UNIVERSITÉ PARIS 7
UFR D'INFORMATIQUE

THÈSE

présentée pour obtenir

LE GRADE DE DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ PARIS 7

Spécialité : INFORMATIQUE

par

Patrice BERTIN

Sujet :

Mémoires actives programmables : conception, réalisation et programmation.

Soutenue le 14 juin 1993 devant la commission d'examen composée de :

Pierre	FAURRE	*Président*
Gérard	BERRY	*Rapporteur*
Dominique	PERRIN	*Examinateur*
Patrice	QUINTON	*Rapporteur*
Charles P.	THACKER	*Examinateur*
Jean	VUILLEMIN	*Directeur de thèse*

Je tiens en tout premier lieu à exprimer mes remerciements et ma plus profonde gratitude à Jean Vuillemin, à qui ce travail doit tout, de l'inspiration initiale à l'impulsion finale. Il a créé et guidé ce projet à partir d'une vision audacieuse mais sûre, et a toujours su développer la motivation permettant de dépasser les obstacles rencontrés en chemin. Sans sa présence constante rien de ce qui est relaté ici n'aurait abouti.

Je tiens aussi à reconnaître l'apport essentiel de Didier Roncin, dont la qualité d'intuition, la compétence technique et la force de travail ont permis de concrétiser réellement ce qui ne serait autrement resté qu'une belle idée.

Je veux ici remercier Pierre Faurre de m'avoir fait l'honneur d'accepter de présider le jury de cette thèse. Je remercie également Gérard Berry et Patrice Quinton d'avoir accepté la lourde et délicate charge de rapporteurs de ce travail, ainsi que Dominique Perrin et Chuck Thacker qui ont bien voulu faire partie du jury.

Je suis très reconnaissant envers Patrick Baudelaire et les membres du Laboratoire de Recherches de Paris de la société Digital Equipment Corporation pour la qualité de l'environnement de recherche qu'ils ont su développer. Je remercie sans pouvoir les citer toutes les personnes de ce laboratoire qui ont contribué d'une façon ou d'une autre à ce travail, ainsi que celles des laboratoires frères Systems Research Center et Western Research Laboratory. Une mention particulière va à Thierry Pudet qui m'a fait l'amitié de relire avec beaucoup de soin le manuscrit de cette thèse.

Enfin, ma participation à ce projet a été rendue possible par un contrat de collaboration entre l'Institut National de Recherches en Informatique et en Automatique et le Laboratoire de Recherches de Paris de la société Digital Equipment Corporation. Que ces deux institutions en soient ici remerciées.

Je dédie cette thèse à mon épouse Sylvie.

Résumé

On appelle *mémoire active programmable* (ou PAM, de l'anglais Programmable Active Memory) un coprocesseur matériel configurable d'usage général connecté à un ordinateur conventionnel. Un ordinateur équipé d'une PAM peut accélérer l'exécution d'un algorithme particulier à un instant donné en configurant cette PAM en un circuit logique spécialisé, calculant directement les opérations critiques de cet algorithme. Pour toute une classe d'applications, un tel couple {Hôte + PAM} permet d'obtenir des performances comparables ou même supérieures à celles des super-ordinateurs, pour un volume et un prix comparables à ceux d'une station de travail traditionnelle.

La fabrication de PAMs de taille conséquente a été rendue possible par l'introduction récente d'une nouvelle famille de composants logiques, les *prédiffusés programmables*. Cette thèse rassemble et analyse l'expérience acquise au cours d'un projet explorant la conception, la réalisation et l'utilisation des PAMs, conduit depuis 1988 au laboratoire DEC-PRL sous la direction de Jean Vuillemin. Au cours de ce projet, deux architectures successives de PAM ont été realisées, un environnement de programmation matérielle spécialisé a été développé, et plus d'une dizaine d'applications majeures ont été programmées.

On commence par définir précisément le concept par la donnée d'un modèle fondamental. On montre que tout circuit logique synchrone peut être réalisé sur une PAM, pourvu que celle-ci soit assez grande et la fréquence de fonctionnement assez faible. On définit une unité fondamentale de mesure de la puissance de calcul des circuits matériels permettant de comparer la puissance des PAMs à celle des processeurs traditionnels.

On étudie ensuite les problèmes architecturaux liés à la conception des PAMs. En s'appuyant sur l'exemple des deux architectures effectivement realisées au laboratoire DEC-PRL, on étudie en détail les étapes successives de conception menant d'un circuit prédiffusé programmable du commerce à une PAM opérationnelle.

On aborde enfin les problèmes logiciels liés à la programmation et à l'utilisation des PAMs. On décrit tout d'abord une suite d'outils permettant d'aborder la programmation matérielle sur PAM avec la même méthodologie que la programmation logicielle classique. On présente ensuite quelques problèmes liés à l'utilisation efficace et sûre des PAMs, et on décrit les solutions qui ont été retenues pour y répondre.

On conclut par une estimation du potentiel d'évolution de cette technologie, en rassemblant les principaux problèmes architecturaux et logiciels encore ouverts dont cette évolution dépend. Deux articles précedemment publiés sont en annexe (en anglais); ils décrivent les principales applications ayant été programmées sur PAM au cours de ce projet, ainsi que les résultats et niveaux de performances obtenus.

Mots-clés: PAM, mémoire active programmable, prédiffusé programmable, coprocesseur configurable, logique synchrone.

Abstract

We call *Programmable Active Memory* (or PAM for short) a general-purpose, configurable, hardware coprocessor connected to a standard computer. When thus equipped, a computer can speed up the execution of a given algorithm, by configuring the PAM on demand into a specialized hardware circuit which directly computes the critical operations of the algorithm. On many applications, such a {Host + PAM} pair can achieve a level of performance comparable to or greater than that of a super-computer, for a price and a size comparable to that of a traditional workstation.

The recent introduction of a new family of logic devices, called Field-Programmable Gate Arrays (FPGAs), made it feasible to build PAMs of a substantial size. This thesis presents the results of a research project aimed at exploring the design, fabrication and use of efficient PAMs, and tries to analyse the experience gained from it. During the course of this project, which was conducted at DEC's Paris Research Laboratory under the direction of Jean Vuillemin, two successive PAM architectures have been built, a specialized hardware programming environment has been developed, and more than ten major PAM applications have been successfully programmed.

We first provide a precise definition of the PAM concept through the introduction of a fundamental model. We show that every synchronous logic circuit can be realized on a large enough PAM for a slow enough clock. We introduce a fundamental unit for measuring hardware computing power, which can be used to compare PAMs to traditional CPUs.

We then study the architectural problems raised by the design of efficient PAMs. Through the example of two PAM architectures built at DEC-PRL, we review in detail the successive design stages leading from an off-the-shelf FPGA to an operational PAM.

We finally consider the software issues related to the programming and use of PAM coprocessors. We first describe a tool suite through which the hardware programming of a PAM can be done with the same methodology as traditional software programming. We then review some problems related to the efficient and safe use of a programmed PAM, and show possible solutions to them.

We conclude by an estimation of the evolution potential of this technology, gathering the main architectural and software-related open problems which will condition this evolution. Two previously published articles are presented in the appendix (in English); they describe the most important PAM applications which have been programmed so far, and the levels of performance which were obtained in each case.

Keywords: PAM, programmable active memory, field-programmable gate-array, configurable co-processor, synchronous logic.

Table des Matières

Introduction **1**

1 Mémoires actives programmables **7**
1.1 Circuits logiques synchrones 7
1.1.1 Définition 8
1.1.2 Réalisation 10
1.1.3 Circuits intégrés traditionnels 17
1.2 Mémoires actives programmables 18
1.2.1 Définition 18
1.2.2 Modèle 19
1.2.3 Mise en œuvre 22
1.2.4 Puissance de calcul 24
1.2.5 Motivation 27
1.2.6 Extensions possibles 28
1.3 Réalisations 29
1.3.1 Un exemple de circuit logique programmable 29
1.3.2 Projet PAM, laboratoire DEC-PRL 31
1.3.3 Autres réalisations 32

2 Architecture **35**
2.1 Blocs de base: circuits programmables 35
2.1.1 Eventail des possibilités 35
2.1.2 Choix retenus 41
2.2 Matrice principale 42
2.2.1 Dimensions 42
2.2.2 Interconnexions 44
2.2.3 Distribution d'horloge 48
2.3 Chemins de données externes 49
2.3.1 Canaux d'entrées-sorties 49

2.3.2 Interconnexion . . . 51
2.4 Contrôle . . . 54
2.4.1 Organisation mémoire . . . 54
2.4.2 Architecture . . . 55
2.5 Support . . . 58
2.5.1 Configuration . . . 58
2.5.2 Relecture . . . 59
2.5.3 Génération d'horloge . . . 60
2.5.4 Interface avec l'hôte . . . 63
2.5.5 Contrôle des mémoires . . . 70
2.6 Autres PAMs . . . 71
2.6.1 Quickturn RPM . . . 71
2.6.2 Splash . . . 73
2.6.3 Splash II . . . 75
2.6.4 Tableau comparatif . . . 77

3 Programmation **79**
3.1 Programmation matérielle . . . 79
3.1.1 Introduction . . . 80
3.1.2 Modèle . . . 83
3.1.3 Description des circuits . . . 85
3.1.4 Implémentation sur PAM . . . 90
3.2 Programmation logicielle . . . 94
3.2.1 Objectifs . . . 94
3.2.2 Bibliothèque d'exécution . . . 95
3.2.3 Mise au point . . . 98

4 Conclusions et perspectives **103**

A Introduction to Programmable Active Memories **115**
A.1 Concept . . . 116
A.2 Implementation . . . 117
A.3 PAM programming . . . 118
A.4 Design experiments . . . 119
A.4.1 Early experiments . . . 119
A.4.2 Long multiplication . . . 119
A.4.3 Data compression . . . 120
A.4.4 Image processing . . . 121

A.5 Conclusion 121

B Programmable Active Memories: a Performance Assessment 123
B.1 PAM concept 124
B.2 Two PAM architectures 125
B.3 PAM programming 126
B.4 Ten PAM applications 127
B.4.1 Long multiplication 128
B.4.2 RSA cryptography 128
B.4.3 Data compression 129
B.4.4 String matching 130
B.4.5 Heat and Laplace equations 130
B.4.6 Newton's mechanics 131
B.4.7 Binary 2D convolution 131
B.4.8 Boltzmann machine 132
B.4.9 3D Geometry 133
B.4.10 Discrete cosine transform 133
B.5 Conclusion 134

Bibliographie 139

Table des Illustrations

1.1 Les deux types de transistor CMOS 11
1.2 Deux portes logiques simples 11
1.3 Symboles conventionnels de représentation des portes logiques 12
1.4 Additionneur binaire complet 12
1.5 Registre synchrone 13
1.6 Fonctionnement du registre synchrone 14
1.7 Réalisation d'un circuit synchrone 15
1.8 Réalisation du compteur modulo 2^n 15
1.9 Bit actif programmable (PAB) 19
1.10 Mémoire active programmable: modèle fondamental 21
1.11 Mémoire active programmable: connexion à l'hôte 22
1.12 Une horloge temps-réel réalisée sur PAM 23
1.13 Configurations canoniques du PAB 25
1.14 Puissances de calcul de diverses architectures matérielles 27
1.15 Architecture du circuit Xilinx XC2064 30
1.16 Xilinx XC2064: bloc logique configurable 30
1.17 Réalisation du PAB fondamental sur XC2064 31

2.1 Xilinx XC2000: cellule de base (schéma simplifié) 36
2.2 Xilinx XC3000: cellule de base (schéma simplifié) 36
2.3 Xilinx XC3000: assemblage des cellules 37
2.4 Xilinx XC4000: cellule de base (schéma simplifié) 38
2.5 Algotronix CAL: PAB (schéma simplifié) 38
2.6 Plessey ERA: architecture générale 40
2.7 Concurrent Logic CFA6006: PAB et schéma d'interconnexion 40
2.8 Connexions directes 44
2.9 Bus 45
2.10 Anneaux 46
2.11 Utilisation des XC3090 pour Perle-1 46
2.12 Performance des interconnexions 47

2.13 Connexion de Perle-1 à sa machine hôte 49
2.14 Ports externes de Perle-1 51
2.15 Performance des entrée-sorties 51
2.16 Canaux d'entrée-sortie de Perle-0 52
2.17 Canaux d'entrée-sortie de Perle-1 52
2.18 Un exemple de chemin de données sur Perle-1 53
2.19 Organisation générale de Perle-0 56
2.20 Organisation générale de Perle-1 57
2.21 Perle-0: circuit de génération d'horloge 61
2.22 Utilisation synchrone de deux horloges 61
2.23 Perle-1: circuit de génération d'horloge 62
2.24 Perle-0: interface avec l'hôte 63
2.25 Perle-0: protocole de communication avec l'hôte 64
2.26 Perle-1: interface avec l'hôte 65
2.27 Un exemple de comportement du FIFO d'entrée en mode autostop 68
2.28 L'exemple de la figure 2.27, projeté sur l'horloge `clk0` 68
2.29 Cycle d'écriture d'une mémoire statique 70
2.30 Perle-1: modèle de l'interface RAM synchrone 71
2.31 Architecture de la machine Quickturn RPM 72
2.32 Architecture de la machine Splash 73
2.33 Architecture de la machine Splash II 76
2.34 Puissances comparées de différentes PAMs 77

3.1 Compteur modulo 2^n décrit en Le-Lisp 86

4.1 Evolution de la puissance des processeurs et de la taille des mémoires 105
4.2 Evolution de la puissance des PAMs 106

A.1 A simple PAM 116
A.2 Perle-0 118
A.3 Perle-0 setup 119
A.4 Two PAM designs 120

Introduction

Algorithmes logiciels ou matériels?

L'implantation des algorithmes est aujourd'hui quasi-universellement obtenue au moyen de réalisations logicielles opérant sur des machines universelles (processeurs) dont l'architecture est à peu près standardisée [HP90]. Pour cela, la suite d'opérations à effectuer est décomposée en une séquence d'instructions élémentaires qui sont directement implémentées par l'architecture matérielle du processeur utilisé. Ces instructions élémentaires sont en petit nombre, et varient peu selon les implémentations: les processeurs courants d'aujourd'hui calculent ainsi typiquement les opérations arithmétiques et logiques usuelles sur des nombres entiers de 32 bits[1], ainsi que quelques opérations arithmétiques simples sur des nombres à virgule flottante de 64 bits.

Lorsque le niveau de performance d'un ordinateur conventionnel est insuffisant, par exemple pour nombre de problèmes d'analyse numérique ou de traitement d'images numériques en temps réel, on peut utiliser des "super-ordinateurs", machines extrêmement coûteuses obtenues par exploitation maximale des technologies disponibles à l'instant courant. Les tendances actuelles en ce domaine reposent ainsi sur l'extension de l'architecture standard dans l'une des directions suivantes:

- les semi-conducteurs ultra-rapides, permettant d'obtenir des processeurs conventionnels à temps de cycle très court;
- le parallélisme massif, par interconnexion d'un grand nombre de processeurs conventionnels.

Il existe cependant une autre voie permettant d'obtenir un niveau de performance comparable à bien plus faible coût: c'est la réalisation d'un circuit intégré *sur mesure*, dont la structure est directement dérivée de la nature algorithmique de l'application, et qui calcule directement la suite d'opérations désirée (ou tout au moins les plus coûteuses d'entre elles). Cette approche est d'autant mieux adaptée que l'application s'exprime inefficacement au moyen des instructions machine conventionnelles. On appellera "réalisation matérielle d'un algorithme" la procédure consistant à fabriquer un tel circuit dédié; celle-ci offre les avantages et inconvénients suivants:

[1] Cette taille est parfois portée à 64 bits, pour les dernières générations en cours d'introduction.

Avantages Dans un volume et pour un prix de revient comparables à ceux d'un micro-ordinateur, on peut disposer d'une puissance de calcul comparable à celle d'un super-ordinateur, voire nettement supérieure dans certains cas.

La fabrication de circuit permet d'aborder toute une classe d'applications algorithmiques intrinsèquement matérielles et traditionnellement inaccessibles aux techniques logicielles (par exemple l'interfaçage avec un réseau de communication, ou l'acquisition et le filtrage en temps réel de données à haut débit).

Inconvénients La machine obtenue n'est utilisable que pour une classe très particulière d'applications, voire une seule.

Le coût et le temps de développement du prototype initial sont très élevés, sans commune mesure avec ceux du développement d'un logiciel pour super-ordinateur.

Mémoires actives programmables

L'apparition récente d'un nouvelle famille de circuits intégrés, les "prédiffusés programmables" [CD+86], a changé les données de ce problème. Destinés initialement au remplacement de la logique discrète sur les cartes électroniques, ce sont des composants logiques dont on peut changer dynamiquement la structure interne (équations logiques et structure d'interconnexion) par simple chargement d'un mot de configuration.

Par assemblage régulier de tels circuits, on peut obtenir une structure arbitrairement grande que l'on pourrait appeler un "méta-circuit", c'est-à-dire un dispositif matériel standard configurable à volonté et de façon logicielle en n'importe quel circuit digital synchrone particulier. En quelque sorte, un circuit configurable de taille infinie est au calcul parallèle ce qu'une machine de Turing est au calcul séquentiel.

On appellera *mémoire active programmable* (ou PAM, de l'anglais Programmable Active Memory) une telle structure connectée à un ordinateur conventionnel "hôte" en tant que *coprocesseur*. Un ordinateur équipé d'une PAM peut utiliser celle-ci à un instant donné pour accélérer un calcul donné, en procédant en deux temps:

1. on configure tout d'abord la PAM en y chargeant la description d'un circuit particulier calculant les opérations essentielles de l'application;

2. on utilise ensuite la PAM de la même façon qu'un coprocesseur conventionnel, en y écrivant les données du calcul et en y relisant les résultats.

Le calcul de la configuration particulière réalisant le circuit voulu est un processus de compilation qui demande typiquement quelques dizaines de minutes; elle est donc faite préalablement. Par contre le processus de configuration lui-même est très rapide (typiquement quelques dizaines de millisecondes). Une même PAM peut donc être reconfigurée plusieurs dizaines de fois par seconde, pour des applications différentes ou les phases successives d'une même application.

Un tel couple $\{$Hôte + PAM$\}$ fournit une plate-forme de développement d'algorithmes matériels à très faible temps de cycle (l'essai d'une nouvelle version ne demandant que le

temps nécessaire au calcul de la nouvelle configuration); à l'inverse des systèmes utilisant la simulation, l'algorithme une fois compilé s'exécutera à la vitesse réelle d'un circuit matériel, et pourra communiquer en temps réel avec des dispositifs matériels extérieurs.

D'autre part, si l'on compare (à technologie égale) les réalisations du même algorithme parallèle par un circuit spécifique et par une PAM, on constate que:

- la fréquence d'horloge maximale (et donc le niveau de performances) d'une réalisation PAM est généralement inférieure à celle d'un circuit spécifique d'un facteur inférieur à 2; dans bien des cas cependant, la fréquence de fonctionnement effective est limitée par des facteurs extérieurs (bandes passantes d'entrée-sortie) et ceci est de peu d'importance;
- la surface d'un circuit spécifique est inférieure de deux ordres de grandeur à celle d'une PAM, soit un composant ($\approx$ 1 cm^2) contre une carte ($\approx$ 100 cm^2);
- le temps de mise au point d'une application PAM est comparable à celui de l'écriture d'un logiciel très optimisé, soit quelques semaines, en regard des mois nécessaires à la réalisation d'un circuit spécifique;
- le coût initial de la solution PAM est nul[2], celui d'un circuit spécifique est de quelques centaines de milliers de francs. Le circuit spécifique ne devient donc rentable qu'à partir de plusieurs centaines de systèmes réalisés, alors qu'une PAM rend économiquement viable la réalisation unitaire d'un circuit pour une application particulière.

Projet PAM

C'est sur ces bases que Jean Vuillemin a initié le projet de recherches "PAM", à l'Institut National de Recherches en Informatique et en Automatique (INRIA) tout d'abord (automne 1987), puis au Laboratoire de Recherches de Paris de la société Digital Equipment Corporation (DEC-PRL, printemps 1988). L'objectif de ce projet est l'exploration des potentialités de cette technologie, à travers l'étude des questions suivantes:

Architecture Comment concevoir et réaliser des PAMs aussi performantes que possibles?

Applications Quelles applications bénéficient au mieux de cette technologie?

Programmation Quels outils faut-il développer pour tirer le meilleur parti de ces PAMs, le plus aisément possible?

Au cours de cette étude, deux architectures successives de PAMs ont été conçues et réalisées (appelées respectivement Perle-0 et Perle-1), un environnement complet de programmation et d'utilisation de celles-ci a été développé, et plus d'une douzaine d'applications représentatives de domaines très différents ont été programmées [BRV89, BRV93].

[2] à condition, bien entendu, de disposer de cette technologie.

Contenu de cette thèse

Cette thèse tente de rassembler et d'analyser l'expérience acquise au cours de ce projet. Elle est organisée en trois parties principales, qui reflètent les différentes facettes de l'étude:

1. Le premier chapitre précise la définition et quelques propriétés du concept de mémoire active programmable. Il commence par introduire les notations et résultats essentiels concernant le modèle logique synchrone, qui serviront de base à toute la suite. Il présente ensuite un modèle fondamental abstrait de mémoire active programmable permettant de définir précisément le concept, ainsi qu'une unité de mesure permettant de quantifier la performance des PAMs et de les comparer aux autres architectures matérielles. Il passe enfin en revue les implémentations connues de ce concept.

2. Le deuxième chapitre étudie en détail les problèmes architecturaux posés par la conception des PAMs. En s'appuyant extensivement sur les exemples des deux architectures réalisées au laboratoire DEC-PRL, il décrit progressivement les différentes étapes de la conception physique d'une PAM. Des comparaisons sont tentées avec les autres architectures apparentées connues.

3. Le troisième chapitre traite des aspects logiciels liés à la programmation et à l'exploitation des PAMs. Il présente tout d'abord le problème de la programmation matérielle sur PAM, et décrit l'environnement de programmation qui a été réalisé afin de pouvoir développer les applications désirées, les mettre au point et en obtenir le niveau de performances escompté. Une deuxième section présente quelques problèmes liés à l'utilisation de la PAM une fois programmée, et décrit succintement l'architecture logicielle qui a été retenue pour y répondre.

On conclut en tentant d'évaluer le potentiel d'évolution de cette technologie. Pour les deux aspects principaux que sont l'architecture et la programmation des PAMs, on présente un certain nombre de problèmes encore ouverts qui semblent conditionner cette évolution.

On trouvera enfin en annexe la réimpression de deux articles précédemment publiés [BRV89, BRV93], qui décrivent les principales applications programmées respectivement sur Perle-0 et Perle-1 ainsi que les résultats et niveaux de performance obtenus.

Contributions

Les résultats présentés ici sont le fruit d'un travail collectif ayant impliqué un nombre variable de personnes pendant plusieurs années. Il est de ce fait impossible de délimiter précisément la contribution de chacun. Pour fixer les idées, une estimation de la contribution personnelle de l'auteur de cette thèse pourrait être résumée comme suit:

Idée initiale L'inspiration initiale, les premières évaluations et les premières grandes lignes d'architecture sont entièrement dues à Jean Vuillemin.

Etude théorique La définition du modèle PAM abstrait, l'étude de ses propriétés et la caractérisation des unités de mesure de performances sont de Jean Vuillemin et de l'auteur.

Architecture Les deux PAMs réalisées ont été conçues en commun par Didier Roncin, Jean Vuillemin et l'auteur.

Réalisation La conception détaillée des PAMs, leur réalisation physique et leur mise au point ont été essentiellement faites par Didier Roncin et l'auteur, avec l'aide d'Alfred Permuy et Philippe Boucard dans le cas de Perle-1 (ce travail n'est pas décrit dans cette thèse).

Logiciel Le logiciel de programmation de Perle-0 ainsi que les logiciels d'utilisation de Perle-0 et Perle-1 ont été conçus et réalisés par l'auteur, avec des apports ponctuels de Mark Shand et Marcin Skubiszewski. Le logiciel de programmation de Perle-1 (qui n'est pas décrit dans cette thèse) est en cours de réalisation par Hervé Touati et d'autres membres du laboratoire DEC-PRL.

Applications L'auteur a peu contribué aux diverses applications qui ont été programmées sur Perle-0 et Perle-1, à l'exception des versions du multiplicateur décrites en [SBV91]. Les auteurs respectifs de ces applications sont citées dans les articles [BRV89] et [BRV93] qui les décrivent.

En plus de celles qui sont nommées ci-dessus, d'autres personnes ont participé à ce projet qui sont trop nombreuses pour être toutes citées ici. Dans toute la suite de cette thèse, l'auteur a essayé de mentionner les personnes impliquées là où c'était possible.

Chapitre 1

Mémoires actives programmables

Ce chapitre est consacré à la définition et à l'étude abstraite du concept de mémoire active programmable, indépendamment de toute réalisation physique. La section 1.1 définit le modèle logique synchrone dans lequel on se place. La section 1.2 présente le concept de mémoire active programmable et étudie ses propriétés essentielles. Enfin, la section 1.3 passe en revue les principales réalisations connues de ce concept.

1.1 Circuits logiques synchrones

La quasi-totalité des machines de calcul et de traitement de l'information modernes sont fabriquées au moyen de circuits électroniques. Selon la fonction qu'ils implémentent et la technologie mise en œuvre pour les réaliser, ceux-ci peuvent être séparés en deux familles bien distinctes:

- Les circuits *analogiques* sont modélisables par la donnée d'un certain nombre de paramètres électriques (tension, intensité...) variant *continûment* dans le temps.
- Les circuits *logiques* sont modélisables par la donnée *à une suite discrète d'instants* d'un certain nombre de paramètres pouvant prendre un *nombre fini de valeurs*.

Les premiers sont traditionnellement utilisés pour des fonctions de "bas niveau" (interfaçage physique, communication). Les seconds sont universellement employés pour toutes les fonctions "algorithmiques" (processeurs universels, circuits arithmétiques, de traitement du signal, etc...)[1].

Les circuits logiques peuvent eux-mêmes être regroupés en plusieurs familles qui répondront à des modèles différents. On distinguera en particulier:

- la logique *booléenne*, pour laquelle les paramètres physiques permettant de représenter le système ne peuvent prendre que deux valeurs, ou au contraire la logique *multivaluée*

[1] On trouvera cependant dans [Mea89] des exemples intéressants d'utilisation de circuits analogiques pour des fonctions algorithmiques.

(ternaire, quaternaire,...) lorsque ces paramètres varient sur un ensemble à trois éléments ou plus;

- la logique *synchrone*, lorsque toutes les grandeurs du système changent de valeur simultanément lors d'une suite discrète d'instants prédéfinis; la logique *biphase*, lorsque certaines grandeurs changent sur les instants pairs et les autres sur les instants impairs d'une suite discrète d'instants; la logique *auto-temporisée* (en anglais *self-timed*) lorsque les grandeurs caractérisant le système peuvent changer de valeur à tout instant $t \in \mathbf{R}$; etc...

On s'intéresse exclusivement dans ce travail au modèle de la *logique booléenne synchrone*. Celui-ci recouvre la grande majorité des circuits algorithmiques performants conçus aujourd'hui. La suite de cette section définit le cadre de ce modèle, et en énonce certaines propriétés essentielles qui seront exploitées ultérieurement. Pour une formalisation plus avancée de ce modèle ainsi qu'une étude approfondie de ses propriétés arithmétiques, on se réfèrera à [Vui93].

1.1.1 Définition

On note $\mathbf{N}$ l'ensemble des entiers naturels et $\mathbf{B} = \{0, 1\}$ celui des valeurs booléennes muni des opérations usuelles de l'algèbre de Boole: $\neg$ (négation), $\wedge$ (conjonction) et $\vee$ (disjonction).

Définition 1.1 *Un* circuit logique synchrone $\mathcal{C}$ *est défini par la donnée d'un quintuplet* $\langle i, o, s, \mathcal{O}, \mathcal{S}\rangle$ *constitué des éléments suivants:*

- $i \in \mathbf{N}$ *est le nombre de variables d'entrée du circuit;*
- $o \in \mathbf{N}$ *est le nombre de variables de sortie du circuit;*
- $s \in \mathbf{N}$ *est le nombre de variables d'état du circuit;*
- $\mathcal{O} : \mathbf{B}^{i+s} \to \mathbf{B}^o$ *est la fonction de sortie du circuit;*
- $\mathcal{S} : \mathbf{B}^{i+s} \to \mathbf{B}^s$ *est la fonction de transition d'état du circuit.*

On munit un tel circuit d'une sémantique de fonction opérant sur des suites binaires infinies $V = (V^t)_{t\in\mathbf{N}} \in \mathbf{B}^{\mathbf{N}}$, appelées par la suite *variables*, et qui représentent l'évolution dans le temps d'une grandeur électrique. Pour cela, on considère $\mathcal{C}$ comme une fonction opérant sur i variables d'entrée $(I_k = (I_k^t)_{t\in\mathbf{N}})_{0\le k<i}$, retournant o variables de sortie $(O_k = (O_k^t)_{t\in\mathbf{N}})_{0\le k<o}$, et définie de la façon suivante:

Définition 1.2 *Un circuit synchrone* $\mathcal{C} = \langle i, o, s, \mathcal{O}, \mathcal{S}\rangle$ *définit la fonction:*

$$\begin{array}{rcl} \mathcal{C} : (\mathbf{B}^{\mathbf{N}})^i & \to & (\mathbf{B}^{\mathbf{N}})^o \\ (I_0, \ldots, I_{i-1}) & \rightsquigarrow & (O_0, \ldots, O_{o-1}) \end{array}$$

où les variables de sortie $(O_k)_{0\le k<o}$ *sont définies par:*

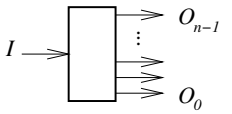
$$\forall t \in \mathbf{N},\quad (O_0^t, \ldots, O_{o-1}^t) = \mathcal{O}(I_0^t, \ldots, I_{i-1}^t, S_0^t, \ldots, S_{s-1}^t)$$

et où les variables d'état $(S_k)_{0 \le k < s}$ *sont elles-même définies par:*

$$(S_0^0, \ldots, S_{s-1}^0) = (0, \ldots, 0)$$
$$\forall t \ge 1,\quad (S_0^t, \ldots, S_{s-1}^t) = \mathcal{S}(I_0^{t-1}, \ldots, I_{i-1}^{t-1}, S_0^{t-1}, \ldots, S_{s-1}^{t-1})$$

Exemple: compteur modulo 2^n

Illustrons la définition ci-dessus par la donnée d'un circuit synchrone réalisant une fonction de *compteur modulo* 2^n, pour $n \in \mathbf{N}^+$ donné.



t	0	1	2	3	4	5	6	7	8	9	$\cdots$
I	0	1	1	0	0	0	1	0	1	0	$\cdots$
$\sum 2^i O_i$	0	0	1	2	2	2	2	3	3	4	$\cdots$

Ce circuit a une variable d'entrée unique I et n variables de sortie $O_0, \ldots, O_{n-1}$. On veut qu'à tout instant $t \in \mathbf{N}$ les sorties soient la représentation binaire du compte (modulo 2^n) du nombre de fois où la variable d'entrée a pris la valeur 1 entre les instants 0 et $t-1$, c'est-à-dire:

$$\forall t \in \mathbf{N},\quad \sum_{i=0}^{n-1} 2^i O_i^t = \sum_{k=0}^{t-1} I^k \pmod{2^n}$$

On réalisera ce circuit en utilisant n variables d'états $S_0 \ldots S_{n-1}$ permettant de mémoriser ce compte en représentation binaire. Pour pouvoir incrémenter ce compte lorsque l'entrée I est active, on définit la fonction Incr_p d'incrémentation modulo 2^p d'un nombre de p bits par un bit, de la façon suivante[2]:

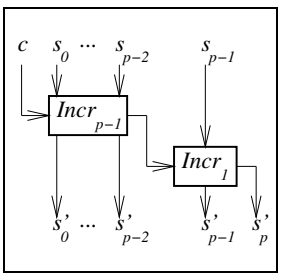
$$\begin{array}{rcl} \mathrm{Incr}_1 : \mathbf{B} \times \mathbf{B} & \to & \mathbf{B}^2 \\ (c, s) & \rightsquigarrow & (c \oplus s, c \wedge s) \end{array}$$

$c \backslash s$	0	1
0	(0, 0)	(1, 0)
1	(1, 0)	(0, 1)

et pour $p \ge 2$:

$$\begin{array}{rcl} \mathrm{Incr}_p : \mathbf{B} \times \mathbf{B}^p & \to & \mathbf{B}^{p+1} \\ (c, s_0, \ldots, s_{p-1}) & \rightsquigarrow & (s'_0, \ldots, s'_{p-1}, s'_p) \end{array}$$

$$\text{où} \begin{cases} (s'_0, \ldots, s'_{p-2}, c') &=& \mathrm{Incr}_{p-1}(c, s_0, \ldots, s_{p-2}) \\ (s'_{p-1}, s'_p) &=& \mathrm{Incr}_1(c', s_{p-1}) \end{cases}$$



[2] Dans toute la suite, le signe $\oplus$ représentera l'opérateur logique "ou exclusif": $a \oplus b = (a \wedge \neg b) \vee (\neg a \wedge b)$

On vérifie aisément par les règles de l'algèbre de Boole (pour $p = 1$) et par récurrence sur p (pour $p \geq 2$) que cette fonction satisfait bien l'équation d'incrémentation:

$$\forall p \in \mathbf{N}^+,\ \forall (c, s_0, \ldots s_{p-1}) \in \mathbf{B} \times \mathbf{B}^p,\ \forall (s'_0, \ldots s'_p) \in \mathbf{B}^{p+1},$$
$$(s'_0, \ldots, s'_p) = \mathrm{Incr}_p(c, s_0, \ldots, s_{p-1}) \qquad ssi \qquad \sum_{k=0}^{p} s'_k 2^k = c + \sum_{k=0}^{p-1} s_k 2^k \pmod{2^p}$$

Le circuit recherché est alors complètement défini par le quintuplet $\langle 1, n, n, \mathcal{O}, \mathcal{S} \rangle$, où les fonctions $\mathcal{O}$ et $\mathcal{S}$ sont définies par:

$$\begin{array}{rcl} \mathcal{O} : \mathbf{B}^{n+1} & \rightarrow & \mathbf{B}^n \\ (I, S_0, \ldots, S_{n-1}) & \rightsquigarrow & (S_0, \ldots, S_{n-1}) \end{array}$$

$$\begin{array}{rcl} \mathcal{S} : \mathbf{B}^{n+1} & \rightarrow & \mathbf{B}^n \\ (S, S_0, \ldots, S_{n-1}) & \rightsquigarrow & (S'_0, \ldots, S'_{n-1}) \end{array}$$
$$\text{où } (c, S'_0, \ldots, S'_{n-1}) = \mathrm{Incr}_n(I, S_0, \ldots, S_{n-1})$$

Expressivité

Au vu de la définition ci-dessus, il est immédiat que tout circuit synchrone $\langle i, o, s, \mathcal{O}, \mathcal{S} \rangle$ est un automate fini à 2^s états, d'alphabet d'entrée $\mathbf{B}^i$ et d'alphabet de sortie $\mathbf{B}^o$. Réciproquement, tout automate fini peut être directement simulé par un circuit logique synchrone, moyennant un encodage binaire de son alphabet d'entrée, de son alphabet de sortie et de son ensemble d'états.

La puissance de calcul des circuits synchrones est donc équivalente à celle des automates finis. En particulier, l'adjonction d'une mémoire virtuellement infinie à un circuit synchrone permet d'obtenir une machine universelle. C'est selon ce principe que sont réalisés les ordinateurs actuels, selon l'architecture dite de Von Neuman.

1.1.2 Réalisation

Cette section décrit la méthodologie la plus couramment utilisée pour la réalisation des circuits logiques synchrones au moyen de circuits électroniques. Elle introduit les notions et les terminologies fondamentales concernant ces derniers, auxquelles on se réfèrera constamment par la suite.

Transistors

Le composant de base fourni par la technologie du silicium est le *transistor*, et on sait aujourd'hui en réaliser un grand nombre (plusieurs millions) sur un seul circuit intégré. Dans la technologie la plus usitée actuellement (CMOS, de l'anglais Complementary Metal Oxyde Semiconductor), ces transistors sont de deux sortes et sont traditionnellement représentés comme sur la figure 1.1.

Figure 1.1: Les deux types de transistor CMOS

En première approximation, le fonctionnement du transistor N (respectivement P) peut être décrit comme suit:

Transistor: Lorsque la tension électrique appliquée entre la borne de commande G et la borne S est supérieure (respectivement inférieure) à un seuil donné, le transistor offre une résistance très faible entre ses bornes S et D; dans le cas contraire il offre une résistance très élevée entre S et D.

En d'autres termes, un transistor se comporte comme un interrupteur placé entre les bornes S et D, dont l'état est commandé par la tension présente sur la borne G.

Portes logiques — Circuits combinatoires

On donnera une sémantique logique aux circuits électroniques en considérant une équipotentielle électrique comme une variable booléenne, vraie si et seulement si son potentiel est supérieur à la moitié de la tension d'alimentation du circuit ("*Vcc*"). Ainsi que le montre la figure 1.2, on sait sous cette hypothèse réaliser les fonctions logiques élémentaires avec des assemblages simples de transistors. Ceux-ci sont traditionnellement appelés *portes logiques*.

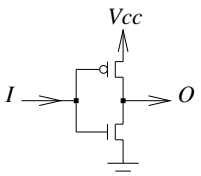
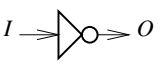
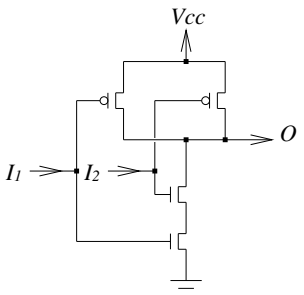
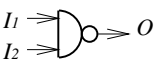
	Réalisation	Représentation
Inverseur $O = \neg I$		
Porte non-et $O = \neg(I_1 \wedge I_2)$		

Figure 1.2: Deux portes logiques simples

La figure 1.3 définit les symboles qui seront utilisés par la suite pour représenter les portes logiques les plus courantes.

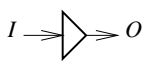
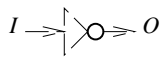
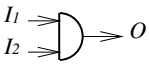
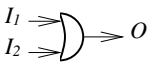
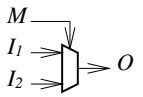
Fonction	Equation	Symbole	Commentaires
Identité	$O = I$		
Négation	$O = \neg I$		"Bulle" sur une entrée ou une sortie de porte
Conjonction	$O = I_1 \wedge I_2$		
Disjonction	$O = I_1 \vee I_2$		
Sélection	$O = (M \wedge I_1) \vee (\neg M \wedge I_2)$		Porte communément appelée *multiplexeur* ou *mux*

Figure 1.3: Symboles conventionnels de représentation des portes logiques

Définition 1.3 (Circuits combinatoires) *On appellera* circuit combinatoire *tout circuit formé d'un* graphe dirigé acyclique de portes logiques. *Sous la modélisation énoncée plus haut, on considérera qu'un tel circuit calcule une fonction de* $\mathbf{B}^n$ *dans* $\mathbf{B}^m$*, où* n *est le nombre de racines du graphe (le nombre de sommets sans prédécesseurs, que l'on appelera les* entrées *du circuit), et* m *le cardinal d'un sous-ensemble distingué des sommets du graphe que l'on appellera les* sorties *du circuit.*

La figure 1.4 montre ainsi une réalisation d'un additionneur binaire complet.

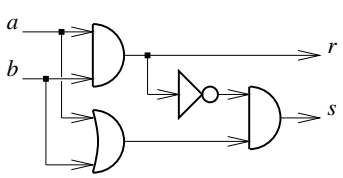
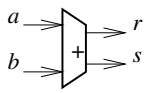
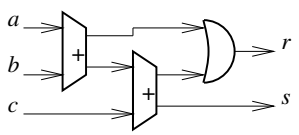
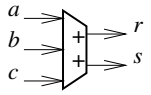
	Réalisation	Représentation
Additionneur binaire $\begin{cases} r = a \wedge b \\ s = a \oplus b \end{cases}$		
Additionneur binaire complet $\begin{cases} r = (a+b+c) \text{ div } 2 \\ s = (a+b+c) \bmod 2 \end{cases}$		

Figure 1.4: Additionneur binaire complet

Rappelons qu'on appelle *base de l'algèbre de Boole* un sous-ensemble minimal des fonctions de $\mathbf{B}^n$ dans $\mathbf{B}$ pouvant engendrer toutes les autres par composition: par exemple deux

telles bases sont $\{0, 1, mux\}$ et $\{1, nand\}$, où $mux(a, b, c) = (a \wedge b) \vee (\neg a \wedge c)$ est la fonction "multiplexeur" et $nand(a, b) = \neg(a \wedge b)$ la fonction "non-et".

Réciproquement, il suffit donc de savoir réaliser un jeu de portes constituant une base de l'algèbre de Boole pour être capable de réaliser toute fonction de $\mathbf{B}^n$ dans $\mathbf{B}^m$ par un circuit logique combinatoire. On énoncera ce résultat comme suit:

Propriété 1.1 (Circuits combinatoires) *Pour toute paire d'entiers $(n, m) \in \mathbf{N}^2$ et toute fonction booléenne $f : \mathbf{B}^n \to \mathbf{B}^m$, il existe un circuit combinatoire calculant f.*

Note: La base $\{0, 1, mux\}$ est très importante en pratique, car elle donne lieu à une décomposition canonique des circuits qui reste très souvent de taille polynomiale, bien que ce problème soit intrinsèquement de complexité exponentielle [Bry86].

Délai combinatoire Il est bien entendu que le temps de commutation d'un transistor d'une part, et le temps de propagation d'un signal électrique le long d'une équipotentielle d'autre part, bien qu'étant très petits, ne sont pas nuls. Le temps nécessaire à l'établissement des valeurs de sortie d'un circuit combinatoire après que les valeurs d'entrée aient été établies sera donc non nul.

On appellera *délai combinatoire* du circuit la borne supérieure de ce temps lorsque l'on fait varier le mot d'entrée du circuit sur les 2^n valeurs possibles. En première approximation, ce délai est proportionnel à la profondeur combinatoire du circuit, c'est-à-dire à la longueur du plus long chemin reliant une entrée à une sortie dans le graphe dirigé acyclique que constitue le circuit.

Registre synchrone

Pour réaliser un circuit synchrone, la dernière fonction manquante est la mémorisation. On utilisera pour cela un élément appelé *registre synchrone* ou *flip-flop*, qu'il n'est pas réellement possible de construire avec des portes logiques mais dont la figure 1.5 schématise le principe approximatif de réalisation.

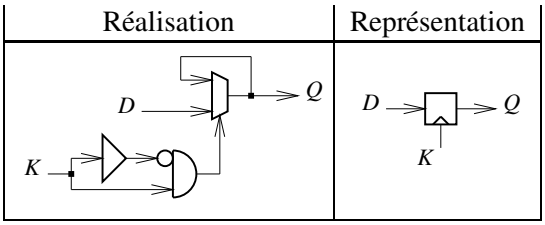


Figure 1.5: Registre synchrone

Ce registre possède une entrée D, une sortie Q, et une entrée auxiliaire dite d'*horloge* K. Son fonctionnement est le suivant (cf. figure 1.6):

- l'état de la sortie Q est normalement constant et égal à la valeur mémorisée, quel que soit l'état de l'entrée D;

- lors d'une transition d'état ascendante de l'entrée d'horloge K (c'est à dire lors de son passage de l'état 0 à l'état 1), la sortie Q passe dans le même état que l'entrée D, et maintient ensuite cet état.

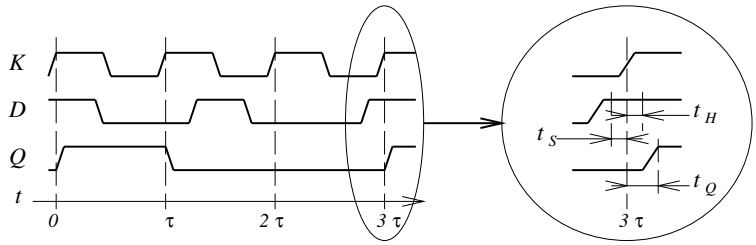


Figure 1.6: Fonctionnement du registre synchrone

Délais La condition de bon fonctionnement de cet élément est que l'entrée D soit stable lors de la transition d'horloge, c'est-à-dire que l'état de D ne change pas durant un intervalle $[t_0 - t_S, t_0 + t_H]$ incluant strictement l'instant t_0 de transition d'horloge. Sous ces hypothèses, le nouvel état de la sortie Q est disponible à partir de l'instant $t_0 + t_Q$ (cf figure 1.6; les valeurs t_S, t_H et t_Q sont des constantes déterminées par l'implémentation exacte du registre en termes de transistors).

Circuits logiques synchrones

Etant donnés les éléments décrits plus haut, on réalisera effectivement un circuit synchrone $\mathcal{C} = \langle i, o, s, \mathcal{O}, \mathcal{S} \rangle$ de la façon suivante (figure 1.7):

- les variables d'entrée et de sortie de $\mathcal{C}$ sont respectivement représentées par i et o équipotentielles $(I_k)_{0 \leq k < i}$ et $(O_k)_{0 \leq k < o}$;
- les variables d'états de $\mathcal{C}$ sont représentées par s registres synchrones. On appellera respectivement $(D_k)_{0 \leq k < s}$, $(Q_k)_{0 \leq k < s}$ et $(K_k)_{0 \leq k < s}$ leurs bornes d'entrée, de sortie et d'horloge;
- la fonction de sortie $\mathcal{O}$ est représentée par un circuit combinatoire; les entrées et les sorties de ce circuit sont respectivement connectées aux équipotentielles $(I_0, \ldots, I_{i-1}, Q_0, \ldots, Q_{s-1})$ et $(O_0, \ldots, O_{o-1})$;
- la fonction de transition d'état $\mathcal{S}$ est représentée par un circuit combinatoire; les entrées et les sorties de ce circuit sont respectivement connectées aux équipotentielles $(I_0, \ldots, I_{i-1}, Q_0, \ldots, Q_{s-1})$ et $(D_0, \ldots, D_{s-1})$;
- les entrées d'horloge des registres $(K_k)_{0 \leq k < s}$ sont toutes connectées à une équipotentielle supplémentaire K, qui sera l'entrée d'horloge du circuit synchrone ainsi construit; on assure, éventuellement à travers l'insertion de délais supplémentaires, que le temps de

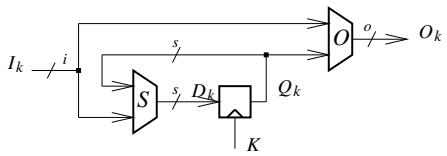


Figure 1.7: Réalisation d'un circuit synchrone

propagation entre K et K_k est constant pour tout $k \in [0, s-1]$, de façon à assurer que les entrées K_k de tous les registres voient des transitions d'horloge aussi *rigoureusement synchrones* que possible.

