# Méthode de conception rapide d'architecture massivement parallèle sur puce : de la modélisation à l'expérimentation sur FPGA

Mouna Baklouti




École Nationale d'Ingénieurs de Sfax &
Université Lille 1 Sciences et Technologies

Thèse

présentée pour obtenir le titre de
docteur spécialité Informatique
par

Mouna BAKLOUTI

# Méthode de conception rapide d'architecture massivement parallèle sur puce : de la modélisation à l'expérimentation sur FPGA

**Composition de jury :**

| | | |
|---|---|---|
| ..... | ..... | **Président** |
| Ahmed Chiheb AMMARI | Maître de conférence INSAT | **Rapporteur** |
| Eric MARTIN | Professeur Université de Bretagne-Sud | **Rapporteur** |
| ..... | ..... | **Examinateur** |
| Jean-Luc DEKEYSER | Professeur Université Lille 1 | **Directeur** |
| Mohamed ABID | Professeur ENIS | **Directeur** |
| Philippe MARQUET | Maître de conférence Université Lille 1 | **Co-Directeur** |

# Table des matières

**Table des matières** iii

**Table des figures** vii

**Liste des tableaux** xi

**Glossaire** xiii

**1 Introduction** 1
- 1.1 Contexte . . . 4
- 1.2 Problématique . . . 8
- 1.3 Contributions . . . 11
- 1.4 Plan . . . 12

**2 État de l'art** 14
- 2.1 Applications de traitement de signal systématique et intensif . . . 15
- 2.2 Architectures SIMD pour le TSS . . . 17
  - 2.2.1 Modèle d'exécution SIMD . . . 18
  - 2.2.2 Motivations : efficacité de SIMD pour le TSS . . . 18
  - 2.2.3 SIMD embarqué . . . 19
- 2.3 Architectures SIMD sur FPGA . . . 20
  - 2.3.1 Intérêt des FPGA . . . 21
  - 2.3.2 Classification des architectures SIMD sur FPGA . . . 22
    - 2.3.2.1 Architectures SIMD non couplées . . . 23
    - 2.3.2.2 Architectures SIMD fortement couplées . . . 25
- 2.4 Conception et prototypage des architectures SIMD . . . 28
  - 2.4.1 Besoin de nouvelles méthodologies d'intégration . . . 29
  - 2.4.2 Intégration d'IP . . . 29
  - 2.4.3 Environnements de conception pour systèmes sur puce . . . 31
- 2.5 Quelle méthodologie d'assemblage ? . . . 33
- 2.6 Conclusion . . . 35

**3 Modèle mppSoC : architecture SIMD paramétrique et générique** 37
- 3.1 Définition du modèle mppSoC . . . 38
- 3.2 Description de l'architecture de mppSoC . . . 41
  - 3.2.1 Le processeur de contrôle : ACU . . . 41
  - 3.2.2 Interface ACU - PE . . . 41

3.2.3 Le réseau de PEs . . . 41
3.2.4 Le global routeur : mpNoC . . . 42
3.3 Analyse du modèle architectural de mppSoC . . . 44
3.3.1 Aspect modulaire . . . 44
3.3.2 Aspect paramétrique et flexible . . . 45
3.3.2.1 Paramétrisation de mppSoC . . . 45
3.3.2.2 Flexibilité de mppSoC . . . 46
3.3.3 Aspect programmable . . . 50
3.4 Validation du modèle mppSoC . . . 52
3.4.1 Algorithme de multiplication de matrices parallèle . . . 52
3.4.1.1 Étude théorique . . . 52
3.4.1.2 Étude expérimentale . . . 55
3.4.2 Algorithme de réduction : Somme de N valeurs sur P processeurs . . . 55
3.4.2.1 Étude théorique . . . 55
3.4.2.2 Étude expérimentale . . . 56
3.5 Comparaison du modèle mppSoC avec d'autres modèles SIMD . . . 57
3.6 Conclusion . . . 58

**4 Méthode de conception rapide et programmation de mppSoC 60**
4.1 Présentation de la méthode de conception de mppSoC . . . 61
4.1.1 Description de la conception de mppSoC . . . 61
4.1.2 Méthodologies de conception des processeurs . . . 63
4.1.2.1 Réduction de processeur . . . 64
4.1.2.2 Réplication de processeur . . . 65
4.1.3 Tâches du concepteur . . . 67
4.2 Bibliothèque d'IPs mppSoC . . . 68
4.2.1 Processeurs . . . 68
4.2.1.1 Choix de processeurs . . . 68
4.2.1.2 Processeur miniMIPS . . . 70
4.2.1.3 Processeur OpenRISC OR1200 . . . 76
4.2.1.4 Processeur NIOS II . . . 82
4.2.2 Mémoires . . . 87
4.2.3 Réseau de voisinage . . . 87
4.2.4 MpNoC . . . 90
4.2.4.1 Bus partagé . . . 93
4.2.4.2 Crossbar . . . 93
4.2.4.3 Réseau multi-étages de type Delta . . . 95
4.2.5 Classification des IPs . . . 97
4.3 Programmation de mppSoC . . . 97
4.3.1 Cas de réduction de processeur . . . 97
4.3.1.1 Processeur miniMIPS . . . 97
4.3.1.2 Processeur OpenRisc . . . 100
4.3.2 Cas de réplication de processeur . . . 101
4.3.3 Jeu d'instructions mppSoC . . . 102
4.4 Conclusion . . . 104

**5 Outil de conception de mppSoC : mise en œuvre et expérimentation sur FPGA 106**
5.1 Introduction à l'IDM et automatisation de la génération de code . . . . . 107
5.1.1 L'IDM pour la conception des systèmes embarqués . . . . . 108
5.1.2 Langage de modélisation UML . . . . . 109
5.1.3 Profil MARTE . . . . . 110
5.1.3.1 Package HRM . . . . . 110
5.1.3.2 Package GCM . . . . . 111
5.1.3.3 Package RSM . . . . . 112
5.2 Proposition d'une méthode d'utilisation de MARTE pour mppSoC . . . . . 113
5.2.1 Intégration dans Gaspard . . . . . 113
5.2.2 Modélisation de mppSoC . . . . . 115
5.2.2.1 Modélisation des composants élémentaires de mppSoC . . . 116
5.2.2.2 Modèle 1D . . . . . 118
5.2.2.3 Modèle 2D . . . . . 120
5.2.2.4 Déploiement en utilisant les IPs . . . . . 123
5.3 Chaîne pour la génération du code VHDL synthétisable . . . . . 125
5.3.1 Organisation de mppSoCLib . . . . . 125
5.3.2 Transformation depuis un modèle Deployed vers du code VHDL . . . 126
5.4 Expérimentation sur FPGA : traitement vidéo temps réel . . . . . 128
5.4.1 Plateforme de prototypage . . . . . 128
5.4.2 Chaîne de traitement vidéo à base de mppSoC . . . . . 129
5.4.3 Application de conversion de couleur . . . . . 133
5.4.3.1 Présentation de l'application . . . . . 133
5.4.3.2 Génération de mppSoC . . . . . 135
5.4.3.3 Mesure de performances . . . . . 137
5.4.4 Application de convolution . . . . . 138
5.4.4.1 Présentation de l'application . . . . . 138
5.4.4.2 Génération de mppSoC . . . . . 138
5.4.4.3 Mesure de performances . . . . . 140
5.5 Comparaison de mppSoC avec d'autres systèmes . . . . . 141
5.6 Conclusion . . . . . 143

**6 Conclusion et perspectives 145**
6.1 Bilan . . . . . 146
6.2 Perspectives . . . . . 147

**Publications personnelles 149**

**Bibliographie 151**

**A Implémentation VHDL de miniMIPS 159**

**B Fichiers de compilation OR32 168**

**C Codage des instructions MIPS32 172**

**D Description des outils de compilation du processeur miniMIPS 175**

**E IPs Mémoires de la bibliothèque mppSoC** 177

**F Écran LCD TRDB-LTM** 182

**Résumé/Abstract** 186

# Table des figures


1.1 *Prévisions de l'évolution des semi-conducteurs [52]* . . . 5
1.2 *Architecture SIMD* . . . 6
1.3 *Compromis performance/flexibilité* . . . 9

2.1 *Composition du TSI* . . . 16
2.2 *Représentation de l'application radar anti-collision* . . . 17
2.3 *Bloc diagramme d'un FPGA* . . . 21
2.4 *Architecture IMAPCar [59]* . . . 22
2.5 *Architecture du coprocesseur lié au processeur hôte* . . . 24
2.6 *Circuit Ter@Core [14]* . . . 25
2.7 *Architecture bloc de Morphosys (M2)* . . . 27
2.8 *Architecture WPPA et structure d'un WPPE [57]* . . . 31

3.1 *Modèle architectural de mppSoC* . . . 39
3.2 *Connexions de voisinage entre les PEs dans une topologie maille* . . . 42
3.3 *Schéma général de mpNoC* . . . 44
3.4 *Topologies du réseau de voisinage* . . . 46
3.5 *Types de réseaux d'interconnexion du mpNoC* . . . 48
3.6 *Classification des réseaux MINs* . . . 50
3.7 *Structure de commutateurs formant un réseau MIN-oméga* . . . 50
3.8 *Partitionnement sur un anneau de PEs* . . . 53
3.9 *Temps de simulation en variant le nombre de PEs* . . . 54
3.10 *Temps de simulation en variant la topologie du réseau de voisinage* . . . 55
3.11 *Somme de huit valeurs sur huit PEs* . . . 55
3.12 *Algorithme de réduction sur des configurations mppSoC avec 64 PEs* . . . 57

4.1 *Conception hiérarchique de mppSoC intégrant 4 PEs* . . . 62
4.2 *Principe de la méthodologie de réduction de processeur* . . . 64
4.3 *Principe de la méthodologie de réplication de processeur* . . . 66
4.4 *Pipeline miniMIPS à 5 étages* . . . 71
4.5 *Schéma bloc de miniMIPS* . . . 71
4.6 *Schéma bloc de l'ACU* . . . 72
4.7 *Réduction du processeur miniMIPS* . . . 72
4.8 *Schéma bloc du PE* . . . 73
4.9 *Connexion du module prédicteur de branchement* . . . 74
4.10 *Architecture du processeur OpenRISC* . . . 78
4.11 *Exemple d'une connexion Wishbone point-à-point au niveau RTL* . . . 78

4.12 *Réduction du piepline de l'OpenRISC* . . . 80
4.13 *Processeur embarqué NIOS d'Altera* . . . 83
4.14 *Configuration mppSoC à base de NIOS* . . . 84
4.15 *Interface de l'adaptateur connecté à l'ACU* . . . 85
4.16 *Interface de l'adaptateur connecté au PE* . . . 85
4.17 *FSM de l'adaptateur de l'ACU* . . . 86
4.18 *FSM de l'adaptateur du PE* . . . 86
4.19 *Architecture du routeur de voisinage* . . . 88
4.20 *Connexions entre les RC dans une topologie maille* . . . 89
4.21 *Architecture de l'unité élémentaire à base du processeur miniMIPS* . . . 90
4.22 *Architecture de mpNoC* . . . 91
4.23 *Intégration de mpNoC dans mppSoC* . . . 91
4.24 *Distribution des ressources de mpNoC* . . . 92
4.25 *Architecture du bus partagé* . . . 94
4.26 *Architecture du crossbar* . . . 95
4.27 *Types de réseaux Delta (8,8)* . . . 96
4.28 *Paquet de donnée dans le réseau Delta MIN* . . . 96
4.29 *Formats d'instructions MIPS32* . . . 97
4.30 *Programmation d'un code parallèle en assembleur miniMIPS étendu* . . . 99
4.31 *Génération du fichier de test "App.or32"* . . . 101

5.1 *Gap entre la conception du matériel et du logiciel [51]* . . . 108
5.2 *Principe de transformation de modèle* . . . 109
5.3 *Structure du profil HRM* . . . 111
5.4 *Vue globale du package GCM* . . . 112
5.5 *Le concept "Shaped" dans le package RSM* . . . 112
5.6 *Vue globale du package RSM* . . . 113
5.7 *L'environnement Gaspard* . . . 114
5.8 *Modélisation du processeur élémentaire* . . . 116
5.9 *Modélisation de la mémoire locale du PE* . . . 116
5.10 *Modélisation de l'ACU et ses mémoires* . . . 117
5.11 *Modélisation du réseau mpNoC* . . . 117
5.12 *Modélisation du périphérique d'entrée/sortie* . . . 118
5.13 *Modélisation du PU dans le cas d'un réseau de voisinage linéaire* . . . 118
5.14 *Configuration mppSoC intégrant un réseau de voisinage linéaire* . . . 119
5.15 *Spécification du nombre de PE* . . . 119
5.16 *Stéréotype InterRepetition* . . . 119
5.17 *Stéréotype Reshape* . . . 119
5.18 *Modélisation du PU dans le cas d'un réseau de voisinage de type maille* . . . 120
5.19 *Configuration mppSoC intégrant un réseau de voisinage de type maille* . . . 121
5.20 *Liaison Est/Ouest* . . . 121
5.21 *Liaison Nord/Sud* . . . 121
5.22 *Connecteur Reshape* . . . 122
5.23 *Modélisation du PU dans le cas d'un réseau de voisinage de type Xnet* . . . 122
5.24 *Configuration mppSoC intégrant un réseau de voisinage de type Xnet* . . . 122
5.25 *Connecteur Nord/Sud* . . . 123
5.26 *Connecteur Est/Ouest* . . . 123

5.27 *Connecteur diag11/diag12* . . . 123
5.28 *Connecteur diag21/diag22* . . . 123
5.29 *Déploiement d'un IP pour le PE* . . . 124
5.30 *Le concept "CodeFile"* . . . 124
5.31 *Chaîne mppSoC intégrée dans l'environnement Gaspard* . . . 125
5.32 *Organisation de mppSoCLib* . . . 126
5.33 *La plateforme DE2 70* . . . 128
5.34 *Chaîne de traitement vidéo à base de mppSoC intégrant 4 PEs* . . . 130
5.35 *Influence du réseau d'interconnexion sur le temps d'exécution* . . . 131
5.36 *Résultats de simulation pour une image RGB de taille 800x480* . . . 132
5.37 *Partitionnement de pixels entre les PEs (cas de 4 PEs)* . . . 135
5.38 *Spécification du nombre de PE* . . . 135
5.39 *Spécification de la taille des mémoires* . . . 136
5.40 *Spécification de la méthodologie de conception* . . . 136
5.41 *Spécification du type de processeur* . . . 136
5.42 *Spécification du réseau d'interconnexion de mpNoC* . . . 137
5.43 *Résultats de simulation de la conversion RGB à YIQ* . . . 137
5.44 *Spécification du nombre de processeurs élémentaires* . . . 139
5.45 *Modélisation du processeur dans le cas de réplication* . . . 139
5.46 *Choix du type de processeur* . . . 139
5.47 *Résultats de simulation de la convolution* . . . 140

D.1 *Étapes d'assemblage* . . . 176

F.1 *Spécification des signaux de l'écran dans la direction horizontale* . . . 183
F.2 *Paramètres des signaux de l'écran dans la direction horizontale* . . . 183
F.3 *Spécification des signaux de l'écran dans la direction verticale* . . . 184
F.4 *Paramètres des signaux de l'écran dans la direction verticale* . . . 184

# Liste des tableaux


1.1 Différences de fonctionnement des systèmes SIMD et MIMD . . . . . 7

3.1 Caractéristiques des topologies régulières extensibles . . . . . 47
3.2 Comparaison entre les réseaux : bus, crossbar et MIN . . . . . 49
3.3 Caractéristiques de modèles SIMD . . . . . 58

4.1 Comparaison entre processeurs . . . . . 69
4.2 Interface du processeur miniMIPS . . . . . 71
4.3 Ressources occupées par l'ACU et le PE dans le cas de réduction du miniMIPS 75
4.4 Ressources occupées par l'ACU et le PE dans le cas de réplication du miniMIPS 77
4.5 Signaux requis pour intégrer l'OpenRisc dans mppSoC . . . . . 79
4.6 Ressources occupées par l'ACU et le PE dans le cas de réduction de l'OpenRisc 81
4.7 Ressources occupées par l'ACU et le PE dans le cas de réplication de l'OpenRisc 82
4.8 Occupation FPGA des mémoires . . . . . 87
4.9 Performances du bus partagé . . . . . 94
4.10 Performances du crossbar . . . . . 95
4.11 Performances du réseau Delta MIN . . . . . 96
4.12 Décodage de OPCOD . . . . . 98
4.13 Macros mppSoC . . . . . 105

5.1 Résultats de synthèse dans le cas de la réduction . . . . . 130
5.2 Résultats de synthèse dans le cas de la réplication . . . . . 131
5.3 Résultats de synthèse . . . . . 140
5.4 Comparaison des temps d'exécution . . . . . 141
5.5 Comparaison entre configurations SIMD et accélérations C2H . . . . . 142
5.6 Comparaison entre configuration mppSoC et autres systèmes embarqués . . . 142
5.7 Comparaison entre configuration mppSoC et autres systèmes embarqués . . . 143

C.1 Cas OPCOD = SPECIAL, décodage de FUNC . . . . . 172

C.2 Cas OPCOD = SPECIALP, décodage de FUNC . . . . . 173
C.3 Cas OPCOD = P_MEM . . . . . 173
C.4 Cas OPCOD = BCOND . . . . . 173
C.5 Cas OPCOD = COPRO . . . . . 173
C.6 Extension du jeu d'instructions . . . . . 174

# Glossaire

| | |
|---|---|
| ACU | Array Controller Unit |
| ALM | Adaptive Logic Module |
| ALU | Arithmetic Logic Unit |
| ALUT | Adaptive Look-Up Table |
| ASIC | Application Specific Integrated Circuit |
| CMP | Chip Multi-Processor |
| DAP | Distributed Array Processors |
| DSP | Digital Signal Processor |
| FIFO | First In First Out |
| FPGA | Field Programmable Gate Arrays |
| GASPARD | Graphical Array Specification for Parallel and Distributed Computing |
| GCM | Generic Component Model |
| GPU | Graphical Processing Unit |
| HRM | Hardware Resource Model |
| IDM | Ingénierie Dirigée par les Modèles |
| IP | Intellectual Property |
| ISS | Instruction Set Simulator |
| ITRS | International Technology Roadmap for Semiconductors |
| MARTE | Modeling and Analysis of Real-Time Embedded Systems |
| MDA | Model Driven Architecture |
| MIN | Multistage Interconnetion Network |
| MMP | Massively MultiProcessor |
| MMX | MultiMedia eXtensions |
| MOPS | Million d'Opération Par Seconde |
| mpNoC | massively parallel Network on Chip |
| mppSoC | massively parallel processing System on Chip |
| NoC | Network on Chip |

| | |
|---|---|
| OMG | Object Management Group |
| PE | Processeur Élémentaire |
| PLD | Programmable Logic Device |
| PU | Processing Unit |
| RAM | Random Access Memory |
| RISC | Reduced Instruction Set Computer |
| RSM | Repetitive Structure Modeling |
| RTL | Register Transfer Level |
| SIMD | Single Instruction Multiple Data |
| SoC | System on Chip |
| SOPC | System On Programmable Chip |
| SSE | Streaming SIMD Extensions |
| TDI | Traitement de Données Intensif |
| TS | Traitement de Signal |
| TSI | Traitement de Signal Intensif |
| TSS | Traitement de Signal Systématique |
| TTM | Time To market |
| UML | Unified Modeling Language |
| VHDL | VHSIC Hardware Description Language |
| VLIW | Very Long Instruction Word |

# Chapitre 1

# Introduction


1.1 Contexte . . . . . 4
1.2 Problématique . . . . . 8
1.3 Contributions . . . . . 11
1.4 Plan . . . . . 12

Plus rapides, plus puissants sont les caractéristiques des systèmes électroniques qui envahissent notre vie quotidienne et professionnelle. Ces systèmes poursuivent une intense évolution et sont largement influencés par les progrès de l'industrie des semi-conducteurs.

### Évolution de la technologie des semi-conducteurs

En 1965, le Dr. Gordon E. Moore, cofondateur d'Intel, avait prédit que les capacités d'intégration sur les technologies silicium augmenteraient de manière exponentielle, voir doubleraient tous les deux ans [73]. Cette loi stipulait la révolution technologique qui a commencé il y a plusieurs décennies et continue jusqu'à nos jours. Cette augmentation des densités d'intégration des puces, suite à l'amélioration des technologies de fabrication, a mis en évidence les limites des méthodologies de conception traditionnelles. Il était donc nécessaire de trouver de nouvelles méthodes de conception plus évolutives qui permettront l'implémentation de larges et complexes systèmes.

### Apparition des systèmes sur puce

Un système embarqué est un système électronique dédié à des applications spécifiques. Il est souvent optimisé en termes de coût, performance, consommation d'énergie et fiabilité. Les systèmes embarqués sont omniprésents dans notre vie quotidienne (automobiles, téléphone portable, transport ferroviaire, aéronautique, etc.). Les premiers systèmes embarqués apparus étaient composés de plusieurs éléments intégrés sur un circuit. Avec l'évolution incessante des technologies de fabrication, conformément à la loi de Moore, l'augmentation des capacités d'intégration des puces ont rendu possible et facile l'intégration de plusieurs composants [73] sur une seule puce. L'ère des systèmes sur puce (*SoC*) a alors vu le jour.

### Intégration des SoC

La méthodologie de conception de SoC offre la possibilité d'intégrer divers composants assurant le calcul, le stockage et d'autres fonctionnalités dans un même système afin d'améliorer la performance, réduire la consommation d'énergie et minimiser les coûts. Contrairement à la conception des circuits spécifiques tels que les ASICs, aujourd'hui les concepteurs s'orientent vers la réutilisation de blocs prédéfinis, nommés propriétés intellectuelles (*IP*), ce qui réduit considérablement le temps de mise sur le marché.

En suivant la loi de Moore, les systèmes embarqués ont évolué de simples petits systèmes composés de quelques blocs d'IP spécialisés, à des systèmes intégrant de multiples processeurs faisant naissance aux systèmes parallèles sur puce, ou systèmes multi-Processeur sur puce (*CMP*). En effet, pendant près de 40 années, les innovations technologiques se sont succédées dans le but de réduire les temps d'exécution. Certaines ont consisté à réduire le temps de traitement et à améliorer les fréquences de fonctionnement. Ces techniques sont néanmoins limitées par les possibilités physiques d'intégration. D'autres ont tenté d'augmenter le débit de traitement des instructions, c'est-à-dire le nombre de traitements effectués par unité de temps s'appuyant de plus en plus sur les architectures parallèles. Ces architectures consistent, en fait, à segmenter le traitement des instructions ou à accroître le nombre de ressources pour exécuter plus d'instructions simultanément. Dans ce contexte, les systèmes parallèles sur puce se basent sur la multiplication du nombre de ressources de calcul travaillant en parallèle et intégrées sur une même puce afin de satisfaire les applications

récentes en termes de vitesse de calcul et performances. De ce fait, on assiste à une augmentation croissante du nombre de processeurs dans les systèmes actuels visant, au travers un recours au parallélisme massif, à satisfaire les besoins des applications en vitesse de calcul.

### Évolution des applications

Corrélativement, les algorithmes des applications de traitement de signal intensif, telles que les applications de télédétection (radar, sonar, etc.), et de l'image deviennent de plus en plus sophistiqués. Ils mettent en jeu un volume considérable de données. Ce domaine d'applications requiert une grande puissance de calcul, que seules les architectures parallèles spécialisées *taillées sur mesure* peuvent satisfaire les contraintes de vitesse, de performance et d'encombrement [68]. La parallélisation et les architectures parallèles semblent être la voie la plus prometteuse pour répondre à la complexité applicative.

### Vers les architectures parallèles

Dans ce contexte, un tour d'horizon sur l'évolution des architectures parallèles s'avère nécessaire. Les besoins croissants des grandes applications scientifiques et militaires ont été le principal moteur économique du développement de ces nouvelles architectures. Les premières machines de calcul intensif, les machines vectorielles, sont apparues au milieu des années 70, notamment avec la création de Cray Research en 1974. Le calcul numérique intensif était donc l'apanage de ces machines vectorielles, et réservé aussi aux super-ordinateurs vectoriels. Plusieurs machines parallèles avec des caractéristiques diverses ont été aussi proposées et exploitées dans les années 80 à savoir Maspar [13], Connection Machine [47], etc. Leur coût technologique au niveau du développement des architectures a freiné leur fabrication d'une façon systématique. Depuis, la technologie des circuits a progressé à un tel point qu'il est devenu possible de fabriquer aujourd'hui des systèmes complexes nécessitant des millions de transistors sur une seule puce. Cette progression est observée à travers la migration remarquable des sociétés de microprocesseurs (AMD, IBM, Intel, Sun) vers les designs multi-cœurs. Ces grands constructeurs s'engagent de plus en plus dans l'exploitation du parallélisme spatial qui avait été abandonné préalablement au profit d'augmentation des fréquences de fonctionnement. Ainsi, pour proposer plus de puissance de calcul, ils misent maintenant sur la multiplication des cœurs homogènes ou hétérogènes de processeurs au sein d'une même puce [43]. Citons le processeur CELL [18] conçu conjointement par IBM, Sony et Toshiba qui est révélé en février 2005. Il équipe notamment la console de jeu vidéo PlayStation 3 de Sony. Son architecture est basée sur un processeur PowerPC et 8 unités SIMD. Intel a aussi développé récemment une puce intégrant 80 cœurs [102]. Cette puce conçue peut atteindre une fréquence de 5.7 GHz et ayant une performance de 1.81 Téra-FLOPS (mille milliards d'opérations flottantes par seconde). Parallèlement à cette évolution, des architectures massivement parallèles sont apparues et commencent à tirer une grande partie du marché des systèmes embarqués. À titre d'exemple, citons le processeur de traitement graphique spécialisé (*GPU*) comprenant plusieurs unités de traitement.

On voit alors clairement que l'exploitation du parallélisme dans les systèmes actuels connaît de nos jours un bond spectaculaire. Les systèmes sur puce suivent cette même évolution en intégrant plusieurs processeurs favorisant un traitement parallèle. Ces systèmes répondent en parti à l'augmentation continuelle des puissances de calcul requises par les applications multimédia. Tout cela a un impact conséquent sur la conception des SoC qui

fait face aujourd'hui à de nombreux défis, à savoir : un temps de mise sur le marché très court, le coût de la conception, la complexité du silicium, la productivité, l'adéquation de l'architecture avec l'application, etc. C'est dans ce contexte que se situent les travaux de cette thèse que nous présenterons dans la section suivante.

## 1.1 Contexte

#### Domaine d'applications : TSS embarqué

Nos travaux de thèse s'inscrivent dans le cadre du traitement de signal intensif (TSI). Le TSI est en effet le sous-ensemble du traitement du signal le plus gourmand en calcul. Les applications de TSI (télécommunications, traitement multimédia, etc.) demandent une grande puissance de calcul nécessaire à leur exécution. Elles doivent souvent répondre à des contraintes de temps afin d'assurer une exécution temps réel et sont souvent embarquées. Des contraintes de taille et de consommation énergétiques se superposent alors aux contraintes de temps. Le TSI se compose de deux parties. Une partie de traitement régulier appelée traitement de signal systématique (TSS) qui consiste à réaliser des calculs sur les signaux indépendamment de leurs valeurs. Cette partie est généralement suivie d'une partie de traitement de données intensif (TDI) plus dépendante de la valeur des données. Dans notre travail, nous nous intéresserons particulièrement aux applications de TSS. À titre d'exemple on peut citer les algorithmes de filtrage dont les données d'entrées proviennent de radar ou de sonar (détection d'une cible, caractérisation de sa position, sa vitesse, etc.) et les algorithmes de traitement d'images. Le TSS se caractérise par une manipulation de grandes quantités de données traitées de façon régulière et/ou de grands calculs systématiques à effectuer. Ces applications sont principalement développées sur des systèmes embarqués avec des unités de calcul à hautes performances telles que des processeurs de traitement de signal (*DSP*) ou des unités parallèles SIMD.

#### Les architectures parallèles au service des applications modernes

Dans ce même contexte, les architectures parallèles, basées sur un grand nombre de processeurs intégrés, sont de plus en plus utilisées pour satisfaire les nouvelles applications embarquées. Ces applications contiennent des fonctionnalités très gourmandes en terme de puissance de calcul et comprennent des algorithmes de traitement de données intensif (traitement d'image, multimédia, infographie, etc.) qui sont en essence parallèles et répétitifs [11, 24, 21]. Leur demande croissante en puissance de calcul se marie parfaitement à la mise à disposition de puces électroniques ayant des architectures massivement parallèles avec des capacités de plus en plus grandes. En effet, les systèmes parallèles sur puce sont particulièrement adaptés aux applications embarquées qui comportent une partie de traitement intensif du signal ou de l'image. Face à l'importance du flot de données à manipuler et à la diversité des opérations rencontrées, la multiplication des ressources de calcul constitue une réponse naturelle pour atteindre des performances satisfaisantes. L'augmentation des capacités d'intégration sur puce favorisent bien cette évolution.

### Augmentation de la capacité d'intégration des SoC

Suivant avec régularité la feuille de route technologique internationale pour les semi-conducteurs (ITRS) [52] (figure 1.1), le nombre de cœurs de calcul et la taille de la mémoire intégrés sur une seule puce augmentent constamment (avec la capacité d'intégration de cent cœurs sur une seule puce pronostiquée pour l'année 2010). En réalité, les constructeurs de matériels embarqués sont déjà à l'étude de solutions massivement multi-processeurs (*MMP*), articulant plusieurs processeurs identiques autour d'un réseau sur puce (*NoC*). Aujourd'hui, les concepteurs de systèmes embarqués comptent sur des architectures systèmes sur puce multi-processeurs parallèles pour fournir le niveau de performance demandé par les applications. La tendance s'oriente soit vers l'utilisation massive de processeurs standard soit vers l'utilisation des architectures parallèles.

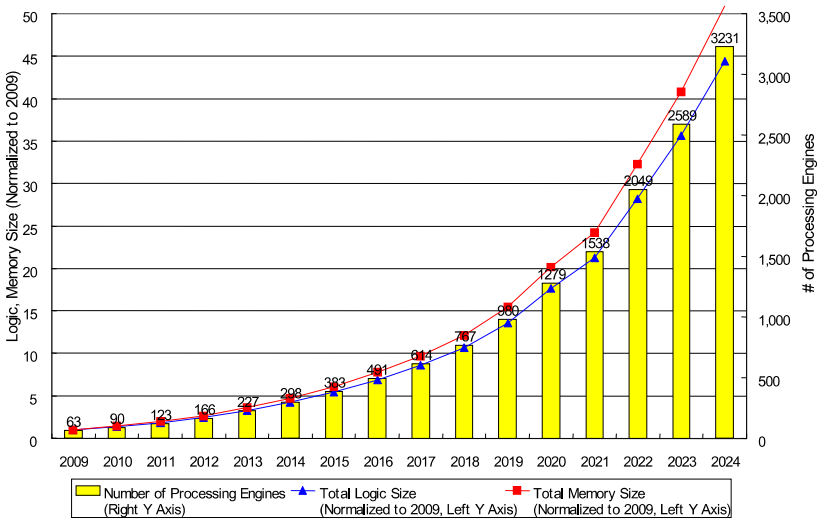



FIGURE 1.1 – *Prévisions de l'évolution des semi-conducteurs [52]*

### Classification de Flynn

Différentes classifications ont été proposées afin de caractériser les types des architectures existantes. La classification la plus connue est celle de Flynn [38]. Elle caractérise les machines selon leur mode de fonctionnement, c'est-à-dire la multiplicité des flots de contrôle et de données. Il en découle quatre catégories qui sont listées ci dessous :

- SISD : Single Instruction stream Single Data stream : exécution séquentielle ;
- MISD : Multiple Instruction stream Single Data stream : exécution sur un flot de données (pipelinée) ;
- SIMD : Single Instruction stream Multiple Data stream : exécution synchrone d'une même instruction par plusieurs processeurs sur différentes données ;
- MIMD : Multiple Instruction stream Multiple Data stream : exécution sur plusieurs processeurs avec chacun son flot de contrôle.

La majorité des architectures parallèles appartiennent à la catégorie SIMD ou MIMD [92]. Les architectures MIMD sont les architectures multiprocesseurs composées d'un nombre de processeurs connectés à travers un réseau d'interconnexion. Les processeurs opèrent de ma-

nière asynchrone chacun exécutant une tâche propre sur un ensemble de données locales ou globales. Les architectures MIMD se divisent en deux groupes : architectures à mémoire partagée et architectures à mémoire distribuée [38].

Dans les architectures SIMD, nous trouvons les processeurs vectoriels pipelinés et les tableaux de processeurs (*Arrays*). Les processeurs vectoriels exploitent la régularité des traitements sur les éléments d'un vecteur (tableau linéaire de nombres). Ce type de processeurs est couramment utilisé sur les supercalculateurs. Dans tout ce qui suit, nous nous intéresserons aux tableaux de processeurs. Ces derniers sont caractérisés par un contrôle centralisé et des données distribuées. Les processeurs d'une machine de ce type n'ont pas de séquenceur et sont juste des éléments simples de calcul appelés processeurs élémentaires (PE). Chacun possède une mémoire de données locale et reçoit ses instructions d'un unique séquenceur de sorte que tous les processeurs exécutent la même instruction de manière synchrone [38].

**Comparaison SIMD/MIMD : efficacité de SIMD pour le TSS**

Les systèmes MIMD peuvent exécuter des instructions différentes sur des données multiples. Ces processeurs parallèles sont alors intéressants pour l'exploitation de processus disjoints comme le parallélisme au niveau tâche. Quant aux architectures SIMD, il s'agit de processeurs parallèles permettant d'effectuer des traitements identiques sur différentes données favorisant ainsi un parallélisme de données inhérent aux applications de TSS. De point de vue intégration sur puce, les architectures MIMD sont plus coûteuses en comparaison avec les architectures SIMD. En effet dans ces architectures, il s'agit de plusieurs flots d'instructions et donc plusieurs décodeurs ce qui consomme beaucoup de puissance et de surface sur la puce. Le contrôle de l'application pose un problème majeur pour ces architectures. En effet, le temps global d'exécution d'un algorithme parallèle peut être très significativement dégradé par une mauvaise utilisation des ressources ou mauvaise répartition des charges de calcul. L'avantage des architectures SIMD tient à ce que seules les unités de calcul doivent être dupliquées. En effet, une seule unité de contrôle suffit à décoder les instructions pour tous les processeurs. Sur ce type d'architectures les instructions traitant des données parallèles sont diffusées à tous les processeurs pour être appliquées aux données réparties dans les mémoires locales de ces processeurs, telle qu'illustrée dans la figure 1.2. Ainsi, tous les processeurs effectuent d'une manière synchrone la même instruction, mais sur des données différentes. Vu leurs caractéristiques, les architectures SIMD sont plus faciles à program-

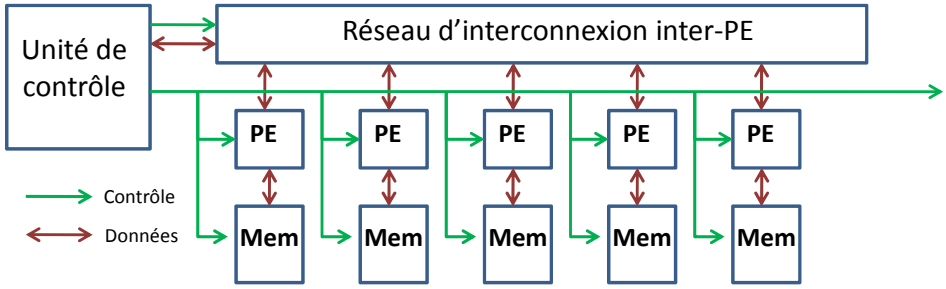



FIGURE 1.2 – *Architecture SIMD*

mer. Du fait du haut niveau de synchronisme, les interactions sur les architectures SIMD sont faciles à mettre en œuvre et implantées de façon efficace. Le tableau 1.1 illustre trois points majeurs faisant la différence entre les systèmes SIMD et MIMD. D'après ce tableau,

TABLE 1.1 – Différences de fonctionnement des systèmes SIMD et MIMD

| | SIMD | MIMD |
|---|---|---|
| Opérations à un instant donné | Identiques | Différentes |
| Adresses de mémoires | Identiques | Différentes |
| Réception des valeurs des voisins | Au même instant | Indépendante |

nous déduisons que les machines MIMD sont adéquates pour les algorithmes parallèles irréguliers, alors que les machines SIMD répondent efficacement aux algorithmes réguliers. Les architectures SIMD sont particulièrement populaires en traitement d'image par exemple, car elles permettent de s'approcher de la configuration spatiale de l'image. Elles présentent un bon support pour les applications data parallèles [35]. Citons par exemple les processeurs de traitement de signal (*DSP*) qui intègrent aujourd'hui des unités SIMD afin de répondre aux traitements numériques intensifs [35], [97]. Les processeurs généralistes sont dotés aussi d'extensions SIMD. À titre d'illustration, Intel, avec sa famille de processeurs Pentium, a ajouté un jeu d'instructions limitées de type SIMD de technologie MMX au début de 1997. Par la suite, l'arrivée des processeurs Pentium III et 4 incluait de nouvelles technologies, SSE et SSE2, avec pour but d'étendre l'utilisation de la technologie SIMD.

Nous concluons que les architectures massivement parallèles de type SIMD sont les cibles d'exécution les plus adaptées et performantes pour les applications data parallèles. Elles sont destinées principalement à assurer un parallélisme de données. Visant les applications régulières de TSS, nous optons alors pour ces architectures vu leur efficacité pour ce type de traitements. Des exemples de machines SIMD incluent la Connection Machine 2 de Thinking Machines Corporation, la MP-1 de MasPar [13] ou les DAP de AMT [39] et la Zephir de Wavetracer.

#### Déclin des machines SIMD traditionnelles

Ces machines ont connu un essor à l'époque de la Connection Machine. Elles étaient conçues comme des circuits spécialisés destinés à des applications bien précises. À la fin des années 90, on assistait à un déclin de ces architectures entièrement propriétaires du fait de la complexité technologique d'un côté et du coût de conception d'un autre côté. En effet, il était difficile de pouvoir synchroniser l'horloge sur plusieurs processeurs intégrés sur des cartes séparées. La conception de processeurs ad-hoc spécifiques pour ces machines a aussi augmenté leur coût de développement et de fabrication surtout avec l'apparition sur le marché de nouveaux processeurs standard et performants qui étaient beaucoup moins chers. Il était alors moins coûteux d'acheter des processeurs standard et de les inclure dans une machine que de concevoir un nouveau processeur spécifique vendu à petite échelle. Ces problèmes ont alors conduit à l'abandon des machines SIMD.

#### Les progrès technologiques récents : renaissance de SIMD

Cependant, l'évolution des performances des processeurs n'a cessé de croître rapidement. En effet, les avancées technologiques concernant l'intégration n'ont cessé d'évoluer en suivant la célèbre loi de Moore. On voit ainsi se profiler des circuits intégrés au milliard de transistors. Ce contexte a poussé les concepteurs à reconsidérer l'intérêt et la faisabilité des architectures parallèles sur une puce. Ajoutons de plus, le grand intérêt et efficacité que

présentaient les architectures SIMD vis à vis des traitements réguliers et parallèles de données. Ainsi, des extensions SIMD, tel qu'énoncé précédemment, sont ajoutées à la plupart des nouveaux processeurs. Du fait de leur structure simple, on voit aussi apparaître des systèmes SIMD embarqués tel est l'exemple du calculateur Symphonie [26], développé par le CEA-LETI, qui fait partie du système de veille infrarouge du Rafale. Ces systèmes répondront aux besoins des applications TSI temps réel.

#### Contraintes de conception d'architectures parallèles sur SoC

Visant le domaine des systèmes sur puce, concevoir de tels circuits pose des contraintes en termes de développement et de coût. Les circuits matériels spécifiques sont capables de surmonter les contraintes de vitesse. Cependant, ces solutions dédiées sont sujettes à des contraintes supplémentaires. En effet, elles doivent faire face à une augmentation rapide des coûts d'accès au silicium et à des besoins de faible consommation et de robustesse. Une autre contrainte est liée à la dynamique toute particulière du marché des systèmes embarqués. Afin de rester compétitif, les constructeurs se doivent de garder les coût et temps de production extrêmement bas tout en optimisant leurs temps de mise sur le marché (*TTM*). Il faut alors faire appel aux nouvelles méthodologies de conception et d'intégration.

## 1.2 Problématique

#### Complexité croissante des SoCs

La complexité des systèmes sur puce ne cesse de croître, d'autant plus lorsqu'il s'agit de systèmes hautes performances. Ces systèmes sont soumis à des contraintes de temps de mise sur le marché (conception, développement, vérification, validation, etc.). Il s'avère alors nécessaire de réduire leur temps de développement tout en assurant leur performance et facilité d'adaptation. En effet, offrir de nos jours un système flexible est d'une importance majeure vue que les applications multimédias récentes sont en pleine évolution et se caractérisent par la diversité des algorithmes (filtrage, corrélation, etc.) ainsi que leurs nombreux paramètres. L'exploration et la comparaison de différentes solutions algorithmiques requièrent l'utilisation d'architectures programmables et flexibles. Par le terme flexibilité, on entend l'adaptabilité du système à répondre aux différents besoins d'un domaine d'applications précis. La flexibilité est requise pour des raisons de coût de conception en permettant de modifier une architecture en fonction de l'évolution ou des caractéristiques très diverses des applications.

#### Compromis performance/flexibilité

La figure 1.3 [5] montre le compromis performance/flexibilité entre une architecture à usage général et une architecture complètement spécialisée. La performance est évaluée en termes de ratios d'efficacité de surface (MOPS/$mm^2$), efficacité énergétique (MOPS/mW) et efficacité de poids (MOPS/Kg). Bien que les architectures spécifiques offrent de meilleures performances vis à vis d'une application donnée, elles restent des solutions dédiées limitant leur flexibilité. Leur spécificité exige aussi des méthodes de conception bien appropriées et l'utilisation des composants spécifiques tel que le recours aux accélérateurs matériels dédiés à l'exécution d'une application bien précise.

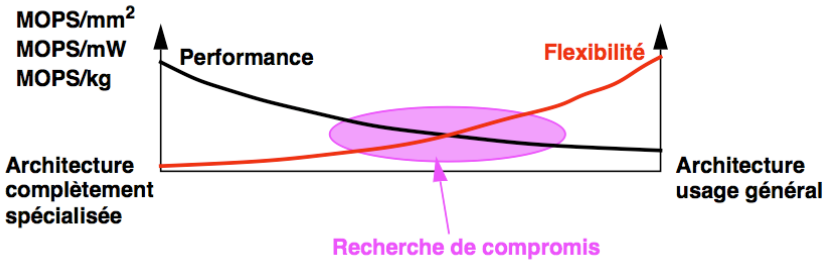



FIGURE 1.3 – *Compromis performance/flexibilité*

### SIMD spécifiques à une application

Ce contexte touche particulièrement les architectures SIMD dont la plupart sont conçues pour être spécifiques à une application particulière. À titre d'exemple, on cite l'architecture SIMD sur puce dédiée au traitement d'images et spécialisée dans la détection de contours [91] et l'architecture SIMD sur puce décrite dans [93] dédiée pour l'application d'estimation de mouvement (*VBSME : Variable Block Size Motion Estimation*). Ces systèmes sont aussi souvent des solutions propriétaires vu qu'ils se basent sur des composants dédiés et destinés pour une tâche bien particulière. Les tendances actuelles de conception des SoCs demandent à ce que le matériel soit de plus en plus programmable, comme une solution à la flexibilité, afin de suivre l'évolution croissante du logiciel. En effet, il faut éviter d'avoir à concevoir une nouvelle architecture aussi souvent qu'un nouvel algorithme est créé, ce qui peut arriver souvent dans un domaine aussi créatif que les multimédias (synthèse d'image, jeu vidéo, etc.). En cela, le fait d'avoir une architecture adéquate et programmable permet de s'adapter efficacement à une grande classe d'algorithmes.

### Nos travaux : intégration d'IP, modélisation haut niveau et validation sur FPGA

Nos travaux traitent plusieurs points clés :

– Pour faire face à la complexité croissante des systèmes sur puce, de nouvelles méthodologies d'intégration se basant sur l'utilisation des composants logiciels ou matériels existants (*IP*) sont apparues pour assurer la flexibilité [12]. En effet, les nouveaux outils et méthodes de conception se basent de plus en plus sur la réutilisation des blocs préconçus. Pour diminuer le coût de développement d'un système sur puce, il faut aussi connaître les bons composants à intégrer (disposer d'un large catalogue d'IP) et savoir comment les intégrer de manière efficace (appliquer une méthodologie de réutilisation) [1]. Cela s'applique aussi parfaitement pour les architectures parallèles (SIMD). Ces architectures sont soumises à de fortes contraintes de performance, de coût de production et de temps de mise sur le marché. La réutilisation de blocs préconçus s'avère donc utile pour accélérer la conception de tels systèmes.
Les composants IP peuvent être de trois sortes : description comportementale du composant (IP soft), description RTL synthétisable du composant avec contraintes de placement-routage (IP synthétisable) ou bloc matériel placé, routé et optimisé selon une filière technologique (IP hard) [108]. La complexité réside donc dans l'assemblage des IPs. Une évaluation et un raffinement préalable de ces composants s'avèrent nécessaires en vue d'intégration dans le système final. Citons comme exemple la conception

des PEs qui est la plus délicate dans une architecture SIMD. Dans la perspective de réaliser une architecture massivement parallèle, la simplicité du PE et de sa conception est un point fondamental [117].

- Afin de faire face aux difficultés de conception des SoCs, des techniques de modélisation et des langages communs ont été utilisés. En effet, une conception à un haut niveau d'abstraction permet d'aborder les systèmes tout en s'affranchissant, dans un premier temps, des détails qui font leur complexité. En revanche, elle doit aussi être accompagnée, dans un deuxième temps, de processus de raffinements permettant de systématiser des mises en œuvre effectives. Dans ce contexte, l'ingénierie dirigée par les modèles (IDM) est considérée aujourd'hui comme l'une des approches les plus prometteuses dans le développement des systèmes. L'IDM offre un cadre conceptuel permettant la génération automatique du code du système à partir d'une modélisation de haut niveau. On cite l'environnement GASPARD qui est un environnement de codesign orienté pour les applications de traitement de signal intensif [28]. Il se base sur l'usage de l'IDM comme méthodologie de conception en allant de la modélisation en UML jusqu'à la génération de code.
- De nouvelles plateformes de prototypage et d'expérimentation sont aussi apparues afin de faciliter la validation et vérification des SoCs, à savoir les circuits reconfigurables (*FPGA*). En effet, une solution intermédiaire entre les processeurs généralistes et les circuits complètement dédiés s'appuie sur la technologie en plein essor des circuits logiques programmables ou PLD, et en particulier la famille FPGA. Les FPGAs telles des ardoises magiques, peuvent être reconfigurés à volonté. Ils peuvent donc être dédiés à une application de manière temporaire puis reprogrammés pour une toute autre application (flexibilité). Les FPGAs offrent de même une souplesse de conception grâce à leur facilité d'utilisation et leur facilité de programmation avec la disponibilité des outils nécessaires ; ce qui raccourcit les cycles de conception et permet d'effectuer des essais réels. L'avantage principal des circuits logiques programmables réside donc dans la rapidité de mise au point qu'ils procurent. Cela permet une mise à disposition très rapide du système embarqué. C'est pourquoi les circuits logiques reprogrammables sont souvent utilisés pour le prototypage rapide de systèmes et aux petits volumes de production. Dans notre travail, nous utiliserons alors les FPGAs comme cible de prototypage et d'expérimentation de part ses capacités, sa vitesse et sa grande flexibilité. Rappelons de plus que notre objectif est de proposer des architectures génériques et ad-hoc en fonction de l'application.

#### Motivations de SIMD sur puce

Différents éléments de réponse par rapport aux approches et méthodologies citées plus haut ont été proposés pour les architectures multiprocesseurs sur puce. Cependant peu de travaux s'intéressent aux méthodologies de conception des architectures SIMD sur des cibles reconfigurables (FPGA). Nous ciblons les architectures SIMD sur puce pour plusieurs raisons :

- les architectures SIMD sont de bon supports pour les applications de TSS (tel que précédemment montré ) ;
- les architectures SIMD sont caractérisées par un nombre important de processeurs élémentaires (simples processeurs) offrant une grande capacité de calcul à un coût relativement faible ;

– les architectures SIMD présentent un modèle de programmation simple : un parallélisme reposant sur la répétition de petites unités d'exécution relativement simples à réaliser et sur un séquencement global ;
– une seule copie du programme est nécessaire et donc une faible consommation mémoire ;
– le parallélisme massif SIMD a une excellente efficacité énergétique sur les traitements réguliers [56, 22].

Il est alors nécessaire de faire appel aux nouvelles méthodologies pour la conception des architectures SIMD afin d'assurer une conception plus rapide et plus efficace. Concevoir rapidement une architecture SIMD flexible aura un grand impact sur les applications de TSS.

#### Contribution

C'est dans ce cadre que se situe notre travail. Il s'agit de définir et implémenter une architecture SIMD sur puce adaptée à une large variété d'applications de traitement de signal systématique. Un environnement facilitant la conception de telles architectures en fonction des besoins s'avère alors nécessaire. Pour répondre aux exigences d'efficacité, de réutilisation et de programmation, il est nécessaire de pouvoir concevoir une architecture avec un processus facile et re-programmable tout en tenant compte d'une application donnée. Afin de maîtriser la complexité, il faut pouvoir réutiliser au mieux les composants ou les blocs de composants déjà existants, qu'ils aient été produits précédemment en interne ou qu'ils soient mis à disposition par un vendeur extérieur. La gestion du parallélisme et de la répétitivité, induite par la réplication des éléments de calcul, dans le système est aussi un point clé qu'il faut pouvoir traiter spécifiquement.

