Faculté des Sciences Appliquées
Laboratoire de Microélectronique

Architecture synchronisée par les données pour système reconfigurable

Jean Pierre DAVID

Thèse soutenue en vue de l'obtention du grade de Docteur en Sciences Appliquées

Composition du jury :

Prof. Jean-Didier Legat (UCL/DICE) - *Promoteur*
Prof. Jean-Luc Danger (ENST, Paris, France)
Prof. Christian Piguet (CSEM, Neuchâtel, Suisse)
Prof. Jean-Jacques Quisquater (UCL/DICE)
Prof. Piotr Sobieski (UCL/TELE) - *Président*
Prof. Anne-Marie Trullemans (UCL/DICE)
Prof. Luc Vandendorpe (UCL/TELE)

Louvain-la-Neuve, Belgique
Juin 2002

Table des matières

Table des figures 7

Remerciements 11

Chapitre 1. Introduction 13

Chapitre 2. Etat de l'art 23
1. Les processeurs orientés flux de données 23
1.1. Introduction 23
1.2. La machine à jetons uniques (flux de données statique) 28
1.3. La machine à jetons colorés (flux de données dynamique) 30
1.4. La machine à stock de jetons explicite 32
2. Les architectures reconfigurables à grain large 32
2.1. Introduction 32
2.2. Les architectures sans processeur 35
2.2.1. ALFA 35
2.2.2. MATRIX 36
2.2.3. PipeRench 40
2.2.4. RaPiD 41
2.2.5. Xputer 44
2.3. Les architectures avec processeur 45
2.3.1. Chameleon 45
2.3.2. FPPA et E-FPPA 47
2.3.3. MorphoSys 50
2.3.4. Raw Machines 51
2.4. Une architecture particulière 53
2.5. Conclusion 56
3. Les architectures reconfigurables à grain fin 61
3.1. Introduction 61
3.2. Les FPGA fabriqués par Altera 64
3.3. Les FPGA fabriqués par Xilinx 67
3.4. Les systèmes multi-FPGA 70
3.5. Conclusion 74

Chapitre 3. Une architecture nouvelle, utilisant des "DDR" 77

1. Introduction 77
2. Les cartes réalisées 79
2.1. La carte 400.000 portes 79
2.2. La carte Virtex 80
3. Une architecture synchronisée par les données 82
3.1. Du concept de réseau de Pétri aux DDR 82
3.2. Le protocole de synchronisation 86
3.3. Une bibliothèque de DDR 91
3.3.1. Introduction 91
3.3.2. Les DDR de traitement de données 92
3.3.3. Les DDR de transfert de données 92
3.3.4. Les DDR de génération de données 96
3.3.5. Les DDR de stockage de données 99
3.3.6. Les DDR de test 105
3.3.7. Les DDR d'entrées/sorties 105
3.3.8. Les DDR spécialisées 105
3.3.9. Conclusion 106
4. Conception d'une architecture DDR 106
4.1. Introduction 106
4.2. La méthode heuristique 108
4.3. Exemple d'application 110
5. L'environnement de développement 113
5.1. Introduction 113
5.2. La saisie de l'architecture DDR 115
5.2.1. Les trois niveaux de hiérarchie 115
5.2.2. Le format VHDL des DDR 116
5.2.3. Le logiciel de génération du VHDL 117
5.3. La simulation 118
5.4. Le test 120
5.5. Conclusion 121

Chapitre 4. Applications - Résultats 123
1. Introduction 123
2. Un exemple simple avec les DDR : La convolution 125
2.1. Approche empirique 126
2.2. Approche basée sur les DDR 128
3. Prototypage rapide 130
3.1. Un algorithme de watermarking 131
3.2. Un algorithme de cryptanalyse 133
4. Les DDR utilisateur : La transformée multi-résolution 135
5. Calcul parallèle : application en robotique 137
5.1. Introduction 137

5.2. Le module BDC 138
5.3. Résultats 142
5.3.1. Caractéristiques des schémas de calcul 142
5.3.2. Le robot APRA 143
5.3.3. Le robot PUMA 145
5.3.4. Un dernier schéma de calcul 147
5.4. Conclusion 147
6. Conclusion 149

Chapitre 5. Conclusion 153
1. Originalité 155
2. Avantages 156
3. Inconvénients 157
4. Perspectives 158
5. L'avenir des systèmes digitaux 159

Publications de l'auteur 161

Références bibliographiques 163

Annexe 1 : Exemple de code VHDL 171
La synchronisation des DDR de traitement de données 171
La DDR additionneur 173

Annexe 2 : Exemple de C pour DDR 177
Déclaration de la carte 177
Déclaration des ressources 178
Génération de l'architecture DDR 178

Annexe 3 : Détail des cartes 181
La carte multi-FPGA 181
La carte PCI-VIRTEX 183

Table des figures

1	Le schéma de calcul comparé à l'approche à contrôle dominant	24
2	Illustration du concept de réseau de Pétri	26
3	Les opérations élémentaires orientées flux de données	27
4	La machine à jetons uniques développée au MIT	28
5	Implémentation de l'exemple de schéma de calcul sur la machine à jetons uniques.	29
6	La machine à jetons colorés développée à l'université de Manchester	31
7	La machine à stock de jetons explicite Monsoon développée au MIT	33
8	Vue générale du système ALFA	35
9	Vue détaillée d'un cluster sur ALFA	36
10	Architecture des unités de traitement dans MATRIX	37
11	Le réseau de connexions dans MATRIX	38
12	L'architecture des ports dans MATRIX	39
13	Architecture PipeRench	40
14	Les unités de traitement de l'architecture PipeRench	41
15	Une unité de traitement de "RaPiD-1"	42
16	Détail de RaPiD-1	43
17	Architecture générale Xputer	44
18	Architecture d'une ALU reconfigurable dans le Xputer	45
19	L'architecture CHAMELEON	46
20	Architecture des unités de traitement de CHAMELEON	47
21	L'architecture FPPA développée au CSEM	48
22	L'architecture E-FPPA élaborée à l'UCL	49

23	Un contrôleur de transferts dans l'architecture E-FPPA	50
24	L'architecture Morphosys	50
25	Le réseau d'unités de traitement de Morphosys	52
26	Les unités de traitement reconfigurables de MorphoSys	52
27	L'architecture RAW développée au MIT	53
28	Exemple d'architecture TTA : le processeur MOVE32INT	55
29	Format des instructions du processeur MOVE32INT	56
30	Les circuits PAL (gauche) et PLA (droite)	63
31	Schéma simplifié de l'architecture MAX (Altera)	65
32	Schéma simplifié de l'architecture FLEX (Altera)	65
33	Schéma simplifié de l'architecture APEX (Altera)	67
34	Schéma simplifié de l'architecture XC3000 (Xilinx)	68
35	Schéma simplifié de l'architecture XC4000 (Xilinx)	69
36	Un système digital (multi-)FPGA	71
37	Quatre types d'interconnexions	72
38	L'architecture "Enhanced Mesh Topology"	79
39	Le concept d'unité de traitement	80
40	Architecture de la carte PCI Virtex	81
41	Example d'un réseau de Pétri	83
42	Example de flambage des transitions sur un réseau de Pétri	84
43	Du réseau de Pétri aux DDR	84
44	Architecture d'une DDR	85
45	Une chaine de DDR	86
46	Le protocole de synchronisation	88
47	Exemple d'implémentation du protocole de synchronisation : un opérateur mathématique à deux opérandes	88
48	Utilisation d'une boucle combinatoire à bon escient	89
49	Les DDR de type traitement de données	92
50	Les DDR mux, demux et copy	93
51	Exemple d'une architecture combinant des sources et des puits de données	94

52 Les ressources autonomes "dispatcher" et "selecter" 96
53 Exemple d'une DDR de type générateur arithmétique de données 97
54 La FIFO et la FIFO étendue 101
55 Exemple d'une architecture utilisant la FIFO étendue comme mémoire cache 103
56 Architecture d'une mémoire à accès partagés 104
57 Architecture générique pour l'implémentation d'un algorithme au moyen de DDR 107
58 Schéma des accès à la mémoire globale 112
59 Les étapes de la réalisation d'une architecture DDR 114
60 Exemple de simulation d'une architecture à DDR 119
61 L'interface graphique utilisée pour le test 121
62 Les deux implémentations de la convolution 126
63 Implémentation empirique d'une convolution 127
64 Implémentation d'une convolution au moyen de l'architecture DDR 129
65 Utilisation des DDR pour le prototypage rapide de l'algorithme du watermarking 132
66 Utilisation des DDR pour le prototypage rapide de la cryptanalyse du DES 134
67 Structure de filtre en treillis pour la transformée multi-résolution 136
68 Le module BDC pour le calcul parallèle de larges schémas de calcul 139
69 Les 4 BDC sur notre architecture de test 140
70 Exemple d'un ordonnancement réalisé par ROBOTRAN 141
71 Cycles d'horloge pour le robot APRA 143
72 Cycles opérateurs pour le robot APRA 144
73 Coût des transferts pour le robot APRA 144
74 Cycles d'horloge pour le robot PUMA 145
75 Cycles opérateurs pour le robot PUMA 146
76 Coût des transferts pour le robot PUMA 146
77 Evolution d'une BDC étendue à 5 opérateurs 149

Remerciements

Je reverrai toujours mon premier entretien d'embauche pour un poste de chercheur au Laboratoire de Microélectronique de l'Université Catholique de Louvain. Le professeur C. Trullemans tentait de voir si je serais capable de travailler sur une carte multi-DSP dont le FPGA de contrôle avait grillé, au sens littéral du terme. Ne comprenant pas de quoi il parlait, je posai une question qui aurait bien pu être fatidique : "C'est quoi, un FPGA ?" J'étais bien de loin d'imaginer un seul instant que je pourrais plus tard écrire une thèse sur ce type de circuits. Heureusement, il enchaîna immédiatement "C'est une sorte de grosse PAL", ce qui rassura tout le monde car je connaissais bien ces circuits ...

Le chemin qui a été parcouru depuis lors est le fruit de relations avec bien des personnes. Sans les nommer, je pense d'abord aux chercheurs qui m'ont accueilli au laboratoire et ont patiemment répondu aux questions que j'avais sur le système informatique, l'utilisation des logiciels et autres matériels de test. C'est à eux que je dois d'avoir accepté, un an après mon arrivée au laboratoire, de commencer une thèse de doctorat en tant qu'assistant. Merci à tous ces collègues qui étaient là avant moi, à ceux qui sont arrivés plus tard, à ceux qui sont déjà partis. Bonne chance et courage à ceux qui restent. Merci Tony, pour le travail que nous avons pu réaliser ensemble, d'abord au niveau de la recherche mais aussi dans l'encadrement du projet intégré sur les robots.

Certains travaillent de façon plus cachée. Bien qu'elles ne liront probablement jamais ces lignes, je voudrais remercier le personnel qui entretient les locaux et inviter le lecteur à partager mon admiration pour ces personnes qui réalisent des tâches plus ingrates pour nous permettre de vivre dans un environnement propre et accueillant. Mon admiration et mes remerciements vont aussi au personnel technique et administratif. Merci Pierrot pour l'accueil simple et amical que tu m'as toujours réservé quand j'avais un problème mécanique à régler. Merci Bernard pour l'accueil chaleureux dont tu as toujours fait preuve quand j'avais besoin de tes services pour moi-même ou pour les étudiants, en sacrifiant au besoin une partie de ton week-end. Merci Brigitte et Pascal pour les

nombreuses fois où j'ai pu faire appel à vos compétences et compter sur votre dévouement pour solutionner des problèmes informatiques. Merci Anne et Marie-Hélène pour les nombreuses commandes en urgence que vous avez encodées dans les plus brefs délais et pour tous les petits services que vous rendez au quotidien. Merci Sabine pour la manière très professionnelle dont tu as toujours traité les petits problèmes de comptabilité qui me concernaient. Merci Jean-Didier et Jean-Jacques d'avoir assumé la direction du laboratoire. Je peux dire que je n'ai jamais manqué de rien d'important tant pour les aspects didactiques que pour la recherche.

Les charges didactiques sont l'occasion d'approfondir la matière que l'on enseigne, du moins les premières années. Rétrospectivement, je me rends compte à quel point le cours de systèmes à logique digitale, dont j'ignorais les fondements avant d'avoir à l'encadrer, m'a aidé tout au long de ce travail de recherche. Je peux citer également les divers projets d'électronique que j'ai dû élaborer et encadrer et qui m'ont permis de concrétiser davantage les concepts théoriques sous-jascents. Je remercie les professeurs pour la confiance qu'ils m'ont témoignée en me laissant beaucoup de liberté pour l'encadrement de ces divers cours et projets.

Mes remerciements s'adressent naturellement aux membres de mon comité d'encadrement et aux membres extérieurs du jury pour leurs remarques et conseils, tant sur la forme que sur le fond, chacun avec un éclairage qui lui est propre selon la spécialité qu'il enseigne. Je remercie plus particulièrement mon promoteur, Jean-Didier Legat, pour l'insistance qu'il a toujours manifestée à présenter des publications malgré mon peu d'enthousiasme (c'est peu dire) à une époque où je n'y voyais vraiment aucune utilité. Je n'imaginais pas que cela pourrait un jour me conduire à un poste académique et que les résultats de mes recherches pourraient prendre plus d'envergure et servir à d'autres.

Finalement, je voudrais remercier celle qui m'aime et me supporte tous les jours, mon épouse Marie-Laurence. Merci pour ton soutien et tes encouragements à poursuivre et achever ce travail dans les moments plus difficiles. Merci pour ton amour et ta tendresse, reflets de Celui qui t'habite et que je remercie également de tout mon coeur.

Merci à ceux que j'oublie aujourd'hui mais qui ont contribué, d'une manière ou d'une autre, à l'aboutissement de ce travail.

CHAPITRE 1

Introduction

La synchronisation des opérations par les données est un concept naturel dont on a progressivement perdu le sens avec l'invention du microprocesseur. La plupart des systèmes que l'on peut observer dans notre environnement sont "synchronisés par l'arrivée des données". La feuille sur l'arbre n'attend l'autorisation de personne pour vibrer quand le vent arrive. La goutte d'eau qui tombe n'a pas besoin qu'on vienne la chercher pour rejoindre la rivière. Notre univers est organisé de manière fondamentalement parallèle. C'est ainsi que les premières machines à calculer mécaniques (16ème siècle) étaient elles-aussi synchronisées par l'arrivée des données. Aujourd'hui, la plupart des circuits électroniques analogiques le sont également dans le sens où ils réagissent simultanément à toutes les variations des signaux d'entrées. Seuls les êtres vivants (et les systèmes qui s'en inspirent) dotés d'une certaine intelligence sont capables de séquentialiser des opérations parce que les ressources de traitement disponibles ne sont pas illimitées. Un être humain comprend facilement qu'il peut remplir une citerne en utilisant un seul seau de manière répétée. La plupart des architectures numériques, issues du modèle de von Neumann, agissent de la même manière. L'architecture que nous proposons est un "retour aux sources", une architecture qui n'est pas basée sur le microprocesseur, même si elle peut l'intégrer. Paradoxalement, c'est grâce aux progrès de la technologie ces cinq dernières années que cette approche a été rendue possible. Il nous paraît fondamental de bien conceptualiser ce que nous entendons par "la synchronisation des opérations par les données" avant d'aborder le vif du sujet. Dans un premier temps, nous proposerons donc deux exemples de la vie courante qui en donnent une bonne image intuitive. Le premier est tiré de la circulation routière tandis que le second vient de l'industrie. Ensuite, nous situerons notre approche dans son contexte historique et par rapport aux autres systèmes digitaux. Finalement, nous détaillerons la structure de ce travail et le contenu des divers chapitres.

Depuis quelques années, les rond-points fleurissent un peu partout dans notre pays, remplaçant avantageusement leurs ancêtres les feux rouges.

Qu'y a-t-il de plus inutile en effet que d'attendre qu'un feu passe au vert alors que la voie est libre ? C'est pourtant un système qui a été utilisé durant de longues années, la priorité de droite étant inefficace lorsqu'un carrefour est bloqué par la présence d'une voiture dans chaque voie. De plus, il est parfois intéressant de favoriser le traffic dans certaines directions. Dans le contexte des systèmes digitaux, nous avons une situation analogue. Qu'y a-t-il de plus inutile pour des opérandes que d'attendre le bon vouloir du processeur pour les envoyer vers l'opérateur qui doit les accueillir alors que celui-ci est disponible ? C'est pourtant un système qui a été employé durant de longues années, le circuit combinatoire étant beaucoup trop limité pour des applications un tant soit peu complexes. La priorité de droite est un système synchronisé par l'arrivée des véhicules. C'est son manque d'efficacité dans certains cas qui a conduit à l'introduction d'un deuxième système : le feu rouge. Celui-ci est diamétralement opposé au précédent. Il fait usage d'un contrôleur extérieur (le feu rouge) qui a tout pouvoir sur le passage des véhicules mais ne tient aucun compte de la présence de ceux-ci. Une amélioration du système a été proposée sous la forme de capteurs de présence dont le rôle est de favoriser l'accès aux véhicules en attente. C'est une approche hybride entre les deux précédentes et nous nous en sommes inspirés. Néanmoins, dans le cadre des feux rouges, il faut regretter que ceux-ci n'anticipent pas l'arrivée des véhicules. Ceux-ci doivent donc le plus souvent s'arrêter avant de pouvoir traverser le carrefour. Finalement, notons qu'il est encore possible d'employer un agent de police qui a la capacité d'anticiper l'arrivée des véhicules et d'offrir théoriquement une circulation idéale. C'est malheureusement aussi la solution la plus "coûteuse". Nous verrons dans la suite de notre travail que le transfert des données au sein de notre architecture peut être comparé à ces véhicules qui doivent franchir des carrefours. Alors que le microprocesseur est proche du système utilisant un simple feu rouge, notre architecture est d'avantage proche du rond point, du feu rouge avec un capteur anticipatif ou encore de l'agent de police.

Le domaine de l'industrie a été plus prompt à utiliser la technique de synchronisation par les données. Au commencement, il n'y avait que des artisans capables de séquentialiser les diverses tâches qu'ils avaient à réaliser. Très vite, les hommes ont étés subordonnés à un contrôleur qui avait autorité sur eux et qui indiquait à chacun ce qu'il avait à réaliser au moment voulu. L'industrie a instauré le travail à la chaîne. Dans ce système, les pièces à fabriquer circulent d'elles-même d'un poste de travail à un autre. Il n'est plus nécessaire d'avoir un contrôleur qui indique à chacun ce qu'il doit faire. Dès que la pièce suivante arrive, l'opération peut commencer. C'est la disposition spatiale des postes de travail et les

moyens de transport automatisés qui les relient qui assurent la bonne complétion du travail. Les artisans agissaient un peu à la manière d'un microprocesseur autonome disposant d'une seule unité de traitement. Le système hiérarchique dans lequel un contrôleur subordonne des ouvriers représente assez bien l'évolution des processeurs qui disposent de plusieurs unités de traitement (VLIW, superscalaires) mais on est toujours dans une approche à contrôle dominant. Finalement, le travail à la chaîne est une juste représentation de l'architecture que nous proposons. Ne nécessitant plus de contrôleur central, les données sont acheminées automatiquement vers les ressources qui doivent les traiter. Celles-ci sont capables de détecter leur présence et de commencer automatiquement leur traitement. Une fois le traitement terminé, les résultats sont envoyés de façon tout aussi automatique vers le système de transfert des données.

Ces deux exemples illustrent trois points forts de notre architecture :

- La circulation automatique des données qui tiennent compte de la présence des autres données au cours de leur cheminement vers l'opérateur qui doit les recevoir.
- L'exécution automatique des opérations dès que les opérandes sont disponibles.
- L'utilisation de la topologie du circuit comme moyen de synchroniser les données entre elles.

L'aspect reconfigurable de notre architecture qui rend possible le changement de topologie, de type d'opérations et les moyens de transfert des données est également un point fort de notre architecture mais nous n'avons pas trouvé son pendant dans les exemples de la vie courante.

Le concept de la synchronisation des opérations par les données en lui-même n'est pas nouveau puisque les premières machines synchronisées par les données remontent aux années 1970. Néanmoins, la technologie de l'époque ne permettait pas d'en tirer un véritable profit par rapport au développement des processeurs conventionnels et cette voie a dû être abandonnée. C'est l'arrivée des systèmes reconfigurables de grande taille qui nous a permis de relancer le défi. La suite de cette introduction veut retracer brièvement l'historique des systèmes synchronisés par les données en les replaçant dans leur contexte.

Les premiers circuits digitaux se limitaient à des fonctions combinatoires. Ces circuits sont intrinsèquement synchronisés par les données puisque l'arrivée d'un résultat intermédiaire entraîne automatiquement l'évaluation du résultat intermédiaire suivant. C'est la découverte des

circuits de mémorisation et la généralisation de leur utilisation qui a ouvert la possibilité de séquentialiser les opérations. Un des problèmes majeurs rencontrés par les pionniers des systèmes digitaux est la taille des circuits. Une des premières machines à calculer électroniques (ENIAC, 1943-1946) contenait 18000 lampes et pesait 30 tonnes [**50**]. Il était impensable à l'époque de réaliser un circuit contenant plusieurs multiplieurs travaillant en parallèle. Un seul additionneur-soustracteur bien utilisé pouvait servir à réaliser multiplications, divisions et même racines carrées pour autant que l'on accepte de séquentialiser les opérations. C'est donc dans le but d'économiser au maximum le nombre de ressources de traitement de données que John von Neumann proposa en 1945 la célèbre architecture qui porte son nom [**85**]. Une unité de contrôle lit l'instruction à réaliser dans une mémoire externe, la décode et l'exécute. Cette dernière étape peut impliquer des transferts avec la mémoire externe ou des entrées-sorties. Ensuite, le cycle recommence. Cette architecture permet de réduire la partie de traitement de données à sa plus simple expression mais le prix à payer est qu'il faut désormais un maitre d'oeuvre qui réalise lui même l'acheminement des données vers les opérateurs et le transfert des résultats vers leur destination finale ou intermédiaire. Nous sommes à l'opposé du circuit combinatoire.

Le transistor a été inventé en 1947 par les laboratoires Bell. Le circuit intégré a suivi en 1958. La complexité des circuits intégrés n'a cessé de croître de manière exponentielle depuis cette époque. Partant de quelques dizaines de transistors, nous sommes aujourd'hui à des dizaines de millions de transistors. De nos jours, les limitations qui ont poussé J. von Neumann a proposer son architecture ne sont plus du tout d'actualité. Néanmoins, son architecture étant parfaitement générique et très efficace dans de nombreux cas, elle est vite devenue un standard universellement répandu et bien ancré dans les esprits. C'est ce qui explique le temps qu'il a fallu pour que d'autres approches voient le jour. C'est en 1975 que la première machine synchronisée par les données a été proposée par Dennis [**26**]. Elle est à l'origine de la famille des machines à jetons uniques. Deux autres familles ont suivi : les machines à jetons colorés [**88**] et les machines à stock de jetons explicite [**67**]. Ces architectures seront détaillées au chapitre suivant. Parallèlement, l'architecture von Neumann a elle-aussi évolué. Une augmentation du parallélisme au niveau des instructions a été proposée avec les architectures RISC, VLIW et les processeurs superscalaires. Une augmentation du parallélisme au niveau des données a vu le jour dans les architectures SIMD, vectorielles, systoliques et neuronales. Enfin, les architectures MIMD proposent diverses méthodes de communications entre processeurs et de répartition des tâches pour obtenir un taux de parallélisme plus élevé tant au niveau

des instructions qu'au niveau des données. Une description détaillée de ces architectures est disponible dans [**75**] tandis qu'une rétrospective des machines réalisées pendant trente années est proposée dans [**92**].

Vers la fin des années 1980, les architectures synchronisées par les données ont été abandonnées car bien que fonctionnelles, leurs performances ne leur permettaient pas de concurrencer les évolutions de l'architecture von Neumann [**74**]. L'état de l'art en matière d'architectures parallèles à cette époque est assez bien résumé dans [**27**]. Néanmoins, c'est au cours de ces dernières années que les circuits logiques programmables ont pris leur essor. Constitués au départ de quelques fonctions logiques élémentaires, ils étaient essentiellement utilisés comme "logique de colle" entre d'autres circuits plus complexes qui devaient communiquer entre eux. Mais les progrès de la technologie ont bientôt permis de réaliser des systèmes de large taille équivalent à des dizaines de milliers de portes logiques [**47**]. Un nouveau monde s'est ouvert dans le domaine de la réalisation de systèmes numériques [**42**]. Alors qu'auparavant la distinction était nette entre le logiciel et le matériel, le circuit logique programmable est venu s'introduire comme un hybride entre les deux approches. Il reste avant tout un dispositif matériel puisqu'il est constitué de portes logiques interconnectées entre elles mais il est aussi logiciel puisque les portes logiques et leurs interconnexions peuvent changer de configuration selon la programmation du composant [**79**]. En parallèle avec le développement de ces systèmes reconfigurables dits de "grain fin", des systèmes reconfigurables de grain "moyen" ou "large" se sont développés [**40**]. Ces circuits diffèrent principalement les uns des autres au niveau des fonctions pré-existantes et des possibilités de les interconnecter entre elles. Les circuits logiques programmables les plus récents disposent d'ailleurs tous de fonctions de haut niveau telles que mémoires, additionneurs et parfois même multiplieurs. Selon les systèmes, la frontière entre le logiciel et le matériel varie fortement.

Un problème majeur rencontré au cours de ces dix dernière années est la répartition des tâches à réaliser en matériel et celles à réaliser par logiciel [**59**]. Il y a en effet peu d'experts en logiciel qui sont capables d'expliquer en termes de portes logiques ce qui se passe dans un processeur et encore moins qui sont capables de réaliser eux-même un tel circuit. Inversément, il y a peu d'experts en matériel qui sont rompus aux techniques de programmation des experts en logiciel. La politique adoptée par le passé était de sur-dimensionner le matériel puisque celui-ci ne pouvait plus être modifié par la suite. Ceci était la cause de manque d'optimisation, de perte de temps (le matériel devant être parfaitement connu pour être utilisé par un logiciel) et parfois même d'échecs lorsque les spécifications

changent en cours de temps. L'arrivée des systèmes reconfigurables n'a fait qu'augmenter l'enjeu d'une solution qui unifierait les deux disciplines. La conception simultanée du logiciel et du matériel "Hardware - Software Codesign" est apparue comme une première réponse à ces problèmes [**55**], [**38**], [**91**]. Ces logiciels permettent la simulation et parfois la synthèse d'un système complet (logiciel et matériel) [**56**], [**73**], [**84**]. Leurs possibilités d'analyse de temps de calcul en fonction de l'implémentation permettent de réaliser la répartition des tâches entre le logiciel et le matériel [**19**], [**51**], [**68**]. Une deuxième manière d'aborder le problème a été proposée en tentant de définir un langage universel de description d'algorithme [**90**]. La totalité ou seulement une partie de l'algorithme serait implémentée en matériel et le reste éventuel en logiciel : SystemC [**32**], SpecC [**34**], Handel-C [**66**], Transmogrifier C [**35**], Esterel [**20**], Lustre [**39**], Hardware C [**54**]. Mais encore faut-il disposer d'un logiciel de synthèse automatique acceptant ce genre de langage et suffisamment intelligent pour réaliser une architecture optimale, ce qui est loin d'être évident. Enfin, une troisième approche consiste à utiliser des transformations automatiques de logiciel en matériel [**53**]. Xilinx investit énormément dans ce domaine en ce moment dans le but de proposer aux firmes qui utilisaient des approches de type logiciel (SimuLink de MATLAB) par le passé de migrer automatiquement vers une implémentation matérielle sur leurs circuits. Toutes ces approches ont pour conséquence d'empêcher les experts en logiciel d'avoir un accès direct aux ressources matérielles ; ce qui présente l'immense avantage de leur simplifier la tâche mais qui en revanche limite aussi les performances pouvant être obtenues. En effet, les nouvelles opportunités offertes par les systèmes reconfigurables de grande taille seraient certainement mieux exploitées par les experts en logiciel s'ils avaient la possibilité de spécifier directement l'implémentation de la partie matérielle. Ceci est particulièrement vrai dans le domaine des coprocesseurs reconfigurables encore appelés "accélérateurs matériels" [**81**]. Ceux-ci sont constitués d'un système reconfigurable directement couplé à un processeur. C'est à eux que sont confiées les tâches les plus gourmandes en calculs répétitifs ou celles qui sont mal adaptées aux processeurs conventionnels (les codeurs/décodeurs cryptographiques par exemple) exactement comme un coprocesseur ordinaire mais avec l'avantage de pouvoir le reconfigurer, parfois même en temps réel [**70**]. Il va de soi que plus les experts en logiciel auront un accès efficace au matériel, plus les systèmes seront performants. Notre approche va dans cette direction en proposant une architecture intuitive très proche du matériel sans pour autant nécessiter de connaissance particulière dans ce domaine.

En 1996, quand nous avons commencé notre travail de recherche, la complexité des circuits reconfigurables de grain fin avait atteint les cent mille portes logiques équivalentes dans un circuit développé par Altera : le FLEX10K100 [**14**]. La technologie était prête pour accueillir la version généralisée d'une architecture synchronisée par les données. La plus grande force de ces systèmes par rapport à l'approche utilisant un processeur est le taux de parallélisme qu'ils offrent. Celui-ci ne dépend que du parallélisme intrinsèque de l'algorithme à réaliser et de la taille du système reconfigurable qui va le supporter. A l'heure actuelle, la taille des plus gros circuits reconfigurables approche les dix millions de portes équivalentes (La famille Virtex II développée par Xilinx [**93**]) et elle n'a cessé de croître à une vitesse fulgurante. Ceci laisse penser que le défi à relever pour les prochaines années sera d'arriver à utiliser le parallélisme intrinsèque d'un algorithme [**46**]. En réponse à ce défi, notre thèse consiste en deux points :

(1) L'architecture que nous proposons est capable de concurrencer, surpasser, ou intégrer selon les cas l'approche traditionnelle utilisant un processeur en exploitant au mieux le parallélisme intrinsèque des algorithmes à implémenter.

(2) Elle permet de simplifier la programmation des systèmes reconfigurables de grain fin en utilisant l'aspect intuitif des architectures synchronisées par les données de sorte que les experts du logiciel y trouvent un moyen d'agir directement sur la spécification du matériel tandis que les experts du matériel se trouvent déchargés des aspects fastidieux de synchronisation des opérations, accès mémoires et entrées-sorties présents dans les approches matérielles traditionnelles.

Par rapport à notre premier objectif, rappelons que tel n'était pas le cas des architectures synchronisées par les données d'avant 1990 et que cela avait conduit à leur abandon. Néanmoins, il paraît évident que les systèmes reconfigurables de grande taille offrent un taux de parallélisme supérieur à celui disponible dans un processeur, même si l'on n'utilise pas notre architecture [**24**]. Mais il faut alors l'intervention d'un expert en matériel pour obtenir une architecture optimale après un temps de conception relativement long. C'est là que notre second objectif prend toute son importance. L'architecture que nous proposons permet de faire le pont entre le monde des experts du matériel et celui des experts du logiciel et d'offrir à ces derniers toutes les capacités des systèmes reconfigurables qui leurs étaient inaccessibles auparavant. Cela permet également d'obtenir très rapidement une architecture fonctionnelle et

disposant d'un taux de parallélisme important. De plus, les experts du matériel pourront toujours développer de nouvelles ressources synchronisées par les données optimisées qui soient utilisables par tous, à la manières des "IP" (Intellectual Property) qui se répandent de plus plus à l'heure actuelle.

Notre architecture peut également être utilisée dans le domaine du prototypage rapide en synthétisant l'architecture à tester dans la logique reconfigurable et en utilisant notre architecture pour l'acheminement des données de test et des résultats. Il faut encore noter que le caractère générique et extensible de cette architecture, indépendant du circuit qui la supporte, lui permettra de tirer profit des nouveaux systèmes reconfigurables, au fur et à mesure qu'ils seront développés.

Ce travail de recherche s'est étalé sur un peu plus de cinq années. De nombreux va-et-vient ont été réalisés entre recherche bibliographique, développement et test, au fur et à mesure que notre architecture prenait forme. Au départ, nous avons étudié l'état de l'art en matière de FPGA et assez vite en matière de systèmes multi-FPGA [**45**] dans le but de proposer une architecture nouvelle baptisée "Enhanced Mesh Topology" [**1**][1] que nous avons validée sur divers algorithmes [**2**][1] [**4**][1]. Ensuite s'est posée la question de trouver la meilleure façon d'utiliser les ressources reconfigurables offertes par le fruit de notre recherche. C'est ainsi que nous nous sommes intéressés de près à la synthèse simultanée du logiciel et du matériel. Parmi les nombreuses branches de ce domaine, l'utilisation de la logique reconfigurable comme accélérateur matériel [**21**], [**86**] s'inscrivait directement dans le cadre de notre travail et nous avons poussé la recherche dans cette voie là, laissant de côté une autre voie qui est la synthèse de processeurs intégrant de la logique reconfigurable [**49**]. Nous nous sommes également intéressés à la synthèse automatique partant d'un algorithme écrit en C ou une de ses évolutions. C'est en examinant l'état de la recherche dans les systèmes reconfigurables de grain large que l'intuition de notre architecture a pris naissance [**3**][1], précisément en étudiant les systèmes Xputer développés à l'université de Kaiserslautern [**41**]. Il y a également des similitudes avec les architectures asynchrônes [**78**], [**89**], [**44**] mais la technologie est complètement différente. Nous avons validé notre architecture sur un premier algorithme de traitement du signal en la comparant à une approche dédiée [**8**][2]. Le domaine de la robotique nous a ensuite permis de valider notre architecture sur des exemples de plus grande envergure [**5**][1]. Finalement, une étude bibliographique poussée des systèmes synchronisés par les données a permis

de situer notre approche dans son contexte de manière plus formelle et d'ouvrir l'horizon à d'autres possibilités [**6**][1] [**7**][1].

Notre architecture est donc le fruit d'un large tour d'horizon en matière de systèmes reconfigurables et d'architectures digitales. Dans le présent document, nous avons seulement repris les éléments fondateurs de cette architecture. C'est ainsi que le chapitre deux propose l'état de l'art en matière de systèmes synchronisés par les données et en matière de systèmes reconfigurables. Ces derniers seront étudiés en deux temps : les systèmes reconfigurables de grain large, dont notre architecture s'inspire et les systèmes reconfigurables de grain fin, constitutifs de notre architecture. Le chapitre trois explicite notre architecture en détail, sans pour autant descendre jusqu'au niveau du code synthétisable. Des exemples de celui-ci peuvent être trouvés en annexe. Nous avons validé notre architecture dans divers contextes et types de problèmes. Le chapitre quatre est consacré aux résultats. Le lecteur y trouvera une description des architectures réalisées et testées, les résultats obtenus en termes de taux de parallélisme et de nombre d'opérations réalisées par seconde. Il propose également une extrapolation des résultats aux FPGA disponibles aujourd'hui sur le marché. Le chapitre cinq donne une conclusion en comparant notre approche à celles décrites dans l'état de l'art et ouvre des perspectives pour continuer la recherche dans le domaine des systèmes reconfigurables synchronisés par les données.

[1]Publications en tant qu'auteur principal

[2]Publication en tant que second auteur

CHAPITRE 2

Etat de l'art

1. Les processeurs orientés flux de données

1.1. Introduction

Historiquement, les premiers processeurs développés sont basés sur une architecture à contrôle dominant de type von Neumann. Comme nous l'avons montré dans le premier chapitre de cet ouvrage, le principe de cette architecture est d'utiliser un nombre limité de ressources de calcul et de mémorisation sur lesquelles règne un contrôleur central. Celui-ci doit gérer l'arrivée des données aux entrées des opérateurs et le stockage des résultats en respectant un programme mémorisé au préalable. Le contrôleur doit gérer le transfert des données à différents niveaux : les registres, les éventuelles caches de premier et second niveau, la mémoire externe et les périphériques. Cette architecture est toujours à la base des processeurs modernes.

L'approche des systèmes orientés flux de données (c'est-à-dire synchronisés par les données) est arrivée dans les années 70. Elle est basée sur le concept suivant : ce sont les données qui, au fur et à mesure qu'elles sont calculées, vont rendre "exécutables" un certain ensemble d'instructions : précisément celles qui utilisent les valeurs qui viennent juste d'être calculées. Ces instructions sont alors envoyées avec leurs opérandes vers les unités de traitement disponibles à ce moment. Il n'y a plus de contrôleur centralisé qui gère seul tous les transferts mais bien un ensemble de données dont la présence va permettre l'exécution des instructions qui en dépendent. Une récente synthèse de l'état de l'art en matière de systèmes orientés flux de données est proposée dans [**62**] tandis qu'une description d'époque est donnée dans [**80**].

A priori, le principe des systèmes orientés flux de données permet d'obtenir le taux de parallélisme le plus important pour une implémentation d'algorithme donnée. En effet, si on admet qu'une opération est réalisée dès que les données dont elle dépend sont présentes, il est impossible d'avoir un système plus performant si l'on reste dans l'hypothèse des

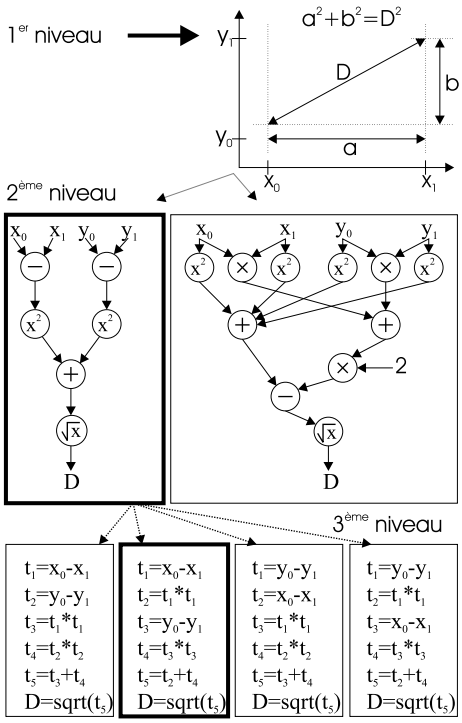


FIG. 1. Le schéma de calcul comparé à l'approche à contrôle dominant

systèmes causals. Néanmoins, ceci suppose un nombre illimité de ressources de calcul et une architecture capable de les gérer, ce qui n'est évidemment jamais le cas pour des architectures réelles.

Les algorithmes ne sont plus écrits sous forme de programme comme c'était le cas pour les systèmes à contrôle dominant mais bien sous forme de schéma de calcul. Il s'agit de graphes de dépendance entre les données d'entrée, les opérateurs et les résultats [**23**]. L'ordre dans lequel les opérations indépendantes seront exécutées de même que l'unité de traitement qui s'en chargera ne sont pas définis à priori. Plusieurs langages permettant la synthèse de ces schémas de calcul ont été développés pour la cause [**11**]. Nous pouvons citer, entre autres : Id [**63**](Irvine Dataflow, utilisé au MIT), Sisal [**31**] (Utilisé à l'Université de Manchester), Val [**12**], FP [**18**], Lucid [**16**] ...

L'utilisation d'un schéma de calcul comme format d'algorithme dans les architectures orientées flux de données permet de ne garder que l'information essentielle, constitutive de l'algorithme à réaliser. Ce concept

est illustré à la figure 1. On y distingue trois niveaux d'abstraction : le principe fondamental sur lequel l'algorithme est construit, le schéma de calcul qui le réalisera et finalement la manière dont ce schéma de calcul sera réalisé. Dans notre exemple, l'algorithme consiste à calculer la distance euclidienne entre deux points. le principe de base repose sur la formule de géométrie élémentaire :

$$\sqrt{(\Delta x)^2 + (\Delta y)^2}.$$

C'est le premier niveau d'abstraction. Il existe plusieurs schémas de calcul qui permettent de réaliser cet algorithme. La figure 1 propose l'implémentation directe :

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

et celle après distribution des produits et simplification :

$$\sqrt{x_1^2 + x_2^2 + y_1^2 + y_2^2 - 2(x_1 x_2 + y_1 y_2)}$$

C'est le deuxième niveau d'abstraction. Finalement, une fois que le schéma de calcul est choisi, il existe encore de multiples façons de réaliser les opérations en faisant varier l'ordre des opérations indépendantes ou le nombre et le type d'unités de traitement disponibles. C'est le troisième niveau d'abstraction. Remarquons que si cet exemple paraît trivial, cela ne l'est pas du tout quand le schéma de calcul contient des milliers d'opérations comme c'est le cas dans le calcul de trajectoires de robots.

En s'arrêtant au deuxième niveau d'abstraction, les architectures orientées flux de données ont davantage de latitude pour utiliser le parallélisme existant dans un algorithme en comparaison aux architectures à contrôle dominant qui, dans les meilleurs cas, essayent de retrouver le parallélisme existant au niveau local comme c'est le cas dans les processeurs superscalaires.

Le concept mathématique sous-jascent aux schémas de calcul est le réseau de Pétri [**69**]. Abondamment utilisé dans des domaines variés, il est encore peu utilisé en électronique malgré le grand intérêt qu'il présente également dans ce domaine [**61**], [**29**]. C'est maintenant chose faite et le présent ouvrage en donne une illustration remarquable dans le domaine des systèmes reconfigurables de grain fin.

Un réseau de Pétri est un graphe constitué de quatre éléments : des "emplacements", des "transitions", un ensemble de "fonctions d'entrée" et un ensemble de "fonctions de sortie". Les emplacements sont destinés à contenir un nombre quelconque de "jetons". Les transitions, associées aux fonctions d'entrée et de sortie, définissent la suppression (pour les

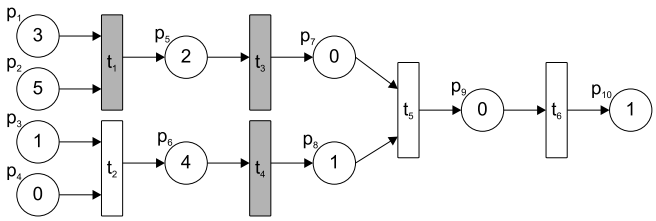


Fig. 2. Illustration du concept de réseau de Pétri

fonctions d'entrée) et la création (pour les fonctions de sortie) de jetons dans les emplacements selon une règle prédéfinie.

La règle de base stipule qu'une transition est prête à "flamber" dès qu'il y a au moins un jeton dans chacun des emplacements associés à cette transition par la fonction d'entrée. Lors du flambage, un jeton est retiré de ces emplacements et un jeton est ajouté à tous les emplacements associés à la transition par la fonction de sortie. Des versions plus sophistiquées de la règle de flambage peuvent prendre en compte un nombre limité de jetons pour chaque emplacement, une couleur pour chaque jeton, une référence au temps pour le flambage etc. La figure 2 permet de visualiser le réseau de Pétri correspondant à notre exemple de calcul de distance. Les transitions sont représentées par des rectangles. Ceux-ci sont représentés en gris lorsque la transition est prête à flamber. Les emplacements sont représentés par des cercles. Les valeurs inscrites à l'intérieur des emplacements représentent le nombre de jetons présents. Dans le cas qui nous concerne, il s'agit des données prêtes à être traitées. Une description plus détaillée des réseaux de Pétri sera donnée au chaptire suivant (point 3.1).

Les opérations fondamentales utilisées en traitement de données par réseau de Pétri sont représentées sur la figure 3.

(1) Joint : Plusieurs données arrivent en entrée. Une entrée de sélection décide de la donnée d'entrée qui sera envoyée vers la sortie. Les autres données attendent d'être sélectionnées à leur tour.
(2) Distribution : Une donnée arrive en entrée. Une entrée de sélection décide de la sortie vers laquelle la donnée d'entrée sera envoyée.
(3) Traitement : Une ou plusieurs donnée(s) arrive(nt) en entrée. Un traitement (logique, arithmétique ou autre) est réalisé sur ces opérandes. Le résultat est envoyé vers la sortie.
(4) Test : Une donnée arrive en entrée. Une entrée de sélection binaire décide si elle sera transmise en sortie ou non.

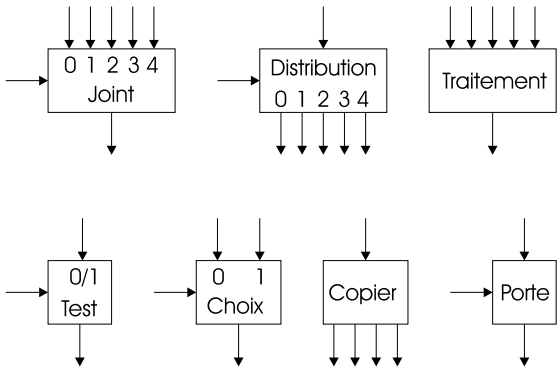


FIG. 3. Les opérations élémentaires orientées flux de données

(5) Choix : Plusieurs données arrivent en entrée. Une entrée de sélection décide de la donnée d'entrée qui sera envoyée vers la sortie. Les autres données sont éliminées.

(6) Copier : Une donnée arrive en entrée. Elle sera recopiée sur chacune des sorties.

(7) Porte : Une donnée arrive en entrée. Elle sera envoyée en sortie dès que la donnée de commande sera présente elle aussi.

Remarque : Dans la littérature, les opérateurs Joint, Distribution et Traitement apparaissent le plus souvent sous leur forme binaire. La forme générale décrite ci-dessus est celle utilisée dans l'architecture que nous proposons.

Dans cette section, nous aborderons d'abord les premiers systèmes à flux de données qui ont été réalisés entre les années 1970 et 1990. Il s'agit de la machine à jetons uniques, de la machine à jetons colorés et de la machine à stock explicite de jetons. L'approche "flux de données" a été ensuite mise de côté par les divers centres de recherche pour aller vers des architectures hybrides (contrôle dominant et flux de données) que nous ne détaillerons pas ici. De plus amples informations peuvent être trouvées dans [**83**]. Nous aborderons néanmoins l'approche TTA (Transfert Triggered Architecture) [**22**], développée dans les années 1990, car elle fait la transition entre les systèmes à flux de données génériques et les architectures reconfigurables développées aujourd'hui. Ces dernières seront présentées à la section 2.

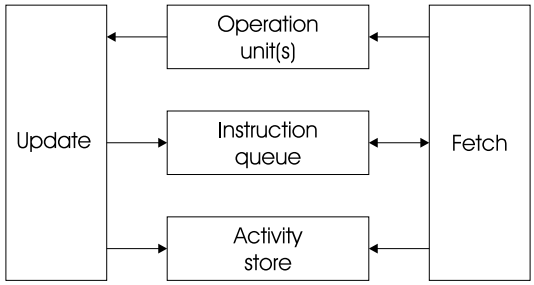


FIG. 4. La machine à jetons uniques développée au MIT

1.2. La machine à jetons uniques (flux de données statique)

Plusieurs architectures ont été développées sur le principe de la machine à jetons uniques : MIT Static Dataflow Architecture (développé au Massachusetts Institute of Technology, Cambridge, Ma, USA), LAU System (développé au CERT, Toulouse, France), TI's Distributed Data Processor (développé par Texas Instrument, USA), DDM1 Utah Data Driven Machine (développé à University of Utah, Salt Lake City, Ut, USA), NEC Image Pipelined Processor (développé par NEC Electronics, Japan) et Hughes Dataflow Multiprocessor (développé par Hughes Aircraft Co., USA). Une brève description de chacune de ces architectures et des références complémentaires peuvent être trouvées dans [**74**]. Dans cette section, nous nous servirons de l'architecture développée au MIT [**26**], [**25**] pour illustrer le concept de la machine à jetons uniques. Comme son nom l'indique, un seul jeton peut se trouver sur un arc donné à un moment donné. Cette architecture est représentée sur la figure 4.

La machine à jetons uniques est constituée de cinq éléments :

(1) Une file d'instructions "Instruction Queue"

Elle contient la liste des adresses des instructions qui sont prêtes à être réalisées, c'est-à-dire celles dont les opérandes sont présentes et qui ont reçu une confirmation que la/les instruction(s) qui en dépende(nt) est/sont prête(s) à recevoir de nouvelles données.

(2) Une unité de lecture "Fetch"

Cette unité se sert de la file d'instructions pour aller lire les instructions prêtes à être réalisées dans la mémoire des instructions et les envoyer à/aux unité(s) de traitement.

