Mozaïc : plate-forme générique de modélisation et de conception d'architectures reconfigurables dynamiquement

Julien Lallet

To cite this version:

Julien Lallet. Mozaïc : plate-forme générique de modélisation et de conception d'architectures reconfigurables dynamiquement. Micro et nanotechnologies/Microélectronique. Université Rennes 1, 2008. Français. ⟨NNT : ⟩. ⟨tel-00446186⟩

HAL Id: tel-00446186
https://theses.hal.science/tel-00446186v1

Submitted on 12 Jan 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

N° d'ordre : 3881

THÈSE

présentée

DEVANT L'UNIVERSITÉ DE RENNES 1

pour obtenir

le grade de : *DOCTEUR DE L'UNIVERSITÉ DE RENNES 1*
Mention : TRAITEMENT DU SIGNAL ET TÉLÉCOMMUNICATIONS

par

Julien LALLET

Équipe d'accueil : Institut de Recherche en Informatique et Systèmes Aléatoires - CAIRN

École Doctorale : Mathématiques, Télécommunications, Informatique, Signal, Systèmes et Électronique

Composante Universitaire : École Nationale Supérieure de Sciences Appliquées et de Technologie

---

MOZAÏC : PLATE-FORME GÉNÉRIQUE DE MODÉLISATION ET DE CONCEPTION D'ARCHITECTURES RECONFIGURABLES DYNAMIQUEMENT

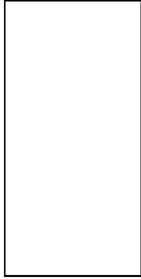
SOUTENUE LE 26 novembre 2008 devant la commission d'Examen

COMPOSITION DU JURY :

Président du jury :
Stanislaw PIESTRAK   Professeur des Universités, Université de Metz

Rapporteurs :
Bertrand GRANADO   Professeur des Universités, ENSEA
Lionel TORRES   Professeur des Universités, Université de Montpellier 2

Examinateurs :
Loïc LAGADEC   Maître de Conférences, Université de Bretagne Occidentale
Sébastien PILLEMENT   Maître de Conférences, Université de Rennes 1
Olivier SENTIEYS   Professeur des Universités, Université de Rennes 1



Remerciements

Je remercie tout particulièrement Monsieur Olivier Sentieys, Professeur des Universités à l'ENSSAT et responsable de l'équipe CAIRN, pour m'avoir accueilli dans son équipe de recherche et pour avoir accepté de diriger cette thèse. Je tiens à remercier également Monsieur Sébastien Pillement, Maîtres de Conférences à l'Université de Rennes pour avoir accepté le co-encadrement de cette thèse. Qu'ils soient assurés de ma gratitude.

Je tiens également à remercier Monsieur Stanislaw Piestrak pour avoir accepté de présider le jury de soutenance, Messieurs Bertrand Granado et Lionel Torres pour avoir accepté de rapporter sur ce manuscrit et enfin Monsieur Loïc Lagadec pour avoir accepté d'examiner ces travaux.

Que tous les membres de l'équipe CAIRN et le personnel de l'ENSSAT acceptent mes remerciements pour avoir contribué à rendre cette expérience enrichissante aussi bien sur le plan professionnel que personnel.

Je remercie également l'ensemble du personnel du département *Mesures Physiques* et de l'IUT de Lannion pour leur accueil pendant ces trois années de monitorat.

Je tiens également à remercier Jürgen Teich, Frank Hannig, Dmitri Kissler et Alexey Kupriyanov de l'Université de Erlangen en Allemagne pour avoir contribué à l'évolution de ce projet.

Merci encore à Auguste Le Berre pour avoir accepté de relire l'intégralité de ce manuscrit.

Enfin, mes derniers remerciements, mais non des moindres, à Agnes, Lucien et Moritz pour avoir su me supporter durant ces longs mois, je vous dédie l'ensemble de ce travail.

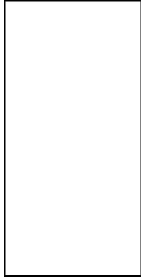


Table des matières

Introduction **1**

Plan du mémoire 4

I Etat de l'art **7**

A Introduction 8

B Historique 8

B.1 Histoire, préhistoire et ancêtres des architectures reconfigurables dynamiquement 8

B.2 Architectures configurables 10

B.3 Architectures reconfigurables 10

B.4 Vers des architectures reconfigurables dynamiquement 11

C Exploration sur la dynamicité de la reconfiguration 12

C.1 Méthodes d'implémentations algorithmiques 12

C.1-1 Implémentation temporelle/logicielle 12

C.1-2 Implémentation spatiale/matérielle 13

C.1-3 Implémentation spatio-temporelle 13

C.1-4 Implémentation et impact sur les caractéristiques F-D-P . . . 14

C.1-5 Synthèse 14

C.2 Notion de granularité dans la dynamicité de la reconfiguration 15

C.2-1 Granularité des motifs de calcul des architectures reconfigurables dynamiquement 15

C.2-2 Impact de la granularité des motifs de calcul 15

C.3 Impact des réseaux d'interconnexion sur la dynamicité de la reconfiguration . . . 16
C.3-1 Réseaux d'interconnexion globaux . . . 16
C.3-2 Réseaux d'interconnexion point-à-point . . . 16
C.3-3 Réseaux d'interconnexion hiérarchiques . . . 17
C.3-4 Impact de la flexibilité d'interconnexion sur la reconfiguration dynamique . . . 18
C.4 Mise en œuvre de la reconfiguration dynamique . . . 18
C.4-1 Mécanismes d'accélération de la reconfiguration dynamique . . . 18
C.4-2 Mécanismes de gestion de la préemption . . . 21
D Architectures reconfigurables dynamiquement . . . 21
D.1 Architectures mono-contexte grain fin . . . 22
D.1-1 ATMEL AT40K . . . 22
D.1-2 Famille Virtex de XILINX . . . 23
D.2 Architectures mono-contexte grain épais . . . 25
D.2-1 DART . . . 25
D.2-2 Systolic Ring . . . 26
D.2-3 WPPA : Weakly Programmable Processor Array . . . 26
D.3 Architectures multi-contexte grain fin . . . 27
D.3-1 DPGA : Dynamically Programmable Gate . . . 28
D.3-2 LATTICE ispXPGA . . . 29
D.3-3 Piperench . . . 29
D.3-4 PicoGa de XiRisc . . . 30
D.4 Architectures multi-contexte grain épais . . . 32
D.4-1 XPP : *eXtreme Processing Platform* . . . 32
D.4-2 ADRES : *Architecture for Dynamically Reconfigurable Embedded Systems* . . . 34
D.4-3 NEC DRP : *NEC Dynamically Reconfigurable Processor* . . . 34
E Plate-formes de développement . . . 35
E.1 Plate-formes dédiées au développement d'architectures grain épais . . . 36
E.1-1 Dresc : *Dynamically Reconfigurable Embedded System Compiler* 36
E.1-2 ArchitectureComposer . . . 37
E.2 Plate-formes dédiées au développement d'architectures grain fin . . . 38
E.2-1 Madeo . . . 38
E.2-2 AMDREL . . . 38
F Synthèse . . . 39

II Modèle Générique de Reconfiguration Dynamique **41**
A Introduction 42
A.1 Présentation globale des concepts de reconfiguration dynamique mis en œuvre par MOZAÏC 42
A.2 Paramètres de connectivité 43
A.3 Paramètres des domaines et ressources de reconfiguration 44
B Concept DUCK 45
B.1 Objectifs 45
B.2 Reconfiguration dynamique et mécanismes de préemption 46
B.3 Homogénéisation des processus de reconfiguration dynamique 47
C DyRIBox : unités d'interconnexion 53
C.1 Routeurs d'interconnexion dans les architectures reconfigurables 53
C.2 Architecture d'une DyRIBox 55
C.3 Principe de reconfiguration et principe de contrôle des multiplexeurs . 56
C.4 De la commutation directe de circuit vers la commutation par paquet . 56
D DyRIOBloc : ressources d'entrée/sortie reconfigurables 58
E Ressources de contrôle et de gestion 58
E.1 *CtxtScheduler* : gestion et ordonnancement des contextes 59
E.1-1 Gestion séquentielle des tâches 59
E.1-2 *IRCtrl* : gestion des interruptions sans priorité 60
E.1-3 Gestion des interruptions avec priorité 61
E.2 *DomainCtrl* : contrôleur de domaine 61
E.2-1 Gestion de la préemption 62
E.2-2 Gestion de la reconfiguration partielle 64
E.3 *CtxtMemCtrl* : Contrôleur de mémoire de configuration 64
E.3-1 Contrôle individuel des domaines 64
E.3-2 Contrôle partagé des domaines 65
E.3-3 Espace mémoire de configuration et format de la trame de configuration 66
F Synthèse 68

III xMAML : ADL pour le reconfigurable dynamiquement **69**
A Introduction 70
A.1 Langages de description structurels 71
A.2 Langages de description comportementaux 71
A.3 Langages de description mixtes 72
B MAML : Présentation 73

B.1 Description au niveau haut : *Processor Array level* 74
B.2 Description d'un élément processeur : *PE level* 74
B.3 Synthèse 76
C xMAML 76
C.1 Présentation générale d'une description xMAML 76
C.2 Modèle xMAML du bloc d'interconnexion DyRIBox 77
C.2-1 Paramètre d'identification 78
C.2-2 Paramètres de reconfiguration dynamique 78
C.2-3 Paramètres structurels 78
C.2-4 Exemple de description d'une DyRIBox en xMAML 79
C.3 Notion de domaine de reconfiguration 80
C.3-1 Introduction générale 80
C.3-2 Description et formalisation des Domaines 81
C.4 Unités de traitement 83
C.5 DyRIOBlocs : blocs d'entrée-sortie 86
C.6 Synthèse 86
D Exploration des paramètres de reconfiguration dynamique 87
D.1 Exploration en surface et consommation des ressources de reconfiguration dynamique 88
D.1-1 Exploration de la ressource DUCK 88
D.1-2 Exploration de la DyRIBox 90
D.2 Données de configuration des interconnexions 91
D.3 Données de configuration des unités de traitement 92
D.3-1 Temps de propagation de contexte 93
D.3-2 Temps de préemption de contexte 93
D.3-3 Détermination du nombre de domaine de reconfiguration . . 93
E Synthèse 93

IV Validation de la plate-forme Mozaïc **95**
A Chaîne de codage et de décodage WCDMA 96
A.1 Emetteur WCDMA 96
A.2 Récepteur WCDMA 97
A.2-1 Réception et échantillonage 98
A.2-2 Filtre de réception 98
A.2-3 Estimation des trajets 98
A.2-4 Décodage des données, estimation de l'amplitude complexe du canal et synchronisation 98

B Implémentation d'un décodeur WCDMA sur FPGA embarqué 99
B.1 Présentation du XC4000 100
B.2 Outils génériques de développement sur FPGA 102
B.2-1 Synthèse logique de l'application : ABC Berkeley 102
B.2-2 Placement et routage : VPR Toronto 103
B.2-3 Génération d'un flot de données de configuration Mozaïc : . . 104
B.3 Adéquation WCDMA-FPGA 106
B.3-1 Synthèse des fonctions du WCDMA sur eFPGA 106
B.3-2 Analyse des performances requises par l'application WCDMA 106
B.4 Description xMAML du eFPGA 107
B.4-1 Description des DyRIBox 108
B.4-2 Description des CLB 108
B.5 Génération des mécanismes de reconfiguration dynamique pour eFPGA 110
B.5-1 Estimation de la taille des données de configurations et des performances de reconfiguration dynamique 110
B.5-2 Estimation du nombre de domaines de reconfiguration 112
B.5-3 Impact de la reconfiguration dynamique sur les caractéristiques de l'architecture 113
C Implémentation d'un décodeur WCDMA sur DART 114
C.1 Présentation de DART et des outils associés 115
C.1-1 DART : processeur reconfigurable dynamiquement 115
C.1-2 Outils dédiés à la compilation sur DART 117
C.2 Adéquation WCDMA-DART 119
C.2-1 Implémentation du filtre FIR 120
C.2-2 Implémentation du décodage des données et combinaison des multi-trajets 121
C.2-3 Synchronisation 122
C.2-4 Estimation de canal 123
C.2-5 Synthèse 124
C.3 Description xMAML de DART 124
C.3-1 Description des DyRIBox 124
C.3-2 Description des DPR 126
C.4 Génération des mécanismes de reconfiguration dynamique pour DART 129
C.4-1 Estimation de la taille des configurations 129
C.4-2 Estimation du nombre de domaines de reconfiguration 131
C.4-3 Estimation de la consommation et de la surface de silicium induits par l'utilisation du concept DUCK 131

C.4-4 Conclusion 132
D Synthèse 133

Conclusions et perspectives **135**

Bibliographie **139**

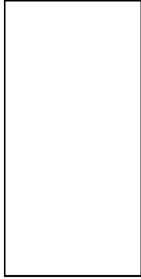
Liste des figures **148**

Liste des tableaux **149**

Listings **151**

Glossaire **153**

Résumé **158**



Introduction

Contexte de l'étude

Que serait la vie quotidienne de chacun d'entre nous, si en 1951, le premier transistor à jonction n'avait pas été conçu ? Le transistor est, par analogie, comparable à une brique de Légo qui, par un assemblage méticuleux et ordonné, permet le traitement de toutes les fonctions logiques. Le procédé d'intégration d'un transistor sur une puce de silicium est le fruit de plusieurs étapes successives complexes, à l'aide de masques. La superposition des couches de différentes natures (substrat, oxyde, métal, polysilicium), de différentes surfaces et de différentes épaisseurs permet la conception physique d'un transistor. La manière dont l'organisation des transistors et leurs interconnexions sont réalisées définissent l'architecture du circuit.

Les architectures spécifiquement conçues pour l'exécution d'une seule application sont appelées ASIC (*Application Specific Integrated Circuit*). Le budget consacré au développement d'une telle architecture est souvent conséquent car la réalisation des masques de production est un processus long et coûteux. Cependant, ce coût tend à diminuer avec le nombre de circuits produits. De plus, ces architectures sont réputées pour leur rapidité d'exécution et leur faible consommation d'énergie.

Si l'on se place à un niveau d'abstraction supérieur, il est possible de concevoir des fonctions flexibles par l'organisation adaptée des transistors, bien que ceux-ci soient figés dans le silicium. La modification de la fonctionnalité est réalisée par configuration. L'UAL (Unité Arithmétique et Logique) d'un micro-processeur est un exemple d'architecture flexible. Selon les besoins de l'application, il est possible de procéder à sa reconfiguration aussi bien pour le traitement d'une addition ou d'une multiplication. Cette flexibilité, intéressante pour les opérations de calculs de nos ordinateurs et autres assistants personnels ou téléphones multimédia, se fait à un coût non négligeable de surface de silicium, de temps d'exécution et de consommation. En revanche, le coût de développement d'une application sur microprocesseur reste relativement modeste comparé au budget de développement nécessaire à la conception

d'un ASIC.

Nous constatons qu'entre ces deux solutions existe un vide architectural qui a été en partie comblé par l'introduction du concept FPGA (*Field Programmable Gate Array*). L'architecture FPGA permet le prototypage rapide d'une application par l'implémentation d'une matrice d'unités de traitement configurables. Ces unités de traitement sont interconnectées par l'intermédiaire de réseaux flexibles et également configurables, de manière à constituer des chemins de traitement de données spécifiques à l'implémentation d'une application. Cependant, le nombre de ressources limité ainsi que les temps de configuration et de reconfiguration trop importants rendent leur utilisation plus difficile pour l'implémentation d'applications nécessitant beaucoup de ressources.

La reconfiguration dynamique se caractérise par des processus de reconfiguration suffisamment rapides pour permettre la réutilisation des unités de traitement au cours de l'exécution d'un algorithme pour différentes opérations. En permettant l'adaptation des chemins de données et des unités de traitement suffisamment rapidement à l'application désirée, le vide architectural présent entre les ASIC et les microprocesseurs tend à se combler.

Le travail présenté dans ce document se situe à l'intersection de deux axes de recherche :

1. les architectures reconfigurables dynamiquement,
2. la conception automatisée de systèmes électroniques, ou encore appelée EDA (*Electronic Design Automation*).

Problématique de l'étude

Dans le domaine de la conception d'architectures, il existe des outils destinés à assister les ingénieurs dans les choix d'implémentations effectués tout au long de la chaîne de développement. Pour le domaine dans lequel nous nous situons dans ce travail de recherche, la modification de certains paramètres influe directement sur trois critères fondamentaux (F-D-P) des architectures reconfigurables dynamiquement :

1. la flexibilité,
2. la dynamicité,
3. et la performance.

La flexibilité d'une architecture a été évoquée dans la première partie de cette introduction, il s'agit de la capacité d'une architecture à s'adapter aux applications nouvelles. La dynamicité caractérise la rapidité à laquelle une architecture pourra s'adapter à ces applications. Et enfin la performance caractérise la rapidité d'exécution des traitements qui pourront être réalisés par une architecture.

Ces trois critères interagissent l'un sur l'autre selon le degré d'importance que l'on donne à chacun d'eux. Plus une architecture est flexible, plus elle aura besoin de temps pour se reconfigurer et moins elle sera efficace pour le traitement des données. À l'inverse, plus une architecture est rapide à se reconfigurer, plus elle est efficace dans le traitement des données moins elle est flexible.

La problématique apparaît alors ici clairement. La conception d'une architecture reconfigurable fait l'objet d'un compromis entre ces trois critères dont l'interaction a été schématisées par la figure 0-1. Selon l'application ou le domaine d'application visé, il sera nécessaire de procéder à des choix d'implémentation qui devront se justifier par rapport aux performances (surface, consommation, rapidité) de l'architecture désirée.

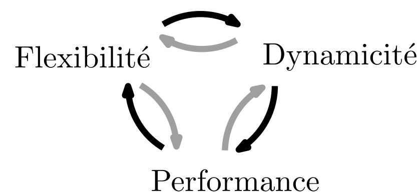


Figure 0-1 – **Interaction entre les critères de flexibilité, de dynamicité de la reconfiguration et de performance (F-D-P) d'une architecture reconfigurable dynamiquement**. *La conception d'un circuit est le fruit permanent de compromis entre ces trois paramètres dont la modification de l'un interagit directement sur les deux autres.*

CONTRIBUTIONS

De notre point de vue, il est nécessaire de proposer des outils d'aide à la conception d'architectures reconfigurables dynamiquement de manière à pouvoir dès le début de la conception, estimer le degré d'interaction entre les trois critères F-D-P. Pour cela nous proposons MOZAÏC, une plate-forme générique de modélisation et de conception d'architectures reconfigurables dynamiquement.

Une plate-forme de développement intègre des outils permettant d'accélérer et de simplifier les processus de conception. Des plate-formes de conception d'architectures reconfigurables existent. Pour la majorité de ces plate-formes, elles se focalisent sur les spécifications des unités de traitement au détriment des caractéristiques de reconfiguration dynamique. Ajoutons que la spécification d'une unité de traitement à haut niveau implique l'abstraction de certains critères générés automatiquement par la plate-forme, ce qui implique un contrôle restreint des performances de traitement pendant la conception de l'architecture.

MOZAÏC est une plate-forme de développement dont les critères de reconfiguration dynamique sont prédominants bien que le contrôle des performances de traitement soit total. En effet, cette plate-forme est basée sur l'intégration d'unités de traitement développés au préalable. L'intérêt de MOZAÏC est de s'intégrer aux plate-formes existantes afin d'apporter ses avantages en matière de gestion de la reconfiguration. Pour cela, nous avons développé des concepts flexibles et paramétrables pour la reconfiguration dynamique. La reconfiguration dynamique mise en œuvre par MOZAÏC intègre des concepts permettant la gestion flexible des unités de traitement implémentées. La figure 0-2 présente l'intégration de MOZAÏC avec une plate-forme de développement hôte. La plate-forme hôte est utilisée afin de générer les paramètres de configuration nécessaires à l'implémentation des applications choisies. MOZAÏC permet la spécification de l'architecture et des mécanismes de reconfiguration devant être implémentés afin de procéder à l'exploration des critères F-D-P.

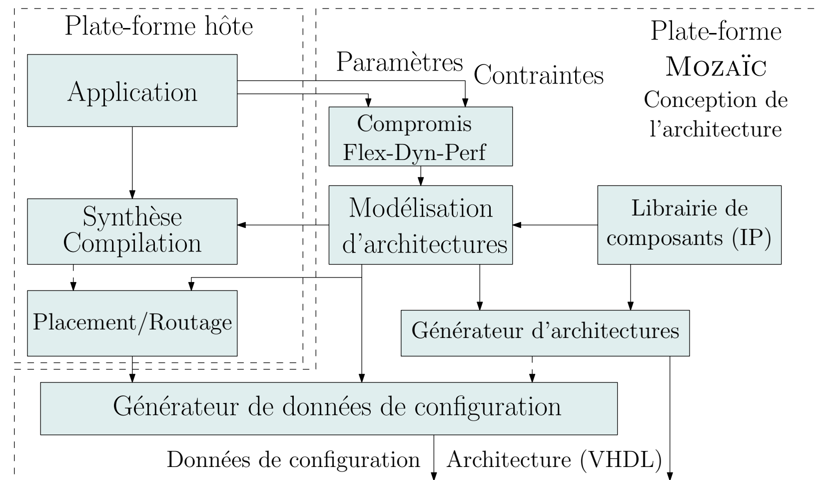


Figure 0-2 – **Intégration de la plate-forme Mozaïc avec une plate-forme de conception hôte**. Mozaïc *permet la spécification des processus de reconfiguration afin de concevoir une architecture adaptées à l'application visée en termes de critères F-D-P. Les unités de traitement propres à la plate-forme de développement hôte sont décrites en termes de reconfiguration dynamique afin de générer les ressources de reconfiguration nécessaires.*

Plan du mémoire

Le premier chapitre de ce document est consacré au concept de reconfiguration dynamique. Tout d'abord, nous présentons les concepts génériques propres à ce mode de reconfiguration ainsi que les différents paramètres qui impactent directement la dynamicité d'une architecture. Nous présentons ensuite quelques architectures reconfigurables, dont chacune présente un intérêt remarquable se devant d'être cité dans ce document. Enfin, nous concluons ce chapitre avec la présentation de quelques plate-formes de développement dédiées à la conception d'architectures reconfigurables dynamiquement.

Le chapitre II présente le modèle générique de reconfiguration dynamique mis en œuvre par la plate-forme de développement Mozaïc. Nous allons détailler les différents mécanismes que nous avons développés pour garantir une reconfiguration suffisamment dynamique et efficace.

Le chapitre III présente le langage de description d'architecture à haut niveau xMAML, dédié à l'intégration sur puce et à la gestion cohérente des unités de traitement reconfigurables dynamiquement. Ce langage permet de procéder rapidement à une spécification des processus de reconfiguration des unités de traitement à implémenter de façon à générer automatiquement les ressources adaptées qui seront nécessaires à la mise en œuvre de la reconfiguration dynamique. xMAML est le fruit direct de la collaboration que nous avons eu avec l'équipe de recherche *Hardware-Software-Co-Design* de l'université d'*Erlangen-Nürnberg* à l'origine du langage de description d'architectures massivement parallèles, MAML.

Le chapitre IV est consacré à la mise en œuvre de Mozaïc, plate-forme générique de modélisation et de conception d'architectures reconfigurables dynamiquement. Dans ce chapitre, nous montrons la généricité de notre méthode par l'exploration de la dynamicité d'une implémentation de décodeur WCDMA (*Wideband Code Division Multiple Access*) sur une cible de processeur reconfigurable, DART, et sur un FPGA embarqué. Nous avons montré la com-

plémentarité de MOZAÏC avec les outils spécifiquement dédiés à ces deux architectures.

Enfin, le dernier chapitre conclut ce document en résumant les apports de notre méthodologie et les perspectives qui s'en dégagent.

Chapitre

1

Etat de l'art

Résumé : De notre point de vue, l'histoire des architectures reconfigurables dynamiquement puise ses origines depuis plus longtemps que l'on pourrait l'imaginer. Il serait trop long de développer tous les évènements qui ont un rapport de près ou de loin avec une architecture reconfigurable dynamiquement, cependant l'évolution des techniques et méthodes qui ont permis d'y aboutir nous semble suffisamment intéressante pour en évoquer quelques dates importantes. Après une introduction, nous présenterons quelques unes des architectures reconfigurables dynamiquement que nous avons choisies de caractériser essentiellement selon leur granularité et leur méthode de reconfiguration. Enfin, nous présenterons les outils permettant le développement d'une architecture, de la description de celle-ci jusqu'aux outils de compilation ou de synthèse dédiés à la reconfiguration dynamique.

Sommaire

A Introduction . . . 8
B Historique . . . 8
B.1 Histoire, préhistoire et ancêtres des architectures reconfigurables dynamiquement . . . 8
B.2 Architectures configurables . . . 10
B.3 Architectures reconfigurables . . . 10
B.4 Vers des architectures reconfigurables dynamiquement . . . 11
C Exploration sur la dynamicité de la reconfiguration . . . 12
C.1 Méthodes d'implémentations algorithmiques . . . 12
C.2 Notion de granularité dans la dynamicité de la reconfiguration . . . 15
C.3 Impact des réseaux d'interconnexion sur la dynamicité de la reconfiguration . . . 16
C.4 Mise en œuvre de la reconfiguration dynamique . . . 18
D Architectures reconfigurables dynamiquement . . . 21
D.1 Architectures mono-contexte grain fin . . . 22
D.2 Architectures mono-contexte grain épais . . . 25
D.3 Architectures multi-contexte grain fin . . . 27
D.4 Architectures multi-contexte grain épais . . . 32
E Plate-formes de développement . . . 35
E.1 Plate-formes dédiées au développement d'architectures grain épais . 36
E.2 Plate-formes dédiées au développement d'architectures grain fin . . . 38
F Synthèse . . . 39

A Introduction

Dans ce chapitre, nous nous attacherons à présenter quelques architectures, pour la plupart reconfigurables dynamiquement, qui, par certains aspects, ont attiré notre attention. Ce chapitre ne se veut pas exhaustif sur le spectre des architectures présentées, mais permettra de mettre l'accent sur les mécanismes qui constituent les éléments essentiels dans la reconfiguration dynamique. Nous nous attacherons également à explorer ces mécanismes en montrant dans quelle mesure ceux-ci influent sur les performances de reconfiguration dynamique. De manière à comprendre le cheminement qui a permis la conception d'architectures reconfigurables dynamiquement, la première partie de ce chapitre est dédiée à la présentation de l'histoire des machines à calculer.

B Historique

B.1 Histoire, préhistoire et ancêtres des architectures reconfigurables dynamiquement

Depuis l'antiquité, les hommes ont cherché à utiliser différentes techniques afin de faciliter leurs calculs. La première de ces techniques consiste tout simplement à symboliser le calcul sur un support d'écriture. Les historiens considèrent l'apparition du premier abaque [1] à l'ère babylonienne (2400 av. JC). Cet outil constitue le premier support spécifiquement destiné à la réalisation de calculs simples. S'en suivirent alors les abaques à boules, ou bouliers dont on ignore si ils apparurent d'abord en Inde, en Mésopotamie ou en Ègypte. Il est étonnant de penser qu'aujourd'hui encore, ces outils soient toujours utilisés en Chine ou au Japon. Ce n'est que beaucoup plus tard, en 1623, que William Schickard (1592-1635) inventa la première machine à calculer mécanique appelée "horloge à calcul". Son but était de faciliter la multiplication de grands nombres. S'en suivit peu de temps après, en 1642, la Pascaline, machine arithmétique créé par Blaise Pascal (1623-1662). Il s'agit d'une machine capable d'effectuer des additions et soustractions, destinée à aider son père, percepteur des taxes. En 1673, Gottfried Wilhelm von Leibniz (1646-1716) ajouta à la Pascaline la multiplication et la division.

En 1834, Charles Babbage (1792-1871) conçoit le prototype d'une machine analytique avec mémoire, qui permet d'évaluer des fonctions. Apprenant qu'une machine à tisser [2] est programmée à l'aide de cartes perforées, il se lance alors dans la construction d'une machine à calculer basée sur ce principe. Il avait auparavant défini les différentes unités qui allaient composer sa machine. Tout d'abord, une unité d'entrée qui permet de communiquer le traitement à effectuer à la machine, une mémoire qui permet de stocker les données et les résultats intermédiaires, une unité de commande qui permet de contrôler l'exécution du traitement, une unité arithmétique et logique qui permet le traitement des calculs et, enfin, une unité de

1. La définition la plus convaincante provient du dictionnaire d'Emile Littré, sa définition est la suivante. *Abaque : Terme d'antiquité. Tableau couvert de poussière, sur lequel on traçait des nombres et on enseignait le calcul.* On peut également ajouter que *abaque* vient du grec *abax* qui veut dire table de comptage recouverte de poussière.

2. métier à tisser de Jacquard

sortie qui permet de lire le résultat. Bien que sa machine n'eut jamais l'occasion de fonctionner, limitée par les compétences mécaniques de l'époque, Babbage avait pourtant défini les bases élémentaires sur lesquelles reposent aujourd'hui encore nos calculateurs modernes.

L'année 1937 marque une nouvelle étape importante. Georges Stibitz construit le premier additionneur binaire basé sur le célèbre algèbre de Boole. C'est ce modèle qui servira de référence au développement d'un calculateur à relais, le "*Complex Number Calculator*". Celui-ci était composé de 450 relais électromécaniques capablent de multiplier des opérandes codées en BCD (Décimal Codé Binaire) en une minute. Pourtant, ce n'est qu'en 1944 que le premier véritable calculateur, le Mark One, va fonctionner de manière autonome à partir d'un programme comme cela avait été l'objectif de Babbage [1]. Le programme était stocké sur une bande de papier perforé, le déroulement des opérations était cadencé par une horloge. Il était capable d'effectuer des opérations sur des opérandes de vingt-trois digits. Une addition prenait trois cents millisecondes de calcul, une multiplication six secondes, et enfin une division un peu plus de onze secondes. L'aspect le plus intéressant de cette machine vient de son architecture. Il s'agit, en effet, de la première machine dont l'architecture est de type Harvard, du nom de l'université où elle a été développée, qui consiste en la séparation physique de la mémoire de données et de la mémoire programme. Cette architecture est aujourd'hui encore à la base de nombreux DSP (*Digital Signal Processor*) et microcontrôleurs.

L'ère des calculateurs électroniques débute en 1946 avec la conception de l'ENIAC (*Electronic Numerical Integrator and Computer*). Composé de plus de dix-sept mille tubes à vide, ses performances sont bien supérieures au Mark One (environ cinq cents fois). Cependant, il occupe une surface de 150 m^2 et pèse plus de trente tonnes. Son principal défaut réside dans sa programmation. Réalisée en modifiant un tableau de connexions, fiche par fiche, la programmation nécessite plusieurs journées pour le reprogrammer. Cette contrainte sera vite dépasée en 1949 [2] par l'EDVAC (*Electronic Discrete Variable Automatic Computer*) dont le projet avait été initié en collaboration avec John Presper Eckert (1919-1995) et John W. Mauchly (1907-1980) créateurs de l'ENIAC, et John Von Neumann (1903-1957) avant que l'ENIAC ne soit opérationnel. En effet, conscients de la limite de leur première machine, ceux-ci avaient déjà sollicité les compétences du mathématicien Von Neumann. Le concept de l'architecture éponyme, qui consiste à partager une seule et même mémoire pour le programme et les données, venait de naître.

Enfin, dernière étape importante dans l'histoire des architectures, le 15 novembre 1971 sorti le premier processeur sur puce. Le 4004 conçu par Intel contenait 2300 transistors et était capable de travailler à une fréquence de 740 kHz sur des mots de quatre bits grâce à un jeu de 46 instructions. Seize registres de données étaient présents, permettant de stocker 1280 demi-octets de données.

1. En 1889, Hermann Hollerith (1860-1929) avait construit une machine à l'aide de cartes perforées mais ne supportaient pas l'exécution universelle de calculs.

2. L'EDVAC est livré au Laboratoire en recherche balistique en août 1949 après qu'un certain nombre de problèmes aient été identifiés et résolus. Il n'est mis en service qu'en 1951 en raison d'un conflit sur le brevet entre l'University of Pennsylvania et Eckert & Mauchly (qui entre-temps ont créé leur propre société, la Eckert-Mauchly Computer Corporation). *sources Wikipédia*

B.2 Architectures configurables

La partie précédente s'est attachée à présenter brièvement un historique des machines à calculer d'une manière très générale. En ce qui concerne le sujet qui nous importe dans ce document, il convient de présenter quelques circuits plus proches en termes d'architectures des circuits reconfigurables dynamiquement, de manière à suivre l'évolution des solutions innovantes apportées ces trente dernières années. Lorsque l'on parle de reconfiguration dynamique, on évoque à l'origine le terme de configuration d'une architecture afin d'adapter celle-ci à un algorithme précis. La première d'entre elle est sans aucun doute la PAL (*Programmable Array Logic*) présentée pour la première fois en 1978.

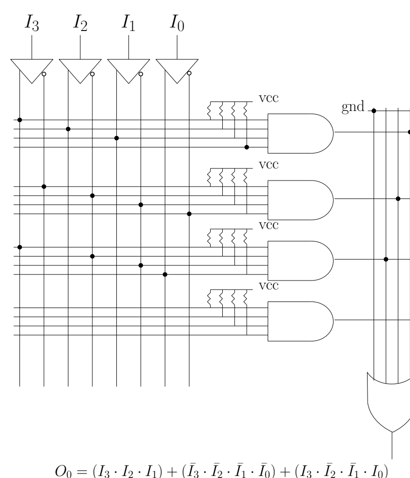


Figure 1-1 – **Implémentation de l'équation logique** $O_0 = (I_3 \cdot I_2 \cdot I_1) + (\bar{I_3} \cdot \bar{I_2} \cdot \bar{I_1} \cdot \bar{I_0}) + (I_3 \cdot \bar{I_2} \cdot \bar{I_1} \cdot I_0)$ **sur une PAL**. *L'entrée I_0 non connectée sur la première porte "ET" est connectée à vcc, aucune entrée de la dernière porte "ET" n'est connectée, celle-ci n'étant pas utilisée.*

La technologie PAL repose sur le principe que toute équation logique peut s'exprimer à l'aide de fonctions "ET" et "OU". Une PAL implémente une matrice de portes logiques "ET" et "OU" pouvant être interconnectées de manière flexible mais définitive. La connexion d'une entrée sur une porte logique se fait par l'intermédiaire d'un fusible. Au cas où ni une entrée ni son inverse n'est connectée à la porte "ET", une entrée de rappel connectée à un état logique haut permet de la rendre inactive. Dans l'exemple figure 1-1, l'équation $O_0 = (I_3 \cdot I_2 \cdot I_1) + (\bar{I_3} \cdot \bar{I_2} \cdot \bar{I_1} \cdot \bar{I_0}) + (I_3 \cdot \bar{I_2} \cdot \bar{I_1} \cdot I_0)$ est implémentée. Pour cela, des connexions appropriées ont été réalisées entre les différentes entrées, les portes "ET" et "OU". Une fois la PAL configurée il n'est plus possible de procéder à une modification des connexions effectuées, ce qui constitue le principal inconvénient de cette technologie.

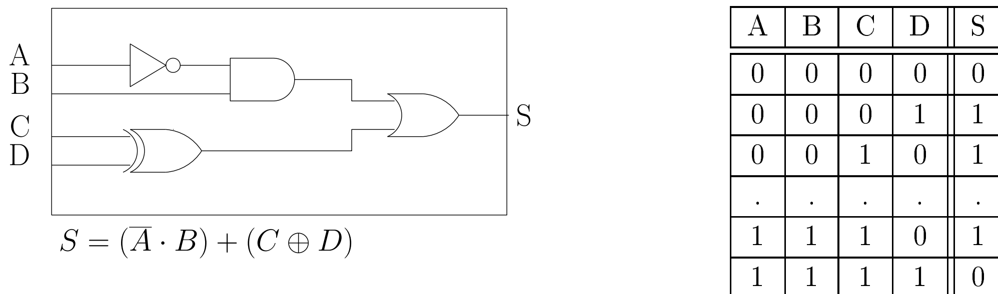
B.3 Architectures reconfigurables

La modification du schéma de connexion d'une architecture PAL étant impossible, de nouvelles technologies se sont développées de manière à palier à cette contrainte. Ainsi, la technologie GAL (*Generic Array Logic*) développée par LATTICE est apparue en 1985. Celle-ci repose sur le même principe que les PALs à ceci près que les connexions sont programmables

à volonté. Le succès de cette technologie ne fut qu'éphémère puisque la même année, Xilinx présente un nouveau type d'architecture appelée FPGA. Le principe de fonctionnement d'un FPGA se différencie complètement des technologies PAL et GAL. Les FPGA sont réalisés par l'assemblage de trois fonctions logiques de base : des LUT (*Look Up Table*), des multiplexeurs et des bascules.

Une LUT est une table à n entrées où sont stockées les 2^n combinaisons possibles de solutions d'une équation à n variables. La solution à l'équation est sélectionnée en fonction des n entrées (figure 1-2). Un réseau de LUT forme ainsi un circuit combinatoire qu'il est possible de rendre synchrone en implémentant des bascules D en sortie des LUT.

Afin de faire communiquer les LUT de la manière la plus flexible possible, celles-ci sont interconnectées par l'intermédiaire de ressources appelées multiplexeurs. Un multiplexeur est une ressource de connexion composée de n entrées pouvant toutes être connectées vers une seule sortie. L'entrée devant être connectée sur la sortie est déterminée par un mot de configuration dont la taille $t = \lceil log_2(n) \rceil$.



A	B	C	D	S
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
.	.	.	.	.
1	1	1	0	1
1	1	1	1	0

Figure 1-2 – **Fonctionnement d'une LUT**. *Afin de réaliser la fonction logique* $S = (\overline{A} \cdot B) + (C \oplus D)$*, un tableau à quatre entrées (A, B, C, D) représentant la table de vérité de l'équation est complété comme le serait une mémoire de sorte que l'adressage se fait par les entrées l'équation.*

B.4 Vers des architectures reconfigurables dynamiquement

Le développement des FPGA a ouvert de nouveaux horizons dans le domaine de la conception d'architectures. En effet, jusqu'alors, seul le développement d'une architecture ASIC permettait l'implémentation d'algorithmes performants aux prix de périodes de développement longues et coûteuses. De plus, l'évolution, même minime, de l'algorithme implémenté conduisait à la mise à l'index du composant ASIC. De part sa reconfigurabilité, les chercheurs et concepteurs ont vu dans la technologie FPGA un moyen de réunir sur une même technologie performance (des ASIC) et flexibilité (des processeurs). Un processeur dispose d'une mémoire programme qui lui permet constamment de procéder à une modification de sa fonctionnalité contrairement à un FPGA qui, pour changer de fonction doit procéder à un arrêt complet des tâches avant de pouvoir procéder à sa reconfiguration. Vers la fin des années 90, ce constat a amené les sociétés ATMEL et Xilinx à concevoir des architectures [1, 2] capables de procéder à une modification d'une partie de la fonctionnalité de leurs ressources pendant le traitement des calculs, donnant alors naissance aux premières architectures reconfigurables

partiellement.

C Exploration sur la dynamicité de la reconfiguration

Le temps mis par une architecture à se reconfigurer va déterminer sa capacité à s'adapter de manière dynamique à son environnement. Selon si la reconfiguration dure quelques microsecondes ou quelques nanosecondes, une architecture sera qualifiable de dynamique ou non. Trois caractéristiques propres à la reconfiguration dynamique peuvent être envisagées lors de la conception d'un circuit. Ces trois caractéristiques sont la flexibilité, la dynamicité et la performance. Cependant, la modification d'un de ces paramètres modifiera en conséquence les deux autres. Plus une architecture est flexible, plus elle aura besoin de temps pour se reconfigurer et moins elle sera efficace pour le traitement des données. À l'inverse, plus une architecture est rapide à se reconfigurer, plus elle est efficace dans le traitement des données moins elle est flexible. Il faudra par conséquent prendre garde à obtenir le meilleur rapport flexibilité-dynamicité-performance (F-D-P) lors de la conception d'un circuit. Cette section va s'attacher à montrer la manière dont les paramètres de flexibilité et de performance interagissent sur le paramètre de dynamicité.

C.1 Caractérisation des méthodes d'implémentations algorithmiques sur architectures et systèmes numériques

La manière dont un algorithme sera implémenté sur une architecture reconfigurable dynamiquement va déterminer d'une part le niveau de performance dans le traitement des données et d'autre part le niveau de dynamicité dont cette architecture pourra faire preuve.

C.1-1 Implémentation temporelle/logicielle

Les architectures Von Neumann et Harvard ont été conçues, à l'origine, afin de répondre de manière flexible à toutes sortes d'algorithmes. L'exécution d'un algorithme se fait par étapes successives en fonction des différentes opérations à exécuter, déterminées au préalable à la compilation. Une seule ressource appelée UAL, configurée pour une opération donnée à un instant donné, est chargée d'effectuer ces opérations successives. Par exemple (figure 1-3 (a)) [3], la résolution de l'équation $y = A.x^2 + B.x + C$ implémentée selon le schéma de Horner, nécessite deux multiplications et deux additions. Les deux multiplications sont $(A) \times (x)$ et $(A \times x + B) \times (x)$. Les deux additions sont $(A \times x) + (B)$ et $(A \times x^2 + B \times x) + (C)$.

La résolution de toute autre équation revient à modifier l'ordonnancement et la nature de ces différentes phases de calculs. Ces modifications sont réalisées par programmation et stockées dans une mémoire que l'UAL viendra lire après chaque opération effectuée. Cette implémentation est appelée temporelle ou logicielle du fait de la nécessité de l'utilisation d'un programme pour l'utilisation de l'UAL et des ressources associées.

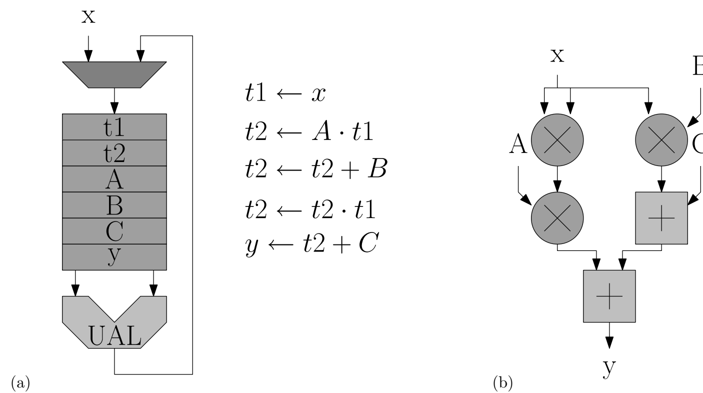


Figure 1-3 – **Exemple d'application exécutée par méthode d'implémentation temporelle et spatiale**. *Dans le cas d'une implémentation temporelle (a) de l'équation* $y = A.x^2 + B.x + C$*, une unité de traitement unique (UAL) est partagée dans le temps entre les différentes opérations. Dans le cas d'une implémentation spatiale (b) de l'équation* $y = A.x^2 + B.x + C$*, les opérations sont assignées à des opérateurs dupliqués en différents points de l'espace. L'équation est ainsi réalisée en parallèle.*

C.1-2 Implémentation spatiale/matérielle

Contrairement à l'implémentation temporelle, certaines architectures réalisent en même temps l'ensemble des opérations nécessaires à l'exécution d'un algorithme. Ainsi, les différentes opérations à effectuer ne sont plus réalisées de manière séquentielle, mais de manière parallèle ou spatiale (figure 1-3 (b)). Toutes les opérations sont réalisées en même temps et permettent ainsi une exécution plus rapide, grâce à l'économie des configurations de l'UAL. Cette solution est plus gourmande en surface à cause de l'implémentation d'opérateurs spécifiques pour chaque opération à effectuer. Ce type d'implémentation est également appelée matérielle du au fait que, pour chaque opération nécessaire, une ressource matérielle propre lui est dédiée.

C.1-3 Implémentation spatio-temporelle

La convergence d'une implémentation logicielle et d'une implémentation matérielle caractérise une forme d'implémentation que nous pouvons qualifier de reconfigurable dynamiquement. Imaginons de remplacer les opérateurs dédiés d'un système dont l'implémentation est spatiale par des opérateurs de type UAL, il devient alors possible de conserver la flexibilité des implémentations logicielles (grâce à l'utilisation des UAL), tout en acquérant des caractéristiques d'exécution aussi rapides que des systèmes dont l'implémentation est matérielle (de part la constitution d'un chemin de données). Cela se traduit par la possibilité d'utiliser ces UAL en parallèle, à condition, d'une part de pouvoir interconnecter ces différentes UAL par l'intermédiaire de multiplexeurs, comme cela est le cas pour les FPGA, et d'autre part que le chemin de données créé par les UAL soit configuré pour un certain temps d'exécution.

C.1-4 Implémentation et impact sur les caractéristiques F-D-P

Chacune de ces trois implémentations présente des avantages et des inconvénients. Les trois caractéristiques principales qui nous intéressent dans l'implémentation d'un algorithme sont souvent les performances en temps d'exécution, en consommation et en surface de silicium nécessaires à l'exécution de cet algorithme. À cela, nous pouvons ajouter la caractéristique de flexibilité en vue des capacités d'évolution de ces architectures. Tout d'abord, argument souvent porteur auprès du grand public, nous pouvons constater que les performances en temps d'exécution sont meilleures pour les implémentations matérielles que pour une implémentation logicielle (figure 1-4). En revanche, si l'on observe la surface de silicium nécessaire à l'exécution d'un même algorithme sur ces deux architectures différentes, nous pouvons constater qu'il est plus économique d'utiliser une implémentation logicielle. Les implémentations logicielles présentent de plus l'avantage d'être très flexibles de part leur capacité à pouvoir exécuter tout type d'algorithme contrairement à une implémentation matérielle spécifiquement dédiée à une application bien précise.

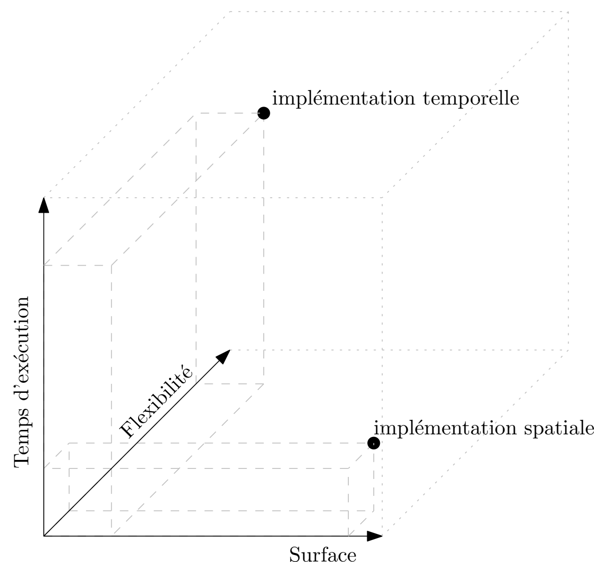


Figure 1-4 – **Caractéristiques des deux types d'implémentation temporelle et spatiale**. *Les systèmes dont les architectures sont dédiées à une implémentation temporelle présentent l'avantage d'occuper moins de surface de silicium au détriment de leur performances en temps et de la consommation nécessaire à l'exécution d'un algorithme donné. À l'inverse, les systèmes dont les architectures sont dédiées à une implémentation spatiale présentent l'avantage d'être performants en temps et en consommation au détriment d'une surface de silicium plus importante.*

C.1-5 Synthèse

La dynamicité de la reconfiguration est déterminée par la capacité d'une architecture à effectuer le changement d'une configuration vers une autre. Agir sur le type d'implémentation lors de la conception d'une architecture a pour conséquence de modifier le paramètre de dynamicité de la reconfiguration. Une implémentation logicielle ne requiert que très peu de données

de configuration qui seront rapidement transmises à la ressource concernée. Le passage d'une configuration à une autre sera alors fait rapidement. En revanche, de part le nombre de ressources à configurer dans une architecture ayant adopté une implémentation matérielle, les données de configurations seront plus nombreuses, ce qui aura pour effet de ralentir les processus de reconfiguration.

C.2 Notion de granularité dans la dynamicité de la reconfiguration

De la même manière que le choix qui est fait sur le type d'implémentation agit directement sur la dynamicité potentielle d'une architecture reconfigurable, la notion de granularité revêt également son importance sur ce critère. Cette partie s'attâche à montrer le niveau d'impact induit par les choix de granularité effectués par les concepteurs sur la reconfiguration d'une architecture.

C.2-1 Granularité des motifs de calcul des architectures reconfigurables dynamiquement

Dans le monde des architectures reconfigurables dynamiquement, nous pouvons recenser deux granularités de motifs de calcul. Les motifs de calcul qui ont la particularité de travailler au niveau bit sont appelés grain fin et les motifs de calculs travaillant sur des mots de plusieurs bits sont appelés grain épais. Un architecture grain fin permet une souplesse dans le spectre des applications réalisables contrairement aux architectures grain épais qui, en imposant une taille de données figées, limitent en conséquence le domaine des applications pouvant être implémentées efficacement. En revanche, une architecture grain fin nécessitera un flot de configuration plus important et donc des phases de configuration plus longues qu'une architecture grain épais.

C.2-2 Granularité des motifs de calcul et impact sur la reconfiguration dynamique

Avec les possibilités d'intégration qui s'offrent aujourd'hui aux concepteurs, il est envisageable de concevoir des architectures reconfigurables dynamiquement capables d'exécuter plusieurs algorithmes en parallèle de granularités éventuellement différentes. L'implémentation d'un algorithme sur une architecture FPGA à grain fin, nécessite un flot de données de configuration dont la taille n'est pas négligeable. Par exemple, la configuration d'une seule LUT à quatre entrées de FPGA nécessite seize bits de configuration auxquels s'ajoutent des bits de configuration des interconnexions ou encore de calculs de retenue pour ne citer que ceux-là. Si l'on considère le plus petit composant Virtex XCV50 de Xilinx [4], celui-ci est composé d'une grille de 16×24 CLB (Configurable Logic Block). Chaque CLB est composé de quatre LUT et de ressources arithmétiques de calcul de retenue et 559200 bits sont nécessaires à la reconfiguration complète du FPGA. On comprend alors qu'un tel choix d'implémentation devra se justifier. Un argument en faveur de cette technologie reste sans aucun doute la flexibilité de telles architectures. Le fait de décomposer chaque unité de traitement en bit permet un

assemblage de blocs logiques flexibles adaptés aux données qui devront être traitées par ces mêmes blocs. À l'inverse, dans le domaine des architectures à grain épais, les processeurs reconfigurables sont spécialisés pour un domaine d'application donné et ont par conséquent les ressources de traitement adaptés aux calculs spécifiques nécessaires à une application donnée. La reconfiguration de telles ressources ne concerne que la fonctionnalité d'une unité de traitement qui, bien souvent, se limite à quelques opérations spécifiques. Dès lors, seuls quelques bits sont utiles à la reconfiguration dynamique. Ces phases de reconfiguration seront alors beaucoup plus rapides au détriment cette fois, de la flexibilité et de la gamme d'applications pouvant être implémentées par ces architectures.

C.3 Impact des réseaux d'interconnexion sur la dynamicité de la reconfiguration

Les réseaux d'interconnexion sont destinés à assurer les communications entre unités de traitement. Selon la topologie d'un réseau d'interconnexion les performances d'exécution, la consommation ainsi que la flexibilité d'un système s'en trouvent changées.

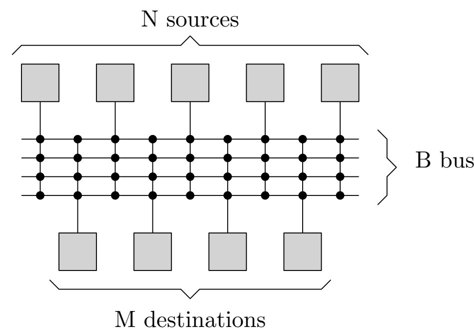


Figure 1-5 – **Exemple de réseaux d'interconnexion global**. *Les M destinataires connectés au réseau peuvent communiquer sans contrainte avec les N sources à condition que le nombre de bus B soit au moins égal au nombre minimum entre les N sources et les M destinations* ($B \geq min(M,N)$).

C.3-1 Réseaux d'interconnexion globaux

Les *crossbar* ou *multi-bus* permettent une connexion totale entre tous les éléments qui s'y connectent (figure 1-5). Cette particularité permet de qualifier ces réseaux comme étant totalement flexibles. Cependant, le prix de cette flexibilité est au dépend d'une forte consommation en énergie et d'un temps de traversée important qui sont proportionnels à leur taille et au nombre de ressources connectées.

C.3-2 Réseaux d'interconnexion point-à-point

Le but des réseaux d'interconnexion point-à-point est de proposer des solutions efficaces en temps et en surface pour des communications locales. De nombreuses topologies existent mais la plus utilisée est sans aucun doute la topologie *mesh* (figure 1-6). Les communications

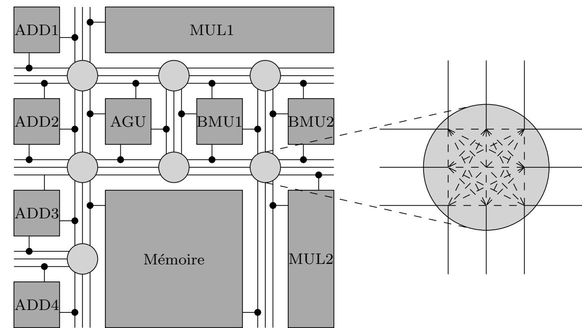


Figure 1-6 – **Exemple de réseau d'interconnexion point-à-point**. *Les différents éléments de l'architecture communiquent par le biais de lignes de communication partagées. Les données sont orientées dans le circuit par des boîtes de commutation (*switch-box*).*

sont basées sur l'utilisation de *switch-box* chargées d'orienter les communications entre les différentes ressources. Le temps de traversée d'une *switch-box* n'est pas pénalisant localement, mais peut devenir contraignant pour les communications de longues distances devant en traverser plusieurs.

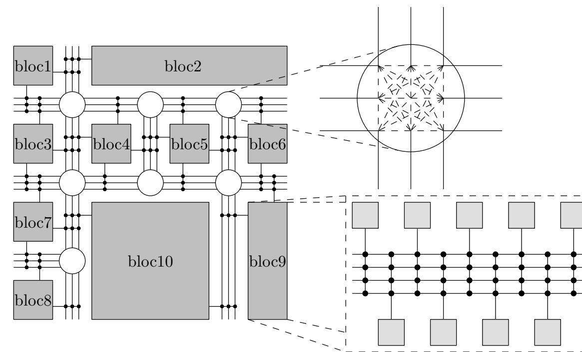


Figure 1-7 – **Exemple de réseaux d'interconnexion hiérarchique**. *Les différents blocs de l'architecture communiquent par le biais d'un réseau mesh généralisé. Les communications locales sont assurées par un réseau de type multi-bus.*

C.3-3 Réseaux d'interconnexion hiérarchiques

La décomposition hiérarchique des réseaux de connexion permet de pouvoir profiter des avantages des deux modes de connexion cités précédemment grâce à l'utilisation de réseaux multibus pour les communications locales et de *switch-box* pour les communications longues distances (figure 1-7). L'architecture est décomposée en blocs dans lesquels les communications locales sont optimisées. Toutefois, une attention toute particulière doit être portée sur les étapes de placement/routage du fait de la complexité des réseaux ainsi créés.

C.3-4 Impact de la flexibilité d'interconnexion sur la reconfiguration dynamique

La granularité d'une architecture dont la flexibilité est variable en termes de calcul induit la mise en œuvre d'une communication entre ressources en adéquation avec ce choix. Nous avons présenté des structures d'interconnexions dans les architectures reconfigurables, qu'il est nécessaire de caractériser afin de connaître leur impact sur la reconfiguration dynamique. L'interconnexion au sein d'un FPGA se doit, si l'on souhaite conserver la flexibilité des blocs logiques programmables, d'être elle aussi la plus flexible possible en proposant des schémas de connexion les plus souples possibles. À l'inverse, la flexibilité d'un processeur reconfigurable étant déjà le plus souvent limitée, les schémas d'interconnexion peuvent également se restreindre aux routages les plus pertinents pour un algorithme donné. Ainsi, cela contribue à la réduction de la taille des mots de configuration et, par voie de conséquence, des temps de reconfiguration, de la consommation et de la surface de silicium.

C.4 Méthodologie pour la mise en œuvre de la reconfiguration dynamique

Le spectre des architectures reconfigurables dynamiquement est limité par deux structures entre lesquelles nous pouvons trouver une multitude de variantes. D'un côté, une architecture FPGA, générique, flexible et capable d'implémenter toutes les applications actuelles et futures, de l'autre un processeur reconfigurable, moins flexible mais dont les unités de traitement sont mieux adaptées à un domaine d'application et par conséquent plus efficaces en termes de rapidité de traitement pour un calcul donné. Il existe entre les deux une multitude de solutions architecturales qui, de part leur niveau de granularité, leur flexibilité en termes de connexions et de leur spécialisation en termes d'unités de traitement se rapprocheront de l'une ou de l'autre des solutions. Dans cette section, nous évoquons les différentes techniques, méthodes et solutions architecturales qui s'offrent à un concepteur d'architectures reconfigurables.

C.4-1 Mécanismes d'accélération de la reconfiguration dynamique

Le développement d'une architecture reconfigurable nécessite la mise en œuvre de processus liés à la gestion de la reconfiguration afin que celle-ci reste dynamique. Les processus de reconfiguration influent sur les performances d'une architecture en termes de surface de silicium et de consommation et surtout en termes de temps d'exécution. Le temps alloué à un traitement doit prendre en compte les contraintes temporelles exigées par une application mais également les contraintes temporelles nécessaires à la configuration de la tâche. C'est pourquoi il est important de caractériser les concepts fondamentaux indispensables à l'implémentation et au développement d'une architecture que le contrôle de la reconfiguration dynamique devra être capable de gérer.

