] W. Sung and K.-I. Kum. Wordlength determination and scaling software for a signal flow block diagram. In *Proc. International Conference on Acoustic, Speech and Signal Processing*, pages 457–460, April 1994.

[154] K. Tadaaki and N. Kiyoshi andK. Hitoshi. A Pipelined Architecture for DLMS Algorithm Considering Both Hardware Complexity and Output Latency. In *European signal processing conference*, pages 503–506, 1998.

[155] C. Tayou Djamegni, P. Quinton, S. Rajopadhye, and T. Risset. Derivation of systolic algorithms for the algebraic path problem by recurrence transformations. *Parallel Computing*, (in press).

[156] J. Teich. *A compiler for application-specific processor arrays*. PhD thesis, Saarbrücken University, 1993.

[157] L. Thiele and U. Arzt. On the synthesis of massively parallel architectures. *Int. J. of High Speed Electronics and Systems*, 4(2):99–131, 1994.

[158] J. Thomas. Pipelined Systolic Architectures for DLMS Adaptive Filtering. *Journal of VLSI Signal Processing*, 12:223–246, 1996.

[159] J. M. Tourreilles. *Conception d'Architectures pour Traitement du Signal en Précision Finie*. Thèse de doctorat, université de Rennes 1, January 1999.

[160] U. Vehlies. DECOMP–a program for mapping DSP algorithms onto systolic arrays. In G. Megson, editor, *Transformational Approaches to Systolic Design*, pages 159–178. Chapman & Hall, 1994.

[161] D. Wilde. A library for doing polyhedral operations. Technical Report 785, Irisa, Rennes, France, 1993.

[162] D. Wilde and O. Sie. Regular array synthesis using Alpha. Technical Report 829, Irisa, Rennes, France, May 1994.

[163] D. K. Wilde. The Alpha language. Research Report 999, Irisa, Campus de Beaulieu, 35042 Rennes Cedex, France, Jan 1994.

[164] M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distributed Systems*, 2(4):452–471, October 1991.

[165] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge MA, 1989.

[166] M. Wolfe. Data dependence and program restructuring. *The Journal of Supercomputing*, 4:321–344, 1990.

[167] M. Wolfe. A loop restructuring research tool. Technical Report CSE 90-014, Oregon Graduate Institute, August 1990.

# Annexe A

# Résultats de la conception de Samba présentée en section 3.3

## A.1 Programme Alpha original

Ce programme (le même que celui de celui de la figure 3.9, page pagereffigSamba1) est celui à partir duquel l'architecture représentée en figure 3.13 a été conçu en section ref-secSamba. Il s'agit du programme Alpha pour l'algorithme de comparaison de séquences Samba étudié dans [73].

```
system sequence: {X,Y| Y>X>2}
        (QS : {i| 1<= i <= X} of integer;
         DB : {j| 1<= j <= Y} of integer)
        returns
        (res : {j| 1<= j <= Y} of integer);
var
M: {i,j|  0<= i <= X; 0<= j <= Y} of integer;
MatchQ: {i,j|  1<= i <= X; 1<= j <= Y} of integer;
let
M[i,j]=case
        {|i=0}: 0[];
        {|i>0; j=0}: 0[];
        {|i>0; j>0}: Max4(0[],M[i,j-1]-8[],
                       M[i-1,j]-8[],M[i-1,j-1]+MatchQ[i,j]);
       esac;
MatchQ[i,j]=if (QS[i]=DB[j]) then 15[] else -12[];
res[j]=M[X,j];
tel;
```