(3) Une mémoire "Activity Store"

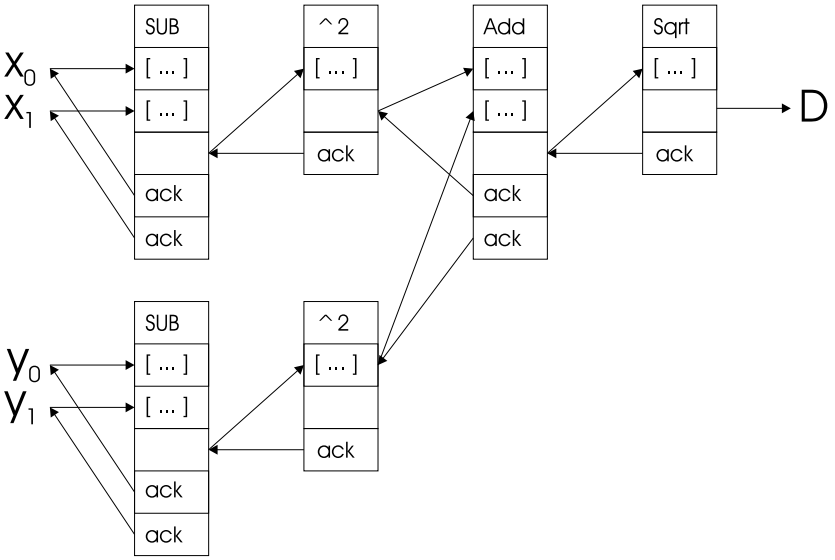


FIG. 5. Implémentation de l'exemple de schéma de calcul sur la machine à jetons uniques.

Les instructions, leurs opérandes, les destinations des résultats et des signaux de contrôle y sont mémorisés.

(4) Une ou plusieurs unité(s) de traitement "Operation Unit(s)"

Les opérations sont alors réalisées par la/les unité(s) de traitement et les résultats sont envoyés à l'unité de rafraîchissement.

(5) Une unité de rafraîchissement "Update"

Cette unité se charge de transmettre les résultats aux instructions concernées et, le cas échéant, d'envoyer l'adresse des instructions à la file d'instructions si elles sont prêtes à être réalisées.

Dans cette architecture, chaque instruction est constituée de plusieurs champs : L'opération à réaliser, deux opérandes, deux pointeurs vers les instructions suivantes et une gestion des confirmations.

A titre d'exemple, l'implémentation de notre schéma de calcul est donné en figure 5.

1.3. La machine à jetons colorés (flux de données dynamique)

Le principal inconvénient de la machine à jetons uniques est, justement, qu'un arc ne peut contenir qu'un seul jeton. Cela veut dire qu'il est impossible d'implémenter des appels de fonctions car les résultats devraient être envoyés à différents endroits selon le point d'appel de la fonction. De plus, il est impossible d'obtenir un parallélisme élevé à l'intérieur d'une boucle puisque la nouvelle itération ne peut commencer que quand la précédente est terminée. Une parade consiste à recopier la partie du schéma de calcul qui correspond au code ré-utilisé mais ce procédé est assez coûteux en taille de programme.

Une approche plus générique a été développée en ajoutant une couleur ("tag" en anglais) aux jetons des réseaux de Pétri. Ainsi, la règle de flambage devient : quand tous les emplacements d'entrée d'une transition contiennent un jeton de la même couleur, le flambage peut avoir lieu. A ce moment, tous les jetons de cette couleur sont retirés des emplacements d'entrée et un jeton de la même couleur contenant le résultat est introduit dans le ou les emplacement(s) de sortie. En pratique, le nombre d'emplacements d'entrée est limité à deux pour n'avoir que des opérations à une ou deux opérandes.

Cette approche a été utilisée simultanément dans divers projets. On peut citer entre autres : Manchester Dataflow Computer (University of Manchester), MIT Tagged-Token Dataflow Machine (développé au Massachusetts Institute of Technology, Cambridge, Ma, USA), SIGMA-1 (développé au Electrotechnical Laboratory, Tsukuba, Japan), PATTSY Processor Array Tagged-Token System (développé à University of Queensland, Brisbane, Australia). Ces architectures sont également brièvement décrites dans [**74**].

Dans cette section, nous utiliserons l'architecture développée à l'université de Manchester [**88**] pour illustrer le concept de la machine à jetons colorés. Cette architecture dérive directement de la machine à jetons uniques. La figure 6 en donne une vue simplifiée.

Notons que dans l'implémentation matérielle de la machine à jetons uniques, les jetons représentaient des instructions avec leurs opérandes tandis que pour la machine à jetons colorés, les jetons représentent des données.

Chaque jeton regroupe quatre informations : la couleur (encore appelée "contexte"), un pointeur vers l'instruction qui en dépend, la valeur

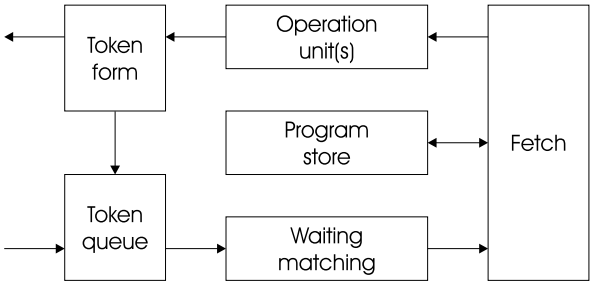


FIG. 6. La machine à jetons colorés développée à l'université de Manchester

elle-même et la place de l'opérande si l'instruction requiert plusieurs opérandes.

Six éléments constituent la machine à jetons colorés :

(1) Une file de jetons "Token Queue" :
Elle reçoit des données à traiter du monde extérieur ou encore des données qui viennent d'être calculées.

(2) Une unité d'assortiment "Waiting Matching" :
C'est la partie la plus critique de cette machine. Elle est chargée de repérer les jetons qui ont la même couleur et qui pointent vers la même instruction qui en dépend. Cette opération devrait en principe être réalisée par une mémoire associative mais en pratique, la taille élevée de la mémoire oblige une implémentation par fonction de hachage. Ceci ralentit considérablement l'opération d'assortiment des jetons. Dès que tous les jetons dont dépend une instruction sont présents, ils sont envoyés à l'unité de lecture.

(3) Une unité de lecture "Fetch" :
Cette unité va lire l'instruction pointée par les jetons qu'elle reçoit et envoie l'ensemble de ces informations à l'unité de traitement.

(4) Une ou plusieurs unité(s) de traitement "Operation Unit(s)" :
Les opérations sont alors réalisées par la/les unité(s) de traitement et les résultats sont envoyés à l'unité de formation des jetons.

(5) Une mémoire de programme "Program Store" :
Elle contient l'ensemble des instructions. Chacune instruction peut être utilisée simultanément par plusieurs jetons si nécessaire grâce au concept de la couleur du jeton.

(6) Une unité de formation des jetons "Token Form" :
Cette unité se charge de reformer un jeton de la même couleur que ses géniteurs et dont le pointeur d'instruction suivante est donné par l'instruction qui vient d'être réalisée.

1.4. La machine à stock de jetons explicite

Cette architecture a été développée pour parer au point critique des machines à jetons colorés : la recherche de concordance entre les jetons de même couleur pointant vers une même instruction. Principalement trois architectures ont été réalisées par des centres de recherche différents : Monsoon [**67**] (développé au Massachusetts Institute of Technology, Cambridge, Ma, USA), EM-4 [**77**](développé au Electrotechnical Laboratory, Tsukuba, Japan) et Epsilon-2 [**36**] (développé à Sandia National Laboratories, Livermore, Ca, USA)

L'innovation de cette approche est de stocker les jetons à une adresse calculée sur la base de leur couleur et de l'instruction qui en dépend. On n'envisage que des opérations à une ou deux opérandes. L'opération à une opérande est triviale. Lorsqu'un jeton d'une opération à deux opérandes arrive, deux possibilités existent : soit c'est le premier jeton et il est stocké dans l'emplacement vide qui correspond à sa couleur et à l'instruction qui en dépend, soit il est le deuxième jeton, auquel cas le premier jeton est récupéré et ils sont tous deux envoyés à une unité de traitement.

A titre d'exemple, la machine de Monsoon est représentée en figure 7. Si l'on compare cette architecture à celle de la machine à jetons colorés, on constate que l'unité d'assortiment a été déplacée en aval de l'unité de lecture des instructions. En effet, le calcul de l'adresse effective requiert une information sur la couleur du jeton (qui est présente sur le jeton lui-même) et une information sur la position de mémorisation du jeton pour l'instruction qui en dépend. Or, cette dernière n'est disponible que quand l'instruction a été lue. Une fois l'adresse effective calculée, il est aisé de stocker le jeton dans la réserve de jetons si c'est le premier qui arrive ou, dans le cas contraire, de récupérer le jeton correspondant pour envoyer la paire d'opérandes à l'unité de traitement.

2. Les architectures reconfigurables à grain large

2.1. Introduction

Le terme "architecture reconfigurable", largement employé dans la littérature anglophone sous le vocable "reconfigurable system", ou "reconfigurable

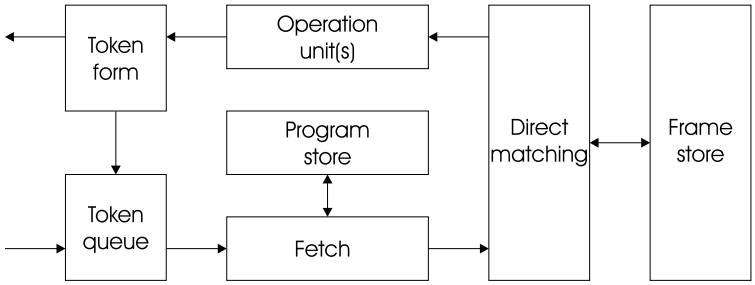


FIG. 7. La machine à stock de jetons explicite Monsoon développée au MIT

computing" recouvre bien des concepts différents. En effet, si jadis il n'existait que des systèmes "non reconfigurables", aujourd'hui, la plupart des architectures disposent d'un certain degré de reconfigurablilité. La possibilité de changer la configuration d'un circuit peut exister à divers niveaux :

(1) Au niveau du programme

C'est le principe même du microprocesseur : un ensemble de fonctions élémentaires sont pré-programmées sous forme de micro-programme. Celui-ci détermine le jeu d'instructions disponibles. Le programme écrit par l'utilisateur est mémorisé en externe et offre donc un certain niveau de reconfigurabilité.

(2) Au niveau du jeu d'instructions

Sur certains microprocesseurs très spécifiques [**17**], il est possible de modifier le micro-programme et donc le jeu d'instructions disponibles à l'utilisateur. Celui-ci peut donc définir divers jeux d'instructions selon les applications ; ce qui offre un degré de reconfigurabilité supplémentaire par rapport au microprocesseur classique.

(3) Au niveau des opérations

On envisage ici des circuits plus élémentaires. Dans ce cas, diverses opérations peuvent être réalisées par un même circuit en fonction de certains signaux de configuration. A tire d'exemple, citons l'ALU (Unité Logique et Arithmétique) et la LUT reconfigurable (chaque sortie correspondant à une configuration d'entrée est mémorisée dans une petite RAM).

(4) Au niveau des connexions

Il existe maintenant des circuits qui ne font que des connexions programmables. Aptix, par exemple, à développé un

circuit qui permet de réaliser toutes les configurations possibles de connexion broche à broche d'un ensemble de 1024 entrées-sorties environ. Selon les circuits, les signaux peuvent être de type analogique ou digital, et dans ce dernier cas, être synchronisés sur une horloge ou non.

(5) Au niveau de la logique

C'est le niveau le plus bas de reconfigurabilité atteint aujourd'hui. La technologie FPGA, par exemple, permet de configurer un ensemble de fonctions logiques et leurs interconnexions. Il est donc possible de configurer un circuit programmable pour qu'il se comporte comme un ASIC. De plus amples informations seront données dans la section 3 qui lui est consacrée.

Dans cette section, nous aborderons le cas des systèmes reconfigurables à grain large. C'est-à-dire, les systèmes qui ne font pas appel à une configuration de la logique elle-même mais plutôt des opérations et des connexions. Nous avons cependant gardé une marge de tolérance pour des systèmes qui, outre leurs possibilités de reconfiguration grain large, incluent aussi une partie reconfigurable de grain fin. Il est en effet délicat de vouloir trop classifier ces systèmes. Développés récemment grâce aux progrès de la technologie VLSI, ils ont été élaborés par les divers centres de recherche sans se soucier de critères spécifiques.

On peut néanmoins comparer ces architectures sous plusieurs angles. Chaque système dispose en effet d'un certain type d'unités de traitement. C'est un premier critère de comparaison. Celles-ci sont reliées entre elles par un certain type de connexions. Ce sera notre deuxième critère de comparaison. Finalement, la possibilité de reconfigurer les unités de traitement et les connexions varie d'une architecture à l'autre. Ce sera notre dernier critère de comparaison.

Nous présenterons d'abord les architectures ne faisant pas appel à un processeur car elles sont plus proches de la forme première de notre architecture. Ensuite, nous aborderons les architectures utilisant un processeur en nous rappelant qu'il est aujourd'hui possible de synthétiser un processeur complet au sein d'un FPGA. Ces architectures peuvent donc nous éclairer sur les avantages qu'il y aurait à inclure cette technologie dans notre architecture. Il faut encore noter que les architectures sont implémentées sous diverses formes (circuit intégré, circuit imprimé, système embarqué, système virtuel). En outre, nous présenterons les aspects originaux de chacune d'entre elles ainsi que les performances annoncées par les concepteurs.

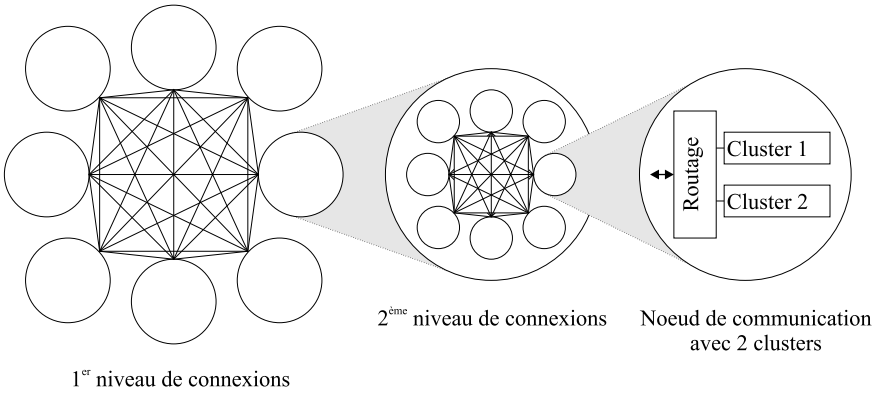


FIG. 8. Vue générale du système ALFA

Nous invitons le lecteur à se rapporter à la fin de cette section (point 2.5) pour une comparaison entre ces différentes architectures. Les points forts et les innovations y sont repris pour en déduire les grandes tendances en matière de systèmes reconfigurables de grain large. Les systèmes reconfigurables à grain fin seront considérés dans la section 3.

2.2. Les architectures sans processeur

2.2.1. *ALFA*

ALFA est un système embarqué (un ensemble de circuits VLSI dédiés montés sur des circuits imprimés interconnectés) qui a été développé au International Computer Science Institute de Berkeley [**82**]. Cette architecture permet l'implémentation directe de larges schémas de calculs (jusqu'à 16384 noeuds) sur un réseau de circuits dédiés. C'est une machine de type flux de données statique, c'est-à-dire autorisant un seul jeton par arc.

Le système est un réseau hiérarchique à deux niveaux qui permet l'interconnexion de 128 clusters (Figure 8). Les transferts de données entre les différents clusters sont effectués par des contrôleurs de transferts dédiés. En local, les clusters sont d'abord regroupés par paires, avec une communication directe entre eux. Ensuite, chaque contrôleur de transfert est connecté à sept autres contrôleurs de même niveau et une huitième connexion donne accès au niveau supérieur. Le dernier niveau est finalement connecté à une ordinateur maître. Chaque cluster contient 128 unités de traitement de 32 bits.

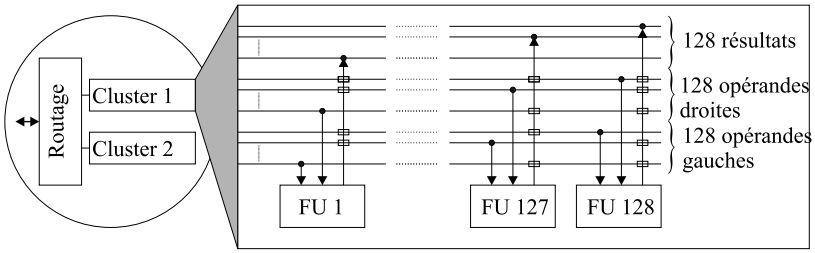


FIG. 9. Vue détaillée d'un cluster sur ALFA

La particularité de l'architecture ALFA consiste à réaliser physiquement une image d'un schéma de calcul au moyen de connexions programmables et d'unités de traitement programmables elles-aussi. Chaque noeud du schéma de calcul dispose donc de sa propre unité de traitement. Une fois cette architecture configurée, seules les données circulent, et ce sous forme de jetons.

Chaque unité de traitement (Figure 9) dispose de deux types de registres : les registres destinés à contenir des données (deux opérandes en entrée et un résultat en sortie) et les registres destinés à la configuration (un registre détermine le type d'opération à réaliser et un deuxième registre détermine les bus sources pour les opérandes et le bus destination pour le résultat).

Les registres contenant les données disposent en plus d'un bit de validité qui indique la présence ou l'absence d'un jeton sur l'arc du schéma de calcul correspondant. Chaque résultat peut être envoyé directement à n'importe quelle opérande d'une autre unité de traitement. Le contrôleur de transferts a la possibilité d'envoyer ou de récupérer une donnée sur n'importe quel bus. Les bits de validité lui permettent de suivre le déroulement des opérations et de récupérer les résultats dès qu'ils sont disponibles.

Aucune réalisation physique n'a été trouvée dans la littérature. L'auteur cite seulement une simulation sur 576 noeuds.

2.2.2. *MATRIX*

Le Massachusetts Institute of Technology (MIT) n'en est pas resté aux machines à flux de données. C'est là également que MATRIX [**60**], un circuit VLSI, a été imaginé et réalisé. Cette architecture est basée sur un réseau hiérarchique à trois niveaux d'unités de traitement reconfigurables. Toutes les unités sont identiques et travaillent sur 8 bits. Chaque

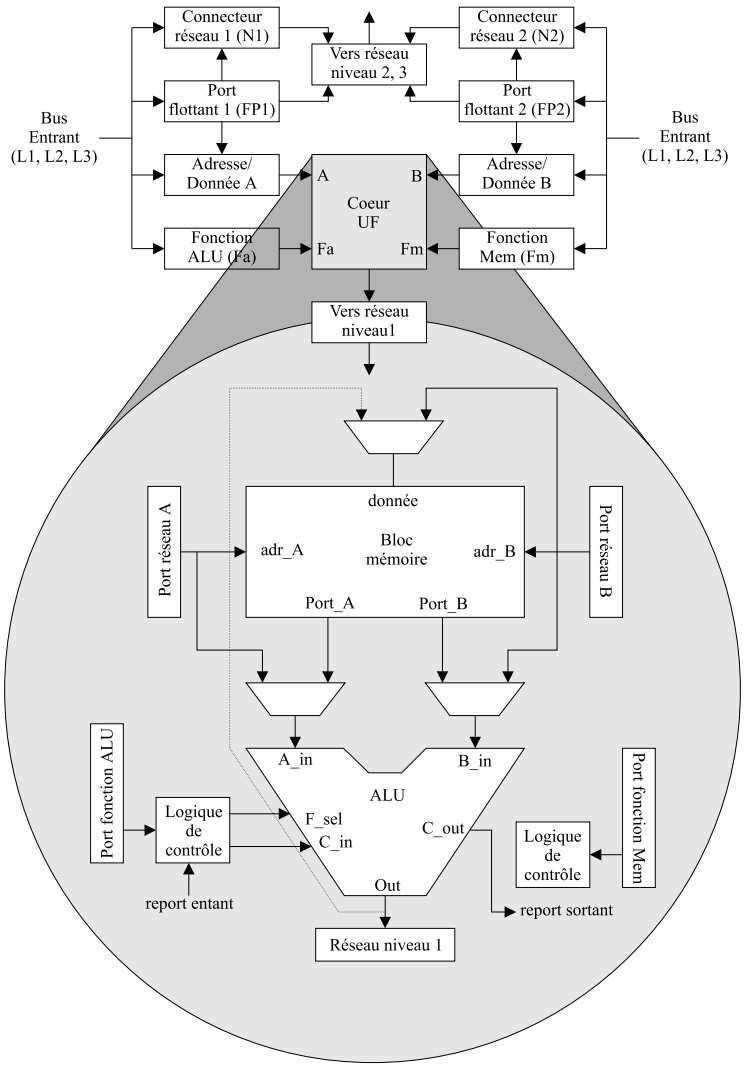


FIG. 10. Architecture des unités de traitement dans MATRIX

unité de traitement (Figure 10) est constituée de trois parties principales :

(1) La mémoire de 256 mots de 8 bits qui sert de ban de registres. Elle peut également être configurée en mémoire à double accès de 128 mots de 8 bits.

(2) L'ALU est assez complète. Elle autorise notamment le calcul des fonctions logiques ET, OU, OU exclusif et leur complément

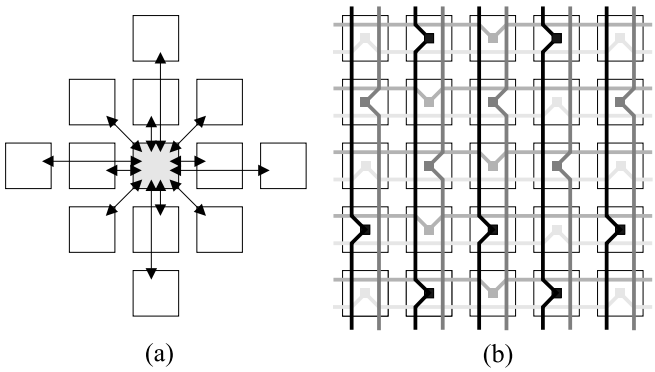


Fig. 11. Le réseau de connexions dans MATRIX

et contient également un multiplieur 8 bits nécessitant 2 cycles pour calculer le résultat sur 16 bits.

(3) La logique de contrôle permet de calculer divers tests (positif, négatif, nul, report de retenue) et d'en déduire la configuration requise pour le cycle suivant. Le calcul du mot de configuration se fait au moyen d'un PLA, en fonction des signaux de contrôle de l'ALU et de ses voisins.

Les unités de traitement sont disposées pour former un réseau physique à deux dimensions. Néanmoins, divers types de connexions existent au sein de ce réseau qui devient finalement un réseau hiérarchique à 3 niveaux : (figure 11)

(1) Les voisins les plus proches : Chaque unité de traitement est connectée à ses douze voisins les plus proches. Les résultats ainsi transmis sont disponibles le cycle suivant.

(2) Des connexions de longueur quatre : Les unités de traitement disposent également de deux connexions vers des unités situées 4 rangées ou 4 colonnes plus loin dans le réseau. Ce type de connexion ajoute au maximum un cycle de pipeline.

(3) Un bus global : Finalement, chaque ligne et chaque colonne du réseau dispose de quatre bus globaux. Ce dernier niveau nécessite toujours un cycle de pipeline pour transférer les données.

Finalement, les ports de communication offrent un très haut degré de reconfigurabilité de l'architecture. Ils disposent de 3 modes de fonctionnement (Figure 12) :

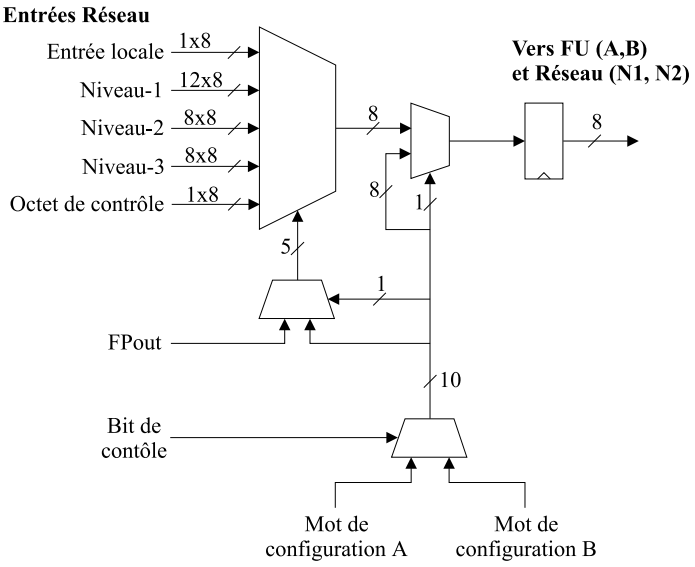


FIG. 12. L'architecture des ports dans MATRIX

(1) Valeur statique : Dans ce mode, le port contient toujours la même valeur qui est utilisée comme constante par l'ALU. Ce mode est notamment utilisé pour implémenter des réseaux systoliques dans lesquels l'unité de traitement effectue toujours la même opération.

(2) Source statique : Ici, le mot de configuration sélectionne l'accès réseau auquel le port est connecté. Cet accès est fixé une fois pour toute au moment de la configuration. C'est le mode classique de configuration d'un réseau d'interconnexions dans un FPGA.

(3) Source dynamique : Ce mode est le plus sophistiqué. Dans ce cas, la configuration est donnée par un port flottant d'une unité de traitement voisine. Elle peut être modifiée à chaque cycle. Ce mode est utilisé lorsqu'une unité de traitement doit traiter alternativement des données présentes sur diverses entrées.

Aucune implémentation physique n'a été trouvée dans la littérature. L'auteur estime qu'un circuit contenant 100 unités de traitement travaillant à du 100Mhz offrirait un taux de calcul maximal de 10Gop/s (en 8 bits).

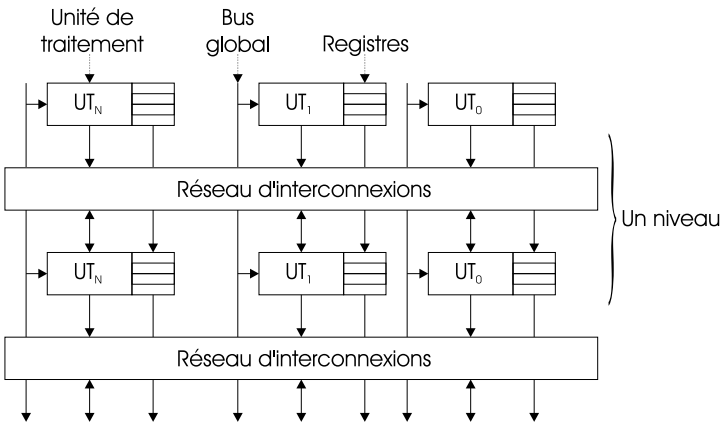


FIG. 13. Architecture PipeRench

2.2.3. *PipeRench*

PipeRench [**72**] est une architecture générique (virtuelle) qui a été développée à Carnegie Mellon University. Elle est à la frontière entre les FPGA et les systèmes reconfigurables à grain large. Son originalité réside principalement dans la possibilité de reconfigurer dynamiquement certaines parties du circuit en cours de pipeline. Le circuit étant générique, sa taille n'est pas fixée à priori. Les largeurs de bus, le nombre d'éléments par niveau de pipeline et le nombre de niveaux sont paramétrables, selon l'application finale désirée. Cette architecture est présentée à la figure 13.

Un niveau de pipeline est constitué d'un certain nombre d'unités de traitement travaillant en parallèle et reliées entre elles par un réseau d'interconnexions programmable. Chaque unité de traitement dispose de registres locaux. Outre le réseau de connexion propre à chaque niveau de pipeline et qui permet un transfert "horizontal" des données, les unités de traitement sont également connectées à des bus globaux permettant le transfert "vertical" de résultats d'un niveau de pipeline à un autre sans devoir passer par toutes les étapes du pipeline.

Les unités de traitement (Figure 14) disposent d'une possibilité de reconfiguration de très bas niveau. En effet, les deux opérandes sont d'abord sélectionnées par un multiplexeur. Elles passent ensuite dans une unité de décalage pour arriver à l'entrée d'une série de LUT à trois entrées. Les trois entrées correspondent à un bit de chaque opérande et un bit commun d'usage générique. La LUT permet donc d'effectuer toutes les opération logiques de deux variables et un bit de contrôle. Pour réaliser

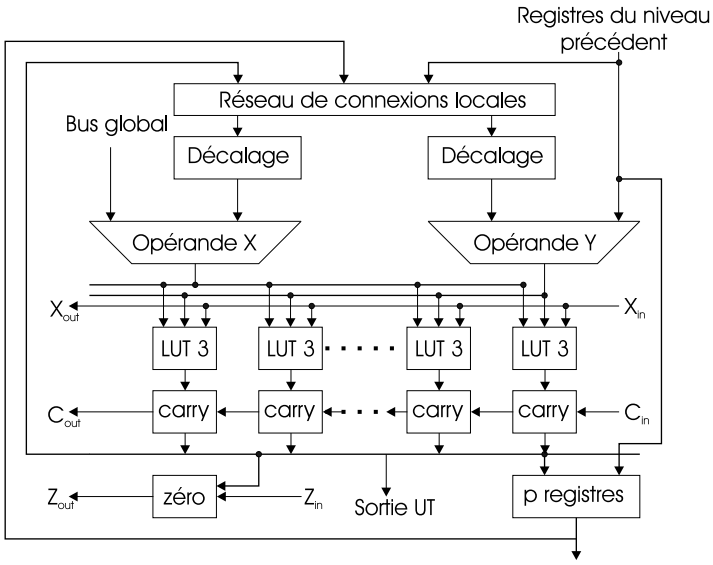


FIG. 14. Les unités de traitement de l'architecture PipeRench

les opérations arithmétiques, une logique dédiée a été ajoutée qui calcule le report de la retenue et du zéro. Finalement, le résultat peut être mémorisé dans un registre local, être réutilisé directement ou être envoyé à l'étage de pipeline suivant.

Un circuit vient d'être réalisé et testé. Une fréquence de 120 Mhz en interne est annoncée par le centre de recherche. Le circuit semble contenir 16 tranches de 16 unités chacune. Le taux de calcul maximal ne peut donc dépasser les 30 Gop/s.

2.2.4. *RaPiD*

L'arhitecture générique RaPiD [**28**] (Reconfigurable Pipelined Datapath) est issue de l'université de Washington. Elle a été réalisée pour parer aux lacunes présentes dans les DSP et dans les ASICs. Les premiers sont trop génériques et donc peu efficaces pour du traitement spécialisé tandis que les seconds n'offrent que très peu de possibilités de modification après synthèse. RaPiD se situe comme intermédiaire entre les deux. Elle est particulièrement bien adaptée pour le produit matriciel.

Cette architecture étant synthétisable, la taille est fonction du problème à traiter. Le circuit lui-même est reconfigurable et permet donc des modifications après synthèse. L'architecture est basée sur l'interconnexion

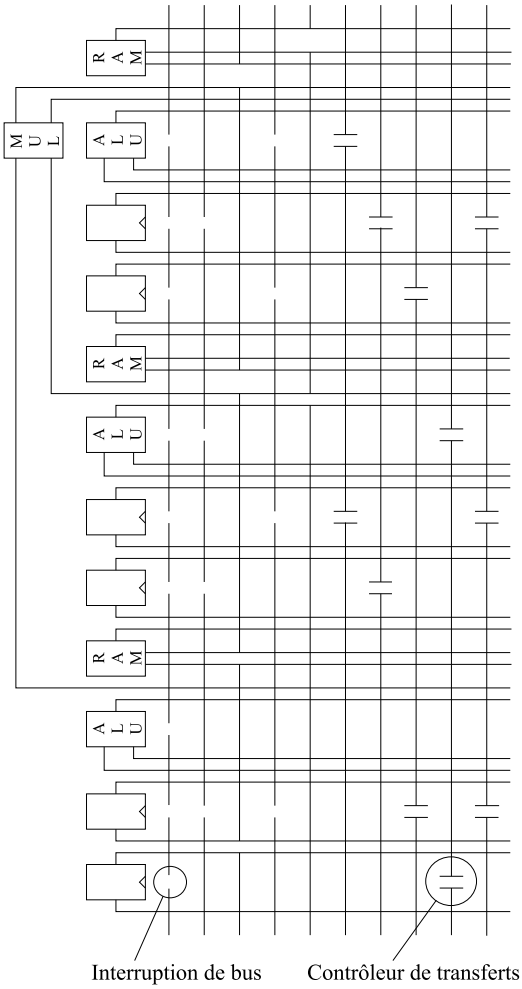


Fig. 15. Une unité de traitement de "RaPiD-1"

linéaire (favorisant le pipeline) d'un certain nombre d'unités de traitements et de mémoires par l'intermédiaire de bus.

Cette architecture générique s'est finalement concrétisée dans un circuit VLSI (RaPiD-1) qui contient 16 unités de traitement telles que celle décrite sur la figure 15. Nous partirons de cette implémentation pour décrire l'architecture RaPiD.

L'unité de traitement de RaPiD-1 dispose de 3 mémoires RAM, 3 ALU, un multiplieur et 6 registres. Les données circulent sur dix bus parallèles. Ces bus sont parfois interrompus, de manière à favoriser un traitement

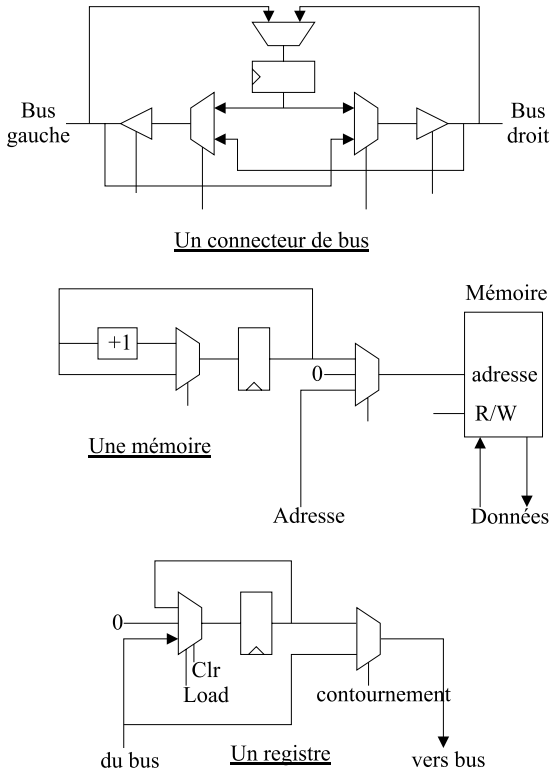


FIG. 16. Détail de RaPiD-1

local de l'information et à disposer de plus de ressources de connexions. A certains endroits, un contrôleur local de bus peut, si nécessaire, assurer le transfert dans les deux sens en insérant un niveau de pipeline. Le détail de ces divers circuit est représenté en figure 16.

La partie de contrôle, qui consiste en la configuration de l'unité de traitement de cycles en cycles, est décomposée en une partie statique et une partie dynamique. La partie statique est stockée dans des mémoires RAM à l'initialisation tandis que la partie dynamique est réalisée par de la logique reconfigurable, assez proche d'une structure FPGA (LUT, registres et connexions programmables).

Cette architecture a été validée sur plusieurs algorithmes dont la convolution (FIR) et la transformée en cosinus (DCT). Un taux de 1.6 GMac/s (en pointe) a été obtenu.

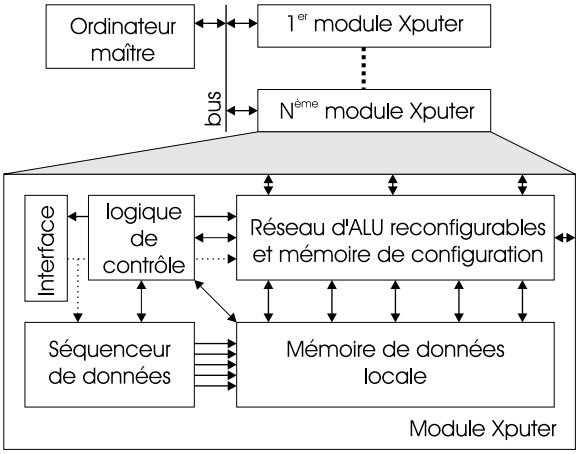


FIG. 17. Architecture générale Xputer

2.2.5. *Xputer*

C'est à l'université de Kaiserslautern que le Xputer [**41**] a vu le jour, succédant à son prédécesseur : le rDPA [**43**] (Reconfigurable Datapath Architecture). L'idée originale consiste à implémenter dans l'espace (c'est-à-dire sous forme de circuits interconnectés) des calculs traditionnellement séquentiels en utilisant des opérateurs synchronisés par les données. Le système complet se compose d'un ordinateur maître connecté à une série de modules identiques reconfigurables par l'intermédiaire d'un bus de données. Chaque module (Figure 17) regroupe 4 entités :

(1) Un séquenceur de données constitué d'un certain nombre de générateurs d'adresses génériques. Ceux-ci sont utilisés pour accéder à des tableaux de données dans un ordre prédéterminé, basé sur une structure de boucles imbriquées. Ils permettent facilement le traitement d'images par blocs.

(2) Une mémoire de données locales dont le bus d'adresse est relié au séquenceur de données tandis que le bus de données est connecté au réseau d'ALU. Cette mémoire est destinée à sauvegarder les résultats intermédiaires.

(3) Un réseau reconfigurable d'ALU pour le traitement des données. Il s'agit d'un réseau 2D dans lequel chaque ALU (Figure 18) est connectée à ses 4 voisins et à un bus local dans les deux sens. La structure de la cellule permet d'exécuter en parallèle une opération avec l'ALU et un routage indépendant de l'ALU. Il

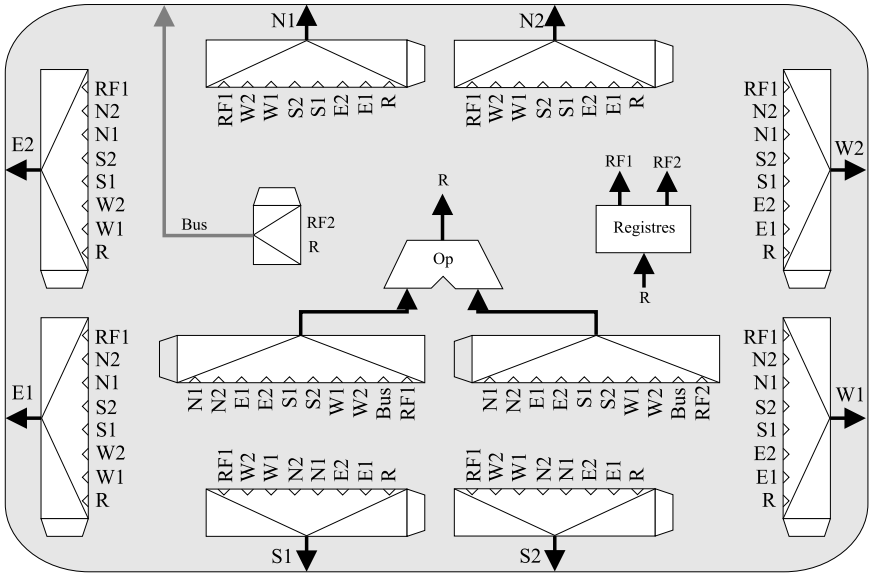


FIG. 18. Architecture d'une ALU reconfigurable dans le Xputer

faut noter que l'ALU dispose d'un micro-programme reconfigurable qui lui permet notamment de réaliser des multiplications en plusieurs cycles. Les opérations sont automatiquement synchronisées par l'arrivée de nouvelles données qui suivent leur chemin au travers des différents noeuds du réseau. Le transfert de données entre ce réseau et la mémoire locale conditionne le séquenceur de données pour la génération des adresses suivantes.

(4) Une interface vers le monde extérieur qui permet à l'ordinateur maître de configurer les modules, leur transmettre les données à traiter et récupérer les résultats.

Aucune performance sur une implémentation physique n'a été trouvée dans la littérature.

2.3. Les architectures avec processeur

2.3.1. *Chameleon*

Chameleon [**52**] est un raccourci pour la société Chameleon Systems. Inc. qui a développé et vend ses circuits VLSI. L'idée novatrice est de regrouper sur un seul circuit un processeur ARC, un contrôleur PCI,

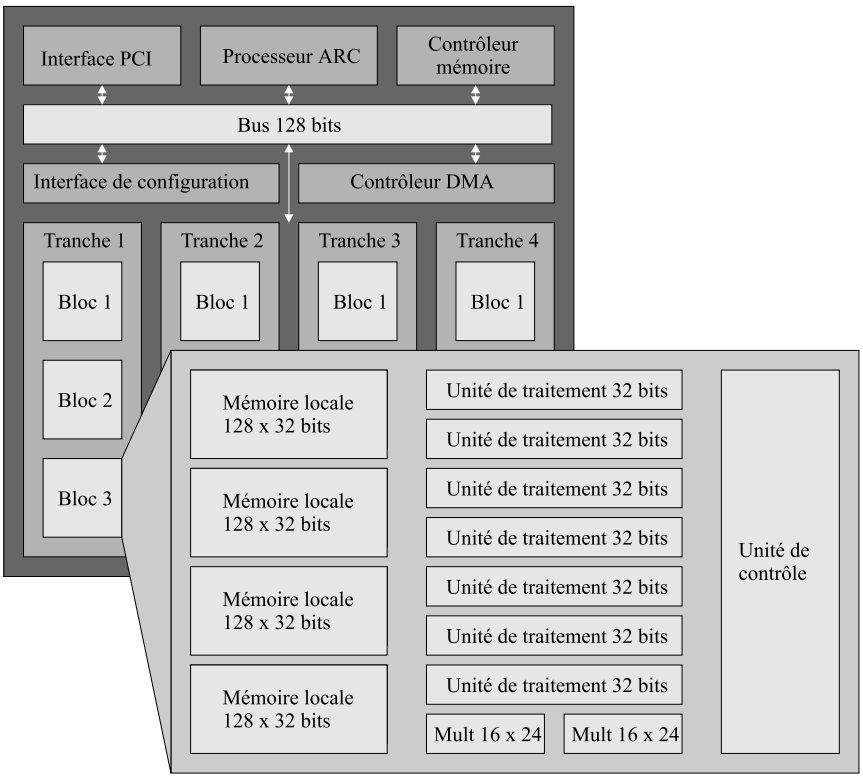


Fig. 19. L'architecture CHAMELEON

un contrôleur de mémoire et un réseau d'unités de traitement reconfigurables de conception originale.

Chaque circuit dispose d'un certain nombre de tranches (Figure 19) configurables individuellement. Une tranche est constituée de trois blocs. Finalement, Chaque bloc regroupe quatre mémoires locales, sept unités de traitement, deux multiplieurs et de la logique de contrôle.

Les unités de traitement (Figure 20) disposent d'un opérateur 32 bits multi-fonctions : ADD, ADD16 (2 additions 16 bits en parallèle), MAX, MAX16, MIN, MIN16, PENC (encodeur de priorité), SADD (addition jusqu'à saturation) et SADD16. Les opérandes sont sélectionnées par des multiplexeurs (24 vers 1) et passent ensuite à travers un opérateur de décalage et des masques.

Pour les transferts de données, il faut compter un délai d'une période d'horolge à l'intérieur d'une tranche et de deux périodes d'horloge si l'on passe d'une tranche à l'autre. Les résultats et les flags sont stockés

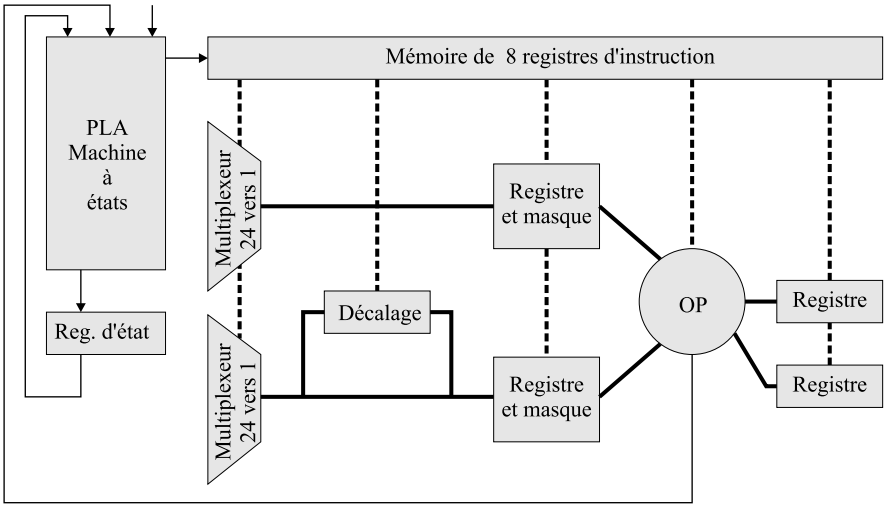


FIG. 20. Architecture des unités de traitement de CHAMELEON

dans les registres de sortie. La configuration de chacun de ces éléments est mémorisée dans un registre d'instruction. Huit instructions peuvent être pré-programmées pour chaque unité de traitement. La sélection de l'instruction courante est réalisée par la logique de contrôle. Celle-ci est simplement une machine à état programmable basée sur un PLA. A chaque cycle d'horloge, un nouvel état et une nouvelle instruction sont calculées par le PLA en fonction de l'état courant, des flags et de signaux de contrôle externes. Cette architecture est un bon exemple ou coexistent les deux niveaux de reconfigurabilité (grain fin et grain large).

Concernant les performances, plusieurs tailles de circuits sont disponibles. Actuellement, le plus performant (le CS2112) dispose de 84 unités de traitement, 24 multiplieurs et 48 mémoires locales. Il obtient un taux de transfert maximal sur ses entrées-sorties de 2 Gbytes/s et un taux de calcul maximal de 24 Gop/s sur 16 bits. Les circuits sont néanmoins prévus pour travailler en 32 bits.

2.3.2. *FPPA et E-FPPA*

L'architecture "Field Programmable Processor Array" [**64**] (Figure 21) a été élaborée au CSEM (Centre Suisse de Microélectronique et Microtechnique). Par analogie au FPGA (Field Programmable Gate Array), l'idée est de construire un circuit intégrant un grand nombre de processeurs interconnectés en réseau à deux dimensions. Cette architecture a été reprise par C. Amerijckx (Université Catholique de Louvain) pour

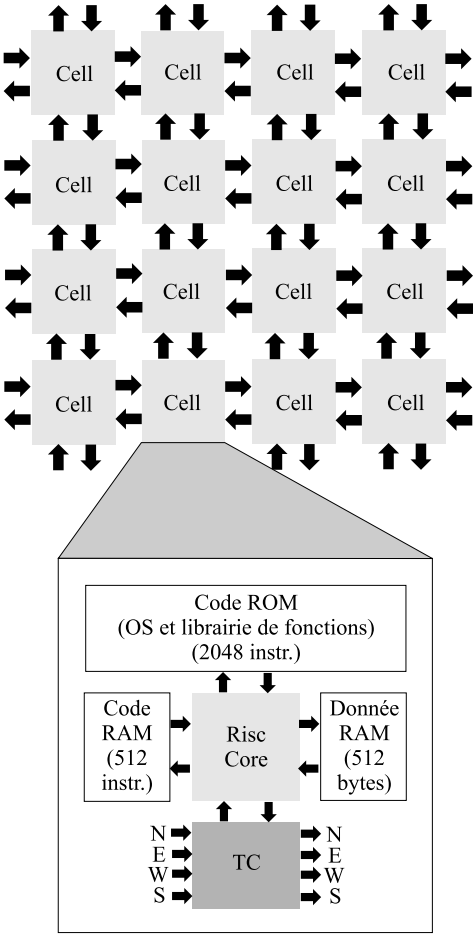


Fig. 21. L'architecture FPPA développée au CSEM

aboutir au E-FPPA [**15**]. Ces architecture sont orientées faible consommation.

Le processeur utilisé est de type RISC (le CoolRisc). L'unité de traitement est constituée d'un processeur connecté à trois mémoires et à un contrôleur de transferts. La mémoire ROM contient les routines d'initialisation du processeur ainsi qu'une librairie de fonctions mathématiques. La mémoire RAM contient d'une part le programme à exécuter et d'autre part les données.