La notion de reconfiguration dynamique implique une certaine rapidité dans l'exécution de celle-ci. Afin de garantir cette dynamicité, il est essentiel de s'assurer que les ressources de reconfiguration utilisée ne viennent pas éclipser les bénéfices atteints par la reconfiguration

dynamique.

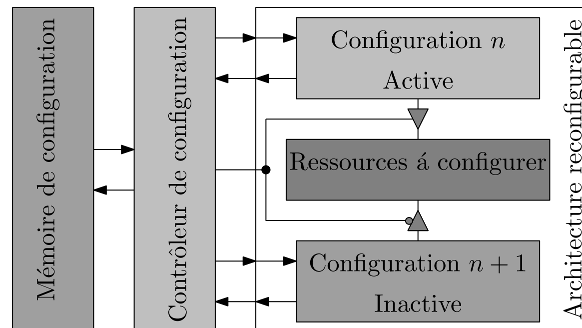


Figure 1-8 – **Exemple d'implémentation à pré-chargement**. *La zone de mémoire supérieure contient la configuration en cours de fonctionnement, alors que la zone de mémoire inférieure contient la configuration future prête à fonctionner. L'une ou l'autre des mémoires est activée par le contrôleur de configuration.*

Préchargement des données configurations le pré-chargement d'une future configuration permet de superposer les phases de calcul et les phases de reconfiguration (figure 1-8). La solution envisagée est la création de plusieurs mémoires de configurations qu'il est possible de venir écrire à tout moment. Cela a été exploité notamment dans les FPGA multi-contexte [5, 6] où chaque contexte est stocké près de l'unité de traitement et peut être instantanément utilisé. Le principal inconvénient de cette méthode vient du fait que des ressources sont intrinsèquement présentes mais non utilisées, ce qui a pour conséquence d'augmenter la surface de silicium et la consommation d'énergie, tout en diminuant l'efficacité de l'implémentation.

Compression des données de configuration et reconfiguration par différence : lorsque les phases de reconfiguration se succèdent très rapidement, le temps alloué au chargement du flot de configuration peut ne pas être suffisant. Une méthode pour diminuer le temps nécessaire à ce chargement est de compresser les données à charger. Une méthode [7] repose sur la compression de la différence entre le flot de données de configuration d'un module et de sa relecture en cours d'exécution. Un *ou-exclusif* logique bit à bit est appliqué entre le flot de données de configuration du module original et le flot de données de relecture. Si les informations sont identiques dans les deux flux, comme des informations de routage ou de configuration de LUT, alors le flux résultant de cette fonction *ou exclusif* sera composé de bits nuls. Seul les états ayant changé de valeur seront représentés par un "1" logique. Étant donné que les bits de configuration sont très largement majoritaires, le nombre de zéros adjacents dans le flux le sera aussi, la compression sera alors d'autant plus efficace. Il est cependant nécessaire de préciser que bien souvent le flot de données de configuration extrait n'est pas identique en termes de formatage des données rendant de fait, cette méthode caduque.

Reconfiguration partielle : la gestion de la reconfiguration partielle est un des critères nécessaire à une reconfiguration efficace. Toutes les ressources d'une zone de reconfiguration ne sont pas nécessairement requises pour l'implémentation d'une tâche. Ou bien, une fois

l'exécution d'une tâche terminée, une zone peut être reconfigurée sans avoir à perturber les autres tâches en cours de traitement (figure 1-9). Par exemple, la première phase A implémente trois tâches (T1, T2, T3), qui seront totalement remplacées par deux autres tâches (T4 et T5) dans la phase B, suivie de la phase C qui remplacera une tâche (T5) par une autre (T1) tout en laissant la deuxième tâche (T4) s'exécuter.

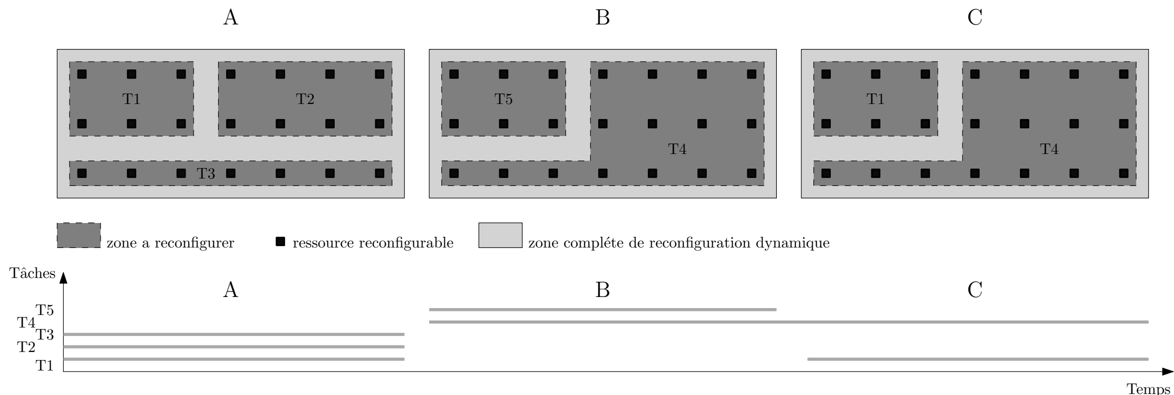


Figure 1-9 – **Implémentation asynchrone de tâches sur une zone de reconfiguration**. *Une architecture reconfigurable dynamiquement doit être capable de procéder à des reconfigurations partielles de ses ressources, sans que cela ne porte préjudice aux ressources voisines, notamment au niveau des communications entre ressources.*

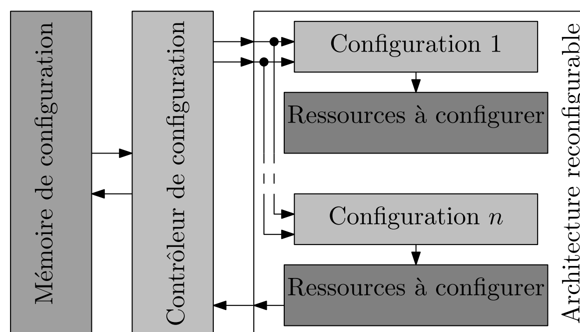


Figure 1-10 – **Exemple d'implémentation à reconfiguration partielle**. *La zone de mémoire de configuration est divisée en plusieurs zones afin de pouvoir accéder à chacune de manière individuelle.*

La reconfiguration partielle (figure 1-10) s'applique très facilement dans les architectures dont la configuration est mémorisée dans les mémoires RAM (*Random Access Memory*) comme les FPGA Xilinx [2] ou le processeur reconfigurable PipeRench [8] entre autres. La méthode consiste à simplement spécifier l'adresse de la mémoire à reconfigurer directement dans le flot de configuration, il est alors possible de procéder à la seule reconfiguration de cette zone. La zone de reconfiguration est le plus souvent indépendante de la zone non reconfigurée ce qui permet d'avoir un parfait recouvrement des phases de reconfiguration et des phases de calculs sur deux zones distinctes. La reconfiguration partielle permet de réduire très fortement la taille du flot de reconfiguration et ainsi le temps nécessaire à cette reconfiguration. Néanmoins, le

gain apporté par la solution d'adressage individuel des mémoires doit être relativisée étant donné que les bits d'adresses contenus dans le flot de données augmente avec le nombre de bits à transférer, et augmente donc la taille du flot de données de configuration et par conséquent le temps de reconfiguration.

C.4-2 Mécanismes de gestion de la préemption

La gestion flexible des phases de reconfiguration est un enjeu important. Cela se traduit par la possibilité de gérer la priorité des tâches dans l'ordonnancement de celles-ci. Pour cela, un mécanisme bien connu du monde des processeurs doit être supporté, la préemption. La préemption est un élément primordial pour aboutir à une exécution temps réel optimisée des calculs par reconfiguration d'un ensemble d'applications non déterministes comme cela pourrait être envisagé dans le cas des architectures reconfigurables dynamiquement. De manière générale, le traitement préemptif consiste à d'abord pouvoir suspendre une tâche ou un traitement en cours d'exécution, d'en sauvegarder le contexte, puis de configurer une nouvelle tâche dont la priorité est plus importante, l'exécuter et enfin reconfigurer et restaurer le contexte de la tâche précédente une fois que la seconde a terminé son exécution.

La préemption sur les architectures reconfigurables reste encore très marginale. Le plus souvent, le remplacement d'une tâche par une autre se fait seulement après l'exécution complète de la tâche précédente. Il n'est donc pas nécessaire de sauvegarder les états des données internes de la structure reconfigurable exécutant le module pour une reprise ultérieure de l'algorithme. Pour appliquer la préemption aux FPGA, c'est à dire pour pouvoir remplacer un module avant la fin de son exécution, il est nécessaire de geler son exécution, puis de lire et sauvegarder les états des registres du FPGA utilisés par ce module avant de charger une nouvelle configuration dans le FPGA. Après reprise d'un module préemptif les états des registres précédemment sauvegardés doivent être restaurés dans leur état d'interruption avant que celui-ci ne puisse recommencer à travailler. Cependant, quelques problèmes s'imposent à la réalisation de cette tâche. Les états des données peuvent êtres répartis à travers tous les modules dont aucun mécanisme ne permet l'accès direct. De plus, les registres d'états d'un module implémenté dans un FPGA contiennent un nombre non négligeable de bits mémoire supplémentaires par rapport aux routines exécutées sur microprocesseur. Il faut donc augmenter fortement la taille des mémoires à utiliser pour la sauvegarde de ces registres et des autres données. Par comparaison, une interruption dans un processeur Intel Pentium II impose la sauvegarde de 104 octets. Dans le cas d'un FPGA Xilinx XCV1000, la sauvegarde de tous les registres, bascules et mémoires internes nécessitent 350 kilo octets.

D Architectures reconfigurables dynamiquement

Nous avons choisi de classifier les architectures reconfigurables selon deux critères distincts : d'une part par la méthode de reconfiguration et d'autre part par la granularité de la reconfiguration. Une architecture mono-contexte est une architecture reconfigurable dynamiquement dont chaque contexte est stocké à l'extérieur du système. Les unités de traitement doivent alors être configurées avant de pouvoir procéder à l'exécution d'un nouvel algorithme. Une

architecture multi-contexte contient sur le même circuit plusieurs contextes qu'il est possible de venir valider de manière successive et indépendante selon les besoins de l'application ou de l'algorithme à exécuter. La deuxième caractéristique notable d'une architecture reconfigurable concerne la granularité de la reconfiguration qui impliquera une flexibilité et des performances plus ou moins intéressantes selon les choix effectués.

D.1 Architectures mono-contexte grain fin

D.1-1 ATMEL AT40K

Architecture : l'ATMEL AT40K [9], aujourd'hui dépassée mais innovante en son temps, mérite pour cela notre attention. L'AT40K fût l'une des premières architectures FPGA a autoriser la reconfiguration partielle de ses ressources. Le bloc logique utilisé (figure 1-11) permet par l'utilisation de deux LUT à trois entrées, la résolution de deux équations à trois variables ou bien d'une équation à quatre variables. L'architecture de l'AT40K est constituée d'une matrice de blocs logiques identiques. Les bus de connexion sont pour leur part différents selon les distances de communications qui devront être parcourues entre deux blocs logiques. Pour les communications longues distance, un répéteur de bus est placé tout les quatre blocs logiques. À chaque intersection de ligne et de colonne de répéteurs, se trouve un bloc SRAM (*Static Random Access Memory*) pouvant stocker jusqu'à trente deux mots de quatre bits. Ces blocs mémoire sont configurables de manière à pouvoir y accéder en simple ou en double ports de manière synchrone ou asynchrone. Chaque bloc logique est connecté au réseau d'interconnexion d'une part, mais également directement aux blocs logiques voisins d'autre part, horizontaux, verticaux et diagonaux (figure 1-12). Cela permet d'économiser des unités d'interconnexions tout en proposant des implémentations performantes.

Reconfiguration dynamique : la reconfiguration de l'architecture AT40K est réalisée selon six modes au choix. Un mode d'autoconfiguration appelé le mode *master*, quatre modes appelés *slave* et enfin le mode *synchronous RAM* permettant d'accéder à la mémoire de configuration directement depuis le port parallèle d'un microprocesseur. Le mode *master* permet une auto-configuration de l'architecture. Celle-ci est exécutée lors des phases de réinitialisation du système. Les modes *slave* sont eux initialisés à partir d'un signal externe. Les mots de configuration peuvent être au choix envoyés en série ou en parallèle sur des mots de huit ou seize bits selon le *slave mode* choisi. La reconfiguration partielle est réalisée en mode *synchronous RAM*. Les mémoires de configuration sont vues par le système comme un espace mémoire à part entière qu'il est possible de venir lire et écrire. Le flot de données de configuration contient trente-deux bits d'information dont vingt-quatre bits d'adresse (adresse de début et adresse de fin de la zone ciblée) et huit bits de données [1]. Bien que cette architecture soit toujours utilisée, notamment dans le domaine de l'aérospatial, sa faible taille (48×48 blocs logiques au maximum) en fait un candidat peu probable pour les applications de nouvelle génération. Notons tout de même que cette architecture accouplée à un processeur en fait un honorable co-processeur reconfigurable (FPSLIC [10]).

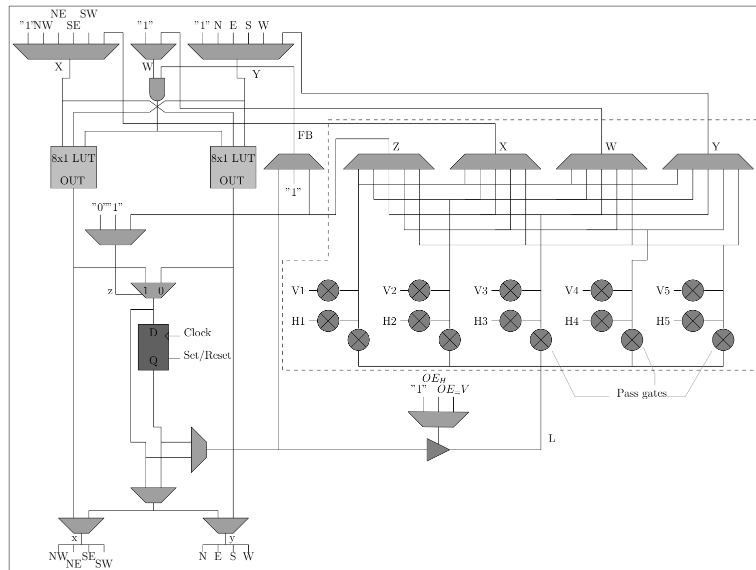


Figure 1-11 – **Bloc logique de l'AT40K**. *Le bloc logique implémenté dans le FPGA AT40K est composé de deux LUT à trois entrées indépendantes. Selon le mode de fonctionnement choisi, le bloc logique permet la résolution de deux équations à trois variables ou bien d'une équation à quatre variables. Cependant, seule une sortie parmi les deux peut être synchrone. Les ressources entourées d'un polygone en pointillé représentent les ressources de connexions aux réseaux de connexion horizontaux et verticaux.*

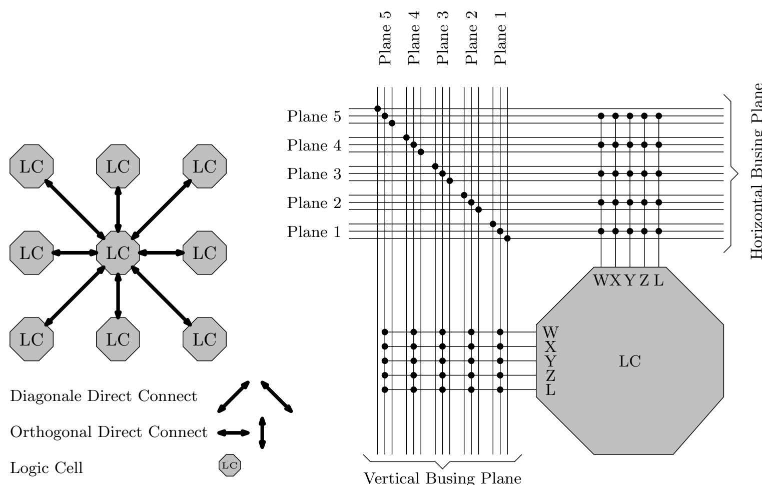


Figure 1-12 – **Interconnexions dans un bloc logique AT40K**. *Les blocs logiques implémentés dans le FPGA AT40K sont interconnectés à des ressources voisines de courte distance ainsi qu'à des ressources plus éloignées par l'intermédiaire de bus dédiés nécessitant l'utilisation de moins de routeurs intermédiaires. Cela permet de minimiser les temps de propagation.*

D.1-2 Famille Virtex de XILINX

Architecture : la famille des FPGA Virtex est très étendue, nous nous intéresserons au premier d'entre eux [4]. Contrairement à l'AT40K, l'architecture du Virtex est hiérarchique. C'est à dire que deux blocs logiques (figure 1-13) sont regroupés en *slice* eux même regroupés

par deux en CLB. Chaque CLB est connecté sur le réseau via une matrice de routage de manière à composer une grille de lignes et de colonnes de CLB. L'architecture d'un bloc logique se compose de deux LUT à 4 entrées pouvant être utilisées de manière commune ou indépendante. Des ressources logiques supplémentaires présentes dans le bloc logique permettent de procéder rapidement à un calcul de retenue au cas où celui-ci serait utilisé pour des calculs arithmétiques. Des multiplexeurs situés en sortie permettent d'orienter les résultats du bloc logique sur les sorties synchrones ou asynchrones. Enfin, chaque LUT peut également être utilisée comme une RAM synchrone de seize mots de un bit accessible sur un ou deux ports.

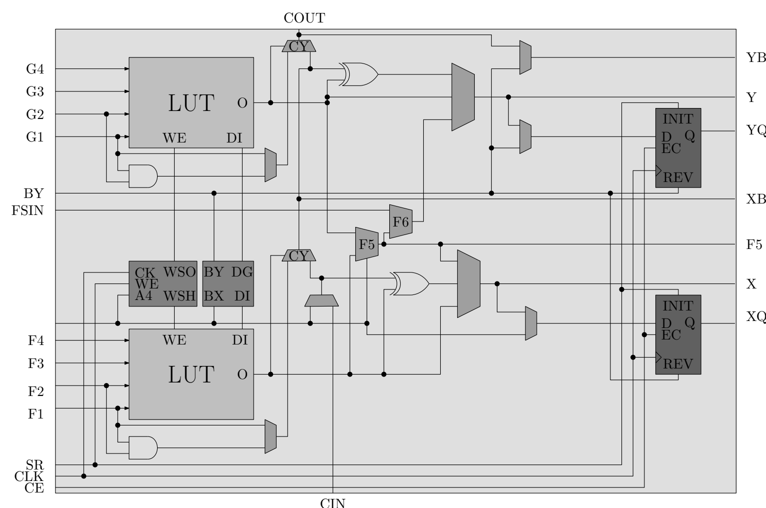


Figure 1-13 – **Bloc logique de type Virtex**. *Le bloc logique implémenté dans le FPGA Virtex est composé de deux LUT à 4 entrées utilisables indépendamment ou en commun. Quel que soit le mode choisi, trois variables seulement peuvent intervenir dans la résolution de l'équation réalisable par les deux LUT.*

RECONFIGURATION DYNAMIQUE : tout comme l'AT40K, plusieurs modes de configuration sont possibles sur le Virtex, mais seuls les modes de configuration *Slave SelectMAP* et *Boundary Scan* permettent la reconfiguration partielle de l'architecture. La reconfiguration partielle peut-être exécutée de deux manières différentes, soit en utilisant la méthode appelée *Module-based partial reconfiguration*, soit en utilisant la méthode appelée *Small-Bit Manipulations*. La méthode de reconfiguration partielle *Module-based* permet de procéder à la reconfiguration sur des zones ayant été définies au préalable. Le nombre global de modules doit être le plus faible possible afin de minimiser les problèmes de communication au sein de systèmes complexes. La méthode de reconfiguration partielle *Module-based* est plus particulièrement dédiée au développement de modules indépendants dont la communication est effectuée à travers des bus dédiés garantissant une bonne communication. La méthode *Small-Bit Manipulations* est quand à elle basée sur la production d'un flot de données de configuration qui ne prend en compte que les modifications éventuelles du flot de données de configuration par rapport au précédent. Ainsi, le passage d'une configuration à une autre peut-être très rapide (de l'ordre de quelques millisecondes [11]) car le flot de données de configuration est alors bien plus petit que le flot de données de configuration complet.

D.2 Architectures mono-contexte grain épais

D.2-1 DART

De manière à rendre la reconfiguration plus rapide, les architectures à grain épais et à chemin de données reconfigurable ont été développées dans l'objectif de spécialiser leurs ressources, et donc de limiter la taille des configurations nécessaires à leur reconfiguration. Ce choix est évidement effectué au détriment de la flexibilité, mais au bénéfice de la performance d'execution et de la consommation d'énergie.

Architecture : DART [12] (figure 1-14) est une architecture à reconfiguration dynamique conçue pour traiter les applications de télécommunication mobiles de troisième génération. L'architecture est composée d'un contrôleur de taches, de ressources de mémorisation et de unités de traitement appelés *Clusters*. Le contrôleur de tâches est chargé d'assigner aux *Clusters* les différents traitements à exécuter. Une fois configuré, chaque *Cluster* travaille de manière autonome et gère ses propres accès à la mémoire de données. Chaque *Cluster* intègre un cœur de traitement dédié et six DPR (*DataPath Reconfigurable*) opérant sur des données de largeur allant de 8 à 32 bits. Les DPR constituent le dernier niveau de la hiérarchie de DART. Ces DPR sont constitués d'unités fonctionnelles interconnectées suivant un motif totalement flexible. Chaque DPR intègre quatre unités fonctionnelles, deux multiplieurs/additionneurs (UFl et UF3) et deux UAL (UF2 et UF4). Toutes ces unités fonctionnelles sont dynamiquement reconfigurables et supportent les traitements SWP (*Sub-Word Processing*).

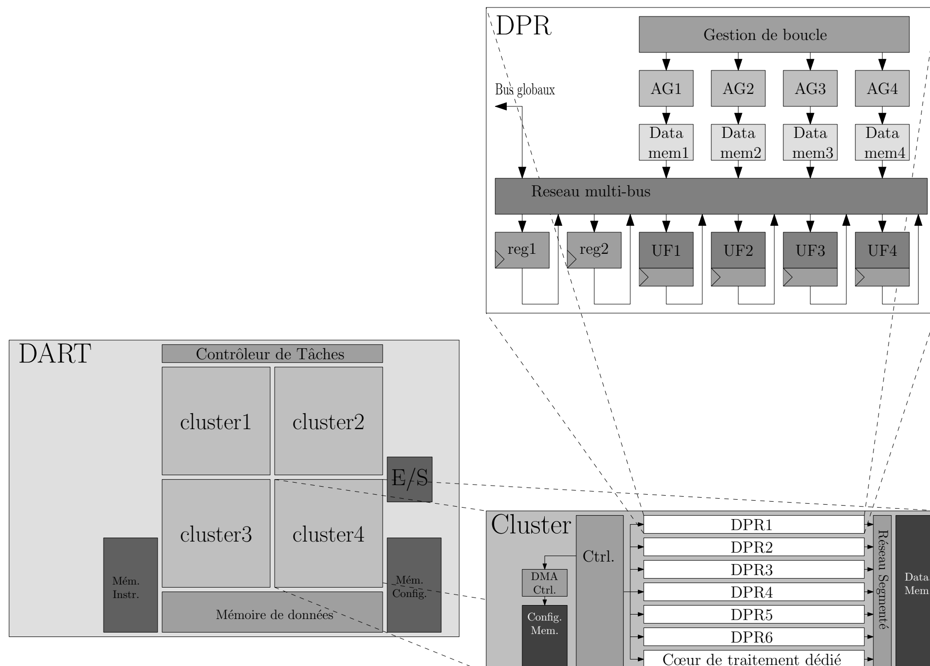


Figure 1-14 – **Architecture d'un *DataPath Reconfigurable* de DART**. *Un DPR est constitué d'unités fonctionnelles interconnectées suivant un motif totalement flexible de type multi-bus. Elles travaillent sur des données stockées dans les mémoires locales auxquelles sont associées des générateurs d'adresses (AG).*

Reconfiguration dynamique : la reconfiguration dynamique sur DART peut être de nature logicielle ou matérielle. Le mode de reconfiguration logicielle a été conçu afin de répondre aux besoins inhérents aux zones de calcul irrégulières, où les motifs de calculs se succèdent très rapidement, sans ordre particulier et de manière non répétitive. La principale caractéristique de ce mode de reconfiguration est d'autoriser une reconfiguration des DPR en un cycle. Ce mode de reconfiguration est qualifié de reconfiguration logicielle. Outre la reconfiguration logicielle, adaptée aux traitements irréguliers, un second mode de reconfiguration a été défini afin de répondre aux besoins des traitements réguliers, tels que les cœurs de boucles, dans lesquels un même motif de calcul est utilisé pendant de longues périodes de temps. Ce mode de reconfiguration est qualifié de matériel. Il s'agit ici d'assurer une totale flexibilité au sein du DPR afin d'autoriser l'optimisation du chemin de données en fonction du motif de calcul. Cette reconfiguration matérielle consiste à spécifier une configuration par le biais d'un flot d'instructions, puis d'adopter un modèle de calcul de type *dataflow* dans lequel la structure du chemin de données est figée. Une fois cette configuration spécifiée, le contrôleur du *Cluster* est donc libéré du contrôle des DPR et n'a plus à accéder à la mémoire d'instructions ni de décoder de nouvelles instructions.

D.2-2 Systolic Ring

Architecture : une des idées intéressantes du *Systolic Ring* [13] est de proposer l'implémentation d'un algorithme pipeliné dans une boucle d'unités de traitement appelés *Dnode* (figure 1-15 a) et dont l'architecture peut s'apparenter à un micro-processeur. Le flot de données est propagé de *Dnode* en *Dnode* de manière circulaire. Ainsi, chaque nœud du pipeline peut être successivement implémenté sur l'un des *Dnode* de manière à pouvoir exécuter le calcul désiré sur les données entrantes. Le *Dnode*, représenté figure 1-15 (b), constitue l'unité de traitement reconfigurable dynamiquement de base du *Systolic Ring*. Celui-ci est composé d'une UAL reconfigurable. Le traitement de type "flot de données" se fait sur des données de seize bits. Un banc de quatre mots de seize bits est également présent pour la sauvegarde de calculs intermédiaires.

Reconfiguration dynamique : la reconfiguration du *Dnode* est réalisée par une micro-instruction de dix-sept bits. Cette micro-instruction est extraite d'une mémoire dédié. Une requête de reconfiguration intervient pendant les phases de traitement et permet une modification de la fonctionnalité du *Dnode* en un cycle d'horloge (d'une addition vers une multiplication par exemple). La gestion des configurations est réalisée par un séquenceur d'instructions, également présent sur la figure 1-15 (b). Les *Dnode* sont organisés en différentes couches dont chacune est connectée au deux couches de *Dnode* suivante et précédente par l'intermédiaire de ressources d'interconnexions reconfigurables de type multi-bus complètement connecté.

D.2-3 WPPA : Weakly Programmable Processor Array

Architecture : l'architecture WPPA (*Weakly Programmable Processor Array*) développée à l'université de Erlangen [14, 15], est une matrice paramétrable d'unités de traitement appelés WPPE (*Weakly Programmable Processor Element*) et représentés sur la figure 1-16.

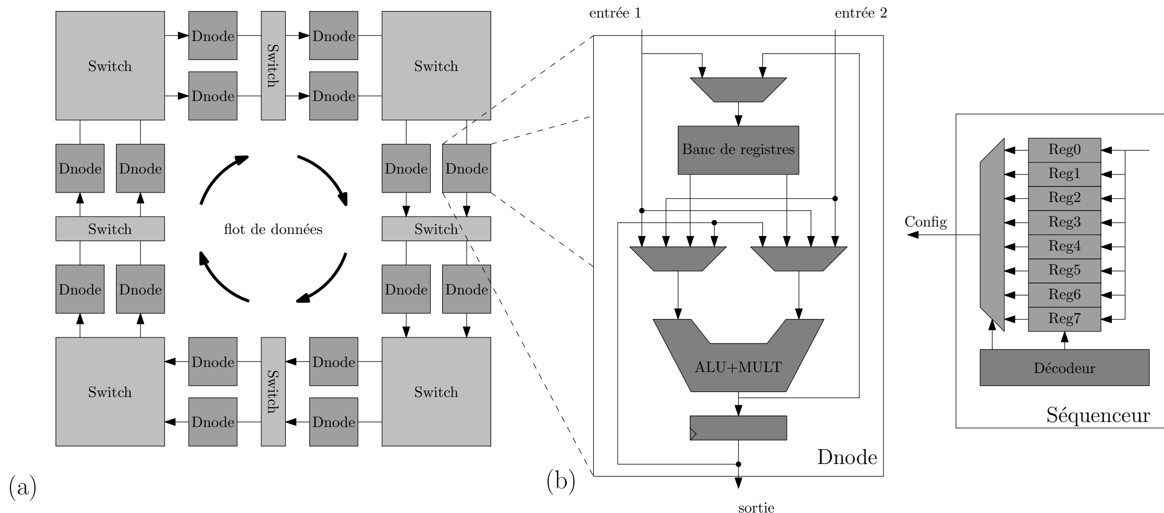


Figure 1-15 – **Architecture *Systolic Ring*.** *L'architecture* Systolic Ring *est composée d'unités de traitement appelés* Dnode *et connectés en anneaux (a). Le flot de données est propagé de* Dnode *en* Dnode *après reconfiguration dynamique. Ainsi, tous les algorithmes pipelinés peuvent être implémentés sans se soucier de la taille de celui-ci. Le* Dnode *(b) constitue l'unité de traitement du* Systolic Ring. *Son architecture est comparable à celle d'un micro-processeur et travaille sur des mots de seize bits. Le processus de reconfiguration est exécuté en un cycle d'horloge par l'intermédiaire d'une configuration de dix-sept bits.*

Chaque WPPE, de type processeur VLIW (*Very Large Instruction Word*), comporte une mémoire ne pouvant contenir que quelques instructions. De plus, les ressources de contrôle sont optimisées de façon à occuper le moins de place possible. Le jeu d'instructions (spécifié lors de la description du WPPE) est restreint aux instructions couramment utilisées dans le domaine du traitement du signal. Les WPPE peuvent être spécifiés de manière à réaliser des additionneurs/soustracteurs, des multiplicateurs, des registres à décalage, ou encore des opérations logiques.

Reconfiguration dynamique : un réseau d'interconnexions flexibles permet de mettre en œuvre la reconfiguration dynamique sur le flot de données. Chaque WPPE est entouré d'un élément d'interconnexion appelé *interconnect wrapper* et est capable de se reconfigurer en cours de traitement dans différentes topologies. Les différentes topologies sont spécifiées pendant la description de l'architecture à l'aide d'une matrice où chaque sortie représente une ligne et chaque entrée une colonne. La connexion entre une entrée et une sortie est alors symbolisée par un "c" au niveau de l'intersection de la ligne et de la colonne concernée ou d'un "0" dans le cas contraire.

D.3 Architectures multi-contexte grain fin

Toujours dans l'objectif de permettre aux architectures reconfigurables de procéder aux phases de reconfiguration le plus rapidement possible, certaines solutions se sont tournées vers l'implémentation de mémoires de configurations supplémentaires.

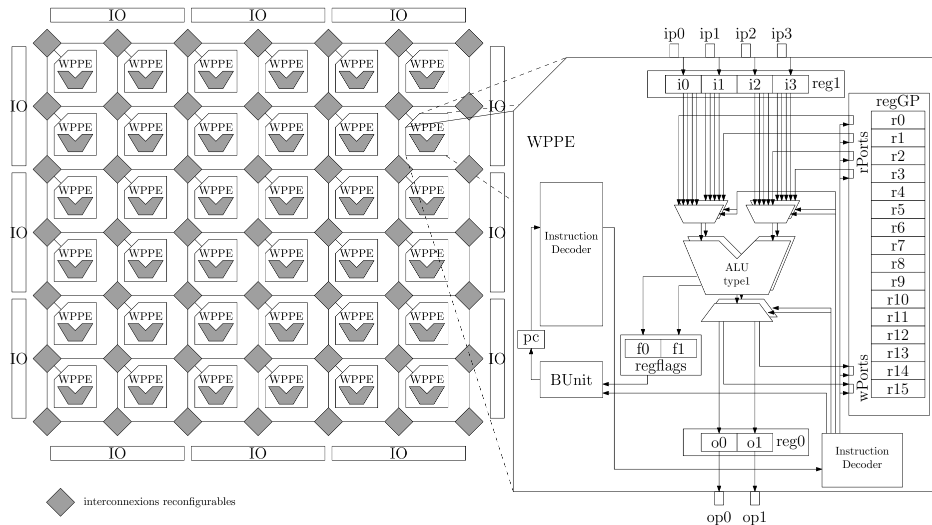


Figure 1-16 – **Architecture de WPPA et de ses WPPE**. *WPPA est une matrice de WPPE interconnectés par des unités d'interconnexions reconfigurables dynamiquement. Un WPPE est une unité de traitement de type processeur VLIW.*

D.3-1 DPGA : Dynamically Programmable Gate

Afin de comprendre le principe d'une architecture multi-contexte, étudions tout d'abord la structure de DPGA (*Dynamically Programmable Gate*) [5] qui a été conçu au MIT (*Massachusetts Institute of Technology*) en 1995. Il s'agit d'une architecture "basique" de FPGA dont les blocs logiques sont composés d'une LUT à quatre entrées et d'une bascule permettant la synchronisation des sorties. Cette architecture est historiquement intéressante puisqu'elle fut la première architecture grain fin à implémenter une technique de reconfiguration par multi-contexte. Seul un contexte parmi quatre est actif à la fois permettant aux contextes inactifs un état de repos. Le passage d'un contexte actif à un autre se fait par l'intermédiaire de deux bits de contrôle appelés *ContextID* (figure 1-17).

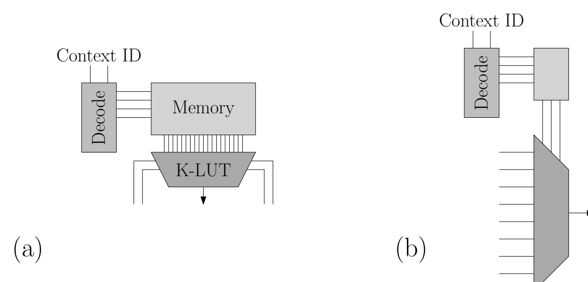


Figure 1-17 – **Architecture DPGA**. *Quatre configurations sont sélectionnées une par une grâce au signal* ContextID *afin de valider le contexte approprié pour les unités de traitement (A) et d'interconnexions (B)*

D.3-2 LATTICE ispXPGA

Architecture : le composant Lattice ispXPGA [16] est une architecture FPGA composée d'une matrice de blocs logiques complexes appelés *Programmable Function Units*, de cellules d'entrées/sorties programmables appelées *Programmable Input Output Cells* et de blocs de RAM embarqués appelés *Embedded Block* RAM. L'architecture d'un *Programmable Function Units* (figure 1-18) se décompose en trois blocs. Le premier est constitué par des ressources de logiques combinatoires assurées par un ensemble de quatre *Configurable Logic Elements*. Chaque *Configurable Logic Elements* est composé d'une LUT à quatre entrées et de ressources permettant la propagation de retenue entre *Configurable Logic Elements*. De plus, la fonctionnalité de la LUT peut-être modifiée de manière à en faire une mémoire 16×1. Le deuxième bloc, appelé *Wide Logic Generator*, permet l'assemblage des sorties des différents *Configurable Logic Elements* de manière à réaliser des fonctions logiques de granularité plus élevée. Enfin, le dernier bloc permet la conversion des signaux combinatoires en logique séquentielle *Configurable Sequential Element* par l'intermédiaire de ressources spécifiques capables de réaliser différentes bascules (D et RS) une fois correctement configurées.

Reconfiguration dynamique : les données nécessaires à la configuration d'un contexte actif sont stockées dans une mémoire de type SRAM. Parallèlement à cette mémoire, une autre de type E^2CMOS (*Electrically Erasable Complementary Metal Oxide Semiconductor*) permet le stockage d'une configuration future dont le chargement peut être fait au préalable en arrière plan. Cela permet la préparation d'un futur contexte sans avoir à stopper l'architecture en fonctionnement. Cette mémoire est également utile pour une première configuration après un démarrage à froid puisque non volatile. Le rafraîchissement de la mémoire SRAM avec les données de la mémoire E^2CMOS dure environ $200\mu s$ [17].

D.3-3 Piperench

Architecture : l'architecture Piperench [8] a la particularité d'être une architecture travaillant sur des données de huit bits mais ses unités fonctionnelles pipelinées sont constituées de LUT travaillant au niveau bit, justifiant de fait cette classification. L'architecture Piperench est décomposée en *stripe*. Chaque *stripe* est composées de seize unités de traitement appelées *Processing Elements* (figure 1-19). L'unité fonctionnelle de chaque *Processing Element* est constituée de huit LUT à trois entrées configurées de manière semblable. L'entrée *Xin* est connectée à toutes les LUT, ce qui est utile pour l'implémentation de multiplexeurs ou de multiplieurs. Le calcul de retenue pour l'implémentation de soustracteurs ou d'additionneurs est effectué à l'aide de ressources dédiées. Il est possible de programmer la connexion de ces signaux afin de pouvoir procéder à l'implémentation d'unités fonctionnelles travaillant sur des données plus larges.

Reconfiguration dynamique : l'architecture Piperench est développée de telle sorte qu'elle supporte la virtualisation du procédé de pipeline tout en limitant son contrôle. La virtualisation du pipeline consiste à décomposer un algorithme pipeliné en différents étages pouvant être configurées de manière successive, au fur et à mesure de son état d'avancement.

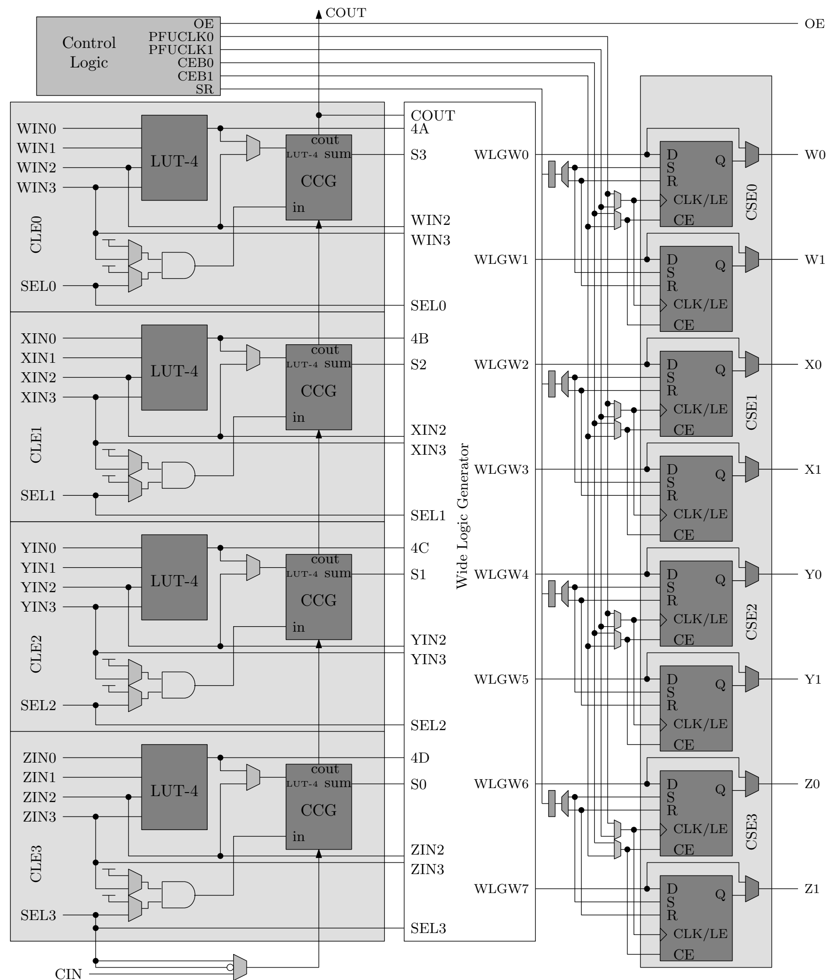


Figure 1-18 – **Bloc logique du Lattice ispXPGA**. *Le bloc logique implémenté dans le FPGA Lattice ispXPGA est composé de trois blocs dédiés à la conversion de logique (combinatoire vers séquentielle) et la conversion de granularité.*

Dans l'exemple décrit (figure 1-20), un pipeline virtuel de six étages virtuel (figure 1-20 A) va être implémenté sur quatre étages physiquement présents (figure 1-20 B). Chaque *stripe* est configuré selon le contenu de la mémoire de configuration embarquée indépendamment de son emplacement physique sur le circuit. La reconfiguration s'effectue en un cycle d'horloge, ce qui permet de libérer l'étage du pipeline concerné au dernier moment. La configuration d'un *Processing Element* nécessite 42 bits, soit 672 bits pour la configuration d'un *stripe* complet. Une mémoire de préemption supplémentaire est utilisée au cas où les ressources matérielles physiquement présentes ne suffiraient pas à assurer un parallélisme satisfaisant pour une application précise.

D.3-4 PicoGa de XiRisc

ARCHITECTURE : XiRisc est une architecture de processeur reconfigurable basée sur un processeur VLIW de type RISC (*Reduced Instruction Set Computer*) dont les capacités ont été étendues par l'introduction d'un chemin de données reconfigurable dynamiquement appelé

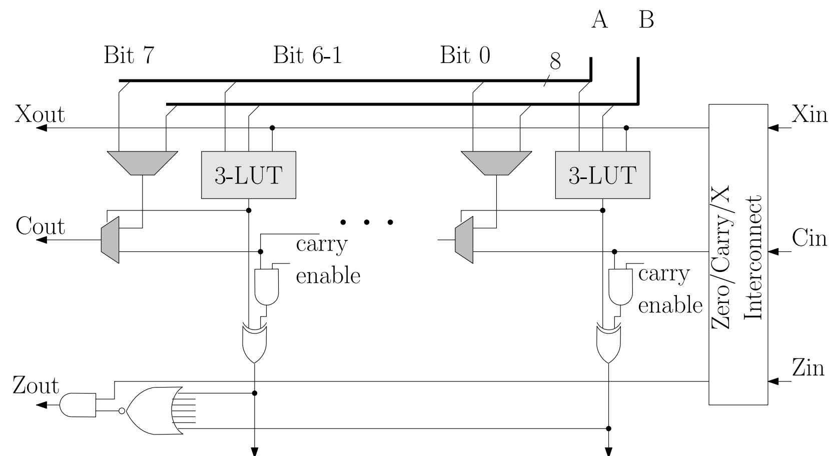


Figure 1-19 – ***Processing Element* de Piperench**. *Une unité de traitement Piperench est composée de huit LUT à trois entrées permettant de classifier cette architecture dans le groupe des grain fin.*

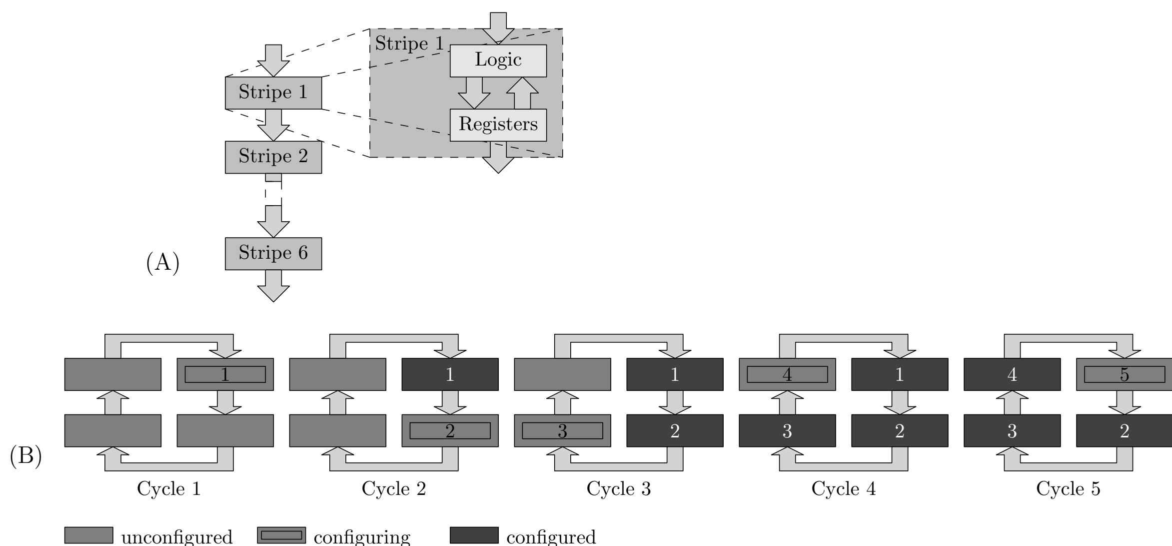


Figure 1-20 – **Phases de configurations et d'exécution de Piperench**. *La gestion du pipeline sur Piperench est telle qu'il est possible de procéder à des phases de reconfiguration rapides sur un nombre limité de ressources de manière à ne pas interrompre le pipleline.*

PiCoGa (*Pipelined Configurable Gate-Array*) [18]. PiCoGa est composé de ressources logiques appelées *Reconfigurable Logic Cells* multi-contexte bidimensionnels. Chaque rangée contient 16 *Reconfigurable Logic Cells* et un signal commun d'activation est donné pour chaque registre dans la rangée de *Reconfigurable Logic Cells* (figure 1-21). Ainsi, chaque rangée peut mettre en application un pipeline paramétrable, pouvant traiter des opérandes d'une taille allant jusqu'à trente-deux bits. Étant donnée la variabilité des données du flux d'entrée et étant donné que l'évolution du pipeline ne peut pas être préalablement spécifiée, la synergie spécifique entre le noyau de processeur et le PiCoGA exige un mécanisme dédié de commande afin de garantir la fonctionnalité de l'architecture.

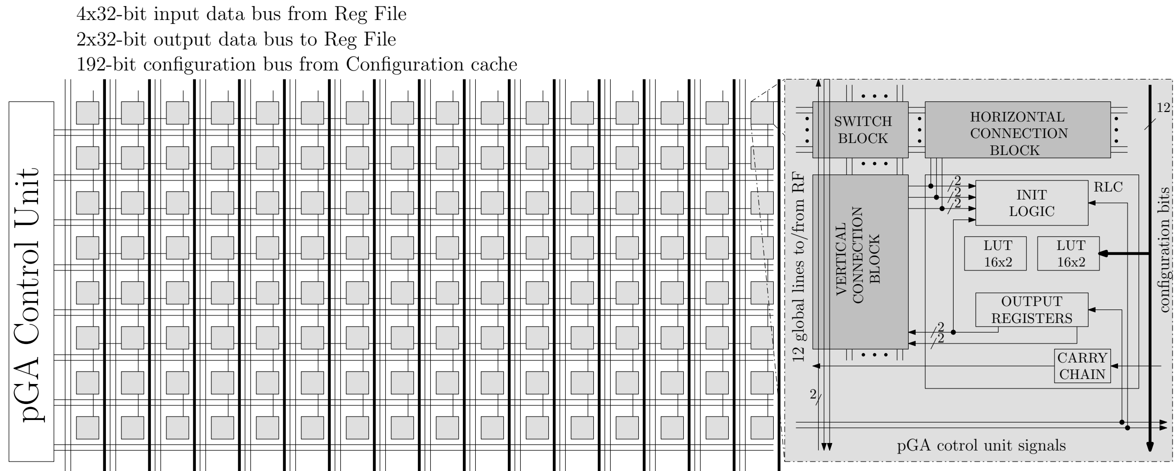


Figure 1-21 – **Architecture PicoGA du processeur reconfigurable XiRisc**. *PiCoGA est une matrice de bloc logique déstiné à l'implémentation matérielle d'algorithmes pipelinés.*

Reconfiguration dynamique : la représentation du calcul sur le PiCoGA est effectuée grâce à un graphe flot de données appelé *Pipelined Data Flow Graph*. Chaque nœud du *Pipelined Data Flow Graph* représente une étape du pipeline alloué sur le PiCoGA et ceci dans une ou plusieurs rangées du PiCoGA. Le contrôle du flot de données est assuré par un contrôleur reconfigurable. L'unité de commande principale est composée de sous unités dont chacune génère le signal de contrôle, appelé *Execution Enable*, afin d'activer les registres de la rangée correspondante. Les signaux *Execution Enable* provenant de chaque nœud suivant et précédent permettent à chaque sous unité de commande de rangée de gérer l'activation d'un état de nœud. En se basant sur la description des connexions du *Pipelined Data Flow Graph*, il est possible d'extraire les connexions correspondantes des différentes sous unités de commande de rangées, permettant à la matrice d'interconnexions programmable d'être configurée correctement.

D.4 Architectures multi-contexte grain épais

D.4-1 XPP : *eXtreme Processing Platform*

Architecture : XPP (*eXtreme Processing Platform*)) [19, 20] est une architecture à base d'unités de traitement appelées *Processing Array Elements*. Celles-ci sont organisées en matrices formant ainsi un *Processing Array*. Cette architecture a été développée afin de permettre l'exécution d'applications à flux de données multiples fonctionnant en parallèle. Un *Processing Array Element* est le résultat de l'instanciation de ressources prédéfinies disponibles dans des bibliothèques. Les ressources instantiées peuvent être de différentes natures, comme par exemple des ressources de routage ou de calcul, ou encore des ressources de type ALU. L'ALU est capable de procéder à des calculs arithmétiques sur des données à virgule fixe, à des opérations logiques ou bien des fonctions de comptage et de classement. Les ALU sont capables de générer des signaux sur des évènements issus de résultats de calcul ou de positionnement de drapeaux comme cela peut-être réalisé sur un micro-processeur. Une autre

ressource pouvant être implémentée dans un *Processing Array Element* est une ressource de mémorisation. Celle-ci peut-être utilisée en tant que FIFO, que RAM, et par extension en tant que LUT. Enfin, il est également possible de procéder au développement complet d'une ressource de type *Processing Array Element*.

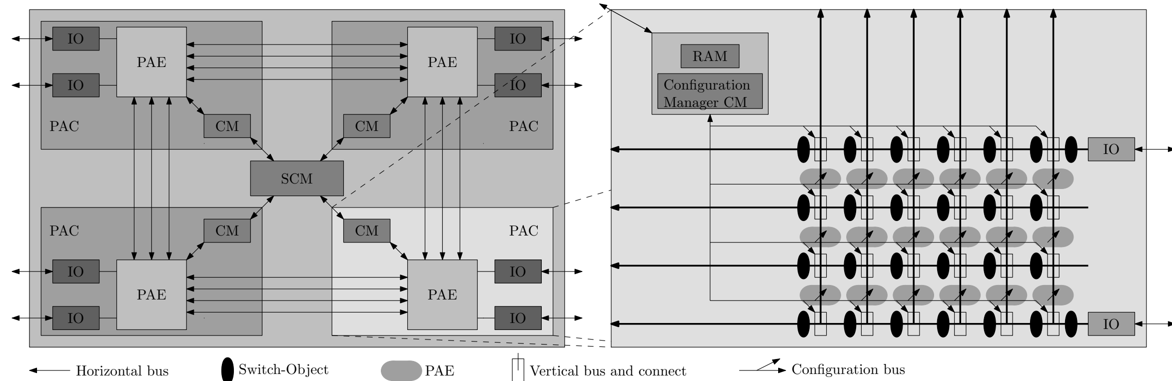


Figure 1-22 – **Architecture XPP**. *XPP est une architecture à plusieurs niveaux de hiérarchie composée de différentes matrices de traitement ou* Processing Array Cluster*. Chaque* Processing Array Cluster *est indépendant l'un de l'autre en termes de traitement de données et de reconfiguration dynamique. La reconfiguration dynamique est gérée de manière hierarchique par les* Configuration Managers *et le* Supervising Configuration Manager*.*

Reconfiguration dynamique : dans XPP, la reconfiguration dynamique est gérée par un contrôleur de configuration hiérarchique appelé *Configuration Manager*. Un *Processing Array* assemblé à un *Configuration Manager* de niveau bas est défini comme étant un *Processing Array Cluster*. Le *Configuration Manager* de niveau bas a la responsabilité d'écrire des données de configuration dans les ressources reconfigurables du *Processing Array*. L'utilisation de plusieurs *Processing Array Cluster* est typiquement nécessaire pour l'implémentation d'une application récente et constitue ainsi un processeur complet de type XPP. L'ensemble de *Configuration Managers* forment un arbre hiérarchique de données de configurations. Le *Configuration Manager* en tête de l'arbre appelé *Supervising Configuration Manager* est connecté à une RAM externe contenant les configurations. Néanmoins, le *Supervising Configuration Manager* peut agir comme un *Configuration Manager* ordinaire afin de soutenir des architectures composées de plusieurs unités XPP. Une application implémentée sur XPP est un ensemble composé de une ou plusieurs configurations. Les *Configuration Managers* peuvent individuellement configurer leur *Processing Array Cluster*, de manière indépendante et concurrente. Si une configuration est déjà chargée, une nouvelle peut être mise en cache du *Configuration Manager*, et ne nécessite pas de nouvelle requête de la part du *Supervising Configuration Manager* au moment où le *Processing Array Element* redevient disponible. Un tel préchargement peut être réalisé de différentes manières suivant l'application. Soit une requête de nouvelle configuration est lancée par une application en cours d'exécution, soit chaque *Configuration Manager* initie lui-même une requête. Le contrôle de la configuration est réalisé matériellement par une modélisation de type machine d'état. Pendant le préchargement de la configuration, toutes les ressources déjà configurées peuvent

immédiatement débuter le traitement des données. Cette stratégie de configuration est appelée *Wave-Reconfiguration*. La *Wave-Reconfiguration* est basée sur la nature orientée paquet du réseau de communication, ce qui assure aux paquets de données de ne pas être perdus au cas où la destination ne serait pas encore configurée.

D.4-2 ADRES : *Architecture for Dynamically Reconfigurable Embedded Systems*

Architecture : Adres (*Architecture for Dynamically Reconfigurable Embedded Systems*) [21] est un modèle d'architecture flexible composé d'un processeur VLIW et d'une architecture CGRA (*Coarse Grain Reconfigurable Array*) dédiée aux calculs complexes. Le processeur VLIW est composé typiquement de plusieurs FU (*Fonctionnal Units*) et de plusieurs bancs de registres (RF) multi-ports représentés à la figure 1-23. Afin de supprimer les ressources de contrôle pour l'exécution de boucles, les FU permettent les prédictions de branchement. La différence avec un VLIW classique est que celui-ci est utilisé en première ligne de la matrice reconfigurable. Quelques FU de la première ligne sont connectées à la hiérarchie mémoire selon le nombre de ports disponibles. Les opérations de chargement et de mémorisation sur ces FU facilitent l'accès aux données de la mémoire unifiée de l'architecture. Le contrôle de boucle est supprimé grâce au support des prédictions d'opérations. Les résultats issus des FU peuvent être soit enregistrés dans les RF distribuées (plus petites mais ayant moins de ports que les RF partagées), soit directement transmises aux FU voisines. Afin de garantir les temps de propagation, un registre est placé en sortie des FU.

Reconfiguration dynamique : dans Adres, la mémoire RAM de configuration permet le stockage local des configurations. Cependant, en contre-partie de délais supplémentaires, les configurations peuvent également être stockées dans la hiérarchie mémoire au cas où les mémoires locales seraient trop petites. Le comportement des unités de traitement et de contrôle est géré par la reconfiguration des connexions des multiplexeurs de sélection.

D.4-3 NEC DRP : *NEC Dynamically Reconfigurable Processor*

Architecture : le processeur reconfigurable dynamiquement NEC DRP (*NEC Dynamically Reconfigurable Processor*) [22] consiste en un ensemble d'éléments appelés *Tiles* (figure 1-24(a)). Chaque *Tile* est composé d'une matrice d'éléments de calculs appelés *Processing Elements* (figure 1-24(b)) d'une taille de huit lignes et huit colonnes. Ces *Processing Elements* fonctionnent au niveau octet et sont reliés par un réseau d'interconnexions programmables. Un *Processing Element* est composé d'une ALU (*Arithmetic and Logical Unit*), d'une ressource appelée *Data Management Unit* et de la ressource *Instruction* chargée de gérer les données de reconfiguration issues des mémoires de contexte. L'ALU peut effectuer des opérations arithmétiques et logiques classiques sur des données de huit bits. Le *Data Management Unit* sert à sélectionner, décaler, masquer, au niveau bit et au niveau octet, les données issues de l'ALU. La ressource *Instruction* gère localement la reconfiguration de l'ALU et du *Data Management Unit* ainsi que des interconnexions entre *Processing Elements*. La matrice de *Processing Elements* est entourée de blocs mémoires en périphérie, disponibles aussi bien pour la sauvegarde de configurations que pour une utilisation classique en mémoire de calcul.