On utilisera un tel circuit comme suit:

- l'entrée globale d'horloge K est alimentée par un signal périodique de période T;
- les données d'entrées du circuit sont présentées sur les fils $(I_0, \ldots, I_{i-1})$ au début de chaque cycle d'horloge, et maintenues constantes pendant tout le cycle;
- les résultats calculés par le circuit sont relus sur les fils $(O_0, \ldots, O_{o-1})$ à la fin de chaque cycle d'horloge.

La figure 1.8 illustre cette construction pour le cas du compteur modulo 2^n dont les équations sont décrites au §1.1.1.

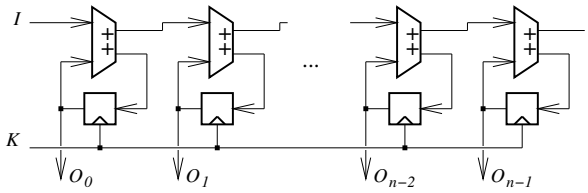


Figure 1.8: Réalisation du compteur modulo 2^n

Délais Soient $t_{\mathcal{O}}$ et $t_{\mathcal{S}}$ les délais combinatoires des circuits implémentant respectivement les fonctions $\mathcal{O}$ et $\mathcal{S}$. Pour que ces circuits aient le temps de calculer leurs résultats avant que ceux-ci ne soient utilisés, il suffit que la période du signal d'horloge délivrée à l'entrée K soit supérieure à la fois à:

- $t_Q + t_{\mathcal{O}}$, le temps maximal d'établissement d'un signal de sortie après une transition d'horloge;
- $t_Q + t_{\mathcal{S}} + t_S$, le temps maximal d'établissement d'un signal à la borne D d'un registre après une transition d'horloge, augmenté du temps minimal d'attente entre cet établissement et la transition d'horloge suivante.

On en déduit la propriété fondamentale de validité du modèle logique synchrone, qui peut s'énoncer comme suit:

Propriété 1.2 *Pour tout circuit logique synchrone $\mathcal{C}$, il existe un graphe $\mathcal{G}$ de portes logiques et de registres, et une valeur $T_{min} \in \mathbf{R}$, $T_{min} > 0$, tels que $\mathcal{G}$ réalise $\mathcal{C}$ pour toute période d'horloge $T \geq T_{min}$.*

Notes

- Réciproquement, il est immédiat qu'un graphe $\mathcal{G}$ de portes logiques et de registres[3] réalise un circuit synchrone si et seulement si il ne possède pas de "boucle combinatoire", c'est-à-dire si le graphe $\mathcal{G}'$ obtenu à partir de $\mathcal{G}$ en supprimant les registres est acyclique.

- On peut maintenant préciser la condition de validité des circuits synchrones énoncée plus haut, selon laquelle les entrées d'horloges des registres doivent voir des transitions "aussi synchrones que possible". Considérons deux registres d'indice i et j reliés par un chemin combinatoire de longueur τ (c'est-à-dire qu'il existe un chemin du graphe reliant Q_i à D_j, ne traversant que des portes logiques, et de délai total τ). On suppose que les entrées d'horloge K_i et K_j sont désynchronisées d'un intervalle de temps δt, choisi positif lorsque K_j est en retard sur K_i. La condition de bon fonctionnement du circuit devient alors:

  $$\delta t + t_H < t_Q + \tau < T + \delta t - t_S$$

  En particulier:

  - si $\delta t < 0$, le circuit ne fonctionnera correctement que si la période d'horloge est en fait supérieure à $t_Q + \tau + t_S - \delta t$, ce qui correspond à une baisse de performance;
  - si $\delta t > t_Q + \tau - t_H$ le circuit *cesse de fonctionner*, et ceci quelle que soit la période d'horloge (le signal D_j arrivant au registre j avant l'horloge apparaît "sauter" celui-ci).

  En pratique on ne fera pas cette analyse individuelle, mais on considérera la borne supérieure δt_{max} des décalages des fronts d'horloge entre deux registres du circuit (en anglais *clock skew*). Les règles de validité énoncées plus haut sont alors raffinées de la façon suivante:

  - δt_{max} doit être inférieur au délai du plus court chemin combinatoire reliant deux registres du circuit;
  - la période minimale de fonctionnement du circuit est $t_Q + t_S + t_S + \delta t_{max}$.

[3] On ne considère bien entendu ici que les graphes *bien formés* pour lesquels chaque équipotentielle comporte exactement 0 (entrée du circuit) ou 1 borne de sortie d'élément logique.

1.1.3 Circuits intégrés traditionnels

Les méthodes traditionnelles de réalisation physique d'un circuit logique synchrone donné — en tout cas dès que celui-ci est d'une taille conséquente — consistent toutes à fabriquer un circuit intégré spécifique, réalisant un graphe de portes et de registres qui implémente ce circuit selon les règles énoncées plus haut. Les principales solutions technologiques disponibles sont les suivantes:

Circuit à façon (en anglais *full custom*) On fournit au fabricant les plans complets du circuit à réaliser, le niveau de détail étant celui du dessin physique des transistors; le fabricant grave ce circuit sur une puce de silicium nue.

On peut ainsi obtenir des densités de plusieurs millions de transistors, avec une flexibilité et une performance maximales, mais la conception d'un tel circuit est extrêmement difficile et coûteuse. Elle demande en particulier une très bonne compréhension du fonctionnement électrique des transistors et du processus physique de fabrication, en sus d'une conception sans faille des équations logiques. Cette approche ne se justifie que pour des circuits qui seront produits en très grandes quantités (microprocesseurs, mémoires, etc. . .).

Réseau prédiffusé (en anglais *gate array*) Le fabricant fabrique en série des circuits possédant une grande quantité de portes logiques élémentaires universelles (typiquement des portes "non-ou"), non encore interconnectées; on fournit la description d'un graphe de ces portes qui implémente le circuit logique désiré; le fabricant grave alors sur le circuit un réseau d'interconnexion permettant de réaliser ce graphe.

On peut ainsi atteindre des densités de plusieurs centaines de milliers de portes. La conception d'un tel circuit est nettement moins difficile que celle d'un circuit à façon, ne demandant qu'une connaissance superficielle des contraintes électriques de fonctionnement. Le processus de fabrication reste néanmoins long et coûteux: ceci en limite l'applicabilité à des circuits fabriqués en quantité, et impose d'avoir des méthodes de conception logique sans faille (afin que le circuit fabriqué soit correct au premier essai; on utilise généralement pour cela de coûteuses simulations, sans pour autant pouvoir jamais garantir complètement la correction complète du résultat).

Le circuit désiré étant réalisé, il faut ensuite l'intégrer dans un système permettant de le faire fonctionner. Un mode usuel d'utilisation d'un tel circuit consiste à le connecter à un ordinateur existant "hôte" en tant que *coprocesseur*: on peut ainsi utiliser la puissance de calcul et les périphériques d'entrée-sortie de l'hôte pour alimenter le circuit en données et utiliser les résultats qu'il produit. Cette connexion sera obtenue au moyen d'une carte électronique d'interface, spécifique au circuit considéré d'une part et au protocole d'interfaçage de la machine hôte d'autre part. La conception et la fabrication de cette carte augmentent encore le temps et le coût de mise en œuvre de l'algorithme considéré.

1.2 Mémoires actives programmables

La section précédente rappelle que les méthodes traditionnelles de réalisation des circuits logiques restent très coûteuses en temps et en argent, à l'opposé de l'implémentation logicielle des algorithmes qui ne coûte que le temps nécessaire à leur conception et à leur mise au point. Or, nombre de problèmes algorithmiques ne peuvent être valablement résolus que par la réalisation d'un circuit, soit qu'ils exigent des performances de calcul excédant les possibilités des réalisations logicielles, soit qu'ils aient des caractéristiques intrinsèquement matérielles (par exemple l'interfaçage physique avec un réseau de communication, ou la saisie de données en temps réel).

Cette section introduit un nouveau mode d'implémentation des algorithmes, intermédiaire entre les réalisations matérielles et logicielles, reposant sur l'utilisation de *circuits logiques programmables*. Ces réalisations auront toutes les caractéristiques des réalisations matérielles, tout en gardant un cycle de développement de nature logicielle.

1.2.1 Définition

Le concept de mémoire active programmable, introduit par Jean Vuillemin en 1987 [Vui87], recouvre un nouveau type d'architecture de coprocesseur. Il peut être brièvement défini comme suit:

Définition 1.4 (Mémoires actives programmables) *On appellera* mémoire active programmable *(ou PAM, de l'anglais Programmable Active Memory) un circuit logique configurable universel, constitué d'un arrangement régulier de cellules logiques élémentaires identiques, attaché à un ordinateur traditionnel "hôte" en tant que coprocesseur.*

Une PAM est un coprocesseur matériel équipant l'ordinateur hôte, permettant comme tout coprocesseur d'accélérer l'exécution d'un algorithme ou de réaliser des fonctions d'entrée-sortie physique. A la différence d'un coprocesseur traditionnel cependant (circuit de calcul arithmétique en virgule flottante, d'encryption de données, d'interface audio-numérique, etc. . .), une PAM est un circuit logique *programmable* d'usage général, qui ne réalise pas une fonction fixe mais peut être configuré à tout moment en un circuit particulier effectuant un ensemble donné d'opérations.

Un ordinateur équipé d'une PAM peut donc configurer celle-ci pour accélérer un algorithme donné, puis la reconfigurer ultérieurement pour effectuer une autre fonction. L'utilisation d'une PAM pour un calcul particulier se fait en deux temps:

1. on *configure* tout d'abord la PAM en y chargeant la description d'un circuit logique qui effectue le calcul désiré;

2. on utilise ensuite cette PAM de la même façon qu'un coprocesseur matériel ordinaire, en y écrivant les données du calcul et en y relisant les résultats.

La programmation d'une PAM, étant un processus purement logiciel, bénéficiera des avantages inhérents au développement logiciel (coût matériel nul, développement et mise au point incrémentaux, etc...). La PAM une fois programmée se comportant comme un circuit matériel par contre, on pourra bénéficier à l'exécution des caractéristiques de ceux-ci (niveau de performance, parallélisme à grain fin, exécution temps-réel, etc...).

1.2.2 Modèle

Cette section décrit complètement une réalisation abstraite du concept défini ci-dessus. Elle en constituera le *modèle fondamental*, dans la mesure où toute autre réalisation de ce concept lui sera équivalente, dans un sens qui sera défini plus loin (§1.2.4).

Bit actif programmable

On appellera *bit actif programmable* (ou PAB, pour l'anglais Programmable Active Bit), le circuit synchrone schématisé par la figure 1.9. Celui-ci possède:

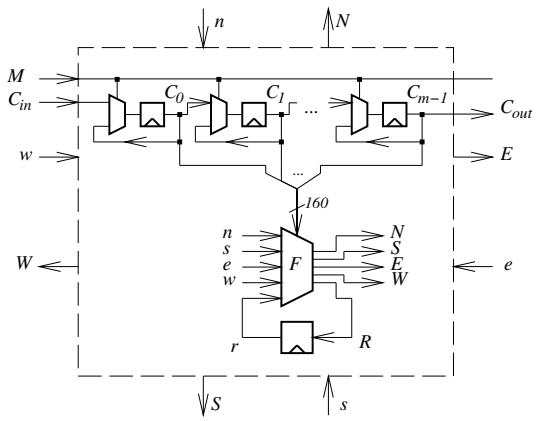


Figure 1.9: Bit actif programmable (PAB)

- 4 entrées de données n, s, e, w;
- 4 sorties de données N, S, E, W;
- un registre dit *bit actif*, d'entrée R et de sortie r;
- $m = 5 \times 2^5 = 160$ registres dits de *configuration* C_i, $i = 0 \ldots m-1$;
- une entrée dite de *mode* M;
- une entrée de configuration C_{in} et une sortie de configuration C_{out}.

On le définira d'abord informellement de la façon suivante:

- La fonction F est la *fonction logique universelle* à cinq entrées et cinq sorties; elle calcule (N, S, E, W, R) à partir de (n, s, e, w, r) en fonction de la table de vérité mémorisée dans le vecteur de bits $C = (C_0, \ldots, C_{m-1})$.
- Lorsque l'entrée de mode est active ($M = 1$), le circuit est en mode *configuration*: pendant les cycles suivants la mémoire de configuration C est chargée sériellement à partir de l'entrée de configuration C_{in}.
- Lorsque l'entrée de mode est inactive ($M = 0$), le circuit est en mode *exécution*. A chaque cycle, la fonction F calcule les bits (N, S, E, W, R) à partir de (n, s, e, w, r) en fonction de la configuration courante C.
- La sortie C_{out} est l'extrémité du registre à décalage de chargement de la configuration C. Elle permet de connecter plusieurs PABs en un seul grand registre à décalage.

Afin de définir précisément ce circuit, on introduira tout d'abord la fonction booléenne f_n définie pour $n \in \mathbf{N}$ par:

$$\begin{array}{rcl} f_n : \mathbf{B}^n \times \mathbf{B}^{2^n} & \rightarrow & \mathbf{B} \\ (i_0, \ldots, i_{n-1}, C_0, \ldots, C_{2^n-1}) & \rightsquigarrow & C_{\sum_{k=0}^{n-1} i_k 2^k} \end{array}$$

Cette fonction sera considérée comme une *fonction booléenne universelle* à n entrées $(i_0, \ldots, i_{n-1})$, dont la table de vérité est donnée par le vecteur $(C_0, \ldots, C_{2^n-1})$. Il est en effet immédiat que l'ensemble des fonctions partielles $f_n^{(C_0,\ldots,C_{2^n-1})}$ obtenues à partir de f_n en fixant les entrées $C_0, \ldots, C_{2^n-1}$ à une valeur particulière, soit:

$$\left\{ f_n^{(C_0,\ldots,C_{2^n-1})} \mid (C_0, \ldots, C_{2^n-1}) \in \mathbf{B}^{2^n} \right\}$$

est égal à l'ensemble $\mathbf{B}^{\mathbf{B}^n}$ de toutes les fonctions booléennes à n entrées.

On peut maintenant définir précisément le bit actif programmable comme le quintuplet $PAB = \langle 6, 5, 161, \mathcal{O}, \mathcal{S} \rangle$, où:

$$\begin{array}{rcl} \mathcal{O} : \mathbf{B}^{m+7} & \rightarrow & \mathbf{B}^5 \\ (n, s, e, w, M, C_{in}, r, C_0, \ldots, C_{m-1}) & \rightsquigarrow & (N, S, E, W, C_{out}) \end{array}$$

$$\text{avec} \left\{ \begin{array}{rcl} N & = & f_5(n, s, e, w, r, C_0, \ldots, C_{31}) \\ S & = & f_5(n, s, e, w, r, C_{32}, \ldots, C_{63}) \\ E & = & f_5(n, s, e, w, r, C_{64}, \ldots, C_{95}) \\ W & = & f_5(n, s, e, w, r, C_{96}, \ldots, C_{127}) \\ C_{out} & = & C_{m-1} \end{array} \right.$$

et:

$$\begin{array}{rcl} \mathcal{S} : \mathbf{B}^{m+7} & \rightarrow & \mathbf{B}^{m+1} \\ (n, s, e, w, M, C_{in}, r, C_0, \ldots, C_{m-1}) & \rightsquigarrow & (R, c_0, \ldots, c_{m-1}) \end{array}$$

$$\text{avec} \begin{cases} R & = f_5(n, s, e, w, r, C_{128}, \ldots, C_{159}) \\ c_0 & = mux(M, C_{in}, C_0) \\ c_1 & = mux(M, C_0, C_1) \\ & \vdots \\ c_{m-1} & = mux(M, C_{m-2}, C_{m-1}) \end{cases}$$

Mémoire active programmable

Le modèle fondamental de mémoire active programmable (PAM) sera défini comme suit:

1. On assemble $N \times M$ bits actifs programmables en une matrice régulière, ainsi qu'il est schématisé sur la figure 1.10:

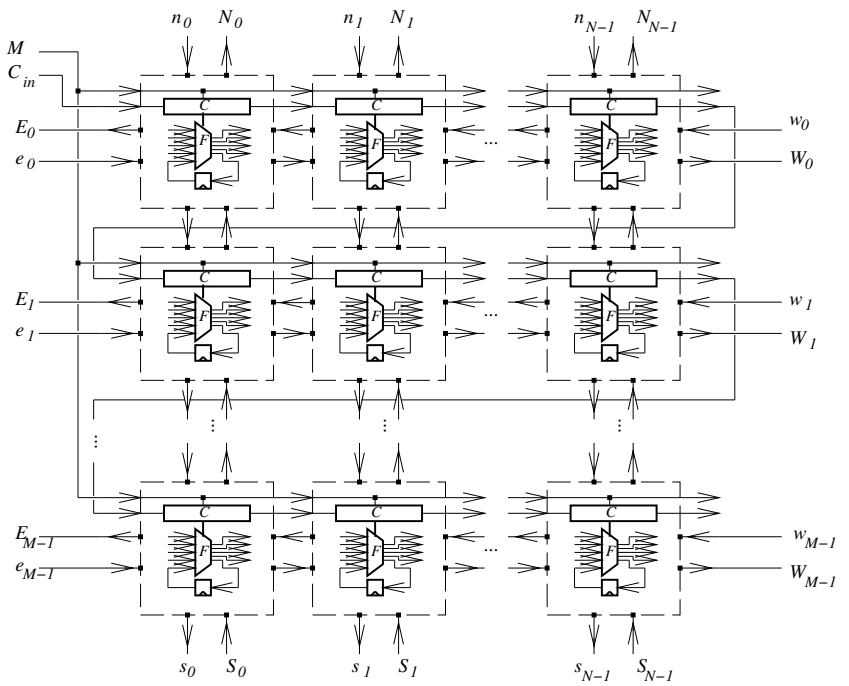


Figure 1.10: Mémoire active programmable: modèle fondamental

- les entrées de données (n, s, e, w) sont connectées par juxtaposition aux sorties de données (N, S, E, W);
- les entrées de mode M sont toutes connectées entre elles pour former l'entrée de mode globale de la PAM;

- les entrées et les sorties de configuration C_{in} et C_{out} sont connectées de proche en proche afin de former un seul registre à décalage permettant de configurer sériellement tous les PABs.

2. De façon à pouvoir configurer la matrice de PABs d'une part, et à pouvoir échanger des données avec celle-ci après configuration d'autre part, on connecte à un ordinateur hôte:

   - les entrées de configuration de la matrice (M et C_{in});
   - les ports d'entrée et sortie de données de PABs restant libres sur les quatre faces de la matrice: $(n_k, s_k, N_k, S_k)_{k=0\ldots N-1}$ et $(e_k, w_k, E_k, W_k)_{k=0\ldots M-1}$.

   Ce mode de connexion ne sera pas détaillé ici. Il suffit que la machine hôte puisse écrire la valeur de chacun des fils d'entrée et lire celle de chacun des fils de sortie (figure 1.11).

3. On ajoute enfin au circuit obtenu un générateur d'horloge à fréquence variable programmable depuis l'hôte. Ce générateur pilote l'entrée d'horloge globale K commune à tous les registres de la matrice de PABs (figure 1.11).

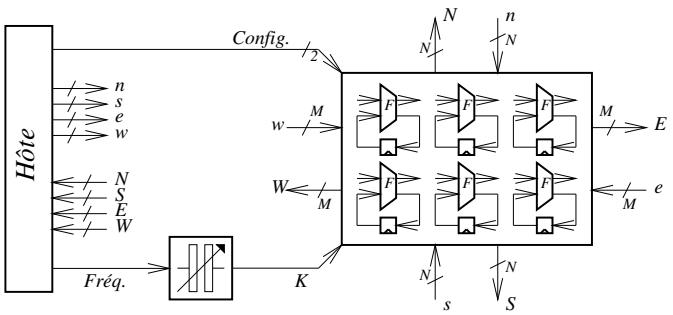


Figure 1.11: Mémoire active programmable: connexion à l'hôte

Le coprocesseur programmable ainsi obtenu ne dépend que de la taille de la matrice de PABs utilisée. On l'appellera par la suite une *PAM[*$N \times M$*]*.

1.2.3 Mise en œuvre

On réalisera un circuit synchrone particulier $\mathcal{C}$ sur cette PAM de la façon suivante:

1. Le circuit $\mathcal{C}$ doit être "compilé" pour une PAM[$N \times M$] donnée. Ceci consiste à identifier une configuration de chaque PAB telle que le circuit synchrone formé par la PAM après application de la configuration soit une réalisation de $\mathcal{C}$. La section 1.2.4 montre qu'il est toujours possible de trouver une telle configuration si la PAM est assez grande.

   La configuration complète de la PAM est obtenue en concaténant dans l'ordre adéquat les vecteurs de bit constituant les configurations individuelles des PABs.

2. On détermine la période minimale T_{min} de fonctionnement du circuit synchrone ainsi obtenu en fonction des délais intrinsèques des PABs (cf. §1.1.2).

3. On configure la PAM en positionnant l'entrée M en mode "configuration" puis en écrivant le vecteur de configuration sur l'entrée C_{in} à raison d'un bit par cycle[4].

4. On ramène l'entrée M en mode "exécution" et on règle le générateur d'horloge sur une période $T \geq T_{min}$.

5. On utilise ensuite le circuit en écrivant les données du calcul sur les ports d'entrée (n_k, s_k, e_k, w_k) et en relisant les résultats sur les ports de sortie (N_k, S_k, E_k, W_k).

Exemple: horloge temps-réel

Illustrons cette mise en œuvre par le traitement d'un exemple complet: la réalisation d'une *horloge temps-réel*. Celle-ci est simplement implémentée par un compteur modulo 2^n, incrémenté de 1 à chaque cycle d'horloge, et relisible à tout moment depuis l'ordinateur hôte (figure 1.12). L'entier n est choisi de façon appropriée en fonction de la période d'horloge utilisée et des durées que l'on veut pouvoir mesurer.

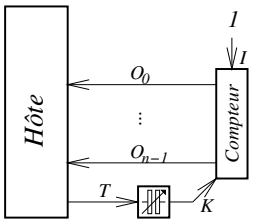
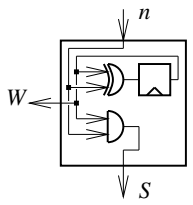


Figure 1.12: Une horloge temps-réel réalisée sur PAM

Cette horloge est implémentée sur une PAM$[1 \times (n+1)]$. Les PABs $(0,1)$ à $(0,n)$[5] sont identiques et implémentent la cellule élémentaire de compteur telle qu'elle est décrite au §1.1.2. Leur configuration est définie par[6]:

$$\forall j = 1 \ldots n, \qquad \mathrm{PAB}(0,j) : \begin{cases} W = r \\ S = n \wedge r \\ R = n \oplus r \end{cases}$$



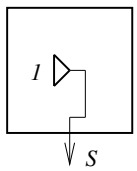
Le PAB $(0,0)$ est utilisé pour générer la constante 1 qui est injectée comme retenue initiale dans le compteur, de façon à ce que celui-ci soit incrémenté à chaque cycle. Sa configuration s'écrit:

[4] En cours de configuration, le circuit peut passer par des états "illégaux" possédant des boucles combinatoires. Ceci peut être ignoré pour ce modèle abstrait, mais devrait bien entendu être pris en compte lors de la réalisation d'une PAM réelle.

[5] On repère les PABs par leurs coordonnées cartésiennes dans la PAM, l'origine étant choisie au coin supérieur gauche

[6] Ne figure ici que la configuration des fonctions logiques universelles qui interviennent effectivement dans le calcul; les autres seront choisies identiquement nulles.

$$\text{PAB}(0,0): \left\{ S = 1 \right.$$



Le plus long chemin combinatoire du circuit ainsi réalisé est aisé à identifier: il s'agit de la chaîne de retenues du compteur, qui joint le bit de poids faible (PAB $(0,1)$) au bit de poids fort (PAB $(0,n)$). Cette chaîne traverse n fonctions combinatoires, donc si on appelle t_f le temps de traversée d'une fonction universelle f_5, il est immédiat que la période minimale de fonctionnement du circuit est:

$$T_{min} = t_Q + n.t_f + t_H$$

Une fois la configuration désirée chargée dans la PAM, on démarre le générateur d'horloge avec une période connue $T \geq T_{min}$. La fonction "horloge" est obtenue en relisant depuis l'hôte le mot binaire présent sur le port $(W_1, \ldots, W_n)$ à l'instant désiré, et en le multipliant par T.

1.2.4 Puissance de calcul

La propriété fondamentale du concept de PAM est sa généralité: tout circuit synchrone peut être réalisé sur une PAM, pourvu que celle-ci soit assez grande et la période d'horloge assez lente. Cette propriété s'énonce précisément comme suit:

Propriété 1.3 *Pour tout circuit synchrone $\mathcal{C}$, il existe deux entiers $(N, M) \in \mathbf{N}^2$ et un réel $T_{min} \in \mathbf{R}^+$ tels qu'il existe une configuration de la PAM[$N \times M$] qui soit une réalisation de $\mathcal{C}$ pour toute période d'horloge $T \geq T_{min}$.*

La preuve de cette propriété n'est pas difficile et découle des résultats énoncés plus haut. On en présente ci-dessous le principe. Considérons un circuit synchrone $\mathcal{C}$:

1. (multiplexeurs et registres) D'après les propriétés 1.1 et 1.2 et en utilisant la base $\{0, 1, mux\}$ de l'algèbre de Boole, il existe un graphe $\mathcal{G}$ de multiplexeurs et de registres qui implémente $\mathcal{C}$ pour une horloge assez lente. [Vui93] décrit un algorithme explicite de construction d'un tel graphe.

2. (placement et routage) On dessine le graphe $\mathcal{G}$ dans un plan infini, de telle façon que les signaux d'entrée et de sortie en soient situés sur la périphérie. Il est alors immédiat que l'on peut construire un graphe $\mathcal{G}'$ isomorphe à $\mathcal{G}$ par assemblage de cellules choisies parmi les configurations canoniques du PAB représentées sur la figure 1.13 ainsi que celles qui en sont dérivées par symétrie et rotation. Soit (N, M) les dimensions du plus petit rectangle à côtés parallèles aux axes contenant $\mathcal{G}'$: la configuration obtenue est une réalisation de $\mathcal{C}$ sur une PAM [$N \times M$] pour une horloge assez lente.

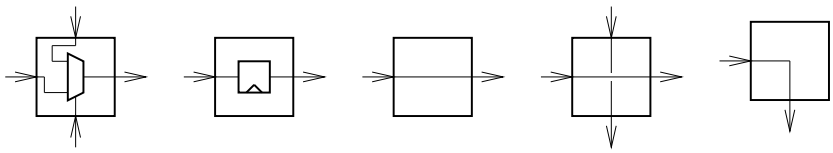


Figure 1.13: Configurations canoniques du PAB

3. (période d'horloge) On détermine la période minimale de fonctionnement du circuit $\mathcal{G}'$ défini ci-dessus en fonction des caractéristiques temporelles t_Q, t_f et t_S du PAB (cf. §1.1.2). Le résultat obtenu est de la forme $T_{min} = t_Q + n.t_f + t_S$, où n est le nombre maximal de cellules séparant deux registres dans le graphe $\mathcal{G}'$.

Equivalence entre PAMs

Le modèle présenté au §1.2.2 n'est qu'une implémentation possible du concept de PAM. Il est bien entendu possible de choisir d'autres structures pour la cellule de base (le PAB) tout en gardant la même propriété de généralité. En accord avec la définition 1.4, on englobera sous la dénomination de PAM tout coprocesseur matériel configurable, formé d'un assemblage régulier de cellules logiques identiques, qui est équivalent à notre modèle fondamental dans le sens suivant:

Définition 1.5 (Equivalence entre PAMs) *On considère deux architectures $\mathcal{A}_1$ et $\mathcal{A}_2$ de coprocesseurs matériels configurables, définies chacune par un arrangement régulier de cellules identiques.*

On dira que l'architecture $\mathcal{A}_1$ est représentable *par l'architecture $\mathcal{A}_2$ s'il existe deux constantes k_S et $k_T \in \mathbf{R}^+$ telles que tout circuit implémenté sur $\mathcal{A}_1$ dans une surface S_1 et pour une période T_1 puisse être implémenté sur $\mathcal{A}_2$ dans une surface $S_2 \leq k_S S_1$ et pour une période $T_2 \leq k_T T_1$.*

On dira que l'architecture $\mathcal{A}_1$ est équivalente *à l'architecture $\mathcal{A}_2$ si $\mathcal{A}_1$ est représentable par $\mathcal{A}_2$ et $\mathcal{A}_2$ est représentable par $\mathcal{A}_1$.*

La section 1.3.1 montre un exemple d'application de cette définition à l'architecture d'un circuit prédiffusé programmable commercial[7].

Mesure de puissance

Un problème récurrent face à la diversité des architectures matérielles permettant de résoudre un problème algorithmique donné, est d'essayer de mesurer la puissance de calcul théorique de chacune sur une échelle commune, afin de pouvoir les comparer.

Parmi les nombreuses unités de mesure utilisées pour les processeurs classiques on trouve traditionnellement le MIPS (million d'instructions par seconde) et le MFLOPS (million

[7] Le même résultat s'applique en fait à *tous* les circuits programmables aujourd'hui existants.

d'opérations en virgule flottante par seconde), selon que le problème manipule plutôt des données entières ou en virgule flottante. Ces unités ne sont pas applicables à la mesure de performance des PAMs – ou des circuits logiques en général –, qui travaillent au grain le plus fin, à savoir celui du bit.

La relation d'équivalence ci-dessus suggère l'existence d'une unité de mesure plus fondamentale, qui s'avère être ici la plus raisonnable. Il s'agit du nombre de bits utiles calculés par seconde, ou pour être plus précis du nombre de fonctions booléennes élémentaires calculées par unité de temps. Ceci n'est bien sûr pas spécifique aux coprocesseurs programmables, et peut être étendu à tout autre modèle de réalisation de circuits utilisant la logique booléenne synchrone (par exemple les circuits à façon ou prédiffusés). On en retire une mesure du volume de logique consacré au problème, que l'on prendra comme *unité fondamentale de mesure de puissance de calcul*.

On choisira comme base, et on mesurera par 1 *bops* (de l'anglais *boolean operation per second*), la puissance d'un circuit calculant une fonction booléenne à quatre entrées au plus, au rythme d'une opération par seconde [BRV93]. Du fait des tailles et des vitesses de circuit disponibles actuellement, on utilisera en pratique les multiples Mbops et Gbops de cette unité (respectivement 10^6 et 10^9 bops).

On dira qu'un circuit synchrone particulier $\mathcal{C}$ opérant à une fréquence de F MHz a une puissance de FS Mbops, si S est le nombre minimal de fonctions universelles f_4 nécessaires à une réalisation de $\mathcal{C}$ n'utilisant que des fonctions f_4 et des registres.

Le choix de la fonction universelle f_4 comme base est arbitraire. Toute autre base de l'algèbre de Boole ne possédant que des fonctions à nombre d'entrées borné serait équivalent, à un choix d'unité près. On choisit celle-ci car elle est représentative d'un grain pratique d'implémentation des fonctions logiques en-dessous duquel les particularités propres à chaque technologie deviennent prédominantes.

Avec cette unité, l'évaluation de la puissance utile théorique d'une PAM est aisée: notre PAB fondamental (hors logique de configuration) correspond à 10 fonctions f_4. Une PAM$[N \times M]$ opérée à la fréquence de F MHz offre donc une puissance maximale de $(10NMF)$ Mbops.

L'évaluation de cette même mesure pour des architectures de processeurs ou de coprocesseurs classiques est plus délicate, demandant l'évaluation de la quantité de logique utile. La figure 1.14 résume ces chiffres pour un certain nombre de machines actuelles, étant bien entendu que les données qu'elles contient ne sont à considérer que comme des estimations d'ordre de grandeur. Pour plus de détails on se réfèrera à [BRV93]. Ceci donne en tout cas une idée de la taille et de la vitesse minimale qu'une PAM doit avoir pour être compétitive face à des machines traditionnelles, si l'usage recherché est l'accroissement de performances (on a mentionné que certains usages comme l'interfaçage peuvent justifier l'utilisation de PAMs indépendamment des questions de performances).

Note On s'appuie implicitement ici sur un résultat sous-jacent important de la théorie des circuits synchrones, selon lequel le produit du nombre de fonctions logiques d'un circuit par sa fréquence de fonctionnement est bien une quantité intrinsèque de l'algorithme considéré (en particulier, indépendamment du nombre des registres). On trouvera dans [Vui93] une discussion détaillée de ce résultat, qui n'est présenté qu'informellement ici.

machine	type	taille (kPABs)	fréquence (MHz)	puissance (Gbops)
Perle-1	PAM	10	25	250
			40	400
MIPS R3000	microprocesseur 32 bits	0,26	25	6,5
			40	10
DEC Alpha 21064	microprocesseur 64 bits (unité entière)	0,56	150	84
			200	110
	(unité flottante)	6,3	150	950
			200	1 300
Thinking Machines CM-1	super-ordinateur massivement parallèle	64	10	640

Figure 1.14: Puissances de calcul de diverses architectures matérielles

1.2.5 Motivation

L'apparition récente d'une nouvelle famille de circuits intégrés, les *prédiffusés programmables* [CD+86] a rendu envisageable la réalisation de PAMs d'une taille et d'un niveau de performances appréciables. On voit immédiatement les partis qu'on peut tirer d'un tel ensemble {Hôte + PAM}:

Coprocesseur universel Lorsque, sur un problème algorithmique donné, le niveau de performances obtenu par une solution logicielle sur un ordinateur traditionnel (ordinateur personnel, station de travail) est insuffisant, il faut soit passer à l'utilisation d'un super-ordinateur – entraînant un coût plusieurs ordres de grandeur supérieur – soit développer un coprocesseur matériel spécialisé – avec un coût et un temps de développement élevés, et la nécessité d'un investissement matériel spécifique. Un ordinateur conventionnel équipé d'une PAM peut sur toute une gamme de problèmes [BRV93] obtenir un niveau de performances comparable, au prix d'un développement purement logiciel. Une PAM unique assortie d'une collection de fichiers de configuration remplace ainsi une collection de coprocesseurs matériels différents.

Prototypage d'algorithme matériel La modification d'un algorithme implémenté sur PAM ne demande qu'une recompilation purement logicielle, dont la durée sera typiquement mesurée en minutes ou en heures (§3.1). A l'inverse la modification d'un algorithme matériel implémenté par des moyens classiques est généralement très coûteuse, et se mesure plutôt en mois, voire en années.

L'ensemble {Hôte + PAM} offre donc un environnement de développement et de mise au point d'algorithmes matériels comparable à ceux dont on dispose pour le logiciel. En particulier il devient possible de *tester incrémentalement* l'algorithme, de l'*exercer à vitesse réelle*, de raffiner expérimentalement le compromis essentiel qu'est la répartition des tâches entre matériel et logiciel.

L'algorithme étant mis au point, il est ensuite éventuellement possible, sans coût de développement supplémentaire, de le réimplémenter sur circuit à façon ou réseau prédiffusé traditionnel, si l'on a un objectif de fort volume et de faible coût unitaire qui le justifie.

Interfaçage, temps-réel Hors considérations de performances, une PAM permet d'aborder certains domaines jusqu'ici exclusivement réservés aux circuits (saisie de données, interfaçage, calcul en temps réel) avec un développement purement logiciel.

Spécialisation du matériel Un paramètre nouveau du développement d'algorithme matériel sur PAM est qu'il devient possible – et même conseillé – de spécialiser à outrance le circuit pour l'application visée, alors que lors du développement d'un circuit classique on cherchera au contraire à le faire aussi général que possible pour amortir les coûts de développement. Un exemple caractéristique d'une telle spécialisation serait un circuit d'encryption de données dans lequel la clé serait *directement compilée* dans la logique du circuit (c'est-à-dire directement incluse dans les équations logiques de celui-ci), économisant ainsi un registre pour la mémoriser, ainsi que le chemin de données, les bornes d'entrée-sortie et la logique permettant de le charger. Le circuit obtenu est plus compact, potentiellement plus efficace et plus facile à développer, et cette approche se justifie dès que le temps de recompilation nécessaire au changement de clé n'est pas pénalisant.

1.2.6 Extensions possibles

On décrit ici quelques extensions intéressantes du modèle de base défini au §1.2.2.

Mémoire locale L'adjonction à la PAM d'une certaine quantité de *mémoire locale*, connectée sur les bords de la matrice de PABs, permet de mémoriser des résultats intermédiaires de calcul, ou de stocker des tables de coefficients, sans consommer de bande passante ni de cycles de calcul sur l'ordinateur hôte. Cette mémoire peut avoir une largeur et un débit adaptés à ceux de la PAM, alors que ceux de l'interface hôte sont typiquement un ordre de grandeur plus faible.

Connecteurs externes L'adjonction sur les bords de la PAM de ports d'interconnexion permet de communiquer directement avec des dispositifs matériels extérieurs, sans passer par l'hôte. La PAM devient alors un dispositif d'interfaçage versatile, offrant une bande passante potentiellement très importante, de la puissance de calcul locale permettant de traiter le flot de données "au vol", et des capacités de traitement temps-réel. Une application typique consiste par exemple à saisir un flux d'images vidéo numériques, à y appliquer un algorithme de compression pour le ramener à un débit supportable par l'hôte, et à le transférer vers ce dernier.

Relecture Au prix d'une modification assez mineure de la cellule élémentaire (PAB), on peut ajouter à la PAM la possibilité de relire à tout moment l'état interne de tous les bits actifs, sous contrôle de l'hôte et indépendamment de la configuration courante (on utilisera typiquement le registre à décalage de configuration). Ceci, combiné à l'extension

suivante (pas-à-pas), fournit un moyen puissant de mise au point des applications, ainsi qu'un moyen d'auto-diagnostic de la PAM elle-même.

Cette même fonctionnalité peut aussi être utilisée pour *relire directement* les résultats de certains algorithmes à très faible bande passante de sortie, permettant comme ci-dessus (§1.2.5, "Spécialisation du matériel") d'omettre complètement la logique et les ressources de communication normalement nécessaire à leur acheminement vers le canal de sortie. Ceci est possible dès que le temps nécessaire à l'opération de relecture n'est pas pénalisant; un exemple typique serait par exemple celui d'un algorithme de comparaisons de séquences génétiques produisant seulement quelques dizaines de bits de résultat (l'index de la meilleure corrélation trouvée) pour plusieurs millions de cycles de calcul.

Mode pas-à-pas On ne tirera réellement parti de l'extension précédente (relecture) qu'en modifiant le générateur d'horloge pour qu'il puisse fonctionner en *pas-à-pas* sous contrôle de l'hôte. On peut ainsi faire fonctionner une application PAM en "mode trace", où l'on effectue une relecture complète de l'état après chaque cycle d'horloge de façon à pouvoir examiner ultérieurement le comportement de chacun des signaux internes (cf. §3.2.3).

Si l'on dispose de plus d'un mode "rafale" (où la source d'horloge génère deux ou plusieurs cycles à vitesse normale puis s'arrête), on pourra même *mesurer effectivement* la fréquence maximale de fonctionnement d'une configuration et identifier exactement son plus long chemin combinatoire. Il suffit pour cela de relire l'état du circuit après chaque rafale et de le comparer à une exécution de référence, et ceci pour des fréquences d'horloge croissantes: le premier bit d'état qui diffère indique l'extrémité du plus long chemin combinatoire utilisé par la séquence particulière de données employée[8].

1.3 Réalisations

Cette section présente les réalisations physiques du concept PAM connues de l'auteur à ce jour. On commence par décrire un exemple concret de circuit prédiffusé programmable – la pièce maîtresse de toutes ces réalisations –, et on montre comment il possède les caractéristiques essentielles à la réalisation d'une PAM. On présente ensuite les deux réalisations effectuées au laboratoire DEC-PRL, qui constituent l'objet principal de ce travail. On passe enfin en revue les travaux apparentés connus. Ceux-ci seront mis en comparaison dans les chapitres ultérieurs.

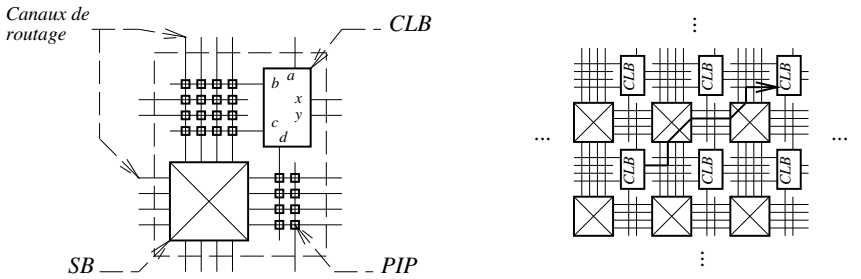
1.3.1 Un exemple de circuit logique programmable

Le premier circuit prédiffusé programmable introduit sur le marché est le XC2064 commercialisé par Xilinx dès 1985 [CD+86]. On décrira d'abord les points importants de son architecture, puis on montrera qu'il est équivalent à notre modèle PAM théorique.

[8] Ce procédé est dû à Mark Shand.

Cellule élémentaire

Le circuit XC2064 est formé d'une matrice 8×8 de cellules programmables élémentaires identiques. Celles-ci sont constituées des éléments suivants (figure 1.15(a)):



(a) Cellule élémentaire

(b) Architecture de la matrice de cellules. En gras: un exemple de routage d'un signal entre deux CLBs.

Figure 1.15: Architecture du circuit Xilinx XC2064

- Un *bloc logique configurable* (CLB), contenant principalement une fonction booléenne universelle à quatre entrées (f_4) et un registre synchrone. Ce bloc est configurable d'un assez grand nombre de façons, dont celle qui est représentée sur la figure 1.16.
- Des *ressources de routage*, comprenant principalement quatre canaux dans chaque direction.
- Des points d'*interconnexion programmable* (PIP) et une *matrice de routage* (SB) permettant de connecter les entrées et les sorties du CLB aux canaux de routage d'une part, les canaux de routage entre eux d'autre part (figure 1.15(b)).

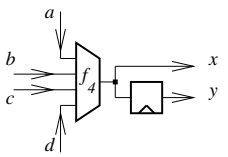


Figure 1.16: Xilinx XC2064: bloc logique configurable

Précisons que l'on n'a décrit ici que les caractéristiques essentielles permettant d'établir le résultat ci-dessous. On se réfèrera à [Xil92] pour une description complète de l'architecture de ce circuit. En particulier, sont présents un réseau global de distribution d'horloge pouvant piloter tous les registres du circuit de façon synchrone, ainsi que des blocs d'entrée-sortie configurables permettant de connecter les bords de la matrice de CLBs au monde extérieur.

Equivalence au modèle

- Il est immédiat que l'on peut réaliser une cellule de XC2064 avec une matrice 5×6 de nos PABs fondamentaux, organisés selon une géométrie strictement identique à celle de la figure 1.15(a).

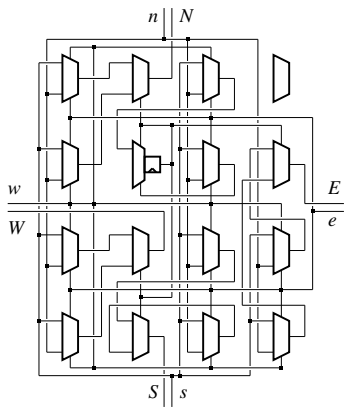


Figure 1.17: Réalisation du PAB fondamental sur XC2064

- La figure 1.17 montre une réalisation du PAB fondamental au moyen d'une matrice 4×4 de cellules de XC2064.

Pour chacune de ces deux constructions, le rapport des fréquences de fonctionnement est borné et aisément déterminable en fonction des caractéristiques temporelles des deux types de cellules. L'architecture du XC2064 est donc bien équivalente à celle de notre modèle PAM fondamental, selon la définition 1.5.

Une cellule de XC2064 calcule une fonction f_4 par période d'horloge. La puissance théorique maximale d'un circuit XC2064 opérant à la fréquence maximale annoncée de $F = 70$ MHz est donc de $64F = 4,48$ Gbops.

1.3.2 Projet PAM, laboratoire DEC-PRL

On a successivement réalisé deux coprocesseurs PAM au Laboratoire de Recherches de Paris de la société Digital Equipment Corporation (DEC-PRL). Ils sont extensivement décrits dans les chapitres suivants.

Perle-0 (1988) Le premier, appelé Perle-0, a été mis au point en 1988 et fabriqué à quelques exemplaires afin de pouvoir servir de support aux premières expériences de programmation et d'utilisation de la technologie PAM. Les principaux résultats obtenus sont décrits dans [BRV89], [Sku90] et [SBV91].

Perle-1 (1991) Le second, appelé DECPeRLe-1 (ou plus brièvement Perle-1), a été mis au point en 1991. Il a été fabriqué à quelques dizaines d'exemplaires, qui sont utilisés à la

fois au laboratoire DEC-PRL et dans plusieurs autres laboratoires français et étrangers, soit en tant que plate-forme de développement d'algorithmes matériels, soit en tant qu'architecture cible pour le développement d'outils d'aide à la conception des circuits logiques. Certains résultats déjà obtenus sont décrits dans [BRV93], [Vui92], [SV93], [Sku92] et [Ber91].

1.3.3 Autres réalisations

Un certain nombre de machines d'architecture similaire ont été réalisées par d'autres équipes. On passe en revue ci-après les plus représentatives d'entre celles qui sont connues de l'auteur à la date de rédaction (1993). Une étude comparée de leur architecture est tentée au chapitre suivant (§2.6). Notons que cette liste est en évolution rapide, le concept de PAM semblant acquérir une popularité croissante (et la généralisation de la technologie des prédiffusés programmables en facilitant la réalisation).