## A.2 Programme Alpha0 dérivé du programme de la section A.1

Le programme suivant est le programme Alpha0 pour l'algorithme de comparaison de séquences Samba étudié dans [73]. On peut directement en déduire l'architecture du réseau représenté en figure 3.13 page 63, toutes les informations structurelles sont là.

```
system sequence :{X,Y | 3<=X<=Y-1}
                 (QS : {i | 1<=i<=X} of integer;
                  DB : {j | 1<=j<=Y} of integer)
       returns   (res : {j | 1<=j<=Y} of integer);
var
  TSep8 : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
  TSep7 : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
  TSep6 : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
  TSep5 : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
  TSep4 : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
  TSep3 : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
  expr1 : {t,p | p-1<=t<=p+Y-1; 1<=p<=X+1} of integer;
  DB_mirr1 : {t,p | 2<=t<=Y+1; p=1} of integer;
  QS_mirr1 : {t,p | -X+3<=t<=2; p=1} of integer;
  TSep2 : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
  TSep1 : {t,p | p+1<=t<=p+Y; 1<=p<=X} of boolean;
  M_ctl2P_reg1loc : {t,p | p-1<=t<=p+Y-1; 2<=p<=X+1} of boolean;
  QQS_ctl1P_reg2loc : {t,p | p<=t<=p+Y-1; 2<=p<=X+1} of boolean;
  QQS_In_reg3loc : {t,p | 2p-X-1<=t<=p; 2<=p} of integer;
  DDB_reg4loc : {t,p | p<=t<=p+Y-1; 2<=p<=X+1} of integer;
  M_reg8 : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
  M_reg7 : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
  M_reg6 : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
  QQS_reg5 : {t,p | p+2<=t<=p+Y; 1<=p<=X} of integer;
  DDB_reg4 : {t,p | p+1<=t<=p+Y; 2<=p<=X} of integer;
  QQS_In_reg3 : {t,p | 2p-X+1<=t<=p+1; 2<=p} of integer;
  QQS_ctl1P_reg2 : {t,p | p+1<=t<=p+Y; 2<=p<=X} of boolean;
  M_ctl2P_reg1 : {t,p | p<=t<=p+Y; 2<=p<=X} of boolean;
  M_ctl2P_Init_In : {t,p | 1<=t<=Y+1; p=1} of boolean;
  M_ctl2P_Init : {t | 1<=t<=Y+1} of boolean;
  M_ctl2P : {t,p | p<=t<=p+Y; 1<=p<=X} of boolean;
  QQS_ctl1P_Init_In : {t,p | 2<=t<=Y+1; p=1} of boolean;
  QQS_ctl1P_Init : {t | 2<=t<=Y+1} of boolean;
  QQS_ctl1P : {t,p | p+1<=t<=p+Y; 1<=p<=X} of boolean;
  QQS_In : {t,p | 2p-X+1<=t<=p+1; 1<=p} of integer;
  DDB : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
  QQS : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
  M : {t,p | p<=t<=p+Y; 0<=p<=X} of integer;
```

```
    MatchQ : {t,p | p+1<=t<=p+Y; 1<=p<=X} of integer;
  let
    TSep8[t,p] = max(TSep5, TSep6);
    TSep7[t,p] = max(0[], TSep4);
    TSep6[t,p] = M_reg8 + MatchQ[t,p];
    TSep5[t,p] = M_reg7 - 8[];
    TSep4[t,p] = M_reg6 - 8[];
    TSep3[t,p] = max(TSep7, TSep8);
    expr1[t,p] = M[t,p-1];
    DB_mirr1[t,p] = DB[t-1];
    QS_mirr1[t,p] = QS[-t+3];
    M_reg8[t,p] = expr1[t-2,p];
    M_reg7[t,p] = expr1[t-1,p];
    DDB_reg4loc[t,p] = DDB[t,p-1];
    DDB_reg4[t,p] = DDB_reg4loc[t-1,p];
    QQS_In_reg3loc[t,p] = QQS_In[t,p-1];
    QQS_In_reg3[t,p] = QQS_In_reg3loc[t-2,p];
    QQS_ctl1P_reg2loc[t,p] = QQS_ctl1P[t,p-1];
    QQS_ctl1P_reg2[t,p] = QQS_ctl1P_reg2loc[t-1,p];
    M_ctl2P_reg1loc[t,p] = M_ctl2P[t,p-1];
    M_ctl2P_reg1[t,p] = M_ctl2P_reg1loc[t-1,p];
    M_reg6[t,p] = M[t-1,p];
    QQS_reg5[t,p] = QQS[t-1,p];
    M_ctl2P_Init_In[t,p] = M_ctl2P_Init[t];
    M_ctl2P_Init[t] =
        case
          case
            { | t=1} : True[];
            { | 2<=t} : False[];
          esac;
        esac;
    QQS_ctl1P_Init_In[t,p] = QQS_ctl1P_Init[t];
    QQS_ctl1P_Init[t] =
        case
          case
            { | t=2} : True[];
            { | 3<=t} : False[];
          esac;
        esac;
    QQS_In[t,p] =
        case
          { | p=1} : QS_mirr1;
          { | 2<=p} : QQS_In_reg3;
        esac;
    DDB[t,p] =
```

```
        case
          { | p=1} : DB_mirr1;
          { | 2<=p} : DDB_reg4;
        esac;
    QQS_ctl1P[t,p] =
        case
          { | p=1} : QQS_ctl1P_Init_In[t,p];
          { | 2<=p} : QQS_ctl1P_reg2;
        esac;
    QQS[t,p] =
        case
          if (QQS_ctl1P[t,p]) then
              case
                { | t=p+1} : QQS_In;
                { | p+2<=t} : 0[];
              esac else
          case
            { | t=p+1} : 0[];
            { | p+2<=t} : QQS_reg5;
          esac;
        esac;
    M_ctl2P[t,p] =
        case
          { | p=1} : M_ctl2P_Init_In[t,p];
          { | 2<=p} : M_ctl2P_reg1;
        esac;
    M[t,p] =
        case
          { | p=0} : 0[];
          { | 1<=p} : if (M_ctl2P[t,p]) then
                case
                  { | t=p} : 0[];
                  { | p+1<=t} : 0[];
                esac else
             case
               { | t=p} : 0[];
               { | p+1<=t} : TSep3;
             esac;
        esac;
    TSep1[t,p] = QQS[t,p] = DDB[t,p];
    TSep2[t,p] = -12[];
    MatchQ[t,p] = if (TSep1) then 15[] else TSep2;
    res[j] = M[j+X,X];
  tel;
```

## A.3 Programme AlpHard dérivé du programme de la section A.2

Cette section regroupe le programme AlpHard qui correspond donc a la structuration du programme Alpha0 présenté en section A.2. Les sous-systèmes sont présentés dans cette ordre: l'interface en section A.3.1, le module en section A.3.2, la cellule 1 en section A.3.3 (la cellule 2 n'est pas représentée, elle est quasiment identique à la cellule 1 avec toutefois quelques différences minimes), et enfin le contrôleur en section A.3.5.