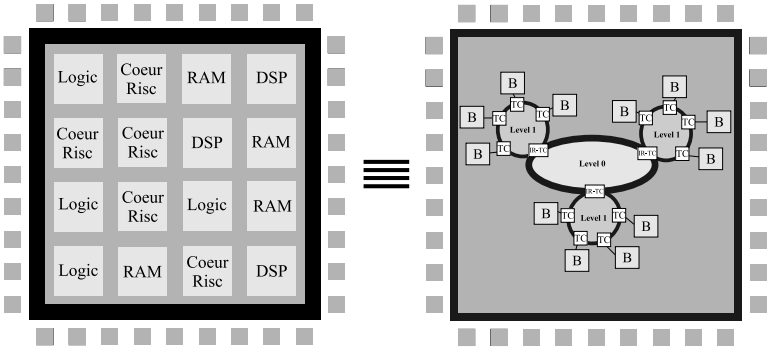


Fig. 22. L'architecture E-FPPA élaborée à l'UCL

Le contrôleur de transferts utilise un protocole asynchrone de type demande-confirmation. Il gère seul les transferts, déchargeant ainsi le processeur pour lui permettre de se consacrer uniquement au traitement des données.

Le nombre d'unités de traitement étant relativement important, la probabilité qu'une erreur de fabrication touche une unité n'est pas nulle. Pour parer à ce problème, le circuit est capable de s'auto-configurer pour supprimer la ligne et la colonne correspondant à l'unité défectueuse. Une re-numérotation des unités est réalisée et l'utilisateur se retrouve avec un circuit de plus petite taille mais fonctionnel.

L'architecture E-FPPA a repris le concept du réseau de processeurs et l'a élargi en diversifiant les unités de traitement disponibles et en proposant une architecture originale pour le transfert de données entre les différentes unités. Les unités de traitement peuvent prendre la forme de processeurs, DSP, mémoires, logique reconfigurable ... Le transfert de données entre les unités est basé sur une communication en anneaux hiérarchiques avec des contrôleurs de transferts dédiés à chaque unité et entre les anneaux (Figure 22).

La structure en anneau permet le transfert parallèle de plusieurs données sur un anneau, ce qui est très avantageux si la distance entre deux unités en communication est assez faible. L'utilisation de FIFO (Figure 23) dans chaque contrôleur de transferts permet une grande souplesse quant à la synchronisation pour l'émission et la réception de données.

L'auteur du FPPA annonce une estimation des performances de 1000 mips pour un circuit qui contiendrait 100 unités travaillant à 10Mhz

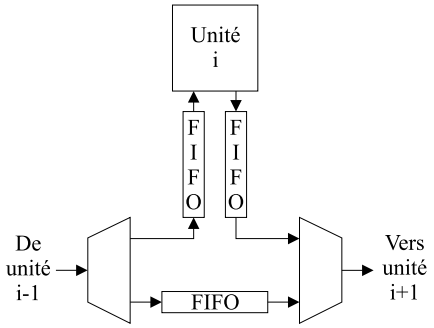


FIG. 23. Un contrôleur de transferts dans l'architecture E-FPPA

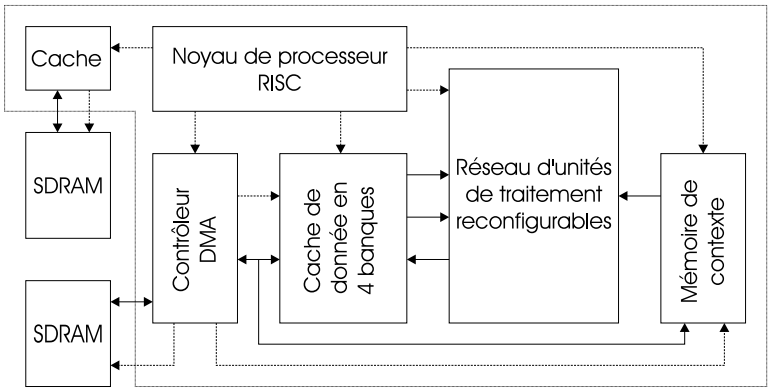


FIG. 24. L'architecture Morphosys

chacune. L'architecture E-FPPA ne connaît pas d'implémentation physique. Ses performances seraient de toutes façons étroitement liées au type des circuits dont il serait constitué.

2.3.3. *MorphoSys*

Développée à l'Université de Californie, Irvine, USA, cette architecture est à la pointe de la recherche dans le domaine des systèmes reconfigurables à grain large. Un processeur RISC, un réseau d'unités de traitement reconfigurables et une interface mémoire très élaborées sont regroupés sur un seul circuit VLSI [**76**]. Le système est capable de se reconfigurer de façon dynamique. Cette architecture est représentée sur la figure 24.

Le mode de fonctionnement de ce circuit est le suivant : les opérations générales sont réalisées par le processeur RISC tandis que les opérations impliquant de nombreux calculs répétitifs sont prises en charge par le réseau d'unités de traitement. L'exécution d'un algorithme se fait en plusieurs étapes :

(1) Le processeur RISC configure le réseau d'unités de traitement à partir de la mémoire externe en programmant le contrôleur DMA.
(2) Les données à traiter provenant de la mémoire externe sont transférées par le processeur dans la première partie du tampon de données au moyen du contrôleur DMA.
(3) Les données sont traitées par le réseau d'unités de traitement. Parallèlement, de nouvelles données sont introduite dans la deuxième partie du tampon de données.
(4) Les nouvelles données sont traitées par le réseau d'unités de traitement sur le temps que les premiers résultats sont transférés en mémoire centrale.
(5) Les étapes 3 et 4 sont répétées jusqu'à ce que toutes les données soient traitées.

Le réseau d'unités de traitement est organisé en un carré de huit sur huit. Le modèle de traitement de données est de type SIMD : chaque ligne ou chaque colonne peut être reconfigure individuellement et partage le même mot de configuration. Les unités de traitement sont connectées à leurs voisines directes ainsi qu'à d'autres plus éloignées, comme représenté sur la figure 25.

Chaque unité de traitement (Figure 26) est constituée de deux multiplexeurs d'entrée, une ALU 16 bits avec multiplieur, un opérateur de décalage et quatre registres. Les unités sont configurées par un registre de contexte global de 32 bits.

Les auteurs des cette architecture espèrent obtenir une fréquence de travail de 100Mhz, ce qui entraînerait des performances de pointe de 6.4 Gop/s

2.3.4. *Raw Machines*

Le Massachusetts Institute of Technology (MIT) est décidément très fécond dans le domaine des architectures pour le traitement de données. Raw [**87**] (Reconfigurable Architecture Workstation) y a été également développé. Cette architecture consiste en un réseau à deux dimensions

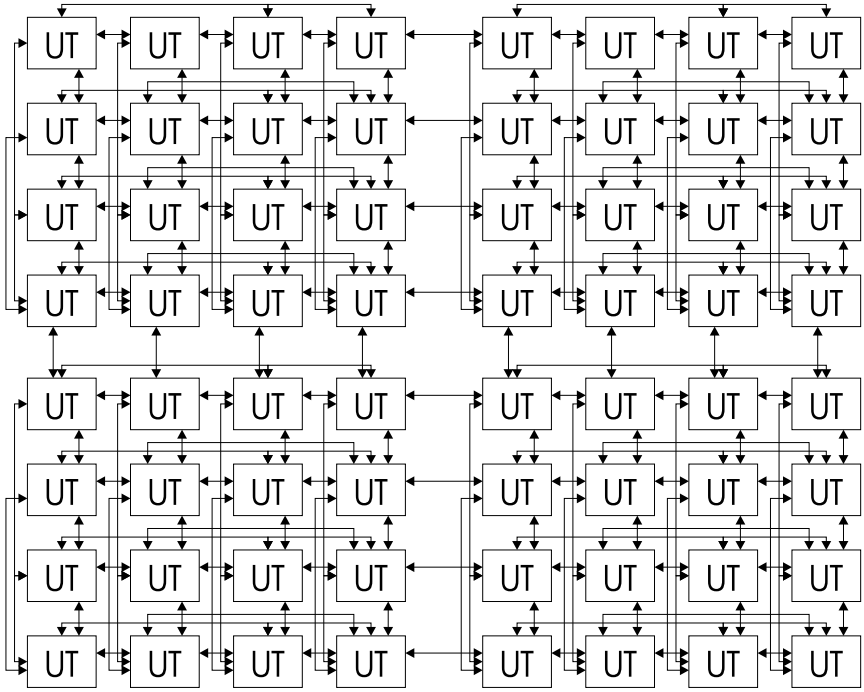


Fig. 25. Le réseau d'unités de traitement de Morphosys

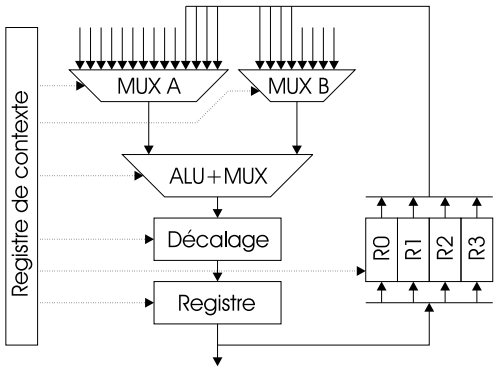


Fig. 26. Les unités de traitement reconfigurables de MorphoSys

d'unités de traitement basées sur un processeur RISC. Chaque unité de traitement (Figure 27) dispose d'une mémoire d'instructions, une mémoire de données, un peu de logique reconfigurable et une connexion programmable au réseau.

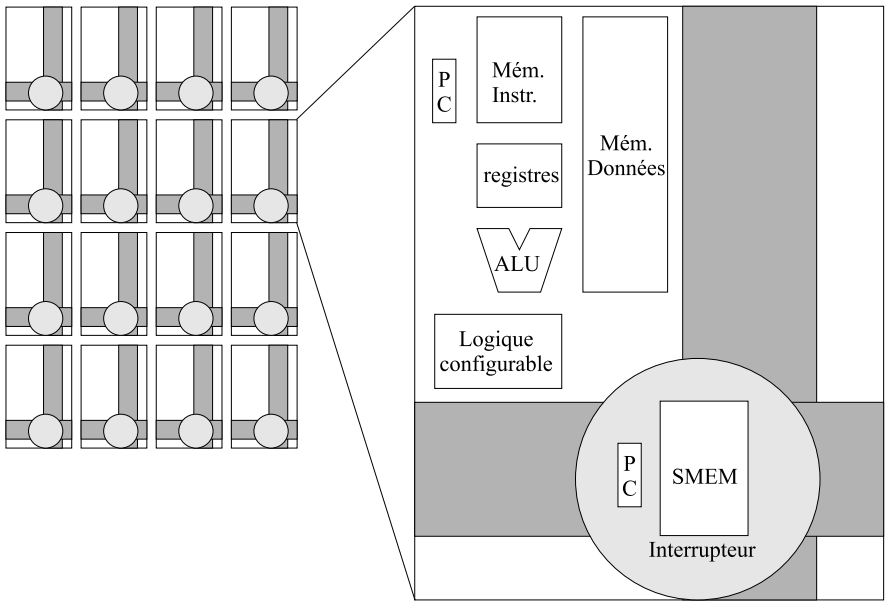


Fig. 27. L'architecture RAW développée au MIT

Les concepteurs de cette architecture ont voulu donner au compilateur un accès aussi complet que possible au hardware. C'est ainsi que le processeur et la connexion programmable au réseau exécutent tous deux une séquence d'instructions générées par le compilateur.

Le système est particulièrement efficace pour les traitements de données qui ne demandent pas de synchronisation entre les diverses unités de traitement. Dans ce cas, les données sont envoyées d'unités de traitement en unités de traitement dans un pipeline régulier et pré-programmé.

Cette architecture a été validée par émulation sur un système comprenant 5 cartes de 64 FPGA chacune. Une implémentation d'un ASIC constitué d'un réseau de 4X4 processeurs travaillant à un fréquence de 225Mhz est en cours.

2.4. Une architecture particulière

Les "Transport Triggered Architectures" [**22**] sont d'un type complètement différent des précédents. Ce type d'architecture n'est ni une machine à flux de données, ni un système reconfigurable semblable aux précédents. Elle se rapproche du processeur classique et de ses évolutions (superscalaire, VLIW) tout en étant différente. Cette architecture est unique en son genre mais elle présente des similarités avec celle que nous avons

développée. C'est pourquoi nous prenons le temps de la présenter quelque peu.

Développées dans le début des années 1990 à la Technische Universiteit Delft, les TTA proposent un nouveau concept : décrire un algorithme en utilisant exclusivement des opérations de transfert des données entre diverses unités. Le nombre d'unités, leur fonctionnalité et le nombre de transferts réalisables en parallèle sont des paramètres du système qui sera finalement implémenté sous la forme d'un ASIC.

Nous partirons d'un processeur développé par Henk Corporaal, inventeur de cette architecture, pour illustrer ce concept. Il s'agit du circuit MOVE32INT représenté à la figure 28.

Quatre bus partagés par l'ensemble des unités sont utilisés pour le transfert de données d'une unité à l'autre. Les unités implémentées pour ce processeur sont : dix bans de registres, deux unités d'addition/soustraction, une unité d'opérations logiques, une unité de décalage, une unité de chargement de constantes, deux unités de comparaison, une unité de stockage externe et une unité de chargement des instructions. Cette dernière est très spécifique puisque c'est elle qui détermine les transferts à effectuer en fonction du programme stocké à l'extérieur.

Chaque unité dispose d'un certain nombre de registres accessibles en écriture et/ou en lecture. De plus, certains registres ont la propriété de déclencher l'opération dès que l'on y écrit une valeur. Une opération peut prendre plusieurs cycles pour être réalisée. Un certain pipeline est donc possible. L'unité d'addition/soustraction, par exemple, dispose d'un registre d'entrée à déclenchement contenant la première opérande, d'un registre d'entrée classique contenant la deuxième opérande et d'un registre de sortie contenant le résultat. Dès qu'un transfert a lieu vers l'entrée de déclenchement, le calcul commence et le résultat est ajourné deux cycles plus tard.

Chaque instruction est codée sur 64 bits et comprend 4 champs identiques représentant chacun un transfert indépendant des autres. L'opérande source est soit un registre, soit une constante. Certaines unités disposent de plusieurs registres virtuels, correspondant à un seul registre réel. L'accès à un de ces registres détermine l'opération qui doit être réalisée. Le format des instructions est donné à la figure 29.

Chaque instruction est réalisable sous la condition définie par le champ de garde. Les 8 combinaisons offrent diverses fonctions logiques des résultats des deux unité de comparaison. Il faut noter qu'un branchement inconditionnel se réduit à charger une nouvelle valeur dans le compteur

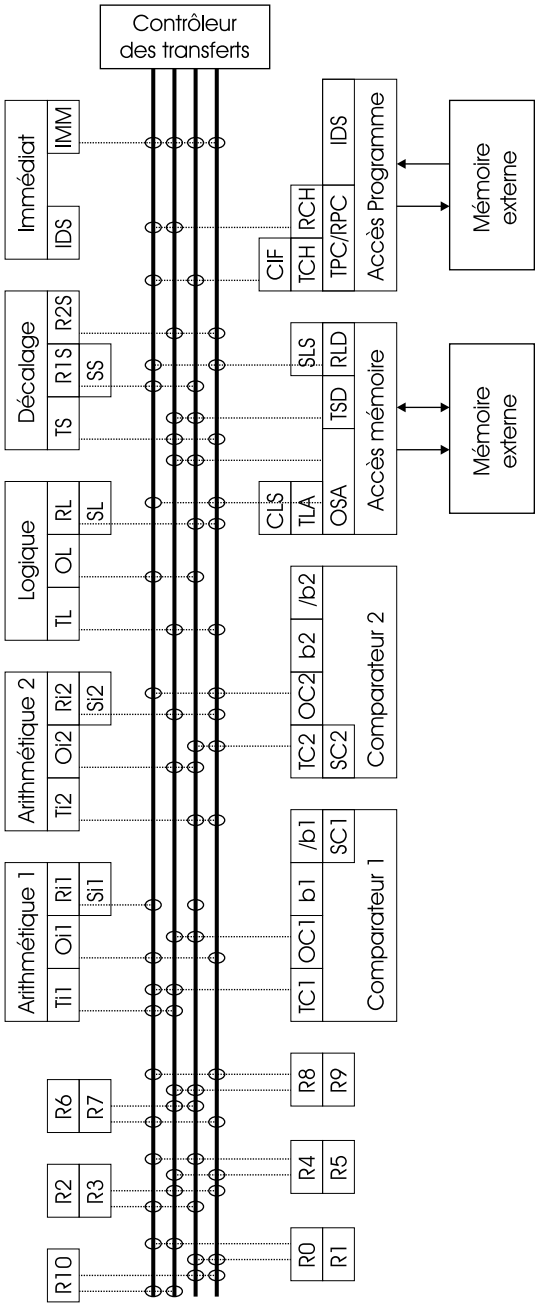


FIG. 28. Exemple d'architecture TTA : le processeur MOVE32INT

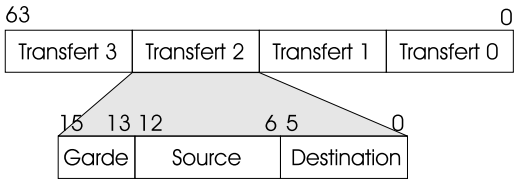


FIG. 29. Format des instructions du processeur MOVE32INT

de programme tandis qu'un branchement conditionnel utilise le même stratagème mais de manière conditionnelle.

Une simulation a permis de déterminer un chemin critique de délai égal à 12.5ns qui autorise une fréquence de travail maximale de 80Mhz, soit 320Mop/s.

2.5. Conclusion

Dans cette dernière section, nous proposons une étude comparative entre les diverses architectures présentées selon les trois critères annoncés :

(1) Le type d'unité de traitement (Tableau 1)
(2) Le type de connexions/communications (Tableau 2)
(3) Les possibilités de reconfiguration (Tableau 3)

Partant des tableaux récapitulatifs pour chaque critère, nous dégagerons les grandes tendances, les originalités intéressantes et les éventuelles faiblesses à éviter. Finalement, nous montrerons les points importants que nous avons pris en compte dans l'élaboration de notre architecture.

Lorsque l'on compare les diverses unités de traitement, on constate qu'il existe deux grandes familles : les systèmes basés sur une ALU et les systèmes basés sur un noyau de processeur (RISC ou DSP). Les premiers disposent le plus souvent de quelques registres et parfois d'une petite mémoire. Les seconds ont besoin d'au moins une mémoire et en disposent parfois de trois (une ROM et deux RAMs). Beaucoup d'ALU disposent de l'opération de décalage, certaines ont un multiplieur, d'autres disposent de logique reconfigurable de grain fin. La majorité des architectures sont capables de traiter 16 bits et certaines vont jusqu'à 32 bits. Enfin, le nombre d'unités de traitement est très variable d'une architecture à l'autre. C'est à ce niveau que ce joue le compromis. Plus l'architecture est simple, plus elle peut être répliquée un grand nombre de fois sur un seul circuit.

Parmi les idées originales intéressantes, nous retenons :

Architecture	Unité de traitement
ALFA	128 ALU 32 bits avec trois registres par ALU. Implémentation directe d'un schéma de calcul statique.
MATRIX	ALU complexe avec multiplieur, mémoire double accès et PLA pour le contrôle
PipeRench	Décalage, LUT à 3 entrées pour effectuer toutes les opérations de deux variables et 1 bit de contrôle, logique dédiée pour les additions-soustractions.
RaPiD-1	3 ALU, 3 RAM, 1 multiplieur et 6 registres. Logique dédiée pour accès linéaires en mémoire et pipeline.
Xputer	Une ALU 32 bits avec micro-programme reconfigurable intégré et des séquenceurs de données.
Chameleon	7 ALU 32 bits avec décalage, 2 multiplieurs 16x24 bits, 4 mémoires 128*32 bits. Possibilité de traiter deux mots 16 bits en un cycle.
(E-)FPPA	Processeur RISC, mémoire, (DSP et logique reconfigurable dans E-FFPA)
MorphoSys	1 ALU avec multiplieur, décalage et 4 registres.
Raw	Un processeur RISC, une mémoire d'instructions et une mémoire de données, un peu de logique reconfigurable.
MOVE32INT	2 opérations arithmétiques, 1 opération logique, un décalage, 2 comparateurs, 2 unités d'accès à de la mémoire externe et 11 registres.

TAB. 1. Comparaison des unités de traitement

- La possibilité d'exécuter 2 opérations 16 bits en parallèle à la place d'une opération 32 bits et les fonctions originales MAX,MIN,SADD et PENC de l'ALU Chameleon.
- La présence simultanée d'unités de traitement très diversifiées dans E-FPPA.
- Les générateurs d'adresses autonomes et la synchronisation par les données de Xputer.
- L'utilisation de logique reconfigurable de grain fin pour le contrôle ou pour le traitement de donnée (Chameleon, E-FPPA, MATRIX, PipeRench, RaPiD et Raw).
- L'utilisation de petites mémoires pour favoriser le traitement local des données (Chameleon, E-FPPA, Xputer, MATRIX, RaPiD et RAW).
- La synchronisation des opérations par l'arrivée de données (une esquisse dans ALFA, Xputer, MOVE32INT).

Architecture	Connexions et communications
ALFA	Réseau hiérarchique à 4 niveaux (8x8x2x128=16384 noeuds). Matrice de connexions au sein du CLUSTER (niveau 4) de 3x128 bus 32 bits.
MATRIX	Réseau hiérarchique 2D à trois niveaux : Connexion directe aux 12 voisins les plus proches, connexion de longueur 4 et bus.
PipeRench	Réseau 2D avec connexions génériques horizontales et bus en vertical, orienté pipeline.
RaPiD-1	Système multi-bus avec contrôleur de transferts entre 2 bus adjacents
Xputer	Architecture bus unique et réseau 2D avec accès aux 4 voisins directs. Traitement et routage en parallèle.
Chameleon	Réseau hiérarchique à deux niveaux (3x4=12 noeuds). Contrôleur DMA, PCI, mémoire externe.
(E-)FPPA	Réseau 2D avec contrôleur de tranferts et réseau hiérarchique en anneaux.
MorphoSys	Réseau 2D avec connexions étendues et contrôleur DMA
Raw	Réseau 2D avec contrôleur de transferts à chaque noeud qui dispose de son propre programme.
MOVE32INT	Système multi-bus.

TAB. 2. Comparaison des types de connexions

– Le caractère générique et synthétisable de certaines architecture qui permet d'offrir un circuit final adapté à l'application désirée tout en offrant de grandes possibilités de reconfiguration après synthèse (ALFA, MATRIX, PipeRench, RaPiD, TTA).

L'observation des différentes architectures de connexions entre les unités de traitement nous montre 3 grandes classes qui parfois se recoupent : Les structures régulières à une ou deux dimensions, les réseaux hiérarchiques et les connexions par bus. Le nombre de connexions entre une unité de traitement et ses voisines varie entre 4 et 24, sans compter les architectures génériques. Les délais occasionnés dépendent du type de connexions. L'utilisation du pipeline pour augmenter le parallélisme est très répandue et les architectures doivent le rendre aisé. Un nouveau concept apparaît depuis quelques années : l'utilisation de contrôleur DMA ou assimilé au sein même des circuits.

Certains concepts sont utilisés dans plusieurs architectures et/ou offrent un intérêt manifeste :

Architecture	Reconfigurabilité
ALFA	Configuration statique des opérations et connexions de bas niveau et programmation des contrôleurs de transferts.
MATRIX	3 types de configuration (valeur statique, source statique et source dynamique).
PipeRench	Reconfiguration dynamique en cours de pipeline ,reconfiguration de grain fin (LUT).
RaPiD-1	Configuration statique et dynamique par PLA.
Xputer	Configuration statique des séquenceurs de données, connexions dans le réseau 2D et micro-programmes des ALU.
Chameleon	Configuration dynamique et partielle. Accès à huit mots de configuration par unité de traitement.
(E-)FPPA	Programmation des processeurs, contrôleurs de transferts et de la logique reconfigurable.
MorphoSys	Configuration dynamique par ligne ou par colonne avec cache de configuration.
Raw	Programmation du processeur, du contrôleur et de la logique.
MOVE32INT	Programme mémorisé en externe.

TAB. 3. Comparaison des possibilités de configuration

- L'utilisation de plusieurs niveau de hiérarchie pour favoriser le calcul local en offrant des connexions rapides (ALFA, Chameleon, E-FPPA, MATRIX et MorphoSys).
- La présence d'un contrôleur DMA ou assimilé (Chameleon, Xputer, MorphoSys, esquisse dans Raw et E-FPPA).
- La structure originale en anneaux (E-FPPA).
- La possibilité d'utiliser le pipeline comme moyen d'augmenter le parallélisme dans presque toutes les architectures.
- Le caractère générique des architectures qui permettent de fixer la taille et le nombre des bus après simulation (RaPiD et TTA).

Finalement, l'observation des différentes possibilités de configuration des architectures met en évidence les divers types de configurations possibles déjà esquissés au point 2.1. Le moyen privilégié des architectures basées sur un processeur est de reconfigurer son programme alors que celles basées sur des ALU doivent reconfigurer le type d'opération qu'elles réalisent et les connexions. Entre les deux, il y a des approches ou l'on reconfigure le micro-programme et d'autres ou l'on définit un certain

nombre d'opérations réalisables par une ALU. Certaines architectures sont configurées uniquement au démarrage tandis que d'autres peuvent se reconfigurer en temps réel.

Ici encore, nous pouvons mettre en évidence certains aspects innovateurs qui offrent des possibilités intéressantes :

- La possibilité de configurer des connexions pour implémenter un schéma de calcul directement en matériel (ALFA).
- La possibilité de mémoriser huit configurations différentes de l'ALU Chameleon, c'est-a-dire huit instructions complexes.
- L'utilisation de logique reconfigurable de grain fin pour gérer les signaux de contrôle et la sélection de la configuration suivante (Chameleon, MATRIX, PipeRench, RaPiD et Raw).
- La possibilité de micro-programmer les ALU offerte par Xputer. Néanmoins, la réalisation des multiplications par micro-programme est longue et n'est sans doute pas la meilleure application de ce concept.
- L'utilisation simultanée de configuration statique pour certaines parties et dynamique pour d'autres (MATRIX, RaPiD).
- L'utilisation de logique reconfigurable de grain fin adjointe à de la logique mathématique proposée par PipeRench.

Au vu de ces diverses caractéristiques, nous constatons que chaque architecture sous-tend des choix pour optimiser certains aspects prépondérants tout en laissant d'autres points moins optimaux. Ainsi, il faut choisir entre un grand nombre d'unités de traitement ou des unités de traitement complexes pour une question de taille du circuit final. Le même compromis existe au niveaux des connexions rendues possibles entre les diverses unités de traitement. Dans ce cas, outre la taille du circuit, il faut également prendre en compte le temps de transition des données d'un point à l'autre du circuit. Enfin, plus un circuit offre de possibilités de se reconfigurer, plus il prend de la place et plus il est lent.

Nous avons pu constater que s'il n'existe pas une architecture "miracle", certaines idées ou concepts rendent une architecture plus performante qu'une autre, toutes autres choses restant égales. C'est finalement ce "savoir-faire", ces "idées novatrices" des inventeurs que nous nous proposons de résumer maintenant pour y faire référence dans la description de l'architecture que nous proposons.

(1) Utiliser le pipeline et/ou une synchronisation des opérations par les données pour éviter des temps morts si les données ne sont pas disponibles pour être traitées.

(2) Utiliser des unités spécialisées pour le calcul des adresses ou un contrôleur DMA couplé à une mémoire double accès. Ainsi les

adresses sont calculées en parallèle avec la lecture ou l'écriture de données en mémoire.

(3) Favoriser le calcul local qui est intrinsèquement plus rapide en utilisant des mémoires locales ou des registres et en disposant de connexions locales rapides telles que celles qui sont présentes dans les réseaux hiérarchiques.

(4) Utiliser des multiplieurs intégrés car la programmation d'une multiplication consomme trop de temps par rapport à une implémentation par circuit.

(5) Combiner les avantages des divers types d'architectures en intégrant sur un même circuit des processeurs, des ALU, et de la logique reconfigurable.

(6) Permettre une reconfiguration partielle dynamique comme alternative à un problème de trop grande taille de circuits. Utiliser les différents niveaux de reconfigurabilité.

(7) Utiliser simultanément l'aspect générique (nombre d'unités de traitement, de connexions, largeur des bus ...) et l'aspect reconfigurable d'une architecture. Bien que difficiles à mettre en oeuvre, ils permettent d'obtenir une architecture qui soit réellement adaptée à l'application visée tout en permettant de modifier ses caractéristiques par la suite.

3. Les architectures reconfigurables à grain fin

3.1. Introduction

La technologie FPGA s'inscrit au sommet de l'évolution des composants logiques. Historiquement, la première forme de circuit logique utilisée est la porte logique élémentaire "NAND" (le complément du "ET" logique) et la porte logique élémentaire "NOR" (le complément du "OU" logique). Tout circuit logique peut en effet être réalisé comme une combinaison d'une seule de ces deux portes. La technologie évoluant, des fonctions logiques dédiées sont apparues sous forme de composants discrets :

- Les fonctions logiques de base : OU, ET, OU exclusif et leurs compléments, éventuellement regroupées en bus.
- Les circuits de multiplexage, codage, démultiplexage et décodage.
- Les fonctions logiques évoluées : décalage, rotation.
- Les fonctions arithmétiques : addition, soustraction, comparaison.
- Les fonctions de mémorisation : registres (à décalage), ROMs, RAMs, FIFOs.

– Les circuits dédiés plus complexes (Compteurs, codeurs, ALU, microprocesseurs...)

Les fonctions logiques de base sont très utilisées comme interface entre des circuits plus complexes. Elles sont couramment appelées "glue logic", c'est-à-dire "logique de colle" entre les composants. Néanmoins, l'utilisation de nombreux composants discrets s'est révélée coûteuse en terme de place nécessaire sur une carte. De plus, il était quasiment impossible de modifier un tant soit peu la fonction logique réalisée si les spécifications changeaient ou si une erreur avait échappé à la simulation. De là est venue l'idée de réaliser des circuits logiques configurables qui réduiraient l'encombrement et permettraient des modifications même quand la carte est réalisée.

Le circuit programmable le plus générique est la mémoire. En effet, une mémoire disposant de N_a lignes d'adresse et N_d lignes de données permet de réaliser N_d quelconques fonctions de N_a variables. Néanmoins, la complexité du circuit étant exponentielle et le délai linéaire par rapport au nombre d'entrées, cette solution n'est pas rationnelle pour de la "logique de colle". Pourtant, diverses technologies ont vu le jour pour stocker les programmes nécessaires aux processeurs et sont encore utilisées aujourd'hui pour contenir les routines élémentaires des systèmes d'exploitation.

– Les ROM (Read Only Memory) : Mémoire figée d'usine.
– Les PROM ("Programmable ROM") : Mémoire que l'on peut programmer une seule fois. Néanmoins, il est toujours relativement facile de changer le circuit si l'on doit changer le programme. Il suffit de monter le circuit sur un soquet.
– Les EPROM ("Erasable PROM") : Mémoire de type PROM mais que l'on peut effacer en la plaçant un certain temps (environ une demi-heure) sous des rayons ultra-violet intenses. Le circuit peut alors être reprogrammé. Cette technologie requiert elle-aussi l'utilisation de soquets si l'on désire reprogrammer le composant.
– Les EEPROM (Electrically Erasable PROM) : Ces mémoires sont du type EPROM mais leur effacement se fait de manière électrique. Certains modèles permettent de reconfigurer le circuit sans le retirer de la carte sur laquelle il est monté ("In System Programming").
– Les mémoires FLASH : Ces mémoires sont une évolution des EEPROM. Elles offrent une interface de programmation et d'effacement plus évoluée.

Les PROM et EPROM requièrent toujours une tension nettement différente de la tension d'alimentation pour être programmées. Selon les modèles, certaines EEPROM et FLASH n'en ont pas besoin.

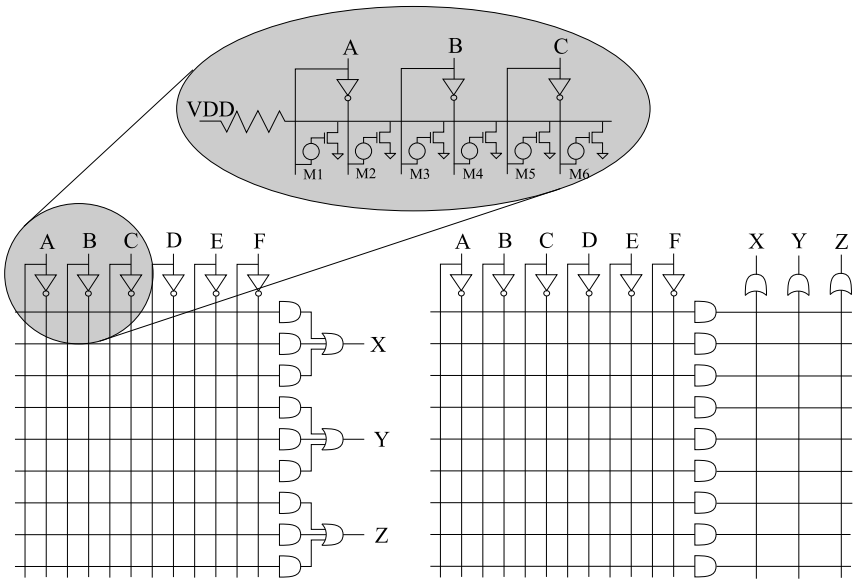


Fig. 30. Les circuits PAL (gauche) et PLA (droite)

Pour parer aux inconvénients sus-mentionnés des mémoires, des composants moins génériques mais plus petits et plus rapides ont été développés : les PLD (Programmable Logic Devices) tels que les PAL et les PLA, les CPLD (Complex PLD) et les FPGA (Field Programmable Gate Array).

- Les PAL : Ces circuits disposent d'une matrice prenant les entrées et leurs compléments en verticale et un certain nombre de produits logiques en horizontale (Figure 30). Chaque variable et son complément peut être connectée de manière programmable à un produit. Les sorties sont finalement la somme des produits qui lui leurs sont attribués. Sur certains modèles, un registre peut être intercalé entre la somme de produits et la sortie qui lui correspond. Dans ce cas, les sorties et leurs compléments sont ajoutés aux variables d'entrée au niveau du matriçage vertical.
- Les PLA : C'est une évolution des PAL. Plutôt que d'assigner un certain nombre de produits à chaque sortie, on a prévu un double matriçage. Le premier réalise tous les produits dont on a besoin tandis que le second réalise toutes les sommes des produits dont on a besoin. Davantage générique, ce circuit est néanmoins un peu plus lent car il dispose de deux niveaux de programmation.
- Les CPLD : La technologie évoluant et la demande de performances augmentant, il est devenu nécessaire de passer à des systèmes de

plus grande taille. Néanmoins, il n'était pas possible d'en rester à des systèmes à deux niveaux pour un grand nombre de variables car le nombre de produits nécessaire (dans le cas d'un OU exclusif par exemple) varie comme l'exponentielle du nombre de variables. Le CPLD est venu en réponse à ce problème. Il s'agit simplement de plusieurs PLD regroupés sur un seul circuit et interconnectés par une matrice d'interconnexion programmable. Dès lors, la taille d'un circuit n'est plus limitée que par la technologie.

– Les FPGA : Ces circuits sont une évolution des CPLD. Ils disposent de plusieurs niveaux hiérarchiques de connexions. Depuis peu, ils intègrent également des mémoires entières, des multiplieurs et même des noyaux de processeur. Les sections suivantes leur sont consacrées.

Actuellement deux fabricants mondiaux se disputent le marché des FPGA : Altera et Xilinx. De nombreux autres fabricants, de moindre envergure, proposent également leurs propres produits. Dans les sections qui suivent, nous passerons rapidement en revue les diverses technologies proposées par Altera et Xilinx pour y faire référence dans l'architecture que nous proposons. Nous aborderons la question des connexions (au sein d'un circuit ou entre plusieurs circuits) au point 3.4 qui est consacré aux systèmes multi-FPGA.

3.2. Les FPGA fabriqués par Altera

Outre les quelques petits circuits développés quand il en était encore à ses balbutiements, Altera propose deux types d'architectures bien différentes. Ces architectures sont basées sur :

(1) Des sommes de produits (Figure 31) : C'est l'architecture classique des PAL et de PLA qui a été un peu améliorée et à laquelle on a adjoint un réseau hiérarchique de connexions. Il s'agit des famille MAX (Multiple Array Matrix). Elles présentent les avantages et inconvénients de leurs ancêtres (rapides mais pas exhaustives car le nombre de produits disponibles pour une somme est limité). Remarquons la présence de la porte OU exclusif qui permet d'augmenter le nombre de fonctions logiques réalisables par rapport à un simple PLA. Un registre peut être intercalé entre la fonction logique réalisée et la sortie qui lui correspond. Des fonctions de "preset" (mise à '1') et "clear" (mise à '0') asynchrones ainsi qu'une commande "enable" sont alors disponibles. Pour des raisons de synchronisation entre tous les registres, ceux-ci partagent généralement une ou plusieurs horloge(s) globale(s).

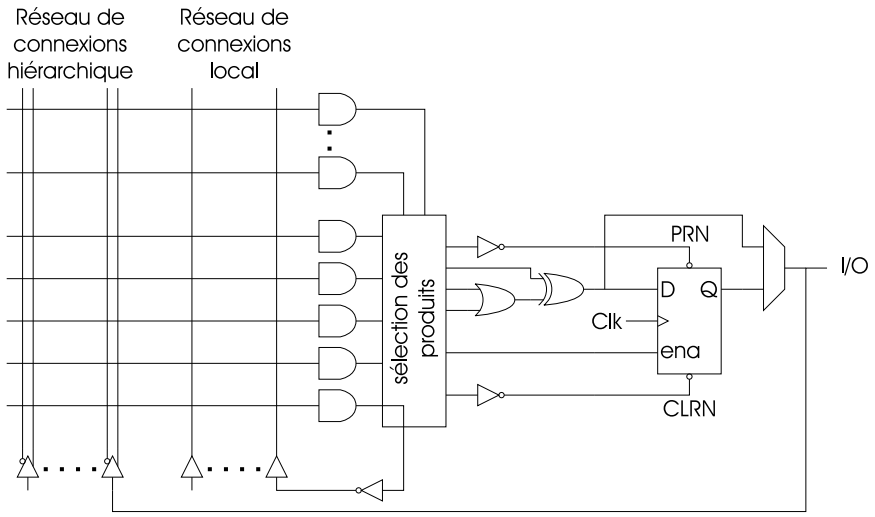


FIG. 31. Schéma simplifié de l'architecture MAX (Altera)

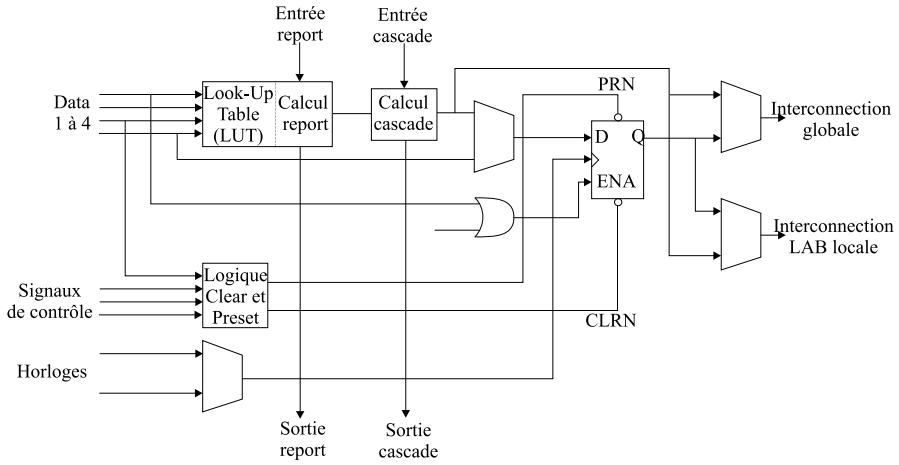


FIG. 32. Schéma simplifié de l'architecture FLEX (Altera)

(2) Des LUT ("Look Up Table") (Figure 32) : Ici, le principe est tout autre. Chaque cellule logique dispose de quatre entrées. Il existe donc $2^4 = 16$ combinaisons différentes de ces entrées. L'idée consiste à mémoriser la sortie correspondant à chaque combinaison d'entrée dans une petite table de 16 bits (la LUT). C'est la famille FLEX (Flexible Logic Element Matrix). Un réseau de connexions hiérarchique assure la connexion des élément

logiques entre eux. Remarquons que la simplicité de cette architecture lui permet néanmoins de réaliser $2^{2^4} = 65536$ fonctions différentes. De plus, de la logique dédiée a été ajoutée pour réaliser les opérations arithmétiques (report de la retenue dans les additions-soustractions) et logiques (report du zéro). Tout comme dans la famille MAX, un registre peut être intercalé entre la fonction logique et la sortie qui lui correspond.

Dans un premier temps, la taille des circuits a augmenté progressivement pour atteindre 512 cellules (le circuit MAX7512) dans le cas de la famille MAX et 1960 cellules (le circuit EPF6024) pour la famille FLEX. Ensuite, un nouveau concept est apparu : ajouter au coeur de la logique des petites mémoires RAM imbriquées. Altera a proposé une nouvelle famille de composants (la famille FLEX 10K) disposant d'un certain nombre de petits blocs de mémoire de 2048 bits configurables en mots de 1,2,4 ou 8 bits. Le concept s'est encore amélioré en offrant la possibilité d'avoir des mémoires à double accès et finalement, des mémoires accessibles par le contenu (CAM). Le plus gros circuit de ce type (le circuit EPF10K200E) contient 24 mémoires imbriquées, 9984 cellules logiques et peut offrir jusqu'à 470 entrées/sorties.

Dernièrement, l'idée est venue de combiner sur un seul circuit les avantages des technologies MAX et FLEX. Cela a donné naissance à la famille APEX (Figure 33). Les APEX disposent d'un certain nombre de "MegaLAB" interconnectés en réseau hiérarchique. Chaque MegaLAB contient 240 LUT (du même type que la série FLEX) associées à un ESB (Embedded System Bloc). Ce bloc a la particularité de pouvoir être configuré en mémoire (ROM, RAM, DPRAM, CAM) ou en PLA semblable à ceux utilisés dans la famille MAX. La taille des circuits a littéralement explosé. Le plus gros APEX (le circuit EP20K1500) dispose de 51840 éléments logiques (de type FLEX) et de 216 ESB. Ce qui serait équivalent à un million et demi de portes logiques (selon Altera). La famille APEX II offre en outre une compatibilité avec divers systèmes de transmission de données (True LVDS, Flexible LVDS, PHY-Link Layer Interface, diverses mémoires et bus).

Finalement, la dernière innovation consiste à ajouter au coeur de la logique un noyau de processeur avec quelques périphériques et de la mémoire : c'est la famille Excalibur. Altera propose des circuits avec 3 processeurs différents : l'ARM, le MIPS et le Nios. Ce dernier est très différent des deux autres car il s'agit d'un processeur synthétisable dans la logique FPGA plutôt que d'un noyau synthétisé.

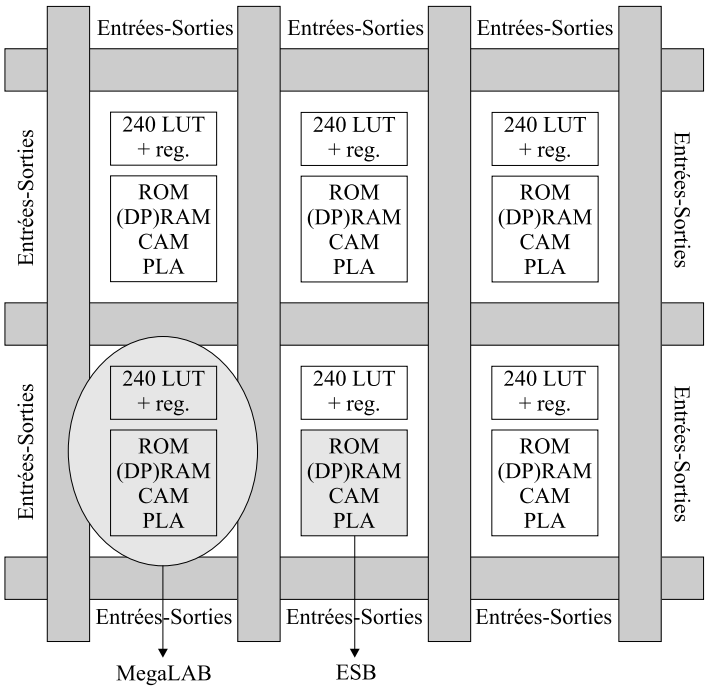


FIG. 33. Schéma simplifié de l'architecture APEX (Altera)

3.3. Les FPGA fabriqués par Xilinx

La technologie développée par Xilinx se prête moins facilement à une description sommaire car elle est intrinsèquement plus complexe. Au départ, Xilinx utilise exclusivement des LUT et des multiplexeurs pour réaliser ses fonctions logiques (les familles XC3000 et XC4000). Néanmoins, pour concurrencer Altera dans le domaine des PLA (famille MAX7000), Xilinx a élaboré deux familles de PLD utilisant la technologie PLA :

- La famille XC9500 : Une copie presque conforme de l'architecture MAX7000.
- La famille XPLA -2-3 : Cette architecture est constituée d'un réseau de type PAL, un réseau de type PLA et un ensemble d'interconnexions programmables. Elle offre donc simultanément les avantages des PAL, des PLA et consomme très peu de puissance.

Ces familles restent cependant assez limitées en nombre de portes logiques et Xilinx ne dispose pas d'équivalent direct à la famille APEX d'Altera qui intègre simultanément la technologie PLA et LUT sur un

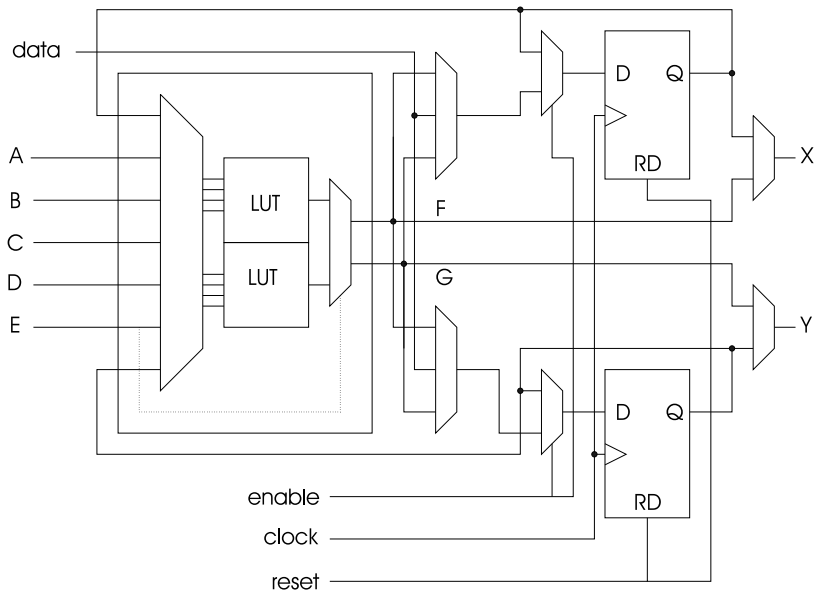


FIG. 34. Schéma simplifié de l'architecture XC3000 (Xilinx)

seul circuit. Il faut cependant noter que la famille Virtex II s'en rapproche assez bien en proposant des sommes de LUT qui elles-mêmes peuvent calculer des produits.

Dans la famille XC3000, la cellule de base (Figure 34) contient deux LUT à 4 entrées qui peuvent être regroupées en une seule LUT de 5 entrées. Selon le mode de fonctionnement, la cellule logique est donc capable de réaliser deux fonctions quelconques de 4 signaux parmi 7 ou une fonction quelconque de 5 signaux parmi 7. Les 7 signaux disponibles consistent en 5 entrées génériques et les retours des deux registres qui sont associés à la cellule logique. Des entrées dédiées sont prévues pour les registres : une donnée "data", un signal de validation "enable", une horloge "clock" et une remise à zéro "reset". C'est cette cellule qui est à la base des architectures développées ultérieurement. Le plus grand circuit de cette famille (le XC3195A) dispose d'un réseau de 22x22 cellules de base et est équivalent à environ 7000 portes logiques (selon Xilinx).