Quickturn RPM (1989) La société Quickturn Systems commercialise depuis 1989 des systèmes d'émulation de circuits intégrés reposant sur un principe similaire à celui des PAMs [Qui91]. Une ou plusieurs cartes électroniques contenant des matrices aussi grandes que possible de circuits prédiffusés programmables sont connectées entre elles et à un ordinateur hôte. Un logiciel de compilation prend en entrée la description d'un circuit intégré en développement (sous la forme d'un graphe de portes logiques élémentaires) et calcule automatiquement une configuration des cartes programmables qui émule ce circuit. Il est ainsi possible de simuler le circuit en développement un ordre de grandeur plus rapidement qu'avec un simulateur logiciel, voire même d'exercer (à vitesse très réduite) le système physique réel auquel le circuit final sera connecté.

L'émulation de circuits intégrés conventionnels par de la logique programmable (à l'opposé de la programmation directe d'algorithmes matériels sur circuits programmables, qui est l'objet du projet PAM à DEC-PRL) introduit une pénalité importante en performance, que l'on peut estimer d'après les données publiées à un facteur d'environ 40 à 50 en surface et 10 à 20 en temps. Les systèmes les plus récents sont ainsi donnés pour pouvoir émuler des circuits d'une taille maximale de 50 000 portes, à une fréquence moyenne de 0.5 à 2 MHz.

Bien que les principes de réalisation d'un tel système soient similaires à ceux d'une PAM au sens premier où elle est définie ici, on verra que l'usage particulier auquel il est destiné introduit des contraintes d'architecture qui conduisent à des réalisations assez différentes (§2.6). On peut donc considérer ce système comme une PAM très spécialisée dans l'émulation de circuits intégrés classiques. L'importance de la pénalité en surface en fait une machine intrinsèquement chère (car demandant de grandes quantités de circuits programmables), ce qui en restreint l'usage aux plus gros projets de développement de circuits (essentiellement les développements de microprocesseurs performants).

Splash (1989) Un système d'inspiration similaire à la PAM mais spécialisé dans l'implémentation des algorithmes systoliques a été construit par D. Lopresti *et al.* au Supercomputer Research Center de Bowie, Maryland (USA) [GHK91]. Ce système, baptisé Splash,

est essentiellement composé d'un assemblage linéaire uniforme de circuits prédiffusés programmables, chacun étant associé à une mémoire locale (§2.6). Un ordinateur hôte peut alimenter une des extrémités de cette chaîne en données d'entrée, et relire les résultats à l'autre extrémité.

Bien que cette architecture possède tous les éléments caractéristiques d'une PAM, sa structure linéaire ainsi que la présence de mémoire uniformément distribuée la spécialisent fortement dans l'implémentation d'algorithmes systoliques [Kun82]. Ce système, qui a été initialement conçu pour la comparaison de séquences d'ADN, a ainsi été utilisé avec succès pour plusieurs autres algorithmes similaires, dans les domaines de la reconnaissance de parole, de la recherche de chaînes de caractères ou de l'analyse génétique.

Splash II (1993) Les concepteurs du système Splash sont en train de terminer la réalisation d'une deuxième génération de ce système, appelée Splash II [ABD92]. Ce système est beaucoup plus complexe et plus général que le premier, et rentre complètement dans le cadre du modèle PAM (§2.6). Il est toujours organisé autour d'un assemblage linéaire uniforme de prédiffusés programmables, possédant chacun un banc de mémoire locale, mais la présence d'un réseau complexe d'interconnexion programmable permet en fait de configurer une importante variété d'architectures.

Ce système n'est pas encore opérationnel, mais plusieurs applications sont déjà envisagées dans les domaines de la recherche et la comparaison de chaînes de caractères (vérificateurs d'orthographe, indexation de textes, comparaisons de séquences génétiques) qui semblent devoir fournir des performances de tout premier plan.

Algotronix CHS 2×4 (1991) La société Algotronix commercialise une carte d'interface pour ordinateur personnel qui est une implémentation complète du modèle PAM: une matrice centrale de circuits prédiffusés programmables est connectée au bus d'entrée-sortie de l'hôte d'une part, à un banc de mémoire locale d'autre part, et peut accepter des données en provenance de dispositifs externes via des connecteurs additionnels [KB92].

Cette carte, de faible coût, possède peu de ressources logiques programmables en comparaison de celles qui ont été citées jusqu'ici. Elle est surtout destinée à des fins de démonstration, d'expérimentation en compilation matérielle ou d'exercices pédagogiques.

ArMen (1991) La machine ArMen réalisée par B. Pottier à l'Université de Bretagne Occidentale constitue une extension du concept de PAM au domaine des machines parallèles [Pot91]. Elle est formée d'un assemblage régulier de nœuds identiques, constitués chacun d'un microprocesseur, d'un banc de mémoire et d'un circuit prédiffusé programmable. Ces nœuds sont connectées en anneau par l'intermédiaire des circuits programmables, et l'ensemble est contrôlé par une station de travail traditionnelle.

Il s'agit donc essentiellement d'une machine multiprocesseur classique dans laquelle l'interconnexion entre processeurs est réalisée par de la logique programmable, de façon à pouvoir réaliser toute une variété d'architectures. En plus de leurs fonctions d'interconnexion, les circuits programmables peuvent par ailleurs être utilisés comme un

ensemble d'accélérateurs matériels configurables locaux à chaque nœud, ou au contraire comme un gros accélérateur matériel distribué sur l'ensemble du réseau. Enfin, un port d'entrée-sortie d'usage général présent sur chaque nœud permet l'interfaçage direct avec des dispositifs matériels externes.

Cette machine fait l'objet d'un programme de recherches actif et a déjà été utilisée avec succès pour l'implantation d'algorithmes systoliques de traitement du signal, ainsi que d'automates cellulaires réalisant des fonctions de traitement d'images.

Chameleon (1992) Beat Heeb and Cuno Pfister, de l'Institut Polytechnique de Zurich, ont réalisé une station de travail expérimentale intégrant complètement le concept de PAM [HP92]. Cette machine comprend en tout et pour tout un microprocesseur, de la mémoire et une matrice de composants logiques programmables. Une petite partie de celle-ci est consacrée aux fonctions traditionnelles d'entrée-sortie (rafraîchissement de l'écran, interfaces sérielles asynchrones pour le clavier et la souris), qui peuvent ainsi être spécialisées et modifiées aisément. Le reste des ressources logiques programmables est disponible pour des fonctions d'accélération d'algorithmes.

Ce projet, qui a été mené à bien en moins d'un an, avait pour but de démontrer la faisabilité du concept. La dernière partie de cette thèse (chapitre 4) tente de discuter les possibilités d'évolution à plus long terme de telles architectures.

Divers Un nombre croissant de périphériques d'ordinateurs intégrant une quantité plus ou moins grande de logique programmable à usage général ont été developpés ou sont en cours de développement. On peut grouper dans cette catégorie la carte d'interfaçage de Perle-1 développée au laboratoire DEC-PRL ("3Mint", cf. §2.3.1) ainsi que les travaux relatés par [BMT92, CSM92, BP91, SGL92, InC90]. Ces architectures possèdent généralement l'essentiel des caractéristiques des PAMs, mais présentent une puissance de calcul insuffisante pour qu'il puisse être réellement intéressant de les utiliser à des fins d'accélération d'algorithmes. Elles trouvent généralement leur meilleur usage dans des applications d'interfaçage en temps-réel avec des dispositifs externes non-standards, ou comme plate-forme de développement d'algorithmes matériels à usage pédagogique.

Chapitre 2

Architecture

> *The second system is the most dangerous a man ever designs... The general tendency is to over-design the second system, using all the ideas and frills that were cautiously sidetracked on the first one.*
>
> F. P. Brooks, Jr., *The Mythical Man-Month* (1975).

Ce chapitre retrace les étapes successives de la conception d'une PAM, en s'appuyant sur l'exemple des cartes Perle-0 et Perle-1 réalisées au laboratoire DEC-PRL, et en mettant l'accent sur la rationalisation des choix d'architecture qui se sont présentés lors de cette conception. On mentionne les variantes ou extensions envisageables, et, lorsque c'est possible, on tente des comparaisons avec les réalisations apparentées connues.

2.1 Blocs de base: circuits programmables

2.1.1 Eventail des possibilités

Le premier choix à faire lorsque l'on décide de fabriquer une PAM est celui de la famille de circuits programmables qui en constituera le cœur. Il n'est pas ici question de se lancer dans l'aventure que représenterait la réalisation de tels circuits; le choix est donc limité à l'offre commerciale à l'instant courant. Cette section passe en revue les différentes familles existantes et tente d'en extraire les caractéristiques principales.

Remarques préliminaires:

- Les "prédiffusés programmables" (en anglais "field-programmable gate arrays") ont été introduits commercialement en 1985 avec le Xilinx XC2064 [CD+86]. Ils représentent un marché qui a crû extrêmement rapidement, et qui est encore en ébullition aujourd'hui. La liste qui suit n'est donc en aucun cas exhaustive, et ne représente qu'une vision instantanée à la date de rédaction.
- Le terme de "prédiffusé programmable" recouvre traditionnellement plusieurs familles de circuits, tous formés d'un arrangement régulier de cellules logiques programmables et assez généraux pour pouvoir en principe implémenter un circuit logique arbitraire. Néanmoins seuls certains de ces circuits mémorisent leur configuration dans des cellules

de mémoire vive et sont par là indéfiniment reprogrammables. Les autres utilisent des technologies à base d'"anti-fusibles" [Act91, MC92] ou de mémoire morte [Xil92c] et ne sont programmables qu'un nombre limités de fois voire une seule fois. Cette caractéristique étant bien entendu rédhibitoire pour l'application qui nous préoccupe, ces derniers ne seront pas considérés ici.

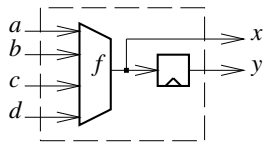


Figure 2.1: Xilinx XC2000: cellule de base (schéma simplifié)

Xilinx XC2000 (1985) [CD+86, Xil92] Premier circuit logique reprogrammable introduit sur le marché, il est organisé autour d'une cellule de base contenant un registre et une fonction booléenne universelle à 4 entrées f (figure 2.1). Cette famille, dont le développement est aujourd'hui arrêté, est disponible dans une taille maximale de 10×10 blocs et une fréquence de fonctionnement maximale de 100 MHz (il s'agit de la fréquence maximale de basculement des registres, ou *toggle rate*: la fréquence maximale d'utilisation sera typiquement inférieure d'un facteur 2).

Ce circuit comme les suivants possède toutes les propriétés fondamentales permettant d'en faire la base de la construction d'une PAM (§1.3.1). Parmi les propriétés additionnelles utiles, il permet la relecture de l'état interne des PABs, à tout moment et de façon transparente.

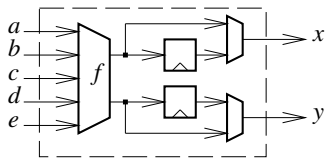


Figure 2.2: Xilinx XC3000: cellule de base (schéma simplifié)

Xilinx XC3000 (1987) [HD+87, Xil92] Cette deuxième génération de circuits développée par Xilinx est la plus usitée aujourd'hui. La cellule de base possède deux registres et une fonction booléenne universelle à cinq entrées f, configurable en deux fonctions indépendantes à quatre entrées (figure 2.2). Elle peut donc être comptée comme deux bits actifs selon l'unité de mesure définie au §1.2.4.

Le circuit est constitué d'une matrice de telles cellules séparées par des canaux de routage (figure 2.3). En simplifiant, chaque intersection des entrées et des sorties de cellules avec les canaux de routage, ou des canaux de routage entre eux, possède un *point d'interconnexion programmable*, c'est-à-dire un transistor de commutation dont la conductivité est déterminée par un bit de configuration du circuit. On peut ainsi router

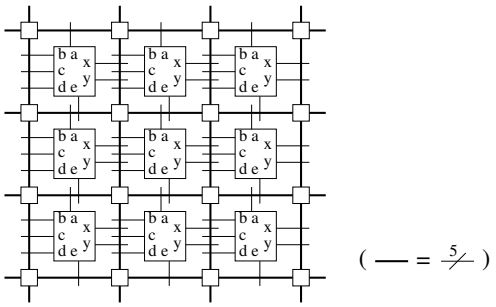


Figure 2.3: Xilinx XC3000: assemblage des cellules

un signal logique de la sortie d'une cellule à l'entrée de n'importe quelle autre cellule en établissant un chemin de proche en proche à travers ces points d'interconnexion.

Le coût (en temps) d'un tel routage est essentiellement proportionnel au nombre de points d'interconnexion programmable traversés, et sera donc au moins proportionnel à la distance de Manhattan entre les extrémités du signal. La plupart des applications possèdent cependant quelques signaux globaux qui sont vus par un grand nombre de cellules (bus de données, contrôles globaux...); pour éviter que ceux-ci ne souffrent de temps de propagation prohibitifs, le circuit possède en plus une certaine quantité de ressources de routage à longue distance ("longues lignes"), non représentées ici, permettant de transporter un signal à coût constant sur toute la largeur du circuit dans une des deux directions horizontale ou verticale.

Enfin cette matrice de blocs logiques et de canaux de routage est entourée d'une rangée de blocs d'entrée-sortie reliés aux pattes physiques du circuit. Ces blocs d'entrée-sortie sont eux-mêmes programmables et peuvent être configurés d'une variété de façons (entrée combinatoire ou synchrone, sortie combinatoire ou synchrone, mode trois-états, inversion de polarité, etc...).

Cette famille est aujourd'hui disponible dans une taille maximale de 16×20 cellules et une fréquence de basculement de 125 MHz. Des circuits plus gros et plus rapides sont annoncés.

Comme le précédent, ce circuit permet la relecture de l'état interne. Il offre de plus la possibilité très intéressante de réaliser des bus internes, par l'existence de points de connexions trois-états sur les longues lignes horizontales.

Xilinx XC4000 (1991) [HC+90, Xil92b] Cette troisième génération de circuits programmables développée par Xilinx est en cours d'introduction. La cellule de base possède deux registres, deux fonctions booléennes à 4 entrées indépendantes (qui ne partagent pas leurs entrées) f et g, et une fonction booléenne supplémentaire à 3 entrées h (figure 2.4). Elle présente de plus plusieurs propriétés nouvelles intéressantes comme la capacité de calculer très rapidement des retenues arithmétiques, ou la possibilité de réécrire dynamiquement les tables de vérité des fonctions f et g, les transformant ainsi en petites *mémoires 16 bits* réparties dans le circuit.

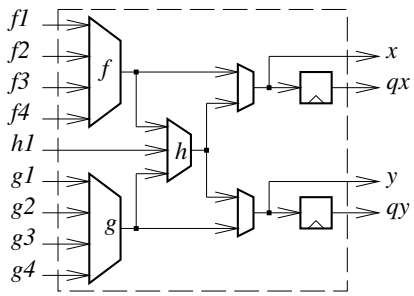


Figure 2.4: Xilinx XC4000: cellule de base (schéma simplifié)

Cette famille est aujourd'hui disponible dans une taille maximale de 20×20 cellules et une fréquence de basculement des registres de 150 MHz. Elle est encore en développement.

Comme le précédent, ce circuit permet la relecture de l'état des registres et possède des bus internes. On peut de plus spécifier l'état initial des registres lors de la configuration, ce qui peut permettre d'interrompre un calcul en cours et de le relancer ultérieurement.

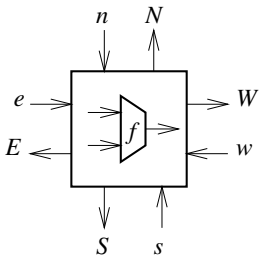


Figure 2.5: Algotronix CAL: PAB (schéma simplifié)

Algotronix CAL (1989) [KG89, Alg90] A l'opposé de l'évolution suivie par les familles précédentes, ce circuit est construit autour d'une cellule de base extrêmement simple (figure 2.5), similaire à celle de notre modèle théorique (§1.2.2). Elle possède quatre entrées et quatre sorties réparties sur les quatre faces, et une fonction universelle à deux entrées f, aussi configurable en *latch* (approximativement la moitié d'un registre synchrone). Des ressources de routage interne à la cellule non représentées ici permettent de connecter chaque sortie N, S, E, W ainsi que chaque entrée de f à l'une quelconque des entrées n, s, e, w ou à la sortie de f.

De la même façon que pour notre modèle, les cellules sont directement connectées par juxtaposition, sans interposition de ressources de routage spécialisées: le routage des signaux entre les bits actifs est donc explicitement réalisé de proche en proche par les cellules logiques elles-même.

Ce circuit est aujourd'hui disponible dans une taille maximale de 32×32 cellules. Une technique originale de gestion des pattes d'entrée-sortie permet d'étendre arbitrairement la taille de cette matrice par juxtaposition de circuits, sans aucune rupture de régularité

de la structure interne. Le temps de traversée d'une cellule est de 1.7 ns pour du routage simple, 6 ns lorsque l'on traverse la fonction f et 20 ns lorsque l'on passe d'un circuit à un circuit adjacent.

Ce circuit permet de plus la reconfiguration partielle dynamique des cellules (avec accès aléatoire), à l'opposé des précédents qui n'offrent que la reconfiguration complète (le programme de configuration étant chargé bit-sériellement). De même, il permet la relecture directe de l'état logique de la sortie de toute cellule, là aussi avec accès aléatoire.

Une telle architecture est intellectuellement attirante par sa simplicité presque ultime; elle souffre néanmoins de plusieurs inconvénients en pratique:

- Pour être compétitive elle demanderait des tailles de matrice plus grandes et des temps de traversée de cellules plus faibles que ce qui est disponible aujourd'hui. Ceci est probablement dû en partie à une différence de niveau technologique avec les circuits précédents (Xilinx), qui restent la référence de ce marché; plus fondamentalement, il semble néanmoins que la granularité de la partie programmable (portes à deux entrées, demi-registres) soit trop faible pour la plupart des applications et que les portes universelles à 4 ou 5 entrées de l'architecture Xilinx constituent un choix mieux adapté en pratique [Woo91]. Contrairement au circuit prédiffusé classique (*gate-array*), la programmabilité des points d'interconnexion d'un prédiffusé programmable introduit en effet un surcoût considérable en surface et en délai, qui doit être compensé par une réduction relative du nombre de ces points et donc un accroissement de la taille de la cellule de base.
- L'interaction fine du placement et du routage dans ce genre d'architecture où les cellules ne communiquent qu'à leurs voisines rend l'implémentation automatique efficace des applications plus difficile que pour les architectures où ces ressources sont différenciées [Bee91].
- L'absence totale de ressources de routage hiérarchiques (c'est-à-dire de ressources de communication à distance ≥ 1) rend difficile voire impraticable l'implémentation efficace de circuits possédant une structure autre que purement systolique; ceci limite considérablement le champ des applications intéressantes.

Plessey ERA (1990) [Ple91] Ce circuit possède une architecture dérivée de celle des circuits prédiffusés dits "mers de portes" (*sea-of-gates*) (figure 2.6). Des portes logiques universelles à deux entrées sont organisées en rangées, séparées par des canaux de routage.

Ce circuit est disponible dans des tailles allant jusqu'à 10000 cellules, et un temps de propagation par cellule de 1.5 ns. Comme le circuit Algotronix, il permet la reconfiguration dynamique sélective des cellules. Il incorpore d'autre part une logique sophistiquée de vérification de l'intégrité des cellules au cours du temps.

L'avantage essentiel de cette architecture est d'être directement calquée sur celle des circuits prédiffusés "mers de portes": elle bénéficie donc directement de l'important investissement logiciel qui a été fait pour ces derniers. Autrement elle ne paraît pas très

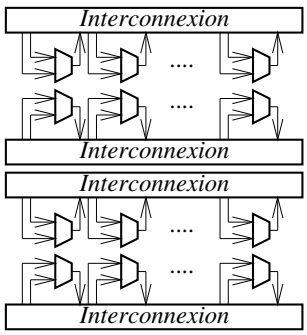


Figure 2.6: Plessey ERA: architecture générale

efficace, souffrant du même problème de granularité que la précédente et ne permettant pas une très bonne utilisation des ressources présentes. Ce circuit ne semble d'ailleurs pas évoluer à un rythme comparable à celui de ses concurrents.

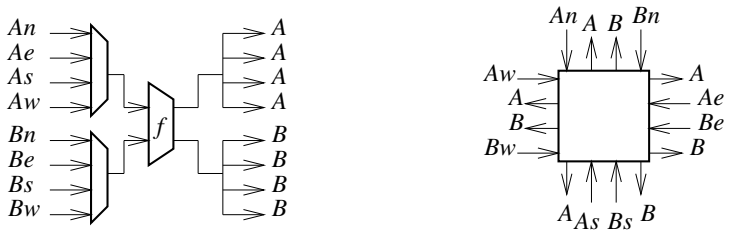


Figure 2.7: Concurrent Logic CFA6006: PAB et schéma d'interconnexion

Concurrent Logic (1991) [CLi92] Ce circuit, le dernier-né du genre, dérive d'un prototype initialement développé par la société Apple [FSJ90]. Sa cellule élémentaire (figure 2.7) possède une fonction logique à deux entrées et deux sorties f, qui peut être configurée soit en fonction identité (pour du routage simple), soit en un additionneur binaire complet, soit en un registre synchrone et un générateur de constante[1]. A l'instar du circuit Algotronix, les cellules sont organisées en matrice et connectées par juxtaposition, mais un réseau hiérarchique de connexions longues est aussi présent.

Ce circuit existe aujourd'hui en matrice 56×56, et est annoncé avec des tailles allant jusqu'à 80×80. Le temps de traversée d'une cellule est d'environ 1.5 ns (routage simple) ou 2.5 ns (fonction logique), pour une fréquence de fonctionnement maximale annoncée de 40 MHz.

Cette architecture est intéressante par sa simplicité, sa régularité et sa symétrie. Elle ne semble pas souffrir des mêmes inconvénients que l'architecture Algotronix, grâce à sa nature clairement synchrone et à son réseau d'interconnexion hiérarchique original. Il est trop tôt pour se prononcer sur son utilisabilité et son efficacité, le circuit étant

[1] Cet ensemble constitue bien une base de l'algèbre de Boole

en cours d'introduction et le logiciel d'utilisation encore balbutiant, mais il semble que cette architecture soit une bonne alternative entre la simplicité du circuit Algotronix et l'efficacité pratique des circuits Xilinx.

Annonces récentes Comme il l'a été dit, le marché des prédiffusés programmables est actuellement en fort développement et voit régulièrement l'introduction de nouvelles familles de produits. A la date de rédaction, les circuits suivants sont en particulier annoncés (mais non encore disponible):

AT&T ORCA [HB+92] Ce circuit possède une architecture du même type que celle des circuits Xilinx XC4000, la différence principale étant que la cellule de base offre une largeur de 4 bits, de façon à pouvoir implémenter plus efficacement des applications possédant un chemin de données régulier assez large.

Altera Flex8000 [Bur92] Ce circuit possède un double niveau de hiérarchie: la cellule de base possède une fonction booléenne arbitraire à quatre entrées et un registre synchrone, ainsi qu'un générateur de retenues arithmétiques; ces cellules sont groupées en blocs de 8 par des ressources de routage locales à délai constant; ces blocs sont eux-mêmes organisés en matrice bidimensionnelle regulière pour former le circuit, et connectés par des ressources de routage globales. Ce circuit est annoncé avec des tailles allant jusqu'à 2 kPABs, pour des fréquences effectives d'utilisation allant de 70 à 150 MHz.

Divers En plus des familles décrites ci-dessus, il existe un nombre assez important de circuits logiques programmables aux architectures plus ou moins *ad hoc* permettant d'implémenter des circuits synchrones relativement arbitraires [WS+89, VA92, Haw91, Jen91, Lat92, Atm91, Xil92c]. Ces circuits sont généralement dérivés de l'architecture PAL (Programmable Array Logic), par adjonction de registres internes ou de macro-cellules spécialisées. Ils sont généralement plus efficaces que les précédents pour l'implémentation de petits circuits, mais ne possèdent pas la structure régulière bidimensionnelle qui est la clé d'une extensibilité quasi-arbitraire. Ils ne seront donc pas considérés ici.

2.1.2 Choix retenus

Perle-0 A l'époque de la conception de Perle-0 (fin 1987), seuls étaient disponibles les circuits de la famille XC2000 et le XC3020 (famille XC3000, matrice 8×8). Le choix naturel était alors le XC3020, qui offrait la plus grande puissance intrinsèque (128 PABs) dans une architecture bien adaptée. Ce circuit était initialement disponible avec une fréquence de basculement des registres de 70 MHz, soit une puissance maximale théorique de 9 Gbops.

Perle-1 La conception de Perle-1 fut commencée en 1989, et étaient alors disponibles les familles XC2000, XC3000, Algotronix et Plessey. Ces deux dernières architectures étant éliminées pour les raisons développées dans la section précédente, le choix fut encore une fois arrêté sur le circuit le plus puissant disponible, à savoir le Xilinx XC3090

(matrice 16 $\times$ 20). Ce circuit était alors disponible avec une fréquence de basculement des registres de 100 MHz, pour une puissance maximale théorique de 64 Gbops.

Choix actuel Dans l'hypothèse où une telle décision serait à reconsidérer aujourd'hui, il est probable que le choix pencherait en faveur du plus gros circuit XC4000 disponible aujourd'hui, à savoir le XC4010 (matrice 20 $\times$ 20), qui offre à la fois le plus gros volume de logique et une architecture qui semble très utilisable. Cette situation est cependant en évolution très rapide, et plusieurs annonces actuelles pourraient – à quelques mois près – la voir reconsidérer, des candidats potentiels étant les suivants:

- des circuits XC3000 plus gros et/ou plus rapides: ceux-ci pourraient reprendre l'avantage en terme de puissance pure, et bénéficient indépendamment d'une base logicielle beaucoup plus mûre;
- des circuits XC4000 plus gros;
- les circuits Concurrent Logic, dès que ceux-ci et leurs logiciels d'utilisation seront réellement disponibles: ils sont plus difficile à comparer dans la mesure où leur architecture est radicalement différente de celle des circuits Xilinx, mais sont certainement prometteurs.

Pour être complets dans la description de ce choix du circuit, il reste deux paramètres à considérer qui ont leur importance et n'ont pas été mentionnés plus haut:

- le grade de vitesse dans lequel le circuit est employé: ceci résulte d'un pur compromis performance-prix; on a à chaque fois retenu les circuits les plus rapides disponibles;
- le boîtier physique dans lequel le circuit est employé: il s'agit là d'un paramètre essentiel, particulièrement par les contraintes technologiques qu'il va induire. Ce sujet ne sera pas développé ici, mais on peut toutefois mentionner les critères importants suivants:
  - il est fondamental d'utiliser un boîtier pour lequel tous les blocs d'entrée-sortie du circuit sont utilisables, de façon à préserver la régularité de la PAM telle qu'elle sera vue par le programmeur;
  - il est tout aussi important d'utiliser un boîtier permettant une densité et une routabilité optimale sur le circuit imprimé, si l'on veut pouvoir fabriquer des matrices suffisamment grandes et rapides.

  Cet idéal n'existait malheureusement pas ni à l'époque de la conception de Perle-0, ni à celle de Perle-1, et il a fallu dans les deux cas rechercher une solution de compromis.

2.2 Matrice principale

2.2.1 Dimensions

Le circuit de base étant choisi, le cœur de la PAM est obtenu en assemblant en matrice rectangulaire, de façon régulière, un certain nombre d'exemplaires de ce circuit. Les dimensions

de cette matrice résulteront d'un compromis dont les données essentielles peuvent se résumer ainsi:

Puissance La surface de la matrice détermine directement la puissance de la PAM obtenue. Les estimations du §1.2.4 montrent qu'il y a là une borne inférieure en-dessous de laquelle la fabrication d'une PAM ne peut plus se justifier du seul point de vue de la puissance de calcul, en comparaison des meilleurs microprocesseurs actuels.

Largeur Les dimensions des côtés de la matrice réalisée (en nombre de PABs) déterminent la largeur des chemins de données qui s'implanteront naturellement dans chaque direction. A moins d'avoir des besoins très particuliers, on choisira donc des dimensions compatibles avec la taille des données de la machine hôte (typiquement une puissance de deux). La symétrie de l'architecture est aussi une question importante, et bien que les circuits Xilinx ne soient pas symétriques intérieurement, nous avons toujours choisi de construire des PAMs symétriques (carrées) de façon à garder la plus grande généralité possible. Un choix opposé sera trouvé dans l'architecture de la machine Splash [GHK91], qui est unidimensionnelle, et du même coup très spécialisée dans l'implémentation d'algorithmes systoliques opérant sur des données de taille fixée.

Coût Les circuits programmables représentant une part importante du coût final de la machine, et le coût du circuit imprimé croissant de façon non linéaire avec la surface, le coût unitaire final est bien évidemment un argument limitant la taille.

Contraintes de fabrication Enfin le dernier argument limitant de ce compromis, et non des moindres, sera donné par un certain nombre de contraintes technologiques: la taille du circuit imprimé est limitée par l'environnement auquel il est connecté (VME, TURBOchannel... cf. §2.3.1), ainsi que par les capacités de réalisation et de montage des fabricants. Il faut aussi prendre en compte la complexité, croissant fortement avec la dimension, du problème de distribution d'une horloge synchrone à tous les circuits: ceci limitera la taille des cartes, et de façon encore plus marquante le nombre de cartes dans une solution multi-cartes, si l'on veut pouvoir conserver des fréquences de fonctionnement élevées.

Il est clair que la fabrication de PAMs de grande taille et rapides est un domaine qui peut tirer immédiatement parti des recherches les plus récentes dans le domaine de l'interconnexion des circuits (modules multi-circuits, hybrides, etc...) [DE+92].

On a ainsi choisi les dimensions suivantes pour les cartes Perle-0 et Perle-1:

Perle-0 La carte Perle-0 est construite autour d'une matrice 5×5 de circuits XC3020, qui correspond à la taille maximale implémentable sans trop de difficultés sur une carte VME (voir §2.3.1). Ceci correspond à une PAM 40×80, ou à 3200 bits actifs. On remarquera que, en contradiction avec les principes énoncés plus haut, la largeur n'est pas une puissance de deux ou un multiple simple de la taille naturelle des mots de la machine hôte. On verra en fait par la suite (§2.4) que Perle-0 est réellement une matrice 32×32 de cellules Xilinx, soit une machine 32 bits, à laquelle on a adjoint une ligne et une colonne supplémentaires destinées à l'interfaçage et au contrôle.

Perle-1 La carte Perle-1 est construite autour d'une matrice 4×4 de circuits XC3090, soit fondamentalement 4 fois plus grosse que Perle-0. Ceci correspond à une PAM 80×128, ou à 10k bits actifs. On verra de la même façon (§2.2.2, commentaire 2) que Perle-1 est réellement une matrice 64×64 de cellules Xilinx, dans laquelle on a réparti de l'espace supplémentaire pour implémenter de la logique de contrôle.

2.2.2 Interconnexions

Principes

Les schémas possibles d'interconnexions internes de la matrice de circuits sont nombreux, et ce point constitue l'un des choix cruciaux qui conditionneront la classe d'algorithmes implémentables sur la machine obtenue. La première contrainte est – autant que faire se peut – de préserver au niveau de la matrice la régularité de structure que l'on observe à l'intérieur d'un circuit, de façon à donner au programmeur l'illusion qu'il travaille sur un seul grand circuit. Ceci est obtenu en installant des "connexions directes" entre les bords adjacents des circuits voisins. La deuxième est d'installer un réseau de communication global entre les circuits, pouvant efficacement accommoder les besoins d'une classe d'algorithmes aussi large que possible. Nous avons pour cela choisi de retenir un schéma d'interconnexions *régulier* et *hiérarchique* à 3 niveaux, décrit ci-après. Rappelons que les données de ce problème sont les blocs d'entrée-sortie présents sur les circuits, soit dans le cas de l'architecture Xilinx deux par rangée et deux par colonne de cellule, une cellule contenant 2 PABs.

Interconnexions de Perle-0 et Perle-1

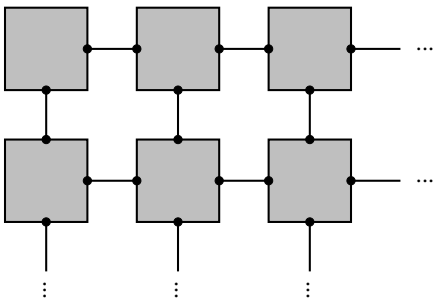


Figure 2.8: Connexions directes

Connexions directes Pour Perle-0, on a choisi de façon un peu arbitraire d'allouer une connexion sur deux aux connexions directes, soit une par rangée et une par colonne de cellules (figure 2.8). Ce choix s'est à l'expérience révélé tout à fait raisonnable, et a été conservé pour Perle-1. Cela veut dire que globalement la quantité de communications entre un PAB et son voisin est de un signal par cellule, ou un signal pour deux PABs (ceci est à comparer aux cinq canaux de routage disponibles entre deux cellules à l'intérieur du circuit, qui doivent aussi accommoder le routage local).

Pour être exact, l'expérience montre que cette valeur, quoique parfaitement utilisable, est une limite inférieure: il y a certaines classes d'opérateurs (par exemple le multiplicateur sériel opérant 4 bits à la fois, qui serait pourtant optimal en technologie Xilinx) pour lesquelles elle est un peu faible, et il est probable qu'en l'absence d'autres contraintes une valeur plus confortable pour la densité de canaux se situerait autour de 1.5 par cellule. Sur Perle-0 en effet, des raisons d'arrondi ont conduit à implémenter en fait une densité de canaux de $9/8$, et ceci a servi plus d'une fois. Dans le cas de Perle-1, il est possible d'augmenter un peu le nombre de canaux si nécessaire en prenant sur la grande quantité de bus, et ceci a aussi été quelquefois utilisé (il faut noter que la plus grande taille des circuits prédiffusés programmables utilisés par Perle-1 rend néanmoins cette limite moins contraignante, dans la mesure où la plupart des opérateurs usuels peuvent être entièrement implémentés à l'intérieur d'un seul circuit).

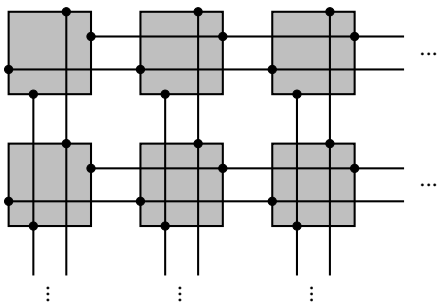


Figure 2.9: Bus

Bus Le deuxième niveau de hiérarchie est constitué par des connexions horizontales ou verticales qui traversent toute la matrice, joignant les faces identiques des circuits. On les appelera par la suite des *bus* (figure 2.9). Ceux-ci constituent le moyen fondamental de communication des données et des résultats, et leur largeur détermine la largeur des chemins de données "naturels" qui seront implémentés. Leur densité est de 0.5 par cellule et par face sur Perle-0 (soit 16 par face), et de 1 par cellule et par face sur Perle-1 (soit 64 par face). L'expérience a montré que le nombre de bus disponibles sur Perle-0 était souvent trop faible et contraignant, alors que sur Perle-1 il est confortable.

Anneaux Le troisième niveau de hiérarchie enfin est constitué de connexions globales, visibles de tous les circuits, que l'on nommera par la suite *anneaux* (figure 2.10). Ces connexions permettent de distribuer des données globales (par exemple le multiplicande d'un multiplicateur sériel, ou le signal de stabilisation de la chaîne de retenues du multiplicateur modulaire décrit en [SBV91]), ou des signaux de contrôle globaux indiquant par exemple un mode de fonctionnement.

Leur nombre est de 3 sur Perle-0, ce qui s'est souvent révélé trop faible, et de 10 sur Perle-1, ce qui est confortable voire large.

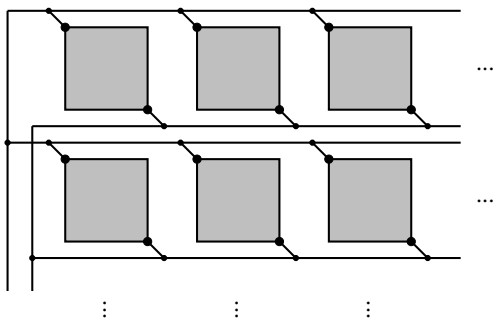


Figure 2.10: Anneaux

Commentaires

- Le compromis entre les différents types de ressources présentés ci-dessus n'est pas toujours facile à faire, mais en présence d'alternatives nous avons toujours choisi de privilégier la *régularité* et la *simplicité*. Ceci permet à la structure obtenue d'être plus intuitive pour le programmeur et plus facile à mémoriser, ce qui est essentiel pour la bonne utilisation ultérieure des ressources cruciales que sont par exemple les bus.

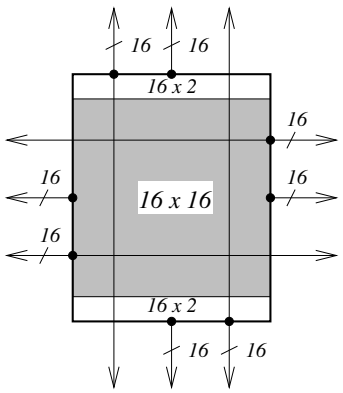


Figure 2.11: Utilisation des XC3090 pour Perle-1

- On remarquera l'utilisation particulière qui est faite des circuits XC3090 dans la matrice de Perle-1 (figure 2.11): ces circuits ont une taille 16×20, mais pour des raisons de régularité, on a choisi de ne les considérer que comme des circuits 16×16, laissant ainsi 2 rangées de "mou" en haut et en bas de chaque circuit.

  Introduite par hasard en raison de la forme curieuse des circuits fournis par Xilinx, cette structure s'est en fait révélée un facteur *essentiel* de la facilité d'utilisation de l'architecture de Perle-1. Cet espace supplémentaire, disponible au niveau de chaque circuit de part et d'autre du chemin de données, fournit en effet l'endroit idéal pour implémenter nombre de structures annexes à ce chemin de données (automates de contrôle délocalisés, décodeurs, registres de *retiming* [LS91], etc...). Ces structures

s'étaient révélées extrêmement difficile à loger dans la structure de Perle-0 sans perdre sur la largeur du chemin de données lui-même (et passer ainsi d'une largeur utile – 32 bits –, à une largeur inférieure bien moins intéressante).

Ce facteur, allié à d'autres (cf. §2.5.4), permet pratiquement aux applications implémentées sur Perle-1 de ne plus avoir de contrôleur central, l'automate de séquencement étant délocalisé et répliqué dans chacun des circuits; on peut ainsi s'abstraire des problèmes que posent généralement le routage des communications entre l'automate et le chemin de données.

Une conséquence de cette remarque est donc que, par raffinement sur le principe énoncé au §2.2.1, la bonne largeur pour les circuits programmables constituant la matrice est "$2^n + \epsilon$". La bonne granularité pour installer ce "mou" est bien en effet celle du circuit, dans la mesure où c'est au niveau inter-circuits que la communication se fait par un nombre restreint de canaux lents, et peut demander de la logique supplémentaire d'encodage et de décodage ou des registres de retiming.

PAM	Connexions	nombre	fréquence (MHz)	bande passante (Mbits/s)
Perle-0	Conn. directes	216	50	11 000
	Bus	64	25	1 600
	Anneaux	3	22	66
Perle-1	Conn. directes	384	60	23 000
	Bus	256	40	10 000
	Anneaux	10	25	250

Figure 2.12: Performance des interconnexions

- Une caractéristique importante de ce schéma d'interconnexion est la bande passante utile disponible pour chaque niveau hiérarchique. Celle-ci dépendra de la fréquence maximale d'utilisation de chaque connexion, elle-même dépendant des caractéristiques électriques des pattes d'entrée-sortie des circuits, du nombre de charges capacitives présentes sur la connexion (*fan-out*) et dans une moindre mesure de la qualité de réalisation physique de la carte.

  La figure 2.12 résume ces données pour les différents types d'interconnexion présents sur Perle-0 et Perle-1.

- Le nombre de niveaux de hiérarchie du schéma d'interconnexion a été ici fixé à trois, mais il pourra (devra?) augmenter si l'on veut réaliser des PAM possédant plus de circuits. A ce sujet, l'extension de loin la plus intéressante de cette architecture serait la conception d'un niveau supplémentaire d'interconnexion multi-cartes, qui offrirait à la PAM l'avantage de la *modularité*, permettant d'étendre *ad infinitum* la surface de logique disponible selon le volume de calcul demandé par l'application.

- Il est possible ici de formuler quelques critiques sur l'architecture Xilinx XC3000, à la lumière des problèmes rencontrés dans cette section:

- la structure interne des pattes d'interconnexion devrait être *parfaitement régulière* vis-à-vis de la matrice de PABs: ceci n'est pas vrai aujourd'hui car d'une part les entrées d'horloge sont prises parmi la structure régulière des blocs d'entrée-sortie, d'autre part les boîtiers physiques disponibles ne permettent pas sauf exception d'utiliser l'intégralité de ces blocs d'entrée-sortie.
- le nombre de canaux d'entrée-sortie disponible est un peu faible pour certaines applications, et surtout il introduit une discontinuité majeure dans la régularité de la matrice principale de calcul: il n'est pas aisé d'améliorer notablement ce facteur en raison des limitations technologiques sur le nombre de pattes des boîtiers, mais des solutions spécialisées à l'interconnexion de circuits voisins sont envisageables, comme le montrent [Alg90] et [DE+92].
- de même, le délai introduit par la communication inter-circuits est un facteur pénalisant qui va conduire à multiplier les étages de *retiming* pour les applications les plus rapides: ce délai est en fait nettement sub-optimal, ayant été délibérément allongé par le fabricant pour fiabiliser les communications synchrones avec le monde extérieur au détriment de la communication inter-circuits [Xil92]. Il semblerait que ce facteur ait été sinon optimisé du moins sensiblement amélioré pour les circuits XC4000.

2.2.3 Distribution d'horloge

Le réseau de distribution d'horloge n'a pas été mentionné dans les ressources d'interconnexion ci-dessus, car il est d'une nature à part lorsque comme ici l'on se place dans le modèle logique synchrone:

- l'horloge est un signal extérieur qui n'a pas a être calculé par la PAM elle-même;
- la distribution aux différents circuits doit être aussi synchrone que possible: tout délai δ entre les fronts d'horloge délivrés à deux circuits se traduira en effet, pour certaines applications, par une augmentation de δ de la période minimale de fonctionnement; si δ devient plus grand que le délai de propagation d'un signal entre les deux circuits, l'application pourra même tout simplement cesser de fonctionner, et ceci quelle que soit la vitesse d'horloge (§1.1.2);
- le circuit de distribution d'horloge véhicule des signaux deux fois plus rapides que tous les autres signaux présents sur la carte; il doit pouvoir supporter une fréquence supérieure ou égale à la fréquence de fonctionnement de l'application synchrone la plus rapide implémentable sur la carte.

Tout ceci impose de réaliser le circuit de distribution d'horloge avec de l'électronique plus rapide que les circuits programmables eux-mêmes. Nous avons ainsi implémenté sur Perle-1 un circuit de distribution d'horloge pouvant véhiculer aux circuits programmables des fréquences allant jusqu'à 100 MHz, avec une dispersion inter-circuits (*skew*) inférieure à 1 ns (mentionnons que cette réalisation s'est en fait avérée hautement non-triviale avec la technologie de l'époque, et serait probablement encore assez délicate aujourd'hui; ceci ne sera pas développé ici).

2.3 Chemins de données externes

La matrice principale de calcul étant réalisée, l'étape suivante consiste à y connecter les chemins d'entrée et de sortie. On fera tout d'abord l'inventaire des différents types de canaux utiles à considérer, puis l'on examinera la manière effective de les connecter sur la PAM.

2.3.1 Canaux d'entrées-sorties

Communication avec l'hôte

Le premier choix à effectuer est celui du canal de communication avec la machine hôte, qui constitue le moyen principal de transfert de données et de commandes entre une application PAM et son programme de pilotage. On sera ici principalement guidé par la disponibilité sur la machine avec laquelle on veut s'interfacer, la bande passante offerte, et la simplicité du protocole d'interfaçage.

Remarquons que beaucoup d'applications PAM – comme l'exemple fondamental qu'est le multiplicateur décrit en [SBV91] – demandent un couplage serré avec l'hôte, qui fait que l'on ne considèrera pas vraiment les possibilités d'interfaçage sur un réseau de communication: même lorsque le débit est comparable, ces solutions souffrent par rapport à la connexion directe sur un bus d'entrées-sorties d'une trop grosse granularité de communication et d'une latence trop importante. On a donc finalement retenu:

Perle-0 une interface VME [Pet89]: bus asynchrone offrant environ 8 Moctets/s de bande passante, raisonnablement simple d'interfaçage et assez répandu. L'hôte choisi est une station de travail Sun-3/110 (puissance: environ 3 Mips).

Perle-1 une interface TURBOchannel [DEC91]: bus synchrone offrant 100 Moctets/s de bande passante et très simple d'interfaçage, c'est l'un des meilleurs bus disponibles actuellement. L'hôte choisi est une station de travail DECstation 5000/200 (puissance: environ 24 Mips).

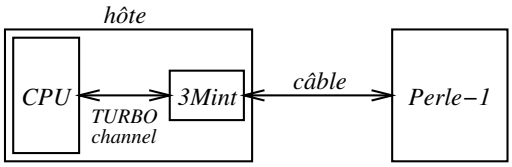


Figure 2.13: Connexion de Perle-1 à sa machine hôte

Pour être précis (figure 2.13), le canal d'entrée-sortie de Perle-1 n'est en réalité pas le TURBOchannel lui-même, mais un canal privé extrêmement simple, similaire au TURBOchannel et de même bande passante. La connexion au TURBOchannel est faite à travers une carte d'interface ("3Mint", elle-même une petite PAM spécialisée) et un câble. Ce choix permet en fait, par reprogrammation ou remplacement de la carte 3Mint,

d'interfacer Perle-1 à tout autre bus d'entrée-sortie ou réseau dont la bande passante est compatible.

Mémoire locale

Il y a une différence d'ordre de grandeur fondamentale entre la bande passante potentielle aux bornes de la PAM (Perle-1: 800 Moctets/s en ne comptant que les bus, et à la fréquence raisonnable de 25 MHz) et la bande passante effective du canal d'entrées-sorties de la machine hôte (Perle-1: 100 Moctets/s maximum, en supposant que l'on peut saturer complètement le TURBOchannel).