## 1.3 Contributions

Notre contribution touche donc le domaine de conception des architectures SIMD parallèles sur puce et consiste à proposer des solutions permettant de résoudre les problèmes évoqués précédemment. En effet, le travail réalisé se compose de quatre axes :

#### La définition d'un modèle pour mppSoC : générique et adaptatif

Un modèle générique et flexible pour un système mppSoC (*massively parallel processing System on Chip*), système SIMD massivement parallèle sur puce, a été proposé. Ce modèle a été inspiré principalement de l'architecture traditionnelle MasPar [13]. Il est flexible, paramétrique et configurable afin de s'adapter à une application donnée. Le système mppSoC est spécifié à un haut niveau d'abstraction dont différentes configurations peuvent être construites.

#### La proposition d'une démarche de conception rapide et modulaire

Une démarche de conception par assemblage de composants a été définie. Différents composants matériels (*IP*) tels que processeur, mémoire, réseau d'interconnexion, etc. ont été conçus et implémentés. Afin de favoriser la réutilisation de ces IPs dans la conception de mppSoC, une bibliothèque de composants, mppSoCLib, a été mise en place. Un processus

de raffinement/transformation des IPs a été aussi réalisé. Ce processus a pour but d'adapter l'IP utilisé aux exigences de l'architecture matérielle et des contraintes de conception.

#### Le développement d'un outil de génération de configurations mppSoC

Un outil de génération de configurations instances du modèle mppSoC a été développé. À partir d'une spécification à un haut niveau d'abstraction (modèle UML utilisant le profil *MARTE*), l'outil permet la génération automatique du code VHDL tout en s'appuyant sur la bibliothèque d'IP dédiée à mppSoC. L'outil permet à l'utilisateur de choisir une configuration SIMD répondant à ses besoins applicatifs. Il a été intégré dans l'environnement Gaspard [28].

#### Expérimentation sur FPGA

En vue de tester et valider le modèle mppSoC ainsi que sa démarche de conception, les circuits FPGA ont été choisis comme des plateformes cibles. Un prototypage de différentes configurations mppSoC variant leurs paramètres ainsi que leurs composants intégrés a été effectué. Une mesure de performances peut se faire en se basant sur les outils de simulation et synthèse. Cette étape est d'un intérêt majeur et permet de valider les performances de la configuration mppSoC définie. Si les performances de l'architecture générée sont insuffisantes alors une simple modification de la modélisation haut niveau peut se faire facilement. Suite à la quelle, l'outil développé permet alors de régénérer le code VHDL de la nouvelle architecture définie. Dans un contexte applicatif réel, le système mppSoC a été utilisé pour définir une chaîne de traitement vidéo temps réel.

## 1.4 Plan

Le manuscrit est organisé selon le plan suivant :

**Chapitre 2 : État de l'art** Dans ce chapitre, nous présentons le contexte de nos travaux qui abordent le domaine de traitement de signal systématique. Nous présentons les motivations qui nous ont conduit à adopter les architectures SIMD pour le TSS. Nous étudions et comparons alors différentes architectures SIMD proposées. Nous décrivons de même les méthodologies de conception optées. À partir de cette étude, nous positionnons nos travaux et nous donnons les grandes lignes de nos contributions.

**Chapitre 3 : Modèle mppSoC : architecture SIMD paramétrique et générique** Ce chapitre a pour but de définir le système mppSoC. Une vue globale du modèle générique de mppSoC est présentée. Nous détaillons alors ses caractéristiques, ses aspects paramétriques et ses différents composants. Cette analyse du modèle est aussi accompagnée par une justification des concepts permettant la flexibilité de mppSoC .

**Chapitre 4 : Méthode de conception rapide et programmation de mppSoC** Dans ce chapitre, la démarche de conception modulaire proposée pour générer des configurations mppSoC est détaillée. À ce niveau, nous présentons les différentes approches proposées pour concevoir et implémenter une architecture SIMD sur puce se basant sur la bibliothèque d'IPs mppSoC, mppSoCLib. Enfin, un jeu d'instructions parallèle développé permettant la programmation du système mppSoC est présenté.

**Chapitre 5 : Outil de conception de mppSoC : mise en œuvre et expérimentation sur FPGA** Ce chapitre décrit la mise en œuvre de l'outil de génération de configurations mppSoC : d'un modèle UML-MARTE jusqu'à la génération du code VHDL synthétisable. Ce code permet l'implémentation de mppSoC sur FPGA. Les outils de synthèse et de simulations sont alors utilisés afin de mesurer les performances d'une configuration donnée.

Une étude de cas d'une application de traitement vidéo embarquée temps réel placée sur différentes configurations mppSoC est présentée. Cette étude permet non seulement de donner un aperçu sur l'outil développé mais valide aussi les différentes propositions et implémentations faites précédemment.

**Chapitre 6 : Conclusion et perspectives** Nous concluons cette thèse par le bilan des travaux effectués et détaillons les contributions apportées avant d'aborder quelques perspectives à nos travaux.

# Chapitre 2

# État de l'art


**2.1 Applications de traitement de signal systématique et intensif . . . . . 15**
**2.2 Architectures SIMD pour le TSS . . . . . 17**
2.2.1 Modèle d'exécution SIMD . . . . . 18
2.2.2 Motivations : efficacité de SIMD pour le TSS . . . . . 18
2.2.3 SIMD embarqué . . . . . 19
**2.3 Architectures SIMD sur FPGA . . . . . 20**
2.3.1 Intérêt des FPGA . . . . . 21
2.3.2 Classification des architectures SIMD sur FPGA . . . . . 22
**2.4 Conception et prototypage des architectures SIMD . . . . . 28**
2.4.1 Besoin de nouvelles méthodologies d'intégration . . . . . 29
2.4.2 Intégration d'IP . . . . . 29
2.4.3 Environnements de conception pour systèmes sur puce . . . . . 31
**2.5 Quelle méthodologie d'assemblage ? . . . . . 33**
**2.6 Conclusion . . . . . 35**

Ce chapitre traite tous les points clés évoqués précédemment. Nous débutons par une introduction aux applications de traitement du signal systématique en mettant l'accent sur leurs spécificités en terme notamment de parallélisme de données. Cette partie a pour but de familiariser le lecteur avec le contexte d'étude de nos travaux. Les applications visées peuvent tirer profit d'exécution sur des architectures parallèles SIMD vu leur régularité. Nous étudions alors l'évolution des architectures SIMD et leurs réalisations récentes. Une attention particulière est accordée aux architectures SIMD à base de FPGA. L'FPGA est utilisé comme une plateforme cible d'expérimentation vu qu'il facilite de tester rapidement différentes configurations architecturales. Les outils de conception et réalisation des systèmes sur puce, particulièrement à architectures parallèles, sont après introduits. Nous mettons l'accent sur les environnements à base de réutilisation de composants. Nous verrons alors que la conception à base d'IP permet de répondre à certains défis de conception de SoC. Mais, nous constaterons que peu de travaux sont en liaison avec les architectures SIMD permettant la génération rapide et facile de telles architectures qui soient flexibles et pouvant s'adapter à une application donnée. Une analyse de cet état de l'art montre les limites et les problématiques rencontrés dans la conception des architectures SIMD sur puce. Au terme de ce chapitre, nous serons alors en mesure d'introduire notre système mppSoC et nous mettrons en exergue les grandes lignes de nos contributions.

## 2.1 Applications de traitement de signal systématique et intensif

Le traitement du signal (TS) est la discipline qui développe et étudie les techniques de traitement, d'analyse et d'interprétation des signaux. Un *signal* peut être défini comme étant la représentation physique d'une information à transmettre. Parmi les types d'opérations possibles sur ces signaux, on peut dénoter le contrôle, le filtrage, la compression de données, la transmission de données, la déconvolution, la prédiction, l'identification, etc [1]. La partie du TS qui nous intéresse est sa partie la plus intensive et la plus gourmande en calcul, dénotée TSI dans la figure 2.1. Une application est dite *intensive* si elle opère sur une grande masse de données, et s'il faut fortement l'optimiser pour obtenir la puissance de calcul requise et répondre aux contraintes d'exécution du système [15].

Le traitement de signal intensif (figure 2.1) se compose d'une phase de Traitement de Signal Systématique (TSS) suivie d'une phase de Traitement de Données Intensif (TDI). Le TSS correspond à la première phase de traitement des signaux et consiste en l'application de traitements très réguliers (indépendants de la valeur des données) appliqués systématiquement aux signaux d'entrée pour en extraire les caractéristiques intéressantes. Celles-ci sont ensuite traitées par des calculs plus irréguliers (dépendants de la valeur de ces grandeurs) dans la phase de TDI. Le TDI consiste le plus souvent à extraire les informations pertinentes d'un ensemble important de données.

Les applications de TSI sont largement utilisées dans des domaines variés allant de l'automobile aux communications sans fils, en passant par les applications multimédias et les télécommunications : filtrages (FIR, LMS...), transformations (DCT, FFT...), codage d'erreurs (Turbo code, viterbi...), compression et décompression d'image et de flux vidéo (JPEG, MPEG), etc. Ces applications sont rencontrées souvent dans les SoC et sont caractérisées par leur grande puissance de calcul ce qui nécessite la parallélisation des traitements.

1. http ://fr.wikipedia.org/wiki/Traitement_du_signal

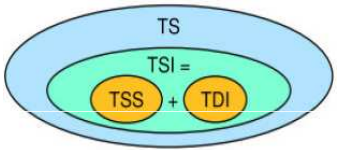



FIGURE 2.1 – *Composition du TSI*

### Exemple d'applications

Quelques exemples représentatifs des applications de TSI composées d'une phase de TSS et une phase de TDI sont illustrés ci dessous :

- **Traitement sonar :** un système de détection sous marine se base sur une première étape de traitement systématique (constituée d'une FFT qui ajoute une dimension fréquentielle aux données traitées) sur les données produites par les hydrophones (microphones répartis autour du sous-marin). Cette première phase produit des données représentant les échos captés. Ces échos sont ensuite analysés par un traitement de données : la poursuite. Ce traitement est dédié à la détection d'objets et à leur poursuite au cours du temps.
- **Récepteur de radio numérique :** cette application en émergence fait appel à une partie frontale de TSS consistant à la numérisation de la bande de réception, la sélection du canal et l'application de filtres permettant d'éviter les parasites. Les données fournies par ces traitements systématiques sont ensuite envoyées dans le décodeur dont le traitement est plus irrégulier (synchronisation, démodulation, etc.).
- **Convertisseur 16/9-4/3 :** Souvent devant notre télévision et en utilisant une télécommande on peut passer d'un format à un autre avec une certaine facilité. En pratique, ce processus se décompose en deux étapes [69]. La première consiste à créer des pixels à partir du signal vidéo au format 16/9 via une interpolation, il en résulte un nouveau signal vidéo. La seconde étape supprime certains pixels de ce signal de manière à ne conserver qu'une partie de l'information, cela permet le passage au format 4/3. Cette application ne considère à aucun moment la valeur des pixels, il s'agit donc de traitement de signal systématique.
- **Radar anti-collisions :** la prévention des collisions dans un système se fait par le biais d'une antenne qui renvoie un signal sous la forme d'un flux de données contenant des informations relatives à la présence d'obstacles, un écho. Ces informations seront filtrées dans une phase de traitement systématique de manière à en faire ressortir les caractéristiques intéressantes (la présence d'obstacles). Cette phase est suivie d'une analyse irrégulière de ce nouveau flux de données de manière à valider la présence d'un obstacle, à déclencher un freinage d'urgence ou à le poursuivre si nécessaire [61]. Cette application est illustrée à la figure 2.2.

### Caractéristiques du TSS

Notre travail se concentre particulièrement sur les applications de traitement de signal systématique et intensif qui jouent un rôle crucial dans le domaine des multimédias et traitement d'images et de la vidéo. Telle qu'illustré par les exemples précédents, une applica-

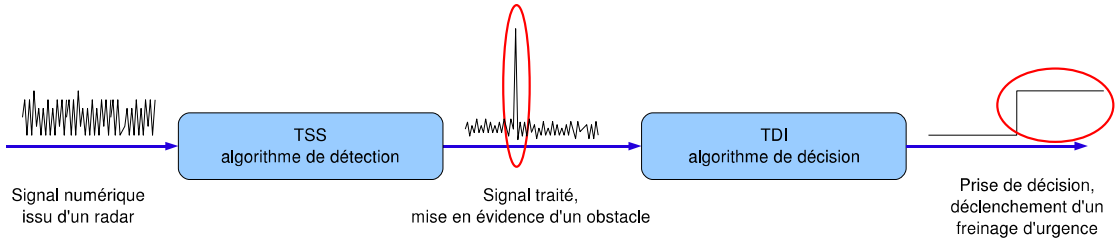



FIGURE 2.2 – *Représentation de l'application radar anti-collision*

tion est dite *systématique* si son traitement consiste principalement en des calculs réguliers et indépendants, appliqués systématiquement sur des données en entrée pour fournir des résultats. Le TSS offre diverses caractéristiques intéressantes du point de vue des données et des calculs : l'organisation des données en tableaux, le changement de la taille des tableaux au fil des calculs et l'évolution possible du nombre de leur dimension. La régularité de ces traitements fait que l'ensemble des calculs effectués sur les données sont indépendants de leurs valeurs. Ces calculs ne sont généralement pas différents d'une application à une autre (composés majoritairement de produits scalaires, transformés de Fourier (FFT), etc.). Ces caractéristiques de TSS démontrent que l'interaction des calculs avec les données est plus importante que la nature des calculs eux-mêmes. L'exécution de ces applications fait souvent appel à des techniques parallèles et distribuées et l'utilisation de circuits spécialisés. La majorité de ces applications exigent un traitement en temps réel et sont souvent embarquées. Des contraintes drastiques de consommation et de surface s'ajoutent alors aux contraintes de temps.

Les difficultés de développement des applications de TSS sont principalement l'exploitation du parallélisme de données et le respect des contraintes de temps et des ressources d'exécution. La complexité croissante des algorithmes implémentés, et l'augmentation continue des volumes de données et des débits applicatifs, requièrent souvent la conception d'architectures dédiées [20]. Il est alors nécessaire de trouver des architectures efficaces offrant les performances suffisantes pour l'exécution de telles applications.

Dans ce cadre et afin de profiter de ce parallélisme, il est logique de penser à l'utilisation des systèmes parallèles sur puce. Parmi les quels on trouve ceux à architecture SIMD qui sont bien adaptés aux applications de TSS. La section suivante introduit les architectures SIMD, leurs spécificités et met l'accent sur leur évolution.

## 2.2 Architectures SIMD pour le TSS

Les architectures SIMD ont toujours constitué un centre d'intérêt important pour les applications de TSI exploitant un parallélisme de données inhérent à ces applications. En particulier, la majorité des applications de TSS manipulent des ensembles de données (typiquement vecteurs, matrices, etc.) en répétant la même opération sur chaque donnée. Ceci fait que le parallélisme SIMD est efficace offrant les performances suffisantes pour exécuter ce type d'applications.

Dans une première partie, nous définissons le modèle d'exécution SIMD. Dans une deuxième partie, nous détaillons ses spécificités en tant qu'architectures adaptées au TSS et nous survolons ses apports par rapport à ce domaine applicatif.

### 2.2.1 Modèle d'exécution SIMD

Le modèle d'exécution SIMD se base sur un parallélisme de données. Ce type de parallélisme est bien adapté dans notre cas puisque nous avons besoin d'effectuer une même opération sur toutes les données ou bien sur un ensemble de données. Les propriétés de régularité des applications de TSS, tant au niveau des traitements que de l'accès aux données, ont motivé l'utilisation de ce parallélisme de type régulier. Ce modèle permet alors d'exprimer efficacement et correctement les applications ciblées.

En terme de programmation, le contrôle est séquentiel alors que l'accès aux données s'effectue en parallèle. Partant d'un seul programme data parallèle contenant à la fois instructions séquentielles et parallèles, l'ordre d'exécution est piloté par une unité de contrôle unique. Cette unité exécute toute instruction séquentielle et de contrôle et envoie les instructions parallèles à un ensemble de processeurs élémentaires (PE) indépendamment des données locales. Celles-ci résident dans la mémoire locale de chaque PE. Du fait du comportement synchrone de l'architecture SIMD, les PEs exécutent la même instruction au même instant sur des données différentes. Ils sont en effet guidés par le processeur de contrôle et n'ont aucune autonomie individuelle. Parmi les instructions exécutées, on trouve des instructions de communication. Les communications au sein d'une architecture SIMD s'effectuent d'une manière synchrone. Elles peuvent être soit des communications de voisinage soit des communications point à point assurant un transfert de données parallèle. Ce modèle se base aussi sur un système de contrôle de l'activité (masquage) qui permet de désigner les PEs qui participent à l'exécution de l'instruction courante et désactiver les autres.

### 2.2.2 Motivations : efficacité de SIMD pour le TSS

D'après le paragraphe précédent, nous avons montré que le modèle d'exécution SIMD assure un parallélisme de données bien adapté aux applications de TSS. Dans le cas des applications de traitement vidéo par exemple (réduction de bruit, etc.), la plupart des opérations sont exactement les mêmes pour toutes les données, et donc profitent efficacement des architectures SIMD. Le fait d'avoir des flots séparés pour traiter les données reste une solution efficace pour le calcul rapide sur un grand nombre de données.

L'aspect synchrone d'une architecture SIMD facilite beaucoup la programmation parallèle. En effet, pour le programmeur, tout se passe comme si son programme s'exécutait sur une architecture séquentielle classique, à la seule différence que les instructions parallèles exécutent une même opération sur des données multiples au lieu d'une opération simple. L'unicité du code à écrire simplifie encore la programmation. En plus de son modèle de programmation, le modèle d'architecture SIMD offre plusieurs avantages. En effet, une architecture SIMD est simple à concevoir. Une seule copie du programme est nécessaire avec un décodage d'instructions simple. Ceci minimise les coûts vu qu'on duplique uniquement les unités d'exécution sans dupliquer l'unité de contrôle (qui demande beaucoup plus de ressources). En effet, la distribution du contrôle entraîne un sur-coût matériel et logiciel qui peut être pénalisant. Les architectures SIMD peuvent traiter un très grand nombre de données dans un même cycle. Elles allient alors la simplicité à l'efficacité. Un autre critère très important est la synchronisation naturelle des processeurs lors des communications ; il n'y a donc pas de problèmes de synchronisation inter-processeur par définition.

Pour ce type d'architecture, le parallélisme est souvent obtenu en appliquant directement les traitements sur la structure. Ainsi, les opérations matricielles se mettent en œuvre

aisément sur des architectures SIMD en matrice 2D de processeurs. Nous notons différents réseaux SIMD bidimensionnels à base de processeurs bit série tels que Goodyear's MPP [94], GAPP [74], DAP [39], etc. Pour les architectures SIMD à structure linéaire, les données seront traitées linéairement tel est l'exemple de CLIP7 [40], SLAP [37], SYMPATI [10], PRINCETON [100], etc.

De point de vue consommation de puissance, les processeurs SIMD s'avèrent économiques. Ils présentent une consommation de puissance réduite et une excellente efficacité énergétique sur les traitements réguliers [34]. Un parallélisme de données exploité sur une architecture SIMD permet de réduire la consommation de puissance tout en réduisant la charge de recherche et décodage des instructions [89]. Dans [87], les auteurs montrent que l'utilisation des instructions SIMD permet de diminuer la puissance dissipée sur un certain nombre de benchmarks représentatifs du traitement d'images. La localité permet de même de réduire la consommation [36]. En effet, plus les unités communicantes (tel est l'exemple du PE et sa mémoire de données) sont proches sur la puce, plus la consommation liée aux transferts de données est réduite de façon drastique.

Il est clair alors que les architectures SIMD présentent de bon candidats pour implémenter efficacement les applications de TSS. Ces applications sont de nos jours de plus en plus embarquées (traitement multimédia, capteurs intelligents, cryptologie, etc.) introduisant des contraintes de consommation, d'encombrement et de temps réel. Elles exigent alors des systèmes embarqués ou SoC, d'ou la nécessité des systèmes SIMD sur puce.

### 2.2.3 SIMD embarqué

La grande majorité des architectures SIMD s'est heurtée vers la fin des années 90 à un problème de coût matériel. En effet, ces architectures s'étaient basées sur des processeurs spécifiques, et n'avaient pas profité de l'évolution des processeurs du commerce (Mips, Alpha, Pentium, etc.). Elles étaient alors dépassées en performances par des composants standard. Plus spécifiquement, les machines SIMD les plus anciennes ont été des machines complètement câblées telle que CM2 [27] et beaucoup de processeurs avaient des fonctionnalités limitées. D'autres machines SIMD de taille importante étaient coûteuses en terme de construction et temps de conception. Cet obstacle est à présent en partie levé grâce aux possibilités d'intégration offertes par les technologies actuelles. De plus, le coût de fabrication d'un système utilisant des microprocesseurs standard du commerce revient moins cher qu'un système bâti autour de quelques processeurs spécifiques puissants. Un autre problème est la limite de l'augmentation de la fréquence de fonctionnement vu qu'il était difficile de diffuser de manière synchrone les instructions à tous les processeurs se trouvant sur des cartes séparées.

Avec l'évolution récente des technologies d'intégration et l'utilisation des circuits programmables ainsi qu'au besoin ressenti par l'industrie du jeu vidéo, de nouveaux processeurs SIMD sont apparus. Dans ce cadre, une attention particulière doit être aussi accordée aux approches SIMD avec les coprocesseurs graphiques GPU. En effet, les cartes graphiques présentent de nos jours un grand intérêt pour le monde du calcul haute performance et il est bon de procéder à une petite description de leurs fonctionnements. Les accélérateurs graphiques présents aujourd'hui sont capables en plus des traitements graphiques d'effectuer des calculs scientifiques. Par exemple, les circuits graphiques de la société NVidia ont évolué vers une architecture massivement parallèle en considérant plusieurs unités de calcul SIMD. Aujourd'hui, un circuit Geforce 8 GTX [66] permet d'exécuter 12288 threads pour 16

multiprocesseurs fonctionnant à une fréquence de 1.5 GHz. Toutefois, la taille et la consommation du circuit sont assez rédhibitoires pour une utilisation embarquée puisque le circuit comporte au minimum 681 millions de transistors sur 470 $mm^2$ pour une consommation de l'ordre de 150 W. D'autres circuits sont aussi développés à savoir le Larrabee [96] fabriqué par Intel qui sert pour le calcul de rendu 3D et du calcul haute performance. Un autre circuit fabriqué par la société Tilera [109] est composé de 64 processeurs VLIW associés via un réseau sur puce et composé de quatre contrôleurs mémoires. Chacun des cœurs est en mesure de faire fonctionner un noyau Linux ce qui peut paraître assez lourd dans le cadre de notre problématique des architectures flots de données pour le TSS.

Ajourd'hui, nous voyons aussi la commercialisation de plus en plus de DSP destinés pour les applications de TS. La plupart de ces DSP présentent une architecture SIMD à savoir Ardbeg de ARM [110], MuSIC de Infineon [90] et Sandblaster de Sandbridge [101].

Ces architectures massivement parallèles ont été pleinement influencées par les progrès technologiques récents des circuits intégrés. L'augmentation de la capacité d'intégration des transistors rendent le placement d'une grille SIMD complète sur une unique puce envisageable. La fréquence de fonctionnement de ces architectures dépend fortement de leur intégration, c'est-à-dire de la proximité physique des processeurs. On voit alors facilement l'avantage qu'on pourrait tirer en les intégrant sur un seul circuit intégré. La diffusion synchrone d'instructions à plusieurs processeurs intégrés sur une seule puce est plus aisée dans ce contexte.

Le secteur de l'embarqué reprend souvent sur une même puce les architectures hautes performances que l'on pouvait trouver il y a quelques années sur de gros calculateurs et l'on peut raisonnablement se dire que ce transfert de technologie deviendra de plus en plus courant. D'ailleurs de nos jours, plusieurs architectures SIMD sur puce sont apparues. C'est la raison pour laquelle nous avons envisagé la création d'un système SIMD massivement parallèle dédié aux applications de TSS qu'il nous est possible d'intégrer au sein du même circuit. Ce système sur puce doit répondre aux besoins applicatifs et à des critères de consommation d'énergie, de performance ou de surface.

Notre travail s'intéresse particulièrement aux systèmes SIMD sur des circuits à architectures programmables à savoir les FPGAs. L'intérêt suscité par les FPGA est essentiellement motivé par leur grande flexibilité (grâce à leur reprogrammabilité), la facilité qu'ils présentent dans la mise en œuvre des applications, etc. La section suivante justifie ce choix et survole des exemples de systèmes SIMD sur FPGA.

## 2.3 Architectures SIMD sur FPGA

Le développement des technologies micro-électroniques a relancé l'intérêt pour certains concepts qui, il y a encore quelques années, trouvaient des difficultés pour leur concrétisation réaliste. Cette évolution croissante des circuits intégrés a favorisé alors la renaissance des systèmes SIMD sur puce. Parmi les technologies en plein essor, nous trouvons les circuits programmables en particulier la famille FPGA. L'intérêt des FPGA est détaillé dans le paragraphe suivant. Une classification des architectures SIMD sur FPGA sera après présentée.

### 2.3.1 Intérêt des FPGA

Deux principales cibles de conception sont offertes aux concepteurs : les circuits spécifiques ASICs et les circuits matriciels programmables FPGAs. Un ASIC est un circuit dédié exclusivement à un type d'application où tout est figé. La possibilité d'effectuer des modifications ou d'améliorer certaines fonctionnalités ne peut se faire que par la conception d'un nouvel ASIC. Ceci a un coût en terme de temps de conception, de complexité et d'argent. Mais en contre partie, ces circuits offrent les meilleures performances temporelles et de consommation.

Un circuit FPGA est une architecture générique se composant de blocs configurables de logique, de blocs de mémoire programmable et d'un réseau d'intercommunications programmable (figure 2.3). Plus récemment, certains FPGAs possèdent des fonctions pré-

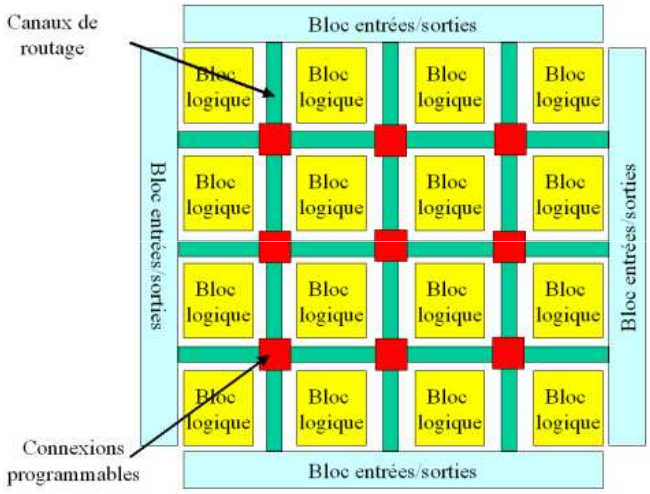



FIGURE 2.3 – *Bloc diagramme d'un FPGA*

câblées comme des multiplieurs et/ou des processeurs spécialisés implantés en hard (PowerPC, ARM) au sein même des réseaux de portes logiques. Ces circuits permettent de spécialiser une architecture à moindre coût, par programme, sans passer par le processus de fonderie long et coûteux des circuits ASIC qui sont complètement spécialisés. Ceci fait que l'FPGA est une bonne plateforme d'expérimentation à coût de développement réduit. Les performances des FPGA se rapprochent de plus en plus de celles des ASICs avec l'avantage de la reconfigurabilité. Tel que décrit dans le premier chapitre, les FPGA sont flexibles vu qu'ils sont reconfigurables à volonté et permettent de tester différents systèmes et applications. Or dans notre travail, notre objectif est de pouvoir tester différentes configurations SIMD, mesurer leurs performances et choisir la configuration la mieux adaptée aux besoins applicatifs. Ceci montre que l'FPGA est la bonne cible pour ces travaux grâce à leur mise à disposition très rapide du système embarqué. Ces circuits logiques reprogrammables sont souvent utilisés pour le prototypage rapide de systèmes et aux petits volumes de production.

Dans la suite, nous nous concentrerons particulièrement sur les implémentations SIMD à base de FPGA.

### 2.3.2 Classification des architectures SIMD sur FPGA

Afin de réduire les temps de calcul, améliorer les performances et satisfaire les exigences des applications de TSS embarquées, de nombreuses recherches se sont tournées vers des solutions SIMD. Ces solutions sont très variées allant des solutions d'accélération SIMD matérielle spécifiques jusqu'aux solutions SIMD programmables en passant par les extensions SIMD pour des processeurs embarqués.

Divers processeurs ou coprocesseurs SIMD ont été implémentés. Notons à titre d'exemple l'architecture lineDancer de Aspex Semiconductor [53] qui est une architecture couplant un processeur SIMD avec une unité reconfigurable grain fin : le ASProCore (4096 PEs). Grape-DR [67] est un autre exemple d'un coprocesseur SIMD intégrant 512 cœurs d'exécution. Le Grape-DR (Grape signifiant tout simplement raisin en anglais) prend place dans une carte au format PCI-X. Le processeur IMAPCar (*Integrated Memory Array Processor for Car*) [59], de la société NEC, est un exemple de circuit parallèle dédié au traitement d'images embarqué pour l'automobile. Cette architecture matérielle, illustrée dans la figure 2.4, vise les applications d'aide à la conduite. Xetal II [2] de la société NXP est un autre

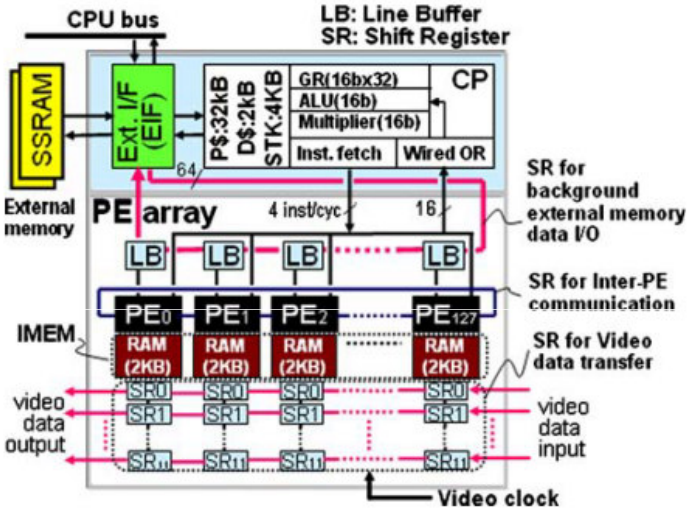



FIGURE 2.4 – *Architecture IMAPCar [59]*

exemple de processeur programmable massivement parallèle intégrant 320 unités de calcul élémentaires. L'architecture est conçue et optimisée afin d'être utilisée dans les systèmes de vision mobile et les caméras intelligentes. Ces circuits mentionnés sont toutefois peu évolutifs, car ils sont mis en place au sein d'un ASIC. Nous ciblons les architectures SIMD flexibles et programmables sur FPGA avec les avantages en termes de reconfigurations et d'évolutions que cela confère. Dans notre travail, nous nous intéresserons donc aux solutions proposées à base de FPGA. En effet, la problématique se pose différemment dès lors que l'on cible un FPGA : différentes contraintes à respecter et des compromis à résoudre.

Diverses solutions SIMD matérielles à base de FPGA ont été proposées afin d'accélérer un traitement particulier. Ces architectures sont câblées et les PEs ne peuvent effectuer que des opérations bien déterminées. Nous notons à titre d'exemple : l'architecture SIMD avec 31 PEs présentée dans [93], qui sert pour le calcul de l'estimation de mouvement. Elle a été intégrée en tant que bloc IP matériel dans la chaîne de l'encodeur H.264. Une autre architecture

SIMD [91] avec 30 PEs implémentée sur FPGA est dédiée au calcul de détection de contours se basant sur la méthode de gradient de Sobel. Un capteur CMOS, implémenté comme une grille 2D de 64x64 PEs, est présenté dans [44]. Dans ce capteur, chaque PE peut faire une convolution sur la valeur de son pixel. Ces différentes solutions matérielles peuvent compenser par leur optimisation et leurs performances souvent élevées leurs coûts et délais de développement. Cependant malgré leurs performances, elles manquent toutefois de flexibilité puisque les traitements intégrés sont figés durant toute la durée du vie du SoC le plus souvent intégré au sein de différentes générations de produits. Ces solutions se considèrent comme des accélérateurs SIMD plutôt que des processeurs SIMD cadencés par un jeu d'instructions parallèle. De telles solutions, peu programmables, restreignent de façon considérable les possibilités d'adaptation et de modification des algorithmes et de leurs nombreux paramètres. Il y a de plus un réel besoin de programmabilité pour satisfaire les exigences des applications de TSS.

Il est évident que c'est dans la programmabilité que réside la pérennité des choix architecturaux. Une limitation possible d'une telle solution programmable par rapport à une solution spécifique purement matérielle est la perte d'un peu d'efficacité dû à son aspect général. Cependant, cette comparaison ne s'applique qu'aux algorithmes qui sont assez simples pour permettre les deux implémentations. Pour des algorithmes data parallèles plus sophistiqués, la solution programmable est la plus facile à choisir. De plus, un informaticien peut mieux comprendre les détails de programmation sur une architecture parallèle et coder seul son algorithme data parallèle que de comprendre les détails d'implémentations sur un FPGA, ce qui peut dépasser ses connaissances. Par conséquent, pour résoudre un problème donné, il peut sélectionner l'algorithme qui conduit à la mise en œuvre la plus efficace dans le cas d'une architecture parallèle programmable, vue qu'il peut tester différentes implémentations. La problématique de la conception s'en trouve néanmoins complexifiée, puisque l'adéquation concerne alors trois relations intimement liées et de même importance : application, architecture, programmation.

Notre critère de classification des architectures SIMD se base sur la nature du jeu d'instructions. Nous trouvons en effet des architectures fortement couplées dans les quelles un traitement data parallèle (contient le traitement séquentiel et le traitement parallèle) existe, d'autres non couplées telles que les coprocesseurs où un traitement séquentiel fait appel parfois à un traitement parallèle. Ce dernier mode de couplage connecte généralement l'architecture SIMD comme un périphérique. Pour chacune des catégories, nous donnons quelques exemples. Cette liste n'est pas exhaustive, de part la multitude de solutions disponibles.

#### 2.3.2.1 Architectures SIMD non couplées

Dans les architectures SIMD non couplées, le processeur exécute d'une manière séquentielle son programme mais peut parfois faire intervenir un coprocesseur ou une unité vectorielle afin d'assurer un traitement parallèle. Les extensions du jeu d'instructions de divers processeurs peuvent être considérées un cas particulier de ce type de solutions. Elles servent à accélérer certaines fonctions. La fameuse instruction SIMD calculant la somme des valeurs absolues des différences de huit pixels, que l'on trouve dans les jeux d'instructions IA-32 (SSE) ou PowerPC (Altivec) utilisée en estimation de mouvement en est un exemple typique [88]. Ces solutions fondées sur des processeurs généralistes restent limitées par les puissances de calcul disponibles, plus particulièrement dans le domaine de TSS temps réel. Le problème de ces extensions réside toujours dans la complexité du compilateur du processeur

afin de tenir en compte les instructions SIMD ajoutées. De plus, cette approche spécialise le système résultant et rend généralement le développement ou modification futurs difficiles. Elle reste aussi spécifique à une seule application et un seul processeur et très dur de porter à un nouveau système avec une autre application.

Parmi les solutions d'ajout de coprocesseur SIMD, nous citons le processeur vectoriel, nommé *VESPA : Vector Extended Soft Processor Architecture* [116]. Ce processeur est portable et extensible. Il est composé d'un processeur MIPS scalaire avec trois étages pipeline, mené d'un jeu d'instructions MIPS-I, avec un coprocesseur VIRAM configurable. Ce coprocesseur est mené d'un jeu d'instruction VIRAM qui supporte les instructions vectorielles sur les entiers à virgule fixe à l'exception de la division et la multiplication-accumulation, les instructions load/store, six opérations sur des drapeaux (*flag*) et cinq instructions de manipulation de vecteurs. Le coprocesseur a été désigné à la main en utilisant le langage de description matérielle verilog et peut être configuré afin de supporter différents paramètres tels que le nombre d'éléments à traiter en parallèle, la taille de données, etc. Il communique avec le processeur scalaire à travers une interface d'instructions coprocesseur MIPS, tout en partageant le flot d'instructions et la mémoire cache de données. L'assembleur MIPS a été modifié afin de supporter les instructions VIRAM vectorielles. Le programmeur doit donc coder les instructions vectorielles en assembleur. En plus de la difficulté de programmation, la conception de ce coprocesseur est coûteuse et un développement d'un nouveau compilateur a été nécessaire afin de tenir en compte les nouvelles instructions vectorielles.

Un autre coprocesseur massivement parallèle à base de FPGA est présenté dans [17]. Il est dédié pour les machines à vecteurs de support (*SVM : Support Vector Machines*) et sert principalement pour les algorithmes d'apprentissage qui incluent les tâches de reconnaissance des scènes, des situations et des concepts ainsi que leur analyse. Ces algorithmes sont caractérisés par un parallélisme massif de données. L'architecture du coprocesseur (figure 2.5) comporte des groupes de processeurs élémentaires vectoriels (*VPE : Vector Processing Element*) opérant dans un mode SIMD. Ces VPE ne communiquent pas ensemble vu qu'il n'y a pas de dépendance de données dans les algorithmes ciblés. Le jeu d'instructions

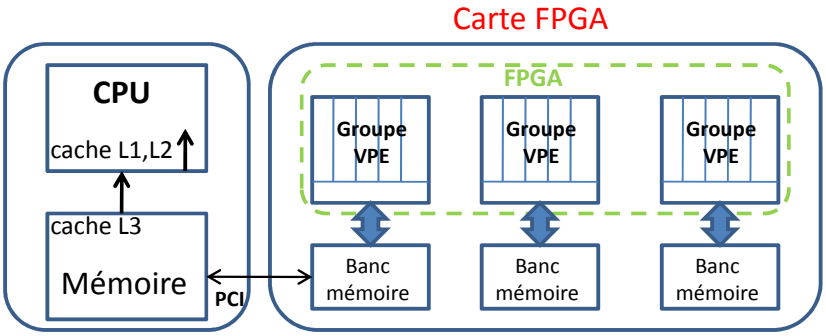



FIGURE 2.5 – *Architecture du coprocesseur lié au processeur hôte*

SIMD contient les instructions pour charger les opérandes dans les blocs RAM du FPGA, calcul des opérations arithmétiques, écriture des résultats dans les registres du FPGA et envoie des résultats dans la mémoire SSRAM (*Synchronous Static RAM*). Les blocs DSP du FPGA sont utilisés servant comme des unités parallèles de multiplication et accumulation. En effet, l'architecture est bien adaptée pour les calculs de base des algorithmes de SVM. Ce coprocesseur a été prototypé sur une plateforme contenant une interface PCI

(*Peripheral Component Interconnect*), un FPGA Xilinx Virtex 5 et 1 GB de mémoire DDR2. L'exécution couplant un processeur hôte et le coprocesseur implémenté sur un exemple d'algorithme SVM-SMO (*Sequential Minimal Optimization*) montre de bonnes performances et une accélération remarquable. Cependant, ce coprocesseur reste une solution spécifique dédiée pour l'accélération des algorithmes SVM. De plus, le couplage en mode périphérique entre le processeur hôte et le coprocesseur SIMD limite fortement les performances des communications entre les deux unités.

Les architectures SIMD évoquées précédemment présentent des caractéristiques très différentes résultant de compromis tels que le niveau de performance recherché, le coût, le volume, l'extensibilité et l'évolutivité de la solution. Elles sont parfois peu flexibles et non extensibles afin de supporter l'évolution des algorithmes de traitement vidéo par exemple. D'autant plus, ces architectures sont difficiles à programmer ; de part le coût de communication à optimiser entre le processeur principal et le coprocesseur utilisé.

Plus simples à mettre en œuvre et plus faciles à programmer, les processeurs à architecture SIMD présentent une deuxième solution pour l'exécution des applications de TSS. On se place alors dans le cadre des architectures fortement couplées.

#### 2.3.2.2 Architectures SIMD fortement couplées

Les architectures SIMD fortement couplées se caractérisent par un grand nombre d'éléments de calculs exécutant la même instruction décodée et diffusée par un processeur de contrôle.

Le circuit Ter@Core [14] de la société THALES, illustré dans la figure 2.6, est un circuit SIMD implémenté sur un FPGA Virtex4 SX55. Il atteint une performance de 19.2 GOPS avec une consommation de puissance de 14 W. Il contient un microprocesseur MicroBlaze, un PU,

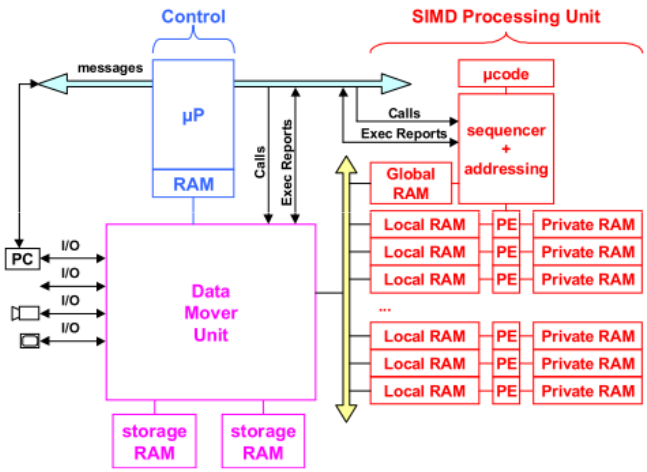



FIGURE 2.6 – *Circuit Ter@Core [14]*

un réseau DMU (*Data Mover Unit*) pour le transfert de données et un bus. Le PU se compose d'un contrôleur et 128 PEs connectés en anneau et a une fréquence de 150 MHz. La conception du PE est limitée par les ressources disponibles sur l'FPGA. Chaque PE contient un DSP servant comme ALU, deux mémoires RAM 1Kx18b et 512x36b et un peu de logique pour

les connexions. Des opérateurs (sous forme de micro-codes) dédiés au traitement d'images, à exécuter par l'unité SIMD, ont été développés. Un opérateur est activé par le microprocesseur à travers un appel de procédure qui indique l'adresse de début du micro-code ainsi que les paramètres d'appel. Les limites de cette architecture proviennent de sa dépendance vis à vis du FPGA utilisé. La conception des PEs est d'autant plus spécifique ce qui demande un temps de conception assez long. Cette architecture n'est pas flexible pour supporter différents modes de communication par exemple satisfaisant les différents besoins des applications de TSS.

Dans [95], les auteurs proposent une architecture SIMD programmable implémentée sur FPGA. Sa conception se base sur un IP processeur soft-core PicoBlaze programmable en langage assembleur. Les PEs sont des processeurs 1-bit réduits à partir de l'IP initial PicoBlaze. Cette manière de conception a beaucoup simplifié la construction des PEs et réduit le temps de conception. Utiliser un IP processeur existant est une solution incontournable pour faciliter l'implémentation de l'architecture. Pour construire un PE, les auteurs ont procédé à un raffinement de l'IP PicoBlaze consistant à ne garder que les unités opérant sur des valeurs 1-bit, alors que toutes les autres unités ont été supprimées. Les PEs ne peuvent alors exécuter que les instructions Load, AND, OR et XOR. La principale limite du processeur utilisé est son petit jeu d'instructions à fonctionnalités très limitées ainsi que la capacité limitée de la mémoire d'instructions. Ce design est dédié déjà à effectuer des traitements simples à savoir la sommation des pixels ou l'érosion pour les images binaires. Il est alors nécessaire d'ajouter de la logique si on veut exécuter un traitement plus complexe avec plus d'instructions. Malgré sa programmabilité, cette architecture demeure spécifique pour un traitement bien particulier. Nous constatons de même l'influence du choix de l'IP processeur pour la conception du PE. Il faut garder en effet un minimum de performance pour les PEs afin de répondre aux exigences de différentes applications de TSS.

Un autre processeur SIMD [63] a été implémenté sur un circuit FPGA Xilinx Virtex 1000E-6. Il est doté d'un jeu d'instructions modifié à partir du processeur PIC de la technologie Microchip. Les instructions, codées en assembleur, sont de taille 16 bits alors que les données sont de taille 8 bits. Se baser sur un processeur existant permet de faciliter la conception ainsi que la programmation du système. Le processeur intègre 95 PEs et a une fréquence de 68 MHz. Il est spécifiquement conçu pour les applications cryptographiques tel que l'algorithme de chiffrement RC4. Cette architecture ne supporte pas des communications inter-PE ce qui pose un problème pour une variété d'applications parallèles.

Un processeur SIMD intégrant jusqu'à 88 PEs est décrit dans [48]. L'architecture a une fréquence de 121 MHz et consomme 17% de la logique sur un FPGA EPS80F1508C6. Une unité de contrôle cherche et décode les instructions et envoie l'instruction décodée aux ALUs. En effet, chaque PE est constitué d'une ALU, une mémoire locale et de deux bus de données : un pour stocker les données dans la RAM et l'autre pour lire les données de la RAM. L'ALU est conçu à partir des blocs DSP présents sur le FPGA et sert pour exécuter les instructions arithmétiques. Deux blocs d'instructions personnalisées AND et OR sur des entrées 8 bits ont été ajoutés afin de supporter les instructions logiques. Dans ce design, les données ne peuvent pas être communiquées entre les PEs. Ceci constitue une grande limite de cette architecture. En effet, les communications inter-PE jouent un rôle important dans les algorithmes data parallèles et doivent être rapides afin d'assurer de bonnes performances [42]. De plus, Nous voyons dans ces deux derniers exemples que la conception des unités de traitement (PEs) est spécifique, nécessitant alors un temps de conception assez long.

Citons de même l'architecture Morphosys [50] qui est une architecture hybride composée

d'un module programmable (TinyRISC) et d'un module matériel reconfigurable (*RC Array*). Le bloc reconfigurable est ici un tableau de 8x8 cellules reconfigurables (*RC : Reconfigurable Cell*) dont toutes les cellules d'une même ligne ou colonne peuvent être reconfigurées de la même façon. Chaque cellule est en effet connectée avec ses quatre voisins les plus proches et aussi avec toutes les cellules de sa ligne et sa colonne. Cela conduit à des applications régulières effectives, mais malheureusement les communications avec des éléments non voisins paraissent être difficiles et consommatrices de temps. La figure 2.7 présente la version M2 de l'architecture Morphosys. Cette architecture opère sur des données de largeur 8 à 16 bits. Le

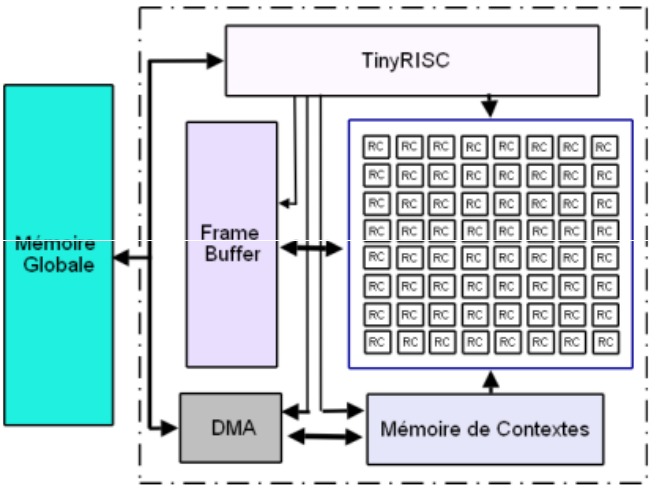



FIGURE 2.7 – *Architecture bloc de Morphosys (M2)*

circuit reconfigurable est configuré par une mémoire de contexte contrôlée par le processeur TinyRISC. Ce processeur se charge par ailleurs de l'exécution des traitements séquentiels et de la gestion des transferts de données entre la mémoire principale et le bloc reconfigurable. Ces transferts se font par l'intermédiaire d'un contrôleur DMA (*Direct Memory Access*) qui a par ailleurs la charge d'assurer les transferts de données entre la mémoire principale et une mémoire tampon de trame (*Frame Buffer*) stockant les données manipulées par le bloc reconfigurable. Cette architecture cible principalement les traitements par blocs disposant d'un parallélisme de données important.

L'équipe de H.-S. S Lee à Georgia Tech propose une architecture multiprocesseur massivement parallèle sur puce, appelée PoD (*Parallel on Die*) se basant sur le modèle SIMD [112]. Son jeu d'instructions se base sur le jeu Intel 64 légèrement modifié pour simplifier la conception. Afin de supporter les instructions parallèles, de nouvelles instructions codées à la main ont été ajoutées permettant d'envoyer des instructions natives aux PEs du processeur hôte. Les auteurs ont montré que le jeu d'instructions utilisé est aussi efficace que les architectures basées sur un jeu d'instructions spécifique. Cette architecture se compose d'un processeur hôte (un cœur du processeur Intel Core 2 Duo), d'un réseau SIMD de PE (8x8) et d'un réseau d'interconnexion PoD pour la communication inter-PE. Les PEs sont des processeurs VLIW exécutant des instructions entières, SIMD et mémoire. La communication irrégulière est totalement contrôlée par le logiciel. Ce travail ne traite pas le sujet de communication point à point, du côté du matériel, qui est assez important dans les architectures parallèles et gère toute sa complexité en logiciel. Une communication irrégulière sera alors assurée par un ensemble de communications locales ce qui peut être

coûteux en temps d'exécution.

Il apparaît que la majorité des architectures SIMD existantes sont conçues spécifiquement pour des applications bien particulières. Bien que certaines architectures se basent sur l'utilisation de l'existant afin de réduire le temps de conception, elles manquent de flexibilité leur empêchant de bien satisfaire une large variété d'applications. L'aspect d'évolution de la solution est fondamental et doit être considéré avec attention. En effet, il est important que l'architecture soit en adéquation avec l'application qu'elle va exécuter afin d'assurer de bonnes performances. En analysant les différentes architectures SIMD, il ressort que les architectures à mémoire distribuée sont celles qui répondent le plus naturellement aux exigences d'extensibilité du parallélisme massif. Assurer l'extensibilité d'une architecture SIMD fait face à de nombreuses limites en termes d'interface, distribution d'horloge, communications synchrones, etc. Il faut de même intégrer un bon réseau assurant un bon compromis connectivité/extensibilité. Pour cela, il faut bien étudier ces différents aspects afin de garantir de bonnes performances.