Cette cellule a subi 2 grandes transformations pour aboutir à la cellule de la famille XC4000 :

– La possibilité de modifier les mémoires des LUT en cours de fonctionnement. En effet, les LUT ne sont rien d'autre que des petites RAM qui étaient configurées au démarrage. L'idée de cette architecture consiste

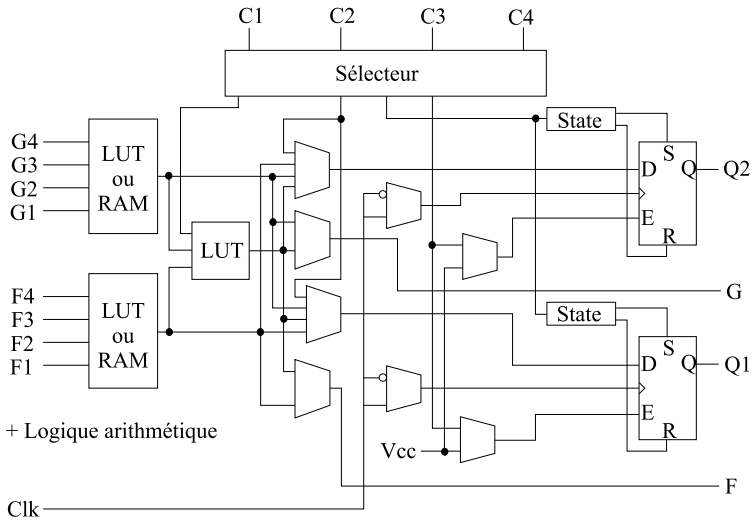


FIG. 35. Schéma simplifié de l'architecture XC4000 (Xilinx)

à pouvoir modifier le contenu de ces mémoires en cours de fonctionnement, c'est-à-dire à les utiliser comme des petites mémoires de 16x1 bits.
– L'adjonction de logique dédiée pour le calcul des opérations arithmétiques (report de retenue).

Cette nouvelle cellule est à la base de la famille XC4000 (Figure 35). D'autres modifications mineures ont également été apportées telles que l'adjonction d'une LUT à 3 entrées, un "set" (mise à "1") asynchrone en complément au "reset" déjà présent, le déclenchement sur l'horloge ou sur son complément et d'autres encore. Le plus large circuit de cette famille (le circuit XC40250) dispose d'un réseau de 92x92 cellules de base et est équivalent à environ 250000 portes logiques (selon Xilinx).

Xilinx propose également des composants "haute densité" avec les familles Virtex (4 millions de portes) et Virtex II (6 millions de portes et le 10 millions de portes est annoncé pour bientôt). La famille Virtex apporte plusieurs nouveautés par rapport à la famille XC4000 :

– l'adjonction de blocs mémoires de 4Kbits au coeur de la logique et même de plus larges mémoires dans la famille "Extended Memory"
– l'utilisation de boucles à verrouillage de phase améliorées : DLL (Digital Delay Locked Loop) qui permettent de synchroniser une horloge interne sur une horloge externe, de travailler en quadrature de phase et de multiplier ou diviser la fréquence.

- La présence d'un anneau de connexions autour du circuit pour faciliter le routage des entrées-sorties.
- La compatibilité avec de nombreux standards de transmission de données et de niveaux logiques.

L'architecture des cellules logiques est toujours basée sur des LUT à 4 entrées configurables également en petites mémoires RAM 16 bits. De plus, de la logique a été ajoutée pour permettre la synthèse de plus larges LUT comme une combinaison des LUT existantes. Le plus large circuit de cette famille (le circuit XCV3200E) contient un réseau de 104x156 cellules logiques et est équivalent à environ 4 millions de portes logiques (selon Xilinx).

Finalement, la famille Virtex II Pro améliore encore un peu le modèle précédent :
- Des blocs de mémoire de 18 Kbits.
- Des multiplieurs signés de 18x18 bits vers 36 bits.
- Des buffer Tri-state en interne pour réaliser des bus.
- Le contrôle des impédances de sortie pour adapter chaque impédance à celle de la piste du circuit imprimé.
- Le DCM (Digital Clock Manager) qui est une évolution du DLL et qui affine encore le déphasage et la synthèse des horloges internes.

Le plus large circuit de cette famille (le circuit XC2V10000) contiendra 192 blocs mémoire de 18Kbits, 192 multiplieurs, un réseau de 128x120 cellules logiques et sera équivalent à environ 10 millions de portes logiques (selon Xilinx).

3.4. Les systèmes multi-FPGA

A l'époque des PAL, il était impensable de réaliser des fonctions très complexes, même avec un grand nombre de circuits. Mais avec l'avènement des FPGA, il est très vite apparu que si l'on connectait ensemble un certain nombre de FPGA (jusqu'à des milliers) associés à des mémoires et à des circuits dédiés, on pouvait réaliser des circuits de plus en plus performants. Cette approche était particulièrement prisée dans le domaine de la simulation des ASIC et du prototypage rapide. Plutôt que de simuler pendant de longues heures le comportement d'un ASIC, il suffisait de configurer un système multi-FPGA et de le tester en temps réel ou du moins beaucoup plus rapidement qu'avec un simulateur logiciel.

Il est intéressant de constater qu'un système multi-FPGA à exactement la même structure qu'un FPGA (Figure 36) :
- Un ensemble d'entrées.
- Un ensemble d'éléments logiques.
- Un ensemble d'éléments de mémorisation.

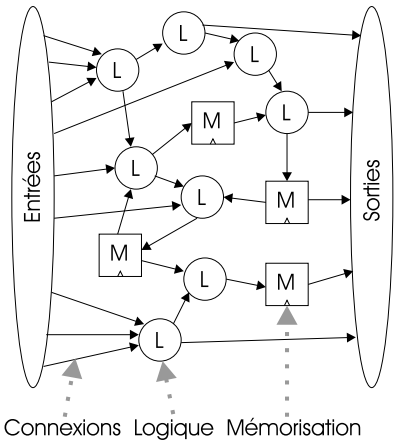


FIG. 36. Un système digital (multi-)FPGA

– Un ensemble de connexions.
– Un ensemble de sorties.

Dans le cas d'un FPGA, les fonctions logiques sont élémentaires (LUT, PLA) et les éléments de mémorisation sont de simples registres. Dans le cas d'un système multi-FPGA, les fonctions logiques sont plus complexes et les éléments de mémorisation peuvent prendre la forme de larges mémoires. C'est pourquoi l'étude des systèmes multi-FPGA d'hier se rapporte à l'étude des FPGA d'aujourd'hui. En effet, la carte que nous avons développée au début de ce travail de recherche est basée sur 4 FPGA FLEX10K100 (Altera). A l'époque, c'était le sommet de la technologie. Depuis peu, la famille APEX (Altera) propose des circuits dont le nombre de cellules logiques dépasse allègrement les 20000 cellules disponibles sur notre carte. Si on extrapole la croissance des FPGA ces dernières années (environ un facteur 15 en 5 ans) aux années à venir, on peut donc s'attendre à des circuits offrant un nombre de portes logiques impressionnant.

Un point crucial est le type de connexions utilisées. On distingue quatre architectures différentes (Figure 37) qui d'ailleurs ne sont pas limitées aux systèmes multi-FPGA mais que l'on retrouve dans tous les systèmes interconnectés (en ce compris les architectures reconfigurables de grain large étudiées à la section précédente) :

(1) L'architecture bus : C'est la plus commune. Tous les circuits sont connectés en parallèle sur un bus. A tout moment, un seul circuit a le contrôle du bus et peut envoyer des données à un ou plusieurs autres circuits. On trouve également des systèmes à

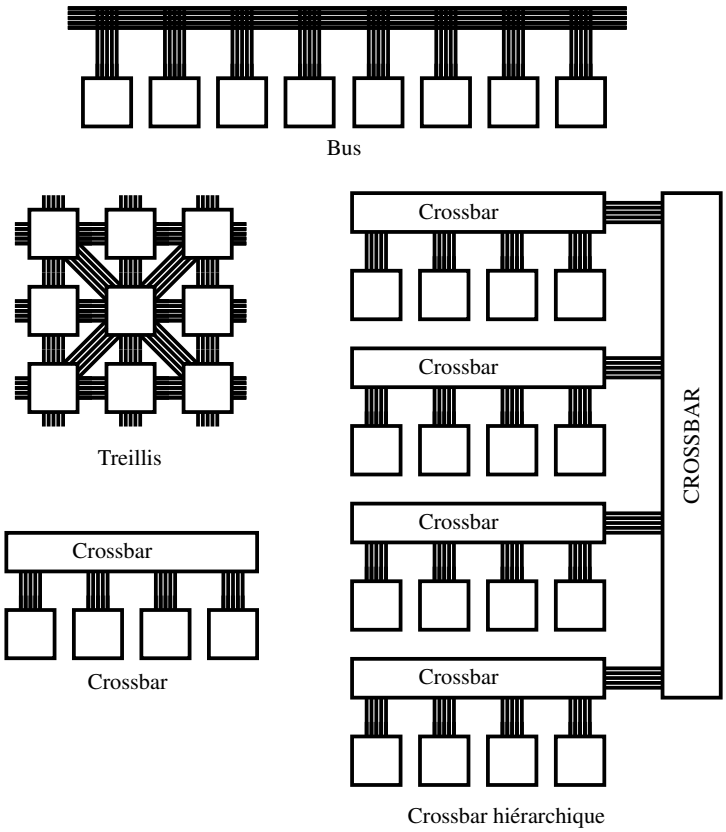


FIG. 37. Quatre types d'interconnexions

plusieurs bus permettant d'augmenter le parallélisme entre les transferts.

(2) L'architecture en treillis : Il s'agit de connexions fixes entre chaque circuit et ses voisins plus ou moins proches.

(3) L'architecture par "crossbar" : Le crossbar est un circuit qui permet d'établir toutes les connexions possibles broche à broche entre tous les circuits qui lui sont connectés.

(4) L'architecture par "crossbar hiérarchique" : Outre les connexions locales entre divers FPGA, des connexions sont prévues vers un niveau supérieur de la hiérarchie.

Historiquement, c'est l'architecture bus qui est apparue la première. Elle offre l'avantage de permettre l'interconnexion d'un nombre illimité de circuits avec un nombre très réduit de connexions. Le prix a payer est

que cette architecture ne supporte qu'un seul transfert à la fois par bus et qu'il faut gérer un protocole d'accès au bus. L'avènement des systèmes multi-FPGA et dernièrement des architectures reconfigurables de grain large nécessitait de trouver une autre alternative pour remplacer, ou du moins compléter l'architecture bus. Ces alternatives font usage de la topologie du circuit pour privilégier des transferts au niveau local, ce qui permet d'obtenir un taux de parallélisme plus important.

L'architecture en treillis offre des connexions très rapides entre deux circuits car elles sont fixes. Elles ne nécessitent donc pas de protocole d'accès et sont presque sans délai (juste le temps de propagation du signal sur un conducteur). Son inconvénient majeur est de rendre la communication difficile et lente entre des circuits éloignés. Dans ce cas, les signaux doivent transiter par des circuits intermédiaires et accaparent leurs ressources de traitement de données et d'entrées-sorties pour effectuer du routage.

L'architecture "crossbar" est théoriquement parfaite puisque toutes les connexions sont possibles entre tous les circuits. Les circuits sont donc utilisés au maximum de leurs possibilités. En pratique, le nombre de circuits qui peuvent être interconnectés est très limité car la taille du circuit de connexion est proportionnelle au carré du nombre de connexions et le plus large circuit disponible aujourd'hui (développé par Aptix) ne dépasse pas les 1024 connexions. Remarquons également que le crossbar introduit un léger délai dans la connexion et qu'il n'est pas reconfigurable instantanément.

L'architecture "crossbar hiérarchique" propose un compromis entre des connexions locales rapides et un grand nombre de circuits accessibles au moyen de la structure hiérarchique. Le grand défi consiste alors à regrouper ensemble les circuits qui ont le plus de connexions (ou qui échangent le plus de données) entre eux et à séparer les parties de circuits qui en ont moins (ou qui échangent moins de données).

C'est cette dernière architecture qui est la plus utilisée dans les FPGA. Pour relever le défi annoncé ci-dessus, les logiciels utilisent deux techniques différentes : le groupement des cellules qui partagent un grand nombre de connexions et la division de large blocs en sous-blocs peu connectés. Par des méthodes itératives, ils arrivent finalement à un optimum local mais sans garantie qu'il n'existe pas de meilleure solution [**48**]. Les systèmes les plus évolués partent de la hiérarchie existant dans les fichiers sources pour effectuer les groupements et séparations les plus judicieux [**30**]. C'est finalement les systèmes qui interagissent avec les concepteurs dans les choix de partitionnement qui donnent les meilleurs résultats.

3.5. Conclusion

Nous voyons que la technologie des circuits reconfigurables de grain fin a considérablement évolué au cours des dix dernières années. Les domaines d'application se sont eux-aussi élargis au fur et à mesure que la taille et la vitesse des circuits augmentaient. Principalement utilisés comme "logique de colle" à l'origine, ils sont maintenant capables de réaliser des fonctions de traitement de données précédemment réservées aux ASIC seuls, diminuant ainsi considérablement le temps de développement. Le nombre de projets universitaires et autres les utilisant dépasse la centaine [**37**], [**65**], sans compter les applications commerciales dédiées.

Les deux fabricants principaux de FPGA (Altera et Xilinx) ont suivi une évolution assez semblable. Partant de quelques cellules logiques, ils disposent maintenant de circuits contenant des dizaines de milliers de cellules. Les fonctions logiques sont basées sur des LUT ou des sommes de produits. Des petites mémoires locales, des multiplieurs (Xilinx uniquement) et des noyaux de processeur ont été ajoutés au coeur de la logique dans les circuits les plus récents. En outre, une technologie nouvelle a été développée pour la gestion des horloges (boucles à verrouillage de phase améliorées), pour garantir la compatibilité avec les nombreux standards de transmisson de données et dernièrement pour adapter l'impédance de sortie aux pistes du circuit imprimé (Xilinx uniquement) afin de permettre des fréquences d'horloge allant jusqu'à 400 Mhz.

Comme nous l'annoncions dans l'introduction, le grand défi consiste maintenant à offrir un langage de programmation qui ne soit pas réservé exclusivement à des experts en matériel. Les spécialistes du traitement de données, habitués à manipuler le langage C/C++ ou SimuLink trouveraient un intérêt certain à pouvoir accéder directement au matériel. Des bibliothèques de circuits logiques dédiés prêts à être utilisés existent déjà. Des synthétiseurs de circuits logiques, sur base de quelques paramètres, existent également et sont développés par des firmes spécialisées. C'est ainsi qu'il est assez facile de réaliser par exemple un filtre à réponse finie de N_c coefficients de N_b bits chacun avec une latence de N_l cycles. Néanmoins, si le travail des experts du FPGA s'en trouve facilité, ceci n'ouvre pas pour autant le monde des FPGA aux non-spécialistes.

L'architecture que nous proposons offre une approche intuitive et une simplicité de mise en oeuvre tout-à-fait à la portée des non-spécialistes du FPGA, pour autant qu'une bonne interface utilisateur soit disponible. Ce qui n'est pas encore le cas à l'heure actuelle car la réalisation d'une interface graphique sort du contexte de ce travail. Notre architecture offre aux non-spécialistes comme aux spécialistes un moyen de penser et

de réaliser de larges systèmes parallèles utilisant habilement les dernières innovations de la technologie. Nous la présentons au chapitre suivant.

CHAPITRE 3

Une architecture nouvelle, utilisant des "DDR"

1. Introduction

Lorsque nous avons développé la première carte en 1996, le but était de regrouper un maximum de logique reconfigurable sur une seule carte PCI pour pouvoir travailler à fréquence élevée sur des systèmes complexes. Le FPGA le plus large à l'époque était le FLEX10K100 (Altera). Sa taille physique (assez énorme) permettait de placer un maximum de quatre circuits sur une seule carte PCI. En outre, de la mémoire SRAM rapide a été ajoutée jusqu'à saturation de l'espace disponible sur la carte. Nous avons alors imaginé une architecture originale pour interconnecter les FPGA, la mémoire et le bus PCI.

Dans un second temps, la question du partitionnement d'un circuit logique sur les quatre FPGA a dû être traitée. Les diverses approches proposées dans la littérature (séparations et regroupements successifs, utilisation de la hiérarchie) mettant en oeuvre des logiciels assez sophistiqués, nous avons préféré opter pour une solution plus élémentaire dans un premier temps, vu la "simplicité" de notre architecture (seulement quatre circuits). Le cas échéant, il serait toujours possible d'utiliser ces techniques plus tard sur des systèmes plus complexes.

Notre approche de la question est donc tout-à-fait différente. Nous voyons le système reconfigurable comme une boite dans laquelle arrivent des données et de laquelle repartent des résultats. Ces données circulent à l'intérieur de la boite et sont traitées. Elles sont parfois stockées temporairement dans des mémoires externes avant d'être réemployées et finalement sont envoyées vers la sortie. Nous avons résolu le problème du partitionnement de la logique en l'effectuant non plus sur de la logique de bas niveau (LUT et registre de un bit) mais bien sur des opérateurs synchronisés par les données. Le partitionnement de bas niveau est ainsi laissé au logiciel de synthèse que nous utilisons tel quel. De cette manière, le nombre de noeuds à partitionner est considérablement réduit et le partitionnement peut se faire de manière intuitive. Chaque opérateur synchronisé par les données peut tourner à haute fréquence puisqu'il

est entièrement contenu dans un FPGA et les connexions entre FPGA ne sont utilisées que pour le transfert de données, sous un protocole d'auto-synchronisation efficace. Cette approche sous-tends un contrôle délocalisé qui se retrouve dans les réseaux de Pétri. C'est pourquoi ce concept mathématique a été utilisé pour soutenir notre implémentation. En outre, le degré de parallélisme n'est limité que par le nombre d'opérateurs implémentés. Le taux de transfert de données ne dépend que du nombre d'entrées-sorties disponibles sur le système (multi-)FPGA.

L'approche par réseau de Pétri nous a conduits à une architecture particulièrement souple, efficace et adaptée aux systèmes reconfigurables, quelle que soit leur taille. De plus, son caractère intuitif convient particulièrement bien aux mathématiciens, programmeurs et spécialistes du traitement du signal qui ne sont pas formés à la programmation de bas niveau des FPGA. Notre approche est à la programmation de bas niveaux ce que le C++ est à l'assembleur. Les spécialistes de l'assembleur seront toujours nécessaires pour augmenter les fonctionnalités disponibles à plus haut niveau. Néanmoins, Il n'est pas nécessaire de comprendre le fonctionnement interne d'un microprocesseur pour écrire un code C++. De la même manière, la bibliothèque des fonctions synchronisées par les données que nous utilisons ne pourra être améliorée ou augmentée que par des spécialistes des FPGA. Néanmoins, leur utilisation est ouverte à tout qui peut comprendre le concept de réseau de Pétri, ce qui est peu limitatif.

Dans un dernier temps, l'opportunité de développer une nouvelle carte s'est présentée. La carte PCI Virtex contient un VIRTEX1000 (Xilinx) et un processeur ARM. La logique reconfigurable qu'elle contient sur un seul FPGA est deux fois et demi supérieure à celle contenue sur les quatre FPGA de la première carte. Néanmoins, notre architecture s'y prête très bien puisqu'elle n'est pas limitée par la taille du système. Cette dernière carte permet donc de valider le caractère générique de notre approche.

Dans ce chapitre, nous commencerons par donner une brève description des deux cartes développées. Ensuite, nous montrerons comment, partant du concept du réseau de Pétri, nous avons construit les ressources synchronisées par les données. Nous décrirons alors la bibliothèque des ressources développées et puis la question de l'élaboration d'une architecture synchronisée par les données sera abordée. Partant d'un algorithme de traitement de données écrit en C (une convolution à 2 dimensions), nous donnerons un algorithme heuristique pour transformer un code C en une architecture synchronisée par les données. Finalement, nous

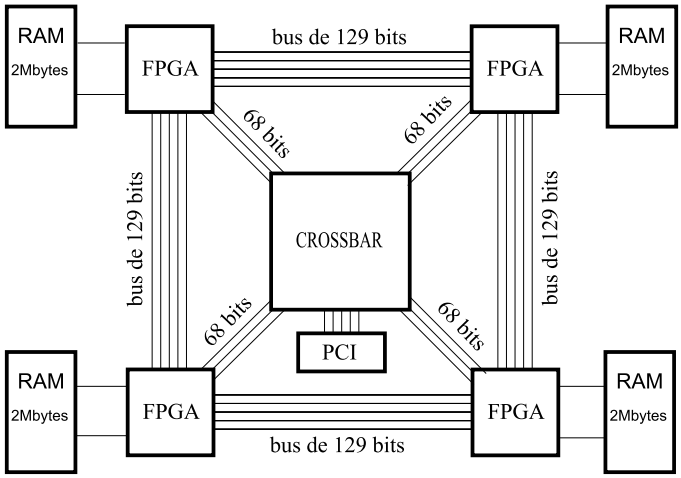


FIG. 38. L'architecture "Enhanced Mesh Topology"

détaillerons l'environnement de développement, de simulation et de test que nous avons réalisés.

2. Les cartes réalisées

2.1. La carte 400.000 portes

Le but poursuivi en réalisant cette carte était double : regrouper un maximum de logique reconfigurable et offrir une fréquence de travail la plus élevée possible. Le premier a été atteint en utilisant le plus large circuit disponible à l'époque (le FLEX10K100 d'Altera) et le second l'a été en regroupant les FPGA sur une seule carte PCI (132 Mo/s) au moyen d'une architecture originale que nous avons baptisée "Enhanced Mesh Topology" (Figure 38). Cette architecture, basée sur l'architecture en treillis et sur l'architecture "crossbar" en offre les avantages combinés (connexions rapides et génériques).

Les quatre FPGA sont connectés en carré avec des connexions fixes de 129 bits. En outre, un crossbar offre 320 connexions programmables pour interconnecter les FPGA et l'interface PCI entre eux. Nous avons ajouté une mémoire de 2 Moctets organisée en mots de 32 bits à chaque FPGA et une chaine JTAG relie tous les FPGA entre eux. L'accès au JTAG et la configuration des FPGA se fait par le bus PCI. Finalement l'interface PCI est réalisée au moyen d'un petit FPGA dont le programme est mémorisé dans une EPROM amovible.

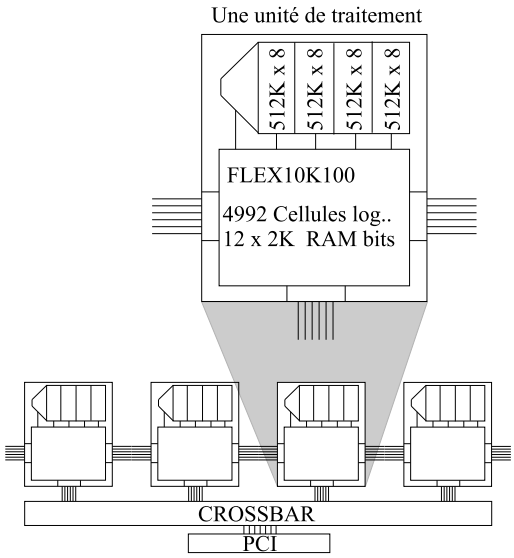


FIG. 39. Le concept d'unité de traitement

L'ensemble FPGA-mémoire peut être vu comme une unité de traitement spécifique (Figure 39). Un algorithme peut donc être décomposé en plusieurs étapes, chacune d'entre elles étant réalisée par une unité de traitement. Les données arrivent du bus PCI, circulent d'une unité de traitement à l'autre en pipeline et les résultats sont renvoyés sur le bus PCI. Chaque unité de traitement dispose de 4992 cellules logiques, 12 mémoires imbriquées de 2Kbits et une mémoire externe de 2 Moctets. La fréquence de travail maximale dépend de la complexité du circuit réalisé. Le circuit le plus simple (un compteur 16 bits par exemple) peut théoriquement tourner à 100 Mhz. Dans la pratique, un circuit optimisé manuellement peut tourner à 40Mhz tandis qu'un circuit synthétisé automatiquement tourne rarement au delà de 20 Mhz.

2.2. La carte Virtex

Cette dernière carte a été développée à la fin du travail de recherche (année 2001). L'architecture est toujours basée sur une carte PCI avec de la logique reconfigurable et de la mémoire. Néanmoins, deux nouveaux concepts ont étés introduits : la présence d'un processeur ARM avec ses périphériques et la présence de diverses interfaces permettant à la carte de fonctionner de manière autonome (USB, I/O génériques, port série).

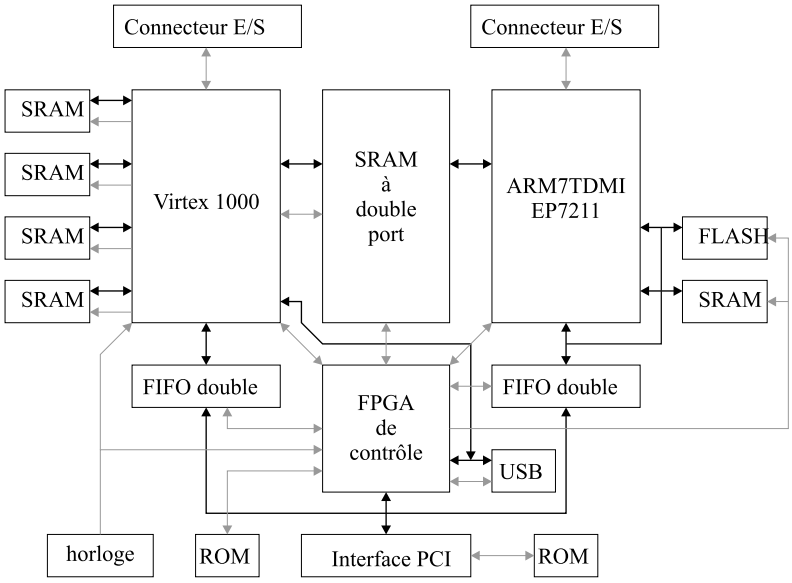


FIG. 40. Architecture de la carte PCI Virtex

La carte Virtex (Figure 40) est basée sur le circuit Virtex1000 (Un FPGA développé par Xilinx équivalent à un million de portes logiques) et le circuit EP7211 (un microcontrôleur développé par Cirrus Logic basé sur un noyau ARM720T avec gestion de mémoire et divers périphériques). Le transfert des données vers le PCI se fait au moyen de FIFO doubles (une pour le FPGA et une pour le processeur) et d'un circuit dédié pour la gestion du protocole PCI. Des mémoires 32 bits ont été connectées au FPGA et au microcontrôleur. Une mémoire à double accès permet une communication souple entre le FPGA et le micrcontrôleur. Un contrôleur USB et divers connecteurs ont été ajoutés pour permettre l'utilisation autonome de la carte. Finalement un petit FPGA de contrôle, utilisé comme "logique de colle intelligente" assure la gestion de tous les signaux de contrôle.

Comme nous l'avons vu dans l'introduction, le Virtex 1000 est équivalent à deux fois et demi les quatre FLEX10K100 de la première carte. En outre, la technologie utilisée permet d'obtenir des fréquences de travail nettement plus élevées. Ce qui multiplie les performances par un bon facteur cinq. Les mémoires utilisées sont des SRAM synchrones. Nous verrons par la suite que l'architecture que nous proposons et qui a été au départ étudiée et validée sur notre première carte est tout aussi adaptée

à l'architecture de la carte Virtex. Quelques nouvelles ressources synchronisées par les données doivent être développées pour être compatibles avec les FIFO, les SRAM synchrones et l'interface PCI qui sont différentes de l'ancienne carte.

3. Une architecture synchronisée par les données

3.1. Du concept de réseau de Pétri aux DDR

Le réseau de Pétri est un concept mathématique constitué de quatre ensembles :

(1) Les emplacements : Ils sont destinés à contenir un certain nombre de jetons. Dans la première version des réseaux de Pétri, le nombre de jetons est illimité. Sur un graphe, ils sont représentés par un cercle contenant un nombre qui représente la quantité de jetons contenus dans l'emplacement.

(2) Les transitions : Le nombre de jetons contenus dans chaque emplacement est appelé à varier en fonction des transitions. Chaque transition est connectée à un ensemble d'emplacements d'entrée et à un ensemble d'emplacements de sortie. Dans la première version des réseaux de Pétri, une transition est prête à "flamber" dès qu'il y a au moins un jeton dans tous les emplacements d'entrée. Lors du flambage, un jeton est prélevé de chaque emplacement d'entrée et un jeton est ajouté à chaque emplacement de sortie. Sur un graphe, elles sont représentées par des rectangles.

(3) Les fonctions d'entrée : Elles définissent les emplacements d'entrée de chaque transition. Sur un graphe, elles sont représentées par des flèches dont l'origine est un emplacement et la destination une transition.

(4) Les fonctions de sortie : Elles définissent les emplacements de sortie de chaque transition. Sur un graphe, elles sont représentées par des flèches dont l'origine est une transition et la destination un emplacement.

La figure 41 montre la représentation graphique du réseau de Pétri défini comme suit :

(1) $P = \{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9\}$

(2) $T = \{T_1, T_2, T_3, T_4, T_5\}$

(3) $I(T_1) = \{P_1, P_2\}; I(T_2) = \{P_3, P_4\};$
$I(T_3) = \{P_5\}; I(T_4) = \{P_6\}; I(T_5) = \{P_7, P_8\}$

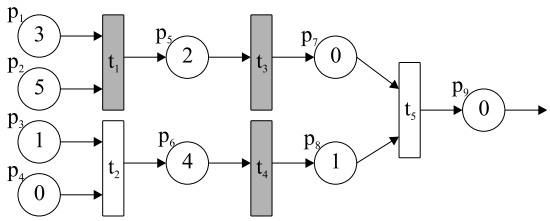


FIG. 41. Example d'un réseau de Pétri

(4) $O(T_1) = \{P_5\}; O(T_2) = \{P_6\}; O(T_3) = \{P_7\};$
$O(T_4) = \{P_8\}; O(T_5) = \{P_9\}$

La figure 42 montre l'évolution du nombre de jetons dans les emplacements d'un réseau de Pétri au fur et à mesure que les transitions flambent. Chaque transition prête à flamber a été représentée en grisé.

Plusieurs variantes, évolutions des réseaux de Pétri existent. Le nombre de jetons dans un emplacement peut être limité. Une couleur peut être attribuée aux jetons. Une notion de temporalité dans le flambage des transitions peut être introduite. D'autres règles de flambage peuvent être utilisées ...

Dans notre approche, les jetons représentent des données tandis que les transitions représentent des opérateurs. Le nombre de jetons dans un emplacement est évidemment limité par la taille physique de la mémoire utilisée. De plus, l'ordre dans lequel les jetons arrivent est également de première importance.

Nous appelons "DDR" (Data Driven Resource, c'est-à-dire ressource synchronisée par les données) (Figure 43) l'association d'une transition avec ses emplacements d'entrées. Ce regroupement a été réalisé pour une question d'efficacité d'implémentation dans la logique. Il est bon en effet que chaque opérateur soit étroitement lié aux registres qui contiennent ses opérandes pour minimiser le délai introduit par le routage. Notons que si à première vue on perd en généralité (car un emplacement d'entrée peut appartenir à plusieurs transitions), il n'en est rien. En effet, il suffit de définir un opérateur de copie de jetons qui recopiera son entrée dans toutes les DDR qui en dépendent.

Vue de l'extérieur, une DDR (Figure 44) est constituée de quatre éléments :

(1) Un ensemble de ports d'entrée synchronisés par les données. C'est par ces ports que les opérandes arrivent à l'opérateur. Nous les appellerons "canaux d'entrée".

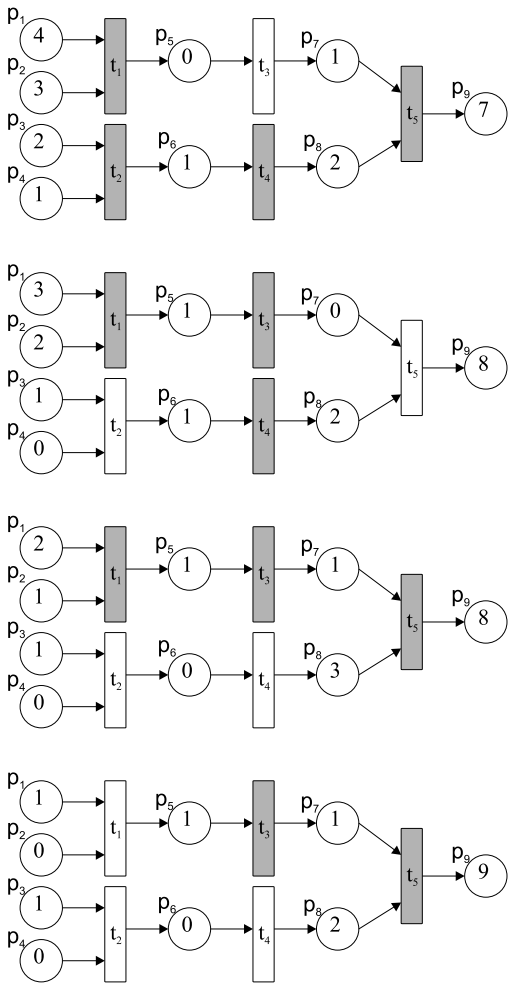


FIG. 42. Example de flambage des transitions sur un réseau de Pétri

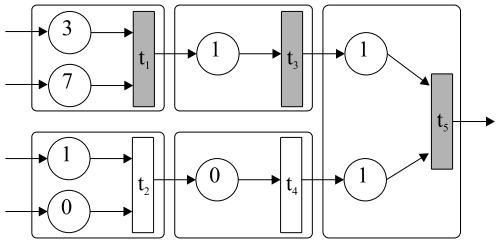


FIG. 43. Du réseau de Pétri aux DDR

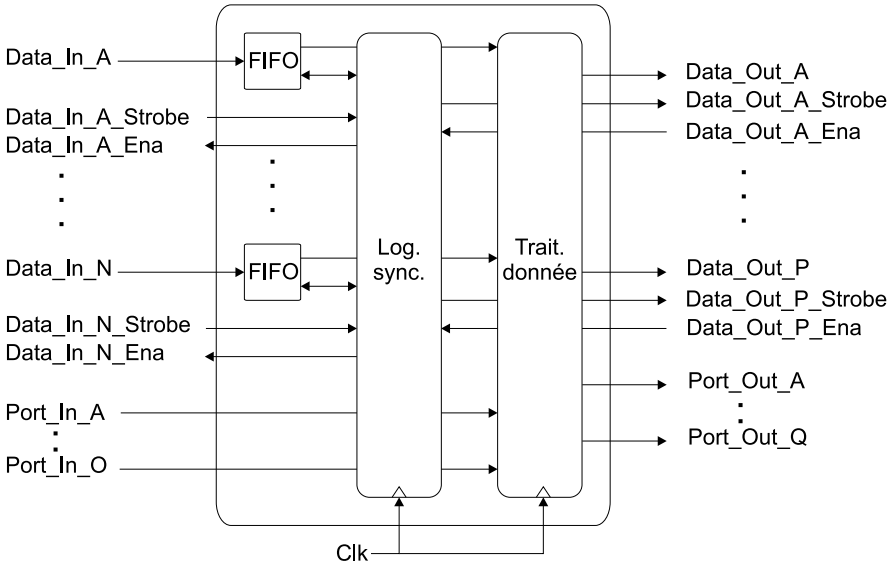


FIG. 44. Architecture d'une DDR

(2) Un ensemble de ports de sortie synchronisés par les données. Ces ports sont utilisés pour transmettre les résultats. Nous les appellerons "canaux de sortie".

(3) Un ensemble de ports d'entrée (non synchronisés).

(4) Un ensemble de ports de sortie (non synchronisés).

Les canaux sont donc utilisés pour transmettre les données à traiter. Outre la donnée elle-même, ils véhiculent deux signaux de synchronisation pour contrôler le transfert des données (voir section suivante).

Les ports non synchronisés sont utilisés pour échanger tous les paramètres qui ne sont pas des données sous forme de "jetons". Ils font plutôt partie du circuit de contrôle que du chemin de donnée. Par exemple, la sélection d'un port d'entrée pour un multiplexeur synchronisé par les données se fera par un port non synchronisé tandis que les données qui y arrivent et en ressortent utilisent des canaux.

Vue de l'intérieur, une DDR est constituée de trois éléments :

(1) Un ensemble de FIFO d'entrée qui accumulent les opérandes destinées à un port d'entrée de l'opérateur. Pour des raisons d'économie de ressources, la FIFO contient le plus souvent un seul mot. Elle se réduit donc à un registre avec un bit de présence.

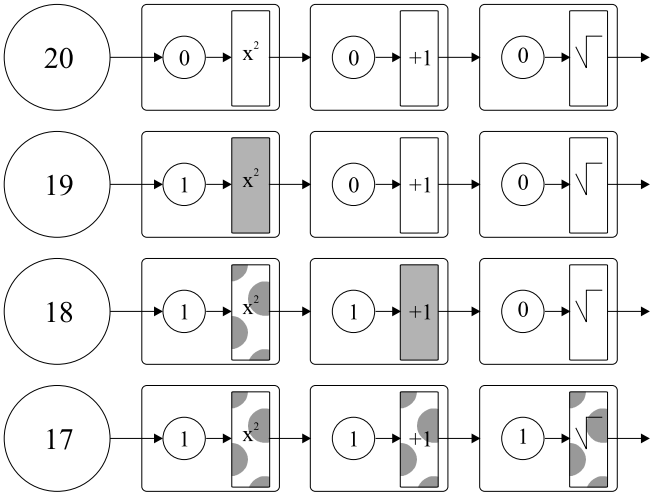


FIG. 45. Une chaine de DDR

(2) De la logique de contrôle qui arbitre les trois types de transferts de données : Extérieur vers FIFO ; FIFO vers Opérateur et Opérateur vers Extérieur. Cette logique est réalisée de manière à autoriser un pipeline régulier d'une ressource à la suivante.

(3) De la logique pour réaliser l'opération. Le terme "opération" doit être pris au sens large. Il peut s'agir bien entendu des classiques opérations logiques et arithmétiques mais également des opérations de routage, mémorisation, accès extérieur, opérateur dédié ... ou encore un générateur de données.

3.2. Le protocole de synchronisation

Toute l'architecture que nous proposons est basée sur une transmission automatique des données dès que celles-ci sont disponibles et que leur destinataire est prêt à les recevoir. C'est le caractère "automatique" qui offre l'avantage de pouvoir se passer de contrôleur centralisé et donc d'avoir un taux de parallélisme très important. En fait, il y a autant de contrôleurs que de DDR mais leur simplicité est telle qu'ils demandent peu de logique pour être réalisés. De cette façon le contrôle est réparti sur toute l'architecture et est très rapide en local.

Le choix du protocole de transmission a été posé en fonction de deux critères : la rapidité (un taux de transfert maximal) et la capacité d'anticiper la production et la destruction de jetons. Ce dernier point mérite d'être détaillé.

La taille des FIFO étant limitée (et souvent même réduite à une seule donnée), les emplacements d'entrée sont le plus souvent remplis. Néanmoins, imaginons le cas d'une chaine de DDR à une entrée et une sortie (Figure 45) qui réalise la fonction $\sqrt{x^2+1}$. Au départ, 20 données doivent être traitées. Dès qu'une valeur arrive dans l'emplacement d'entrée du premier opérateur, celui-ci est plein. A priori, il est donc impossible de rentrer une nouvelle valeur au cycle suivant. Cependant, au cycle suivant, ce premier résultat sera stocké dans l'emplacement d'entrée de l'opérateur suivant et l'emplacement d'entrée du premier opérateur sera vide alors qu'il aurait déjà pu contenir la donnée suivante à traiter. Il faut donc un protocole qui permette de "prévoir" que l'emplacement du premier opérateur va se vider pour autoriser un transfert malgré que l'emplacement de destination est déjà plein. C'est ce qui est représenté par le motif en demi-cercle grisé sur la figure. Ces transitions sont prêtes à flamber à condition que celle du successeur flambe également. Remarquons que ces conditions s'enchaînent. En effet, au cycle suivant, l'autorisation de transférer une donnée dans l'emplacement du premier opérateur (qui est déjà plein) dépend de l'autorisation qu'à le deuxième opérateur d'envoyer sa donnée au troisième ...

Le protocole que nous avons choisi est inspiré de celui utilisé par le bus PCI. L'émetteur ET le récepteur valident un signal de synchronisation quand ils sont prêts. Si les deux signaux sont valides sur un même flanc montant de l'horloge, le transfert a lieu. De cette manière, le taux de transfert maximal est la fréquence de l'horloge elle-même. Si un périphérique est plus lent, c'est lui qui impose son propre taux de transfert. En fait, ce protocole a été un peu modifié pour diminuer le chemin critique du circuit. En effet, la condition de transfert d'une valeur dépend de l'état de la FIFO et éventuellement de l'état des FIFO des successeurs. Par ailleurs, le chargement lui-même d'une donnée dans la FIFO introduit un délai. A l'instar des processeurs RISC, il nous a paru intelligent d'effectuer ces deux opérations en pipeline. A chaque cycle, la condition de transfert est calculée. Au cycle suivant, le transfert est effectivement réalisé. Cette architecture est mise en défaut si le transfert dépend de la donnée (Le pipeline est cassé). Néanmoins, de nombreux calculs intensifs ne dépendent pas des données et dans ce cas l'obtention d'une fréquence de travail supérieure est appréciable.

Les différents cas de figure sont représentés à la figure 46. L'émetteur fait passer le signal "Strobe" à '1' s'il est prêt à envoyer un résultat (au coup suivant à cause du pipeline). Le récepteur fait de même avec le signal "Enable". Si les deux signaux sont valides sur un même flanc montant de l'horloge, la donnée sera transférée au cycle suivant. L'implémentation

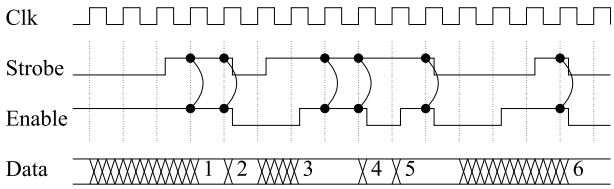


FIG. 46. Le protocole de synchronisation

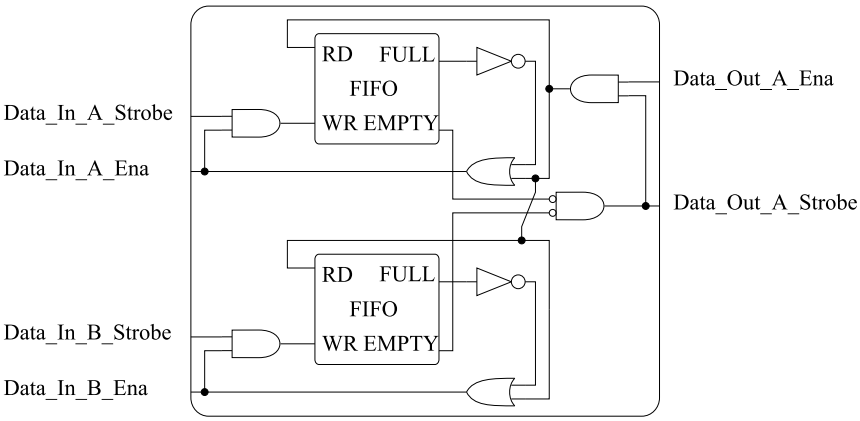


FIG. 47. Exemple d'implémentation du protocole de synchronisation : un opérateur mathématique à deux opérandes

de ce protocole dépend du type d'opérateur, principalement en ce qui concerne le nombre d'entrées, de sorties et la latence. Néanmoins, afin de permettre une bonne compréhension des cette notion de transfert automatique des données, nous allons le décrire dans le cas de figure le plus commun.

La figure 47 montre l'implémentation du protocole de synchronisation pour un opérateur arithmétique à deux entrées avec une latence de un cycle. Nous constatons que le signal "Data_Out_A_Strobe" est validé si et seulement si les deux FIFO contiennent au moins une donnée chacune. En effet, le résultat peut alors être envoyé. Si le signal "Data_Out_A_Ena" est également validé, ce qui veut dire que la ressource suivante est prête à recevoir le résultat que nous souhaitons envoyer, alors un mot sera retiré de chacune des FIFO et les signaux Data_In_x_Ena sont validés puisqu'il y a au moins une place de libre dans la FIFO. Dans le cas contraire, ces signaux ne sont validés que si les FIFO correspondantes ne sont pas pleines. Remarquons la simplicité, la rapidité (quelques

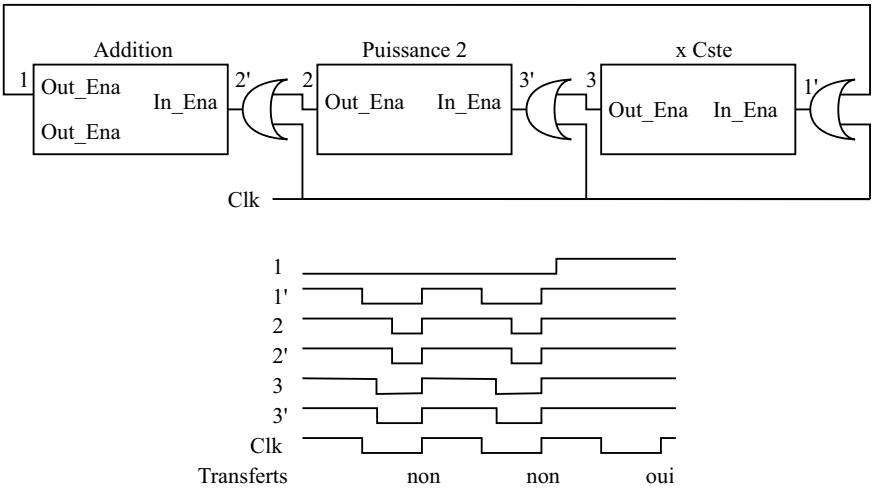


FIG. 48. Utilisation d'une boucle combinatoire à bon escient

portes logiques) et la puissance de cette architecture qui permet une implémentation directe d'un quelconque réseau de Pétri avec un taux de calcul de une opération par cycle pour chaque ressource. Un restriction doit néanmoins être faite à l'égard des systèmes bouclés. En effet, cette architecture introduit une dépendance combinatoire entre sorties et entrées. La connexion de plusieurs DDR en boucle amènerait inévitablement une boucle combinatoire sur le signal "Enable". Lorsque toutes les FIFO d'entrée des ressources de la boucle sont pleines, elles peuvent théoriquement toutes envoyer leur résultat à la suivante et acquérir le résultat de la précédente sur le même flanc montant de l'horloge. Néanmoins, la boucle combinatoire autorise deux états stables : toutes les ressources de la boucle sont bloquées ou toutes les ressources de la boucle sont autorisées à transférer leurs résultats. Cette dernière solution est la seule acceptable évidemment. Pour contourner ce problème, nous proposons deux solutions :

(1) Identifier chaque boucle et insérer une ressource spéciale qui ne contient pas de dépendance combinatoire entre les entrées et les sorties de synchronisation. Le registre s'y prête très bien (voir la section sur la bibliothèque de DDR).

(2) Utiliser la boucle combinatoire à bon escient en forçant la boucle à prendre le seul état stable qui nous intéresse : tous les signaux "Enable" sont validés.

La deuxième solution se base sur le fait que les boucles combinatoires n'introduisent pas d'instabilité mais au contraire, deux états stables. Ainsi, il faudrait forcer la boucle à prendre l'état stable dans lequel tous les signaux "Enable" sont valides. Pour y parvenir, nous proposons d'introduire entre le signal "Data_Out_..._Ena" et "Data_In_..._Ena" une porte logique OU dont une entrée est connectée à l'horloge (Figure 48).

Nous voyons que dans la première partie du cycle d'horloge, toutes les ressources reçoivent un signal "Data_Out_..._Ena" (1', 2' et 3') valide puisque l'horloge l'est aussi. Chaque ressource est donc amenée à calculer ses propres signaux "Data_In_..._Ena" comme si elle était sûre de pouvoir transférer une donnée au cycle suivant.

Si chaque donnée est prête à transférer une donnée pour autant que la suivante soit prête à en recevoir, alors un transfert peut avoir lieu et aura lieu au prochain flanc montant de l'horloge. En effet, dans ce cas chaque porte OU va voir ses deux entrées validées et quand l'horloge repassera à zéro, la boucle combinatoire gardera sa valeur valide pour le prochain flanc montant de l'horloge. Ce cas de figure est illustré au dernier cycle représenté sur la figure 48.