Si l'on veut éviter que la très grande majorité des applications ne soient limitées par le débit d'entrées-sorties, il est nécessaire d'ajouter à la PAM une certaine quantité de mémoire locale, de bande passante adaptée, selon une technique comparable à celle de la *mémoire cache* ("antémémoire" en français officiel) utilisée de nos jours par tous les microprocesseurs performants. Selon les applications, cette mémoire pourra être utilisée soit en lecture-écriture pour le stockage de résultats intermédiaires, soit en lecture seule pour le stockage de tables de coefficients.

Cette mémoire doit avoir une largeur compatible avec la largeur naturelle de la PAM, et pouvoir soutenir des fréquences de fonctionnement comparables. On a ainsi choisi d'ajouter respectivement:

Perle-0 deux bancs mémoires de 32 bits de large, soit un horizontal et un vertical;

Perle-1 quatre bancs mémoires de 32 bits de large, soit un par face.

Dans les deux cas ces bancs ont été réalisés avec des circuits mémoires capables de soutenir des temps de cycle comparables à ceux de la PAM elle-même. Ceci correspondait d'ailleurs à peu près aux circuits de mémoire les plus rapides qu'il était économiquement possible de trouver à cette période-là, à savoir justement les circuits destinés à implémenter les mémoires cache des microprocesseurs rapides.

Entrées-sorties directes

Il est enfin extrêmement utile de pouvoir connecter directement des dispositifs externes à la PAM, soit à des fins d'interfaçage temps-réel (par exemple pour s'interconnecter avec un réseau de communication, ou un dispositif audio numérique), soit encore pour obtenir des débits d'entrées-sorties supérieurs à ceux que peut fournir la machine hôte (par exemple pour transmettre des images vidéo).

Il est alors nécessaire d'installer des ports d'entrées-sorties versatiles et rapides sur la PAM elle-même. On a ainsi choisi d'installer sur Perle-1 trois ports d'entrée-sortie compatibles mécaniquement et électriquement avec le standard TURBOchannel (c'est-à-dire que Perle-1 peut piloter directement des cartes d'entrées-sorties TURBOchannel en tant que maître), ainsi que quatre ports 64 bits complètement généraux pour une bande passante maximale, et un port parallèle 8 bits compatible avec le standard audio numérique pour des petits besoins d'interfaçage (figure 2.14).

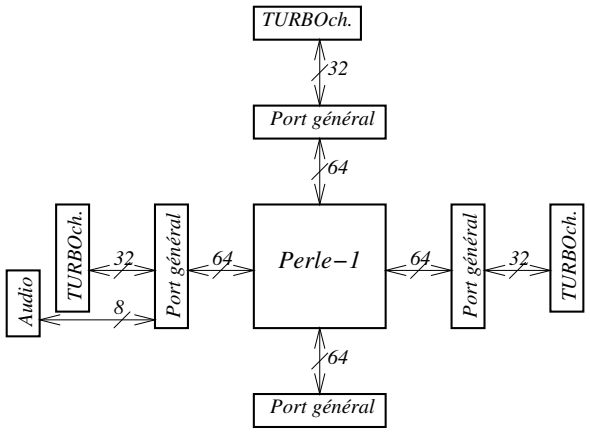


Figure 2.14: Ports externes de Perle-1

Bandes passantes

La figure 2.15 résume les bandes passantes disponibles sur les différents canaux de communication de Perle-0 et Perle-1. On la mettra en comparaison avec la figure 2.12 donnant les chiffres de bande passante interne.

PAM	Canal	largeur	fréquence (MHz)	bande passante (Mbits/s)
Perle-0	Hôte	32	2	64
	Mémoire	64	25	1 600
	Externe	64	20	1 280
Perle-1	Hôte	32	25	800
	Mémoire	128	35	4 500
	Externe	256	25	6 400

Figure 2.15: Performance des entrée-sorties

2.3.2 Interconnexion

L'ensemble des ports d'entrée-sortie étant défini comme ci-dessus, le problème est donc maintenant de les connecter sur la matrice principale de calcul. Rappelons qu'en vertu des schémas d'interconnexion interne définis au §2.2.2, cette dernière peut être attaquée par les ports suivants:

- les connexions directes périphériques, sur les quatre faces;
- les bus, dans les deux directions;
- les anneaux.

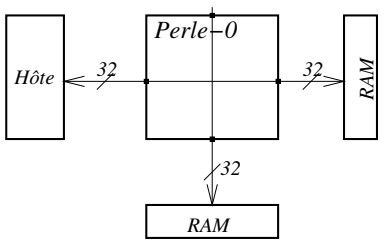


Figure 2.16: Canaux d'entrée-sortie de Perle-0

Perle-0

Ainsi qu'on peut le voir sur la figure 2.16, les canaux d'entrée-sortie de Perle-0 sont connectés directement sur les bus de la matrice. La raison de ce choix est – ainsi qu'on le reverra fréquemment par la suite – qu'en l'absence d'expérience préalable la conception de Perle-0 a été guidée par un souci de simplicité maximale. En l'occurrence cependant, l'utilisation de Perle-0 a montré que ce point était souvent contraignant, et a conduit à adopter une architecture plus complexe pour Perle-1:

granularité cette structure impose, si l'on veut communiquer avec la mémoire ou le bus VME durant un cycle, d'occuper l'intégralité des 32 bits du bus correspondant, bloquant ainsi toute communication à l'intérieur de la PAM (pour être précis, la granularité des entrées-sorties de Perle-0 est en fait de 16 bits, mais l'argument reste le même);

multiplexage il est nécessaire de multiplexer en permanence ces bus entre leurs différents usages (entrées, sorties, accès mémoire, communications internes); ce multiplexage complexifie l'application et son contrôle, et peut souvent pénaliser le temps de cycle. Les bus deviennent alors des goulots de bande passante.

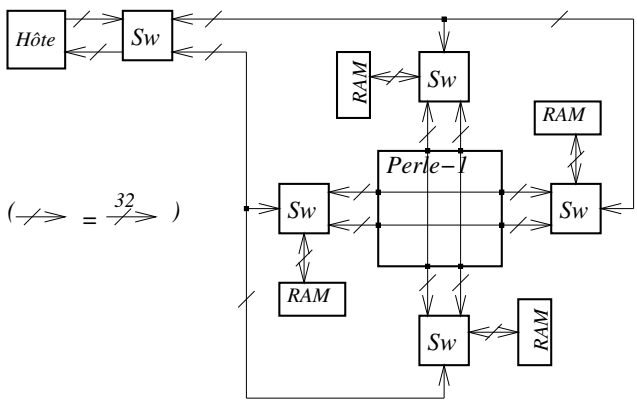


Figure 2.17: Canaux d'entrée-sortie de Perle-1

Perle-1: utilisation de *switches*

L'architecture de Perle-1 a été sur ce point complètement modifiée, comme en témoigne la figure 2.17. L'entrée-sortie dans la matrice se fait toujours par le truchement des bus, mais on a interposé des circuits de multiplexage, que l'on appellera par la suite *switches*, entre la matrice et les sources de données principales (mémoire, hôte).

- Ces switches sont réalisés avec les mêmes circuits programmables que le reste de la PAM: même s'il existe sur le marché des circuits spécialisés plus efficaces que les circuits programmables en multiplexage pur, ces derniers restent beaucoup plus intéressants par leur versatilité:
  - ils permettent d'implanter toute sorte de reformattage et de *retiming*, comme par exemple des conversions série-parallèle, et peuvent donc délivrer les données à la matrice principale de calcul sous le format optimum;
  - ils peuvent implémenter leurs propres automates de contrôle spécialisés, et n'être commandés que par des ordres de haut niveau; les circuits de multiplexage spécialisés, à l'opposé, demandent une quantité phénoménale de signaux de contrôle;
  - ils sont tout à fait efficaces en comparaison des circuits spécialisés du point de vue de la surface occupée sur le circuit imprimé (en conséquence du moindre nombre de pattes de contrôle pour un même nombre de pattes de données);
  - pourvu que l'on implémente un ou deux niveaux de *retiming* à l'intérieur, ils offrent des performances comparables au reste de la PAM et ne pénalisent pas les temps de cycle;
  - la réalisation de la PAM au moyen d'un type unique de circuits programmables – surtout dans la partie vue par le programmeur – ajoute un argument de simplicité et d'uniformité qui, s'il reste secondaire, n'est pas à négliger.

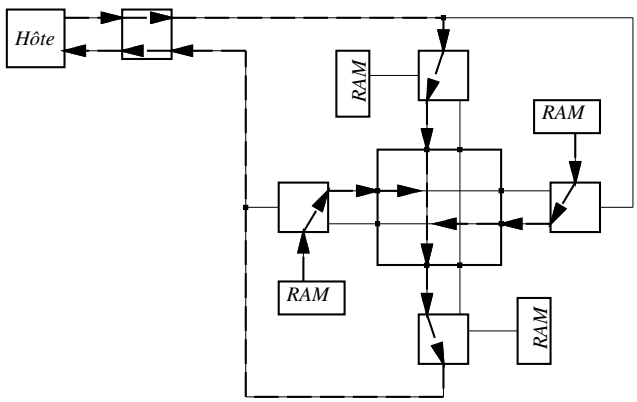


Figure 2.18: Un exemple de chemin de données sur Perle-1

- On a doublé la largeur des chemins d'entrée-sortie par rapport à Perle-0, de façon à ce que ceux-ci puissent être *statiques*. La figure 2.18 montre ainsi un exemple de chemin de données de 32 bits de large qui part du canal d'entrée, traverse la matrice et retourne sur le canal de sortie, pendant que les bancs mémoire Est et Ouest délivrent des coefficients, le tout sans aucun multiplexage dynamique.

Notes

- On pourrait envisager de connecter les différentes sources de données à la matrice principale de calcul par ses connexions directes périphériques plutôt que par ses bus. L'argumentation en faveur de l'une ou l'autre possibilité n'est pas évidente et dépendra de la nature de l'application (selon par exemple qu'il s'agit de données délivrées à un pipeline d'opérateurs, ou de coefficients délivrés à une batterie d'opérateurs fonctionnant en parallèle). On a choisi la solution "bus" à la suite de l'expérience acquise sur Perle-0, qui en a validé l'utilisation pratique. La présence de switches programmables minimise de toute façon la différence entre les deux solutions.

- Les canaux d'entrée-sortie directes décrits par la figure 2.14 ne sont pas inclus dans le schéma présenté ci-dessus: ils ont été connectés directement sur les connexions directes périphériques de Perle-1. La raison de cette simplification est encore une fois qu'en l'absence d'expérience préliminaire on a préféré implémenter la solution la plus économique permettant de mener à bien les premiers essais. Il est néanmoins vraisemblable qu'il y aurait tout avantage à connecter ces canaux à la matrice à travers le même système de switches, ou un deuxième système de switches parallèle au premier.

- Des annonces récentes [Apt92, ICu93] prévoient la prochaine mise sur le marché de circuits d'interconnexion programmables, qui sont essentiellement des matrices de transistors de commutation dont on peut charger la table de connectivité dans une mémoire de configuration. Ces circuits, éventuellement associés à un peu de logique configurable, pourraient probablement prendre avantageusement la place des switches.

2.4 Contrôle

On regroupe dans cette section la génération des adresses pour les bancs mémoire et les contrôles divers (commande des transactions mémoire, protocole de communication avec l'hôte, contrôle des switches, etc...). Tous ces signaux seront typiquement générés par l'automate d'état de l'application programmée dans la PAM.

2.4.1 Organisation mémoire

La largeur de la mémoire a été choisie plus haut, mais il reste à en déterminer la profondeur et la granularité, deux paramètres importants qui conditionneront la classe d'applications implémentables sur la PAM obtenue.

profondeur Ce point résulte presqu'exclusivement d'un compromis performance/coût: doubler la taille de la mémoire n'a qu'un impact faible sur l'architecture de la machine (ajoute un bit d'adresse) et permettra à une classe plus large d'algorithmes d'être efficacement implémentables. En contrepartie les circuits mémoire représentent une fraction importante du coût total de la machine ainsi que de la surface occupée sur le circuit imprimé.

Le bancs mémoire de Perle-0 font 64k mots de profondeur (16 bits d'adresse); ceux de Perle-1 font 256k mots de profondeur (18 bits d'adresse). Dans les deux cas ce choix a résulté d'un compromis performance/coût/contraintes de réalisation, à partir de contraintes minimales sur la classe d'applications que l'on voulait pouvoir implémenter (par exemple un banc mémoire devait être assez grand pour contenir une image vidéo à la résolution standard).

granularité Ce choix est plus délicat: à taille égale, diviser la granularité de la mémoire par deux, par exemple pour la ramener de 16 à 8 bits, double le nombre de fils d'adresses à gérer, mais permet une plus grande flexibilité et une meilleure utilisation de la mémoire par les applications. Par exemple une application ayant besoin d'accéder de façon aléatoire à une table de coefficients de 8 bits risque fort de devoir gaspiller la moitié de la taille et de la bande passante d'une mémoire 16 bits.

La mémoire de Perle-0 est organisée en bancs de 16 bits de large. Celle de Perle-1 est organisée en bancs de 32 bits de large, avec néanmoins un certain support pour des modes 16 et 8 bits dégradés.

2.4.2 Architecture

L'organisation de la mémoire étant choisie, il faut maintenant distribuer les bus d'adresses sur la PAM, et ceci est à nouveau une décision cruciale qui conditionnera la versatilité et la facilité de programmation de la machine obtenue. On décrira successivement les architectures retenues pour les parties "contrôle" de Perle-0 et Perle-1.

Perle-0

Comme il a déjà été mentionné, en l'absence d'expérience préalable, les fils conducteurs de la conception de Perle-0 ont toujours été la simplicité et la régularité maximales. La partie programmable en est en particulier constituée d'une seule matrice régulière de circuits. On a d'autre part mentionné plus haut que les données en provenance de la mémoire ou de l'hôte sont distribuées directement sur les bus de cette matrice (figure 2.19, label d).

Il ne reste alors qu'une façon raisonnable de compléter ce schéma, en connectant les adresses mémoires sur les connexions directes périphériques de la matrice (figure 2.19, label a). On constatera alors que l'on a en fait induit une organisation interne virtuelle à l'intérieur de cette matrice 40×40, où l'on distinguera:

- une matrice de calcul proprement dite D (32×32), dans l'alignement des bus de données d;

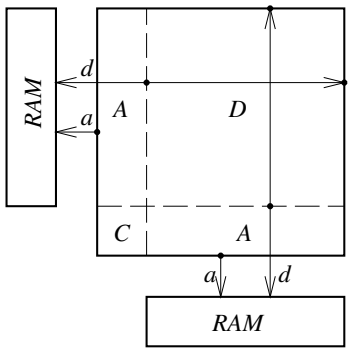


Figure 2.19: Organisation générale de Perle-0

- deux générateurs d'adresse A (8×32 et 32×8) qui contrôlent les fils d'adresse a;
- une partie purement contrôle C (8×8). Celle-ci recevra les autres contrôles (essentiellement ceux du protocole VME), et pilotera d'une part les générateurs d'adresse par l'intermédiaire des 8 bus restant à gauche et en bas (non représentés), d'autre part la matrice de calcul par l'intermédiaire des anneaux.

Cette organisation est virtuelle dans la mesure où elle n'apparaît nulle part dans le schéma de Perle-0, mais sera naturellement adoptée par l'essentiel des applications "standard" du fait de la répartition des connexions externes. Elle pourra néanmoins être remodelée voire complètement bouleversée par une application dont les besoins sont non-standard.

Notes additionnelles:

- Le schéma précédent laisse inutilisées les connexions directes périphériques des faces droite et supérieure. On a choisi de les connecter respectivement sur les faces gauche et inférieure (bus d'adresse a), obtenant ainsi une topologie "torique" parfaitement symétrique. Ceci permettra soit de déporter (ou de dupliquer) un générateur d'adresse sur la face opposée, soit de faire communiquer deux opérateurs qui seraient situés sur des faces opposées de la matrice.
- Des "connecteurs de bus" non représentés ici ont été ajoutés à cette structure de base pour plus de versatilité et de symétrie. Il s'agit de composants externes pouvant, avec une granularité de 16 bits, connecter entre eux les différents bus d'adresses et de données présents sur la carte, et ce dans les deux directions. Ceci permet par exemple à un unique générateur d'adresses de piloter simultanément tous les bancs mémoire, ou encore de "couder" des données (transférer un mot de 32 bit vertical en un mot de 32 bits horizontal), opération qui aurait un coût prohibitif dans la PAM elle-même.

Perle-1

L'expérience acquise sur l'utilisation de Perle-0 a conduit à repenser complètement l'architecture de cette partie de la PAM lors de la conception de Perle-1. Il s'est avéré que, pour

l'essentiel des applications implémentées:

- les générateurs d'adresses et les contrôles proprement dits sont fortement couplés, et font partie d'un même sous-ensemble que l'on appellera "Contrôle" au sens large;
- il y a très rarement communication entre les bus de données et les bus d'adresses.

Ceci correspond naturellement à la classe d'applications que l'on pourrait appeler "primitives récursives", constituées d'un pipeline d'opérateurs agissant sur les données: les mémoires servent de sources ou destinations de données, ou de sources de coefficients, mais il n'y a pas d'interaction des données elles-même sur la séquence d'adresses. Le seul cas (rare) où ceci se produit en pratique est celui de l'application "microcodée" où un mot extrait de la mémoire doit directement être transféré sur le bus d'adresse un certain nombre de cycles plus tard.

On a donc choisi dans l'architecture de Perle-1 de regrouper les génerateurs d'adresses et les contrôles en un circuit supplémentaire, que l'on nommera *contrôleur*, distinct de la matrice principale de calcul, rendant ainsi explicite la spécialisation qui était implicite dans Perle-0.

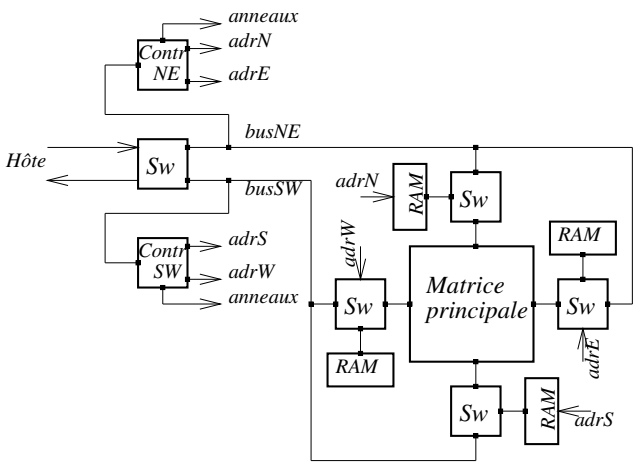


Figure 2.20: Organisation générale de Perle-1

Pour des raisons de nombres de connexions, ce contrôleur est en fait implementé dans deux circuits programmables (identiques à ceux qui implémentent le reste de la PAM, et ci-après nommés *Contrôleur Nord-Est* et *Contrôleur Sud-Ouest*), qui sont l'exacte duplication l'un de l'autre excepté que chacun commande deux des quatre bancs mémoires. On arrive ainsi à l'architecture décrite par la figure 2.20, où ces contrôleurs regroupent chacun:

- deux générateurs d'adresses entrelacés;
- les contrôles des bancs de mémoire correspondants;
- les contrôles des switches correspondants;
- un accès sur le bus d'entrée-sortie correspondant, aligné en face des générateurs d'adresses, de façon à permettre un éventuel transfert données-adresses;

- les anneaux de la matrice;
- les contrôles divers (quasi-inexistants, voir ci-dessous §2.5.3 à §2.5.5);
- le port asynchrone de communication avec l'hôte ("LCBus") discuté plus loin (§2.5.4).

Cette décision est majeure, et potentiellement limitante car elle spécialise assez fortement l'architecture en faveur d'une classe d'application donnée. L'expérience acquise après maintenant un an d'utilisation de cette architecture est néanmoins nettement positive, celle-ci s'étant à chaque fois très bien prêtée à l'implémentation des applications envisagées [BRV93].

2.5 Support

Cette section décrit la dernière part de la conception d'une PAM, à savoir les circuits de support et d'interfaçage avec l'hôte. Ceux-ci constituent la partie non programmable de la carte, c'est-à-dire celle qui n'est pas vue directement par le programmeur PAM mais lui permet d'utiliser la partie programmable le plus optimalement et le plus aisément possible.

De la même façon que précédemment, ces circuits étaient pour Perle-0 réduits au strict minimum, dans un but de simplicité. On constatera que l'expérience acquise sur cette première carte a conduit à les développer considérablement lors de la conception de Perle-1, de façon à améliorer selon les cas: la performance, la fiabilité, la facilité de programmation.

Dans la mesure du possible, ces circuits ont été réalisés au moyen des mêmes circuits programmables que ceux qui constituent le cœur de la PAM (soit pour Perle-0: deux XC3020, pour Perle-1: un XC3090), ceci afin de bénéficier d'une plus grande souplesse dans l'implémentation, et de réduire le nombre de composants différents présents. Ainsi les seules fonctions réalisées en composants discrets sont-elles celles qui nécessitent un niveau de performances supérieur aux capacités des circuits programmables (par exemple le générateur d'horloge).

Les circuits programmables de support de Perle-0 étaient eux-mêmes configurés automatiquement à la mise sous tension à partir d'une mémoire morte (ROM) résidente sur la carte. La complexification de ces circuits pour Perle-1 a rendu préférable l'implémentation en logique discrète de la circuiterie permettant de configurer (et même de relire, cf. §2.5.2) le circuit d'interface depuis l'hôte. On peut ainsi développer et télécharger des nouvelles versions, voire des versions spécialisées pour telle ou telle application, sans manipulation matérielle.

2.5.1 Configuration

La première fonction que doit remplir l'interface est la configuration des circuits programmables de la PAM. Il y peu de commentaires à faire sur ce point, sinon que la *vitesse de configuration* va être un critère déterminant lors de l'utilisation future de la carte: plus celle-ci est rapide, plus il devient possible de découper une application complexe en plusieurs applications simples, qui se succèdent dans le temps et communiquent par l'intermédiaire des mémoires. L'exemple de base d'une telle pratique est celui de l'application qui implémente un premier circuit pour initialiser la RAM depuis l'hôte, un deuxième pour effectuer le calcul voulu sur les données en

RAM, puis un troisième pour relire les résultats dans la RAM. Il est plus simple dans ce cas de développer trois circuits séparés, plutôt qu'un seul qui devrait multiplexer les trois fonctions, et les sous-circuits obtenus sont plus facilement réutilisés et/ou optimisés séparément (dans ce cas précis un circuit optimisé de chargement et de relecture de la mémoire fait bien évidemment partie de la bibliothèque standard).

Cette approche devient pratique dès que le temps de configuration devient non déterminant devant le temps de calcul lui-même, et si l'on a pris soin d'implémenter la logique de configuration de façon à assurer qu'elle préserve de façon fiable le contenu de la mémoire. On a ainsi obtenu pour Perle-1 un temps de configuration inférieur à 20 ms, égal à la limite inférieure tolérée par les circuits Xilinx eux-mêmes.

Notes

La vitesse de configuration est un des points (peu nombreux) sur lesquels les circuits Xilinx, conçus pour un autre usage, sont assez nettement sub-optimaux par rapport à un usage PAM: il serait probablement assez facile de gagner un ordre de grandeur sur ce point.

Une autre caractéristique qu'il eût été intéressant que les circuits programmables utilisés offrissent est le choix à la configuration de l'*état initial* des registres (flip-flops) de l'application. On pourrait ainsi recharger une application en cours de fonctionnement et utiliser une carte PAM en *temps partagé*, ou simplement intervenir sur la valeur des registres internes en tant qu'outil de mise au point. Cette caractéristique est maintenant partiellement disponible sur les circuits Xilinx XC4000 [Xil92b].

2.5.2 Relecture

Une propriété des circuits Xilinx, originellement secondaire mais qui s'avère ici fondamentale, est qu'il est possible à tout moment et de façon transparente pour l'application de relire la configuration et l'état interne de tous les registres. Nous avons implémenté au niveau de la PAM le support pour cette fonction, que l'on appellera *readback*, avec les avantages suivants (cf. §1.2.6):

mise au point et diagnostic cette caractéristique, alliée à une horloge pouvant fonctionner en pas-à-pas (§2.5.3), fournit un outil de mise au point et de diagnostic irremplaçable. La PAM peut ainsi être utilisée comme son propre simulateur, de façon à rechercher soit des erreurs de conception de l'application, soit des problèmes matériels sur la carte elle-même (court-circuit, piste de circuit imprimé coupée, mauvaise soudure, etc...).

relecture de résultats lorsqu'une application ne produit que peu de résultats (par exemple l'application de comparaison de séquences génétiques décrite en [BRV93], qui compare un mot à tout un dictionnaire pour en trouver la meilleure approximation), il peut devenir plus intéressant de relire directement ces résultats par readback que d'implémenter un chemin de données qui les ressort via l'interface hôte. Ceci devient pratique dès que le temps de readback n'est pas pénalisant par rapport au temps de calcul. L'application et sa mise au point s'en simplifient d'autant.

Ceci montre qu'à l'instar de la configuration il est essentiel d'obtenir une relecture aussi rapide que possible. Une fois encore nous avons implémenté sur Perle-1 la logique permettant d'effectuer cette relecture aussi vite que les circuits Xilinx le permettent, soit en moins de 10 ms. A noter que là encore les circuits Xilinx sont très sub-optimaux, puisque ceci correspond à une bande passante utile inférieure à 80 kbits/s. Il serait vraisemblablement aisé de gagner au moins deux ordres de grandeur sur ce temps si l'on voulait redéfinir un circuit programmable spécifiquement pour l'utilisation en PAM.

Note

Selon la même approche, il serait très intéressant d'implémenter dans les circuits de support d'une PAM la logique et les chemins de données permettant la relecture – voire même l'écriture – des bancs mémoires depuis l'hôte, de façon transparente pour l'application: on en tirerait les mêmes avantages de facilité de mise au point, et cela permettrait de simplifier notablement bon nombre d'applications. Ceci a été étudié en détail lors de la conception de Perle-1, mais n'a pas été jugé faisable à un coût acceptable. Il semble néanmoins que ce serait aujourd'hui possible en utilisant certaines facilités nouvelles de mise au point qui ont été introduites dans la dernière génération de circuits programmables Xilinx [Xil92b].

La méthode alternative, permettant quand même de faire de la mise au point lorsque comme sur Perle-1 cette propriété n'est pas présente, consiste bien sûr à arrêter l'application sur le cycle désiré (cf. §2.5.3), et à télécharger l'application standard d'accès mémoire. On peut ainsi vérifier à tout moment l'état des mémoires, mais cette vérification est destructive car on ne peut plus ensuite relancer l'application initiale à partir du cycle où on l'a arrêtée.

2.5.3 Génération d'horloge

On a vu qu'un générateur d'horloge à vitesse variable constitue l'une des parties incontournables du modèle PAM de base. Celui-ci doit pouvoir générer une horloge synchrone, dans une plage de fréquences déterminée par la plus courte et la plus longue période d'horloge à laquelle il est envisageable de faire fonctionner une application. Il est aussi souhaitable que celui-ci puisse fonctionner en mode pas-à-pas, pour la mise au point ou des applications particulières.

Encore une fois, la complexité de l'implémentation Perle-0 a été réduite au strict minimum (figure 2.21): on fournit sur les bords de la matrice utilisateur trois sources d'horloge (deux oscillateurs fixes à 10 et 40 MHz, et une entrée externe), ainsi que deux bits contrôlés par logiciel par l'intermédiaire du circuit d'interface pour implémenter le pas-à-pas. On trouve aussi dans la même région les entrées de deux amplificateurs d'horloge, qui pilotent deux réseaux de distribution desservant les entrées horloge des circuits Xilinx (les circuits Xilinx possèdent deux réseaux d'horloge indépendants). Chaque application doit ainsi implémenter dans sa propre logique les diviseurs et multiplexeurs nécessaires au choix de la fréquence d'horloge désirée, ainsi que le mode pas-à-pas quand nécessaire.

Perle-1 est sur ce point à la fois nettement plus complexe et plus spécialisée, les limitations suivantes étant apparues lors de l'utilisation de Perle-0:

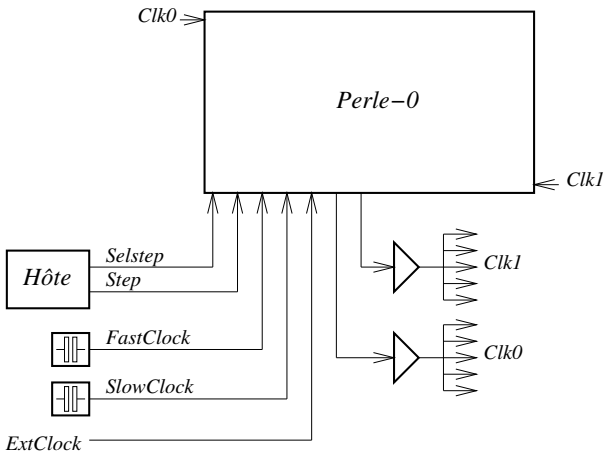


Figure 2.21: Perle-0: circuit de génération d'horloge

- la manipulation des horloges par les circuits programmables eux-mêmes est un art difficile truffé d'embûches (on sort du modèle synchrone pour tomber dans un monde où les délais de routage et l'implémentation fine des fonctions logiques sont cruciaux); ceci n'est que partiellement soluble par la fourniture de modules pré-optimisés; de plus la manipulation des horloges demande intrinsèquement de l'électronique plus rapide que l'utilisation de ces horloges: ceci introduit une limitation en performance.

- la sélection de la vitesse d'horloge et/ou du mode pas-à-pas ne doit pas demander de recompilation de l'application, selon le principe général qu'il est malsain et malcommode d'avoir à modifier l'objet en cours de développement pour pouvoir le mettre au point ou en mesurer les performances.

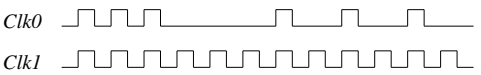


Figure 2.22: Utilisation synchrone de deux horloges

- une PAM n'est réellement faite que pour implémenter des circuits qui répondent parfaitement au modèle logique synchrone. A l'intérieur de ce modèle, la seule utilisation légale de deux horloges distinctes est celle où les deux horloges sont synchrones, l'une des deux pouvant être un sous-multiple de l'autre ou s'arrêter (figure 2.22).

On a ainsi implémenté sur Perle-1 le circuit d'horloge décrit sur la figure 2.23, avec les caractéristiques suivantes:

Fréquence L'horloge est délivrée par un générateur à fréquence variable sélectionnable depuis l'hôte; la fréquence maximale est de 100 MHz, supérieure à la spécification des circuits Xilinx; la fréquence minimale est de 300 kHz (il est possible de simuler des fréquences

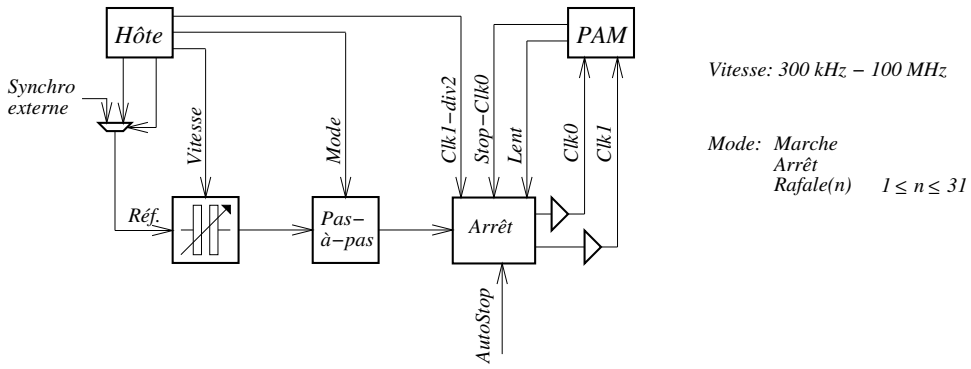


Figure 2.23: Perle-1: circuit de génération d'horloge

inférieures par pas-à-pas si nécessaire); le pas élémentaire de fréquences est d'environ 0.01%, inférieur à la variabilité en température des délais internes des circuits Xilinx; cette horloge est optionnellement synchronisable sur une référence extérieure, pour le cas où la PAM serait directement interfacée avec un dispositif externe qui délivre des données à fréquence imposée (par exemple une source audio numérique); le temps de stabilisation de la fréquence d'horloge après un ordre de changement est inférieur au temps de chargement d'une application dans la PAM.

Pas-à-pas Sous contrôle purement logiciel, l'horloge peut opérer en mode pas-à-pas ou en mode "rafale", pour des rafales de 1 à 31 pas. Ceci permet une mesure fine des chemins critiques des applications (voir §3.2.3).

Double horloge Le réseau de distribution d'horloge délivre deux horloges aux circuits programmables, ci-après nommées `clk0` et `clk1`. Celles-ci sont strictement synchrones, sous les restrictions suivantes:

- `clk0` peut s'arrêter au cycle près sous contrôle de l'application (voir aussi §2.5.4, "contrôle de flux"). Ceci est très utile pour "geler" un chemin de données en attente de données, et beaucoup plus économique que de distribuer à tous ses registres un signal de validation (*clock enable*).
- `clk1` est optionnellement deux fois plus lente que `clk0`. Ceci permet d'implémenter des applications dont le cycle interne est inférieur à la période minimale de fonctionnement des dispositifs d'entrée-sortie (bancs mémoire et interface hôte), en faisant fonctionner le chemin de données sur `clk0` et les entrées-sorties sur `clk1`.

Mode lent Sous contrôle de l'application embarquée dans la PAM, les horloges peuvent ralentir dynamiquement leur fréquence (en fait la diviser par 4), tout en restant synchrone. L'intérêt de cette fonction est de permettre aux applications dont la fréquence de fonctionnement pourrait être très élevée, mais qui doivent néanmoins effectuer une opération lente de temps à autre (par exemple accéder à la mémoire, ou arrêter l'horloge `clk0`), de fonctionner la plupart du temps à horloge rapide.

Pour être utile, cette fonction de ralentissement doit elle-même pouvoir être effectuée quelle que soit la vitesse d'horloge. Or, étant une manipulation d'horloge, il s'agit d'une opération intrinsèquement difficile donc lente. Pour résoudre cette contradiction, on a choisi de donner à cette fonction une interface *logiquement asynchrone* (bien qu'électriquement synchrone), comme suit:

- Lorsque l'application requiert un ralentissement d'horloge, cette opération n'est pas effectuée tout de suite, mais un certain nombre n de cycles plus tard; ce nombre n est compris entre 1 et une limite supérieure n_{max} connue (pour Perle-1, $n_{max} = 6$).
- Une application devant effectuer une opération critique du cycle N au cycle M ($M \geq N$) demande le ralentissement d'horloge entre les cycles $N - n_{max}$ et M; ce ralentissement sera en fait effectif entre les cycles $N - k_1$ et $M + k_2$, où k_1 et k_2 sont des nombres non-déterministes compris entre 0 et $(n_{max} - 1)$ dépendant de la vitesse d'horloge courante et des délais de routage.
- Il est immédiat que si l'application est conforme au modèle logique synchrone, la sémantique de l'exécution ne dépend pas des valeurs de k_1 et k_2. L'exécution est donc correcte, sans qu'il ait été nécessaire de régler la vitesse d'horloge globale au niveau du cycle le plus lent.

Note

De la même façon que pour la distribution d'horloge mentionnée au §2.2.3, la réalisation d'un générateur d'horloge satisfaisant ces contraintes s'est révélée être un défi technologique, que ce soit pour la partie "oscillateur programmable" ou pour la partie "mode/arrêt". Ce sujet ne sera pas développé ici. Mentionnons toutefois que cette situation semble être en évolution rapide avec l'annonce récente de composants performants intégrant l'une ou l'autre de ces fonctions [ICD92].

2.5.4 Interface avec l'hôte

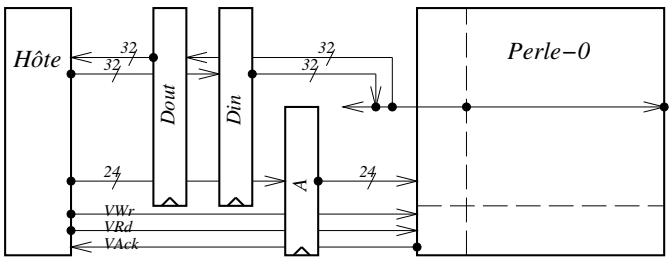


Figure 2.24: Perle-0: interface avec l'hôte

Perle-0

De même que pour les autres parties, l'interface hôte de Perle-0 a été conçue aussi simple que possible (figure 2.24). Elle est réduite à un registre d'adresse `A`, un registre de données d'entrée `Din`, un registre de données de sortie `Dout` et trois bits `VWr`, `VRd` et `VAck` implémentant un protocole de transaction asynchrone classique (*handshake*):

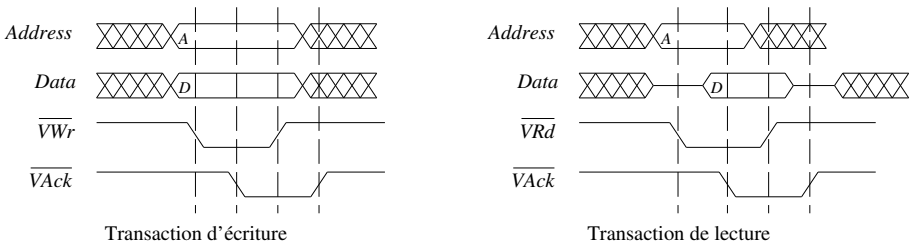


Figure 2.25: Perle-0: protocole de communication avec l'hôte

- Sur une transaction d'écriture (figure 2.25, partie gauche), l'hôte écrit respectivement l'adresse et la donnée dans `A` et `Din`, puis active `VWr`; la PAM lit ces registres et active `VAck`; l'hôte désactive `VWr`; la PAM désactive `VAck`.
- Sur une transaction de lecture (figure 2.25, partie droite), l'hôte écrit l'adresse dans `A` puis active `VRd`; la PAM lit l'adresse, écrit la donnée résultat dans `Dout` puis active `VAck`; l'hôte lit la donnée puis désactive `VRd`; la PAM désactive `VAck`.

Un tel schéma a l'avantage de la simplicité, mais présente plusieurs désavantages qui ont conduit à adopter une interface nettement plus sophistiquée pour Perle-1:

- le protocole possède une composante *temps réel*: la PAM doit répondre à une transaction dans un intervalle de temps donné, sinon celle-ci sera annulée par l'hôte (le processus de pilotage recevant une "erreur de bus"). Ceci contredit la propriété fondamentale du modèle logique synchrone qui est qu'un circuit fonctionne correctement à toute fréquence inférieure à sa fréquence critique. En particulier il devient impossible d'observer en mode pas-à-pas les cycles situés à l'intérieur d'une transaction bus, d'où des difficultés potentielles de mise au point.
- les signaux utilisés par le protocole sont *asynchrones* et doivent donc être manipulés avec extrême précaution à l'intérieur des circuits programmables.
- le couplage entre l'application et son logiciel de pilotage est trop serré (chaque transaction de lecture ou d'écriture étant en fait une synchronisation par *rendez-vous*), et s'est avéré être source d'inefficacités: par nature, la partie matérielle de l'application (implémentée dans la PAM) consomme et produit généralement un flot de données régulier, alors que la partie logicielle (implémentée sur le processeur de la machine hôte) est sujette à de forts à-coups de bande passante, le processeur devant effectuer d'autres tâches. Il en

résulte aisément un faible taux d'utilisation de l'une ou de l'autre, qui passe l'essentiel de son temps à attendre les rendez-vous.

L'écriture de programmes logiciels qui sont efficaces sous de telles contraintes (temps-réel à très faible granularité) est un challenge généralement déraisonnable et qui va à l'encontre des principes standards d'utilisation des processeurs. Un exemple extrême est représenté par le logiciel de pilotage de l'application d'encryption à clé publique développée par Mark Shand et décrite en détail dans [SBV91]. Ce logiciel devait piloter trois cartes Perle-0 en parallèle, tout en assurant lui-même une quantité de calcul non négligeable. Afin d'obtenir un bon taux d'utilisation de ces ressources (trois PAMs et un processeur), M. Shand a du développer des techniques permettant d'entrelacer au niveau de l'instruction machine les quatre séquences d'instructions correspondant au pilotage des quatre différents processus.

Il est possible de remédier partiellement aux deux premiers de ces inconvénients par l'utilisation dans toutes les applications d'un module d'interface soigneusement pré-optimisé qui présente une interface synchrone. C'est une version améliorée de ce module qui a été implémentée directement dans la logique de support de Perle-1. Le troisième sera résolu par l'introduction de *files d'attente*.

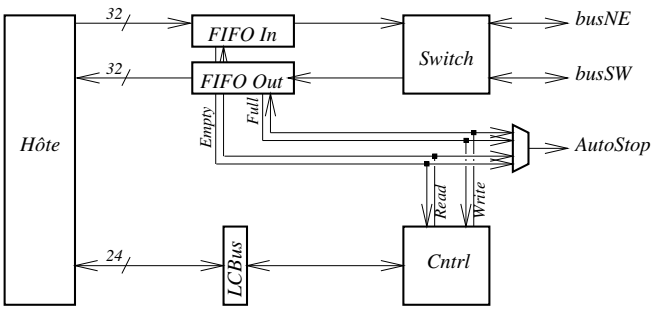


Figure 2.26: Perle-1: interface avec l'hôte

Perle-1

Le schéma de principe de l'interface hôte de Perle-1 est décrit sur la figure 2.26.

FIFOs Il a été inséré des *files d'attente* (par la suite dénommées *FIFOs*, de l'anglais First In, First Out), dans les canaux d'entrée et de sortie de la PAM. L'avantage principal que celles-ci procurent est qu'elles découplent les bandes passantes de la PAM et de son logiciel de pilotage au niveau le plus bas: l'application PAM typique va consommer ses données et produire ses résultats à un rythme régulier, par exemple un mot tous les N cycles; le logiciel de pilotage typique va par contre avoir un débit beaucoup plus erratique, produisant ou consommant un paquet de mots rapidement puis s'interrompant un temps assez long pour effectuer des calculs, ranger les résultats sur disque, etc... La

présence de ces tampons permet, pourvu que les bandes passantes soient globalement du même ordre, d'amortir ces à-coups et de minimiser le temps d'attente de part et d'autre.

Ces FIFOs fournissent de plus l'endroit idéal pour encapsuler la barrière de synchronisme qui existe obligatoirement entre la PAM et sa machine hôte: chaque FIFO présente une interface purement synchrone à chacune de ses extrémités (sur des horloges différentes), et la gestion de l'asynchronisme est entièrement repoussée dans un circuit d'arbitrage des indicateurs de remplissage qui est implémenté dans le FIFO lui-même.

Pour la très grande majorité des applications implémentables sur PAM, cette solution est beaucoup mieux adaptée que la solution adoptée pour Perle-0, et permettra d'obtenir un bien meilleur rendement tout en simplifiant considérablement la tâche de l'implémenteur. Il existe néanmoins une certaine classe d'applications qui peuvent être pénalisées, à savoir celles qui dépendent d'une faible latence de couplage entre la PAM et le processeur hôte, ou de la possibilité d'effectuer des transactions de lecture synchrone (ces dernières ne sont pas implémentables simplement par une interface avec FIFOs). Un exemple du premier type serait un coprocesseur arithmétique de bas niveau similaire aux circuits classiques de calcul en virgule flottante. Un exemple du deuxième type serait une application de contrôleur d'écran graphique intelligent (*smart frame buffer*), qui requiert la possibilité d'effectuer des lectures dans la mémoire d'écran.

Cet inconvénient potentiel n'a néanmoins pas été retenu, sous les arguments suivants:

- Les applications qui en souffrent sont très largement minoritaires par rapport à l'ensemble des applications intéressantes à implémenter sur PAM.
- La tendance des architectures modernes d'ordinateurs est de reléguer les périphériques d'entrée-sortie de plus en plus loin du processeur central, et de favoriser la bande passante plutôt que la latence dans la communication avec ces périphériques (en fait, les canaux de communication entre le processeur et les périphériques contiennent souvent eux-mêmes des files d'attente à cet effet). Il serait donc de toute façon inefficace d'implémenter une telle application sur une PAM qui comme Perle-1 est connectée à l'ordinateur hôte comme un périphérique classique.

Transactions Les transactions suivantes ont été implémentées entre la machine hôte et la PAM (voir aussi §3.2):

- `WriteFifo`: Ecriture d'un mot de données (32 bits) dans le FIFO d'entrée.
- `ReadFifo`: Lecture d'un mot de données (32 bits) dans le FIFO de sortie.
- `WritePAM`: Ecriture d'un mot d'adresse (24 bits) et d'un mot de données (32 bits) dans le FIFO d'entrée. Par rapport à la transaction WriteFIFO, celle-ci fournit une bande passante presque deux fois supérieure, ainsi que des garanties d'atomicité qui peuvent être utiles en cas d'utilisation de la PAM par plusieurs sources concurrentes.
- `ReadPAM`: Ecriture d'un mot d'adresse (24 bits) dans le FIFO d'entrée et lecture d'un mot de données (32 bits) dans le FIFO de sortie. Cette transaction est d'une utilité discutable et n'est présente que par souci de complétude.

- `GetFifoFlags`: Lecture de l'état des FIFOs, mesuré sur une échelle à 5 valeurs variant de "vide" à "plein". Il s'agit en fait d'une approximation par défaut du nombre de cases libres dans le FIFO d'entrée – ou du nombre de cases pleines dans le FIFO de sortie –, de façon à assurer la correction de l'algorithme décrit au paragraphe "contrôle de flux" ci-dessous.
- `DMA Read`: Transfert d'un bloc de n mots de données contigus de la mémoire de l'hôte au FIFO d'entrée, sans intervention du processeur central (*Direct Memory Access*). La bande passante disponible est alors maximale, soit pratiquement 100 Moctets/s. Grâce à une logique de contrôle sophistiquée implémentée par P. Boucard, la taille n du bloc n'est pas bornée supérieurement (elle varie cependant par multiples de 64).
- `DMA Write`: Transfert d'un bloc de n mots de données contigus du FIFO de sortie à la mémoire de l'hôte, sans intervention du processeur central (*Direct Memory Access*). Comme ci-dessus, la bande passante disponible est de 100 Moctets/s, et n peut varier par pas de 64 sans borne supérieure.