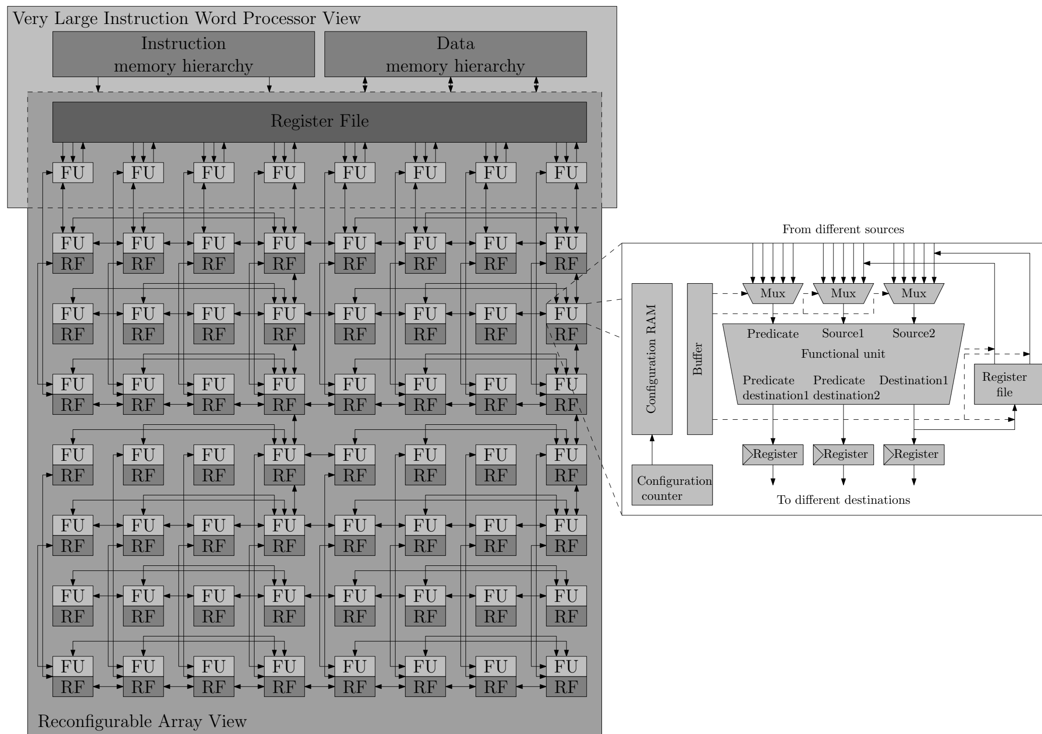


Figure 1-23 – **Architecture Adres**. *L'architecture Adres se compose d'un processeur VLIW* Core *et d'un CGRA (a). Celui-ci est constitué d'une matrice d'unités fonctionnelles reconfigurables dynamiquement représentée en (b). La reconfiguration dynamique d'une unité fonctionnelle est gérée localement par un compteur de configurations. La partie calcul de cette unité fonctionnelle est capable d'opérer sur des données provenant des unités fonctionnelles voisines.*

Ces mémoires ont une capacité de 256 mots de huit bits utilisables en FIFO pour certaines et d'une capacité de 8192 mots de huit bits pour les autres.

Reconfiguration dynamique : le processus de reconfiguration est géré par une ressource appelée *State Transition Controller* [23]. Celui-ci exploite la reconfiguration par multicontexte en générant en cours de fonctionnement les pointeurs de configuration. Chaque *Processing Element* dispose de sa mémoire de configuration dont le contenu sera sélectionné par le *State Transition Controller*. Une mémoire de contexte permet de contenir seize contextes. De plus, le *State Transition Controller* peut recevoir un maximum de quatre signaux de condition de branchement provenant des *Processing Element* permettant ainsi la gestion de la priorité des tâches.

E Plate-formes de développement dédiées aux architectures reconfigurables dynamiquement

Parmi les architectures reconfigurables dynamiquement que nous avons présentées précédemment, quelques unes telles que Adres ou XPP sont en fait des plate-formes d'architectures

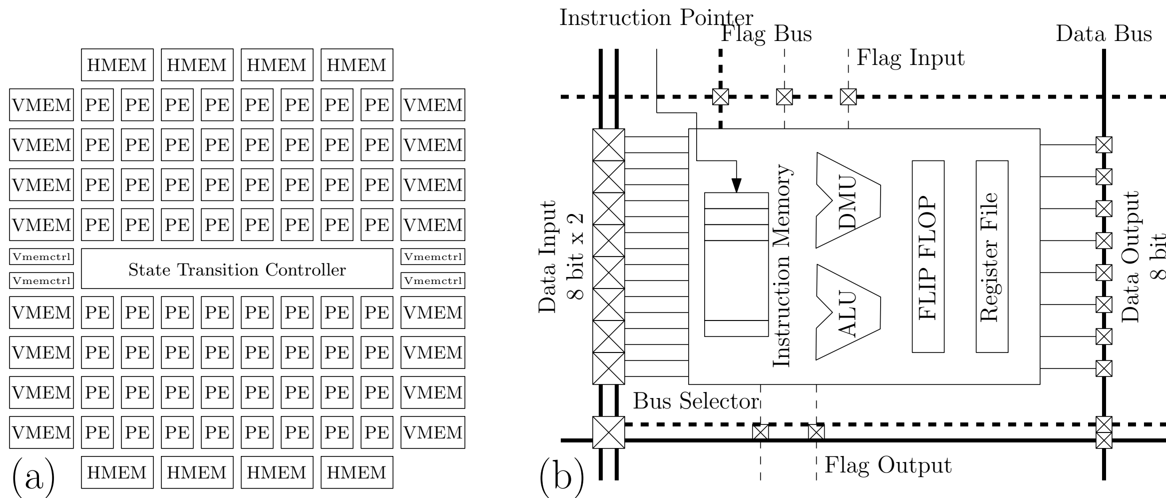


Figure 1-24 – ***Tile* composant l'architecture du NEC-DRP**. *L'architecture NEC-DRP est une matrice d'éléments de calculs (PE) entourées de mémoires locales (HMEM, VMEM). Un élément de calcul consiste en un processeur pouvant travailler sur des données de huit bits.*

qu'il est nécessaire de paramétrer afin de générer une architecture synthétisable spécifique à un domaine d'application. Ces spécifications concernent des paramètres tels que la taille des données à traiter, les tailles des mémoires, ou encore le nombre et le type d'unités de traitement pour ne citer que ceux-ci. Dans cette partie, nous présentons les outils de développement mis à disposition afin de procéder à la conception de ces modèles d'architectures, depuis la conception de l'architecture, jusqu'à la génération des flots de données de configuration qui permettront de mettre en œuvre les processus de reconfiguration dynamique propres à chaque architecture et adaptées à l'implémentation d'un algorithme.

E.1 Plate-formes dédiées au développement d'architectures grain épais

L'implémentation d'un algorithme sur architecture grain épais nécessite la mise en œuvre d'outils spécifiques (compilation, débogage, simulation). L'intérêt d'une plate-forme de développement est de proposer des outils capables de s'adapter aux particularités de l'architecture conçue. Nous présentons les plate-formes Dresc et ArchitectureComposer et plus particulièrement les différents outils qui les composent ainsi que leurs interactions.

E.1-1 Dresc : *Dynamically Reconfigurable Embedded System Compiler*

Dresc (*Dynamically Reconfigurable Embedded System Compiler*) [21] est une suite logicielle dédiée au développement d'une architecture de type Adres et de ses outils de compilation.

Outils dédiés à l'application : la première étape dans le processus de développement consiste en une description C de l'application à implémenter. Cette description est ensuite partitionnée par *Impact* afin d'en extraire les boucles qui requièrent des ressources matérielles

adaptées pour une exécution la plus efficace possible. Les ressources de la matrice reconfigurable sont adaptées à l'implémentation de ce type de boucle. Cependant, par souci d'efficacité, un flot de reconfiguration adapté se doit d'être correctement généré. Pour cela, un code émis en sortie de *Impact*, le *Lcode*, permettant une représentation intermédiaire de l'algorithme, sera utilisé par l'outil d'ordonnancement.

Outils dédiés à l'architecture : suivant les résultats issus de l'outil de partitionnement, une description de l'architecture cible est faite. Celle-ci est réalisée par l'intermédiaire d'un langage de description de type XML garantissant la génération rapide d'un modèle bas niveau de l'architecture pouvant être synthétisée par les outils commerciaux.

Outils dédiés au co-développement algorithme-architecture : suite à la description de l'architecture, un nouvel algorithme d'ordonnancement est généré afin de concevoir une réelle optimisation des mécanismes de parallélisation des cœurs de boucles. Les communications entre le processeur VLIW et la matrice reconfigurable sont alors automatiquement gérées et ordonnancées. Enfin, dernière étape de la conception, la co-simulation est rendue possible grâce à la description de l'architecture et du code source ordonnancé. Notons enfin l'existence d'un outil d'exploration capable de fournir le nombre de bits de configuration et la surface des ressources de reconfiguration utilisées.

E.1-2 ArchitectureComposer

Tout comme Dresc, ArchitectureComposer [24] est une plate-forme de développement issue des travaux de l'université de Erlangen en Allemagne. Cette plate-forme est dédiée à la conception d'architectures massivement parallèles de type WPPA et de leurs outils de compilation. Cette plate-forme se compose d'outils permettant de procéder à des phases d'exploration semi-automatique des caractéristiques de l'architecture facilitant ainsi la détermination du meilleur compromis implémentation matérielle-logicielle.

Outils dédiés à l'architecture : la première étape du développement consiste à décrire, de manière graphique, les chemins de données de contrôle et de calcul. Cela est fait par l'intermédiaire de librairies contenant des IP (*Intellectual Property*) de ressources paramétrables tels que des files de registres, des mémoires, des unités arithmétiques et/ou logiques ou encore des ressources de connexions. Cette description graphique permet à tout moment la génération automatique d'un modèle synthétisable de l'architecture basé également sur la spécification du jeu d'instructions et des règles grammaticales du générateur de code. À partir de la description de l'architecture, un modèle de machine d'état est également créé permettant ainsi la génération automatique d'un environnement de débogage et de simulation. Il est ainsi aisé de valider les fonctionnalités de l'architecture à différents niveaux d'abstraction comme une description de pipeline ou de jeu d'instructions.

Outils dédiés au co-développement algorithme-architecture : après simulation, un compte rendu analysé par *ArCoExplorer* permet d'explorer le meilleur compromis entre

une implémentation matérielle et logicielle spécifiquement au domaine d'application visé en fonction de la taille du code, du temps d'exécution et des ressources matérielles disponibles.

E.2 Plate-formes dédiées au développement d'architectures grain fin

La conception d'une architecture grain fin nécessite l'utilisation d'outils différents des architectures grain épais. À notre connaissance, une architecture grain fin ne permet que des implémentations matérielles d'algorithmes, ce qui nécessite des processus de configuration différents d'une implémentation logicielle. Madeo et AMDREL sont deux plate-formes qui permettent la conception d'architectures grain fin. Nous présentons ici les outils utilisés par ces deux plate-formes.

E.2-1 Madeo

Madeo [25] est une plate-forme de développement permettant la synthèse logique sur architecture reconfigurable de type FPGA. Madeo est composé de Madeo-Fet (Madeo *Front-End Tool*) et de Madeo-Bet (Madeo *Back-End Tool*) [26].

Outils dédiés à l'architecture : Madeo-Fet permet la génération d'un modèle synthétisable d'architecture FPGA basé sur une description *Smalltalk*. Depuis la description haut niveau de l'architecture, Madeo-Fet supporte la prospection architecturale et le prototypage rapide de FPGA. La modélisation de FPGA commerciaux, tels que la famille Virtex de Xilinx et la modélisation de FPGA plus prospectifs tels que LPPGA [27], ont été réalisés avec succès.

Outils dédiés au co-développement algorithme-architecture : la synthèse Madeo se traduit par la construction de tables de vérité qui seront ensuite converties sous un format binaire pouvant être utilisé par l'outil SIS de Berkeley [28]. Le résultat, sous forme de graphiques hiérarchiques de modules logiques optimisés, peut ensuite être directement utilisé par Madeo-Bet. Madeo-Bet comprend une suite d'outils incluant le placement/routage de l'application synthétisable, ainsi que l'allocation de ressources nécessaires à la génération du flot de données de configuration. Le spectre d'architectures visées par Madeo-Bet s'étend également au delà des FPGA, puisque celui-ci a été utilisé pour cibler des architectures à base de chemins de données reconfigurables ou bien encore des nano-technologies émergentes telle que l'architecture NASIC en partenariat avec l'Umass (*University of Massachusetts*) [29], [30].

E.2-2 AMDREL

AMDREL [31] est une plate-forme de développement d'architectures FPGA génériques similaire en termes de fonctionnalités à Madeo. Les outils qui composent cette plate-forme ont été développés par l'Université de Thrace en Grèce et s'appuient sur des outils issus d'universités comme SIS de Berkeley et VPR (*Virtual Place and Route*) de Toronto.

Outils dédiés à l'application : la synthèse logique constitue la première étape dans le processus de conception d'architectures avec AMDEL. L'outil dédié à la synthèse logique, DIVINER (*Democritus University of Thrace RTL Synthesizer*), permet de convertir la description VHDL (*Very High Speed Integrated Circuit Hardware Description Language*) d'une application en une liste de composants basiques au format EDIF. Cependant, DIVINER ne permettant pas d'effectuer une optimisation booléenne, il est nécessaire de faire exécuter cette étape par SIS [28] comme cela est le cas avec Madeo. Le fichier de description issu de SIS est une liste de portes logiques. À partir de cette description, DRUID (*DemocRitus University of Thrace*) est capable de générer une configuration de LUT et de bascules indépendamment de l'architecture FPGA qui pourra en supporter l'implémentation.

Outils dédiés à l'architecture : la description de l'architecture FPGA est générée sous un format de type XML par l'outil DUTYS (*Democritus University of Thrace Architecture file generator synthesizer*). Depuis cette description, compatible avec l'outil de placement/routage VPR [32], il est possible de procéder au placement/routage de l'application visée. Pour fonctionner, DUTYS requiert quelques paramètres tels que le nombre de bloc logiques, le nombre d'entrées/sorties ou encore la taille des bus de communications entre autres. De plus, parallèlement à la description XML de l'architecture, une estimation de la consommation énergétique est également produite.

Outils dédiés au co-développement algorithme-architecture : TVPack et VPR sont des outils développés par l'université de Toronto dont l'utilisation permet d'obtenir la configuration exacte des ressources logiques et des ressources de routage qui serviront à la génération du flot de données de configuration d'une application sur un FPGA. VPR est basé sur un algorithme d'optimisation de placement appelé *Simulated Annealing* [33] et réputé pour son efficacité tout comme l'algorithme de routage basé sur l'algorithme *Pathfinder Negotiated Congestion* [34, 35]. Enfin, dernière étape de la conception, la génération du flot de données de configuration est assuré par DAGGER (*Democritus University of Thrace eFPGA bitstream generator*) conformément à la description de l'architecture fournie et des fichiers de placement-routage générés. DAGGER est capable de supporter à la fois la reconfiguration dynamique et la reconfiguration partielle d'une architecture. DAGGER intègre également des méthodes de réallocation, de défragmentation, et de compression des données de configuration et enfin de détection d'erreurs de type CRC (*Cyclic Redundancy Checking*).

F Synthèse

Il est possible d'aborder les architectures reconfigurables sous différents aspects. Nous avons choisi de les présenter selon leur méthode de reconfiguration et la granularité de leurs unités de calculs.

Tout d'abord, nous avons présenté des architectures mono-contexte. Les ressources nécessaires à l'implémentation de ces architectures sont dimensionnées au plus juste de manière à être le plus efficace possible en termes de surface de silicium et de consommation. Le principal inconvénient, au niveau reconfiguration, vient du fait que la reconfiguration d'une zone néces-

site l'arrêt des traitements de données en cours de réalisation comme cela est le cas pour les FPGA Virtex. Dans le cas de DART, les configurations sont suffisamment petites pour que la reconfiguration se fasse rapidement. En revanche, dans le cas de l'AT40K, la reconfiguration peut nécessiter plusieurs millisecondes ce qui est, pour des applications actuelles, beaucoup trop important.

Nous avons ensuite présenté les architectures multi-contexte. Le principal avantage de ces architectures réside dans la possibilité de procéder à une reconfiguration quasi immédiate grâce à l'implémentation locale des différentes configurations. La plupart des architectures présentées permettent la modification du contexte contenu dans ces mémoires locales qui peuvent, pour certaines d'entre elles, sauvegarder plus de huit contextes différents. Cela implique, en termes d'organisation des mémoires, une redondance inefficace de la sauvegarde des différents contextes et par conséquent une consommation d'énergie plus importante. De plus, l'impact de la reconfiguration sur les caractéristiques de surface et de consommation sont souvent ignorées par ces architectures.

Nous avons également présenté des plate-formes de développement dédiées à la conception d'architectures reconfigurables dynamiquement. Les architectures ainsi conçues sont accompagnées alors d'outils nécessaires à leur bonne exploitation tels que des compilateurs, des simulateurs, des outils de placement-routage ou encore de synthèse. Cependant, selon la plate-forme choisie, l'architecture générée aura les caractéristiques de calcul et de topologie propres à la plate-forme.

Dans les chapitres suivants, nous présentons la plate-forme de développement Mozaïc. Mozaïc a été développée de manière à permettre la conception d'architectures reconfigurables dynamiquement par la génération automatique des mécanismes de reconfiguration indifféremment des particularités architecturales (topologie, granularité, mécanismes de reconfiguration) des unités de traitement implémentées. L'intérêt des architectures reconfigurables dynamiquement est souvent critiqué. La principale critique vient du fait que l'implémentation des mécanismes de reconfiguration requiert parfois plus de surface et plus d'énergie que n'aurait nécessitée une implémentation statique. Mozaïc est basée sur l'intégration d'unités de traitement développés au préalable. L'intérêt de Mozaïc est de s'intégrer aux plate-formes existantes afin d'apporter ses compétences en matière de gestion de la reconfiguration. Pour cela, nous avons développé des concepts de reconfiguration dynamiques flexibles et paramétrables. La reconfiguration dynamique mise en œuvre par Mozaïc intègre des concepts permettant la gestion partielle et flexible des unités de traitement implémentées aussi hétérogènes quelles soient en termes de granularité et de gestion de la reconfiguration dynamique.

Chapitre

2

Modèle Générique de Reconfiguration Dynamique mis en œuvre par Mozaïc

Résumé : Dans ce chapitre, nous présentons le modèle générique de reconfiguration dynamique mis en œuvre par la plate-forme de développement Mozaïc. Ce modèle est basé sur le concept des architectures multi-contexte. L'optimisation consiste à minimiser les ressources de mémorisation qui constituent le principal point faible de cette technique. Le concept tel que nous l'avons développé est capable de gérer de manière homogène des ressources reconfigurables hétérogènes. Cette hétérogénéité peut s'exprimer aussi bien en termes de connectivité qu'en termes de mécanismes de reconfiguration dynamique.

Sommaire

A Introduction 42
- A.1 Présentation globale des concepts de reconfiguration dynamique mis en œuvre par Mozaïc 42
- A.2 Paramètres de connectivité 43
- A.3 Paramètres des domaines et ressources de reconfiguration 44

B Concept DUCK 45
- B.1 Objectifs 45
- B.2 Reconfiguration dynamique et mécanismes de préemption 46
- B.3 Homogénéisation des processus de reconfiguration dynamique 47

C DyRIBox : unités d'interconnexion 53
- C.1 Routeurs d'interconnexion dans les architectures reconfigurables 53
- C.2 Architecture d'une DyRIBox 55
- C.3 Principe de reconfiguration et principe de contrôle des multiplexeurs 56
- C.4 De la commutation directe de circuit vers la commutation par paquet 56

D DyRIOBloc : ressources d'entrée/sortie reconfigurables 58

E Ressources de contrôle et de gestion 58
- E.1 *CtxtScheduler* : gestion et ordonnancement des contextes 59
- E.2 *DomainCtrl* : contrôleur de domaine 61
- E.3 *CtxtMemCtrl* : Contrôleur de mémoire de configuration 64

F Synthèse 68

A Introduction

La conception d'une architecture reconfigurable dynamiquement implique la mise en œuvre de mécanismes de reconfiguration dédiés aux ressources de calculs implémentées. Les architectures conçues de nos jours intègrent de plus en plus de ressources toujours plus hétérogènes. Par exemple, les Virtex 5 intègrent aussi bien des LUT, qu'un processeur embarqué, que des mémoires RAM ou encore des unités de traitement (multiplieurs, DSP...). Dans le contexte de la conception d'architecture, l'ajout d'une nouvelle entité de calcul implique l'intégration totale des caractéristiques de connectivité et de contrôle de l'architecture hôte. L'objectif de Mozaïc est de permettre la conception d'architectures reconfigurables dynamiquement hétérogènes par la virtualisation des caractéristiques des ressources de traitement utilisées. Pour cela, nous avons développé un modèle de reconfiguration dynamique paramétrable et suffisamment flexible pour permettre la conception de tous types d'architectures. Ce chapitre va s'attacher à présenter les différents concepts que nous avons développés.

A.1 Présentation globale des concepts de reconfiguration dynamique mis en œuvre par Mozaïc

La figure 2-1 présente une vue globale et générique d'une architecture générée par la plateforme Mozaïc. D'une manière générale, les différentes ressources qui constituent une architecture peuvent être regroupées en trois types d'unités. Les unités d'entrée/sortie (Uio), les unités de traitement ou de mémorisation (Ut) et les unités d'interconnexion (Ui). Si les unités d'entrée/sortie, ainsi que les unités d'interconnexion reposent le plus souvent sur des concepts simples, et maîtrisés ; cela n'est pas le cas des unités de traitement. La diversité des unités de traitement utilisées pour l'exécution d'algorithmes de plus en plus complexes, rend les architectures hétérogènes. Cette hétérogénéité complexifie la gestion de l'architecture et les intéractions entre Ut. Le rôle de Mozaïc est de prendre en considération ces différences par l'analyse des paramètres de chaque Ut, de manière à procéder ensuite à la génération automatique :

1. des unités d'entrée/sortie,
2. des unités d'interconnexion,
3. des ressources de reconfiguration et de leur contrôleur.

La génération automatique des ressources de reconfiguration se traduit par l'introduction de nouvelles unités (Ur) chargées de gérer localement les reconfigurations de chacune des unités (Uio, Ui, Ut). Ces unités de reconfiguration sont, en quelque sorte, des mémoires locales qui contiennent une configuration destinée à l'unité avec laquelle chacune de celle-ci est connectée. Les unités de reconfiguration sont également reliées entre elles de manière à pouvoir propager les configurations de la première Ur, jusqu'à la dernière. Selon le nombre d'unités de traitement, d'interconnexion et d'entrée/sortie, le nombre d'unités de reconfiguration augmente, de sorte que la propagation des configurations ne soit plus suffisamment rapide pour maintenir les contraintes temporelles imposées par les applications. Afin de remédier à cela, des zones de reconfiguration sont créées (domaines) et contrôlées individuellement (Cc). Un domaine est donc constitué d'un ensemble d'unités de reconfiguration formant une seule

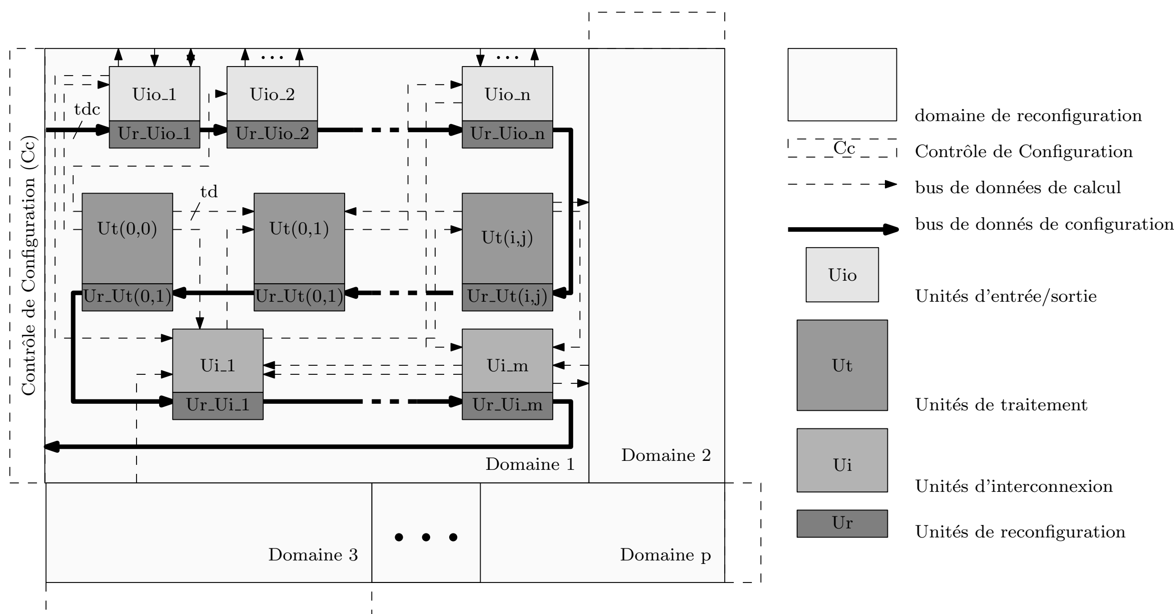


Figure 2-1 – **Vue simplifiée des différentes ressources qui constituent une architecture issue de la plate-forme Mozaïc**. *D'une manière générale, cette architecture est composée d'unités d'entrée/sorties (Uio), d'unités de traitement (Ut) et d'unités d'interconnexion (Ui).*

chaîne de propagation. Nous allons maintenant détailler l'ensemble des paramètres à prendre en compte pour la génération automatique des ressources de reconfiguration.

A.2 Paramètres de connectivité

unités d'interconnexion (Ui) : les unités d'interconnexion générées par Mozaïc doivent accepter tout type de connexion. Par conséquent, celles-ci sont paramétrables en nombres de ports d'entrée, en nombre de ports de sortie ainsi qu'en nombre de ports bidirectionnels. De plus, selon les applications devant être implémentées, la taille des données qui seront traitées peut varier. Cela constitue également un paramètre. Il doit être également possible de modifier un schéma de connexion pendant le traitement d'un algorithme, les unités d'interconnexion doivent donc être reconfigurables dynamiquement. La mise en œuvre de la reconfiguration implique des données de reconfiguration qu'il est important de pouvoir réduire si besoin. Les unités d'interconnexion peuvent donc accepter une restriction dans le schéma des connexions en interdisant une entrée de se connecter sur une sortie par exemple. Enfin, les unités d'interconnexion permettent la conception de topologies de calcul diverses, telle qu'une topologie hiérarchique comme celle présentée par la figure 2-2.

Unités d'entrée/sortie (Uio) : les unités d'entrée/sortie ont pour rôle de permettre les communications des Ut avec le monde extérieur. La nature de ces communications va déterminer le nombre, la taille et la nature des ports de communication. De plus, les données qui devront transiter par les Uio sont destinées ou proviennent de ressources fonctionnant

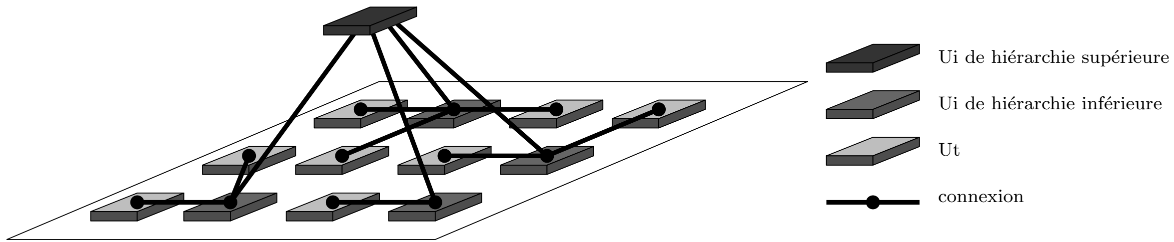


Figure 2-2 – **Modélisation des hiérarchies d'interconnexion**. *Dans cet exemple, deux niveaux de hiérarchie sont présentés. Les Ut proches les une des autres sont interconnectées par l'intermédiaire d'Ui de niveau inférieur, et peuvent communiquer avec les Ut plus éloignées par l'intermédiaire des Ui de niveau supérieur.*

selon un mécanisme et une horloge différente de l'architecture concernée. Afin de prévenir les problèmes de compatibilité, des mémoires tampon paramétrables en taille sont présentent dans les Uio et permettent des adaptations de protocoles ou de tension par exemple.

A.3 Paramètres des domaines et ressources de reconfiguration

Unités de reconfiguration (Ur) : les unités de reconfiguration sont générées de manière à réaliser une interface entre les ressources de reconfiguration produites par Mozaïc et les Ut implémentées au sein d'un même domaine. Pour cela, les Ur doivent stocker les configurations localement puis les transférer vers l'Ut concernée selon un protocole faisant partie de la liste des paramètres à spécifier à Mozaïc. Afin de gérer la reconfiguration de manière efficace, il est nécessaire de connaître le nombre de bits et de cycles nécessaires à la reconfiguration, ainsi que le nombre et la taille des ports réservés à cet usage. La génération de chaque Ur nécessite également la spécification de la taille du bus qui relie l'ensemble des Ur du domaine. La spécification du protocole de reconfiguration n'est pas nécessaire pour les unités d'interconnexion ni les unités d'entrée/sortie étant elles-même générées par Mozaïc. Les autres paramètres concernant la reconfiguration dynamique et la génération des Ur adaptés aux Ui et Uio ont été décrits précédemment.

Ressources de reconfiguration : la gestion d'un domaine de reconfiguration permet plusieurs ajustements en termes de flexibilité. La taille du bus de configuration reliant les Ur est un de ces paramètres. Plus la taille de celui-ci est grande, plus rapide seront les processus de propagation. Il est également possible de réunir plusieurs domaines de reconfiguration de manière à ce qu'ils soient gérés par un seul contrôleur. Chaque domaine peut être reconfiguré partiellement de façon à permettre le parallélisme asynchrone de tâches. Une tâche peut être vue comme une configuration particulière, ou un contexte particulier, des ressources de traitement et d'interconnexion qui permettent son implémentation. Des mécanismes de contrôle on été mis en œuvre par Mozaïc afin d'augmenter l'efficacité de la reconfiguration. Parmi ces mécanismes, notons la gestion des processus de préemption et de la priorité d'implantation des tâches (configurations ou contextes). La gestion des tâches prioritaires implique la spécification de paramètres tel que : la taille de la mémoire tampon qui servira à tem-

poriser l'ordonnancement des différentes interruptions, les différents niveaux d'interruptions ainsi que le nombre d'interruptions par niveau. Un autre paramètre concerne la taille de la mémoire de configuration qui détermine le nombre de contextes pouvant être implémentés sur l'architecture.

Grâce à l'introduction des concepts de reconfiguration dynamique appliquée dans Mozaïc, nous pouvons concevoir des architectures reconfigurables couvrant des paradigmes allant des FPGA aux réseaux dynamiques de processeurs tels que WPPA, en passant par des processeurs reconfigurables comme DART. La partie qui suit va s'attacher à montrer plus précisément de quelle manière ces différents concepts sont intégrés dans les ressources de reconfiguration générées par Mozaïc.

B Le concept DUCK : support matériel pour la reconfiguration dynamique

Dans cette section, nous présentons en détail les particularités architecturales et conceptuelles de l'unité de reconfiguration que nous avons appelée DUCK (*Dynamic Unifier and reConfiguration blocK*).

B.1 Objectifs

La reconfiguration dynamique par multi-contexte permet une accélération des processus de reconfiguration plutôt significative, au détriment de l'augmentation des ressources de mémorisation. Cela contribue à une inefficacité énergétique incompatible avec les contraintes actuelles de développement d'architecture. La méthode que nous présentons dans ce chapitre repose sur l'utilisation d'une seule mémoire de contexte parallèlement aux ressources de configuration. Cela implique par conséquent l'utilisation de solutions architecturales afin de pouvoir maintenir les contraintes temporelles et la flexibilité requises par les applications actuelles. Cela se traduit par la séparation du chemin de données de configuration et des ressources de configuration à proprement parlé (figure 2-3). Une ressource dédiée à cette tâche a été développée ; baptisée DUCK, celle-ci permet de faire l'interface entre le chemin de reconfiguration et l'unité à reconfigurer. Un DUCK est une mémoire de configuration locale par laquelle passe le bus de configuration. Cette mémoire locale est une mémoire parallèle dans laquelle est stockée la future configuration. Le DUCK est aussi bien dédié à une unité de traitement qu'à une unité d'interconnexion ou d'entrée/sortie. La configuration stockée dans le DUCK peut être acheminée vers la zone de configuration de l'unité concernée via l'interface de configuration de cette unité.

Prenons par exemple l'implémentation d'un accélérateur matériel de type FPGA embarqué (figure 2-4). L'implémentation montre une matrice d'unités d'interconnexion, schématisée par un hexagone ⓐ, chacune associée avec un bloc logique configurable, schématisée par un polygone de même couleur ⓑ. Chaque unité, qu'elle soit de type interconnexion ou logique, se voit associée à un DUCK. L'ensemble des DUCK crée ainsi le chemin de configuration, schématisé par une flèche noire épaisse. Dès qu'une reconfiguration est requise, chaque DUCK permute son contexte depuis ses registres internes vers les registres de contrôle contenus dans

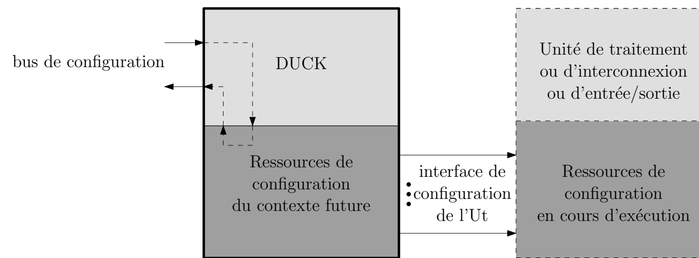


Figure 2-3 – **Synoptique d'un DUCK simplifié**. *Un DUCK est une mémoire de configuration locale parallèle dans laquelle est stockée la future configuration. Lors d'un processus de reconfiguration, la configuration stockée dans le DUCK peut être acheminée vers les ressources de configuration de l'unité concernée.*

l'unité concernée. Les configurations sont permutées à travers le chemin de données schématisé par des flèches fines. Une fois la configuration permutée, il est possible d'extraire l'ancienne configuration par le chemin de configuration. De manière synthétique, nous pouvons affirmer que l'introduction du concept DUCK tend à résoudre trois problématiques majeures dans le développement d'architectures reconfigurables dynamiquement :

1. l'accélération des processus de reconfiguration,
2. la gestion des mécanismes de préemption,
3. la gestion homogène de la reconfiguration dynamique quelles que soient les ressources reconfigurables implémentées.

B.2 Reconfiguration dynamique et mécanismes de préemption : principe de fonctionnement

L'utilisation des registres DUCK permet une exécution parallèle des processus de reconfiguration et des processus de traitement. Autrement dit, l'acheminement des contextes futurs se fait pendant les phases de calculs sans avoir à arrêter les traitements en cours. Pour cela, le concept DUCK repose sur le principe des registres *scanpath* utilisés dans les techniques de DFT (*Design For Test*). L'utilisation de tels registres permet à tout moment de configurer les connexions de sorte que l'ensemble des registres concernés constitue un seul et unique registre à décalage. Cela permet, dans les techniques de DFT, d'extraire un contexte afin de détecter d'éventuelles erreurs dans les calculs dûes à des défaut de fonctionnement. Nous avons apporté quelques modifications à cette technologie, aujourd'hui maîtrisée, afin de la rendre suffisamment flexible et adaptée aux mécanismes de reconfiguration dynamique. De plus, cela permet une minimisation des données de configuration en supprimant les données d'adressage. Le concept *scanpath* est exploité lors des phases de propagation qui consistent à acheminer les nouveaux contextes dans les registres DUCK. En revanche, des mécanismes ont été introduits pour l'échange entre le contexte "futur" contenu dans le DUCK et le contexte "passé" à modifier dans une ressource reconfigurable. Les registres DUCK connaissent quatre phases dans la reconfiguration (I, II, III et IV figure 2-5 et figure 2-6) réalisées grâce à trois configurations de registres DUCK (A, A', B, C). La configuration A correspond à une configuration de propagation de contexte. Les registres DUCK sont interconnectés en *scanpath* de

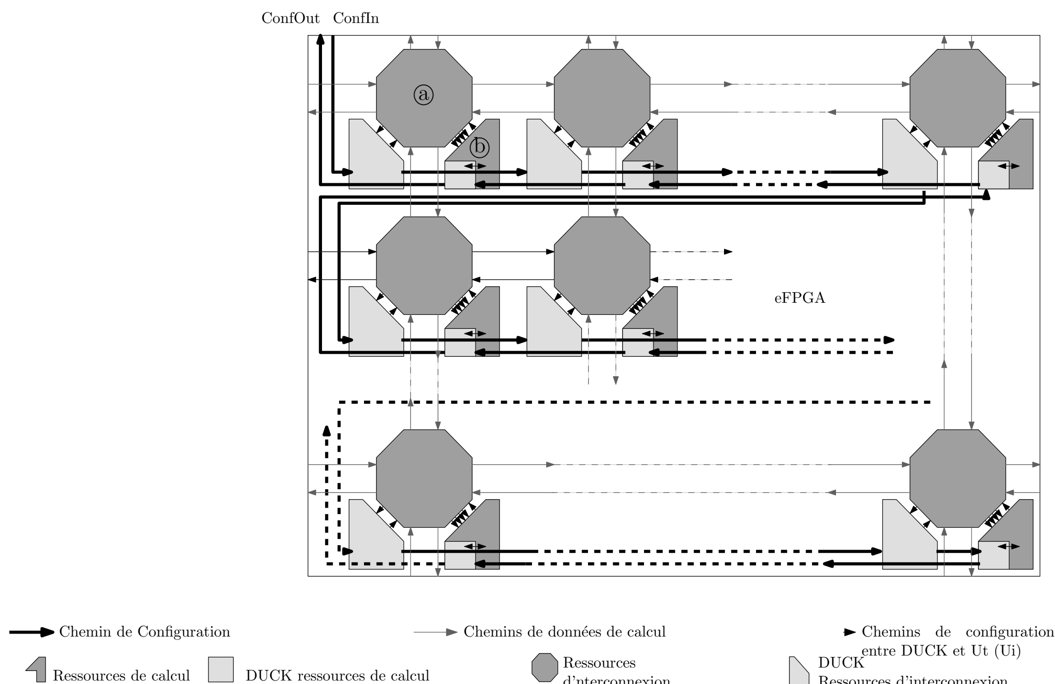


Figure 2-4 – **Exemple de l'utilisation du réseau de DUCK dans un accélérateur matériel**. *Le réseau qui constitue le chemin de configuration est implémenté de sorte que celui-ci passe par les DUCK dédiés aux unités d'interconnexion avant de constituer le chemin de configuration dédié au unités de traitement.*

manière à propager le contexte "futur". La configuration A' est, d'un point de vue architectural, identique à la configuration A, mais correspond à une phase de préemption où le contexte "passé" est sauvegardé en mémoire par la méthode de *scanpath*. La configuration B est celle utilisée pendant la phase de reconfiguration où les registres DUCK sont interconnectés aux registres de configuration des ressources. La nouvelle configuration prend la place de la configuration "passée". Enfin, la configuration C qui succède aux configurations A et A' si celle-ci est nécessaire, permet l'attente de la fin d'exécution de la tâche avant la reconfiguration.

B.3 Homogénéisation des processus de reconfiguration dynamique

Dans le domaine de la reconfiguration dynamique, plusieurs modes de reconfiguration sont possibles. Dans les FPGA Virtex de chez Xilinx, la reconfiguration se fait en adressant des mémoires RAM de configuration [11], alors que dans le processeur reconfigurable DART [12], les configurations sont changées par l'intermédiaire de registres de configuration. Le concept DUCK permet une adaptation des différentes méthodes de reconfiguration et réalise ainsi une interface entre chaque ressource reconfigurable et un processus de reconfiguration homogène à toute l'architecture. Le reste de la section détaille les trois modes de reconfiguration considérés.

Reconfiguration séquentielle : un DUCK est conçu pour s'adapter à la ressource à laquelle il est lié. L'acheminement d'une nouvelle configuration dans un DUCK adapté à

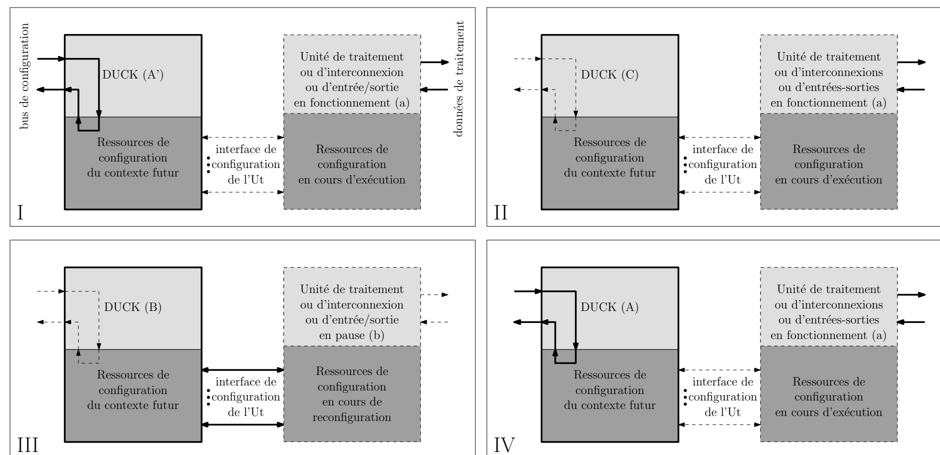


Figure 2-5 – **Phases de reconfiguration d'un DUCK et interaction avec une unité de traitement ou d'interconnexion**. *Quatre phases sont nécessaires à la reconfiguration d'une unité de traitement ou d'interconnexion. La phase I représente la propagation d'un nouveau contexte dans le DUCK. La phase II représente la phase d'attente de la fin du traitement. La phase III représente la phase de reconfiguration à proprement parlé. Le contexte contenu dans le DUCK est propagé (ou échangé si l'on se place dans le cadre d'une phase de reconfiguration suite à une interruption) dans l'unité concernée. La phase IV représente la phase d'extraction en cas de préemption.*

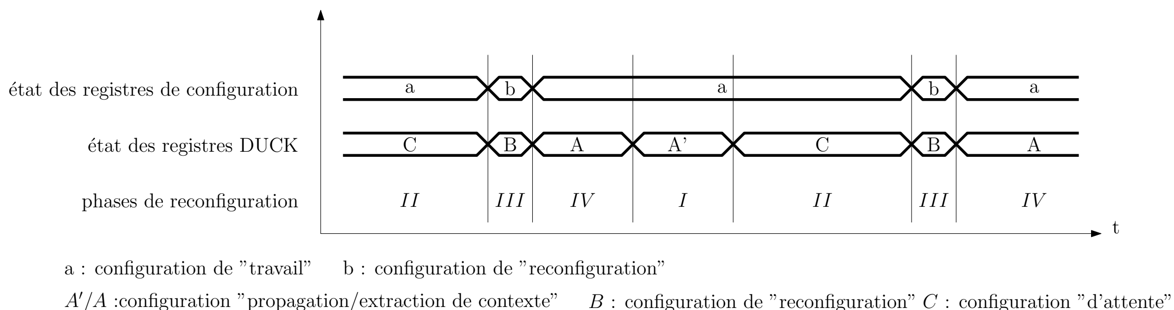


Figure 2-6 – **Logigramme représentant les phases de reconfiguration à l'aide d'un DUCK**. *En dehors de la phase de reconfiguration à proprement parler, les registres du DUCK préparent et extraient les configurations parallèlement aux éléments de stockage de configuration*

une reconfiguration série se fait en trois étapes. La première (figure 2-7(a)) est l'étape de repos. Cela veut dire qu'une nouvelle configuration est présente dans le DUCK et prête à être configurée. La seconde étape, (figure 2-7(b)) schématise l'acheminement d'une nouvelle configuration dans le DUCK. Comme cela a déjà été expliqué, les registres du DUCK sont connectés en *scanpath* de sorte que les configurations sont acheminées cycle par cycle. Au cas où une phase de préemption est nécessaire, le processus reste le même pour extraire la configuration en ayant prie soins de prévoir un port de communication dédié entre le DUCK et la ressource. La dernière étape, propre à ce mode de reconfiguration (figure 2-7(c)), montre la manière dont les registres DUCK sont interconnectés avec les registres de configuration.

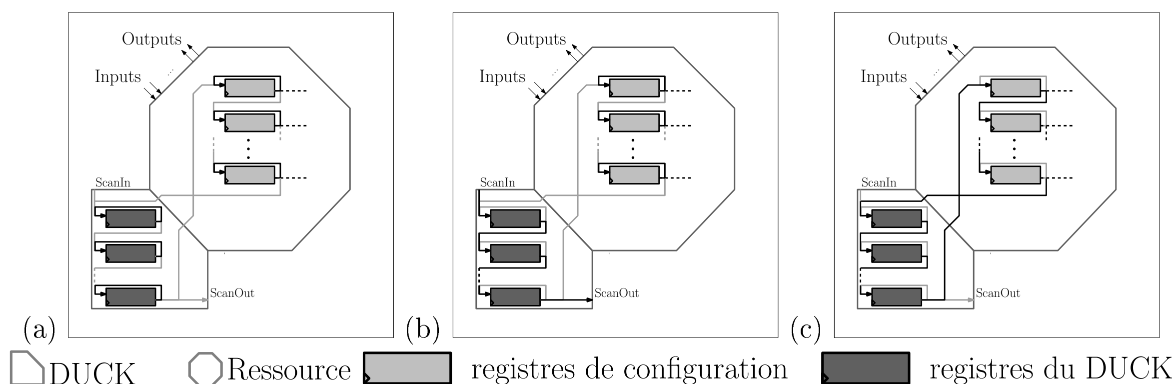


Figure 2-7 – **Phases de reconfiguration d'une ressource par l'intermédiaire d'un DUCK en mode séquentiel**. *Trois configurations différentes permettent d'effectuer une reconfiguration en quatre phases. La première phase constitue la phase de "repos" (configuration a), le DUCK n'est pas sollicité. La deuxième phase (configuration b) est la phase de propagation d'un nouveau contexte. La troisième phase (configuration c) correspond à la reconfiguration de la ressource connectée au DUCK. Enfin la dernière phase correspond à l'extraction du contexte précédent dont la configuration du DUCK est identique à la phase de propagation. Les connexions actives sont représentées par une flèche noire. Les connexions inactives sont représentées par une flèche grise.*

Dans notre exemple, seul le dernier registre DUCK est connecté à l'entrée des registres de configuration, permettant une reconfiguration en n cycles, n étant le nombre de registres de configuration nécessaires à la ressource.

Par exemple, si nous considérons une architecture de FPGA embarqué tel que celui de la figure 2-4, celui-ci implémente une série de bloc logique (figure 2-8) dont le temps de reconfiguration est égal au temps nécessaire à la propagation du prochain contexte à travers toute l'architecture. Les registres de configuration (aire Ⓐ de la figure 2-8) permettent de paramétrer certaines ressources en fonction de l'utilisation du bloc logique en mode séquentiel ou combinatoire ou encore de choisir l'entrée utilisée pour le calcul de retenue. Le réseau de multiplexeurs (aire Ⓑ) permet l'utilisation des ressources de LUT en mode RAM. Les ressources nécessaires au calcul de la retenue pour les opérations arithmétique sont représentées dans l'aire Ⓒ. Enfin, la LUT utile à l'implémentation de la fonction logique est représentée aire Ⓓ avec sa bascule.

Le chemin de configuration d'un bloc logique passe par l'ensemble des registres de configuration et des registres de LUT. Dans cet exemple, 19 cycles d'horloges sont nécessaires au processus de reconfiguration pour acheminer l'ensemble du contexte depuis les DUCK (aire Ⓔ figure 2-8) jusqu'aux registres de configuration. Ces 19 cycles correspondent à 19 bits nécessaires à la configuration des 16 bits de la LUT à quatre entrées, et de trois registres. Les trois registres permettent de spécifier la pré-configuration du registre de sortie ainsi que l'utilisation de celui-ci en sortie du bloc logique et enfin la validation de l'entrée *cin* comme entrée de LUT. Ce processus étant exécuté en parallèle pour l'ensemble des DUCK, cela se traduit, pour un FPGA embarqué composé de $n \times m$ blocs logiques, à un temps de propagation t_a valant $t_a = n \times m \times 19$ cycles d'horloges pour l'ensemble du FPGA. Grâce à l'utilisation du

concept DUCK l'ensemble des cellules logiques est alors configuré en $t_r = 19$ cycles d'horloge.

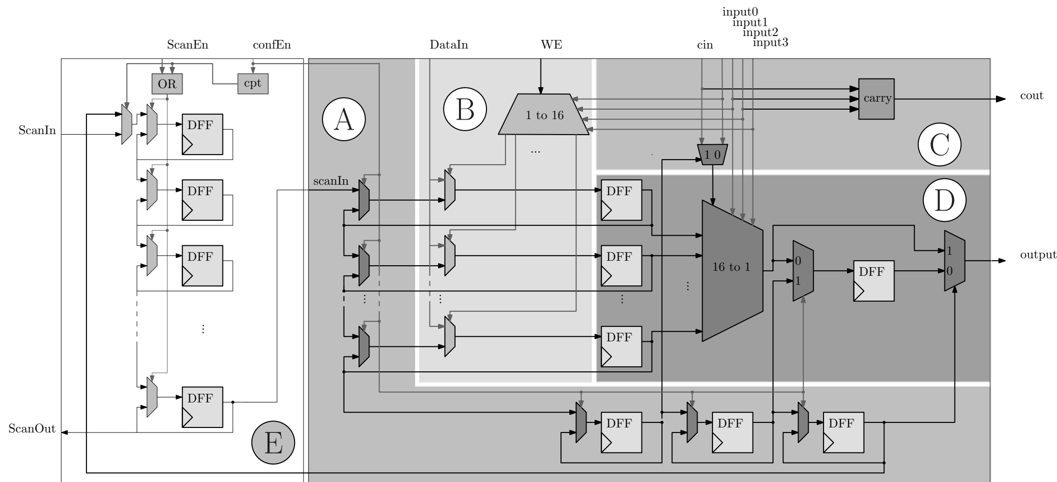


Figure 2-8 – **Exemple d'implémentation d'un DUCK dédiée à une ressource reconfigurable séquentielle**. *Dans cet exemple, un DUCK à reconfiguration séquentielle représenté zone E, est connecté à un bloc logique de FPGA. Le bloc logique est divisé en cinq zones. La zone A regroupe les registres de configuration, la zone B contient les ressources nécessaires à l'utilisation du bloc logique en RAM, la zone C contient les ressources de calcul de retenue et la zone D les ressources de LUT.*

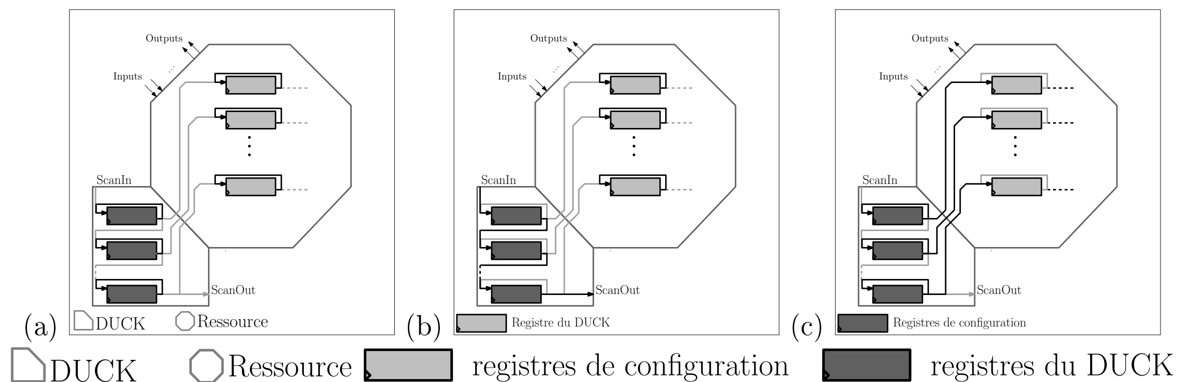


Figure 2-9 – **Phases de reconfiguration d'une ressource par l'intermédiaire d'un DUCK en mode parallèle**. *Le passage d'une configuration à une autre par la méthode parallèle se fait en un cycle d'horloge. Chaque registre de configuration est connecté à un registre DUCK contenant le mot de configuration du prochain contexte.*

Reconfiguration parallèle : l'association du chemin de reconfiguration à la manière *scanpath* avec le concept DUCK permet au système de se reconfigurer en un cycle si nécessaire. Cela est pleinement fonctionnel avec une reconfiguration parallèle. L'acheminement d'une nouvelle configuration dans un DUCK adapté à une reconfiguration parallèle s'exécute selon le même principe que pour les DUCK à reconfiguration séquentielle en ce qui concerne les deux premières étapes (figure 2-9(a) et (b)). La dernière étape, propre au mode de reconfiguration parallèle (figure 2-9(c)) montre la manière dont les registres DUCK sont interconnectés

avec les registres de configuration. Dans notre exemple, chaque registre de configuration est directement connecté à son registre DUCK, permettant une reconfiguration en un cycle. Ce type de reconfiguration peut être utilisé par le processeur reconfigurable DART (figure 2-10). Dans cet exemple, un registre permet la configuration d'une UAL de DART. La configuration permet le paramétrage de fonctionnement du SIMD (*Single Instruction Multiple Data*) ou du SWP. Le DUCK adapté à une reconfiguration parallèle est représenté par l'aire Ⓐ de la figure 2-10, les registres DUCK sont directement connectés aux registres de configuration représentés dans l'aire Ⓑ. Enfin, ces registres commandent la configuration des unités de traitement schématisés aire Ⓒ.

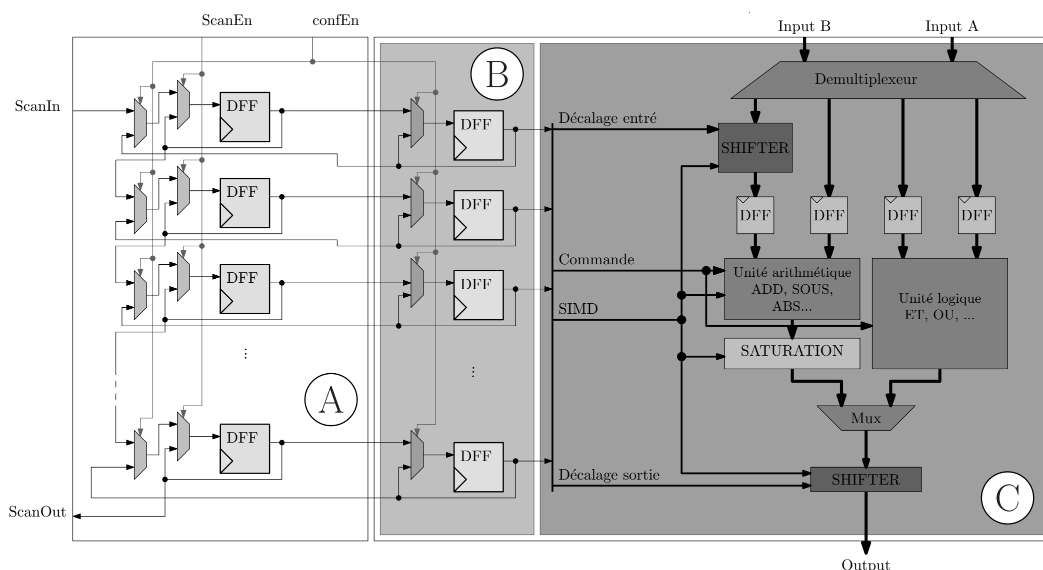


Figure 2-10 – **Exemple d'implémentation d'un DUCK dédié à une ressource reconfigurable parallèle**. *Une unité fonctionnelle d'un DPR de DART est capable de se reconfigurer de manière à pouvoir effectuer des calculs différents (addition, multiplication en mode SWP si besoin) et des recadrages de données en entrée et en sortie. La configuration d'une unité fonctionnelle est réalisée par l'intermédiaire d'un registre de configuration qu'il est possible de venir écrire directement en parallèle, ce qui permet de reconfigurer cette ressource en un cycle d'horloge.*

RECONFIGURATION PAR ADRESSAGE : il existe des unités de traitement, comme le CLB des Virtex Xilinx, dont la mémoire de configuration est une SRAM. Le DUCK doit cette fois pouvoir adresser cette RAM et aller y écrire les configurations stockées dans ses registres. La figure 2-11 illuste cette reconfiguration en mode par adressage. Les deux premières étapes (figure 2-11(a) et (b)) sont identiques à la reconfiguration séquentielle et parallèle. La dernière étape est exécutée grâce à l'introduction d'un générateur d'adresse capable de gérer la mémoire RAM contenue dans la ressource à reconfigurer (figure 2-11(c)). Dans la figure 2-12, la fonction LUT d'un bloc logique de FPGA est remplie par une mémoire RAM (représentée aire Ⓑ). L'aire Ⓐ correspond toujours au DUCK configuré cette fois-ci pour une configuration dynamique par adressage. L'aire Ⓒ présente les ressources en sortie du bloc logique permettant une sélection entre sortie synchrone et sortie asynchrone.