Dans le cas contraire, au moins une porte OU aura ses deux entrées invalidées au moment ou l'horloge repassera à zéro. Sa sortie sera donc invalidée également et se propagera de ressource en ressource aboutissant à l'état stable invalide. Aucun transfert n'aura donc lieu au prochain flanc montant de l'horloge. C'est ce qui est illustré aux deux premiers cycles de la figure 48. Dans cet exemple, l'additionneur n'est pas prêt à recevoir de nouvelles données (parce qu'il n'a pas encore reçu sa deuxième opérande par exemple). Ceci entraîne (avec un petit délai) que l'opérateur "x Cste" n'est plus prêt non plus et de même pour l'opération "Puissance 2" avec encore un petit délai.

Cette solution introduit néanmoins un délai de propagation du signal "Enable" proportionnel au nombre de ressources dans la boucle. Il est donc parfois plus intéressant de garder la première solution. Remarquons qu'il serait également possible de recalculer la fonction combinatoire de tous les signaux "Enable" d'une boucle en fonction de l'état de chacune des ressources. Néanmoins, ceci entraînerait des fonctions combinatoires complexes et demanderait une formalisation de l'état de chacune des ressources qui rendrait l'implémentation plus compliquée et moins générique. La deuxième solution est en tous cas intéressante dans le cas très fréquent des boucles de une ou deux ressources (quand un résultat est directement redirigé vers une opérande comme dans le cas d'un accumulateur par exemple).

3.3. Une bibliothèque de DDR

3.3.1. *Introduction*

Une fois le système d'auto-synchronisation élaboré, toutes les opérations réalisables par réseau de Pétri sont à la portée d'un FPGA, dans la limite des ressources disponibles. Nous avons donc décidé de réaliser un ensemble de ressources paramétrables que nous pourrions utiliser par la suite comme briques de base pour réaliser des architectures plus complexes. Nous avons réalisé plusieurs classes de DDR :

(1) Les DDR de traitement de données : Ces ressources réalisent les opérations logiques et arithmétiques courantes. Elles sont facilement extensibles à toute opération combinatoire d'un nombre quelconque d'opérandes.

(2) Les DDR de transfert de données : Il est souvent nécessaire de transmettre la sortie d'une DDR à une ou plusieurs autre(s) parmi un ensemble pré-défini. A l'inverse, une entrée peut prendre ses valeurs depuis plusieurs sources différentes. Les DDR de transfert de données permettent d'aiguiller les données depuis les sources vers les puits.

(3) Les DDR de stockage de données : Quel que soit le système considéré, il est presque toujours nécessaire de stocker temporairement des données pour les récupérer par la suite. Diverses DDR sont prévues selon le nombre de données à mémoriser et le type d'accès qui doit y être fait.

(4) Les DDR de génération de données : Ces ressources ont la particularité de ne pas prendre de données d'entrée mais de générer elles-même des séquences pré-programmées.

(5) Les DDR de test : Ces ressources sont utilisées pour conditionner le transfert de données aux valeurs de celles-ci.

(6) Les DDR d'entrées/sorties : Tous les périphériques connectés au FPGA doivent pouvoir envoyer ou recevoir des données. Ainsi, pour chaque type de périphérique, il faut une DDR adaptée qui convertit le protocole propre au périphérique dans le protocole d'auto-synchronisation que nous proposons.

(7) Les DDR spécialisées : Certaines fonctions ne peuvent être classées dans les rubriques ci-dessus. Cette rubrique reprend aussi toutes les fonctions définies par l'utilisateur.

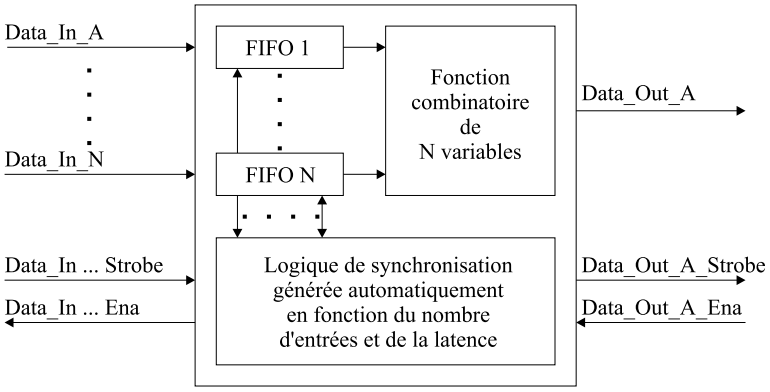


FIG. 49. Les DDR de type traitement de données

3.3.2. *Les DDR de traitement de données*

Toute fonction logique dont la sortie ne dépend que de ses entrées (par opposition aux fonctions disposant d'une mémoire) peut être réalisée au moyen d'une DDR de traitement de données. C'est le cas par exemple des fonctions logiques et arithmétiques de base mais également de fonctions plus complexes telles que les fonctions trigonométriques, les fonctions d'encryption et bien d'autres encore. En effet, la DDR de traitement de données est constituée de deux parties (Figure 49) :

(1) Une partie de synchronisation automatique des données qui gère les signaux "enable", "strobe" et l'accès aux FIFO d'entrée. Cette logique est générée automatiquement selon le nombre d'opérandes et la latence de la fonction.
(2) Une fonction combinatoire dont les entrées sont les opérandes et les sorties sont le résultat.

La logique de synchronisation fait en sorte d'attendre qu'au moins une donnée soit présente dans chaque FIFO. A ce moment, les données sont présentées à la fonction combinatoire qui en calcule le résultat. Une latence de plusieurs cycles peut être prise en compte le cas échéant. Cette logique assure un taux de transfert maximal en fonction de la latence et de la disponibilité de la ressource suivante. Dans les meilleurs cas (qui sont assez fréquents), un résultat est calculé à chaque cycle.

3.3.3. *Les DDR de transfert de données*

Les données doivent nécessairement transiter entre les diverses ressources de traitement de données, éléments de mémorisation, entrées-sorties

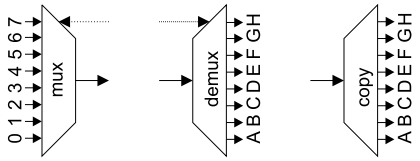


FIG. 50. Les DDR mux, demux et copy

et autres types de DDR. Trois ressources de base (Figure 50) ont été développées pour gérer le transfert des données provenant de plusieurs sources et allant vers divers puits :

(1) Le multiplexeur "mux" : Cette ressource dispose d'un nombre paramétrable de canaux d'entrée, d'un port de sélection asynchrone et d'un canal de sortie. Un et un seul canal d'entrée (déterminé par le port de sélection) est connecté au canal de sortie. Ceci veut dire que tous les signaux "enable" des canaux d'entrée sont inactifs sauf celui qui correspond au canal sélectionné, lequel recopie le signal "enable" du canal de sortie. Le signal "strobe" du canal de sortie recopie simplement celui de l'entrée sélectionnée et il en va de même pour le signal de donnée. Lors d'un changement de configuration sur le port de sélection, la modification de la sélection des données s'effectue un cycle après celle de la sélection des signaux de contrôle pour rester cohérent avec le protocole de synchronisation. De cette manière, le flux de donnée sur le canal sélectionné est transmis sans délai à la sortie et un changement de sélection se fait sans perdre de cycle d'horloge. Les canaux non sélectionnés sont bloqués en attendant leur tour.

(2) Le démultiplexeur "demux" : Cette ressource est le dual de la précédente. Elle dispose d'un canal d'entrée, d'un port de sélection asynchrone et d'un nombre paramétrable de canaux de sortie. Un et un seul canal de sortie (déterminé par le port de sélection) est connecté au port d'entrée. Les signaux "strobe", "enable" et les données sont gérés de manière semblable à celle décrite dans le multiplexeur. Ainsi, le canal d'entrée peut être aiguillé vers une sortie arbitraire sans délai.

(3) Le copieur "copy" : Cette dernière ressource dispose d'un seul canal d'entrée et d'un nombre variable de canaux de sortie. Pour une donnée qui arrive sur le canal d'entrée, une donnée de la même valeur est transmise sur chacun des canaux de sortie.

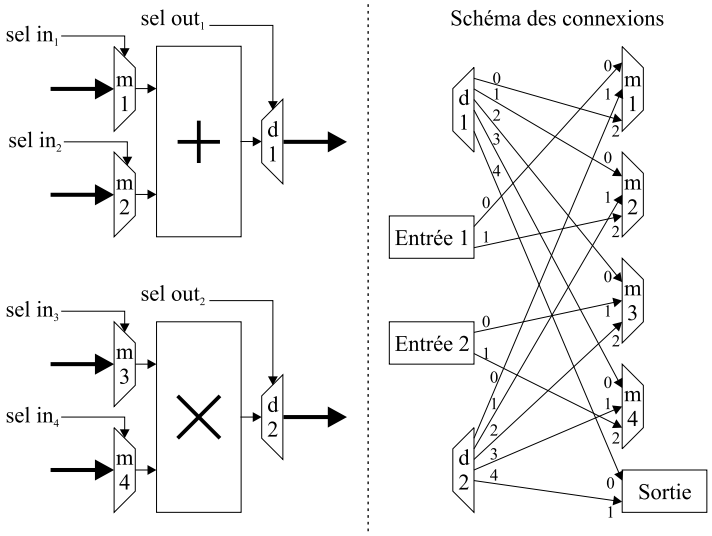


FIG. 51. Exemple d'une architecture combinant des sources et des puits de données

Les ressources "mux" et "demux" sont de véritables "aiguillages" de données. Leur combinaison avec un nombre quelconque de sources et de puits de données permet de réaliser des connexions programmables en logique synchronisée par les données. La figure 51 donne un exemple d'une architecture permettant de réaliser n'importe quelle somme de produit (ou produit de somme) avec seulement deux ressources de traitement de données. Pour y parvenir, ces opérateurs (un additionneur et un multiplieur) peuvent prendre leurs opérandes ("m_1","m_2","m_3" et "m_4") de l'extérieur ("Entrée 1" et "Entrée 2") ou réutiliser un résultat intermédiaire ("d_1" et "d_2"). Remarquons que le transfert entre une source et un puits de données ne peut se faire que si le multiplexeur et le démultiplexeur qui leurs sont associés sont tous les deux positionnés pour être connectés ensemble. Ceci permet d'attendre que les transferts en attente sur un canal soient réalisés avant de passer aux suivants et de respecter ainsi l'ordre dans lequel les calculs doivent être réalisés.

Chaque ressource de transfert de données dispose de son port de sélection asynchrone qui doit donc être commandé par une ressource lui aussi. La philosophie que nous avons choisie est d'opter pour un contrôle délocalisé. Ce qui signifie que contrairement à l'approche traditionnelle dans laquelle un contrôleur spécialisé gère tous les ports de commande, nous préférons que ce soit le passage des données qui agisse sur ces

mêmes ports de commande. Cela revient à utiliser une multitude de petits contrôleurs. Nous voyons en effet qu'une fois qu'un multiplexeur ou un démultiplexeur a été positionné d'une certaine manière, c'est qu'un transfert doit avoir lieu sur ce canal. Il n'y a donc pas de raison de changer la commande de sélection tant que le transfert attendu n'est pas réalisé. L'exemple suivant permet de mieux comprendre ce concept. Il s'agit d'utiliser l'architecture décrite sur la figure 51 pour réaliser le calcul suivant :

$$Sortie = x_1x_2x_3 + y_1y_2y_3 + z_1 + z_2 + z_3$$

Dans les tableaux qui suivent, nous allons étudier le comportement des diverses ressources constitutives de notre architecture. Les mentions T_i réfèrent au numéro du transfert qui a lieu sur la ressource concernée.

(1) Les données d'entrée :
- "Entrée 1" envoie successivement z_1, z_2, z_3.
- "Entrée 2" envoie successivement x_1, x_2, x_3, y_1, y_2, y_3.

(2) La sélection des canaux de sortie des démultiplexeurs :

Source	T_1	T_2	T_3	T_4	T_5	T_6
"Entrée 1"	0 (m_1)	1 (m_2)	1 (m_2)			
"Entrée 2"	0 (m_3)	1 (m_4)	1 (m_4)	0 (m_3)	1 (m_4)	1 (m_4)
"d_1"	0 (m_1)	0 (m_1)	0 (m_1)	4 (Sortie)		
"d_2"	2 (m_3)	1 (m_2)	2 (m_3)	1 (m_2)		

(3) La sélection des canaux d'entrée des multiplexeurs :

Puits	T_1	T_2	T_3	T_4
"m_1"	0 (Entrée 1)	2 (d_1)	2 (d_1)	2 (d_1)
"m_2"	2 (Entrée 1)	2 (Entrée 1)	1 (d_2)	1 (d_2)
"m_3"	0 (Entrée 2)	2 (d_2)	0 (Entrée 2)	2 (d_2)
"m_4"	2 (Entrée 2)	2 (Entrée 2)	2 (Entrée 2)	2 (Entrée 2)
"Sortie"	0 (d_1)			

(4) L'évolution temporelle des opérandes :

Cycle	m_1	m_2	m_3	m_4	OP	Sortie
t1	z_1		x_1			
t2		z_2		x_2		
t3	$z_1 + z_2$	z_3	x_1x_2	x_3	+ X	
t4	$z_1 + z_2 + z_3$	$x_1x_2x_3$	y_1		+ X	
t5	$z_1 + z_2 + z_3 + x_1x_2x_3$			y_2	+	
t6	$z_1 + z_2 + z_3 + x_1x_2x_3$		y_1y_2	y_3	X	
t7	$z_1 + z_2 + z_3 + x_1x_2x_3$	$y_1y_2y_3$			X	
t8					+	OK

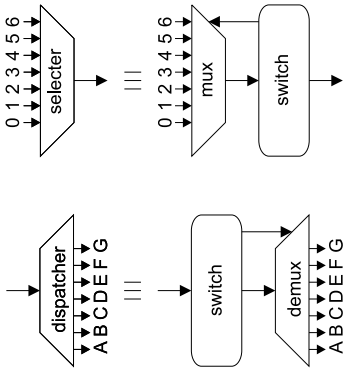


FIG. 52. Les ressources autonomes "dispatcher" et "selecter"

Constatons que le parallélisme a été réalisé à tous les endroits possibles en tenant compte qu'il n'y avait que deux canaux d'entrée et que le chemin critique de ce schéma de calcul est de 4 opérations.

La commande des ports de sélection des multilexeurs et démultiplexeurs est prise en charge par une ressource spécialisée que nous avons développée à cet effet : le "switch". Celle-ci dispose d'un canal d'entrée, d'un canal de sortie et d'un port de sortie asynchrone destiné à commander le port de sélection qui lui est attaché. Le canal de sortie recopie le canal d'entrée mais il en surveille les transferts. Dès qu'un transfert a lieu, une nouvelle valeur est chargée sur le port de sortie asynchrone. La combinaison switch-demultiplexeur a été baptisée "dispatcher" tandis que la combinaison switch-multiplexer a été baptisée "selecter" (Figure 52). Ces ressources sont maintenant autonomes. Le stockage des mini-programmes dans le switch se fait aisément au moyen de LUT (voir la section suivante).

3.3.4. *Les DDR de génération de données*

Les données en question sont le plus souvent des adresses destinées aux accès mémoire. Elles sont également utilisées par les ressources de transfert de données (switch) pour mémoriser les séquences de transferts. Il faut remarquer que tous les générateurs de données travaillent en parallèle, permettant ainsi de ne pas perdre de temps en calcul d'adresses. Trois type de générateurs ont étés développés :

(1) Les générateurs arithmétiques : Leur nom vient du fait que l'adresse suivante est calculée sur base d'un calcul qui dépend d'un certain nombre de paramètres plutôt que d'être mémorisée.

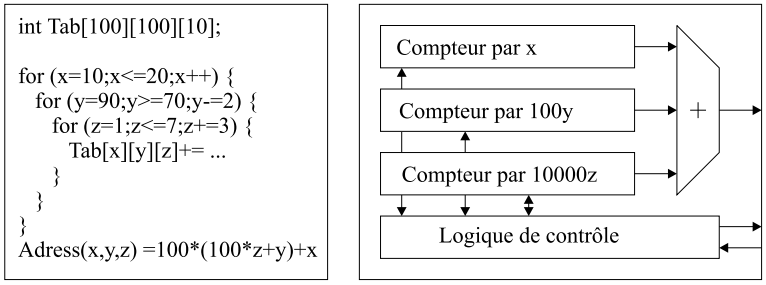


FIG. 53. Exemple d'une DDR de type générateur arithmétique de données

Ils permettent de générer une séquence d'accès mémoire à des tableaux de une, deux ou trois dimensions correspondant à une structure en boucles imbriquées (Figure 53). Trois ressources ont été réalisées : simple boucle, boucle imbriquée à l'intérieur d'une autre et triple boucle imbriquée. Ces générateurs ne sont pas limités à l'accès à des tableau de données. En effet, l'accès à un tampon circulaire dans le cas d'une convolution est réalisable avec un générateur à deux dimensions. Les paramètres de chaque boucle sont la position de départ, l'incrément et le nombre d'itérations.

(2) Les générateurs utilisant les LUT : Comme leur nom l'indique, ces générateurs utilisent les propriétés des LUT pour mémoriser une séquence de données. La LUT est particulièrement bien adaptée à la mémorisation d'un certain nombre de données lorsque ce nombre n'est pas trop élevé. En effet, N_n LUT de N_i entrées permettent de mémoriser une séquence de 2^{N_i} données de N_n bits. Un peu de logique de synchronisation est adjointe aux LUT pour gérer les envois selon le protocole que nous avons défini. Ces générateurs sont principalement utilisés par les ressources de transfert de données (switch). Les LUT contiennent le mini-programme de commande des transferts. Lorsque les tailles de séquences à mémoriser deviennent plus importantes, le nombre de LUT devient prohibitif et il vaut mieux employer les générateurs utilisant les mémoires imbriquées.

(3) Les générateurs utilisant les mémoires imbriquées : Au cours de l'exécution de larges calculs dans lesquels aucune structure arithmétique régulière ne peut être extraite, il est nécessaire de mémoriser de longues séquences d'accès à des mémoires. C'est le cas par

exemple pour les larges schémas de calculs utilisés en robotique. Dans ces cas, l'utilisation de mémoires s'impose. Les générateurs utilisant les mémoires imbriquées offrent simplement la logique de synchronisation nécessaire pour envoyer séquentiellement une table de données préalablement mémorisée dans une ROM interne selon le protocole que nous avons défini.

Nous avons également réalisé des générateurs spécifiques à des utilisations courantes telles que la mémoire FIFO et la mémoire FIFO étendue. De plus amples informations sur ces ressources sont données à la section traitant des DDR de stockage (point 3.3.5).

(1) Le générateur d'adresses de FIFO : Une FIFO peut être vue comme une mémoire classique à laquelle on a adjoint une logique spécialisée pour la gestion des adresses d'écriture et de lecture. En effet, si l'on peut garantir que la séquence d'adresses de lecture est la même que celle des adresses d'écriture et qu'une lecture ne peut avoir lieu que quand une écriture a eu lieu précédemment à la même adresse, alors l'effet FIFO est observé. Dans le cas qui nous concerne, le générateur dispose donc de deux canaux de sortie : un pour les adresses d'écriture et l'autre pour les adresses de lecture. Les adresses d'écriture et de lecture sont générées linéairement selon le protocole utilisé. Une logique basée sur les adresses permet d'interrompre la génération des adresses de lecture quand la FIFO est vide ou d'interrompre la génération des adresses d'écriture quand la FIFO est pleine. Ce générateur d'adresses est destiné à être connecté à la DDR mémoire SRAM qui garantit qu'une adresse d'écriture acceptée correspond à une écriture en mémoire le cycle d'horloge suivant.

(2) Le générateur d'adresses de FIFO étendue : Ce générateur est le plus évolué que nous ayons développé. Il est utilisé pour générer les accès à des mémoires caches. Le principe de ces mémoires est de stocker les résultats qui arrivent linéairement, comme dans une FIFO normale. Néanmoins, les accès peuvent se faire de manière aléatoire pour autant qu'une donnée ait été sauvegardée précédemment à l'adresse demandée. Pour obtenir cet effet, le générateur d'adresses de FIFO étendue dispose d'un canal d'entrée (destiné à recevoir la séquence d'accès aléatoires) et de deux canaux de sortie (un pour l'adresse d'écriture et l'autre pour l'adresse de lecture). La génération des adresses d'écriture se fait de la même manière que pour une FIFO normale. Les adresses de lecture sont simplement une copie du

canal d'entrée. Néanmoins, cette génération est interrompue si l'adresse demandée n'a pas encore été générée par le générateur des adresses d'écriture et acceptée par la mémoire SRAM qui lui est connectée. Ceci garantit que les données ne sont pas envoyées hors de la cache avant d'y être arrivées.

3.3.5. *Les DDR de stockage de données*

L'exemple étudié dans le cadre des ressources de transfert de données (Figure 51) peut paraître exhaustif pour tout calcul ne nécessitant que des additions et des multiplications et il semble assez simple à étendre à davantage d'opérations. Ce n'est que partiellement vrai. Toute somme de produit peut effectivement être réalisée au moyen de l'architecture en question. Néanmoins, décomposer un calcul en somme de produits ou en produit de sommes peut conduire à un nombre d'opérations beaucoup plus grand que si on avait mémorisé quelques résultats intermédiaires, sans compter la perte de précision éventuellement occasionnée. Par ailleurs, toute architecture traitant des tableaux de données doit nécessairement les mémoriser dans des ressources dédiées lorsqu'elles ne sont pas utilisées.

Nous avons donc développé trois types de ressources de base pour le stockage des données : Le tampon, le registre et la mémoire SRAM. Cette dernière, lorsqu'elle est couplée à d'autres DDR, offre des possibilités plus complexes telles que la FIFO, la FIFO étendue et la mémoire à accès partagés.

(1) Le tampon de taille N : Cette ressource dispose d'un canal d'entrée, d'un canal de sortie et d'un port asynchrone de statut. Elle s'intercale entre la sortie d'une DDR et l'entrée de la suivante lorsque l'on désire éviter des blocages intempestifs des transferts sans perdre de cycle d'horloge. En effet, cette mémoire se comporte comme une FIFO de petite taille. Elle est transparente tant que la ressource suivante est capable d'accepter des données. Dès que celle-ci est saturée, la ressource joue son rôle de FIFO. En outre, le port de statut indique si le tampon est vide ET que la ressource suivante est prête à recevoir de nouvelles données. Auquel cas la ressource précédente est certaine de pouvoir transférer au moins N+1 données sans interruption. Cette fonctionnalité est bien utile en aval d'une ressource de traitement de donnée disposant d'une latence supérieure à un et capable d'utiliser le pipeline. En effet, si un tampon de taille N est vide, une ressource de latence N+1 peut accepter N+1

données d'entrée tout en étant sûre que les résultats correspondants à ces N+1 données déjà engagées seront acceptés en sortie. Si la ressource qui suit le tampon est capable de suivre le débit, on obtient un taux de calcul de une opération à chaque cycle tout en utilisant le pipeline au sein même de la ressource de traitement de données.

(2) Le registre : Cette ressource est également du type FIFO de petite taille. Elle dispose néanmoins de deux places de mémorisation pour les données. En revanche, elle n'est pas aussi "transparente" que le tampon car si cette mémoire est capable d'accepter et d'envoyer un mot à chaque cycle d'horloge, elle introduit cependant un délai minimum d'un cycle d'horloge. Elle est surtout utilisée pour sa capacité à interrompre une chaîne combinatoire de synchronisation trop longue (entrainant une fréquence d'horloge trop faible) ou pour casser une boucle combinatoire formée par une topologie en boucle de DDR. En effet, contrairement à pratiquement toutes les autres DDR, il n'existe aucune dépendance combinatoire entre les entrées de synchronisation du registre et ses sorties.

(3) La mémoire SRAM : C'est la ressource la plus évoluée des ressources de stockage. Elle dispose de 3 canaux d'entrée (adresse d'écriture, adresse de lecture, donnée d'écriture) et un canal de sortie (donnée de lecture). En outre, elle utilise un certain nombre de ports asynchrones qui sont destinés à être connectés à une SRAM synchrone (interne ou externe au FPGA) et un port asynchrone de validité d'accès double. La taille de la mémoire est fonction de la SRAM qui est connectée à ses ports. Une opération d'écriture a lieu dès qu'une donnée arrive sur le canal d'adresse d'écriture ET qu'une donnée arrive sur le canal de donnée d'écriture. Ces deux données ne sont acceptées par la ressource que quand elles sont présentes simultanément. Ceci garantit que quand une adresses d'écriture est acceptée, l'écriture aura lieu au cycle suivant car l'opération d'écriture est prioritaire sur l'opération de lecture. C'est capital pour l'utilisation des mémoires en tant que FIFO ou FIFO étendue (voir ci-dessous). Une opération de lecture est enclenchée dès qu'une adresse de lecture est acceptée. La donnée correspondant à l'adresse en question arrive avec un délai d'un cycle d'horloge. L'acceptation d'une adresse de lecture est subordonnée à trois conditions :

- La ressource connectée au canal de sortie est prête à recevoir une nouvelle donnée.

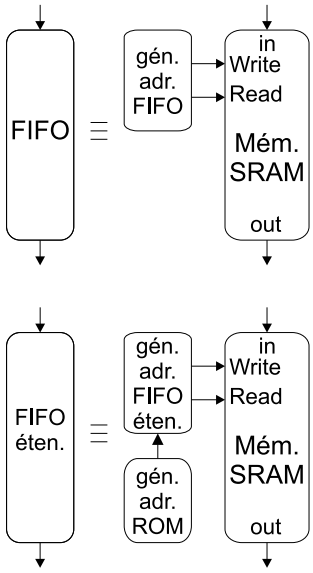


FIG. 54. La FIFO et la FIFO étendue

– Si une opération de lecture était déjà en cours, le port d'entrée asynchrone d'autorisation d'accès double doit être actif.
– Il n'y a pas d'opération d'écriture possible à ce moment là. Sinon, ce serait elle qui serait prioritaire.

La deuxième condition vient du délai d'un cycle d'horloge introduit par cette ressource. Il se peut en effet qu'une deuxième demande de lecture soit introduite avant que la première donnée ne soit acceptée par la ressource suivante. Si cette ressource n'est pas capable d'accepter deux données sans délai, la deuxième lecture ne pourra être acceptée et sera perdue. C'est pourquoi on introduit généralement une ressource tampon entre la sortie de la RAM et la ressource suivante. Celle-ci permet d'obtenir un taux de transfert d'une donnée à chaque cycle (dans un pipeline régulier) si le récepteur en est capable.

Des fonctions de mémorisation plus complexes peuvent être obtenues en interconnectant une mémoire SRAM avec diverses autres ressources telles que générateurs de données, dispatchers et selecters. Trois DDR complexes ont été réalisées de cette manière : la FIFO, la FIFO étendue et la mémoire à accès partagés.

(1) La FIFO : Son but est de permettre à deux ressources travaillant à des cadences différentes de se transmettre des données dans un ordre déterminé. Pour y parvenir, l'émetteur peut stocker des données à son rythme tant que la FIFO n'est pas pleine (indiqué par un signal "full"). Le récepteur de son coté est averti qu'il peut commencer à récupérer des données dès qu'au moins une donnée est présente en mémoire (indiqué par la négation du signal "empty"). Il peut alors récupérer la ou les données à son propre rythme. Compte tenu du protocole de synchronisation automatique des données, les signaux "full" et "empty" ne sont plus nécessaires en tant que tels dans notre architecture. La FIFO se réduit simplement à une ressource disposant d'un canal d'entrée et d'un canal de sortie construits autour d'une mémoire SRAM (Figure 54). Le canal d'entrée accepte des données tant que la mémoire n'est pas pleine tandis que le canal de sortie tente d'envoyer ses données dès que la mémoire en contient. La gestion des adresses d'écriture et de lectures est effectuée par le générateur de données spécialisé pour les FIFO (voir ci-dessus).

(2) La FIFO étendue : Nombreux sont les algorithmes qui demandent un stockage temporaire des résultats intermédiaires. Il était nécessaire de développer une ressource de type "mémoire locale" ou encore "mémoire cache" capable de remplir cette fonction. Une première approche est présentée dans la ressource FIFO étendue. Cette ressource est capable de mémoriser les données qui lui sont présentées dans leur ordre d'arrivée et de les redistribuer ensuite dans un ordre arbitraire. Elle peut également envoyer plusieurs fois la même donnée si nécessaire. Basée elle-aussi sur la ressource mémoire SRAM (Figure 54), elle dispose de deux canaux d'entrée (les données à mémoriser et les adresses de lecture) et d'un canal de sortie (les données lues). Les données arrivent donc sur le canal d'entrée qui leur est affecté et sont mémorisées dans leur ordre d'arrivée exactement comme pour la FIFO. Les lectures sont effectuées au moyen du canal d'adresses de lecture. Cependant, l'adresse de lecture n'est acceptée que si une donnée a été préalablement mémorisée à cette adresse. Dans le cas contraire, l'accès est bloqué jusqu'à ce que la condition soit remplie (c'est-à-dire que la donnée attendue soit présente en mémoire). Ceci réalise l'effet mémoire cache. La génération des adresses de lecture et d'écriture de la mémoire SRAM constitutive de cette ressource

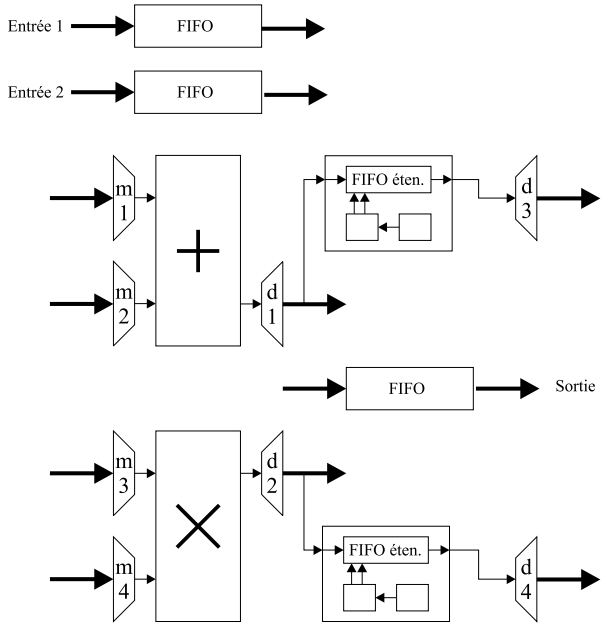


FIG. 55. Exemple d'une architecture utilisant la FIFO étendue comme mémoire cache

est assurée par un générateur de données spécialisé pour FIFO étendue (voir ci-dessus).

Partant de l'architecture qui avait servi à illustrer les ressources de traitement et de transfert de données (Figure 50), l'adjonction des ressources FIFO et FIFO étendue permet d'obtenir une unité de traitement complète et autonome (Figure 55). A chaque opération, les résultats peuvent être directement ré-injectés comme opérandes, transmis comme résultats ou stockés en mémoire cache. Cette cache dispose de son propre programme de lecture de données (réalisé au moyen d'un générateur de données basé sur une mémoire ROM par exemple). Ainsi, dès que la donnée attendue est disponible en cache, elle est transmise au dispatcher qui lui-même la fera transiter vers sa destination finale dès qu'elle pourra être acceptée par celle-ci. Les FIFO aux entrées et sorties permettent des transferts par blocs des données à traiter et des résultats. Ceci permet de ne pas monopoliser les entrées-sorties pendant tout le traitement des données et de rendre compatibles les divers taux de transfert et de traitement des données. Il est bon également de noter

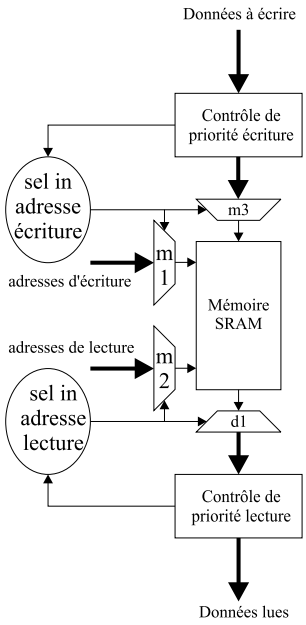


FIG. 56. Architecture d'une mémoire à accès partagés

la facilité de synchroniser l'arrivée des données à traiter et la récupération des résultats. Cette synchronisation est implicite et intégralement effectuée par les DDR.

(3) La mémoire à accès partagés : Lorsque divers algorithmes ou parties d'algorithmes doivent accéder à de larges quantités de données stockées dans une même mémoire (externe), il est nécessaire de prévoir un accès multiple en lecture comme en écriture. Cette fonctionnalité peut être facilement réalisée en interconnectant une mémoire SRAM, un multiplexer par canal d'entrée, un demultiplexer pour le canal de sortie et un système de gestion des priorités d'accès tant pour les lectures que pour les écritures (Fig 56). Ce système monitorise les transferts sur chaque canal, les données prêtes à être écrites et les ressources prêtes à recevoir les données pour déterminer le canal qui aura accès à la mémoire à un moment donné. Plusieurs algorithmes de priorité peuvent être réalisés. Nous avons implémenté un algorithme qui balaye systématiquement toutes les entrées/sorties jusqu'à ce que l'une d'entre elles soit prête à envoyer/recevoir des données. L'accès lui est alors attribué pour un nombre déterminé de transferts. Ensuite la séquence de balayage continue.

3.3.6. *Les DDR de test*

Ces ressources disposent de deux canaux d'entrée et d'un canal de sortie. C'est un sous-ensemble des fonctions de traitement de données. Le canal de sortie contient le résultat de la comparaison des deux nombres. Divers types de comparaisons peuvent être réalisées. Ces ressources peuvent servir de générateur de données pour une ressource "switch". Ainsi, les données sont aiguillées différemment selon le résultat de la comparaison.

3.3.7. *Les DDR d'entrées/sorties*

Ces ressources assurent l'acheminement des données depuis le monde extérieur vers différentes parties de notre architecture et réciproquement. Il y a autant de ressources d'entrées/sorties potentielles que de type d'entrées/sorties. On peut citer les grandes familles : série (RS232, CAN, IEEE1394, USB ...), parallèle (port parallèle PC, HP-IB, SCSI ...) et bus (PCI, VME ...). Chaque standard de communication a défini son propre protocole et ses caractéristiques physiques. Néanmoins, la généralité du protocole de synchronisation que nous avons défini nous autorise à penser que tout système de communication peut être converti à notre protocole.

Lors du test de notre architecture, nous avons utilisé le bus PCI puisque c'était le seul accès avec les entrées/sorties dont nous disposions sur la première carte développée. Deux types de ressources ont été développées : les ports PCI virtuels d'écriture et de lecture. Ces ressources permettent de définir jusqu'à 64 canaux d'entrée et 64 canaux de sortie disposés au sein de l'architecture à la convenance de l'utilisateur. Chaque canal dispose de sa propre adresse PCI. Ecrire ou lire une donnée à une adresse PCI attachée à un canal revient à envoyer ou recevoir une donnée sur ce canal. Les cas de canaux d'entrée bloqués ou des canaux de sortie vides doivent être traités de manière logicielle.

Nous avons également développé une ressource d'entrée-sortie de type RS232 ou plus généralement de type parallèle-série puisque le nombre de bits de donnée et la fréquence de transmission sont des paramètres de la ressource.

3.3.8. *Les DDR spécialisées*

Tout système logique peut être mis au format DDR. Il suffit de convertir ses ports de communication selon le protocole que nous utilisons. Rappelons qu'un des buts poursuivis est l'aspect intuitif, facilement accessible aux non spécialistes des FPGA. Il est donc important que les spécialistes,

eux, puissent développer des applications sous forme de DDR, qui viendront s'emboîter comme une pièce de puzzle dans un système créé par un non spécialiste.

3.3.9. *Conclusion*

Nous avons explicité les ressources de base constitutives des architectures synchronisées par les données que nous proposons. Certaines sont elles-même constituées de ressources encore plus élémentaires. Par un exemple, nous avons déjà entrevu comment l'interconnexion des sept familles de DDR que nous proposons pourrait déboucher sur une architecture plus complexe de traitement de données. L'étude de ces architectures est proposée à la section suivante.

4. Conception d'une architecture DDR

4.1. Introduction

Comme nous l'avons vu précédemment, n'importe quel réseau de Pétri peut être réalisé sour forme de DDR. Ces ressources dépassent néanmoins les possibilités des réseaux de Pétri en offrant des DDR spécifiques telles que les ressources de stockage et de transfert de données qui ne fonctionnent pas selon la règle de flambage habituelle (un jeton présent dans tous les emplacements d'entrée induit un jeton dans tous les emplacements de sortie).

La bibliothèque des fonctions élémentaires développées offre un large éventail de possibilités de traitement de données comme le montrera le chapitre consacré aux résultats. La présente section propose une architecture générale pour l'implémentation de ces algorithmes. Par une approche empirique, nous montrons comment transformer un algorithme conventionnel de traitement de données (écrit en C par exemple) en une architecture orientée DDR. Il s'agit d'un premier pas vers la réalisation d'un compilateur autonome pour architecture DDR. Un tel compilateur n'existe pas encore. A cette heure, le savoir faire de l'utilisateur est nécessaire pour réaliser les différentes étapes de la transformation.

L'architecture finale à laquelle on se propose d'aboutir est représentée à la figure 57. Nous y retrouvons les DDR de base décrites à la section précédente. L'algorithme est décomposé en une série de tâches travaillant en parallèle.

Chaque tâche prend ses données directement des entrées/sorties et leurs transmet ses résultats. Le cas échéant, elles font appel à la capacité de

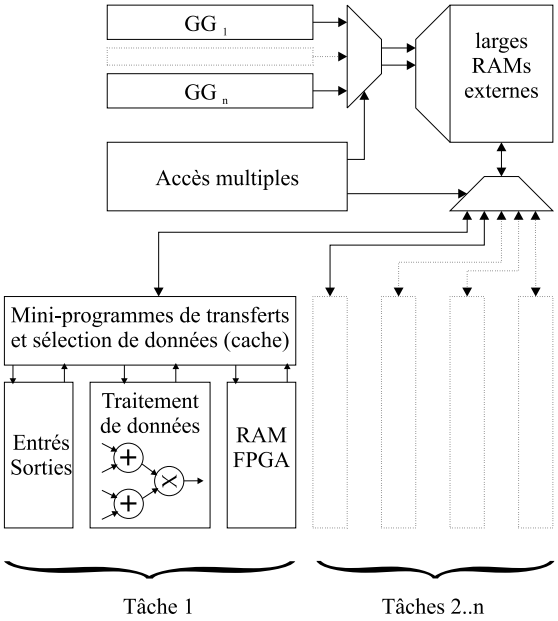


FIG. 57. Architecture générique pour l'implémentation d'un algorithme au moyen de DDR

stockage des mémoires extérieures (appelées mémoires globales) pour mémoriser les données qu'elles ne peuvent garder en local.

Une tâche est constituée au minimum d'une ressource de traitement de données. Si nécessaire, plusieurs ressources sont combinées ensemble et on peut également faire usage de dispatchers, selecters, FIFO, FIFO étendues, mémoires locales partagées, DDR spécialisées avec leurs divers mini-programmes et générateurs d'adresses comme cela a été montré dans la section décrivant la librairie de DDR.

L'accès aux mémoires globales se fait par l'intermédiaire d'une ressource mémoire partagée. Chaque tâche dispose d'un certain nombre de canaux d'accès (en écriture et/ou en lecture) à ces mémoires, selon ses besoins. Les adresses sont générées par des générateurs génériques travaillant en parallèle avec les tâches, ce qui autorise un transfert d'une donnée à chaque cycle (pour chaque mémoire) dans les meilleurs cas.

Dans les sections qui suivent, nous explicitons la méthode empirique que nous avons élaborée pour convertir un algorithme écrit en C en une architecture semblable à celle que nous venons de décrire. A titre d'illustration, nous utilisons l'exemple du calcul d'une convolution à deux dimensions sur une image.

4.2. La méthode heuristique

(1) Le partitionnement de l'algorithme en tâches : Certains algorithmes sont constitués de plusieurs parties distinctes pouvant travailler en parallèle ou en pipeline. Dans ce cas, il est bon d'identifier chacune de ces tâches de manière à implémenter un maximum de parallélisme de manière intrinséque.

(2) Le partitionnement de la mémoire (locale - globale) : Lorsque des données doivent être accessibles à plusieurs tâches, elles sont systématiquement globales. Dans le cas contraire, cela dépend de leur taille. Si elles peuvent être contenues dans une mémoire interne au FPGA, on peut les sauvegarder en mémoire locale. Dans le cas contraire, elles doivent être considérées comme faisant partie de la mémoire globale.

(3) Le partitionnement de l'algorithme (local - global) : Le code de chacune des tâches va être modifié de manière à obtenir deux programmes qui interagissent. Le premier (appelé programme local) effectue des opérations uniquement sur les données locales. Partant du code initial, tous les accès à des mémoires globales sont remplacés par les opérations virtuelles :

"Get_Global(Canal)" (pour les lectures)

"Send_Global(Donnée, Canal)" (pour les écritures)

Parallèlement, le second code (appelé programme global) est écrit pour chaque tâche. C'est lui qui gère les accès à la mémoire globale. Il échange les données avec la tâche qui lui correspond par les instructions virtuelles :

"Send_Local(Donnée, Canal)" (envoi)

"Get_Local(Canal)" (réception)

Le paramètre "Canal" correspond au numéro de canal employé pour accéder à une mémoire. Il peut en effet arriver que plusieurs tableaux soient traités simultanément dans une même mémoire et demandent plusieurs canaux d'accès. Nous voyons que dans cette décomposition, seul le programme global gère les adresses des données globales. C'est ainsi qu'elles pourront être générées par la suite par des ressources de génération de données autonomes de manière parallèle.

(4) L'étude du temps de vie des variables globales : Le but de cette étape est de déterminer les variables globales qui sont utilisées plus d'une fois à l'intérieur d'une tâche de manière à les conserver dans une mémoire locale utilisée comme cache. De cette manière, on réduit le nombre de transferts avec la mémoire

globale qui est parfois plus lente et surtout plus sollicitée puisqu'elle est commune à toutes les tâches. L'analyse du temps de vie permet de déterminer la taille de la cache. Elle correspond au nombre maximum de variables vivantes à un moment donné.

(5) L'optimisation des transferts : C'est ici qu'intervient l'effet de la mémoire cache. Il est inutile que le programme global envoie la même donnée à quelques cycles d'intervalle. Ces données sont avantageusement mémorisées dans une cache locale. Cette transformation vise donc à éliminer du programme global tous les transferts multiples d'une même valeur s'ils sont assez rapprochés que pour être utilement réalisés par une cache. Ici, c'est le savoir-faire de l'utilisateur, la connaissance de l'algorithme, et bien entendu l'étude du temps de vie des variables qui permettront un choix judicieux des valeurs à mémoriser en cache et des valeurs à transférer directement.

(6) L'implémentation des tâches : Cette partie correspond à l'implémentation des programmes locaux. Le but est de transformer le programme local en une architecture faisant intervenir des DDR de traitement de données, des FIFO, FIFO étendues, mémoires à accès partagés, dispatchers, selecters, et autres ressources synchronisées. Ici encore, c'est le savoir-faire de l'utilisateur qui apportera une solution élégante au problème posé. Le choix des ressources de traitement de données, leur nombre et leurs paramètres influenceront sur les performances du système en termes de temps de calcul et de taille du circuit. Il faut se souvenir en effet que les DDR ne se limitent pas aux simples opérations logiques ou arithmétiques mais peuvent faire intervenir des systèmes aussi complexes qu'un noyau de processeur.

(7) L'implémentation de la gestion des mémoires globales : Cette étape découle naturellement du programme global obtenu à l'issue du partitionnement de l'algorithme entre local et global. Celui-ci ne fait que mémoriser les données qui lui arrivent des tâches et envoyer les données demandées par celles-ci avec parfois une dépendance entre l'arrivée d'une donnée et l'envoi de celle-ci vers un autre destinataire. Cette fonctionalité est facilement réalisable au moyen des divers générateurs d'adresses disponibles et de la mémoire à accès partagés. Cette dernière nécessite un gestionnaire de priorité pour gérer l'attribution des accès aux différents canaux qui sont connectés. L'utilisateur peut aussi définir un algorithme de priorité des requêtes qui soit adapté à l'application visée.

4.3. Exemple d'application

L'example suivant illustre la méthode heuristique que nous venons de décrire. L'algorithme est écrit en langage C. Il réalise la convolution d'une image de 256x256 pixels par une matrice de 3x3 coefficients.

```
Int NewImage[256*256] ;
Int Image[256*256] ;
Int Coef[3*3] ;
For (line=0 ; line¡256 ; line++) {
    For (col=0 ; col¡256 ; col++) {
        Sum=0 ;
        For (x=0 ; x¡3 ; x++) {
            For (y=0 ; y¡3 ; y++) {
                Sum+=Image[256*(line+y)+col+x] * Coef[3*y+x] ;
            }
        }
        NewImage[256*line+col]=Sum/9 ;
    }
}
```

Nous voyons que l'algorithme est constitué de deux boucles imbriquées de 256 itérations chacune balayant toute l'image d'abord horizontalement (variable col) et ensuite verticalement (variable line). A l'intérieur de ces boucles, deux autres boucles imbriquées de 3 itérations chacune effectuent une somme pondérée de neufs pixels qui forment le carré de 3x3 dont le premier pixel est [line,col] et le dernier [line+2,col+2]. Chaque somme pondérée implique donc 9 lectures dans le tableau "Image" et une écriture dans le tableau "NewImage".

(1) Le partitionnement de l'algorithme en tâches : Cette partie paraît triviale puisqu'il n'y a qu'une seule tâche. Néanmoins, un utilisateur averti peut profiter du parallélisme présent dans cet algorithme pour réaliser plusieurs tâches identiques travaillant sur des parties différentes de l'image. Le nombre de tâches dépendra du nombre de ressources disponibles. On peut par exemple imaginer quatre tâches, chacune traitant un quart de l'image.

(2) Le partitionnement de la mémoire (Locale - Globale) : Sur notre architecture de test, la taille des mémoires internes varie entre 256 et 2048 mots de donnée. Les deux tableaux de données utilisés par notre algorithme pour contenir l'image source et l'image traitée doivent donc nécessairement résider en mémoire globale. Par contre, le tableau de coefficients peut très bien résider en mémoire locale. Ces coefficients étant constants, on peut même envisager de les implémenter sous forme d'un générateur de données.

(3) Le partitionnement de l'algorithme (Local - Global) :

Voici les deux programmes obtenus après application de la méthode proposée :

(a) Programme local

```
For (x=0 ; x¡3 ; x++) {
    For (y=0 ; y¡3 ; y++) {
        TEMP=Get_Global(0) ;
        Sum+=TEMP * Coef[3*y+x] ;
    }
}
Send_Global(Sum/9,0) ;
```

(b) Programme global

```
For (line=0 ; line¡256 ; line++) {
    For (col=0 ; col¡256 ;col++) {
        For (x=0 ; x¡3 ; x++) {
            For (y=0 ; y¡3 ; y++) {
                Send_Local(Image[256*(line+y)+col+x],0)
            }
        }
        Get_Local(NewImage[256*line+col],0) ;
    }
}
```

(4) L'étude du temps de vie des variables globales : La figure 58 permet de visualiser les accès qui sont faits à la mémoire globale. La première somme pondérée (point A) se fait sur 9 variables globales qui n'ont pas encore été utilisées auparavant. Les 255 sommes pondérées suivantes (point B) impliquent chacune 6 variables qui avaient été utilisées à la somme précédente et 3 nouvelles variables. A partir de la ligne suivante (point

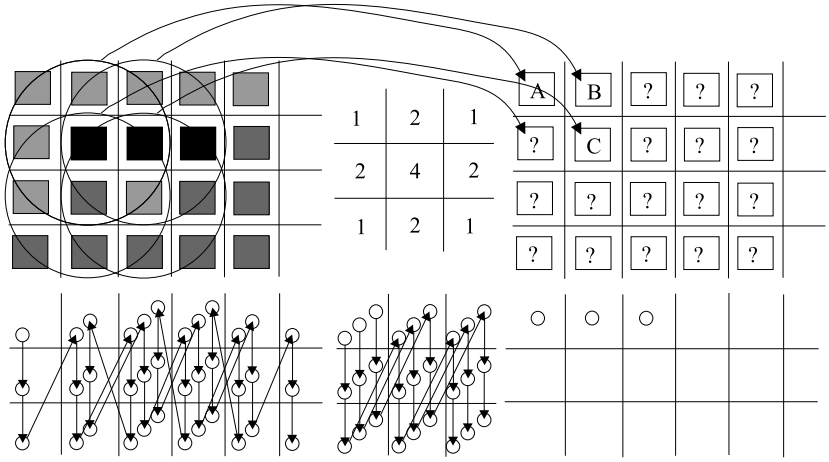


FIG. 58. Schéma des accès à la mémoire globale

C), les sommes pondérées impliquent 6 variables utilisées lors des 2 lignes précédentes, 2 variables utilisées dans les 2 sommes précédentes et finalement une seule nouvelle variable doit être lue en mémoire globale.