Contrôle de flux — Autostop La présence de ces FIFOs supprime la nécessité d'une synchronisation de la PAM et de l'hôte à chaque transaction, mais impose d'avoir un mécanisme de *contrôle de flux* permettant d'éviter que les FIFOs ne débordent (c'est-à-dire qu'on ne lise un FIFO vide ou qu'on n'écrive un FIFO plein).

- Du point de vue du logiciel il n'est pas nécessaire d'implémenter un quelconque mécanisme automatique, car le programme de pilotage peut relire à tout moment l'état des FIFOs et adapter son comportement en conséquence. On utilisera typiquement l'algorithme suivant (écrit ici pour le transfert d'un bloc B de N mots de l'hôte vers la PAM):

  $n \leftarrow 0$
  tant que $n < N$ **faire**
  $\quad s \leftarrow \min(\text{GetFifoState}(\text{FifoIn}), N - n)$
  $\quad$ **pour** i **variant de** 1 **à** s **faire**
  $\quad\quad \text{WriteFifo}(B[n+i])$
  $\quad n \leftarrow n + s$

  Il est immédiat qu'en utilisant un algorithme de ce type, le surcoût attaché au test de l'état du FIFO tend vers 0 lorsque le FIFO devient plus grand et la PAM plus rapide que le processeur hôte, ce qui garantit bien une utilisation efficace du processeur.
- Du point de vue de la PAM, le même mécanisme est possible (avec $0 \leq s \leq 1$) puisque les circuits de contrôle ont accès à l'information "FIFO d'entrée vide" et "FIFO de sortie plein" (figure 2.26). Pour répondre à l'usage de loin le plus courant, on a néanmoins choisi d'implémenter un mode optionnel de *contrôle de flux automatique* ci-après nommé *autostop*: lorsque ce mode est en service, une application verra son horloge principale `clk0` automatiquement coupée dès qu'elle

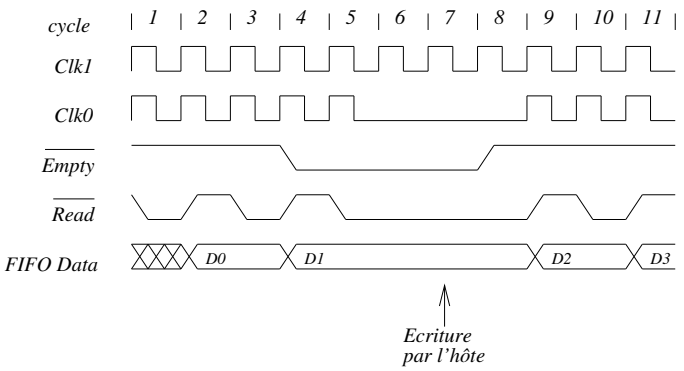


Figure 2.27: Un exemple de comportement du FIFO d'entrée en mode autostop

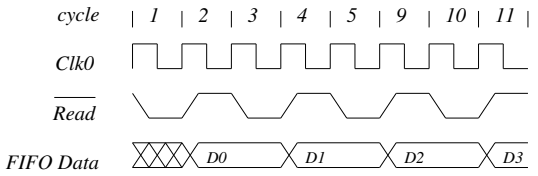


Figure 2.28: L'exemple de la figure 2.27, projeté sur l'horloge `clk0`

essaie de faire une écriture dans un FIFO de sortie plein, ou une lecture dans un FIFO d'entrée vide.

La figure 2.27 montre un exemple d'utilisation de ce mode par une application qui lit le FIFO d'entrée un cycle sur deux: au cycle 5, l'application requiert une lecture alors que le FIFO est vide; son horloge principale `clk0` s'arrête alors jusqu'au cycle 9, qui est le premier cycle pour lequel un nouveau mot de données est disponible. La figure 2.28 montre la même séquence, mais du point de vue de l'horloge `clk0` uniquement: la coupure n'apparaît plus, et tout se passe comme si le FIFO n'était jamais vide dans les cycles où l'on a besoin de le lire. Le comportement sur le FIFO de sortie serait exactement similaire.

Lorsque ce mode est en service, une application qui n'utilise que l'horloge principale `clk0` travaille donc dans un *monde logique synchrone parfait*, dans lequel il est possible de lire un mot d'entrée et/ou d'écrire un mot de sortie à chaque cycle d'horloge, sans se préoccuper de l'état des FIFOs. Ceci élimine tout souci de synchronisation de la conception de la grande majorité des applications.

Port asynchrone Un des inconvénients de l'interface simplifiée de Perle-0 est que le seul moyen de communiquer avec l'application PAM depuis le logiciel de pilotage est à travers le chemin de données. Or, il est souvent nécessaire de distinguer les transactions de "données" (entrée de données ou sortie de résultat) des transactions de "contrôle" (un ordre de grandeur moins fréquentes mais souvent indispensables, par exemple pour changer un mode de fonctionnement, ou signaler la fin d'un calcul). Cela a

fréquemment conduit les applications programmées sur Perle-0 à implémenter des protocoles compliqués et des mécanismes de multiplexages permettant de séparer les transactions de données des transactions de contrôle dans l'unique flux d'entrée.

On a donc choisi d'implémenter dans Perle-1 un canal de communication direct entre l'hôte et les circuits de contrôle de l'application, sous la forme d'un port parallèle bi-directionnel d'usage général ("LCBus"): il est ainsi possible au logiciel de communiquer des informations de "mode" à l'application sans passer par les FIFOs, et à l'application de renvoyer des informations similaires (par exemple "calcul terminé"). A noter que ce port est logiquement asynchrone par rapport au flot des données, mais bien sûr électriquement synchrone par rapport à l'horloge, de façon à rester là encore au sein du modèle logique synchrone décrit au chapitre 1.

Interruptions Sous contrôle logiciel, un des signaux du port asynchrone décrit ci-dessus peut optionnellement être utilisé comme signal d'interruption du processeur hôte (cf. §3.2.2). Ceci permet d'optimiser l'écriture des logiciels de pilotage des applications ne demandant que peu d'interventions du processeur, en évitant le recours à l'attente active.

Tags Les circuits intégrés implémentant les FIFOs offrent fréquemment des largeurs du type "$2^n+\epsilon$" (originellement prévues pour implémenter des chemins de données avec contrôle de parité ou code correcteur d'erreur); pour Perle-1 ils ont ainsi une largeur de 36 bits au lieu de 32. Dans le cas du FIFO d'entrée (c'est-à-dire dans le sens hôte vers PAM) on a choisi d'utiliser ces bits supplémentaires comme des marqueurs – ci-après appelés *tags* – permettant d'identifier la provenance de chaque mot. On a ainsi implémenté:

- un bit indiquant si le mot est une adresse (écrite par une transaction `WritePAM` ou `ReadPAM`) ou une donnée;
- un bit indiquant si le mot fait partie d'un paquet écrit par DMA ou non;
- deux bits d'usage général dont la valeur peut être librement choisie depuis le logiciel de pilotage.

Ces bits supplémentaires sont délivrés à l'application embarquée dans la PAM en même temps que le mot de donnée auquel ils sont attachés. Bien qu'ayant été introduits par hasard du fait de la largeur des FIFOs disponibles commercialement, ils se sont révélés très utiles lorsque l'application traite un flot continu de données de longueur variable, ou un flot de commandes et de données entremêlées: ils permettent en effet de *typer* les mots d'entrée (par exemple "ceci est la donnée A" ou "ceci est une commande") de façon synchrone avec ce flux de données. Sans ce mécanisme, les solutions alternatives possibles seraient:

- utiliser quelques bits du mot d'entrée à cet effet, réduisant effectivement sa largeur à moins de 32 bits; ceci est la plupart du temps inacceptable;
- utiliser le canal "LCBus" pour passer cette information; ceci nécessite une synchronisation logiciel – PAM (c'est-à-dire le vidage complet du FIFO d'entrée) à chaque modification de l'état de ces bits, et peut fortement pénaliser les performances;

- utiliser un protocole de communication entre l'hôte et la PAM qui inclut ou rend inutile cette information (par exemple en transmettant systématiquement la longueur et le type d'un bloc de mots avant celui-ci); ceci impose alors l'implémentation d'un décodeur de ce protocole dans la partie matérielle de l'application.

2.5.5 Contrôle des mémoires

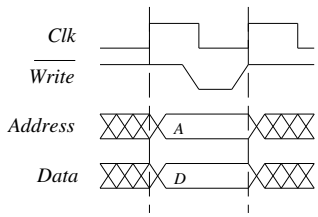


Figure 2.29: Cycle d'écriture d'une mémoire statique

Les circuits utilisés pour implémenter les bancs mémoire de Perle-0 et Perle-1 sont des mémoires statiques conventionnelles. En lecture, ils se comportent de façon purement combinatoire, mais en écriture ils demandent la génération d'une impulsion électrique calibrée strictement incluse à l'intérieur du cycle d'écriture (figure 2.29, signal $\overline{Write}$). Pour l'utilisation directe depuis une PAM, ceci laisse plusieurs solutions au programmeur, dont aucune n'est satisfaisante:

- utiliser trois cycles d'horloge pour une transaction d'écriture (en générant l'impulsion d'écriture durant le cycle médian), ce qui n'est bien sûr possible que si l'on n'a pas besoin d'utiliser la mémoire à son maximum de bande passante;

- sortir du modèle logique synchrone et générer l'impulsion d'écriture à partir de boucles combinatoires et de délais calibrés expérimentalement.

Lors de l'utilisation de Perle-0, cette dernière approche s'est révélée extrêmement délicate et source de nombreux problèmes de mise au point, malgré l'utilisation de modules de bibliothèques optimisés manuellement. Ceci confirme le résultat désormais maintes fois énoncé qu'il est extrêmement imprudent de sortir du modèle logique synchrone lorsque l'on utilise des circuits programmables du type des circuits Xilinx.

On a donc inclus dans l'électronique de support de Perle-1 des circuits de "conditionnement" des commandes permettant à la mémoire de présenter une interface purement synchrone à la PAM, de façon telle qu'une transaction de lecture ou d'écriture puisse être effectuée à chaque cycle sur chaque banc, sans précaution particulière. La figure 2.30 montre le modèle de mémoire ainsi obtenu, tel qu'il est vu par le programmeur PAM (on a ici représenté le bit j du mot d'adresse i).

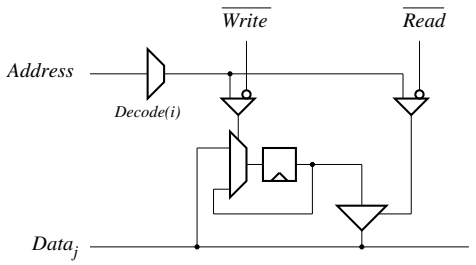


Figure 2.30: Perle-1: modèle de l'interface RAM synchrone

Note Ces circuits de synchronisation introduisent intrinsèquement une limite supérieure à la fréquence de fonctionnement de la PAM (pour Perle-1: 35 Mhz). Ils sont donc optionnellement débranchables sous contrôle logiciel de façon à ce que le très petit nombre d'applications qui voudraient utiliser des fréquences supérieures puissent quand même utiliser la mémoire en mode asynchrone (les modes *Slow* et *Clk1Div2* du générateur d'horloge décrits au §2.5.3 fournissent d'autres moyens de contourner cette limitation éventuelle).

Remarquons enfin que face à l'accroissement de fréquence des microprocesseurs modernes – et ainsi que c'est déjà le cas pour les FIFOs mentionnés précédemment –, des circuits de mémoire intégrant directement cette interface synchrone sont en train de faire leur apparition sur le marché, pour résoudre les mêmes difficultés d'utilisation que celles qui nous préoccupent ici. A la date de conception de Perle-1, ces circuits n'étaient encore ni assez gros ni assez rapides pour être utilisables, mais il est probable que cette situation va évoluer rapidement.

2.6 Autres PAMs

On tente ici de décrire les points importants de l'architecture de quelques-unes des autres PAMs connues (§1.3.3), pour autant que l'information soit disponible, et de les mettre en comparaison avec Perle-0 et Perle-1.

2.6.1 Quickturn RPM

Assez peu d'information a été publiée sur l'architecture des machines d'émulation produites par la société Quickturn [Qui91]. Apparues dès la disponibilité des premiers composants prédiffusés programmables, celles-ci reposent sur un principe similaire à celui des PAMs mais sont destinées à un usage différent. Il s'agit ici d'*émuler* un circuit intégré classique en cours de développement de façon à pouvoir le mettre en œuvre aussi tôt que possible, soit par simulation (avec l'objectif d'une vitesse d'exécution plus rapide que par les techniques logicielles classiques de simulation), soit par émulation proprement dite, c'est-à-dire connecté physiquement à l'environnement matériel dans lequel le circuit final évoluera (avec l'espoir d'obtenir une vitesse d'exécution qui, bien que très lente, permettra la validation de la conception du circuit en développement).

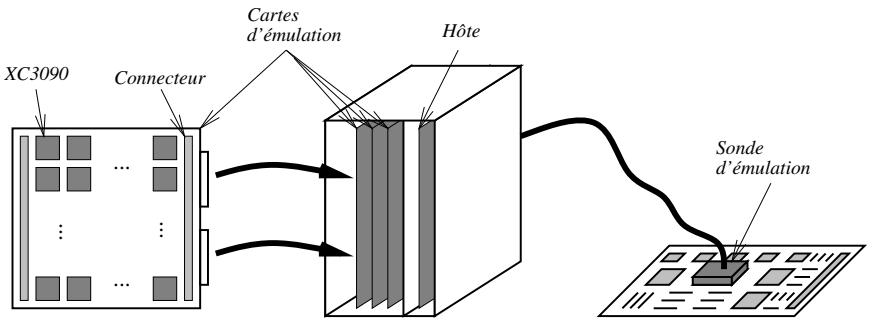


Figure 2.31: Architecture de la machine Quickturn RPM

Dans les deux cas, la méthode consiste à prendre la description exacte du circuit en développement (graphe de portes logiques élémentaires tel qu'il sera fourni au fabricant de semi-conducteurs, cf. §1.1.3) et à la transformer de façon entièrement automatique en une réalisation sur circuits prédiffusés programmables. Cette contrainte impose de disposer d'une quantité considérable de ressources logiques programmables, en raison du surcoût entrainé par le processus d'émulation. Les machines Quickturn RPM possèdent ainsi les caractéristiques suivantes (figure 2.31):

- Des cartes électroniques pavées de composants logiques programmables constituent le cœur du système. Dans la version la plus performante, chaque carte comporte une matrice 7×9 de composants Xilinx XC3090, soit environ 4 fois la taille de la matrice principale de calcul de Perle-1, pour une capacité d'émulation annoncée de 12 500 portes logiques élémentaires.

- Le système complet comporte 1 à 4 de ces cartes, connectées entre elles via une grande quantité de connecteurs périphériques, et reliées à une station de travail hôte par l'intermédiaire d'un bus VME.

- Une sonde d'émulation permet de relier le système à l'environnement physique du circuit en développement (il s'agit d'un câble terminé par un connecteur identique à celui qu'aura le circuit final, qui se connecte directement à la carte électronique sur laquelle celui-ci sera implanté; cette liaison comprendra typiquement plusieurs centaines de signaux logiques).

- Le système possède des connecteurs permettant de connecter des composants physiques pouvant se substituer à la logique programmable pour l'émulation plus efficace d'une partie du circuit final (par exemple, on utilisera typiquement des boîtiers mémoire standard pour émuler la mémoire cache des microprocesseurs, car la logique programmable demanderait pour ce faire un volume totalement prohibitif).

- Jusqu'à 20 de ces systèmes peuvent être connectés en réseau pour augmenter les capacités d'émulation jusqu'à un maximum annoncé de un million de portes.

L'architecture de cette machine est donc très similaire à celle de notre modèle PAM virtuel (une matrice bidimensionnelle de PABs de taille virtuellement infinie), et l'algorithme de compilation employé est très certainement proche dans ses principes de l'algorithme formel esquissé au §1.2.4. En particulier, cette machine ne possède pas de mémoire locale comme c'est le cas pour une PAM, car il serait très difficile de l'utiliser efficacement lors d'une compilation automatique partant d'un graphe de portes logiques (l'utilisation de circuits mémoire pour émuler directement des parties du circuit final est une optimisation manuelle faite directement par les concepteurs de ce circuit).

Dans l'état actuel des techniques de compilation, la pénalité en surface provoquée par ce processus d'émulation semble être de l'ordre d'un facteur 40 à 50 (un circuit XC3090 pouvant émuler en moyenne 250 portes élémentaires). Le facteur limitant est ici le nombre de pattes d'entrée-sortie sur les circuits programmables, qui limite la capacité de routage inter-circuits. La pénalité en temps est essentiellement une conséquence de cette pénalité en surface, et semble être de l'ordre de 10 à 20, les circuits émulés fonctionnant généralement à une fréquence de 0.5 à 2 MHz.

La machine obtenue est donc intrinsèquement chère, du fait du volume de logique programmable nécessaire à l'émulation de circuits modernes. Elle semble néanmoins recevoir un certain succès dans les équipes de développement des microprocesseurs actuels les plus complexes, par la vitesse de simulation qu'elle permet d'obtenir.

2.6.2 Splash

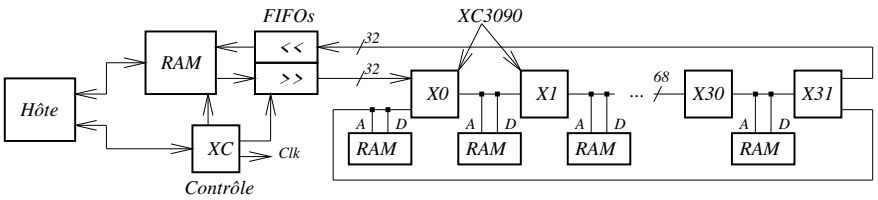


Figure 2.32: Architecture de la machine Splash

La machine Splash [GHK91] repose elle aussi sur le principe PAM, mais est orientée vers l'implémentation d'architectures systoliques (cf. §1.3.3), c'est-à-dire de structures régulières formées d'éléments identiques communicant de proche en proche. Les algorithmes systoliques s'expriment généralement en terme de structures multidimensionnelles, mais on sait souvent les réaliser sur des architectures mono-dimensionnelles. L'architecture de cette machine est donc centrée autour d'une chaîne de cellules identiques organisées en "pipeline" (figure 2.32):

- Chaque cellule comporte des ressources logiques, sous la forme d'un circuit Xilinx XC3090, et de la mémoire, sous la forme d'un banc de 128k$\times$8 bits.
- Les cellules sont reliées de proche en proche par un chemin de données de 68 bits de large; les bancs mémoires sont en fait connectés sur ces connexions inter-circuits (adresses, données et contrôles).

- La première et la dernière cellule reçoivent respectivement le canal d'entrée et le canal de sortie permettant la connexion à l'hôte; ces canaux n'ayant que 32 bits de large, les 36 connexions restantes sont connectées entre elles de façon à former une boucle.

- La communication avec l'hôte se fait par un bus VME, à travers un banc mémoire tampon additionnel (8 Moctets) permettant de doubler la bande passante disponible, soit 2 Moctets/s.

Bien que les conceptions se soient faites sans interférence entre les équipes, un certain nombre de points communs peuvent être retrouvés avec les architectures de Perle-0 et Perle-1:

- Les circuits programmables sont configurables depuis l'hôte; la logique d'interface et de configuration est implémentée dans deux circuits programmables supplémentaires configurés à la mise sous tension depuis une mémoire morte résidente (c'était le cas de Perle-0; les circuits d'interface de Perle-1 comme ceux de Splash II sont configurables depuis l'hôte pour plus de flexibilité).

- Le générateur d'horloge est à vitesse variable programmable depuis l'hôte; il possède des modes pas-à-pas et rafale.

- La relecture de l'état des circuits programmables est implémentée; son couplage avec le mode pas-à-pas ci-dessus fournit un outil de mise au point qui rend peu nécessaire le recours à la simulation.

- La communication avec l'hôte se fait à travers un jeu de FIFOs qui découplent les bandes passantes et isolent la barrière de synchronisme; un contrôle de flux automatique similaire à celui de Perle-1 est disponible (arrêt de l'horloge lorsque les FIFOs sont vides ou pleins).

Un certain nombre de limitations peuvent néanmoins être trouvées (et sont reconnues par les auteurs dans [GHK91]), qui sont imputables au manque d'expérience préalable des concepteurs (comme Perle-0, cette machine a été conçue dès l'apparition des premiers circuits logiques programmables):

- La plus importante concerne le niveau de performance atteignable, par rapport à ce que les circuits programmables pourraient délivrer:

  - La bande passante d'entrée-sortie est limitée à 1 Moctet/s (ou au maximum le double dans les cas rares où les données chargées dans la mémoire d'interface sont recirculées plusieurs fois dans le réseau); la présence de la mémoire d'interface allonge encore la latence de la communication.
  - La fréquence d'horloge est choisie par divisions par 2 successives d'une base à 32 MHz, ce qui offre une précision bien trop faible; elle est de toute façon souvent limitée à 1 ou 2 MHz par la fréquence de fonctionnement des FIFOs d'entrée-sortie; les temps de propagation entre circuits sont aggravés par une dispersion (*skew*) de la distribution d'horloge pouvant aller jusqu'à 20 ns.

- Les bancs de mémoire locale sont souvent trop étroits pour une utilisation efficace en tables (il faut quatre lectures pour obtenir un mot de la largeur naturelle du chemin de données); leur connexion directe sur les fils d'interconnexion entre circuits est souvent une nuisance, comme c'était le cas pour Perle-0.

- L'accès aux bancs de mémoire locale ne peut se faire qu'à travers l'application embarquée dans la PAM, ce qui est un frein à la mise au point et une possible limitation à l'utilisation. Cette contrainte a été aussi reconnue lors de la conception de Perle-1 mais n'a pas pu être simplement résolue. Splash II contourne cette limitation par une complication d'architecture qui semble très chère.

- Les temps de chargement et de relecture sont longs (une demi-seconde, par rapport aux 20 et 10 ms de Perle-1), ce qui limite les possibilités de coopération inter-applications.

- Les circuits d'interface ne sont pas relisibles, ce qui complique leur mise au point; cette limitation était aussi présente dans Perle-0, et a été corrigée dans Perle-1 (et Splash II).

Malgré ces limitations en performance, dont on peut aisément chiffrer l'impact à un facteur 10 ou 20, les résultats obtenus sur les quelques algorithmes systoliques implémentés qui ont été publiés sont de premier ordre, dépassant largement ceux des réalisations sur super-ordinateurs. Ceci démontre l'intérêt de la technologie PAM, même sous une forme simple, dans ce cas particulier où le volume de logique consacré à un problème algorithmique peut aisément être multiplié par un facteur quasi-arbitraire.

2.6.3 Splash II

La machine Splash II (qui n'est pas encore opérationnelle à la date de rédaction, mais dont l'architecture a déjà été publiée [ABD92]) reprend l'architecture précédente en l'étendant dans plusieurs directions:

- le volume de logique disponible a été considérablement augmenté, en tirant parti de l'évolution technologique des circuits sous-jacents d'une part, et en jouant la carte de la *modularité* (assemblage multi-cartes) d'autre part;

- les fréquences de fonctionnement possibles ont été relevées par utilisation de meilleures technologies de réalisation, et par l'utilisation d'un bus d'entrée-sortie plus performant;

- l'architecture est plus flexible que la précédente, par l'introduction d'un système complexe d'interconnexion permettant de s'écarter des structures purement systoliques;

- un certain nombre des limitations mentionnées plus haut ont été corrigées.

L'architecture de cette machine est schématisée sur la figure 2.33. Bien que peu d'informations soit encore disponible, on relèvera les points importants suivants:

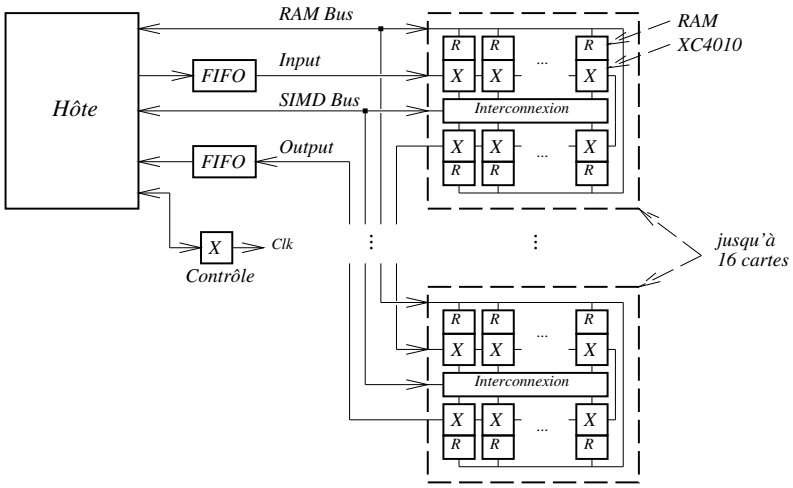


Figure 2.33: Architecture de la machine Splash II

- L'élément essentiel de l'architecture est une carte comportant 16 circuits Xilinx XC4010 organisés comme précédemment en chaîne systolique et connectés entre eux par un chemin de données de 36 bits de large. Compte-tenu de la puissance accrue de ces nouveaux circuits, cela représente environ la même capacité de calcul que la carte Splash originale toute entière. Chaque circuit possède un banc de mémoire locale de $256\text{k} \times 16$ bits, qui lui est relié directement sans interférence avec le chemin de données principal.

- Le système complet contient 1 à 16 de ces cartes, connectées par leurs chemins systoliques en une seule grande chaîne. L'ensemble est piloté par une carte d'interface offrant une bande passante de communication avec l'hôte de plusieurs dizaines de Moctets/s.

- En plus du chemin de données systolique, un jeu de circuits d'interconnexion spécialisés permet au niveau de chaque carte d'établir toute une famille de schémas d'interconnexion, allant de la diffusion globale à des schémas réguliers à 2 ou 4 dimensions. Le canal d'entrée du système fait aussi partie des ports de ce réseau d'interconnexion, ce qui permet de diffuser en parallèle des données à toutes les cartes.

- Un générateur d'horloge plus sophistiqué que le précédent permet le réglage fin de la fréquence, ainsi que les modes pas-à-pas et rafale.

- Des chemins supplémentaires d'adresses et de données permettent d'accéder directement les bancs de mémoire locale depuis l'hôte.

- Un connecteur externe sur la carte d'interface permet de s'interfacer à des dispositifs matériels externes (à faible débit).

Il est encore trop tôt pour se prononcer sur les qualités de cette architecture, même s'il est certain que – comme cela a été le cas pour Perle-1 – ses concepteurs ont été dans une certaine

mesure victimes du "syndrome du second système[2]" en corrigeant les défauts remarqués dans la première version. Cette machine possède certainement une puissance de calcul considérable, mais aussi une architecture complexe. Il est certain que certaines classes d'algorithmes – et en particulier les algorithmes systoliques – peuvent tirer parti d'un tel volume de logique et obtenir des performances de premier ordre. Néanmoins, encore plus que dans le cas de Perle-1, il paraît difficile d'automatiser la compilation des circuits sur une architecture qui comme celle-ci semble possèder de nombreuses idiosyncrasies. L'avenir proche le dira d'autant plus que les architectes annoncent vouloir utiliser la synthèse logique comme leur principal outil de programmation.

2.6.4 Tableau comparatif

La figure 2.34 rassemble les caractéristiques (estimées) de taille et de vitesse d'un certain nombre de PAM connues. On la mettra en relation avec la table présentée par la figure 1.14.

PAM	Surface (PABs)	Fréquence (MHz)	Puissance (GBops)
Perle-0	3k	16	48
Perle-1	12k	25	300
Quickturn [Qui91]	40k	1	40
(4 cartes)	160k	1	160
Splash [GHK91]	20k	2	40
Splash II [ABD92]	19k	20	380
(16 cartes)	300k	20	6 000
CHS 2×4 [KB92]	2k	15	30
Chameleon [HP92]	2k	15	30
AnyBoard [BMT92]	3k	15	45

Figure 2.34: Puissances comparées de différentes PAMs

[2] "When one is designing the successor to a relatively small, elegant and successful system, there is a tendency in one's success to become grandiose and design an elephantine feature-laden monstrosity."

Chapitre 3

Programmation

Le “programme” que l’on doit mettre au point pour implémenter une application donnée sur PAM est constitué de deux parties bien distinctes (bien que leurs conceptions soient fortement couplées et généralement menées en parallèle):

- une partie matérielle: c’est la configuration de la PAM, c’est-à-dire la description du circuit logique synchrone qu’elle doit implémenter;
- une partie logicielle: c’est la séquence d’instructions qui s’exécute sur l’hôte, et qui exerce ce circuit logique pour obtenir le résultat recherché.

Ce chapitre décrit les outils que l’on a été amené à développer pour programmer et utiliser les PAMs Perle-0 et Perle-1 (§1.3.2). La première section est consacrée au développement de la partie matérielle des applications. La deuxième section traite du développement de la partie logicielle.

3.1 Programmation matérielle

On s’intéresse ici aux outils et à l’environnement de développement permettant d’écrire et de mettre au point une configuration matérielle pour une PAM donnée.

Ce problème entre dans le cadre général de la conception assistée par ordinateur des circuits intégrés, un champ de recherches et de développements dont l’importance est considérable depuis de nombreuses années. Néanmoins, les PAMs représentaient jusqu’à récemment un domaine suffisamment nouveau et contraint pour que les technologies développées dans ce domaine ne soient pas directement applicables. On a donc été amené à développer une suite d’outils simples, mais spécifiques et bien adaptés, permettant de réaliser les premières applications.

Le travail décrit ici est relativement ancien, ayant été principalement effectué pour la programmation de la première génération de PAM réalisée au laboratoire DEC-PRL [BRV89]. Une réécriture complète en a été entreprise [Tou93], à la lumière de l’expérience acquise et pour tirer parti des travaux extérieurs qui commencent à se développer dans ce domaine. On ne décrira donc que les grandes lignes de ce système, en essayant de dégager les apports

originaux spécifiques à l'utilisation PAM ainsi que les principales critiques apparues au cours de trois années d'utilisation.

3.1.1 Introduction

Description du problème

La configuration d'une PAM pour une application donnée est spécifiée par la donnée de la configuration de chacun des PABs élémentaires qui la composent. Pour une PAM de la taille de Perle-1, il s'agira d'un système de plus de 14 000 équations booléennes à 4 entrées.

Même pour une PAM de taille plus réduite comme Perle-0, ceci représente beaucoup trop d'information pour qu'il soit praticable de construire cette configuration à partir des outils commerciaux normalement utilisés pour les circuits logiques programmables [Xil91]. Cette approche traditionnelle peut être résumée par la succession d'étapes suivantes (on se réfèrera à la section 1.3.1 pour un rappel des grandes lignes de l'architecture cible considérée):

Schémas Les équations logiques décrivant le circuit à implémenter sont décrites sous la forme d'un graphe source interconnectant des blocs standard (portes, registres, compteurs, etc...), et généralement entré sous forme graphique par l'intermédiaire d'un éditeur de schémas. Exceptionnellement, certains de ces blocs peuvent être directement décrits par un jeu d'équations logiques.

Mapping Un processus automatique (en anglais "technology mapping") transforme ce graphe source en un graphe objet ne contenant plus que des instances de la cellule logique élémentaire du prédiffusé programmable cible (l'équivalent du PAB), en expansant les blocs standards d'après une bibliothèque et en faisant appel à des algorithmes de minimisation booléenne. Le nombre des cellules élémentaires ainsi obtenues doit être inférieur au nombre de cellules disponibles dans le circuit programmable visé.

Placement Un processus automatique essaie de trouver une affectation "optimale" des cellules de ce graphe objet dans l'ensemble des cellules offertes par le circuit, en fonction de leur schéma d'interconnexion. Ce problème étant $\mathcal{NP}$-complet dans toute sa généralité, on utilise typiquement des algorithmes de type "recuit simulé", très lourds en temps de calcul et aux résultats incertains. De la qualité du résultat obtenu dépend crucialement la faisabilité de la phase suivante, ainsi que la qualité du résultat final (c'est-à-dire le niveau de performances obtenu).

Routage Un processus automatique essaie de trouver une configuration des ressources d'interconnexion du circuit (canaux de routage, points d'interconnexion programmables et matrices de routage) permettant d'établir l'interconnexion désirée entre les cellules, en minimisant les temps de propagation des signaux. S'il échoue, on peut éventuellement intervenir manuellement sur le placement et le routage obtenus, jusqu'à obtention du résultat escompté.

Les obstacles principaux à l'utilisation de ces outils – qui étaient les seuls disponibles lors de la réalisation de Perle-0 (1988) – sont les suivants:

- La description par schémas, même hiérarchisés, est complètement inadaptée à la description de circuits de taille conséquente; il devient nécessaire d'utiliser des techniques algorithmiques permettant de *calculer* le circuit à réaliser [MV85].
- Les algorithmes actuels de placement automatique obtiennent très difficilement des résultats acceptables sur des prédiffusés programmables de taille moyenne; le traitement d'une structure de la taille d'une PAM est complètement hors de leur portée. Il devient donc nécessaire de spécifier initialement la géométrie du circuit à réaliser au même titre que ses équations logiques.
- Aucun outil n'est disponible pour le traitement global de réalisations faisant intervenir plusieurs circuits prédiffusés programmables, comme c'est le cas pour une PAM.
- L'approche traditionnelle est lourde et difficilement incrémentale (il est par exemple conseillé d'intervenir manuellement sur le résultat final si la phase de routage n'a pas pu être achevée automatiquement; ce travail est perdu si on veut ensuite modifier le circuit). Une PAM permettant d'essayer à peu de frais un algorithme matériel, on veut impérativement pouvoir développer, tester et mettre au point les circuits de façon incrémentale.

On a donc été amené à développer un "environnement de programmation" plus adapté, beaucoup plus proche dans ses principes de ce qui est traditionnellement utilisé pour le développement logiciel (techniques dont on sait qu'elles permettent de maîtriser la construction et la mise au point de très gros édifices logiques).

Objectifs

Les objectifs principaux que l'on a poursuivis lors de ce développement peuvent se résumer comme suit. Précisons que ces points sont à considérer plus comme des fils conducteurs guidant la direction du développement, que comme des objectifs qu'il est impératif d'atteindre effectivement.

Abstraction de la machine physique Les outils essaient de présenter au programmeur un monde abstrait aussi proche que possible du modèle décrit au §1.2.2, et de l'affranchir des détails d'implémentation physique de la PAM utilisée. La principale raison en est bien entendu la simplification du modèle que le programmeur doit avoir à l'esprit pour travailler. En théorie cela permet aussi de porter plus facilement des applications d'une architecture de PAM à une autre.

Performance On s'est donné comme objectif impératif de ce développement de ne *rien sacrifier en performance*. Cela veut dire qu'il doit être possible de prendre le contrôle complet sur le résultat obtenu, partout où c'est nécessaire. La – relative – petitesse des PAMs actuelles, ainsi que la médiocrité des outils de synthèse automatique, fait que ceci reste encore bien souvent la clé de l'implémentation performante d'un algorithme.

On veut donc pouvoir définir le circuit à implémenter à travers des constructions de "haut niveau", mais pouvoir ajouter là où c'est nécessaire les indications de bas niveau (en

quantité minimale) qui permettent d'obtenir le résultat escompté. Il faut comparer cela à l'approche qui consiste, lors du développement d'un logiciel, à ajouter des directives de compilation ou à recoder des procédures critiques en langage machine, de façon à obtenir le niveau de performances souhaité. On peut espérer que pour la construction de circuits comme pour le logiciel, l'accroissement de la puissance et de la taille des circuits d'une part, la maturité croissante des algorithmes de compilation d'autre part, rendront cette approche de moins en moins nécessaire.

Compilation Il est indispensable que l'obtention finale de la configuration souhaitée puisse être obtenue par *compilation automatique* d'un jeu de *fichiers source* mis au point par le programmeur. Ceci veut dire que si une erreur est trouvée, ou si le circuit obtenu n'est pas assez performant, le programmeur modifie les fichiers source en conséquence et recompile l'ensemble du circuit. On exclut totalement les solutions consistant à intervenir manuellement sur le résultat de la compilation ou une quelconque phase intermédiaire.

Mise au point De façon à pouvoir développer et mettre au point les circuits de façon incrémentale, on veut disposer d'outils de mise au point qui travaillent au même niveau d'abstraction que les fichiers source: ceci conduit à développer des outils de *mise au point symbolique* comparables à ceux qui sont classiquement utilisés pour le développement logiciel.

Il est important de remarquer que pour la technologie qui nous préoccupe, la rapidité de recompilation du circuit d'une part, la capacité de relecture offerte par la PAM d'autre part (§1.2.6) relèguent au second plan le besoin en outils de simulation, pourtant omniprésent dans tous les autres domaines traitant de réalisation matérielle. La PAM elle-même peut effectivement être utilisée comme son propre simulateur, avec un niveau de performance supérieur à bien des réalisations logicielles et une correction de résultat garantie: la première génération d'applications PAM développées au laboratoire DEC-PRL l'ont en fait été avec cet unique outil de mise au point. En deuxième approche néanmoins, la disponibilité d'un simulateur logiciel permet de valider des modules logiques sans à avoir à concevoir de géométrie pour eux, ni à devoir les inclure dans une application PAM complète; la réécriture des outils de développement en cours à DEC-PRL inclut donc un simulateur complet [Tou93].

Travaux apparentés

La tendance actuelle des recherches en ce domaine est essentiellement centrée sur la synthèse logique automatique à partir de langages de description de circuits de "haut niveau", sous l'impulsion des besoins créés par la fabrication des circuits intégrés classiques. Les résultats obtenus à ce jour pour les prédiffusés programmables – qui constituent un monde beaucoup plus contraint – sont encore peu satisfaisants, surtout à l'échelle multi-circuits d'une PAM, essentiellement par la difficulté qu'il y a à synthétiser la géométrie des circuits. La plupart des utilisateurs "lourds" de prédiffusés programmables ont ainsi choisi comme nous de travailler directement au niveau des équations logiques et de la géométrie [GK+90, Pfi91]. Une exception

notable est constituée par le logiciel de compilation de la société Quickturn [Qui91] qui par nature n'incorpore aucune description géométrique; le prix à payer est un accroissement considérable du volume de logique programmable nécessaire ainsi que des temps de cycle obtenus.

Les outils de description de circuits que l'on a implémentés (§3.1.3) et tout particulièrement la description par langage de programmation sont essentiellement dérivés de techniques connues. [Ser84, BSS88, DM92] représente la lignée de travaux dont on s'est le plus inspiré.

Les outils de synthèse et de minimisation logique que l'on a utilisés pour l'optimisation logique des structures générées à partir du langage Esterel (§3.1.3) ont été développés par H. Touati à partir du logiciel SIS de l'Université de Berkeley [SS+92, BT93].

Il n'est pas possible de citer tous les travaux actuellement en cours portant sur des sujets apparentés à la programmation des prédiffusés programmables; la plupart de ceux-ci sont d'ailleurs postérieurs à ceux que l'on décrit ici. On mentionnera néanmoins les travaux récents de W. Luk [PL91], qui portent sur une approche originale et très simple de la description de circuits au moyen de langages de programmation; en première approximation ces travaux relèvent de la même approche que ceux effectués par G. Berry pour la compilation d'Esterel sur PAM [Ber91], bien que les langages utilisés répondent à des modèles différents.

3.1.2 Modèle

Le modèle que le programmeur utilise pour programmer sur PAM est proche du modèle idéal décrit au §1.2.2. En particulier, il considère la PAM comme un seul (grand) espace logique programmable, et non pas comme une collection de prédiffusés programmables distincts. La description complète d'un circuit dans ce modèle sera composée de trois parties:

Logique Comme pour notre définition initiale (§1.1.2), le circuit désiré est décrit comme un graphe de fonctions combinatoires et de registres – ou de façon équivalente comme un système d'équations booléennes synchrones –, à partir des primitives suivantes:

- un jeu de fonctions logiques élémentaires contenant une base de l'algèbre de Boole (on a retenu 0, 1, $\wedge$, $\vee$, $\oplus$, $\neg$ et *mux*);
- le registre synchrone;
- un certain nombre de blocs spécifiques à la technologie cible, les plus importants étant les ports d'entrée-sortie et le *buffer trois-états* (ce dernier permet de construire des entrées-sorties bidirectionnelles ainsi que quelques fonctions combinatoires très larges).

Géométrie On considère la PAM comme un espace géométrique bidimensionnel discret, en la munissant d'un système de coordonnées entières dont le grain est le PAB. La géométrie du circuit désiré devra être décrite en annotant chaque noeud du graphe logique ci-dessus par un *placement explicite*, sous la forme d'un rectangle $(x_{min}, y_{min}, x_{max}, y_{max})$ de cet espace.

Indications Le circuit obtenu est éventuellement annoté d'indications supplémentaires permettant d'en contraindre l'implémentation. On peut ainsi donner des indications de routage pour certains signaux critiques, ou spécifier la position physique que tel signal logique doit avoir sur tel bloc logique auquel il est connecté (cette précision est souvent essentielle pour la qualité de réalisation d'un circuit en technologie Xilinx).

La sémantique de ce modèle est que le circuit logique décrit sera implémenté sur la PAM de façon à ce que chaque bloc logique soit placé dans la zone rectangulaire qui lui a été assignée. Ces indications de géométrie et les indications supplémentaires éventuellement présentes sont considérées comme *impératives*: si une solution ne peut pas être trouvée qui les satisfasse toutes, la compilation échoue.

Une PAM physique donnée ne constitue pas un monde bidimensionnel homogène et isotrope parfait: les idiosyncrasies des circuits prédiffusés employés d'une part, les discontinuités introduites par les frontières entre ces circuits d'autre part, introduisent des disparités entre les quantités de ressources programmables disponibles aux différents points de la grille de PABs sous-jacente. En technologie Xilinx par exemple, une position (x, y) "standard" disposera typiquement de deux fonctions f_4 à quatre entrées et deux registres, alors qu'une position située sur la périphérie d'un prédiffusé disposera en plus de deux ports d'entrée-sortie, quatre registres et deux inverseurs. Il est de la responsabilité du programmeur travaillant sur une PAM donnée d'assurer que son circuit n'utilise en aucun point de la matrice plus de ressources que ce qui est effectivement disponible.

Notes

- La description ci-dessus est matérialisée sous la forme d'une *structure de données abstraite*, c'est-à-dire d'une bibliothèque de fonctions pour un langage de programmation donné permettant de construire le graphe désiré. On a ainsi utilisé successivement les langages Modula-2+ [RLW85], Le-Lisp [Cha86] et C++ [ES86].

- Cette structure de données a une structure *plate*: les éventuelles structurations de blocs ou de signaux présentes dans le langage source (§3.1.3) n'y sont pas représentées, ayant été complètement expansées au moment de sa création. Ce choix a été initialement fait dans un but de simplicité, mais n'est pas forcément le meilleur: l'expérience a montré qu'il serait quelquefois utile de conserver cette information dans la structure de données abstraite, de façon à pouvoir en tirer parti dans les programmes ultérieurs opérant sur cette structure (§3.1.4).

- Le "grain" à choisir pour le système de coordonnées est en fait la plus petite dimension en-dessous de laquelle la notion de placement géométrique n'a plus de sens pour le circuit prédiffusé cible. Il correspondra typiquement à la cellule élémentaire de ce prédiffusé.

- La géométrie globale de Perle-1 n'est pas réductible à une simple matrice de PABs comme l'était celle de Perle-0 (§2.2 à 2.4). Par extension du modèle ci-dessus, on considérera donc Perle-1 comme un ensemble de matrices de PABs connectées entre

elles et aux dispositifs externes. La spécification de géométrie d'un élément de circuit sera alors en fait constituée d'un couple (identificateur de matrice, rectangle de cette matrice).

- Le modèle décrit ici est plus général que l'implémentation actuelle telle qu'elle est décrite au §3.1.4 (en particulier, d'importantes restrictions sont imposées sur la spécification des rectangles de placement de façon à simplifier les algorithmes de compilation). On considère néanmoins que ce modèle simple est celui vers lequel l'implémentation doit tendre; en promouvant dès le début l'objectif final on peut donc continuer à développer les algorithmes de compilation sans avoir à redéfinir fréquemment le modèle fondamental utilisé par le programmeur.

3.1.3 Description des circuits

La structure de données abstraite introduite à la section précédente contient toute l'information permettant de décrire complètement un circuit particulier à implémenter sur la PAM. Cette section décrit différents moyens permettant de construire effectivement cette structure de données. Ces moyens sont complémentaires, en cela qu'un circuit complet pourra être obtenu par assemblage de sous-circuits ayant été indifféremment construits par l'un ou l'autre de ceux-ci.

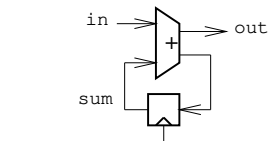
Description par langage de programmation

Le premier moyen de description de circuit consiste en l'écriture directe d'un programme, utilisant les primitives de construction de la structure de données modèle (§3.1.2), dont l'exécution génère effectivement une description complète du circuit.

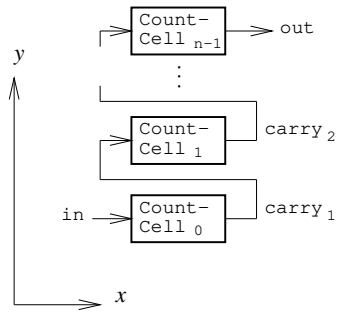
Cette méthode est la plus fondamentale, et c'est celle qui a été de loin la plus employée pour la construction des divers circuits implémentés au laboratoire DEC-PRL [BRV89, BRV93]. Afin de la faciliter et d'en simplifier l'emploi, on a en fait construit une *bibliothèque utilisateur* spécialisée à cet effet au-dessus de la bibliothèque primitive de construction de la structure de données abstraite. Cette bibliothèque utilisateur contient un nombre réduit de fonctions de plus haut niveau que la bibliothèque primitive, et est optimisée pour que l'écriture manuelle de code décrivant les circuits à implémenter soit la plus concise et la plus aisée possible. Dans un souci de simplicité maximale, son expressivité est limitée aux constructions les plus usuelles; en cas de besoin particulier, les fonctionnalités complètes de la bibliothèque primitive restent de toute façon disponibles.