### A.3.1 Interface du programme AlpHard

```
system sequence :{X,Y | 3<=X<=Y-1}
                 (QS : {i | 1<=i<=X} of integer;
                  DB : {j | 1<=j<=Y} of integer)
       returns  (res : {j | 1<=j<=Y} of integer);
var
  DB_mirr1 : {t,p | 2<=t<=Y+1; p=1} of integer;
  QS_mirr1 : {t,p | -X+3<=t<=2; p=1} of integer;
  M : {t,p | p<=t<=p+Y; 0<=p<=X} of integer;
let
  DB_mirr1[t,p] = DB[t-1];
  QS_mirr1[t,p] = QS[-t+3];
  res[j] = M[j+X,X];
  use  sequenceModule[X,Y] (DB_mirr1, QS_mirr1) returns  (M) ;
tel;
```

### A.3.2 Module du programme AlpHard

```
system sequenceModule :{X,Y | 3<=X<=Y-1}
     (DB_mirr1In : {t,p | 2<=t<=Y+1; p=1} of integer;
      QS_mirr1In : {t,p | -X+3<=t<=2; p=1} of integer)
       returns
     (MOut : {t,p | p<=t<=p+Y; 0<=p<=X} of integer);
var
  M_ctl2P_Init : {t | 1<=t<=Y+1} of boolean;
  QQS_ctl1P_Init : {t | 2<=t<=Y+1} of boolean;
  M : {t,p | p<=t<=p+Y; 0<=p<=X} of integer;
  M1 : {t,p | p<=t<=p+Y; 2<=p<=X} of integer;
  expr1 : {t,p | p-1<=t<=p+Y-1; 1<=p<=X+1} of integer;
  DDB1 : {t,p | p+1<=t<=p+Y; 2<=p<=X} of integer;
  DDB_reg4loc : {t,p | p<=t<=p+Y-1; 2<=p<=X+1} of integer;
  QQS_In1 : {t,p | 2p-X+1<=t<=p+1; 2<=p} of integer;
  QQS_In_reg3loc : {t,p | 2p-X-1<=t<=p; 2<=p} of integer;
  QQS_ctl1P1 : {t,p | p+1<=t<=p+Y; 2<=p<=X} of boolean;
  QQS_ctl1P_reg2loc : {t,p | p<=t<=p+Y-1; 2<=p<=X+1} of boolean;
  M_ctl2P1 : {t,p | p<=t<=p+Y; 2<=p<=X} of boolean;
  M_ctl2P_reg1loc : {t,p | p-1<=t<=p+Y-1; 2<=p<=X+1} of boolean;
  M2 : {t,p | 1<=t<=Y+1; p=1} of integer;
  DB_mirr1 : {t,p | 2<=t<=Y+1; p=1} of integer;
  QS_mirr1 : {t,p | -X+3<=t<=2; p=1} of integer;
  DDB2 : {t,p | 2<=t<=Y+1; p=1} of integer;
  QQS_In2 : {t,p | -X+3<=t<=2; p=1} of integer;
  QQS_ctl1P2 : {t,p | 2<=t<=Y+1; p=1} of boolean;
  M_ctl2P2 : {t,p | 1<=t<=Y+1; p=1} of boolean;
  M_ctl2P_Init_In : {t,p | 1<=t<=Y+1; p=1} of boolean;
  QQS_ctl1P_Init_In : {t,p | 2<=t<=Y+1; p=1} of boolean;
  M4 : {t,p | 0<=t<=Y; p=0} of integer;
let
  DB_mirr1[t,p] = DB_mirr1In[t,p];
  DDB_reg4loc[t,p] =
      case
        { | 3<=p} : DDB1[t,p-1];
        { | p=2} : DDB2[t,p-1];
      esac;
  expr1[t,p] =
      case
        { | 3<=p} : M1[t,p-1];
        { | p=2} : M2[t,p-1];
        { | p=1} : M4[t,p-1];
      esac;
  M[t,p] =
```

```
        case
          { | 2<=p} : M1[t,p];
          { | p=1} : M2[t,p];
          { | p=0} : M4[t,p];
        esac;
    M_ctl2P_Init_In[t,p] = M_ctl2P_Init[t];
    M_ctl2P_reg1loc[t,p] =
        case
          { | 3<=p} : M_ctl2P1[t,p-1];
          { | p=2} : M_ctl2P2[t,p-1];
        esac;
    MOut[t,p] = M[t,p];
    QQS_ctl1P_Init_In[t,p] = QQS_ctl1P_Init[t];
    QQS_ctl1P_reg2loc[t,p] =
        case
          { | 3<=p} : QQS_ctl1P1[t,p-1];
          { | p=2} : QQS_ctl1P2[t,p-1];
        esac;
    QQS_In_reg3loc[t,p] =
        case
          { | 3<=p} : QQS_In1[t,p-1];
          { | p=2} : QQS_In2[t,p-1];
        esac;
    QS_mirr1[t,p] = QS_mirr1In[t,p];
    use  ControlsequenceModule[X,Y] ()
         returns  (M_ctl2P_Init, QQS_ctl1P_Init) ;
    use {p | 2<=p<=X} cellsequenceModule1[p,X,Y]
        (expr1, DDB_reg4loc,QQS_In_reg3loc,
         QQS_ctl1P_reg2loc, M_ctl2P_reg1loc)
          returns  (M1, DDB1, QQS_In1, QQS_ctl1P1, M_ctl2P1) ;
    use {p | p=1} cellsequenceModule2[p,X,Y]
          (expr1, DB_mirr1, QS_mirr1, M_ctl2P_Init_In, QQS_ctl1P_Init_In)
          returns  (M2, DDB2, QQS_In2, QQS_ctl1P2, M_ctl2P2) ;
    use {p | p=0} cellsequenceModule4[p,X,Y] () returns  (M4) ;
  tel;
```