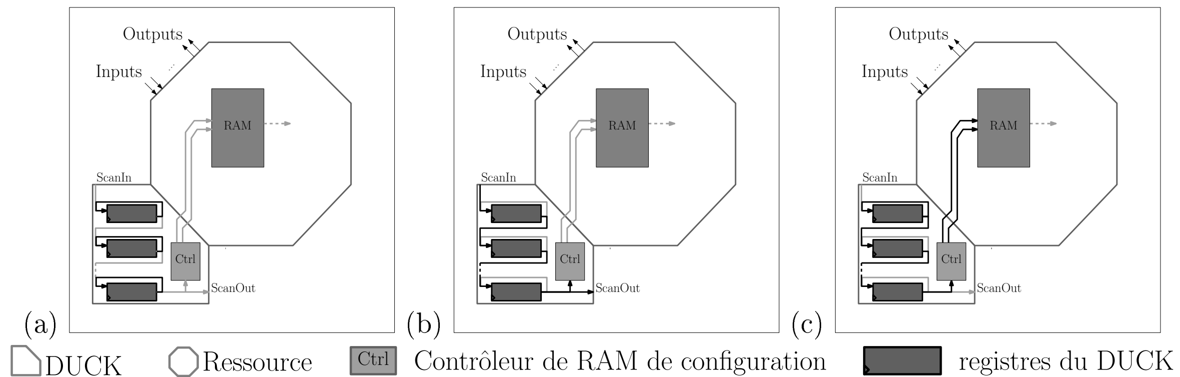


Figure 2-11 – **Phases de reconfiguration d'une ressource par l'intermédiaire d'un DUCK en mode adressage**. *Les phases de propagation de contextes se font selon le même principe que pour les DUCK à reconfiguration parallèle et séquentielle. La reconfiguration est gérée ensuite par un contrôleur interne au DUCK chargé de gérer les adresses de configuration.*

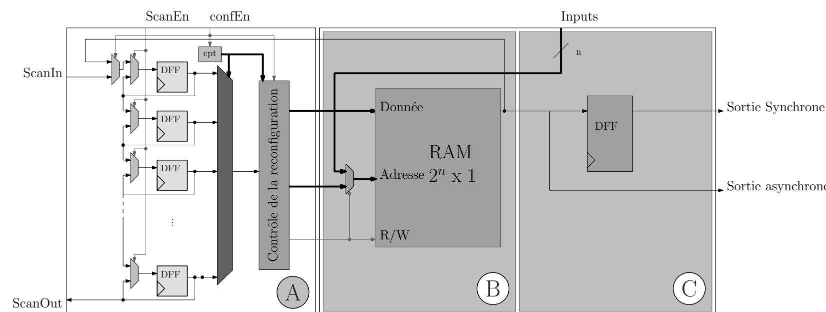


Figure 2-12 – **Exemple d'implémentation d'un DUCK dédié à une ressource reconfigurable par adresse**. *Un DUCK, représenté zone A, est connecté à une unité de traitement dont la fonctionalité peut être assimilée à une LUT. Cette LUT, zone B, est réalisée par une mémoire RAM asynchrone. La zone C représente la logique nécessaire à la génération d'un signal de sortie synchrone ou asynchrone.*

REGISTRES DUCK "FANTÔMES" : le chemin de configuration que constituent les registres des DUCK est homogène en termes de taille et d'interconnectivité. Cela permet de gérer simplement les processus de reconfiguration. En revanche, chaque DUCK peut avoir en charge une ressource dont les registres/mémoires de reconfiguration ont des tailles de données différentes du DUCK et ce quel que soit le mode de reconfiguration choisi. L'une des particularités du DUCK est sa capacité à pouvoir gérer cette hétérogénéité. Lors de l'instanciation d'un DUCK, son architecture est adaptée pour une ressource particulière et des connexions sont créées de sorte que la reconfiguration puisse se dérouler comme prévu. Prenons un exemple simple (figure 2-13), avec l'instanciation d'un DUCK adapté à une reconfiguration parallèle. Le chemin de configuration prévoit l'utilisation de plusieurs DUCK composés de registres de six bits alors que le registre de configuration de la ressource est lui de seize bits. Les trois bus de six bits, des registres DUCK sont assemblés de manière à former un seul bus de seize bits. Nous pouvons envisager toute combinaison possible, cela implique bien souvent la possibilité de connecter les bus de sortie du DUCK bit par bit. La taille des registres de configuration

DUCK étant homogène sur l'ensemble de l'architecture, il est parfois nécessaire de procéder à l'implémentation de registres DUCK "fantômes" de manière à ce que le nombre de bits de configuration des DUCK soit supérieur ou égal au nombre de bits de configuration de la ressource. Dans l'exemple précédent de la figure 2-13, cela implique l'implémentation de deux bascules "fantômes".

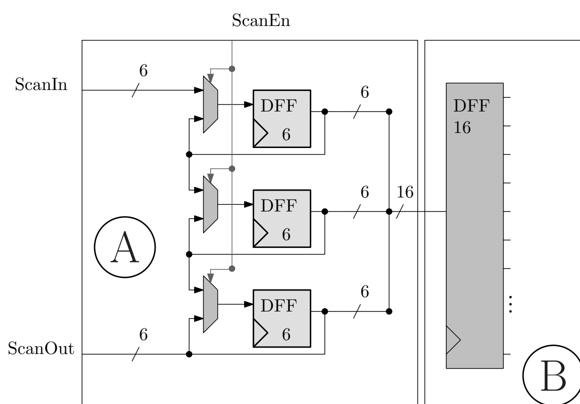


Figure 2-13 – **Adaptation du DUCK à la reconfiguration de la ressource**. *L'adaptation de la ressource DUCK à la configuration d'une unité à reconfigurer nécessite parfois l'utilisation de registres "fantômes". Le nombre de registres "fantômes" est déterminé en fonction de la taille du bus de configuration et du nombre de bits de configuration de l'unité considérée. Dans cet exemple, 18 registres composent le DUCK, dont seuls 16 sont utiles et dont les 2 restant constituent les registres "fantômes".*

C DyRIBox : unités d'interconnexion paramétrables et reconfigurables dynamiquement

Dans toute architecture, les unités de traitement ont besoin d'un réseau d'interconnexion permettant la communication entre ressources. Les schémas de communication entre ressources d'une architecture reconfigurable sont amenés à être modifiés au cours des différentes phases de configuration. Le réseau d'interconnexions doit donc être lui-même également reconfigurable dynamiquement. De plus, l'éventuelle hétérogénéité des ressources qui viendront composer l'architecture, implique l'obligation de prévoir un réseau d'interconnexions flexible tant en terme de connectivité qu'en terme de temps de reconfiguration. Cette section présente le concept de routeur d'interconnexion reconfigurable dynamiquement que nous avons baptisé DyRIBox (*Dynamically Reconfigurable Interconnection Box*). Avant cela, nous allons brièvement présenter les deux méthodes d'interconnexion utilisées dans le domaine des systèmes sur puce.

C.1 Routeurs d'interconnexion dans les architectures reconfigurables

Les routeurs d'interconnexions sont les garants d'une communication homogène au sein des architectures reconfigurables. En effet, eux seuls peuvent garantir la continuité de la com-

munication avant, après et pendant les phases de reconfiguration. Afin de concrétiser leur mise en application, deux types de solutions existent : le routage par paquet de données et le routage par commutation de circuit.

ROUTEURS À COMMUTATION PAR PAQUET : dans le cas des réseaux à commutation par paquet de données, l'information à transmettre est encapsulée dans une série d'informations de routage qui serviront à orienter en temps réels les paquets de données à la réception de ceux-ci par le routeur. Un canal de communication virtuel est alors créé entre l'émetteur et le récepteur. On peut facilement considérer que cette technique est particulièrement adaptée aux architectures reconfigurables, si l'on considère la possibilité de router et placer les tâches en temps réel en tout point de l'architecture. C'est la raison pour laquelle, cette technique de commutation est privilégiée dans le cadre du développement des NOC (*Network On Chip*) [36]. Cela est notamment le cas de [37] ou [38], où l'idée principale est basée sur l'utilisation massive de routeurs pouvant être désactivés en cours d'exécution et dont les ressources matérielles libérées peuvent être converties en ressources de calculs reconfigurables dynamiquement. L'association d'une utilisation massive de routeurs de communications et d'un algorithme de routage dédié garantit ainsi le routage d'un paquet à travers toute l'architecture. Le principal inconvénient est bien entendu la surface de silicium supplémentaire induite par cette technique et la latence encore plus accrue par cette méthode. Nous pouvons encore citer les travaux publiés dans [39] où l'architecture reconfigurable est basée sur une technique de routage par paquets. Leur approche, plus classique, consiste à concevoir les routeurs d'interconnexions comme une ressource reconfigurable dynamiquement de la même manière que cela est fait pour les ressources de calculs. Dans la plupart des cas, l'architecture des NOC utilisés est basée sur des routeurs utiles seulement pour le routage du flot de données de calcul. Leur présence est donc nécessaire pour un type de calcul donné et n'est donc plus nécessaire une fois le calcul effectué. Les auteurs estiment donc que ces routeurs doivent être aussi facilement insérés ou retirés comme pour les ressources de calculs. Cette fonctionnalité de reconfiguration est assurée par des tables de routage dynamiques. Ces tables sont configurées par le même contrôleur en charge de la reconfiguration des processus de reconfiguration. Une évolution intéressante est présentée dans [40], où les auteurs ont développé un système dont les tables de routage sont modifiées en temps réel, de manière à décongestionner les routeurs trop souvent sollicités. Ces différents exemples montrent la flexibilité apportée par un réseau d'interconnexions de type commutation par paquet de données. Cette flexibilité se situe au niveau du placement des tâches en temps réel. Cependant, le contrôle induit par cette technique pendant l'exécution, introduit de la latence et une hausse de la consommation d'énergie et de la surface de silicium et complexifie les algorithmes de routage adaptatifs.

ROUTEURS PAR COMMUTATION DIRECTE : les routeurs par commutation directe de circuit consistent à établir une connexion directe entre l'émetteur et le récepteur par l'intermédiaire d'un canal dédié. Cette méthode permet une faible latence dans la communication car la connexion est directe entre les ressources communicantes. Cependant, la multiplication des ressources de calculs et des communications qui en découlent au sein d'un seul et même SOC (*System On Chip*) [41] tend plutôt à l'utilisation de structure à base de routeurs par paquet. Dans le contexte des architectures reconfiguables, le problème est légèrement différent.

Tout d'abord, l'une des caractéristiques de la reconfiguration dynamique est de minimiser le nombre de ressources de calculs en permettant une optimisation temporelle de leur utilisation. Cela veut donc dire que, pour l'exécution d'une tâche, peu de communications devront être partagées par l'ensemble de l'architecture. De plus, de part leur faible besoin en énergie, les routeurs à commutation directe sont plus attractifs pour le domaine des systèmes sur puces. Dans ce domaine, les auteurs de [42] ont démontré que la technique de commutation par circuit direct permet une réduction de la consommation d'énergie de l'ordre d'un facteur trois et demi comparé à une solution équivalente en commutation par paquet. Une autre étude intéressante publiée dans [43] montre que ce mode d'interconnexion permet une prévision plus facile de la routabilité et de la surface utilisée pour l'implémentation des éléments d'interconnexion.

La faible consommation d'énergie, et la prédictibilité de la routabilité et des latences de transmissions font partis, de notre point de vue, des caractéristiques essentielles à prendre en compte lors de la conception d'une architecture reconfigurable dynamiquement. La DyRIBox que nous présentons est basée sur le principe de la commutation directe et dont les capacités ont été améliorées par l'introduction de ressources spécifiques à la reconfiguration dynamique. Cependant, pour des raisons de flexibilité, ses compétences ont également été étendues afin de permettre la mise en œuvre d'un réseau basé sur le routage par paquets.

C.2 Architecture d'une DyRIBox

La DyRIBox est une unité d'interconnexion (Ui) décrite par la figure 2-14. Elle oriente les données depuis ses ports d'entrée vers ses ports de sorties selon un schéma défini et stocké au préalable dans une configuration. Une DyRIBox est composée de I_{db} ports d'entrée et O_{db} ports de sortie dédiés aux communications entre DyRIBox et de I_{pe} ports d'entrée et de O_{pe} ports de sortie dédiés aux communications avec une ou plusieurs unités de traitement. Les ports de communication entre DyRIBox se répartissent sur chacun des quatre côtés (Nord, Sud, Est, Ouest). Le nombre total de ports d'entrée I et de ports de sortie O d'une DyRIBox est alors respectivement de $I = I_{db} + I_{pe}$ et de $O = O_{db} + O_{pe}$ avec :

$$I_{db} = \sum_{i=0}^{3} I_i \tag{2-1}$$

$$O_{db} = \sum_{i=0}^{3} O_i \tag{2-2}$$

où I_{db} représente le nombre de ports d'entrée connectés à d'autres DYRIBox, O_{db} représente le nombre de ports de sortie connectés à d'autres DYRIBox, I_i représente le nombre d'entrées du côté i, O_i représente le nombre de sortie du côté i. En fonction de la valeur du registre de configuration, chaque entrée peut être connectées à une ou plusieurs sorties. Afin de réduire la complexité et la taille du flot de données de configuration de la DyRIBox, le nombre d'entrées pouvant être connectées à une sortie est fixé à E, avec $E \leq I$. Par conséquent, une DyRIBox contient O registres de configuration de $e = \lceil \log_2 E \rceil$ bits. Par exemple, une DyRIBox qui serait composée de vingt ports de sortie et dix-sept ports d'entrée aurait vingt registres de configuration de cinq bits si l'on considère que toutes les entrées peuvent être connectées à toutes les sorties.

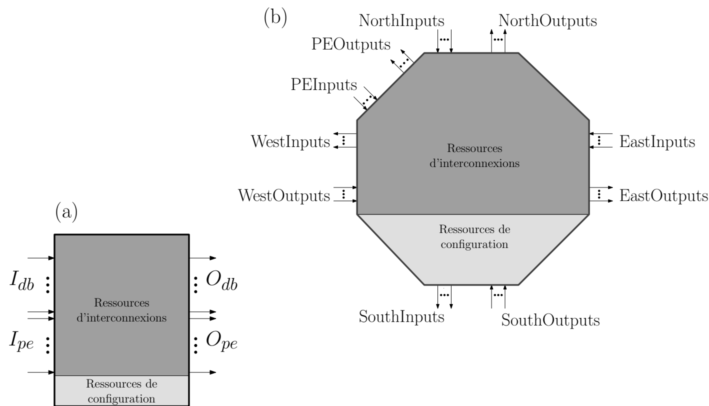


Figure 2-14 – **Synoptique d'une DyRIBox simplifiée**. *(a) Une DyRIBox permet de connecter I_{db} et I_{pe} entrées sur O_{db} et O_{pe} sorties. (b) Les I_{db} entrées et O_{db} sorties se répartissent uniformément sur les quatre côtés d'une DyRIBox.*

C.3 Principe de reconfiguration et principe de contrôle des multiplexeurs

Après avoir présenté le principe général de fonctionnement d'une DyRIBox, examinons à présent le principe de reconfiguration interne de la DyRIBox (figure 2-15). Chaque multiplexeur de configuration des sorties est contrôlé par un registre dont la valeur permet de sélectionner l'entrée désirée (figure 2-15(a)). Dans le même esprit que la méthode de reconfiguration utilisée par le DUCK, les multiplexeurs des DyRIBox sont commandés par des registres avec une entrée de validation (figure 2-15(b)) interconnectés avec le DUCK en configuration parallèle (cf section B.3). Cela permet, avec une faible quantité de ressources supplémentaires, de procéder à des reconfigurations dynamiques en un cycle. Outre le fait de pouvoir procéder à une reconfiguration rapide, cela permet en plus de pouvoir procéder à des modifications de chemin de données pour certaines sorties sans perturber la configuration d'autres sorties non concernées par la reconfiguration. En effet, le remplacement d'un mot de configuration par un autre mot dont la valeur est identique à la précédente ne fera pas commuter le multiplexeur et ne perturbera donc pas les chemins de données concernés.

C.4 De la commutation directe de circuit vers la commutation par paquet

Le système de communication basé sur des DyRIBox est à commutation directe de circuit. Les informations d'orientation des données sont stockées dans la mémoire de configuration et acheminées localement lorsque cela est nécessaire. Nous pouvons faire évoluer cette méthode vers une commutation par paquet. Cela est envisageable grâce à la possibilité de la DyRIBox de se reconfigurer en un cycle, permettant la réduction maximale de la latence. Il est alors possible de profiter des avantages du routage en temps réel au prix d'une augmentation de surface, de consommation et d'une difficulté à estimer de manière précise la latence dans les communications. Cette partie est encore en cours d'exploration, nous nous contenterons de présenter ici le concept général de ce type de routeur.

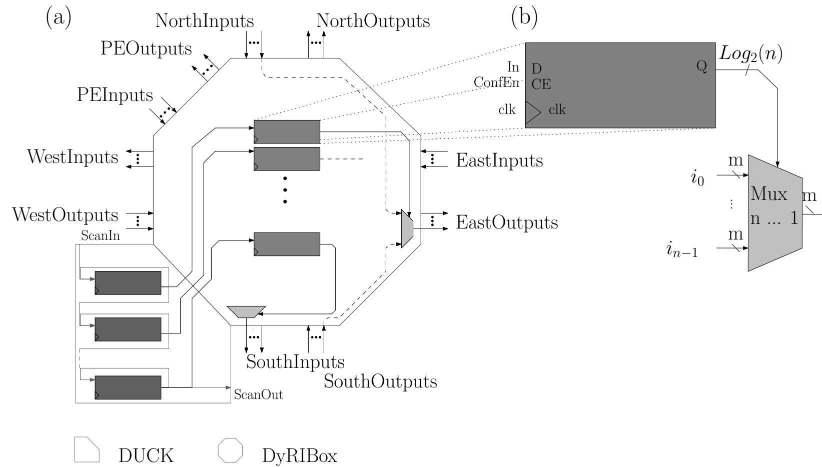


Figure 2-15 – **Schéma synoptique d'une DyRIBox simple**. *La représentation logique d'une DyRIBox simple (a) revient à l'implémentation d'un réseau de multiplexeurs commandé par un réseau de registres dont la valeur détermine l'entrée à connecter sur la sortie. Les registres de commande (b) ont été implémentés avec une entrée de validation afin de prendre une nouvelle configuration en compte.*

Méthode de décodage : la méthode de décodage que nous avons choisi d'implémenter est basée sur l'utilisation de mémoire CAM (*Content Access Memory*) utilisées dans les mécanismes de mémoires caches [44]. Lorsque le routeur détecte le début d'une trame, celui-ci décode l'entête afin de retrouver dans sa mémoire la direction vers laquelle la trame devra être propagée. L'adressage de la mémoire se fait par le contenu de façon équivalente avec les contrôleurs des mémoires cache (figure 2-16).

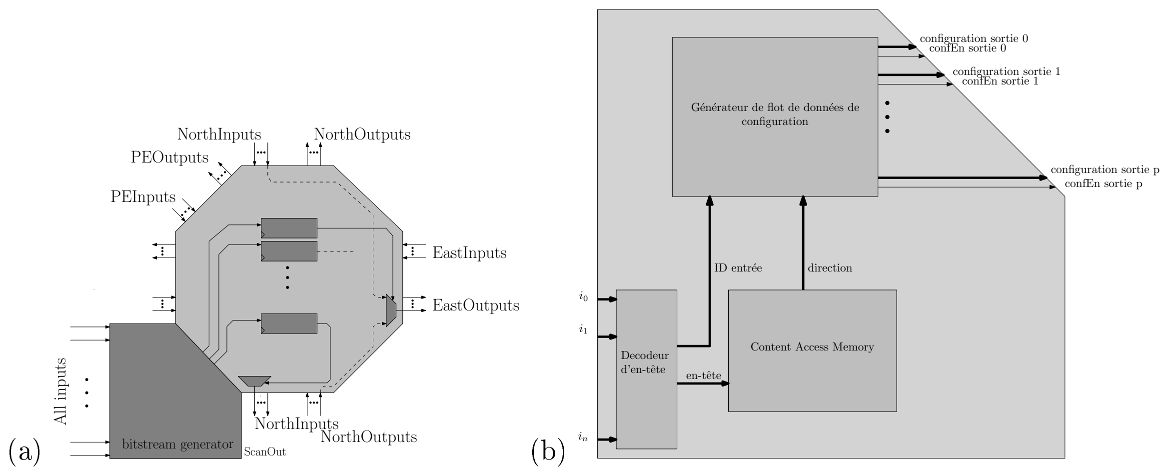


Figure 2-16 – **DyRIBox de type commutation par paquets**. *La gestion des registres de configuration n'est plus effectuée par le DUCK mais par une mémoire de type CAM. Cette CAM peut être reconfigurée par l'intermédiaire d'un DUCK de type adresse.*

À chaque début de trame, un décodeur recherche dans une table de correspondance la valeur du registre de commande du multiplexeur concerné. La mémoire CAM est adressée avec la

valeur de l'en-tête et présente alors sur son port de donnée l'identificateur de la sortie où devra être orientée la trame. Une phase de configuration est bien évidemment nécessaire afin d'établir correctement la table de routage à implémenter dans la mémoire CAM. Une fois l'identificateur de la sortie déterminé, la valeur du registre du multiplexeur de commande est remplacée par la valeur binaire du numéro de l'entrée d'où provient le décodage de la trame afin de garantir la bonne orientation de la communication pour toute la trame.

D DyRIOBloc : ressources d'entrée/sortie reconfigurables

Souvent utilisées comme accélérateurs matériels, les architectures reconfigurables ont besoin de pouvoir communiquer avec le monde extérieur. Dans le cadre d'une architecture reconfigurable, un contexte peut prévoir l'utilisation d'un port de communication dans un sens, et un autre contexte l'utilisation de ce port dans le sens inverse. Cette sous section présente le principe de l'unité d'entrée/sortie que nous avons baptisé DyRIOBloc (*Dynamically Reconfigurable Input Output Block*). Les ressources DyRIOBloc (figure 2-17 (A)) sont dédiées aux communications de l'architecture avec l'extérieur. La spécification d'un port consiste essentiellement à définir si celui-ci est un port d'entrée, un port de sortie ou un port bidirectionnel, le sens de la communication sera alors spécifié par reconfiguration dynamique. Un paramètre intéressant à prendre en compte pour les architectures reconfigurables, consiste à spécifier la taille de la mémoire tampon qui fera office d'interface entre l'architecture et le monde extérieur. Aucune particularité majeure n'est à noter si un port de l'architecture reconfigurable est spécifié de manière à être utilisé exclusivement en entrée ou exclusivement en sortie. En revanche, certaines précautions doivent être prises si le port est configuré en entrée, ou en sortie. Il est nécessaire, pour le bon fonctionnement de la mémoire tampon, de réguler les communications avec des portes à trois états (figure 2-17 (B.a)). Si le port est configuré en entrée (figure 2-17 (B.b)), alors seules les portes trois états actives à l'état haut sont passantes. À l'inverse, si le port est configuré en sortie (figure 2-17 (B.c)), alors seules les portes trois états actives à l'état bas sont passantes. Un seul registre est nécessaire à la configuration de l'ensemble des portes trois états d'un port. Seuls les ports reconfigurables dynamiquement sont censés être connectés au chemin de configuration. Un contrôleur de configuration est dédié à l'ensemble de ces ports. Le principe de la reconfiguration appliquée est le même que pour les unités d'interconnexion DyRIBox. Chaque registre de configuration est connecté en entrée à un registre du DUCK dédié à la configuration des entrées/sorties. Il est alors possible de procéder à la reconfiguration d'un port en un cycle d'horloge, laissant ainsi le temps maximum aux phases de travail.

E Ressources de contrôle et de gestion de la reconfiguration dynamique

La définition d'un modèle générique de reconfiguration dynamique nous a amené à proposer le choix le plus large possible en termes d'implémentation et de caractéristique de reconfi-

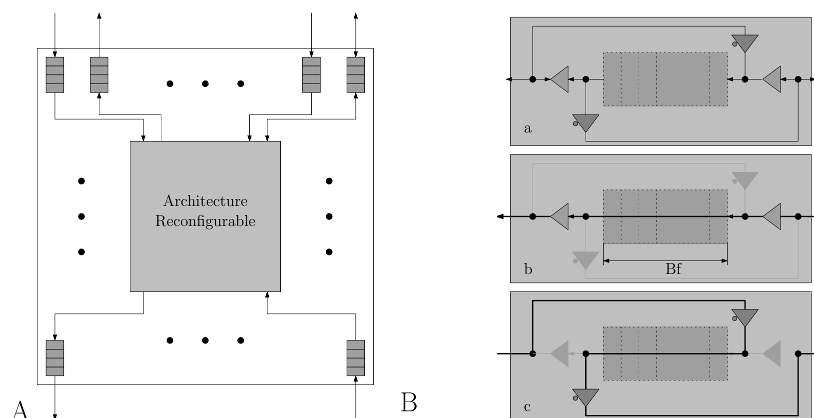


Figure 2-17 – **Schéma synoptique d'un DyRIOBloc**. *Un DyRIOBloc permet d'interfacer les données de communication entre l'architecture et le monde extérieur. Des mémoires servent de tampon dans les communications. Les mémoires tampon des ports bidirectionnel doivent être contrôlés à l'aide de portes trois états. Si le port est configuré en entrée (B.b), alors seules les portes trois états actives à l'état haut sont passantes. À l'inverse, si le port est configuré en sortie (B.c), alors seules les portes trois états actives à l'état bas sont passantes.*

guration. Une fois définies les caractéristiques principales à atteindre, il est ensuite facile de restreindre les possibilités offertes par notre modèle de manière à réaliser le meilleur compromis entre flexibilité, dynamicité et performances (F-D-P). Selon les performances de reconfiguration requises, un contrôleur de reconfiguration peut gérer un ou plusieurs domaines. Le contrôleur de reconfiguration (figure 2-18) est composé de trois fonctions principales que nous présentons dans cette section :

1. un ordonnanceur de contextes (*CtxtScheduler*),
2. un contrôleur mémoire de contextes (*CtxtMemCtrl*),
3. un contrôleur de domaine (*DomainCtrl*).

E.1 *CtxtScheduler* : gestion et ordonnancement des contextes

Le *CtxtScheduler* est chargé de fournir la première adresse du contexte à implémenter au contrôleur de mémoire. Cette adresse est déterminée en fonction de l'ordonnancement des contextes effectué à la compilation et des éventuelles interruptions qui pourraient intervenir en cours de traitement. Le fonctionnement d'un *CtxtScheduler* sera détaillé dans les pages suivantes.

E.1-1 Gestion séquentielle des tâches

D'un point de vue pratique, les adresses des tâches sont stockées dans une mémoire (*CtxtAddr* figure 2-19) dans un ordre d'exécution préalablement configuré. À chaque requête de reconfiguration, l'adresse de la mémoire *CtxtAddr* est incrémentée de façon à extraire la première adresse de la prochaine configuration. Étant donné que l'interruption d'une exécution séquentielle est possible au cours de son traitement, il est nécessaire de gérer l'exécution des

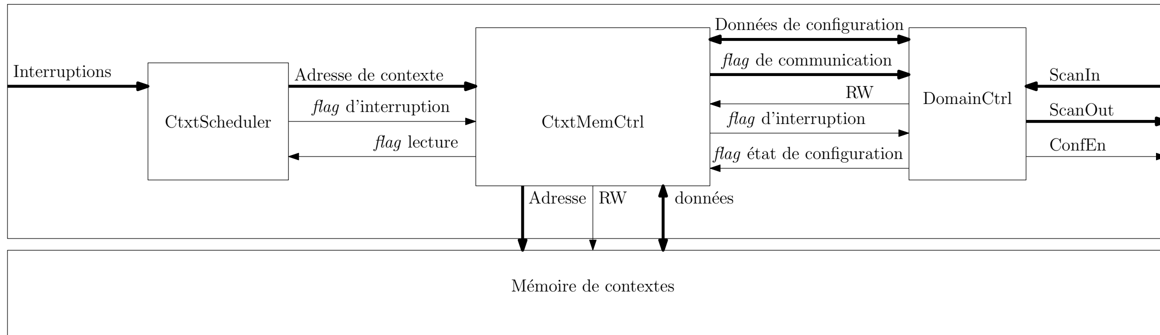


Figure 2-18 – **Schéma synoptique du contrôleur de reconfiguration**. *L'ordonnanceur de contextes détermine la première adresse mémoire du prochain contexte à propager. Le contrôleur de mémoire est chargé d'adresser la mémoire de configuration et de gérer les flots de données entrant et sortant. Enfin, le contrôleur de domaine, détermine les instants auxquels il est nécessaire de procéder à une nouvelle configuration en fonction des états d'avancement des tâches implémentées. Des signaux spécifiques de communication permettent d'informer les blocs voisins de l'état de chacun.*

tâches d'interruption (*IRCtrl*, FIFO et *IRIDConverter*) et d'indiquer au contrôleur mémoire la priorité de reconfiguration du contexte par l'intermédiaire d'un drapeau.

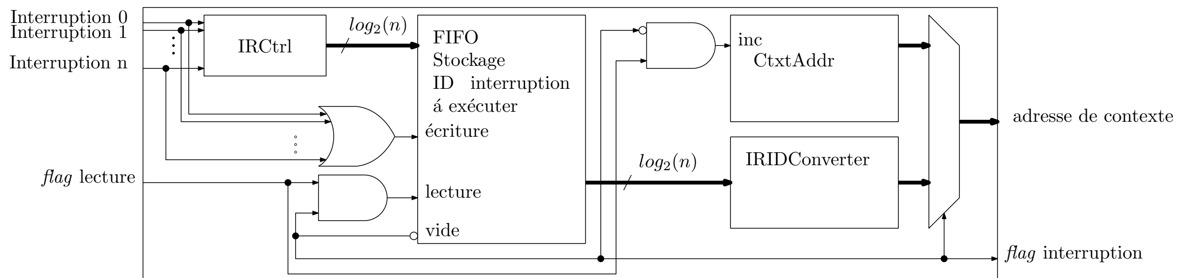


Figure 2-19 – **Schéma synoptique du *CtxtScheduler***. *Le* CtxtScheduler *est en charge de gérer les adresses de début de contexte et d'ordonnancer leur exécution en fonction des entrées d'interruption ou du pointeur de configuration interne (*CtxtAddr*) configuré en fonction du séquencement des tâches défini à la compilation.*

E.1-2 *IRCtrl* : gestion des interruptions sans priorité

Lorsque une requête d'interruption est détectée par l'activation de l'un des n signaux d'interruption externes (*interruption_i* avec $1 \leq i \leq n$ figure 2-19), l'identificateur du signal concerné est stocké dans une mémoire FIFO afin de gérer ces interruptions dans leur ordre d'apparition. L'identificateur d'interruption a la valeur i codée en binaire. La première adresse mémoire de la tâche d'interruption concernée est ensuite extraite d'une table de correspondance (*IRIDConverter*) qui sera lue prioritairement par rapport à la table *CtxtAddr*.

E.1-3 Gestion des interruptions avec priorité

La gestion de priorité de tâche à p niveaux est également supportée (figure 2-20). Pour cela, p blocs *IRCtrl*, *IRIDConverter* et mémoires FIFO sont nécessaires. Le principe de fonctionnement est sensiblement identique au fonctionnement du contrôleur présenté précédemment. Il est cependant nécessaire de gérer la priorité des interruptions. Cette tâche est accomplie par le bloc *ReadFlagCtrl* (figure 2-20) chargé de séquencer la lecture des FIFO selon le niveau de priorité des interruptions. Son principe de fonctionnement est simple puisqu'il sélectionne les FIFO de niveaux de priorité supérieures tant que celles-ci ne sont pas vides. Une fois la mémoire FIFO supérieure vidée, le contrôleur sélectionne la mémoire FIFO de niveau de priorité inférieur, jusqu'à ce que toutes les mémoires FIFO soient vides.

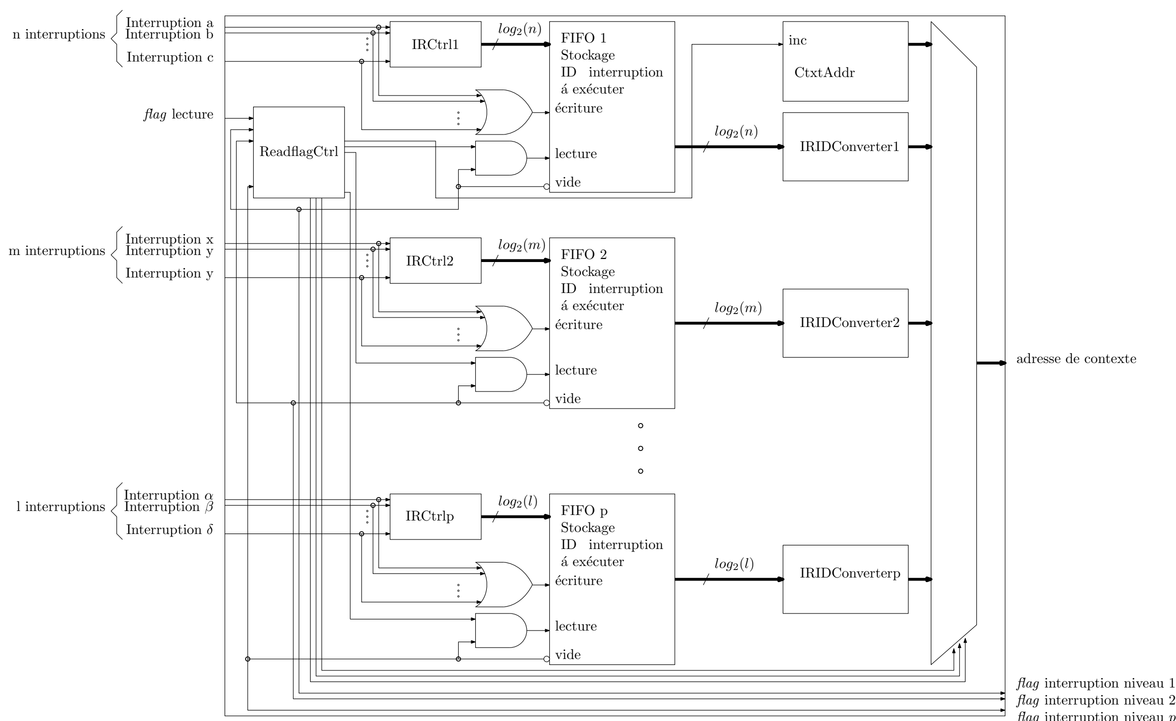


Figure 2-20 – **Schéma synoptique du *CtxtScheduler* à plusieurs niveaux de priorité d'interruptions**. *Le* CtxtScheduler *présenté ici est capable de fournir les adresses des contextes d'interruption par ordre d'apparition et de priorité.*

E.2 *DomainCtrl* : contrôleur de domaine

Le contrôleur de domaine (*DomainCtrl*) présenté dans la figure 2-18, est chargé de gérer la propagation des configurations à travers le domaine. Il commande les bits de contrôle des DUCK en fonction des données de configuration provenant du contrôleur de mémoire. Le *DomainCtrl* gère certains mécanismes de reconfiguration dynamique telles la gestion de la préemption et la gestion de la reconfiguration partielle.

E.2-1 GESTION DE LA PRÉEMPTION

La gestion de la préemption est effectuée par les *DomainCtrl*. Ceux-ci déterminent l'instant où la configuration du contexte doit être effectuée. Deux possibilités sont envisagées. Dans un premier cas, le temps d'exécution nécessaire est déjà défini à la compilation. Dans ce cas, le compilateur est capable de déterminer une itération de N_c cycles avant de configurer un nouveau contexte. Pour cela, un compteur est configuré de manière à signaler la fin de la tâche après N_c cycles d'horloge. Dans le deuxième cas, la fin de l'exécution n'est pas définie à la compilation, le temps d'implémentation n'est alors pas connu à la compilation et la condition d'exécution de la préemption nécessite l'implémentation d'un algorithme de contrôle flexible. Concrètement, une boucle *for* (listing 2-1 a) dont la condition de fin et le nombre d'itérations sont connus à la compilation peut être contrôlée par un compteur dont la valeur d'initialisation est contenue dans le flot de données de configuration. En revanche, une boucle *while* (listing 2-1 b), dont la condition de sortie n'est pas connue à la compilation nécessite un contrôle complexe. Comme défini dans le modèle de reconfiguration, cette exécution est conditionnée sur la valeur d'un bit de contrôle. L'implémentation de fonction logique sur une cible de type bloc logique configurable à base de LUT se prête parfaitement à cet objectif. Le compilateur se doit ensuite de générer le flot de données de configuration en fonction de la condition logicielle codée dans le code source. Le concept général de contrôle des zones de reconfiguration se fait par une machine d'état que nous ne détaillons pas ici. Un graphe *flowchart* simplifié de cette gestion est présenté figure 2-21.

```
1  //a: exemple de boucle finie      // b: exemple de boucle indéterminée
2  for(i=0;i<10;i++){                do{
3     X++;                              X++;
4  }                                 } while (X<Y);
5  ...
```

Listing 2-1 – Exemple de boucles en C : finie et indéterminée

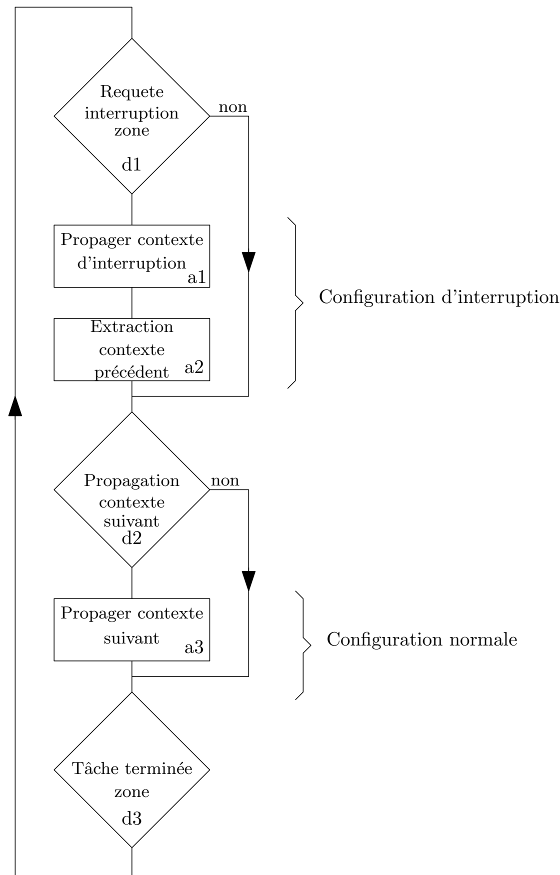


Figure 2-21 – **Graphe *flowchart* du contrôleur de domaine**. *Le contrôleur de domaine est basé sur une machine d'état. Ce contrôleur est chargé de contrôler les ressources de configuration en propagation ou en extraction selon les instructions du contrôleur de pointeur de configuration par l'intermédiaire du contrôleur mémoire en cas d'interruption, ou selon son propre compteur interne en cas de configuration normale. Après avoir vérifié qu'aucune tâche d'interruption n'est requise (d1), il lui faut s'assurer que la tâche en cours de traitement n'a pas terminé son exécution (d3) après avoir vérifié que la future est contenue dans les DUCK (d2), et procédé à la propagation de celui-ci si besoin est (a3). Au cas où une requête d'interruption intervient (d1), le contrôleur prépare la nouvelle configuration, procède à la reconfiguration des ressources dès que la configuration est complètement chargée localement dans les DUCK (a1) puis extrait le contexte interrompu (a2). La tâche devant être interrompue à continuer son exécution pendant la phase de chargement de la tâche d'interruption.*

E.2-2 GESTION DE LA RECONFIGURATION PARTIELLE

La gestion de la reconfiguration partielle est un des critères de contribution à une reconfiguration dynamique efficace. Toutes les ressources d'un domaine ne sont pas nécessairement requises pour l'implémentation d'une tâche. Par exemple, une fois l'exécution d'une tâche terminée, une zone est définie dynamiquement et peut être reconfigurée sans avoir à perturber les autres tâches en cours de traitement. Nous avons mis en œuvre la reconfiguration partielle de manière à ce que chaque entité reconfigurable de l'architecture se voit attribuer une position virtuelle dans une grille recouvrant l'ensemble du domaine gérée par le contrôleur. La position d'une ressource sur cette grille s'exprime en ligne et en colonne de sorte que les signaux de configuration générés par le contrôleur forment un masque de configuration qui viendra valider ou non le processus de reconfiguration d'une ressource (figure 2-22).

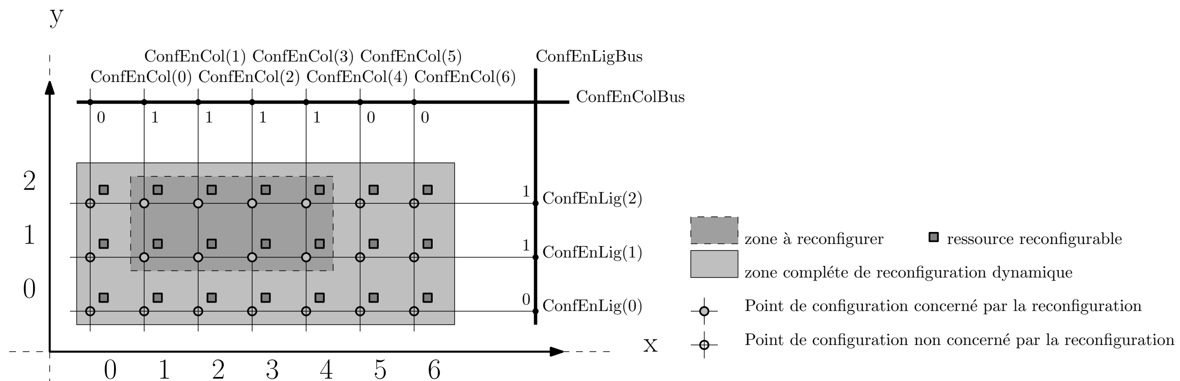


Figure 2-22 – **Principe de la reconfiguration partielle**. *Deux bus sont utilisés afin de rendre la reconfiguration partielle possible. Un bus est chargé de valider les lignes à reconfigurer tandis que l'autre est chargé de valider les colonnes. Seuls les éléments se trouvant à l'intersection d'une ligne et d'une colonne à reconfigurer auront une validation de la reconfiguration.*

E.3 *CtxtMemCtrl* : CONTRÔLEUR DE MÉMOIRE DE CONFIGURATION

E.3-1 CONTRÔLE INDIVIDUEL DES DOMAINES

Un domaine est défini comme étant une zone contrôlée individuellement et composée de DUCK chargés de gérer les processus de reconfiguration de façon homogène. Le contrôleur de mémoire de configuration se doit de gérer les flux provenant de la mémoire et de les orienter vers les ressources de configuration. Il agit sur requête des contrôleurs de domaines, et adresse l'espace mémoire selon la valeur du pointeur de contexte correspondant. La gestion de ces adresses et des phases de reconfiguration/préemption est réalisée en fonction des indicateurs d'interruption fournis par le bloc *CtxtScheduler*. Si une tâche intervient avec un niveau d'interruption supérieur à la tâche en cours d'exécution, alors le contrôleur procède à la propagation du contexte correspondant, fait stopper le traitement en cours, reconfigure puis procède à l'extraction du contexte précédent. Techniquement, ce contrôle est géré par

une machine d'état dont le fonctionnement à l'aide d'un graphe *flowchart* est donné sur la figure 2-23 (A).

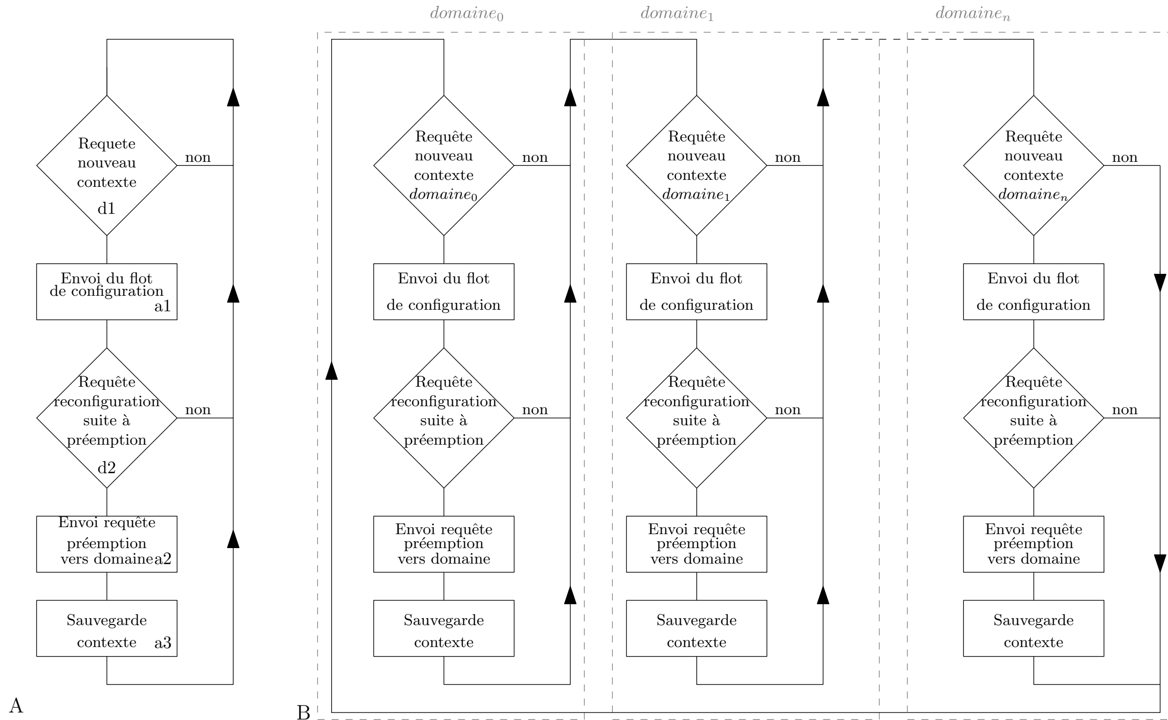


Figure 2-23 – **Graphe *flowchart* du contrôleur de mémoire de configurations**. *Le graphe A concerne un* CtxtMemCtrl *en charge d'un seul domaine, alors que le graphe B concerne un* CtxtMemCtrl *en charge de plusieurs domaines. Tout d'abord, le système attend qu'une requête de propagation de nouveau contexte arrive (d1). Cette requête peut venir soit du contrôleur de domaine concerné, soit du gestionnaire d'ordonnancement des tâches en cas d'interruption. Si tel est le cas (d2), alors le contrôleur de zone de configuration se doit de l'indiquer au domaine (a2) après avoir envoyé la configuration (a1) afin que celui-ci puisse procéder à la sauvegarde du contexte (a3).*

E.3-2 Contrôle partagé des domaines

Dans le cas où les contraintes temporelles de l'application à implémenter le permettent, la gestion séparée des domaines de reconfiguration n'est pas nécessaire. Nous avons également développé un contrôleur générique de mémoire multi-zones (figure 2-23 B). Dans ce cas, une seule mémoire de configuration est nécessaire à l'ensemble de l'architecture (figure 2-24). Le contrôleur partagé est alors chargé de gérer l'accès à la mémoire pour un ensemble de domaines. Cependant, il est secondé par un *CtxtScheduler* et par *DomainCtrl* spécifiques à chaque domaine. Le principe de base du fonctionnement de ce contrôleur est identique à un contrôleur dédié à un domaine unique.

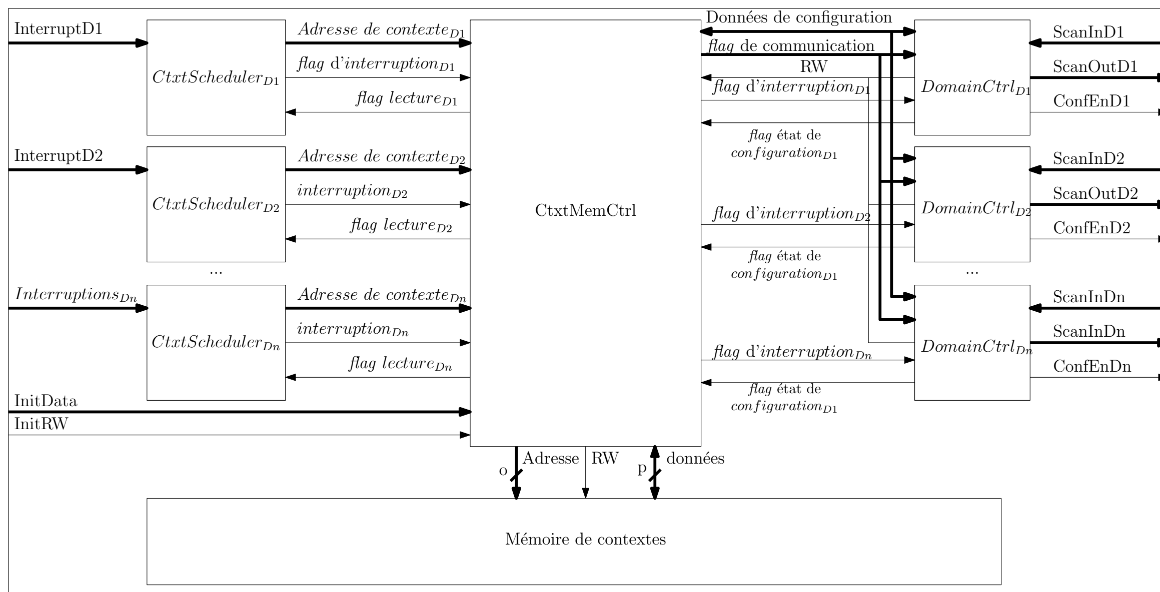


Figure 2-24 – **Schéma synoptique du contrôleur de mémoire de configuration multi-zones**. *Un contrôleur mémoire devant gérer les reconfigurations dynamiques de plusieurs zones est secondé par un contrôleur de pointeur de configuration et d'un contrôleur de domaine par domaine. Celui-ci séquence les accès à la mémoire en fonction des requêtes des différentes fonctions.*

E.3-3 Espace mémoire de configuration et format de la trame de configuration

L'organisation de l'espace mémoire est présentée à la figure 2-25. Sa taille est fonction du nombre de tâches à sauvegarder et de la taille en bit (q) des mots de configuration. Dans l'organisation de celle-ci, les mots de configuration des ressources de calculs se doivent d'être placées avant les mots de configuration des interconnexions. En cas de préemption, cela permet de n'avoir qu'une partie du flot de configuration à extraire. Le format des trames de configuration sauvegardées en mémoire est détaillé dans la section suivante. L'espace A' représente la taille occupée en mémoire pour stocker une configuration d'unité de traitement (C_t). L'espace A représente la taille occupée en mémoire pour stocker une configuration d'une unités d'interconnexion (C_i). L'espace B représente la taille occupée en mémoire pour stocker la configuration d'un contexte complet. L'espace C représente la taille occupée en mémoire pour stocker toutes les configurations d'un domaine. Chaque mot de configuration est composé de q bits. Enfin, les données nécessaires aux différents contrôleurs sont représentées par C_{ctrl}.

Le contrôleur de configuration gère la configuration des unités d'interconnexion indépendamment des unités de traitement. Cela se justifie lors des phases de préemption où la configuration des connexions n'a pas évolué au cours de l'exécution d'un calcul et ne nécessite pas par conséquent d'être extraite. L'implémentation même des ressources nécessaires à la reconfiguration permet cette séparation. Une trame de configuration complète est présentée sur la figure 2-26. Dans cette trame, les données concernant la configuration des unités de

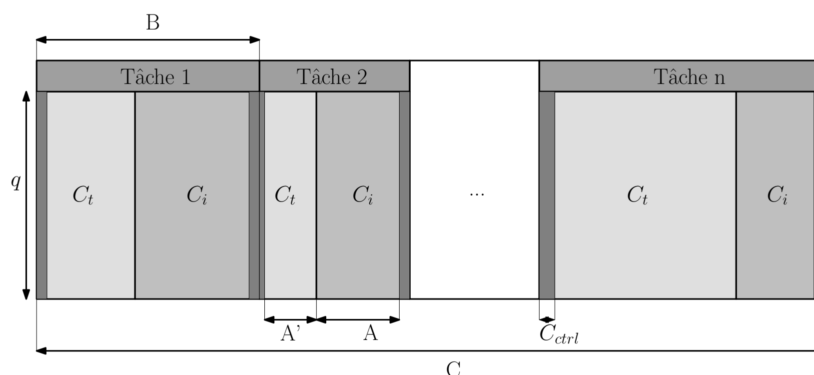


Figure 2-25 – **Organisation de la mémoire de configuration**. *La taille de cette mémoire est fonction du nombre de zones de reconfiguration ainsi que du nombre de configurations par zone.*

traitement sont physiquement séparées des données concernant les unités d'interconnexion. Une trame de flot de données de configuration est composée de différentes informations. Tout d'abord, le premier paquet de données (Tbc) indique le nombre de bits contenus dans le flot de données de configuration complet. Le paquet de données (Tbt) quand à lui contient le nombre de bits contenus dans le paquet de données qui concerne la configuration des unités de traitement (C_t). (C_i) représente le flot de données de configuration des unités d'interconnexion. Le paquet (Wct) contient le temps nécessaire à l'exécution complète de la tâche, si celle-ci est connue. Dans le cas contraire, il est composé uniquement de "un". Dans le cas d'une phase de configuration préemptive, le paquet (Wcr) permet de connaître la durée de traitement déjà réalisée dans une configuration précédente. Enfin, le paquet (Mr), dont la taille varie en fonction du nombre de lignes et de colonnes de la zone à reconfigurer, contient le masque à appliquer sur les bus de masquage afin de sélectionner la zone dynamique de reconfiguration.

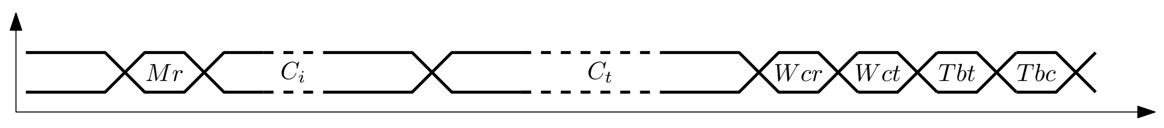


Figure 2-26 – **Trame de configuration**. *Une trame de configuration est composée des données de configuration des unités de traitement (C_t) et d'interconnexions (C_i). Les données de configuration nécessaires au contrôleur DomainCtrl sont Tbc, Tbt, Wcd, Wcr et Mr. Tbt est la valeur binaire du nombre de cycles d'horloge total nécessaire à l'acheminement des données de configuration des unités de traitement, et Tbc de la trame complète. Wct est la valeur binaire de la durée (en cycles d'horloge) de traitement de la tâche en cours de configuration alors que Wcr est la durée de traitement déjà effectué. Cette information est utile lors de la reconfiguration d'une tâche préalablement préemptée. Mr est la valeur binaire de masque nécessaire à une reconfiguration partielle.*

F Synthèse

Dans ce chapitre, nous avons présenté le modèle de reconfiguration dynamique que nous considérons comme étant suffisamment flexible pour supporter les applications actuelles et futures. Cette flexibilité a été atteinte grâce au développement de ressources reconfigurables dynamiquement tels que les DUCK, les DyRIBox et les contrôleurs de configuration associés. Ainsi, nous avons pu répondre aux exigences des contraintes que nous nous étions fixés tels que le temps de reconfiguration, la gestion des interruptions et des mécanismes de préemption. La flexibilité permise par ces ressources est nécessaire afin de pouvoir répondre de manière générique à tout types d'application ce qui n'est pas forcement le cas une fois l'architecture conçue. Par conséquent, pour des raisons de coût en surface de silicium et de consommation, il est nécessaire de pouvoir procéder à une spécialisation de l'architecture en cours de conception tout en s'assurant que les choix effectués répondent au maximum aux exigences de départ. C'est à partir de ce constat que nous avons procédé à la spécification d'un langage de description d'architecture qui nous permet de mettre en œuvre les concepts présentés dans ce chapitre en adéquation avec les contraintes factuelles de l'application destinée à être implémentée. Le langage de description permet d'une part, en faisant abstraction des détails architecturaux, de procéder rapidement à des phases de simulation et d'exploration. D'autre part, cela permet un support aux outils de compilation génériques adaptés à chaque architecture décrite.

Chapitre

3 xMAML : langage de description d'architectures reconfigurables dynamiquement

Résumé : Dans le chapitre précédent, Nous avons présenté un modèle générique de reconfiguration dynamique dont l'objectif est d'être utilisé pour la conception d'architectures. Cependant, ce modèle étant générique, il est nécessaire de pouvoir procéder rapidement à une spécification des processus de reconfiguration afin de répondre aux exigences de l'application destinée à être implémentée. Une méthode de spécification est la description d'architecture de haut niveau. Cette méthode permet d'une part, en faisant abstraction des détails architecturaux, de procéder rapidement à des phases de simulation et d'exploration. D'autre part, cela permet un support aux outils de compilation génériques adaptés à chaque architecture décrite. Le langage de description que nous présentons dans ce chapitre, xMAML, est basé sur le langage de description MAML. xMAML est le fruit direct de la collaboration que nous avons pu avoir avec l'équipe de recherche Hardware-Software-Co-Design de Universität Erlangen-Nürnberg à l'origine de MAML.

Sommaire

A Introduction . . . 70
A.1 Langages de description structurels . . . 71
A.2 Langages de description comportementaux . . . 71
A.3 Langages de description mixtes . . . 72
B MAML : Présentation . . . 73
B.1 Description au niveau haut : *Processor Array level* . . . 74
B.2 Description d'un élément processeur : *PE level* . . . 74
B.3 Synthèse . . . 76
C xMAML . . . 76
C.1 Présentation générale d'une description xMAML . . . 76
C.2 Modèle xMAML du bloc d'interconnexion DyRIBox . . . 77
C.3 Notion de domaine de reconfiguration . . . 80
C.4 Unités de traitement . . . 83
C.5 DyRIOBlocs : blocs d'entrée-sortie . . . 86
C.6 Synthèse . . . 86
D Exploration des paramètres de reconfiguration dynamique . . . 87
D.1 Exploration en surface et consommation des ressources de reconfiguration dynamique . . . 88
D.2 Données de configuration des interconnexions . . . 91
D.3 Données de configuration des unités de traitement . . . 92
E Synthèse . . . 93

A INTRODUCTION

Nous avons présenté, dans le chapitre II, un modèle générique de reconfiguration dynamique. Ce modèle est voué à être utilisé pour différentes cibles architecturales. Un certain nombre de plate-formes de développement ont été conçues ces dernières années. Celles-ci permettent le développement d'architectures reconfigurables dynamiquement basées sur un seul concept de traitement des données. Nous proposons, en complément de ces plate-formes de développement, un outil dédié à la mise en œuvre de la reconfiguration dynamique, quel que soit les concepts de traitement utilisés sur l'architecture cible. À ce titre, MOZAÏC constitue un élément des plate-formes sur lesquelles nous nous appuyons. La figure 3-1 présente un exemple d'intégration de MOZAÏC dans un flot de conception d'architecture. La description xMAML de l'architecture permet de procéder à une exploration des performances de la reconfiguration dynamique par l'analyse des résultats de consommation ou de la surface de silicium utilisée dès le début de la conception. Après validation des résultats de l'exploration, une deuxième analyse de la description xMAML conduit à la production automatique du code synthétisable des ressources de reconfiguration dynamique adaptées. Indépendamment de cela, selon l'architecture visée, l'application devant être implémentée peut être synthétisée (ou compilée) selon les méthodologies définies par les plate-formes en question. Une phase de traduction sur les configurations générées, propres à chaque plate-forme, doit alors être effectuée de manière à produire les trames de configuration compatible avec le format définit par MOZAÏC.