Nous voyons donc qu'une variable globale est vivante durant 2 lignes et 2 points, ce qui correspond à 514 données. Néanmoins, on peut imaginer de réduire le temps de vie des variables et de considérer qu'une variable a un temps de vie de 2x6 points, qui correspond aux trois carrés successifs utilisant un même point. Dans ce cas, chaque somme pondérée demande 3 nouveaux points plutôt qu'un à partir de la deuxième ligne.

(5) L'optimisation des transferts : Cette étape est sans doute la plus critique car il s'agit de décider de la stratégie de mémorisation en cache qu'il faut adopter. Comme l'étude du temps de vie des variables l'a montré, il existe deux possibilités : Utiliser une mémoire cache de minimum 514 mots de donnée ou une mémoire cache de minimum 6 mots de donnée. Dans cet exemple, nous retenons la deuxième solution car la première conduirait à l'utilisation de trop de ressources mémoires locales par rapport aux ressources de traitement et l'on se retrouverait dans une situation ou les ressources logiques du FPGA sont peu utilisées alors qu'il n'y a plus de ressources mémoires disponibles. L'inconvénient de cette solution est que le nombre de transferts avec la mémoire globale est trois fois plus élevé. Il reste

quand même trois fois moindre que sans aucune cache, ce qui signifie que l'on peut efficacement faire tourner trois tâches en parallèle, chacune sur un tiers d'image.

Au niveau local, rien ne change. La tâche continue de recevoir ses données sans tenir compte du fait qu'elles viennent maintenant d'une cache plutôt que directement de la mémoire globale. Il faut seulement réécrire le programme global et rajouter une cache et ses générateurs d'adresses entre la tâche et la mémoire globale.

Programme global avec cache

```
For (line=0 ; line¡256 ; line++) {
    For (col=0 ; col¡256 ;col++) {
        For (y=0 ; y¡3 ; y++) {
            Send_Local(Image[256*(line+y)+col+x],0)
        }
        Get_Local(NewImage[256*line+col],0) ;
    }
}
```

(6) L'implémentation des tâches : La tâche à implémenter est assez simple. Il suffit de récupérer l'architecture qui avait été proposée comme exemple à la section décrivant la bibliothèque de DDR. On choisira des dispatchers et des selecters qui réalisent la somme de 9 produits, transfèrent le résultat et recommencent ensuite.

(7) L'implémentation de la gestion de la mémoire globale : Ici aussi l'architecture est assez simple : Une mémoire globale, une ressource de mémoire partagée si l'on désire faire travailler plusieurs tâches en parallèle et quelques générateurs arithmétiques à trois dimensions pour générer les séquences d'adresses.

5. L'environnement de développement

5.1. Introduction

Bien que parfaitement définis de façon mathématique par les quatre ensembles décrits à la section les concernant (point 3.1), les réseaux de Pétri utilisent intensément l'aspect graphique pour visualiser la fonctionnalité qu'ils représentent. Il en est de même pour les DDR. Une architecture utilisant les DDR découle directement de la visualisation qu'en a le concepteur. Cela fait partie de l'aspect intuitif associé aux DDR.

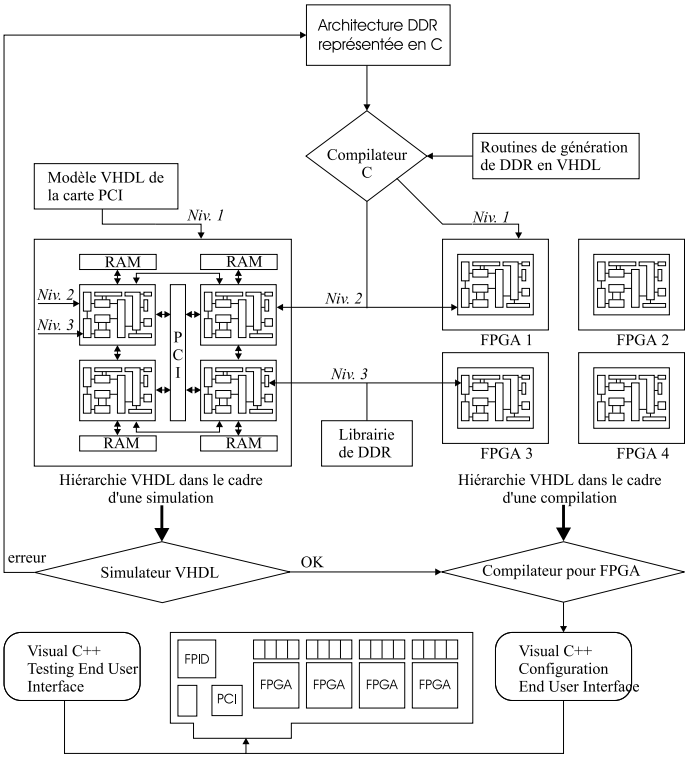


FIG. 59. Les étapes de la réalisation d'une architecture DDR

Néanmoins, il n'existe pas de langage graphique universellement reconnu en matière de synthèse de circuits logiques. Il était donc nécessaire de formaliser le concept de DDR dans un tel langage. Nous avons choisi le VHDL pour des raisons de compatibilité avec la plupart des systèmes de synthèse et de simulation de circuits logiques et également par expérience personnelle. Dans cette section, nous décrivons le format VHDL utilisé pour décrire une architecture DDR et les différents logiciels qui ont été réalisés pour générer, simuler et tester une architecture complète.

La figure 59 propose une vision globale des diverses étapes de la réalisation d'une architecture DDR depuis sa description jusqu'au test final. La description en langage C, la simulation et le test seront explicités dans les sections suivantes. La partie "Compilateur pour FPGA" est réalisée de manière automatique par le logiciel de synthèse pour FPGA. Néanmoins, nous prenons le temps de décrire brièvement le travail réalisé par celui-ci. Nous avons dans l'ordre :

(1) Description de l'algorithme dans un langage de haut niveau : C'est le code source.

(2) Décomposition en fonctions logiques : Le code source est transformé pour ne plus faire apparaître que des fonctions logiques simples telles que logique combinatoire, registres, et éventuellement des primitives plus complexes telles que mémoires, additionneurs, multiplieurs ...

(3) Adaptation à la technologie : La logique combinatoire et les registres sont implémentés en utilisant les ressources propres aux FPGA (LUT et PLA). Le résultat consiste en un graphe dont chaque noeud est une ressource disponible sur le FPGA et chaque arc représente une connexion entre deux ressources.

(4) Placement dans le FPGA : Dans cette étape, un emplacement physique du FPGA est assigné à chaque noeud du graphe à implémenter.

(5) Routage dans le FPGA : Finalement, les connexions programmables sont établies entre toutes les ressources qui doivent être interconnectées conformément aux arcs du graphe.

5.2. La saisie de l'architecture DDR

5.2.1. *Les trois niveaux de hiérarchie*

L'architecture dispose de plusieurs niveaux hiérarchique de VHDL. Selon l'utilisation qui est faite de ce code (simulation ou synthèse), le premier niveaux représente des réalités différentes tandis que les niveaux deux et trois sont communs.

(1) Dans le cadre d'une simulation, le premier niveau représente l'entièreté de la carte avec ses mémoires, ses connexions programmables (programmées) et ses accès au bus PCI. Il s'agit d'une description fonctionnelle non synthétisable. Dans le cadre d'une synthèse, ce niveau représente chacun des quatre FPGA destinés à être compilés. C'est à ce niveau que l'association des broches est réalisée, de même que la sélection de l'horloge principale et la génération d'un signal de reset interne.

(2) Le deuxième niveau consiste en l'interconnexion des diverses DDR entre elles, avec le bus PCI, avec les mémoires externes et avec les canaux d'entrée-sortie des FPGA pour réaliser les connexions inter-FPGA.

(3) Le dernier niveau contient le code VHDL propre à chaque DDR. C'est à ce niveau que la bibliothèque de DDR a été réalisée.

C'est ici aussi que l'on trouvera la définition des ressources spécialisées définies par un utilisateur expert en FPGA.

5.2.2. *Le format VHDL des DDR*

Le format VHDL d'une DDR est le suivant :

```
Entity Une_DDR is
    Generic (Param_1, ... Param_N) ;
    Port (
        Reset : in std_logic ; – reset général
        Clk : in std_logic ; – horloge générale
        "Canaux d'entrée"
        "Canaux de sortie"
        "Ports asynchrones d'entrée"
        "Ports asynchrones de sortie"
    )
```

$Param_1$, ... $Param_N$ sont utilisés pour définir un certain nombre de paramètres tels que les largeurs de bus, le nombre d'entrées (resp. sorties) d'un multiplexeur (resp. démultiplexeur) et d'autres encore. Rappelons que les canaux sont constitués de trois signaux (données, enable, strobe) tandis que les ports asynchrones n'en contiennent qu'un. Les canaux d'entrée et de sortie peuvent prendre deux formes différentes :

(1) Normale : Dans ce cas, les canaux sont énumérés dans l'ordre alphabétique à partir de :

Pour les canaux d'entrée :
- Data_In_A : in std_logic_vector (M-1 downto 0) ;
- Data_In_A_Strobe : out std_logic ;
- Data_In_A_Ena : in std_logic ;

Pour les canaux de sortie :
- Data_Out_A : out std_logic_vector (N-1 downto 0) ;
- Data_Out_A_Strobe : in std_logic ;
- Data_Out_A_Ena : out std_logic ;

(2) Multiple de taille P : Dans ce cas, chaque signal est remplacé par un vecteur de taille P. Ce mode est utilisé pour les ressources dont le nombre de canaux d'entrée ou de sortie est un paramètre de synthèse comme c'est le cas dans les (dé)multiplexeurs. Ceci pourrait également concerner d'autres types de ressources telles qu'une opération logique ou arithmétique à P opérandes.

Les ports asynchrones sont également énumérés par ordre alphabétique en commençant par :

– Port_In_A : in std_logic_vector (P-1 downto 0) ;
– Port_Out_A : out std_logic_vector (Q-1 downto 0) ;

5.2.3. *Le logiciel de génération du VHDL*

Le premier logiciel qui a été développé dans le cadre de la réalisation d'une architecture DDR est un générateur de VHDL orienté DDR. Ce générateur est écrit en C. Il est capable de générer l'instantiation de n'importe quelle DDR sur base des quelques paramètres qui la définissent. Ces paramètres sont regroupés dans une structure C. Ainsi, une ligne de code C est capable de représenter une quelconque DDR.

Dans un deuxième temps, les fonctions d'interconnexions ont été ajoutées au logiciel. Il s'agit de commandes C générant les lignes de code VHDL correspondant à l'interconnexion d'un canal vers un autre ou d'un port asynchrone vers un autre. A ce stade, il devenait déjà possible de générer automatiquement le code VHDL de n'importe quelle architecture constituée de DDR interconnectées entre elles formant une nouvelle ressource plus complexe.

Dans un troisième (et dernier) temps, la carte tout entière a été modélisée comme une architecture DDR. Les ports PCI virtuels ont été générés automatiquement et les connexions inter-FPGA ont été modélisées comme des DDR spéciales de transfert de données.

L'utilisateur commence donc par définir le nombre de ports PCI virtuels (d'entrée et de sortie) pour chaque FPGA ainsi que le nombre et la taille des communications inter-FPGA. Ensuite il définit une structure C pour chaque DDR qu'il désire utiliser et l'assigne à un FPGA. Remarquons qu'une fois qu'une structure a été déclarée en C, on peut facilement en faire un fichier à inclure chaque fois que l'on a besoin de cette ressource dans une architecture. Finalement, il utilise les commandes de connexions de canaux et de ports asynchrones pour interconnecter les DDR, les ports PCI virtuels, les mémoires externes et les FPGA entre eux. Une fois le code compilé et exécuté, deux fichiers sont générés pour chaque FPGA : le fichier VHDL de premier niveau pour la synthèse et le fichier VHDL de niveau deux pour la synthèse et/ou la simulation. L'organisation du fichier de niveau deux est la suivante :

– Déclaration de l'entité globale pour être compatible avec le code VHDL de niveau un.
– Déclaration de toutes les DDR utilisées.
– Déclaration de tous les signaux d'interconnexion.
– Instantiation des ports PCI virtuels.
– Instantiation des DDR utilisées.

– Connexions des canaux (3 signaux par canal).
– Connexions des ports asynchrones.
– Connexion éventuelle de la mémoire externe.
– Ajout éventuel de déclarations demandées par un utilisateur averti.

Ce logiciel a avantageusement remplacé l'interconnexion manuelle des DDR qui s'est vite révélée fastidieuse et propice à des erreurs. Un fichier de niveau deux généré automatiquement fait plus de deux mille lignes de code pour une architecture de taille moyenne. Il serait bien-entendu intéressant de disposer d'une interface graphique qui permettrait à l'utilisateur de dessiner directement son architecture plutôt que de la décrire dans un langage intermédiaire. Néanmoins, cette interface graphique représente un investissement en temps qui déborderait largement le cadre de ce travail compte tenu du peu d'intérêt scientifique qu'il offrirait.

5.3. La simulation

Pour la simulation, nous avons écrit un modèle VHDL fonctionnel de la carte que nous avons utilisée qui constitue le premier niveau hiérarchique (voir le point 5.1). En simulation, l'interface PCI est remplacée par une ressource de transfert de données entre les ports PCI virtuels et des fichiers. Ceci nous permet d'ignorer le protocole PCI qui est assez encombrant lorsqu'il s'agit de faire des simulations fonctionnelles. Les données d'entrée sont mémorisées dans un fichier source et les résultats sont envoyés automatiquement vers un autre fichier destination.

Il existe fondamentalement deux types de simulation :

(1) La simulation avant synthèse : Il s'agit de vérifier la fonctionnalité d'un circuit tel qu'il est décrit dans le code source. Cette simulation ne prend pas en compte les délais inhérents à la technologie. En outre, on peut utiliser des descriptions fonctionnelles qui ne sont pas directement synthétisables pour simuler le comportement d'une partie de circuit. Cette simulation donne de bons résultats sur le comportement du circuit final pour autant qu'un certain nombre de règles de bonne pratique soient respectées. L'avantage de cette simulation est d'être simple et donc (relativement) rapide. Le prix à payer est qu'elle ne donne aucune indication sur la fréquence maximale et les délais.

(2) la simulation après synthèse : Celle-ci correspond à la simulation de l'architecture une fois que toutes les étapes de la synthèse (compilation) ont été réalisées. Ceci signifie que chaque

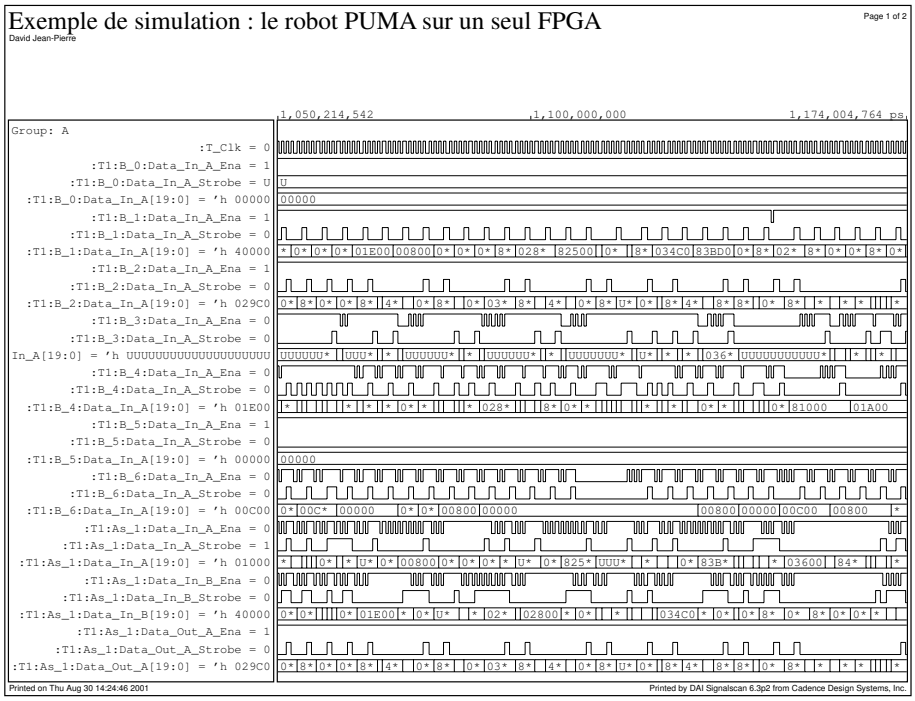


FIG. 60. Exemple de simulation d'une architecture à DDR

ressource du FPGA a été modélisée avec ses délais et ses conditions de stabilité de signaux d'entrée, et que les délais introduits par le routage sont pris en compte. L'avantage est d'avoir une simulation réaliste prenant en compte les délais et permettant de calculer la fréquence maximale. Le prix à payer est un temps de simulation excessivement long dû au grand nombre de signaux à prendre en compte dans les modèles et au surcroît d'équations logiques temporelles qu'ils induisent.

On peut également simuler des parties de circuits synthétisées connectées à d'autres qui ne le sont pas. Il faut encore noter que pour qu'une simulation (avant ou après synthèse) soit totalement fiable, il faut pouvoir prouver que tous les cas de figure ont été envisagés, ce qui n'est généralement pas le cas.

Dans le cadre de notre travail, nous avons cherché à obtenir une simulation avant synthèse qui soit tout-à-fait fidèle à la simulation après synthèse pour éviter des pertes de temps inutiles :

(1) Une simulation après synthèse demande que la synthèse soit réalisée auparavant (!). Une compilation demandant environ

une heure par FPGA, cela représente quatre heures de calcul avant même de pouvoir commencer la simulation de la carte.

(2) La simulation après synthèse étant beaucoup plus lente que la simulation avant synthèse, il faut à nouveau attendre plusieurs heures avant d'avoir des résultats significatifs.

Nous avons donc opté pour une description des DDR en VHDL de très bas niveau et pour l'emploi d'une seule horloge globale. Ces deux conditions assurent une bonne adéquation entre les résultats de simulation avant et après synthèse. De plus, cela augmente la compatibilité avec d'autres technologies. L'expérience le montre, les simulations avant et après synthèse donnent les même résultats pour autant que la fréquence maximale ne soit pas dépassée. Celle-ci est déterminée par une simulation après synthèse une fois que les simulations avant synthèse ont validé la fonctionnalité du circuit.

La figure 60 montre un exemple de l'évolution temporelle des signaux logiques de divers canaux au sein d'une architecture réalisant un large schéma de calcul utilisé dans le cadre de la robotique.

5.4. Le test

Le test est l'ultime étape lors de la mise au point d'un circuit. Ce n'est qu'après l'étape de compilation que l'on peut estimer la fréquence maximale à laquelle notre architecture peut travailler. Si cette fréquence s'avère insuffisante, il est encore possible de changer l'architecture. La limitation en fréquence peut provenir de deux sources :

(1) Des chaines de synchronisation trop longues.

(2) Des ressources combinatoires trop lentes.

Dans le premier cas, on peut améliorer la situation en réduisant la taille de ces chaînes de synchronisation en changeant la topologie de l'architecture ou en introduisant une ressource qui interrompt la dépendance combinatoire entre les entrées et les sorties. Néanmoins cela se fait au prix d'un nombre de cycles d'horloge plus important pour un même calcul. Dans le second cas, on peut définir une latence plus importante dans les ressources concernées et éventuellement utiliser le pipeline.

Pour procéder au test, nous avons développé une librairie de routines de gestion du bus PCI et de configuration de la carte. Chaque FPGA et le Crossbar doivent en effet d'abord être configurés avant de pouvoir être utilisés. Ces routines permettent à une application d'accéder aux ports PCI virtuels en écriture et en lecture.

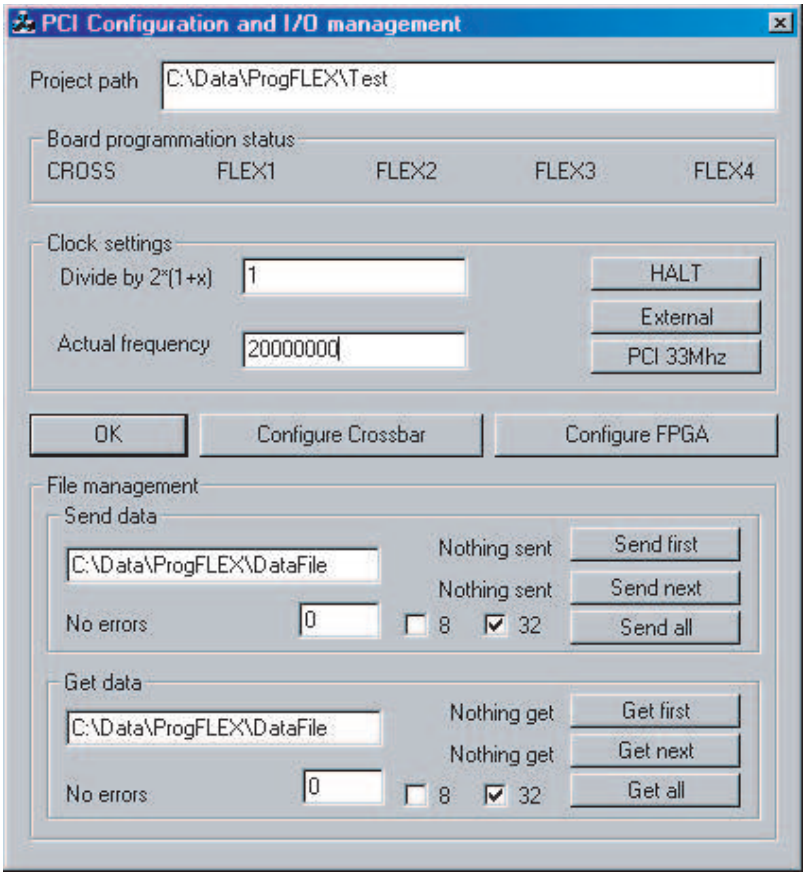


FIG. 61. L'interface graphique utilisée pour le test

De plus, une interface graphique (Figure 61) a été développée en Visual C++ . Elle permet de configurer la carte et d'envoyer des fichiers complets de données (en divers formats) vers un port PCI virtuel ou d'en récupérer. Il faut néanmoins mentionner que l'accès à la carte n'est pas géré par un gestionnaire spécialisé "Windows Driver" et ne permet donc pas d'accès DMA ou burst. Le développement d'un tel logiciel ferait appel à des ressources dont nous ne disposons pas au laboratoire et qui n'apporteraient pas grand chose au contenu scientifique de notre approche.

5.5. Conclusion

Nous avons présenté une architecture nouvelle qui consiste en l'utilisation de ressources synchronisées par les données. Ces ressources, issues

des réseaux de Pétri, permettent de se passer d'un contrôleur central qui était nécessaire dans les architectures de type Von Neumann en confiant ce rôle à des mini-contrôleurs qui monitorisent les transferts de donnée. Il y a autant de contrôleurs que de ressources, ce qui permet d'obtenir un taux de parallélisme maximal. Ces ressources sont destinées à être implémentées sur une architecture de type FPGA (ou multi-FPGA). Une librairie de ressources de base a été développée et présentée. Nous avons également proposé une méthodologie pour les interconnecter et former ainsi une architecture complète pour le traitement des données. Finalement, divers logiciels ont été développés et présentés pour prendre en charge la génération automatique du code compilable et le test de l'architecture finale.

CHAPITRE 4

Applications - Résultats

1. Introduction

L'architecture DDR est l'aboutissement d'un long travail de recherche et de développement dans le domaine de la synthèse sur système multi-FPGA. Il est également le fruit de diverses collaborations, principalement avec le département de robotique. Différentes étapes ont marqué son développement au cours desquelles des résultats de diverses natures ont pu être observés :

(1) Le parallélisme intrinsèque : Il indique le nombre maximum d'opérations pouvant être réalisées en parallèle sur un cycle. Il correspond au nombre de ressources de calcul physiquement implémentées dans le FPGA.

(2) Le taux de calcul : Il ajoute une référence temporelle en donnant le nombre maximum d'opérations réalisables en une seconde. Cette valeur est directement liée à la technologie utilisée et est donc donnée pour les FPGA que nous avons utilisés.

(3) Le taux d'utilisation des opérateurs de traitement de donnée : Il indique l'utilisation effective qui est faite de chaque ressource en considérant l'entièreté du temps de traitement.

(4) Le caractère générique : Il assure que les nouveaux circuits (disposant de davantage de ressources) pourront également bénéficier au maximum de notre architecture.

(5) La facilité de mise en oeuvre : C'est notre dernier atout qui doit garantir une collaboration efficace entre les utilisateurs experts en FPGA et les autres.

Chronologiquement, les résultats que nous avons obtenus découlent de l'évolution de notre architecture au fur et à mesure qu'elle a été confrontée à diverses utilisations :

(1) Un premier algorithme de convolution : C'est la toute première application complète qui tourné sur la carte. Elle a été entièrement réalisée à la main, dans le langage du plus bas niveau (AHDL).

Elle a permis de valider la carte et d'en mesurer les performances maximales puisque tout a été réalisé et optimisé manuellement.

(2) Le prototypage d'un algorithme de watermarking : C'est la première utilisation de type "prototypage rapide". Disposant déjà d'un ensemble élémentaire de DDR permettant de gérer les entrées-sorties, la carte et son architecture DDR a été utilisée pour valider une architecture réalisée par des étudiants dans le cadre de leur travail de fin d'études. Utilisant le protocole de synchronisation propre aux DDR, ils ont pu bénéficier de l'environnement de travail déjà développé pour finaliser leur architecture.

(3) La librairie de DDR : Les quelques ressources DDR développées ont démontré une facilité d'utilisation et des performances (au niveau du parallélisme) qui nous ont poussés à développer une librairie plus complète de DDR et à optimiser certaines ressources.

(4) La génération automatique du code VHDL : L'interconnexion manuelle des DDR entre elles s'est vite révélée fastidieuse et source d'erreurs. le développement du logiciel de génération automatique du code VHDL a démontré une efficacité en terme de rapidité et de fiabilité très appréciable.

(5) Le développement de DDR spécifiques : Nous avons également voulu comparer les performances de notre architecture en comparaison à l'approche ASIC. Pour y parvenir, nous avons réalisé une ressource élémentaire de transformation multi-résolution. L'utilisation de cette ressource couplée à celles présentes dans la librairie a démontré des performances incroyablement élevées par rapport à l'approche ASIC.

(6) Le prototypage d'un algorithme de cryptanalyse : Un deuxième travail de fin d'études a permis de valider, une deuxième fois, le caractère pratique et rapide de l'architecture dans le domaine du prototypage rapide.

(7) L'architecture pour les larges schémas de calcul : Cette dernière architecture est le fruit d'une longue collaboration entre le département de robotique et le nôtre. Il s'agit de la conception d'une cellule de base pouvant être utilisée dans l'implémentation de larges schémas de calculs. Générique, elle peut être répliquée autant de fois que nécessaire ou augmentée selon la place disponible sur le système qui la supporte. Cette cellule démontre la possibilité de réaliser efficacement de longs calculs complexes sur notre architecture alors qu'il est impossible d'utiliser le parallélisme

intrinsèque de ces schémas de calcul sur une architecture traditionnelle.

2. Un exemple simple avec les DDR : La convolution

La convolution est un calcul intensément employé en traitement de signal, notamment dans les domaines du filtrage audio, vidéo et dans les télécommunications. Elle correspond à l'implémentation d'un filtre numérique à réponse impulsionnelle finie (FIR). Nous avons :

x_k : Les échantillons d'entrée.

h_k : La réponse impulsionnelle du filtre (N coefficients).

y_k : Les échantillons de sortie.

$$y_k = \sum_{i=0}^{N-1} x_{k-i} h_i$$

Chaque échantillon demande donc le calcul de N multiplications et (N-1) additions. Dans la littérature, on trouve deux implémentations différentes de ce calcul (Figure 62) :

(1) L'implémentation directe : A l'arrivée d'un nouvel échantillon x_k, on calcule la fonction telle qu'elle est décrite ci-dessus. Cette architecture nécessite la mémorisation des N échantillons d'entrée les plus récents. En cas de changement des coefficients, la transition d'un filtre à l'autre est très brutale.

(2) L'implémentation inverse : A l'arrivée d'un nouvel échantillon x_k, on calcule sa contribution pour chacune des sorties y_k ... y_{k+N-1}. Ceci implique qu'il faut mémoriser N-1 valeurs de sortie d'une étape à l'autre. Or, ces valeurs nécessitent généralement une plus grande précision que les échantillons d'entrée puisqu'elles sont le résultat d'un produit. Néanmoins, la transition d'un filtre à l'autre est beaucoup plus souple et l'architecture se prête bien à un nombre important de coefficients et à un partitionnement sur plusieurs entités.

Les coefficients du filtre peuvent être constants ou varier au cours du temps. Dans le premier cas, des techniques très spécifiques aux FPGA peuvent être mises en oeuvre pour optimiser à la fois la fréquence de travail et le nombre de cellules logiques nécessaires. Ces méthodes sont basées sur le pré-calcul du produit des coefficients fixes par un certain nombre de bits (dépendant de la taille des LUT disponibles sur le FPGA). Les résultats de ces pré-calculs sont stockés dans les LUT

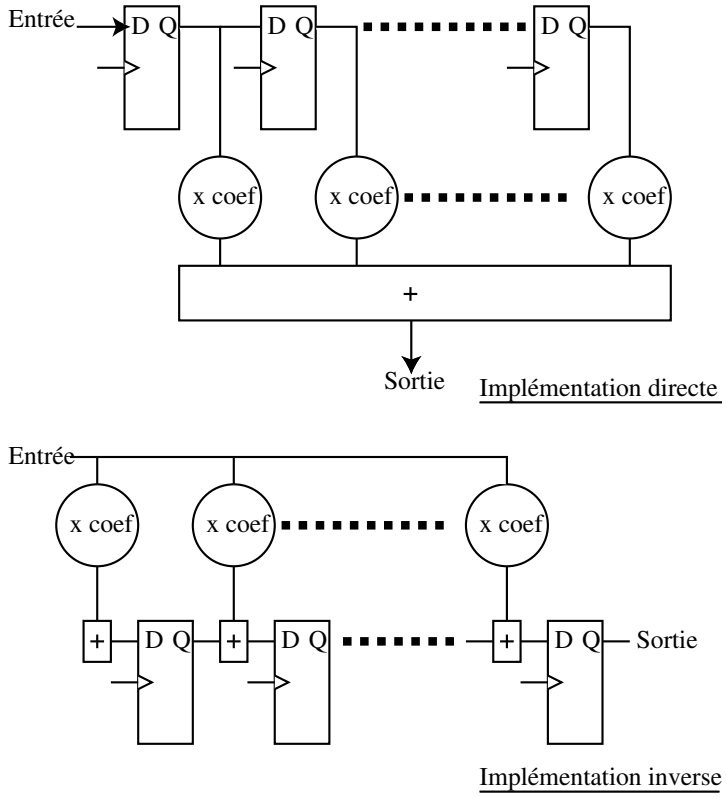


FIG. 62. Les deux implémentations de la convolution

en question. Un registre à décalage sur plusieurs bits et quelques additionneurs suffisent alors à réaliser une architecture extrêmement efficace pour ce type d'application. Le lecteur intéressé peut se référer à [**94**].

Nous nous attacherons particulièrement au second cas, plus général, tout en mentionnant que le premier cas trouve également sa place dans une architecture à DDR mais en tant que ressource dédiée, réalisée par un utilisateur expert.

2.1. Approche empirique

La première architecture que nous avons testée sur la première carte développée est une convolution contenant un nombre élevé de coefficients. Ceux-ci sont mémorisés dans les larges mémoires externes tandis que les échantillons sont stockés dans les mémoires internes du FPGA. De cette façon il est possible de mémoriser un grand nombre de filtres

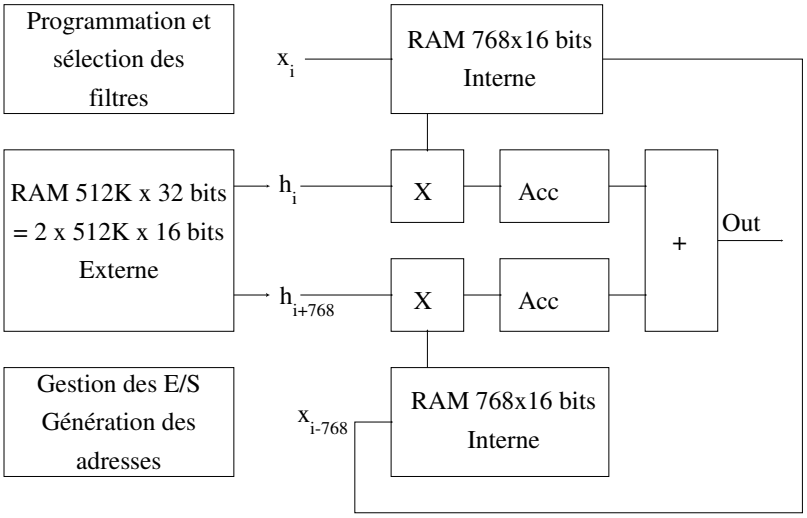


FIG. 63. Implémentation empirique d'une convolution

différents et de passer instantanément d'un filtre à l'autre. Nous avons opté pour une résolution de 16 bits qui était le standard à l'époque pour la numérisation d'un signal audio.

Chaque FPGA dispose de 12 mémoires de 256 mots de 8 bits. Ceci nous autorise à mémoriser jusqu'à 1536 échantillons de 16 bits dans chaque FPGA. D'autre part, la taille de la mémoire externe est de 2Moctets, soit 1M coefficients. Si l'on réserve un espace de 2048 coefficients par filtre (la puissance de 2 directement supérieure ou égale à 1536), il est possible de mémoriser jusqu'à 512 filtres. La carte complète (constituée des quatre FPGA et mémoires associées) permet donc l'implémentation d'une convolution de 6144 coefficients de 16 bits.

La figure 63 décrit l'architecture que nous avons utilisée pour réaliser l'implémentation de cette convolution. Chaque FPGA peut recevoir deux coefficients de 16 bits à chaque cycle puisqu'ils sont mémorisés dans la mémoire externe de 32 bits. Nous sommes donc limités à deux MAC (Multiplication - Addition) par cycle d'horloge et par FPGA. Nous avons opté pour une architecture pipeline des multiplieurs et des accumulateurs qui permet d'atteindre la fréquence des 40 Mhz. Une unité indépendante gère les entrées/sorties avec le bus PCI (acquisition d'un nouvel échantillon et récupération du résultat), l'initialisation du calcul, la génération des adresses et la mémorisation du résultat. Par ailleurs, une fonction de programmation des mémoires externes et de sélection de filtre a également dû être programmée.

Cette architecture dispose donc de 6 ressources de calcul travaillant en parallèle : 2 multiplieurs et 2 additionneurs pour le calcul de la convolution elle-même et 2 additionneurs pour le calcul des adresses. La carte entière offre ainsi un parallélisme de 24 ressources. A 40 Mhz, nous disposons d'un taux de calcul de 80Mmac/s par FPGA, soit 320 Mmac/s pour l'ensemble de la carte. Si l'on prend en compte le calcul des adresses, on arrive à un total de $24 \times 40 = 960$ Mop/s pour la carte complète. Le taux d'utilisation des opérateurs est de 100%. Il faut remarquer que ces calculs sont réalisés sans perte de précision (résultat final sur 43 bits). Par ailleurs, la moitié seulement des ressources logiques de la carte sont utilisées. Il serait donc théoriquement possible de doubler encore le taux de calcul si l'on disposait de deux mémoires externes par FPGA à la place d'une seule ou si on décidait de n'utiliser que des structures de filtres symétriques. Une convolution demande environ 20 us (1536/80Mmac/s), ce qui nous fournit une fréquence d'échantillonnage maximale de 50 KHz. La fréquence standard d'échantillonnage de l'époque dans le domaine de l'audio étant de 48 Khz, notre architecture était très réaliste.

Il faut considérer que ces performances sont le maximum que l'on peut attendre d'une telle architecture. En effet, la fréquence de 40Mhz a été rendue possible grâce à l'implémentation et l'optimisation manuelle de l'architecture. La contre partie à payer est bien entendu le temps de développement et l'expérience de l'utilisateur. L'écriture d'une telle architecture (relativement simple pourtant) dans le langage de plus bas niveau demande un temps considérable dû au fait qu'il faut gérer la synchronisation de tous les événements et en particulier les divers accès aux mémoires et au bus PCI. Les aspects "caractère générique" et "facilité de mise en oeuvre" sont très médiocres. C'est ce qui nous a poussés à développer l'architecture DDR.

2.2. Approche basée sur les DDR

Une fois notre travail de recherche achevé, nous avons voulu implémenter à nouveau la calcul d'une convolution, mais cette fois au moyen des outils que nous avions développés. Cette démarche nous donne un bon moyen de juger de l'efficacité des DDR à divers niveaux : taux de calcul, temps de conception, portabilité... L'architecture que nous avons utilisée est décrite à la figure 64.

Il faut d'ores et déjà remarquer que cette architecture est entièrement réalisée avec des DDR déjà développées dans la librairie et que l'utilisateur ne doit pas nécessairement connaître un langage de programmation

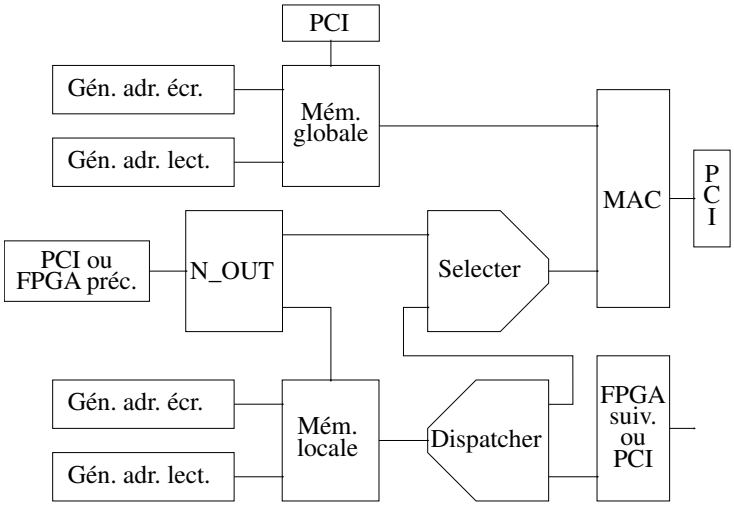


FIG. 64. Implémentation d'une convolution au moyen de l'architecture DDR

de circuits. La seule connaissance intuitive du fonctionnement de chacune des ressources de la librairie est suffisante pour réaliser l'architecture complète, le code VHDL étant d'une part contenu dans la librairie de DDR et d'autre part généré automatiquement par notre générateur de VHDL.

Voyons brièvement le fonctionnement de cette architecture. La mémoire globale contient toujours les coefficients et la mémoire locale les échantillons. La gestion de la programmation des coefficients dans la carte est très simple : nous avons connecté un port PCI virtuel sur le canal d'entrée de données de la mémoire globale et un générateur d'adresses à une dimension (c.a.d de type incrémental) sur le canal d'adresses d'écriture. De cette manière, toutes les données qui arrivent par ce port PCI virtuel sont automatiquement mémorisées comme coefficients à la bonne adresse. L'envoi des coefficients est réalisé lui-aussi par un générateur d'adresses incrémental connecté au canal d'adresses de lecture de la mémoire globale. La génération de la séquence d'échantillons est intrinsèquement plus complexe puisqu'il faut réaliser un tampon circulaire pour l'écriture des échantillons et un double tampon circulaire pour leur envoi vers l'unité de multiplication-accumulation. L'architecture est prévue pour fonctionner sur N FPGA connectés en série. Le premier FPGA prend ses échantillons d'un port PCI virtuel tandis que les suivants les reçoivent du FPGA qui les précède directement. Le dernier FPGA renvoie ses échantillons sur un port PCI virtuel pour

une éventuelle utilisation ultérieure. Lorsqu'un échantillon arrive, il est d'abord dédoublé en passant dans la DDR N_OUT. De là, il est envoyé d'une part à la mémoire locale et d'autre part à l'unité MAC (au travers du selecter) qui peut dès lors commencer son premier calcul. Dès que le selecter a reçu son échantillon, il bascule pour se connecter sur la sortie du dispatcher qui est elle-même connectée à la sortie de la mémoire locale. Celle-ci lui envoie tous les échantillons précédemment mémorisés. Après l'avant dernier échantillon, le dispatcher bascule vers le FPGA suivant (ou le port PCI virtuel s'il s'agit du dernier FPGA) pour transmettre le dernier échantillon (c.a.d l'échantillons sortant) à la ressource de traitement suivante. Le système se met alors en attente d'un nouvel échantillon. Notons que pour réaliser la mémorisation des échantillons dans la mémoire locale nous avons utilisé un générateur incrémental d'adresses tandis que pour envoyer la séquence d'échantillons vers l'unité MAC, nous avons utilisé un générateur incrémental à deux dimensions. Finalement, l'opérateur MAC est constitué d'une ressource multiplication connectée à une ressource accumulateur dont le nombre d'additions à réaliser est fixé au moment de la synthèse.

Le parallélisme et le taux d'utilisation des opérateurs de traitement de données sont évidemment les mêmes que dans l'approche empirique. Néanmoins, la fréquence maximale utilisable est 20Mhz, ce qui diminue les performances par deux. Par contre, le temps de développement est réduit à très peu de chose. l'architecture est totalement générique et facilement exportable sur une autre technologie. Et finalement, aucune connaissance spécifique en matière de technologie FPGA n'est nécessaire.

3. Prototypage rapide

Deux travaux de fin d'études ont utilisé la carte que nous avons développée pour valider expérimentalement leurs résultats de simulation. L'algorithme de watermarking a été réalisé peu après l'implémentation de la convolution longue, en parallèle avec le développement de la bibliothèque de DDR et du générateur automatique de VHDL. Il a donc contribué à leur élaboration et optimisation. L'algorithme de cryptanalyse a été réalisé beaucoup plus tard, utilisant l'architecture disponible qui était bien suffisante pour l'utilisation demandée. Ce type d'application a également donné lieu à une publication [**9**].

3.1. Un algorithme de watermarking

Le watermarking consiste à insérer une petite quantité d'information invisible à l'intérieur d'un paquet d'information plus important sans en altérer la qualité. Parmi les applications on trouve notamment la protection des droits d'auteur dans le domaine de l'image, du son et même du logiciel. Le défi consiste à insérer une information qui soit invisible, robuste aux diverses transformations et récupérable uniquement si l'on dispose de la clé qui a servi à son insertion.

Le cas qui nous concerne consiste en l'insertion de watermark dans une séquence télévisée. La taille des images est de 720x576 pixels. Le principe utilisé consiste à décomposer l'image en 6480 blocs de 8x8 pixels et d'incruster un bit de donnée dans chaque bloc si possible. L'incrustation se fait en modifiant légèrement la luminance des points du bloc en fonction de leur répartition et d'une grille de codage qui représente la clé. De plus amples information peuvent être trouvées dans [**13**].

Le corps de l'algorithme a été écrit en VHDL par les étudiants avec pour consigne que les données leur arrivaient selon le protocole que nous avons défini et que les résultats devaient ressortir selon le même protocole. La bibliothèque de DDR et le générateur automatique de VHDL ont été utilisés pour générer un environnement qui permette l'acheminement des données de diverses manières :

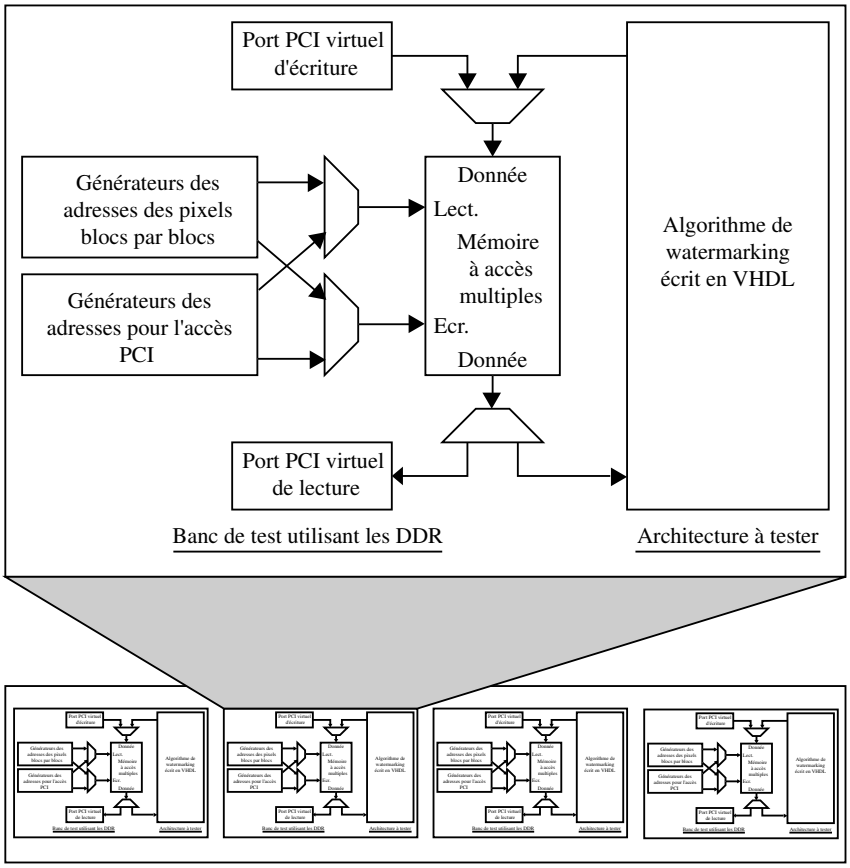
(1) Transfert de l'image depuis le bus PCI jusqu'à la mémoire globale.

(2) Transfert des blocs de l'image depuis la mémoire globale vers l'architecture de watermarking dans un ordre pré-défini et retour des blocs marqués vers la mémoire globale en parallèle.

(3) Récupération de l'image qui a subi le watermarking depuis la mémoire globale vers le PCI.

En outre, certains signaux de synchronisation étaient nécessaires à l'architecture et on pu être fournis par l'intermédiaire d'un port PCI virtuel supplémentaire.

La figure 65 illustre l'architecture DDR utilisée. Remarquons que c'est au cours de ce développement que l'utilisation de la ressource mémoire à accès multiples est apparue nécessaire et a été développée. De même, les générateurs arithmétiques ont été réalisés à la même époque. Nous voyons que la même architecture a été répliquée sur chacun des 4 FPGA. Pour commencer l'algorithme, on enclenche le générateur d'adresses correspondant à l'entrée des données par le port PCI virtuel connecté à la mémoire partagée. Les adresses sont générées incrémentalement par



La carte PCI complète avec les 4 FPGA

FIG. 65. Utilisation des DDR pour le prototypage rapide de l'algorithme du watermarking

un générateur d'adresses à une dimension jusqu'à ce que l'entièreté de l'image soit acquise. Ensuite, un signal de synchronisation est envoyé pour démarrer les générateurs d'adresses correspondant aux transferts de données avec l'algorithme. Celui-ci reçoit les données qui arrivent de la mémoire partagée dans l'ordre prédéfini (bloc par bloc) grâce au générateur d'adresses correspondant. Les résultats sont ensuite stockés dans le même ordre après traitement. Finalement, on enclenche le dernier générateur d'adresses pour récupérer les données par le PCI.