Exemple La figure 3.1 montre ainsi un exemple de code Le-Lisp [Cha86] construisant le circuit "compteur modulo 2^n" introduit à la section 1.1.2. Dans cet exemple, le compteur est placé verticalement dans un rectangle de taille $1 \times n$.

Sans entrer dans le détail de cet exemple, on notera l'usage des deux fonctions fondamentales permettant de décrire la logique: `(function net eqn)` pour exprimer que le signal `net` est défini par l'équation combinatoire `eqn`, et `(register net eqn)` pour exprimer que le signal `net` est défini par l'équation combinatoire `eqn` immédiatement suivie d'un registre.




```
(defun CountCell (name in out x y)
  (let ((sum (net name)))
    (function out (AND in sum))
    (register sum (XOR in sum))
    (place out x y)
    (place sum x y)))
```




```
(defun Counter (n in out x y)
  ; Compteur n bits:
  ; in: retenue entrante
  ; out: retenue sortante
  ; (x,y): position de la cellule 0
  (let ((c (NetVector "carry" (+ n 1))))
    (vset c 0 in)
    (vset c n out)
    (for (i 0 1 (- n 1))
      (CountCell (catenate "sum" i)
                 (vref c i) (vref c (+ i 1))
                 x (+ y i)))))
```

Figure 3.1: Compteur modulo 2^n décrit en Le-Lisp

On notera d'autre part l'usage de la fonction fondamentale de description de la géométrie: `(place net x y)`, signifiant que le bloc logique qui calcule le signal `net` doit être placé à la position $(\mathtt{x}, \mathtt{y})$ de la PAM.

On remarquera enfin comment l'usage des facilités de structuration des données (vecteurs) et du code (procédures, boucles) du langage hôte permettent d'exprimer directement la régularité de la structure logique et géométrique du circuit construit.

Notes L'idée d'utiliser un langage de programmation général pour la description de circuits n'est pas nouvelle [MV85], bien qu'elle n'ait pas encore trouvé le développement industriel qu'elle mérite. Une telle approche présente plusieurs avantages importants:

- On dispose de toute la puissance algorithmique d'un langage de programmation général pour calculer des circuits complexes. A l'opposé, les langages de description de circuit spécialisés sont très rarement des langages de programmation complets, et même lorsqu'ils le sont, ne peuvent pas bénéficier du même volume d'investissement dans des bibliothèques sophistiquées.

- On bénéficie directement des facilités de structuration des données (types structurés) et du code (procédures, paramètres, structure de contrôle, récursivité) présents dans les langages de programmation généraux. La taille de la description d'un circuit devient alors proportionnelle à sa complexité réelle, et non pas à la taille du résultat: une telle approche est particulièrement bien adaptée à la description de structures régulières.

- Un programme complexe est généralement bien plus aisé à développer et à mettre au point dans un langage de programmation général que dans un langage spécialisé: le programmeur économise l'apprentissage du nouveau langage, et on peut directement bénéficier de l'important investissement qui a été fait dans les environnements de développement et les outils de mise au point traditionnels.

A l'opposé, un point négatif relativement important est le suivant:

- Pour les circuits simples devant être décrits par énumération plutôt qu'algorithmiquement, cette description par code est moins concise et moins lisible qu'une description graphique ou par langage spécialisé. C'est typiquement le cas pour la structure interne des quelques cellules élémentaires qui forment la base de tout circuit complexe, et on pourra s'en convaincre en examinant la description de la cellule élémentaire de compteur `CountCell` de la figure 3.1.

C'est en particulier pour remédier à ce dernier inconvénient que l'on a développé les autres moyens de description décrits ci-après, ainsi que la génération de dessins décrite au §3.1.4.

Description par langage spécialisé

Au cours des années d'utilisation de Perle-0 et Perle-1, on a mené plusieurs expériences d'utilisation de langages spécialisés dans la programmation de PAMs, en parallèle avec la description "native" par langage de programmation. Les plus importantes sont les suivantes:

Lustre Le langage Lustre conçu par N. Halbwachs à l'université de Grenoble est un langage déclaratif spécialisé dans la manipulation de flots de données synchrones [CH+85]. Bien qu'originalement conçu pour la description de réalisations logicielles telles qu'on en trouve dans les automates industriels, sa sémantique synchrone claire et sa concision en font un candidat naturel à la description de circuits.

En collaboration avec le laboratoire DEC-PRL, F. Rocheteau a développé une version de ce langage spécialisée dans la description d'algorithmes matériels [Roc92], essentiellement en étendant la syntaxe de façon à pouvoir spécifier des contraintes géométriques, et en introduisant des opérateurs permettant la manipulation globale de vecteurs de signaux.

On a donc implémenté les modules d'interfaces permettant d'utiliser cette version de Lustre comme moyen primaire de description de circuits pour PAMs. Plus de deux ans d'utilisation pour des applications complexes ont maintenant validé l'utilité du concept: bien que Lustre ne soit pas un langage de programmation général au sens usuel (ses structures de contrôle se limitent à l'appel de fonction et à la réplication d'éléments en tableaux), cette version permet de répondre de façon élégante et lisible à une grande majorité des besoins courants de description de circuits.

Esterel Le langage Esterel développé par G. Berry à l'Ecole des Mines de Sophia-Antipolis est un membre important de la famille des langages "réactifs" spécialisés dans la description d'automates finis [BC84]. Ce langage a lui aussi été initialement développé

pour la programmation de systèmes logiciels industriels, tels que des tableaux de bord de machines par exemple. En collaboration avec H. Touati de DEC-PRL, G. Berry a développé des techniques de compilation originales permettant d'implémenter efficacement un sous-ensemble de ce langage – "signal pur", c'est-à-dire où le seul type de données est le booléen – sur circuits logiques synchrones [Ber91]. On a donc là encore implémenté les modules d'interfaces permettant d'utiliser ce langage comme moyen primaire de description de circuits pour PAM.

A la différence fondamentale du précédent, la description de circuits en Esterel n'est pas structurelle mais comportementale. Elle permet de construire incrémentalement et hiérarchiquement la description des automates d'état des applications, alors qu'une description manuelle directe possède généralement la mauvaise propriété d'être entièrement à refaire au moindre changement de spécification.

Le revers de cette médaille est qu'il est impossible de spécifier initialement la géométrie du circuit synchrone produit, la structure et la taille de celui-ci étant généralement sans rapport avec celles du code source. Ceci semble contredire les principes énoncés au §3.1.2, mais n'est pas un inconvénient majeur en pratique vu la relative petite taille des structures concernées: l'automate de contrôle d'une application représente généralement une toute petite part de la surface totale de la PAM (bien que sa mise au point soit bien souvent une des tâches les plus complexes du développement); en accord avec le modèle initial, on peut donc réserver dans la structure géométrique globale de l'application une zone de taille suffisante dans laquelle seront placées toutes les cellules logiques produites par le compilateur Esterel; la géométrie finale à l'intérieur de ce sous-ensemble sera déterminée par les outils de placement automatique, mais l'ensemble est généralement suffisamment petit pour que le résultat produit soit acceptable, tout du moins si on ne vise pas des niveaux extrêmes de performance.

Cette approche est toujours aujourd'hui en développement actif, mais paraît prometteuse au vu des premières expériences. Un langage du type d'Esterel semble rendre beaucoup plus accessible la conception d'automates de contrôle complexes; on profite par ailleurs immédiatement d'un environnement de développement maintenant bien mûr, offrant des outils puissants de mise au point ainsi que des possibilités de vérification formelle.

2Z F. Bourdoncle et J. Vuillemin terminent actuellement la conception d'un nouveau langage de spécification de circuits synchrones baptisé *2Z* [BV93]. Celui-ci incorpore à la fois les résultats de l'expérience menée avec le langage Lustre décrit ci-dessus, et les techniques de synthèse de circuits arithmétiques décrites dans [Vui93]. Il devrait devenir le langage principal de description de circuits en usage à DEC-PRL.

Les processus de compilation de tous ces langages s'appuient bien entendu sur les primitives de construction de la structure de données modèle du §3.1.2 pour la description des circuits à réaliser. Ils peuvent donc aisément interagir entre eux et tirer immédiatement parti des algorithmes d'implémentation sur PAM décrits plus loin (§3.1.4).

Description graphique

Le moyen traditionnel universellement employé dans le monde de l'électronique pour la description des circuits est le *schéma*, c'est-à-dire un graphique similaire à tous ceux qui illustrent cette thèse. Par rapport aux moyens algorithmiques (langages de programmation) ou textuels (langages spécialisés) précédemment considérés, celui-ci possède les avantages et inconvénients suivants:

- il est plus limité dans ses structures de contrôle, la seule réellement disponible étant la hiérarchisation des blocs et des signaux (c'est-à-dire l'équivalent des procédures d'une part, des structures de données "tableau" et "enregistrement" d'autre part);
- sa mise en œuvre est souvent pénalisée par le manque de disponibilité de logiciels d'édition suffisamment simples, versatiles et ergonomiques; la plupart des éditeurs de schémas sont effectivement incorporés dans des logiciels intégrés complexes (et chers) spécialisés dans le développement de circuits imprimés ou de circuits intégrés;
- par contre, ce moyen possède l'avantage essentiel et incontestable d'une bien meilleure *lisibilité*, tout au moins en ce qui concerne les descriptions de type énumératives[1].

A titre expérimental, on a réalisé les bibliothèques et modules d'interface permettant d'utiliser un logiciel d'édition de schémas disponible par ailleurs à DEC-PRL (un des outils du progiciel commercial utilisé pour le développement des PAMs elles-même) comme moyen de description de circuits sur PAM. L'utilisation de ce système pour la description de quelques applications programmées sur Perle-0 a validé le concept (bien meilleure lisibilité, plus de duplication manuelle entre la description textuelle et la documentation schématique) mais mis en apparence les inconvénients mentionnés plus haut (mal adapté tel quel à la description de structures régulières, par manque de structures de contrôle "algorithmiques"). Cette expérience devait de toute façon être interrompue par la lourdeur et la faible disponibilité de l'outil d'édition employé.

Un tel outil utilisé dans de bonnes conditions permettrait néanmoins certainement un gain de confort, et peut-être de productivité; à notre opinion, les facteurs essentiels qu'il convient de rassembler sont les suivants:

- un bon éditeur de schémas, simple et versatile, ne faisant pas d'hypothèse sémantique sur les structures manipulées au-delà de ce qui est strictement nécessaire, aisément disponible sur toute une gamme de machines (à la date de rédaction, l'auteur n'a pas connaissance d'un tel oiseau rare);
- un programme de traduction de ces schémas dans la structure de données commune utilisée par le reste des outils de description, si possible à travers l'un des autres langages utilisés, de telle sorte qu'on puisse aisément faire interagir description textuelle et description schématique; on peut ainsi n'utiliser la description schématique que pour les parties du circuit pour lesquelles elle apporte un gain réel;

[1] Cet avantage est tel qu'on trouvera fréquemment des descriptions textuelles de circuits documentées par des schémas annexes exprimant *exactement* la même information.

- l'extension de la syntaxe reconnue dans les schémas de façon à y introduire des primitives algorithmiques; bien qu'étant toujours du domaine de la recherche, de telles extensions ont maintenant été réalisées plusieurs fois avec succès, et apportent un gain extrêmement notable de puissance d'expression tout en préservant les caractéristiques naturelles de lisibilité: on se réfèrera notamment à [DM92] pour la description d'un système mêlant librement schémas et code C++.

Le développement d'un outil suivant ces directives est prévu dans la réécriture de l'environnement de programmation pour PAM qui a été entreprise à DEC-PRL. Il est néanmoins subordonné à la disponibilité (ou au développement préalable) d'un logiciel d'édition adapté

3.1.4 Implémentation sur PAM

Cette section décrit les principaux algorithmes mis en œuvre pour réaliser effectivement sur PAM un circuit décrit au moyen des techniques présentées ci-dessus. Rappelons qu'en vertu des principes énoncés au §3.1.1, on veut une compilation *entièrement automatique* transformant la structure de données contenant l'information "source" définie par le programmeur, jusqu'au fichier final de configuration de la PAM.

Les principales étapes de cette compilation sont les suivantes[2]:

Mapping On utilise l'information de placement contenue dans la structure source pour transformer le graphe de portes logiques et de registres entré par le programmeur en un graphe ne contenant plus que des cellules élémentaires du prédiffusé programmable (CLB).

Le modèle décrit au §3.1.2 spécifie que chaque bloc logique constituant le circuit source doit être placé par le programmeur dans un rectangle donné de l'espace offert par la PAM. Dans cette première implémentation, on a en fait imposé une restriction très importante sur ces directives de placement, demandant qu'elles soient de l'un des deux types suivants uniquement:

- un rectangle de taille 1×1 correspondant en fait à un placement exact, ou
- un rectangle correspondant exactement à la taille et à la position d'un des prédiffusés programmables composant la PAM.

Sous ces restrictions, l'implémentation de cette phase devient simple:

- par une première passe sur le circuit, on classe les primitives logiques placées exactement en fonction de leur position (x, y);
- pour chacune des positions ainsi trouvées, on tente de construire un CLB implémentant l'ensemble de la logique placée par le programmeur à cette position[3]; si cette construction échoue, le circuit demandé n'est pas implémentable;

[2] Dans toute la suite, on utilisera les acronymes standard de la terminologie Xilinx, soit *LCA* (Logic Cell Array) pour désigner un circuit prédiffusé et *CLB* (Configurable Logic Block) pour désigner une cellule élémentaire de ce circuit.

[3] M. Shand et M. Skubiszewski ont contribué à l'implémentation fine de cet algorithme de construction, qui est assez délicate dans le cas particulier des cellules élémentaires Xilinx XC3000.

- les ressources logiques qui ne sont pas placées exactement sont simplement classées en fonction du LCA auquel elles ont été assignées; leur conversion en CLBs sera faite par les outils automatiques fournis par Xilinx (voir plus loin).

Notons que cette restriction, quoique importante, n'a pas été très handicapante pour le style de circuits réalisés jusqu'ici: il est en effet très courant que le programmeur veuille placer exactement l'ensemble des ressources logiques constituant le chemin de données de son application, qui constituent l'essentiel du volume de logique et pour lesquelles il conçoit généralement une géométrie régulière précise; au contraire, les structures annexes (automate de contrôle généré par Esterel, chemins d'entrées-sorties) sont généralement peu importantes et sans structure régulière apparente, et peuvent être efficacement placées par un outil automatique.

La levée quasi-totale de cette restriction constitue cependant un des objectifs de la réécriture de ces outils qui a été amorcée. Elle permet de ne plus se soucier de la structure fine des cellules Xilinx, et de travailler par allocation de la surface disponible (*floorplanning*). L'implémentation en devient néanmoins beaucoup plus complexe, demandant en particulier l'intervention d'algorithmes de minimisation logique.

Partitionnement en circuits Le circuit logique synchrone obtenu après la phase ci-dessus est partitionné en autant de sous-circuits qu'il y a de prédiffusés programmables sous-jacents. L'identification du LCA dans lequel placer une ressource logique donnée est immédiate à partir des indications de placement contenues dans la structure source, compte tenu de la restriction énoncée au point "Mapping" ci-dessus. Le travail à effectuer pour cette phase consiste donc essentiellement à:

- Ajouter les blocs d'entrée-sortie nécessaires à chaque signal logique traversant une frontière physique de circuit (entrées, sorties ou bidirectionnels);
- Incorporer les modules standard d'interconnexion avec les périphériques de la PAM (horloge, mémoires, ...), d'après une bibliothèque.

Routage inter-circuits On doit ensuite affecter les blocs d'entrée-sortie introduits ci-dessus aux canaux de communication inter-LCAs disponibles sur la PAM. Il s'agit là d'une phase de *routage global* au niveau de la PAM toute entière, guidée par les éventuelles indications du programmeur.

On a pour cela utilisé un algorithme simple et direct (*greedy*) qui s'est révélé bien adapté en pratique:

- les signaux dont le routage a été explicitement spécifié par le programmeur (à travers des annotations spécifiques) sont affectés en premier; si cette affectation n'est pas possible, cette phase échoue;
- les signaux inter-circuits restant sont routés sur une grille régulière dont la granularité est le LCA, en fonction des canaux de connexion restant disponibles, et en privilégiant l'utilisation des ressources de communication globales de la PAM pour ceux d'entre eux qui vont à une distance strictement supérieure à 1; le résultat de cette phase est une liste de signaux à router à travers chaque frontière inter-LCA;

- pour chacune des frontières ci-dessus, l'affectation exacte des signaux aux canaux de communication est faite au moyen d'un algorithme de minimisation des distances à l'intérieur de chaque circuit, en tenant compte de la position précise des points de connexion à ces signaux dans les LCAs; on prend soin d'assurer que les configurations régulières produisent des solutions régulières.

Génération des fichiers intermédiaires On génère les fichiers intermédiaires suivants, permettant aux outils spécifiques du fabricant [Xil91] de terminer la compilation:

- un fichier de description logique pour chaque circuit prédiffusé programmable, dans la syntaxe spécifique au fabricant, contenant les équations à implémenter dans ce circuit;
- un fichier de directives de placement et de routage pour chaque prédiffusé programmable, dans la syntaxe spécifique au fabricant, contenant les indications données par l'utilisateur ainsi que celles qui proviennent des phases de compilation ci-dessus;
- un fichier d'"automatisation de compilation", dans la syntaxe du système d'exploitation hôte (on utilise ici l'utilitaire `make(1)` d'Unix), permettant d'enchaîner les phases finales de compilation sans intervention de l'utilisateur et de filtrer les éventuels messages d'erreurs que celles-ci produisent; optionnellement, on fournit la possibilité d'utiliser plusieurs machines en parallèle pour ces phases finales, qui sont typiquement les plus lourdes en temps de compilation (au plus, on peut utiliser une machine différente par prédiffusé programmable).

Additionnellement, on génère les fichiers suivants destinés à la mise au point:

- un fichier de "symboles" permettant la mise au point symbolique (cf. §3.2.3); celui-ci contient la localisation précise qui a été assignée à chacun des objets nommés apparaissant dans la structure source;
- un fichier de "dessins" (description graphique bidimensionnelle de la structure géométrique obtenue) pouvant être imprimé ou inspecté à l'écran; ceci fournit un outil essentiel d'aide à la mise au point des indications géométriques contenues dans la structure source[4].

Phase finale Sous contrôle du fichier de commande mentionné ci-dessus, on utilise les outils spécifiques à la technologie fournis par le fabricant pour effectuer les phases finales de la compilation (de façon indépendante pour chaque LCA):

- terminer le *mapping* (fabriquer des CLBs pour la logique non placée par le programmeur);
- terminer le placement (placer exactement les blocs logiques non placés par le programmeur et ceux issus de la phase précédente);
- effectuer le routage détaillé à l'intérieur de chaque LCA;

[4] M. Skubiszewski est l'auteur de l'ingénieux algorithme de mise en page utilisé.

- générer les fichiers de configuration finaux de chaque LCA.

Enfin, à la fin de tout ceci: ces fichiers de configuration individuels sont repris par un programme final qui les assemble en un fichier de configuration de la PAM toute entière; on extrait l'information de placement final des blocs placés automatiquement de façon à compléter le fichier de symboles.

Notes

- Les phases de calcul décrites ci-dessus constituant un ensemble assez lourd (surtout en ce qui concerne les outils Xilinx utilisés en phase finale), on essaie à chaque étape de faire une analyse de validité du circuit aussi fine que possible de façon à détecter les erreurs potentielles le plus tôt possible; on minimise ainsi la durée du cycle édition–compilation–mise au point dans les phases initiales de développement, mais surtout ceci permet de présenter au programmeur un diagnostic d'erreur aussi précis que possible.

- Les outils Xilinx utilisés en phase finale étant lourds en temps de calcul (particulièrement en ce qui concerne la phase de placement automatique) mais travaillant indépendamment au niveau du LCA, on a été amené à développer un filtre permettant de reconnaître qu'un LCA donné n'a pas été modifié par rapport à la version précédente (même si par exemple un renommage global des signaux a eu lieu). La recompilation suivant une modification locale d'un circuit en développement s'en trouve ainsi considérablement accélérée (quelques minutes au lieu de plusieurs dizaines de minutes).

- Cette séparation nette entre les outils de description des circuits et les outils d'implémentation permet d'expérimenter aisément des outils nouveaux travaillant soit en amont soit en parallèle de la chaîne de compilation décrite. H. Touati a ainsi développé une passerelle permettant d'appliquer des outils de minimisation logique [BT93] à une partie du circuit, ce qui est essentiel à l'implémentation efficace des automates générés par le compilateur Esterel; l'auteur a implémenté un traducteur inverse de circuits synchrones en Esterel permettant de prouver formellement la correction de la traduction d'Esterel en circuit; N. Shenoy a implémenté un algorithme expérimental de *retiming* automatique de circuits synchrones préservant la géométrie; etc...

- Le système décrit ci-dessus, bien que très simple, s'est avéré tout à fait utilisable en pratique. Ses principales limitations sont néanmoins les suivantes:

  – Le plus important maillon manquant est un algorithme de placement automatique simple et direct (*greedy*) pouvant prendre en charge la géométrie des structures régulières au niveau de la PAM toute entière. Il devrait être possible de ne spécifier dans le source qu'une allocation globale de surface (*floorplanning*) et les indications en nombre minimal permettant de contraindre l'implémentation à choisir la solution voulue (par exemple en spécifiant la position des pattes d'entrée et de sortie d'un chemin de données parfaitement régulier); on ferait ainsi l'économie d'un travail fastidieux et source d'erreurs.

- Les principaux autres outils manquants sont d'une part un analyseur de délai statique travaillant au niveau de la PAM toute entière et intégrant la base de données des temps de commutation des différentes liaisons inter-LCAs, d'autre part un simulateur global travaillant lui aussi au niveau de la PAM toute entière et intégrant des modèles de ses périphériques standard (mémoire, FIFOs, générateur d'horloge, etc...). Ce dernier outil est en cours de réalisation dans le cadre de la réécriture entreprise au laboratoire DEC-PRL [Tou93].
- La structure de données source devrait pouvoir être hiérarchisée au niveau des signaux et des blocs logiques: d'une part, cela permettrait d'implémenter plus aisément l'allocation de surface décrite ci-dessus, d'autre part on pourrait envisager, lors de la compilation ou de la simulation, de substituer des implémentations pré-optimisées à des structures logiques particulières sans que ces structures aient à être reconnues par tous les programmes travaillant sur le graphe source.

3.2 Programmation logicielle

On s'intéresse ici aux outils et à l'environnement de développement permettant d'écrire et de mettre au point la partie logicielle d'une application implémentée sur PAM, c'est-à-dire le programme s'exécutant sur l'hôte et pilotant une configuration matérielle particulière de la PAM.

Il s'agit en fait essentiellement de développement logiciel conventionnel, pour lequel on peut utiliser les techniques standard. On n'abordera ici que les points particuliers spécifiques à la nature PAM de l'application.

3.2.1 Objectifs

L'objectif premier est la fourniture de bibliothèques offrant les fonctionnalités essentielles à l'interfaçage entre un programme logiciel conventionnel et le dispositif matériel qu'est une PAM connectée à la machine hôte, soit principalement la configuration, la communication (transferts de données) et quelques procédures de gestion diverses (pilotage du générateur d'horloge (§2.5.3), relecture de l'état (§2.5.2), etc...).

Ces bibliothèques sont fournies pour le ou les langages de programmation que l'on prévoit d'utiliser pour l'écriture des programmes pilotant la PAM. Dans le cas présent, les langages supportés sont respectivement Le-Lisp [Cha86] et C [ISO90] pour Perle-0, C et C++ [ES86] pour Perle-1.

Dans l'écriture de ces bibliothèques, on a principalement poursuivi les objectifs suivants:

Performance Ainsi que c'était déjà le cas pour les outils d'aide à la programmation matérielle (§3.1.1), l'objectif prioritaire est de permettre l'obtention des performances maximales pour une application donnée.

Sécurité Le développement des programmes de pilotage de dispositifs matériels, utilisant la machine hôte à très bas niveau, peut être intrinsèquement dangereux par la portée

des conséquences potentielles des erreurs de programmation (cf. note 5, page 97). On s'est attaché à fournir un bon niveau de protection contre les erreurs pouvant menacer l'intégrité physique de la PAM elle-même d'une part, l'intégrité des données portées par la machine hôte d'autre part.

Facilité d'utilisation On s'est efforcé de rendre aussi simple que possible le développement des applications "standard", c'est-à-dire de celles qui ne font appel qu'au sous-ensemble le plus courant des fonctionnalités disponibles.

Mise au point On fournit un certain nombre d'outils d'aide à la mise au point (§3.2.3), permettant d'une part d'aider au développement de la partie logicielle de l'application, d'autre part d'exploiter les informations d'aide à la mise au point de la partie matérielle générées par les outils décrits précédemment (§3.1.4).

3.2.2 Bibliothèque d'exécution

Les fonctionnalités nécessaires à l'utilisation de la PAM depuis le logiciel s'exécutant sur l'hôte sont regroupées en trois catégories selon leur nature, et fournies à travers trois interfaces de programmation distinctes. Elle sont citées ci-après en ordre hiérarchique, c'est-à-dire que chacune est implémentée au moyen des précédentes:

1. Un *gestionnaire de périphériques* spécifique intégré dans le noyau du système d'exploitation (en anglais *device-driver*; on ne s'est ici intéressé qu'aux systèmes d'exploitation de la famille Unix). Cette interface contrôlée fournit les fonctionnalités les plus critiques au niveau de la sécurité et de l'intégrité de la machine hôte (essentiellement l'accès physique à la PAM).

2. Une *interface de bas niveau*, offrant l'accès à l'intégralité des fonctionnalités et des performances offertes par la PAM considérée. Cette interface est implémentée avec le minimum de surcoût possible, et fait le minimum d'hypothèses sur la nature et les besoins des programmes qui l'utiliseront. En particulier les considérations de sécurité y sont systématiquement secondaires aux considérations de performances et de fonctionnalité.

3. Une *interface de haut niveau*, offrant seulement le sous-ensemble minimum de fonctionnalités utiles aux applications "standard", avec une sécurité en principe totale face aux erreurs de programmation (dans le sens suivant: une application n'utilisant que la bibliothèque de haut niveau ne peut pas menacer l'intégrité physique de la PAM ou de la machine hôte, ni celle des données portées par la machine hôte). L'objectif principal est la simplicité d'utilisation; l'implémentation offre le niveau de performances maximal compatible avec la notion de sécurité définie ci-dessus.

L'intention est bien entendu que les programmes d'application n'utilisent que l'interface haut-niveau; là où c'est impossible pour des considérations de performances ou de fonctionnalité particulières, le programmeur peut construire lui-même la ou les fonctions de haut niveau qui lui manquent en utilisant la bibliothèque bas-niveau; il prend alors lui-même la responsabilité de la sécurité de ces nouvelles fonctions.

Bibliothèque haut-niveau

La suite de cette section décrit les grandes lignes des fonctionnalités présentées par l'interface haut-niveau de Perle-1, à titre d'exemple représentatif de ce que doit typiquement comporter la bibliothèque standard d'utilisation d'une PAM.

Initialisation Une première fonction (dite d'"ouverture") permet d'établir le lien entre le programme et une PAM donnée présente sur la machine hôte. Elle supporte une notion d'*accès exclusif* permettant de synchroniser l'accès à une même PAM depuis plusieurs processus logiciels:

- par défaut, une application sera bloquée lors de l'ouverture d'une PAM si cette PAM est déjà utilisée par un autre processus; son exécution reprendra dès que cet autre processus aura terminé;
- si ce comportement ne convient pas, une application peut demander à ce que l'ouverture échoue immédiatement si la PAM est déjà en usage, ou au contraire à ouvrir la PAM malgré tout (ce dernier mode étant utile pour certaines fonctions de mise au point).

Ce schéma simple suffit amplement pour l'usage recherché (quelques personnes partageant une même PAM pour le développement d'applications). Il demanderait néanmoins certainement à être raffiné pour un usage plus intensif ou plus critique.

Il est bien entendu possible à un même programme d'ouvrir plusieurs PAMs simultanément et de les utiliser en parallèle, si celles-ci sont disponibles sur la machine hôte. A cet effet, la fonctionnalité standard offerte par la bibliothèque d'utilisation de Perle-0 était en fait l'ouverture de n'importe laquelle des PAMs couramment inutilisée présente sur la machine, et seulement lorsqu'il était nécessaire d'adresser une PAM particulière (pour y chercher un problème matériel, ou parce qu'un dispositif externe y avait été connecté) devait-on préciser celle qui était concernée. Ce mode n'a pas encore été réimplémenté pour Perle-1 car les machines hôtes potentielles n'avaient jusqu'à récemment pas une grande probabilité d'accueillir plusieurs PAMs.

Configuration Une seconde fonction permet de *configurer* la PAM ainsi ouverte pour une application donnée, en fonction de deux paramètres: le fichier de configuration de la PAM elle-même d'une part (tel qu'il résulte de la phase de compilation décrite au §3.1), la fréquence de fonctionnement du générateur d'horloge d'autre part.

Notes:

- Cette fonction, étant typiquement la première action physique effectuée par le programme d'exploitation sur la PAM, ne fait *aucune hypothèse* sur l'état de cette dernière: elle la réinitialise complètement, la configure comme désiré, et la laisse dans un état initial standard complètement déterminé (sauf en ce qui concerne l'état des mémoires, voir ci-dessous). Ceci permet de ne pas se soucier de l'état dans lequel une application doit laisser une PAM après exécution ou en cas d'erreur.

- Ce processus de configuration *préserve l'état des mémoires*. Ceci permet d'enchaîner plusieurs phases de traitement successives au sein d'une même application (§2.5.1).
- Cette fonction est très soigneusement *optimisée en temps d'exécution*: ceci maximise le nombre d'applications pour lesquelles il est possible d'utiliser plusieurs configurations différentes qui se succèdent (§2.5.1). En particulier il a été ici nécessaire de paralléliser le processus de modification de la fréquence d'horloge (la technologie des boucles à verrouillage de phase utilisée par Perle-1 est intrinsèquement lente) et la configuration de la PAM proprement dite.
- Cette fonction vérifie l'intégrité des fichiers de configuration chargés en calculant une *signature* de ces fichiers au cours de la configuration. Ceci rend infinitésimale la probabilité que soit chargé dans la PAM un vecteur de bits aléatoire non issu de la phase de compilation décrite au §3.1, à la suite d'une erreur matérielle ou logicielle (une configuration aléatoire a toute chance de *détruire physiquement* la PAM, par le nombre massif de court-circuits internes qu'elle peut comporter[5]). Cette signature est en fait calculée par la logique de configuration de la PAM elle-même, de façon à ne pas pénaliser le temps de configuration.

Transferts de données Les fonctions les plus importantes sont ensuite celles qui permettent le transfert de données entre la PAM et son application de pilotage, en écrivant dans le FIFO d'entrée ou en relisant le FIFO de sortie (§2.5.4).

On fournit en fait deux jeux indépendants de telles fonctions, permettant d'une part de transférer atomiquement un mot (en utilisant les instructions d'entrée-sortie natives du processeur), d'autre part de transférer un bloc de mots consécutifs de taille arbitraire (en utilisant les opérations d'"accès direct mémoire" de la machine hôte). Ces deux modes sont complémentaires, le premier offrant une flexibilité maximale et le second une bande passante maximale dans le cas particulier des transferts de blocs de grande taille.

Notes:

- La bande passante des transferts de données entre la PAM et son logiciel de pilotage étant très souvent un facteur contraignant des applications, ces fonctions ont été très soigneusement optimisées pour la performance.
- Une fonction d'interrogation par défaut de l'état des FIFOs est fournie pour que le programmeur utilisant les fonctions de transfert de mots puisse effectuer le contrôle de flux nécessaire (en utilisant typiquement un algorithme du type similaire à celui qui est présenté page 67). Ce contrôle de flux est effectué automatiquement par les fonctions de transfert de blocs. Les erreurs éventuelles de débordement des FIFOs (écriture d'un FIFO plein ou lecture d'un FIFO vide) sont reportées à l'application sans surcoût de performance.
- Les applications demandant la plus grande bande passante d'entrée-sortie (par exemple la manipulation de flux audio ou vidéo en temps réel) ont fréquemment

[5] Cette fonctionnalité essentielle a en fait été introduite *après* qu'une telle catastrophe – initialement considérée comme très improbable – se soit effectivement produite au laboratoire DEC-PRL.

besoin d'utiliser des techniques de double tampon (*double-buffering*) pour soutenir la cadence de transfert. Pour cela, les fonctions de transfert de blocs peuvent être utilisées en *mode asynchrone*, ce qui permet par exemple d'écrire un tampon de résultats sur disque ou sur l'écran pendant que la PAM remplit le suivant.

Contrôle Un certain nombre de fonctions annexes donnent accès aux fonctions les plus courantes de gestion de la PAM:

- réinitialisation de l'état de l'application embarquée dans la PAM ou de celui des FIFOs;
- lecture ou écriture du port de communication asynchrone ("LCBus", cf. §2.5.4);
- démarrage ou arrêt du générateur d'horloge;
- autorisation des interruptions (voir ci-dessous).

Interruptions La partie matérielle de l'application (embarquée dans la PAM) peut interrompre la partie logicielle pour signaler la fin d'un calcul ou une circonstance exceptionnelle (§2.5.4). La bibliothèque standard permet d'enregistrer une procédure qui sera appelée de façon asynchrone par cette interruption. Les interruptions déclenchées par la PAM pour une autre cause (erreurs de FIFOs, de DMA, etc. . .) sont transformées en erreurs.

Mise au point Un certain nombre de fonctionnalités permettent la mise au point de la partie matérielle ou de la partie logicielle de l'application. Elle font l'object de la section suivante (§3.2.3).

3.2.3 Mise au point

Capture de traces d'exécution

Un des outils essentiels de mise au point de la partie matérielle d'une application, ou de l'interaction du couple matériel–logiciel, est l'exécution en mode "trace", c'est-à-dire avec capture périodique de l'état pour analyse ultérieure. Cette exécution peut prendre plusieurs formes:

- l'exécution simulée, où la partie matérielle de la PAM est en fait remplacée par un simulateur fonctionnel (rappelons que les outils de compilation de la configuration de Perle-1 permettent d'obtenir automatiquement un tel simulateur à partir de la description source [Tou93]);
- l'exécution en *pas-à-pas*, où l'on capture l'état complet de la PAM après chaque cycle d'horloge par *readback* (§2.5.2);
- l'exécution en *double-pas* utilisée pour rechercher et caractériser le chemin critique des applications PAM, où l'on envoie des paires de coups d'horloge en succession de plus en plus rapide, et où l'on capture l'état de la PAM après chaque paire pour le comparer à une séquence de référence (§1.2.6).

Dans tous les cas, ces modes d'exécution demandent une version spécialisée du programme de pilotage de l'application, obtenue à partir de la version originale en insérant aux endroits judicieux des ordres de capture d'état ou d'avancement de l'horloge.

Il s'est avéré extrêmement malcommode de développer ces versions spécialisées, selon le principe général qu'il est malsain d'avoir à modifier extensivement un objet en cours de développement de façon à pouvoir le mettre au point. La bibliothèque standard d'exécution de Perle-1 incorpore donc les plus communes de ces fonctionnalités, sous une forme telle qu'*aucune modification du code source* ne soit nécessaire pour les utiliser. Ceci est obtenu en fournissant des implémentations alternatives des fonctions de l'interface de haut niveau, offrant les fonctionnalités suivantes:

- l'exécution utilisant un simulateur au lieu de la PAM réelle;
- l'exécution en mode pas-à-pas ou double-pas sur la PAM réelle, à diverses fréquences d'horloge dans le deuxième cas;
- la capture optionnelle de l'état interne (que ce soit de la PAM réelle ou du simulateur) pendant un certain intervalle de cycles;
- l'exécution à vitesse normale mais avec toutes les vérifications de cohérence en service (celles de ces vérifications qui ont un coût à l'exécution ne sont normalement pas présentes dans la bibliothèque standard; ceci est analogue au mode "vérification des indices de tableaux" d'un compilateur Pascal par exemple).

La plupart de ces fonctionnalités sont immédiates à implémenter. L'une d'entre elles cependant est intéressante: il s'agit du choix du moment auquel faire avancer l'horloge de la partie matérielle de l'application (que ce soit pour les modes pas-à-pas, double-pas ou simulation). On utilise pour cela le fait que l'interface entre le logiciel et la PAM est spécifiée comme étant *asynchrone*. En particulier, selon ce modèle, tout entrelacement des cycles d'horloge du processeur hôte et de la PAM est légal, pourvu que le nombre maximal de cycles de l'une contenus dans un cycle de l'autre soit borné. On peut mettre cette observation à contribution en implémentant deux de ces modes, qui sont aisés à réaliser:

PAM infiniment rapide: après chaque accès du processeur hôte dans la PAM (c'est-à-dire chaque appel d'une fonction de la librairie de haut niveau), l'horloge de la PAM est cyclée jusqu'à ce que celle-ci se bloque (note: ce mode n'est implémentable tel quel qu'avec une application utilisant le mode "Autostop" décrit au §2.5.4, car autrement il n'existe pas de moyen simple de déterminer que le calcul en cours dans la PAM est bloqué en attente de l'hôte; dans les autres cas on peut mettre une limite supérieure au nombre de cycles à effectuer);

PAM infiniment lente: l'horloge de la PAM est avancée de un cycle à chaque fois que l'hôte effectue un *rendez-vous* avec la PAM, c'est-à-dire ici à chaque transaction de lecture synchrone de l'état de la PAM.

On a implémenté le deuxième de ces modes pour Perle-1, l'horloge de la PAM (ou du simulateur) n'étant avancée que par deux des fonctions de l'interface haut-niveau: `GetFifoFlags` (interrogation par défaut de l'état des FIFOs) et `GetLCBus` (lecture de l'état du port de communication asynchrone). Il est à noter que la présence du FIFO de sortie fait que la lecture d'un mot de donnée n'est pas une opération de synchronisation entre les deux horloges.

Il serait probablement intéressant d'implémenter aussi le premier de ces modes (PAM infiniment rapide). Bien que le deuxième suffise à générer des traces d'exécution valides permettant la mise au point, la disponibilité de deux modes d'exécution radicalement différents permettrait de détecter plus aisément les logiciels d'application qui ne sont pas complètement corrects vis-à-vis de l'hypothèse d'asynchronisme, et risquent d'être sujets à un faible taux d'erreurs aléatoires extrèmement difficiles à identifier (il est par exemple relativement fréquent de trouver dans les applications des constructions du genre "écrire une donnée; attendre un peu; relire le résultat sans vérifier car il doit être prêt depuis longtemps").

Examen des traces d'exécution

La chaîne de compilation des circuits matériels décrite au §3.1.4 fournit entre autres résultats un fichier de "symboles" permettant d'associer à chaque bit d'état de la PAM le nom que le programmeur lui a donné dans la description source. Les fonctions de lecture et d'analyse de cette table de symboles sont présentes dans la bibliothèque d'exécution, de façon à ce que lors de la phase de mise au point, le programmeur puisse à tout moment relire l'état de la PAM, retrouver les bits particuliers qui l'intéressent d'après leur nom symbolique et en suivre l'évolution (si par exemple l'application possède un chemin de données de 32 bits de large, il est probable que l'on va vouloir relire les 32 bits d'un même registre et les recombiner en une valeur entière avant de vérifier leur évolution).

La présence de ces fonctions dans la bibliothèque permet au programmeur de disposer de la flexibilité maximale, lui offrant la possibilité de développer des programmes complexes spécialisés pour la vérification de son application, tout en restant au même niveau d'abstraction que la description source de cette application. Néanmoins, pour répondre aux usages courants sans avoir à écrire de code spécialisé, on a trouvé utile de développer un certain nombre d'outils permettant d'inspecter aisément les traces d'exécution capturées par les fonctions de la section précédente:

- H. Touati a adapté à Perle-1 un outil d'inspection de traces de simulation de circuits intégrés baptisé `Krono`, originellement développé par L. Monier et S. Karsenty. Cet outil permet d'observer des signaux ou des ensembles de signaux sous une forme identique à celle qui est usuellement présentée sur les analyseurs logiques, en utilisant les noms symboliques des signaux pour les référencer.

- M. Shand a développé un outil original pour analyser les traces d'exécution de Perle-1. Baptisé `showRB`, celui-ci permet d'afficher graphiquement sur l'écran l'état complet de la PAM à un instant donné, sous une géométrie similaire à celle du modèle utilisé lors de la conception, et de faire évoluer cet affichage dans le temps avec une présentation inspirée de celle d'un magnétoscope. Dérivé d'un prototype originellement conçu par

l'auteur et J. Vuillemin à des fins uniquement démonstratives, cet outil s'est révélé étonnament utile à la recherche des défauts de fonctionnement des applications, l'œil s'avérant être capable de détecter aisément des défauts de régularité dans la structure ou l'évolution d'une image représentant pourtant plus de 18 000 bits. M. Shand a depuis lors augmenté les capacités de cet outil d'une facilité de comparaison de deux traces d'exécution, qui s'est avéré être un outil efficace d'aide à la recherche des chemins critiques des applications par la méthode du *double-pas*[6].

L'expérience de plusieurs années d'utilisation a prouvé que les outils ci-dessus, bien que simples, étaient bien adaptés à l'aide à la mise au point des applications PAM, les points essentiels étant la disponibilité de l'information de mise au point au même niveau d'abstraction que la description source, l'interactivité permettant une recherche rapide des problèmes les plus courants et une programmabilité complète permettant de développer des outils spécialisés sophistiqués pour les applications complexes. Leur développement a d'ailleurs été largement inspiré des facilités de mise au point symbolique qui sont aujourd'hui présentes dans tous les environnements de développement logiciel classiques.

Travaux apparentés

Les concepteurs de la machine Splash [GHK91] décrivent un système d'inspiration similaire dans [GK+90]. Ce dernier permet la mise au point symbolique des applications, une certaine interactivité et la programmabilité; il n'offre cependant pas les fonctionnalités de visualisation géométrique ni de capture automatique de traces d'exécution qui ont été présentées ici.

[6] M. Shand est par ailleurs l'inspirateur initial et le premier implémenteur de cette méthode *double-pas*.

Chapitre 4

Conclusions et perspectives

Ce dernier chapitre rassemble un certain nombre de réflexions – plus ou moins formalisées – issues de l'expérience acquise au cours de cette étude. On aborde d'abord brièvement la question de l'applicabilité de cette technologie. On examine ensuite plus en profondeur les perspectives que celle-ci semble offrir, en passant en revue un certain nombre de problèmes ouverts dans les deux domaines complémentaires que sont l'architecture et la programmation des PAMs.

Domaines d'application

On présente en annexe deux articles précédemment publiés qui rassemblent les principaux résultats obtenus après respectivement une et trois années d'avancement de ce projet. On avait alors l'objectif initial d'explorer des domaines aussi variés que possible, afin d'identifier les champs d'applications les plus immédiatement prometteurs de cette nouvelle technologie; ces résultats montrent ainsi un certain nombre de problèmes pour lesquels on a pu apporter une solution un ordre de grandeur meilleure que ce qui existait préalablement en termes de rapport performance-coût.

Pour chaque problème individuel, on aurait bien entendu pu construire un dispositif matériel spécifique apportant une solution au moins équivalente, pour un coût unitaire de réalisation probablement moindre que celui de la PAM elle-même. L'intérêt essentiel de la technologie PAM repose donc sur sa versatilité, faisant qu'un dispositif matériel unique – donc préalablement construit et pour la conception et la fabrication duquel on bénéficie de l'économie d'échelle – peut répondre de façon satisfaisante à toute une famille de problèmes qu'on ne saurait autrement résoudre qu'à travers des développements matériels spécifiques.

Il est important de tenter d'identifier les caractéristiques communes de ces différentes classes d'applications potentielles. En première analyse, les applications considérées lors de cette phase initiale peuvent être regroupées en un petit nombre de familles:

Calcul: une première catégorie de besoins pourrait être regroupés sous le terme d'*algorithmiques*. Les caractéristiques communes en sont les suivantes:

- un fort besoin en puissance de calcul;

- des algorithmes simples, mais mal adaptés à l'architecture des processeurs conventionnels;
- des algorithmes suffisamment peu stables, trop nouveaux ou d'usage pas assez fréquent pour pouvoir justifier de l'investissement important d'un circuit intégré spécifique.

C'est cette catégorie qui s'est naturellement présentée la première, mais on verra par la suite que c'est celle qui risque d'être la plus difficile à satisfaire à plus long terme.

Interfaçage: une deuxième famille d'applications entrent dans la catégorie *interfaçage* ou *temps-réel*. Elles sont caractérisées par un ou plusieurs des besoins suivants:

- des contraintes temps-réel à faible granularité (sub-milliseconde ou sub-microseconde);
- l'interfaçage physique avec un dispositif non normalement connectable à un ordinateur standard;
- des débits élevés d'entrée-sortie.

Cette catégorie semble être le champ d'applications le plus prometteur de ceux que l'on a seulement récemment commencé à explorer.

Prototypage: enfin, un dernier domaine qu'il est plus difficile de distinguer des précédents concerne le *prototypage d'algorithmes matériels*, consistant à développer, tester ou mettre au point des algorithmes nouveaux sur PAM de façon à pouvoir les essayer en configuration réaliste, tout en sachant que l'implémentation définitive sera faite sur circuit intégré classique. Au moins deux des algorithmes présentés dans [BRV93] rentrent dans cette catégorie.

Perspectives

On doit maintenant se poser la question des capacités potentielles d'évolution de cette technologie, au regard de ce qu'il est possible de prévoir aujourd'hui quant à l'évolution des technologies sous-jacentes ou concurrentes. On examine ci-après ces potentialités sous un certain nombre de facettes différentes, et on essaie dans chaque cas d'identifier les problèmes importants qui restent à résoudre pour que cette évolution puisse être aussi prometteuse que possible.

Puissance de calcul

On a vu qu'une part importante de l'applicabilité de la technologie PAM vient de sa capacité à délivrer sur certains problèmes des puissances de calcul supérieures à celles des ordinateurs classiques, voire même des super-ordinateurs. Cet avantage provient principalement de la souplesse d'architecture qu'offre la logique configurable, qui permet de tirer le meilleur parti d'un volume de logique donné là où l'architecture figée d'un processeur classique peut être très mal adaptée; il est néanmoins essentiel que la PAM possède une puissance de calcul

intrinsèque suffisante (au sens qui a été défini au §1.2.4) si on ne veut pas que cet avantage soit contrebalancé ou limité à une classe d'applications de plus en plus marginale.