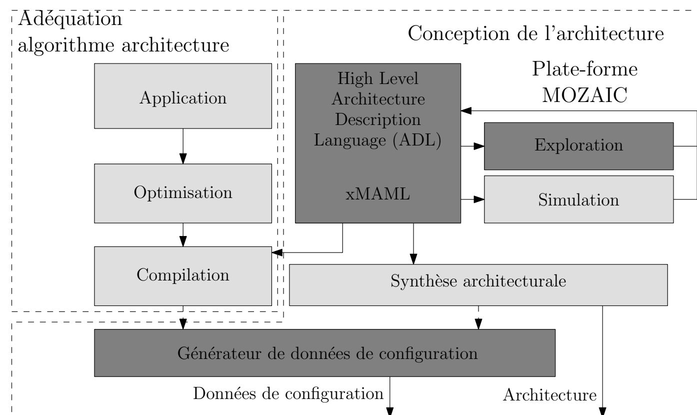


Figure 3-1 – **Plate-forme de développement MOZAÏC et son environnement**. *La conception d'un système reconfigurable dynamiquement à l'aide de* Mozaïc *commence par la description haut niveau de l'architecture. À partir de cette description, il est envisageable de procéder à une exploration des performances par l'analyse des résultats de simulation, des consommations ou de la surface de silicium utilisée. L'exploration terminée, la description haut niveau est ensuite analysée et conduit à la production automatique du code synthétisable de l'architecture finale.*

Les langages de description d'architectures ADL (*Architecture Description Language*) ont été développés de manière à pouvoir procéder aux phases de simulation et d'exploration (surface, rapidité, consommation) dès les premières étapes de la conception. En effet, moins la description des unités de traitement et d'interconnexion est détaillée, plus celles-ci sont implicites et peuvent être rapidement évaluées par les outils d'exploration. De plus, ces langages sont utili-

sés de manière à fournir des informations d'implémentation aux compilateurs flexibles qui ont ensuite la tâche de générer des codes machines adaptés à ces architectures. Dans ce chapitre, nous ferons un bref résumé sur les ADL. Des études plus détaillées, [45] et [46], apporteront d'avantages d'informations. Il convient de discerner trois catégories d'ADL, remarquables par leurs caractéristiques : structurel, comportemental et mixte.

A.1 Langages de description structurels

La description d'une architecture à l'aide d'un langage ADL de type structurel consiste en une description de cette architecture au niveau transferts de registres (RTL). Ce type de description typiquement matérielle se rapproche des langages VHDL ou Verilog. L'avantage d'une telle description réside dans la possibilité de modéliser concrètement les structures d'une architecture et de maîtriser de manière très précise les ressources logiques implémentées. Ceci constitue également le principal inconvénient lorsqu'il s'agit de décrire une architecture complexe de part la quantité de ressources à décrire, surtout lorsqu'il s'agit de procéder à l'exploration et à la simulation de cette architecture. Par exemple, MIMOLA [47] utilise une *netlist* de composants définis au niveau RTL. Une description MIMOLA est très proche d'une description VHDL. Dans l'exemple présenté listing 3-1, la description débute par la déclaration d'un module nommée `Alu` suivi de ses ports d'entrée-sortie. L'architecture est ensuite définie entre les balises `CONBEGIN` et `CONEND` pour les différentes opérations réalisées dans l'ALU. Certains langage ADL structurels comme UDL/I [48] sont spécialisés dans les descriptions particulières comme les processeurs superscalaires ou les processeurs VLIW, ce qui limite leur intérêt.

```
1  MODULE Alu
2     (
3        IN i1, i2 : (15:0);
4        OUT outp : (15:0);
5        IN ctr: (1:0)
6     );
7  CONBEGIN
8     outp <- CASE ctr OF
9        0: i1+i2;
10       1: i1-i2;
11       2: i1 AND i2;
12       3: i1;
13       END AFTER 1;
14 CONEND;
```

Listing 3-1 – Exemple de description MIMOLA d'une ALU

A.2 Langages de description comportementaux

À l'opposé des langages ADL structurels se trouvent les ADL de type comportementaux. Les ADL de type comportemental permettent une spécification de l'architecture sous forme de sémantiques d'instructions. Cela présente l'avantage d'ignorer les détails de la structure matérielle de manière à se concentrer sur l'aspect fonctionnel de l'architecture. Il est alors

très rapide et très facile de simuler et de procéder à une exploration des performances de l'architecture produite. Mais, encore une fois, le principal avantage des ADL comportementaux se trouve être également le principal inconvénient puisque cette fois-ci, il devient très difficile de contrôler les ressources qui seront générées. Par exemple, nML [49] permet la description d'un jeu d'instructions de manière hiérarchique par l'intermédiaire d'instructions partielles. Dans l'exemple proposé dans le listing 3-2, la définition de l'instruction `num_instruction` combine trois instructions partielles réalisables en parallèle. Par exemple, la première instruction partielle `num_action` permet l'exécution au choix de `add` ou `sub` ou `mul` ou `div`. Toutes les dérivations possibles de l'instruction `num_instruction` sont alors le produit de la taille des instructions partielles `num_action`, `SRC` et `DST`.

```
1  op num_instruction(a:num_action, src:SRC, dst: DST)
2  action {
3      temp_src = src;
4      temp_dst = dst;
5      a.action;
6      dst = temp_dst;
7  }
8  op num_action = add | sub | mul | div
9  op add()
10 action = {
11     temp_dst = temps_dst+temp_src
12 }
```

Listing 3-2 – Exemple de description nML d'une ALU

A.3 Langages de description mixtes

Enfin, les langages ADL de type mixte sont à mi-chemin entre les ADL comportementaux et structurels. Il s'agit d'une extension des ADL comportementaux pour lesquels il a été introduit la possibilité d'inclure des notions de ressources matérielles permettant ainsi de trouver un compromis acceptable entre la rapidité de l'exploration et la simplicité de la description. Citons par exemple l'ADL ARMOR [50]. Un jeux d'instructions de type ARMOR se fait par la définition de règles. L'ensemble de ces règles forme une grammaire dont chaque dérivation représente les instructions "légales". Une description d'architecture consiste à définir une règle de haut niveau (*InstructionSet*) qui décrit le jeux d'instructions et les règles de groupes (*gp*). Le comportement de chaque instruction est défini par les règles *df* (*dataflow instructions*) qui modélisent le chemin des données et les règles de contrôle pour les instructions correspondantes. Ces règles permettent de définir les opérateurs soit au niveau transferts de registres, soit à l'aide d'une liste d'opérations. La définition du parallélisme de tâche est également possible. Par exemple, examinons la description d'une architecture homogène présentée dans le listing 3-3. L'architecture décrite est composée d'une file de registres et de trois unités de traitement. Le jeu d'instructions est composé soit de trois instructions de type *dataflow* en parallèle appelées `instDF` soit d'une instruction de contrôle appelée `instCTRL`. En revanche, une restriction est spécifiée en ce qui concerne les instructions `store` et `memAccess` qui ne peuvent être exécutées en parallèle.

À notre connaissance, il n'existe pas de réel ADL dédié à la description générique d'archi-

```
1        InstructionSet = [ instDF || instDF || instDF ] | instCTRL
2        restriction ! [ memAccess || store]
3        gp instDF = compute | move | memAccess | nop
4        gp memAccess = load | store
```

Listing 3-3 – Exemple de description ARMOR

tectures reconfigurables dynamiquement. MADEO [25] ou VPR [32] sont spécialisés dans le domaine des FPGA, tandis que MAML [51] et Adres [21] sont respectivement spécialisés dans les architectures massivement parallèles ou pour un modèle d'architecture paramétrable. Cependant, les langages de description mixtes ne sont pas sans intérêt pour le domaine du reconfigurable. En effet, des concepts de reconfiguration peuvent être automatisés et leur utilisation dans un ADL ne requiert pas une description approfondie des mécanismes mis en œuvre. Au contraire, les unités de traitement ont, à priori, besoin d'être détaillées pour une implémentation très ciblée d'algorithmes spécifiques.

Dans ce chapitre, nous présentons un langage de description innovant dédié à la description d'architecture reconfigurables. Celui-ci, xMAML, basé sur MAML (*MAchine Marked up Language*) [51], permet la spécification de paramètres de reconfiguration dynamique propre aux mécanismes présentés dans le chapitre II.

B MAML : langage de description d'architectures massivement parallèles

L'ADL mixte MAML a été développé à l'origine afin de faciliter la conception d'architectures de processeurs massivement parallèles. La description d'une architecture massivement parallèle avec MAML se conçoit à deux niveaux d'abstractions. Le premier niveau, le niveau bas appelé *Processor Element Level*, concerne la description de l'architecture de chaque processeur qui composera l'architecture. Cette description se fera en termes de ressources ou de capacité de calcul. Le deuxième niveau, le plus élevé appelé *Processor Array Level*, est quand à lui dédié à la description des interactions entre chaque élément processeur. Cela concerne les interconnexions ou encore la position de chaque élément processeur (figure 3-2).

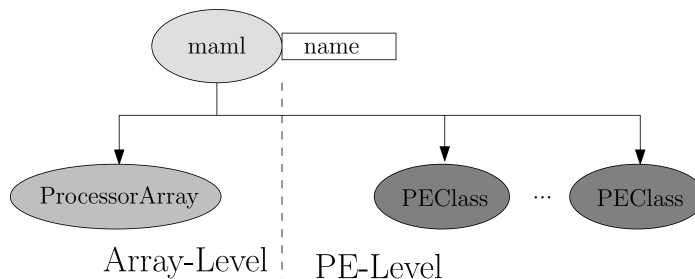


Figure 3-2 – **Caractéristiques d'une description d'architecture en MAML**. *Deux niveaux d'abstraction se distinguent, le niveau bas, niveau d'architecture d'un élément processeur (*PE-Level*) et le niveau haut, niveau d'interaction entre éléments processeurs (*Array-Level*).*

B.1 Description au niveau haut : *Processor Array level*

Le *Processor Array level* est nécessaire à la spécification des paramètres génériques de toute l'architecture (figure 3-3). Le premier de ces paramètres `<PElements>` définit le nombre d'éléments processeurs qui composeront l'architecture. Le nombre d'éléments processeurs est défini sous forme de matrice fixant par la même occasion le nombre de lignes et le nombre de colonnes de l'architecture. Cependant, le nombre de colonnes multiplié par le nombre de lignes n'est pas nécessairement égal au nombre d'éléments processeurs réellement implémentés dans l'architecture. La grille ainsi constituée permet seulement de servir de référentiel d'implémentation des différentes ressources tels que processeurs, mémoires ou ressources d'entrée-sortie. Le paramètre `<PEInterconnectWrapper>` spécifie le schéma d'interconnexion de la matrice par l'intermédiaire des IW (*Interconnect Wrapper*) que l'on peut traduire par "enveloppe d'interconnexion". Chaque élément processeur (PE), dont l'architecture peut être individuellement spécifiée, se voit placé à l'intérieur d'un IW. Tous les IW d'une matrice sont identiques. C'est pourquoi les paramètres de l'IW sont spécifiés au niveau *Array-Level* de la description MAML. Le paramètre `<ICDomain>` pour *Interconnect Domain* spécifie l'ensemble des éléments processeurs qui partageront la même topologie d'interconnexion. Par exemple, la figure 3-4 montre une architecture partagée en quatre domaines d'interconnexion. L'ensemble des PE qui compose chaque domaine partage le même schéma d'interconnexion comme cela est présenté dans la figure 3-5). Le paramètre `<ClassDomain>` spécifie l'ensemble des éléments processeurs qui partagent la même architecture.

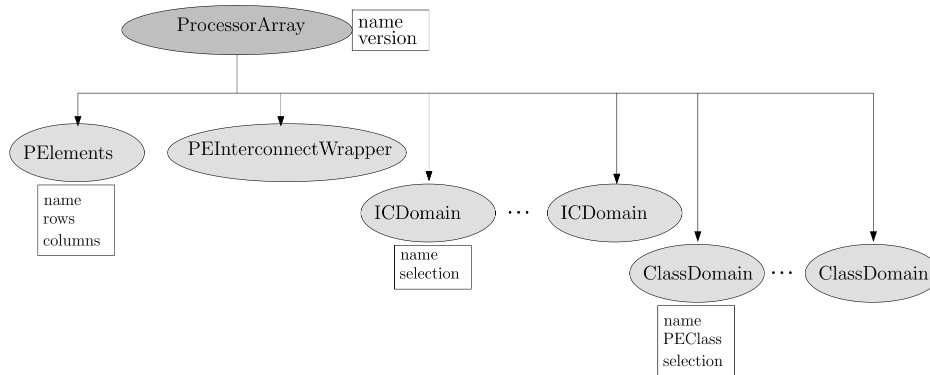


Figure 3-3 – **Paramètres nécessaire à la description d'une architecture au niveau *Array Level*.** *Le niveau d'abstraction* Array Level *permet de définir des schémas d'interconnexion pour les éléments processeurs qui composeront l'architecture. Les éléments processeurs qui partagent le même schéma d'interconnexion seront regroupés en* domaines.

B.2 Description d'un élément processeur : *PE level*

La spécification de l'architecture d'un élément processeur est décrite dans la partie *PE-level* d'une description MAML (figure 3-6). Il est, par exemple, possible de spécifier les caractéristiques des ports d'entrée/sortie en termes de taille de données, de direction (entrée, sortie ou bidirectionnel), ou de définir si il s'agit d'un port de contrôle ou de données. Ensuite, il est nécessaire de préciser les ressources internes qui composeront l'élément processeur tels que des

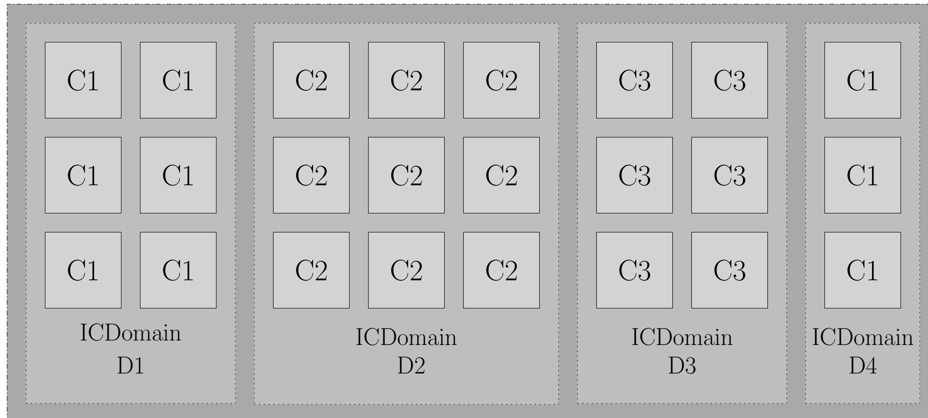


Figure 3-4 – **Exemple de domaines d'interconnexion**. *Les domaines définissent des zones de calculs dont les ressources partagent un même schéma d'interconnexion par l'intermédiaire des IW. Dans cet exemple, quatre domaines sont définis, dont deux domaines D1 et D4 qui partagent les mêmes architectures d'éléments processeurs.*

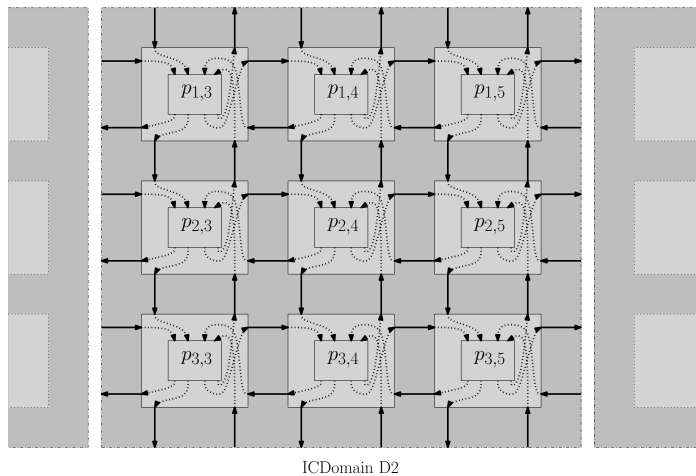


Figure 3-5 – **Exemple d'implémentation d'un domaine au niveau des interconnexions**. *Cet exemple montre que tous les éléments processeurs du domaine D2 partagent un même schéma d'interconnexion.*

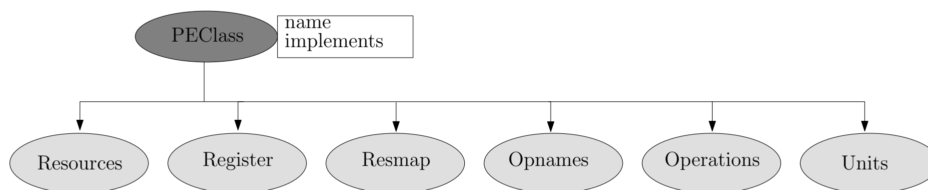


Figure 3-6 – **Paramètres nécessaires à la description d'un élément processeur**. *L'architecture d'un élément processeur est flexible, sa spécification se fait par l'intermédiaire de paramètres.*

unités fonctionnelles, des bus ou encore des éléments mémoires (files de registres, mémoires d'instructions ou FIFO). La description possible d'un élément processeur par l'intermédiaire de son jeu d'instructions permet de caractériser MAML comme étant un ADL mixtes.

B.3 Synthèse

Cette section a permis de montrer que la description d'architectures massivement parallèles est simplifiée par l'intermédiaire des concepts adaptés de MAML. Cependant, notre objectif étant la conception d'architecture reconfigurables dynamiquement, il nous faut apporter certaines extensions à ce langage. Les extensions apportées permettent la description d'architectures hétérogènes aussi bien en termes de motifs de calculs que de processus de reconfiguration. Ainsi, grâce à l'introduction de ces concepts spécifiques à la reconfiguration dynamique, nous pourrons exploiter le modèle générique de reconfiguration dynamique présenté au chapitre précédent, et ce, de la manière la plus efficace possible.

C xMAML : Modélisation d'une architecture reconfigurable dynamiquement

Dans cette section, nous présentons le langage ADL xMAML spécifiquement dédié à la description d'architectures reconfigurables dynamiquement. xMAML est nécessaire à Mozaïc afin de fournir les informations utiles à la mise en œuvre du modèle de reconfiguration dynamique. Dans ce but, nous avons développé des concepts nécessaires à simplifier la description en plus de ceux déjà présents dans MAML.

C.1 Présentation générale d'une description xMAML

Une description xMAML est composée d'un ensemble de ressources dont le nombre varie en fonction de l'architecture à décrire. La figure 3-7 résume ces différentes ressources. Tout d'abord, les DyRIBox, unités d'interconnexion utiles aux communications internes à l'architecture. Les unités de traitement sont spécifiées selon leur emplacement dans l'architecture (ClassDomain) et selon leurs ports de communication nécessaires aux traitements des données (PEInterface), ainsi qu'à la reconfiguration dynamique. La spécification d'un domaine (DBDomaine) permet de définir les unités de traitement et d'interconnexion qui font partis de la même zone de reconfiguration. Enfin, les communications externes à l'architecture se font à travers les ports spécifiés dans la partie *PortMapping* de la description.

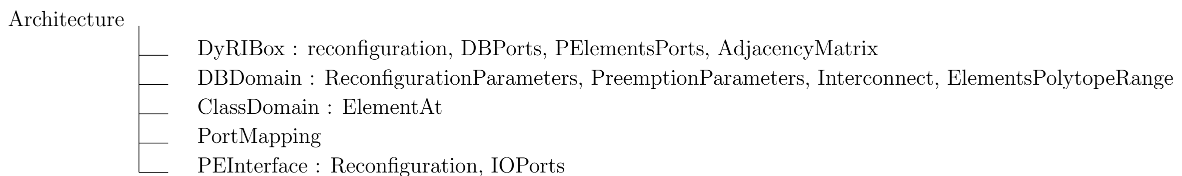


Figure 3-7 – **Paramètres nécessaires à la description d'une architecture en xMAML.** *Une architecture est un ensemble d'unités d'interconnexion (*DyRIBox*), de domaines de reconfiguration (*DBDomain*), d'unités de traitement (*ClassDomain *qui assigne chaque emplacement de la matrice à un PE et* PEInterface *qui spécifie les ports d'interface d'un PE), de ports d'entrée/sortie (*PortMapping*).*

C.2 Modèle xMAML du bloc d'interconnexion DyRIBox

Une architecture reconfigurable est constituée d'un ensemble de ressources organisées autour d'un réseau de connexion fléxible. La première des modifications concerne donc la spécificité des interconnexions. En effet, MAML permet la description et la spécification d'architectures régulières en termes de ressources de calculs et d'interconnexion. Si l'on observe quelques architectures reconfigurables dynamiquement telles que DART [12] ou l'ATMEL AT40K [1] entre autres, les communications internes sont réalisées avec plusieurs niveaux hiérarchiques d'interconnexion. Des interconnexions courtes distances permettent des communications entre ressources voisines, alors que des interconnexions longues distances permettent des communications plus rapides entre des ressources éloignées. Ces concepts n'existent pas à l'origine dans MAML. Les architectures reconfigurables implémentent aujourd'hui des ressources très différentes en termes de motif de calcul. Ces ressources, qu'elles soient de type algorithmique, tel qu'un processeur, ou bien de type logique, tel qu'un FPGA ou encore de type sauvegarde, telles que des mémoires RAM, ROM, FIFO, files de registres, doivent être capables de communiquer par l'intermédiaire d'un réseau d'interconnexion flexible et reconfigurable. En effet, l'intégration et l'implémentation de ressources différentes peut nécessiter la mise en place de ressources d'interface chargées de la régularisation des processus de reconfiguration. L'unité d'interconnexions, DyRIBox (section C du chapitre II), est chargée d'assurer la continuité des communications entre les unités de traitement hétérogènes implémentées sur l'architecture, même pendant la reconfiguration. La description xMAML d'une DyRIBox, représentée sur la figure 3-8, est effectuée par l'intermédiaire de trois catégories de paramètres :

1. le paramètre d'identification,

2. les paramètres de reconfiguration dynamique,

3. les paramètres structurels (ports d'entrée/sortie et schéma d'interconnexion).

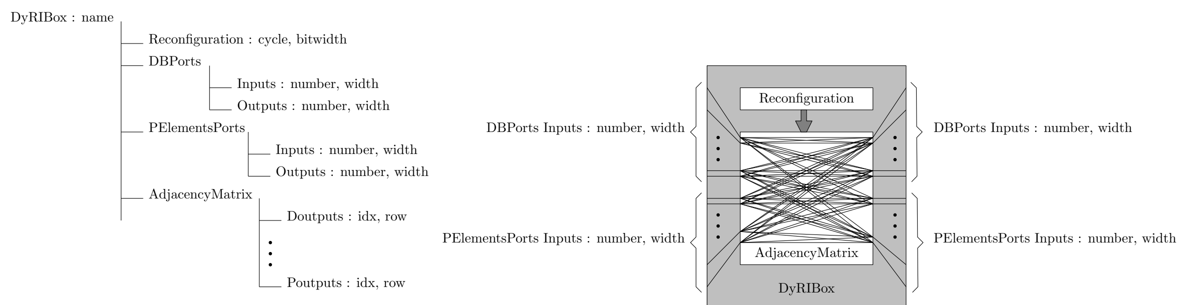


Figure 3-8 – **Paramètres nécessaires à la description d'une DyRIBox**. *Trois catégories de paramètres sont nécessaires à la description d'une DyRIBox. Après lui avoir donné un nom, il est nécessaire de spécifier les paramètres de reconfiguration, les paramètres structurels tels que les ports d'entrée/sortie (DBPorts, PElementsPorts), ainsi que le schéma de connexion qui permettra, le cas échéant, de restreindre le nombre de connexions possibles (AdjacencyMatrix).*

C.2-1 Paramètre d'identification

Au sein de l'architecture, l'identification d'une DyRIBox est nécessaire afin d'autoriser les instanciations de plusieurs modèles d'interconnexion dans l'architecture. Comme montré ligne 1 du listing 3-4, cette identification se fait par l'intermédiaire de la balise `name` placée après la balise `PEInterconnectDyRIBox` qui permet d'identifier le début d'une description de ressource de type DyRIBox.

C.2-2 Paramètres de reconfiguration dynamique

La description d'une DyRIBox à l'aide de xMAML permet de définir les paramètres nécessaires à la génération du modèle synthétisable du routeur de connexion. Le premier paramètre, `Reconfiguration`, concerne la reconfiguration dynamique. Ce paramètre sera utilisé afin de générer les ports d'entrée et de sortie du flot de données de configuration selon les sous paramètres `bitwitdh` et `cycle`. Le paramètre `bitwitdh` permet de spécifier la taille du bus de configuration auquel sera connecté le DUCK de la DyRIBox. Le sous paramètre `cycle` permet de spécifier le nombre de cycles de reconfiguration nécessaires à la DyRIBox pour modifier son schéma d'interconnexion. Ce paramètre modifie directement le nombre de ports d'échange de données entre la DyRIbox et le DUCK qui lui est associé. Il pourrait être légitime de s'interroger sur la pertinence de ce paramètre dans la description d'une DyRIBox. En effet, nous avons porté toute notre attention sur la possibilité de pouvoir reconfigurer une DyRIBox en un cycle afin de modifier un schéma d'interconnexion suffisamment rapidement sans avoir à perturber les communications non concernées par la reconfiguration. Cependant, le coût en surface induit n'est pas indispensable dans certains cas. Par exemple, si une tâche occupe la totalité de la zone, et que celle-ci est remplacée par une autre tâche tout aussi grande, aucune communication ne se fera entre deux configurations. Il est donc possible de procéder à des reconfigurations un peu plus longues en temps, mais qui permettront d'économiser de la surface.

C.2-3 Paramètres structurels

Le nombre et le type des ports d'entrée-sortie qui composeront la DyRIBox sont des paramètres à inclure dans la spécification xMAML. Il existe deux types de port dont la distinction est utile aux outils de placement-routage. Tout d'abord, les `DBPorts` spécifient les ports de communication de DyRIBox à DyRIBox ou bien de DyRIBox vers un port d'entrée-sortie de l'architecture. Ensuite, les `PElementsPorts` sont dédiés aux communications depuis/vers une DyRIBox vers/depuis une unité de traitement. La direction vers laquelle le port devra envoyer les données est spécifiée par la balise `Inputs` si il s'agit d'une entrée, `Outputs` si il s'agit d'une sortie, et `IO` si il s'agit d'un port bidirectionnel. La taille des ports de communication est spécifiée par la balise `bitwidth`. Cette valeur s'exprime en nombre de bits. Enfin, le nombre de ports de chaque direction est spécifié par la balise `number` dont la valeur sera un entier positif.

Le dernier paramètre qui compose la description d'une DyRIBox est celui qui permet la spécification du schéma d'interconnexion. `AdjacencyMatrix` permet de limiter le nombre de

connexions réalisables entre les ports d'entrée et les ports de sortie, chaque sortie est repérée par son index (idx) spécifié par une valeur binaire. Chaque bit de cette valeur binaire sert à modéliser la connexion entre la sortie en cours de description et une entrée (bit à 1 pour autoriser la connexion, bit à 0 sinon). L'attribution d'une valeur à un bit se fait dans l'ordre croissant des index en commençant par les entrées de type DBPorts, puis les entrées de type PElementsPorts. L'ensemble de ces bits est appelé row. Concrètement, si l'on considère une sortie ayant n connexions possibles et dont le row a pour valeur $bit_{n-1}...bit_m bit_{m-1}...bit_0$, alors le bit_{n-1} modélise la connexion avec l'entrée provenant d'une DyRIBox ayant pour index un, le bit_{n-2} modélise la connexion avec l'entrée ayant pour index deux et ainsi de suite jusqu'au bit_0. Il en va de même pour la modélisation des connexions avec les entrées provenant des unités de traitement. Dans l'exemple de la figure 3-9, la DyRIBox est composée de trois ports d'entrée et un port de sortie de type *DBPorts*, de trois ports d'entrée et un port de sortie de type *PElementsPorts*. Seules quatre connexions peuvent être configurées sur la première sortie (DBPorts S0). Le code xMAML qui permet de spécifier ce schéma de connexion est alors : <DOutput idx="0" row="101101" />. En ce qui concerne la sortie "PElementsPorts S0", seules trois connexions sont possibles (DBPorts E1, PElementsPorts E0 et PElementsPorts E2), la spécification xMAML de cette connexion est donc : <POutput idx="0" row="010110" />.

Figure 3-9 – **Exemple de description de type *AdjacencyMatrix***. *Dans cet exemple, la DyRIBox est composée de trois ports d'entrée et un port de sortie de type* DBPorts*, de trois ports d'entrée et un port de sortie de type* PElementsPorts*. La spécification de l'*AdjacencyMatrix *permet de restreindre les connexions possibles entre les entrées et les sorties.*

C.2-4 EXEMPLE DE DESCRIPTION D'UNE DYRIBOX EN xMAML

La description xMAML d'une unité d'interconnexion est donnée en exemple. Cette description reprend les concepts présentés précédemment et donne un exemple précis de la manière dont une description doit être effectuée tel que décrit dans la figure 3-10. La description donnée spécifie une unité d'interconnexions qui sera directement connectée à quatre DyRIBox en entrée et en sortie. Parmi les cinq sorties, quatre ports seront connectés à des DyRIBox, et un port sera connecté à une unité de traitement. Le listing 3-4 présente la description xMAML associée à cet exemple de DyRIBox. Quatre ports d'entrée (ligne 4) et de sortie (ligne 5) de type DBPorts sont spécifiés comme ayant une taille de huit bits. Un port d'entrée (ligne 8) et un port de sortie (ligne 9) de type PElementsPorts sont également présents. La reconfiguration des interconnexions est réalisée en un cycle (ligne 2). Les lignes 11 à 17,

spécifient les restrictions d'interconnexions. Par exemple, la ligne 12 indique que la sortie d'index "0" connectée à une DyRIBox accepte les connexions provenant du port d'entrée `DBPorts` d'index "1", "2", "3" et du port d'entrée `PElementsPorts` d'index "0", mais pas du port d'entrée `DBPorts` d'index "0". À cela s'ajoute deux ports de communication destinés à une unité de traitement dont la sortie ne peut également pas être connectée directement sur l'entrée de cette unité de traitement.

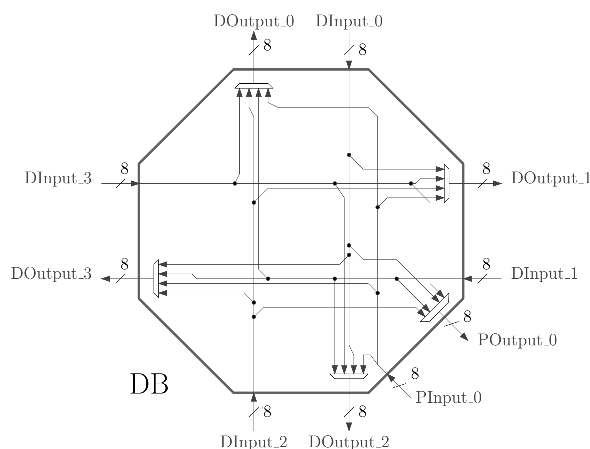


Figure 3-10 – **Exemple de DyRIBox**. *Cette DyRIBox est composée de cinq ports d'entrées et de cinq port de sorties. Parmi les ports d'entrées, quatre ports proviennent eux même de DyRIBox, et un port provient de l'unité de traitement.*

```
1  <PEInterconnectDyRIBox name="DB">
2     <ReconfigurationTime cycle="1"/>
3     <DBPorts>
4        <Inputs number="4" bitwidth="8" />
5        <Outputs number="4" bitwidth="8" />
6     </DBPorts>
7     <PElementsPorts>
8        <Inputs number="1" bitwidth="8" />
9        <Outputs number="1" bitwidth="8" />
10    </PElementsPorts>
11    <AdjacencyMatrix>
12       <DOutput idx="0" row="01111" />
13       <DOutput idx="1" row="10111" />
14       <DOutput idx="2" row="11011" />
15       <DOutput idx="3" row="11101" />
16       <POutput idx="0" row="11110" />
17    </AdjacencyMatrix>
18 </PEInterconnectDyRIBox>
```

Listing 3-4 – Description en xMAML de la DyRIBox de la figure 3-10

C.3 NOTION DE DOMAINE DE RECONFIGURATION

C.3-1 INTRODUCTION GÉNÉRALE

L'un des enjeux de la reconfiguration dynamique se situe dans la possibilité de procéder à la reconfiguration partielle d'une architecture. Il est alors nécessaire, dès la description de

l'architecture, de définir des zones de reconfiguration indépendantes les unes des autres. Dans le langage de description original, la notion de domaine était utilisée afin de regrouper les éléments processeurs en fonction de leur schéma d'interconnexion. Nous avons étendu la notion de domaines afin de regrouper les ressources appartenant à la même zone de reconfiguration. Un domaine est une zone où la reconfiguration dynamique est exécutée de manière homogène en termes de taille de mots de configuration et en termes de temps nécessaire à la reconfiguration. Cela permet de pouvoir procéder à des reconfigurations dynamiques partielles et indépendantes et donc parallèles. Ces aspects de la reconfiguration dynamique ont été présentés dans le chapitre précédent. À cela s'ajoute la possibilité de procéder à la reconfiguration partielle du domaine en question, de manière à permettre l'implémentation de sous tâches indépendantes entre elles. Un domaine requiert quelques paramètres de spécification listés dans la figure 3-11. Tout d'abord, la spécification débute par les paramètres de reconfiguration dynamique (`ReconfigurationParameters`) et de gestion de preemption (`PreemptionParameters`). S'en suit la spécification concernant les paramètres structurels permettant l'instanciation des unités d'interconnexion DyRIBox ainsi que la description des interconnexions internes non reconfigurables entre les différentes ressources (`interconnect`). Enfin, l'instanciation des unités de traitement est réalisée lors de la spécification du `ElementsPolytopeRange`.

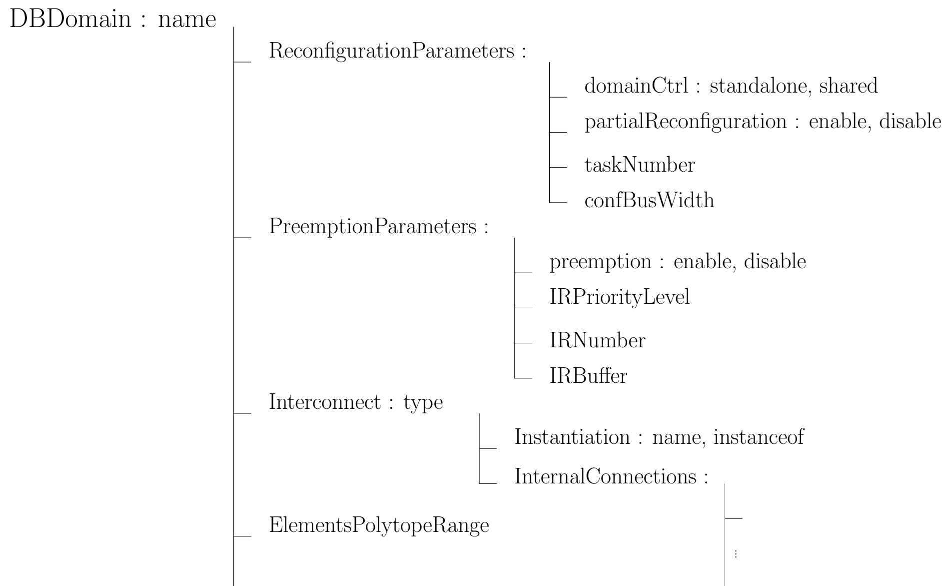


Figure 3-11 – **Paramètres nécessaires à la description d'un Domaine**. *Tout comme la spécification d'une DyRIBox, trois catégories de paramètres entrent dans la spécification d'une architecture : le paramètre d'identification, les paramètres de reconfiguration dynamique (*`ReconfigurationParameters` *et* `PreemptionParameters`*) et les paramètres structurels (*`Interconnect` *pour ce qui concerne les interconnexions internes au domaine et* `ElementsPolytopeRange` *pour ce qui concerne le placement des unités de traitement).*

C.3-2 Description et formalisation des Domaines

Paramètre d'identification : la création d'un domaine commence tout d'abord par la déclaration de son nom `<DBDomain name="`*nom_du_domain*`">` (figure 3-11). Ceci permettra

par la suite de pouvoir procéder à la description des interconnexions entre domaines.

Paramètres de reconfiguration dynamique : les paramètres de reconfiguration dynamique sont divisés en deux catégories : les paramètres liés à la gestion de la reconfiguration (`ReconfigurationParameters`) et les paramètres liés à la gestion des interruptions (`PreemptionParameters`). Afin de générer les ressources de reconfiguration qui seront nécessaires au contrôle du domaine spécifié, il convient de définir les paramètres de reconfiguration. Ceux-ci sont identifiés par la balise `ReconfigurationParameters`. Des paramètres de contrôle doivent ensuite être précisés. Le premier d'entre eux concerne la manière dont sera gérée la reconfiguration sur le domaine. Cela se fait par l'intermédiaire du paramètre `domainCtrl`. Une gestion séparée et individuelle de la reconfiguration dynamique du domaine nécessitera l'initialisation de ce paramètre à la valeur `standalone` alors qu'une gestion commune des processus de reconfiguration avec un contrôleur général imposera la valeur `shared`. La justification de la création de plusieurs domaines vient du fait que pour certaines cibles d'architecture, les flots de données de configuration sont trop importants pour pouvoir procéder à des configurations rapides. Cela permet, de plus, de pouvoir procéder à des reconfigurations partielle de l'architecture. Le paramètre qui permet la génération des ressources utiles à la mise en œuvre de la reconfiguration partielle est `partialReconfiguration`. Celui-ci peut prendre les valeurs `enable` ou `disable`. La taille de la mémoire de configuration est calculée en fonction du paramètre `taskNumber`. Ce paramètre spécifie le nombre de tâches implémentables par le domaine et prend une valeur n, avec $n \in \mathbb{N}$. Le dernier paramètre concerne la taille du bus de configuration qui sera utlisé par les registres DUCK dont on spécifie la valeur par l'intermédiaire de la balise `confBusWidth`.

Les paramètres qui concerne la gestion des interruptions, et sous entendu le processus de préemption sont regroupés dans les `PreemptionParameters`. Les interruptions ne sont pas gérées par l'attribution de la valeur `disable` au paramètre `preemption`. Dans le cas contraire, la valeur `enable` implique la spécification du nombre de niveau de priorité géré (`IRPriorityLevel`), ainsi que le nombre d'entrée d'interruption (`IRNumber`) par niveau d'interruption et la taille de la FIFO qui permet de sauvegarder les différents appels d'interruptions intervenues et non executées (`IRBuffer`).

Paramètres structurels : un domaine est une instanciation de plusieurs ressources que l'on peut classer en deux catégories. Les unités d'interconnexion que nous avons présentées dans la section précédente, et les unités de traitement que nous présenterons par la suite. Les interconnexions, quelles soient reconfigurables ou non, sont spécifiées dans la partie `Interconnect`. Les unités d'interconnexion mettant en œuvre la reconfiguration dynamique, sont associées à un domaine par l'instruction `Instantiation` suivie du nom choisi à donner à cette instanciation (`name`) et du nom de la ressource d'origine avec `instanceOf`. Cela permet d'utiliser un seul modèle de schéma d'interconnexion que l'on peut répliquer dans l'architecture, et ainsi simplifier la description. Le paramètre `InternalConnections` permet de définir les différentes connexions statiques entre tous les éléments qui composent un domaine. Le format de spécification d'une connexion à une DyRIBox est légèrement différent du format de spécification d'une connexion à une unité de traitement. Prenons d'abord le cas de la

DyRIBox. Le port à connecter devra être spécifié de la manière suivante : `nom_de_la_DyRIBox : type_du_port (index_du_port )`, où `nom_de_la_DyRIBox` est le nom de l'instance de la DyRIBox spécifiée dans la description du domaine, `type_du_port` a pour valeur soit `in`, `out` ou `io` selon si il s'agit d'un port d'entrée, de sortie ou bi-directionnel, et enfin `index_du_port` est le numéro du port que l'on souhaite connecter.

De la même manière, le format de spécification de la connexion à une unité de traitement devra être déclaré de la manière suivante : `nom_de_la_ressource : nom_du_port (plage_index_port)`, où `nom_de_la_ressource` est le nom de l'instance de la ressource spécifiée dans la description des classes d'unités de traitement (que nous détaillerons par la suite), `nom_du_port` est le nom correspondant au port que l'on souhaite connecter et qui a été spécifié dans la déclaration de l'unité de traitement, et enfin `plage_index_port` est le numéro du port que l'on souhaite connecter. Par exemple si l'on souhaite connecter l'entrée "1" de la DyRIBox nommée `MultBus` aux bits "0" à "15" de la sortie `output_0` de la ressource `DataMem1`, la description sera : `<MultBus:in(1) = DataMem1:output_0(0:15)/>`.

La méthode d'instanciation des unités de traitement est différente. Celle-ci hérite directement de la méthode initialement utilisée par le langage de description MAML [51]. À l'origine, MAML étant utilisé pour la description d'architectures massivement parallèles, les unités de traitement étaient repérées par leur position sur une grille. L'association d'une ressource à un domaine consiste alors à délimiter les bornes de cette grille par l'instruction `ElementsPolytopeRange`. Le principe de cette spécification consiste à décrire sous forme matricielle les équations des droites qui définissent l'espace du domaine (`ElementsPolytopeRange`). Prenons par exemple un espace de domaine dont les unités de traitement forment un rectangle de huit éléments sur quatre, les quatre droites α, β, γ et δ, ayant pour équation $y_\alpha = 0$, $y_\beta = 4$, $x_\gamma = 0$ et $x_\delta = 8$ (figure 3-12) délimitent alors cet espace. Il suffit de représenter ces équations sous forme matricielle afin de pouvoir les modéliser dans le langage. Par exemple, la matrice 3-1) représente les équations délimitant l'espace d'intégration des unités de traitement à un domaine d'après l'exemple de la figure 3-12.

$$matrixA \times \begin{bmatrix} x \\ y \end{bmatrix} = vectorB \Longleftrightarrow \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 8 \\ 0 \\ 4 \end{bmatrix} \tag{3-1}$$

Notons que les ressources ainsi implémentées ne sont pas nécessairement homogènes en termes d'architecture. De plus, cette partie de la description MAML ne concerne pas directement la spécification de l'architecture de l'unité de traitement à implémenter. Le code xMAML, correspondant à la spécification de l'`ElementsPolytopeRange` de la matrice 3-1 et de la figure 3-12, est représenté dans le listing 3-5 lignes 14 à 23.

C.4 Description et fonctionnement des unités de traitement

Les unités de traitement qui devront être implémentées dans un domaine ne sont qu'une instance d'éléments existants dont le modèle synthétisable aura été préalablement décrit. Mozaïc permet d'interconnecter ces différentes ressources. Pour cela, il est nécessaire de

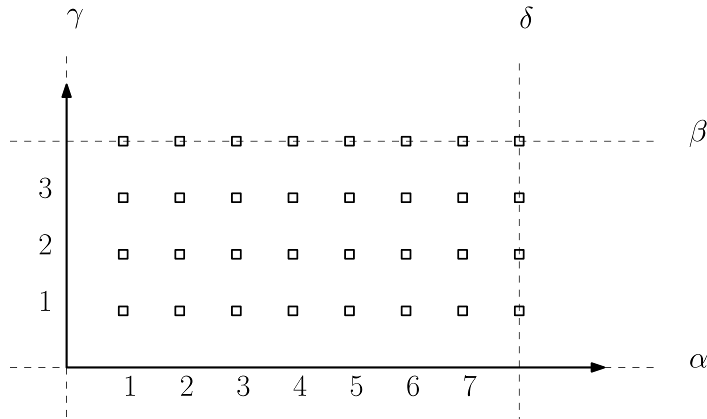


Figure 3-12 – **Exemple de droites délimitant les bornes extrêmes du polyèdre définissant l'espace du domaine**. *Le domaine est défini par les quatre droites α, β, γ et δ, ayant pour équation $y_\alpha = 0$, $y_\beta = 4$, $x_\gamma = 0$ et $x_\delta = 8$. La norme définie en MAML impose la description de ces équations sous forme matricielle.*

```
1  <DBDomain name="DPR_1" >
2     <ReconfigurationParameters preemption="disable" domainCtrl="shared"
          partialReconfiguration="enable" IRPriorityLevel="3" taskNumber="10"
          confBusWidth="8"/>
3     <Interconnect type="manual" >
4     <Instantiation name="MultBus" instanceOf="DBox"/>
5     <InternalConnections >
6        <MultBus:in(1) = DataMem1:output_0(0:15)/>
7        <MultBus:in(2) = DataMem2:output_0(0:15)/>
8        .
9        .
10       .
11       <AG4:output_0(0:15) = DataMem4:input_0(0:15)/>
12    </InternalConnections >
13    </Interconnect >
14    <ElementsPolytopeRange >
15       <MatrixA row = " 1 0"/>
16       <MatrixA row = " 1 0"/>
17       <MatrixA row = " 0 1 "/>
18       <MatrixA row = " 0 1 "/>
19       <VectorB value = " 0"/>
20       <VectorB value = " 8"/>
21       <VectorB value = " 0 "/>
22       <VectorB value = " 4"/>
23    </ElementsPolytopeRange >
24 </DBDomain >
```

Listing 3-5 – Exemple de description d'un domaine

procéder à une description précise des ports d'interconnexion ainsi que des paramètres de reconfiguration qui serviront à la génération des ressources d'interface. Les ressources d'interface permettent une homogénéité dans le processus de reconfiguration définis par le concept DUCK de Mozaïc. Le reste de cette section détaille la manière dont les unités de traitement sont décrites en xMAML.

Les différentes balises nécessaires à la description des interfaces d'une unité de traitement

sont présentées figure 3-13. La description d'une unité de traitement commence par la balise `PEInterface`. Celle-ci est suivie du nom de la ressource à instantier `name="`*nom_ressource*`"`. Par exemple, l'instanciation d'une unité de traitement de type CLB est présentée à la première ligne du listing 3-6. Le paramètre *nom_ressource* doit être identique au nom du fichier du modèle synthétisable se trouvant dans la librairie de composants. Afin de prévoir une interface de configuration adaptée à la ressource, il faut ensuite spécifier le nombre de cycles nécessaires à sa reconfiguration (`cycle`), ainsi que le nombre de bits contenus dans une configuration (`bits`). Le paramètre `preemption` permet de prévoir les ressources pour l'extraction de la configuration venant d'être remplacée au cas où l'architecture de la ressource concernée le permettrait.

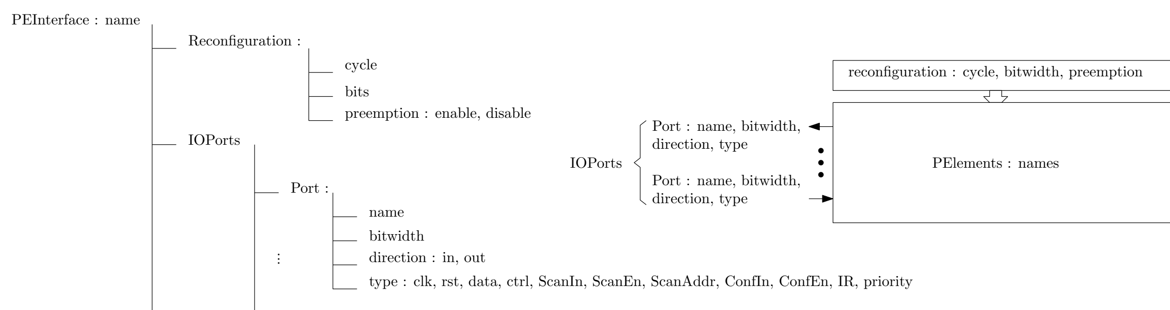


Figure 3-13 – **Paramètres nécessaires à la description d'un PEInterface**. *La description d'une unité de traitement nécessite avant tout les paramètres nécessaires à la génération du DUCK adapté. Cela inclue les paramètres de reconfiguration dynamique (taille, temps de reconfiguration et gestion de la préemption). Les ports de communications sont également spécifiés par la balise `IOPorts` afin de permettre la spécification des connexions entre les différentes unités du domaine.*

Il convient ensuite de préciser dans la partie `IOports` le nombre et la nature des ports qui devront être interconnectés. La description d'un port (`Port`) inclut son nom (`name`), sa taille (`bitwidth`), son sens (`direction`) et enfin son type (`type`). Le type du port peut être soit `data` spécifiant que celui-ci permettra l'acheminement des données nécessaires aux calculs à exécuter, soit `ctrl` spécifiant que les données qui transitent sont utiles au contrôle de la ressource, soit `rst` spécifiant le port de réinitialisation, soit `clk` specifiant le port d'horloge. Les ports de configuration doivent être traités à part, étant donnée leur complexité. En effet, nous avons vu dans le chapitre précédent que deux méthodes majeures de reconfiguration sont utilisées dans les architectures reconfigurables. La première consiste à adresser une mémoire qui contient localement tous les mots de configuration dont a besoin l'unité de traitement. La deuxième consiste à venir écrire directement le mot de configuration dans des registres pouvant être adressés directement par des ports spécifiques. Dans le cas d'une configuration dans une mémoire RAM, il nous faut spécifier le port d'écriture qui sera alors de type `ScanIn`, le port de validation qui sera alors de type `ScanEn` et enfin le port d'adresse qui sera de type `ScanAddr`. Dans le cas d'une configuration directe dans des registres, il est possible de spécifier le port d'écriture qui sera alors de type `ConfIn` et le port de validation qui sera alors de type `ConfEn`. Enfin, dernier type exclusivement réservé aux sorties, le type `IR` qui doit nécessairement être d'une largeur de un bit et suivi d'un indice de priorité dont la balise est `priority`. Le type

`IR` permet de générer les connexions avec le contrôleur d'interruptions.

```
1  <PEInterface name="clb">
2     <Reconfiguration cycle="16" bits="1" preemption="no"/>
3     <IOPorts>
4        <Port name="luti0" bitwidth="1" direction="in" type="data" />
5        <Port name="luti1" bitwidth="1" direction="in" type="data" />
6        <Port name="luti2" bitwidth="1" direction="in" type="data" />
7        <Port name="luti3" bitwidth="1" direction="in" type="data" />
8        <Port name="clbout" bitwidth="1" direction="out" type="data" />
9        <Port name="cin" bitwidth="1" direction="in" type="data" /
10       <Port name="cout" bitwidth="1" direction="out" type="data" />
11       <Port name="reset" bitwidth="1" direction="in" type="rst"/>
12     <Port name="H" bitwidth="1" direction="in" type="clk"/>
13       <Port name="ConfigIn" bitwidth="1" direction="in" type="RAMConfIn"/>
14     <Port name="RW" bitwidth="1" direction="in" type="RAMConfEn"/>
15     <Port name="ConfigAdre" bitwidth="4" direction="in" type="RAMConfAdr"/>
16    </IOPorts>
17 </PEInterface>
```

Listing 3-6 – Exemple de description d'une interface d'unité de traitement de type CLB

C.5 DyRIOBlocs : blocs d'entrée-sortie

La description d'une entité de type DyRIOBloc dans la description xMAML permet de définir les paramètres nécessaires à la génération du modèle synthétisable d'un bloc d'entrée-sortie. Un DyRIOBloc permet de faire communiquer les domaines avec le monde extérieur. Le premier paramètre, `Reconfiguration`, concerne la reconfiguration dynamique. Ce paramètre sera utilisé afin de générer les ports d'entrée et de sortie du flot de données de configuration selon le sous paramètre `bitwitdh`. Le sous paramètre `cycle` permettra de déterminer le nombre de cycle de reconfiguration nécessaires au DyRIOBloc pour modifier son schéma d'interconnexion. Cependant, si l'on souhaite pouvoir procéder à des reconfigurations dynamiques sans perturber les éventuelles communications ne devant pas être reconfigurées, ce paramètre doit être fixé à "1". Le sens des communications est fixé par différentes balises selon la configuration choisie : `in`, `out` et `inout`. La taille en nombre de bits du port de connexion est spécifiée par la balise `bitwidth` et la taille de la mémoire tampon associée par la balise `buffer`. Tout comme les DyRIBox, ces ports d'entrée-sortie sont reconfigurables dynamiquement. Cette reconfiguration est nécessaire pour les ports bidirectionnels afin de déterminer le sens de communication pour une tâche donnée. Cette reconfiguration est également nécessaire pour déterminer la taille des mémoires tampon d'entrée-sortie pour les ports qui en seront pourvus. Par exemple, dans le listing 3-7, un bloc d'entrée-sortie est spécifié de sorte que le port bidirectionnel `io1` puisse être configuré en un cycle. Les deux autres ports `i1` et `o1` ne nécessitent une reconfiguration que pour limiter la taille du *buffer*.

C.6 Synthèse

Dans cette section, nous avons présenté les paramètres introduits dans le langage MAML et qui ont donné naissance au langage xMAML. La spécification de ces paramètres permet la génération de tout types d'architectures reconfigurables existantes ou nouvelles. La conception

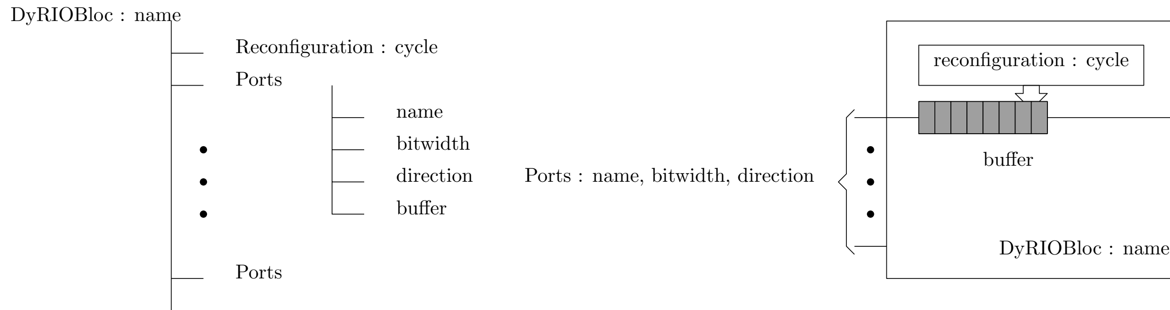


Figure 3-14 – **Paramètres nécessaires à la description d'un DyRIOBloc**. *La spécification des paramètres d'une unité d'interconnexion de type DyRIOBloc commence par son identification, et la spécification du temps de reconfiguration. Viennent ensuite les descriptions propres aux différents ports présents sur le DyRIOBloc.*

```
1 <DyRIOBloc name="IO">
2    <Reconfiguration cycle="1"/>
3    <Port name="i1" bitwidth="1" direction="in" buffer="4"/>
4    <Port name="io1" bitwidth="1" direction="inout" buffer="4"/>
5    <Port name="o1" bitwidth="1" direction="out" buffer="4"/>
6 </DyRIOBloc>
```

Listing 3-7 – Description d'un DyRIOBloc

de nouvelles architectures se fait par l'intégration d'unités de traitement développées au préalable dont la gestion individuelle de la reconfiguration dynamique est assurée par la génération automatique des DUCK adaptés. Dans la section suivante, nous allons présenter l'outil dédié à la génération automatique de l'ensemble des ressources nécessaires à la gestion individualisée de chaque unité de l'architecture.

D Outil d'exploration des paramètres de reconfiguration dynamique

La spécification d'une architecture à l'aide de xMAML détermine certains paramètres de reconfiguration. Selon la valeur donnée à ces paramètres, les caractéristiques F-D-P de l'architectures seront modifiées. La phase d'exploration permet de déterminer comment les performances de reconfiguration et les résultats de synthèse seront modifiés par les choix qui seront faits à la conception. Dans cette section, nous détaillons la manière dont les différents paramètres sont évalués. Tout d'abord, nous analysons l'impact de l'introduction des ressources de reconfiguration sur la surface et la consommation. Puis, dans une deuxième partie, nous présentons les méthodes permettant l'estimation de la taille des configurations des ressources d'interconnexion, et dans une troisième partie, l'estimation de la taille des configurations des unités de traitement. Enfin, la dernière partie de cette section montre l'évolution de l'ensemble de ces paramètres pour différentes tailles d'architecture à base de DPR de DART et de CLB du Xilinx XC4000.