Pour répondre aux différentes exigences des applications de TSS, il est crucial de rendre flexible et programmable l'ensemble de la structure du système SIMD. Pour introduire plus de flexibilité au choix d'une solution architecturale qui soit adéquate à une application donnée, le système SIMD implémenté doit être paramétrique, reconfigurable et programmable. La programmabilité évite de figer les fonctionnalités et permet ainsi d'expliciter le comportement du système par un programme. La flexibilité et la reconfigurabilité permettent la souplesse de l'architecture SIMD. D'après cet état de l'art, nous concluons que l'architecture SIMD doit comporter des PEs assez performants pour répondre aux exigences de diverses applications. Les PEs doivent être aussi conçus d'une manière simple et rapide afin d'éviter les longs temps de développement et ainsi réduire le temps de mise sur le marché. De plus, la programmation de l'architecture doit être simple et facile pour un programmeur, d'autant plus elle ne doit pas demander un développement complexe pour simplifier la tâche de conception. Une architecture SIMD performante doit de même assurer les communications nécessaires aux applications data parallèles qui jouent un rôle très important dans un grand nombre d'applications.

Différentes solutions et méthodologies sont utilisées afin de faciliter la conception de systèmes qui peuvent s'adapter aux besoins applicatifs. La section suivante introduit alors certaines méthodologies proposées pour la conception et le prototypage des systèmes sur puce plus particulièrement ceux à architecture SIMD.

## 2.4 Conception et prototypage des architectures SIMD

Afin de faciliter la conception des systèmes SIMD sur puce spécialisés pour une application donnée, différentes solutions ont été proposées à savoir l'automatisation de la conception de l'architecture SIMD à partir de la description de l'algorithme et l'utilisation des composants d'architecture programmable tels que les FPGAs.

### 2.4.1 Besoin de nouvelles méthodologies d'intégration

Les méthodes de conception traditionnelles sont de moins en moins adaptées aux circuits de grandes complexités. En effet, les cycles de conception actuels des systèmes sur puce prennent beaucoup de temps alors qu'on demande à l'heure actuelle un temps de conception de plus en plus court. Les concepteurs sont amenés à gérer, non seulement un nombre énorme de transistors, mais aussi une incroyable complexité dans le design. Les possibilités architecturales pour concevoir de tels systèmes sont extrêmement nombreuses. Par conséquent, l'évaluation de différentes solutions architecturales devient un passage obligé de la conception moderne de circuits intégrés. Pour lutter contre l'écart grandissant entre la productivité et la technologie et pour répondre à une complexité croissante des fonctions à développer, de nouveaux paradigmes ont été mis en œuvre à savoir :

- Synthèse haut niveau : cette démarche de conception permet la réutilisation de composants au niveau portes logiques (association de transistors) puis la réutilisation des fonctions (assemblage de portes logiques) associées.
- Conception conjointe logicielle/matérielle, *Codesign* : l'évolution des méthodologies se porte sur la conception conjointe des parties logicielles et des parties matérielles.
- Utilisation d'IP : le rapprochement des milieux logiciel et matériel a permis d'utiliser au niveau du silicium un concept bien connu dans le monde de la conception logicielle à savoir la réutilisation de composants préconçus aussi nommés propriétés intellectuelles. Ces composants virtuels, lorsqu'ils sont correctement définis au niveau de description RTL, permettent une intégration plus aisée donc un gain de temps de conception.

L'intégration de systèmes complets sur une seule puce, couplée à une demande de plus en plus forte de réduction du temps de mise sur le marché avec des performances de plus en plus pointues, font qu'il n'est plus possible de développer un système en partant de zéro. La solution à ces problèmes est la réutilisation de blocs déjà conçus et validés. La notion de composants génériques réutilisables s'impose ainsi de plus en plus. Les environnements de co-modélisation ou de synthèse de haut niveau tirent profit du concept de réutilisation afin d'accélérer encore la conception de SoC. Nous remarquons que la conception matérielle d'un SoC repose sur l'utilisation d'outils de synthèse, de placement-routage, de plan de masse et d'assemblage au niveau du circuit, et de bibliothèques de composants réutilisables (IP).

Les questions qui se posent à ce niveau sont : est ce qu'on peut appliquer une conception à base d'IPs pour les architectures SIMD massivement parallèles sur puce ? Comment font les concepteurs de processeurs SIMD vis à vis de la notion de réutilisation ou du placement/routage ? Comment rendre une architecture SIMD dédiée à usage plus général et pouvant s'adapter aux exigences de différentes applications ? Les sections suivantes essaient d'apporter des éléments de réponse à ces différentes questions.

### 2.4.2 Intégration d'IP

Face à la complexité croissante des applications et des technologies, les nouvelles approches envisagées reposent sur l'accélération du processus de conception par la réutilisation de composants pré-définis et pré-vérifiés, dénommés composants virtuels ou blocs IP, plutôt que leur re-conception. Ces blocs de base ou IP sont rendus disponibles sous forme de bibliothèques. En effet, un IP est un élément d'une bibliothèque dont le concepteur dispose et qu'il peut directement inférer ou instancier sans avoir à le concevoir. L'assemblage

de ces composants permet de décrire rapidement des applications complexes. Par ailleurs, les ressources de conception étant devenues très précieuses, plutôt que d'en gaspiller à refaire ce qui a déjà été fait, il vaut mieux les concentrer sur les parties plus innovantes, celles qui font réellement la valeur du système, et s'appuyer sur la réutilisation pour les autres : c'est ce qu'on appelle l'*IP Reuse*.

Les IPs peuvent être des cœurs de processeurs, des cœurs dédiés, des mémoires, des cœurs reconfigurables, des MEMS [2], etc. Ces différents blocs se divisent en trois groupes :

- les *Soft-Cores* : sont par exemple des blocs décrits dans un langage de description matérielle synthétisable. Ils sont indépendants de toute technologie et sont donc très souples d'utilisation et flexibles.
- les *Firm-Cores* : sont des blocs ayant subis une optimisation en surface et en vitesse par des techniques de placement relatif. Décrits en HDL structurel, ils font appel à des composants élémentaires d'une librairie générique. Cette implantation autorise des évaluations plus fines de ressources utilisées et de performances. Le Firm-Core n'est pas routé. Ce type de macro bloc est donc un compromis entre les IPs qui sont complètement logiciels et ceux complètement matériels.
- les *Hard-Cores* : sont optimisés et placés-routés pour une technologie donnée. Le bloc est donc implicitement caractérisé en termes de performances et de surface. L'inconvénient est qu'ils sont moins flexibles et portables car le processus est dépendant de la technologie (plate-forme).

Différents standards ont été proposés pour l'interfaçage et la connexion des IPs. Ces IPs peuvent être aussi fournis avec des interfaces propriétaires.

Trois principales organisations proposent des protocoles configurables, point à point, non spécifiques à un bus tel que : VSIA (*Virtual Socket Interface Alliance*), OCP-IP (*Open Core Protocol-International Partnership*) et SPIRIT (*Structure for Packaging, Integrating, and Re-using IP within Tool flows*). VSIA est le consortium qui a développé et mis en place le protocole VCI (*Virtual Component Interface*). Cette norme décrit une communication point à point et asymétrique entre deux acteurs : un initiateur et une cible. La norme VCI est en réalité constituée de trois types d'interfaces offrant des niveaux de performance différents que le concepteur pourra choisir selon ses besoins :

- PVCI : *Peripheral VCI*, destinée aux composants d'entrée/sortie nécessitant un faible niveau de performance tels que les périphériques d'entrée/sortie (clavier, souris, etc.) ;
- BVCI : *Basic VCI*, pour des applications monothread ;
- AVCI : *Advanced VCI*, pour des applications multithread.

OCP-IP est un consortium qui définit le protocole OCP (*Open Core Protocol*). OCP [76] ressemble au protocole VCI, ajoutant de plus des signaux de contrôle et de test pour unifier toutes les communications entre les IPs. OCP définit un protocole extensible, configurable et point à point utilisant les adaptateurs et aussi les notions d'initiateurs et de cibles. Le consortium SPIRIT [99] définit le standard IP-XACT. Il se base sur le format XML pour permettre de décrire des composants au niveau RTL et au niveau TLM (*Transaction Level Model*). SPIRIT est en effet un mécanisme de spécification commun pour la description de la propriété intellectuelle, qui permet une configuration et une intégration automatisée à travers des outils plug-in.

Afin de définir une méthodologie d'intégration d'IPs pour une architecture SIMD, il faut décomposer l'architecture en IPs, définir une bibliothèque d'IPs, définir les interconnexions

2. Micro Electro-Mechanical System

entre ces IPs et développer un flot de conception partant d'une spécification jusqu'à une génération de l'architecture en se basant sur un assemblage de ces IPs fournis. Voyons alors s'il existe déjà des environnements de conception d'architectures SIMD à base d'IPs.

### 2.4.3 Environnements de conception pour systèmes sur puce

Afin de faciliter la conception des SoC, différents outils de description et simulation des circuits à différents niveaux d'abstraction ont été développés. De plus, avec la complexité accrue de ces systèmes, l'automatisation de certaines phases a été réalisée permettant de limiter le temps de mise sur le marché. Le point crucial dans un flot de conception de SoC est le choix de l'architecture adéquate car c'est elle qui va largement fixer le compromis de performances. Ce contexte est aussi valable pour les systèmes SIMD sur puce.

Dans le contexte des architectures massivement parallèles, citons l'environnement de co-design, présenté dans [46], qui permet la modélisation, la simulation et l'évaluation des architectures massivement parallèles sur puce. Les architectures cibles s'appellent WPPA (*Weakly-Programmable Processor Arrays*) qui se composent d'un ensemble de simples PEs (*WPPE : Weakly Programmable Processing Elements* ) inter-connectés par des structures régulières, illustrées dans la figure 2.8. Le jeu d'instructions peut être configuré dynamiquement.

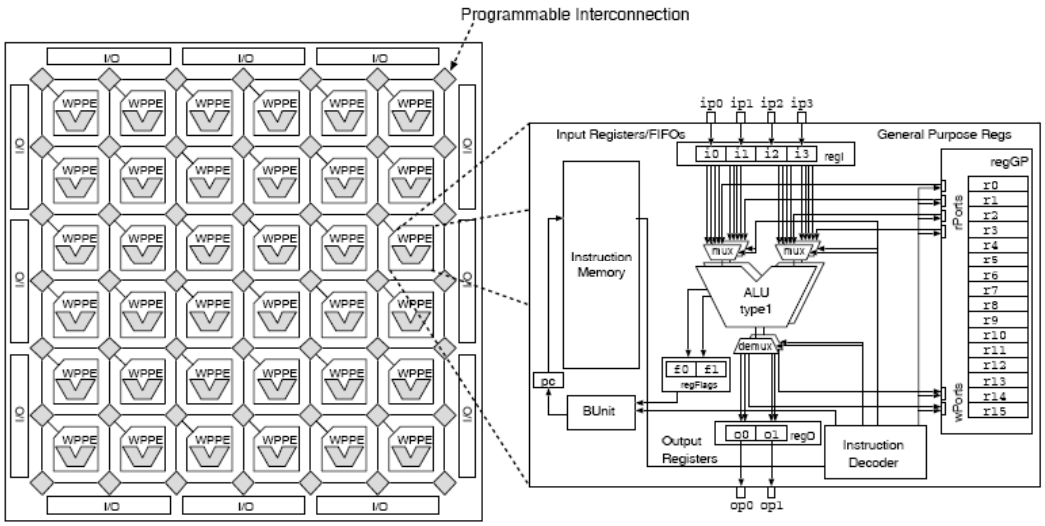



FIGURE 2.8 – *Architecture WPPA et structure d'un WPPE [57]*

La spécification de l'architecture se base sur un langage appelé MAML (*MAchine Markup Language*) qui a été modifié et étendu. Afin d'évaluer différentes architectures de processeurs, une simulation au niveau RTL est réalisée.

L'environnement MMAlpha [30], développé à l'IRISA et basé sur le langage fonctionnel Alpha, permet de manipuler des systèmes structurés d'équations récurrentes (uniformes et affines), en s'aidant d'une boîte à outils de transformations de programme (uniformisation, ordonnancement, allocation, etc.). Sous certaines restrictions, l'outil est capable de générer semi-automatiquement, à partir d'une spécification de boucle en Alpha, à la fois une description matérielle (en VHDL) d'un réseau de processeurs (unidimensionnel ou bidimensionnel) et de son interface logicielle/matérielle. L'outil a par ailleurs été validé sur de nombreuses applications, essentiellement issues du traitement du signal.

Dans [41], les auteurs présentent un environnement appelé CCSIMD ( *Concurrent Communication and Computation Framework for SIMD*) qui tend à explorer certaines alternatives SIMD afin de trouver le modèle de communication le plus efficace, pour les communications régulières et irrégulières, minimisant le temps de communication. Cet outil s'arrête au niveau simulation et ne permet pas la génération ou l'implémentation matérielle de l'architecture adéquate.

Dans le cadre de la conception à base d'IPs, notons l'environnement IPCHINOOK [23]. C'est un outil de synthèse pour les systèmes embarqués distribués. Il est orienté vers la réutilisation de composants. L'entrée de IPCHINOOK est une description comportementale, une description de l'architecture cible et une fonction d'allocation qui définit les relations entre les deux descriptions. La description comportementale contient plusieurs modules concurrents et communicants appelés *modal processes*. La description de l'architecture cible décrit les processeurs, les entrées/sorties, les bus de communication et la topologie du système cible. La fonction d'allocation décrit l'affectation des *modal processes* aux processeurs du système. Cette description structurée en trois parties permet de changer une partie indépendamment des deux autres. Une phase de synthèse consiste à transformer la représentation de haut niveau vers une description plus proche de l'implémentation. Cette synthèse comprend deux étapes : la synthèse du mode-manager et la synthèse des communications et des interfaces. La synthèse des communications permet d'explorer plusieurs affectations des tâches, plusieurs topologies de bus et plusieurs protocoles de communication ; ce qui est nécessaire pour trouver une solution efficace qui vérifie les contraintes. L'approche proposée permet de synthétiser les communications d'un système en utilisant une bibliothèque décrivant les protocoles que l'on souhaite pouvoir utiliser. De plus, l'utilisation de technique de routage permet à tout processeur du système de communiquer avec tous les autres. Ces deux techniques doivent permettre de pouvoir utiliser cette méthode pour réaliser des systèmes dont les architectures sont complexes comme celles comportant des hiérarchies de bus. Cependant, la description du système est très singulière et fort complexe. Le comportement des tâches y est décrit comme des gestionnaires d'évènements. Cette description semble inadaptée par exemple à la description de systèmes de type flot de données.

Plusieurs autres environnements, dédiés pour la conception et la synthèse de systèmes multiprocesseurs sur puce existent. Citons à titre d'exemple les travaux présentés dans [25] développant une architecture appelée FAUST (*Flexible Architecture of Unified System for Telecom*) basée sur un réseau sur puce. Ces travaux permettent de concevoir des architectures sur puce connectant un ensemble d'IPs hétérogènes.

Dans [65], les auteurs montrent la conception d'une plateforme SIMD multimédia sur puce. La plateforme est composée d'un cœur PLX, un contrôleur RISC 8051 8-bits pour gérer les entrées/sorties et un accélérateur matériel H.264. PLX a une unité fonctionnelle SIMD supportant des données de tailles 8, 16, 32 et 64 bits. Un prototype virtuel au niveau SystemC est développé pour la vérification de la plateforme et une méthodologie de génération de code SIMD est proposée. Le prototype SystemC aide le concepteur à modéliser conjointement le logiciel et le matériel au niveau système. Le cœur PLX est implémenté comme un ISS. Les modules H.264 sont implémentés au niveau transactionnel. Ce prototype permet de tester l'application et son comportement temps réel avant de procéder à la conception matérielle.

D'autres environnements de développement sont proposés pour la programmation des applications de traitement de signal et exécution sur des systèmes sur puce. Un flot de conception des applications data parallèles de TS sur des architectures multiprocesseurs sur

puce à base de NoC est présenté dans [64]. L'architecture multiprocesseur se compose d'un ensemble d'IPs processeurs MicroBlaze connectés par un réseau de topologie maille et sont interfacés à base du protocole OCP-IP.

Nous voyons alors que divers environnements existent mais qui tiennent compte de la programmation sur des architectures parallèles, tout en supposant que l'architecture matérielle cible existe, ou la synthèse de systèmes multi-cœurs sur puce. Malgré la multitude de travaux intéressés à faciliter la conception et la génération d'architectures multiprocesseurs sur puce, moins sont ceux qui considèrent les architectures SIMD sur puce. Peu de travaux exploitent l'utilisation d'IP pour la construction d'un système massivement parallèle sur puce. La plupart des outils [60, 114, 49] développés dans ce contexte sont plutôt proposés afin de faciliter la parallélisation de codes supportant des instructions SIMD. Ils consistent généralement à étendre un compilateur spécifique pour supporter des instructions SIMD sur une architecture cible.

Suite à cet état de l'art, plusieurs questions peuvent se poser. Parmi les quelles, la méthodologie de construction d'une architecture SIMD présente une question fondamentale d'autant plus dans un contexte d'assemblage d'IPs.

## 2.5 Quelle méthodologie d'assemblage ?

Afin de définir notre méthodologie facilitant la conception d'un système SIMD, nous avons examiné les travaux qui proposent un assemblage de composants. Le fameux exemple est celui de la MasPar qui peut comporter de 1024 à 16384 PEs pour la MP1 [75]. La machine est extensible à travers une méthodologie de réplication des éléments clés à savoir les PEs qui sont de simples unités d'exécution (sans mémoire d'instructions). L'ACU est implémenté sur un circuit imprimé. Le réseau de processeurs est extensible de 1 jusqu'à 16 puces. Chaque puce comporte 1024 PEs organisés en 64 groupes ou *clusters* de 16 PEs chacun. Les PEs sont connectés par un réseau de voisinage Xnet et un réseau global multi-étages. Deux réseaux de diffusion et de réduction connectent l'ACU avec le réseau de PEs. Une puce MP1 contient 32 PEs (deux *clusters*) consommant 450.000 transistors. Un système MP1 peut être alors étendu par l'ajout de circuits de PEs. Cela implique une augmentation en nombre de PE, taille mémoire et ressources de communication. L'ajout de circuits n'est pas si facile et fait face à plusieurs problèmes tels que la distribution d'horloge, la diffusion synchrone des instructions, la complexité des communications, etc. Les inter-connecteurs utilisés dans la Maspar sont bien conçus afin de ne pas dégrader les performances en augmentant la taille du système. La soustraction de circuit est de même difficile vu qu'il y aura des trous dans le design à combler.

Symphonie (SYstème Massivement Parallèle à Haut Niveau d'Intercommunication Embarqué) est un autre exemple de système extensible [26]. Ce calculateur parallèle SIMD peut intégrer de 32 à 1024 processeurs super-scalaires organisés en anneau. En effet, quatre processeurs et 64 KOctets de mémoire sont intégrés sur une seule puce et quatre puces sont intégrées par module multi-puces (*MCM : Multi-Chip Module*). Seize processeurs et 256 KOctets de mémoire sont ainsi intégrés dans un seul boîtier.

Si nous examinons de même le système super-extensible de Cray, le Cray 3/SSS (*Super Scalable System*), nous remarquons qu'il se compose de processeurs simples 1bit associés à chaque colonne d'une RAM dans une puce propriétaire appelée PIM (*Processing-In-*

*Memory*). Ces puces constituent une partie de l'espace d'adressage mémoire du système. Cela semble être une approche prometteuse pour une extensibilité de plus d'un millions de processeurs. La technologie PIM permet de réduire le gap entre la vitesse des mémoires et celle des processeurs dans les applications de données intensives. Cela consiste à intégrer le processeur sur puce utilisant des mémoires DRAM. L'intégration de la mémoire et du processeur sur la même puce permet de diminuer la latence de la mémoire et améliore les transferts de données entre les deux composants.

Suite à cet état de l'art, nous avons dégagé plusieurs problèmes et limites rencontrés dans les architectures SIMD existantes. Afin d'atteindre de bonnes performances pour une large gamme d'applications de TSS, il faut assurer un minimum de flexibilité et extensibilité de l'architecture SIMD.

Dans le système Ter@Core [14], les PEs sont conçus à partir des DSP incorporés dans l'FPGA Virtex4 SX55. Ceci limite le nombre de PEs intégrés dépendamment du nombre de blocs DSP et fait que le design n'est pas portable pour une future implémentation sur un autre FPGA. Dans cette architecture SIMD, le réseau de voisinage est fixe ayant une topologie en anneau. Un réseau en anneau n'est pas toujours en mesure de satisfaire les communications inter-PE d'une manière efficace. Ceci peut ralentir l'exécution de certains algorithmes (multiplication de matrices, convolution, etc.). Les questions qui se posent alors sont :

> $Q_1$ : Comment rendre le système matériel en adéquation avec une application donnée ? Comment un système SIMD peut s'adapter aux besoins applicatifs ?

Dans l'architecture SIMD décrite dans [95], les PEs sont conçus à partir d'IP processeur disponible ce qui a largement réduit le temps de conception. Les PEs ont été réduits afin de n'opérer que sur des données 1-bit. Cette manière de réduction n'est pas assez efficace vu la limite du jeu d'instructions. De plus, il s'avère que les PEs doivent être suffisamment performants pour répondre à la complexité des applications de TSS. En effet, les applications de TSS visées nécessitent de faire des calculs sur des variables de taille raisonnable contrairement à des applications de traitement d'images binaires. Pour de nombreuses raisons le gros grain est plus performant que le grain fin [54]. En considérant les calculs élémentaires par exemple, un processeur 32-bits produit une multiplication de deux nombres de 32-bits par cycle alors qu'il faudra 1024 cycles pour faire la même opération sur un processeur 1-bit. Nous avons de même constaté que la spécificité de la conception des PEs oblige le développement de nouvelles instructions SIMD avec un jeu d'instructions différent du processeur de contrôle. Ceci complexifie la programmation et le développement d'un compilateur approprié. Une des solutions est de garder le même jeu d'instructions tant pour le PE que pour l'ACU. Dans ce même contexte, le jeu d'instructions du réseau SIMD dans l'architecture PoD [112] se base sur le jeu du processeur hôte ce qui a beaucoup simplifié la conception. Les questions qui se posent alors sont :

> $Q_2$ : Quelle méthode de réduction à base d'IP opter pour concevoir un PE ? Comment être le moins intrusif possible pour diminuer le temps de conception ?

Dans les travaux [63, 48], les architectures SIMD ne supportent pas de communications inter-PE. Ceci présente une grande limite vis à vis d'une multitude d'applications multimédia. La conception des PEs au sein de ces architectures est spécifique et propriétaire. Cela demande un temps de développement assez long. C'est pour cette raison, nous allons procéder à une conception à base d'IP afin d'accélérer la construction d'un PE et de toute l'architecture SIMD.

Dans plusieurs architectures SIMD telle que Morphosys [50], les communications irrégulières ne sont pas gérées. Ceci pose une limite concernant les communications point à point et en particulier les transferts de données parallèles.

$Q_3$ : Quel dispositif intégrer pour gérer les communications irrégulières ? Comment assurer les entrées/sorties parallèles ?

La majorité des architectures SIMD citées précédemment sont efficaces et spécifiques pour une application bien particulière. De plus, elles ne sont ni évolutives ni extensibles pour répondre aux différents besoins applicatifs. Or, le choix d'une architecture parallèle efficace repose sur le choix du nombre de PEs à mettre en œuvre et celui du réseau d'interconnexion leur permettant d'échanger les données à traiter [71]. Ceci nous mène à poser la question ci dessous :

$Q_4$ : Comment peut-on faire face aux limites des architectures SIMD existantes et à leur manque de flexibilité ? Quels sont les paramètres à introduire dans une architecture SIMD ?

En analysant les divers environnements de conception proposés pour les systèmes sur puce, nous remarquons que ceux à base d'IPs accélèrent et facilitent la conception. Cependant, ils sont destinés essentiellement pour des systèmes multiprocesseurs sur puce. La majorité des systèmes SIMD sont conçus manuellement et nécessitent un long temps de développement. La problématique est alors :

$Q_5$ : Comment faciliter et accélérer la conception d'une architecture SIMD ? Quel flot de conception facile et simple à opter pour de tels objectifs ?

La suite de ce document tends à répondre progressivement à ces interrogations et proposer des solutions pour les problématiques évoquées précédemment.

## 2.6 Conclusion

Dans ce chapitre, nous avons présenté les travaux qui sont en relation avec les architectures SIMD massivement parallèles sur puce. Notre étude a montré certaines limites dans ces architectures en termes de performance, de flexibilité et d'adéquation avec différentes applications. Nous avons aussi abordé les méthodologies de conception proposées afin de faciliter et accélérer la conception des systèmes sur puce. Nous avons remarqué que la plupart des méthodologies de conception d'architectures SIMD sont propriétaires et spécifiques

soit pour une application donnée, soit pour une architecture particulière bien définie et non évolutive. Ce chapitre a conduit à se poser diverses questions suite aux problématiques dégagées. Cela rejoint parfaitement la problématique énoncée dans le premier chapitre.

Notre travail apporte une réponse pour ces différentes questions. Nous nous intéressons à proposer une méthode de conception rapide d'une architecture SIMD massivement parallèle en se basant sur un assemblage de composants. Cette approche a plusieurs avantages facilitant la conception et réduisant le temps de mise sur le marché. De plus, nous visons à proposer une architecture qui soit flexible et paramétrique afin de s'adapter à une application donnée. Plusieurs raisons militent en faveur d'une approche où le système peut être taillé pour une application donnée. Le modèle de notre architecture parallèle sur puce sera introduit dans le chapitre suivant.

# Chapitre 3

# Modèle mppSoC : architecture SIMD paramétrique et générique


**3.1 Définition du modèle mppSoC . . . . . 38**
**3.2 Description de l'architecture de mppSoC . . . . . 41**
3.2.1 Le processeur de contrôle : ACU . . . . . 41
3.2.2 Interface ACU - PE . . . . . 41
3.2.3 Le réseau de PEs . . . . . 41
3.2.4 Le global routeur : mpNoC . . . . . 42
**3.3 Analyse du modèle architectural de mppSoC . . . . . 44**
3.3.1 Aspect modulaire . . . . . 44
3.3.2 Aspect paramétrique et flexible . . . . . 45
3.3.3 Aspect programmable . . . . . 50
**3.4 Validation du modèle mppSoC . . . . . 52**
3.4.1 Algorithme de multiplication de matrices parallèle . . . . . 52
3.4.2 Algorithme de réduction : Somme de N valeurs sur P processeurs . . 55
**3.5 Comparaison du modèle mppSoC avec d'autres modèles SIMD . . . . . 57**
**3.6 Conclusion . . . . . 58**

Les chapitres précédents ont montré l'intérêt des architectures SIMD à efficacement exécuter des applications de TSS. Une analyse de l'état de l'art a dévoilé les limites des architectures SIMD existantes en termes de conception et adéquation avec différentes applications. En effet, l'architecture doit pouvoir répondre aux besoins de l'application pour être efficace. Ces besoins varient d'une application à une autre bien qu'elles appartiennent au même domaine. Proposer un système massivement parallèle sur puce qui peut être adapté aux exigences de l'application s'avère alors d'une importance majeure. C'est dans ce cadre que se situe le projet mppSoC. Ce projet s'oriente vers la définition et la conception d'un système SIMD appelé *mppSoC*. Notre objectif est de reconsidérer l'intérêt des architectures massivement parallèles en exploitant les nouvelles méthodologies d'intégration basées sur la réutilisation d'IPs. Nous visons de même un système qui soit flexible, paramétrique et configurable permettant de couvrir une variété d'applications de traitement de signal systématique.

Ce chapitre a pour but essentiel de définir le modèle général de mppSoC et est structuré comme suit. La section 2 décrit le modèle mppSoC en se basant sur son flot d'exécution. L'architecture de mppSoC est détaillée dans la section 3. Une analyse du modèle de mppSoC est présentée dans la section 4. Ce modèle paramétrique est justifié et validé dans la section suivante. La section 6 compare entre le modèle mppSoC et d'autres modèles d'architectures SIMD sur FPGA. Enfin, la section 7 présente une synthèse de ce chapitre et donne un aperçu sur la méthode de conception du système mppSoC.

## 3.1 Définition du modèle mppSoC

Dans ces travaux, nous proposons un système programmable multi-applicatifs reposant sur une architecture SIMD massivement parallèle. Rappelons les trois verrous principaux que ce système vise à lever et à concilier :

- une solution réutilisable multi-applications dans le domaine de TSS ;
- une architecture extensible massivement parallèle ;
- un système programmable sur puce efficace en coût de développement.

Afin de concilier ces différents objectifs, nous avons défini une architecture SIMD massivement parallèle autonome, nommée mppSoC, acronyme de *massively parallel processing System on Chip*, dans laquelle les applications sont implémentées au moyen d'un modèle d'exécution unifié. Cette architecture est flexible (programmable et reconfigurable) pour une approche multi-applications dans le domaine de TSS.

MppSoC est un système SIMD massivement parallèle sur puce, inspiré des systèmes SIMD traditionnels plus particulièrement la MasPar [13] et conçu pour être implémenté sur des circuits FPGA. Un modèle typique du système mppSoC est illustré dans la figure 3.1. Ce modèle se caractérise par :

- un contrôle centralisé : un processeur principal appelé *ACU* présente le processeur performant qui contrôle tout le système et effectue les traitements séquentiels. Dans ce modèle, on trouve une seul unité de contrôle qui gère le flot d'instructions.
- un réseau de processeurs élémentaires dénotés PE : chaque PE présente une unité d'exécution simple chargée d'exécuter les traitements parallèles. Dans ce modèle, les PEs sont alors dupliqués pour former le réseau massivement parallèle. Le nombre de PEs est paramétrique.

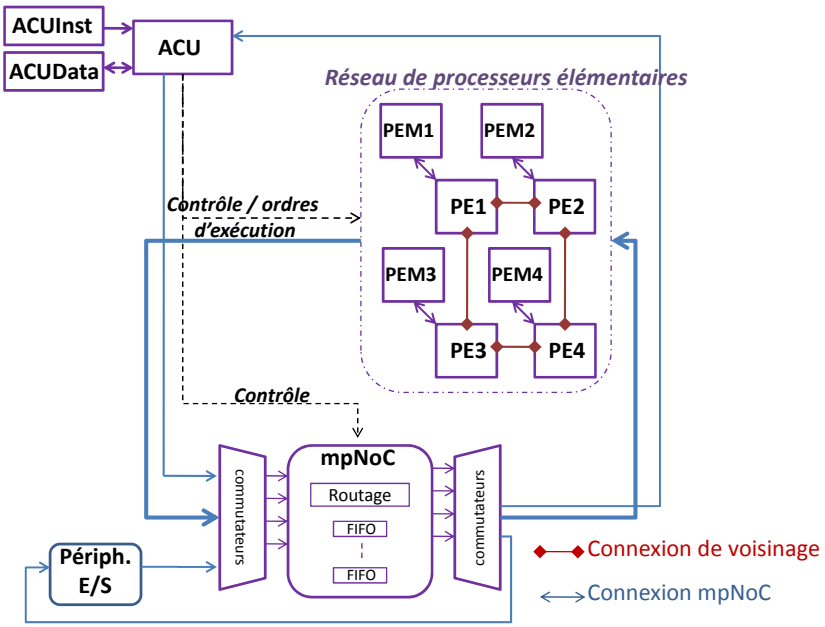



FIGURE 3.1 – *Modèle architectural de mppSoC*

– une mémoire séquentielle : l'ACU est connecté à une mémoire de programmation centralisée. Cette mémoire peut être divisée en deux mémoires distinctes : une mémoire d'instructions (dénotée ACUInst dans la figure 3.1) et une mémoire de données (dénotée ACUData dans la figure 3.1). La mémoire d'instructions contient le programme data parallèle. La mémoire de données contient les données sur les quelles opère l'ACU. La connexion entre l'ACU et sa mémoire d'instructions est modélisée par un lien unidirectionnel permettant la lecture des instructions. Cependant, la connexion entre l'ACU et sa mémoire de données est modélisée par un lien bidirectionnel permettant la lecture/écriture des données. Toutefois, le concepteur a le choix d'intégrer une seule mémoire qui contient à la fois données et instructions.
  Un programme data parallèle est composé d'instructions parallèles et d'instructions séquentielles. L'ACU exécute toute instruction séquentielle et diffuse les instructions arithmétiques parallèles et de traitement de données au réseau de PEs.
– une interface de distribution des ordres d'exécution : l'ACU est responsable de diffuser les traitements parallèles au réseau de PEs. De ce fait, une interface de diffusion des ordres d'exécution relie l'ACU aux PEs. Cette interface est modélisée par un bus reliant l'ACU à tous les PEs du système.
– une mémoire distribuée parallèle : chaque PE est connecté à sa propre mémoire locale de données. Il traite simultanément, avec le même ordre d'exécution reçu de l'ACU, une partie de données qui lui est propre. De ce fait, un parallélisme de données est assuré. Chaque PE va ensuite stocker son résultat dans sa mémoire. La connexion entre le PE et sa mémoire est modélisée par un lien bidirectionnel assurant les opérations de lecture/écriture.
  L'exécution parallèle s'effectue d'une manière synchrone.
– une communication de voisinage synchrone : les PEs sont inter-connectés par un réseau de voisinage. Ce dernier est modélisé par des liens bidirectionnels connectant les

PEs ensemble. En effet, le réseau de voisinage connecte chaque PE avec un nombre de ses voisins au sens géométrique déterminé selon la topologie du réseau. La structure SIMD est généralement caractérisée par une topologie régulière, c'est-à-dire que la règle de connexion d'un PE est la même pour tous les PEs (modulo les bords).
Les communications inter-PE s'effectuent d'une manière synchrone. Elles sont bidirectionnelles et gérées par des instructions de communication.

- une communication irrégulière : un réseau, appelé *mpNoC : massively parallel Network on Chip*, assure les communications point à point entre les divers composants du système (ACU, PEs et périphériques). Il est alors connecté à ces composants via des liens de communication point à point unidirectionnels. Ces liens connectent les composants en entrée et en sortie du mpNoC.
  Les communications via le mpNoC sont aussi gérées par des instructions de communication.
- un mécanisme d'activation des PEs : l'exécution des PEs est conditionnée par un bit d'activité. Ce bit est contrôlé par l'ACU à travers des instructions de contrôle. Un PE peut être en effet soit actif soit inactif ; seulement les PEs actifs exécutent les instructions alors que les autres sont à un état de repos. Ce minimum d'autonomie est généralement utile pour permettre l'exécution des instructions conditionnelles.
- un mécanisme d'identification des PEs : chaque PE est aussi identifié dans le système par son identité ou numéro.
- un mécanisme de Or Tree : ce mécanisme de OU global sert de test pour un contrôle de l'état du réseau de PEs en manipulant le bit d'activité. Il permet à l'ACU de savoir en permanence si au moins un des PEs est actif. Il vaut 0 si et seulement si tous les PEs sont inactifs. L'Or Tree est réalisable par un arbre de portes logiques OU faisant remonter l'information de chaque PE, d'où son nom. Son utilisation est indispensable pour la détection d'états stables ou encore de points de convergence dans beaucoup d'applications : réseaux de neurones, algorithmes de squelétisation, etc.

Nous voyons que le modèle mppSoC est simple et facile à concevoir. Une telle organisation SIMD permet une grande simplicité dans l'implémentation de la partie contrôle qui est unique alors que la partie exécutive est répliquée.

Pour faire face aux limites des architectures SIMD évoquées à la fin du chapitre 2 et en répondant en partie à la question $Q_4$, nous proposons le modèle mppSoC. Ce modèle est générique, à la base du quel plusieurs configurations SIMD possibles (instances du modèle mppSoC) peuvent être construites. Pour modéliser une configuration mppSoC, l'utilisateur a besoin de définir les paramètres et les entités présentes dans le modèle et les interconnecter ensemble. En effet, une instance mppSoC peut aller d'une simple configuration composée d'un processeur de contrôle guidant un simple réseau paramétrique de PEs, à une configuration où les PEs sont inter-connectés par le biais d'un réseau de voisinage configurable, jusqu'à une configuration SIMD intégrant de plus un réseau de communication globale assurant ainsi les communications point à point entre les divers composants. De ce fait, le modèle mppSoC peut être adapté selon l'application.

Une description plus détaillée de l'architecture est présentée dans la section suivante.

## 3.2 Description de l'architecture de mppSoC

Le modèle mppSoC peut donner naissance à différentes configurations SIMD qui se caractérisent par la nature de ses composants intégrés ainsi que ses interconnexions. Dans ce qui suit, nous mettons en évidence les points clés de l'architecture de mppSoC tout en faisant abstraction des détails d'implémentation.

### 3.2.1 Le processeur de contrôle : ACU

L'ACU est le processeur principal qui contrôle toute l'architecture. C'est un élément de base qui doit toujours exister dans une configuration mppSoC. Il est le seul processeur qui a accès à la mémoire d'instructions. L'ACU peut être connecté à deux mémoires différentes : une pour stocker les instructions et une pour les données ; ou bien à une seule mémoire qui stocke à la fois données et instructions. Ce choix est laissé libre pour le concepteur et est possible grâce à la modularité de mppSoC. Ces deux cas varient dans la manière de connexion de l'ACU. En effet, si le concepteur intègre deux mémoires, l'ACU sera connecté seulement en mode lecture à la mémoire d'instructions et en mode lecture/écriture à la mémoire de données. Cependant dans le cas d'une unique mémoire, l'ACU sera connecté en mode lecture/écriture. L'ACU est chargé de lire le code à exécuter, de décoder les instructions, puis de les exécuter s'il s'agit d'instructions séquentielles sinon les diffuse à tous les PEs dans le cas des instructions parallèles sous forme d'ordres d'exécution (instructions ou micro-instructions ; voir détails dans le chapitre 4). L'ACU peut aussi communiquer (envoie ou lecture de données) avec n'importe quel PE ou autre composant de mppSoC à travers le mpNoC. Pour cela, une connexion entre l'ACU et le mpNoC doit être établie. Cette connexion est assurée par deux liens de communication unidirectionnels (lecture/écriture) entre l'ACU et le mpNoC.

### 3.2.2 Interface ACU - PE

L'ACU est connecté aux PEs à travers une interface de diffusion des ordres d'exécution. Cette interface est un bus communiquant les ordres d'exécution, provenant de l'ACU, à tous les PEs. L'ACU pilote tout le système mppSoC. Il est responsable d'activer les PEs et leur envoyer les traitements parallèles à effectuer. Chaque PE reçoit donc simultanément le même ordre que les autres. L'ordre indique directement l'action qui sera exécutée par tous les PEs sur leurs données locales. L'OR Tree sert aussi à connecter les PEs à l'ACU. Il s'agit d'un arbre de portes OU logiques effectuées sur les signaux de bit d'activité des PEs et est connecté en entrée à l'ACU.

### 3.2.3 Le réseau de PEs

Le réseau de PEs constitue le cœur d'exécution du système mppSoC. Il consiste à répliquer un PE un nombre paramétrique de fois, chacun est connecté à sa propre mémoire de données. Les PEs sont guidés par l'ACU et n'exécutent que les instructions parallèles qui leur y sont envoyées. À chaque instant, tous les PEs actifs exécutent exactement la même instruction, mais ne possèdent pas les mêmes données. En effet, chaque PE possède son propre jeu de registres ainsi que sa mémoire de données. Du point de vue accès mémoire, les PEs ne disposent pas d'autonomie d'adressage. En effet, lors d'un accès à la mémoire locale, les

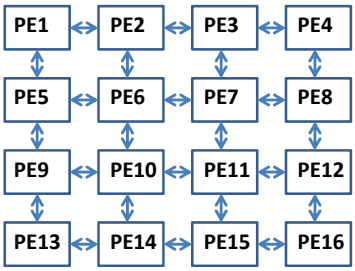



FIGURE 3.2 – *Connexions de voisinage entre les PEs dans une topologie maille*

PEs accèdent tous aux mêmes adresses. Chaque PE possède un identifiant unique le distinguant qui peut s'avérer très utile pour divers calculs. Les instructions parallèles sont toutes les instructions qui ne manipulent pas le compteur du programme (*PC : Program Counter*) (que seul l'ACU possède). Nous nous sommes inspirés principalement du jeu d'instructions MasPar pour décider des instructions parallèles à étendre.

Les PEs peuvent être reliés entre eux grâce à un réseau de voisinage. Ce réseau présente une structure régulière ayant une topologie paramétrique. Ces topologies diffèrent par la nature des connexions reliant les PEs. Les connexions de voisinage sont assurées par des bus bidirectionnels pour le transfert de données entre les PEs voisins. La figure 3.2 illustre les interconnexions reliant les PEs, placés dans un réseau 4x4, dans le cas d'une topologie maille. Les communications de voisinage sont gérées par des instructions de communication et s'effectuent d'une manière synchrone. Il en résulte qu'à un instant donné, tous les PEs exécutent la même instruction de communication parallèle avec leurs voisins situés dans la même direction. Une instruction de communication est de type lecture/écriture mémoire (*Load/Store*) de/vers une adresse mémoire du PE voisin désigné par sa direction.

Chaque PE peut aussi réaliser une communication point à point avec un autre PE ou avec un périphérique à travers le mpNoC qui est dédié au trafic de données intensif. Dans ce cas, les PEs doivent être connectés en entrée et en sortie du mpNoC.

### 3.2.4 Le global routeur : mpNoC

Les communications globales sont menées au sein de mppSoC via le réseau mpNoC. Ce réseau connecte potentiellement chaque PE à un autre assurant ainsi une communication irrégulière effective. Il assure de même les communications point à point entre les autres composants de mppSoC à savoir l'ACU et les périphériques (figure 3.3). MpNoC accomplit, en effet, trois principales fonctions :

- connecter en parallèle tout PE avec un autre ;
- connecter les PEs aux périphériques assurant des transferts de données parallèles ;
- connecter l'ACU avec n'importe quel PE.

Chaque PE est relié en lecture comme en écriture au réseau. Lors d'une communication, chaque PE sera en mesure d'envoyer une donnée à un autre PE. Puis chaque PE receveur (qui était éventuellement aussi envoyeur) pourra réceptionner la donnée. En utilisant le mpNoC, il est possible d'intégrer une gestion des accès des PEs à divers périphériques. L'idée est de faire partager à tous les PEs leur point d'entrée/sortie sur le mpNoC avec un périphérique. MpNoC est un réseau souple et d'une importance majeure permettant de réaliser tout type de communication, particulièrement des transferts de flots de données avec les

périphériques. Il permet d'assurer trois modes de communications bidirectionnels à savoir :

- mode PE - PE ;
- mode ACU - PE ;
- mode PE - Périph E/S.

Nous pouvons constater qu'il n'existe pas de mode ACU vers périphériques. Nous considérons en fait que l'ACU accède directement au périphérique. En effet, une connexion devra dans tous les cas exister entre ces composants, car le périphérique a besoin d'être piloté par l'ACU pour ses accès au mpNoC. La gestion du mode d'envoie se fait à travers un gestionnaire de mode (*mode manager*) qui permet d'activer les connexions désirées. Une fois un mode établi, les communications sont gérées par le réseau d'interconnexion qui peut avoir différent types dans mpNoC. Par ailleurs, le mpNoC a un fonctionnement indépendant du type de réseau utilisé. Les communications irrégulières sont aussi gérées par des instructions de communication.

La question $Q_3$ posée à la fin du chapitre 2 de l'état de l'art exprime le besoin d'intégrer un dispositif assurant les communications irrégulières au sein d'une architecture SIMD. Nous proposons alors le mpNoC comme solution pour ce type de communications. En effet, le mpNoC est le réseau massivement parallèle qui peut être intégré dans mppSoC afin d'assurer les communications point à point entre tous les composants. Il peut fonctionner selon divers modes de communication (tous-vers-un, un-vers-tous ou un-vers-un). Ce réseau gère les communications entre les périphériques d'entrée/sortie et les processeurs (ACU/PE). Il peut ainsi assurer les entrées/sorties parallèles.

Le modèle mppSoC est caractérisé par sa flexibilité. Il peut être adapté afin de couvrir diverses applications de TSS. Cette flexibilité est assurée à travers la paramétrisation de l'architecture, son extensibilité et sa reconfigurabilité. En bref, le modèle architectural générique de notre système mppSoC est :
- modulaire pour maîtriser la complexité ;
- flexible pour s'adapter rapidement à diverses applications de TSS et à leurs évolutions ;
- extensible pour traiter une large classe d'applications de différentes tailles ;
- conçu d'une manière systématique pour réduire le temps de mise sur le marché.

Ce modèle permet alors de faire face aux limites des architectures SIMD existantes et à leur manque de flexibilité, exprimé par la question $Q_4$ présentée à la fin du chapitre de l'état de l'art. Les paramètres introduits dans le système mppSoC peuvent être classés en deux groupes : des paramètres généraux et des paramètres spécifiques. Alors que le premier type de paramètres peut être librement choisi (nombre de PEs, taille mémoire), le second type doit être sélectionné parmi une liste de choix offerte (topologie du réseau de voisinage, type de réseau de mpNoC, etc.). Néanmoins, ceci ne limite pas la flexibilité de mppSoC qui peut être encore plus étendue.

Une analyse des différentes caractéristiques du modèle mppSoC est présentée dans la section qui suit.

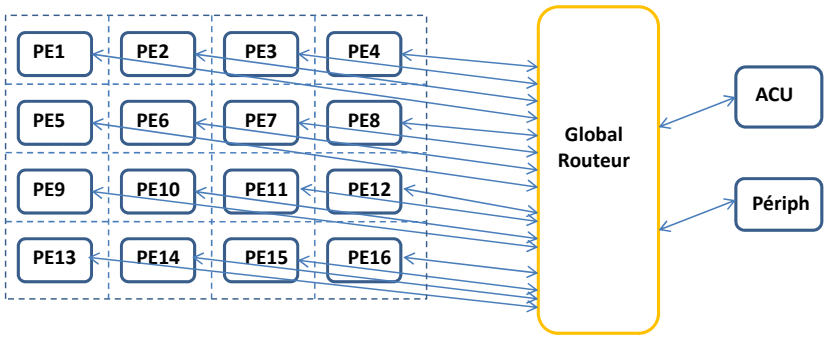



FIGURE 3.3 – *Schéma général de mpNoC*

## 3.3 Analyse du modèle architectural de mppSoC

Le système mppSoC est conçu d'une manière simple et efficace afin d'offrir les performances nécessaires aux applications data parallèles. Il est aussi portable et indépendant de la technologie. De ce fait, le système mppSoC peut être implémenté sur n'importe quelle FPGA. Ce système se caractérise par sa modularité, sa paramétrisation et sa programmabilité. Ces différents aspects seront détaillés dans les paragraphes suivants.

### 3.3.1 Aspect modulaire

La modularité est obtenue grâce à un assemblage de différents composants pour former un système mppSoC. Les composants, tels que définis dans le paragraphe précédent, incluent les processeurs (ACU et PE), les mémoires et les réseaux de communication (réseau de voisinage et mpNoC). En effet une configuration mppSoC, instance du système général, peut aller d'une simple configuration parallèle incluant juste un ACU avec un ensemble de PE à une configuration assurant de plus les communications inter-PE jusqu'à finalement une configuration complète avec des communications point à point entre les différents composants.

Cet aspect modulaire est bien justifié. En effet, la modularité favorise la conception à base de composants du système. Elle facilite de même l'extensibilité de mppSoC. Vu que nous ciblons une implémentation sur puce, il est dans ce cas primordial d'optimiser l'architecture et n'inclure que les composants nécessaires pour une application donnée. Choisir d'intégrer un seul, les deux ou aucun réseau de communication offre au concepteur plusieurs configurations SIMD dont il peut sélectionner une qui satisfait ses contraintes et répond à ses besoins. Cette liberté dans le choix d'intégration d'un réseau de communication (voisinage ou mpNoC) convient à une diversité des besoins des applications de TSS. Certaines applications par exemple n'ont besoin que d'effectuer des communications régulières, tel est l'exemple des algorithmes de convolution, filtrage, etc. Ajoutons de plus que la réalisation d'une communication globale est coûteuse et forme parfois une barrière importante pour l'intégration de plusieurs processeurs sur une même puce ; tel est le cas de la Connection Machine qui intègre 16 PEs sur une même puce. Une communication globale, sur un hypercube, est reportée à un organe externe. Par ailleurs, nous trouvons d'autres algorithmes élémentaires de traitement d'images qui nécessitent des communications point à point. Ces communications impliquent alors le transfert de données d'une manière non régulière entre les PEs d'où la

nécessité de mpNoC. Un algorithme de rotation d'images par exemple consiste à échanger les pixels entre les PEs d'une manière irrégulière afin de former l'image résultante. Le réseau mpNoC ne se limite pas à assurer ce type de communications mais il est aussi vital pour réaliser les transferts de données parallèles et gérer les entrées/sorties parallèles. En effet, pour un bon nombre d'applications data parallèles les performances globales des systèmes sont fortement limitées faute d'un transfert suffisamment rapide entre les unités de calcul et les dispositifs de stockage. MpNoC permet d'assurer un transfert parallèle de données. Il est aussi extensible et s'adapte à la taille du réseau de PEs.

Cette qualité modulaire du modèle architectural permet d'inclure les seuls composants dont on a besoin pour une application donnée. Elle est indispensable pour maîtriser la complexité des systèmes surtout dans le cadre d'une implémentation sur puce. Notre objectif est de concevoir une machine SIMD ad-hoc en fonction de l'application. À partir d'une architecture générique, nous pourrons déduire une solution taillée selon les besoins d'un ensemble d'applications en terme de ressources de calcul, de mémorisation et de communication. Il est à noter que la modularité de mppSoC est possible grâce à une méthode de conception systématique qui s'appuie sur un assemblage de composants. Cette méthode fera l'objectif du chapitre suivant.