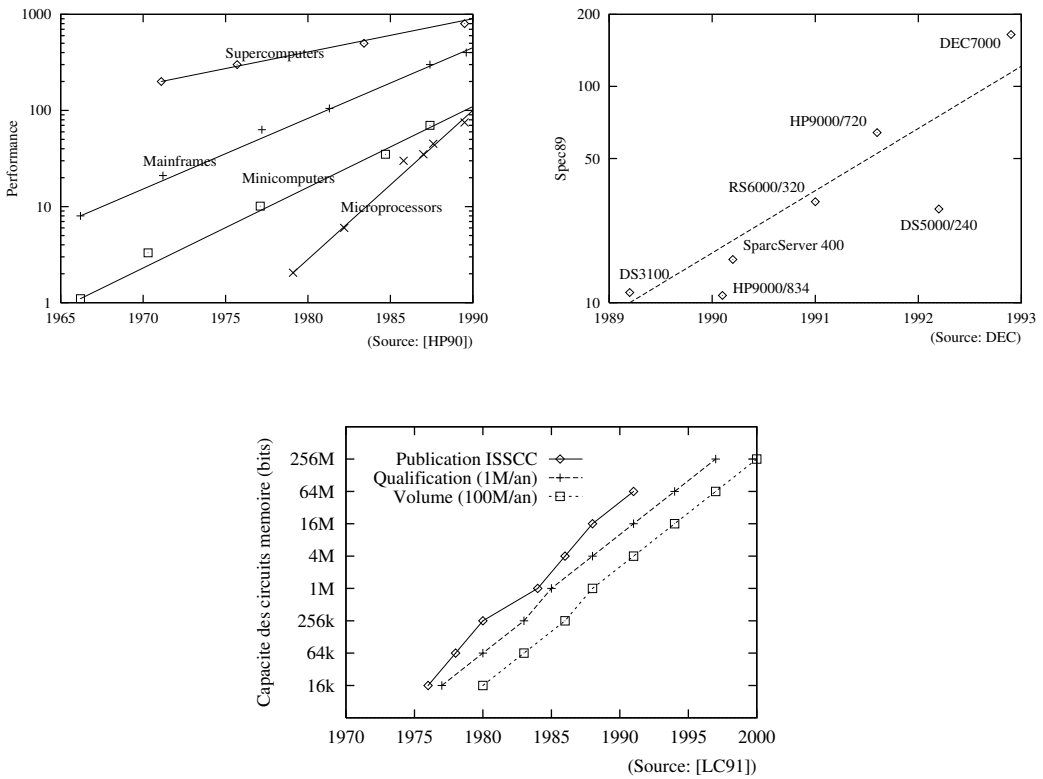


Figure 4.1: Evolution de la puissance des processeurs et de la taille des mémoires

L'étude de l'évolution de la technologie du silicium depuis maintenant près de 30 ans permet de constater le phénomène unique, mais solidement établi expérimentalement, qu'un certain nombre d'indicateurs de performance de cette technologie affichent une *croissance exponentielle* soutenue. Deux des indicateurs affichant ce comportement sont en particulier la puissance de calcul des microprocesseurs et la capacité des circuits mémoire (figure 4.1), et les prévisions actuelles semblent s'accorder sur une continuation de ce phénomène pendant encore plusieurs années [HP90, LC91]. Ces courbes globales sont en fait les enveloppes des courbes de progression individuelle de chaque génération, dont la pente est généralement supérieure à la moyenne en début de cycle pour devenir inférieure en fin de cycle.

La technologie des prédiffusés programmables est relativement récente, mais il y a tout lieu de penser qu'elle va suivre ou dépasser ce taux de croissance global pendant un certain nombre d'années: la capacité de ces circuits est étroitement liée à celle des mémoires (un prédiffusé programmable est essentiellement une mémoire statique dans laquelle sont dispersées des portes logiques "utiles"); leur vitesse dépend des mêmes paramètres que le temps de cycle des microprocesseurs; enfin étant une technologie jeune, on peut espérer y trouver un taux d'innovation architecturale supérieur à la moyenne pendant un certain nombre d'années. Le

dernier facteur important est celui de la demande, mais la croissance importante de celle-ci pendant ces premières années montre clairement que ces circuits ont trouvé une forte applicabilité et sont là pour rester[1].

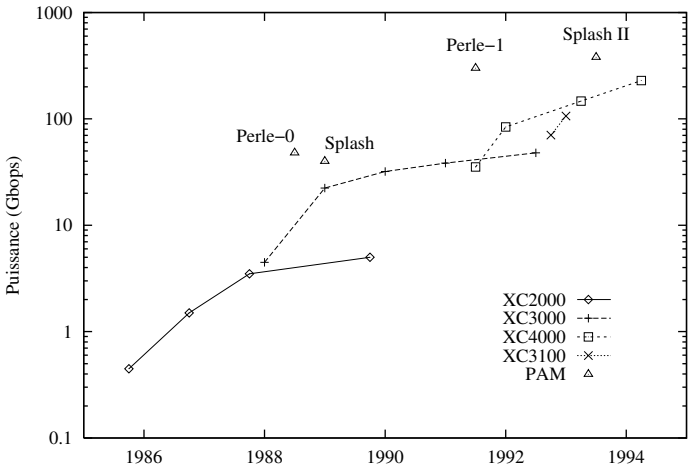


Figure 4.2: Evolution de la puissance des PAMs

On peut donc parier sur le fait que la technologie des prédiffusés programmables, sur laquelle repose fondamentalement le concept de PAM, va évoluer au moins aussi rapidement que celle des processeurs. Si les autres facteurs considérés au point suivant ne sont pas bloquants, le rapport de performances théoriques existant aujourd'hui ne devrait donc pas se dégrader. La figure 4.2 rassemble les caractéristiques des premières générations de prédiffusés programmables et de PAMs, montrant l'évolution potentielle qui se profile si on ne considère que ce seul facteur.

Architecture

Les challenges posés par la conception des PAMs sont d'une part le suivi de la courbe technologique en ce qui concerne le niveau de performances, d'autre part la versatilité permettant l'applicabilité au plus grand nombre d'applications. On considérera séparément ces différents aspects.

- Le niveau de performances nécessaire peut être obtenu par augmentation de la surface (nombre de PABs) ou de la vitesse (fréquence de fonctionnement); malheureusement, ces objectifs sont essentiellement contradictoires et il faut essayer d'obtenir le meilleur compromis. Un certain nombre de réalisations actuelles semblent ainsi privilégier fortement la surface au détriment de la vitesse ([Cas93, Qui91] et dans une moindre

[1] Les prédiffusés programmables constituent le segment du marché des composants électroniques ayant affiché la plus forte croissance ces dernières années.

mesure [ABD92]). Bien que ceci soit technologiquement plus facile, c'est à notre opinion un choix discutable: on risque ainsi de limiter la gamme d'applications intéressantes à une niche de plus en plus étroite, à savoir celles qui peuvent accommoder efficacement un énorme volume de logique (la principale classe d'applications entrant dans cette catégorie étant constituée par les algorithmes purement systoliques). En contrepartie la vitesse de fonctionnement des processeurs augmente régulièrement, grâce au bénéfice apporté par l'intégration, et on peut illustrer cette progression par les simples comparaisons suivantes:

- à 20 MHz, la fréquence de fonctionnement moyenne de Perle-0 était supérieure à celle de son hôte (16 MHz);
- à 25 MHz, la fréquence de fonctionnement moyenne de Perle-1 était initialement identique à celle de son hôte; elle est maintenant inférieure à celle-ci (40 MHz)[2];
- à la date de rédaction (début 1993), des processeurs fonctionnant à 150 MHz sont en cours de commercialisation, avec des versions 200 MHz à l'horizon proche.

Il semble donc plus intéressant de viser la taille maximale qu'il est possible d'atteindre sans trop compromettre la vitesse d'exécution (voir aussi le point suivant: modularité). Les PAMs possèdent ici un désavantage intrinsèque, devant être composées de plusieurs circuits pour compenser la perte de densité due à la reconfigurabilité. On devra le minimiser en utilisant les meilleures technologies disponibles dans au moins trois domaines:

- les prédiffusés programmables eux-mêmes (en ce qui concerne la vitesse et la densité);
- l'interconnexion et la mise en boîtier (*packaging*) des circuits, de façon à maximiser la densité et la vitesse de communication (les technologies hybrides semblent ici prometteuses, mais sont au-delà de la limite technologique de ce qu'une équipe de recherche restreinte peut aborder);
- la distribution d'horloge (sur ce dernier point la technologie nécessaire existe, mais est d'un emploi délicat).

- La flexibilité est l'autre atout essentiel des PAMs, puisqu'en permettant la plus large applicabilité d'un même système elle permet d'en amortir autant les coûts de conception et de réalisation. Elle peut être considérée sous deux angles différents:

  - ainsi qu'on l'a mentionné au §2.2.2, les schémas d'interconnexion entre les prédiffusés programmables d'une part, entre ceux-ci et leurs périphériques d'autre part, conditionnent largement la classe d'applications efficacement implémentables; ceci est principalement dû au fait que les prédiffusés programmables actuels sont beaucoup moins efficaces pour le brassage des données qu'ils ne le sont pour la logique régulière. La solution généralement retenue aujourd'hui semble être l'utilisation de composants d'*interconnexion programmable* spécialisés [ABD92,

[2] Elle pourrait néanmoins rejoindre cette valeur par une mise à jour technologique des composants utilisés.

CS93, Cas93]. Ceux-ci, d'introduction très récente [Apt92, ICu93], semblent en effet constituer à terme le complément idéal des prédiffusés programmables pour réaliser des PAMs; à notre opinion cependant, leur (relativement) faible niveau de performance actuel introduit une penalité telle qu'elle rend peu envisageable leur emploi si comme on l'a dit ci-dessus on cherche à réaliser des PAMs pouvant fonctionner à fréquence élevée.

Il est cependant possible que cette situation change, surtout si des circuits d'interconnexion programmables intégrant des registres de *retiming* sont introduits un jour. Dans l'immédiat cependant, il semble que la solution que l'on a retenue pour Perle-1 (utilisant des prédiffusés programmables pour ces fonctions d'interconnexion, cf. §2.3.2) soit toujours d'actualité; rappelons qu'en plus du brassage et du *retiming* des données, elle permet de plus d'implémenter efficacement toutes sortes de conversions de format, ainsi que la délocalisation des automates de contrôle associés.

– l'autre aspect de la versatilité d'une PAM concerne sa taille. Le défi à relever ici semble être celui de la *modularité* [ABD92], c'est-à-dire la conception de modules PAM élémentaires pouvant être utilisés tels quels ou assemblés en structures arbitrairement grandes. On pourra ainsi développer un même matériel à la fois pour les applications demandant peu de volume mais à faible coût (par exemple les applications d'interfaçage, qui sont potentiellement nombreuses) et pour la petite famille d'applications potentiellement utilisatrices de PAMs de très grande taille. La granularité de ces modules est donc à déterminer en fonction de considérations de taille physique, de performances et de prix de revient; bien que cela soit difficile à estimer il semble qu'à l'heure actuelle un module de la puissance de Perle-1 réalisé avec les technologies les plus compactes disponibles soit un ordre de grandeur intéressant.

  Il est à noter qu'une telle structure de PAM *hiérarchique* devra très probablement intégrer une hiérarchie similaire dans les vitesses d'horloge d'une part, dans la latence et la bande passante des canaux de communication d'autre part (ainsi que c'est le cas pour les systèmes de mémoire des ordinateurs modernes par exemple). On pourrait alors plus aisément utiliser des circuits d'interconnexion programmables du type de [Apt92], qui présentent des capacités d'interconnexion considérables au prix d'un temps de traversée relativement élevé.

- Un troisième facteur important est constitué par les possibilités de connexion à la machine hôte. L'architecture des ordinateurs s'oriente effectivement de plus en plus vers une structure hiérarchique de niveaux de mémorisation dont le processeur est le centre et où les dispositifs périphériques additionnels sont repoussés de plus en plus loin [HP90]. Plus précisément, la bande passante disponible sur ces périphériques est favorisée au détriment de la latence de communication entre ces périphériques et le processeur. Dans le cas présent, ce phénomène risque encore une fois de restreindre la classe des applications "algorithmiques" intéressantes à implémenter sur PAM aux seules plus grosses d'entre elles, en raison du nombre croissant d'instructions que l'hôte est capable d'effectuer pendant le temps d'un simple aller-retour de données entre le

processeur et la PAM.

Il est donc important de choisir le meilleur canal d'entrée-sortie disponible sur la machine hôte. Bien que ce choix soit généralement assez limité, on peut noter qu'il existe sur la plupart des stations de travail une possibilité de connexion d'écrans graphiques haute-résolution, avec des besoins similaires de bande passante élevée et de latence pas trop importante. Ceux-ci fournissent probablement la garantie qu'une possibilité raisonnable de connexion continuera à exister dans le futur proche. A plus long terme bien sûr, on peut envisager une prise en compte directe du concept PAM dans la conception même des machines, si celui-ci s'avère suffisamment utile.

- De façon beaucoup plus spéculative, on peut envisager les possibilités suivantes d'évolution à plus long terme de cette technologie:
  - ainsi qu'on l'a mentionné à plusieurs reprises, les circuits prédiffusés actuels sont conçus pour un autre usage, et sont assez nettement sub-optimaux sur certains points vis-à-vis de l'utilisation en PAM. On peut donc imaginer des versions de prédiffusés programmables plus directement adaptées à cet usage – par exemple avec des facilités d'interconnexion efficace entre circuits adjacents [Alg90], une meilleure distribution d'horloge, une configuration plus rapide, etc... – voire même radicalement différentes des versions existant aujourd'hui – intégrant des ressources globales de brassage de signaux, de la mémoire, des processeurs, etc... Ceci demande bien entendu pour être économique que la technologie PAM obtienne préalablement un succès de diffusion qu'elle est encore loin d'avoir, mais n'est certainement pas inenvisageable à moyen terme, d'autant plus que les fabricants de prédiffusés programmables ayant maintenant atteint une certaine taille commencent à disposer de ressources leur permettant plus de liberté dans l'exploration de technologies "exotiques" apparentées.
  - à plus long terme un débouché naturel de cette technologie – si l'on veut bien y croire – est bien entendu l'intégration de ressources logiques programmables, sous une forme ou sous une autre, directement au sein des microprocesseurs eux-mêmes. A terme, c'est bien évidemment la seule façon efficace de répondre aux interrogations présentées ci-dessus concernant les canaux de communication avec l'hôte, tout particulièrement en ce qui concerne les applications "algorithmiques". C'est aussi une façon envisageable d'utiliser efficacement la surface de silicium toujours croissante qui est offerte aux concepteurs de processeurs, maintenant que les phases essentielles d'intégration du premier puis du deuxième niveau d'antémémoire, du coprocesseur arithmétique, des tables de conversion d'adresses virtuelles, etc... ont été réalisées, ainsi que tout récemment l'augmentation de la largeur des chemins de données à 64 bits. Ceci ne sera bien sûr envisageable que si le concept de PAM s'avère être un atout majeur dans l'architecture des ordinateurs. Il est par ailleurs important de noter qu'un tel concept va à l'encontre du principe de base d'utilisation des ordinateurs par les systèmes d'exploitation actuels, où le processeur partage son temps de façon transparente entre plusieurs tâches indépendantes: l'introduction de logique programmable augmente en effet

considérablement le nombre de bits représentant l'état d'un calcul en cours, pénalisant le temps de changement de tâche de façon inacceptable. Ce dernier facteur permet probablement de prédire que si la logique programmable fait son introduction dans les processeurs, ce sera initialement sous une forme différente et très limitée, ou pour des machines à usage spécialisé.

Programmation

La programmation des PAMs est aussi un domaine qui est encore balbutiant. Les techniques existantes sont héritées de la conception des circuits intégrés mais sont mal adaptées: d'une part le problème physique est ici assez différent, d'autre part la différence fondamentale dans l'effort de fabrication fait que les utilisateurs ne veulent pas utiliser la même méthodologie (effort important de préparation, conception détaillée, vérifications lourdes), mais cherchent au contraire à utiliser des méthodes similaires à celles de la conception logicielle (construction et mise au point incrémentale, description de "haut-niveau").

Sur beaucoup de points, la méthode utilisée actuellement pour les PAMs de taille conséquente [BRV93, GK+90] est similaire à celle qui était utilisée pour la programmation des premiers ordinateurs: les outils étant frustres, la taille du problème encore gérable et les besoins en efficacité élevés, la conception est essentiellement faite "à la main" au niveau le plus bas. Il est néanmoins probable que comme ce fut le cas pour le logiciel, l'accroissement de taille des machines d'une part, le développement des techniques de compilation d'autre part, rendront cette conception détaillée de moins en moins nécessaire, ou tout au moins en limiteront l'usage aux parties vraiment critiques des applications[3].

La tendance actuelle des recherches en la matière semble nettement pencher en faveur d'une compilation entièrement automatisée, à partir de descriptions aussi "logicielles" que possible; ceci reflète aussi le fait que beaucoup de programmeurs PAM potentiels proviennent des domaines de la programmation logicielle. A notre opinion cependant, l'obtention de résultats performants à partir d'une telle approche est à peu près sans espoir, sauf à se placer à l'intérieur d'une classe très particulière d'applications. La conception d'une configuration de PAM comporte effectivement, en plus de la logique, une composante géométrique qui ne semble pas pouvoir être synthétisée complètement automatiquement dans un futur proche; on postulera donc par la suite que le concepteur doit initialement décrire à la fois la logique et la géométrie de son application, et on aura comme objectif que l'une comme l'autre de ces descriptions puissent être faites de la façon la plus aisée et au plus haut niveau possible.

On peut envisager toute une gamme d'améliorations des algorithmes ou des outils allant dans ce sens, depuis la mécanisation d'opérations délicates ou fastidieuses aujourd'hui faites à la main sans changement fondamental du résultat obtenu (l'équivalent pour le logiciel serait le travail effectué par les tout premiers compilateurs Fortran ou C par exemple), jusqu'à l'application d'algorithmes sophistiqués produisant des résultats que l'on n'aurait

[3] La même prédiction a été faite maintes fois dans le passé pour les circuits intégrés classiques et ne s'est réalisée que marginalement dans le cas des circuits performants. Ainsi qu'on l'a mentionné cependant, la nature intrinsèquement longue et chère de la fabrication de ces circuits influence fondamentalement l'approche utilisée.

manuellement qu'au prix d'efforts considérables (l'équivalent logiciel serait l'optimisation de code pour un processeur RISC pipeliné moderne).

On passe ci-dessous en revue (sans prétention d'exhaustivité) un certain nombre de domaines qui semblent devoir évoluer si l'on veut progresser dans la direction que l'on a indiquée. Il est à noter que ce découpage est néanmoins assez artificiel, une des principales complexités venant du fait que la plupart de ces domaines sont en fait assez intimement inter-dépendants.

Géométrie Un des points les plus cruciaux aujourd'hui concerne la description de la géométrie des circuits. La plupart des efforts actuels portent sur la synthèse automatique complète de celle-ci; les résultats obtenus étant décevants à l'échelle globale d'une PAM, les outils existants ne sont pas utilisables et le programmeur doit finalement faire la conception complète par lui-même. Il semble cependant que deux étapes intermédiaires intéressantes consisteraient en:

- la synthèse automatique de structures régulières, en fonction d'indications suffisamment précises données par le programmeur (ceci rentre dans la catégorie "mécanisation" ci-dessus);
- la synthèse automatique *locale* des structures irrégulières, selon une allocation globale de surface (*floorplanning*) faite par le programmeur.

Notons néanmoins que ces problèmes géométriques sont étroitement mélés aux problèmes de synthèse logique, de bibliothèques et de *retiming* mentionnés ci-après.

Bibliothèques Il s'est avéré très difficile[4], au cours des premières années d'utilisation des PAMs, de mettre au point des bibliothèques de modules d'usage courant (compteurs, opérateurs arithmétiques et logiques usuels) pouvant être réutilisés d'application en application. Deux raisons au moins peuvent être identifiées:

- le grand nombre de variantes possibles d'un même opérateur selon le contexte (plus précisément selon la représentation des données dans l'espace et le temps);
- les nombreux degrés de liberté supplémentaires introduits par les contraintes d'implémentation (géométrie, retiming).

Le challenge semble ici reposer sur la sélection d'un nombre restreint mais bien choisi de types de données d'une part, de stratégies d'implémentation d'autre part, alliée aux techniques d'optimisation logique et de retiming automatique abordées plus loin; on devrait alors pouvoir constituer des bibliothèques d'opérateurs définis par un nombre raisonnable de paramètres, remplissant l'essentiel des besoins usuels.

Synthèse logique Les techniques de synthèse et d'optimisation logique destinées aux circuits intégrés classiques sont aujourd'hui relativement mûres en ce qui concerne la logique combinatoire, et sont en développement actif pour la logique séquentielle [BCM90, STB91]. Elles ne sont cependant pas souvent directement applicables, du fait de la

[4] On peut même considérer cela comme un échec par rapport aux objectifs initiaux.

différence des modèles technologiques et des contraintes d'implémentation à prendre en compte dans chaque cas. On assiste néanmoins avec le développement des prédiffusés programmables à un effort de recherche croissant dans ce domaine spécifique. Bien que l'interaction avec les contraintes géométriques présente là encore une difficulté essentielle, on peut dès aujourd'hui obtenir de bons résultats d'optimisation si l'on se limite à un niveau local; la mise en œuvre de ces techniques constitue certainement une des clés de la simplification du problème de conception pour PAM, ainsi que de la production de bibliothèques.

Retiming Un problème traditionnellement considéré au sein du précédent, mais qui prend ici une importance toute particulière est celui du *retiming* des circuits logiques, c'est-à-dire de l'insertion dans un circuit existant de registres de pipeline en nombre et aux endroits judicieux permettant le fonctionnement à une fréquence d'horloge optimale [LS91]. La forte importance relative des délais de propagation dans la technologie qui nous préoccupe, surtout lors des communications inter-circuits, entraîne généralement un recours massif au retiming, le plus souvent qui plus est de façon irrégulière dans la structure du circuit en développement.

Cette opération est très délicate à effectuer manuellement car elle affecte globalement toute la structure du circuit, et dépend des délais de propagation fins obtenus. Un des premiers outils à réaliser concernerait donc le retiming global de circuits implémentés sur PAM – sous contrôle du programmeur –, incluant éventuellement les modifications appropriées des automates de contrôle. Cette opération est bien comprise théoriquement; le point difficile est encore une fois l'interaction avec la géométrie, mais il ne devrait pas être trop difficile à surmonter dans ce cas particulier, les prédiffusés programmables possédant intrinsèquement une grande densité de registres.

Description Enfin, le "bon" langage de description de circuit n'existe probablement pas encore. Ainsi qu'on l'a esquissé au §3.1.3, il est probable que celui-ci devra intégrer de façon mixte les différentes méthodes de description utilisées aujourd'hui (schémas graphiques, description textuelle et synthèse algorithmique), en plus de la capacité à décrire conjointement les structures logique et géométrique des circuits.

Conclusions

On a décrit dans cette thèse les premières expériences menées sur une technologie nouvelle, les mémoires actives programmables, sous l'impulsion initiale de Jean Vuillemin. Pour les trois domaines complémentaires que constituent l'architecture, la programmation matérielle et la programmation logicielle, on a montré comment il était possible de construire un premier prototype opérationnel très simple, puis une deuxième version plus sophistiquée conçue à partir de l'expérience acquise; dans chaque cas, on a essayé de procéder à une analyse critique des solutions ainsi obtenues.

Les résultats obtenus par cette première phase valident le concept initial, en montrant que:

- il est possible de construire des coprocesseurs matériels d'usage général que l'on appellera PAMs, sur la base des nouveaux circuits dits "prédiffusés programmables",

avec une puissance de calcul comparable à celle que fournit l'état de l'art de la technologie des circuits prédiffusés usuels;

- ces PAMs permettent d'obtenir sur toute une gamme de problèmes un niveau de performance comparable à celui des meilleures réalisations sur super-ordinateurs; la solution apportée est alors plusieurs ordres de grandeur meilleure en termes de rapport performance-coût;

- le développement d'une application sur PAM se mesure usuellement en semaines; ce temps est donc du même ordre que celui qui est nécessaire au développement d'une solution hautement optimisée sur super-ordinateur; il est par contre un ordre de grandeur inférieur au temps de développement d'une solution équivalente à base de circuits intégrés spécifiques.

Bien que la technologie PAM soit encore à un stade initial de son développement, on peut d'ores et déjà dire que les résultats de cette première phase d'expérimentations font désormais partie de l'acquis: à titre d'illustration, les minutes de la première conférence internationale consacrée spécifiquement au sujet [FCCM93] font état de plus de dix présentations traitant de systèmes matériels différents rattachables au concept que l'on a ici baptisé PAM.

Cette technologie semble posséder un important potentiel de croissance, bien qu'un certain nombre de problèmes architecturaux et logiciels restent à résoudre. Il est probable que le champ des applications potentielles n'ait encore été qu'effleuré; en particulier les PAMs semblent pouvoir fournir des solutions nouvelles extrêmement intéressantes aux problèmes d'acquisition et de filtrage en temps réel de flots de données à haut débit.

Annexe A

Introduction to Programmable Active Memories

Patrice Bertin
Didier Roncin
Jean Vuillemin

June 1989

Abstract

We introduce the concept of PAM, *Programmable Active Memory* and present results obtained with our Perle-0 prototype board, featuring:

- A software silicon foundry for a 50K gate array, with a 50 milliseconds turn-around time.
- A 3000 one bit processors universal machine with an arbitrary interconnect structure specified by 400K bits of nano-code.
- A programmable hardware co-processor with an initial library including: a long multiplier, an image convolver, a data compressor, etc. Each of these hardware designs speeds up the corresponding software application by at least an order of magnitude.

Ce chapitre est la réimpression du *Rapport de recherches n° 3*, publié par Digital Equipment Corporation, Paris Research Laboratory, 85, av. Victor-Hugo, 92563 Rueil-Malmaison Cedex, France (1989).

Une version antérieure de ce rapport apparaît, sous le même titre, dans *Systolic Array Processors*, édité par J. McCanny, J. McWhirter, and E. Swartzlander Jr. et publié par Prentice-Hall (1989).

A.1 Concept

Definition (Programmable Active Memory: PAM)
A PAM is a uniform *array of identical cells all connected in the same repetitive fashion. Each cell, called a PAB for* programmable active bit*, must be general enough so that the following holds true:* **Any** *synchronous digital circuit can be realized (through suitable programming) on a large enough PAM for a slow enough clock.*

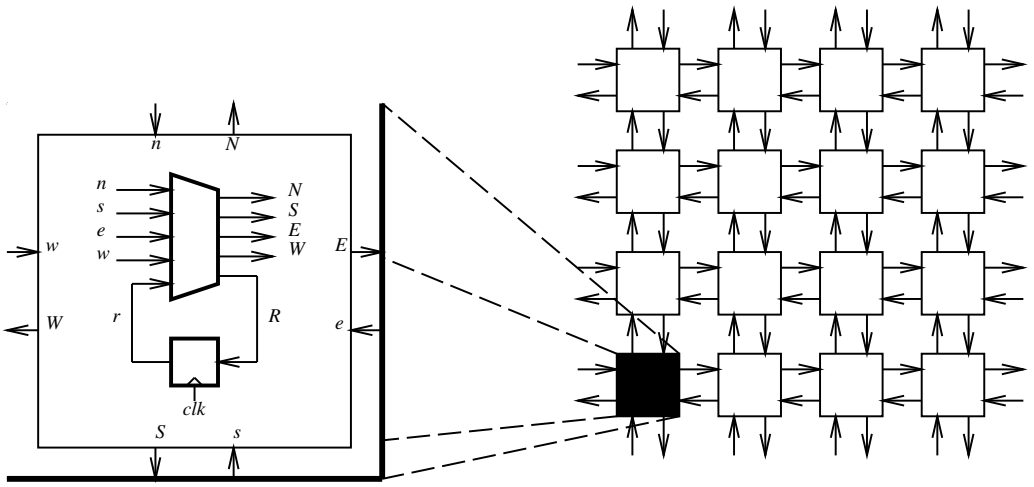


Figure A.1: A simple PAM

Figure A.1 shows a simple PAM implementation as a regular matrix of Manhattan connected identical PABs. Each PAB has:

- 4 inputs $< n, s, e, w >$;
- 4 outputs $< N, S, E, W >$;
- 1 register (flip-flop) with input R and output r, synchronous with the PAM's global clock clk;
- 1 combinatorial gate g connected so that: $g(n, s, e, w, r) = < N, S, E, W, R >$;
- $160 = 5 \times 32$ control bits which specify the truth table of function g.

A *program* for such a PAM with m active bits is a sequence of $160m$ control bits (*bitstream*) representing the truth tables for each PAB. This program can be *downloaded* into the control part of the PAM. From this instant, and until the program is changed again, our PAM behaves as the particular finite state machine specified by the bitstream.

Other PAM implementations are possible:

- the *Configurable Array Logic* (CAL) reported in [GK] is close to the one above;
- the *Logic Cell Array* (LCA) is a cMOS component designed by Xilinx, Inc. [X]. Except for minor details, it is similar to the above description with each of our $< n, s, e, w >$ and $< N, S, E, W >$ variables carrying 4 signals instead of one. The truth table of function g is reduced, for obvious practical reasons: only one general 5 input boolean function is actually specified; the remaining part of g is used to specify various *switchboxes*, usable to route the signals.

Our investigation aims at answering the following questions:

- How to build large PAMs?
- How fast can it run?
- Which class of design can we efficiently map onto a PAM?
- How do we implement PAM designs?

A.2 Implementation

Using a 5×5 array of LCA chips, we have built a $40 \times 80 \simeq 3K$ bit PAM named Perle-0 on a 25×25 cm^2 printed circuit board. Perle-0 has a VME bus interface, which makes it a general-purpose configurable hardware co-processor tightly coupled to a host CPU (Figure A.3).

The configuration data for Perle-0 (about 400K bits) is downloaded by the host itself in 50 milliseconds. The logic controlling the download process, as well as the host bus communication protocol are *programmed* into two extra LCAs, statically configured at power-up time from a PROM. By merely changing the content of that PROM, we are able to quickly adapt to different bus protocols, or to add extra features to the bus interface.

The PAM cycle being much faster than that of the host bus, we added 4 Megabits of fast static RAM to Perle-0, directly connected to the PAM (the bandwidth of that memory is up to 1.5 Gigabits per second, while the host bus bandwidth is typically around 50 Megabits per second). Besides a few mandatory buffers for driving the host bus, Perle-0 is thus built out of two kinds of components: LCA and static RAM (Figure A.2).

PAM designs are synchronous logic circuits, each of the registers being updated on each cycle of a global clock signal. The maximal clock speed for such a design is directly determined by its critical combinatorial path, which varies from one design to another. It has thus been necessary to design a clock distribution system whose speed can be *programmed* as part of the design configuration, for speeds up to 70 MHz (the present maximum clock cycle of LCA chips).

Last but not least, we take advantage of an extra feature of the LCA component which makes it possible to dynamically *read back* the content of each PAB. Together with a "software stepping" facility (stop the main clock and trigger clock cycles one at a time from the host)

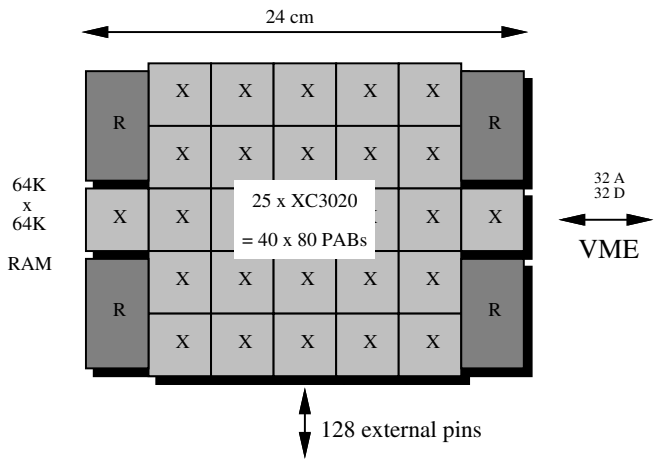


Figure A.2: Perle-0

this provides a powerful debugging tool which can take a snapshot of the internal state of the design after each clock cycle.

A.3 PAM programming

The logical description of a PAM design consists of the truth table for each PAB and the description of their interconnection. In order to derive the final configuration bitstream for each LCA, we also need to specify the placement of each block within the PAM, as well as the actual routing of each signal through the available channels and switching boxes.

At the LCA level, both placement and routing may be automatically generated by Xilinx software. We have found this to be practical only for routing; automatic placement is far too time-consuming and it generally yields poor results: in this technology, each programmable interconnection point adds about 2 nanoseconds to the signal propagation delay. Therefore, a poor relative block placement may considerably affect the final design performance.

In our CAD system, the designer needs to specify the logic equations and net names for each PAB; optionally he may also control the actual placement of each PAB. This is achieved through a program in a high-level language (we use Modula-2+ and Lisp) or through schematic entries. Netlists are then generated for each of the 25 LCAs of Perle-0 and we use Xilinx software to finish the placement, do the actual routing and produce the bitstream for each LCA. Finally, the 25 bitstreams are merged into a 400K bits long file ready to be downloaded by the host into the PAM.

The above steps typically take from 15 minutes to a few hours, depending on the complexity of the design; subsequent minor modifications are generally much faster. The actual downloading is then performed in 50 ms. Since debugging the designs can be done through stepping and readback, this effectively brings hardware design times very close to these of software. In particular, it provides a vehicule for experimenting with hardware algorithms which would otherwise be too time-consuming and costly to implement.

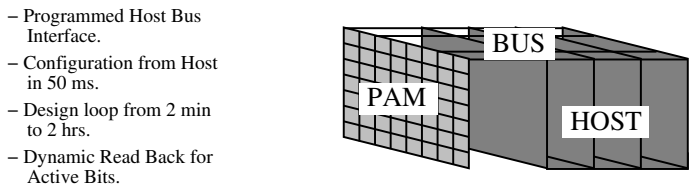


Figure A.3: Perle-0 setup

A.4 Design experiments

We describe here some of the designs which have been successfully tested on Perle-0. More designs are currently in progress.

A.4.1 Early experiments

Our first designs aim at debugging and testing the performance of Perle-0 itself:

- A software-controlled variable clock generator, used as a basic block in building other designs.
- A combinatorial link established through all the pins, in order to test the printed circuit board traces for electrical continuity; it also implements one of the worst-case combinatorial delay achievable through routing, and runs at 4 MHz.
- A 3K shift-register through all of the PAM's active bits. It provides an example of the fastest possible synchronous design and proves operational at clock speeds up to 50 MHz.
- Designs to read and write Perle-0's RAM from the host. Their speed is limited by the VME bus bandwidth, at about 50 Mbits/s.

Such simple designs typically take a few hours to build, compile and debug.

A.4.2 Long multiplication

We have programmed Perle-0 into a fast 512-bit multiplier (see [L]) computing

$$P = A \times B + S \text{ where:}$$

- A is a m bit integer, with $m \leq 512$;
- B is a n bit integer, with $n > 0$ arbitrary;
- S is a $n + m$ bit integer;
- P is a $n + m + 1$ bit integer.

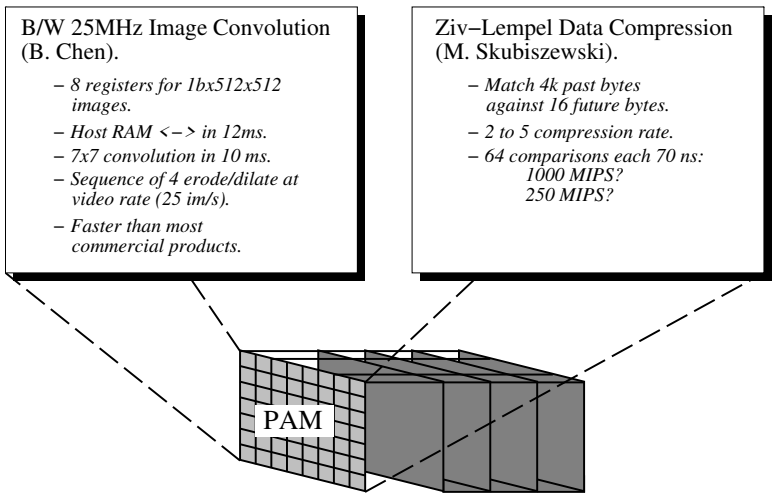


Figure A.4: Two PAM designs

The host writes the A, B and S operands into the PAM, and reads back the resulting P. The speed-up factor achieved by this hardware co-processor can be greater than 25 for long operands, compared to carefully hand-coded software. It computes a 1K bit product in 210 microseconds, which makes this PAM design faster than *any* known machine for which we could obtain benchmark measures. It is 4 times faster than a Cyber 170/750.

The multiplier is interfaced with an arbitrary-precision arithmetic package [SVH] so that any program based on that software takes advantage of the PAM without modification. For example, Hans Boehm has run computations for exact real arithmetic [B] with typical speedups from 3 to 7. We have written code for modular products and used it for RSA cryptography [RSA]. The PAM computes RSA encryption/decryption at 3900 bits per second for arbitrary 512-bit keys; this is about 4.5 times faster than our best software version on the same host.

The first operational version of the multiplier was developed in less than a week. Two subsequent versions, which refined the design on the basis of actual performance measurements, were each developed in less than 5 man-days.

A.4.3 Data compression

A hardware accelerator for data compression was implemented in four months by Marcin Skubiszewski, who had no previous hardware experience. Based on an algorithm described in [FG], it is a massively parallel method (64 byte comparisons each 70 ns), which repetitively matches the next 16 bytes in the file to be compressed against the last 4k bytes seen (stored in the local RAM) in order to detect the longest substring previously seen (see [ZL]). The resulting design achieves an average compression ratio varying from 2 for English text to 3 for source code.

A.4.4 Image processing

J. Bradley Chen has implemented the operations of erosion and dilation (two-dimensional boolean convolution) on $512 \times 512 \times 1$-bit images, with 7×7-bit masks. The design runs at 25 MHz, computing one bit of the transformed image on each cycle, for a total of 10 ms per image. Our best software solution for that problem is at least two orders of magnitude slower. This design was implemented in about 3 months of work, once again with minimal previous hardware experience.

A.5 Conclusion

PAM is an exciting new technology, with a lot of growth potential. Based on our first year of experiments, we derive the following conclusions:

Prototyping. While custom hardware prototyping times are at best measured in months, prototyping with PAM is achieved in weeks. Design times for PAM must be compared with those of software, not of hardware.

Speed. Assuming equal technology, custom hardware designs are faster than their PAM implementation; yet the speed loss appears to be less than 50% for systolic designs with short connections.

Size. While custom hardware designs are obviously smaller than their PAM counterpart, the *same* PAM can be used to run *any number* of hardware designs! We regard the PAM as a *hardware cache*: it can be programmed to perform arbitrary tasks, at speeds comparable to those of custom hardware.

Size vs. Speed. All our designs are such that we can double the bandwidth on a twice bigger PAM. This provides a strong incentive to build much larger ones.

Architecture. As a general purpose computing device, Perle-0 exhibits some unique features:

- The bitstream is a program for Perle-0 which is thus the ultimate one-instruction RISC machine (400K wide nano-code!).
- The PAM is a parallel machine with 3000 one-bit processors. The interconnect structure between these processors can be *arbitrary*, since it is specified by the program. Exotic interconnect structures (hypercube and such) are feasible, but they use up more PAM area and run with slower clocks.
- The host and PAM system is a *silicon foundry* with a 50 ms turn-around time. The host can also *read back* the content of each active bit in the PAM, providing an invaluable design independent debugging tool.
- The PAM is a flexible structure which can be *tailored* (logic, memory and interconnect) to the efficient implementation of *any* well designed massively parallel bit-level algorithm. This should be contrasted with the usual situation where the algorithm must be chosen so as to adapt to the structure of the implementation machine.

References

[B] H. J. Boehm, Constructive real interpretation of numerical programs. In *Proceedings of the ACM Conference on Interpretors*, 214–221 (1987).

[FG] E. R. Fiala and D. H. Greene, Data compression with finite windows. *Communications of the ACM*, 32(4):490–505 (April 1989).

[GK] T. Kean and J. Gray, Configurable hardware: two case studies of micro-grain computation. In J. McCanny, J. McWhirter and E. Swartzlander Jr., editors, *Systolic Array Processors*, Prentice-Hall, 310–319 (1989).

[L] R. F. Lyon, Two's complement pipeline multipliers. *IEEE Transaction on Communications*, 24:418–425 (1976).

[RSA] R. L. Rivest, A. Shamir, and L. Adleman, A method for obtaining digital signatures and public-key cryptography. *Communications of the ACM*, 21(2):120–126 (February 1978).

[SVH] B. Serpette, J. Vuillemin, and J.C. Hervé, *BigNum: A Portable and Efficient Package for Arbitrary-Precision Arithmetic*. Research Report No. 2, Digital Equipment Corp., Paris Research Laboratory, Rueil-Malmaison (1989).

[X] Xilinx, Inc., The Programmable Gate Array Data Book. Product Specifications (1988).

[ZL] J. Ziv and A. Lempel, A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343 (March 1977).

Annexe B

Programmable Active Memories: a Performance Assessment

Patrice Bertin
Didier Roncin
Jean Vuillemin

March 1993

Abstract

We present some quantitative performance measurements for the computing power of Programmable Active Memories (PAM), as introduced by [BRV89]. Based on Field Programmable Gate Array (FPGA) technology, the PAM is a universal hardware co-processor closely coupled to a standard host computer. The PAM can speed up many critical software applications running on the host, by executing part of the computations through a specific hardware design. The performance measurements presented are based on two PAM architectures and ten specific applications, drawn from arithmetics, algebra, geometry, physics, biology, audio and video. Each of these PAM designs proves as fast as any reported hardware or super-computer for the corresponding application. In cases where we could bring some genuine algorithmic innovation into the design process, the PAM was measured to be an order of magnitude faster than any previously existing system (see [SBV91] and [Sku92]).

Ce chapitre est la réimpression du *Rapport de recherches n° 24*, publié par Digital Equipment Corporation, Paris Research Laboratory, 85, av. Victor-Hugo, 92563 Rueil-Malmaison Cedex, France (1993).

Une version antérieure de ce rapport apparaît, sous le même titre, dans *Proc. of the Symposium on Integrated Systems*, Seattle, WA, USA, March 1993, publié par MIT Press (1993).

B.1 PAM concept

Like any RAM memory module, a PAM is attached to the system bus of a host computer. The processor can write into, and read from the PAM. Being an active hardware co-processor however, the PAM processes data between write and read instructions. The specific processing is determined by the content of its *configuration memory*. The host can change the PAM configuration by *downloading* a new design into it, within a few milliseconds.

We speed up a specific software application running on the host, by executing its *critical inner-loop* through an appropriate hardware design downloaded into the PAM. Ten examples of such designs and applications are presented below.

In selecting them, we have attempted to cover as many application areas as we could. In each, we picked basic and frequent enough problems, where a large inner-loop speedup through a specific PAM design results in a significant speedup of the application system, software and hardware combined.

For the sake of specification and debugging, the functionality of each hardware design is matched, bit-wise at the I/O level, by a software implementation. Through successive refinements, both hardware and software implementations are optimized for speed, while retaining identical I/O behavior. This is how we derived our experimental data in this assessment of PAM's computing power.

The traditional measurement unit here is the Mips[1], or its multiple the Gips ($1G = 10^9$) which is more appropriate to the levels of performance at hand. As explained in [HP90], quantitative performance comparisons between different computer architectures is a challenging art. First, our Mips are measured in different units: for the 32b MIPS R3000 or the 64b Alpha AXP instruction sets? for which clock speed? cache size? bus bandwidth? Second, in a number of cases we have external data for both VLSI and super-computer implementations of systems close enough to some of ours, so that we could make meaningful speed comparisons. Such external data is typically given in Gflops (Giga floating point operations per second) for super computers; for VLSI, number of transistors with operating clock frequency are normally quoted.

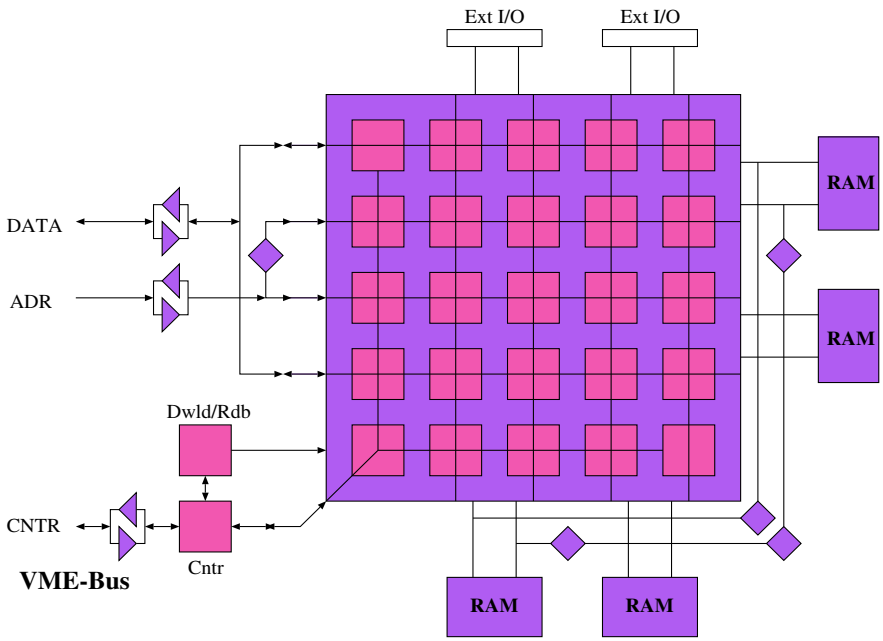
To make quantitative comparisons significant across such a wide spectrum of feasible implementation technologies, we introduce a common unit for measuring all forms of computing power, the *Gbops* (billion of binary operations per second) where each useful boolean operation with up to 4 inputs, be it clocked or not, counts for one. For example the computing power of a standard carry-ripple 32b adder operating at 100 Mhz is 6.4 Gbops, accounting for both carries and sums. That of a n bits multiplier at 1 GHz is $2n^2$ Gbops. Any specific computer operation can be similarly decomposed at the bit level, so we can evaluate its corresponding power in Gbops.