D.1 EXPLORATION EN SURFACE ET CONSOMMATION DES RESSOURCES DE RECONFIGURATION DYNAMIQUE

Afin de permettre une première estimation des ressources de reconfiguration produites automatiquement suite à l'introduction du concept DUCK, nous avons procédé à une série de synthèses en technologie CMOS 130 nm avec l'outil Synopsys. La méthode d'exploration que nous avons définie consiste à analyser le code xMAML de l'architecture et d'en extraire les paramètres des DUCK qui seront générés. Trois paramètres sont nécessaires à une première estimation de la surface et de la consommation. Le premier concerne l'architecture des DUCK spécifique à la méthode de reconfiguration de l'unité à laquelle chaque DUCK est dédié, séquentielle, par adressage ou parallèle. Chacune de ces architectures influe sur la surface et la consommation de manière différente. Le deuxième et le troisième paramètre concernent respectivement le nombre et la taille des registres utilisés dans un DUCK. Ces trois paramètres constituent donc la base de l'exploration sur la reconfiguration dynamique de la plate-forme MOZAÏC. Trois séries de synthèse ont donc été effectuées en faisant varier ces trois paramètres les uns après les autres et sont regroupées par architecture de DUCK explorée.

D.1-1 EXPLORATION DE LA RESSOURCE DUCK

L'homogénéité de la reconfiguration est assurée par l'introduction des DUCK dans le chemin de configuration. L'impact de cette introduction est analysée en termes de surface et de consommation.

RÉSULTATS DE SYNTHÈSE SUR DES DUCK ADAPTÉS À UNE RECONFIGURATION SÉQUENTIELLE : les résultats de synthèse représentés sur la figure 3-15, montrent que l'évolution de la quantité de ressources et de la consommation induite peut être estimée en fonction du nombre de registres DUCK (Rn) et de la taille de chacun de ces registres (Rs). La courbe représentant la surface utilisé peut être approximée par l'équation :

$$SD_{seq} = 174,3 + Rn \times Rs \times 27,6 \tag{3-2}$$

avec SD_{seq} représentant la surface totale en μm^2 d'un DUCK à reconfiguration séquentielle, Rn représentant le nombre de registres DUCK et Rs représentant la taille en bits des registres DUCK. La courbe représentant la consommation peut être approximée par l'équation :

$$CD_{seq} = 0,2852 + Rn \times Rs \times 0,0155 \tag{3-3}$$

avec CD_{seq} représentant la consommation totale en mW d'un DUCK à reconfiguration séquentielle.

RÉSULTATS DE SYNTHÈSE SUR DES DUCK ADAPTÉS À UNE RECONFIGURATION PAR ADRESSAGE : les mêmes séries de synthèse ont été faites sur des DUCK à reconfiguration par adressage (figure 3-16). Les résultats obtenus sont très proches des résultats de synthèses effectuées sur des DUCK à reconfiguration séquentielle car leur architecture est également très proche.

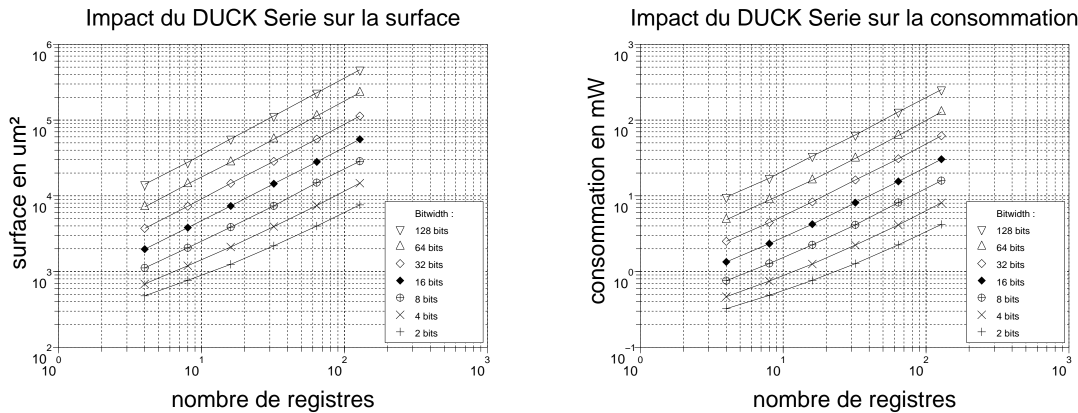


Figure 3-15 – **Exploration de la surface et de la consommation induite par les DUCK en mode séquentiel**. *Les données représentées sur ce graphique permettent d'estimer rapidement la surface de silicium et la consommation d'un DUCK en mode de configuration séquentielle.*

Seules quelques ressources supplémentaires sont nécessaires au DUCK à reconfiguration par adressage afin de gérer les adresses mémoires comme il se doit. La différence est plus notable pour des ressources DUCK qui implémentent peu de registres et de petite taille alors que les résultats convergent pour un nombre et des tailles de registres plus conséquents, la proportion de registres parmi l'ensemble des ressources d'un DUCK étant alors très importante. Les droites obtenues peuvent être approximées par les mêmes équations que pour les DUCK à reconfiguration séquentielle.

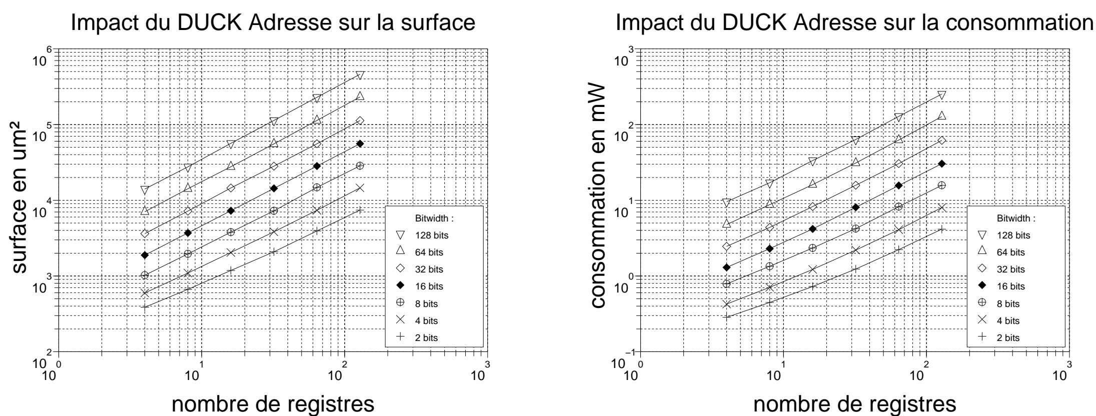


Figure 3-16 – **Exploration de la surface et de la consommation induite par les DUCK en mode adressage**. *Les données représentées sur ces graphiques permettent d'estimer rapidement la surface de silicium et la consommation induites par l'introduction du concept DUCK en mode de configuration par adressage.*

RÉSULTATS DE SYNTHÈSE SUR DES DUCK ADAPTÉS À UNE RECONFIGURATION PARALLÈLE : la configuration parallèle ne nécessite pas l'implémentation de compteur. Seuls

quelques ports supplémentaires sont nécessaires pour connecter les registres DUCK directement aux registres de configuration. Par conséquent, la reconfiguration s'effectue en un cycle. Les résultats obtenus à la figure 3-17 montrent tout de même que cela implique une surface de silicium quasiment doublée par rapport aux deux autres architectures de DUCK précédentes. La courbe représentant la surface (SD_{para}) utilisé peut être approximée par l'équation :

$$SD_{para} = 562,9 + Rn \times Rs \times 49,8 \tag{3-4}$$

La courbe représentant la consommation (CD_{para}) peut être approximée par l'équation :

$$CD_{para} = 1,16 + Rn \times Rs \times 0,0246 \tag{3-5}$$

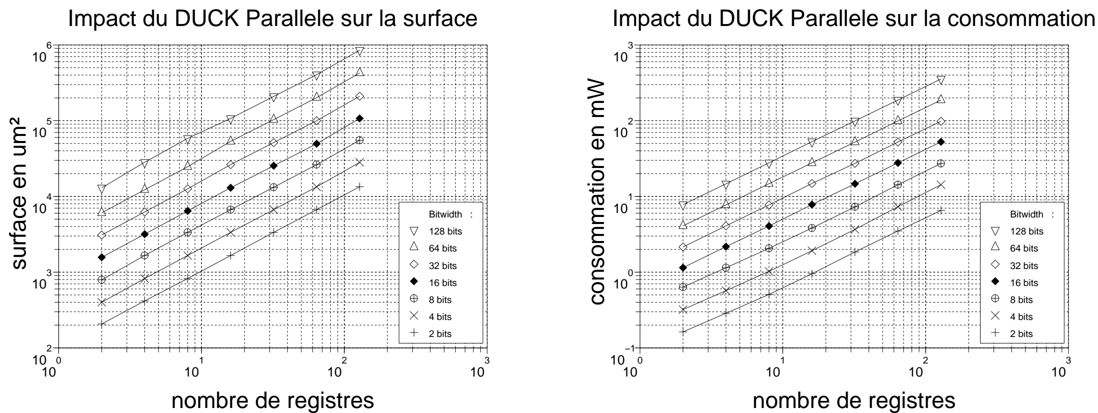


Figure 3-17 – **Exploration de la surface et de la consommation induite par les DUCK en mode parallèle**. *Les données représentées sur ce graphique permettent d'estimer rapidement la surface de silicium et la consommation induites par l'introduction du concept DUCK en mode de configuration parallèle.*

D.1-2 Exploration de la DyRIBox

Afin de caractériser la DyRIBox, nous avons procédé à une série de mesures effectuées sur des modèles de DyRIBox puis analysé les paramètres de surface de silicium et de consommation. Ces paramètres sont analysés en fonction de la taille des entrées/sorties et du nombre de connexions possibles sur une sortie. Les figures 3-18 (a) et (b) montrent que l'importance de la taille des données sur la surface totale de silicium et sur la consommation tend à diminuer pour des tailles de plus en plus grandes. Cependant, cette différence tend à s'estomper lorsque l'on augmente le nombre de connexions possibles sur une sortie. Ceci est notamment dû au fait que le contrôle de la connexion est indépendant de la taille des données à traiter, ce qui implique qu'à partir d'une certaine taille de multiplexeur utilisé, ceux-ci ont proportionnellement plus d'influence sur la surface de silicium utilisée. En revanche, une plus faible consommation sera atteinte lorsque l'on privilégiera des connexions restreintes sur des données les plus larges possible.

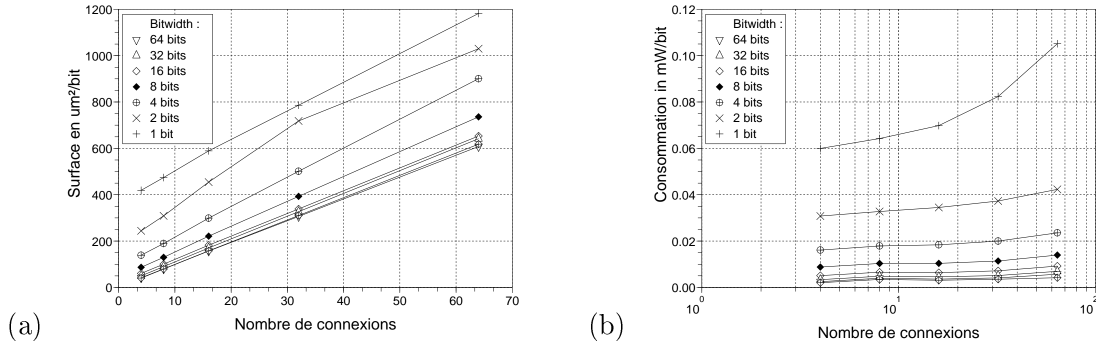


Figure 3-18 – **Influence de la taille des données et du nombre de connexions par sortie sur la surface de silicium (a) et sur la consommation (b).**

Solution	Surface en mm^2	Fréquence en MHz	Consommation en $\mu W/MHz$
4S project	0.0506	1075	16.12
DyRIBox	0.0526	1444	5.00

Tableau 3-1 – Résultats de synthèse de DyRIBox et comparaison avec la méthode du projet 4S.

Comparons maintenant les performances de la DyRIBox pour un modèle de connexion utilisé dans le projet 4S [42]. L'article présente un ensemble de routeurs à connexion directe interconnectés entre eux. Un routeur est composé de 5 ports bidirectionnels de 16 bits interconnectés par un *crossbar* 16×20. Les résultats de synthèse de notre DyRIBox ayant les mêmes caractéristiques, pour une même technologie sont présentés dans le tableau 3-1 et montrent que notre routeur permet un gain en fréquence de reconfiguration de 25% ainsi qu'un gain en consommation de 69% pour une augmentation de la surface de silicium de seulement 4%.

D.2 Données de configuration des interconnexions

L'unité d'interconnexions DyRIBox a besoin d'un certain nombre de bits de configuration pour un bon fonctionnement. La taille des registres de configuration est déterminée par le nombre d'entrées pouvant être connectées sur une sortie. Toutefois, tous les registres se doivent d'être homogènes en termes de taille pour une même DyRIBox. C'est pourquoi, la taille des registres de l'ensemble des registres de configuration d'une DyRIBox sera définie par le plus grand d'entre eux. Si l'on considère C comme étant le plus grand nombre de connexions parmi toutes les sorties d'une DyRIBox, alors chaque registre aura pour taille $T_r = log_2(C)$. Pour une DyRIBox, le nombre total de registres de configuration (TC_r) sera alors la somme des entiers supérieurs au logarithme de base deux de C :

$$TC_r = \sum_{n=0}^{No-1} \lceil log_2(C) \rceil \quad (3\text{-}6)$$

avec No représentant le nombre de sorties à configurer et C le plus grand nombre de connexions

sur une sortie parmi l'ensemble des sorties de la DyRIBox. La configuration de l'ensemble des DyRIBox d'un domaine (TCR_r) a donc pour taille :

$$TCR_r = \sum_{i=0}^{N_{db}-1} \sum_{n=0}^{No_i-1} \lceil log_2(C_i) \rceil \quad (3\text{-}7)$$

avec N_{db} représentant le nombre de DyRIbox contenues dans le domaine, No_i le nombre de sorties sur la DyRIBox i, et C_i le plus grand nombre de connexions sur une sortie parmi l'ensemble des sorties de la DyRIBox. TCR_r permet d'estimer le nombre brut de bits de configuration nécessaires à la configuration des DyRIBox. Il est cependant nécessaire de prendre en compte les éventuels registres "fantômes" induits par l'interface entre les registres DUCK et les registres de configuration. La section B.3 du chapitre II détaille l'intérêt de l'utilisation des registres "fantômes". Le calcul du nombre de ces registres "fantômes" est déterminé en fonction de la taille du bus de configuration qui traverse l'ensemble des DUCK d'un domaine et la taille des registres de chaque DyRIBox. Si l'on note T_d la taille des registres du DUCK, le nombre total de registres utilisés pour la configuration d'une DyRIBox (TC_{fr}), registres "fantômes" compris vaut :

$$TC_{fr} = \left\lceil \frac{T_r}{T_d} \right\rceil \times T_d \quad (3\text{-}8)$$

avec T_r représentant la taille brute des registres de la DyRIBox concernée. La taille du flot de configuration complet concernant les DyRIBox $TCRN_d$ est alors :

$$TCRN_d = \sum_{n=0}^{N_{db}-1} TC_{fr}n \quad (3\text{-}9)$$

avec $TC_{fr}n$ représentant le nombre net de bits de configuration de la DYRIBox n, N_{db} représentant le nombre de DyRIbox contenues dans le domaine.

D.3 DONNÉES DE CONFIGURATION DES UNITÉS DE TRAITEMENT

La taille du flot de données de configuration des unités de traitement est déterminée par la partie `PEInterface` de la description xMAML. Celle-ci intègre le paramètre `bit` qui permet de définir la taille de la configuration de la ressource concernée. La valeur `bit` détermine directement la variable $TCpe_n$ représentant la taille de la configuration de l'unité de traitement n. La taille brute, en nombre de registres, du flot de configuration des unités de traitement (TCR_{pe}) est alors :

$$TCR_{pe} = \sum_{n=0}^{N_{pe}-1} TCpe_n \quad (3\text{-}10)$$

avec N_{pe} représentant le nombre d'unités de traitement à configurer. La taille nette ($TCRN_{pe}$), en nombre de registres du flot de configuration des unités de traitement est elle donnée par :

$$TCRN_{pe} = \sum_{n=0}^{N_{pe}-1} \left\lceil \frac{TCpe_n}{T_d} \right\rceil \times T_d \quad (3\text{-}11)$$

D.3-1 Temps de propagation de contexte

Le temps nécessaire à la propagation d'un contexte complet ($Prop_t$) est déterminé en fonction de la taille des flots de données de configuration des DyRIBox ($TCRN_d$), de la taille des configurations des unités de traitement ($TCRN_{pe}$), de la fréquence de lecture de la mémoire de configuration (Fr_{mem}), et de la taille du bus de configuration (T_d). On a alors :

$$Prop_t = \frac{TCRN_d + TCRN_{pe}}{T_d \times Fr_{mem}} \tag{3-12}$$

D.3-2 Temps de préemption de contexte

Le temps nécessaire à la propagation d'un contexte complet ($Pree_t$) est déterminé en fonction de la taille des flots de données de configuration des unités de traitement ($TCRN_{pe}$), de la fréquence maximale d'écriture de la mémoire de configuration (Fw_{mem}), et de la taille du bus de configuration (T_d). En effet, la configuration des DyRIBox n'évolue pas au cours du traitement, il n'est donc pas nécessaire de procéder à leur préemption. On obtient :

$$Pree_t = \frac{TCRN_{pe}}{T_d \times Fw_{mem}} \tag{3-13}$$

D.3-3 Détermination du nombre de domaine de reconfiguration

La plupart du temps, les contraintes temporelles imposées par les applications nécessitent le partage des domaines de reconfiguration. Le nombre de domaine de reconfiguration (N_D) est déterminé par le temps de propagation d'un contexte ($Prop_t$) et le temps disponible entre deux phases de reconfiguration selon l'équation :

$$N_D = \left\lceil \frac{Prop_t}{Ctxt_t - Pree_t} \right\rceil \tag{3-14}$$

avec $Ctxt_t$ représentant le temps disponible d'implémentation d'un contexte. Cette valeur dépend de l'application à implémenter.

E Synthèse

Dans ce chapitre, nous avons présenté un langage de description de haut niveau permettant l'exploitation des mécanismes de reconfiguration présenté au chapitre précédent. Grâce à xMAML, il est possible d'intégrer rapidement des unités de traitement par l'intermédiaire d'une spécification d'interface permettant la génération d'un réseau de DUCK et des contrôleurs de reconfiguration adaptés. Une analyse de la description xMAML permet une première estimation rapide de la surface de silicium et de la consommation d'énergie induite par la mise en œuvre de la reconfiguration dynamique. De plus l'analyse de la description permet également de déterminer la taille des flots de données de configuration propres aux ressources de connexions et aux unités de traitement ainsi qu'à l'estimation du nombre de domaines de reconfiguration nécessaires selon l'application choisie.

Chapitre

4

Validation de la plate-forme Mozaïc

Résumé : Dans ce chapitre, nous mettons en œuvre la plate-forme Mozaïc par la génération des ressources nécessaires à la reconfiguration dynamique sur deux architectures reconfigurables dynamiquement. La première est basée sur les unités de traitement implémentées par le processeur reconfigurable à grain épais DART. La deuxième est basée sur une architecture de FPGA embarqué basé sur une matrice de cellules logiques comparables à la famille XC4000 de chez Xilinx à laquelle on ajoute des possibilités de reconfiguration dynamique. Pour ces deux architectures, le concept de reconfiguration dynamique implémenté est directement généré par Mozaïc. L'application que nous avons choisie est un décodeur WCDMA particulièrement adapté à la mise en œuvre de la reconfiguration dynamique.

Sommaire

A Chaîne de codage et de décodage WCDMA . . . 96
A.1 Emetteur WCDMA . . . 96
A.2 Récepteur WCDMA . . . 97
B Implémentation d'un décodeur WCDMA sur FPGA embarqué . 99
B.1 Présentation du XC4000 . . . 100
B.2 Outils génériques de développement sur FPGA . . . 102
B.3 Adéquation WCDMA-FPGA . . . 106
B.4 Description xMAML du eFPGA . . . 107
B.5 Génération des mécanismes de reconfiguration dynamique pour eFPGA 110
C Implémentation d'un décodeur WCDMA sur DART . . . 114
C.1 Présentation de DART et des outils associés . . . 115
C.2 Adéquation WCDMA-DART . . . 119
C.3 Description xMAML de DART . . . 124
C.4 Génération des mécanismes de reconfiguration dynamique pour DART 129
D Synthèse . . . 133

Dans ce chapitre, nous allons mettre en application la plate-forme Mozaïc. Pour cela, nous allons procéder à l'implémentation d'un décodeur WCDMA sur deux architectures reconfigurables dynamiquement. La première architecture est une architecture de FPGA embarqué (eFPGA) à base de cellules logiques de la famille XC4000 de chez Xilinx [52]. La deuxième architecture est basée sur les unités de traitement du processeur reconfigurable grain épais DART [53]. Pour ces deux architectures, nous nous appuyons sur les outils de compilation et de synthèse spécifiques à chacune d'elle, de manière à générer les données de configuration qui seront utilisées par les ressources générées par Mozaïc. Pour chacune de ces deux architectures, nous présenterons tout d'abord l'application en adéquation avec leurs propres ressources de traitement. Puis, nous procéderons à leur description xMAML permettant la génération automatique des ressources propres à la gestion de la reconfiguration dynamique. Enfin, par des phases d'exploration, nous déterminerons l'impact sur la surface de silicium et de la consommation des ressources. La première partie de ce chapitre va s'attacher à présenter une chaîne de codage-décodage de type WCDMA.

A Chaîne de codage et de décodage WCDMA

L'application choisie comme démonstration est une application typique du domaine des télécommunications mobiles. Une communication de type WCDMA [54] se fait entre deux interlocuteurs : le terminal mobile et la station de base. Ces communications orientées sont différenciées selon le sens dans lequel l'information transite. Les voies montantes se réfèrent aux communications amorcées par le terminal mobile vers la station de base. Les voies descendantes se réfèrent, à l'inverse, aux communications amorcées depuis la station de base vers le terminal mobile. Les techniques de télécommunications de générations précédentes attribuaient à chaque utilisateur une fréquence d'émission-réception particulière. Une communication de type WCDMA exploite mieux la bande passante sur une seule fréquence en utilisant les techniques d'accès par étalement de spectre. Dans le cadre du développement de nos architectures, nous nous focaliserons sur les traitements réalisés au niveau du terminal mobile.

A.1 Emetteur WCDMA

L'émetteur d'un terminal mobile (figure 4-1) correspond, si l'on se réfère à la définition précédente, à la voie montante de la communication. La nature des informations à transmettre va déterminer le canal logique qui sera utilisé. Les principaux canaux logiques pouvant être transmis par le terminal mobile transportent soit des données, appelées DPDCH (*Dedicated Physical Data CHannel*) soit des informations de contrôle, appelées DPCCH (*Dedicated Physical Control CHannel*). Ces informations de contrôle concernent la gestion de la puissance, le facteur d'étalement utilisé pour les données transmises et enfin une séquence de bits permettant d'estimer le canal à la réception (bits pilotes). Ces canaux sont organisés en trames de 10 ms composées chacune de 15 *slots*. Les données $d(k)$ à transmettre sur la voie montante possèdent un débit Ds variant de 15 $Ksymboles.s^{-1}$ à 960 $Ksymboles.s^{-1}$. La mise en canal consiste à multiplier les données par un code OVSF (*Orthogonal Variable*

Spreading Factor) orthogonal de longueur variable et unique par utilisateur. L'élément correspondant à un symbole du code est dénommé *chip*. Sa durée Tc est inférieure à la durée Ts du symbole à transmettre. Le débit Dc du code est fixé à 3,84 $Mbits.s^{-1}$. Le rapport entre la durée d'un symbole et la durée d'un *chip* définit le facteur d'étalement SF. L'identification et la séparation des différents canaux se fait grâce à l'intercorrélation entre deux codes OVSF dans la mesure où ceux-ci sont synchronisés. Enfin, une modulation de phase à quatre états, QPSK (*Quadrature Phase Shift Keying*), est appliquée au signal brouillé.

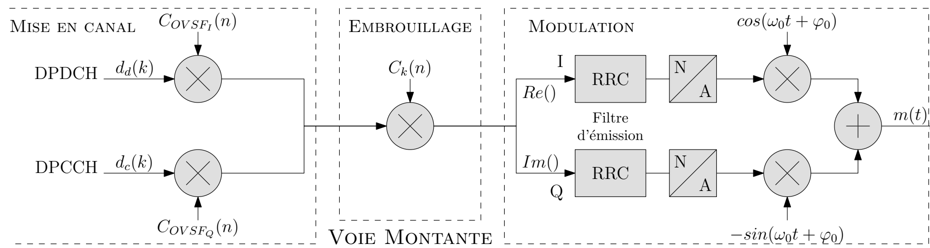


Figure 4-1 – **Synoptique d'un émetteur WCDMA de terminal mobile**. *La séparation et l'identification des différents canaux de transmissions se fait en les multipliant par un code OVSF. La modulation de phase à quatre états adapte le signal au support de transmission.*

A.2 Récepteur WCDMA

De part la présence de multiples chemins de propagation de l'onde radioélectrique entre l'émetteur et le récepteur, le signal reçu se compose de plusieurs répliques du signal émis. Chaque trajet se caractérise par un retard et une amplitude complexe. De plus, du fait de la nature même d'un terminal mobile, son déplacement peut conduire à une variation de la puissance du signal reçu appellée *fading*. Lorsque la combinaison des trajets est destructrice, le *fading* produit des évanouissements du signal qui conduisent à une réponse impulsionnelle du canal.

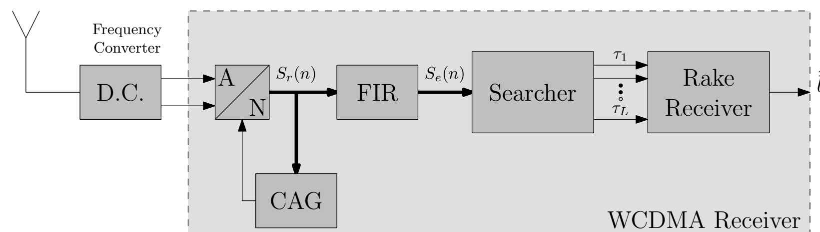


Figure 4-2 – **Synoptique d'un récepteur WCDMA de terminal mobile**. *Le signal reçu par l'antenne est d'abord translaté en bande de base avant d'être échantillonné (éventuellement sur-échantillonné pour une meilleur post-synchronisation). Le module CAG gère automatiquement le gain pour une meilleur exploitation de la dynamique d'échantillonnage. Les interférences entre symbole sont ensuite éliminées par le FIR. Enfin, le* Rake Receiver *combine linéairement toutes les sorties afin de réaliser une décision du symbole optimale.*

A.2-1 Réception et échantillonage

Dans un récepteur mono-utilisateur WCDMA (figure 4-2) tel qu'un téléphone mobile, le signal issu de l'antenne est premièrement translaté en bande de base avant d'être numérisé. Les données à traiter se retrouvent sous forme complexe, suite à la démodulation QPSK. Afin de compenser le phénomène de *fading*, le gain est géré automatiquement par le bloc de CAG (Contrôleur Automatique de Gain) permettant ainsi d'exploiter au mieux la dynamique du convertisseur analogique-numérique. L'étape de numérisation s'exécute à une fréquence de sur-échantillonage multiple de celle d'un *chip* ($f_{chip} = 3.84\ MHz$), ceci afin de faciliter la synchronisation entre l'émetteur et le récepteur. Cependant, le facteur de sur-échantillonnage N_{se}, devra faire l'objet d'un compromis entre une bonne précision de synchronisation et une puissance de calcul trop élevée.

A.2-2 Filtre de réception

Afin d'éliminer les interférences entre symboles, on applique un filtre de demi-Nyquist sur les voies réelles et imaginaires. Ces filtres FIR (*Finite Impulse Response*) contiennent 64 coefficients réels. La complexité de ces filtres est donc de $2 \times N_{SE} \times f_{chip} \times 64 MAC.s^{-1}$, soit 1966 MMACS (*Million Multiply Accumulate Cycles per Second*).

A.2-3 Estimation des trajets

Le bloc *searcher* de la figure 4-2 permet de déterminer le nombre de trajets utiles dans le signal reçu. Seuls les trajets dont la puissance est la plus importante pour le décodage seront transférés au *Rake Receiver*. Il existe plusieurs algorithmes développés pour réaliser cette fonction. La méthode classique, appelée PDP (*Power Delay Profile*), se déroule en trois phases comme montré dans la figure 4-3. La première de ces étapes consiste à multiplier le signal complexe filtré $r(p)$ du $n^{ième}$ *slot*, par le code pseudo-aléatoire propre à l'utilisateur concerné $s(p)$. Le signal corrélé $y(n,m)$ contient le nombre de pics pour les differents délais $\tau_{i,l}$.

La seconde étape consiste à déterminer les chemins les plus significatif. Pour ce faire, le *Power Delay Profile* $z(n,m)$ est déterminé sur la base du signal $y(n,m)$. Ce PDP correspond à la mesure de puissance pour chaque chemin trouvé.

Enfin, seul un nombre limité de chemins du PDP sont sélectionnés. Pour cela, une valeur du seuil η est déterminée pour la puissance du chemin.

A.2-4 Décodage des données, estimation de l'amplitude complexe du canal et synchronisation

L'étape de synchronisation est nécessaire avant le décodage des données. En effet, les codes générés au sein du récepteur ne sont pas forcément synchrones avec la réception des données issues des différents trajets. Cette synchronisation est réalisée en deux temps. Dans un premier temps, la méthode consiste à rechercher de manière séquentielle les maximums dans la fonction d'auto-corrélation entre le signal reçu et le code généré en interne. Seuls les trajets

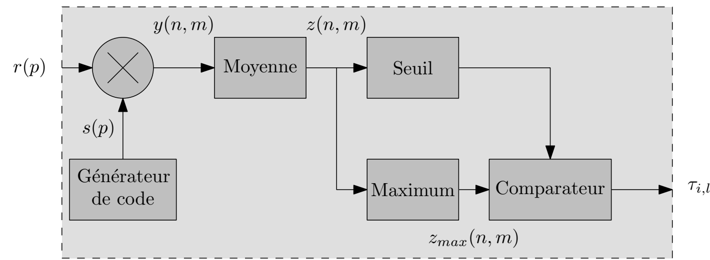


Figure 4-3 – **Synoptique d'un estimateur de trajets basé sur le *Power Delay Profile*.** *La méthode d'estimation des trajets* Power Delay Profile *est éxecutée en trois phases.*

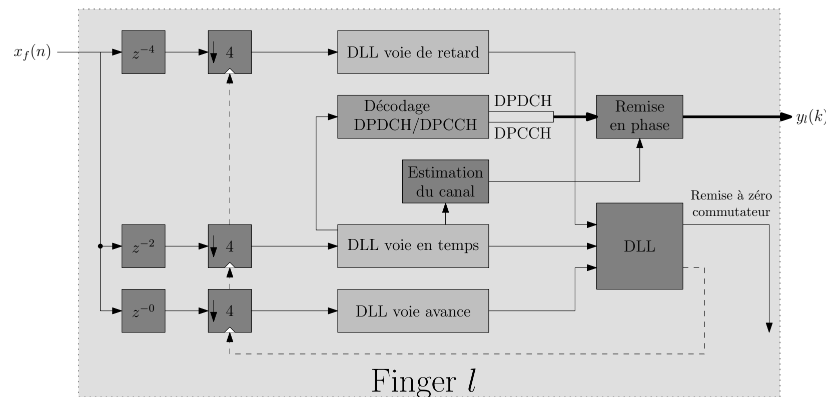


Figure 4-4 – **Synoptique d'un *finger*.** *Un* finger *a pour fonction l'estimation de canal, le décodage des données et enfin la synchronisation (réalisée par les DLL sur les voies en retard, en temps et en avance.*

dont la puissance est la plus élevée seront ensuite affectés aux L *fingers*. Chaque *finger* réalise le décodage des données, l'estimation de l'amplitude complexe du canal et la synchronisation pour un seul trajet l (figure 4-4). L'amplitude complexe du canal est estimée, pour chaque trajet, à chaque nouveau *slot*. Ce qui signifie que le canal est supposé invariant sur la durée d'un *slot*. Cette estimation consiste à corréler les bits pilotes reçus avec une séquence déterministe reconstituée au niveau du récepteur. Une séparation entre signaux ne peut se faire que si ceux ci sont reçus avec un retard supérieur à la durée d'un *chip*. Dans le cas d'un signal issu d'un multi-trajet, les autres signaux sont perçus comme des interférences et éliminés par le gain de traitement.

La combinaison des signaux issus des différents multi-trajets, au niveau de plusieurs *fingers* à raison de un par trajet, permet au *Rake Receiver* (figure 4-5) une estimation du symbole transmis.

B IMPLÉMENTATION D'UN DÉCODEUR WCDMA SUR FPGA EMBARQUÉ

Depuis l'apparition des premiers FPGA jusqu'à aujourd'hui, les architectures des cellules logiques ont, sur le principe, très peu changées. Les architectures actuelles de FPGA ne dif-

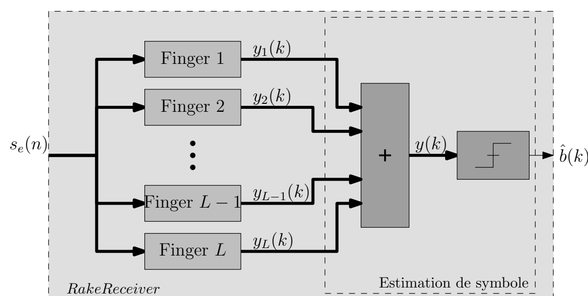


Figure 4-5 – **Synoptique d'un *Rake Receiver* comportant L fingers**. *Chaque* finger *prend en charge le traitement d'un trajet particulier. La combinaison linéaire de chaque symbole décodé au sein des différents* fingers *permet une prise de décision sur la valeur du symbole émis.*

fèrent que dans la densité des ressources implémentées et l'intégration de ressources dédiées supplémentaires. C'est pourquoi nous avons choisi de présenter l'implémentation d'un décodeur WCDMA sur un modèle d'architecture de FPGA embarqué à base de CLB de la famille XC4000 de chez Xilinx.

Afin de réduire le nombre de ressources nécessaire à une implémentation statique du décodeur WCDMA, nous allons procéder à son implémentation dynamique grâce à la plate-forme Mozaïc. Dans cette partie, les spécificités de l'architecture d'un bloc logique de XC4000 sur lequel nous nous basons pour la génération du FPGA embarqué sont détaillées. Ensuite, après avoir présenté les différents outils qui seront utilisés pour la conception de l'architecture et de l'implémentation du décodeur WCDMA, nous procéderons à l'exploration des performances de la reconfiguration dynamique obtenues et de l'impact de l'introduction des concepts sur la surface et la consommation du circuit.

B.1 Présentation du XC4000

Le FPGA XC4000 est basé sur une matrice de CLB. Un CLB, représenté par la figure 4-6, est composé de trois LUT, deux LUT à quatre entrées et une LUT à trois entrées. Des ressources de connexions flexibles sont également implémentées dans un CLB permettant une utilisation des LUT de manière individuelle ou de manière commune. Chacune des LUT est accompagnée de ressources utiles aux calculs arithmétiques et notamment de calcul de retenue afin d'accélérer et de simplifier les additions. Enfin, il est possible de configurer un CLB de manière à ce que le contenu des LUT puisse être modifié en cours de fonctionnement par le chemin de donnée. Ainsi, un CLB est utilisable comme le seraient deux mémoires RAM de 16×1 bits.

Le routage des communications entre deux CLB est configuré en fonction de la distance qui les séparent. Pour cela, différents types de bus sont disponibles. La figure 4-7 présente les cinq bus de communications pouvant être utilisés pour le routage. Les bus *direct connect* sont destinés aux communications entre CLB voisins, les bus *single*, *double*, *quad* et *long* sont utilisés respectivement par ordre croissant des distances à parcourir.

La conception d'une architecture de type FPGA nécessite l'utilisation conjointe de divers

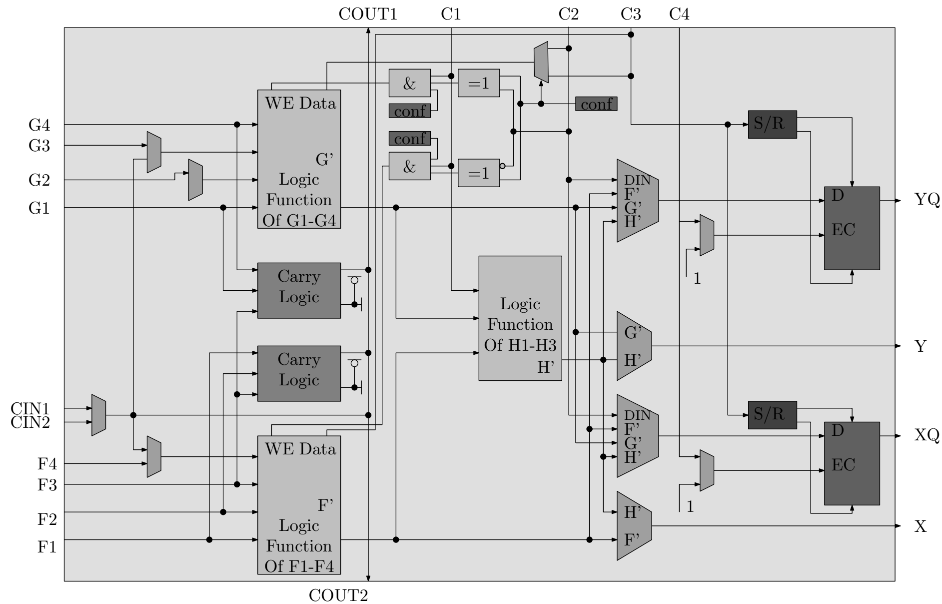


Figure 4-6 – **Schéma logique d'un CLB du XC4000**. *Un CLB de type XC4000 est composé de trois LUT, deux LUT à quatre entrées et une LUT à trois entrées. Celles-ci peuvent être utilisées soit de manière individuelle, soit de manière commune. Deux ressources arithmétiques de calcul de retenue sont également présentes afin d'accélérer et de simplifier les additions.*

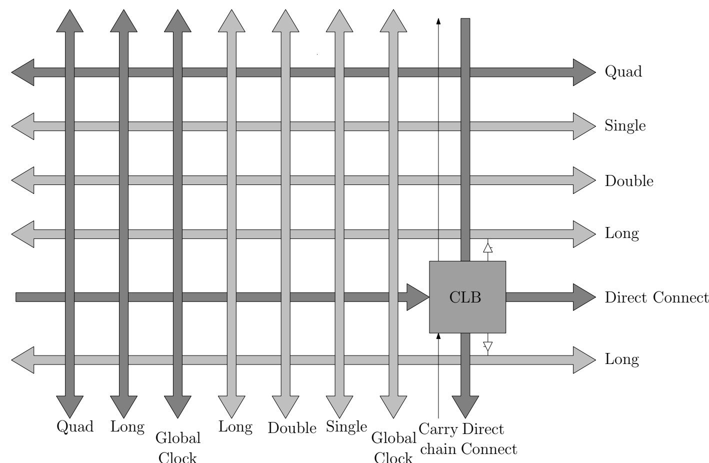


Figure 4-7 – **Plan des connexions sur XC4000**. *Selon les distances des connexions à établir entre les CLB, plusieurs routages sont possibles. L'outils de routage privilégiera les connexions directes pour les connexions entre CLB proches, ou par l'intermédiaire des bus* single, double, quad *ou* long *par ordre croissant des distances qui séparent deux CLB devant communiquer.*

outils tels qu'un outil de synthèse, de placement-routage ou encore d'un environnement de simulation. Afin de valider la plate-forme de développement Mozaïc, nous avons eu recours à des outils de recherche universitaires auxquels nous avons adjoint une série d'outils que nous avons développés. Ceux-ci, représentés sur la figure 4-8, sont utilisés pour la synthèse logique (ABC de l'université de Berkeley), et le placement/routage (VPR de l'université de Toronto).

La section suivante vise à détailler de quelle manière MOZAÏC s'intègre au flot de conception.

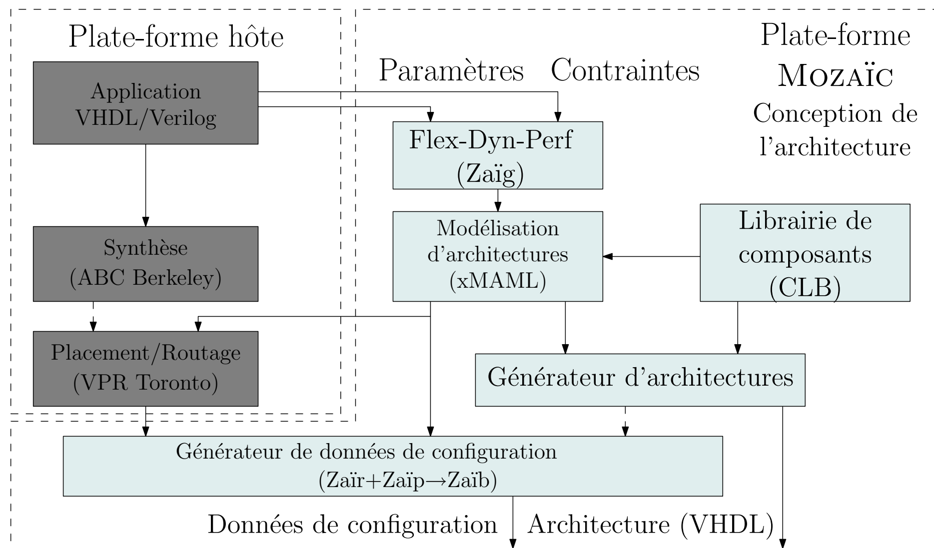


Figure 4-8 – **Plate-forme de développement MOZAÏC et son environnement**. *Le flot de conception* MOZAÏC *présenté dans cette figure s'intègre à un environnement dédié aux cibles FPGA. En plus des outils développés par nos soins s'ajoutent des outils dédiés à la synthèse (ABC de Berkeley) et au placement-routage (VPR de Toronto).*

B.2 OUTILS GÉNÉRIQUES DE DÉVELOPPEMENT SUR FPGA

B.2-1 SYNTHÈSE LOGIQUE DE L'APPLICATION : ABC BERKELEY

L'outils de synthèse ABC, développé par l'université de Berkeley [55], permet la synthèse logique à partir de modèles d'applications tels que Verilog. Le fichier de synthèse est généré au format BLIF (*Berkeley Logic Interchange Format*). Une description BLIF consiste en une liste de cellules standards tels que des portes logiques ou des bascules. Dans le cadre de la synthèse logique sur FPGA, cette description est uniquement composée de LUT et de bascules comme dans l'exemple présenté par le listing 4-1. Le résultat de la synthèse montre la manière dont une LUT à deux entrées (`n318` et `n324`) est utilisée, les valeurs binaires permettent de définir l'état de sortie du signal `n191`, la porte logique réalisée par la LUT est une fonction XOR réalisée entre les deux entrées. Ensuite, nous pouvons voir que le signal `g14` est une regénération synchronisée du signal `n191` par l'intermédiaire d'une bascule synchrone sur le signal `Clock`.

```
1  #utilisation d'une LUT
2  .names n318 n324 n191
3  10 1
4  01 1
5
6  #utilisation d'une bascule
7  .latch  n191  g14  Reset Clock 2
```

Listing 4-1 – Description d'une LUT et d'une bascule au format BLIF

B.2-2 Placement et routage : VPR Toronto

L'outil T-VPACK, faisant parti de la suite logicielle VPR [56] de l'université de Toronto au Canada, permet le reformatage d'un fichier BLIF de manière à regrouper les LUT et les bascules générées par ABC en un ensemble de cellules logiques hiérarchiques. Ces cellules sont paramétrables en termes de nombre et de taille de LUT ainsi que de bascules. Dans l'exemple présenté par le listing 4-2, deux LUT sont utilisées. La première (`g14`), est configurée de manière à ce que la sortie passe par la bascule. Il s'agit en fait de la concaténation des deux ressources (LUT et bascule) de la description BLIF présentée précédemment (listing 4-1). En effet, la fonction logique XOR réalisée entre les entrées `n318` et `n324` sort directement synchronisés sur `g14`. La deuxième LUT spécifiée est une LUT à trois entrées (`n322 n389` et `n323`) et la fonction réalisée (`n324`) est combinatoire, celle-ci n'utilisant pas la bascule.

```
1  # utilisation d'une LUT et de son registre.
2  .clb g14
3  pinlist: n318 n324 open open g14 Clock
4  subblock: g14 0 1 open open 4 5
5
6  # utilisation d'une LUT seulement.
7  .clb n324
8  pinlist: n322 n389 n323 open n324 open
9  subblock: n324 0 1 2 open 4 open
```

Listing 4-2 – Description d'une LUT au format T-VPACK

L'outil VPR permet, d'une part, l'organisation et le placement des cellules logiques générées par T-VPACK. Le FPGA est représenté comme une matrice de cellules logiques où celles-ci sont placées par ligne et par colonne. Dans l'exemple donné par le listing 4-3, la cellule logique `g14` est placée colonne "1", ligne "3". D'autre part, VPR permet également le routage des signaux conformément à l'application et à l'architecture cibles. Dans l'exemple présenté (listing 4-4), le chemin spécifié part de la sortie "1" de la cellule logique se trouvant en position (1,3), puis est connectée à la piste "3" de la ressource de connexion `CHANY` en position (1,3) vers la piste "2" de la ressource de connexion `CHANX` en position (1,3) pour enfin se connecter à la broche d'entrée "0" en position (1,3).

```
1  Netlist file: add.net   Architecture file: k4-n1.xml
2  Array size: 5 x 5 logic blocks
3
4  #block name x  y  subblk   block number
5  #---------- -  -  ------   ------------
6  ...
7  g14        1  3  0  #56
8  ...
```

Listing 4-3 – Description du fichier de placement VPR

```
1  Array size: 5 x 5 logic blocks.
2  Routing:
3
4  ...
5  Net 0 (A[0])
6
7    SOURCE (1,3)  Pad: 1
8     OPIN (1,3)  Pad: 1
9     CHANY (1,3)  Track: 3
10    CHANX (1,3)  Track: 2
11    IPIN (1,3)  Pin: 0
12   SINK (1,3)  Class: 0
13 ...
```

Listing 4-4 – Description du fichier de routage VPR

B.2-3 Génération d'un flot de données de configuration Mozaïc :

Le flot de données de configuration généré par la plate-forme Mozaïc pour une architecture cible de type FPGA utilise l'ensemble des logiciels présentés précédemment (ABC, T-VPACK et VPR). Les fichiers issus de ces outils ne sont pas directement exploitables par Mozaïc, il est nécessaire de développer des outils d'interprétation et de conversion des fichiers de placement-routage produits par VPR.

Zaïp, conversion des fichiers de placement VPR : l'outil Zaïp (figure 4-8) est chargé de produire une description à la fois localisée et à la fois de la configuration des LUT. Cela implique que, pour chaque emplacement de la matrice, une configuration de la cellule logique concernée est spécifiée, incluant les bits de configuration de la LUT ainsi que le bit de configuration concernant l'activation de la sortie en mode synchrone ou combinatoire. Par exemple, une ligne de la forme suivante, `(1,3):10000000000110;` indique que la cellule logique placée colonne "1", ligne "3", active sa bascule de sortie (premier bit du mot de configuration à "1") et que la mémoire de LUT à quatre entrées dont seules deux sont utilisées pour l'implémentation de la fonction logique XOR (`0110`). Ce fichier de configuration est généré à partir de la synthèse issue de l'outils ABC pour la configuration et du fichier de placement issu de VPR.

Zaïr, conversion des fichiers de routage VPR : l'outil Zaïr permet la génération d'un fichier de configuration des DyRIBox à partir de la description de routage issue de VPR. Les ressources de commutation utilisées par VPR, présentées par la figure 4-9, sont au nombre de trois. Les CHANX sont chargés des communications de canaux horizontaux, les CHANY, chargés des communications de canaux verticaux et enfin les SB, chargées d'effectuer le passage d'une communication d'un canal horizontal vers un de canal vertical et inversement. Ces trois fonctionnalités sont très bien supportées par la ressource de connexion DyRIBox. La conversion des ressources de connexion de type VPR vers les unités d'interconnexion DyRIBox est réalisée par Zaïr. Cette conversion se fait de manière à ce que chaque piste entrante ou sortante corresponde à un port bidirectionnel de la DyRIBox, et que chaque port connecté à une ressource logique le soit par l'intermédiaire de ports unidirectionnels. La description

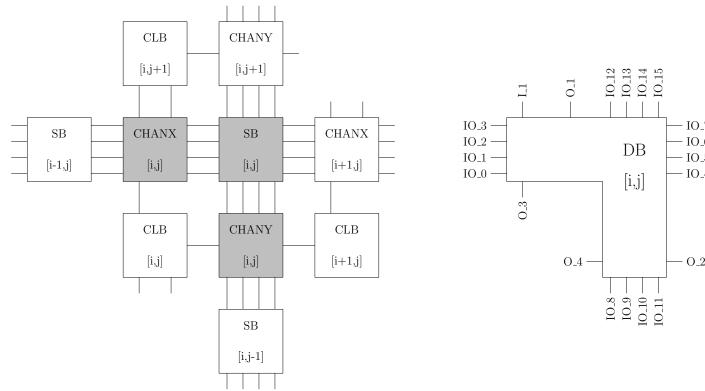


Figure 4-9 – **Conversion d'un routage type VPR vers un routage type DyRIBox**. *L'analyse du fichier de routage issue de VPR impose une conversion des données de configuration. VPR utilise trois ressources différentes de routage, CHANX, CHANY et switch-box pour le passage d'une piste horizontale à une piste verticale. Le principe de la DyRIBox permet de regrouper ces trois ressources.*

de l'exemple du chemin donné par le listing 4-4 dans le paragraphe de présentation de VPR, passe par une ressource de type CHANY, puis vers une ressource de CHANX, impliquant de fait le passage intermédiaire d'une SB. La conversion vers une configuration de type DyRIBox donne alors la configuration suivante : `(1,3):2<=3;`. Cette configuration indique que la sortie "2" de la DyRIBox chargée de la connexion du bloc logique situé colonne "1", ligne "3", est connectée à l'entrée "3".

Zaïb, générateur de données de configurations binaires : les deux fichiers issus des outils Zaïp et Zaïr sont ensuite lus par le générateur de données de configuration Zaïb. Zaïb est chargé de formater ces données en binaire et dans un ordre tel que le format produit soit compatible avec les ressources de reconfiguration générées par la plate-forme Mozaïc. Conformément au format du flot de données de configuration présenté à la section E.3-3 du chapitre II, les données de configuration des LUT sont placées au début de la trame par ordre chronologique de la description xMAML. Suivent ensuite les données de configuration des DyRIBox par ordre croissant des entrées/sorties par DyRIBox. Cette configuration est générée de manière flexible en fonction de la taille de la matrice de cellules logiques, de la taille du bus de configuration, du nombre d'entrées et de sorties des LUT, et de la taille du contexte d'une cellule logique.

Zaïg, génération automatique de code xMAML d'architecture FPGA : à partir des paramètres de l'architecture FPGA générés par VPR et à partir des caractéristiques du CLB, il est possible de procéder à la génération automatique du code xMAML permettant de modéliser une architecture FPGA synthétisable accompagnée des ressources de reconfiguration dynamique conforme au modèle implémenté par Mozaïc. Ces paramètres concernent le nom du modèle d'architecture synthétisable à générer, la taille de la matrice de cellules logiques en termes de lignes et de colonnes, les ports d'interface de la cellule logique de base utilisée pour le FPGA, et enfin le nombre de pistes de communications entre les DyRIBox.

B.3 Adéquation WCDMA-FPGA

Dans cette partie, nous étudions les ressources nécessaires à l'implémentation de chacune des fonctions du décodeur WCDMA sur le eFPGA que nous ciblons. La fonction qui nécessitera le plus de ressources permettra alors de déterminer la taille minimale de notre architecture. Ainsi, il sera possible de déterminer le nombre de domaines de reconfiguration dont nous aurons besoin pour maintenir les contraintes temporelles de l'application WCDMA. Cette étude est basée sur un modèle de décodeur WCDMA développé par Taofik Saïdi dans le cadre de ses travaux de recherche [57]. Nous avons retravaillé ce modèle de façon à pouvoir implémenter ses différentes fonctions dynamiquement.

B.3-1 Synthèse des fonctions du WCDMA sur eFPGA

Le décodeur WCDMA est composé de trois fonctions principales. L'ordonnancement des différentes fonctions est représenté sur la figure 4-10. Le cycle complet de décodage et de reconfiguration doit être effectué sur une période égale à la réception d'un *slot*. Les échantillons sont mémorisés en mémoire RAM au fur et à mesure de la réception du *slot* et seront traités à la réception du *slot* suivant.

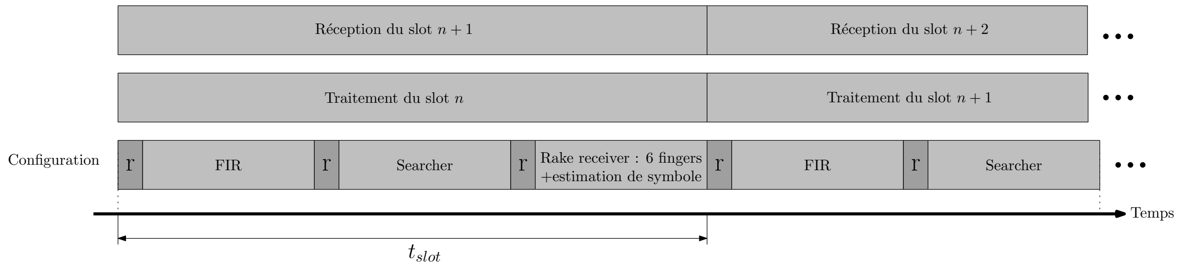


Figure 4-10 – **Ordonnancement des reconfigurations du décodeur WCDMA sur FPGA**. *L'implémentation d'un récepteur WCDMA sur un eFPGA nécessite, tel que nous l'avons développée, trois fonctions principales. Chacune de ces fonctions est implémentée l'une après l'autre à une période correspondant à la réception d'un* slot. *Pendant que les échantillons du* slot $n+1$ *sont en cours de réception, l'architecture traite les échantillons du* slot n

Les résultats issus de la synthèse des différentes fonctions sont reportés dans le tableau 4-1. Ces résultats montrent que l'implémentation nécessite 1117 CLB pour le filtre *FIR*, 1235 pour le *Searcher*, 245 par *finger* du *Rake Receiver* et 50 pour l'estimation de symbole faisant également partie de la fonction *Rake Receiver*. Par conséquent, au moins 1235 CLB sont nécessaires pour une implémentation reconfigurable dynamiquement dont le principe repose sur la reconfiguration successive des différentes fonctions. La structure générique du eFPGA que nous devrons générer sera alors constituée d'une matrice de 36×36 CLB.

B.3-2 Analyse des performances requises par l'application WCDMA

Le temps nécessaire au filtrage et au décodage des données est déterminé par la durée de l'acquisition des échantillons. Il est nécessaire d'acquérir 256 échantillons pour le filtrage et

Fonction	FIR	Searcher	Rake Receiver	
			finger	symbole
CLB	1117	1235	245	50
			Total : 1030	

Tableau 4-1 – Nombre de CLB nécessaires à l'implémentation du module WCDMA par la méthode reconfigurable dynamiquement.

le décodage. La fréquence d'échantillonnage est de $3,84\ MHz$, l'acquisition des échantillons (2×8 bits, partie réelle et partie imaginaire) se fait à un facteur de suréchantillonage de 4 ce qui implique l'acquisition de $4 \times 256 = 1024$ échantillons à $4 \times 3,84\ MHz$ soit 1024 échantillons de 2×8 bits à une fréquence de $15,36\ MHz$. Le temps total maximum pour le traitement d'un *slot* complet, caractérisé par trois reconfigurations successives des fonctions du décodeur WCDMA, doit donc être inférieur à $65,16\ \mu s$.

B.4 Description xMAML du eFPGA

De manière à procéder à la génération des ressources nécessaires à la mise en œuvre de la reconfiguration dynamique, Mozaïc a besoin d'une description précise des interconnexions ainsi qu'une spécification des mécanismes de reconfiguration utilisés par les blocs logiques. Cette description se fait par l'intermédiaire du langage de description xMAML. La figure 4-11 représente un schéma synoptique de l'architecture qui sera produite. Celle-ci est composée d'une matrice de CLB organisés en 36 lignes par 36 colonnes. Chaque CLB est connecté à ses quatre DyRIBox voisines, chacune avec trois entrées et une sortie.