### 3.3.2 Aspect paramétrique et flexible

La flexibilité est définie comme la facilité ou la possibilité d'adaptation du système aux besoins applicatifs. Elle est obtenue grâce à l'utilisation de modèles génériques et paramétriques et grâce à la modularité et la méthode de conception suivie. Il n'est pas toujours possible de prévoir toutes les configurations qu'un système devra avoir pour prendre en charge diverses applications. Néanmoins, on pourra satisfaire une large variété d'applications en assurant un minimum de flexibilité.

#### 3.3.2.1 Paramétrisation de mppSoC

MppSoC est paramétrique en termes de nombre de PEs intégrés et la taille des mémoires. Une application peut par exemple nécessiter un grand nombre de PE avec une petite taille mémoire ou bien un petit nombre de PE avec une grande taille mémoire. Les PEs peuvent être agencés en réseau linéaire ou en grille 2D. Si on considère le traitement d'images, il est clair que les données dans ces algorithmes s'expriment bien sous forme d'une matrice. Par conséquent, une grille 2D de PEs semble être la mieux adaptée à la structure d'une image. Par contre, si on se place dans le cas d'un traitement de signal radar, un arrangement linéaire serait plus efficace puisqu'on manipule un tableau de données. Le fait de pouvoir incrémenter le nombre de PEs ainsi que la taille des mémoires fait que le système mppSoC est extensible. Ceci permet d'adapter le modèle mppSoC à des applications de différentes tailles et complexités. La principale restriction d'extensibilité de mppSoC est la disponibilité des ressources au sein d'un circuit FPGA puisqu'on cible une implémentation sur une seule puce.

Le besoin de la paramétrisation de la taille des mémoires vient aussi du fait que la taille de données à traiter dans une application de TSS est variable d'une application à une autre. Les données peuvent être des données binaires, de type caractère, des entiers signés ou non signés, des couleurs codées sur 24 bits, etc. La variété de ces données demande à pouvoir gérer la taille des mémoires afin d'assurer l'efficacité du système.

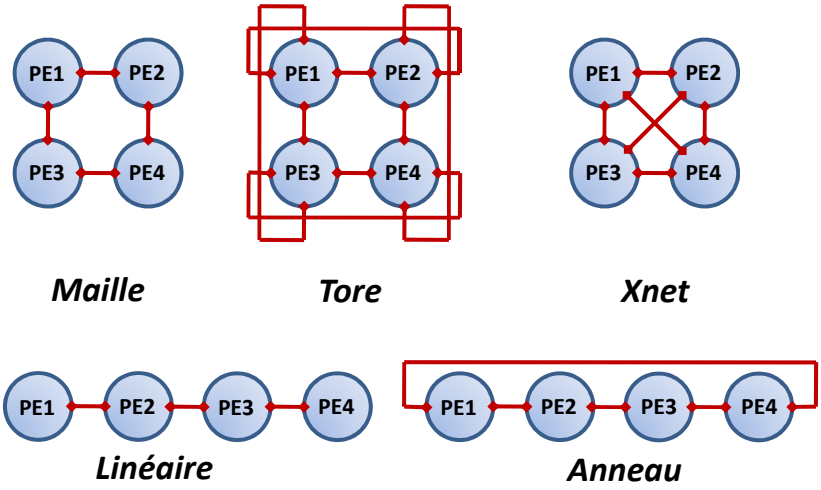



FIGURE 3.4 – *Topologies du réseau de voisinage*

#### 3.3.2.2 Flexibilité de mppSoC

On entend par flexibilité la possibilité de supporter une variété d'applications avec des performances intéressantes. La flexibilité de mppSoC est assurée par sa paramétrisation ainsi que sa facilité de configuration. Elle reste néanmoins limitée par les paramètres définis et les possibilités d'intégration fournies.

MppSoC se caractérise par une flexibilité de communication qui peut varier selon les besoins. En effet, ses réseaux de communication peuvent être de différents types étant donné que :

- le réseau de voisinage : peut avoir une topologie configurée selon les besoins. Différentes topologies, présentées dans la figure 3.4 peuvent être choisies à savoir une topologie anneau ou linéaire dans le cas d'un réseau linéaire de PEs ou bien maille, tore ou Xnet (grille régulière offrant à chaque PE des communications directes avec ses voisins situés sur les diagonales) dans le cas d'une grille 2D de PEs.
- le réseau mpNoC : se base sur un réseau d'interconnexion afin d'assurer les communications irrégulières. Ce réseau peut varier aussi selon les contraintes et besoins et peut aller d'un simple bus hiérarchique jusu'au réseau multi-étages.

Rendre le réseau de voisinage configurable selon les besoins a de grands avantages. En effet, la topologie du réseau d'interconnexion qui relie les PEs entre eux est un critère important. Le réseau de voisinage est un composant particulièrement critique dans la réalisation d'une machine parallèle. Étudier la parallélisation d'un algorithme donné ainsi que les communications nécessaires entre ses différentes parties permet de choisir le réseau le plus adéquat. Dans ce contexte, différentes topologies pour les architectures parallèles se présentent : maille, tore, anneau, réseau linéaire, pyramide, hypercube, etc. La connexion d'une communication locale occupe un pourcentage très significatif de la surface de silicium. Cela limite le nombre de processeurs. Cette raison fait qu'une topologie simple et régulière est en général retenue. Le tableau 3.1 résume les différences entre les topologies régulières extensibles les plus utilisées. Nous nous sommes intéressés aux topologies faisables pour un arrangement de PEs en grille 2D ou en ligne. Dans le tableau 3.1, les $N_k$ correspondent au nombre de processeurs dans la dimension k. La fonction E() est la fonction partie entière. Pour l'arbre, n

TABLE 3.1 – Caractéristiques des topologies régulières extensibles

| Topologie | Processeurs | Liens par processeur ou degré | Routages maximum ou diamètre | Connexions |
|---|---|---|---|---|
| Linéaire | N | 1 à 2 | N-1 | N-1 |
| Anneau | N | 2 | $\frac{N}{2}$ | N |
| Maille | $\prod_{k=1}^{2} N_k$ | 2 à 4 | $\sum_{k=1}^{2} N_k - 2$ | $2.\prod_{k=1}^{2} N_k - \sum_{k=1}^{2} N_k$ |
| Tore | $\prod_{k=1}^{2} N_k$ | 4 | $\sum_{k=1}^{2} E(\frac{N_k}{2})$ | $2.\prod_{k=1}^{2} N_k$ |
| Xnet | $\prod_{k=1}^{2} N_k$ | 3 à 8 | $N_k$-1 | $2.\prod_{k=1}^{2} N_k - \sum_{k=1}^{2} N_k + 2$ si $N_k = 2$<br>$2.\prod_{k=1}^{2} N_k - \sum_{k=1}^{2} N_k$ $+2.\sum_{k=1}^{2}(N_k - 1)$ si $N_k > 2$ |
| hypercube nD | $2^n$ | n | n | $n.2^{n-1}$ |
| octagon | 8 | 3 | 2 | 12 |
| arbre(n,p) | $\sum_{k=0}^{p} k.n + 1$ | 1 ou n | p | $\sum_{k=0}^{p} k^n$ |

représente le nombre de branches et p représente la profondeur de l'arbre. L'arbre est supposé homogène et équilibré. Nous remarquons directement que la topologie octagon n'est pas appropriée dans notre cas. La topologie arbre a l'inconvénient de présenter une forme triangulaire qui croît en $N \log N$. La congestion est aussi un obstacle à l'utilisation de ce réseau. Structure simple et proche de beaucoup d'applications (traitement d'images par exemple), la grille (maille/tore) a été très usitée dans la conception des machines massivement parallèles. Son succès s'explique parce qu'elle est caractérisée par un diamètre raisonnable et une parfaite extensibilité. La topologie hypercube est également populaire. Par contre, l'extensibilité des machines construites à partir d'un hypercube est moins aisée qu'avec les structures précédentes. D'une part, dans un hypercube le nombre de processeurs est toujours une puissance de 2, et il faut donc au moins doubler le nombre de processeurs pour augmenter la taille du réseau. D'autre part, le degré n'est pas fixe et est égal à la dimension de l'hypercube, ce qui en fait une topologie plus difficilement extensible. La topologie Xnet a été choisie vue sa popularité et efficacité dans la machine MasPar.

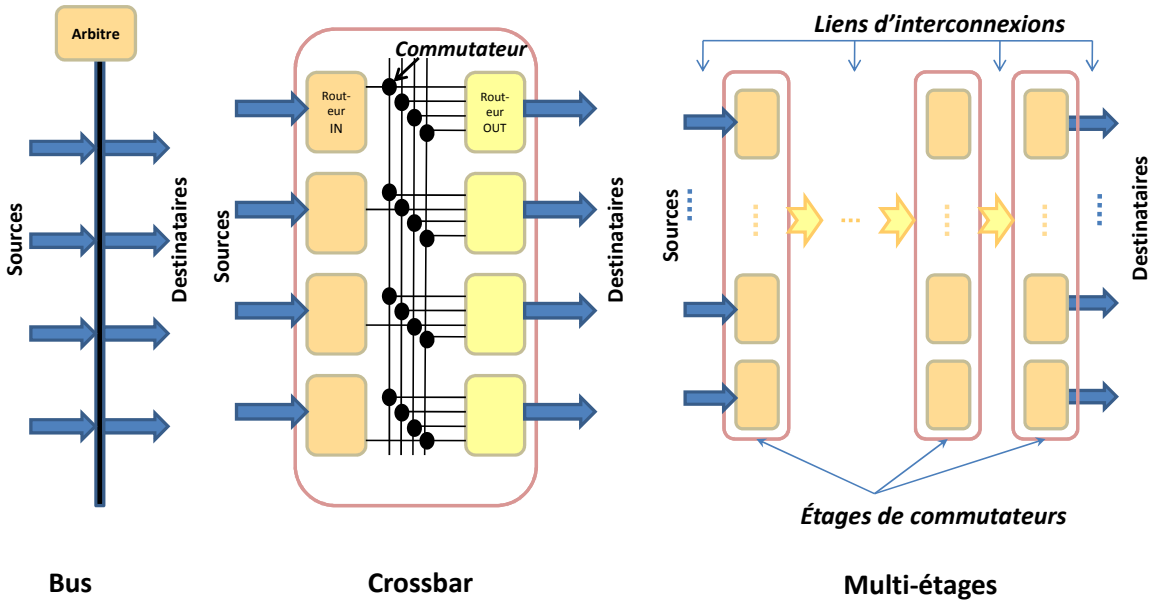



FIGURE 3.5 – *Types de réseaux d'interconnexion du mpNoC*

Les topologies engendrées par un réseau matriciel (maille, tore, Xnet) forment un choix très apprécié aussi bien pour leur simplicité et leur faible coût que par la possibilité de mettre en œuvre diverses applications. Les réseaux linéaires (linéaire et anneau) sont un cas particulier des réseaux bidimensionnels. Ces cinq topologies ont été retenues pour configurer automatiquement le réseau de voisinage de mppSoC.

Concernant le mpNoC, un parmi trois types de réseaux d'interconnexion dynamiques peut être choisi directement pour implémenter une configuration mppSoC. Les réseaux dynamiques sont en effet des réseaux dont on peut changer la topologie de communication en cours d'exécution des programmes ou entre deux exécutions de programmes. Contrairement aux réseaux statiques qui présentent une topologie fixe, les réseaux dynamiques sont beaucoup plus souples à manier. Les réseaux possibles de mpNoC sont au nombre de trois classés par ordre croissant de coût et de performance comme suit :

- un bus hiérarchique ;
- un réseau multi-étages de type Delta (*multistage*) ;
- un réseau matriciel (*crossbar*).

Ces différents réseaux se diffèrent par un certain nombre de paramètres à savoir : le débit et la latence, la surface, la flexibilité, la fiabilité et la topologie. Selon les contraintes de coût et de performance, le concepteur peut choisir le réseau le plus adapté à ses besoins (voir figure 3.5). Le tableau 3.2 récapitule les principales propriétés des trois réseaux. En effet, lorsque la complexité du traitement à réaliser est limitée avec un nombre de PE faible et que le taux d'utilisation du bus par les PEs n'est pas trop élevé et ne nécessite pas des transferts parallèles de données, un réseau en bus ne pose pas de problèmes particuliers. Il est de plus simple et relativement peu coûteux. Par contre, lorsque les contraintes sont plus exigeantes et donc que le nombre de PEs augmente ainsi que leur taux d'utilisation du bus, il se produit un problème de goulot (*bottleneck*). Celui-ci résulte du fait que seulement deux éléments

TABLE 3.2 – Comparaison entre les réseaux : bus, crossbar et MIN

| Réseau | Propriétés | Nature | Nombre de commutateurs | Nombre de liaisons |
|---|---|---|---|---|
| Bus | extensible en terme de coût, pas en termes de performances | bloquant | 0 | o(n) |
| Crossbar | extensible en termes de performances, pas en terme de coût | non bloquant | o($n^2$) | o(2n) |
| MIN | solution intermédiaire | bloquant | o($n \log n$) | o($n \log n$) |

peuvent communiquer entre eux à chaque instant. Le bus a aussi une bande passante limitée. Si la conception d'une machine parallèle à base de bus est plus aisée, elle ne permet cependant pas d'atteindre un degré élevé de parallélisme. Le partage du bus exige en effet un arbitrage et devient rapidement un goulot d'étranglement.

Vu les limites du bus, nous avons pensé à intégrer un réseau de commutation non bloquant tel que le crossbar au sein du mpNoC. C'est un réseau à un étage qui permet de relier simultanément n'importe quelle paire de nœuds inoccupés après un délai de commutation fixe et indépendant du nombre de nœuds dans le réseau. Ce réseau est performant puisqu'il n'a pas le problème de congestion tel est le cas du bus. Cependant, son inconvénient est le coût très élevé de sa programmation matérielle sur la puce. Il est donc difficilement extensible car ce coût croît avec le nombre de nœuds (n) nécessaires soit en O($n^2$). Dans un système bâti autour d'un réseau crossbar, dit fortement couplé, le degré de parallélisme reste faible : le nombre maximal de processeurs est de l'ordre de la centaine.

Dans le contexte d'un système massivement parallèle, les réseaux multi-étages, dénotés MIN, présentent une solution intermédiaire entre les réseaux en bus et les réseaux crossbar. Ils sont proposés afin de connecter un grand nombre de processeurs. Les réseaux multi-étages sont des réseaux crossbar non complètement connectés. Ils essayent donc de s'approcher autant que possible des performances du réseau crossbar tout en nécessitant beaucoup moins de commutateurs mais plus de liaisons et plus de temps de traversée. Dans un tel réseau, le nombre de commutateurs est d'ordre ($N \log_2 N$), comparé à $N^2$ pour le crossbar. Ces réseaux emploient des crossbars de tailles restreintes et les arrangent pour obtenir l'interconnexion d'un grand nombre d'éléments. Ils peuvent également fournir d'autres propriétés comme la limitation du nombre de transistors pour réduire le coût de fabrication. Notre travail s'est concentré par un type bien connu de ces réseaux multi-étages, nommé réseaux Delta [83]. La figure 3.6 montre une classification des réseaux MINs. Les réseaux Delta appartiennent à la classe des réseaux Banyan. Ces interconnexions sont utilisées depuis longtemps dans les super-calculateurs parallèles haut de gamme [98]. Nous distinguons trois types de réseaux Delta : omega, baseline et butterfly. Tous ces réseaux sont équivalents du point de vue topologique. Il suffit de réordonner les positions des commutateurs sans rompre les connexions pour passer d'un réseau à un autre. Dans un réseau multi-étages, les commutateurs sont assemblés sous forme d'un tableau rectangulaire dont les lignes correspondent au nombre de PEs et les colonnes correspondent au diamètre D du réseau qui sert de base (voir figure 3.7). Il est à remarquer que le temps de traversée d'un réseau dynamique est fixe (il est égal à D*$T_s$ où $T_s$ est le temps de traversée d'un commutateur). Il ne dépend

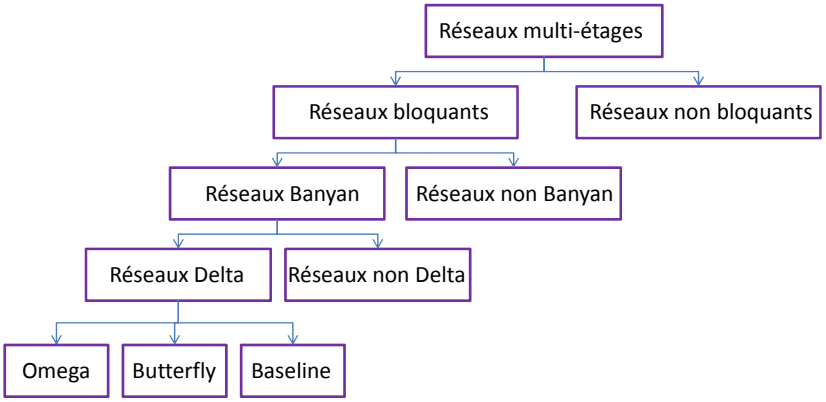



FIGURE 3.6 – *Classification des réseaux MINs*

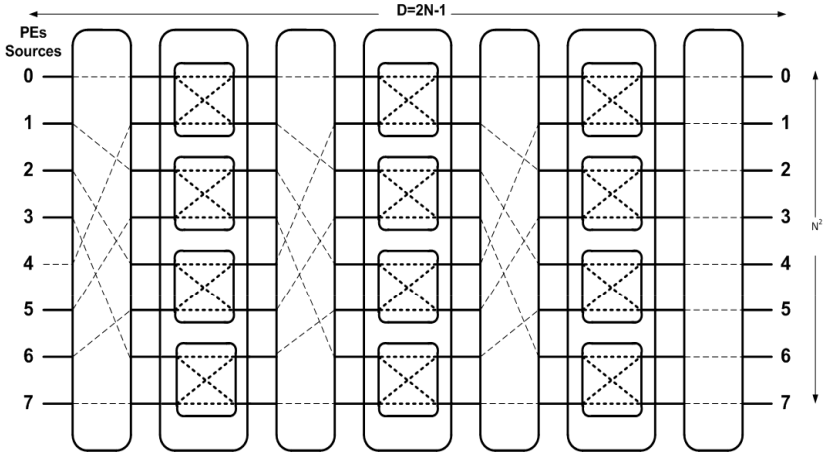



FIGURE 3.7 – *Structure de commutateurs formant un réseau MIN-oméga*

donc pas de la distance entre la source et la destination contrairement à ce qui se passe généralement sur un réseau statique non complètement connecté. Ceci est un avantage en mode SIMD vu qu'il y a une préservation de la synchronisation.

Il serait intéressant d'adapter le choix du réseau d'interconnexion (parmi les trois cités précédemment) en fonction des besoins de l'application, de la taille de la grille de PEs utilisée ainsi que de l'espace disponible sur la puce. Il est à noter que le modèle mppSoC ne se limite pas aux différents types de réseaux proposés (voisinage et mpNoC). Le concepteur peut intégrer le réseau de voisinage de son choix (avec une autre topologie) ainsi qu'un autre réseau d'interconnexion avec le mpNoC (ne figurant pas dans les réseaux listés précédemment). Pour ce fait, il doit respecter l'interface requise par mppSoC pour une intégration directe avec le reste de l'architecture. Cette liberté favorise encore la flexibilité du modèle mppSoC.

### 3.3.3 Aspect programmable

La diversité des algorithmes ainsi que leurs nombreux paramètres laissent au concepteur une grande liberté lors de l'implémentation. L'exploration et la comparaison de différentes

solutions algorithmiques requièrent l'utilisation d'architectures programmables. Donc à la différence des accélérateurs SIMD matériels spécifiques, mppSoC peut être adapté à l'application tout en gardant son aspect programmable. Cet aspect est garanti par l'utilisation de processeurs programmables dans l'implémentation de l'architecture. MppSoC exécute un programme data parallèle, un modèle de programmation qui convient bien aux applications de TSS. De ce fait, les fonctions de traitement élémentaire (convolution, corrélation, etc.) peuvent maintenant être développées en microcode et non plus en langage de description matérielle, tel que VHDL, ce qui réduit drastiquement le coût de développement. Cela permet en outre de faire des évolutions à moindre coût en fonction des évolutions de l'algorithme tout le long du développement du système. La programmabilité permet donc de tester de nouveaux algorithmes et ainsi répondre à leur évolution continue.

Les aspects liés à la programmation jouent de plus un rôle majeur dans la définition de l'architecture. En effet, très souvent, la programmation des architectures spécialisées est jugée extrêmement difficile en raison des faiblesses des environnements de programmation, de l'inadaptation du langage, et parfois de l'impossibilité de produire un code faisant un bon usage des originalités de l'architecture. C'est la raison pour laquelle il est indispensable de définir un langage parallèle ou langage adapté au modèle d'exécution de l'architecture conçue, tel est le cas pour mppSoC.

Une des questions prédominantes suite à l'analyse de l'état de l'art est de rendre l'architecture SIMD adaptée à l'application considérée (exprimée à travers la question $Q_1$ posée à la fin du chapitre 2). La section 3.3 apporte des éléments de réponse pour pouvoir adapter le système mppSoC aux besoins applicatifs. Dans cette section, nous avons dévoilé les aspects et les caractéristiques qui peuvent être paramétriques et flexibles dans une architecture SIMD. En effet, mppSoC est conçu comme un système modulaire se basant sur un assemblage de composants. Le concepteur a donc le choix de construire une configuration intégrant les composants dont il a besoin pour exécuter son application. Une configuration SIMD de base doit contenir une unité de contrôle avec son réseau de PEs. Les réseaux de communications et les périphériques sont les composants auxiliaires à intégrer selon les besoins. Le concepteur doit aussi spécifier les paramètres de son système. Paramétrer le nombre de PEs intégrés ainsi que la taille des mémoires permet de répondre aux exigences de diverses applications avec différentes tailles et complexités. Ceci favorise l'extensibilité du système.
Les réseaux de communication à intégrer dans mppSoC sont relativement flexibles. Le concepteur sélectionne un choix parmi ceux existants. En effet, le réseau de voisinage peut avoir différentes topologies : anneau ou linéaire dans le cas d'un arrangement linéaire de PEs ; maille, tore ou Xnet dans le cas d'une grille 2D de PEs. Le mpNoC intègre un réseau d'interconnexion qui peut varier ayant un type au choix parmi trois possibles (bus, crossbar ou MIN) selon les besoins et contraintes. Cette flexibilité introduite au niveau des communications augmente les performances du système parallèle.
La programmabilité de mppSoC favorise encore la flexibilité du système et permet d'exécuter différents algorithmes de TSS. Coder l'algorithme dans un langage data parallèle adapté au modèle d'exécution SIMD de mppSoC tire profit des spécificités de l'architecture conçue et accroît son efficacité.
Ayant un aspect paramétrique, relativement flexible ainsi que programmable, mppSoC peut être adapté à divers besoins applicatifs.

À travers tout ce que nous venons de voir, il convient que dans ces travaux, nous sommes entrain de traiter un problème d'adéquation architecture/application/programmation. Nous partons d'un programme data parallèle qui tourne bien sur un système mppSoC à architecture SIMD paramétrique, elle même peut être adaptée selon les besoins applicatifs. Certains aspects architecturaux clés de mppSoC seront encore justifiés et validés dans la section suivante.

## 3.4 Validation du modèle mppSoC

Afin de montrer la pertinence des aspects flexibles et paramétriques de mppSoC, cette section survole quelques algorithmes data parallèles et présente la performance de mppSoC à répondre à leurs besoins. Les algorithmes choisis sont une multiplication de matrices et un algorithme de réduction. Ils jouent un rôle important dans les applications de TSS.

### 3.4.1 Algorithme de multiplication de matrices parallèle

Dans ce paragraphe, nous étudions le comportement d'un algorithme de multiplications de matrices (MM) carrées d'ordre N sur mppSoC en variant le nombre de PEs, la topologie du réseau de voisinage ainsi que l'arrangement des PEs. Nous voulons calculer C = AxB selon l'équation :

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj} \quad avec\ 1 \leq i, j \leq N \tag{3.1}$$

Nous procédons alors à une étude théorique puis une étude expérimentale.

#### 3.4.1.1 Étude théorique

Sur une architecture séquentielle, cet algorithme nécessite N multiplications et N additions sur $N^2$ éléments ce qui fait un total de $2N^3$ opérations. La complexité séquentielle d'un tel algorithme est alors de l'ordre de o($N^3$). Vis à vis de la taille mémoire, nous avons besoin d'une mémoire pour stocker trois matrices de taille $N^2$ et donc le besoin en mémoire est de o($N^2$).

Nous allons maintenant comparer les performances de cet algorithme en considérant une exécution parallèle sur P processeurs. Nous assumons que N est divisible par P. Chacun des processeurs doit d'abord disposer des données de départ (des matrices A et B). Donc l'ACU va envoyer $2N^2$ données vers P processeurs ce qui engendre un coût de communication de l'ordre de o($2PN^2$). Si les P PEs sont agencés en un réseau linéaire inter-connectés en anneau, alors une manière de parallélisation de l'algorithme de MM est de répartir A en blocs de lignes circulés entre les PEs alors que chacun va avoir une colonne de B et en résultat pourra calculer une colonne de C. La figure 3.8 illustre cette partition. Cette exécution permet de calculer en parallèle toutes les colonnes de C nécessitant N étapes. L'algorithme parallèle est le suivant :

*Copier A($id_{PE}$) dans Tmp*
*C($id_{PE}$) = C($id_{PE}$) + Tmp * B($id_{PE}$, $id_{PE}$)*

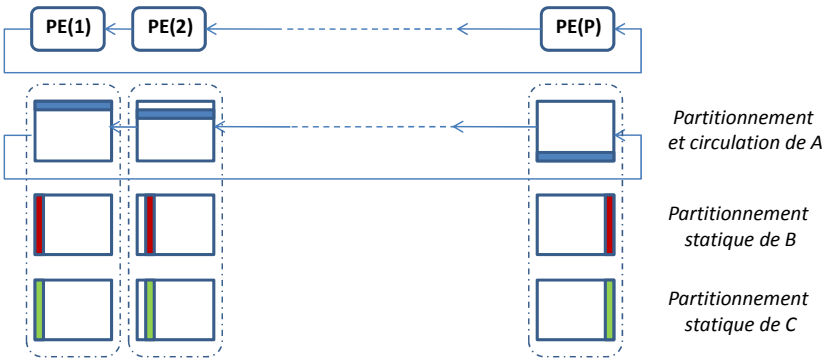



FIGURE 3.8 – *Partitionnement sur un anneau de PEs*

*Pour j=1 jusqu'à (P-1)*
*Envoyer Tmp au processeur ((id$_{PE}$ + 1) mod P)*
*Recevoir Tmp du processeur ((id$_{PE}$ - 1) mod P)*
*C(id$_{PE}$) = C(id$_{PE}$) + Tmp * B((id$_{PE}$ - j) mod P, id$_{PE}$)*

Chaque calcul nécessite : $(2\frac{N}{P}) * \frac{N}{P} * N = \frac{2N^3}{P^2}$ opérations. Si nous assumons que le temps de communication suit un modèle linéaire tel que :

$$t(k) = t_{lat} + k * t_{bw}, \tag{3.2}$$

avec k est le nombre de données à transférer, $t_{lat}$ est le temps de latence et $t_{bw}$ est le temps de transfert d'une donnée dans le réseau. Chaque itération dans la boucle aura un temps :

$$t = t_{lat} + \frac{N^2}{P} * t_{bw}. \tag{3.3}$$

Le temps de communication total vaut alors :

$$(P-1)[t_{lat} + \frac{N^2}{P} * t_{bw}]. \tag{3.4}$$

Le temps de calcul vaut :

$$P * \frac{2N^3}{P^2}. \tag{3.5}$$

Donc : le temps total = temps de calcul + temps de communication.

$$T_{p_{anneau}} = P * \frac{2N^3}{P^2} + (P-1)[t_{lat} + \frac{N^2}{P} * t_{bw}] = \frac{2N^3}{P} + (P-1)t_{lat} + (P-1)\frac{N^2}{P} * t_{bw} \tag{3.6}$$

L'accélération S théorique est calculée en divisant $2N^3$ par le temps obtenu dans le cas de la parallélisation. L'efficacité E sera alors :

$$E = \frac{S}{P} = \frac{1}{1 + \frac{P(P-1)t_{lat}}{2N^3} + \frac{(P-1)t_{bw}}{2N}} \tag{3.7}$$

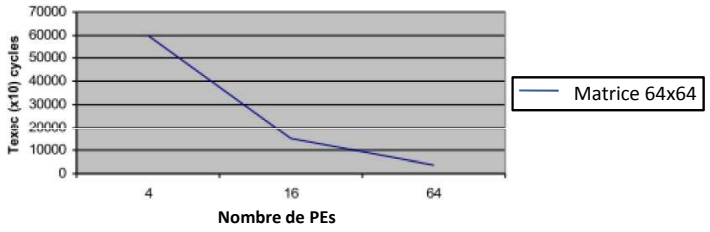



FIGURE 3.9 – *Temps de simulation en variant le nombre de PEs*

Nous remarquons que l'efficacité tend vers 1 avec un grand nombre de N. Ce qui prouve les performances de cet algorithme parallèle en exécutant une MM de grande taille. Notons de même que le temps de calcul est de o($\frac{N^3}{P}$), par contre celui de communication est de o($\frac{N^2}{P}$). Ceci montre que le coût de communication est bien plus petit que celui du calcul. Si nous assumons de plus que le nombre de processeurs est égal à N, on voit clairement que le temps de calcul est de o($N^2$) et celui de communication est de o(N). Ceci montre bien l'apport de paralléliser un algorithme de MM.

Considérons maintenant l'algorithme de multiplication de matrices parallèle sur une architecture SIMD avec un réseau de voisinage en tore. Supposons que le nombre de processeurs est pair : P=$s^2$ arrangés en une grille sxs. Nous allons dans ce cas appliquer l'algorithme de Cannon [62] :

*CANNON(A,B,C)*

*Décalage de A et B*
*Pour k=1 à s en parallèle :*
    *Produit en local de A, B et C*
    *Rotation verticale de B*
    *Rotation horizontale de A*

La première étape de décalage de la matrice A ainsi que B nécessite un même temps de communication égal à :

$$(s-1)(t_{lat} + (\frac{N}{s})^2 t_{bw}) \tag{3.8}$$

L'étape de rotation coûte $t_{lat} + (\frac{N}{s})^2 * t_{bw}$ et chaque multiplication locale nécessite $2(\frac{N}{s})^3$ opérations. Le temps parallèle théorique est alors évalué à :

$$T_{p_{tore}} = 2\frac{N^3}{P} + 2\sqrt{P}t_{lat} + 2\frac{N^2}{\sqrt{P}}t_{bw} \tag{3.9}$$

Nous remarquons dans ce cas que le temps de communication est de l'ordre de o($\sqrt{P}$) ce qui est inférieur par rapport au cas d'une topologie de PEs en anneau (o(P)). Cette démonstration prouve alors qu'un système mppSoC où les PEs sont agencés en grille et inter-connectés par un réseau de voisinage en tore est le plus adéquat pour exécuter une multiplication de matrices parallèle.

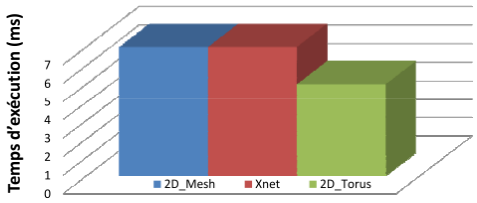



FIGURE 3.10 – *Temps de simulation en variant la topologie du réseau de voisinage*

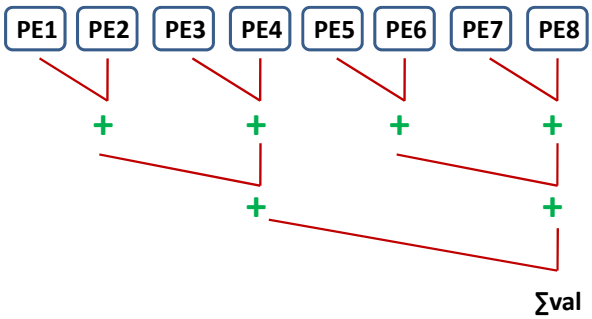



FIGURE 3.11 – *Somme de huit valeurs sur huit PEs*

#### 3.4.1.2 Étude expérimentale

Un algorithme parallèle de multiplication de matrices a été exécuté sur différentes configurations mppSoC fonctionnant à 50 MHz. Premièrement, nous avons varié le nombre de PEs et deuxièmement la topologie du réseau. La figure 3.9 montre les temps de calcul d'un algorithme de MM avec des matrices de taille 64x64 en variant le nombre de PEs [7]. Nous voyons clairement dans ce cas que plus le nombre de PEs est important plus les performances du système sont meilleurs. L'accélération est presque linéaire avec le nombre de PEs. Ceci est du à l'absence des communications qui ont une grande influence sur l'accélération du système. Ces résultats confirment bien ce que nous avons présenté dans la section précédente.

La figure 3.10 montre les temps d'exécution de l'algorithme de MM avec des matrices de taille 128x128 et un nombre de PEs égal à 64. Dans cette expérimentation, différentes topologies 2D de réseaux de voisinage ont été testées. Ces résultats prouvent que la topologie tore est la plus adéquate, telle que montrée dans la partie théorique.

### 3.4.2 Algorithme de réduction : Somme de N valeurs sur P processeurs

Dans ce paragraphe nous étudions un algorithme de réduction qui consiste à faire la somme de N valeurs sur P processeurs.

#### 3.4.2.1 Étude théorique

Dans un premier cas, nous prenons N=P. Nous voulons obtenir la somme des N valeurs dans le dernier processeur. Les PEs sont inter-connectés par un réseau linéaire. La figure 3.11 illustre cet algorithme.

En général, il faut $\log_2(P)$ étapes ; chacune consiste en une phase de communication et une phase d'addition. Tous les PEs communiquent en même temps et calculent en même temps. Si nous supposons qu'une étape dure $T_{comm} + T_{add}$ où $T_{comm}$ est le temps de communication et $T_{add}$ est le temps d'addition, alors le temps parallèle vaut :

$$T_{par} = (T_{add} + T_{comm}) * \log_2(P) \tag{3.10}$$

Cependant, le temps séquentiel, en assumant que le nombre de valeurs à additionner est égal à P, est :

$$T_{seq} = (P-1) * T_{add} \tag{3.11}$$

L'accélération obtenue est :

$$S = \frac{T_{seq}}{T_{par}} = \frac{(P-1) * T_{add}}{(T_{add} + T_{comm}) * \log_2(P)} = \frac{(P-1)}{(1 + \frac{T_{comm}}{T_{add}}) * \log_2(P)} \approx \frac{P}{(1 + \frac{T_{comm}}{T_{add}}) * \log_2(P)} < P \tag{3.12}$$

Dans un deuxième cas, nous allons augmenter le nombre de N. Ceci justifie bien l'intérêt d'avoir une taille mémoire paramétrable qui s'adapte aux besoins de l'algorithme en question. Nous supposons que chaque processeur dispose de $\frac{N}{P}$ valeurs. Une manière d'effectuer cet algorithme en parallèle est que tous les PEs commencent par sommer leurs valeurs en local. Cela prend un temps égal à :

$$((\frac{N}{P} - 1) * T_{add} \tag{3.13}$$

Ensuite, nous appliquons la somme des P valeurs sur les P PEs. Nous obtenons donc :

$$T_{par} = (\frac{N}{P} - 1) * T_{add} + (T_{add} + T_{comm}) * \log_2(P) \tag{3.14}$$

L'équation de l'accélération est la suivante :

$$S = \frac{(N-1) * T_{add}}{(\frac{N}{P} - 1) * T_{add} + (T_{add} + T_{comm}) * \log_2(P)} = \frac{P}{\frac{N-P}{N-1} + \frac{P}{N-1}(1 + \frac{T_{comm}}{T_{add}}) * \log_2(P)} \tag{3.15}$$

En assumant $N >> P$ donc $\frac{N-P}{N-1} \approx 1$ et $(N-1) \approx N$, nous obtenons :

$$S \approx \frac{P}{1 + (1 + \frac{T_{comm}}{T_{add}}) * \frac{P \log_2(P)}{N}} \tag{3.16}$$

Avec un très grand nombre de N, nous remarquons que $S \approx P$ ce qui confirme l'efficacité de l'algorithme parallèle et de même l'adéquation du réseau linéaire pour ce type de problème.

#### 3.4.2.2 Étude expérimentale

Dans ce paragraphe, nous montrons les résultats expérimentaux en exécutant un algorithme de réduction - somme de 16384 valeurs entières - sur deux différentes configurations mppSoC [8]. Ces configurations diffèrent par la topologie du réseau de voisinage (réseau linéaire ou maille) connectant 64 PEs. La figure 3.12 montre les temps d'exécution obtenus sur une carte FPGA 2S180 et une fréquence du système à 50 MHz.

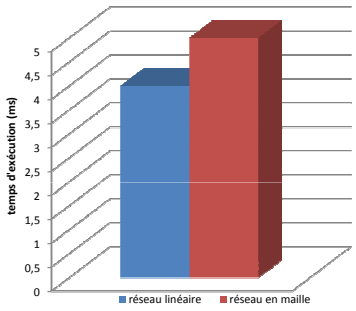



FIGURE 3.12 – *Algorithme de réduction sur des configurations mppSoC avec 64 PEs*

Ces résultats confirment bien la partie théorique et nous remarquons que le temps d'exécution avec un réseau linéaire est plus petit que celui en utilisant un réseau en maille.

D'après les algorithmes précédents, nous constatons le fait qu'il existe toujours un nombre de PEs optimal pour le quel on a un temps de communication acceptable. Ce temps doit être réduit afin de garder de bonnes performances du système parallèle. Si le temps de communication excède le temps de calcul, nous aurons alors une dégradation des performances. Ce fait justifie l'intérêt d'avoir un nombre de PEs paramétrique dans le système. Nous avons de même remarqué que ce temps dépend fortement de la topologie du réseau de voisinage. Donc avoir un réseau paramétrique est très important. Une bonne efficacité du système parallèle ne peut être atteinte que si un bon mariage entre l'application et l'architecture est trouvé.

Cette section a montré la pertinence des aspects paramétriques et flexibles de mppSoC, plus particulièrement en terme de nombre de PEs, agencement des PEs et topologie du réseau de voisinage. D'autres aspects et paramètres de mppSoC nécessitent d'être aussi analysés à savoir la variation du réseau d'interconnexion dans le mpNoC. Ces aspects seront plus détaillés dans les chapitres qui suivent.

## 3.5 Comparaison du modèle mppSoC avec d'autres modèles SIMD

Divers modèles d'architectures SIMD massivement parallèles sur puce ont été proposés. Des exemples intègrent des processeurs très simples opérant sur des données 1-bit tel est l'exemple décrit dans [95]. Cela a l'avantage de pouvoir intégrer un grand nombre sur une seule puce. Cependant, intégrer de très simples PEs peut parfois dégrader les performances [72]. Notre design intègre des PEs 32-bits simples et rapides à exécuter les opérations. Ces mêmes opérations peuvent par contre nécessiter plusieurs PEs 1-bit ou 4-bits et/ou plusieurs cycles d'exécution.

Le tableau 3.3 analyse le modèle de trois exemples de systèmes SIMD. Un des points clés de mppSoC est sa conception modulaire où différentes configurations peuvent être implémentées selon les besoins. De par sa conception, il est possible d'adapter la capacité de calcul de l'architecture mppSoC en fonction des besoins, en augmentant le nombre de processeurs, changeant la structure (linéaire ou matricielle), configurant les connexions entre les PEs et en intégrant le réseau assurant les entrées/sorties parallèles.

En comparaison avec le coprocesseur Imagine [55] qui contient des ALU dédiées qui

TABLE 3.3 – Caractéristiques de modèles SIMD

| Système | Conception de PEs | Nombre de PEs | Taille mémoire | Réseau de voisinage | Réseau global | Applications |
|---|---|---|---|---|---|---|
| [95] | à base d'IP Picoblaze PE 1-bit | Archi. divisée en partitions para-métriques | 64 byte RAM | maille | comm. par *buffer* | prog. TI[3] très simple |
| Ter@core [14] | dédiée | 128 | 1Kx18b RAM 512x36b RAM | anneau | bus : messages + réseau DMU (*Data Mover Unit*) : données | prog. (MIMD pour algo. complexes (rotation d'images)) |
| PoD [112] | à base d'unité SSE 128 bits | 8 x 8 | 128 KB SRAM | maille ou tore complet | comm. *all-to one* à base de bus | prog. (jeu d'ins. Intel 64) |

opèrent sur des flots de données en suivant le modèle producteur-consommateur, notre système utilise des mémoires locales à chaque PE afin de permettre la réutilisation des données locales. Comparé avec le circuit Ter@core [14], mppSoC présente plus de flexibilité et simplicité de conception étant donné que ses PEs se base sur des IPs standard et non conçus à la main. Le réseau de voisinage de mppSoC est aussi configurable à la différence de celui de Ter@core qui est toujours un réseau en anneau.

Comparé à l'architecture PoD [112], notre système intègre un réseau massivement parallèle sur puce, mpNoC, performant pour gérer les entrées/sorties parallèles et assurer tout type de communication irrégulière. Par contre, l'architecture PoD supporte seulement des communications de voisinage en topologie maille ou tore. Un seul type de communication globale "tous vers un" (*all-to-one*) est aussi permis : soit sur une même ligne ou sur une même colonne. Des bus sont utilisés afin de transférer les données d'une mémoire partagée vers tous les PEs ou vice versa. Ce mécanisme de bus n'est pas efficace et peut limiter les transferts de données parallèles.

D'autres processeurs SIMD, malgré programmables, restent encore spécifiques à des applications très particulières. C'est l'exemple du processeur décrit dans [63] dédié à un algorithme de chiffrement RC4 et n'inclue pas de communications entre les PEs.

## 3.6 Conclusion

Tout au long de ce chapitre nous avons défini et présenté une analyse détaillée des différents aspects de mppSoC. Ce système SIMD sur puce se présente comme une architecture

3. Traitement d'Images

parallèle répondant aux besoins des applications régulières data parallèles, modulaire, programmable, suffisamment générale et flexible. L'analyse dans ce chapitre a montré une forte dépendance entre ces aspects. Tous se basent sur une méthode de conception systématique et générique. La conception d'une architecture massivement parallèle sur puce est motivée par plusieurs critères :

- Les contraintes de mise sur le marché deviennent si serrées que la réutilisation massive d'unités de calcul sera l'un des moyens pour obtenir rapidement la puissance de calcul requise par les générations à venir d'applications embarquées.
- Une architecture parallèle régulière et homogène est plus facile à programmer qu'une architecture irrégulière.
- Un modèle d'exécution SIMD s'adapte bien aux applications multimédia ayant un parallélisme de données massif.

Dans le chapitre suivant, nous étudierons de façon plus précise la conception de mppSoC. La méthode de conception proposée permettra la réduction du temps de développement du système et pourra servir dans un processus d'exploration de l'architecture.

# Chapitre 4

# Méthode de conception rapide et programmation de mppSoC


**4.1 Présentation de la méthode de conception de mppSoC . . . . . 61**
4.1.1 Description de la conception de mppSoC . . . . . 61
4.1.2 Méthodologies de conception des processeurs . . . . . 63
4.1.3 Tâches du concepteur . . . . . 67
**4.2 Bibliothèque d'IPs mppSoC . . . . . 68**
4.2.1 Processeurs . . . . . 68
4.2.2 Mémoires . . . . . 87
4.2.3 Réseau de voisinage . . . . . 87
4.2.4 MpNoC . . . . . 90
4.2.5 Classification des IPs . . . . . 97
**4.3 Programmation de mppSoC . . . . . 97**
4.3.1 Cas de réduction de processeur . . . . . 97
4.3.2 Cas de réplication de processeur . . . . . 101
4.3.3 Jeu d'instructions mppSoC . . . . . 102
**4.4 Conclusion . . . . . 104**

Dans le chapitre précédent, le modèle architectural de mppSoC a été présenté. Nous avons ainsi montré les différents aspects du système mppSoC le rendant flexible pouvant s'adapter aux exigences applicatives. L'un de ses principaux aspects est la modularité qui s'appuie sur une conception à base de composants. Un système massivement parallèle complexe doit être conçu de manière efficace afin de réduire le temps de mise sur le marché. Notre principal objectif est de réduire le coût et le temps de conception du système mppSoC afin d'obtenir un circuit fiable et performant. Tout ceci doit être construit selon une méthode de conception favorisant l'abstraction de l'architecture, la validation et facilitant la génération d'une description de bas niveau d'une architecture SIMD. Dans ce chapitre, nous détaillons la méthode de conception de mppSoC. Cette conception se base sur l'utilisation d'une bibliothèque d'IPs qui sera aussi présentée.

Ce chapitre est structuré comme suit. La première section présentera la méthode de conception de mppSoC. La deuxième section décrira la bibliothèque d'IPs utilisée dans le flot de conception. La programmation de mppSoC sera présentée dans la troisième section. Le jeu d'instructions conçu pour mppSoC sera aussi décrit. Finalement, la dernière section donnera une synthèse de ce chapitre.

## 4.1 Présentation de la méthode de conception de mppSoC

Différentes méthodologies de conception, telles que notées dans le chapitre 2, sont apparues poussées par les contraintes de temps de mise sur le marché et de complexité galopante des systèmes sur puce. À part ces méthodologies de conception, il faut bien tenir compte de l'évolution rapide des besoins imposant alors des délais de mise sur le marché d'un nouveau produit de plus en plus brefs. De nouvelles méthodologies de conception sont par conséquent nécessaires. Celles-ci doivent permettre de répartir efficacement la complexité des systèmes à concevoir sur les ressources humaines et matérielles par le biais d'une automatisation des tâches de conception les plus coûteuses [33]. Une accélération du flot de conception est indispensable afin de conserver des délais raisonnables de mise sur le marché d'un produit et d'exploiter au mieux les nouvelles technologies au moment où elles sont disponibles. Une voie prometteuse pour l'accélération des tâches de conception consiste à réutiliser autant que faire des blocs matériels ou logiciels déjà conçus et vérifiés.

Dans ce sillage, notre principal objectif dans ces travaux de thèse consiste à la génération d'architectures SIMD massivement parallèles, se basant sur l'utilisation et la réutilisation d'IPs dans la conception. Notre méthode de conception de mppSoC s'intègre parmi les méthodologies actuelles de conception des systèmes qui promeuvent les approches à base de composants et exploitent l'utilisation d'IP. Ces techniques de conception sont cruciales pour surmonter la complexité des systèmes embarqués.

Tout au long de ce chapitre, nous dénotons "IP" les différents composants utilisés dans l'architecture mppSoC.

### 4.1.1 Description de la conception de mppSoC

La conception de mppSoC se base sur l'utilisation d'IPs disponibles. D'autres IPs sont de même conçus et implémentés pour être intégrés dans mppSoC et qui seront réutilisés dans tout nouveau cycle de conception si nécessaire. Cette conception est alors incrémentale et flexible. À partir de blocs IPs, les concepteurs adapteront rapidement le système à l'applica-

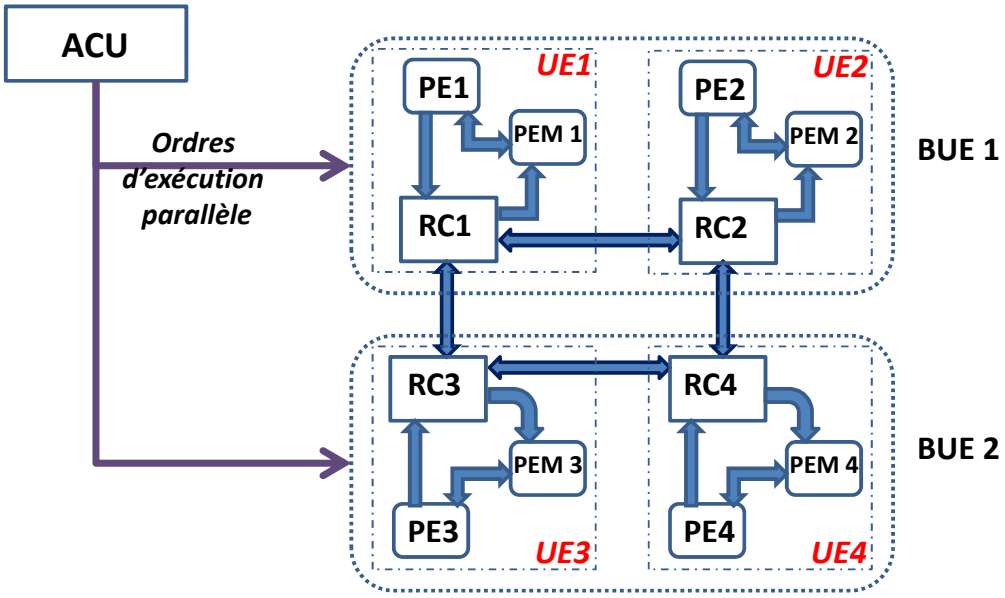



FIGURE 4.1 – *Conception hiérarchique de mppSoC intégrant 4 PEs*

tion visée. Ce contexte permet en grande partie d'alléger les coûts de conception. La conception à base de réutilisation d'IPs est aussi incontournable pour répondre aux exigences de diverses applications.

Pour diminuer le coût de développement d'un système sur puce, il faut :

– connaître les bons composants à intégrer (utiliser une bibliothèque d'IPs) ;
– savoir comment les intégrer de manière efficace (appliquer une méthodologie de réutilisation).

Tel que présenté dans le chapitre précédent, mppSoC est un système modulaire conçu par un assemblage de divers composants ou IPs. Ce système est aussi générique pouvant être paramétré dépendamment des besoins applicatifs.