### A.3.3 Cellule 1 du programme AlpHard

```
system cellsequenceModule1 :{p | 2<=p<=10}
    (expr1 : {t | p-1<=t<=p+99} of integer;
     DDB_reg4loc : {t | p<=t<=p+99} of integer;
     QQS_In_reg3loc : {t | 2p-11<=t<=p} of integer;
     QQS_ctl1P_reg2loc : {t | p<=t<=p+99} of boolean;
     M_ctl2P_reg1loc : {t | p-1<=t<=p+99} of boolean)
returns
    (M : {t | p<=t<=p+100} of integer;
     DDB : {t | p+1<=t<=p+100} of integer;
     QQS_In : {t | 2p-9<=t<=p+1} of integer;
     QQS_ctl1P : {t | p+1<=t<=p+100} of boolean;
     M_ctl2P : {t | p<=t<=p+100} of boolean);
var
  M_ctl2Ploc5 : {t | p<=t<=p+100} of boolean;
  QQS_ctl1Ploc4 : {t | p+1<=t<=p+100} of boolean;
  QQS_Inloc3 : {t | 2p-9<=t<=p+1} of integer;
  DDBloc2 : {t | p+1<=t<=p+100} of integer;
  Mloc1 : {t | p<=t<=p+100} of integer;
  DDB_reg4 : {t | p+1<=t<=p+100} of integer;
  MatchQ : {t | p+1<=t<=p+100} of integer;
  M_ctl2P_reg1 : {t | p<=t<=p+100} of boolean;
  M_reg6 : {t | p+1<=t<=p+100} of integer;
  M_reg7 : {t | p+1<=t<=p+100} of integer;
  M_reg8 : {t | p+1<=t<=p+100} of integer;
  QQS : {t | p+1<=t<=p+100} of integer;
  QQS_ctl1P_reg2 : {t | p+1<=t<=p+100} of boolean;
  QQS_In_reg3 : {t | 2p-9<=t<=p+1} of integer;
  QQS_reg5 : {t | p+2<=t<=p+100} of integer;
  TSep1 : {t | p+1<=t<=p+100} of boolean;
  TSep2 : {t | p+1<=t<=p+100} of integer;
  TSep3 : {t | p+1<=t<=p+100} of integer;
  TSep4 : {t | p+1<=t<=p+100} of integer;
  TSep5 : {t | p+1<=t<=p+100} of integer;
  TSep6 : {t | p+1<=t<=p+100} of integer;
  TSep7 : {t | p+1<=t<=p+100} of integer;
  TSep8 : {t | p+1<=t<=p+100} of integer;
let
  M_ctl2P[t] = M_ctl2Ploc5[t];
  QQS_ctl1P[t] = QQS_ctl1Ploc4[t];
  QQS_In[t] = QQS_Inloc3[t];
  DDB[t] = DDBloc2[t];
  M[t] = Mloc1[t];
  TSep8[t] = max(TSep5[t], TSep6[t]);
  TSep7[t] = max(0[], TSep4[t]);
```

```
    TSep6[t] = M_reg8[t] + MatchQ[t];
    TSep5[t] = M_reg7[t] - 8[];
    TSep4[t] = M_reg6[t] - 8[];
    TSep3[t] = max(TSep7[t], TSep8[t]);
    M_reg8[t] = expr1[t-2];
    M_reg7[t] = expr1[t-1];
    DDB_reg4[t] = DDB_reg4loc[t-1];
    QQS_In_reg3[t] = QQS_In_reg3loc[t-2];
    QQS_ctl1P_reg2[t] = QQS_ctl1P_reg2loc[t-1];
    M_ctl2P_reg1[t] = M_ctl2P_reg1loc[t-1];
    M_reg6[t] = Mloc1[t-1];
    QQS_reg5[t] = QQS[t-1];
    QQS_Inloc3[t] = QQS_In_reg3[t];
    DDBloc2[t] = DDB_reg4[t];
    QQS_ctl1Ploc4[t] = QQS_ctl1P_reg2[t];
    QQS[t] =
        case
          { | t=p+1} : if (QQS_ctl1Ploc4[t]) then QQS_Inloc3[t] else 0[];
          { | p+2<=t} : if (QQS_ctl1Ploc4[t]) then 0[] else QQS_reg5[t];
        esac;
    M_ctl2Ploc5[t] = M_ctl2P_reg1[t];
    Mloc1[t] =
        case
          { | t=p} : if (M_ctl2Ploc5[t]) then 0[] else 0[];
          { | p+1<=t} : if (M_ctl2Ploc5[t]) then 0[] else TSep3[t];
        esac;
    TSep1[t] = QQS[t] = DDBloc2[t];
    TSep2[t] = -12[];
    MatchQ[t] = if (TSep1[t]) then 15[] else TSep2[t];
  tel;
```