Les résultats montrent que l'architecture développée par les étudiants ne peut pas dépasser les 10Mhz. Notre environnement de prototypage rapide basé sur les DDR ne diminue donc en rien les performances de

l'architecture testée. Chaque FPGA est configuré pour traiter une image entière et peut opérer selon deux modes : incrustation du watermark ou extraction de celui-ci. Le premier prend 148,8 ms/image tandis que le second ne demande que 128 ms/image. Lorsque les 4 FPGAs travaillent en parallèle, on peut donc obtenir une fréquence de traitement d'image légèrement supérieure à 25 images/seconde, qui était le but à atteindre. Précisons toutefois que pour des raisons logicielles de transfert de donnée sur le bus PCI, cette architecture n'a pas été testée en temps réel. Néanmoins, elle a été testée à 10Mhz et les résultats obtenus sont conformes aux simulations.

Il n'est pas judicieux de caractériser les performances en termes de parallélisme, de taux d'utilisation des ressources ou de taux de calcul car notre architecture est seulement utilisée comme environnement pour le test d'une autre architecture, écrite directement en VHDL. Par contre, le critère "caractère générique" est bien satisfait. En effet, notre architecture offre la plus grande latitude quant au nombre de canaux d'entrée et de sortie ainsi que le nombre de mémoires à accès multiple. Pour ce qui est du critère "Facilité de mise en oeuvre", il faut se rappeler que le prototypage rapide s'adresse à des experts en FPGA. Dans ce contexte, l'approche DDR est très facilement mise en oeuvre pour l'acheminement des données et des résultats, pour autant que le design à tester puisse s'adapter au protocole, ce qui n'est pas très contraignant.

3.2. Un algorithme de cryptanalyse

La cryptologie est un domaine d'application dans lequel la technologie FPGA offre une puissance de traitement de loin supérieure à une approche conventionnelle (utilisant un processeur). Un traitement est très répétitif et l'utilisation d'opérations travaillant directement sur des bits de données plutôt que sur des mots entiers en sont les raisons principales. Les opérations à réaliser sont facilement implémentables dans des LUT ou au pire dans des petites mémoires ROM imbriquées.

L'algorithme utilisé dans le cadre de ce travail de fin d'études consiste à mesurer des propriétés statistiques du DES (le standard de l'époque en matière d'algorithme de chiffrement) dans le but de retrouver la clé sur base d'un certain nombre de chiffrés. Cette approche nécessite le calcul d'un très grand nombre de chiffrements. A chaque chiffrement, des compteurs sont incrémentés ou non selon l'état des bits observés. Le détail de l'algorithme et de son implémentation peuvent être trouvés dans [**57**].

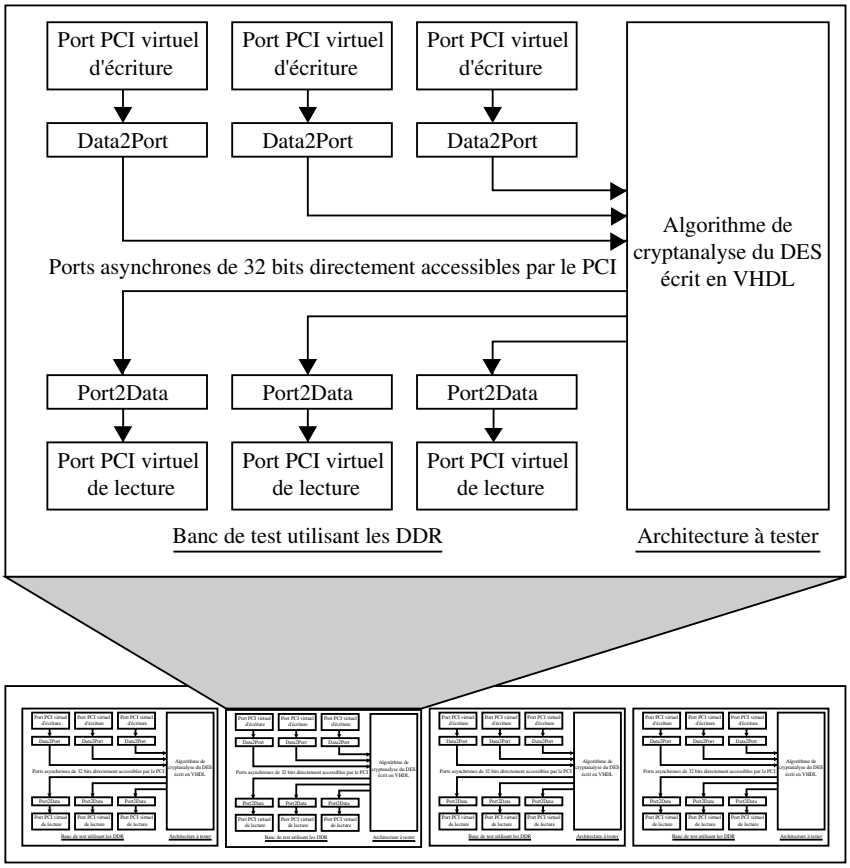


FIG. 66. Utilisation des DDR pour le prototypage rapide de la cryptanalyse du DES

Cette fois encore, l'algorithme complet est écrit directement en VHDL. Néanmoins, plutôt que d'imposer notre protocole de communication pour le transfert de données entre le PCI et l'architecture testée, nous avons préféré employer une approche aussi générique que possible. L'étudiant n'a donc pas eu à se soucier de la manière dont l'architecture DDR fonctionnait pour valider son algorithme.

Notre architecture repose sur deux ressources DDR spécialisées mais très simples (Figure 66). La première copie les données qui lui arrivent depuis son canal d'entrée sur un port de sortie non synchronisé. Elle a été baptisée "Data2Port". La seconde fait l'opération inverse et porte le nom "Port2Data". Ces deux ressources sont connectées à des ports PCI

virtuels. De cette manière, il est possible d'écrire sur n'importe quelle entrée de l'architecture à tester de manière asynchrone et de récupérer les résultats de sortie également de manière asynchrone.

L' architecture développée par l'étudiant se prête très bien à ce genre de banc de test car très peu de données doivent être fournies (quelques paramètres d'initialisation et un signal de démarrage) et quelques valeurs sont mesurées au cours du temps (les valeurs des compteurs). La fréquence maximale utilisable était de 37Mhz. Néanmoins, nous avons utilisé une fréquence de 20Mhz pour ne pas surchauffer les FPGAs. Un chiffrement est réalisé dans chaque FPGA à chaque cycle. Ceci donne un taux de chiffrement de 80M/s. Les 2^{45} (soit environ 35×10^{12}) chiffrés nécessaires ont donc demandé un peu plus de 122 heures de calcul.

Nous voyons que, comme dans l'architecture précédente, les critères "aspect générique" et "facilité de mise en oeuvre" sont pleinement satisfaits. Dans ce cas-ci, il n'est même plus nécessaire que l'architecture à tester se conforme à notre protocole puisque tous les accès se font de manière asynchrone. Néanmoins, cela diminue aussi le taux de transfert maximal réalisable et cette approche n'est certainement pas la meilleure pour des architectures qui font un usage intensif des accès d'entrée-sortie.

4. Les DDR utilisateur : La transformée multi-résolution

Nous envisageons ici le cas ou un utilisateur expert est amené à réaliser une ressource optimisée pour une application très précise. Néanmoins, il désire utiliser l'environnement DDR pour le reste de son architecture. Nous avons déjà annoncé que n'importe quelle fonction combinatoire pouvait être transformée automatiquement en une DDR de traitement de données grâce à une ressource de synchronisation développée à cette fin. Néanmoins, il y a des situations ou il est intéressant de développer une ressource faisant appel à de la logique séquentielle. Le cas des DDR d'entrée-sortie est un exemple ou il est nécessaire de gérer un protocole de communication qui est séquentiel dans la plupart des cas. Nous avons vu également qu'une convolution réalisée avec des coefficients fixes pouvait être réalisée de manière plus optimale en développant une DDR spécifique. De nombreuses applications utilisant des DDR définies par l'utilisateur peuvent ainsi être envisagées.

Dans cette section, nous avons choisi un algorithme de traitement de signal qui réalise la transformée multi-résolution par une structure de filtre en treillis. La choix de cette architecture a été motivé par la réalisation antérieure au sein de notre laboratoire d'un circuit spécialisé (ASIC)

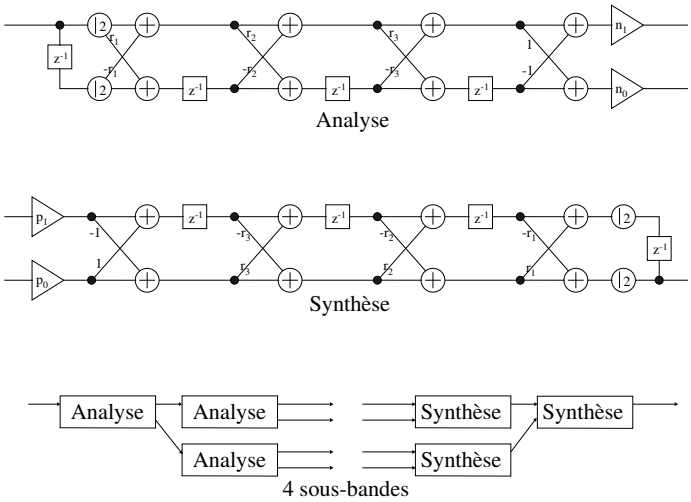


FIG. 67. Structure de filtre en treillis pour la transformée multi-résolution

pour ce genre de traitement de donnée, ce qui offre un point de comparaison intéressant [**8**].

La transformée multi-résolution par filtre en treillis est une architecture qui permet de décomposer un signal en deux bandes de fréquences par passage dans des filtres FIR complémentaires (un passe bas et un passe haut). Chacun des canaux peut alors être sous-échantillonné par un facteur deux. Le cas échéant, la décomposition est effectuée à nouveau sur l'ensemble des canaux de sortie ou sur une partie seulement, pour affiner la résolution de certaines bandes. En imagerie, ce traitement doit être utilisé dans les deux dimensions, résultant en un ensemble de sous images de résolutions diverses. L'image correspondant aux fréquences basses dans les deux dimensions représente l'original en miniature. De plus amples informations peuvent être obtenues dans [**58**].

Le type de filtre utilisé est présenté à la figure 67. Nous voyons que les ressources d'analyse et de synthèse nécessitent chacune 8 additions et 8 multiplications par des coefficients constants. La précision choisie est de 16 bits pour les données et 5 bits pour les coefficients. L'utilisation intelligente des LUT pour le calcul des produits permet de réaliser une ressource d'analyse avec 476 cellules logiques et une ressource de synthèse avec 491 cellules logiques. Il est donc possible de mettre jusque 8 ressources sur un FPGA si on se limite à un remplissage à 80%. Bien que le parallélisme ne soit pas un fruit exclusif de l'architecture DDR mais

aussi de la manière dont la ressource a été développée, il est intéressant de calculer le taux de parallélisme :

$$(8\,additions + 8\,multiplications) \times 8\,ressources = 128$$

Le taux d'utilisation de chaque opérateur est de 50% puisque le calcul complet des deux sorties nécessite l'arrivée de deux données. A une fréquence de 20Mhz, nous obtenons un taux de calcul de :

$$128 \times 50\% \times 20Mhz = 1.25GOp/s$$

pour un seul FPGA. Ces performances sont très largement au dessus de celles de l'ASIC développé précédemment au laboratoire puisqu'il contient 3 multiplieurs et qu'il ne peut pas dépasser la fréquence de 55Mhz. Enfin, remarquons qu'avec un FPGA plus récent tel que l'APEX20K1500, il serait encore possible de multiplier les performances par 10.

Le critère "aspect générique" est bon mais il pourrait être amélioré. En effet, cette DDR peut être générée automatiquement pour une précision quelconque des données d'entrée comme celles de sortie. De plus, elle peut être combinée à ses semblables pour affiner la largeur des bandes de fréquences. En revanche, elle ne dispose que de coefficients constants optimisés pour cette application et cette technologie. Néanmoins, le changement des coefficients peut se faire manuellement de façon relativement simple.

Pour ce qui est de la facilité de mise en oeuvre, le système recouvre tous les avantages d'une architecture DDR si l'on se contente des coefficients programmés. Sinon, le changement manuel des coefficients ne peut être réalisé que par un utilisateur expert.

5. Calcul parallèle : application en robotique

5.1. Introduction

Une fois la librairie de DDR et le générateur de VHDL achevés, il nous fallait valider notre approche sur un problème de taille réelle. Outre la facilité de mise en oeuvre, l'avantage principal de notre architecture est le parallélisme important qu'elle offre. Il fallait une application qui puisse en tirer profit. Les algorithmes classiques tels que FFT, DCT, FIR, IIR et bien d'autres ont déjà fait l'objet de recherches et il existe aujourd'hui des implémentations optimisées pour FPGA proposées sour la forme d'IP (Intellectual Property) par des firmes spécialisées. Il suffit alors d'imbriquer un tel code dans une ressource de synchronisation

pour obtenir une DDR spécialisée dont les performances seront celles de l'IP. Néanmoins, il existe un domaine ou il est difficile d'utiliser le parallélisme intrinsèque, quelle que soit l'architecture considérée. C'est celui des larges schémas de calculs utilisés par exemple en robotique.

Notre département de mécanique est spécialisé dans la génération automatique de schémas de calculs pour la simulation ou la commande des systèmes mécaniques. Le logiciel ROBOTRAN [**33**], développé par le département lui-même, est capable de générer le schéma de calcul demandé, de réorganiser les opérations pour effectuer un certain nombre de simplifications et finalement de proposer un ordonnancement des opérations en fonction des ressources de calcul disponibles [**71**]. Le nombre d'opérations à réaliser croît très vite avec la complexité du système considéré. Le calcul de la dynamique inverse (les couples en fonction de la trajectoire à suivre) d'un robot à quatre degrés de liberté demande déjà 222 opérations en virgule flottante. Un robot à six degrés de liberté en demande 337. Un système complexe tel que la simulation d'une voiture peut demander jusqu'à 20000 opérations. Néanmoins, de nombreuses opérations peuvent être réalisées en parallèle. Ainsi, le cas de la voiture peut théoriquement utiliser une centaine de ressources travaillant en parallèle avec une utilisation presque ininterrompue de chaque ressource.

Cette section décrit une architecture générique construite au moyen de la librairie de DDR. Cette architecture est basée sur une cellule de base qui peut être répliquée autant de fois que nécessaire (selon la place disponible dans le FPGA qui doit l'accueillir). L'ensemble de l'architecture est destiné à réaliser des schémas de calcul de grande taille en utilisant au maximum les possibilités de parallélisme de ceux-ci. La cellule de base a été pensée pour utiliser au mieux les ressources disponibles sur un FPGA FLEX10K100. Néanmoins, sa structure est universelle et peut être augmentée très facilement pour des systèmes plus performants, laissant ainsi un grande ouverture aux nouveaux FPGA et même aux circuits futurs.

5.2. Le module BDC

Le module BDC (Basic Data-driven Cell) est principalement constitué de 2 ressources de traitement de données, trois ressources de mémorisation, un certain nombre de ressources de transfert de données, un port PCI virtuel d'entrée et un de sortie (Figure 68). Elle s'inspire de l'architecture générique de traitement de données décrite à la section concernant la librairie de DDR (chapitre 3, point 3.3).

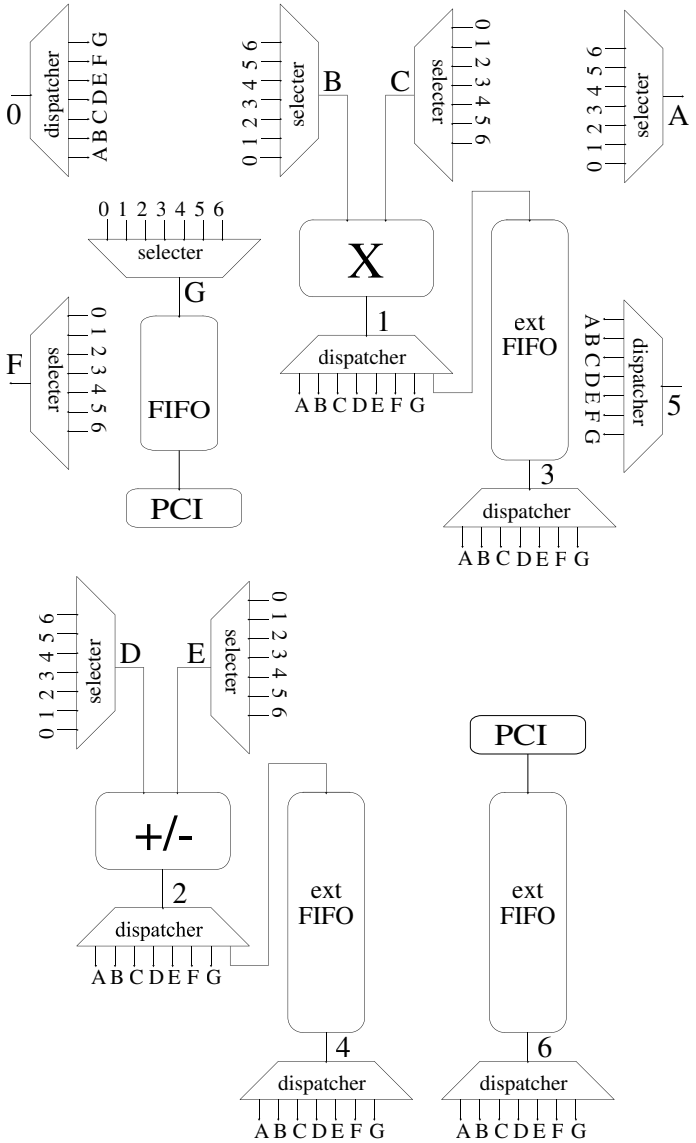


FIG. 68. Le module BDC pour le calcul parallèle de larges schémas de calcul

La cellule doit être vue comme un ensemble de canaux de communications interconnectés au moyen de dispatchers (pour les canaux de sorties) et de selecters (pour les canaux d'entrée). Nous avons 7 canaux d'entrées et 7 canaux de sortie répertoriés comme suit :

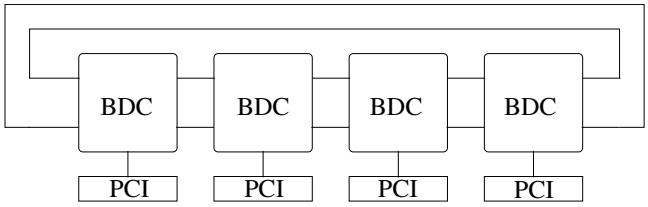


FIG. 69. Les 4 BDC sur notre architecture de test

Les canaux d'entrée

Entrée	Codage	Description
A	000	Extension droite
B	001	Opérande gauche du multiplieur
C	010	Opérande droite du multiplieur
D	011	Opérande gauche de l'additionneur
E	100	Opérande droite de l'additionneur
F	101	Extension gauche
G	110	FIFO de sortie vers le PCI
(H)	111	FIFO étendue locale

Les canaux de sortie

Sortie	Codage	Description
0	000	Extension gauche
1	001	Résultat présent du multiplieur
2	010	Résultat présent l'additionneur
3	011	Résultats mémorisés du multiplieur
4	100	Résultats mémorisés de l'additionneur
5	101	Extension droite
6	110	FIFO étendue venant du PCI

Le choix d'avoir placé un multiplieur et un additionneur est arbitraire. Dans une architecture utilisant plusieurs cellules, l'utilisateur peut attribuer les opérations comme il le désire pour obtenir finalement une architecture optimale par rapport au nombre d'opérations de chaque type nécessaires. De plus, il est possible d'utiliser des opérations plus diversifiées (division, racine carrée, fonctions trigonométriques ...)

Le système multi-FPGA à notre disposition offre la possibilité d'interconnecter 4 BDC (Figure 69). Les canaux d'extension permettent une communication bi-directionnelle avec le voisin de gauche et le voisin de droite. Le cas échéant, il est même possible à une donnée de traverser un FPGA pour aller au suivant. Le point critique du système consiste évidemment en la programmation des divers selecters, dispatchers et

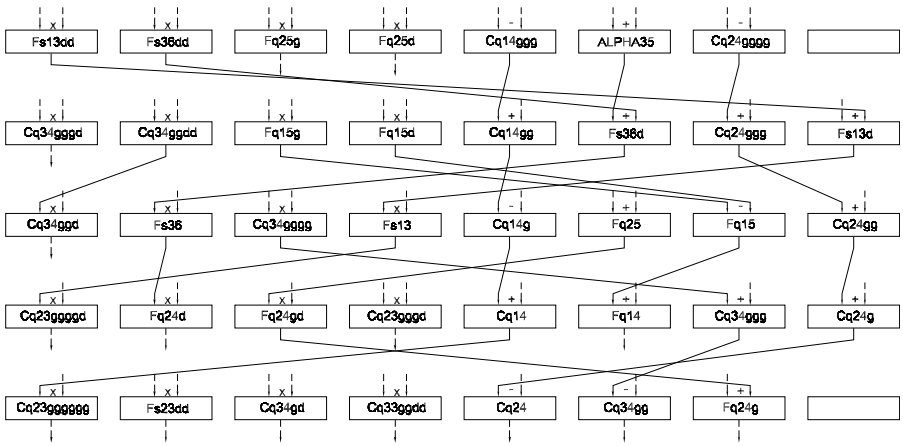


FIG. 70. Exemple d'un ordonnancement réalisé par ROBOTRAN

FIFO étendues pour acheminer les données selon le schéma de calcul qu'il faut réaliser. Pour y parvenir, nous avons fait appel aux possibilités du logiciel ROBOTRAN.

Partant d'une description mécanique du système, ROBOTRAN génère d'emblée un schéma de calcul optimisé en ce qui concerne le nombre total d'opérations à réaliser. Le chemin critique en est alors déduit. Celui-ci fixe le nombre minimal d'étapes nécessaires à la réalisation du calcul. Le nombre effectif d'étapes dépendra de la quantité d'opérateurs disponibles à chaque étape. L'utilisateur peut demander au logiciel de calculer un ordonnancement pour un nombre donné d'opérateurs de chaque type. Il est alors possible de déterminer de manière heuristique le nombre d'opérateurs de chaque type qu'il est intéressant d'implémenter pour obtenir un bon compromis entre la taille de l'architecture et le pourcentage d'utilisation temporelle de chaque ressource. La figure 70 montre un exemple d'ordonnancement réalisé par ROBOTRAN pour 4 additionneurs et 4 multiplieurs. Le passage des données d'une étape à la suivante est représenté par des flèches pleines. Ces transferts se font sans perdre de cycle horloge s'ils sont effectués au sein d'un BDC. Les transferts vers une étape ultérieure sont représentés par des flèches en pointillé. Ces tranferts nécessitent le stockage temporaire dans les FIFO étendues.

Sur base de l'ordonnancement effectué, il faut encore répartir les opérations sur les diverses BDC disponibles en favorisant au maximum les transferts de données au niveau local pour éviter de ralentir l'exécution par un excès de transferts de données. Cette étape n'est malheureusement

pas encore implémentée sur le logiciel et les tests que nous avons réalisés ont été faits sur base d'une répartition arbitraire. Ceci conduira, comme nous le verrons dans les résultats, à une chute des performances quand le nombre de BDC augmente trop. Finalement, les programmes des switch (utilisés dans les dispatchers et selecters) sont générés automatiquement par ROBOTRAN ainsi que les générateurs d'adresses pour les FIFO étendues.

5.3. Résultats

5.3.1. *Caractéristiques des schémas de calcul*

Deux schémas de calcul de trajectoire de robots (en dynamique inverse) ont étés implémentés et testés complètement sur notre architecture :

(1) L'APRA
- 4 degrés de liberté.
- 33 variables d'entrée et 4 variables de sortie.
- 132 multiplications, 57 additions and 33 soustractions.
- Des chemins critiques de 133 (1 BDC), 67 (2 BDC), 46 (3 BDC) et 35 (4 BDCs).

(2) Le PUMA
- 6 degrés de liberté.
- 46 variables d'entrée et 6 variables de sortie.
- 198 multiplications, 87 additions and 52 soustractions.
- Des chemins critiques de 199 (1 BDC), 100 (2 BDC), 68 (3 BDC) et 52 (4 BDCs).

Pour chaque robot, nous avons considéré une implémentation sur 1,2,3 ou 4 BDC avec des opérateurs utilisant 1,2,3 ou 4 cycles pour réaliser leur opération (le délai). Nous comparons successivement :

(1) Le nombre de cycles d'horloge : Il représente le nombre total de périodes d'horloge pour réaliser le schéma de calcul. Il croît avec le délai et diminue avec le nombre de BDC jusqu'à une certaine limite.

(2) Le nombre de cycles opérateurs : Il correspond au nombre de cycles d'horloge total divisé par le délai. On ramène donc la référence temporelle au temps de calcul d'une opération plutôt qu'au nombre de cycles d'horloges. Cette vision des choses est intéressante car le temps de calcul minimum nécessaire pour réaliser une opération sur un FPGA ne dépend pas de la fréquence

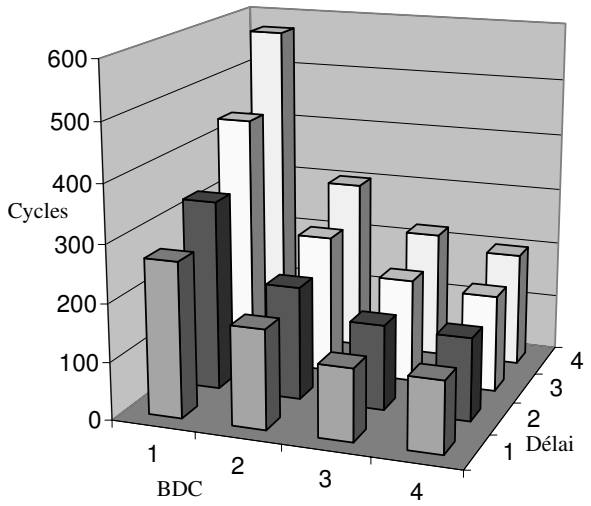


FIG. 71. Cycles d'horloge pour le robot APRA

de l'horloge (l'implémentation combinatoire étant la plus rapide même si elle ne permet aucun pipeline). Plus l'horloge sera rapide, plus il faudra de cycles pour réaliser l'opération. Par ailleurs, les transferts ne demandent que très peu de temps pour être réalisés (un seul cycle d'horloge). Nous voyons que le nombre de cycles opérateurs décroît avec le délai et avec le nombre de BDC, jusqu'à une certaine limite.

(3) Le coût des transferts : Le logiciel ROBOTRAN calcule le nombre de cycles opérateurs minimum pour réaliser le schéma de calcul. C'est ce que nous avons appelé le "chemin critique". Néanmoins, les résultats montrent que le calcul prend plus de temps que nécessaire car il prend en compte le temps des transferts. Le coût des transferts est la différence entre le temps réel et le temps théorique minimum (selon ROBOTRAN) pour réaliser le schéma de calcul. Il est exprimé en pourcentage de ce temps minimum. Nous observons qu'il augmente avec le nombre de BDC et qu'il diminue avec le délai.

5.3.2. *Le robot APRA*

(1) Nombre de cycles d'horloge

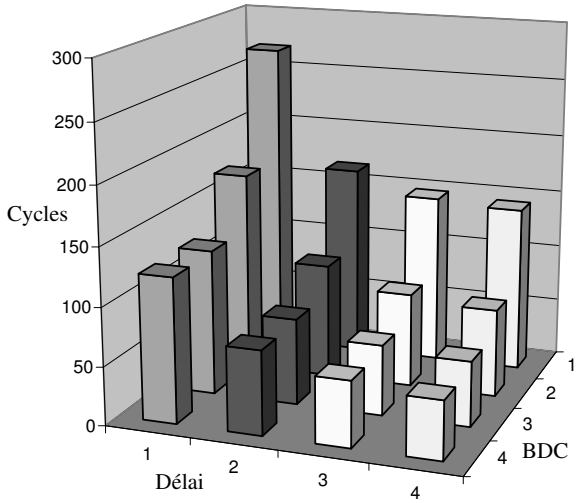


FIG. 72. Cycles opérateurs pour le robot APRA

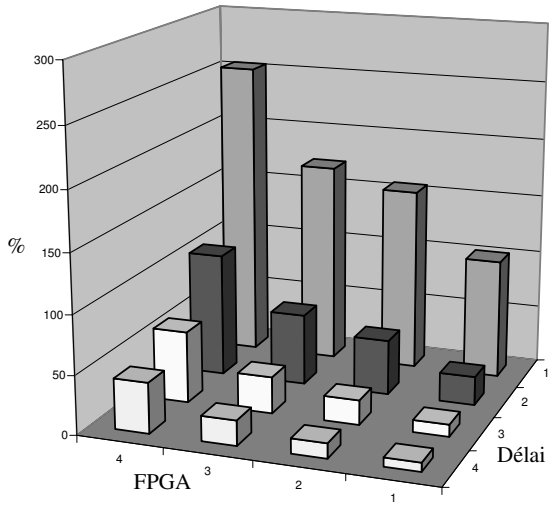


FIG. 73. Coût des transferts pour le robot APRA

Cycles	1 BDC	2 BDC	3 BDC	4 BDC
1	269	172	125	124
2	331	197	148	144
3	440	243	181	169
4	572	302	223	200

(2) Nombre de cycles opérateurs

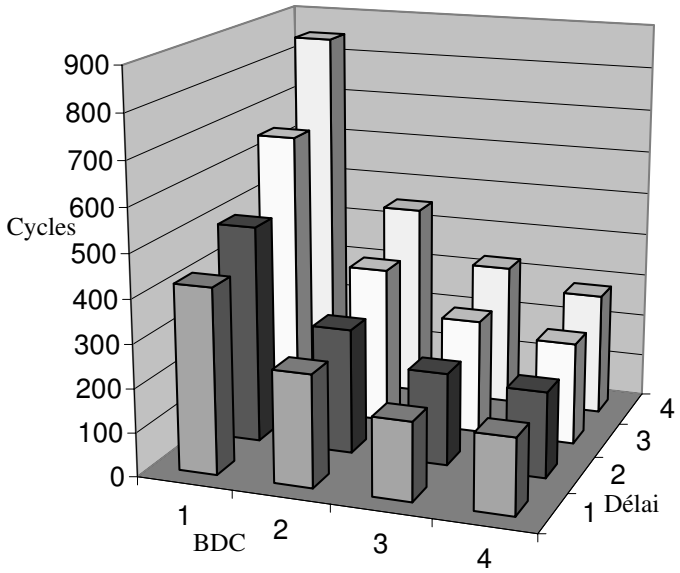


FIG. 74. Cycles d'horloge pour le robot PUMA

Cycles	1 BDC	2 BDC	3 BDC	4 BDC
1	269	172	125	124
2	166	99	74	72
3	147	81	60	56
4	143	76	56	50

(3) Coût des transferts (en %)

Cycles	1 BDC	2 BDC	3 BDC	4 BDC
1	102	157	172	254
2	24	47	61	106
3	10	21	31	61
4	8	13	21	43

5.3.3. *Le robot PUMA*

(1) Nombre de cycles d'horloge

Cycles	1 BDC	2 BDC	3 BDC	4 BDC
1	423	256	179	173
2	499	286	211	196
3	656	362	263	235
4	849	451	329	283

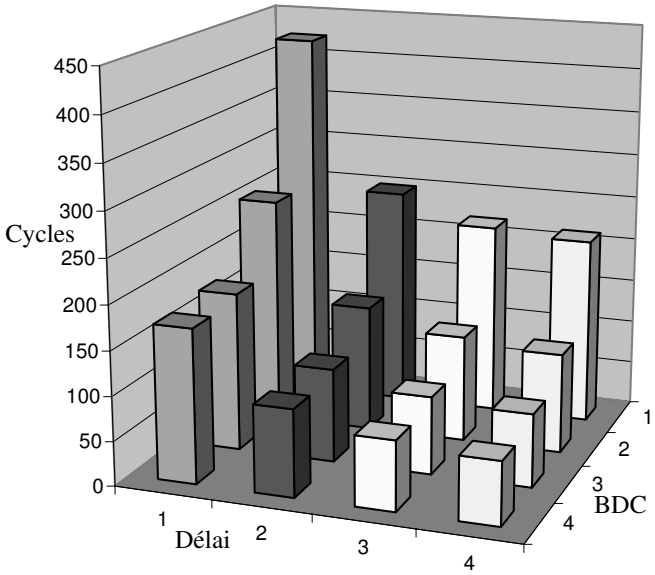


FIG. 75. Cycles opérateurs pour le robot PUMA

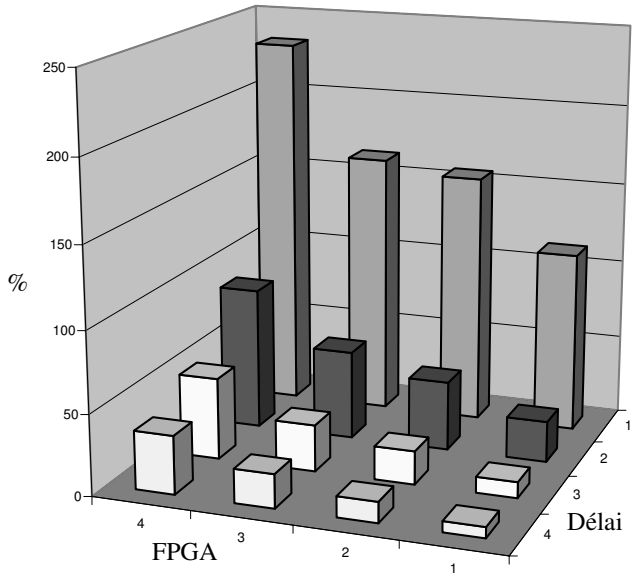


FIG. 76. Coût des transferts pour le robot PUMA

(2) Nombre de cycles opérateurs

Cycles	1 BDC	2 BDC	3 BDC	4 BDC
1	423	256	179	173
2	250	143	106	98
3	219	121	88	78
4	212	113	82	71

(3) Coût des transferts (en %)

Cycles	1 BDC	2 BDC	3 BDC	4 BDC
1	113	156	163	233
2	25	43	55	88
3	10	21	29	51
4	7	13	21	36

5.3.4. *Un dernier schéma de calcul*

Nous avons également désiré tester un schéma de calcul de beaucoup plus grande taille. Le logiciel ROBOTRAN a été utilisé pour générer les équations correspondant à la simulation du quart d'une voiture (2CV). Le schéma de calcul engendré comporte 985 opérations (612 multiplications, 211 additions et 162 soustractions). La quantité d'opérations imposait d'utiliser les 4 BDC car la taille des mémoires disponibles sur le FLEX10K100 ne permet pas de mémoriser plus de 256 données par fifo étendue. Les résultats sont semblables à ceux déjà évoqués ci-dessus :

Délai	1	2	3	4
coût	231	81	38	22

Il semble même que le coût des transferts diminue avec la taille du système. Nous pouvons donc étendre les propriétés de notre architecture testée sur des exemples réduits à de très larges schémas de calcul.

5.4. Conclusion

Les performances se mesurent facilement en comparant les résultats concernant le coût des transferts. Ce paramètre représente l'efficacité de notre approche. Nous voyons que le coût des transferts varie entre 7% et 254 %, ce qui signifie que dans le meilleur des cas, les transferts ajoutent moins d'un dixième du temps de calcul tandis que dans le pire des cas, ceux-ci multiplient le temps de calcul par trois et demi.

Le meilleur des cas correspond à un seul BDC et un délai de quatre cycles. Dans ce cas, un minimum de transferts est nécessaire puisque les données restent au sein du même BDC. Néanmoins, toutes les opérations

qui nécessitent une mémorisation des résultats intermédiaires ajoutent au moins un cycle d'horloge car notre architecture ne permet pas d'envoyer une donnée vers deux endroits simultanément (ce qui pourrait être envisagé cependant). Ce problème est néanmoins compensé par le fait qu'une opération de transfert prend seulement le quart du temps d'une opérations de traitement de données puisqu'elles demandent 4 cycles pour chaque opération.

Le pire des cas correspond au cas de figure inverse : quatre BDC et un seul délai. Outre les raisons mentionnées au point précédent, il faut se souvenir que le logiciel ROBOTRAN n'optimise pas les transferts entre divers BDC. De plus, a partir de quatre BDC, certaines données doivent transiter par leur voisin pour arriver au BDC opposé, ce qui entraine une perte de temps considérable. Cette architecture est donc très performante si le délai est supérieur à deux et que le problème des transferts inter-BDC peut être résolu. Ce dernier point est possible si on dote ROBOTRAN de règles heuristiques pour regrouper les opérations (ou groupes d'opérations) utilisant les mêmes données et séparer celles qui sont indépendantes.

Si l'on considère le taux de calcul, on obtient pour la carte complète : $4\,BDC\times$ 2 opérateurs en parallèle $\times 10\,Mhz$ soit 80 Mflop/s. Ce taux n'est à-priori pas extraordinaire si on le compare à une approche à DSP. Néanmoins, il faut tenir compte du fait que, outre les opérations du schéma de calcul proprement dit, l'architecture prend en compte tous les calculs d'adresses, ce qui doit aussi être pris en charge par un DSP. Les opération étant binaires et sans structure régulière, deux calculs d'adresses sont nécessaires pour la lecture des opérandes et un calcul d'adresse est nécessaire pour l'écriture du résultat. On peut donc multiplier ces performances par 4 si l'on prend en compte les opérations de calcul des adresses, ce qui nous amène à un taux de 320 Mop/s pour la carte.

Les nouveaux circuits développés par Altera (APEX20K1500) contiennent environ dix fois plus de ressources que celui que nous avions utilisé à l'époque et l'on peut espérer travailler à une fréquence opérateur de 20 Mhz. Un calcul rapide montre que ce genre de circuit pourrait contenir environ 36 DDR de traitement de donnée travaillant en parallèle, ce qui porte le taux de calcul à 720Mflop/s (ou 2.88 Gop/s si l'on prend en compte le calcul des adresses). Par ailleurs, si l'on regarde l'état de l'art chez Xilinx, les derniers circuits annoncés contiennent jusqu'à 192 multiplieurs intégrés et promettent des performances encore bien supérieures sur ce genre d'architecture.

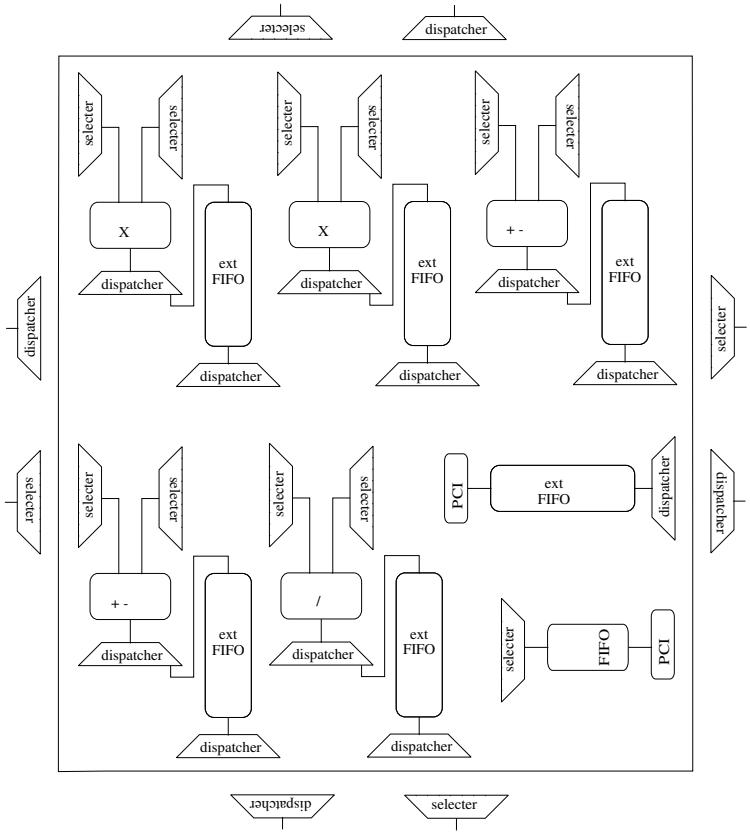


FIG. 77. Evolution d'une BDC étendue à 5 opérateurs

Finalement, notons encore que cette cellule a été développée pour tenir dans un FPGA FLEX10K100. Les circuits disponibles aujourd'hui permettent d'imaginer des cellules beaucoup plus grosses. Notre BDC utilise seulement trois bits de contrôle par switch, ce qui limite le nombre total de canaux d'entrée et de sortie à 8 de chaque sorte. Si l'on passe à 4 bits, on peut réaliser une cellule contenant 5 ressources de calcul et 4 canaux de communication avec des cellules voisines, autorisant ainsi une architecture en treillis dans laquelle chaque cellule peut communiquer de façon bidirectionnelle avec ses quatre voisines. Cette possibilité est illustrée à la figure 77.

6. Conclusion

Dans ce chapitre, nous avons montré que notre architecture autorise diverses utilisations. L'implémentation d'un algorithme générique (illustré

par la convolution), le prototypage rapide (illustré par les algorithmes de watermarking et de cryptanalyse) et enfin l'implémentation de larges schémas de calculs (illustrés par les deux robots et la simulation de la voiture). L'utilisateur ne doit pas nécessairement avoir une connaissance approfondie des FPGA pour l'utiliser. Néanmoins, un utilisateur expert pourra en tirer le maximum en réalisant ses propres ressources optimisées (illustré par la transformée multi-résolution) au sein d'une architecture utilisant les ressources génériques déjà développées.

Les taux de calcul offerts par la carte varient d'une application à l'autre allant de 320 Mop/s (dont 80 Mflop/s) pour les larges schémas de calcul à théoriquement 5 Gop/s (si l'on utilisait les FPGA uniquement avec des ressources de calcul) pour la transformée multi-résolution. Par ailleurs, la technologie FPGA ne cesse de s'améliorer puisque Xilinx annonce un circuit équivalent à dix millions de portes logiques (le Virtex II), soit cent fois plus que le FPGA que nous avons utilisé et avec des fréquences quadruples. A l'heure où nous écrivons ces lignes, Altera vient de proposer une nouvelle architecture "Stratix" pour concurrencer le Virtex II.

Un avantage certain de notre architecture est de pouvoir s'adapter directement aux progrès de la technologie. Le tableau suivant permet une comparaison entre les performances des diverses architectures présentées dans l'état de l'art et l'architecture DDR. Outre les résultats mesurés, nous y présentons une extrapolation des performances de l'architecture DDR pour les circuits FPGA les plus récents en utilisant les chiffres annoncés par les fabricants. Toutes les performances entre parenthèses représentent une prévision ou une extrapolation.

Architecture	Performances	Unités
Alpha	(576 simulés) (16384)	Opérateurs
Matrix	(10)	Gop/s
PipeRench	(30)	Gop/s
Rapid	1.6	Gmac/s
Chameleon	24	Gop/s
FPPA	(1000)	mips
Morphosys	(6.4)	Gop/s
Raw Machines	(3.6)	Gop/s
TTA	(320)	Mop/s
DDR (FLEX10K100)	120	Mop/s/FPGA
DDR (APEX1K1500)	(2.88)	Gop/s/FPGA
DDR (Stratix)	(56)	Gop/s/FPGA
DDR (Virtex 2)	(90)	Gop/s/FPGA

Ce tableau révèle deux choses importantes :

(1) Peu d'architectures reconfigurables de large grain ont été implémentées physiquement dans un circuit dont on peut mesurer les performances. La plupart des centres de recherche s'arrêtent devant l'obstacle du coût d'une telle réalisation et le risque de se faire dépasser très rapidement par les progrès de la technologie. Il faut néanmoins mentionner que ce genre de recherche est très récente. Plusieurs centres sont maintenant sur le point d'aboutir à une implémentation physique.

(2) Notre architecture, utilisée sur un circuit récent (APEX 20K1500), permet d'obtenir des performances tout-à-fait honorables par rapport aux prévisions des autres architectures. De plus, l'utilisation des FPGA permet de s'adapter instantanément aux innovations technologiques et d'avoir accès directement à des taux de calcul largement supérieurs à ceux annoncés pour des architectures reconfigurables à large grain, même les plus récentes.

CHAPITRE 5

Conclusion

Ce travail était consacré à l'implémentation d'algorithmes au moyen d'une architecture reconfigurable qui utilise un processus de synchronisation des opérations par l'arrivée des données. Il a commencé en 1996 avec l'arrivée sur le marché d'un FPGA de haute densité (cent mille portes logiques équivalentes). L'avènement de ce type de circuit (qui fut suivi par de nombreux autres de plus en plus larges et rapides) a marqué un tournant dans l'ère des systèmes digitaux en ouvrant des perspectives de traitement numérique des données impensables auparavant.

Pressentant la puissance de calcul qui se cachait derrière cette technologie novatrice (au vu notamment des systèmes multi-FPGA de grande taille développés à l'époque), notre premier travail a été de réaliser une carte regroupant un maximum de ces circuits auxquels nous avons adjoint de la mémoire statique très rapide. Ce travail s'est concrétisé sous forme d'une carte PCI de type long contenant quatre FPGA FLEX10K100 (Altera), chacun associé à une mémoire 32 bits de 2 Moctets interconnectés par des liaisons fixes et d'autres programmables. Nous avions donc à disposition assez de ressources logiques pour élaborer ce qui deviendrait à terme

Une architecture reconfigurable synchronisée par les données.

Après avoir validé notre carte sur quelques configurations de test, l'implémentation manuelle d'un algorithme de convolution longue nous a montré que nous pouvions obtenir un taux de calcul de 320 millions de MAC (sans compter les calculs d'adresses) avec un résultat final sans perte de précision sur 43 bits. Cette puissance de calcul prouvait que notre architecture était assez puissante pour élaborer des systèmes complexes jusque là réservés aux seuls ASIC ou processeurs dédiés.

C'est en considérant le temps passé à développer et optimiser manuellement cet algorithme que l'idée nous est venue d'étudier la faisabilité d'une implémentation "automatisée" d'algorithmes sur notre architecture dans un premier temps et puis sur toute architecture utilisant de la

logique reconfigurable de grain fin. En parcourant la littérature concernant les systèmes reconfigurables de l'époque, notre attention a été attirée par l'architecture Xputer (Université de Kaiserslautern), qui offrait un parallélisme important grâce à la synchronisation des opérations par les données. Cependant, le concept était utilisé dans le cadre d'un système reconfigurable de grain large, ce qui limitait l'utilisation de ce type de synchronisation aux seuls cas de figure prévus d'avance. Nous avons alors imaginé un système qui pourrait reprendre à son compte les avantages de l'architecture Xputer et les élargir jusqu'à obtenir une architecture beaucoup plus générique utilisable sur des systèmes reconfigurables de grain fin.

Une recherche bibliographique a montré que la synchronisation des opérations par les données est un concept déjà ancien (milieu des années 1970) qui avait dû être abandonné dans les années 1990 car les architectures utilisant cette approche étaient moins performantes que les évolutions de l'architecture von Neumann (RISC, VLIW, superscalaires...) Néanmoins, ces dernières années nous ont montré que ce type d'architecture n'était pas sans intérêt avec l'arrivée des systèmes reconfigurables. Plusieurs architectures reconfigurables de grain large ont été développées dans des centres de recherche et surpassent les processeurs conventionnels pour des applications données.

Notre approche est une tentative de généraliser toutes les solutions particulières proposées dans divers centres de recherche en une architecture simple, générique et performante qui doit être accessible non seulement aux spécialistes de la synthèse sur FPGA mais également aux experts en logiciel. Pour y parvenir, nous utilisons le concept des réseaux de Pétri que nous modifions pour aboutir aux DDR (Data Driven Resources) qui requièrent une technologie reconfigurable de grain fin pour être implémentées. De cette manière, les ressources de calcul et les connexions sont définies en fonction de l'application plutôt que l'inverse comme c'est le cas dans les systèmes utilisant un grain de configuration plus large. Ceci optimise la taille de l'architecture, les délais de connexion et le parallélisme tout en ne nécessitant pas de connaissances spéciales en matière de FPGA.