Une conception hiérarchique est suivie pour la construction de mppSoC. En effet, les processeurs élémentaires sont inclus dans une unité hiérarchique appelé Unité Élémentaire (UE). Chaque UE, comme le montre la figure 4.1, est composée d'un processeur (PE), d'une mémoire et d'un routeur de communication (dénoté RC). Ce routeur est un élément important dans l'architecture de l'UE. Il contient une logique de contrôle permettant d'assurer les communications de voisinage demandées par le PE pour un autre PE voisin. Sa conception est mieux détaillée dans ce qui suit. Un autre niveau de hiérarchie, appelé Bloc d'Unités Élémentaires (BUE), correspond à une ligne dans la grille d'unités élémentaires. Ce niveau est créé afin de profiter de la compilation incrémentale et permet de limiter le synthétiseur utilisé dans ses optimisations. Ces optimisations indésirables créent souvent de la logique centrale et augmentent la difficulté de routage. Cette manière de conception permet alors d'organiser et de faciliter le placement du réseau de PEs sur l'FPGA.

Un autre problème rencontré dans la conception de mppSoC est la diffusion synchrone des ordres d'exécution parallèle à tous les PEs avec une consommation de puissance efficace. Afin d'assurer une exécution SIMD où chaque PE exécute la même instruction au même cycle, nous distinguons deux méthodes :

– exécuter l'instruction immédiatement dés son arrivée à une ligne de PEs, ce qui en-

gendre un délai de propagation entre le premier PE de la ligne et le dernier, appelé (*timezone effect*) [113].
– mémoriser chaque instruction pendant un temps suffisant jusqu'à ce que chaque PE exécute la même instruction au même instant.

La deuxième méthode introduit un coût supplémentaire de surface sur la puce et nécessite l'utilisation d'une logique dédiée afin de guider l'exécution des PEs. Cette approche n'est pas alors générale, non extensible et dépend du processeur utilisé. L'effet *timezone* rend de même la programmation synchrone de l'architecture un peu difficile. Une des solutions possibles est de procéder à des tests dans le programme à travers le *Or Tree*. Ce test s'avère nécessaire pour connaître si l'instruction courante a été exécutée par tous les PEs actifs avant de procéder à l'instruction qui suit. Cette méthode est plus simple tant en implémentation qu'au codage.

Afin d'assembler les processeurs dans une architecture SIMD (instance de mppSoC) et assurer les connexions entre l'ACU et les PEs, deux méthodologies ont été proposées selon les besoins. Elles sont détaillées dans la sous section suivante.

### 4.1.2 Méthodologies de conception des processeurs

Les processeurs dans mppSoC présentent l'ACU et les PEs. Ils sont conçus en utilisant le même IP processeur. Ils ont alors la même architecture, ce qui a l'avantage de minimiser les développements matériel et logiciel. La caractéristique principale des organisations SIMD est l'homogénéité. Cette propriété nous semble nécessaire pour parvenir à une solution aisément programmable. En utilisant le même IP, le haut coût de conception lié aux machines SIMD est considérablement réduit.

Parmi les limites des architectures SIMD existantes telle que montrée dans le chapitre 2 de l'état de l'art, l'utilisation de PEs 1-bit assez simples ne permet pas de répondre aux performances exigées par les applications modernes de TSS. Pour cela, l'architecture de mppSoC se base sur l'utilisation de processeurs standard 32-bits. En effet, avec la croissance de la technologie d'intégration, il est possible de mettre sur une seule puce des processeurs assez performants. MppSoC se caractérise par son réseau homogène de PEs où le parallélisme est supporté par la duplication des IPs PEs de même type. Nous retiendrons prioritairement des processeurs à architecture RISC élémentaires libres d'utilisation (*open source*) ou du moins pour les quels il existe des outils de simulation et synthèse. Donc les solutions, pour lesquelles un environnement de développement ouvert est disponible, sont privilégiées. Vu la simplicité de ses instructions, un processeur RISC permet un gain en rapidité d'exécution, à part sa surface réduite qui est un critère très important à prendre en compte pour une implémentation massivement parallèle sur puce. Citons à titre d'exemple les processeurs MIPS qui sont de petits processeurs très utilisés dans le domaine des systèmes embarqués, Leon, ou encore les processeurs soft-cores tels que NIOS, MicroBlaze, etc.

Nous proposons deux méthodologies de conception de processeurs, réduction et réplication, pour le système mppSoC. Chacune présente différentes caractéristiques et l'utilisateur pourra choisir l'une ou l'autre selon ses besoins et contraintes. Les paragraphes suivants présentent ces deux méthodologies.

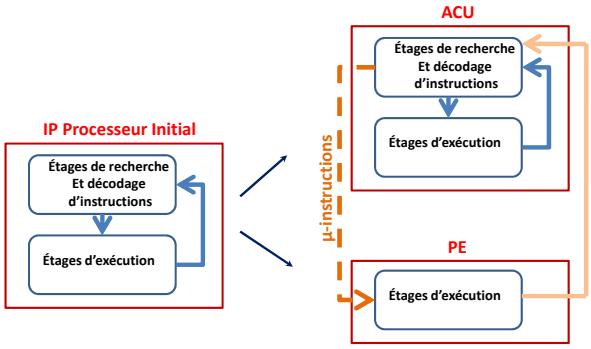



FIGURE 4.2 – *Principe de la méthodologie de réduction de processeur*

#### 4.1.2.1 Réduction de processeur

Comme son nom l'indique, la méthodologie de réduction de processeur consiste à réduire un processeur initial afin d'obtenir un processeur simple et réduit jouant le rôle du PE. L'architecture du processeur élémentaire constitue un critère intéressant. On trouve dans les architectures SIMD existantes divers styles de processeurs : du processeur bit-série organisé autour d'une UAL et de quelques registres 1-bit aux processeurs disposant de plusieurs unités fonctionnelles 16 voire 32-bits travaillant en parallèle. La simplicité de conception du PE est un point fondamental afin de réaliser une architecture massivement parallèle. Pour cela, nous avons proposé la méthodologie de réduction. Plus le PE est petit, plus l'efficacité performance/surface est plus grande. Cette méthodologie nécessite l'utilisation d'un processeur dont les sources sont disponibles (*open source*) pour pouvoir le modifier. De cette manière, le temps requis pour implémenter et tester un processeur propriétaire est réduit.

Notre méthodologie s'applique particulièrement aux processeurs pipelinés avec cinq étages. Dans cette méthodologie, l'ACU est conçu par le même IP processeur alors que le PE est un processeur réduit qui dérive de ce même IP. En effet, la majeure différence entre les deux composants est que les PEs n'ont pas de partie pour décoder les instructions : ils ne font que les exécuter. Ils contiennent en fait seulement les étages d'exécution. Dans l'optique d'une évaluation de l'espace silicium pris par le mppSoC, nous trouvons qu'un PE compte 20 à 30 % moins de transistors que l'ACU grâce au tronquage de la partie décodage des instructions. Toutes les instructions dédiées aux PEs sont envoyées par l'ACU sous forme de micro-instructions. Plus spécifiquement, l'ACU est conçu comme un processeur modifié qui produit des micro-instructions à la sortie de son étage de décodage. Ces micro-instructions sont en effet des instructions décodées et prêtes à être directement exécutées. Elles présentent alors les entrées des étages d'exécution du PE [7]. Pour un processeur RISC pipeliné, l'ACU contiendra tous les étages du pipeline avec modification au niveau de son étage de décodage. Alors que le PE est implémenté avec les étages d'exécution seulement telle qu'illustrée par la figure 4.2. La modification du pipeline du processeur requiert une grande attention pour ne pas perturber le fonctionnement. Cette méthodologie est suivie d'une modification du jeu d'instructions du processeur pour pouvoir diffuser les micro-instructions aux PEs. De nouvelles instructions parallèles sont alors ajoutées. Un décodage différent est développé pour permettre au processeur de distinguer entre une instruction séquentielle de celle parallèle. Les signaux de Or Tree et des bits d'activité et d'identité des PEs sont ajoutés. Une manière appropriée est de modifier certains registres

internes du processeur pour supporter ces différentes fonctionnalités.

En général, la méthodologie de réduction se résume principalement dans les étapes suivantes :

- Pour l'ACU :
  1. choisir un processeur *open source* ;
  2. tester le fonctionnement du processeur ;
  3. étudier l'interface entre l'étage de décodage et l'étage d'exécution ;
  4. dupliquer les signaux (sorties de décodage/ entrée à l'exécution) ;
  5. ajouter un test au niveau de décodage afin de distinguer entre instruction séquentielle ou parallèle ;
  6. ajouter tous les signaux nécessaires provenant des PEs pour les dispositifs avancés tel que le module de prédiction de branchement ou de renvoie.
- Pour le PE :
  1. éliminer les étages de recherche et décodage ;
  2. garder le banc de registres ;
  3. connecter les signaux dupliqués sorties de l'étage de décodage de l'ACU à l'étage d'exécution.
- simuler la connexion ACU-PE pour tester son bon fonctionnement.

La conception de quelques exemples de processeurs en suivant cette méthodologie sera décrite dans la section 4.2. En appliquant cette méthodologie, nous pouvons avoir un PE simple avec une surface minimale tout en se basant sur la réutilisation d'IP. Ce gain nous permet d'introduire un grand nombre de PEs sur une même puce. Cependant, l'inconvénient majeur de cette méthodologie est sa difficulté et son long temps de développement ; ce qui nous a poussé à proposer une deuxième méthodologie présentée dans le paragraphe suivant.

#### 4.1.2.2 Réplication de processeur

Comme son nom l'indique, la méthodologie de réplication de processeur consiste à concevoir le PE par un même processeur initial choisi et de le répliquer afin d'obtenir le réseau de PEs. Cette méthodologie facilite et accélère ainsi la conception. La connexion entre l'ACU et les PEs dans ce cas est un bus d'instructions parallèles (voir figure 4.3). Pour gérer les bits d'activité et d'identité des PEs, deux registres externes sont ajoutés. Le mécanisme du Or Tree est aussi assuré par l'implémentation d'un OU logique entre les bits d'activité des PEs.

En général, la méthodologie de réplication se résume dans les étapes suivantes :

1. étudier l'interface entre l'étage de décodage et l'étage de recherche d'instruction ;
2. ajouter un test au niveau de décodage de l'ACU afin de distinguer entre instruction séquentielle ou parallèle ;
3. ajouter la sortie d'instruction à communiquer de l'étage de décodage de l'ACU à l'étage de décodage du PE (désactiver alors son entrée initiale provenant de son étage de recherche d'instruction) ;

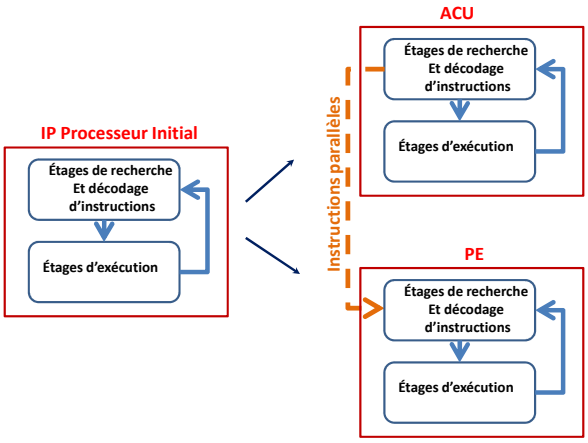



FIGURE 4.3 – *Principe de la méthodologie de réplication de processeur*

4. simuler la connexion ACU-PE pour tester son bon fonctionnement.

La méthodologie de réplication est facile à reproduire sur différents processeurs avec un temps de développement réduit. Cependant, son inconvénient est la difficulté d'introduire un grand nombre de PEs sur la même puce. Le critère d'utiliser cette méthodologie est de choisir de très simples processeurs qui consomment une petite surface sur l'FPGA.

Une des limites des architectures SIMD existantes est la spécialisation du PE intégré. Ces processeurs sont habituellement conçus pour un unique usage et ne peuvent donc pas répondre aux exigences évolutives des applications de TSS. De plus, la conception de tels PEs est propriétaire et nécessite un temps de développement assez long. Dans notre travail, nous proposons une diminution du coût de ce design dédié en utilisant un seul et préexistent IP processeur comme une base de conception de l'ACU et des PEs. Utiliser un IP processeur existant facilite la conception d'une part et réduit considérablement le temps de développement d'autre part. Réutiliser le même IP réduit encore le coût de conception et facilite la programmation de l'architecture. Nous avons proposé deux méthodologies d'assemblage des processeurs dans mppSoC : réduction et réplication. Alors que la réduction consiste à réduire un processeur et supprimer ses étages de décodage non nécessaires pour un PE, la réplication vient pour simplifier la construction de l'architecture massivement parallèle et consiste à répliquer directement les PEs. De cette manière, nous avons proposé une méthode de conception rapide et simple pour construire le réseau de PEs.
La méthodologie de réduction réponds alors à la problématique de construction d'un PE réduit à base d'IP. La méthodologie de réplication est une deuxième alternative de conception du PE permettant d'être le moins intrusif possible et accélérer ainsi le temps de conception (réponse à la question $Q_2$ posée à la fin du chapitre 2).

Nous voyons que chacune des deux méthodologies de conception des processeurs offre un compromis temps de développement/intégration de PEs. L'utilisateur peut choisir celle qui le convient en fonction de ses besoins.

### 4.1.3 Tâches du concepteur

Dans ces travaux, nous proposons un environnement de génération de mppSoC facilitant le choix d'une configuration mppSoC aux utilisateurs. Cet environnement utilise une bibliothèque d'IPs conçus afin d'être réutilisés dans n'importe quelle instance de mppSoC. L'utilisateur, disposant de l'environnement de mppSoC, n'a qu'à choisir les IPs à intégrer ainsi que saisir certains paramètres pour avoir en sortie un système SIMD sur puce décrit en VHDL. Les étapes de génération sont les suivantes :

1. sélectionner l'IP processeur à utiliser depuis la bibliothèque d'IPs mppSoC. L'utilisateur n'a qu'à choisir un seul IP vu qu'il servira pour la conception de l'ACU et aussi des PEs. Il a aussi la possibilité d'intégrer son propre IP sous la contrainte qu'il doit présenter les signaux requis pour une intégration dans mppSoC. Cette interface sera décrite dans la section suivante.

2. choisir la méthodologie de conception des processeurs : réduction ou réplication. Tous les processeurs open-source disponibles dans la bibliothèque sont offerts avec les deux méthodologies. Par ailleurs, si l'utilisateur choisit d'intégrer un autre, il doit appliquer la méthodologie de conception sélectionnée sur son processeur.

3. choisir l'IP mémoire à intégrer. Le principe du choix est le même que celui du processeur.

4. saisir les paramètres de mppSoC à savoir : le nombre/arrangement de PEs et la taille des mémoires de l'ACU et des PEs. Vu la conception hiérarchique de mppSoC, le nombre et l'arrangement des PEs sont gérés par deux paramètres génériques qui sont le nombre de lignes et le nombre de colonnes de la grille de PEs.

5. choisir ou non d'intégrer un réseau de voisinage. Il faut sélectionner alors une topologie parmi cinq offertes, qui soit adéquate avec l'agencement des PEs (linéaire ou grille).

6. choisir ou non d'intégrer le réseau mpNoC. Dans ce cas, l'utilisateur aura à sélectionner le type du réseau d'interconnexion (bus, crossbar ou Delta MIN). Dans le cas du réseau Delta MIN, l'utilisateur pourra aussi sélectionner une parmi trois configurations : omega, baseline ou butterfly. Cependant, il est à noter que les IPs Delta MIN présents dans la bibliothèque mppSoC se limitent à un nombre précis de PEs (4, 8, 16, 32, 64 et 128). L'utilisateur a la possibilité d'intégrer son propre réseau à condition qu'il présente l'interface requise par mpNoC. Cette interface sera détaillée dans la section suivante.

En suivant ces six étapes, l'utilisateur peut générer une architecture SIMD sur puce, selon le modèle mppSoC. L'architecture, décrite en VHDL, pourra après être simulée ou prototypée sur FPGA (en utilisant les outils de CAO disponibles avec l'FPGA) afin d'étudier les performances. Toute modification de l'architecture est possible et facile en utilisant l'environnement proposé et en répétant les étapes décrites ci dessus.

Une des questions fondamentales posée à la fin de l'état de l'art est la question $Q_5$ qui s'interroge sur la manière d'accélérer la conception d'une architecture SIMD. Notre travail réponds à cette question en proposant un environnement de génération de configurations SIMD qui dérivent du modèle mppSoC. Cet environnement rend facile la conception de mppSoC dans un temps réduit. Mené de cet environnement, un utilisateur n'a qu'à suivre six étapes afin de générer la configuration SIMD de son choix. Les étapes consistent à choisir les composants à intégrer, fixer les paramètres de l'architecture et générer après le code VHDL correspondant. L'environnement proposé permet alors de faciliter la tâche de conception de mppSoC aux concepteurs, abstrait les détails d'implémentation et est facile à utiliser.

La section suivante décrira la bibliothèque d'IPs tout en précisant la manière de conception de chacun.

## 4.2 Bibliothèque d'IPs mppSoC

Dans ce travail, nous avons choisi de construire une bibliothèque d'IP dédiée pour notre architecture parallèle. L'idée est d'assembler après une combinaison choisie de blocs selon les besoins pour développer rapidement une configuration mppSoC et réduire les délais de conception. Il faut en effet une nouvelle méthodologie d'interconnexion pour développer des systèmes sur puce contenant un milliard de transistors ou plus, organisés en centaines, voire en milliers de blocs IP. Notre choix d'interface propriétaire est aussi bien justifié. En effet, en prenant le cas du protocole VCI par exemple, l'interface VCI représente 135 bits. Un composant interfacé VCI relié à 1024 PEs par exemple, engendre plus de 100000 fils ce qui est très coûteux. De ce fait, nous avons pensé à utiliser un protocole moins coûteux et plus adapté au système mppSoC.

Pour la conception des IPs, nous avons utilisé le langage de description matérielle VHDL. Ce langage, initialement développé pour la modélisation et la simulation, est également utilisé pour la description comportementale ou RTL (transfert de registres) d'une architecture logique pour la synthèse automatique. Les IPs conçus se composent principalement de processeurs, mémoires et réseaux de communications. Ils seront décrits dans les sous-sections suivantes.

### 4.2.1 Processeurs

Dans ce paragraphe, nous détaillons la conception des processeurs disponibles dans la bibliothèque d'IPs mppSoC. Nous justifierons dans un premier lieu le choix des processeurs. Dans un deuxième lieu, nous décrirons leur conception en utilisant les deux méthodologies citées précédemment.

#### 4.2.1.1 Choix de processeurs

Les cœurs de processeurs sont toutefois bien souvent conçus pour un domaine applicatif privilégié ou avec des contraintes spécifiques d'efficacité silicium (MOPS/$mm^2$), d'efficacité énergétique (MOPS/mW) ou de programmabilité. De ce fait l'analyse de performances de ces cœurs fait apparaître une grande diversité de performance. Notre travail s'est basé sur le

TABLE 4.1 – Comparaison entre processeurs

| Processeur | NIOS II (fast) | MicroBlaze | OpenRISC 1200 | Leon3 | miniMIPS |
|---|---|---|---|---|---|
| Fréq. Max MHz | 200 (FPGA) | 200 (FPGA) | 300 (ASIC) | 400/125 (ASIC/FPGA) | 50 (FPGA) |
| ASIC/FPGA | -/Stratix et StratixII | -/Virtex4 | 0.18$\mu$m/- | 0.13$\mu$m/- | 0.6$\mu$m/- |
| DMIPS | 150 | 166 | 250 | 85 | N/D [4] |
| Mémoire cache (I/D) | jusqu'à 64 KB | jusqu'à 64 KB | jusqu'à 64 KB | jusqu'à 256 KB | 4 KB |
| Pipeline | 6 étages | 3 étages | 5 étages | 7 étages | 5 étages |
| Instructions personnalisées | jusqu'à 256 | non | limite non spécifiée | non | limite non spécifiée |
| Implém. | FPGA Altera | FPGA Xilinx | FPGA/ ASIC | FPGA/ ASIC | FPGA/ ASIC |
| Surface | 700-1800 LUTs | 1269 LUTs | 4626 LUTs | 3500 LUTs | 3060 LUTs |
| Licence | fourni avec Altera EDS [5] | fourni avec Xilinx EDK | GNU/ LGPL | GNU/ LGPL | GNU/ LGPL |
| Format | soft VHDL | EDIF [6] | Verilog | VHDL | VHDL |
| ISA [7] | NIOS II ISA | MicroBlaze ISA | ORBIS 32 | SPARC V8 | MIPS I |
| Interface | Bus Avalon | LMB, IBM OPB V2.0 | Wishbone SoC | AMBA - AHB | interface miniMIPS |

concept de "Open Hardware". Il s'agit de choisir des processeurs *open-source* ou des processeurs embarqués sur FPGA. Opencores [80] est un groupe qui propose un téléchargement d'un nombre varié de composants ou blocs IPs réutilisables. Le tableau 4.1 montre, d'une manière non exhaustive, une comparaison entre les divers processeurs courants (de type RISC 32 bits) appartenant à ces deux catégories [106].

Trois processeurs ont été choisis et implémentés dans la bibliothèque d'IPs : miniMIPS, OpenRISC 1200 et NIOS II. Les deux premiers processeurs sont libres (*open source*) et peuvent être téléchargés gratuitement depuis le site OpenCores (sous la licence GNU LGPL : *GNU Lesser General Public License*). Le choix du processeur miniMIPS était principalement basé sur des raisons pragmatiques : le MIPS est un processeur simple, standard et bien établi. Le processeur miniMIPS est de taille réduite caractérisé par son code source libre. Il est aussi bien codé et extensible. Par ailleurs, il possède certains dispositifs avancés permettant une augmentation des performances lors de son implémentation en tant qu'ACU. Il intègre en effet un module de prédiction de branchement, de gestion des exceptions et des interruptions, de détection d'aléa et de renvoie vers l'étage de décodage. La version initiale présente sur le site d'Opencores a été testée et corrigée car elle présentait certains bogues (au niveau

4. Non Disponible
5. Embedded Design Suite
6. Le VHDL code est disponible commercialement
7. Instruction Set Architecture

des branchements). Le processeur miniMIPS était le premier processeur au quel nous avons appliqué la méthodologie de réduction.

Le processeur OpenRISC 1200 a été choisi après pour étudier encore la faisabilité de la méthodologie de réduction et la refaire en facilitant certaines étapes. Il présente une plus grande fréquence sur FPGA que le leon3 (250 MHz). Il a la capacité d'étendre le jeu d'instructions par l'ajout de composants additionnels tel que l'ajout d'une unité flottante (*FPU : Floating Point Unit*). Le concepteur est aussi libre d'ajouter des instructions personnalisées. Pour le développement logiciel, les outils sont disponibles et permettent de compiler des programmes écrits en C/C++, Java et fortran pour être exécutés sur le processeur OpenRISC. Un SDK (*Software Development Kit*) complet intègre les outils GNU composés d'un compilateur GCC, Binutils pour l'édition de liens et GDB (*Gnu DeBugger*) pour le débogage. Le processeur OR1200 possède un code source bien clair et une architecture relativement simple (cinq étages pipeline) comparé à l'architecture de leon3 (sept étages pipeline). Il présente alors un bon candidat pour notre méthodologie de réduction. Ce processeur est interfaçable grâce à un bus Wishbone. Cette interface présente des bus synchrones de données et d'adresses avec de multiple maîtres et esclaves.

Le NIOS II a été choisi finalement pour tester un processeur soft-core dans l'implémentation d'une architecture massivement parallèle. Vu que nous disposons des FPGA Altera, ce choix est bien justifié et élimine le choix du MicroBlaze. Étant donné que son code source n'est pas disponible, ce processeur n'est utilisé que dans la méthodologie de réplication.

Dans la section suivante, nous détaillons les deux méthodologies de conception appliquées au processeur miniMIPS. Le même principe est suivi pour les autres processeurs. Nous nous contenterons alors de résumer leur conception tout en mettant en exergue les différences d'implémentation si elles existent.

#### 4.2.1.2 Processeur miniMIPS

Le miniMIPS [81] est un processeur pipeliné à cinq étages (voir figure 4.4) avec un jeu d'instructions MIPS1, dans le quel les instructions nécessitent généralement un seul cycle pour s'exécuter. Les cinq étages du pipeline sont :

1. IF : *Instruction Fetch* : recherche de l'instruction ;
2. ID : *Instruction Decode* : décodage de l'instruction, extraction des opérandes et calcul de l'adresse de l'instruction suivante ;
3. IE : *Instruction Execute* : exécution de l'instruction ;
4. Mem : *Memory* : accès mémoire ;
5. WB : *Write Back* : écriture du résultat.

De point de vue implémentation, ce processeur présente les modules VHDL suivants :
- pf : calcul du compteur du programme ;
- ei : recherche de l'instruction ;
- di : décodage ;
- ex : exécution de l'instruction ;
- mem : accès mémoire ;
- bus_ctrl : gère l'interface avec la mémoire ;
- renvoi : module de détection d'aléas ;

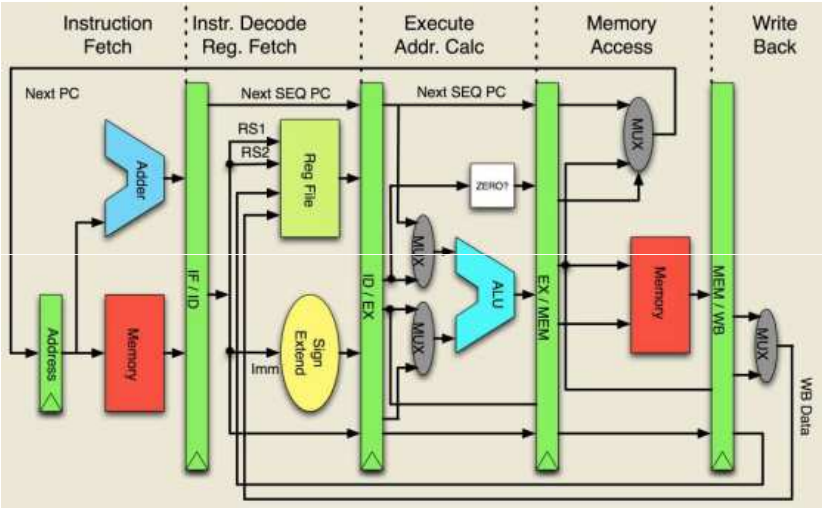



FIGURE 4.4 – *Pipeline miniMIPS à 5 étages*

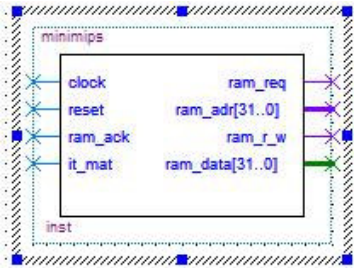



FIGURE 4.5 – *Schéma bloc de miniMIPS*

TABLE 4.2 – Interface du processeur miniMIPS

| Signal | E/S[8] | Taille (bits) | Description |
|---|---|---|---|
| clock | E | 1 | horloge système |
| reset | E | 1 | mise à zéro |
| ram_adr | S | 32 | adresse |
| ram_r_w | S | 1 | bit de lecture ou écriture |
| ram_data | E/S | 32 | données |
| ram_ack | S | 1 | acquittement |
| ram_ben | S | 4 | type de transaction |
| ram_req | S | 1 | requête d'accès à la mémoire |
| it_mat | E | 1 | interruption matérielle |

– syscop : opère sur les instructions du coprocesseur ;
– predict : prédicteur de branchement.

Le schéma bloc du processeur miniMIPS, après compilation et synthèse, est présenté dans la figure 4.5.

8. Entrée/Sortie

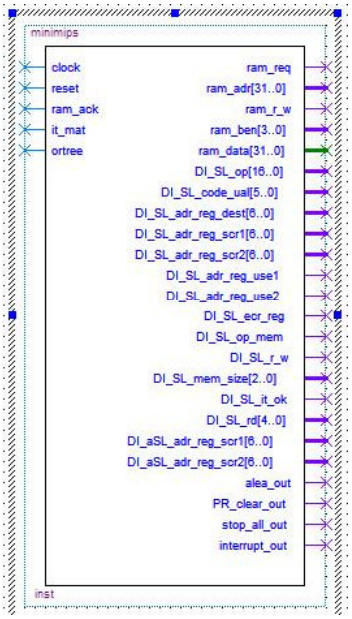



FIGURE 4.6 – *Schéma bloc de l'ACU*

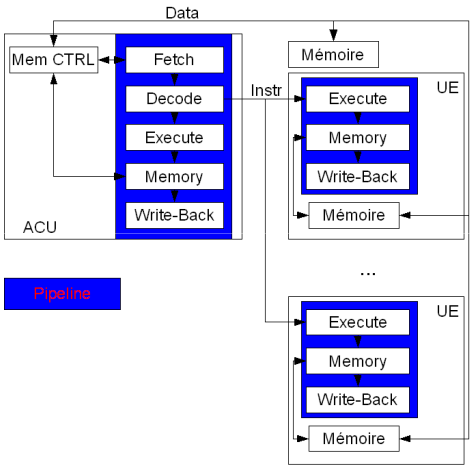



FIGURE 4.7 – *Réduction du processeur miniMIPS*

Ce processeur présente une interface simple avec des signaux suffisants pour assurer le bon fonctionnement de l'architecture parallèle. Vu qu'il était aussi le premier processeur utilisé pour construire le système mppSoC, son interface va être adoptée comme référence pour les autres IP processeurs. Le tableau 4.2 détaille cette interface. Les six premiers signaux listés dans le tableau 4.2 seront toujours retenus pour définir les interfaces de connexion entre les différents IPs de mppSoC. Le bus bidirectionnel de transfert de données peut être remplacé par deux bus (entrée et sortie) selon le processeur.

### Méthodologie de réduction

Afin d'appliquer la méthodologie de réduction, quelques mineures additions sont ajoutées au processeur miniMIPS pour concevoir l'ACU, alors que ses parties d'extraction d'instructions et de décodage sont supprimées dans le PE assurant ainsi une meilleure intégration sur puce et réduisant la consommation. Seulement une petite partie du décodage concernant l'extraction des registres est laissée. Une particularité intéressante est que les modules du pipeline sont les mêmes pour l'ACU et les PEs. La différence est que plusieurs signaux d'entrée non utilisés de ces modules sont mis à 0 pour qu'ils soient retirés des PEs. Les connexions entre l'ACU et les PE sont des fils qui relient l'étage de décodage de l'ACU aux étages d'exécution des PE. Une micro-instruction dans le cas du processeur miniMIPS est de longueur 73 bits. La figure 4.6 montre le schéma bloc de l'ACU en présentant l'interface des micro-instructions envoyées vers les PEs.

Dans l'ACU, seul l'étage de décodage des instructions a été modifié permettant le décodage des instructions parallèles. Si une instruction est parallèle, l'ACU envoie une micro-instruction NOP (*No OPeration*) dans son propre pipeline sinon il envoie NOP dans le pipeline des PEs. Cette manière de décodage est détaillée dans l'annexe A. La figure 4.7 illustre la méthodologie de réduction appliquée au processeur miniMIPS. Le PE sélectionne les deux

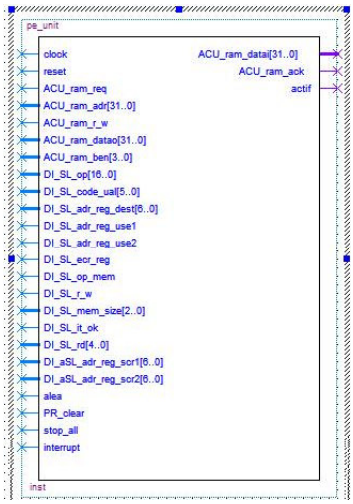



FIGURE 4.8 – *Schéma bloc du PE*

opérandes à envoyer à son étage d'exécution (soit la valeur reçue de l'ACU, soit la valeur de ses registres). Le choix est déterminé par deux signaux qui viennent de l'ACU. Si un PE n'est pas actif, il annule l'écriture dans les registres de l'instruction et annule les opérations mémoires. La figure 4.8 présente le schéma bloc du PE. Nous voyons clairement qu'il présente l'interface des micro-instructions qu'il reçoit à partir de l'ACU. L'interface entre l'étage de décodage et l'étage d'exécution de l'ACU ou des PE n'est pas la même. L'interface vers les PE contient moins de fils. Ce choix a pour but de diminuer le nombre de connexions entre l'ACU et les PE afin de diminuer la complexité du routage et augmenter la fréquence du système.

Dans un pipeline d'instruction, il faut faire attention aux dépendances inter-instructions. Le processeur miniMIPS contient un prédicteur de branchement qui va être enlevé dans le PE. Il faut cependant assurer les signaux nécessaires, sorties de l'étage d'exécution du PE, pour ce module. La figure 4.9 montre la connexion de ce prédicteur aux autres étages du processeur. Le processeur contient aussi un module de renvoi pour détecter les dépendances de données (aléas) entre les instructions. La détection des dépendances de données est calculée dans l'ACU. Le module de renvoi implémenté prend les résultats des registres de fin des étages de décodage, d'exécution et de mémoire et l'envoie vers l'étage de décodage pour l'ACU et vers l'étage d'exécution pour les PEs (vu qu'ils n'ont pas d'étage de décodage). De ce fait, ce module a été modifié dans les PEs. L'implémentation de ce module introduit une perte de un cycle pour les dépendances entre instructions arithmétiques et deux cycles entre une instruction arithmétique et une instruction mémoire.

Du côté de la programmation, le processeur miniMIPS est composé de 57 instructions réparties comme suit :

- 33 instructions arithmétiques et logiques ;
- 12 instructions de branchement ;
- 7 instructions mémoire ;
- 5 instructions système.

Notre idée est de garder le même jeu d'instructions pour l'ACU et pour les PEs. En effet, dans la plupart des travaux de conception des PEs, nous trouvons qu'ils sont menés par

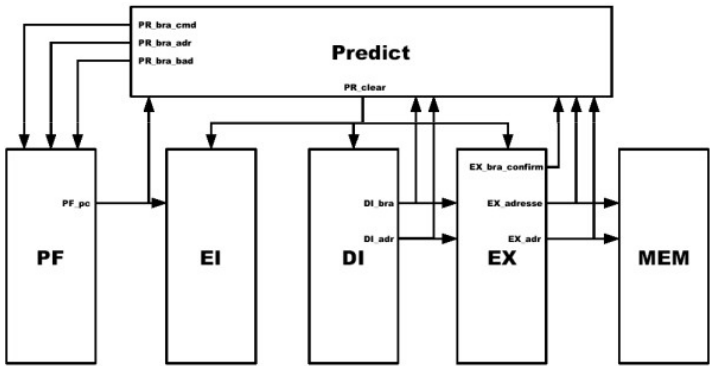



FIGURE 4.9 – *Connexion du module prédicteur de branchement*

un jeu d'instructions spécifique. De cette manière, nous perdons l'avantage que l'ACU et les PEs ont les mêmes instructions et nous déplaçons la complexité vers la chaîne d'outil de développement. Une extension a été ajoutée au jeu d'instructions miniMIPS contenant : des instructions parallèles d'opérations arithmétiques et binaires, d'accès-mémoire par les PEs, de gestion de l'activité des PEs, de communication et des instructions séquentielles d'accès-mémoire par l'ACU. Donc il y a des instructions dupliquées et d'autres qui sont créées pour les PEs. Les instructions de branchement n'ont pas été dupliquées et ne peuvent s'exécuter que par l'ACU. L'implémentation est générique et peut être réutilisée pour d'autres instructions en les ajoutant dans la table d'instructions. Chaque instruction 32-bits présente un code opération codé sur 6 bits appelé OPCODE et les autres 24 bits sont utilisés pour désigner les opérandes. L'OPCODE permet de transporter l'identifiant de l'instruction à exécuter : c'est un code différent de celui utilisé pour le codage binaire des instructions. Avec ce format, on est limité à l'utilisation de 64 instructions pour les PEs. Cela nous a permis d'ajouter des instructions parallèles avec de nouveaux codes. L'ACU doit être capable de dissocier les instructions standard ou étendues qui lui sont destinées des instructions étendues qui sont destinées aux PEs. Pour cette raison, un bit "*op_vector*" a été ajouté dans le tableau des instructions. Il est à 1 si et seulement si l'instruction est parallèle. En effet, l'implémentation est basée sur un tableau de record. Chaque ligne du tableau décrit le traitement d'une instruction. L'annexe A montre le tableau d'instructions implémenté dans le cas de miniMIPS. Pour ajouter les instructions de l'extension, il a fallu modifier les records du tableau pour contenir plus d'informations pour les nouveaux modes de traitement. Ces informations ont été ajoutées dans de nouvelles colonnes.

Le miniMIPS possède un coprocesseur. Il est le même tant pour l'ACU que pour le PE. Cependant, les PEs et l'ACU n'ont pas la même définition pour tous les registres. Pour accéder au coprocesseur, nous pouvons utiliser les instructions *mfc0* et *mtc0* pour l'ACU ou *p_mfc0* et *p_mtc0* pour les PEs. Les registres communs entre les PEs et l'ACU sont :

- registre 8 : Badvaddr (adresse causant une exception d'accès-mémoire) ;
- registre 12 : statut ;
- registre 13 : cause (cause de la dernière exception).

Les registres dédiés à l'ACU sont :

- registre 14 : EPC (compteur du programme lors de la dernière exception) ;
- registre 22 : Or Tree.

Le registre dédié aux PEs est le registre 22 qui sert pour enregistrer l'identifiant (numéro)

TABLE 4.3 – Ressources occupées par l'ACU et le PE dans le cas de réduction du miniMIPS

| Processeur | Entité | ALUTs | Registres |
|---|---|---|---|
| ACU | | 3339 | 1936 |
| | Étage d'extraction d'instruction | 4 | 65 |
| | Contrôleur du compteur de programme | 171 | 32 |
| | Banc de registre | 58 | 992 |
| | Contrôleur du bus mémoire | 102 | 34 |
| | Étage de décodage | 1321 | 212 |
| | Étage d'exécution | 649 | 182 |
| | Étage mémoire | 155 | 80 |
| | Prédicteur de branchement | 345 | 207 |
| | Renvoie | 50 | 2 |
| | Coprocesseur | 30 | 129 |
| PE | | 1242 | 326 |
| | Banc de registre | 33 | 0 |
| | Étage d'exécution | 729 | 143 |
| | Étage mémoire | 149 | 42 |
| | Renvoie | 20 | 2 |
| | Coprocesseur | 27 | 139 |

du PE. Ce registre 22 qui diffère entre l'ACU et les PEs a été ajouté dans l'extension. Le registre de statut contient un bit spécial qui est le $16^{eme}$ bit servant comme bit d'activité. Sa déclaration en VHDL est la suivante : *actif <= scp_reg(12)(16).*

Le paragraphe suivant présente une évaluation de la surface consommée par l'ACU et par le PE après la réduction.

► **Évaluation de performances** Le tableau 4.3 donne l'occupation de chaque module des processeurs ACU et PE en termes de nombre de ALUT combinatoire et des registres dédiés (*Dedicated Logic registers*). L'FPGA Stratix 2S180 a été utilisé dans ce cas. Nous remarquons que le total d'éléments logiques des entités n'est pas égal au total nécessaire pour l'implémentation du processeur approprié. Ceci est bien justifié par le fait qu'il y a de la logique directement dans le module sans être dans une unité. Le tableau 4.3 montre bien qu'un PE est beaucoup plus petit que l'ACU. La taille du PE est environ trois fois plus réduite que celle de l'ACU. Ceci prouve le gain important de la méthodologie de réduction en terme de coût sur l'FPGA. Par ailleurs, cette analyse de l'utilisation de la surface FPGA montre que l'étage de décodage est le plus grand étage de l'ACU, et que l'étage d'exécution est le deuxième étage qui consomme plus de surface sur l'FPGA. Par conséquent, le fait de ne pas inclure les étages de recherche et décodage des instructions et le module de prédiction de branchement au sein d'un PE ainsi que laisser les instructions de contrôle et de branchement s'exécuter uniquement par l'ACU, permet d'atteindre une rapidité d'exécution tout en assurant une surface minimale. La comparaison entre l'ACU et le PE montre l'avantage de cette méthodologie en terme d'intégration sur une seule puce. Dans ce cas, l'FPGA Stratix 2S180 permet d'intégrer jusqu'à 96 PEs avec une petite mémoire locale de 0.5 Ko sans compter les réseaux de communication.

**Méthodologie de réplication**

Dans cette méthodologie, nous gardons le processeur miniMIPS tel quel et nous le répliquons pour obtenir le réseau de PEs. La seule différence entre l'ACU et les PEs est que l'ACU accède à sa mémoire d'instructions pour lire l'instruction à exécuter alors que les PEs reçoivent directement leurs instructions de l'ACU. Un simple test a été ajouté dans le module de décodage de l'instruction de l'ACU. En effet, notre objectif est de simplifier les PEs sans changer radicalement le jeu d'instructions afin de faciliter l'implémentation. Nous avons alors proposé de délimiter tout code séquentiel ou parallèle à travers deux instructions spécifiques. Dans le cas de miniMIPS, nous avons choisi :

- l'instruction coprocesseur *MFC0 R0,R1*, codée en hexa 0x40000800, qui consiste à stocker le contenu du registre du coprocesseur R1 dans le registre R0. Puisque R0 est toujours à 0, cette instruction n'a aucun effet. Elle est choisie pour indiquer le début d'un code séquentiel.
- l'instruction coprocesseur *MFC0 R0,R2*, codée en hexa 0x40001000, qui consiste à stocker le contenu du registre du coprocesseur R2 dans le registre R0. Cette instruction, sans effet, est utilisée pour indiquer le début d'un code parallèle.

Le test des instructions se fait juste avant le décodage des instructions lues de la mémoire d'instructions. Un bit nommé *activ_parallel*, initialisé à 0, se positionne à 1 si l'instruction est parallèle sinon il est à 0 si l'instruction est séquentielle, tel qu'illustré par le code VHDL suivant :

*activ_parallel <= '1' when EI_instr=X"40001000" else '0' when EI_instr=X"40000800";*

Le choix de ces instructions est libre au concepteur à condition de l'adapter dans le code VHDL. Les instructions doivent être choisies de telle manière qu'elle n'auront aucune influence sur le code à exécuter. De la même manière que la méthode de réduction, l'ACU envoie un NOP dans son pipeline quand il s'agit d'une instruction parallèle sinon il envoie NOP aux PEs dans le cas d'une instruction séquentielle.

▶ **Évaluation de performances** Le tableau 4.4 résume les ressources nécessaires pour chaque module des processeurs ACU et PE dans le cas de la réplication sur la carte FPGA Stratix 2S180. Nous voyons clairement que les tailles de l'ACU et du PE sont comparables. L'étage de décodage de l'ACU consomme un peu plus de ressources que celui du PE du fait du processus ajouté pour le test des instructions séquentielles/parallèles. En comparant ces résultats avec ceux obtenus dans le tableau 4.3, nous constatons que l'ACU consomme plus de surface dans le cas de la réduction comme prévu. En effet, dans la réduction du miniMIPS, l'ACU est beaucoup plus modifié avec un jeu d'instructions étendu.

Dans ce cas, un maximum de 64 PEs, avec une mémoire de 0.5 Ko chacun, peut être intégré dans l'FPGA Stratix 2S180 sans intégrer les réseaux de communication.

#### 4.2.1.3 Processeur OpenRISC OR1200

OpenRISC OR1200 [82] est un processeur libre ; son code source est disponible sous licence GNU GPL. Il présente un modèle synthétisable décrit en Verilog d'un processeur 32 bits conforme à l'architecture RISC. Son architecture interne est présentée dans la figure 4.10. Ce processeur est constitué d'un cœur CPU/DSP, d'un bloc de gestion de mémoire MMU (*Memory Management Unit*), d'une mémoire cache configurable (ICache, Dcache), d'une unité

TABLE 4.4 – Ressources occupées par l'ACU et le PE dans le cas de réplication du miniMIPS

| Processeur | Entité | ALUTs | Registres |
|---|---|---|---|
| ACU | | 3247 | 1880 |
| | Étage d'extraction d'instruction | 3 | 67 |
| | Contrôleur du compteur de programme | 171 | 32 |
| | Banc de registre | 57 | 992 |
| | Contrôleur du bus mémoire | 102 | 34 |
| | Étage de décodage | 1210 | 159 |
| | Étage d'exécution | 646 | 183 |
| | Étage mémoire | 146 | 80 |
| | Prédicteur de branchement | 348 | 202 |
| | Renvoie | 47 | 2 |
| | Coprocesseur | 30 | 129 |
| PE | | 3241 | 1885 |
| | Étage d'extraction d'instruction | 3 | 67 |
| | Contrôleur du compteur de programme | 169 | 32 |
| | Banc de registre | 56 | 992 |
| | Contrôleur du bus mémoire | 102 | 34 |
| | Étage de décodage | 1202 | 161 |
| | Étage d'exécution | 648 | 183 |
| | Étage mémoire | 155 | 80 |
| | Prédicteur de branchement | 348 | 202 |
| | Renvoie | 47 | 2 |
| | Coprocesseur | 30 | 129 |

de débogage (*Debug Unit*), d'un contrôleur d'interruption programmable (*PIC : Programmable Interrupt Controller*), d'un bloc de gestion de puissance (*Power managment*), et d'un *Tick Timer* qui mesure le temps nécessaire pour exécuter des programmes. Il peut être connecté à une mémoire à travers une interface Wishbone. Toute la documentation pour programmer ce processeur est disponible sur le site d'OpenCores [80].

Le processeur OR1200, comme le miniMIPS, possède un pipeline d'exécution de profondeur 5 étages. Il est doté de 53 instructions de taille fixe de 32 bits. Ce processeur présente les modules suivants :

- genPC : gestion du compteur de programme ;
- IF : recherche d'instruction ;
- Control : module de contrôle et décodage d'instructions ;
- Register file : registres ;
- Operand mux : lecture d'opérandes ;
- ALU : unité arithmétique et logique ;
- LSU (*Load Store Unit*) : module d'accès mémoires ;
- WB mux : écriture de résultat.

Après une phase de test du processeur, une première étape a consisté à éliminer tous les modules non nécessaires à savoir la MMU (instructions et données), le tick timer, le bloc de gestion de puissance et l'unité de débogage. Le processeur peut communiquer avec d'autres composants en échangeant des données à travers un système de bus Wishbone. Une concep-

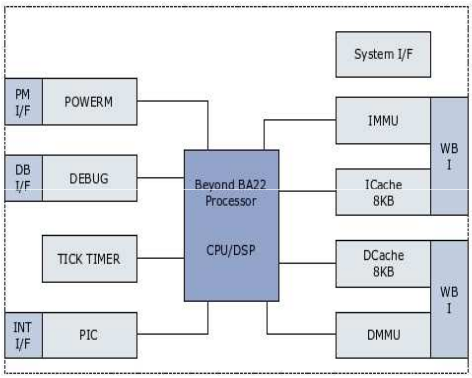



FIGURE 4.10 – *Architecture du processeur OpenRISC*

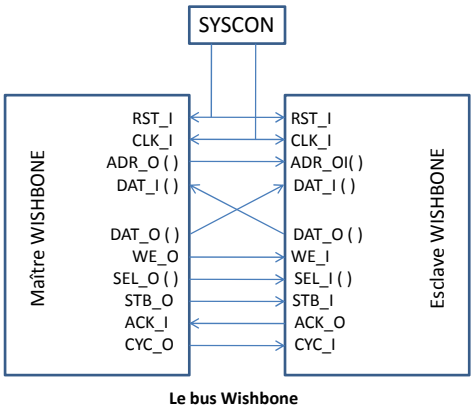



FIGURE 4.11 – *Exemple d'une connexion Wishbone point-à-point au niveau RTL*

tion typique pour un système contenant le bus wishbone implique les composants suivants :

- Maître/Esclave :
  1. Les modules maîtres qui lancent des opérations lecture/écriture.
  2. Les modules esclaves qui répondent aux opérations de lecture/écriture provenant des maîtres. Les modules esclaves ont chacun leur propre plage d'adressage sur le bus, définie lors de la description de l'architecture.
- L'arbitre du bus qui autorise le transfert de données pour un seul maître à la fois.
- Le décodeur du bus qui décode l'adresse de chaque transfert sur le bus et fournit les signaux sélectionnés.

La figure 4.11 montre un exemple du bus wishbone avec une connexion point-à-point au niveau RTL. Pour des fins de simplicité, nous avons gardé l'interface Wishbone du processeur pour ne pas perturber le fonctionnement du système. En examinant cette interface, nous avons pu sélectionner les signaux dont nous avons besoin afin d'assurer la connexion avec les réseaux de communication. Les autres signaux ont été désactivés. Les signaux utilisés, extraits de l'interface Wishbone, sont détaillés dans le tableau 4.5. Ce tableau montre bien que les signaux utilisés sont ceux de données, d'adresse, d'accès à la mémoire et d'acquittement. Ceci se rapproche de l'interface montrée dans le cas du processeur miniMIPS (voir tableau 4.2).

TABLE 4.5 – Signaux requis pour intégrer l'OpenRisc dans mppSoC

| Type | Signal | E/S | Taille (bits) | Description |
|---|---|---|---|---|
| Maître et esclave | CLK_I | E | 1 | horloge système |
| | RST_I | E | 1 | remise à zéro |
| | DAT_I | E | 32 | données d'entrée |
| | DAT_O | S | 32 | données de sortie |
| Maître | ACK_I | E | 1 | acquittement |
| | ADR_O | S | 32 | adresse de sortie |
| | WE_O | S | 1 | bit de lecture/écriture en mémoire |
| Esclave | ADR_I | E | 32 | adresse d'entrée |
| | ACK_O | S | 1 | acquittement |
| | WE_I | E | 1 | bit de lecture/écriture en mémoire |

Les paragraphes suivants décrivent les deux méthodologies de conception appliquées à ce processeur.

**Méthodologie de réduction**

La réduction du processeur OR1200 a fait l'objet du projet de mastère de l'étudiante Bouthaina Dammak (soutenu le 15 juillet 2010) au quel j'ai participé à son encadrement. Le même principe de réduction présenté avec le processeur miniMIPS est appliqué à l'OR1200. Les étages de recherche d'instructions et de décodage sont éliminés du PE, seulement les étages d'exécution ont été gardés. Par contre, ils ont été modifiés dans l'ACU pour permettre la diffusion des micro-instructions aux PEs.