### A.3.4 Cellule 4 du programme AlpHard

```
system cellsequenceModule4 :{p,X,Y | p=0; 3<=X<=Y-1}
                            ( )
       returns              (M : {t | 0<=t<=Y} of integer);
let
  M[t] = 0[];
tel;
```

### A.3.5 Controleur du programme AlpHard

```
system ControlsequenceModule :{X,Y | 3<=X<=Y-1}
         ( )
       returns
       (M_ctl2P_Init : {t | 1<=t<=Y+1} of boolean;
        QQS_ctl1P_Init : {t | 2<=t<=Y+1} of boolean);
let
   M_ctl2P_Init[t] =
       case
        { | t=1} : True[];
        { | 2<=t} : False[];
      esac;
  QQS_ctl1P_Init[t] =
       case
        { | t=2} : True[];
        { | 3<=t} : False[];
      esac;
tel;
```

## A.4 Code VHDL généré par MMAlpha

Cette section présente le code VHDL généré automatiquement à partir du programme AlpHard de la section précédente avec X ayant pour valeur 10 et Y ayant pour valeur 100. À chaque système Alpha de la section A.3 correspond une entité VHDL sauf pour l'interface qui n'est pas traduite. On présente les entités VHDL correspondant au module, à la cellule 2 et au contrôleur dans cet ordre.