[1] Million of instructions per second

B.2 Two PAM architectures

Our assessment is based on two PAM architectures realized at DEC Paris Research Lab., DECPeRLe-0 (see [BRV89]) and DECPeRLe-1 which we respectively refer to as P_0 and P_1 in the following.

Each PAM is built around a large array of bit-level configurable logic cells (hereafter called Programmable Active Bits or PABs) in which the application-specific hardware operator is programmed. This array is surrounded with local RAM banks used as a cache (wide and fast enough to match the PAM's processing bandwidth), a programmable clock generator, and some additional non-configurable logic to manage the host bus interface and the download process.



The above figure sketches the architecture of P_0 (1988). The central computational array is made of a 5×5 matrix of Xilinx XC3020 Programmable Gate Arrays [Xil87]; it has two 32-bit wide RAM banks on the south and east sides, a VME bus interface on the west side and general-purpose interface connectors on the north side. The control and bootstrap logic is implemented in two extra XC3020 (non user-programmable). Finally, additional bus switching resources are provided for global data routing (represented here with diamond-shape boxes).

The architecture of P_1 was designed after two years of P_0 usage . It features a 4-times-bigger central computational matrix with accordingly wider RAM, a faster host interface, and a much more flexible global interconnection network for data routing and switching.

For the purpose of evaluating Xilinx-based designs such as ours, it is convenient to define the Programmable Active Bit as in [BRV89]: a PAB consists of a universal 4 input combinatorial gate and a synchronous flip-flop. Using this measurement unit, our two PAM architectures P_0 and P_1 have the following vital statistics:

PAM	part	number	PABs	Fmax (MHz)	Power (Gbops)	RAM (MB)	Host bus (MB/s)
P_0	XC3020	25	3.2K	25	80	0.5	8
P_1	XC3090	23	14K	40	588	4	100

This chart exhibits the three most important architectural parameters conditioning which application benefits most from a PAM speed-up:

- The number of PABs (3.2K for P_0 and 14K for P_1), together with the application-dependent maximal clock frequency (25 MHz and 40 MHz) at which we can reliably operate. The product of these two numbers is the maximum theoretical computing power of the PAM, expressed in Gbops (80 Gbops for P_0 and 588 Gbops for P_1).
- The host bus bandwidth: 8 MB/s for P_0 through a VME bus, and 100 MB/s for P_1 through a TURBOchannel interface [DEC91].
- The size of the local (fast) RAM: 0.5 MB for P_0 and 4 MB for P_1.

B.3 PAM programming

Programming an algorithm on the PAM is similar to casting it in conventional hardware (gate-array or VLSI), with two very important differences: first the target hardware provides a clean implementation of the synchronous logic model, so there is no need to worry about low-level electrical details; second the whole design process is entirely software operated with a fast turnaround time (5 to 30 minutes edit-compile-run loop), so it can be approached with the same methodology of piecewise testing and successive refinements as a software design. For a given application, it involves:

1. identifying the critical computations which are best implemented in hardware; this is usually done by successive refinement, under constraints of communication bandwidth and load balancing between the software and hardware;
2. implementing the hardware part on the PAM and gradually optimizing it;
3. implementing the software part on the host processor and gradually optimizing it.

Step 3 above is done with conventional techniques. Step 2 consists in describing the logic design to implement in the PAM down to the individual bit level, as well as its geometric layout, much as is done for a conventional VLSI design.

For this, we have developed a tool suite in which the hardware design is described by the writing of a program in a conventional programming language (we use Lisp, C++ [Tou92]

and Esterel [Ber92]) using a specialized library. This program will describe the various logic modules by their bit-level logic equations, or by using standard library modules (adders, registers, standard interfaces...). It will also contain layout (geometric) information to the level of details the designer wishes to specify: this usually starts with global floorplanning and detailed layout of the important datapath components only, to be enriched with more precise descriptions along the optimization process.

Executing this program produces a partially placed, hierarchical netlist which can either be simulated, or compiled to the final PAM configuration in a fully automatic way, through tools developed in-house (board-level routing, logic optimization) and standard Xilinx back-end software (chip-level placement and routing, bitstream generation).

The design can then be downloaded into the PAM for debugging and testing under real conditions; its maximum running speed can be characterized, and its critical paths identified (we have developed specialized interactive tools to help visualize the latter). The designer can then gradually optimize it, for example by adding extra levels of pipeline to increase the clock speed, or optimizing the geometric layout, or adding new functionalities to further unload the software. The most important point there is that, as compiling and trying a new version costs no money and only a few minutes to an hour of time, it is possible to incrementally design, test and optimize the algorithms and implementations involved, as is usually done with software, as opposed to having to have them correct and optimal on the first try.

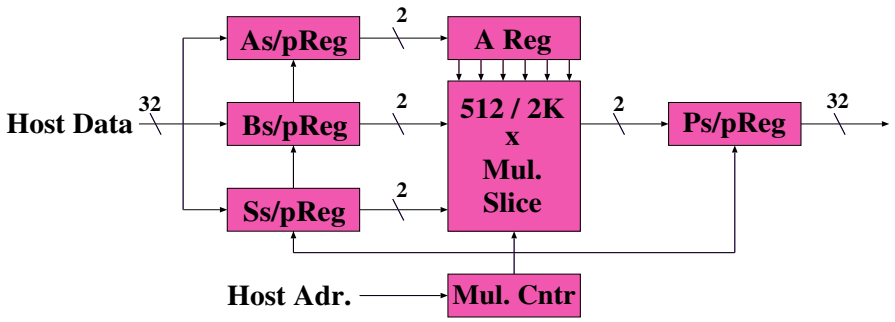
B.4 Ten PAM applications

The following applications were chosen to span a wide range of current leading edge computational challenges. In each case, we provide a brief description of the design, the names of the implementors, and a performance comparison with similar reported work. In each case, the following simple paradigm has been applied:

compile the inner loop in PAM hardware, and let software handle the rest!

In what follows, we let $(a \div b)$ represent the quotient and $(a \cdot|\cdot b)$ the remainder in the integer division.

B.4.1 Long multiplication



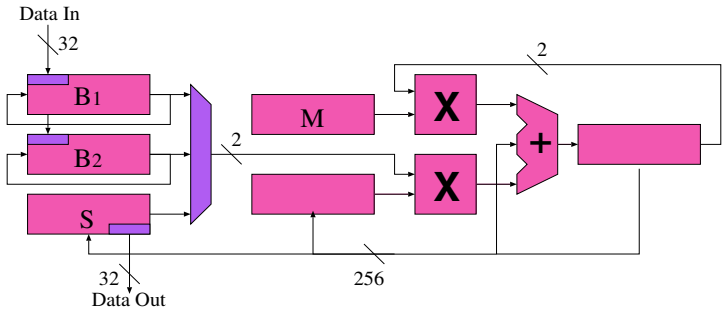
We have programmed both PAMs into long multipliers ($n = 512$ bits for P_0, and $n = 2K$ bits for P_1) computing $P = A \times B + S$, with A a n-bit multiplier, and B, S arbitrary size multiplicands and summands (see [Lyo76], [BRV89] and [SBV91]). These multipliers are interfaced with an arbitrary-precision arithmetic package *BigNum* (see [SVH89]) so that any program based on that software takes advantage of the PAM without modification, by simply relinking with a modified BigNum library. This respectively speeds up raw multiplication by a factor up to 24 (P_0) and 30 (P_1) for long operands, as compared to optimized assembly code running on the host workstation. For example, P_0 equipped with this design computes RSA encryption/decryption at 1500 bits per second for arbitrary 512-bit keys (see [RSA79]); this is about 10 times faster than our best software version on the same host.

The P_1 implementation produces product bits at 66 Mbits/s, which makes it faster than *any* known machine for which we could obtain benchmark measures. It is at least 16 times faster than the best figures for a Cray II or a Cyber 170/750 reported by [BW89]. This multiplier can be used to compute a 50 coefficient (16-bits) polynomial convolution (FIR filter) at 16 times *audio real time* (2×24 bits samples at 48 kHz).

B.4.2 RSA cryptography

To further investigate the tradeoffs which are possible in our hybrid hardware/software system we focused on the RSA cryptosystem (see [RSA79]), which can be cast entirely in terms of long multiplications. Starting from the above general-purpose multiplier, M. Shand from DEC-PRL implemented a series of hardware/software systems spanning two orders of magnitude in performance. The latest version is based on an original hardware design for computing modular products at the rate of two bits per cycle [SV93].

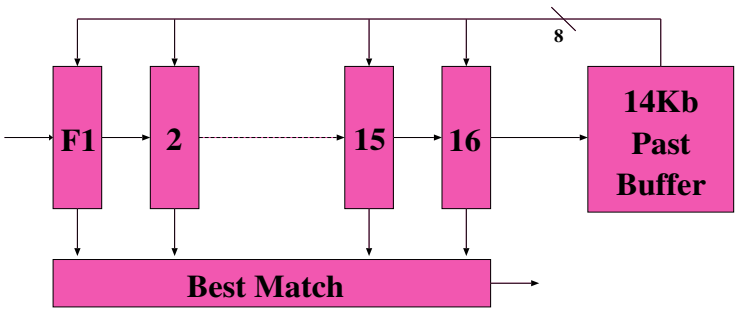
The system originally used three differently programmed P_0 boards, all operating in parallel with the host (see [SBV91]). At 200 kbit/s decoding speed, it was faster than *any* currently existing 512 bits RSA implementation, in *any* technology, as of February 1990. A survey by [Bri90] grants the previous speed record for 512 bits keys RSA decryption to a VLSI from AT&T, at 19 kbits/sec.



M. Shand recently ported this RSA system to a single P_1 board; at 40 MHz, this design provides either two independent 600 Kb/s RSA encryption channels for 480b keys, or one 175 Kb/s RSA encryption channel for 970b keys.

B.4.3 Data compression

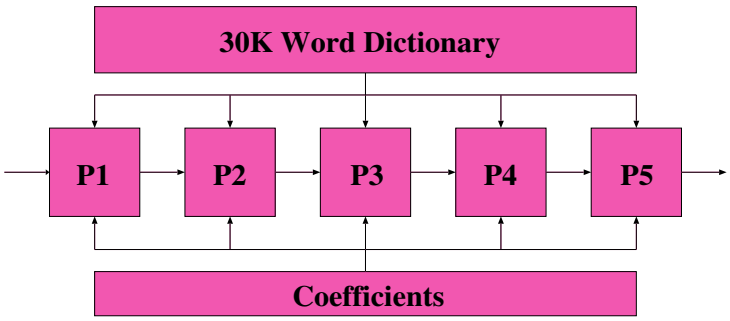
M. Skubiszewski from DEC-PRL has implemented a P_0 design to speed up the algorithm of [ZL77], which is well known to achieve an average data compression ratio varying from 2 for English (or French, or Polish...) plain text to 3 for C (or Lisp, or Pascal...) source code.



The design is a massively parallel method which computes 64 byte comparisons on each (70 ns) cycle; it matches the next 16 bytes in the file to be compressed against the last 4k bytes seen (stored in the local RAM), in order to detect the longest substring previously seen. While this design performs a respectable 1 Gops (8 bits integer comparison), it ends up in a disappointing factor two speed-up, when compared to optimized software such as Unix `compress`. Indeed, such optimized software avoids most of the comparisons performed in the hardware, by detecting early that they are irrelevant to the final output. A more elaborate hardware design is needed to genuinely speed up this particular algorithm.

B.4.4 String matching

Given an alphabet $\mathcal{A} = (a_1, \ldots, a_n)$, a probability $(S_{ij})_{i,j=1\ldots n}$ of substitution of a_i by a_j, and a probability $(I_i)_{i=1\ldots n}$ (resp. $(D_i)_{i=1\ldots n}$) of insertion (resp. deletion) of a_i, one can use a classical dynamic programming algorithm to compute a probability of transformation of w_1 into w_2; this defines a *distance* between any two words w_1 and w_2 over $\mathcal{A}$.

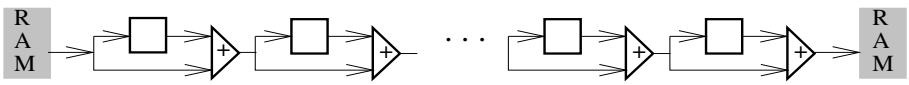


D. Lavenier from IRISA (Rennes, France) has implemented this algorithm with a P_0 design which computes the distance between an input word and all 30K words in a dictionary; it reports the k words found in the dictionary which are closest to the input. The system processes 200K words/sec: this is faster than a solution previously implemented at CNET using 12 Transputers; it has only half of the performance obtained by a system previously developed at IRISA, based on 28 custom VLSI chips and 2 PC boards.

Applications of this algorithm include automated mail sorting using OCR scanners, on-the-fly keyboard spelling corrections, and DNA sequence matching (see [Lop87]).

B.4.5 Heat and Laplace equations

[Vui93] shows how to adapt the classical finite difference method (see [FLS63]) to compute solutions of the heat and Laplace equations in n dimensions with help from special purpose hardware. An implementation of the method on P_1 operates with a pipe-line depth of 128 operators:



Each operator computes:

$$\mathcal{O}(v_0, v_1) = \begin{cases} v_0, \text{ when } v_0 \cdot|\cdot 2 = 1, \\ 2(v_0 \div 4 + v_1 \div 4 + (v_0 \cdot|\cdot 4) \div 2), \text{ otherwise,} \end{cases} \tag{B.1}$$

all with 24b fixed-point data format. At 20 MHz, this amounts to 5 Gops (24b adds, tests and shifts); it is easy to show (see [Vui93]) that fixed-point gives the same results as floating-point operations for this specific problems; the achieved performance thus exceeds those reported in [McB89] and [MF+91] for solving the same problem with super-computers. A sequential

computer needs to execute 25 billion instructions per second (25 Gips), to reproduce the same computation.

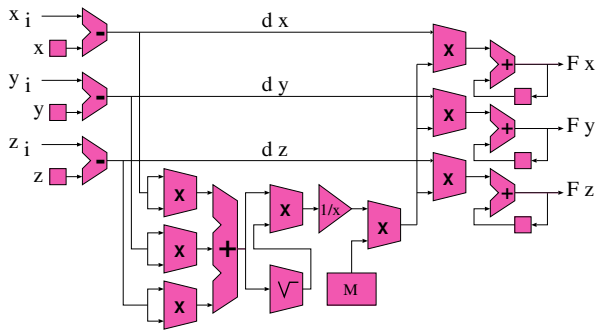
The heat and Laplace equations have many applications in mechanics, circuit technology, fluid dynamics, electrostatics, optics, finance, and so on.

B.4.6 Newton's mechanics

J. Vuillemin has specified a P_1 design for computing the evolution of a n-body system, using Newton's equations. The design computes the gravitational field acting on body k by summing the individual fields induced at k by each other body in the system. This amounts to the following 18 operations:

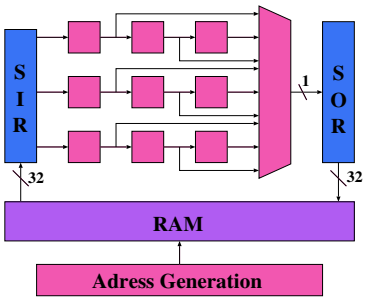
$(x_i - x_k) \Rightarrow dx$	$(dx2 + dz2) \Rightarrow dxz$	$fm \times dx \Rightarrow fdx$
$(y_i - y_k) \Rightarrow dy$	$(dxz + dy2) \Rightarrow d2$	$fm \times dy \Rightarrow fdy$
$(z_i - z_k) \Rightarrow dz$	$\sqrt{d2} \Rightarrow d$	$fm \times dz \Rightarrow fdz$
$(dx \times dx) \Rightarrow dx2$	$d \times d2 \Rightarrow d3$	$fx + fdx \Rightarrow fx$
$(dy \times dy) \Rightarrow dy2$	$\frac{1}{d3} \Rightarrow fd$	$fy + fdy \Rightarrow fy$
$(dz \times dz) \Rightarrow dz2$	$fd \times m_i \Rightarrow fm$	$fz + fdz \Rightarrow fz$

Positions and forces are represented as 20 bit floating-point numbers. Assuming a 40 ns internal cycle (achievable through deep pipe-line) the expected throughput exceeds 2.5 GFlops (this design has not been tested at publication time).



B.4.7 Binary 2D convolution

B. Chen and J. Vuillemin have implemented a 7×7 binary 2D convolver on P_0, for performing erosion, dilation and matching on black and white images, as defined in [Ser82].



The convolver runs at 25 MHz, generating one pixel each 40 ns; it completes a single convolution pass over one 512×512 image in 10 milliseconds; this allows for up to 4 successive operations (erosion, dilation, or matching) at video rate. Reproducing this performance through optimized software would require a 200 Mips computer.

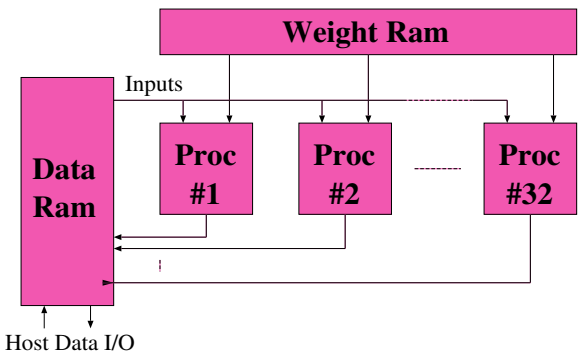
B.4.8 Boltzmann machine

M. Skubiszewski has implemented two successive versions of a hardware emulator for binary neural networks, based on the *Boltzmann machine* model (see [Sku90] and [Sku92]).

The Boltzmann Machine is a probabilistic algorithm which minimizes quadratic forms over binary variables, i.e. expressions of the form

$$E(\vec{N}) = \sum_{i=0}^{n-1} \sum_{j=0}^{i} w_{i,j} N_i N_j \tag{B.2}$$

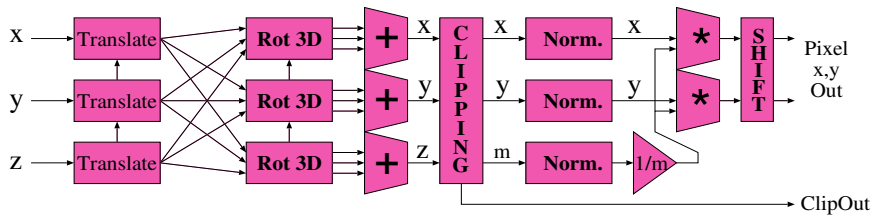
where $\vec{N} = (N_0, \ldots, N_{n-1})$ is a vector of binary variables and $(w_{i,j})_{0 \le i,j < n}$ is a fixed matrix of *weights*. It is typically used to find approximate solutions to $\mathcal{NP}$-hard problems, such as graph partitioning or circuit placement.



The latest realization, on P_1, can solve problems with up to 1400 variables, using 16-bit weights, for a total computing power of 500 *megasynapses per second* (the megasynapse is the traditional unit used in this field, it amounts to one million additions and multiplications, or one million terms of (B.2)).

B.4.9 3D Geometry

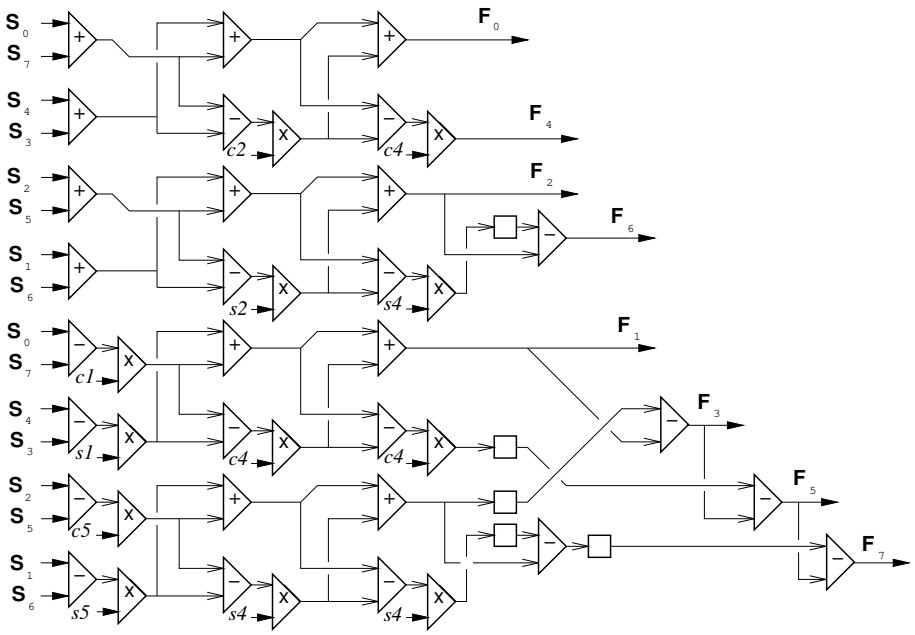
H. Touati from DEC-PRL has implemented a 3-D graphic accelerator for P_1, which supports translation, rotation, clipping and perspective projection, to directly compute the screen image of a cloud of points in 3-D space.



At 25MHz, it has a peak performance of 1.56 million points per second, using 16 bit fixed point coordinates for the input and output, and up to 32 bits for the intermediate results. One needs a 300 Mips processor to achieve the same throughput in software.

B.4.10 Discrete cosine transform

This design (by J. Vuillemin and D. Martineau on P_1) compresses a video stream in *real time* through multi-dimensional fast discrete cosine transform. The fDCT implements the following network:



The overall design computes 48 fixed-point (32 bit outputs) operations (add, subtract, multiply and shift) on each 40 ns cycle, for a total of 1.4 Gops. To match this performance through software would require a 15 Gips processor.

B.5 Conclusion

The following chart summarizes the practical PAM performance achieved by each of our ten designs:

Design	PABs	MHz	Gbops	Gops	Gips	PAM
Multiplier	8k	33	264	0.8	2.6	P_1
RSA	8k	32	256	0.5	4	P_1
DCT	10k	25	250	1.4	15	P_1
Newton	10k	25	250	2.5 †	5	P_1
Laplace	10k	20	200	7.5	20	P_1
Boltzmann	8k	25	200	1	1.5	P_1
3D Geometry	3k	25	75	0.5	0.7	P_1
Ziv-Lempel	3k	15	45	1	2	P_0
String	3k	10	30	0.15	0.3	P_0
2D convolution	1k	25	25		1	P_0

†These are Gflops, with 20 bit floating-point numbers.

The applications are ranked according to the most reliable performance measure, namely the *Gbops*. As a comparative measure of resource utilisation in such systems, the following table charts the maximum theoretical performance of generic PAM hardware (in Gbops) obtained by multiplying the maximal clock frequency (in MHz) by the area (in PABs):

PAM	Area	1 MHz	20 MHz	50 MHz
XC3020	128	0.1	2.5	6.4
XC3090	640	0.6	12.8	32
P_0	3.2K	3.2	64	
P_1	14K	14	280	700

Three years of PAM design lead us to believe the following:

1. For each of the chosen application, we have shown that the level of performance achieved with the PAM is comparable to the best figures reported using super-computer or custom silicon circuits.

   Our applications have been carefully selected for having a clearly identified (PAM implementable) inner-loop, which accounts for a vast percentage of the software run-time. For such low level processing, the PAM proves more cost effective than any super-computer.

   Due to their software complexity, many current super-computer applications still remain outside the possibilities of current PAM technology.

2. Each mentioned PAM design was implemented and tested within one or two months, starting from the delivery of the specification software. This is roughly equivalent to the time it takes to implement a *highly optimized* software version of the same system with a super-computer; both are technically challenging, yet remain an order of magnitude faster than the time it takes to cast a system into silicon.

3. The cost of P_1 is comparable to that of a high-end workstation. This is orders of magnitude lower than the cost of a super-computer. Based on figures from [McB89], we find that the price (in $ per operation per second) of solving the heat and Laplace equations is 100 times higher with super-computers than with the PAM.

4. Another field of applications, not covered by any existing supercomputer, is open to PAM technology: high-bandwidth interfaces to the external world, with *fully programmable real-time* capabilities. The P_1 PAM has a 256b wide connector, capable to deliver up to 6.4 Gb/s of external bandwidth. It is a "simple matter of hardware programming" to interface directly with any electrically-compatible external device, by programming its communication protocol into the PAM itself. Applications for this capability are numerous, including interfaces to high-bandwidth networks, audio and video input or output devices, and on-the-fly data acquisition and filtering.

References

[Ber92] Gérard Berry, A Hardware Implementation of Pure Esterel. *Academy Proceedings in Engineering Sciences, Indian Academy of Sciences*, Sadhana, 17:1:95–130, 1992. Also available as *PRL report 15*, Digital Equipment Corporation, PRL, 85, avenue Victor Hugo, Rueil Malmaison, France.

[BRV89] Patrice Bertin, Didier Roncin and Jean Vuillemin, Introduction to Programmable Active Memories. *Systolic Array Processors, J. McCanny, J.McWhirther, E. Swartslander Jr. editors*, Prentice Hall, 300–309, 1989. Also available as *PRL report 3*, Digital Equipment Corp., Paris Research Lab., 85, Av Victor Hugo. 92563 Rueil Malmaison Cedex, France, 1989.

[Bri90] E.F. Brickell, A survey of hardware implementations of RSA. *Proceedings of Crypto'89, Lecture Notes in Computer Science*, Springer Verlag, 1990.

[BW89] D.A. Buell and R.L. Ward, A multiprecise integer arithmetic package. *The Journal of Supercomputing, Kluwer Academic Publishers*, Boston, 3:89–107, 1989.

[DEC91] Digital Equipment Corporation, TURBOchannel hardware specification. *DEC document EK-369AA-OD-007A*, Digital Equipment Corp., 1991.

[FLS63] R.P. Feynman, R.B. Leighton and M. Sands, *The Feynman lectures on PHYSICS*. Addison-Wesley, 1963.

[GK89] J.P. Gray and T. Kean, Configurable hardware: two case studies of micro-grain computation. *Systolic Array Processors, J. McCanny, J.McWhirther, E. Swartslander Jr. editors*, Prentice Hall, 310–319, 1989.

[HP90] J.L. Hennessy and D.A. Patterson, *Computer architecture: a quantitative approach. Morgan Kaufmann Publishers, Inc*, 1990.

[Lyo76] R.F. Lyon, Two's complement pipeline multipliers. *IEEE Trans. Comm.*, 24:418–425, 1976.

[Lop87] D.P. Lopresti, P-NAC: A systolic array for comparing nucleic acid sequences. *Computer Magazine*, 20(7):98–99, 1987.

[McB89] O.A. McBryan, Connection machine application performance. In *Scientific Applications of the Connection Machine, World Scientific*, 94–114, 1989.

[MF+91] O.A. McBryan, P.O. Frederickson, J. Linden, A. Shüller, K. Solchenbach, K. Stüben, C.A Thole and U. Trottenberg, Multigrid methods on parallel computers - a survey of recent developments. *Impact of Computing in Science Engineering, Academic Press*, 3(1):1–75, 1991.

[RSA79] R.L. Rivest, A. Shamir and L. Adleman, Public key cryptography. *CACM 21*, 120–126, 1979.

[Ser82] J.P. Serrat, Image Analysis and Mathematical Morphology. *Academic Press, N.Y.*, 1982.

[SVH89] B. Serpette, J. Vuillemin and J.-C. Hervé, BigNum: a portable efficient package for arbitrary-precision arithmetic. *PRL report 2*, Digital Equipment Corp., Paris Research Lab., 85, av. Victor Hugo. 92563 Rueil Malmaison Cedex, France, 1989.

[Sku90] Marcin Skubiszewski, A hardware emulator for binary neural networks. *Proceedings of the International Neural Network Conference, Paris*, 2:555–558, 1990.

[Sku92] Marcin Skubiszewski, An exact hardware implementation of the Boltzmann Machine. *PRL report in preparation*, Digital Equipment Corp., Paris Research Lab., 85, av. Victor Hugo. 92563 Rueil Malmaison Cedex, France, 1992.

[SBV91] Mark Shand, Patrice Bertin and Jean Vuillemin, Hardware speedups in long integer multiplication. *Computer Architecture News*, 19(1):106–114. 1991.

[SV93] Mark Shand and Jean Vuillemin, Fast Implementations of RSA Cryptography. *11th IEEE Symposium on COMPUTER ARITHMETIC*, 1993.

[Tou92] Hervé Touati, Perle1DC: a C++ library for the simulation and generation of DECPeRLe-1 designs. *PRL Technical Note 4*, Digital Equipment Corp., Paris Research Lab., 85 av. Victor Hugo, 92563 Rueil Malmaison, France, 1992.

[Vui93] Jean E. Vuillemin, Contribution à la résolution numérique des équations de Laplace et de la chaleur. To appear in Mathematical Modelling and Numerical Analysis, RAIRO, 1993. Also available as *PRL report 16*, Digital Equipment Corp., Paris Research Lab., 85 av. Victor Hugo, 92563 Rueil Malmaison, France, 1992.

[Xil87] Xilinx, Inc., *The Programmable Gate Array Data Book. Product Briefs, Xilinx, Inc.*, 1987.

[ZL77] J. Ziv and A. Lempel, A universal algorithm for sequential data compression. *IEEE Trans. on Information Theory*, IT-23(3):337-343, 1977.

Bibliographie

[ABD92] J. Arnold, D. Buell and E. Davis, *Splash II*, in *4th ACM Symposium on Parallel Algorithms and Architectures*, San Diego, California, USA (1992).

[Act91] Actel, *ACT Family FPGA Databook*, Actel Corporation, Sunnyvale, CA, USA (1991).

[Alg90] Algotronix Ltd., *The Configurable Logic Data Book*, Edinburgh, UK (1990).

[Apt92] Aptix Corporation, *FPIC AX1024D Preliminary Data Sheet, v1.0*, Aptix Corporation, 225, Charcot Ave., San Jose CA 95131, USA (1992).

[Atm91] Atmel Corporation, *Integrated Circuit Data Book*, Atmel Corporation, San Jose, CA, USA (1991).

[BC84] G. Berry and L. Cosserat, *The synchronous programming language Esterel and its mathematical semantics*, in *Seminar on Concurrency*, S. Brookes and G. Winskel, editors, Springer-Verlag Lecture Notes in Computer Science 197, 389–448 (1984).

[BCM90] C. Berthet, O. Coudert and J. C. Madre, *New Ideas on Symbolic Manipulations of Finite State Machines*, in *IEEE International Conference on Computer Design*, October 1990.

[Bee91] J. F. Beetem, *Simultaneous placement and routing of the Labyrinth reconfigurable logic array*, in *FPGAs*, W. R. Moore and W. Luk editors, 232–243, Abingdon EE&CS Books, Abingdon, UK (1991).

[Ber91] G. Berry, *A Hardware Implementation of Pure Esterel*, PRL report 15, Digital Equipment Corporation, Paris Research Laboratory, 85, av. Victor-Hugo, 92563 Rueil-Malmaison Cedex, France (1991).

[BMT92] D. E. Van den Bout, J. N. Morris, D. Thomae, S. Labrozzi, S. Wingo and D. Hallman, *AnyBoard: An FPGA-Based, Reconfigurable System*, in *IEEE Design & Test of Computers*, 21–30 (Sept. 1992).

[BP91] O. Bigalli and F. Pirri, *HPGI: A Hardware Programmable Generic Interface for the IBM PC*, in *FPGAs*, W. R. Moore and W. Luk editors, 408–413, Abingdon EE&CS Books, Abingdon, UK (1991).

[BRV89] P. Bertin, D. Roncin and J. Vuillemin, *Introduction to Programmable Active Memories*, in *Systolic Array Processors*, J. McCanny, J. McWhirter, E. Swartzlander Jr. editors, pp 301–309, Prentice-Hall (1989). Aussi disponible en tant que PRL report 3, Digital Equipment Corporation, Paris Research Laboratory, 85, av. Victor-Hugo, 92563 Rueil-Malmaison Cedex, France (1989). En annexe.

[BRV93] P. Bertin, D. Roncin and J. Vuillemin, *Programmable Active Memories: a Performance Assessment*, in *Symposium on Integrated Systems*, Seattle, WA, USA, March 1993, MIT Press (1993). Aussi disponible en tant que PRL report 24, Digital Equipment Corporation, Paris Research Laboratory, 85, av. Victor-Hugo, 92563 Rueil-Malmaison Cedex, France (1993). En annexe.

[Bry86] R. E. Bryant, *Graph-based Algorithms for Boolean Functions Manipulation*, in *IEEE Transactions on Computers*, C-35(8):677–691 (1986).

[BSS88] R. Barth, B. Serlet and P. Sindhu, *Integrating Schematics and Programs for Design Capture*, in *Advanced Research in VLSI*, proceedings of the 1988 MIT conference, MIT Press, 351–366 (1988).

[BT93] G. Berry and H. Touati, *Optimized Controller Synthesis Using Esterel*, in *International Workshop on Logic Synthesis*, Lake Tahoe, CA, USA (May 1993).

[Bur92] D. Bursky, *RAM-Based logic arrays up density, cut delays*, in *Electronic Design*, 40(20):45–49 (October 1992).

[BV93] F. Bourdoncle et J. Vuillemin, *The 2Z Report*, rapport de recherches en préparation, Digital Equipment Corporation, Laboratoire de Recherches de Paris, 85, av. Victor-Hugo, 92563 Rueil-Malmaison Cedex, France (1993).

[Cas93] S. Casselman, *Virtual Computing*, in *IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, California, USA (April 1993).

[CD+86] W. S. Carter, K. Duong, R. H. Freeman, H. C. Hsieh, J. Y. Ja, J. E. Mahoney, L. T. Ngo and S. L. Sze, *A user programmable reconfigurable logic array*, in *Proc. IEEE 1986 Custom Integrated Circuits Conference*, 233–235 (1986).

[Cha86] J. Chailloux, M. Devin, J.-M. Hullot, B. Serpette et J. Vuillemin, *Le-Lisp version 15.2: le manuel de référence*, I.N.R.I.A., Domaine de Voluceau, Rocquencourt, 78153 Le Chesnay Cedex (1986).

[CH+85] P. Caspi, N. Halbwachs, D. Pilaud and J. Plaice, *Lustre: a declarative language for programming synchronous systems*, in *Proc. 14th Annual ACM Symposium on Principles of Programming Languages* (1987).

[CLi92] Concurrent Logic, Inc., *Cli6000 Series Field-Programmable Gate Arrays*, Concurrent Logic Inc., 1270 Oakmead Parkway, Sunnyvale, CA94086, USA (1992).

[CS93] P. K. Chan and M. D. F. Schlag, *Architectural Tradeoffs in Field-Programmable-Device-Based Computing Systems*, in *IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, California, USA (April 1993).

[CSM92] P. K. Chan, D. F. Schlag and M. Martin, *BORG: A Reconfigurable Prototyping Board using Field-Programmable Gate Arrays*, in *1st International ACM Workshop on Field-Programmable Gate Arrays*, 47–51, Berkeley, CA, USA (1992). Also available as UCSC Technical Report UCSC-CRL-91-45, University of California, Santa Cruz, CA 95064, USA (1991).

[DEC91] Digital Equipment Corporation: *TURBOchannel Hardware Specification*, Digital Equipment Corporation, TRI/ADD Program, 100, Hamilton Ave., Palo Alto, CA 94301, USA (1991).

[DE+92] I. Dobbelaere, A. El Gamal, D. How and B. Kleveland, *Field-programmable MCM systens – Design of an interconnection frame*, in *1st International ACM Workshop on Field-Programmable Gate Arrays*, 52–56, Berkeley, CA, USA (1992).

[DM92] J. Dion and L. Monier, *Design Tools for BIPS-0*, Technical note 32, Digital Equipment Corporation, Western Research Laboratory, 250 University Ave., Palo-Alto CA 94301, USA (1992).

[ES86] M. A. Ellis and B. J. Stroustrup, *The Annotated C++ Reference Manual*, Addison Wesley (1986).

[FCCM93] *IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, California, USA (April 1993).

[FSJ90] F. Furtek, G. Stone and I. Jones, *Labyrinth: a homogeneous computational medium*, in *Proc. IEEE 1990 Custom Integrated Circuits Conference*, paper 31.1 (1990).

[GK+90] M. B. Gokhale, A. Kopser, A. P. Lucas and R. G. Minnich, *The Logic Description Generator*, in *Int. Conf. on Application-Specific Array Processing* (1990). Also available as SRC Technical Report SRC-TR-90-011, Supercomputing Research Center, 17100 Science Drive, Bowie MD 20715, USA (1990).

[GHK91] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, A. Sweely and D. Lopresti, *Building and using a highly parallel programmable logic array*, Computer 24(1):81–89 (1991).

[Haw91] D. Hawley, *Advanced PLD architectures*, in *FPGAs*, W. R. Moore and W. Luk editors, 11–23, Abingdon EE&CS Books, Abingdon, UK (1991).

[HB+92] D. D. Hill, B. K. Britton, B. Oswald, N. S. Woo, S. Singh, T. Poon, B. Krambeck, *A new architecture for high-performance FPGAs*, in *2nd International Workshop on Field-Programmable Logic and Applications*, paper 2.1, Vienna, Austria (1992).

[HC+90] H. C. Hsieh, W. S. Carter, J. Ja, E. Cheung, S. Schreifels, C. Erikson, P. Freidin, L. Tinkey and R. Kanazawa, *Third-generation architecture boosts speed and density of field-programmable gate arrays*, in *Proc. IEEE 1990 Custom Integrated Circuits Conference*, paper 31.2 (1990).

[HD+87] H. C. Hsieh, K. Duong, J. Y. Ja, R. Kanazawa, L. T. Ngo, L. G. Tinkey, W. S. Carter and R. H. Freeman, *A second generation user-programmable gate array*, in *Proc. IEEE 1987 Custom Integrated Circuits Conference*, 515–521 (1987).

[HP90] J. L. Hennessy and D. A. Patterson, *Computer Architecture: a Quantitative Approach*, Morgan Kaufman (1990).

[HP92] B. Heeb and C. Pfister, *Chameleon, a Workstation of a different colour*, in *2nd International Workshop on Field-Programmable Logic and Applications*, paper 5.6, Vienna, Austria (1992).

[ICD92] IC Designs, *ICD2053A Programmable Clock Oscillator*, in *Standard Components Data Book*, IC Designs, 12020, 113th Ave. NE, Kirkland, WA 98034, USA (1992).

[ICu93] I-Cube Design Systems, *IQ160 Field Programmable Interconnect Device*, I-Cube Design Systems, 2328-C Walsh Ave., Santa Clara CA 95051, USA (1993).

[InC90] InCA, *Personal Prototyping Machine PPM 1000, Reference Manual*, Integrated Circuit Applications Ltd., Gibbs House, Kennel Ride, Ascot, Berkshire SL5 7NT, UK (1990).

[ISO90] ISO/IEC, *Langages de programmation – C*, Standard international ISO/IEC 9899:1990, ISO/IEC, Case Postale 56, CH-1211 Genève 20, Confédération helvétique (1990).

[Jen91] J. Jenkins, *Programmable Macro Logic: the PML 2852*, in *FPGAs*, W. R. Moore and W. Luk editors, 24–33, Abingdon EE&CS Books, Abingdon, UK (1991).

[KB92] T. Kean and I. Buchanan, *The Use of FPGAs in a Novel Computing Subsystem*, in *1st International ACM Workshop on Field-Programmable Gate Arrays*, 60–66, Berkeley, CA, USA (1992).

[KG89] T. A. Kean and J. P. Gray, *Configurable hardware: two case studies of micro-grain computation*, in *Systolic Array Processors*, J. McCanny, J. McWhirter, E. Swartzlander Jr. editors, 310–319, Prentice-Hall (1989).

[Kun82] H. T. Kung, *Why Systolic Architectures?*, in *Computer*, 15(1):37–46 (1982).

[Lat92] Lattice, *pLSI and ispLSI Data Book and Handbook*, Lattice Semiconductor Corporation, 5555, Northeast Moore Ct., Hillsboro, OR 97124, USA (1992).

[LC91] G. Larrabee and P. Chatterjee, *DRAM Manufacturing in the '90s*, in *Semiconductor International*, 84–101, May 1991.

[LS91] C. Leiserson and J. Saxe, *Retiming Synchronous Circuitry*, in *Algorithmica*, 6(1):5–35, 1991.

[MC92] D. Marple and L. Cooke, *An MPGA compatible FPGA architecture*, in *1st International ACM Workshop on Field-Programmable Gate Arrays*, 39–44, Berkeley, CA, USA (1992).

[Mea89] C. Mead, *Analog VLSI and Neural Systems*, Addison-Wesley (1989).

[MV85] L. Monier and J. Vuillemin, *Using Silicon Assemblers*, in *International Conference on Very Large Scale Integration*, E. Hörbst, editor, Tokyo, Japan, August 1985.

[Pet89] W. D. Peterson, *The VMEbus Handbook: a User's Guide to the IEEE 1014 and IEC 821 Microcomputer Bus*, VMEbus International Trade Association, 10229 N. Scottsdale Road, Suite E, Scottsdale, AZ 85253, USA (1989).

[Pfi91] C. Pfister, *The LABRYS Project*, in *FPGAs*, W. R. Moore and W. Luk editors, 223–231, Abingdon EE&CS Books, Abingdon, UK (1991).

[PL91] I. Page and W. Luk, *Compiling OCCAM into FPGAs*, in *FPGAs*, W. R. Moore and W. Luk editors, 271–283, Abingdon EE&CS Books, Abingdon, UK (1991).

[Ple91] GEC Plessey Semiconductors, *ERA60100 Electrically Reconfigurable Array Data Sheet*, GEC Plessey Semiconductors Ltd., Swindon, Wiltshire SN2 2QW, UK (1991).

[Pot91] B. Pottier, *ArMen: une machine parallèle intégrant un réseau de circuits logiques programmables*, thèse de l'Université de Rennes I (1991).

[Qui91] Quickturn Systems, Inc., *RPM Emulation System Data Sheet*, Quickturn Systems, Inc., 325 East Middlefield Road, Mountain View, CA 94043, USA (1991).

[RLW85] P. Rovner, R. Levin and J. Wick, *On Extending Modula-2 for building large, integrated systems*, SRC report 3, Digital Equipment Corporation, Systems Research Center, 130, Lytton Ave., Palo-Alto CA 94301, USA (1985).

[Roc92] F. Rocheteau, *Extension du langage Lustre et application à la conception de circuits: le langage Lustre V4 et le système Pollux*, thèse de l'Université de Grenoble (juin 1992).

[SBV91] M. Shand, P. Bertin and J. Vuillemin, *Hardware Speedups in Long Integer Multiplication*, in *Computer Architecture News*, 19(1):106–114 (1991).

[Ser84] B. Serlet, *Description structurelle et simulation des circuits intégrés*, thèse de 3$^{\text{ème}}$ cycle de l'Université d'Orsay (1984).

[SGL92] M. J. S. Smith, T. Gilley, K. Lau, J. Otaguro, S. Jilla, M. Lwin and L. Xu, *Reprogrammable Hardware for the Apple Macintosh*, in *2nd International Workshop on Field-Programmable Logic and Applications*, paper 2.5, Vienna, Austria (1992).

[Sku90] M. Skubiszewski, *A Hardware Emulator for Binary Neural Networks*, in *Proceedings of the International Neural Network Conference*, 2:555–558, Paris, France (1990).

[Sku92] M. Skubiszewski, *An Exact Hardware Implementation of the Boltzmann Machine*, in *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, Arlington, Texas, USA (December 1992).

[SS+92] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton and A. Sangiovanni-Vincentelli, *Sequential Circuit Design Using Synthesis and Optimization*, in *IEEE International Conference on Computer Design* (1992).

[STB91] H. Savoj, H. Touati and R. K. Brayton, *Extracting Local Don't Cares for Network Optimization*, in *International Conference on Computer-Aided Design*, November 1991.

[SV93] M. Shand and J. Vuillemin, *Fast Implementations of RSA Cryptography*, in *11th IEEE Symposium on Computer Arithmetic*, Windsor, Ontario, Canada, (1993).

[Tou93] H. Touati, *Perle1DC: a C++ library for the Simulation and Generation of DECPeRLe-1 Designs*, PRL technical note 4, Digital Equipment Corporation, Paris Research Laboratory, 85, av. Victor-Hugo, 92563 Rueil-Malmaison Cedex, France (1993).

[VA92] S. S. Vij and B. Ahanin, *A high-density, high-speed, array-based erasable programmable logic device with programmable speed-power optimization*, in *1st International ACM Workshop on Field-Programmable Gate Arrays*, 29–32, Berkeley, CA, USA (1992).

[Vui87] J. Vuillemin, *L'Ordinateur Modulaire*, communication privée (1987).

[Vui92] J. Vuillemin, *Contribution à la résolution numérique des équations de Laplace et de la chaleur*, PRL report 16, Digital Equipment Corporation, Paris Research Laboratory, 85, av. Victor-Hugo, 92563 Rueil-Malmaison Cedex, France (1992).

[Vui93] J. Vuillemin, *On Circuits and Numbers*, PRL report 25, Digital Equipment Corporation, Paris Research Laboratory, 85, av. Victor-Hugo, 92563 Rueil-Malmaison Cedex, France (1993).

[Woo91] N. S. Woo, *A study on the structure of the intermediate network in an FPGA technology mapping*, in *FPGAs*, W. R. Moore and W. Luk editors, 170–178, Abingdon EE&CS Books, Abingdon, UK (1991).

[WS+89] S. C. Wong, H. C. So, J. H. Ou and J. Costello, *A 5000-gate CMOS EPLD with multiple logic and interconnect arrays*, in *IEEE 1989 Custom Integrated Circuits Conference*, paper 5.8, 1989.

[Xil91] Xilinx, *User Guide and Tutorials*, Xilinx, 2100 Logic Drive, San Jose, CA 95124, USA (1991).

[Xil92] Xilinx, Inc., *The Programmable Gate Array Data Book*, Xilinx, 2100 Logic Drive, San Jose, CA 95124, USA (1992).

[Xil92b] Xilinx, Inc., *The XC4000 Data Book*, Xilinx, 2100 Logic Drive, San Jose, CA 95124, USA (1992).

[Xil92c] Xilinx, Inc., *The EPLD Data Book*, Xilinx, 2100 Logic Drive, San Jose, CA 95124, USA (1992).