Une analyse des signaux de ces principaux étages est alors faite. Suite à la quelle, nous avons pu conclure les signaux nécessaires à faire sortir pour servir comme entrées directement aux étages d'exécution des PEs. Une manière simple de faire est de dupliquer les signaux sorties de l'étage de décodage de l'ACU qui sont destinés à être connectés à l'entrée de son étage d'exécution. Ces signaux dupliqués ne seront alors activés que dans le cas des instructions parallèles. La figure 4.12 montre la réduction du pipeline de l'OR1200. Le pipeline à droite est celui de l'ACU dont le module de décodage a été modifié. Alors que le pipeline à gauche est celui du PE où les étages de décodage et d'extraction ont été éliminés. Cependant, une partie du décodage concernant les registres doit être gardée dans le PE telle qu'illustrée dans la figure 4.12. Pour la distinction entre un code séquentiel de celui parallèle, nous avons procédé à la manière la plus simple décrite dans la méthodologie de réplication de miniMIPS. Deux instructions spécifiques sont choisies pour cette fin. Ceci a demandé alors la modification de l'étage de décodage par l'ajout d'un processus de test sensible à toute nouvelle instruction communiquée via le module d'extraction. Un signal de longueur 1 bit, nommé PAR, initialisé à 0 se positionne à 1 si l'instruction est parallèle. Ce processus est implémenté en verilog comme suit :

```
always @(if_insn)
if (if_insn==32'h9ce70000)
PAR<=#1 1'b0 ;
else if (if_insn==32'he0e70000)
PAR<= #1 1'b1 ;
```

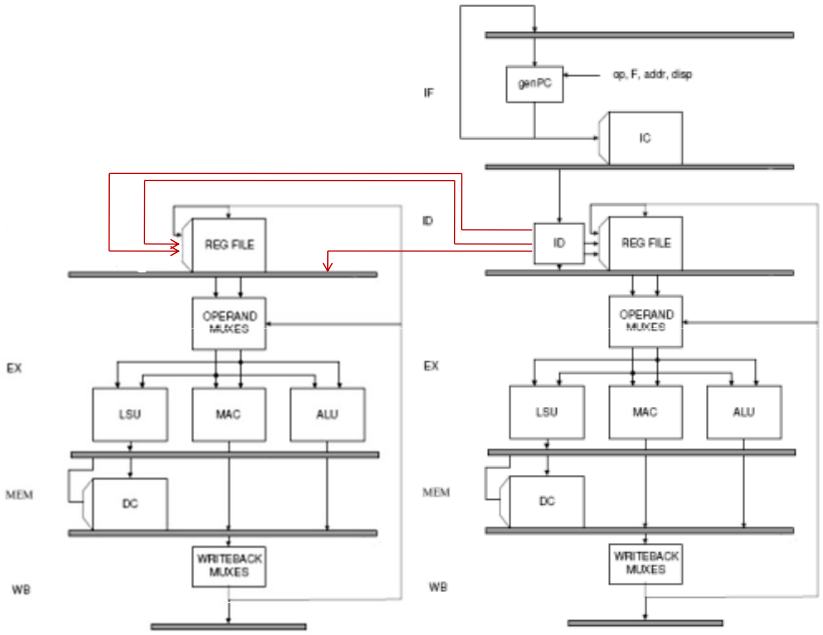



FIGURE 4.12 – *Réduction du piepline de l'OpenRISC*

En effet, les deux instructions spécifiques utilisées dans ce cas sont :

- L'instruction arithmétique d'addition immédiate *l.addi r7,r7,0* qui consiste à ajouter au contenu du registre r7 la valeur immédiate 0, d'où aucune modification ne sera effectuée sur ce registre. Cette instruction est codée 0X9CE70000 en hexa. Elle indique le début d'un traitement séquentiel.
- L'instruction arithmétique d'addition *l.add r7,r7,r0* qui consiste à ajouter au contenu du registre r7 le contenu du registre r0 (qui vaut toujours 0). Elle est codée 0XE0E70000 en hexa et indique le début d'un traitement parallèle.

Le module de décodage active ou désactive les sorties dédiées aux PEs dépendamment du signal PAR. Le code suivant montre l'affectation de quelques signaux d'entrées aux étages d'exécution des PEs :

*assign rf_addra_pe = PAR ?(if_insn[20 :16]) :(5'bzzzzz) ;*
*assign rf_addrb_pe = PAR ?(if_insn[15 :11]) :(5'bzzzzz) ;*
*assign rf_rda_pe = PAR ?(if_insn[31]) : (1'bz) ;*
*assign rf_rdb_pe = PAR ?(if_insn[30]) : (1'bz) ;*

Le bus de micro-instructions dans ce cas est de taille 142 bits. De la même manière que le miniMIPS, nous avons implémenté les registres d'identité et d'activité ainsi que le mécanisme du OR Tree. Cette méthodologie a nécessité environ deux mois sans compter la phase de test et configuration du processeur.

Comparé au processeur miniMIPS, nous remarquons que le bus de diffusion des micro-instructions, dans le cas de l'OpenRisc, consomme plus de ressources. Ceci s'explique par la différence de performances et des implémentations entre les deux processeurs. De même, nous constatons que le fait de simplifier la conception et de ne pas modifier le jeu d'instructions du processeur offre un gain très important en terme de temps de développement.

TABLE 4.6 – Ressources occupées par l'ACU et le PE dans le cas de réduction de l'OpenRisc

| Processeur | Entité | ALUTs | Registres |
|---|---|---|---|
| ACU | | 2791 | 1005 |
| | Contrôleur de programme | 217 | 30 |
| | Étage d'extraction d'instruction | 70 | 68 |
| | banc de registres | 125 | 114 |
| | Étage de décodage | 324 | 313 |
| | Étage d'exécution | 419 | 16 |
| | Étage mémoire | 173 | 66 |
| | Étage Write Back | 344 | 33 |
| | Contrôleur de bus wishbone | 128 | 9 |
| PE | | 1316 | 431 |
| | Banc de registre | 122 | 114 |
| | Étage d'exécution | 416 | 16 |
| | Étage mémoire | 173 | 64 |
| | Étage Write Back | 313 | 32 |
| | Contrôleur de bus wishbone | 84 | 9 |

► **Évaluation de performances** Le tableau 4.6 donne l'occupation des ressources des processeurs ACU et PE sur l'FPGA Stratix 2S180. Nous remarquons alors que la taille du PE est environ deux fois plus réduite que celle de l'ACU. En analysant les ressources consommées par chaque module, nous constatons qu'elles sont comparables. Les étages les plus grands sont ceux d'exécution et d'écriture de résultats. Ceci explique le petit pourcentage de réduction gagné dans ce cas par rapport à celui du miniMIPS.

Sans intégrer les réseaux de communication, une configuration simple de mppSoC comportant un ACU et un réseau de PEs peut avoir jusqu'à 40 PEs sur l'FPGA Stratix 2S180, où chaque PE est connecté à une mémoire locale de données de taille 0.5 Ko. Ce nombre peut être encore augmenté en utilisant un FPGA de plus grande taille.

### Méthodologie de réplication

Dans ce cas, les processeurs OpenRISC ont été répliqués afin de construire le système parallèle. Aucune simplification du processeur n'a eu lieu. La seule modification est effectuée au niveau de l'étage de décodage de l'ACU permettant de distinguer entre instruction séquentielle ou parallèle. L'implémentation se base sur le même principe que celui utilisé dans la méthodologie de réduction. Dans le cas d'une instruction parallèle, l'ACU envoie toute l'instruction de taille 32 bits aux PEs. Les PEs reçoivent simultanément des instructions complètes et chacun s'occupe à la décoder et l'exécuter dans son propre pipeline. Cette méthodologie simplifie beaucoup la conception. Cependant, dans ce cas les PEs consomment plus de ressources sur FPGA ce qui limite leur nombre.

► **Évaluation de performances** Le tableau 4.7 donne l'occupation des ressources des processeurs ACU et PE sur l'FPGA Stratix 2S180. Nous constatons alors que les deux processeurs occupent approximativement les mêmes ressources sur FPGA. L'ACU est un peu plus grand que le PE vu que son étage de décodage a été légèrement modifié.

TABLE 4.7 – Ressources occupées par l'ACU et le PE dans le cas de réplication de l'OpenRisc

| Processeur | Entité | ALUTs | Registres |
|---|---|---|---|
| ACU | | 2488 | 961 |
| | Étage d'extraction d'instruction | 70 | 65 |
| | Contrôleur de programme | 208 | 30 |
| | banc de registres | 126 | 114 |
| | Étage de décodage | 280 | 291 |
| | Étage d'exécution | 420 | 16 |
| | Étage mémoire | 174 | 65 |
| | Étage Write Back | 329 | 33 |
| | Contrôleur wishbone | 129 | 9 |
| PE | | 2462 | 968 |
| | Étage d'extraction d'instruction | 70 | 65 |
| | Contrôleur de programme | 209 | 30 |
| | Banc de registres | 127 | 114 |
| | Étage de décodage | 275 | 293 |
| | Étage d'exécution | 420 | 16 |
| | Étage mémoire | 173 | 65 |
| | Étage Write Back | 329 | 33 |
| | Contrôleur wishbone | 130 | 9 |

Si nous considérons juste l'intégration de l'ACU et du réseau de PEs sans compter les réseaux de communication, un maximum de 28 PEs, où chaque PE est connecté à une mémoire locale de données de taille 0.5 Ko, peut être intégré sur l'FPGA Stratix 2S180. Ceci montre le faible taux d'intégration comparé à un design conçu avec la méthodologie de réduction.

#### 4.2.1.4 Processeur NIOS II

Le processeur « softcore » NIOS II, de Altera, est basé sur une architecture RISC. Ce processeur existe en trois versions : rapide NIOSII/f (*fast*), économique NIOSII/e et standard NIOSII/s. C'est un cœur de processeur configurable (taille des instructions, nombre de registres, ajout de périphériques, etc.) à volonté selon les besoins. Il possède un nombre paramétrable de différents périphériques (voir figure 4.13) à savoir : mémoire RAM/ROM, UART pour gérer une ligne série, timer, PIO (*Parallel Input Output*), etc. Hormis ces périphériques implémentés par Altera, l'utilisateur a la possibilité d'en ajouter qu'il aura lui-même écrits. Ils sont vus par le NIOS comme de simples emplacements mémoires accédés autant en lecture qu'en écriture avec la possibilité de gestion d'interruptions. Ce processeur embarqué est optimisé pour la logique programmable d'Altera et l'intégration système sur un circuit programmable (SOPC). Avec l'outil de développement de système SOPC Builder, les concepteurs peuvent adapter le processeur NIOS et ses périphériques pour créer le système exact dont ils ont besoin.

L'architecture interne du processeur NIOS est conçue pour fournir plusieurs avantages tels que :

– Implémentation efficace dans les composants FPGA d'Altera :

1. Nombre minimal d'éléments logiques ;

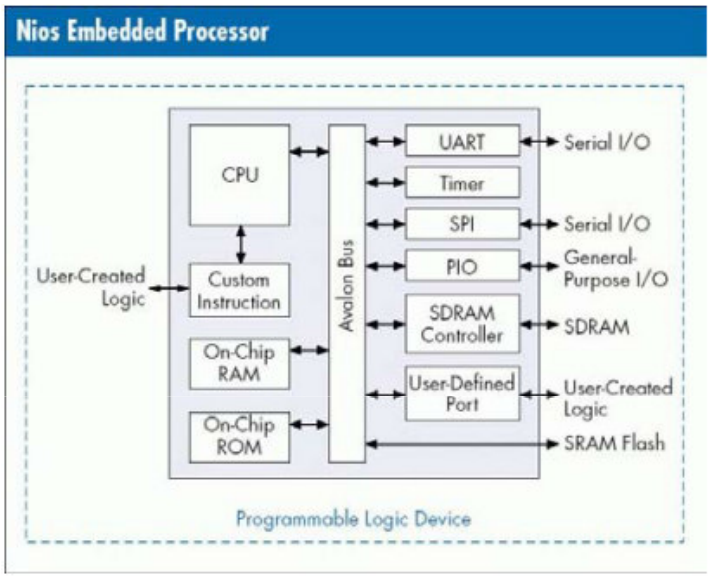



FIGURE 4.13 – *Processeur embarqué NIOS d'Altera*

2. Utilisation du minimum de mémoire ;
3. Fréquence d'horloge maximale.

– Intégration de système sans effort :

1. Interface mémoire simple ;
2. Placement de périphériques configurables standard ;
3. Outil SOPC Builder qui crée la logique d'interface du bus Avalon entre l'unité centrale de traitement, les périphériques et la mémoire.

Étant donné que le code source de ce processeur est non accessible, la seule méthodologie de conception qu'on peut appliquer est celle de réplication. Cette méthodologie est mieux expliquée dans le paragraphe suivant. L'adaptation de NIOS aux besoins de mppSoC est aussi détaillée.

### Méthodologie de réplication

Le processeur NIOS peut communiquer avec les composants qui l'entourent à travers un bus Avalon [3]. Avalon est une interface qui indique les connexions de ports entre les composants maître et esclave, et indique la synchronisation par laquelle ces composants communiquent. Afin d'intégrer un grand nombre de processeurs NIOS sur l'FPGA, nous avons choisi d'utiliser la version économique puisqu'elle qui consomme moins de surface. Un adaptateur a aussi été développé permettant de connecter le processeur NIOS aux réseaux de communication de mppSoC. Il présente une interface avalon d'un côté et de l'autre côté une interface adéquate avec les réseaux mppSoC. En utilisant le SOPC Builder, notre architecture parallèle peut être construite comme un assemblage de sous-systèmes. Chaque sous-système se compose d'un processeur NIOSII/e connecté par un *timer*, un périphérique ID *SysID* (qui donne une identité unique pour le processeur dans le système) et une mémoire locale de données (dénotée Mi et DATAM pour le PEi et l'ACU respectivement). De

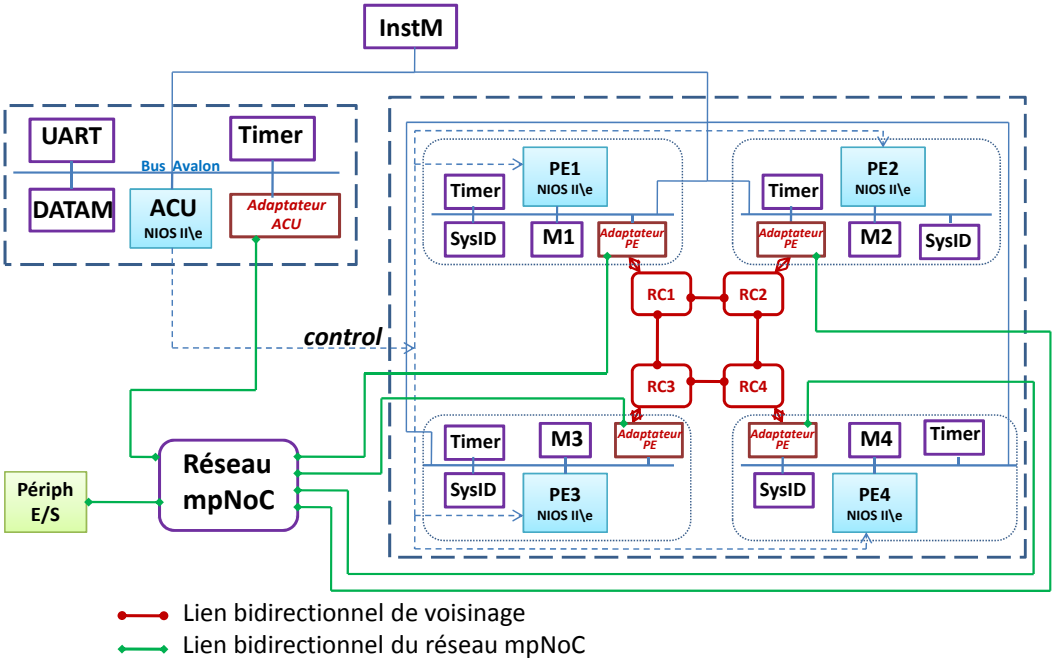
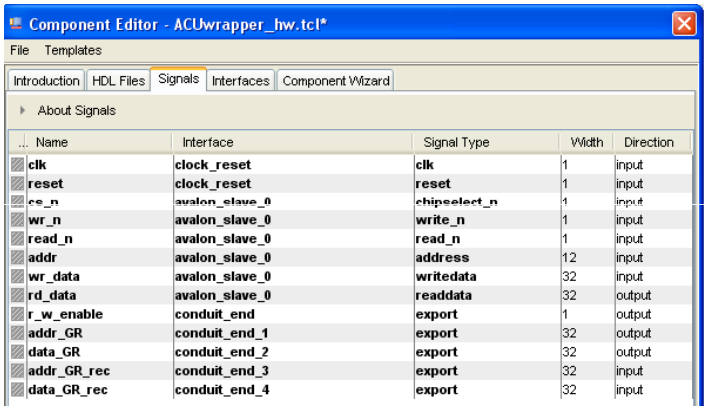



FIGURE 4.14 – *Configuration mppSoC à base de NIOS*

plus, l'ACU se distingue des PEs par sa connexion à travers une interface JTAG UART au PC en utilisant le câble USB Blaster qui permet de déboguer l'application. Chaque processeur dans le système est aussi connecté à un périphérique propriétaire (*custom peripheral*), appelé adaptateur mppSoC. Cet adaptateur permet d'assurer les communications entre les processeurs et les réseaux. La figure 4.14 illustre le système mppSoC à base de NIOS. Pour des raisons de simplification, seulement quatre PEs ont été présentés. Tous les processeurs peuvent recevoir leurs instructions à partir d'une mémoire d'instructions. Vu que le code source du NIOS est non accessible, les PEs reçoivent directement leurs instructions de la mémoire d'instructions et non de l'ACU. Un contrôle s'effectue au niveau de la mémoire d'instructions pour distinguer entre instruction séquentielle ou parallèle. Le même principe de test décrit dans le cas de l'OpenRisc est suivi. Si l'instruction est séquentielle, elle sera envoyée à l'ACU alors qu'un NoP sera envoyé aux PEs et vice versa s'il s'agit d'une instruction parallèle.

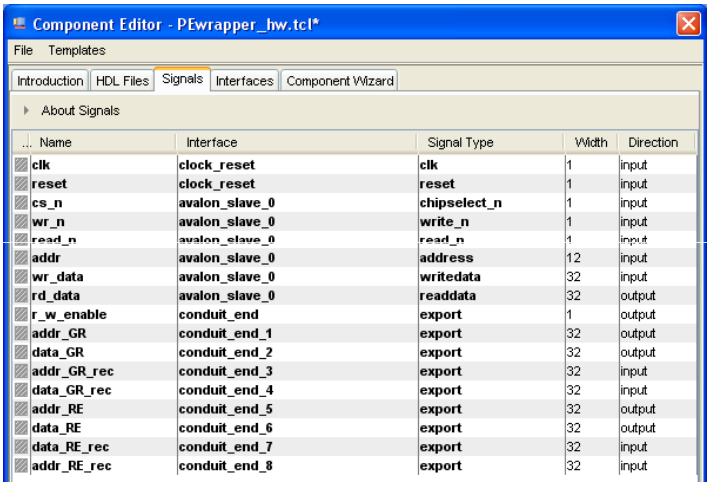
Le composant clé dans ce système est alors l'adaptateur qui a été conçu spécifiquement afin d'utiliser le NIOS dans mppSoC. Il sera détaillé dans le paragraphe suivant.

► **Adaptateur NIOS/MPPSoC** Avalon est une spécification d'interfaces pour des composants sur puce. Elle définit les transferts entre un (ou plusieurs) périphériques et une structure d'interconnexion. Les composants connectés au processeur NIOS sont mappés dans l'espace mémoire, ce qui est référencé comme Avalon-MM (*Memory Mapped*). Le système de communication est maître/esclave. L'adaptateur implémenté se base sur cette interface. Son rôle est de connecter les processeurs aux réseaux de communication appropriés (voisinage et mpNoC). Il a deux interfaces : une interface avalon esclave connectée au NIOS et une interface propriétaire dite *conduit interface* permettant d'exporter les signaux vers l'extérieur du système. Afin d'assurer les communications, nous avons besoin des adresses et données ve-




| Name | Interface | Signal Type | Width | Direction |
|---|---|---|---|---|
| clk | clock_reset | clk | 1 | input |
| reset | clock_reset | reset | 1 | input |
| cs_n | avalon_slave_0 | chipselect_n | 1 | input |
| wr_n | avalon_slave_0 | write_n | 1 | input |
| read_n | avalon_slave_0 | read_n | 1 | input |
| addr | avalon_slave_0 | address | 12 | input |
| wr_data | avalon_slave_0 | writedata | 32 | input |
| rd_data | avalon_slave_0 | readdata | 32 | output |
| r_w_enable | conduit_end | export | 1 | output |
| addr_GR | conduit_end_1 | export | 32 | output |
| data_GR | conduit_end_2 | export | 32 | output |
| addr_GR_rec | conduit_end_3 | export | 32 | input |
| data_GR_rec | conduit_end_4 | export | 32 | input |

FIGURE 4.15 – *Interface de l'adaptateur connecté à l'ACU*




| Name | Interface | Signal Type | Width | Direction |
|---|---|---|---|---|
| clk | clock_reset | clk | 1 | input |
| reset | clock_reset | reset | 1 | input |
| cs_n | avalon_slave_0 | chipselect_n | 1 | input |
| wr_n | avalon_slave_0 | write_n | 1 | input |
| read_n | avalon_slave_0 | read_n | 1 | input |
| addr | avalon_slave_0 | address | 12 | input |
| wr_data | avalon_slave_0 | writedata | 32 | input |
| rd_data | avalon_slave_0 | readdata | 32 | output |
| r_w_enable | conduit_end | export | 1 | output |
| addr_GR | conduit_end_1 | export | 32 | output |
| data_GR | conduit_end_2 | export | 32 | output |
| addr_GR_rec | conduit_end_3 | export | 32 | input |
| data_GR_rec | conduit_end_4 | export | 32 | input |
| addr_RE | conduit_end_5 | export | 32 | output |
| data_RE | conduit_end_6 | export | 32 | output |
| data_RE_rec | conduit_end_7 | export | 32 | input |
| addr_RE_rec | conduit_end_8 | export | 32 | input |

FIGURE 4.16 – *Interface de l'adaptateur connecté au PE*

nant du processeur ainsi que le signal de lecture/écriture indiquant la nature de l'opération. Les figures 4.15 et 4.16 montrent les différents signaux requis pour implémenter l'adaptateur connecté à l'ACU et aux PEs respectivement. Les signaux avec un type *export* sont exportés à l'extérieur du système SOPC et utilisés pour se connecter aux réseaux.

L'adaptateur est responsable de transférer les demandes du processeur (addr et wr_data) au réseau utilisé (selon le décodage de l'adresse). De la même manière, il achemine les données et adresses reçus vers les processeurs (addr_GR_rec et data_GR_rec dans le cas d'une communication via le mpNoC ; addr_RE_rec et data_RE_rec dans le cas d'une communication de voisinage).

Les deux figures 4.17 et 4.18 présentent les machines à états finis (*FSM : Finite State Machines*) des deux adaptateurs dédiés pour l'ACU et les PEs respectivement. Les deux FSM présentées correspondent aux interfaces de lecture et écriture du côté de l'interface Avalon connectée au processeur NIOS. Les états "Envoi" et "Recep" correspondent respectivement à l'envoi ou à la réception de données à travers l'interface avalon. L'état "Envoi_IP" indique le transfert de données de l'adaptateur vers le réseau approprié et vice versa pour l'état "Recep_IP". Nous remarquons que ces FSM sont simplifiées afin de réduire les ressources FPGA

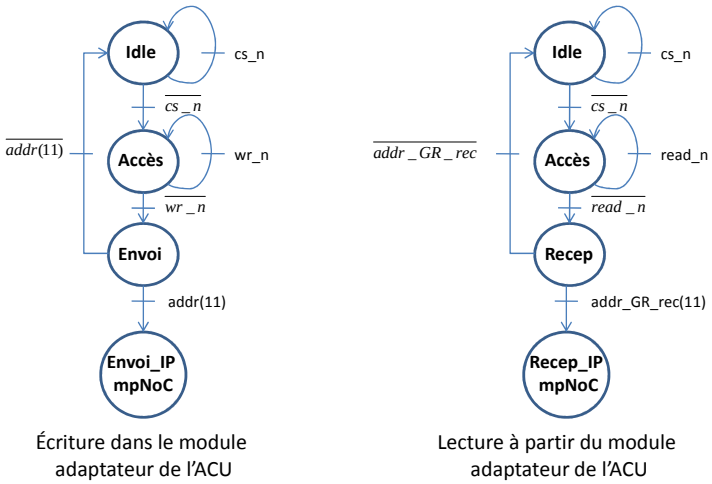



FIGURE 4.17 – *FSM de l'adaptateur de l'ACU*

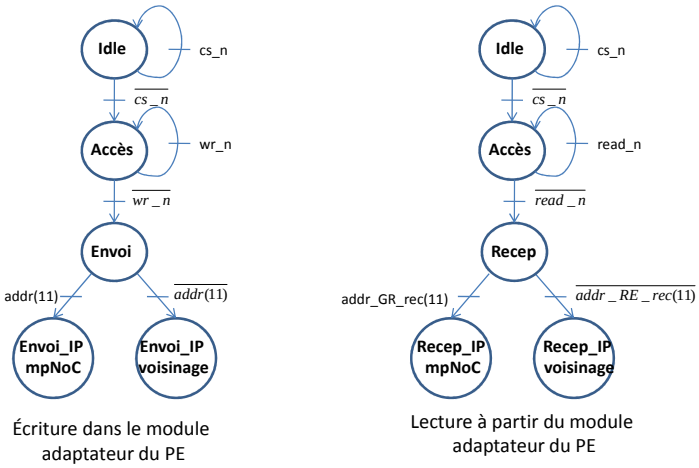



FIGURE 4.18 – *FSM de l'adaptateur du PE*

nécessaires pour l'implémentation des adaptateurs.

► **Évaluation de performances** Après compilation et synthèse, nous avons trouvé que l'adaptateur dédié au PE a besoin de 36 ALUTs et 118 registres, ce qui fait une surface totale de 122 éléments logiques (<1% de la surface totale de l'FPGA). L'adaptateur dédié à l'ACU nécessite 2 ALUTs et 75 registres seulement vu qu'il ne peut être connecté qu'avec le mpNoC. Sa surface totale est de 77 éléments logiques (<1% de la surface totale de l'FPGA). Ces résultats de synthèse montrent bien que cette solution d'adaptateur n'est pas coûteuse en terme de surface. L'FPGA stratix 2S180 peut intégrer jusqu'à 72 NIOS avec une mémoire locale de 0.5 Ko chacun sans les réseaux de communication. Ceci confirme les résultats prévus vu que le NIOS est un processeur optimisé pour être intégré dans des FPGA de technologie Altera.

TABLE 4.8 – Occupation FPGA des mémoires

| Type de mémoire | ALUTs [9] | ALMs [10] | Bits mémoires | M4Ks [11] |
|---|---|---|---|---|
| Mémoire Altera | 0 | 0 | 32768 | 8 |
| Mémoire portable | 3557 | 1888 | 0 | 0 |

### 4.2.2 Mémoires

La mémoire forme un composant ayant une grande influence sur notre système. En effet, les algorithmes sur des flots de données effectuent un grand nombre d'accès mémoire. Celle-ci est une grande consommatrice d'énergie et peut occuper jusqu'à 90% de la surface du système. Bien que la conception des processeurs suit une méthodologie précise, celle des mémoires est assez simple et consiste à utiliser directement les IPs existants. Dans mppSoC, nous distinguons deux types de mémoires : séquentielle et parallèle. La mémoire séquentielle est constituée de la mémoire d'instructions et de données connectées à l'ACU ; alors que la mémoire parallèle forme l'ensemble des mémoires de données locales attachées aux PEs. MppSoC se base sur une architecture à mémoire distribuée. Celle-ci offre plusieurs avantages :

- temps d'accès à la mémoire court ;
- grand nombre de PEs pouvant être intégrés ;
- architecture flexible et extensible ;
- petites mémoires distribuées.

Nous notons de plus que la taille des mémoires est paramétrique, ce qui offre au concepteur le choix d'implémenter la quantité de mémoire nécessaire à exécuter son application.

De point de vue implémentation, chaque PE peut accéder à sa mémoire à une même adresse vu que chacun est connecté à sa mémoire locale privée. Cela permet de simplifier l'adressage. Ce mécanisme est assuré à travers un composant matériel nommé *chip select* connecté au processeur PE et permettant d'activer le transfert de données vers la mémoire locale en cas de besoin.

Deux types de mémoires sont offerts dans la bibliothèque d'IPs mppSoC : une mémoire spécifique Altera et une mémoire portable. Les fichiers VHDL de ces deux mémoires sont donnés en annexe E. L'utilisateur peut aussi intégrer son propre IP mémoire. Le tableau 4.8 montre les ressources consommées par les deux mémoires, chacune de taille 1 Ko, sur un FPGA stratix 2S180F1020C5. Nous remarquons alors qu'une mémoire Altera est optimisée afin d'être implémentée sur les blocs mémoires de l'FPGA Altera ; alors qu'une mémoire portable consomme les éléments logiques de l'FPGA. En fonction des besoins et contraintes, le concepteur pourra alors choisir l'IP mémoire qui le convient.

### 4.2.3 Réseau de voisinage

Le réseau de voisinage joue un rôle important dans les systèmes parallèles. Il assure les communications inter-PE. Il permet en effet de communiquer les données entre les PEs voisins et de charger les mémoires des PEs avec les données envoyées. Dans mppSoC, le réseau de voisinage peut être configuré selon cinq topologies différentes : linéaire, anneau, maille,

9. Adaptive Look Up Table
10. Adaptive Logic Module
11. Blocs mémoire composés de 4608 bits

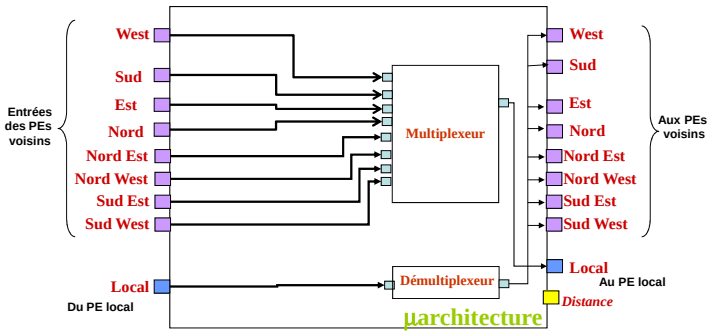



FIGURE 4.19 – *Architecture du routeur de voisinage*

tore ou Xnet (maille avec des connexions sur les diagonales). Se basant sur le concept de la réutilisation d'IPs, le réseau est formé par un contrôleur et un ensemble de routeurs, qui sont au nombre de PEs, chacun connecté au PE correspondant. Ces routeurs sont connectés selon une topologie définie contrôlée par le contrôleur de réseau et permettent de communiquer les données entre les PEs [9].

Le routeur présente l'élément principal du réseau. Cet IP peut transférer la donnée dans une direction spécifique (Nord, Sud, Est, Ouest, Nord Est, Nord Ouest, Sud Est ou Sud Ouest selon la topologie) telle que demandée par la source. La figure 4.19 montre l'architecture interne du routeur. Ce routeur reçoit des données de son PE et d'un autre routeur et les transmets au routeur correspondant et à son PE respectivement. Il contient en effet un multiplexeur 8 :1 afin de détecter la présence d'une donnée venant d'un PE voisin et de l'acheminer vers le PE local. Cette donnée sera alors directement stockée dans la mémoire du PE à l'adresse demandée. Un démultiplexeur 1 :8 permet de faire le travail inverse vu qu'il est responsable de communiquer la donnée du PE local vers la destination requise. Les destinations possibles sont au nombre de huit (nombre de destinations maximales dans le cas de Xnet) qui sont activées ou désactivées dépendamment de la topologie implémentée.

Les données communiquées via le routeur ne sont pas stockées dans des files de type FIFO vu que les communications de voisinage sont synchrones s'effectuant toutes dans une seule direction à un instant donné. En effet dans une communication SIMD synchrone où tous les PEs effectuent la même opération à un instant donné, nous n'avons pas besoin d'inclure des tampons ou bien de la logique supplémentaire pour gérer la congestion. Une fonction d'interconnexion SIMD est une bijection, et donc le transfert de données s'effectue sans conflits. À chaque communication, chaque routeur active seulement un lien vers la direction requise. Les autres liens vers les autres directions seront désactivés. Cela permet de réduire la consommation de puissance.

Le réseau généré peut aussi supporter des communications inter-PE entre deux voisins qui ne sont pas directement connectés, par exemple espacés par un PE intermédiaire. Par conséquent, un port de sortie permet d'indiquer la distance demandée entre le PE source et le PE destinataire. L'implémentation actuelle de mppSoC permet des communications inter-PE distantes entre deux PEs séparés par un PE intermédiaire, telle qu'illustré dans la figure 4.20. Cependant, nous pouvons ajouter d'autres communications distantes. Ceci demande d'ajouter plus de logique de contrôle ce qui consomme plus de ressources sur l'FPGA. La connexion entre les routeurs est gérée au niveau du fichier VHDL principal connectant les différents composants du système (*top level*). À ce niveau, nous utilisons un code générique

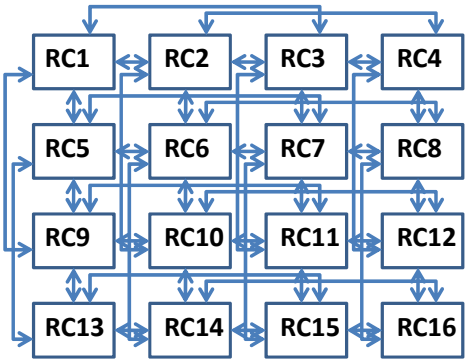



FIGURE 4.20 – *Connexions entre les RC dans une topologie maille*

afin d'établir les connexions voulues en fonction de la topologie du réseau choisie. L'exemple de code VHDL suivant montre les connexions des routeurs pour une topologie de voisinage linéaire :

```
cond1 : if (Topreg=4) generate - - Topologie linéaire
datamesh(0)(0) <= DataReg(0)(0)(2) ;
adrmesh(0)(0) <= AdrReg(0)(0)(2) ;
 d0 : for i in 1 to (sl_nb_column-2) generate
datamesh(0)(i) <= DataReg(0)(i-1)(3) ;
datamesh(0)(i) <= DataReg(0)(i+1)(2) ;
adrmesh(0)(i) <= AdrReg(0)(i-1)(3) ;
adrmesh(0)(i) <= AdrReg(0)(i+1)(2) ;
end generate d0 ;
datamesh(0)(sl_nb_column-1) <= DataReg(0)(sl_nb_column-2)(3) ;
adrmesh(0)(sl_nb_column-1) <= AdrReg(0)(sl_nb_column-2)(3) ;
end generate cond1 ;
```

Étant donné que chaque routeur communique dans un seul sens à un instant donné, sa conception est simple avec une petite taille et assure des communications de voisinage limitées seulement par le temps de propagation des fils. Un routeur nécessite environ 4 ALUTs, 62 ALMs et 64 registres sur un FPGA stratix 2S180 (<1%), ce qui montre la simplicité de son architecture. Les communications de voisinage sont gérées par des instructions de communication, qui seront détaillées dans la section 4.3, permettant à chaque PE de communiquer des données à son voisin. Étant donné que cette communication est totalement contrôlée par des instructions et vu que l'exécution est orchestrée par l'ACU, les liens de communication sont déterministes et exempts de problèmes de congestion ou de blocage. En outre, chaque PE n'a besoin que d'un multiplexeur 8 :1 et d'un démultiplexeur 1 :8 pour ses communications de voisinage, telle que décrit précédemment. Cela confirme bien la simplicité du routeur.

Les instructions de communication se basent sur des lectures/écritures en mémoire. Un processus teste continuellement l'adresse émise par le processeur élémentaire. Dés qu'une adresse de voisinage est détectée (format connue) la donnée sera acheminée vers le routeur qui s'occupe par la communiquer vers la direction demandée.

Prenons l'exemple du processeur miniMIPS. La figure 4.21 illustre les connexions entre le routeur, le PE ainsi que sa mémoire locale. Chaque PE est connecté à son routeur de voisinage via les signaux d'adresse, de donnée et de bit de lecture/écriture mémoire. Dés qu'une adresse de voisinage est détectée accompagnée par une demande d'écriture mémoire (bit de lecture/écriture activé à 1), elle sera connectée avec la donnée aux entrées du routeur. Le routeur décode la destination demandée à partir de l'adresse reçue et active alors son port

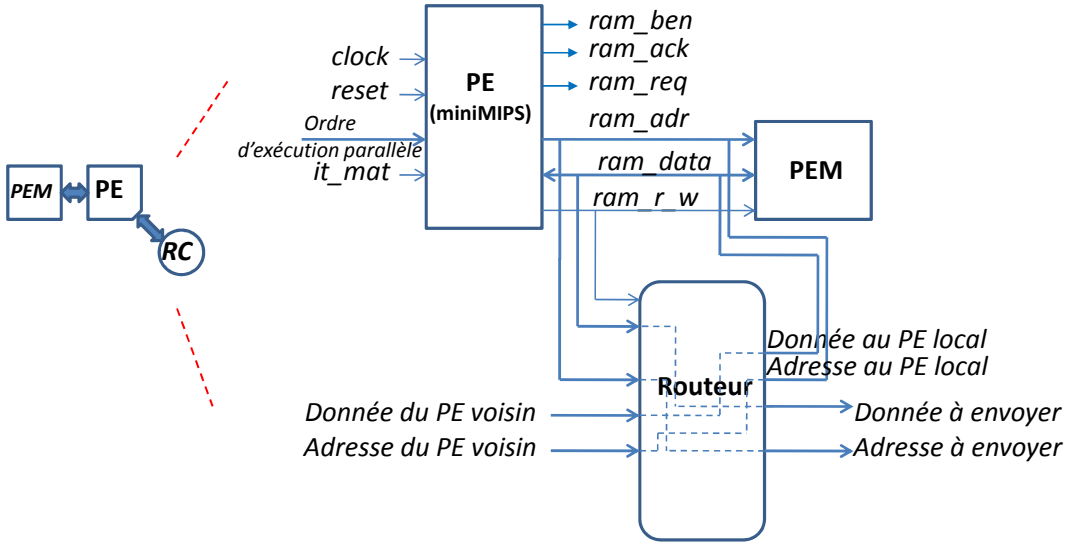



FIGURE 4.21 – *Architecture de l'unité élémentaire à base du processeur miniMIPS*

de sortie approprié. Il peut ensuite acheminer la donnée avec l'adresse mémoire spécifiée (où sera stockée la donnée dans le PE voisin) à la sortie correspondante. Dés que reçue par le routeur du PE receveur, la donnée sera directement stockée dans la mémoire à l'adresse spécifiée.

### 4.2.4 MpNoC

Le mpNoC est désigné comme un IP réseau global irrégulier adapté aux machines SIMD sur puce qui ne consomme pas beaucoup de ressources et qui est rapide. Un tel réseau est très utile et permet des communications parallèles entre les PEs ainsi qu'assurer des entrées/sorties parallèles. En effet, supposons qu'on veut effectuer l'addition de deux matrices A et B de taille NxN sur une structure de NxN processeurs organisés en 2D. Il suffit alors de distribuer les coefficients $A_{ij}$, $B_{ij}$ respectivement des deux matrices A et B, sur le processeur $PE_{ij}$ qui effectue ainsi aisément l'addition $A_{ij}$+$B_{ij}$. Tous les coefficients de la matrice résultat sont ainsi calculés en parallèle sur les $N^2$ processeurs. Toutefois, pour effectuer de telles opérations, il est nécessaire de distribuer les opérandes sur les PEs, de même qu'il est nécessaire de récolter le résultat distribué afin de l'exploiter. Ces architectures doivent donc proposer des mécanismes d'entrées/sorties rapides afin de ne pas dégrader les performances.

Intégrer le réseau mpNoC au sein d'une configuration mppSoC pose un compris coût/-performances. Ce choix est guidé par les besoins applicatifs. Ce réseau intègre deux composants : un gestionnaire de mode de communication et un IP réseau d'interconnexion (voir figure 4.22). Le premier est responsable d'établir le mode de communication requis. Le deuxième composant assure le transfert des données des sources vers les destinataires. Le mpNoC est conçu comme un IP capable d'assurer trois fonctions dans un système mppSoC [31] :

- utilisé comme un global routeur afin de connecter en parallèle n'importe quel PE avec un autre ;
- connecter les PEs aux périphériques de mppSoC ;

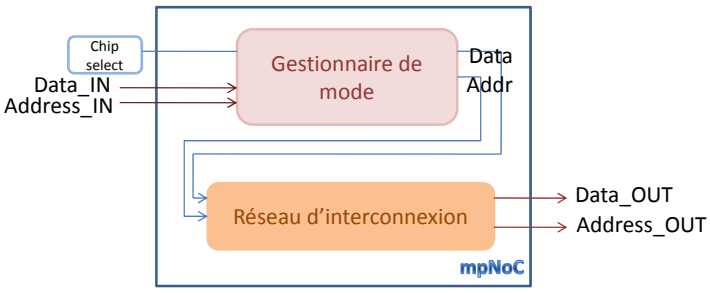



FIGURE 4.22 – *Architecture de mpNoC*

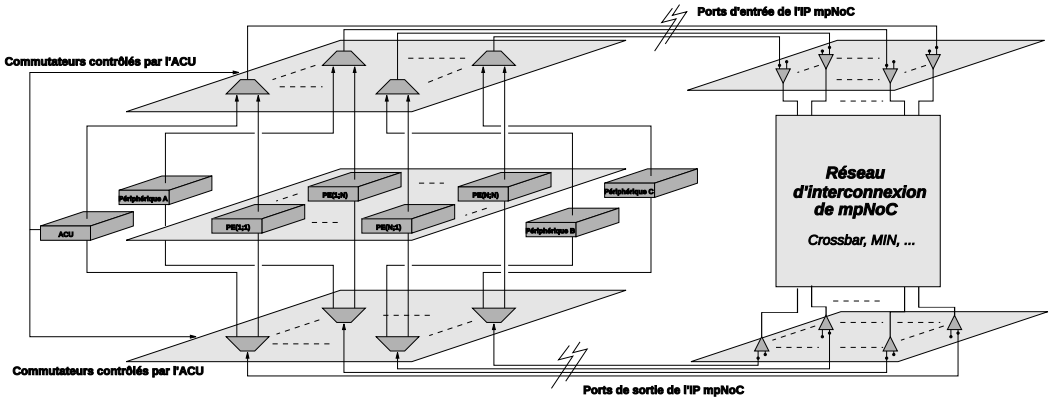



FIGURE 4.23 – *Intégration de mpNoC dans mppSoC*

– connecter l'ACU à n'importe quel PE.

Suite à ces fonctions, le mpNoC peut être configuré selon trois modes de communications bidirectionnels :

– PE - PE ;
– ACU - PE ;
– PE - périphérique E/S.

Ces cinq différents modes sont définis comme des constantes dans le fichier de configuration de mppSoC. Les entrées et sorties du réseau mpNoC sont en effet connectées à des commutateurs contrôlés par l'ACU, telle qu'illustrée dans la figure 4.23. Ces commutateurs permettent de connecter soit les PEs, soit l'ACU ou les périphériques au mpNoC, selon le mode de communication établit. En effet, le mpNoC a N entrées et N sorties, avec N est le nombre de PEs dans le système. Pour un mode PE-PE toutes les entrées/sorties sont activées. Dans le cas par exemple du mode ACU-PE, seulement la première entrée du réseau est activée et connectée à l'ACU (les autres entrées sont désactivées), alors que toutes les sorties sont connectées aux PEs.

Le fonctionnement du gestionnaire de mode est indépendant du réseau d'interconnexion. Par défaut, c'est le mode PE-PE qui est établit. La sélection du mode est assurée via une instruction de communication (détaillée dans la section 4.3). Le réseau d'interconnexion interne du mpNoC peut avoir différents types. Trois IPs sont fournis dans la bibliothèque mppSoC à savoir : un bus partagé, un crossbar et un réseau multi-étages. La flexibilité de ce réseau permet d'augmenter les performances du système. Le concepteur peut de même

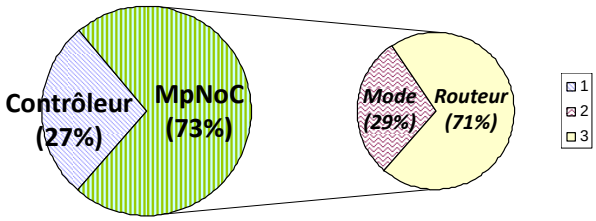



FIGURE 4.24 – *Distribution des ressources de mpNoC*

utiliser son propre IP après l'avoir adapté aux besoins de mppSoC. La topologie du réseau d'interconnexion dans mpNoC est configurable dépendant de plusieurs paramètres tels que la performance, la possibilité d'intégration, le nombre de PE, etc. De plus, mpNoC est conçu d'une manière générique avec une taille configurable (4x4, 32x32, par exemple) et peut directement être intégré dans toute configuration mppSoC puisqu'il est paramétrique en termes de nombre d'entrées et de sorties (égal au nombre de PEs) [6]. L'interface de mpNoC, incluant une connexion vers le port VGA (*Video Graphic Array*) par exemple, est détaillée ci dessous :

*clock : signal d'horloge*
*reset : signal de mise à zéro*
*cs : signal d'activation*
*- - - - Processeurs - - - -*
*- - - - Entrées - - - -*
*datainPE : données des PEs (vecteur de données 32-bits (taille du vecteur = nombre de PEs))*
*requestinPE : adresses des PEs (vecteur d'adresses 32-bits (taille du vecteur = nombre de PEs))*
*ram_wr_PE : lecture/écriture PE (vecteur de bits R/W (taille du vecteur = nombre de PEs))*
*datainACU : donnée ACU (32-bits)*
*requestinACU : adresse ACU (32-bits)*
*write_en : signal lecture/écriture ACU*
*- - - - Sorties - - - -*
*dataoutPE : données des PEs (vecteur de données 32-bits (taille du vecteur = nombre de PEs))*
*requestoutPE : adresses des PEs (vecteur d'adresses 32-bits (taille du vecteur = nombre de PEs))*
*dataoutACU : donnée ACU (32-bits)*
*requestoutACU : adresse ACU (32-bits)*
*- - - - Périphériques - - - -*
*dataoutVGA : donnée VGA (32-bits)*
*reqoutVGA : adresse VGA (32-bits)*

Le mpNoC présente toujours les signaux d'entrée et de sortie des processeurs (ACU et PEs). Le concepteur n'a qu'à ajouter les signaux vers les périphériques de mppSoC en cas de besoin. Une telle interface peut être facilement utilisée dans différentes configurations mppSoC.

Le mpNoC mets un temps non connu à l'avance (dépendant de la nature du réseau d'interconnexion) pour acheminer les données de communication ; d'autant plus lorsqu'il s'agit de multiple envois ou de multiples réceptions. Pour cela, il faut assurer la synchronisation des PEs dans le système et la réalisation de toutes les communications désirées. Afin de ne pas gérer cette complexité au niveau logiciel, nous avons implémenté un module matériel permettant de contrôler les communications via le mpNoC. Ce contrôleur vérifie si les données transférées par le réseau sont acheminées vers les bons destinataires. Dés qu'une instruction de communication à travers le réseau est exécutée par les PEs, l'ACU arrête d'envoyer les ordres d'exécution jusqu'à un acquittement envoyé par le module de contrôle. En effet, l'IP de contrôle effectue un traitement permettant de s'assurer que chaque destinataire

a bien reçu la donnée transférée par la source correspondante. Il envoie alors une notification à l'ACU pour continuer à exécuter les instructions. Plus précisément, c'est l'ACU qui décode les instructions et pilote les PEs et le mpNoC. Lorsque l'ACU traite une nouvelle requête sur le mpNoC, il commence par diffuser les ordres d'exécution aux PEs. Au cycle suivant, les PEs envoient leurs données (donnée à envoyer et adresse du destinataire). Au cycle d'après, le mpNoC lit les données des PEs puis effectue la requête dont l'exécution prend plusieurs cycles. À la fin de la transmission, le mpNoC indique à l'ACU et aux PEs que la requête est terminée à travers le contrôleur. La figure 4.24 compare entre la taille du contrôleur et les autres composants de mpNoC. Le contrôleur occupe 27% de la surface totale dédiée à l'mpNoC. Nous voyons de même que le réseau d'interconnexion consomme le plus de ressources, avec un pourcentage de 71%, au sein du mpNoC.

Les sous sections suivantes détaillent les implémentations des différents réseaux d'interconnexion de mpNoC existants dans la bibliothèque mppSoCLib.

#### 4.2.4.1 Bus partagé

Le bus permet une technique de communication très simple entre les composants et reste encore très largement utilisé dans beaucoup de systèmes sur puce. Il peut être vu comme étant fonctionnellement équivalent à un multiplexeur qui sélectionne, par une fonction d'arbitrage, une source unique qui se voit attribuer le droit de communiquer avec un destinataire défini par son adresse. Il est clair que l'atomicité des accès induit de ce fait une sérialisation des requêtes concurrentes pouvant être émises par différentes sources. La sérialisation dans le temps des accès se traduit par des latences moyennes d'autant plus élevées que le nombre de sources augmente. Ce problème est connu dans la littérature sous l'appellation de scalabilité limitée. En contre partie, le bus possède un avantage important de simplicité avec un canal de communication unique et un algorithme de diffusion simple à mettre en œuvre. Cependant une architecture bâtie autour d'un bus partagé est peu extensible. De plus, avec un seul canal de communication le débit disponible entre les unités sur le bus est limité par leur nombre. Le réseau implémenté présente l'architecture illustrée dans la figure 4.25. Dans ce cas, chacun des PEs envoie sa requête et la donnée à communiquer qui seront stockées dans une file d'attente de type FIFO. Un arbitre gère les priorités (selon un arbitrage Round Robin) entre les différentes requêtes et un décodeur décode l'adresse envoyée afin de transférer la requête au destinataire approprié. Les performances du bus partagé en termes de surface, débit et latence se résument dans le tableau 4.9. La latence correspond aux délais (généralement exprimés en nombre de cycles) nécessaires à l'acheminement d'un message depuis un nœud émetteur jusqu'au nœud destinataire. Dans le tableau 4.9 les valeurs de latence sont bornées entre la valeur minimale et celle maximale. Le débit de données reflète les capacités d'écoulement de trafic du réseau en terme de quantité de données par unité de temps. Il est exprimé en Mega Bytes par seconde.