### A.4.1 Module

```
- VHDL Model Created for "system sequenceModule"
 -  20/8/2000 17:23:53

library IEEE;
use IEEE.std_logic_1164.all;
 library test;
 use test.definition.all;


entity sequenceModule is
      Port ( Ck : In std_logic;
      CE : In std_logic;
      Rst : In std_logic ;
      DB_mirr1In : In DB_mirr1InType;
      QS_mirr1In : In QS_mirr1InType;
      MOut : Out MOutType );
end sequenceModule;


architecture Behavioral of sequenceModule is

  signal M_ctl2P_Init : std_logic;
  signal QQS_ctl1P_Init : std_logic;
  signal M : MType;
  signal M1 : M1Type;
  signal expr1 : expr1Type;
  signal DDB1 : DDB1Type;
  signal DDB_reg4loc : DDB_reg4locType;
  signal QQS_In1 : QQS_In1Type;
  signal QQS_In_reg3loc : QQS_In_reg3locType;
  signal QQS_ctl1P1 : std_logic_vector(2 to 10);
  signal QQS_ctl1P_reg2loc : std_logic_vector(2 to 11);
  signal M_ctl2P1 : std_logic_vector(2 to 10);
  signal M_ctl2P_reg1loc : std_logic_vector(2 to 11);
  signal M2 : M2Type;
```

```
    signal DB_mirr1 : DB_mirr1Type;
    signal QS_mirr1 : QS_mirr1Type;
    signal DDB2 : DDB2Type;
    signal QQS_In2 : QQS_In2Type;
    signal QQS_ctl1P2 : std_logic_vector(1 to 1);
    signal M_ctl2P2 : std_logic_vector(1 to 1);
    signal M_ctl2P_Init_In : std_logic_vector(1 to 1);
    signal QQS_ctl1P_Init_In : std_logic_vector(1 to 1);
    signal M4 : M4Type;

  Component cellsequenceModule1
        Port ( Ck : In std_logic;
        CE : In std_logic;
        Rst : In std_logic ;
        expr1 : In Integer range -32767 to 32767;
        DDB_reg4loc : In Integer range -32767 to 32767;
        QQS_In_reg3loc : In Integer range -32767 to 32767;
        QQS_ctl1P_reg2loc : In std_logic;
        M_ctl2P_reg1loc : In std_logic;
        M : Out Integer range -32767 to 32767;
        DDB : Out Integer range -32767 to 32767;
        QQS_In : Out Integer range -32767 to 32767;
        QQS_ctl1P : Out std_logic;
        M_ctl2P : Out std_logic );
  end Component;


  Component cellsequenceModule2
        Port ( Ck : In std_logic;
        CE : In std_logic;
        Rst : In std_logic ;
        expr1 : In Integer range -32767 to 32767;
        DB_mirr1 : In Integer range -32767 to 32767;
        QS_mirr1 : In Integer range -32767 to 32767;
        M_ctl2P_Init_In : In std_logic;
        QQS_ctl1P_Init_In : In std_logic;
        M : Out Integer range -32767 to 32767;
        DDB : Out Integer range -32767 to 32767;
        QQS_In : Out Integer range -32767 to 32767;
        QQS_ctl1P : Out std_logic;
        M_ctl2P : Out std_logic );
  end Component;


  Component cellsequenceModule4
```

```
      Port ( Ck : In std_logic;
      CE : In std_logic;
      Rst : In std_logic ;
      M : Out Integer range -32767 to 32767 );
end Component;


Component ControlsequenceModule
      Port ( Ck : In std_logic;
      CE : In std_logic;
      Rst : In std_logic ;
      M_ctl2P_Init : Out std_logic;
      QQS_ctl1P_Init : Out std_logic );
end Component;


begin

ETIQUETTE1 : FOR i IN 1 to 1 GENERATE
   DB_mirr1(i) <= DB_mirr1In(i);
END GENERATE ETIQUETTE1;

ETIQUETTE2 : FOR i IN 3 to 11 GENERATE
   DDB_reg4loc(i) <= DDB1(i-1);
END GENERATE ETIQUETTE2;

ETIQUETTE3 : FOR i IN 2 to 2 GENERATE
   DDB_reg4loc(i) <= DDB2(i-1);
END GENERATE ETIQUETTE3;

ETIQUETTE4 : FOR i IN 3 to 11 GENERATE
   expr1(i) <= M1(i-1);
END GENERATE ETIQUETTE4;

ETIQUETTE5 : FOR i IN 2 to 2 GENERATE
   expr1(i) <= M2(i-1);
END GENERATE ETIQUETTE5;

ETIQUETTE6 : FOR i IN 1 to 1 GENERATE
   expr1(i) <= M4(i-1);
END GENERATE ETIQUETTE6;

ETIQUETTE7 : FOR i IN 2 to 10 GENERATE
   M(i) <= M1(i);
END GENERATE ETIQUETTE7;
```

```
ETIQUETTE8 : FOR i IN 1 to 1 GENERATE
   M(i) <= M2(i);
END GENERATE ETIQUETTE8;

ETIQUETTE9 : FOR i IN 0 to 0 GENERATE
   M(i) <= M4(i);
END GENERATE ETIQUETTE9;

ETIQUETTE10 : FOR i IN 1 to 1 GENERATE
   M_ctl2P_Init_In(i) <= M_ctl2P_Init;
END GENERATE ETIQUETTE10;

ETIQUETTE11 : FOR i IN 3 to 11 GENERATE
   M_ctl2P_reg1loc(i) <= M_ctl2P1(i-1);
END GENERATE ETIQUETTE11;

ETIQUETTE12 : FOR i IN 2 to 2 GENERATE
   M_ctl2P_reg1loc(i) <= M_ctl2P2(i-1);
END GENERATE ETIQUETTE12;

ETIQUETTE13 : FOR i IN 0 to 10 GENERATE
   MOut(i) <= M(i);
END GENERATE ETIQUETTE13;

ETIQUETTE14 : FOR i IN 1 to 1 GENERATE
   QQS_ctl1P_Init_In(i) <= QQS_ctl1P_Init;
END GENERATE ETIQUETTE14;

ETIQUETTE15 : FOR i IN 3 to 11 GENERATE
   QQS_ctl1P_reg2loc(i) <= QQS_ctl1P1(i-1);
END GENERATE ETIQUETTE15;

ETIQUETTE16 : FOR i IN 2 to 2 GENERATE
   QQS_ctl1P_reg2loc(i) <= QQS_ctl1P2(i-1);
END GENERATE ETIQUETTE16;

ETIQUETTE17 : FOR i IN 3 to 11 GENERATE
   QQS_In_reg3loc(i) <= QQS_In1(i-1);
END GENERATE ETIQUETTE17;

ETIQUETTE18 : FOR i IN 2 to 2 GENERATE
   QQS_In_reg3loc(i) <= QQS_In2(i-1);
END GENERATE ETIQUETTE18;
```

```
ETIQUETTE19 : FOR i IN 1 to 1 GENERATE
   QS_mirr1(i) <= QS_mirr1In(i);
END GENERATE ETIQUETTE19;

ETIQUETTE20: ControlsequenceModule Port Map(Ck,CE,Rst, M_ctl2P_Init,QQS_ctl1P_Init);

ETIQUETTE21 : FOR i IN 2 to 10 GENERATE
   ETIQUETTE22: cellsequenceModule1 PORT MAP
      ( Ck,CE,Rst, expr1(i),DDB_reg4loc(i),QQS_In_reg3loc(i),QQS_ctl1P_reg2loc(i),
        M_ctl2P_reg1loc(i),M1(i),DDB1(i),QQS_In1(i),QQS_ctl1P1(i),M_ctl2P1(i));
 END GENERATE  ETIQUETTE21;

ETIQUETTE23 : FOR i IN 1 to 1 GENERATE
   ETIQUETTE24: cellsequenceModule2 PORT MAP
       ( Ck,CE,Rst, expr1(i),DB_mirr1(i),QS_mirr1(i),M_ctl2P_Init_In(i),
        QQS_ctl1P_Init_In(i),M2(i),DDB2(i),QQS_In2(i),QQS_ctl1P2(i),M_ctl2P2(i));
 END GENERATE  ETIQUETTE23;

ETIQUETTE25 : FOR i IN 0 to 0 GENERATE
   ETIQUETTE26: cellsequenceModule4 PORT MAP( Ck,CE,Rst, M4(i));
 END GENERATE  ETIQUETTE25;

end Behavioral;
```