Au chapitre premier de cet ouvrage, nous avons commencé par introduire le concept de la synchronisation des opérations par les données de manière intuitive. Nous l'avons placé dans son contexte historique en relation avec l'approche proposée par J. von Neumann et celles plus récentes qui s'en inspirent. Nous parcourons également l'évolution des systèmes digitaux de ces dernières années. Dans le deuxième chapitre,

nous avons d'abord présenté l'état de l'art en matière de systèmes reconfigurables de grain large avant de considérer ceux de grain fin. Chaque architecture de grain large a été présentée puis analysée selon trois critères : le type d'unité de traitement, le type de connexions et les possibilités de reconfiguration. Une séparation a été réalisée entre les approches utilisant un microprocesseur et les autres car l'architecture que nous proposons peut l'intégrer mais elle n'en a pas besoin. Un ensemble de points forts, qu'il serait intéressant de retrouver dans l'architecture que nous proposons, en a été déduits. Pour les systèmes de grain fin, nous avons restreint notre état de l'art aux deux constructeurs principaux (Altera et Xilinx) dont nous avons décrit l'évolution de leurs technologies jusqu'aujourd'hui. Le troisième chapitre est consacré à la présentation et l'étude de notre architecture. Après une brève description des cartes développées, on y trouve une présentation des Réseaux de Pétri dont elle est issue, différentes cellules de base qui ont été regroupées pour former une librairie, une méthodologie heuristique pour implémenter un algorithme avec les DDR et finalement les outils logiciels que nous avons développés et utilisés pour valider notre approche. Au chapitre quatre, nous avons relaté les diverses architectures que nous avons testées et qui illustrent plusieurs types d'applications de notre architecture (implémentation d'un algorithme en utilisant la librairie, prototypage rapide, définition et utilisation de nouvelles DDR par des utilisateurs experts et implémentation de larges schémas de calcul en robotique).

Au terme de ce travail, nous résumons maintenant son originalité, ses points forts, ses faiblesses relatives et les possibilités qu'il ouvre dans diverses directions qui restent encore à explorer.

1. Originalité

L'originalité principale de ce travail consiste dans la méthode que nous proposons pour transformer un réseau de Pétri appliqué au traitement de données en une architecture directement synthétisable sur FPGA et ce sans connaissance particulière de ces derniers pour offrir un accès direct au matériel aussi bien aux experts en logiciel qu'en matériel. Cette méthode permet en effet de reconsidérer l'immense travail qui avait déjà été accompli dans le domaine des systèmes à flux de données mais qui avait dû être mis de côté car la technologie n'était pas prête à l'époque pour accueillir une architecture telle que celle que nous proposons. Notre approche, a l'instar d'autres utilisant de la logique reconfigurable de grain large, permet ainsi une nouvelle vision sur la manière de réaliser un traitement numérique des données.

2. Avantages

Les avantages de cette approche sont multiples. Citons d'abord le taux de parallélisme important qu'elle offre, notamment parce que les calculs d'adresses se font de manière parallèle avec les opérations sur les données proprement dites. Notons ensuite le faible temps de développement d'une telle architecture si l'on dispose d'une interface graphique adéquate. Insistons enfin sur le fait qu'elle est accessible aux experts en matériel qui pourront en tirer un profit maximum mais également aux experts en logiciel qui pourront utiliser les ressources déjà développées par les premiers sans devoir entrer dans toute la complexité d'un système reconfigurable de grain fin.

Si l'on compare les tableaux récapitulatifs des architectures reconfigurables de grain large présentés au chapitre 2 point 2.5, et particulièrement les sept points principaux présentés en fin de section, nous voyons que notre architecture est pensée pour intégrer les avantages de ses prédécesseurs.

(1) Le pipeline est présent à deux niveau : le réseau de Pétri lui-même et au niveau des opérateurs nécessitant plusieurs cycles d'horloge pour réaliser une opération. Dans le premier cas, ce pipeline vient directement de la règle de flambage des réseaux de Pétri qui prévoit que toutes les transitions prêtes flambent en même temps, ce qui a été concrétisé par notre protocole de synchronisation anticipatif. Le second cas concerne les ressources qui demandent plusieurs cycles d'horloge pour réaliser une opération mais qui sont capables de réaliser plusieurs opérations en pipeline grâce à l'utilisation des DDR tampons.

(2) Les générateurs d'adresses autonomes travaillent tous en parallèle et se comportent donc comme un contrôleur DMA synthétisé sur mesure quand ils sont couplés à une ressource de mémoire partagée.

(3) Le calcul local est favorisé par l'utilisation des mémoires imbriquées du FPGA. La topologie du réseau de Pétri peut être organisée de manière à favoriser un maximum de transferts en local en utilisant un réseau de connexions hiérarchiques.

(4) La technologie FPGA permet de réaliser diverses architectures de multiplieurs, notamment une version pipeline rapide. Par ailleurs, les nouveaux circuits développés par Xilinx disposent de multiplieurs intégrés dans la logique.

(5) L'intégration de processeurs au sein de notre architecture peut être envisagée à-priori mais ce point sera discuté un peu plus loin.

(6) La possibilité de reconfigurer localement certaines parties du FPGA est un point qui, à priori, n'est pas incompatible avec l'architecture que nous proposons. Néanmoins, cela demanderait d'une part le développement de ressources spécialisées et d'autre part une technologie capable de les supporter, ce qui n'est pas encore le cas sur les FPGA de grande taille.

(7) La combinaison de l'aspect générique et reconfigurable est certainement un des points les plus intéressant de notre architecture. En effet, le problème le plus critique dans toute architecture de grain large est le fait qu'il faut adapter l'algorithme au nombre des ressources disponibles, au type et au nombre de communications réalisables etc. Notre architecture, en utilisant une configuration de grain fin permet de définir les ressources à utiliser en fonction de l'algorithme à réaliser plutôt que l'inverse. Par ailleurs, nous voyons que des structures telles que celle utilisée pour réaliser les larges schémas de calcul sont à la fois figées (l'architecture générale est donnée) et à la fois reconfigurables (le type d'opération et l'ordre des opérations sont déterminés par les programmes mémorisés dans les générateurs d'adresses et de sélection de canal).

3. Inconvénients

Le fait d'utiliser de la logique reconfigurable de grain fin présente cependant des faiblesses par rapport à d'autres approches. En règle générale, les circuits sont plus larges et donc plus lents car ils doivent intégrer toutes les ressources nécessaires au caractère reconfigurable de chacune des petites cellules logiques. Néanmoins, il faut aussi remarquer que le développement des systèmes reconfigurables de grain large sont principalement réalisés par des laboratoires de recherche (à l'exception de Chameleon) tandis que les FPGA sont développés par des sociétés qui distribuent leurs produits dans le monde entier car ils sont beaucoup plus génériques. En conséquence, la technologie utilisée dans les systèmes reconfigurable de grain large est toujours en retard par rapport aux FPGA, ce qui compense les vitesses plus faibles de ces derniers. A tire d'exemple, les FPGA réalisés maintenant par Xilinx disposent d'entrées/sorties avec adaptation de l'impédance de ligne du circuit imprimé pour travailler à des fréquence de 400Mhz ! Par ailleurs, les fabricants de FPGA commencent à inclure des fonctions précablées au sein de leurs circuits telles que multiplieurs et noyaux de processeurs de sorte que les problèmes de délai au sein du FPGA sont moins critiques. Notons enfin le problème de la puissance consommée. C'est sans doute le point le plus faible

des FPGA. Les circuits actuellement disponibles ne disposent pas de logique dédiée pour économiser la puissance en arrêtant l'horloge par exemple. C'est regrettable car notre architecture pourrait en tirer profit sans grand changement. En effet, le pipeline qui existe entre la validation de la condition de transfert et le transfert effectif des données permettrait directement de commander l'horloge quand un transfert et un calcul ont lieu. On pourrait envisager également une synthèse automatique de cette logique lorsque la cible est un ASIC plutôt qu'un FPGA.

4. Perspectives

Au fur et a mesure que ce travail de recherche progressait, des possibilités nouvelles s'offraient à nous et nous avons dû limiter le cadre de notre recherche pour ne pas nous disperser. Nous présentons maintenant diverses voies de recherche qui mériteraient d'être explorées.

(1) La réalisation d'une interface graphique : Décrire les diverses ressources et connexions en langage C est un choix historique qui a été posé à une époque ou nous ne soupçonnions pas encore que notre architecture pourrait un jour s'apparenter aux réseaux de Pétri. Au terme de ce travail, il apparaît clairement que ceux-ci offriraient un moyen privilégié de représenter une architecture DDR, de la simuler, et finalement de la synthétiser, au prix de quelques extensions au concept du réseau de Pétri. Vu le nombre impressionnant de logiciels qui existent déjà dans le domaine, il nous paraîtrait intéressant de repartir d'un tel logiciel pour aboutir à une interface utilisateur qui soit conviviale et qui permettrait un usage aisé du concept de DDR à toute personne capable de comprendre un réseau de Pétri. Une autre approche possible serait l'utilisation de SystemC pour modéliser à un niveaux plus élevé les transferts entre DDR avant de synthétiser l'architecture.

(2) Utiliser les mémoires adressables par le contenu : Une limitation des machines à jetons colorés de l'époque était la difficulté d'apparier les jetons correspondants à une même instruction. La présence de mémoires adressables par le contenu au sein des FPGA récents permettrait de développer des ressources pour le traitement de données par machines à jetons colorés, ce qui augmenterait encore le parallélisme.

(3) Intégrer le processeur : C'est certainement le point le plus prometteur. L'architecture von Neumann reste d'application dans

bon nombre d'applications, ne serait-ce que pour gérer un système d'exploitation. Comment dès lors intégrer cette technologie au sein des DDR ? Le système le plus évident consisterait à le doter d'un certain nombre de canaux d'entrées-sorties qui feraient l'interface entre les DDR et le processeur. C'est en fait l'architecture que nous avons utilisée en développant une carte PCI et en assignant certaines adresses à des ports PCI virtuels répartis au sein de la logique. Néanmoins, il serait sans doute intéressant de formaliser davantage cette approche pour aboutir à un système multi-processeur générique efficace. Nous pensons également qu'il serait bon de développer des outils de partitionnement (semi-)automatique des tâches à réaliser par processeur et par DDR et de travailler sur une implémentation automatique des deux parties, ce qui est loin d'être trivial.

5. L'avenir des systèmes digitaux

Nous ne pouvons pas terminer cet ouvrage sans considérer l'avenir de notre architecture et des systèmes digitaux en général. Tous les centres de recherche tentent de trouver le système "parfait" qui parallélise, partitionne, compile, synthétise, simule et fournit directement les fichiers de programmation et d'interconnexion des composants utilisés. Malgré les immenses progrès réalisés dans les architectures numériques et en génie informatique, un tel système n'existe pas. Peut-être n'existera-t-il jamais ?

Tous les algorithmes de traitement de données sont différents. Il est peu probable qu'une méthode unique puisse satisfaire les exigences des divers cas de figure. Si l'on considère par exemple les systèmes à contrôle dominant par rapport à ceux qui sont synchronisés par les données, on constate que les propriétés des premiers sont pratiquement à l'opposé de celles des seconds :

- Les architecture à contrôle dominant sont très efficaces pour les algorithmes contenant beaucoup de tests et peu de calculs entre chaque test car dans ce cas, le parallélisme local peut être exploité par des architectures VLIW ou superscalaire. Par contre, pour le traitement massif de données structurées (FIR 1D, 2D, 3D) ou non (larges schémas de calcul), les processeurs actuels ne permettent pas de tirer profit de tout le parallélisme existant au sein de l'algorithme.
- Les architecture synchronisées par les données possèdent à peu près les caractéristiques inverses. Très efficaces pour le traitement parallèle de nombreuses données, elles sont plus lourdes à mettre en oeuvre lorsque

de fréquents tests doivent être réalisés ou encore pour répondre à des interruptions - changements de contexte ...

"La solution", si elle existe, fera nécessairement intervenir les deux approches simultanément. De même qu'une solution parfaite pour régler le traffic des voitures n'a pas encore été découverte à ce jour et que les deux approches sont utilisées selon le type de problème à résoudre, une solution parfaite pour acheminer les opérandes aux opérateurs n'a pas non plus encore été trouvée mais nous avons aujourd'hui une alternative au système à contrôle dominant. Ainsi, de même que dans de nombreux cas, nous préférons utiliser le rond point plutôt que d'attendre stupidement à un feu rouge, l'architecture que nous proposons pourra être utilisée avantageusement dans bon nombre de situations. Il reste à espérer que l'on découvre un moyen de combiner les avantages des deux approches tant pour les voitures à l'entrée d'un carrefour que pour les données à traiter à l'intérieur d'un circuit logique. L'avenir le dira ...

Publications de l'auteur

[1] J.P. David and J.D. Legat. A 400kgates 8 Mbytes SRAM multi-FPGA PCI system. In *int. Workshop on Logic and Architecture Synthesis*, pages 133–137, Grenoble, December 1997.

[2] J.P. David and J.D. Legat. A multi-FPGA based coprocessor for digital signal processing. In *IEEE Benelux Signal Processing Systems Symposium*, pages 59–62, Leuven, March 1998.

[3] J.P. David and J.D. Legat. A data-flow oriented codesing for reconfigurable systems. In *IEEE Rapid System Prototyping (RSP'98)*, pages 207–211, Leuven, June 1998.

[4] J.P. David and J.D. Legat. A FPGA-based implementation of adaptive sound filtering. In *Proc. of CSSP'98*, pages 107–111, Mierlo (Netherlands), November 1998.

[5] J.P. David, J.D. Legat, P. Fisette, and T. Postiau. Implementation of very large dataflow graphs on a reconfigurable architecture for robotic application. In *Proc. of the 15th International Parallel & Distributed Processing Symposium (IPDPS'2001)*, San Francisco, 23-27 April 2001.

[6] J.P. David and J.D. Legat. Architecture synchronisée par les données pour système reconfigurable. In *SympA'7 7ième Symposium en Architectures nouvelles de machines*, pages 75–82, Paris, 24-26 avril 2001.

[7] J.P. David and J.D. Legat. A Generic DSP Data-Driven Architecture for Fine Grain Reconfigurable Systems. In *(en voie de soumission)*, 2002.

[8] J.D. Legat, J.P. David, and P. Desneux. Programmable architectures for subband coding : FPGA-based systems versus dedicated VLSI chip. In *Proc. of IEEE 2nd IMACS Int. Multiconf., CESA'98 Computational Engineering in Systems Applications*, pages 301–305, Nabeul-Hammamet (Tunisia), April 1-4 1998.

[9] J.D. Legat and J.P. David. Design of a multi-FPGA system for rapid prototyping experimentation. In *Proc. of the 2nd European workshop on Microelectronics Education*, pages 231–234, Noordwijkerhout (NL), May 14-15 1998. "Microelectronics Education", ed. T.J. Mouthaan and C. Salm, Kluwer Academic Publ. 0-7923-5107-X.

[10] A.M. Trullemans, R. Ferreira, J.P. David, and J.D. Legat. A multi-FPGA system for prototyping power conscious algorithms. In *15th Design of Circuits and Integrated Systems Conference (DCIS 2000)*, pages 41–46, Montpellier, November 21-24 2000.

Références bibliographiques

[11] W.B. Ackerman. Data flow languages. In *Computer*, pages 15–25, February 1982.

[12] W.B. Ackerman and J.B. Dennis. VAL-A Value-Oriented Algorithmic Language : Preliminary Reference manual. In *MIT Laboratory for Computer Science Technical Report TR-218*, MIT, Cambridge, Mass., June 1979.

[13] R. Alexandre and A. Germain. Implémentation sur une carte multi-FPGA d'un algorithme de watermarking destiné à la vidéo numérique. Master's thesis, Laboratoire de microélectronique, UCL, Louvain-La-Neuve, Belgium, Juin 1999.

[14] Altera. "Altera corporation : The programmable solutions company". http ://www.altera.com.

[15] C. Amerijckx. *Architecture d'un réseau d'interconnexion au sein d'un composant programmable complexe.* PhD thesis, Laboratoire de microélectronique, UCL, Louvain-La-Neuve, Belgium, 2000.

[16] E.A. Ashcroft and W.W. Wadge. Lucid, a non-procedural language with iteration. In *Comm. ACM*, volume 20 No. 7, pages 519–526, July 1977.

[17] P.M. Athanas and H.F. Silverman. Processor reconfiguration through instruction set metamorphosis. In *Computer*, pages 11–19, March 1993.

[18] J. Backus. Can programming be liberated from the von Neumann style ? a functionnal style and its algebra of programs. In *Comm. ACM*, volume 21 No. 8, pages 613–641, August 1978.

[19] S. Bakshi and D.D. Gajski. Hardware/Software partitioning and pipelining. In *DAC*, pages 713–716, 1997.

[20] F. Boussinot and R. De Simone. The Esterel language. In *Proceedings of the IEEE*, volume 79, pages 1293–1304, September 1991.

[21] D.A. Buell, J.M. Arnold, and W.J. Kleinfelder. *Splash 2 FPGAs in a Custom Computing Machine.* IEEE Computer Society Press, Los Alamitos, California, 1996.

[22] H. Corporaal. *Transport Triggered Architectures Design and Evaluation.* PhD thesis, Technische Universiteit Delft, Delft, September 1995.

[23] A.L. Davis and R.M. Keller. Data flow program graphs. In *Computer*, pages 26–41, February 1982.

[24] A. Dehon. The density advantage of configurable computing. In *Computer*, pages 41–49, April 2000.

[25] J.B. Dennis. Dataflow supercomputers. In *Computer*, pages 48–56, November 1980.

[26] J.B. Dennis and D.P. Misunas. A preliminary architecture for a basic data-flow processor. In *Proc. 2nd ISCA*, pages 126–132, January 1975.

[27] R. Duncan. A survey of parallel computer architectures. In *Computer*, pages 5–16, February 1990.

[28] C. Ebeling, D.C. Cronquist, and P. Franklin. Rapid - Reconfigurable Pipelined Datapath. In *The 6th International Workshop on Field-Programmable Logic and Applications*, 1996.

[29] R. Esser, J. Janneck, and M. Naedele. Using an object-oriented Petri net tool for heterogeneous systems design : A case study. In *Workshop Algorithmen und Werkzeuge fuer Petri Netze*, Berlin, Germany, October 2-3 1997.

[30] W.J. Fang and A.C.H. Wu. A hierarchical functionnal structuring and partitioning approach for multiple-FPGA implementations. In *IEEE Transactions on computer-aided design of integrated circuits and systems*, volume 16 No. 10, pages 1188–1195, October 1997.

[31] J.T. Feo, D.C. Cann, and R.R. Oldehoeft. A report on the Sisal language project. In *Journal of Parallel and Distributed Computing*, volume 10 No. 4, pages 349–366, December 1990.

[32] A. Fin, F.Fummi, and M.Signoretto. SystemC : A homogenous environment to test embedded systems. In *Proc. IEEE Codesign Conference (CODES)*, pages 17–22, Copenhagen, Denmark, March 2001.

[33] P. Fisette and J.C. Samin. Robotran : Symbolic generation of multibody system dynamic equations. In *Advanced Multibody System Dynamics : Simulation and Software Tools*, pages 373–378, Dordrecht, 1993. Kluwer Academic Publishers.

[34] D.D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC : Specification Language and Methodology.* Kluwer Academic Publisher, 2000.

[35] D. Galloway. The transmogrifier C hardware description language and compiler for FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 136–144, 1995.

[36] V.G. Grafe and J.E. Hoch. The Epsilon-2 hybrid dataflow architecture. In *Proc. of Compcon90*, pages 88–93, San Francisco, CA, March 1990.

[37] S. Guccione. "List of FPGA based computing machines". http ://www.io.com/ guccione/HW_list.html.

[38] R.K. Gupta, C.N. Coelho, and G. De Micheli. Program implementation schemes for hardware software systems. In *Computer*, pages 48–55, January 1994.

[39] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language Lustre. In *Proceedings of the IEEE*, volume 79, pages 1305–1320, September 1991.

[40] R. Hartenstein. A decade of reconfigurable computing : a visionary retrospective. In *DATE 2001, Int'l Conference on Design Automation and Testing in Europe - and Exhibit*, Munich, Germany, March 12-15 2001.

[41] R. W. Hartenstein and J. Becker. Hardware/Software co-design for data-driven Xputer-based accelerators. In *Proc. of 10th Int. Conf. on VLSI Design (Theme : VLSI in Multimedia Applications)*, Hyderabad, India, January 4-7 1997.

[42] R.W. Hartenstein. The microprocessor is no more general purpose : why future reconfigurable platforms will win. In *Proceedings of the International Conference on Innovative Systems in Silicon, ISIS'97*, Austin, Texas, USA, October 8-10 1997.

[43] R.W. Hartenstein, R. Kress, and H. Reinig. A new FPGA architecture for word-oriented datapaths. In *4th Int. Workshop On Field Programmable Logic And Applications, FPL'94*, Prague, September 7-10 1994.

[44] S. Hauck. Asynchronous design methodologies : An overview. In *Proceedings of the IEEE*, volume 83 No. 1, pages 69–93, January 1995.

[45] S. Hauck. *Multi-FPGA Systems*. PhD thesis, University of Washington, Dept. of C.S.E., September 1995.

[46] S. Hauck. The future of reconfigurable systems. In *5th Canadian Conference on Field Programmable Devices*, Montreal, June 1998.

[47] S. Hauck. The roles of FPGAs in reprogrammable systems. In *Proceedings of the IEEE*, volume 86, pages 615–638, April 1998.

[48] S. Hauck and G. Borriello. An evaluation of bipartitioning techniques. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 16 No. 8, pages pp 849–866, August 1997.

[49] J.R. Hauser and J. Wawrzynek. Garp : A mips processor with a reconfigurable coprocessor. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '97)*, April 16-18 1997.

[50] J.P. Hayes. *Computer Architecture and Organization.* Mc Graw Hill, Boston, 1998.

[51] J.R. Henkel. A Hardware/Software partitioner using a dynamically determined granularity. In *DAC*, pages 691–696, 1997.

[52] Chameleon Systems. Inc. "Chameleon systems – your communications platform". http ://www.chameleonsystems.com.

[53] A. Krukowski and I. Kale. Simulink/matlab-to-vhdl route for full-custom/FPGA rapid prototyping of dsp algorithms. In *Matlab DSP Conference (DSP'99)*, Tampere, Finland, November 16-17 1999.

[54] D.C. Ku and G. De Micheli. HardwareC - a language for hardware design, version 2.0. In *CSL TR 90-419*, Stanford University, Stanford, CA, April 1990.

[55] S. Kumar, J.H. Aylor, B.W. Johnson, and W.A. Wulf. *The codesign of embedded systems A unified Hardware/Software Representation.* Kluwer Academic Publisher, Norwell, 1996.

[56] E.A. Lee. Overview of the Ptolemy project. In *Technical Memorandum UCB/ERL M01/11*, University of California, Berkeley, March 6 2001.

[57] S. Martin. Implémentation du DES et cryptanalyse sur FPGA. Master's thesis, Laboratoire de microélectronique, UCL, Louvain-La-Neuve, Belgium, Septembre 1999.

[58] J.Y. Mertès. *Optimisation et dimmensionnement de la transformée multirésolution en précision finie.* PhD thesis, Laboratoire de microélectronique, UCL, Louvain-La-Neuve, Belgium, Octobre 1995.

[59] G. De Micheli and R.K. Gupta. Hardware/Software co-design. In *Proceedings of the IEEE*, volume 85, March 1997.

[60] E. Mirsky and A. Dehon. Matrix : A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *Proc. of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 157–166. IEEE CS Press, 1996.

[61] M.Naedele and J.W. Janneck. Design patterns in Petri net system modeling. In *Proceedings ICECCS'98*, pages 47–54, 1998.

[62] W.A. Najjart, E.A. Lee, and G.R. Gao. Advances in the dataflow computational model. In *Parallel Computing*, volume 25, pages 1907–1929, 1999.

[63] R.S. Nikhil. Id version 90.0 reference manual. In *Computation Structures Group Memo 284-1*, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1990.

[64] P. Nussbaum, B. Girau, and A. Tisserand. Field programmable processor arrays. In *Proceedings of the International Conference on Evolvable Systems*, pages 311–322, October 1998.

[65] Optimagic. "Programmable logic jump station". http ://www.optimagic.com.

[66] I. Page. Constructing hardware-software systems from a single description. In *Journal of VLSI Signal Processing*, volume 12, pages 87–107, 1996.

[67] G.M. Papadopoulos and D.E. Culler. Monsoon : An explicit token store architecture. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, Washington, May 1990.

[68] M.F. Parkinson and S. Parameswaran. Profiling in the ASP codesign environment. In *Proceedings of the The Eighth International Symposium on System Synthesis (ISSS'95)*, pages 128–133, 1995.

[69] J.L. Peterson. *Petri net theory and the modeling of systems.* Prentice-Hall, Inc., Englewood Cliffs, London, 1981.

[70] M. Porrmann, H. Kalte, U. Witkowski, J.C. Niemann, and U. Rückert. A dynamically reconfigurable hardware accelerator for self-organizing feature maps. In *Proceedings of The 5th World Multi-Conference on Systemics, Cybernetics and Informatics, SCI 2001*, volume 3, pages 242–247, Orlando, Florida, USA, July 2001.

[71] T. Postiau, P. Fisette, and J.D. Legat. Fine grain parallelization of multibody system equations of motion. In *ParCo'99 (Parallel Computing)*, pages 193–200, Delft (The Netherlands), August 1999.

[72] H. Schmit. Incremental reconfiguration for pipelined applications. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 47–55, 1997.

[73] S. Schulz and J.W. Rozenblit. Model-based codesign. In *Computer*, pages 60–67, August 1998.

[74] J. Silc, B. Robic, and Th. Ungerer. *Processor Architecture : From Dataflow to Superscalar and Beyond.* Springer-Verlag, 1999.

[75] D. Sima, T. Fountain, and P. Kacsuk. *Advanced Computer Architectures a design space approach.* Addison-Wesley, Harlow, 1997.

[76] H. Singh, M-H Lee, G. Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M.C. Filho. Morphosys : An integrated reconfigurable system for data-parallel and computation-intensive applications. In *IEEE transactions on computers*, volume 49, No. 5, pages 465–481, May 2000.

[77] S.Sakai, Y.Kodama, and Y.Yamaguchi. Prototype implementation of a highly parallel dataflow machine EM-4. In *IPPS'91*, pages 278–286, 1991.

[78] H. Terada, S. Miyata, and M. Iwata. Ddmp's : Self-timed superpipelined data-driven multimedia processors. In *Proceedings of the IEEE*, volume 87, pages 282–296, February 1999.

[79] R. Tessier and W. Burleson. Reconfigurable computing and digital signal processing : A survey. In *Journal of VLSI Signal Processing*, May/June 2001.

[80] P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins. Data-driven and demand-driven computer architecture. In *Computing Surveys*, volume 14 No. 1, pages 4–54, March 1982.

[81] VCC. "Virtual Computer Corporation – manufacturer of reconfigurable computer systems". http ://www.vcc.com.

[82] L. Verdoscia. *ALFA fine grain dataflow machine.* International Programming, m.a. orgun and e.a. ashcroft edition, 1996.

[83] X. Verians. *Parallelism extraction between instruction flows with superscalar-like techniques for multiprocessor or multithreaded architectures.* PhD thesis, Laboratoire de microélectronique, UCL, Louvain-La-Neuve, Belgium, July 2000.

[84] D. Verkest, K. Van Rompaey, I. Bolsens, and H. De Man. Coware a design environment for heterogeneous hardware/software systems. In *Design Automation for Embedded Systems*, volume 1, pages 357–386, October 1996.

[85] J. von Neumann. First draft of a report on the edvac. In *Contract No. W-670-ORD-492*, Moore School of Electrical Engineering, Univ. of Penn., Philadelphia, June 30 1945.

[86] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable Active Memories : the coming of age. In *IEEE Trans. on VLSI*, volume 4, pages 56–69, March 1996.

[87] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and

A. Agarwal. Baring it all to software : Raw machines. In *Computer*, pages 86–93, September 1997.

[88] I. Watson and J. Gurd. A practical data flow computer. In *Computer*, pages 51–57, February 1982.

[89] T. Werner and V. Akella. Asynchronous processor survey. In *Computer*, pages 67–76, November 1997.

[90] N. Wirth. Hardware compilation : Translating programs into circuits. In *Computer*, pages 25–31, June 1998.

[91] W.H. Wolf. Hardware - Software co-design of embedded systems. In *Proceedings of the IEEE*, volume 82, pages 967–989, July 1994.

[92] P.R. Woodward. Perspectives on supercomputing. In *Computer*, pages 99–111, October 1996.

[93] Xilinx. "Xilinx : Programmable Logic Devices, FPGA & cpld". http ://www.xilinx.com.

[94] Xilinx. "The role of distributed arithmetic in FPGA-based signal processing". http ://www.xilinx.com/appnotes/theory1.pdf, December 1996.

Annexe 1 : Exemple de code VHDL

La synchronisation des DDR de traitement de données

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity synchro is
  generic (n_input: integer:=2;
    latency: integer:=2);

  port(
    reset: in STD_LOGIC;
    clk: in STD_LOGIC;
    In_Strobe: in STD_LOGIC_VECTOR(n_input-1 downto 0);
    In_Ena: out STD_LOGIC_VECTOR(n_input-1 downto 0);
    In_Latch: out STD_LOGIC_VECTOR(n_input-1 downto 0);
    In_Full: out STD_LOGIC_VECTOR(n_input-1 downto 0);
    Out_Strobe: out STD_LOGIC;
    Out_Ena: in STD_LOGIC);
end synchro;

architecture Arch_Simu of synchro is
  signal Buffer_Full: STD_LOGIC_VECTOR(n_input-1 downto 0);
  signal Enable_In: STD_LOGIC_VECTOR(n_input-1 downto 0);
  signal And_In_Full: STD_LOGIC;
  signal Out_Strobe_tmp: STD_LOGIC;
  signal Computed: STD_LOGIC_VECTOR(1 to Latency);
  signal Computed_tmp: STD_LOGIC_VECTOR(1 to Latency);

  begin
  process(Out_Ena,Buffer_Full,Out_Strobe_tmp)
  variable var_And_In_Full: STD_LOGIC;
```

```
    begin
    var_And_In_Full:='1';
    for i in 0 to n_input-1 loop
      var_And_In_Full:=var_And_In_Full and Buffer_Full(i);
    end loop;
    for i in 0 to n_input-1 loop
      Enable_In(i)<=not Buffer_Full(i) or Out_Strobe_tmp;
    end loop;
    And_In_Full<=var_And_In_Full;
  end process;

  In_Full<=Buffer_Full;
  In_Ena<=Enable_In;

  Computed_tmp(1)<=And_In_Full;  -- unused
  Computed(1)<=And_In_Full;

  L1:if (Latency=2) generate
    process(clk,reset)
      begin
      if (reset='1') then
        Computed_tmp(2)<='0';
      elsif (clk'event and (clk='1')) then
        Computed_tmp(2)<=And_In_Full and not Out_Strobe_tmp;
      end if;
    end process;
    Computed(2)<=Computed_tmp(2);
  end generate;

   L2:if (Latency>2) generate
    process(clk,reset)
      begin
      if (reset='1') then
        for i in 2 to Latency loop
          Computed_tmp(i)<='0';
        end loop;
      elsif (clk'event and (clk='1')) then
        for i in 2 to Latency loop
          if (i=2) then
            Computed_tmp(i)<=And_In_Full and
                              not Out_Strobe_tmp;
          else
```

```
          Computed_tmp(i)<=Computed_tmp(i-1) and
                            not Out_Strobe_tmp;
        end if;
      end loop;
    end if;
  end process;
  Computed(2 to Latency)<=Computed_tmp(2 to Latency);
end generate;

process(clk,reset)
  begin
  if (reset='1') then
    In_Latch<=(others=>'0');
    Buffer_Full<=(others=>'0');
  elsif (clk'event and (clk='1')) then
    In_Latch<=Enable_In;
    for i in 0 to n_input-1 loop
      if (Enable_In(i)='1') then
        Buffer_Full(i)<=In_Strobe(i);
      end if;
    end loop;
  end if;
end process;

Out_Strobe_tmp<=Computed(Latency) and Out_Ena;
Out_Strobe<=Out_Strobe_tmp;
end Arch_Simu;
```

La DDR additionneur

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity adder_s is
  generic (width: integer:=16;
    Latency: integer:=3);

  port(
    reset: in STD_LOGIC;
    clk: in STD_LOGIC;
```

```
    Data_In_A: in STD_LOGIC_VECTOR(width-1 downto 0);
    Data_In_A_Strobe: in STD_LOGIC;
    Data_In_A_Ena: out STD_LOGIC;
    Data_In_A_Fast_Ena: out STD_LOGIC;
    Data_In_B: in STD_LOGIC_VECTOR(width-1 downto 0);
    Data_In_B_Strobe: in STD_LOGIC;
    Data_In_B_Ena: out STD_LOGIC;
    Data_In_B_Fast_Ena: out STD_LOGIC;
    Data_Out_A: out STD_LOGIC_VECTOR(width-1 downto 0);
    Data_Out_A_Strobe: out STD_LOGIC;
    Data_Out_A_Ena: in STD_LOGIC);
end adder_s;

architecture Arch_Simu of adder_s is
  CONSTANT n_input: integer:=2;
  signal s_In_Strobe: STD_LOGIC_VECTOR(n_input-1 downto 0);
  signal s_In_Ena: STD_LOGIC_VECTOR(n_input-1 downto 0);
  signal s_In_Latch: STD_LOGIC_VECTOR(n_input-1 downto 0);
  signal s_In_Full: STD_LOGIC_VECTOR(n_input-1 downto 0);

  signal In_Latch_A: STD_LOGIC;
  signal In_Latch_B: STD_LOGIC;

  signal Data_In_A_Latch: STD_LOGIC_VECTOR(width-1 downto 0);
  signal Data_In_B_Latch: STD_LOGIC_VECTOR(width-1 downto 0);

begin
  S1: Synchro
    generic map(n_input,Latency)
    port map(
      reset=>reset,
      lk=>clk,
      In_Strobe=>s_In_Strobe,
      In_Ena=>s_In_Ena,
      In_Latch=>s_In_Latch,
      In_Full=>s_In_Full,
      Out_Strobe=>Data_Out_A_Strobe,
      Out_Ena=>Data_Out_A_Ena);

  s_In_Strobe<=Data_In_B_Strobe&Data_In_A_Strobe;
  In_Latch_B<=s_In_Latch(1);
  In_Latch_A<=s_In_Latch(0);
```

```
  Data_In_B_Ena<=s_In_Ena(1);
  Data_In_A_Ena<=s_In_Ena(0);
  Data_In_B_Fast_Ena<=not s_In_Full(1);
  Data_In_A_Fast_Ena<=not s_In_Full(0);

  process(clk,reset)
    begin
    if (reset='1') then
      Data_In_A_Latch<=(others=>'0');
      Data_In_B_Latch<=(others=>'0');
    elsif (clk'event and (clk='1')) then
      if (In_Latch_A='1') then
        Data_In_A_Latch<=Data_In_A;
      end if;
      if (In_Latch_B='1') then
        Data_In_B_Latch<=Data_In_B;
      end if;
    end if;
  end process;
  Data_Out_A<=Data_In_A_Latch+Data_In_B_Latch;
end Arch_Simu;
```

Annexe 2 : Exemple de C pour DDR

Le code suivant teste un générateur de données basé sur une ROM en envoyant son contenu sur un port PCI virtuel

Déclaration de la carte

Chacun des FPGA et chacune des connexions inter-FPGA est caractérisées par :

(1) Le nombre et la taille des canaux d'écriture.

(2) Le nombre et la taille des canaux de lecture.

(3) Le nombre et la taille des ports d'écriture.

(4) Le nombre et la taille des ports de lecture.

Dans le cas des FPGA, les canaux représentent des ports PCI virtuels tandis que pour les connexions inter-FPGA, les canaux représentent des échanges de données inter-FPGA.

```
#include <stdio.h>
#include "flex_main_function.h"

struct s_function *Cur_FLEX;
struct s_function FLEX1={"FLEX1","","",NORMAL,0,1,{32},{32},
   0,0,{0},{0}};
struct s_function FLEX2={"FLEX2","","",NORMAL,0,0,{32},{32},
   0,0,{0},{0}};
struct s_function FLEX3={"FLEX3","","",NORMAL,0,0,{32},{32},
   0,0,{0},{0}};
struct s_function FLEX4={"FLEX4","","",NORMAL,0,0,{32},{32},
   0,0,{0},{0}};

struct s_function *Cur_FLEX_LIEN_next;
struct s_function FLEX1_LIEN_2={"NEXT1","lien_next","",
   NORMAL,1,1,{32},{32},1,1,{1},{1}};
struct s_function FLEX2_LIEN_3={"NEXT2","lien_next","",
```

```
   NORMAL,1,1,{32},{32},1,1,{1},{1}};
struct s_function FLEX3_LIEN_4={"NEXT3","lien_next","",
   NORMAL,1,1,{32},{32},1,1,{1},{1}};
struct s_function FLEX4_LIEN_1={"NEXT4","lien_next","",
   NORMAL,1,1,{32},{32},1,1,{1},{1}};
```

Déclaration des ressources

Chacune des ressource est caractérisée par tous les paramètres nécessaires à sa déclaration VHDL :

(1) Son nom.

(2) Son type (par rapport à la bibliothèque de DDR).

(3) Les paramètres nécessaires au modèle VHDL : generic (...)

(4) Le type, le nombre et la taille des canaux d'écriture et de lecture.

(5) Le nombre et la taille de ports asynchrones d'écriture et de lecture.

```
struct s_function B_0={"B_0","register_s","20,0",NORMAL,
   1,1,{20},{20},0,0,{0},{0}};
struct s_function B_5={"B_5","register_s","20,0",NORMAL,
   1,1,{20},{20},0,0,{0},{0}};
struct s_function GENG0={"GENG0","data_gen_ROM_s",
   "8,8,13,\"ROM10.mif\"",NORMAL,1,1,{1},{8},0,0,{0},{0}};

struct s_function *(F1_List[100])={&B_0,&B_5,&GENG0,NULL};
```

Génération de l'architecture DDR

Diverses fonctions de génération de code VHDL sont appelées :

(1) L'en-tête du fichier VHDL de niveau 2

(2) Les connexions au ports PCI virtuels

(3) Les connexions entre les DDR et avec les FPGA voisins

(4) Les connexions à la mémoire globale

(5) La fin du fichier VHDL de niveau 2

(6) L'en-tête du fichier VHDL de niveau 1

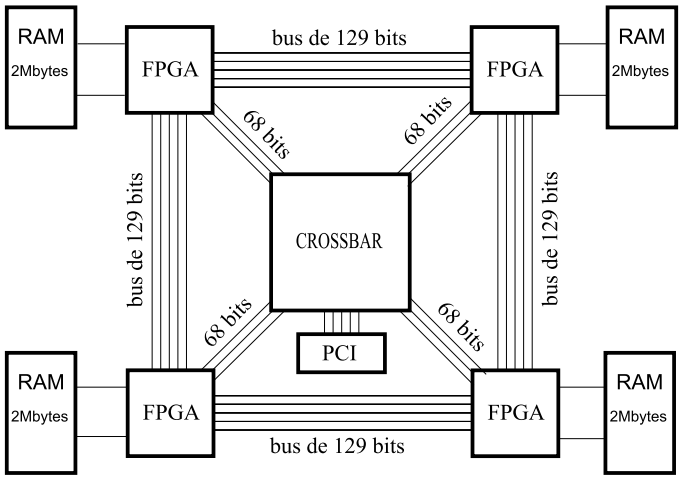
(7) La description de toutes les entrées sorties du FPGA

(8) La fin du fichier VHDL de niveau 1

```
int main() {

   init(1);

   generate_top_of_file();
   generate_list_of_components(F1_List);

   generate_connection_s(&GENG0,0,Cur_FLEX,0);

   generate_connection_s(Cur_FLEX_LIEN_prev,0,&B_0,0);
   generate_connection_s(&B_0,0,Cur_FLEX_LIEN_next,0);

   generate_connection_s(Cur_FLEX_LIEN_next,0,&B_5,0);
   generate_connection_s(&B_5,0,Cur_FLEX_LIEN_prev,0);

   //DEBUG
   generate_string("\ts_GENG0_Data_In_A<=\"0\";");
   generate_string("\ts_GENG0_Data_In_A_Strobe<='0';\n");
   //END DEBUG

   generate_connection_p(Cur_FLEX_LIEN_prev,0,
                         Cur_FLEX_LIEN_prev,0);
   generate_constant_p("0",Cur_FLEX_LIEN_next,0);

   generate_connection_RAM(NULL);

   generate_bottom_of_file();
   terminate();

   init_FLEX(1);
   generate_FLEX();
   terminate_FLEX();

   return 0;
}
```

Annexe 3 : Détail des cartes

La carte multi-FPGA



(1) 4 FPGA EPF10K100GC503-3DX : Chaque circuit contient 4992 cellules logiques et 12 SRAM imbriquées de 2048 bits configurables en mots de 1,2,4 ou 8 bits. Sur les 503 broches du boîtier, 406 peuvent être utilisées en entrée/sortie. La fréquence de travail maximale dépend de l'architecture réalisée et peut monter jusqu'à 100Mhz. Ce circuit est équivalent à cent mille portes logiques selon Altera.

(2) 16 SRAM MCM6246WJ20 : Ces mémoires contiennent chacune 512Koctets et offrent un temps d'accès de 20ns. Elles sont regroupées par quatre pour obtenir un bus de 32 bits sur chacun des FPGA.

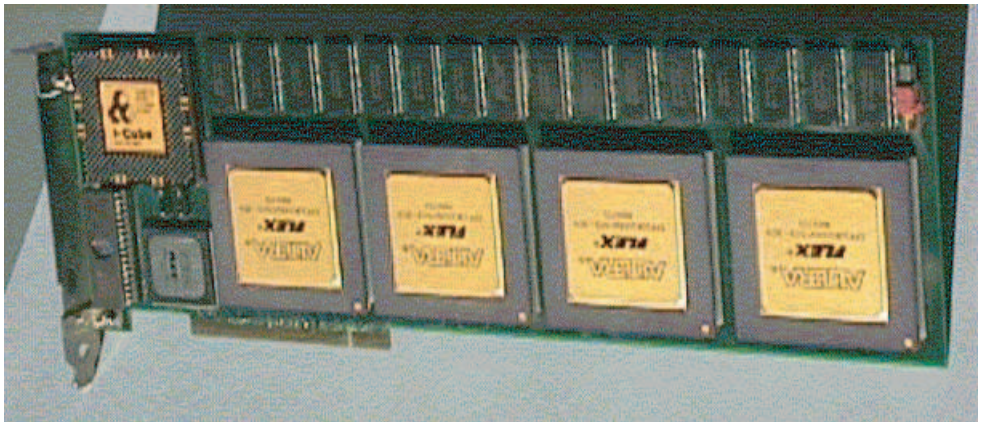
(3) 1 Crossbar IQ320-12 : Ce circuit permet de réaliser l'interconnexion de 320 entrées/sorties de toutes les manières possibles avec un registre en entrée ou en sortie le cas échéant.

(4) 1 FPGA EPF8822ARC208-3 : C'est lui qui réalise l'interface entre le bus PCI et le reste de la carte. Il gère donc la configuration des FPGA et du Crossbar, la transmission des données et l'accès au JTAG.

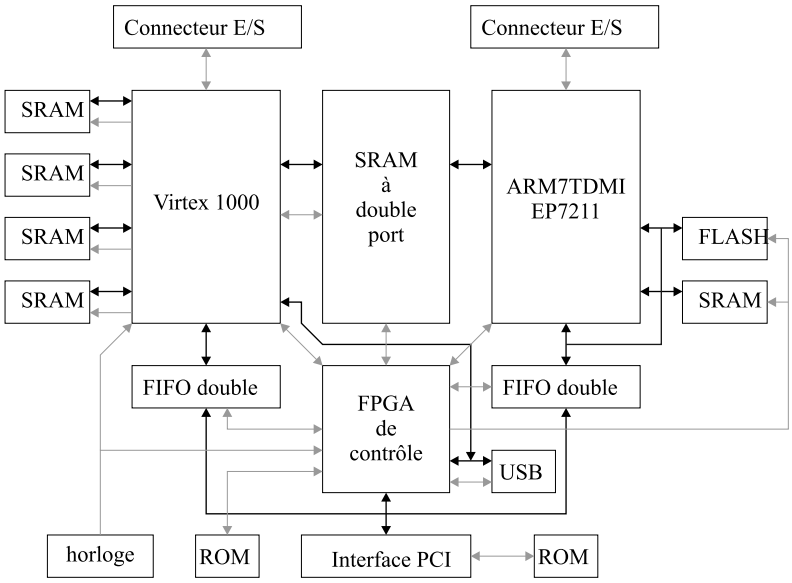
(5) Une EPROM 27C020 : Montée sur support, elle contient le programme de configuration du FPGA qui réalise l'interface PCI.

(6) Un oscillateur intégré : Monté sur support lui aussi, il permet de générer une horloge de fréquence quelconque (selon le type d'oscillateur que l'on connecte) qui sera utilisée par les FPGA.

(7) Un interrupteur : Pour le reset général de la carte.



La carte PCI-VIRTEX



(1) 1 FPGA XCV1000BG560 : Il contient 27648 cellules logiques et 32 mémoires SRAM imbriquées de 4096 bits chacune. 512 broches sur un total de 560 peuvent être configurées en entrée/sortie. Sa fréquence de travail maximale est 200 Mhz.

(2) 1 micro-contrôleur EP7211 : Basé sur une architecture ARM720T (un ARM7TDMI avec MMU), ce micro-contrôleur 32 bits offre également de nombreuses interfaces telles que mémoires - clavier - LCD - série(2x) - CODEC - ADC - PWM - entrées/sorties génériques ... Il peut travailler à une fréquence maximale de 74Mhz en interne tandis que son bus externe travaille à une fréquence moitié.

(3) 5 SRAM CY7C1361TA-6.5 : Ces mémoires (1 Moctets organisés en mots de 32 bits) peuvent travailler en burst jusqu'à une fréquence maximale de 133Mhz. Quatre d'entre elles sont utilisées comme mémoires globales associées au FPGA tandis que la dernière sert de mémoire de données pour le micro-contrôleur.

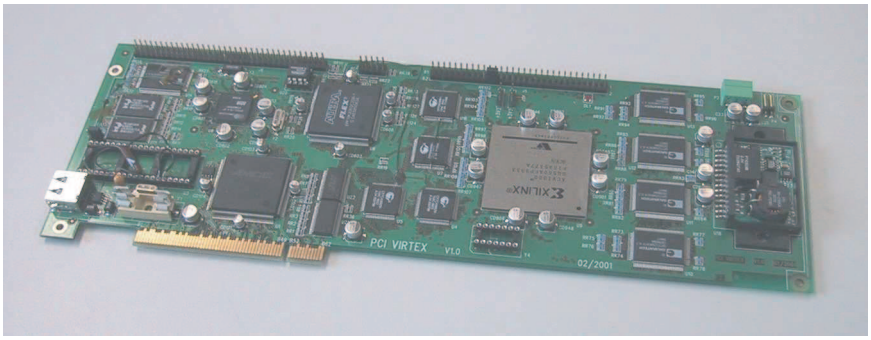
(4) 2 FLASH 28F160 : Ces mémoires offrent un total de 4 Moctets organisés en mots de 32 bits. C'est là que sera mémorisé le système d'exploitation et toutes les données qui doivent demeurer une fois l'alimentation coupée.

(5) 2 DPRAM CY7C09289-9 : Ces mémoires (double accès) sont communes au FPGA et au micro-contrôleur EP7211 et leur permettent d'échanger jusqu'à 256 Koctets organisés en mots de 32 bits à une fréquence maximale de 67Mhz.

(6) 2 FIFO CY7C43682V-15 : Ces circuits permettent les transferts entre le bus PCI, le FPGA et le micro-contrôleur. Ce sont des tampons bi-directionnels qui peuvent contenir jusqu'à 64 Koctets organisés en mots de 32 bits dans chaque sens.

(7) 1 FPGA EPF10K30EQC208-1 : Ce circuit, configuré par une petite ROM série ou par le JTAG, contient toute la logique de contrôle de la carte. Il synchronise les transferts entre les différents circuits en adaptant au besoin le protocole de communication.

(8) 1 micro-contrôleur PIC16C765 : Ce petit micro-contrôleur 8 bits est connecté à l'interface USB. Il doit gérer les transferts entre le bus USB et le reste de la carte.

(9) Un oscillateur intégré : Monté sur support, il permet au FPGA Virtex d'utiliser une horloge différente de celles qui sont déjà présentes sur la carte (33Mhz PCI et 37Mhz EP7211).

(10) Divers connecteurs : Ces connecteurs permettent d'utiliser les entrées/sorties du micro-contrôleur EP7211 et celles du FPGA Virtex non utilisées. Un connecteur JTAG permet d'accéder au FPGA Virtex, au FPGA de contrôle et au micro-contrôleur EP7211 par ce port.