D'un côté, nous voyons clairement que le bus est peu extensible et ne supporte pas des débits élevés. Sa latence augmente considérablement avec l'ajout du nombre de sources. De l'autre côté, il se caractérise par sa petite taille.

#### 4.2.4.2 Crossbar

Vu que le bus a une bande passante limitée et atteint une saturation dés lors qu'on intègre une dizaine de processeurs, nous avons alors pensé au réseau complètement connecté à

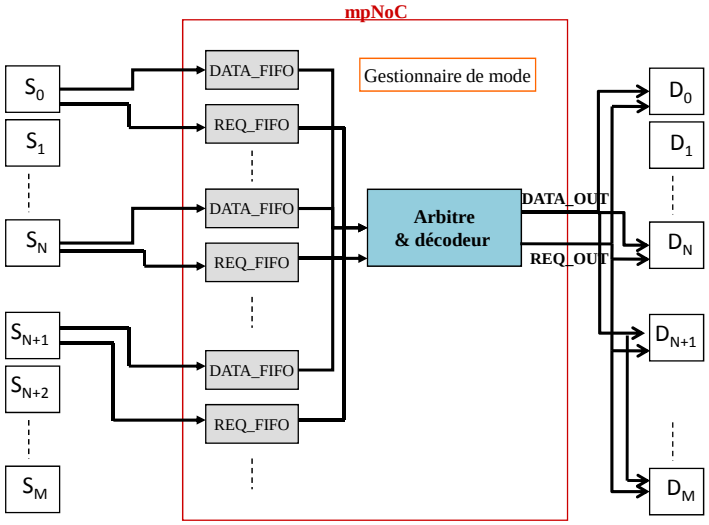



FIGURE 4.25 – *Architecture du bus partagé*

TABLE 4.9 – Performances du bus partagé

| Nombre | Ressources logiques | | Latence | Débit |
|---|---|---|---|---|
| PEs | ALUTs | registres | (cycles) | (MB/s) |
| 4 | 183 | 442 | 3-12 | 347.16 |
| 8 | 407 | 739 | 3-24 | 303.42 |
| 16 | 726 | 1319 | 3-48 | 229.64 |
| 32 | 1496 | 2467 | 3-96 | 185.91 |
| 64 | 2981 | 4781 | 3-192 | 120.25 |
| 128 | 4440 | 6888 | 3-384 | 63.5 |

savoir le crossbar. Son architecture est présentée dans la figure 4.26. Elle est basée sur deux principaux blocs : N routeurs d'entrée (*Router_IN*) et N routeurs de sortie (*Router_OUT*), avec N est le nombre de PEs dans le système. Le routeur d'entrée joue le rôle d'un commutateur qui va acheminer la requête et la donnée (chacune de taille 32 bits) de ses deux ports d'entrée vers le routeur de sortie approprié selon le décodage de l'adresse. Dés que le routeur de sortie détecte une donnée sur son port d'entrée, il l'emmagasine dans une FIFO (de taille 2) et l'envoie après vers le destinataire demandé. Ce routeur contient une unité de routage et d'arbitrage (par priorité) qui assure la fonction d'aiguillage et gère les situations de conflits. Si une FIFO qui a la priorité n'a pas encore eu de donnée à transmettre alors sa voisine (dans l'algorithme round robin) qui présente une donnée va l'envoyer au destinataire. Toutes ces transactions sont synchrones au signal d'horloge global du système [6]. Les performances du crossbar en termes de surface, débit et latence se résument dans le tableau 4.10.

Nous voyons clairement l'augmentation exponentielle de la surface du crossbar en augmentant le nombre de sources. En effet, la complexité est de $N^2$ avec N est le nombre de commutateurs dans le réseau. Comparé à un bus, le crossbar offre une meilleure bande passante et donc une meilleure latence du fait que les chemins sources-destinataires sont séparés les uns des autres. Néanmoins le nombre de connexions physiques nécessaires à l'implémentation d'un crossbar complet limite son utilisation.

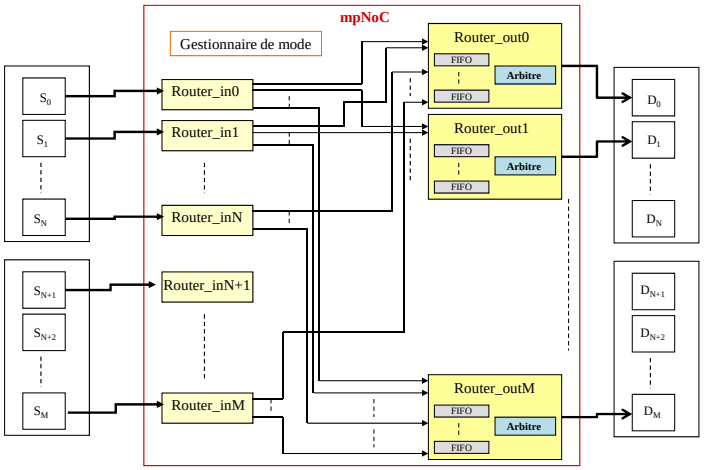



FIGURE 4.26 – *Architecture du crossbar*

TABLE 4.10 – Performances du crossbar

| Number | Ressources logiques | | Latence | Débit |
|---|---|---|---|---|
| PEs | ALUTs | registres | (cycles) | (MB/s) |
| 4PEs | 281 | 483 | 2-5 | 527.16 |
| 8PEs | 548 | 806 | 2-9 | 463.84 |
| 16PEs | 957 | 1469 | 2-17 | 420.56 |
| 32PEs | 1905 | 2818 | 2-33 | 360.64 |
| 64PEs | 4191 | 5917 | 2-65 | 340.22 |
| 128PEs | 10184 | 13609 | 2-129 | 303.76 |

#### 4.2.4.3 Réseau multi-étages de type Delta

Les réseaux Delta sont des réseaux d'interconnexion multi-étages de type Banyan qui, d'après la définition formelle de Patel [83], comprennent axb crossbars, dont tous les ports d'entrées et de sorties sont connectés. Le réseau delta est caractérisé par un accès total. Ainsi, par une configuration correcte des commutateurs à chaque étage, n'importe quelle entrée doit être capable d'atteindre n'importe quelle sortie. Le réseau delta se caractérise aussi par sa capacité de routage automatique des messages depuis la source vers la destination. En ce sens le canal de sortie choisi à chaque commutateur ne dépend pas de la source mais seulement de la destination. Une autre propriété pour les réseaux delta est l'équivalence topologique : il suffit de réordonner les positions des commutateurs sans rompre les connexions pour passer d'un réseau à un autre. En effet, il existe plusieurs formes (types) de réseaux Delta, dépendamment de leurs connexions. Dans le cadre des travaux de mastère des étudiants Bilel Neji (soutenu en Juin 2008) et Ramzi Tligue (soutenu en Juin 2009), trois réseaux Delta à savoir : Oméga, butterfly et baseline ont été implémentés. Ces réseaux ont été utilisés et adaptés au système mppSoC. La figure 4.27 présente les trois types de réseaux de taille 8x8 chacun.

Un réseau Delta général $a^n x b^n$ se compose de $a^n$ sources et $b^n$ destinations, avec n est le nombre d'étages. Le $i^{eme}$ étage a $a^{n-1} x b^{n-1}$ crossbars ou éléments de routage (*Switching Element*) de taille axb (dans notre cas des crossbars de taille 2x2). L'élément de routage est le

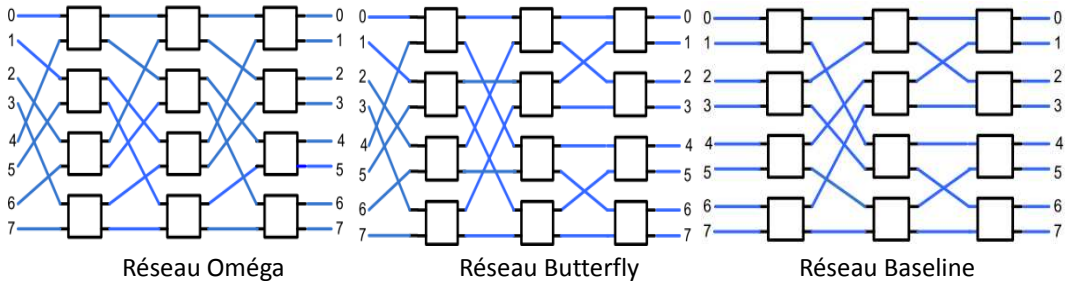



FIGURE 4.27 – *Types de réseaux Delta (8,8)*

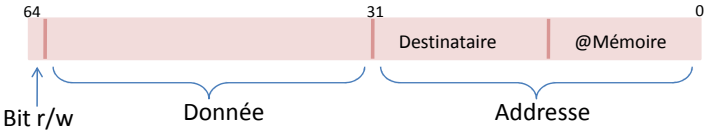



FIGURE 4.28 – *Paquet de donnée dans le réseau Delta MIN*

TABLE 4.11 – Performances du réseau Delta MIN

| Number | Ressources logiques | | Latence | Débit |
|---|---|---|---|---|
| PEs | ALUTs | registres | (cycles) | (MB/s) |
| 4 | 402 | 1404 | 6-9 | 504.08 |
| 8 | 1101 | 3885 | 9-16 | 477.28 |
| 16 | 1456 | 5401 | 12-27 | 474.12 |
| 32 | 2099 | 7746 | 15-46 | 354.44 |
| 64 | 2905 | 9022 | 18-81 | 324.96 |
| 128 | 3834 | 10397 | 21-148 | 290.14 |

composant clé d'un tel réseau. Il est composé de deux FIFO et un ordonnanceur qui décide de transférer les données aux destinations désirées en se basant sur un algorithme d'arbitrage round robin [105]. Le réseau implémenté est un réseau de transfert de données par paquets. Un paquet (de longueur 65 bits) se compose de trois parties telle que montrée dans la figure 4.28 :

- entête (1 bit) : contient le bit de lecture/écriture pour indiquer la nature de l'accès mémoire ;
- donnée (32 bits) ;
- queue (32 bits) : composée de la destination et de l'adresse mémoire.

Les performances du réseau Delta en termes de surface, latence et débit se résument dans le tableau 4.11. Comparé au crossbar, nous constatons que le réseau Delta a une latence plus importante. Cependant, la surface du MIN est plus réduite. Elle est proportionnelle à N$log_2$N, comparé à $N^2$ pour le crossbar. Le débit d'un réseau MIN est comparable à celui du crossbar pour de petites configurations. Cependant, il est inférieur pour de grandes configurations.

Le mode de communication ainsi que les transferts de données via le mpNoC sont gérés par des instructions de communications qui seront présentées dans la section 4.3.

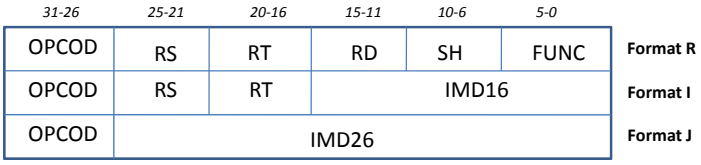



FIGURE 4.29 – *Formats d'instructions MIPS32*

### 4.2.5 Classification des IPs

Dans la bibliothèque mppSoC, nous distinguons deux types d'IPs : des IPs composants de mppSoC et des IPs d'assemblage. Le premier type présente les IPs de base constituant une configuration mppSoC tels que les processeurs, mémoires ou réseaux pour les quels le concepteur peut intégrer son propre IP. Ce type d'IPs est fourni afin de simplifier la tâche de conception et accélérer la construction d'une configuration mppSoC. Le deuxième type inclue les IPs nécessaires à être intégrés afin d'assurer le bon fonctionnement de mppSoC à savoir : le gestionnaire de mode, le contrôleur de mpNoC et les IPs de réplication (registres d'activité et d'identité et module de OU logique).

Les aspects relatifs à la programmation et au portage d'applications sur mppSoC sont présentés dans la section suivante.

## 4.3 Programmation de mppSoC

Dans le cadre de ces travaux, nous avons défini un langage data parallèle pour mppSoC. MppSoC est basé sur un modèle d'exécution SIMD. Dans le modèle de programmation à parallélisme de données qui lui est sous-jacent, le contrôle est séquentiel alors que l'accès aux données est parallèle.

Dans notre travail, divers langages de programmation ont été utilisés selon le processeur : assembleur MIPS, assembleur OpenRisc et C. Dans la programmation, nous distinguons de même entre une méthodologie de réduction et une méthodologie de réplication. Ceci est dû au fait que nous avons proposé une modification du jeu d'instructions dans le cas de réduction de miniMIPS. Nous détaillons alors en premier lieu ces différences de programmation. Nous définissons dans un deuxième lieu un langage de programmation de mppSoC dont l'implémentation varie selon le processeur choisi.

### 4.3.1 Cas de réduction de processeur

Dans la bibliothèque d'IPs mppSoC, nous trouvons deux processeurs réduits à savoir miniMIPS et OpenRisc. Leur programmation est définie dans les paragraphes suivants.

#### 4.3.1.1 Processeur miniMIPS

Tel que décrit dans la sous section 4.2.1.2, l'assembleur MIPSI a été étendu afin de supporter les instructions parallèles. Toutes les instructions sont compatibles avec MIPS32 ayant une longueur fixe de 32 bits. Classées en trois formats (R, I et J) présentés dans la figure 4.29, chacune dispose d'un opcode codé sur les 6 bits de poids fort. Dans la table de décodage des instructions miniMIPS, nous avons alors pu ajouter d'autres opcodes et donc d'autres

TABLE 4.12 – Décodage de OPCOD

| | | INS 28 :26 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| INS 31 :29 | 000 | *SPECIAL* | *BCOND* | J | JAL | BEQ | BNE | BLEZ | BGTZ |
| | 001 | ADDI | ADDIU | SLTI | SLTIU | ANDI | ORI | XORI | LUI |
| | 010 | *COPRO* | | ***SPECIALP*** | | | | | |
| | 011 | **P_ADDI** | **P_ADDIU** | **P_SLTI** | **P_SLTIU** | | | | |
| | 100 | LB | LH | | LW | LBU | LHU | | ***P_MEM*** |
| | 101 | SB | SH | | SW | **P_ANDI** | **P_ORI** | | |
| | 110 | | | | | **P_XORI** | | **S_LBP** | **S_LWP** |
| | 111 | | | | | **P_LUI** | | **S_SBP** | **S_SWP** |

instructions parallèles. Le tableau 4.12 montre le décodage des différents opcodes et l'ajout d'instructions parallèles (préfixées par P_). Il y'a d'autres opcodes d'indirection pour lesquels d'autres bits sont à analyser. Ils apparaissent en italique dans le tableau 4.12. Ces différents cas sont analysés en annexe C.

Plusieurs instructions du jeu MIPSI ont été dupliquées après avoir été préfixées par P_. Lorsque l'ACU décode une de ces instructions, il demande à tous les PEs de l'exécuter de manière parallèle et synchrone. Le choix des instructions a été réalisé en s'inspirant de la machine MasPar. Les instructions arithmétiques et logiques ou d'accès aux mémoires ont été dupliquées. Les instructions de branchement ou d'appel système ont été supprimées, car les PEs n'ont pas de code en mémoire. Seul l'ACU conserve ces instructions, c'est lui qui gère les sauts. D'autres instructions ont été ajoutées ; elles sont décrites en annexe C. Elles sont toutes préfixées soit par S_ soit par P_ :

- S_ : "Séquentiel", pour les instructions exécutées uniquement par l'ACU ;
- P_ : "Parallèle", pour les instructions exécutées de manière parallèle et synchrone par tous les PEs.

Les registres dédiés du processeur sont inaccessibles par les instructions classiques (qui utilisent les 32 registres principaux). Le programmeur peut toutefois consulter certains d'entre eux, voire les modifier, via les instructions mfc0 et mtc0 (respectivement). Pour accéder à la valeur du Or Tree (bit valant 1 ssi après la dernière instruction envoyée aux PEs, un au moins d'entre eux est encore actif), l'ACU doit utiliser mfc0. Le numéro de registre dédié correspondant au Or Tree est 22. Du côté des PEs, p_mfc0 et p_mtc0 permettent les mêmes opérations. Pour les PEs, le registre 22 présente le registre d'identité (numéro du PE). Afin d'ajouter le bit d'activité pour les PEs, nous avons choisi d'utiliser le $16^{eme}$ bit du registre statut (registre 12) comme bit d'activité.

### Techniques de programmation

Avec le processeur miniMIPS, les programmes sont codés en assembleur étendu. Vu que nous avons besoin d'assembler de nouvelles instructions, deux outils *generation* et *fusion* ont été développés (en C). Le script-shell *mppsocassembly* permet alors de générer un binaire exécutable à partir d'un fichier source assembleur étendu, en utilisant les outils développés ainsi que *sde-as* [12] [7]. *Generation* est le premier outil à utiliser pour la génération d'un

12. assembleur MIPS

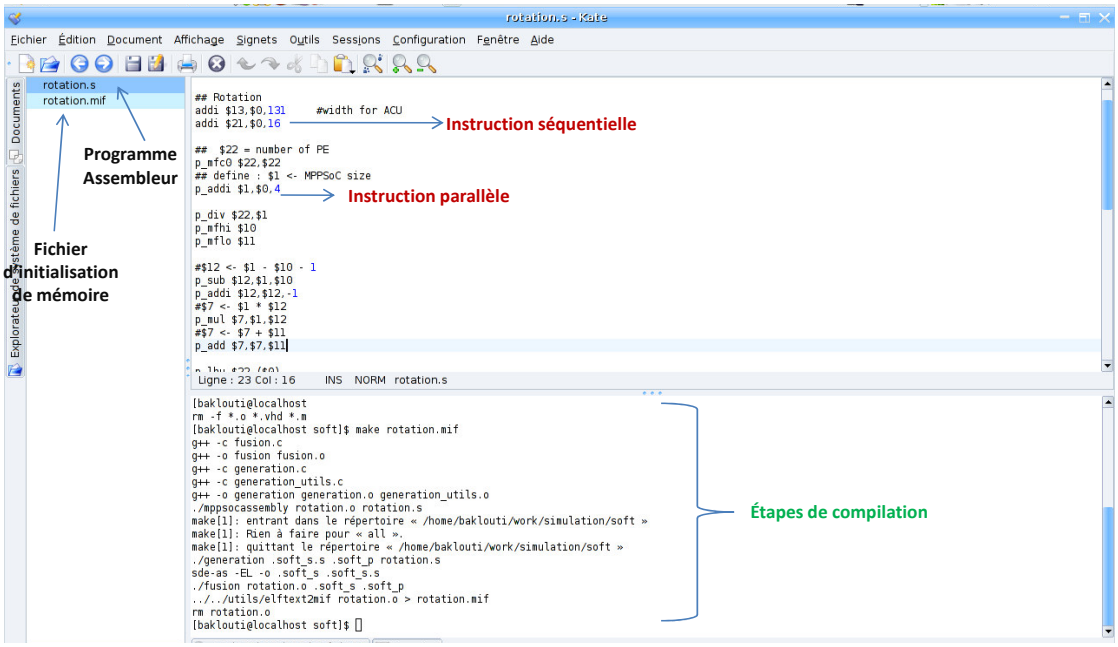



FIGURE 4.30 – *Programmation d'un code parallèle en assembleur miniMIPS étendu*

fichier exécutable. Il permet, à partir d'un fichier source assembleur étendu, d'en séparer le code MIPSI du code étendu. *Fusion* est le deuxième outil utilisé qui permet d'obtenir le fichier binaire exécutable final. Finalement, *mppsocassembly* est un simple script-shell permettant directement de générer un exécutable étendu à partir d'un fichier source en assembleur étendu.

Un script *elftext2mif* a aussi été développé afin de transformer le binaire obtenu en un format (*MIF : Memory Initialisation File*) compréhensible par les outils de compilation et synthèse. Le fichier final sert alors pour l'initialisation de la mémoire d'instructions de l'ACU. Ces différents outils sont encore mieux détaillés en annexe D. La figure 4.30 montre la génération du binaire d'un code assembleur MIPS étendu.

Le code suivant en assembleur MIPS étendu montre un exemple d'une multiplication parallèle (sur 4 PEs) de matrice de taille 8x8 par une constante fournie par l'ACU :

```
.globl main
.text
main :
ori $1, $0, 0x0
lw $8,0($1) #constante c
ori $2, $0, 0x9003
addi $4,$0,1 #mode ACU-PE
sw $4,0($2)
sw $8,0($1) #envoie de la constante à tous les PEs
addi $2,$0,0
addi $3,$0,16
p_ori $1, $0, 0x0
p_lw $3,0($1) #lecture de la constante par tous les PEs
loop :
p_ori $1, $0, 0x4
p_lw $7,0($1) #lecture des éléments de matrice
p_mult $7,$3 #multiplication parallèle
p_mflo $7
p_sw $7,0($1) #stockage des résultats dans les mémoires locales
p_addi $1,$1,4
addi $2,$2,1
subu $5,$3,$2
bgez $5,loop #rebouclage
end :nop
j end
```

#### 4.3.1.2 Processeur OpenRisc

Le processeur OpenRisc peut être programmé par un langage assembleur ou un langage C. Afin d'accélérer et faciliter l'écriture d'un algorithme parallèle, nous avons choisi de programmer en C. Par ailleurs, nous utilisons des instructions assembleur imbriquées pour définir certaines tâches telles que les instructions de début de code séquentiel ou parallèle, lecture/écriture dans une adresse mémoire spécifique, etc.

Afin de compiler des programmes pour l'architecture OR32, les outils GNU doivent être installés dans un environnement linux. Nous retrouvons : la collection de compilateurs GNU (GCC), les outils binaires GNU (*binutils*), le simulateur OpenRisc (*OR1Ksim : OpenRisc Architectural Simulator*), etc. Ces outils sont disponibles en libre téléchargement sur le site OpenCores [82] :

- Or1K_binutils : est une collection d'outils de développement de logiciels contenant un éditeur de liens, un assembleur et d'autres outils pour travailler avec des fichiers objet et archives.
- Or1K_GCC : est le compilateur permettant de compiler des codes C pour l'OR32.
- Or1K_gdb : nous permet de débuguer un programme en cours d'exécution (en le déroulant instruction par instruction ou en examinant et modifiant ses données). Il permet donc de traquer les bugs/erreurs qui se trouvent dans tout programme.
- Or1Ksim : est un logiciel libre et configurable conçu spécialement pour la simulation des systèmes à base de processeur OpenRisc. Il est conçu à base d'ISS permettant de modéliser une architecture et de simuler le fonctionnement de cette dernière.

Après l'installation des outils, un code C peut être cross-compilé. Pour cela, il faut regrouper dans un dossier les fichiers suivants :

- Makefile (voir annexe B) : un script qui décrit toutes les commandes de génération du fichier de test.
- App.c : contient le programme C.
- Boot.s (voir annexe B) : contient le programme permettant une initialisation des re-

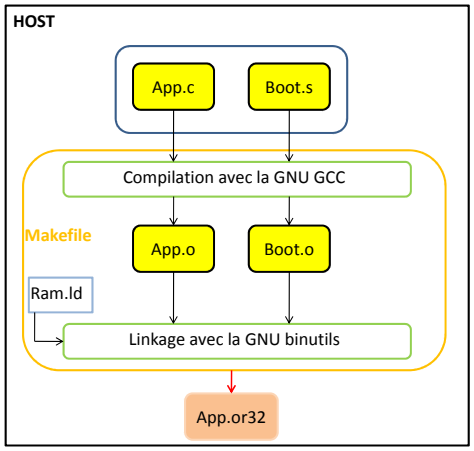



FIGURE 4.31 – *Génération du fichier de test "App.or32"*

gistres.

– Ram.ld (voir annexe B) : contient le programme qui décrit la taille et l'organisation de la mémoire utilisée par le fichier de test ; il sera utilisé par l'éditeur de lien pour classer les adresses du fichier test dans des vecteurs.

La figure 4.31 illustre les différentes étapes d'exécution du fichier Makefile jusqu'à la génération du fichier "App.or32". Le fichier or32 généré peut être transformé en un fichier hex afin de servir comme fichier d'initialisation pour la mémoire d'instructions. Il suffit de taper les commandes suivantes :

– Conversion binaire : Or32-elf-objcopy –O binary App.or32 App.bin
– Conversion hex : bin2vmem App.bin > sram.vmem

Avec OpenRisc, les applications sont codées en C tout en utilisant des instructions assembleur (avec la commande **asm**). Le code C suivant montre un exemple d'une multiplication parallèle (sur 4 PEs) de matrice de taille 8x8 par une constante fournie par l'ACU :

```
int main() {
int c ;
int i,k ;
int *p = 0x0 ;
int y [4][4] ;
int R[4] ;
asm ("l.addi %%r7,%%r7,0 ;" : : : "%r7") ; //code séquentiel
asm ( "l.lwz %0, 0x0(%%r0) ;" : "=r"(c) : :"%r0") ; //lecture de la constante
asm ("l.add %%r7,%%r7,r0 ;" : : : "%r7", "%r0") ; //code parallèle
for (i=0 ;i<4 ;i++) {
for (k=0 ;k<4 ;k++) {
* (R + i)= *(p+ k + 4*i) ;
y[i][k] = c * R[i] ;
asm ("l.sw\t0x00(r8) ,%0" : : "r" (y[i][k])) ; //stockage des résultats dans les mémoires locales
asm ("l.addi %%r8,%%r8,4 ;" : : : "%r8") ;
 } }
return 0 ; }
```

### 4.3.2 Cas de réplication de processeur

Dans la méthodologie de réplication, si nous choisissons le processeur OpenRisc nous suivons la même méthode de programmation que celle décrite dans la sous section précé-

dente. Par contre si nous choisissons le processeur miniMIPS, nous programmons alors avec l'assembleur MIPSI standard en ajoutant les instruction spécifiques dans le code afin de délimiter les instructions dédiées à l'ACU de celles dédiées aux PEs.

En utilisant le processeur NIOS II, la programmation est comparable à celle utilisée avec le processeur OpenRisc. La seule petite différence est dans la manière d'écriture des instructions asm. Le code C suivant montre l'exemple d'une multiplication parallèle (sur 4 PEs) de matrice de taille 8x8 par une constante fournie par l'ACU, exécutée sur une configuration mppSoC à base de NIOS :

```
#include <stdio.h>
#include "system.h"
#include "nios2.h"
int main() {
int c,i,k ;
int *p = 0x0 ;
int y [4][4], R[4] ;
asm ("addi r7,r7,0 ;" : : : "r7") ; //code séquentiel
asm ( "ldw \%0, 0x0(r0) ;" : "=r"(c) : :"r0") ; //lecture de la constante
asm ("add r7,r7,r0 ;" : : : "r7","r0") ; //code parallèle
for(i = 0 ;i< 4 ; i++) {
for(k = 0 ; k < 4 ; k++) {
* (R + i)= *(p+ k + 4*i) ;
 y[i][k] = c * R[i] ;
asm ( "stw \%0, 0x0(r8) ;" : "=r"(y[i][k]) : : "r8") ; //stockage des résultats dans les mémoires locales
asm ("addi r8,r8,4 ;" : : : "r8") ; } }
return 0 ; }
```

### 4.3.3 Jeu d'instructions mppSoC

Le jeu d'instructions mppSoC est dérivé du jeu d'instructions du processeur utilisé. Pour simplifier l'écriture d'un programme data parallèle pour mppSoC, nous avons défini quelques macros facilitant l'utilisation des instructions spécifiques. Ce langage spécialisé de haut niveau constitue une alternative avantageuse au micro-codage et rend plus facile la programmation bas niveau de mppSoC. L'approche haut niveau permet de simplifier de nombreuses étapes lors de la programmation, de réduire les risques d'erreur et les temps de développement. Abstraire les détails architecturaux du système sous-jacent permet de faciliter la programmation des algorithmes. Parmi les instructions spécifiques, nous notons celles de communication et de contrôle. Ces instructions sont alors codées en se basant sur les instructions mémoires à savoir *Load : LW* et *Store :SW*. Les adresses utilisées avec ces instructions sont aussi bien définies.

Notre ultime objectif est d'utiliser les outils et compilateurs existants et ne pas développer d'autres plus spécifiques ou complexes. Le tableau 4.13 donne la liste de ces macros et leur décodage selon le jeu d'instructions du processeur utilisé. Le paragraphe suivant détaille le décodage des instructions de communication utilisées pour programmer les deux réseaux de mppSoC. Ces instructions sont dérivées des instructions du processeur utilisé qui sont les instructions d'accès mémoire : *LW* et *SW*. Nous distinguons les instructions dédiées au réseau mpNoC de celles dédiées au réseau de voisinage.

#### Instructions pour le réseau mpNoC

Afin d'établir le mode de communication requis pour mpNoC, nous utilisons une instruction particulière. Cette dernière consiste à écrire la valeur correspondante au mode dans

une adresse spécifique. Elle se base alors sur l'instruction *SW* :

$$SWcst, @ModeManager \tag{4.1}$$

Avec :
- @ModeManager est une adresse spécifique utilisée pour activer le mpNoC ;
- cst est la constante du mode qui peut avoir une valeur parmi cinq possibles.

Les modes de communication mpNoC sont définis dans le fichier de configuration VHDL de mppSoC comme suit :

```
–MPNoC Modes
constant Mode0 : positive := 0 ; – PE − > PE
constant Mode1 : positive := 1 ; – ACU − > PE
constant Mode2 : positive := 2 ; – PE − > ACU
constant Mode3 : positive := 3 ; – PE − > Device
constant Mode4 : positive := 4 ; – Device − > PE
```

Après le choix du mode de communication, les transferts via le mpNoC seront assurés via les instructions SEND et RECEIVE.

► **Instruction SEND** permet d'envoyer une donnée à travers le réseau d'une source vers un destinataire, en se basant sur l'instruction mémoire *SW : SW data,address*. Le champ "data" est la donnée à envoyer sur 32 bits et le champ "address" correspond à l'adresse transmise sur 32 bits. L'adresse peut avoir divers formats selon le mode de communication du mpNoC :
- Modes PE-PE, ACU-PE et périphérique-PE : l'identité du PE destinataire (de longueur 32-SL_add_width bits) + l'adresse mémoire où stocker la donnée (SL_add_width bits). Dans le cas d'un transfert de données vers tous les PEs, il suffit de spécifier seulement l'adresse mémoire ;
- Mode PE-ACU : l'adresse mémoire de l'ACU (MS_add_width bits) ;
- Mode PE-périphérique : l'adresse du périphérique (32 bits) ;

MS_add_width et SL_add_width sont les tailles paramétriques des adresses mémoires de l'ACU et du PE respectivement. Les modes de communication ACU-périphérique ou périphérique-ACU sont établis par des connexions point à point gérées au niveau matériel.

► **Instruction RECEIVE** permet de recevoir la donnée en se basant sur l'instruction mémoire *LW : LW data,address*. Elle prend le même champ d'adresse que l'instruction SEND.

À travers ces différentes instructions, nous remarquons que le temps d'exécution d'une instruction varie selon le mode de communication. Par exemple dans le cas d'une communication de multiple sources vers un seul destinataire, la donnée est envoyée une à une séquentiellement vu que le destinataire ne peut recevoir qu'une donnée à la fois. Le temps d'exécution vaut alors le temps d'une seule communication multiplié par le nombre de sources (le même cas se trouve dans les machines SIMD traditionnelles telle que la MasPar).

#### Instructions pour le réseau de voisinage

Le réseau de voisinage est aussi géré par des instructions SEND et RECEIVE.

► **Instruction SEND** permet d'envoyer la donnée d'un PE à un autre selon une direction précise. En effet, elle se base comme le mpNoC sur l'instruction mémoire SW. Son adresse se compose de trois champs :

- la distance (29-SL_add_width bits) qui définit le nombre de liens entre le PE source et le PE destinataire ;
- la direction (3 bits) qui peut avoir une valeur parmi huit possibles selon la topologie du réseau : Nord(000), Est(010), Sud(001), Ouest(011), Nord Est(100), Nord Ouest(101), Sud Est(110) et Sud Ouest(111) ;
- l'adresse mémoire (SL_add_width bits) où stocker la donnée.

► **Instruction RECEIVE** permet de recevoir la donnée en se basant sur l'instruction mémoire *LW : LW data,address*. Elle prend le même champ d'adresse que l'instruction SEND décrite précédemment.

Ces différentes instructions de communication décrites servent alors à piloter les réseaux de communication. Elles sont aussi applicables quelque soit l'IP processeur utilisé puisqu'elles se basent sur les instructions standard d'accès mémoire.

## 4.4 Conclusion

Dans ce chapitre, nous avons détaillé la conception des IPs présents dans la bibliothèque d'IPs mppSoC. Cette bibliothèque offre des processeurs, mémoires et réseaux d'interconnexion afin de constituer la configuration mppSoC désirée. Une configuration donnée d'un système mppSoC décrit les composants devant être utilisés et les paramètres de dimensionnement du système (nombre de processeurs, taille mémoire associée à chaque processeur, etc.). L'utilisateur aura le choix d'intégrer les IPs prédéfinis et nécessaires pour son architecture, piloté par les exigences de son application. Nous avons alors vu qu'en partant du modèle mppSoC générique, différentes configurations SIMD peuvent être construites et adaptées selon les besoins. La configuration implémentée peut être soit simulé par les outils de simulation ou prototypée sur un circuit FPGA. Il est aussi intéressant de monter dans le niveau d'abstraction de la conception afin d'encore faciliter la tâche au concepteur. Pour cette fin, un outil capable de générer automatiquement une configuration choisie de mppSoC spécifiée à un haut niveau d'abstraction sans se préoccuper des détails d'implémentation s'avère utile. Cet outil sera présenté dans le chapitre suivant. Nous exposerons de même les résultats expérimentaux et discutons les performances du système mppSoC.

TABLE 4.13 – Macros mppSoC

| Macro ASM | Description | Codage | | |
|---|---|---|---|---|
| | | miniMIPS | OpenRisc | NIOS |
| **SET_MODE_MPNOC (reg)** | Sélectionner le mode de communication de mpNoC parmi 5 modes (listés dans la sous section 4.3.3). Le registre reg contient la valeur du mode. | SW reg,0x9003(r0) | l.addi r1,r0,0x9003<br>l.sw 0x0(r1),reg | IOWR (WRP_ACU_B, 0x800, data)<br>Avec : data est la valeur du mode. |
| **Modes :0,1 et 4 : P_/MPNOC_SEND (reg,dest,adr)** | Envoi via mpNoC : transférer la donnée (qui se trouve dans reg) à la destination requise. | p_/addi r1,r0,dest<br>p_/addi r1,r1,adr<br>p_/SW reg,0(r1) | l.addi r1,r0,dest<br>l.addi r1,r1,adr<br>l.sw 0x0(r1),reg | IOWR (WRP_B, addr, data)<br>Avec : addr(11)='1' et addr(10 :0)contient dest. |
| **Modes :2 et 3 : P_/MPNOC_SEND (reg,adr)** | | p_/addi r1,r0,adr<br>p_/SW reg,0(r1) | l.addi r1,r0,adr<br>l.sw 0x0(r1),reg | Avec : addr(11)='1' et addr(10 :0) contient adr. |
| **Modes :0,1 et 4 : P_/MPNOC_REC (reg,src,adr)** | Réception via mpNoC : recevoir la donnée (la stocker dans reg) de la source. | p_/addi r1,r0,src<br>p_/addi r1,r1,adr<br>p_/LW reg,0(r1) | l.addi r1,r0,src<br>l.addi r1,r1,adr<br>l.lwz reg,0x0(r1) | data=IORD (WRP_B, addr)<br>Avec : addr(11)='1' et addr(10 :0) contient src. |
| **Modes :2 et 3 : P_/MPNOC_REC (reg,adr)** | | p_/addi r1,r0,adr<br>p_/LW reg,0(r1) | l.addi r1,r0,adr<br>l.lwz reg,0x0(r1) | Avec : addr(11)='1' et addr(10 :0) contient adr. |
| **P_REG_SEND (reg,dir,dis,adr)** | Envoi via le réseau de voisinage : transférer la donnée (stockée dans reg) de la source à la destination. | p_addi r1,r0,dir<br>p_addi r1,r1,dis<br>p_addi r1,r1,adr<br>p_SW reg,0(r1) | l.addi r1,r0,dir<br>l.addi r1,r1,dis<br>l.addi r1,r1,adr<br>l.sw 0x0(r1),reg | IOWR (WRP_B, addr, data)<br>Avec : addr(11)='0' et addr(10 :3)=dis et addr(2 :0)=dir. |
| **P_REG_REC (reg,dir,dis,adr)** | Réception via le réseau de voisinage : recevoir la donnée (la stocker dans reg) de la source. | p_addi r1,r0,dir<br>p_addi r1,r1,dis<br>p_addi r1,r1,adr<br>p_LW reg,0(r1) | l.addi r1,r0,dir<br>l.addi r1,r1,dis<br>l.addi r1,r1,adr<br>l.lwz reg,0x0(r1) | data=IORD (WRP_B, addr)<br>Avec : addr(11)='0' et addr(10 :3)=dis et addr(2 :0)=dir. |
| **P_GET_STATUS (reg,ident)** | lire le bit d'activité du PE identifié par "ident". | p_lui r1,0x9<br>p_ori r1,r1,0<br>p_addi r1,r1,ident<br>p_LW reg,0(r1) | l.movhi r1,0x9<br>l.addi r1,r1,ident<br>l.lwz reg,0x0(r1) | |
| **P_SET_STATUS (val,ident)** | modifier le bit d'activité du PE (identifié par "ident") par la valeur val. | p_lui r1,0x9<br>p_ori r1,r1,0<br>p_addi r1,r1,ident<br>p_addi r2,r0,val<br>p_SW r2,0(r1) | l.movhi r1,0x9<br>l.addi r1,r1,ident<br>l.addi r2,r0,val<br>l.sw 0x0(r1),r2 | |
| **P_GET_IDENT (reg)** | lire l'identité | p_lui r1,0x2<br>p_ori r1,r1,0<br>p_LW reg,0(r1) | l.movhi r1,0x2<br>l.lwz reg,0x0(r1) | NIOS2_READ_ CPUID(id) |
| **GET_OR_TREE (reg)** | lire la valeur du OR_Tree | addi r1,r0,0x9005<br>LW reg,0(r1) | l.addi r1,r0,0x9005<br>l.lwz reg,0x0(r1) | |

# Chapitre 5

# Outil de conception de mppSoC : mise en œuvre et expérimentation sur FPGA


**5.1 Introduction à l'IDM et automatisation de la génération de code . . . . . 107**
5.1.1 L'IDM pour la conception des systèmes embarqués . . . . . 108
5.1.2 Langage de modélisation UML . . . . . 109
5.1.3 Profil MARTE . . . . . 110
**5.2 Proposition d'une méthode d'utilisation de MARTE pour mppSoC . . . . 113**
5.2.1 Intégration dans Gaspard . . . . . 113
5.2.2 Modélisation de mppSoC . . . . . 115
**5.3 Chaîne pour la génération du code VHDL synthétisable . . . . . 125**
5.3.1 Organisation de mppSoCLib . . . . . 125
5.3.2 Transformation depuis un modèle Deployed vers du code VHDL . . 126
**5.4 Expérimentation sur FPGA : traitement vidéo temps réel . . . . . 128**
5.4.1 Plateforme de prototypage . . . . . 128
5.4.2 Chaîne de traitement vidéo à base de mppSoC . . . . . 129
5.4.3 Application de conversion de couleur . . . . . 133
5.4.4 Application de convolution . . . . . 138
**5.5 Comparaison de mppSoC avec d'autres systèmes . . . . . 141**
**5.6 Conclusion . . . . . 143**

Les flots de développements et les outils de conception de SoC n'ont pas été aussi prompts à suivre les grandes évolutions matérielles ainsi que logicielles. Il en résulte aujourd'hui un gap de productivité accentué par les lois économiques. En effet, les méthodes et outils de conception et de vérification couramment utilisés ne sont pas adaptés à gérer la complexité croissante des systèmes et de leur support physique. Une récente étude de l'ITRS [51] illustrée dans la figure 5.1 montrait que la demande du logiciel double chaque dix mois. Le même rapport mentionnait que les capacités d'intégration des SoC double chaque 36 mois et la productivité du logiciel double aussi chaque cinq ans. D'où le fossé entre capacité d'intégration théorique et pratique ne cesse de se creuser. Pour réduire les temps et les coûts de développement, deux aspects orthogonaux se révèlent indispensables : la montée dans les niveaux d'abstraction pour la spécification du système et la réutilisation des blocs prédéfinis communément appelés IPs. Alors que le premier aspect encourage à s'abstraire des détails d'implémentation, le second aspect encourage à capitaliser des développements qui ont aboutit à la validation d'un bloc.

Afin d'accélérer la conception de SoC, des approches de spécification orientées modèles sont alors apparues. La modélisation permet d'abstraire les aspects les plus importants pour les communiquer, les analyser et les valider avant toute implémentation. L'Ingénierie Dirigée par les Modèles (IDM) se situe dans ce cadre. Dans ce chapitre, nous allons montrer comment une approche dirigée par les modèles peut faciliter le développement du système mppSoC. Une démonstration de génération de code VHDL automatique à partir de la modélisation de l'architecture parallèle à un très haut niveau d'abstraction montrera tout le potentiel en matière de développement de ce type de systèmes embarqués et l'intégration dans un FPGA pour une évaluation réelle.

Ce chapitre présentera les grandes lignes de notre flot de conception de mppSoC. En premier lieu, nous brosserons un aperçu des standards OMG qui ont contribué à la réalisation de notre flot de conception. À ce titre, nous nous attarderons sur les différents concepts de l'ingénierie dirigée par les modèles ainsi que le profil MARTE. Après avoir brièvement présenté l'environnement GASPARD, nous ferons un petit tour d'horizon des métamodèles que cet environnement utilise pour la génération automatique du code. Une description bien détaillée de la modélisation de mppSoC sera ensuite donnée. Le dernier maillon de notre travail est la génération du code. Une illustration des étapes suivies afin d'aboutir à ce code sera alors mentionnée. La section d'après présentera les résultats expérimentaux de l'utilisation de mppSoC pour la définition d'une chaîne de traitement vidéo. Dans la dernière section, nous discuterons les performances de mppSoC en comparaison avec d'autres systèmes.

## 5.1 Introduction à l'IDM et automatisation de la génération de code

Toute personne ayant déjà tenté de lire et de comprendre du code afin de le réutiliser ou de le modifier pour répondre à ses besoins, sait à quel point cette tâche est longue et difficile quelque soit le langage de programmation et malgré les indispensables lignes de commentaires. La solution est donc de combler cette rupture sémantique par des abstractions qui nous permettront de spécifier notre programme dans un langage métier proche du domaine en question (*DSL : Domain Specific Language*). De plus, la spécification du système à un tel niveau d'abstraction permet d'abstraire les détails et techniques d'implémentation. Pour répondre efficacement au défi de conception posé par mppSoC, un important ingré-

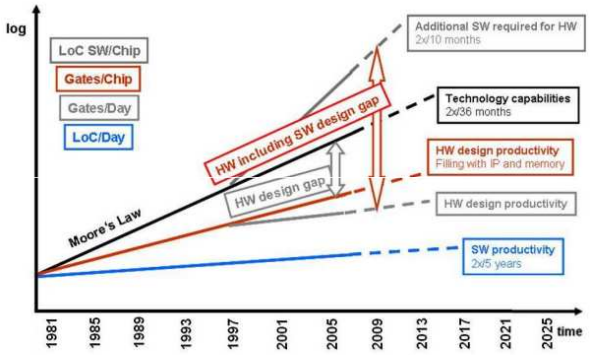



FIGURE 5.1 – *Gap entre la conception du matériel et du logiciel [51]*

dient doit être pris en compte : les moyens de conception à un haut niveau d'abstraction. Ces moyens permettraient d'aborder les systèmes tout en s'affranchissant, des détails qui font leur complexité. En revanche, ils doivent impérativement être accompagnés de processus de raffinement. L'ingénierie dirigée par les modèles vient répondre à ce besoin d'abstraction en plaçant les modèles au centre de développement logiciel. L'IDM permet de répondre à la complexité croissante des systèmes, en améliorant l'expressivité, la portabilité, la lisibilité et la communication des intentions entre flots de conception. Elle autorise également la séparation de préoccupations et la réutilisation. Forte de ces avantages, l'IDM recherche l'efficacité dans le processus de développement et se traduit par des gains en temps appréciables. L'IDM utilise en général le formalisme de spécification UML pour la description des modèles. Bien que l'IDM repose sur des concepts fondateurs (abstraction, modélisation, méta-modélisation, etc.) posés depuis longtemps, il s'agit avant tout d'un paradigme rassemblant de nombreux principes et pratiques existants autour de la notion centrale de modèle et cherchant à généraliser la proposition MDA [77] de l'OMG. MDA est une variante particulière de l'IDM. Le principe de base du MDA est l'élaboration de différents modèles, en partant d'un modèle métier indépendant de l'informatisation (*CIM : Computation Independent Model*), la transformation de celui-ci en modèle indépendant de la plate-forme (*PIM : Platform Independent Model*) et enfin la transformation de ce dernier en modèle spécifique à la plate-forme cible (*PSM : Platform Specific Model*) pour l'implémentation concrète du système. Les techniques employées sont donc principalement des techniques de modélisation et des techniques de transformation de modèles.

Dans le paragraphe suivant, nous exposerons les concepts fondamentaux de l'IDM.

### 5.1.1 L'IDM pour la conception des systèmes embarqués

De façon générale, l'IDM peut être définie autour de trois concepts de base : les modèles, les méta-modèles et les transformations.

**Modèle** : un modèle est une abstraction d'un système, modélisé sous la forme d'un ensemble de faits construits dans une intention particulière. Un modèle doit pouvoir être utilisé pour répondre à des questions sur le système modélisé.

**Métamodèle** : un métamodèle est un modèle qui définit le langage d'expression d'un modèle, ç.à.d. le langage de modélisation. Un modèle représente une vue abstraite de la

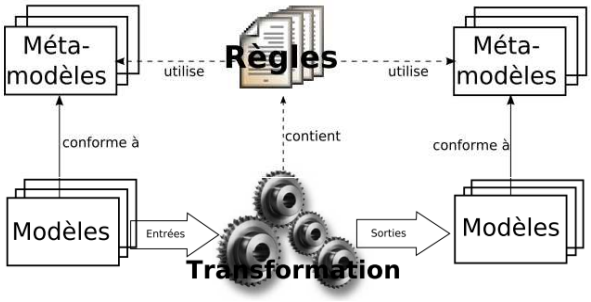



FIGURE 5.2 – *Principe de transformation de modèle*

réalité et est conforme à un métamodèle qui définit précisément les concepts présents à ce niveau d'abstraction ainsi que les relations entre ces concepts.

**Transformation de modèles** : présente un autre point clé de l'IDM. Elle permet de passer d'un modèle source décrit à un certain niveau d'abstraction à un modèle destination décrit éventuellement à un autre niveau d'abstraction. Ces modèles source et destination sont conformes à leur métamodèle respectif (métamodèle source et destination) et le passage de l'un à l'autre (i.e. la transformation) est décrit par des règles de transformation. Ces règles sont exécutées sur les modèles source afin de générer les modèles destination, comme illustré à la figure 5.2.

Dans notre travail, nous avons utilisé le langage de modélisation UML qui sera décrit dans la sous section suivante.

### 5.1.2 Langage de modélisation UML

Standardisé par l'OMG, UML a pour but de permettre la modélisation de n'importe quel système informatique. UML est riche en notation, il définit treize diagrammes organisés en trois catégories : six diagrammes de structure (qui décrivent la structure statique du système) dont le diagramme de classe et le diagramme de structure composite, trois diagrammes de comportement dont le diagramme d'activité et quatre diagrammes d'interactions dont le diagramme de séquence. La dernière évolution majeure de ce langage, UML 2, permet une modélisation encore plus précise de la structure et du comportement d'un système. Elle autorise en fait le passage d'une approche centrée sur le code à une approche dirigée par les modèles.

Une des particularités d'UML est qu'il peut se spécialiser pour des domaines particuliers via le mécanisme de profil. Un profil est une collection d'extensions et éventuellement de restrictions qui décrivent un domaine particulier. Ce mécanisme simple et efficace pour étendre UML, a permis de définir plusieurs profils adaptant UML à la majorité des domaines d'application. Plus d'une dizaine de profils ont été standardisée par l'OMG. L'extension d'UML via profil s'opère par les trois aspects suivants :

**Stéréotype** : un stéréotype permet d'étendre la sémantique des éléments de modélisation. Il s'agit alors d'un mécanisme d'extensibilité du métamodèle d'UML. Il peut posséder des propriétés et être sujet à des contraintes.

**Tag definition** : les propriétés du stéréotype sont un moyen d'annoter des caractéristiques supplémentaires au concept UML étendu. Elles ont impérativement un type et ce type

doit être un type défini dans UML, sinon il doit être défini et annexé au profil.

**Constraint** : on peut spécifier des règles OCL (*Object Constraint Language*) [78] qui viennent contraindre le métamodèle UML afin de le spécifier.

On se sert en général du profil pour introduire les concepts manquants pour les systèmes que l'on cherche à représenter comme par exemple les systèmes embarqués temps réel, tel est l'exemple du profil MARTE (*Modeling and Analysis of Real-Time and Embedded Systems*) présenté dans le paragraphe suivant.

### 5.1.3 Profil MARTE

Un profil UML MARTE [70] a été proposé par l'OMG afin de tenir compte des aspects logiciels et matériels des systèmes embarqués. Le profil MARTE a pour objectif d'étendre UML pour l'utiliser dans une approche de développement dirigée par les modèles. MARTE définit les fondations de la modélisation des systèmes embarqués temps réel. L'intention n'est pas de définir de nouvelles méthodologies de conception ou de nouvelles techniques d'analyse des systèmes embarqués temps réel, mais de les soutenir par une riche base d'annotations.