### A.4.2 cellule 1

```
- VHDL Model Created for "system cellsequenceModule1"
 -  23/8/2000 16:59:3

library IEEE;
use IEEE.std_logic_1164.all;

entity cellsequenceModule1 is
      Port ( Ck : In std_logic;
      CE : In std_logic;
      Rst : In std_logic ;
      expr1 : In Integer range -32767 to 32767;
      DDB_reg4loc : In Integer range -32767 to 32767;
      QQS_In_reg3loc : In Integer range -32767 to 32767;
      QQS_ctl1P_reg2loc : In std_logic;
      M_ctl2P_reg1loc : In std_logic;
      M : Out Integer range -32767 to 32767;
      DDB : Out Integer range -32767 to 32767;
      QQS_In : Out Integer range -32767 to 32767;
      QQS_ctl1P : Out std_logic;
      M_ctl2P : Out std_logic );
end cellsequenceModule1;


architecture Behavioral of cellsequenceModule1 is

  signal M_ctl2Ploc5 : std_logic;
  signal QQS_ctl1Ploc4 : std_logic;
  signal QQS_Inloc3 : Integer range -32767 to 32767;
  signal DDBloc2 : Integer range -32767 to 32767;
  signal Mloc1 : Integer range -32767 to 32767;
  signal DDB_reg4 : Integer range -32767 to 32767;
  signal MatchQ : Integer range -32767 to 32767;
  signal M_ctl2P_reg1 : std_logic;
  signal M_reg6 : Integer range -32767 to 32767;
  signal M_reg7 : Integer range -32767 to 32767;
  signal M_reg8 : Integer range -32767 to 32767;
  signal QQS : Integer range -32767 to 32767;
  signal QQS_ctl1P_reg2 : std_logic;
  signal QQS_In_reg3 : Integer range -32767 to 32767;
  signal QQS_reg5 : Integer range -32767 to 32767;
  signal TSep1 : std_logic;
  signal TSep2 : Integer range -32767 to 32767;
  signal TSep3 : Integer range -32767 to 32767;
  signal TSep4 : Integer range -32767 to 32767;
```

```
    signal TSep5 : Integer range -32767 to 32767;
    signal TSep6 : Integer range -32767 to 32767;
    signal TSep7 : Integer range -32767 to 32767;
    signal TSep8 : Integer range -32767 to 32767;
    signal M_reg8_S1 : Integer range -32767 to 32767;
    signal QQS_In_reg3_S1 : Integer range -32767 to 32767;

  begin


    M_ctl2P <= M_ctl2Ploc5;

    QQS_ctl1P <= QQS_ctl1Ploc4;

    QQS_In <= QQS_Inloc3;

    DDB <= DDBloc2;

    M <= Mloc1;
  process(TSep8,TSep5,TSep6)
     begin
     if TSep5>TSep6 then TSep8 <= TSep5;
      else TSep8 <= TSep6;
      end if;
   end process; process(TSep7,0,TSep4)
     begin
     if 0>TSep4 then TSep7 <= 0;
      else TSep7 <= TSep4;
      end if;
   end process;
    TSep6 <= (M_reg8 + MatchQ);

    TSep5 <= (M_reg7 - 8);

    TSep4 <= (M_reg6 - 8);
  process(TSep3,TSep7,TSep8)
     begin
     if TSep7>TSep8 then TSep3 <= TSep7;
      else TSep3 <= TSep8;
      end if;
   end process;
    process(ck)
    begin
      if (ck='1' AND ck'event) then
       if (CE='1') then
```

```
     M_reg8_S1 <= expr1;
    end if;
 end if;
end process;

process(ck)
begin
if (ck='1' AND ck'event) then
   if (CE='1') then
     M_reg8 <= M_reg8_S1;
    end if;
 end if;
end process;

process(ck)
begin
if (ck='1' AND ck'event) then
   if (CE='1') then
     M_reg7 <= expr1;
    end if;
 end if;
end process;

process(ck)
begin
if (ck='1' AND ck'event) then
   if (CE='1') then
     DDB_reg4 <= DDB_reg4loc;
    end if;
 end if;
end process;

process(ck)
begin
  if (ck='1' AND ck'event) then
   if (CE='1') then
     QQS_In_reg3_S1 <= QQS_In_reg3loc;
    end if;
 end if;
end process;

process(ck)
begin
if (ck='1' AND ck'event) then
   if (CE='1') then
```

```
     QQS_In_reg3 <= QQS_In_reg3_S1;
    end if;
 end if;
end process;

process(ck)
begin
if (ck='1' AND ck'event) then
   if (CE='1') then
     QQS_ctl1P_reg2 <= QQS_ctl1P_reg2loc;
    end if;
 end if;
end process;

process(ck)
begin
if (ck='1' AND ck'event) then
   if (CE='1') then
     M_ctl2P_reg1 <= M_ctl2P_reg1loc;
    end if;
 end if;
end process;

process(ck)
begin
if (ck='1' AND ck'event) then
   if (CE='1') then
     M_reg6 <= Mloc1;
    end if;
 end if;
end process;

process(ck)
begin
if (ck='1' AND ck'event) then
   if (CE='1') then
     QQS_reg5 <= QQS;
    end if;
 end if;
end process;

QQS_Inloc3 <= QQS_In_reg3;

DDBloc2 <= DDB_reg4;
```

```
  QQS_ctl1Ploc4 <= QQS_ctl1P_reg2;

  QQS <=
    QQS_Inloc3 when QQS_ctl1Ploc4 = '1'  else
    0 when QQS_ctl1Ploc4 = '1'  else QQS_reg5;

  M_ctl2Ploc5 <= M_ctl2P_reg1;

  Mloc1 <=
    0 when M_ctl2Ploc5 = '1'  else
    0 when M_ctl2Ploc5 = '1'  else TSep3;

  TSep1 <= (QQS = DDBloc2);

  TSep2 <= ( - 12);

  MatchQ <=
    15 when (TSep1 = '1' ) else TSep2;


end Behavioral;
```