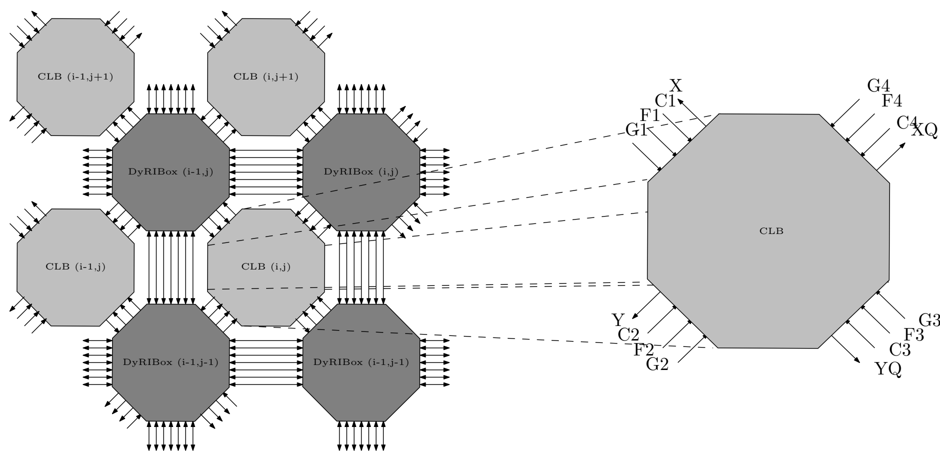


Figure 4-11 – **Schéma synoptique des connexions entre DyRIBox et CLB du eFPGA généré**. *L'architecture à générer est une matrice de CLB dont les connexions avec les DyRIBox voisines sont réparties uniformément sur les quatre côtés du CLB. Le CLB de base étant comparable aux CLB de la famille XC4000, trois ports d'entrée et un port de sortie sont connectés vers les DyRIBox voisines.*

Fonction	Taille de configuration en bits
Deux LUT à quatre entrées	32
LUT à trois entrées	8
Connexions d'entrée/sortie	5
Connexions internes	10
Ressource de calcul des retenues	8
Configuration fonction RAM	3
Total	66

Tableau 4-2 – Tailles des configurations pour un CLB de la famille XC4000S

B.4-1 Description des DyRIBox

Le motif de connexion des unités d'interconnexion du FPGA XC4000 est réalisé de telle manière que les différentes entrées et sorties des CLB soient uniformément connectées sur les quatre DyRIBox qui les entourent. La description xMAML donnée par le listing 4-5, va permettre de générer les DUCK adaptés à la taille de la configuration de la DyRIBox. La figure 4-11 présente les connexions entre DyRIBox et CLB. Chaque DyRIBox est connectée à ses quatre CLB voisins par l'intermédiaire de trois ports de sortie et un port d'entrée chacun. Les connexions totales d'une DyRIBox avec des CLB se résume donc à douze ports de sortie et quatre ports d'entrée. De plus, chaque DyRIBox est également connectée à ses quatre DyRIBox voisines par l'intermédiaire de sept ports bidirectionnels chacun, soit 28 ports bidirectionnels au total. Chaque `DBPort` est restreint en connexion vers quatre autres ports au maximum, alors que chaque sortie `PElementsPort` accepte jusqu'à sept connexions possibles. Cette restriction est spécifiée par l'`AdjacencyMatrix` des lignes 10 à 51.

B.4-2 Description des CLB

Conformément à la spécification du CLB XC4000 représenté sur la figure 4-6, la description xMAML donnée par le listing 4-6 est composée de huit entrées de données (G1, G2, G3, G4, F1, F2, F3, F4) et de quatre sorties de données (X, XQ, Y, YQ). À cela s'ajoutent quatre entrées nécessaires à l'utilisation du CLB en fonction mémoire RAM (C1, C2, C3, C4) ainsi que les ports d'entrée/sortie nécessaires au calcul et à la propagation des retenues (cin_up, cin_down, cout_up, cout_down). L'ensemble de ces entrées sont d'une largeur de un bit. La taille du flot de configuration nécessaire à un CLB est résumé dans le tableau 4-2. Tout d'abord, les trois LUT nécessitent 40 bits de configuration (deux LUT à quatre entrées et une LUT à trois entrées). Les différents multiplexeur de sélection des signaux d'entrée à connecter aux LUT et de sortie du CLB nécessitent en tout cinq bits. Les multiplexeurs de connexions internes nécessitent dix bits. Les ressources de calcul de retenue nécessitent elles huit bits et enfin la configuration du CLB en mémoire RAM nécessite trois bits. Les registres de configuration internes au CLB sont chaînés de façon à former un registre à décalage de sorte que 66 cycles d'horloges sont nécessaires à l'acheminement d'une nouvelle configuration.

```
1  <PEInterconnectDyRIBox name="DBox">
2      <ReconfigurationTime cycle="1"/>
3      <DBPorts>
4          <InOut number="28" bitwidth="1" />
5      </DBPorts>
6      <PElementsPorts>
7          <Inputs number="4" bitwidth="1"/>
8          <Outputs number="12" bitwidth="1" />
9      </PElementsPorts>
10         <AdjacencyMatrix>
11         <DInOut idx="0"  row="00000000000001000000110000000000"/>
12         <DInOut idx="1"  row="00000000000010000001001000000100"/>
13         <DInOut idx="2"  row="00000000000100000010000100001000"/>
14         <DInOut idx="3"  row="00000000001000000100000010000000"/>
15         <DInOut idx="4"  row="00000000010000001000000001000100"/>
16         <DInOut idx="5"  row="00000000100000010000000000101000"/>
17         <DInOut idx="6"  row="00000001000000100000000000010000"/>
18         <DInOut idx="7"  row="00000010000000100000000000010000"/>
19         <DInOut idx="8"  row="00000100000000010000000000101000"/>
20         <DInOut idx="9"  row="00001000000000001000000001000100"/>
21         <DInOut idx="10" row="00010000000000000100000010000000"/>
22         <DInOut idx="11" row="00100000000000000010000100000000"/>
23         <DInOut idx="12" row="01000000000000000001001000001000"/>
24         <DInOut idx="13" row="10000000000000000000110000000100"/>
25         <DInOut idx="14" row="00000011000000000000000000010001"/>
26         <DInOut idx="15" row="00000100100000000000000000100010"/>
27         <DInOut idx="16" row="00001000010000000000000001000000"/>
28         <DInOut idx="17" row="00010000001000000000000010000001"/>
29         <DInOut idx="18" row="00100000000100000000000100000010"/>
30         <DInOut idx="19" row="01000000000010000000001000000000"/>
31         <DInOut idx="20" row="10000000000001000000010000000000"/>
32         <DInOut idx="21" row="10000000000001000000010000000010"/>
33         <DInOut idx="22" row="01000000000010000000001000000000"/>
34         <DInOut idx="23" row="00100000000100000000000100000001"/>
35         <DInOut idx="24" row="00010000001000000000000010000010"/>
36         <DInOut idx="25" row="00001000010000000000000001000000"/>
37         <DInOut idx="26" row="00000100100000000000000000100001"/>
38         <DInOut idx="27" row="00000011000000000000000000010000"/>
39         <POutput idx="0" row="00000000000000111111000000000000"/>
40         <POutput idx="1" row="11111100000000000000000000000000"/>
41         <POutput idx="2" row="00000001111110000000000000000000"/>
42         <POutput idx="3" row="00000000000000000000011111100000"/>
43         <POutput idx="4" row="00000000000000111111000000000000"/>
44         <POutput idx="5" row="11111100000000000000000000000000"/>
45         <POutput idx="6" row="00000001111110000000000000000000"/>
46         <POutput idx="7" row="00000000000000000000011111100000"/>
47         <POutput idx="8" row="00000000000000111111000000000000"/>
48         <POutput idx="9" row="11111100000000000000000000000000"/>
49         <POutput idx="10" row="00000001111110000000000000000000"/>
50         <POutput idx="11" row="00000000000000000000011111100000"/>
51     </AdjacencyMatrix>
52 </PEInterconnectDyRIBox>
```

Listing 4-5 – Description xMAML d'une DyRIBox XC4000

```
1  <PEInterface name="CLB" >
2      <Reconfiguration cycle = "66"  bits = "66" preemption="no"/>
3      <IOPorts>
4        <Port name="G1" bitwidth="1" direction="in" type="data"/>
5        <Port name="G2" bitwidth="1" direction="in" type="data"/>
6        <Port name="G3" bitwidth="1" direction="in" type="data"/>
7        <Port name="G4" bitwidth="1" direction="in" type="data"/>
8        <Port name="F1" bitwidth="1" direction="in" type="data"/>
9        <Port name="F2" bitwidth="1" direction="in" type="data"/>
10       <Port name="F3" bitwidth="1" direction="in" type="data"/>
11       <Port name="F4" bitwidth="1" direction="in" type="data"/>
12       <Port name="X" bitwidth="1" direction="out" type="data"/>
13       <Port name="XQ" bitwidth="1" direction="out" type="data"/>
14       <Port name="Y" bitwidth="1" direction="out" type="data"/>
15       <Port name="YQ" bitwidth="1" direction="out" type="data"/>
16       <Port name="C1" bitwidth="1" direction="in" type="data"/>
17       <Port name="C2" bitwidth="1" direction="in" type="data"/>
18       <Port name="C3" bitwidth="1" direction="in" type="data"/>
19       <Port name="C4" bitwidth="1" direction="in" type="data"/>
20       <Port name="Cin_up" bitwidth="1" direction="in" type="data"/>
21       <Port name="Cin_down" bitwidth="1" direction="in" type="data"/>
22       <Port name="Cout_up" bitwidth="1" direction="out" type="data"/>
23       <Port name="Cout_up" bitwidth="1" direction="out" type="data"/>
24       <Port name="rst" bitwidth="1" direction="in" type="rst"/>
25       <Port name="clk" bitwidth="1" direction="in" type="clk"/>
26       <Port name="Config" bitwidth="1" direction="in" type="ScanIn"/>
27       <Port name="ConfigEn" bitwidth="1" direction="in" type="ScanEn"/>
28     </IOPorts>
29   </PEInterface>
```

Listing 4-6 – Description xMAML d'un CLB XC4000

B.5 Génération des mécanismes de reconfiguration dynamique pour eFPGA

À partir de la description xMAML présentée précédemment, Mozaïc procéde à la génération des ressources de reconfiguration dynamique. Les sections suivantes détaillent les calculs effectués automatiquement par Mozaïc pour la spécification des différentes ressources produites.

B.5-1 Estimation de la taille des données de configurations et des performances de reconfiguration dynamique

Dimensionnement des DUCK de CLB : chaque implémentation d'une des fonctions du décodeur WCDMA requiert une reconfiguration dynamique appliquée à l'ensemble des CLB et des DyRIBox. Chaque CLB nécessite 66 bits de configurations qui devront être placés dans les registres DUCK pendant les phases de calculs. Si l'on considère un bus de configuration d'une largeur de huit bits, alors 72 bits de configuration sont nécessaires au total par CLB incluant six bits "fantômes". Cela implique donc que pour l'ensemble de l'architecture du eFPGA, 88920 bits de configurations sont nécessaires à la configuration des 1235 CLB.

Dimensionnement des DUCK de DyRIBox : en ce qui concerne les unités d'interconnexion, chaque CLB est associé à une DyRIBox dont le schéma de connexion est limité à sept

Ressource	Bits de configuration	
	par ressource	Total
CLB	72	88920
DyRIBox	120	148200
Total	237120 bits	

Tableau 4-3 – Tailles des configurations pour les 1235 CLB et 1235 DyRIBox du eFPGA

connexions possibles par sortie. Les sorties sont au nombre de 40. La taille d'une configuration de DyRIBox de CLB est notée DB_{clb} et a pour valeur :

$$DB_{FPGA} = \sum_{n=1}^{40} \lceil log_2(7) \rceil = 120 \; bits \tag{4-1}$$

Par conséquent, la configuration de l'ensemble des DyRIBox de l'architecture (DBT_{FPGA}) a une taille de DBT_{FPGA} = 120×1235 = 148200 bits. Notons qu'un bus de configuration de huit bits permet de garder une architecture efficace et de ne pas avoir recours à l'implémentation de bits "fantômes". Par conséquent, pour l'ensemble de l'architecture FPGA, un contexte de configuration aura pour taille 237120 bits soit l'implémentation de 29640 registres DUCK de huit bits. Les différentes tailles de configuration sont résumées par le tableau 4-3.

Estimation de la consommation et de la surface de silicium des structures DUCK : la mise en œuvre des processus de reconfiguration selon le principe propre à la plate-forme Mozaïc implique l'utilisation de DUCK à reconfiguration séquentielle pour les configuration des CLB. Les différentes synthèses effectuées sur des échantillons de DUCK à reconfiguration séquentielle ont permis d'extraire deux équations d'exploration pour la surface de silicium et la consommation induite par l'utilisation de ces DUCK. Celles-ci sont fonctions d'un nombre de bits de configuration nécessaires au stockage d'un contexte. La surface de silicium supplémentaire totale pour l'ensemble des DUCK de CLB ($Sduck_{CLB}$) est alors de $Sduck_{CLB} = 174,3 + 88920 \times 27,6 = 2,45 \; mm^2$.

L'étude de la consommation se doit de rester au niveau de la consommation d'une ressource. En effet, les résultats de consommation intermédiaires ne peuvent pas s'ajouter pour une estimation globale. Cependant, il est possible d'estimer la consommation d'une ressource DUCK ($Cduck_{CLB}$) composée de neuf registres dédiée à la configuration d'un CLB, celle-ci est de $Cduck_{CLB} = 0,2852 + 72 \times 0,0155 = 1,40 \; mW$.

Les DUCK utilisés pour la reconfiguration dynamique des DyRIBox sont pour leur part à reconfiguration parallèle. Les DUCK utiles à la sauvegarde de 148200 bits de configuration pour la mise en œuvre de la reconfiguration dynamique nécessitent alors une surface de $Sduck_{DB} = 562,9 + 148200 \times 49,8 = 7,38 \; mm^2$.

De même que pour les DUCK de CLB, nous nous interresserons uniquement à la consommation d'un DUCK de DyRIBox ($Cduck_{DB}$) estimée à $Cduck_{DB} = 1,16 + 120 \times 0,0246 = 4,11 \; mW$.

B.5-2 ESTIMATION DU NOMBRE DE DOMAINES DE RECONFIGURATION

Le nombre de domaines de reconfiguration (N_D) est déterminé par le temps de propagation d'un contexte ($Prop_t$) et le temps disponible entre deux phases de reconfiguration. Comme nous l'avons calculé précédemment, chaque fonction est implémentée pendant $21,5\ \mu s$ soit autant de temps à disposition pour la propagation complète d'un contexte. Cependant, la mémoire de configuration embarquée implémentée pour la sauvegarde des 29640×8 bits de contexte permet une vitesse de lecture de 300 MHz [1]. Le temps de propagation d'un contexte complet est alors réalisé en :

$$Prop_t = \frac{29640}{300E^6} = 98,8\ \mu s \tag{4-2}$$

Par conséquent, le nombre de domaines (N_D) nécessaire au maintient des contraintes temporelles imposées par l'application WCDMA est de :

$$N_D = \left\lceil \frac{98,8E^{-6}}{21,5E^{-6}} \right\rceil = 5 \tag{4-3}$$

Un diagramme de Gantt (figure 4-12) permet de résumer les différentes phases de reconfiguration et de calcul des trois fonctions de l'application WCDMA. Cinq domaines sont nécessaires à l'implémentation successive des trois fonctions. Pour cela, les phases de propagations ($Conf_s$,$Conf_r$ et $Conf_f$ respectivement les configurations de la fonction *Searcher*, *Rake Receiver* et *FIR*) se font en parallèle des phases de traitement de données. Les phases de propagation ($t_{propagation}$) sont dimensionnées de manière à être plus courtes que les phases de traitement (t_{calcul}). L'ensemble des ressources de l'architecture sont nécessaires pour l'implémentation des fonctions filtre *FIR* et *Searcher*. En revanche, l'implémentation des *fingers* et de l'estimation de symbole sont implémentées en parallèle chacune sur un domaine. La figure 4-13 permet une vue schématisée de l'ensemble des différentes configurations sur les cinq domaines du eFPGA.

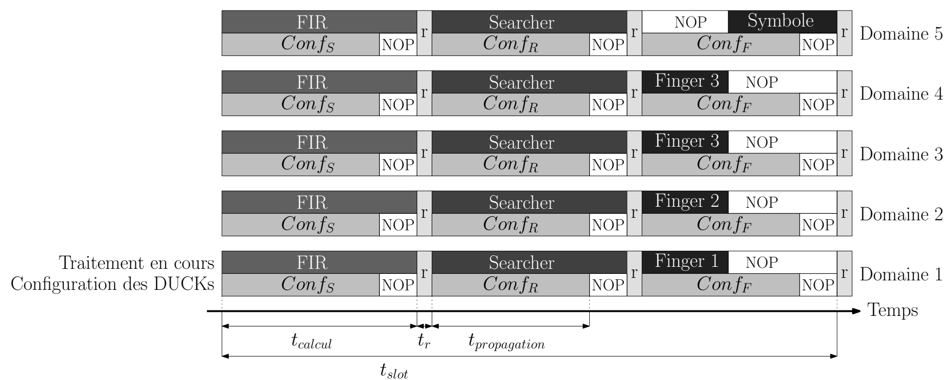


Figure 4-12 – **Diagramme de Gantt des processus de reconfiguration**. *Les phases de propagation de configuration ($Conf_s$,$Conf_r$ et $Conf_f$) sont exécutées en parallèle aux phases de traitement des données (*Searcher, Fingers n *et FIR).*

1. Données fournies par STMicroelectronics

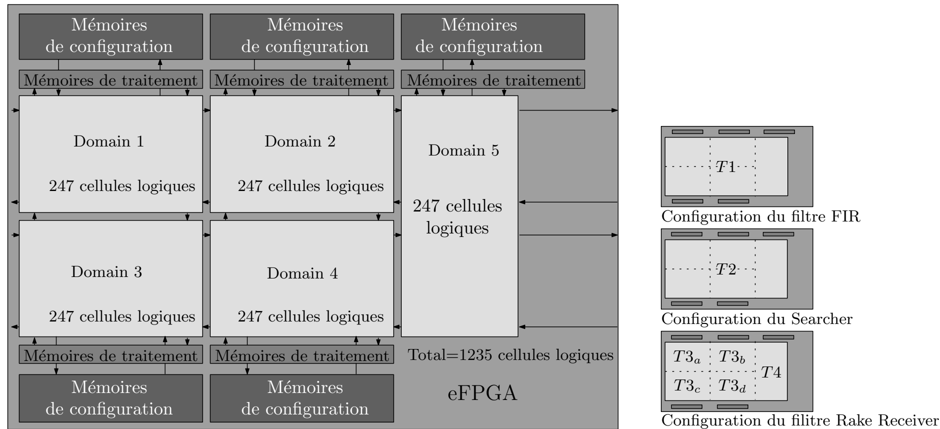


Figure 4-13 – **Schéma synoptique de l'implémentation des domaines pour WCDMA sur eFPGA**. *Ce schéma synoptique permet de visualiser la manière dont sont réparties les différentes tâches selon la configuration en cours de traitement.*

B.5-3 Impact de la reconfiguration dynamique sur les caractéristiques de l'architecture

Nous venons de montrer que l'introduction du concept DUCK nécessaire à la mise en œuvre de la reconfiguration dynamique sur une architecture augmente la consommation et la surface de silicium. Cependant, l'application de la reconfiguration dynamique permet une économie au niveau de la quantité de CLB nécessaires au traitement des données par rapport à une implémentation statique. C'est pourquoi, il est intéressant de comparer la surface nécessaire à l'implémentation dynamique de l'application du décodeur WCDMA avec la surface nécessaire à une implémentation statique.

Surface de silicium et consommation d'une implémentation statique : la synthèse d'un CLB par Synopsys, dans une technologie CMOS 130 nm, nous donne une surface de 1539 μm^2. La synthèse d'une *switch-box* montre que la surface nécessaire à son implémentation est de 11653 μm^2. Si l'on additionne l'ensemble des ressources de CLB nécessaires à chaque fonction de WCDMA, l'architecture statique du décodeur WCDMA nécessiterait 3382 CLB. Par conséquent, une estimation rapide de la surface totale du FPGA, pour une implémentation statique d'un décodeur WCDMA, indique une surface de $44,61$ mm^2 pour l'ensemble des 3382 CLB et DyRIBox.

Surface de silicium d'une implémentation dynamique : nous avons estimé précédemment la surface de silicium et la consommation induite par l'introduction des DUCK. Celles-ci étaient de $11,76$ mm^2 pour l'ensemble de l'architecture. À cette surface, ajoutons la surface des 1285 CLB et DyRIBox d'une valeur de $16,29$ mm^2 soit un total pour l'architecture reconfigurable de $28,05$ mm^2.

En conclusion : bien que l'introduction du concept DUCK a un impact sur la surface et la consommation, celui-ci est très largement minimisé grâce à l'économie des unités de traitement

Implémentation	Surface en mm^2
Statique (3382 CLB)	44,61
Dynamique (1235 CLB)	28,05
Différence	16,56 (-37,12%)

Tableau 4-4 – Résumé des différentes implémentations

et de routage effectuée par la reconfiguration dynamique. Les différents calculs montrent sur le tableau 4-4 qu'une économie de 37% est envisageable sur la surface de silicium.

C Implémentation d'un décodeur WCDMA sur DART

Dans cette section, nous présentons la conception d'une architecture de type DART et l'implémentation d'une application sur celui-ci. Pour cela, nous allons nous appuyer, d'une part sur les outils associés à DART spécifiques à la compilation d'algorithme sur cette architecture, et d'autre part sur Mozaïc afin de générer automatiquement les ressources de reconfiguration dynamique ainsi qu'à l'exploration de sa mise en œuvre. La figure 4-14 présente le flot d'exploration Mozaïc adapté à DART par l'utilisation des outils de compilation développés pour cette architecture [12] et [58]. Dans cette section, nous allons présenter les outils spécifiques de DART ainsi que l'intégration de Mozaïc dans le flot de conception. Ensuite, nous présenterons l'implémentation du décodeur WCDMA sur DART, exploitant les principes de reconfiguration dynamique propres à Mozaïc. Enfin, nous explorerons les caractéristiques de cette implémentation en termes de reconfiguration dynamique.

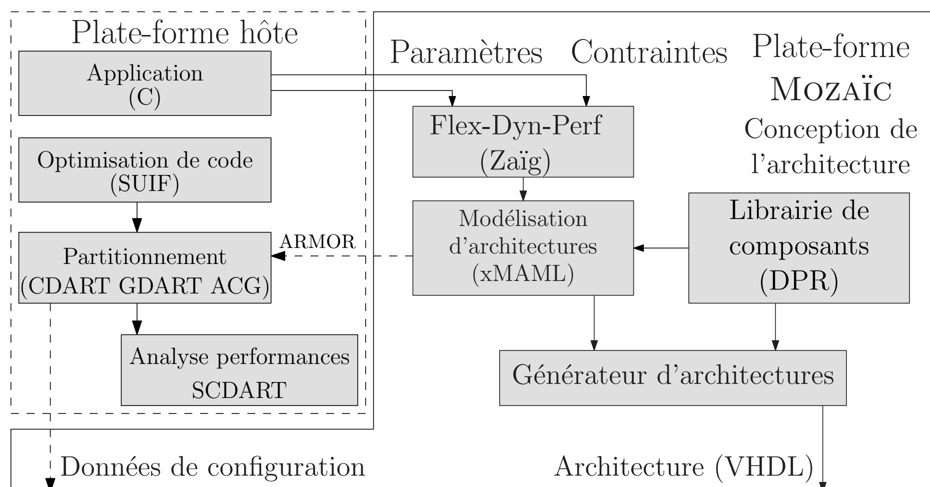


Figure 4-14 – **Plate-forme de développement Mozaïc et interaction avec le flot de développement DART**. *Le flot de conception* Mozaïc *présenté dans cette figure s'intègre à un environnement dédié au développement d'applications sur DART. Tout d'abord SUIF est un* front-end *de compilation utile à la génération des configurations matérielles par gDART et logicielles par cDART en fonction de la description ARMOR de l'architecture. Enfin SCDART permet l'analyse des performances d'exécution et de consommation.*

C.1 Présentation de DART et des outils associés

C.1-1 DART : processeur reconfigurable dynamiquement

Le processeur reconfigurable dynamiquement DART [59, 53] a été développé en 2003 pour le domaine des télécommunications mobiles. La reconfiguration dynamique mise en œuvre concerne les chemins de données, les fonctionnalités des unités de traitement, ainsi que la taille des données traitées.

Architecture : l'architecture DART représentée sur la figure 4-15, est une architecture à reconfiguration dynamique conçue pour traiter les applications de télécommunication mobiles de troisième génération. L'architecture est composée d'un contrôleur de tâches, de ressources de mémorisation et d'unités de traitement appelées *Clusters*. Le contrôleur de tâches est chargé d'assigner aux *Clusters* les différents traitements à exécuter. Une fois configuré, chaque *Cluster* travaille de manière autonome et gère ses propres accès à la mémoire de données. La représentation d'un cluster sur la figure 4-16 montre un cœur de traitement dédié et six DPR (*DataPath Reconfigurable*) opérant sur des données de largeur allant de 8 à 32 bits. Les DPR représentés sur la figure figure 4-17, constituent le dernier niveau de la hiérarchie de DART. Ces DPR sont constitués d'unités fonctionnelles interconnectées suivant un motif totalement flexible. Chaque DPR intègre quatre unités fonctionnelles, deux multiplieurs/additionneurs (UFl et UF3) et deux UAL (UF2 et UF4). Toutes ces unités fonctionnelles sont dynamiquement reconfigurables et supportent les traitements SWP et SIMD (*Single Instruction Multiple Data*). Ces unités permettent par ailleurs la réalisation de décalages en parallèle avec les traitements arithmétiques (un en sortie de chaque multiplieur ainsi qu'en entrée et en sortie des UAL). Enfin, notons que par soucis d'accroissement de la dynamique de calcul, les UAL travaillent avec huit bits de garde.

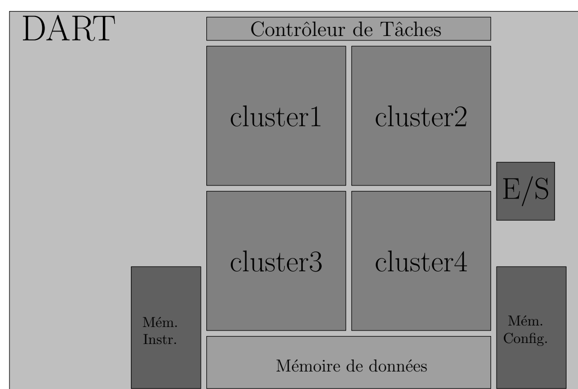


Figure 4-15 – **Architecture de DART**. *L'architecture DART est composée d'un contrôleur de tâches chargé d'assigner aux* clusters *les différents traitements devant être exécutés. Une fois configuré, chaque* cluster *travaille alors de manière autonome et gère ses propres accès à la mémoire de données.*

Reconfiguration dynamique : le contrôleur de reconfiguration des DPR de DART est réalisé sous la forme de flots d'instructions de configuration. Sa principale tâche est de sé-

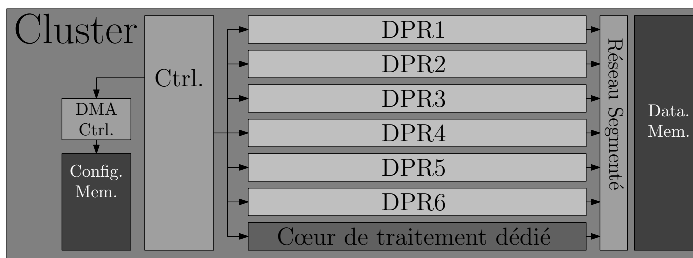


Figure 4-16 – **Architecture d'un *cluster* de DART**. *Un* cluster *se compose d'un cœur de traitement dédié, de chemin de données reconfigurable (DPR) pouvant être interconnectés via un réseau segmenté, et d'un contrôleur gérant ces unités de traitement. Les DPR et le cœur de traitement manipulent des données stockées dans une mémoire partagée.*

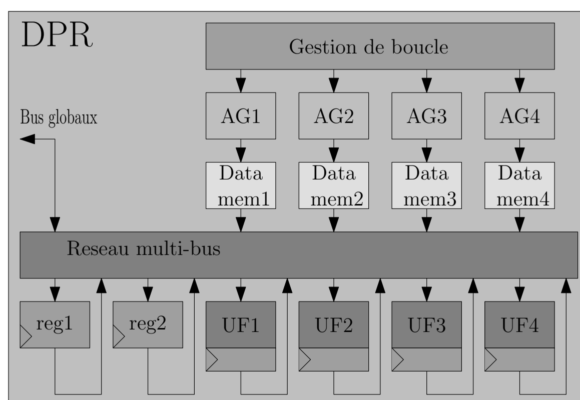


Figure 4-17 – **Architecture d'un *DataPath Reconfigurable* de DART**. *Un DPR est constitué d'unités fonctionnelles interconnectées suivant un motif totalement flexible de type multi-bus. Elles travaillent sur des données stockées dans les mémoires locales auxquelles sont associées des générateurs d'adresses (AG).*

quencer ces instructions de configuration. Son architecture est donc comparable à celle de tout contrôleur de processeur programmable, à ceci près qu'il n'a pas à gérer des mécanismes complexes d'interruptions et qu'il ne séquence pas des instructions (au sens instructions microprocesseur), mais des configurations du matériel. Ainsi, il n'y a pas lieu de systématiser de coûteux accès à la mémoire d'instructions et des décodages d'instructions non moins coûteux. En effet, ces accès n'interviendront que lors des reconfigurations et seront donc très occasionnels. Des mécanismes de mise en attente ont été mis en place afin de minimiser le nombre de lectures et de décodages d'instructions. Ceci permet de réduire considérablement la consommation d'énergie.

La reconfiguration dynamique sur DART peut être de nature logicielle ou matérielle. Le mode de reconfiguration logicielle a été conçu afin de répondre aux besoins inhérents aux zones de calcul irrégulières, où les motifs de calculs se succèdent très rapidement, sans ordre particulier et de manière non répétitive. La principale caractéristique de ce mode de reconfiguration est d'autoriser une reconfiguration des DPR en un cycle. Pour cela, la flexibilité de ces derniers a été limitée en adoptant un motif de calcul de type *Read-Modify-Write*. Dans ce cas, le modèle de calcul est donc comparable à celui des processeurs VLIW conventionnels. À chaque cycle

les données sont lues, traitées, puis les résultats sont rangés en mémoire. Ainsi, ce mode de reconfiguration est qualifié de reconfiguration logicielle. Il n'est ici en aucun cas possible de chaîner des opérateurs ou de réaliser des traitements SWP. Les cibles de la reconfiguration sont, dans ce mode de reconfiguration, en nombre limité puisqu'il s'agit de préciser quelles mémoires sont utilisées, et quelles sont les opérations à réaliser. Pour limiter plus encore le nombre de bits nécessaires à cette reconfiguration, chaque unité doit toujours écrire son résultat dans la même mémoire. Les reconfigurations logicielles sont spécifiées par le biais d'instructions de cinquante deux bits, issues du contrôleur de *Clusters*.

Outre la reconfiguration logicielle, adaptée aux traitements irréguliers, un second mode de reconfiguration a été défini afin de répondre aux besoins des traitements réguliers, tels que les cœurs de boucles, dans lesquels un même motif de calcul est utilisé pendant de longues périodes de temps. Ce mode de reconfiguration est qualifié de matériel. Il s'agit ici d'assurer une totale flexibilité au sein du DPR afin d'autoriser l'optimisation du chemin de données en fonction du motif de calcul. Cette reconfiguration matérielle consiste à spécifier une configuration par le biais d'un flot d'instructions, puis d'adopter un modèle de calcul de type *dataflow* dans lequel la structure du chemin de données est figée. En contrepartie du fort potentiel d'optimisation de ce mode de reconfiguration, le passage d'un motif de calcul à un autre nécessite une phase de reconfiguration dont la durée varie typiquement de trois à dix-neuf cycles d'horloge. Le contrôleur de *Clusters* distribue une instruction par cycle. Le nombre de cycle est fonction de la régularité de la configuration comme par exemple de l'efficacité du SCMD (*Single Configuration Multiple Data*) [1] et du nombre de DPR utilisés. Une fois cette configuration spécifiée, le contrôleur du *Cluster* est donc libéré du contrôle des DPR et n'a plus à accéder à la mémoire d'instructions ni de décoder de nouvelles instructions.

C.1-2 OUTILS DÉDIÉS À LA COMPILATION SUR DART

Les outils qui accompagnent la conception d'une architecture DART sont un *front-end* de compilation basé sur SUIF (*Stanford University Intermediate Format*), gDART, cDART, ACG et SCDART [58]. Tout d'abord, SUIF permet de modifier le code C d'une application afin de faciliter les manipulations effectuées par les outils suivants de la chaîne de conception. gDART est chargé de récupérer le code produit par SUIF et d'en extraire les traitements réguliers afin d'augmenter l'efficacité d'une implémentation matérielle. À l'inverse, cDART permet l'extraction de code irrégulier plus adapté à une implémentation logicielle. Selon la nature de la reconfiguration déterminée, ACG est chargé de produire les configurations de connexions de manière à assurer les communications entre les mémoires et les unités fonctionnelles. Enfin, SCDART est capable de fournir des informations sur la consommation d'énergie de DART en fonction de l'application et des méthodes de configuration qui seront choisies.

DÉROULAGE DE BOUCLE ET OPTIMISATION DE CODE : à partir de la description d'une application à haut niveau tel que C, une représentation intermédiaire sous forme de graphe

1. La reconfiguration SCMD permet de reconfigurer plusieurs DPR en parallèle avec une seule et même instruction partagée.

de données et de contrôle (CDFG) peut être obtenue grâce à l'utilisation de SUIF [60]. La représentation CDFG est générée après optimisation par passes successives de déroulage de boucles et l'extraction de code régulier.

Lors de l'analyse du code C de l'application, il est possible de déterminer les parties critiques en termes de temps d'exécution. Cette étape donne lieu à la génération d'un code C identique où les parties concernées sont désignées par l'ajout d'un commentaire adapté. Lors d'une seconde passe, grâce aux annotations, les traitements réguliers peuvent être extraits et transmis à gDART. Dans le cadre de l'utilisation de SUIF par DART, seuls les cœurs de boucles sont considérés comme des zones de code régulier. Lorsqu'un cœur de boucle est trouvé, celui-ci est recopié dans un nouveau fichier et remplacé dans le CDFG par une succession de lectures et d'écritures des mémoires utilisées par cette boucle. L'extraction d'une boucle donne lieu à la génération de deux fichiers. Le premier fichier qui décrit le motif de calcul de la boucle est nécessaire au module gDART pour une reconfiguration matérielle. Le second fichier contient les informations relatives au contrôle de la boucle ainsi qu'une description de la gestion et de la manipulation des données permettant la synchronisation des configurations des DPR et des générateurs d'adresses.

Le déroulage de boucles est nécessaire à l'exploitation du parallélisme de tâches intrinsèque à la plupart des boucles extraites précédemment. Par exemple, une boucle *for* qui nécessite à priori 128 itérations peut-être modifiée de manière à procéder à un certain nombre de traitements en parallèles, réduisant alors le nombre d'itérations.

Cette transformation de code est réalisée en deux étapes. Tout d'abord, les instructions contenues dans la boucle *for* sont dupliquées puis les indices sont modifiés en conséquence à condition que le nombre d'itérations soit connu dès la compilation.

Génération des configurations matérielles, partitionnement, ordonnancement (gDART) : gDART est chargé de déterminer la structure du chemin de données permettant d'implémenter au mieux le graphe flot de données de chaque cœu de boucle issu de SUIF puis de transformer cette structure en une configuration matérielle. Cependant, dans un souci d'optimisation, le plus souvent en terme de consommation, plusieurs étapes préliminaires sont initiées par gDART avant la génération du code binaire de configuration. La première consiste en une réduction de boucle qui se justifie lors de phases de traitement où il est nécessaire de maintenir des résultats intermédiaires sur plus d'un cycle. Dans ce cas de figure, certaines techniques, telle que la permutation des opérations, permettent de réduire la latence tout en respectant les règles d'associativité et de commutativité des opérations arithmétiques.

Un autre exemple d'implémentation inefficace peut se produire dans le cas de boucles imbriquées. Plus particulièrement, lorsque l'amorçage d'un pipeline d'exécution est relativement long et que le nombre d'itérations de la boucle de plus bas niveau est faible, les avantages apportés par une implémentation pipeline peuvent alors être nuls voire contre productifs. Afin de remédier à ces problèmes, il est nécessaire de réduire ces temps d'amorçage. Pour cela, gDART procède à une parallélisation des séquences d'opérations du début de pipeline.

Au niveau de la gestion des mémoires, deux hiérarchies sont disponibles. Les mémoires locales contenues dans les DPR et la mémoire de données au niveau *cluster*. La politique de gestion de l'allocation mémoire est fonction de la durée de vie des variables. Si une variable est censée

être utilisée plus longtemps que la durée de traitement d'un contexte, alors celle-ci sera sauvée en mémoire de *cluster*. Dans le cas contraire, elle pourra être sauvée dans la mémoire locale du DPR.

Compilation (CALIFE associé à ARMOR, cDART et ACG) : après avoir présenté les outils dédiés à la génération des configurations matérielles, étudions les outils dédiés à la génération des configurations logicielles. Ces outils font appel à des notions traditionnelles de compilation sur processeurs et sont issus de l'environnement de compilation reciblable CALIFE [50].

CALIFE permet la transformation du code source de l'application de manière à adapter celui-ci aux spécificités de l'architecture cible. Pour cela, une description du processeur est nécessaire (ARMOR présenté au chapitre III section A.3) ainsi qu'une bibliothèque d'algorithmes de transformation et une infrastructure transformant la description du processeur en paramètres de transformation de code. Cette bibliothèque est constituée de modules tels que la sélection de code, l'allocation de ressources, l'ordonnancement et l'optimisation de code.

cDART a pour rôle de générer les configurations logicielles des DPR. La configuration logicielle est considérée comme étant un mode de fonctionnement dégradé de l'architecture où seule la fonctionnalité des DPR et les sources des opérandes sont à spécifier lors de la reconfiguration.

ACG est chargé de générer les instructions de génération d'adresses nécessaires aux transferts des données entre la mémoire *cluster* et les mémoires des DPR ainsi que de générer les programmes devant être exécutés sur les générateurs d'adresses pendant les phases de reconfiguration matérielle.

Analyse des performances (SCDART) : SCDART vient compléter la suite logicielle dédiée à DART. Selon la ressource considérée, cet outil est capable d'estimer rapidement la consommation. Cette estimation se fait sur la base d'une description en SystemC, ce qui permet de combiner une modélisation précise de l'architecture et une gestion abstraite du déroulement de l'exécution. La méthode d'estimation est différente selon la ressource considérée (opérateur, interconnexion, registre, mémoire) et permet ainsi une estimation plus précise.

C.2 Adéquation WCDMA-DART

Cette partie présente les résultats d'implémentation des fonctions du décodeur WCDMA sur DART. Chacune de ces fonctions est implémentée indépendamment l'une de l'autre et travaille sur des données stockées en mémoire. Ces résultats d'implémentation sont issus des travaux de thèse de Raphaël David [12]. Comme le montre la figure 4-18, l'implémentation d'un décodeur WCDMA sur DART n'implémente pas de fonction *searcher* nécessaire à l'estimation des trajets. Les fonctions implémentées, filtrage, décodage des données, estimation de canal, synchronisation et combinaison des multi-trajets, sont ordonnancées selon le schéma synoptique de la figure 4-19. Chacune de ces fonctions est implémentée indépendamment l'une de l'autre. Les échanges de données sont effectués par l'intermédiaire des mémoires locales. Le passage d'une fonction à une autre est géré par les ressources de reconfiguration généré par Mozaïc.

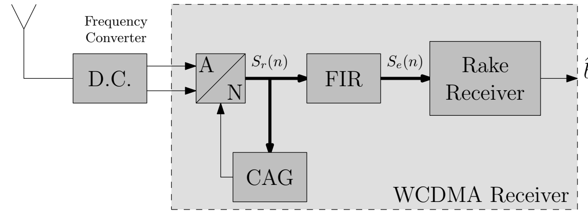


Figure 4-18 – **Schéma synoptique d'une implémentation de décodeur WCDMA sur DART**. *La fonction* searcher *permettant une implémentation optimisée des fonctions* finger *du* Rake Receiver *n'est pas implémenté sur DART. La fonction* Rake Receiver *implémente les fonctions de décodage des données, d'estimation de canal, de synchronisation et de combinaison des multi-trajets.*

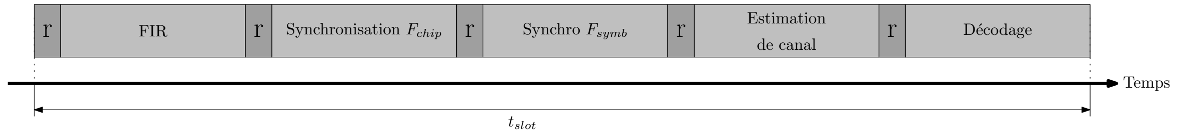


Figure 4-19 – **Ordonnancement des fonctions du décodeur WCDMA sur DART**. *Le décodeur WCDMA implémente successivement les tâches présentées ici dans leur ordre de reconfiguration. Sur la durée d'un* slot, *cinq tâches se succèdent après une phase de reconfiguration (r).*

C.2-1 Implémentation du filtre FIR

Le comportement algorithmique du filtre FIR est décrit par l'équation :

$$Y(n) = \sum_{i=0}^{N-1} X(n+i) \times H(N-i) \forall n \in [0, Nech_{slot}] \tag{4-4}$$

L'implémentation de ce filtre exploite le parallélisme de tâche et permet un traitement en parallèle sur six échantillons par *cluster* de DART. Les six équations de traitement sont alors :

$$\forall p \in \left[0, \frac{Nech_{slot}}{6}\right], \begin{cases} Y(6p) = \sum_{i=0}^{N-1} X(6p+i) \times H(N-i) \\ Y(6p-1) = \sum_{i=0}^{N-1} X(6p-1+i) \times H(N-i) \\ Y(6p-2) = \sum_{i=0}^{N-1} X(6p-2+i) \times H(N-i) \\ Y(6p-3) = \sum_{i=0}^{N-1} X(6p-3+i) \times H(N-i) \\ Y(6p-4) = \sum_{i=0}^{N-1} X(6p-4+i) \times H(N-i) \\ Y(6p-5) = \sum_{i=0}^{N-1} X(6p-5+i) \times H(N-i) \end{cases} \tag{4-5}$$

Six filtres sont implémentés sur les six DPR à raison de un chacun. L'avantage de cette

implémentation est que chaque filtre travaille sur les mêmes coefficients et sur des données retardées de zéro à cinq échantillons. La figure 4-20 montre l'implémentation de ces algorithmes sur les six DPR. Les coefficients sont propagés vers les unités fonctionnelles à travers les réseaux multi-bus. La propagation des retards est implémentée par l'intermédiaire des registres connectés de façon a réaliser un registre à décalage.

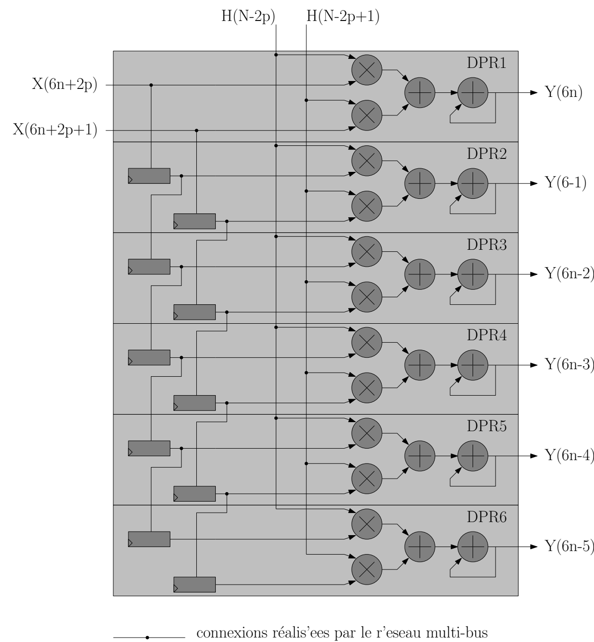


Figure 4-20 – **Exploitation du parallélisme de tâche pour l'implémentation du filtre FIR**. *Les six DPR se partagent les mêmes coefficients à traiter sur six échantillons différents. Les registres sont utilisés pour l'implémentation d'une chaîne de retard sous la forme d'un registre à décalage.*

C.2-2 Implémentation du décodage des données et combinaison des multi-trajets

Chaque *finger* du *Rake Receiver* est chargé de retrouver, dans les échantillons filtrés et synchronisés, l'information émise par l'émetteur. L'algorithme effectué est représenté figure 4-21. Le produit des codes OVSF et de Kasami sont supposés avoir été réalisés au préalable et stockés en mémoire.

Le codage des données sur huit bits permet d'exploiter le parallélisme des données pour une implémentation sur DART. De plus, le remplacement d'une soustraction par une addition de la même valeur inversée permet l'application du décodage complexe des données en mode SWP du a une parfaite symétrie entre les voie réelles et imaginaires.

La figure 4-22 représente l'implémentation réalisée. Le parallélisme de tâche implicite du *Rake Receiver* peut être implémenté facilement sur DART en attribuant un *finger* par DPR.

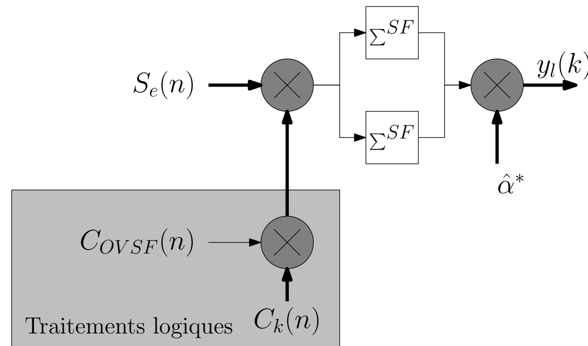


Figure 4-21 – **Schéma synoptique du décodage des données d'un terminal mobile**. *En partant du principe que le produit des codes OVSF et de Kasami a été réalisé au préalable, celui-ci est multiplié et accumulé sur SF cycles. Huit multiplications complexes sont nécessaires.*

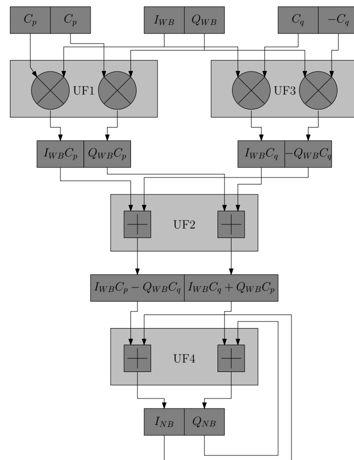


Figure 4-22 – **Schéma synoptique de l'implémentation d'un *finger* sur un DPR**. *La symétrie des calculs opérés entre les données réelles et les données imaginaires permet l'implémentation en mode SWP d'un finger sur chaque DPR.*

C.2-3 Synchronisation

Le décodage des données issues des différents trajets nécessite une synchronisation précise entre les codes générés au sein du récepteur et le code contenu dans le signal transmis pour les opérations de corrélation. Deux étapes sont effectuées. Tout d'abord, une estimation du retard associé à chaque trajet est réalisée avec une précision de $\pm\frac{Tc}{2}$. Cela est réalisé par la recherche séquentielle des maximums dans la fonction d'autocorrélation entre les données reçues et le code généré en interne sur une fenêtre de largeur Tc. Le décodage des données par les *fingers* du *Rake Receiver* ne se fera alors seulement que pour les six trajets dont les puissances sont les plus élevées. Cette étape de synchronisation est divisée en deux phases de calculs selon la fréquence des signaux manipulés. La première phase calcule la corrélation entre le signal d'entrée et les codes générés en interne à la fréquence *chip* ($3,84\ MHz$). La seconde phase permet l'extraction des bits de commande et l'isolation de la composante basse fréquence du module. Cette fois les données sont cadencées à la fréquence symbole

($F_{symbole} = 15\ kHz$). Ces deux phases traitent des données de huit bits.

Traitements à la fréquence *chip* : la configuration qui permet l'execution de cette phase est identique à une configuration de décodage des données présentée au paragraphe C.2-2. Par conséquent, l'implémentation de ce traitement est réalisé selon les mêmes configurations.

Traitements à la fréquence symbole : La difficulté d'implémentation de ce traitement tient dans le degré de parallélisme de données qui évolue entre les étapes du graphe flot de données représenté figure 4-23. Ce graphe flot de données distingue la multiplication par les bits de référence et la mise au carré du calcul de module, qui peuvent être implémentées en mode SWP, de la fin du calcul modulo et du filtre IIR (*Infinite Impulse Response*) dont les parallélismes de données sont insuffisants pour exploiter les capacités SWP de DART. Une phase de conversion de données est donc nécessaire et réalisée par deux masquages parallèles. Ces dix opérations peuvent être implémentés sur deux DPR. Les six DPR peuvent donc traiter trois voies d'un *finger* de manière concurrente réduisant les accès mémoire. En effet, les données de référence et les coefficients sont partagées entre les différentes voies du filtre IIR du *finger*. Les données en entrée des trois voies de synchronisation sont ensuite propagées par une chaîne de retard, selon le même principe que pour l'implémentation du filtre FIR.

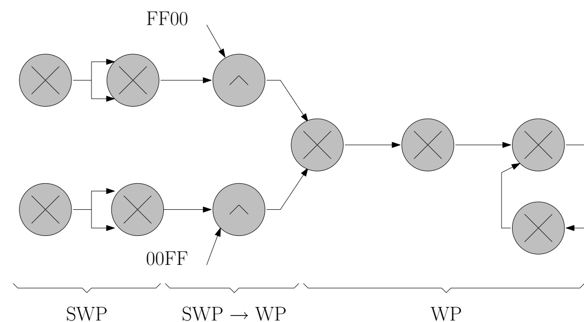


Figure 4-23 – **Graphe flot de données du traitement de la synchronisation à la fréquence symbole**. *La multiplication des signaux d'entrée par le code de référence, la mise au carré de ce produit sont traités en mode SWP. En revanche, la fin du calcul du modulo et le filtrage IIR sont eux réalisés au niveau mot, WP (*Word Processing*). La conversion SWP vers WP nécessite l'utilisation de deux opérateurs.*

C.2-4 Estimation de canal

L'amplitude complexe du canal est estimée, pour chaque trajet, à chaque nouveau *slot*. Cette estimation consiste à corréler les bits pilotes reçus avec une séquence déterministe reconstituée au niveau du récepteur. Le traitement présenté figure 4-24 commence par une multiplication complexe avec le code de Kasami puis avec le code OVSF et accumule le résultat sur 256 échantillons. Ces opérations sont implémentées en parallèle en exploitant les capacités SWP de DART. Afin d'estimer l'amplitude complexe du canal, les résultats calculés préalablement sont réutilisés. La corrélation du signal s_4 avec les bits de référence , puis l'échange des voies

réelle et imaginaire pour l'estimation de la valeur du conjugué de l'amplitude complexe du canal permet d'éliminer les données de contrôle encore présentes. La valeur estimée est alors moyennée avec la valeur estimée du *slot* précédent, par le biais d'un filtre FIR composé de deux cellules dont les coefficients sont à 0,5. La mise en œuvre de l'implémentation de six *fingers* nécessite six DPR configurés de manière identique et permettre donc l'utilisation du SCMD.

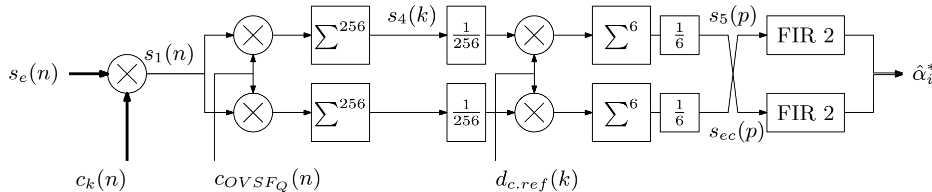


Figure 4-24 – **Schéma synoptique du traitement d'estimation de canal**. *La première partie du traitement corrèle le signal d'entrée avec le code OVSF et le code de Kasami. Les données de contrôle présentes au sein du signal sont ensuite éliminiées par la corrélation de ce signal avec les bits de référence. Afin d'estimer le conjugué de l'amplitude complexe du canal, ses parties réelle et imaginaire sont échangées avant d'attaquer un filtre moyenneur.*

C.2-5 Synthèse

Dans cette section, nous avons présenté l'implémentation des différents algorithmes nécessaires à un décodeur WCDMA. Le cycle de traitement s'effectue sur l'ensemble des données reçue à une période *slot*. Grâce à l'utilisation des méthodes de reconfiguration mis en œuvre par MOZAÏC, chacune des phases de reconfiguration s'effectue en 1 cycle d'horloge, laissant le maximum de temps à l'architecture pour le traitement des données. Les différents temps de configuration ainsi que les temps d'implémentation de chaque fonction sont résumés dans la figure 4-25. Á partir de ces informations, nous pouvons également définit la fréquence de fonctionnement du circuit à 93 MHz. Dans la section suivante, nous allons procéder à la description xMAML de DART, nécessaire à la génération automatique des ressources de reconfiguration dynamique par MOZAÏC, ainsi qu'à l'étude de l'impact de l'introduction de ces ressources sur l'ensemble du circuit.

C.3 Description xMAML de DART

De manière à procéder à la génération des ressources nécessaires à la mise en œuvre de la reconfiguration dynamique, MOZAÏC a besoin de la description xMAML de l'architecture DART.

C.3-1 Description des DyRIBox

Deux DyRIBox différentes sont nécessaires aux interconnexions des ressources. La première DyRIBox, la DyRIBox de DPR, permet de connecter les entrées des unités fonctionnelles et des registres de garde aux sorties de ces mêmes ressources ainsi qu'aux bus globaux. La deuxième DyRIBox, la DyRIBox de *cluster*, est placée sur quatre des huit bus globaux et

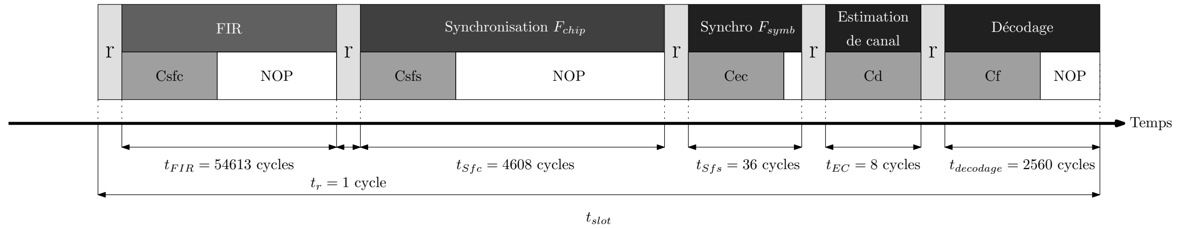


Figure 4-25 – **Temps d'implémentation et de reconfiguration des tâches du décodeur WCDMA sur DART**. *Chacune des fonctions devant être implémentée sur DART se fait par une reconfiguration de 1 cycle d'horloge. Les différentes configurations,* Csfc, Csfs, Cec, Cd, Cf *respectivement dédiées aux fonctions* synchronisation à la fréquence chip, synchronisation à la fréquence symbole, estimation de canal, decodage et filtre, *sont acheminées en parallèle aux phases de traitement.*

permet de définir quelles sorties parmi celles des registres de garde et des unités fonctionnelles seront connectées à chacun des quatre bus globaux.

DYRIBOX DE DPR : la DyRIBox de DPR est chargée d'orienter les données issues des mémoires, des sorties des registres et des unités fonctionnelles. Comme le montre la figure 4-26, pour chacune des deux entrées de chaque unité fonctionnelle et pour chacun des deux registres, des connexions sont réalisables avec les quatre mémoires locales, les huit bus globaux, et les six sorties provenant des registres de garde et des unités fonctionnelles. La description xMAML de cette DyRIBox donnée dans le listing 4-7 montre l'implémentation de huit ports d'entrée d'une largeur de 32 bits provenant des bus globaux (ligne 4), de huit ports d'entrée d'une largeur de 32 bits provenant des quatre mémoires, des deux registres et des UF 2 et UF4 (ligne 7) et de deux ports d'entrée de 40 bits provenant des UF1 et UF3 (ligne 8). Au niveau des sorties, cette DyRIBox est composée de six ports de sortie de 32 bits, connectés aux deux entrées des registres et aux quatre entrées des UF2 et UF4, ainsi que de quatre ports de sortie de 40 bits connectés aux entrées des UF1 et UF3.

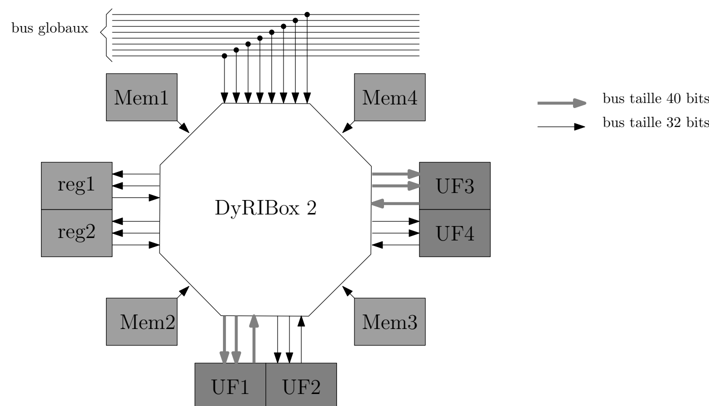


Figure 4-26 – **Schéma synoptique de la DyRIBox de DPR de DART**. *La DyRIBox de DPR permet de connecter huit bus globaux, dix sorties issues des registres, des unités fonctionnelles et des mémoires vers les entrées des unités fonctionnelles et des registres.*

```
1     <PEInterconnectDyRIBox name="DBox_1">
2        <Reconfiguration cycle = "1"  bitwidth = "8" />
3        <DBPorts>
4           <Inputs number="8" bitwidth="32" />
5        </DBPorts>
6        <PElementsPorts>
7           <Inputs number="8" bitwidth="32" />
8           <Inputs number="2" bitwidth="40" />
9           <Outputs number="6" bitwidth="32" /> # registres, UF2 et UF4
10          <Outputs number="4" bitwidth="40" /> # UF1 et UF3
11       </PElementsPorts>
12    </PEInterconnectDyRIBox>
```

Listing 4-7 – Description xMAML de la DyRIBox de DPR

```
1     <PEInterconnectDyRIBox name="DBox_2">
2        <Reconfiguration cycle = "1"  bitwidth = "8" />
3        <DBPorts>
4           <Outputs number="8" bitwidth="32" />
5        </DBPorts>
6        <PElementsPorts>
7           <Inputs number="60" bitwidth="32" />
8        </PElementsPorts>
9        <AdjacencyMatrix>
10          <DOutput idx="0" row="00000011111100000011111000000111111" />
11          <DOutput idx="1" row="11111100000011111100000011111000000" />
12          <DOutput idx="2" row="00000011111100000011111000000111111" />
13          <DOutput idx="3" row="11111100000011111100000011111000000" />
14          <DOutput idx="4" row="00000011111100000011111000000111111" />
15          <DOutput idx="5" row="11111100000011111100000011111000000" />
16          <DOutput idx="6" row="00000011111100000011111000000111111" />
17          <DOutput idx="7" row="11111100000011111100000011111000000" />
18       </AdjacencyMatrix>
19    </PEInterconnectDyRIBox>
```

Listing 4-8 – Description xMAML de la DyRIBox de Cluster

DYRIBOX DE *cluster* : comme le montre la figure 4-27, la DyRIBox de *cluster* est chargée de définir quelles sorties parmi les 30 sorties provenant des unités fonctionnelles et des registres peuvent être connectées aux bus globaux. Seuls quatre bus parmi les huit sont accessibles par DPR. La description xMAML de cette DyRIBox, présentée dans le listing 4-8, permet de spécifier cette restriction (lignes 10 à 17). Seules les connexions provenant des DPR pairs sont acceptées sur les bus globaux pairs, et seules les connexions provenant des DPR impairs sont acceptées sur les bus globaux impairs. Cette DyRIBox est composée de huit ports de sorties connectés aux bus globaux et de 30 ports d'entrée connectés aux sorties de chacune des ressources de l'ensemble des DPR d'un *cluster* (registres de garde, unités fonctionnelles).