### A.4.3 Controleur

```
- VHDL Model Created for "system ControlsequenceModule"
 -  20/8/2000 17:23:44

library IEEE;
use IEEE.std_logic_1164.all;

entity ControlsequenceModule is
      Port ( Ck : In std_logic;
      CE : In std_logic;
      Rst : In std_logic ;
      M_ctl2P_Init : Out std_logic;
      QQS_ctl1P_Init : Out std_logic );
end ControlsequenceModule;


architecture state_machine of ControlsequenceModule is
  signal cpt: Integer;

  type M_ctl2P_InitStates is (E0,E0bis,E1,E2);
  signal currentM_ctl2P_InitState, nextM_ctl2P_InitState :M_ctl2P_InitStates;

  type QQS_ctl1P_InitStates is (E0,E0bis,E1,E2);
  signal currentQQS_ctl1P_InitState, nextQQS_ctl1P_InitState :QQS_ctl1P_InitStates;
BEGIN

reset_sm : PROCESS
 begin
- compass stateMachine adj currentM_ctl2P_InitState
- compass stateMachine adj currentQQS_ctl1P_InitState

  WAIT UNTIL (Ck = '1' AND Ck'event);

  IF Rst ='1' THEN
   cpt <= -1;
   currentM_ctl2P_InitState <= E0;
   currentQQS_ctl1P_InitState <= E0;
  ELSE
   cpt <= cpt + 1;
   currentM_ctl2P_InitState <= nextM_ctl2P_InitState;
   currentQQS_ctl1P_InitState <= nextQQS_ctl1P_InitState;
  END IF;
 END PROCESS;

evolution_M_ctl2P_Init : PROCESS(cpt, currentM_ctl2P_InitState)
```

```
 begin
 CASE currentM_ctl2P_InitState IS
  WHEN E0 => IF (cpt < 0) THEN nextM_ctl2P_InitState <= E0;
    ELSIF (cpt = 0) THEN nextM_ctl2P_InitState <= E1;
 ELSE  nextM_ctl2P_InitState <= E0bis; -error
 END IF;
  WHEN E1 =>  IF (cpt = 1) THEN nextM_ctl2P_InitState <= E2;
 ELSE nextM_ctl2P_InitState <= E0bis; - error
 END IF;
  WHEN E2=> IF (cpt >= 2) THEN nextM_ctl2P_InitState <= E2;
 ELSE nextM_ctl2P_InitState <= E0bis; - error
 END IF;
  WHEN OTHERS => - erreurs et hors service
   nextM_ctl2P_InitState <= E0bis ;
 END CASE;
END PROCESS;

output_M_ctl2P_Init : PROCESS(currentM_ctl2P_InitState)
 begin
 CASE currentM_ctl2P_InitState IS
  WHEN E1=>M_ctl2P_Init <= '1';
  WHEN E2=>M_ctl2P_Init <= '0';
  WHEN OTHERS =>
   M_ctl2P_Init <= 'X';
 END CASE;
END PROCESS;
evolution_QQS_ctl1P_Init : PROCESS(cpt, currentQQS_ctl1P_InitState)
 begin
 CASE currentQQS_ctl1P_InitState IS
  WHEN E0 => IF (cpt < 1) THEN nextQQS_ctl1P_InitState <= E0;
    ELSIF (cpt = 1) THEN nextQQS_ctl1P_InitState <= E1;
 ELSE  nextQQS_ctl1P_InitState <= E0bis; -error
 END IF;
  WHEN E1 =>  IF (cpt = 2) THEN nextQQS_ctl1P_InitState <= E2;
 ELSE nextQQS_ctl1P_InitState <= E0bis; - error
 END IF;
  WHEN E2=> IF (cpt >= 3) THEN nextQQS_ctl1P_InitState <= E2;
 ELSE nextQQS_ctl1P_InitState <= E0bis; - error
 END IF;
  WHEN OTHERS => - erreurs et hors service
   nextQQS_ctl1P_InitState <= E0bis ;
 END CASE;
END PROCESS;

output_QQS_ctl1P_Init : PROCESS(currentQQS_ctl1P_InitState)
```

```
 begin
 CASE currentQQS_ctl1P_InitState IS
  WHEN E1=>QQS_ctl1P_Init <= '1';
  WHEN E2=>QQS_ctl1P_Init <= '0';
  WHEN OTHERS =>
   QQS_ctl1P_Init <= 'X';
 END CASE;
END PROCESS;

END state_machine;
```

# Annexe B

# Liste de packages de MMAlpha