C.3-2 DESCRIPTION DES DPR

Un DPR est composé de quatre mémoires locales associées à quatre générateurs d'adresses, AG (*Address Generator*), de quatre unités fonctionnelles, UF1, UF2, UF3 et UF4 et de deux registres de garde.

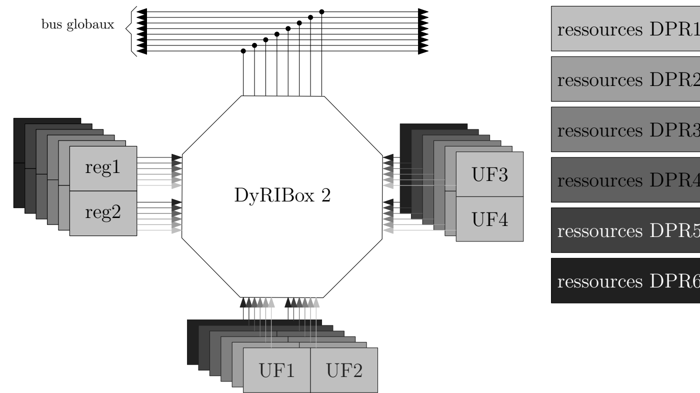


Figure 4-27 – **Schéma synoptique de la DyRIBox de *cluster* de DART**. *La DyRIBox de* cluster *permet de connecter les sorties de toutes les unités fonctionnelles et de tous les registres composant un* cluster *de DART. Seuls quatre bus globaux sont accessibles par DPR.*

```
1  <PEInterface name="AG" >
2      <Reconfiguration cycle = "1"  bits = "4" preemption="no"/>
3      <IOPorts>
4        <Port name="RW_1" bitwidth="1" direction="out" type="ctrl"/>
5        <Port name="Addr_1" bitwidth="8" direction="out" type="ctrl"/>
6        <Port name="RW_2" bitwidth="1" direction="out" type="ctrl"/>
7        <Port name="Addr_2" bitwidth="8" direction="out" type="ctrl"/>
8        <Port name="RW_3" bitwidth="1" direction="out" type="ctrl"/>
9        <Port name="Addr_3" bitwidth="8" direction="out" type="ctrl"/>
10       <Port name="RW_4" bitwidth="1" direction="out" type="ctrl"/>
11       <Port name="Addr_4" bitwidth="8" direction="out" type="ctrl"/>
12       <Port name="rst" bitwidth="1" direction="in" type="rst"/>
13       <Port name="clk" bitwidth="1" direction="in" type="clk"/>
14       <Port name="Config" bitwidth="4" direction="in" type="ScanIn"/>
15       <Port name="ConfigEn" bitwidth="1" direction="in" type="ScanEn"/>
16     </IOPorts>
17   </PEInterface>
```

Listing 4-9 – Description de l'interface des générateurs d'adresses de DART dans MOZAÏC

AG : les générateurs d'adresses sont les garants d'une bonne communication entre les unités fonctionnelles et les mémoires. Quatre bits sont nécessaires à sa configuration. La taille des configurations étant assez modeste, nous avons regroupé l'ensemble des générateurs d'adresses de manière à ne générer qu'un DUCK pour l'ensemble de ceux-ci. La description xMAML (listing 4-9) permet de fournir les informations nécessaires à la génération d'un DUCK adapté. Les entrées `RW_n` sont utilisées pour la validation en lecture ou en écriture à l'adresse `Addr_n` de données présentent en entrée des mémoires locales.

UF1 ET UF3 : pour les mêmes raisons que pour les générateurs d'adresses, nous avons regroupé la gestion des configurations des unités fonctionnelles "1" et "3" , celles-ci étant identiques. Les calculs sont effectués sur les entrées `In_1_UF_n` et `In_2_UF_n` (listing 4-10) et le résultat des calculs est fourni sur la sortie `S_n`. Trois bits sont nécessaires à la configuration de chaque unité fonctionnelle afin de définir l'opération à effectuer.

```
1 <PEInterface name="UF_1_3" >
2     <Reconfiguration cycle = "1"  bits = "6" preemption="no"/>
3     <IOPorts>
4       <Port name="In_1_UF_1" bitwidth="32" direction="in" type="data"/>
5       <Port name="In_2_UF_1" bitwidth="32" direction="in" type="data"/>
6       <Port name="In_1_UF_3" bitwidth="32" direction="in" type="data"/>
7       <Port name="In_2_UF_3" bitwidth="32" direction="in" type="data"/>
8       <Port name="S_1" bitwidth="32" direction="out" type="data"/>
9       <Port name="S_3" bitwidth="32" direction="out" type="data"/>
10      <Port name="rst" bitwidth="1" direction="in" type="rst"/>
11      <Port name="clk" bitwidth="1" direction="in" type="clk"/>
12      <Port name="Config" bitwidth="6" direction="in" type="ScanIn"/>
13      <Port name="ConfigEn" bitwidth="1" direction="in" type="ScanEn"/>
14    </IOPorts>
15   </PEInterface>
```

Listing 4-10 – Description de l'interface des unités fonctionnelles "1" et "3" de DART dans Mozaïc

```
1 <PEInterface name="UF_2_4" >
2     <Reconfiguration cycle = "1"  bits = "22" preemption="no"/>
3     <IOPorts>
4       <Port name="In_1_UF_2" bitwidth="40" direction="in" type="data"/>
5       <Port name="In_2_UF_2" bitwidth="40" direction="in" type="data"/>
6       <Port name="In_1_UF_4" bitwidth="40" direction="in" type="data"/>
7       <Port name="In_2_UF_4" bitwidth="40" direction="in" type="data"/>
8       <Port name="S_2" bitwidth="40" direction="out" type="data"/>
9       <Port name="S_4" bitwidth="40" direction="out" type="data"/>
10      <Port name="rst" bitwidth="1" direction="in" type="rst"/>
11      <Port name="clk" bitwidth="1" direction="in" type="clk"/>
12      <Port name="Config" bitwidth="22" direction="in" type="ScanIn"/>
13      <Port name="ConfigEn" bitwidth="1" direction="in" type="ScanEn"/>
14    </IOPorts>
15   </PEInterface>
```

Listing 4-11 – Description de l'interface des unités fonctionnelles "2" et "4" de DART dans Mozaïc

UF2 et UF4 : encore une fois, nous avons regroupé la gestion des configurations des unités fonctionnelles "2" et "4". Les calculs sont effectués sur les entrées `In_1_UF_n` et `In_2_UF_n` (listing 4-11) et génèrent la sortie `S_n`. Onze bits sont nécessaires à la configuration des UAL pour la spécification des traitements SWP par exemple.

Registres : un bit de configuration par registre de garde est nécessaire. Six registres de garde sont présents dans un DPR. Le regroupement de la gestion de leur reconfiguration conduit à la description xMAML présentée par le listing 4-12. Cette description définit les entrées D_i et les sorties Q_i de chaque registre i.

Mémoires locales : les mémoires locales n'ont pas besoin d'être configurées. Par conséquent, l'interface xMAML (listing 4-13) ne sera utile que pour la spécification des interconnexions avec les autres ressources tel que le bus d'adresse provenant des AG ou les bus de données allant vers les DyRIBox.

```
1  <PEInterface name="reg_garde" >
2      <Reconfiguration cycle = "1"  bits = "6" preemption="no"/>
3      <IOPorts>
4        <Port name="D_1" bitwidth="32" direction="in" type="data"/>
5        <Port name="D_2" bitwidth="32" direction="in" type="data"/>
6        <Port name="D_3" bitwidth="32" direction="in" type="data"/>
7        <Port name="D_4" bitwidth="32" direction="in" type="data"/>
8        <Port name="D_5" bitwidth="32" direction="in" type="data"/>
9        <Port name="D_6" bitwidth="32" direction="in" type="data"/>
10       <Port name="Q_1" bitwidth="32" direction="out" type="data"/>
11       <Port name="Q_2" bitwidth="32" direction="out" type="data"/>
12       <Port name="Q_3" bitwidth="32" direction="out" type="data"/>
13       <Port name="Q_4" bitwidth="32" direction="out" type="data"/>
14       <Port name="Q_5" bitwidth="32" direction="out" type="data"/>
15       <Port name="Q_6" bitwidth="32" direction="out" type="data"/>
16       <Port name="rst" bitwidth="1" direction="in" type="rst"/>
17       <Port name="clk" bitwidth="1" direction="in" type="clk"/>
18       <Port name="Config" bitwidth="6" direction="in" type="ScanIn"/>
19       <Port name="ConfigEn" bitwidth="1" direction="in" type="ScanEn"/>
20     </IOPorts>
21    </PEInterface>
```

Listing 4-12 – Description de l'interface des registres de garde de DART dans Mozaïc

```
1        <PEInterface name="Mem" >
2      <Reconfiguration cycle = "0"  bits = "0" preemption="no"/>
3      <IOPorts>
4        <Port name="input" bitwidth="16" direction="in" type="data"/>
5        <Port name="output" bitwidth="16" direction="out" type="data"/>
6        <Port name="Addr" bitwidth="8" direction="in" type="ctrl"/>
7        <Port name="RW" bitwidth="1" direction="in" type="ctrl"/>
8        <Port name="rst" bitwidth="1" direction="in" type="rst"/>
9        <Port name="clk" bitwidth="1" direction="in" type="clk"/>
10     </IOPorts>
11    </PEInterface>
```

Listing 4-13 – Description de l'interface des registres de garde de DART dans Mozaïc

C.4 Génération des mécanismes de reconfiguration dynamique pour DART

À partir de la description xMAML présentée précédemment, Mozaïc peut procéder à la génération des ressources de reconfiguration dynamique. Nous allons détailler les calculs effectués automatiquement par Mozaïc pour l'estimation du coût des différentes ressources produites. La propagation des configurations requiert un temps qui varie en fonction de la taille des configurations, du nombre de domaines de configurations et de la taille du bus de configurations. Afin de déterminer si les contraintes temporelles peuvent être respectées, il est nécessaire d'estimer ces paramètres.

C.4-1 Estimation de la taille des configurations

Chaque implémentation d'une fonction du décodeur WCDMA requiert une reconfiguration dynamique de type matérielle. Chaque DPR nécessite 38 bits de configurations pour les

CIBLE DE LA RECONFIGURATION	NB BITS /RESSOURCE	NB RESSOURCES /DPR	NB BITS /DPR	NB BITS /CLUSTER
Gardes des générateurs d'adresses	1	4	4	24
Gardes registres	1	6	6	36
Multiplieurs/additionneurs	3	2	6	36
UAL	11	2	22	132
TOTAL			38	228

Tableau 4-5 – Répartition des bits de configuration par ressource de DPR.

CIBLE DE LA RECONFIGURATION	NB DUCK /DPR	NB DUCK /CLUSTER	NB REGISTRES /DUCK	NB BITS /CLUSTER
Gardes des générateurs d'adresses	1	6	4	24
Gardes registres	1	6	8	48
Multiplieurs/additionneurs	1	6	8	48
UAL	1	6	24	144
TOTAL			44	264

Tableau 4-6 – Répartition des configurations DUCK pour un *cluster* de DART.

ressources de calculs (tableau 4-5) qui devront être placés dans les registres DUCK pendant les phases de traitement. Nous utiliserons un bus de quatre bits de largeur pour l'interconnexion des DUCK. Par conséquent, le nombre de bit total de configuration des unités de traitement, des générateurs d'adresse et des registres d'un DPR (TC_{dpr}), registres "fantômes" compris, sera de :

$$TC_{dpr} = 4 \times \left\lceil \frac{4}{4} \right\rceil + 4 \times \left\lceil \frac{6}{4} \right\rceil + 4 \times \left\lceil \frac{6}{4} \right\rceil + 4 \times \left\lceil \frac{22}{4} \right\rceil = 44\ bits \qquad (4\text{-}6)$$

Cela implique donc que pour l'ensemble de l'architecture DART, 264 bits de configurations seront nécessaires à la configuration d'un *cluster* hors configuration des interconnexions.

Le tableau 4-6 résume le nombre de DUCK et leur contenu utilisés pour les différentes ressources qui composent un *cluster* de DART, ainsi que le nombre de registres "fantômes" inclus par la génération automatique de ces DUCK. Une configuration des ressources de calcul d'un *cluster* complet est stockée dans 44 DUCK. Six DUCK de quatre bits sont nécessaires aux configurations des générateurs d'adresse, douze DUCK de huit bits pour les registres de garde et les multiplieurs/additionneurs, et six DUCK de 24 bits pour les UAL.

En ce qui concerne les unités d'interconnexion, chaque DPR intègre un réseau multi-bus complètement connecté. Dans le cadre d'une conception basée sur le principe de reconfiguration implémenté par MOZAÏC, le réseau multi-bus est réalisé par une DyRIBox présentée à la section C.3. Cette DyRIBox permet de connecter dix-huit entrées sur dix sorties. La taille d'une configuration de la DyRIBox de DPR ($TCDB_{dpr}$) a donc pour valeur :

$$TCDB_{dpr} = 4 \times \left\lceil \frac{\sum_{n=0}^{9} log_2(18)}{4} \right\rceil = 52\ bits \qquad (4\text{-}7)$$

Au niveau des interconnexions du *cluster*, une DyRIbox est chargée d'orienter les communications issues des 60 registres et unités fonctionnelles des six DPR. Cette DyRIBox a également

été présentée à la section C.3. Seuls 30 connexions par sortie sont possibles. La taille d'une configuration de DyRIBox de *cluster* ($TCDB_{cluster}$) a donc pour valeur :

$$TCDB_{cluster} = \sum_{n=0}^{7} \lceil log_2(30) \rceil = 40\ bits \tag{4-8}$$

Le total des bits de configuration des interconnexions d'un cluster et de six DPR (TCDB) requiert donc $TCDB = 6 \times 52 + 40 = 352$ bits.

Si nous additionnons la taille définie précédemment nécessaire à la configuration des ressources et des interconnexions, nous pouvons alors déterminer que la taille d'une configuration complète de DART (TC_{DART}) a pour valeur $TC_{DART} = 264 + 352 = 616$ bits.

C.4-2 Estimation du nombre de domaines de reconfiguration

Le nombre de domaines de reconfiguration (N_D) est déterminé par le temps disponible entre deux phases de reconfiguration et le temps de propagation d'un contexte ($Prop_t$). $Prop_t$ est déterminé par la taille de la configuration en nombre de registres et la vitesse de lecture de la mémoire contenant les bits de configuration. Une mémoire RAM intégrée sur puce d'une capacité de 32768×8 bits suffisante pour le stockage des différentes configurations permet une vitesse de lecture de plus de 300 MHz [1]. Le temps de propagation d'un nouveau contexte complet par l'intermédiaire d'un bus de huit bits peut alors être réalisé en :

$$Prop_t = \frac{616/8}{300E^6} = 257\ ns \tag{4-9}$$

Le temps de propagation le plus court apparaît pendant l'exécution de la fonction d'estimation de canal. Ce contexte est implémenté pendant huit cycles d'horloges. Si l'on considère que les unités de traitement opèrent à une fréquence de 93 MHz alors le temps de propagation d'un contexte disponible est de $86,02E^{-9}\ s$. Par conséquent, le nombre de domaines nécessaires pour maintenir les contraintes temporelles imposées par l'application WCDMA est de :

$$N_D = \left\lceil \frac{257E^{-9}}{86,02E^{-9}} \right\rceil = 3 \tag{4-10}$$

C.4-3 Estimation de la consommation et de la surface de silicium induits par l'utilisation du concept DUCK

Étant donné que 264 bits de configuration sont répartis sur des registres de quatre bits. La surface de silicium (S) supplémentaire dû à l'utilisation des DUCK est alors $S = 562,9 \times 264 \times 49,8 = 0,0137\ mm^2$

Afin d'estimer l'impact de l'introduction d'un DUCK sur une ressource dans le chemin de configuration, nous avons comparé la surface de celui-ci relativement à la surface de la ressource qui y est associée. Cette comparaison n'ayant que très peu d'intérêt pour les registres de garde, nous ne nous intéresserons qu'aux unités fonctionnelles 1 et 3 (Multiplieurs/additionneurs)

1. Données fournies par STMicroelectronics

Ressource du DUCK	Nb de registres /DUCK	surface en mm^2		Rapport surface DUCK/ressource
		/DUCK	/ressource	
Multiplieurs/additionneurs	2	0,0008	$0,0368 \times 2$	1,08%
UAL	6	0,0018	$0,0363 \times 2$	2,47%

Tableau 4-7 – Comparaison et impact de l'introduction du DUCK sur la surface des unités fonctionnelles.

Ressource du DUCK	Nb de registres /DUCK	Consommation en mW		Rapport Conso DUCK/ressource
		/DUCK	/ressource	
Multiplieurs/additionneurs	2	0,41	9,32	4,4%
UAL	6	0,65	6,9	9,4%

Tableau 4-8 – Comparaison et impact de l'introduction du DUCK sur la consommation des unités fonctionnelles.

et les unités fonctionnelles 2 et 4 (UAL). Les résultats de cette étude comparative sont regroupées dans le tableau 4-7 et montrent que pour les deux DUCK dédiés aux UF1,2,3,4, l'impact sur la surface reste modeste puisque ceux-ci ne représentent respectivement que 1,08% et 2,47% de la surface totale par unité de traitement.

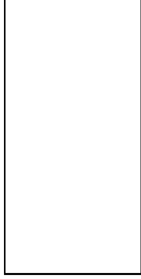
Si l'addition de chacune des surfaces de DUCK estimée permet d'évaluer approximativement la surface totale, cela n'est pas possible pour le calcul de la consommation. Par conséquent, nous ne rapporterons ici que la consommation des DUCK de manière individuelle, par ressource. Pour cela nous utilisons l'équation établie au chapitre III à la section D.1-1 qui est fonction du nombre de registre (Rn) et de leur taille (Rs) $CD_{para} = 1,16 + Rn \times Rs \times 0,0246$. Les résultats de comparaison ont été reporté dans le tableau 4-8. L'impact des DUCK sur la consommation est relativement modeste, puisque la consommation du DUCK dédié aux UF1 et 3 ne représente que 4,4% de leur consommation en énergie. La consommation du DUCK dédié aux UF2 et 4 ne représente pour sa part que 9,4% de leur consommation en énergie.

C.4-4 Conclusion

L'architecture DART a été conçue dans l'objectif d'implémenter de manière reconfigurable dynamiquement des applications de télécommunications mobiles et à faible consommation. Cette partie s'est attachée à démontrer que la conception d'une architecture de type DART est envisageable avec la plate-forme Mozaïc. Les contraintes temporelles ont été respectées grâce à l'introduction des concepts de reconfiguration dynamique de celle-ci. De plus, l'impact de l'introduction des ressources DUCK reste modeste comparés à la surface et la consommation des unités fonctionnelles implémentées dans les DPR. Les données binaires de configuration issues des outils cDART, gDART et ACG sont stockées en mémoire et lues par le contrôleur de configuration après un reformatage des données de manière à assurer une compatibilité entre les différents outils et l'architecture produite.

D Synthèse

Dans ce chapitre, nous avons présenté la généricité de la plate-forme Mozaïc par l'implémentation d'un décodeur WCDMA sur un processeur reconfigurable, DART, et sur un modèle de FPGA embarqué basé sur les CLB de la famille XC4000 de chez Xilinx. Nous avons montré de quelle manière la plate-forme Mozaïc s'intègre aux différents environnements de développement associés. L'utilisation de Mozaïc a permis de générer automatiquement l'ensemble des ressources nécessaires à la mise en œuve de la reconfiguration dynamique. De plus, grâce à Mozaïc, il nous a été permis d'estimer l'impact de la reconfiguration dynamique, et ainsi de montrer les avantages de celle-ci par rapport à une implémentation statique.



Conclusions et perspectives

Synthèse des travaux

Les travaux présentés dans ce document sont dédiés à la définition d'une plate-forme générique de modélisation et de conception d'architectures reconfigurables dynamiquement. La plate-forme que nous présentons, Mozaïc, est dédiée à la spécification des critères de reconfiguration dynamique. Les mécanismes de reconfiguration dynamique mis en œuvre par Mozaïc intègrent des concepts permettant la gestion flexible des unités de traitement implémentées. Ces unités de traitement, conçues au préalable et spécifiées à un niveau d'abstraction permettant leur synthèse, sont intégrées quel que soit leur granularité ou leur méthode de reconfiguration. Depuis la spécification de l'architecture implémentée, Mozaïc permet l'exploration des critères de flexibilité, de dynamicité et de performances apportés par les mécanismes de reconfiguration dynamique générés.

Tout d'abord, nous avons présenté le modèle de reconfiguration dynamique sous-jacent à Mozaïc. Le temps de reconfiguration réduit au minimum, la reconfiguration partielle, la gestion des interruptions et des mécanismes de préemption sont autant de concepts supportés qui permettent de répondre aux exigences des contraintes temporelles des applications ciblées. Ces mécanismes sont supportés localement par la ressource DUCK. Le DUCK est une mémoire de configuration locale par laquelle passe le bus de configuration. Cette mémoire locale est une mémoire parallèle dans laquelle est stockée la future configuration. Le DUCK est aussi bien dédié à une unité de traitement qu'à une unité d'interconnexion ou d'entrée/sortie. La configuration stockée dans le DUCK peut être acheminée vers la zone de configuration de l'unité concernée via l'interface de configuration de cette unité aussi rapidement que le permet celle-ci. Nous avons également introduit le concept DyRIBox, interconnexion reconfigurable dynamiquement capable de supporter l'éventuelle hétérogénéité des ressources qui viendront composer l'architecture ainsi que de procéder à la reconfiguration d'une partie de ses connexions sans perturber les communications entre les ressources qui ne sont pas concernées par la reconfiguration.

De manière à exploiter les caractéristiques de reconfiguration dynamique mis en œuvre par Mozaïc de la manière la plus adaptée aux critères F-D-P recherchés, nous avons développé un langage de description d'architecture à haut niveau xMAML. xMAML est dédié à l'intégration et à la gestion cohérente des unités de traitement reconfigurables dynamiquement. Ce langage permet de procéder rapidement à une spécification des processus de reconfiguration des unités de traitement à implémenter de façon à générer automatiquement les ressources adaptées qui seront nécessaires à la mise en œuvre de la reconfiguration dynamique. Une analyse de la description xMAML permet une première estimation rapide de la surface de silicium et de la consommation d'énergie induite par la mise en œuvre de la reconfiguration dynamique. De plus l'analyse de la description permet également de déterminer la taille des flots de données de configuration propres aux ressources de connexions et aux unités de traitement ainsi qu'à l'estimation du nombre de domaines de reconfiguration nécessaires selon l'application choisie.

Enfin, nous avons démontré la généricité de notre plate-forme en proposant deux solutions d'implémentations de décodeur WCDMA sur un FPGA embarqué à base de CLB de la famille des XC4000 de chez Xilinx et sur le processeur reconfigurable DART pour lesquels les ressources de gestion de la reconfiguration dynamique ont été générés par Mozaïc. Nous avons montré de quelle manière la plate-forme Mozaïc s'intègre aux différents environnements de développement associés. De plus, grâce à Mozaïc, il nous a été permis d'estimer l'impact de la reconfiguration dynamique, et ainsi de montrer les avantages de celle-ci par rapport à une implémentation statique.

Perspectives

L'état de l'art que nous avons présenté au début de ce document montre à quel point le domaine des architectures reconfigurables dynamiquement est étendu. Dans le cadre des travaux présentés dans ce document, plusieurs perspectives sont envisageables.

Parmi ces problématiques, nous pouvons citer l'intégration totale de Mozaïc avec les plate-formes de développement d'architectures et de leur outils tels que VPR et les FPGA ou CALIFE et DART. En effet, dans le chapitre IV, qui présente l'utilisation complémentaire de ces plate-formes avec Mozaïc, nous avons été obligés de procéder à certaines modifications des résultats de compilation ou de synthèse de manière manuelle afin de générer les données de configuration. Certaines de ces étapes ont fait l'objet de développement d'outils spécifiques, notamment pour l'analyse et la génération automatique des données de configuration sur FPGA, mais il serait intéressant de proposer des langages intermédiaires qui permettraient d'automatiser le développement de passerelles de conception avec "toutes" les plate-formes existantes. De plus, grâce à ces outils, il serait possible d'intégrer entièrement la conception et le développement d'application multi-support tels que l'utilisation d'un processeur DART couplé à un FPGA embarqué pour le traitement dynamique de boucle par exemple.

Le modèle de reconfiguration dynamique que nous avons présenté est focalisé sur la gestion des processus de reconfiguration. Cependant, l'intégration de ressources de calcul hétérogènes contribue à augmenter la quantité de données de configuration de manière inefficace. Cette inefficacité est due à l'introduction de registres fantômes pénalisant les résultats de surface et de consommation de l'architecture. Une possibilité de palier à ce problème serait de proposer la

génération automatique des unités de traitement par l'intermédiaire de langages de description d'architectures que nous avons présentés dans le chapitre III. Cela apporterait l'avantage de proposer des phases d'exploration portant sur l'ensemble de l'architecture et de proposer un modèle complet de l'architecture aux outils de compilation/synthèse et placement/routage.

D'une manière plus pragmatique, dans le chapitre II, nous avons établi les principes d'une communication par paquets à base de DyRIBox. Cependant des travaux restent encore à effectuer concernant la gestion et la configuration de ces routeurs. Les tables d'orientation à la base de notre système ont besoin d'être maintenues à jours à chaque configuration de domaine. Une solution envisageable consisterait à générer des trames de configuration empruntant les voies de communications de données. Ainsi, à la réception de l'entête de trame, chaque DyRIBox serait capable de détecter celles qui lui sont destinées.

Un des aspects les plus contraignants de nos outils concerne l'estimation de la consommation. En effet, de part la généricité des architectures modélisables, il est important de concevoir des modèles de consommation plus "réalistes" que l'approche que nous en avons faite. Une étude microscopique, par ressource générée reste possible, mais l'étude plus macroscopique de la consommation de l'architecture générée se heurte aux barrières de la généricité des architectures produites. De plus, d'une manière générale, il serait sans aucun doute profitable de réfléchir à des méthodes d'optimisation dans les processus de reconfiguration de manière à réduire l'énergie consommée.

Nous avons vu que la description d'une architecture régulière, tels que le FPGA embarqué présentée au chapitre IV, est une opération fastidieuse de par la quantité de paramètres de connexions à spécifier. Une évolution efficace de l'outil de description serait de permettre l'automatisation de la génération de topologies pré-définies qu'il suffirait de paramétrer. Nous avons d'ailleurs, à ce titre, exploré quelques possibilités dans la génération d'un réseau *mesh*, cependant, quelques problèmes restaient encore à éclaircir au niveau des connexions locales des unités de traitement. Une amélioration de la description xMAML pourrait également intervenir au niveau des architectures composées de ressources dupliquées. La description générique de ces ressources, ou la répétition de code serait gérée par des instructions de boucle permettant une simplification des descriptions. Par exemple, la description xMAML de DART, tel que l'outil le permet aujourd'hui, nécessite la description de chacun des DPR qui compose un *cluster*. Grâce à cette méthode de description, il serait possible de procéder, par boucle, à une instanciation automatique des cinq DPR restants.

Bibliographie

[1] ATMEL. AT40K FPGAs. Technical report, ATMEL Inc., 1999.

[2] Xilinx. Xc6200 field programmable gate arrays. Technical report, Xilinx, 1997.

[3] K. Compton and S. Hauck. Reconfigurable computing : a survey of systems and software. *ACM Comput. Surv.*, 34 :171–210, 2002.

[4] Xilinx. Virtex 2.5 V Field Programmable Gate Arrays : Product Specification. Technical report, Xilinx, 2001.

[5] A. DeHon. Dynamically programmable gate arrays : A step toward increased computational density. In *Proc. of Fourth Canadian Workshop of Field Programmable Devices*, Toronto, Canada, 1996.

[6] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed FPGA. In *FCCM '97 : Proc. of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, pages 22–27, Washington, DC, USA, 1997.

[7] D. Koch, A. Ahmadinia, C. Bobda, H. Kalte, and J. Teich. FPGA architecture extensions for preemptive multitasking and hardware defragmentation. In *Proc. of the IEEE International Conference on Field-Programmable Technology (FPT)*, pages 433–436, 2004.

[8] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. R. Taylor. Piperench : A virtualized programmable datapath in 0.18 micron technology. In *Proc. of the IEEE Custom Integrated Circuits Conference (CICC)*, pages 63–66, 2002.

[9] ATMEL. 5K - 50K Gates Coproessor FPGA with FreeRAM. Technical report, ATMEL Inc., 2002.

[10] G. Lecurieux Lafayette. Programmable system level integration brings system-on-chip design to the desktop. In *FPL '00 : Proc. of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, pages 789–792. Lecture Notes in Computer Science (LNCS), vol. 1896, 2000.

[11] Application Note : Virtex Series. Virtex series configuration architecture. User guide, Xilinx Inc, 2004.

[12] R. DAVID. *Architecture reconfigurable dynamiquement pour applications mobiles.* PhD thesis, Université de Rennes I, 2003.

[13] G. Sassatelli, L. Torres, P. Benoit, T. Gil, C. Diou, G. Cambon, and J. Galy. Highly Scalable Dynamically Reconfigurable Systolic Ring-Architecture for DSP Applications. In *Proc. of the conference on Design, automation and test in Europe (DATE '02)*, page 553, Washington, DC, USA, 2002.

[14] D. Kissler, F. Hannig, A. Kupriyanov, and J. Teich. A Dynamically Reconfigurable Weakly Programmable Processor Array Architecture Template. In *Proc. of the 2nd International Workshop on Reconfigurable Communication Centric System-on-Chips (ReCoSoC)*, pages 31–37, Montpellier, France, July 2006.

[15] D. Kissler, F. Hannig, A. Kupriyanov, and J. Teich. A Highly Parameterizable Parallel Processor Array Architecture. In *Proc. of the IEEE International Conference on Field Programmable Technology (FPT)*, pages 105–112, Bangkok, Thailand, December 2006. IEEE.

[16] Lattice. ispXPGA Family. Technical report, Lattice Semiconductor Corporation, 2007.

[17] Lattice. ispXP Configuration Usage Guidelines. Technical report, Lattice Semiconductor Corporation, 2002.

[18] A. Cappelli, A. Lodi, C. Mucci, M. Toma, and Fabio Campi. A dataflow control unit for c-to-configurable pipelines compilation flow. In *FCCM '04 : Proc. of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 332–333, Washington, DC, USA, 2004.

[19] PACT. The XPP white paper : A technical perpective. Technical report, Realease 2.1, 2002.

[20] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. PACT XPP—A Self-Reconfigurable Data Processing Architecture. *J. Supercomput.*, 26(2) :167–184, 2003.

[21] B. Mei, A. Lambrechts, D. Verkest, J. Y. Mignolet, and R. Lauwereins. Architecture Exploration for a Reconfigurable Architecture Template. *IEEE Design and Test*, March.

[22] M. Suzuki, Y. Hasegawa, V. M. Tuan, S. Abe, and H. Amano. A cost-effective context memory structure for dynamically reconfigurable processors. In *Proc. of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, pages 8–14, 2006.

[23] H. Amano, T. Inuo, H. Kami, T. Fujii, and M. Suzuki. Techniques for virtual hardware on a dynamic reconfigurable processor -An approach to tough cases. In *Proc. of the International Conference on Field Programmable Logic and Application (FPL2004)*, pages 464–473. Lecture Notes in Computer Science (LNCS), vol. 3203, 2004.

[24] D. Fischer, J. Teich, R. Weper, and M. Thies. BUILDABONG : A Framework for Architecture/Compiler Co-Exploration for ASIPs. *Journal of Circuits, Systems, and Computers*, 12(3) :353–372, 2003.

[25] L. Lagadec and B. Pottier. Object-oriented meta tools for reconfigurable architectures. In John Schewel ; Peter M. Athanas ; Chris H. Dick ; John T. McHenry, editor, *Reconfigurable Technology : FPGAs for Computing and Applications II*, volume 4212, pages 69–79, 2000.

[26] L. Lagadec, D. Lavenier, E. Fabiani, and B. Pottier. Placing, routing, and editing virtual fpgas. In *Proc. of the 11th International Conference on Field-Programmable Logic and Applications (FPL '01)*, pages 357–366. Lecture Notes in Computer Science (LNCS), vol. 2147, 2001.

[27] V. George, H. Zhang, and J. Rabaey. The design of a low energy fpga. In *Proc. of the 1999 international symposium on Low power electronics and design (ISLPED '99)*, pages 188–193, New York, NY, USA, 1999.

[28] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Sis : A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.

[29] C. Dezan, L. Lagadec, M. Leuchtenburg, T. Wang, P. Narayanan, and A. Moritz. Building CAD Prototyping Tool for Emerging Nanoscale Fabrics. 2008.

[30] L. Lagadec, B. Pottier, and A. Poungou. A layered methodology for fast deployment of new technologies. In *European Nano Systems Worshop*, pages 20–24. HAL - CCSD, 2007.

[31] K. Siozios, G. Koutroumpezis, K. Tatas, N. Vassiliadis, V. Kalenteridis, H. Pournara, I. Pappas, D. Soudris, A. Thanailakis, S. Nikolaidis, and S. Siskos. A Novel FPGA Architecture and an Integrated Framework of CAD Tools for Implementing Applications. *IEICE - Trans. Inf. Syst.*, E88-D(7) :1369–1380, 2005.

[32] V. Betz and J. Rose. VPR : A new packing, placement and routing tool for FPGA research. In Wayne Luk, Peter Y. K. Cheung, and Manfred Glesner, editors, *Field-Programmable Logic and Applications*, pages 213–222. Lecture Notes in Computer Science (LNCS), vol. 1304, 1997.

[33] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220 :671–680, 1983.

[34] V. Betz and J. Rose. Directional bias and non-uniformity in fpga global routing architectures. In *Proc. of the 1996 IEEE/ACM international conference on Computer-aided design (ICCAD '96)*, pages 652–659, Washington, DC, USA, 1996.

[35] C. Ebeling, L. McMurchie, S. Hauck, and S. M. Burns. Placement and routing tools for the Triptych FPGA. *IEEE Trans. VLSI Syst.*, 3(4) :473–482, 1995.

[36] W. J. Dally and B. Towles. Route packets, not wires : on-chip interconnection networks. In *Proc. of Design Automation Conference, (DAC'01)*, pages 684–689, 2001.

[37] C. Bobda and A. Ahmadinia. Dynamic interconnection of reconfigurable modules on reconfigurable devices. *IEEE Design and Test*, 22(5) :443–451, 2005.

[38] R. Vaidyanathan and J. L. Trahan. *Dynamic Reconfiguration : Architectures and Algorithms.* Series in Computer Science (Kluwer Academic/Plenum Publishers), 2004.

[39] T. Pionteck, C. Albrecht, and R. Koch. A dynamically reconfigurable packet-switched network-on-chip. In *Proc. of the conference on Design, automation and test in Europe (DATE 06')*, pages 136–137, 2006.

[40] B. Ahmad and T. Arslan. Dynamically reconfigurable NoC for reconfigurable MPSoC. In *Proc. of the Custom Integrated Circuits Conference (CCIC)*, pages 277– 280, 2005.

[41] L. Benini and G. DeMicheli. Networks on Chips : A New Paradigm for Component-Based MPSoC Design, 2002.

[42] P. T. Wolkotte, G. J. M. Smit, and J. E. Becker. Energy-Efficient NoC for Best-Effort Communication. In *Proc. of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 197–202, 2005.

[43] S.J.E. Wilton, N. Kafafi, B. Mei, and S. Vernalde. Interconnect architectures for modulo-scheduled coarse-grained reconfigurable arrays. In *Proc. of the IEEE International Conference on Field-Programmable Technology (FPT)*, pages 33–40, 2004.

[44] K. Pagiamtzis and A. Sheikholeslami. Content-addressable memory (CAM) circuits and architectures : A tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3) :712–727, March 2006.

[45] N. Dutt and P. Mishra. Architecture description languages for programmable embedded systems. *IEE Proc. : Computers and Digital Techniques*, pages 285–297, 2005.

[46] W. Qin and S. Malik. *The Compiler Design Handbook : Optimizations & Machine Code Generation*, chapter Architecture Description Languages for Retargetable Compilation, pages 535–564. CRC Press Y. N. Srikant and Priti Shankar, 2002.

[47] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. The MIMOLA Language - Version 4.1. Technical report, 1994.

[48] O. Karatsu. UDL/I standardization effort another approach to HDL standard. In *Proc. of the Euro ASIC Conference*, pages 388–393, 1991.

[49] A. Fauth, J. V. Van Praet, and M. Freericks. Describing instruction set processors using nml. In *Proc. of the European conference on Design and Test*, Washington, DC, USA, 1995.

[50] F. Charot and V. Messé. A flexible code generation framework for the design of application specific programmable processors. In *Proc. of the seventh international workshop on Hardware/software codesign (CODES '99)*, pages 27–31, New York, NY, USA, 1999.

[51] A. Kupriyanov. MAML - An Architecture Description Language for Modeling and Simulation of Processor Array Architectures Part I. Technical report, Department of Computer Science 12, Hardware-Software Co-Design, University of Erlangen-Nüremberg, 2006.

[52] Xilinx. XC4000XLA/XV Field Programmable Gate Arrays. Technical report, Product Specification DS015 (v1.3) October 18, 1999.

[53] S. Pillement, O. Sentieys, and R. David. DART : a functional-level reconfigurable architecture for high energy efficiency. *EURASIP J. Embedded Syst. Vol.2008*, (1) :1–13, 2008.

[54] T. Ojanpera and R. Prasad. *Wideband CDMA For Third Generation Mobile Communication*. Artech House Publishers, 1998.

[55] J. Pistorius, M. Hutton, A. Mishchenko, and R. Brayton. Benchmarking Method and Designs Targeting Logic Synthesis for FPGAs. In *Proc. of the International Workshop on Logic and Synthesis (IWLS 07)*, 2007.

[56] I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. In *Proc. of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 21–30, New York, NY, USA, 2006.

[57] T. Saidi. *Architectures matérielles pour la technologie WCDMA étendue aux systèmes mulit-antennes*. Ph.d. thesis, University of Rennes 1, ENSSAT, July 2008.

[58] R. David, D. Chillet, S. Pillement, and O. Sentieys. A compilation framework for a dynamically reconfigurable architecture. In *Proc. of the International Conference on Field-Programmable Logic and Applications (FPL)*, pages 1058–1067, 2002.

[59] R. David, S. Pillement, and O. Sentieys. Energy-Efficient Reconfigurable Processsors. In C. Piguet, editor, *Low Power Electronics Design*, Computer Engineering, Vol 1. CRC Press, August 2004.

[60] R. Wilson. SUIF : An Infrastructure for Research on Parallelizing and Optimizing Compilers. Technical Report CA 94305-4055, Computer Systems Laboratory, Stanford University, May 1994.

Table des figures

0-1 Interaction entre les critères de flexibilité, de dynamicité de la reconfiguration et de performance (F-D-P) d'une architecture reconfigurable dynamiquement 3
0-2 Intégration de la plate-forme Mozaïc avec une plate-forme de conception hôte 4

1-1 Implémentation de l'équation logique $O_0 = (I_3 \cdot I_2 \cdot I_1) + (\bar{I}_3 \cdot \bar{I}_2 \cdot \bar{I}_1 \cdot \bar{I}_0) + (I_3 \cdot \bar{I}_2 \cdot \bar{I}_1 \cdot I_0)$ sur une PAL . . . 10
1-2 Fonctionnement d'une LUT . . . 11
1-3 Exemple d'application exécutée par méthode d'implémentation temporelle et spatiale . . . 13
1-4 Caractéristiques des deux types d'implémentation temporelle et spatiale . . . 14
1-5 Exemple de réseaux d'interconnexion global . . . 16
1-6 Exemple de réseau d'interconnexion point-à-point . . . 17
1-7 Exemple de réseaux d'interconnexion hiérarchique . . . 17
1-8 Exemple d'implémentation à pré-chargement . . . 19
1-9 Implémentation asynchrone de tâches sur une zone de reconfiguration . . . 20
1-10 Exemple d'implémentation à reconfiguration partielle . . . 20
1-11 Bloc logique de l'AT40K . . . 23
1-12 Interconnexions dans un bloc logique AT40K . . . 23
1-13 Bloc logique de type Virtex . . . 24
1-14 Architecture d'un *DataPath Reconfigurable* de DART . . . 25
1-15 Architecture *Systolic Ring* . . . 27
1-16 Architecture de WPPA et de ses WPPE . . . 28
1-17 Architecture DPGA . . . 28
1-18 Bloc logique du Lattice ispXPGA . . . 30

1-19 *Processing Element* de Piperench . . . 31
1-20 Phases de configurations et d'exécution de Piperench . . . 31
1-21 Architecture PicoGA du processeur reconfigurable XiRisc . . . 32
1-22 Architecture XPP . . . 33
1-23 Architecture Adres . . . 35
1-24 *Tile* composant l'architecture du NEC-DRP . . . 36

2-1 Vue simplifiée des différentes ressources qui constituent une architecture issue de la plate-forme MOZAÏC . . . 43
2-2 Modélisation des hiérarchies d'interconnexion . . . 44
2-3 Synoptique d'un DUCK simplifié . . . 46
2-4 Exemple de l'utilisation du réseau de DUCK dans un accélérateur matériel . . . 47
2-5 Phases de reconfiguration d'un DUCK et interaction avec une unité de traitement ou d'interconnexion . . . 48
2-6 Logigramme représentant les phases de reconfiguration à l'aide d'un DUCK . . . 48
2-7 Phases de reconfiguration d'une ressource par l'intermédiaire d'un DUCK en mode séquentiel . . . 49
2-8 Exemple d'implémentation d'un DUCK dédiée à une ressource reconfigurable séquentielle . . . 50
2-9 Phases de reconfiguration d'une ressource par l'intermédiaire d'un DUCK en mode parallèle . . . 50
2-10 Exemple d'implémentation d'un DUCK dédié à une ressource reconfigurable parallèle . . . 51
2-11 Phases de reconfiguration d'une ressource par l'intermédiaire d'un DUCK en mode adressage . . . 52
2-12 Exemple d'implémentation d'un DUCK dédié à une ressource reconfigurable par adresse . . . 52
2-13 Adaptation du DUCK à la reconfiguration de la ressource . . . 53
2-14 Synoptique d'une DyRIBox simplifiée . . . 56
2-15 Schéma synoptique d'une DyRIBox simple . . . 57
2-16 DyRIBox de type commutation par paquets . . . 57
2-17 Schéma synoptique d'un DyRIOBloc . . . 59
2-18 Schéma synoptique du contrôleur de reconfiguration . . . 60
2-19 Schéma synoptique du *CtxtScheduler* . . . 60
2-20 Schéma synoptique du *CtxtScheduler* à plusieurs niveaux de priorité d'interruptions . . . 61
2-21 Graphe *flowchart* du contrôleur de domaine . . . 63
2-22 Principe de la reconfiguration partielle . . . 64
2-23 Graphe *flowchart* du contrôleur de mémoire de configurations . . . 65

2-24 Schéma synoptique du contrôleur de mémoire de configuration multi-zones . . 66
2-25 Organisation de la mémoire de configuration 67
2-26 Trame de configuration 67

3-1 Plate-forme de développement MOZAÏC et son environnement 70
3-2 Caractéristiques d'une description d'architecture en MAML 73
3-3 Paramètres nécessaire à la description d'une architecture au niveau *Array Level* 74
3-4 Exemple de domaines d'interconnexion 75
3-5 Exemple d'implémentation d'un domaine au niveau des interconnexions . . . 75
3-6 Paramètres nécessaires à la description d'un élément processeur 75
3-7 Paramètres nécessaires à la description d'une architecture en xMAML 76
3-8 Paramètres nécessaires à la description d'une DyRIBox 77
3-9 Exemple de description de type *AdjacencyMatrix* 79
3-10 Exemple de DyRIBox 80
3-11 Paramètres nécessaires à la description d'un Domaine 81
3-12 Exemple de droites délimitant les bornes extrêmes du polyèdre définissant l'espace du domaine 84
3-13 Paramètres nécessaires à la description d'un PEInterface 85
3-14 Paramètres nécessaires à la description d'un DyRIOBloc 87
3-15 Exploration de la surface et de la consommation induite par les DUCK en mode séquentiel 89
3-16 Exploration de la surface et de la consommation induite par les DUCK en mode adressage 89
3-17 Exploration de la surface et de la consommation induite par les DUCK en mode parallèle 90
3-18 Influence de la taille des données et du nombre de connexions par sortie sur la surface de silicium (a) et sur la consommation (b) 91

4-1 Synoptique d'un émetteur WCDMA de terminal mobile 97
4-2 Synoptique d'un récepteur WCDMA de terminal mobile 97
4-3 Synoptique d'un estimateur de trajets basé sur le *Power Delay Profile* 99
4-4 Synoptique d'un *finger* 99
4-5 Synoptique d'un *Rake Receiver* comportant *L fingers* 100
4-6 Schéma logique d'un CLB du XC4000 101
4-7 Plan des connexions sur XC4000 101
4-8 Plate-forme de développement MOZAÏC et son environnement 102
4-9 Conversion d'un routage type VPR vers un routage type DyRIBox 105
4-10 Ordonnancement des reconfigurations du décodeur WCDMA sur FPGA . . . 106

4-11 Schéma synoptique des connexions entre DyRIBox et CLB du eFPGA généré 107
4-12 Diagramme de Gantt des processus de reconfiguration 112
4-13 Schéma synoptique de l'implémentation des domaines pour WCDMA sur eFPGA 113
4-14 Plate-forme de développement MOZAÏC et interaction avec le flot de développement DART 114
4-15 Architecture de DART 115
4-16 Architecture d'un *cluster* de DART 116
4-17 Architecture d'un *DataPath Reconfigurable* de DART 116
4-18 Schéma synoptique d'une implémentation de décodeur WCDMA sur DART . 120
4-19 Ordonnancement des fonctions du décodeur WCDMA sur DART 120
4-20 Exploitation du parallélisme de tâche pour l'implémentation du filtre FIR . . 121
4-21 Schéma synoptique du décodage des données d'un terminal mobile 122
4-22 Schéma synoptique de l'implémentation d'un *finger* sur un DPR 122
4-23 Graphe flot de données du traitement de la synchronisation à la fréquence symbole 123
4-24 Schéma synoptique du traitement d'estimation de canal 124
4-25 Temps d'implémentation et de reconfiguration des tâches du décodeur WCDMA sur DART 125
4-26 Schéma synoptique de la DyRIBox de DPR de DART 125
4-27 Schéma synoptique de la DyRIBox de *cluster* de DART 127

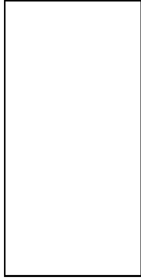
Liste des tableaux

3-1 Résultats de synthèse de DyRIBox et comparaison avec la méthode du projet 4S. 91

4-1 Nombre de CLB nécessaires à l'implémentation du module WCDMA par la méthode reconfigurable dynamiquement. 107
4-2 Tailles des configurations pour un CLB de la famille XC4000S 108
4-3 Tailles des configurations pour les 1235 CLB et 1235 DyRIBox du eFPGA . . 111
4-4 Résumé des différentes implémentations 114
4-5 Répartition des bits de configuration par ressource de DPR. 130
4-6 Répartition des configurations DUCK pour un *cluster* de DART. 130
4-7 Comparaison et impact de l'introduction du DUCK sur la surface des unités fonctionnelles. 132
4-8 Comparaison et impact de l'introduction du DUCK sur la consommation des unités fonctionnelles. 132

Listings

2-1 Exemple de boucles en C : finie et indéterminée 62
3-1 Exemple de description MIMOLA d'une ALU 71
3-2 Exemple de description nML d'une ALU 72
3-3 Exemple de description ARMOR 73
3-4 Description en xMAML de la DyRIBox de la figure 3-10 80
3-5 Exemple de description d'un domaine 84
3-6 Exemple de description d'une interface d'unité de traitement de type CLB . . 86
3-7 Description d'un DyRIOBloc 87
4-1 Description d'une LUT et d'une bascule au format BLIF 102
4-2 Description d'une LUT au format T-VPACK 103
4-3 Description du fichier de placement VPR 103
4-4 Description du fichier de routage VPR 104
4-5 Description xMAML d'une DyRIBox XC4000 109
4-6 Description xMAML d'un CLB XC4000 110
4-7 Description xMAML de la DyRIBox de DPR 126
4-8 Description xMAML de la DyRIBox de Cluster 126
4-9 Description de l'interface des générateurs d'adresses de DART dans Mozaïc . 127
4-10 Description de l'interface des unités fonctionnelles "1" et "3" de DART dans Mozaïc 128
4-11 Description de l'interface des unités fonctionnelles "2" et "4" de DART dans Mozaïc 128
4-12 Description de l'interface des registres de garde de DART dans Mozaïc . . . 129
4-13 Description de l'interface des registres de garde de DART dans Mozaïc . . . 129



Glossaire

ADL : *Architecture Description Language*, 70

Adres : *Architecture for Dynamically Reconfigurable Embedded Systems*, 34

AG : *Address Generator*, 126

ALU : *Arithmetic and Logical Unit*, 34

ASIC : *Application Specific Integrated Circuit*, 1

BCD : Décimal Codé Binaire, 9

BLIF : *Berkeley Logic Interchange Format*, 102

CAG : Contrôleur Automatique de Gain, 98

CAM : *Content Access Memory*, 57

CGRA : *Coarse Grain Reconfigurable Array*, 34

CLB : Configurable Logic Block, 15

CRC : *Cyclic Redundancy Checking*, 39

DAGGER : *Democritus University of Thrace eFPGA bitstream generator*, 39

DFT : *Design For Test*, 46

DIVINER : *Democritus University of Thrace RTL Synthesizer*, 39

DPCCH : *Dedicated Physical Control CHannel*, 96

DPDCH : *Dedicated Physical Data CHannel*, 96

DPGA : *Dynamically Programmable Gate*, 28

DPR : *DataPath Reconfigurable*, 25, 115

Dresc : *Dynamically Reconfigurable Embedded System Compiler*, 36

DRUID : *DemocRitus University of Thrace*, 39

DSP : *Digital Signal Processor*, 9

DUCK : *Dynamic Unifier and reConfiguration blocK*, 45

DUTYS : *Democritus University of Thrace Architecture file generator synthesizer*, 39

DyRIBox : *Dynamically Reconfigurable Interconnection Box*, 53

DyRIOBloc : *Dynamically Reconfigurable Input Output Block*, 58

E^{2}CMOS : *Electrically Erasable Complementary Metal Oxide Semiconductor*, 29

EDA : *Electronic Design Automation*, 2

EDVAC : *Electronic Discrete Variable Automatic Computer*, 9

ENIAC : *Electronic Numerical Integrator and Computer*, 9

FIR : *Finite Impulse Response*, 98

FPGA : *Field Programmable Gate Array*, 2

FU : *Fonctionnal Units*, 34

GAL : *Generic Array Logic*, 10

IIR : *Infinite Impulse Response*, 123

IP : *Intellectual Property*, 37

IW : *Interconnect Wrapper*, 74

LUT : *Look Up Table*, 11

Madeo-Bet : Madeo *Back-End Tool*, 38

Madeo-Fet : Madeo *Front-End Tool*, 38

MAML : *MAchine Marked up Language*, 73

MIT : *Massachusetts Institute of Technology*, 28

MMACS : *Million Multiply Accumulate Cycles per Second*, 98

NEC DRP : *NEC Dynamically Reconfigurable Processor*, 34

NOC : *Network On Chip*, 54

OVSF : *Orthogonal Variable Spreading Factor*, 97

PAL : *Programmable Array Logic*, 10

PDP : *Power Delay Profile*, 98

PiCoGa : *Pipelined Configurable Gate-Array*, 31

QPSK : *Quadrature Phase Shift Keying*, 97

RAM : *Random Access Memory*, 20

RISC : *Reduced Instruction Set Computer*, 30

SCMD : *Single Configuration Multiple Data*, 117

SIMD : *Single Instruction Multiple Data*, 51, 115

SOC : *System On Chip*, 54

SRAM : *Static Random Access Memory*, 22

SUIF : *Stanford University Intermediate Format*, 117

SWP : *Sub-Word Processing*, 25

UAL : Unité Arithmétique et Logique, 1

Umass : *University of Massachusetts*, 38

VHDL : *Very High Speed Integrated Circuit Hardware Description Language*, 39

VLIW : *Very Large Instruction Word*, 27

VPR : *Virtual Place and Route*, 38

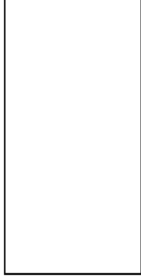
WCDMA : *Wideband Code Division Multiple Access*, 5

WP : *Word Processing*, 123

WPPA : *Weakly Programmable Processor Array*, 26

WPPE : *Weakly Programmable Processor Element*, 26

XPP : *eXtreme Processing Platform*, 32



Résumé

L'évolution constante des applications et le besoin toujours croissant de performances imposent le développement de nouvelles architectures compétitives et évolutives au sein de systèmes reconfigurables dynamiquement sur puces. Ces contraintes ont amené à une complexification des architectures, de leurs mécanismes de reconfiguration et de leur conception. De manière à répondre efficacement à ce problème, des plate-formes de développement ont été conçues et permettent ainsi d'automatiser certains processus constituant la chaîne de conception d'une architecture. Cela est rendu possible par l'intermédiaire d'un langage de description haut niveau (ADL) qui permet, par une spécification rapide de certains paramètres matériels, de procéder rapidement à la génération d'une architecture et de ses outils de développement adaptés tels que des outils de simulation, de compilation ou encore de synthèse. Cette thèse se place dans le contexte de la modélisation haut niveau des architectures ainsi que dans le contexte de l'aide à la conception et à l'exploration d'architectures reconfigurables dynamiquement. Ce document présente la plate-forme de développement Mozaïc dont l'objectif est de permettre la conception d'architectures reconfigurables dynamiquement par l'introduction automatique de ressources matérielles dédiées et adaptées. Dans une première partie, nous détaillons les concepts de reconfiguration dynamique qui ont été développés et mis en œuvre dans Mozaïc. Dans une deuxième partie, nous présentons le langage de description haut niveau xMAML qui permet la spécification de l'architecture et de l'exploitation efficace des mécanismes précédemment présentés. Ce langage est basé sur l'ADL MAML développé à l'université d'Erlangen, auquel nous avons ajouté certains paramètres de spécifications nécessaires à la mise en œuvre de la reconfiguration dynamique ainsi qu'à la spécification d'architectures hétérogènes. Enfin, dans un dernier chapitre, nous présentons les différentes phases de développement, et les outils associés, de deux architectures reconfigurables dynamiquement que sont les FPGAs et le processeur reconfigurable DART. Cette présentation inclut les phases d'exploration et l'implémentation d'un décodeur WCDMA par reconfiguration dynamique sur le FPGA modélisé par xMAML.

Mots clé : Reconfiguration dynamique, ASIC, FPGA, prototypage rapide, langage de description d'architectures, conception de circuits.

Abstract

Constant evolution of applications and an ever increasing need for performances make necessary the use of new dynamically reconfigurable systems on chip every times more highly capable and scalable. Consequently, these architectures constantly grow in complexity in terms of reconfiguration process and conception. The major issue was to design development frameworks based on high level architecture description languages (ADL). The ADLs are useful for the fast specification of hardware implementation and useful for giving architecture informations to the front-end tools. The area of this thesis is the architecture description, computer-aided design (CAD) and exploration of dynamically reconfigurable architectures. This document presents the development framework Mozaïc which aims at designing dynamically reconfigurable architecture by automatic generation of hardware resources needed. In the first part of this document, we detail the dynamic reconfiguration concepts developed and used by Mozaïc. In the second part, we present the ADL xMAML which permits the description and the efficient exploration of the concepts presented in the first part. This ADL is based on the MAML language developed at the University of Erlangen in Germany, to which was extended by adding new parameters required to make feasible dynamic reconfiguration of heterogeneous computing units. The last part of the document is dedicated to the presentation of the framework itself and of the various tools used to develop two reconfigurable architectures : FPGA and the DART. In particular is included the dynamic reconfiguration exploration and implementation of a WCDMA receiver on both architectures.

Keywords : Dynamic reconfiguration, FPGA, processing architecture, rapid prototyping, High Level Architecture Description Language (ADL), system design, adaptive architecture.