Le profil MARTE est identifié autour de trois packages permettant de spécifier à la fois les aspects logiciels et matériels des systèmes embarqués temps réel à savoir :

- Le modèle d'application : décrit les fonctionnalités du système.
- Le modèle des ressources : décrit la plateforme d'exécution en prenant compte des propriétés non fonctionnelles.
- Le modèle d'allocation : décrit l'allocation de l'application sur les ressources.

Nous détaillerons trois packages du profil MARTE que nous avons utilisés dans notre travail : HRM (*Hardware Resource Model*), GCM (*Generic Component Model*) et RSM (*Repetitive Structure Modeling*). Les packages du profil MARTE sont détaillés dans [79].

#### 5.1.3.1 Package HRM

HRM est un profil UML pour décrire du matériel existant ou bien en concevoir un nouveau, à travers plusieurs vues. Il regroupe la plupart des concepts du matériel sous une taxinomie hiérarchique avec plusieurs catégories selon leur nature, fonctionnalités, technologie et forme. Il comprend deux vues, une vue logique qui classifie les ressources du matériel selon leurs propriétés fonctionnelles, et une vue physique qui les distingue selon leurs propriétés physiques. Le profil HRM se compose de deux sous profils, le profil HwLogical explicitant le côté fonctionnel du matériel et le profil HwPhysical explicitant sa partie physique. Ces deux profils sont des spécialisations d'un modèle général commun. La structure du profil HRM est présentée dans la figure 5.3. Les stéréotypes introduits dans HRM sont organisés en un arbre d'héritages successifs depuis les stéréotypes génériques jusqu'aux spécifiques. Ceci explique l'aptitude du profil HRM à supporter plusieurs niveaux de détail. Cet aspect est aussi renforcé par des *tags definitions* facultatives qui ne sont spécifiées qu'une fois nécessaire. Le profil HRM a une autre qualité, celle de supporter la plupart des concepts du matériel grâce à un grand nombre de stéréotypes et encore une fois à son architecture en étages d'héritages. En effet, si aucun stéréotype spécifique ne correspond à un composant particulier, un stéréotype générique devrait convenir. Ceci permet donc d'inclure les nouveaux concepts du matériel de nouvelle nature ou issus d'une nouvelle technologie. En plus des stéréotypes, plusieurs types et plusieurs règles OCL sont spécifiés pour enrichir la

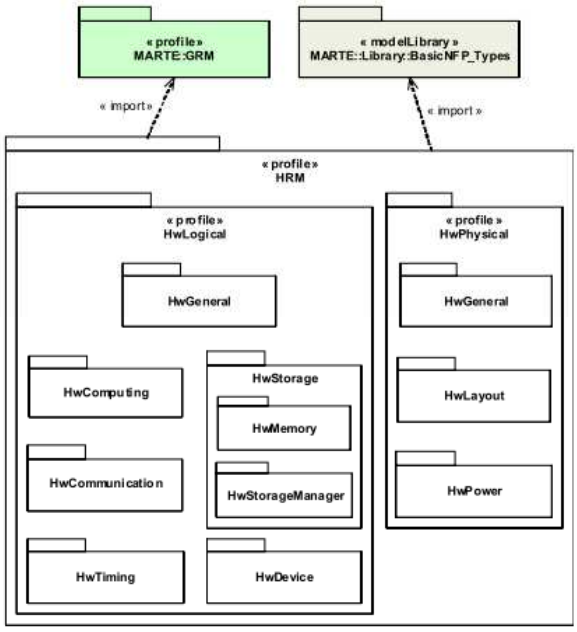



FIGURE 5.3 – *Structure du profil HRM*

sémantique et garantir la cohérence du métamodèle. Afin de faciliter l'utilisation du profil HRM, les noms des stéréotypes et les noms de leurs attributs sont choisis selon la terminologie conventionnelle du monde du matériel. De plus, ils sont préfixés par le label Hw pour lever l'ambiguïté entre logiciel et matériel.

Dans notre travail, nous avons tiré avantage du profil HwLogical. L'objectif du modèle logique est de classifier les ressources du matériel selon leurs fonctionnalités, en distinguant les ressources de calcul, de stockage, de communication, de temps et les ressources auxiliaires. Cette classification est basée principalement sur les services que chaque ressource offre et ne porte ni sur sa nature ni sa forme. Le modèle logique comprend six packages (voir figure 5.3), chacun pour une catégorie fonctionnelle. Dans la modélisation de mppSoC, nous avons utilisé trois packages : HwComputing, HwMemory et HwCommunication.

#### 5.1.3.2 Package GCM

L'objectif principal de ce package consiste à définir un support pour les approches de modélisation orientées composants en permettant la représentation de la structure d'un composant indépendamment de son environnement d'utilisation. Chaque composant est considéré comme un élément indépendant qui peut communiquer avec son environnement externe via la notion d'InteractionPort. La structure et la composition hiérarchiques d'un composant sont définies via la notion d'AssemblyPart et d'AssemblyConnector. Une vue globale du package GCM est illustrée dans la figure 5.4. La structure d'un composant peut être décrite alors via un ensemble de parts reliées entre elles et reliées aux ports du composant englobant par des connecteurs. Une part représente une instance potentielle de composant et peut être utilisée pour définir la structure d'un autre composant.

Le concept de FlowPort est introduit pour supporter le paradigme de flux orienté de communication entre les composants. Il permet d'identifier la nature de flux de communi-

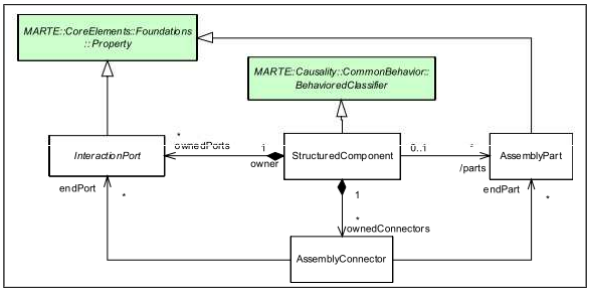



FIGURE 5.4 – *Vue globale du package GCM*

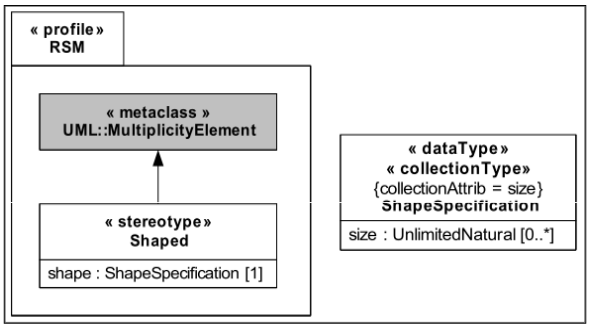



FIGURE 5.5 – *Le concept "Shaped" dans le package RSM*

cation émis ou reçu. Un FlowPort peut manipuler les flux entrants (*in*), sortants (*out*) ou bidirectionnels (*inout*).

#### 5.1.3.3 Package RSM

Le package RSM regroupe des concepts permettant de représenter d'une manière compacte des systèmes répétitifs réguliers. La sémantique dans ce package est basée sur le langage Array-OL (*Array Oriented Language*) [16] dédié aux applications de traitement de signal intensif. Le premier concept introduit dans ce package est celui de Shaped. Une vue globale de ce concept est illustrée dans la figure 5.5. Ce concept est appliqué sur une instance de composant ou un port afin de spécifier la multiplicité sur cet élément sous forme d'un tableau multidimensionnel. Une Shape est décrite avec un vecteur d'entiers strictement positifs représentant le nombre d'éléments dans chaque dimension du tableau. À titre d'exemple, pour modéliser une instance de composant répétée 40 fois sous forme d'un tableau bidimensionnel 4x10, nous représentons une seule instance avec une Shape de (4 ;10).

Le package RSM illustré dans la figure 5.6, caractérise également les topologies de lien pour exprimer la connexion entre des éléments répétés sous une forme compacte. Les trois concepts Tiler, InterRepetition et Reshape sont des exemples de ces topologies.

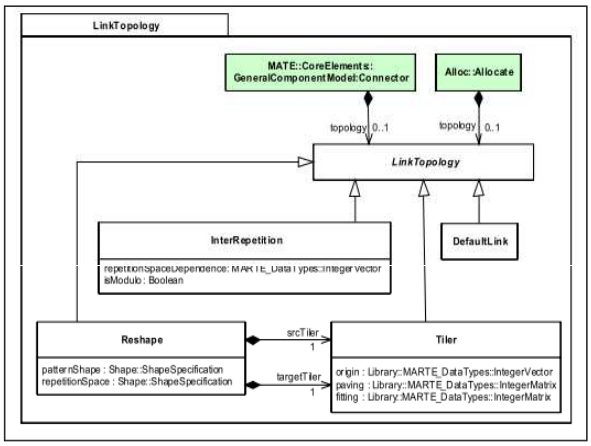



FIGURE 5.6 – *Vue globale du package RSM*

## 5.2 Proposition d'une méthode d'utilisation de MARTE pour mppSoC

Cette section présentera notre méthodologie de conception de mppSoC depuis la modélisation jusqu'à la mise en œuvre suivant les concepts de l'ingénierie dirigée par les modèles. Cette approche vise à fournir un cadre de développement logiciel dans lequel les modèles passent d'un état contemplatif à un état productif. L'état contemplatif correspond aux modèles de haut niveau utilisés pour exprimer les différentes parties de notre architecture ; tandis que l'état productif correspond au code exécutable du système. En particulier, nous nous intéresserons à la génération automatique du code VHDL synthétisable de la configuration SIMD choisie. Le passage entre les deux états contemplatif et productif se fait à travers des phases intermédiaires via un processus de transformation de modèles.

### 5.2.1 Intégration dans Gaspard

Gaspard (*Graphical Array Specification for Parallel and Distributed Computing*) [28] est un environnement de développement qui est basé sur le schéma en "Y" de Gajski et Kuhn [33]. Développé par l'équipe DaRT, Gaspard est dédié aux applications et architectures à parallélisme régulier. Se basant sur l'IDM, Gaspard sert pour la modélisation à haut niveau des applications et des plateformes matérielles. Gaspard offre la possibilité de concevoir à haut niveau des systèmes embarqués à hautes performances. Il permet la modélisation d'applications parallèles ayant des traitements intensifs et réguliers de données, ainsi que la génération automatique de code ciblant différentes technologies. Pour cela, Gaspard utilise le modèle de calcul dit répétitif et l'IDM. Le modèle de calcul répétitif est largement inspiré du langage Array-OL pour permettre la description de parallélisme de tâches et de données, de manière compacte. Il est également utilisable pour décrire des topologies et structures régulières au niveau matériel. Quant à l'IDM, il est utilisé en tant qu'approche de développement souple dans laquelle une abstraction de la complexité est rendue possible par le biais des modèles et de leurs transformations associées. De cette manière, les détails d'implémentation de bas niveau sont introduits progressivement lors de raffinement pour simplifier la modélisation ainsi que l'analyse d'un système.

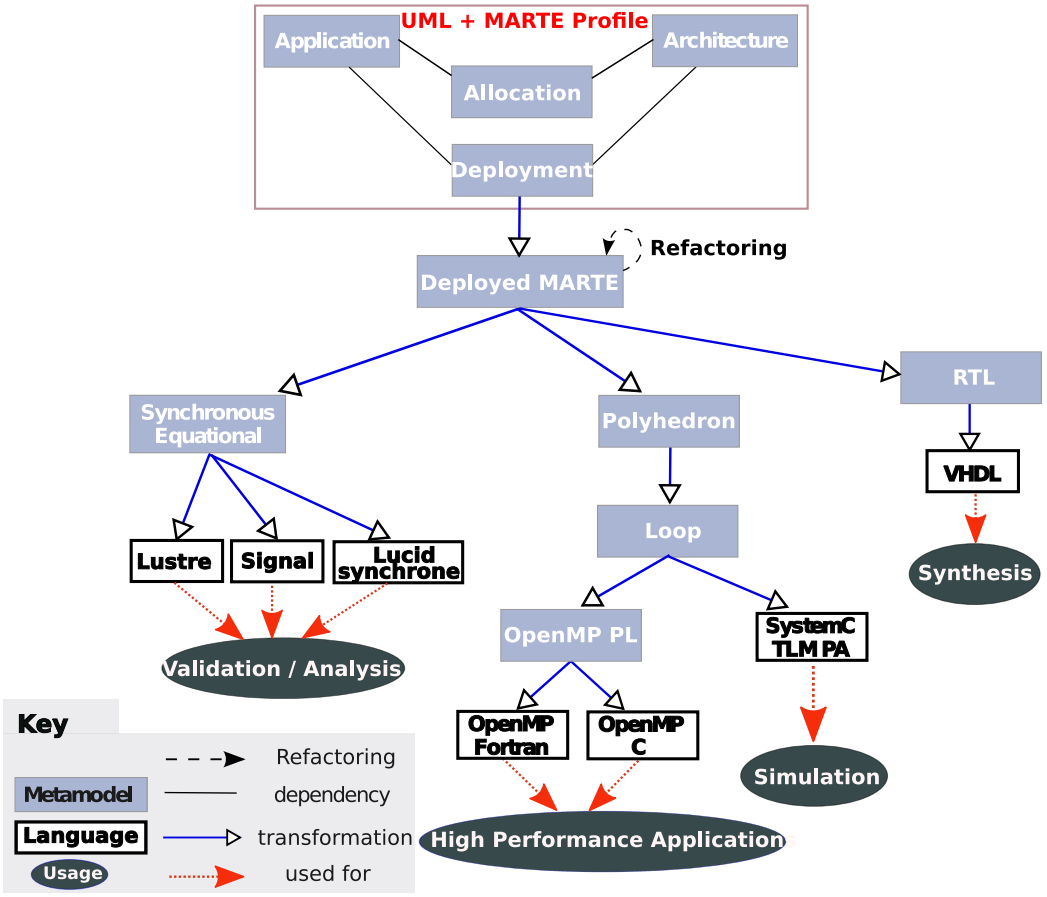



FIGURE 5.7 – *L'environnement Gaspard*

Gaspard permet au concepteur de SoC de modéliser indépendamment le comportement applicatif et l'architecture matérielle. En suivant le flot de conception en Y, le concepteur peut ensuite spécifier une association de l'application sur l'architecture. À partir de ces informations, des transformations de modèles permettent de générer des simulations à des niveaux de plus en plus précis, et après générer le code du SoC complet, à la fois matériel et logiciel. L'outil offre d'un côté l'utilisation de techniques de vérification puissante avant la phase de prototypage et de l'autre côté la réalisation automatique de l'allocation et de l'ordonnancement des applications sur l'architecture [29]. Les technologies MDA contribuent alors à exprimer les transformations de modèles entre les différents niveaux d'abstraction pour lesquels il existe un métamodèle associé. Ceci revient à réaliser des raffinements successifs entre les niveaux d'abstraction de la description depuis une spécification formelle jusqu'à un modèle synthétisable sur FPGA. Ainsi des métamodèles ont été développés ou sont en cours de développement pour spécifier l'application, l'architecture et l'association entre l'application et l'architecture, chacun présente des possibilités d'exprimer l'aspect répétitif et régulier du système.

La figure 5.7 représente l'implémentation de l'environnement Gaspard en IDM. Chaque boîte représente un métamodèle et les flèches entre ces boîtes dénotent des transformations de modèles. Le métamodèle de déploiement (*Deployed*) est celui de plus haut niveau d'abs-

traction à partir duquel différentes transformations permettent de raffiner un modèle jusqu'à la génération de code. Le mécanisme de déploiement permet de relier les composants atomiques de l'application et de l'architecture avec des IPs en bibliothèques. Il est possible d'utiliser des IPs pour les différents langages cibles de façon indépendante aux représentations de l'application et de l'architecture.

À partir d'un modèle conforme au métamodèle *Deployed*, l'environnement Gaspard peut générer plusieurs implémentations de ce modèle pour différentes plateformes d'exécution. En effet, Gaspard vise la génération de différents codes à partir d'un même placement d'une application sur une architecture à savoir :

- la génération du code VHDL correspondant à un accélérateur matériel capable d'exécuter l'application initialement modélisée à haut niveau.
- la génération de langages synchrones déclaratifs (tels que Lustre ou Signal) permettant de vérifier formellement la modélisation d'une application.
- la génération de langages procéduraux tels que Fortran/OpenMP rend possible l'exécution concurrente de différents processus sur une architecture multiprocesseur (architectures à mémoire partagée dans l'état actuel de Gaspard).
- la génération de code SystemC permet la simulation du comportement d'un SoC à différents niveaux d'abstraction.

Notre travail s'intègre en partie dans l'environnement Gaspard, plus spécifiquement dans la chaîne de génération VHDL. La chaîne actuelle de Gaspard ne peut pas être utilisée vu qu'elle manque des transformations, en plus elle ne tient pas compte des concepts UML haut niveau d'une architecture parallèle. Nous avons alors commencé par modéliser mppSoC en utilisant le métamodèle Gaspard. Ensuite, en tirant profit des chaînes de générations existantes, nous avons réalisé une chaîne de transformation du métamodèle Gaspard au code VHDL de mppSoC. En effet, l'environnement de conception Gaspard fournit un métamodèle pour l'expression de différentes parties constituant un système mppSoC. Ce métamodèle est défini autour de cinq packages : component, factorisation, application, HardwareArchitecture et association. Ces trois derniers présentent les aspects essentiels du modèle Y de Gaspard. L'objectif des packages component et factorisation est de regrouper les concepts communs utilisés par les différentes parties du modèle Y pour favoriser leur processus de réutilisation. Afin de profiter des outils standard de modélisation graphique, un profil UML équivalent à notre métamodèle est disponible. Cela permet aux utilisateurs de manipuler les différents concepts tout en ayant une présentation visuelle de leurs modèles. Ultérieurement, une transformation de modèles est utilisée pour passer du modèle conforme au profil UML vers le modèle conforme au métamodèle. Dans ce qui suit, nous présentons la manière de modélisation de mppSoC. Cette modélisation fait appel au profil MARTE tout en intégrant des solutions sophistiquées de Gaspard. Dans ce qui suit, nous nous intéresserons premièrement à définir les modèles de mppSoC. Deuxièmement, une étape de déploiement sera expliquée pour la génération complète du code de mppSoC.

### 5.2.2 Modélisation de mppSoC

Notre méthodologie de modélisation est incrémentale. Elle commence par modéliser les ressources élémentaires, et avec les compositions successives, elle finit en un modèle complet du système. Une incrémentation correspond donc à une composition. Cette méthodologie emploie principalement le diagramme UML de composite structure car il est adapté à la représentation du matériel. Dans cette partie, nous illustrons pas à pas notre méthodo-

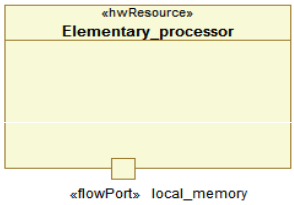



FIGURE 5.8 – *Modélisation du processeur élémentaire*

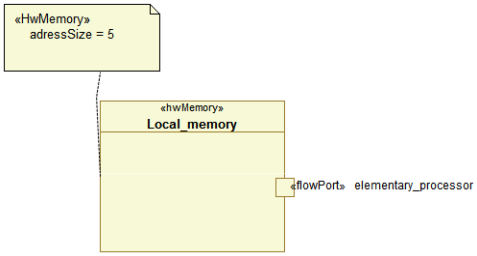



FIGURE 5.9 – *Modélisation de la mémoire locale du PE*

logie de modélisation de la structure entière de mppSoC. En premier lieu, nous définissons la configuration mppSoC sélectionnée en précisant l'arrangement et le nombre des PEs. En deuxième lieu, nous choisissons les réseaux qui vont être intégrés. Le réseau d'interconnexion va alors permettre d'identifier le schéma de communication. Après avoir identifié les parties élémentaires de notre système, nous en développons un modèle UML de haut niveau. Dans cette optique, nous commencerons par la modélisation des topologies 1D. Ensuite, nous présenterons notre stratégie pour modéliser les topologies 2D.

#### 5.2.2.1 Modélisation des composants élémentaires de mppSoC

Dans mppSoC, nous distinguons sept composants élémentaires : les PEs, les mémoires locales des PE, l'ACU, la mémoire d'instruction de l'ACU, la mémoire de donnée de l'ACU, le mpNoC et le périphérique d'entrée/sortie en cas de besoin.

La figure 5.8 illustre le composant élémentaire constituant la grille de processeurs : le processeur élémentaire. Ce dernier est modélisé par une classe nommée Elementary_processor. Cette classe est stéréotypée "hwResource" dans le cas d'une méthodologie de réduction de processeurs. Si le concepteur choisit d'adopter une méthodologie de réplication de processeurs alors cette classe doit être stéréotypée "hwProcessor". La classe Elementary_processor englobe un seul port stéréotypé "FlowPort" permettant de relier chaque PE à sa propre mémoire locale. Ce port doit être spécifié avec une direction bidirectionnelle (inout) pour indiquer qu'il s'agit bien d'une mémoire de donnée. La mémoire locale décrite dans la figure 5.9 est stéréotypée "hwMemory" indiquant qu'il s'agit d'une ressource de stockage. Le "tagged value" adressSize de la classe Local_memory est utilisé afin de fixer la taille de l'adresse mémoire de chaque mémoire locale qui présente un aspect paramétrique dans mppSoC et sa valeur est à fixer par le concepteur. Dans cet exemple, nous avons fixé sa valeur à 5, ce qui veut dire que la mémoire est de taille 128 bytes.

Trois autres types de composants élémentaires sont illustrés dans la figure 5.10 : l'ACU et ses mémoires locales. L'ACU est stéréotypé "hwProcessor" car il s'agit toujours

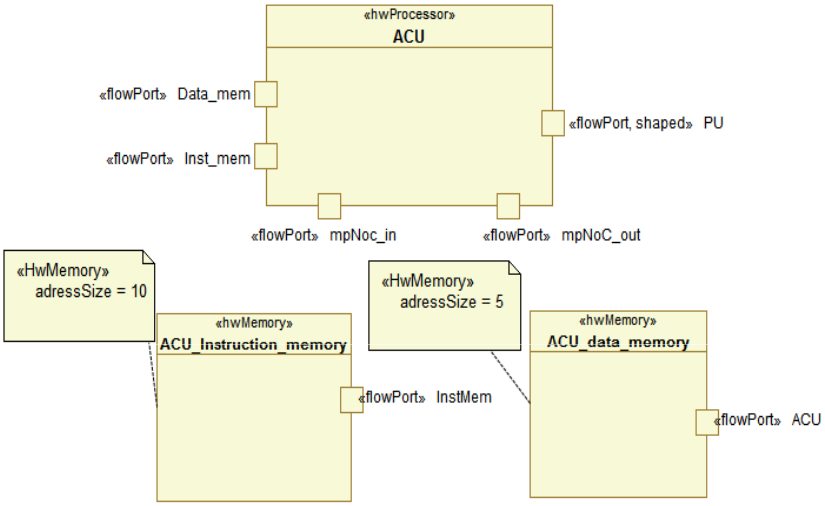



FIGURE 5.10 – *Modélisation de l'ACU et ses mémoires*

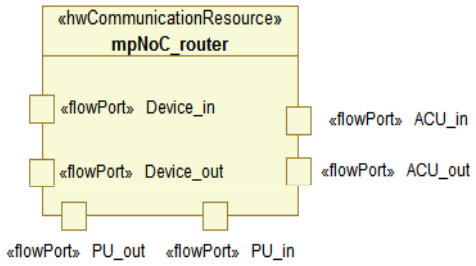



FIGURE 5.11 – *Modélisation du réseau mpNoC*

d'un processeur complet. Les deux mémoires, stéréotypées "hwMemory", sont reliées à l'ACU grâce aux ports ACU de la classe ACU_data_memory et InstMem de la classe ACU_Instruction_memory. La taille de l'adresse de ces deux mémoires doit être spécifiée par le concepteur en remplissant le "tagged value" adressSize du stéréotype "hwMemory". Telle qu'illustré dans la figure 5.10, la classe ACU englobe cinq ports. Les deux ports Data_mem et Inst_mem relient l'ACU à sa mémoire de donnée et sa mémoire d'instruction respectivement. Le port Data_mem doit être alors spécifié avec une direction bidirectionnelle (inout), alors que le port Inst_mem doit être spécifié avec une direction d'entrée (in) pour indiquer qu'il est connecté à une mémoire d'instructions. Le port PU (de direction *out*) permet de relier l'ACU à la grille des PE, son "shape" est égale à 1 indiquant qu'il s'agit d'un port de multiplicité 1. Les deux ports mpNoC_in et mpNoC_out permettent de relier l'ACU au réseau mpNoC. Ces deux ports sont à supprimer si la configuration mppSoC n'intègre pas un réseau mpNoC. Ce dernier est modélisé par la classe mpNoC_router illustrée dans la figure 5.11. Il contient deux ports d'entrée et de sortie à connecter à l'ACU nommés ACU_in et ACU_out respectivement, deux ports d'entrée et de sortie à connecter avec les PEs nommés PU_in et PU_out respectivement et deux ports en cas de besoin afin d'être connecté à un périphérique d'entrée/sortie. Le nombre d'une paire de ports (port d'entrée et port de sortie) destinée aux périphériques dépend de leur nombre dans la configuration.

Un périphérique d'entrée/sortie, dont le code est disponible dans la bibliothèque d'IP

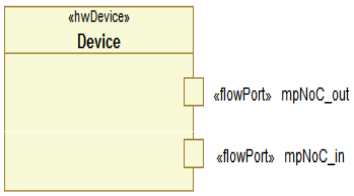



FIGURE 5.12 – *Modélisation du périphérique d'entrée/sortie*

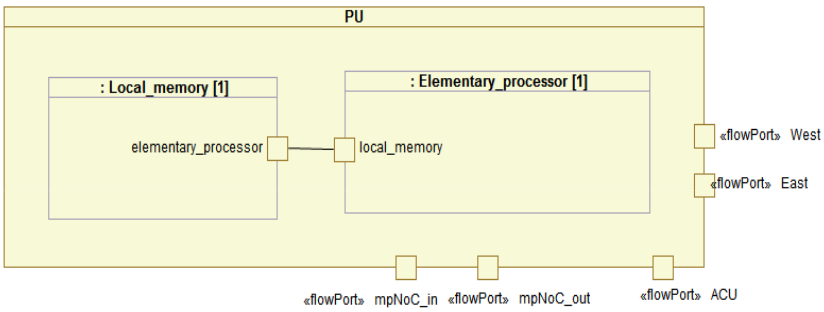



FIGURE 5.13 – *Modélisation du PU dans le cas d'un réseau de voisinage linéaire*

mppSoC, peut être modélisé dans un système mppSoC. Dans ce cas une classe stéréotypée "hwDevice" doit être modélisée intégrant deux ports (figure 5.12). Ces derniers permettent la connexion du périphérique au réseau mpNoC.

Après avoir modélisé chaque composant à part, reste à assembler les composants ensemble afin de former une configuration SIMD 1D ou 2D.

#### 5.2.2.2 Modèle 1D

Dans une configuration 1D de mppSoC, nous distinguons deux topologies de réseaux de voisinage : linéaire et anneau. La figure 5.13 illustre le composant élémentaire constituant un réseau de voisinage linéaire : l'unité élémentaire. Cette dernière est modélisée par une classe nommée (*PU*). Une unité de traitement élémentaire est composée d'un processeur élémentaire connecté à sa propre mémoire locale. La structure du réseau linéaire est spécifiée comme étant une répétition d'unités élémentaires sur une dimension, avec pour limite de répétition la valeur du "tagged value : shape" du stéréotype "Shaped" appliqué à la classe PU (voir figure 5.14). Cette valeur indique le nombre de PE dans la configuration mppSoC et doit être remplie par le concepteur. Dans la figure 5.15 nous avons choisi d'intégrer 4 PE dans l'architecture, pour cela la valeur {4} est attribuée au "tagged value" shape.

Chaque PU est connecté à ses voisins situés à l'est et à l'ouest, de façon linéaire. Le connecteur stéréotypé "InterRepetition" spécifie la topologie de connexion des PU. Les PU situés sur les bords du réseau ne sont pas connectés aux voisins situés sur le bord opposé. Cette contrainte est spécifiée via la valeur booléenne false du "tagged value : isModulo" du stéréotype "InterRepetition" illustré dans la figure 5.16. L'attribut "repetitionSpaceDependence" est un vecteur de translation sur l'espace du tableau multidimensionnel, qui identifie la position d'un voisin donné. Sa valeur est égale à {1} indiquant qu'un élément de position [i] est connecté à son voisin de position [i+1].

Le connecteur stéréotypé "Reshape" entre le port de l'élément répété et le port de l'ins-

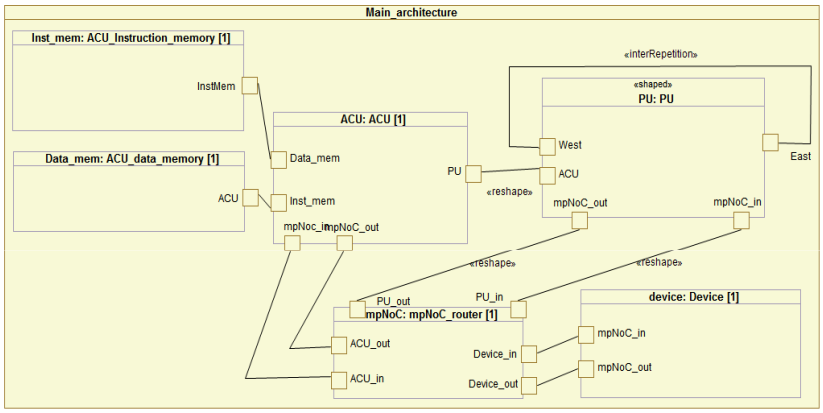



FIGURE 5.14 – *Configuration mppSoC intégrant un réseau de voisinage linéaire*

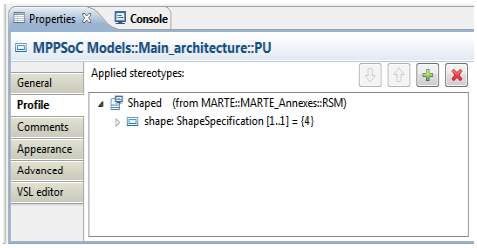



FIGURE 5.15 – *Spécification du nombre de PE*

tance ACU signifie que chaque port de la répétition est connecté au port PU de cette instance (voir figure 5.17). Chaque PU est aussi connecté à une entrée et une sortie du réseau mpNoC assurant les communications irrégulières. Pour relier chaque PE sur un port in et un port out du réseau d'interconnexion irrégulier, nous utilisons deux "Reshape" qui indiquent comment distribuer les deux ports de chaque PE sur les deux ports du réseau mpNoC de multiplicité 1 (voir figure 5.17). Le "repetitionSpace" correspond à l'espace de répétition parcouru pour remplir le tableau de sortie. Dans notre exemple, il s'agit d'une architecture contenant quatre processeurs donc la valeur du "repetitionSpace" est égale au nombre de processeurs. La valeur du patternShape est égale à 1 indiquant qu'il s'agit d'une distribution d'un port de multiplicité 1 sur les ports des quatre processeurs.

Un deuxième type de réseau de voisinage de topologie 1D peut être modélisé : le réseau de voisinage en anneau. Le réseau en anneau est composé aussi d'une répétition de

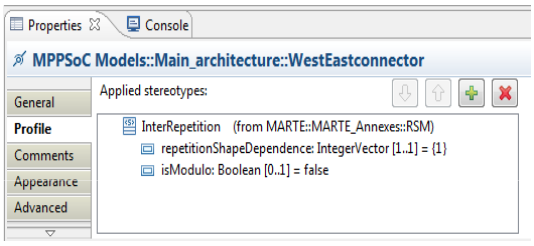



FIGURE 5.16 – *Stéréotype InterRepetition*

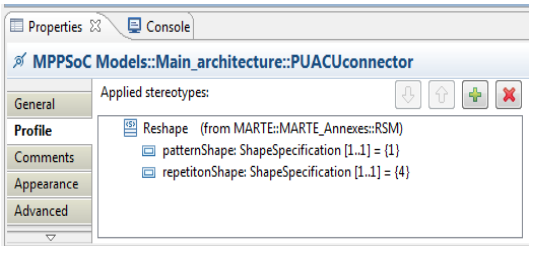



FIGURE 5.17 – *Stéréotype Reshape*

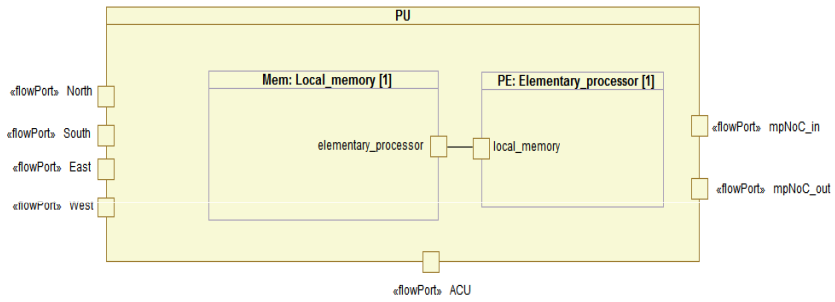



FIGURE 5.18 – *Modélisation du PU dans le cas d'un réseau de voisinage de type maille*

l'élément PU. Les mêmes concepts décrits ci-dessus sont présents dans ce cas. Une seule différence concerne la valeur du "tagged value : isModulo" du stéréotype "InterRepetition". Cette valeur est égale dans ce cas à true. En effet, les PU situés sur les bords du réseau sont connectés à leurs voisins qui sont situés sur le bord opposé.

Les deux topologies précédentes sont à une seule dimension. Dans ce qui suit, nous décrirons la modélisation des réseaux de voisinage à deux dimensions.

#### 5.2.2.3 Modèle 2D

Une configuration 2D de mppSoC peut avoir un réseau de voisinage avec une topologie parmi trois possibles : maille, tore ou Xnet.

**À base de réseau de voisinage tore ou maille**

Commençons par décrire l'unité de traitement élémentaire constituant un réseau de voisinage de type maille ou tore, illustrée dans la figure 5.18. Dans la structure du composant PU, nous distinguons deux parts. La première est une instance de la classe Elementary_processor. Elle dénote un processeur contenant un port. Le port local_memory assure la communication entre le processeur et la deuxième part nommée Mem qui présente la mémoire locale de données du PE.

La figure 5.19 présente une configuration mppSoC intégrant un réseau de voisinage de type maille. La "Shape" de l'instance de PU dénote qu'il s'agit d'une répétition d'éléments sur deux dimensions, dévoilant aussi le nombre de PE. Les connecteurs stéréotypés "InterRepetition" exprimés entre les bords de l'élément répété PU démontrent que chaque unité de traitement est connectée à ses voisins situés au nord, au sud, à l'est et à l'ouest. Dans la topologie maille, les voisins situés sur les bords du réseau ne sont pas connectés aux voisins situés sur le bord opposé (*isModulo = false*).

Le réseau de voisinage de type tore est modélisé en utilisant les mêmes concepts que nous avons exploité pour modéliser le réseau de voisinage de type maille. Seulement la valeur isModulo des deux stéréotypes "InterRepetition" est égale à true mentionnant que les unités de traitement présentes dans les bords du réseau sont liées à celles existantes dans les bords opposés. Les figures 5.20 et 5.21 dénotent les valeurs des "tagged value" du stéréotype "InterRepetition" liant les ports Ouest/Est et Nord/Sud respectivement. La valeur du tagged value "repetitionSpaceDependence" dans la figure 5.20 est égale à {1,0} indiquant qu'un élément de position [i,j] est connecté à son voisin de position [i+1,j] ; tandis qu'elle est égale

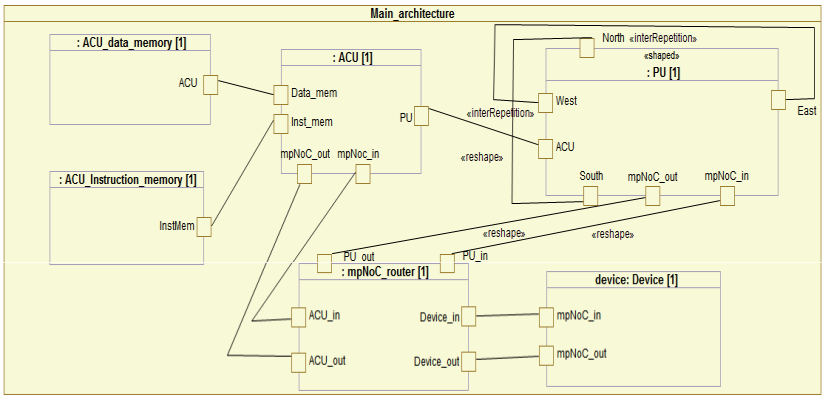



FIGURE 5.19 – *Configuration mppSoC intégrant un réseau de voisinage de type maille*

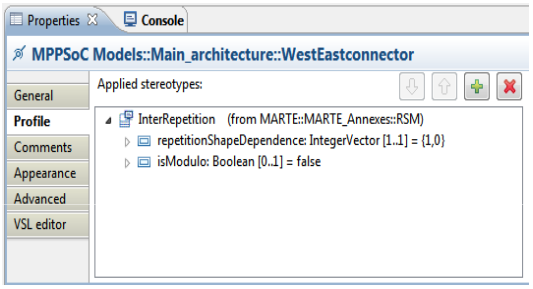



FIGURE 5.20 – *Liaison Est/Ouest*

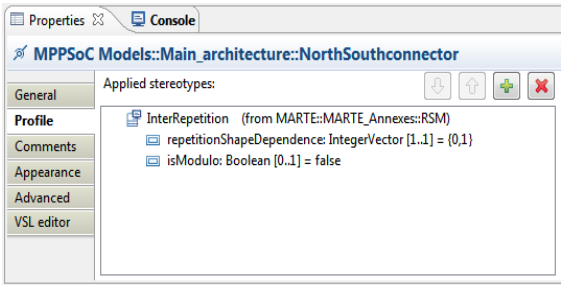



FIGURE 5.21 – *Liaison Nord/Sud*

à {0,1} dans la figure 5.21 dénotant que chaque élément ayant une position [i,j] est connecté à son voisin de position [i,j+1].

La modélisation globale d'une configuration 2D mppSoC est composée principalement d'un réseau de voisinage 2D où chaque unité de traitement constituant ce réseau est reliée à une entrée et une sortie de l'mpNoC grâce à deux connecteurs "Reshape" (figure 5.22). Dans notre exemple, il s'agit d'une architecture contenant 16 processeurs donc la valeur du "repetitionSpace" est égale au nombre de processeurs soit quatre lignes et quatre colonnes de processeurs. La valeur du patternShape est égale à 1 indiquant qu'il s'agit d'une distribution d'un port de multiplicité 1 sur les ports des 16 PEs. Une unité de traitement maître se présente aussi comme un composant principal de cette architecture englobant l'ACU et deux mémoires locales (instructions et données). Le concepteur peut intégrer un périphérique dans sa propre configuration selon ses besoins.

### À base de réseau de voisinage Xnet

Le réseau Xnet, tel que décrit dans le chapitre 3, section 3.4.3, assure des communications sur les quatre directions Nord, Sud, Est et Ouest en plus des connexions sur les diagonales : Nord Est, Nord Ouest, Sud Est et Sud Ouest. La figure 5.23 montre la modélisation de l'unité élémentaire en intégrant un réseau Xnet. Le modèle d'une configuration mppSoC à base du réseau Xnet est alors constitué d'une répétition à deux dimensions de la classe PU, comme présenté dans la figure 5.24. La seule différence entre la modélisation d'une configuration mppSoC à base d'une topologie maille ou une topologie en Xnet réside dans l'ajout des

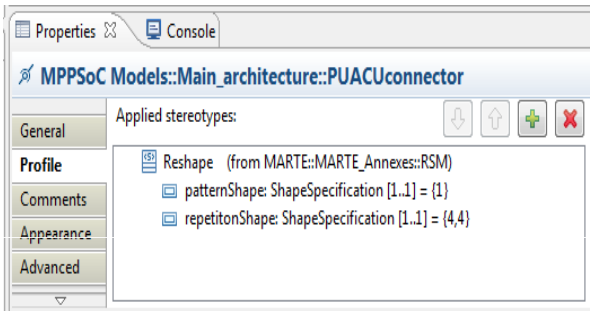



FIGURE 5.22 – *Connecteur Reshape*

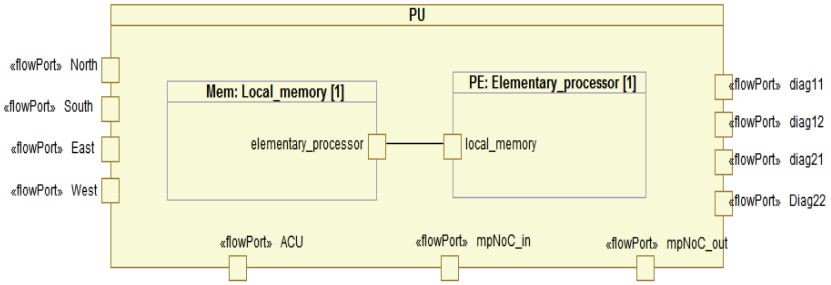



FIGURE 5.23 – *Modélisation du PU dans le cas d'un réseau de voisinage de type Xnet*

connexions sur les diagonales pour le deuxième type de réseau. Chaque instance est liée à ses huit voisins en utilisant des connecteurs stéréotypés "InterRepetition". Le connecteur liant les ports North et South lie chaque PU à ses voisins au nord et au sud respectivement. La valeur du tagged value "repetitionSpaceDependence" dans la figure 5.25 est égale à {0,1} dénotant que chaque élément ayant une position [i,j] est connecté à son voisin de position [i,j+1]. Le connecteur associant les ports West et East révèle que chaque PU est associé à ses voisins à l'ouest et à l'est respectivement. La valeur du tagged value "repetitionSpaceDependence" dans la figure 5.26 est égale à {1,0} dénotant que chaque élément ayant une position [i,j] est connecté à son voisin de position [i+1,j]. Une connexion entre un PU et ses voisins suivant

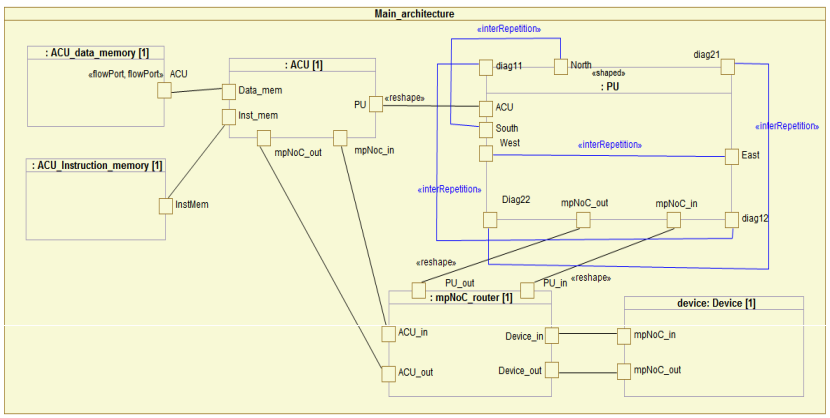



FIGURE 5.24 – *Configuration mppSoC intégrant un réseau de voisinage de type Xnet*

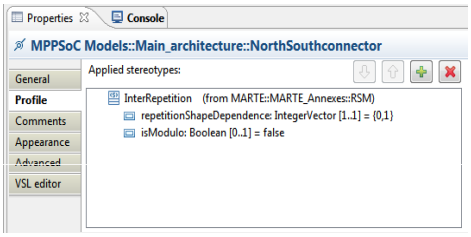



FIGURE 5.25 – *Connecteur Nord/Sud*

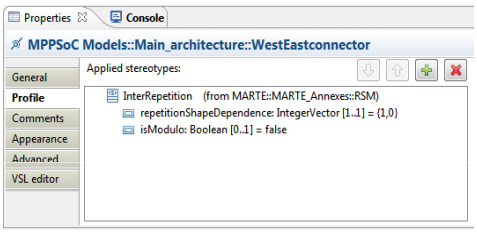



FIGURE 5.26 – *Connecteur Est/Ouest*

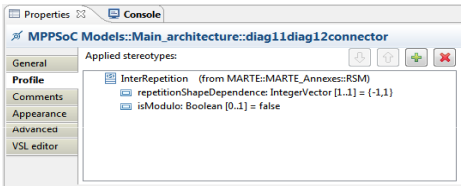



FIGURE 5.27 – *Connecteur diag11/diag12*


FIGURE 5.28 – *Connecteur diag21/diag22*

la première diagonale est assurée en utilisant un connecteur entre les ports diag11 et diag12. La valeur du tagged value "repetitionSpaceDependence" dans la figure 5.27 est égale à {-1,1} dénotant que chaque élément ayant une position [i,j] est connecté à son voisin de position [i-1,j+1]. Le même concept est utilisé pour les ports diag21 et diag22. La valeur du tagged value "repetitionSpaceDependence" dans la figure 5.28 est égale à {1,1} dénotant que chaque élément ayant une position [i,j] est connecté à son voisin de position [i+1,j+1].

La deuxième étape, après la modélisation de la configuration mppSoC voulue, consiste au déploiement des IPs. Cette étape est expliquée dans la sous section suivante.

#### 5.2.2.4 Déploiement en utilisant les IPs

Jusqu'à maintenant, nous avons présenté comment modéliser une configuration choisie de mppSoC à un haut niveau d'abstraction. Les différents aspects de l'architecture sont représentés en se basant sur le concept de composant et plus particulièrement celui de structure élémentaire. Néanmoins, cette description est dépourvue de toute description interne. En effet, le modèle mppSoC de haut niveau ne contient pas les informations nécessaires pour générer un modèle exécutable de bas niveau. Pour répondre à ce besoin les composants élémentaires doivent être associés à une implémentation déjà existante sous forme de code. Dans notre travail, nous nous basons pour cette fin sur la bibliothèque d'IPs mppSoCLib. La phase d'expression des liens entre la modélisation de haut niveau d'abstraction et les IP utilisés dans Gaspard est appelée déploiement.

Dans l'environnement Gaspard, la modélisation de haut niveau des IP logiciels et matériels a été concrétisée par le développement d'une bibliothèque nommée GaspardLib. Cette bibliothèque comporte différents composants (processeurs, caches, mémoires, etc.). Ils sont tous orientés pour la génération de codes à plusieurs niveaux d'abstraction. Nous avons alors intégré les composants IP de mppSoC dans la bibliothèque GaspardLib visant à générer un code complet de la configuration SIMD choisie par le concepteur.

Le passage de l'état contemplatif de notre modèle mppSoC à un état productif ou exé-

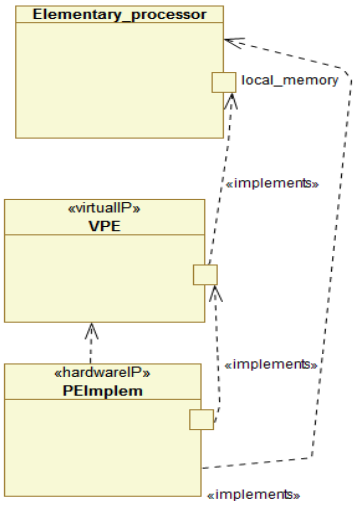



FIGURE 5.29 – *Déploiement d'un IP pour le PE*

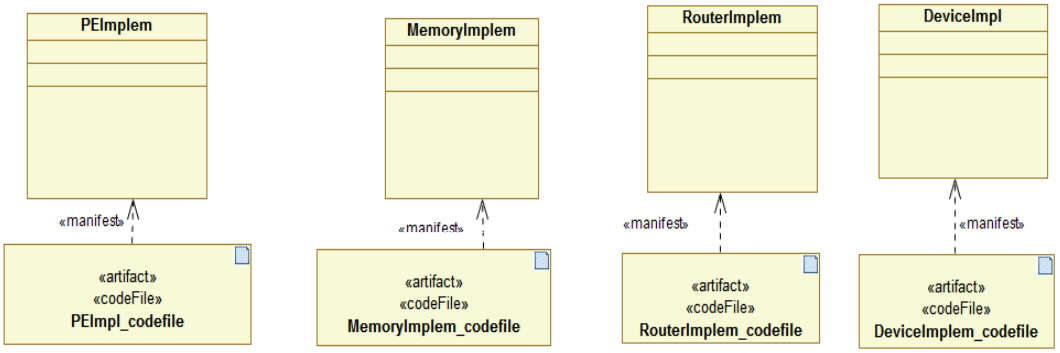



FIGURE 5.30 – *Le concept "CodeFile"*

cutable nécessite tout d'abord la définition d'une cible de compilation. Dans notre cas, nous nous intéressons à la génération automatique du code VHDL. Afin d'atteindre notre objectif, une phase de déploiement est nécessaire. Elle permet d'associer les composants élémentaires de haut niveau à des IPs réels intégrés dans la bibliothèque mppSoCLib. À chaque composant élémentaire de notre configuration mppSoC modélisée, lui correspond une fonction matérielle étant disponible dans la bibliothèque mppSoCLib. Le lien entre les composants modélisés en haut niveau et ces fonctions sera réalisé via les différents concepts introduits par le métamodèle de déploiement. La figure 5.29 présente une partie du déploiement des composants élémentaires. Il s'agit dans ce cas du déploiement d'un "hardwareIP" sur le processeur élémentaire. Un déploiement est aussi réalisé pour la mémoire d'instructions de l'ACU, le réseau mpNoC et le périphérique d'entrée/sortie.

Afin de permettre la génération de code, nous avons besoin de compiler ce code avec des fichiers sources de chaque IP utilisé. Pour cela, nous avons utilisé la notion de "CodeFile" pour préciser les informations concernant chacun de ces fichiers. La figure 5.30 illustre ce mécanisme en présentant le déploiement des composants élémentaires nécessaires pour générer le code d'une configuration mppSoC.

Jusqu'à cette phase de conception, nous avons décrit comment créer un modèle UML de

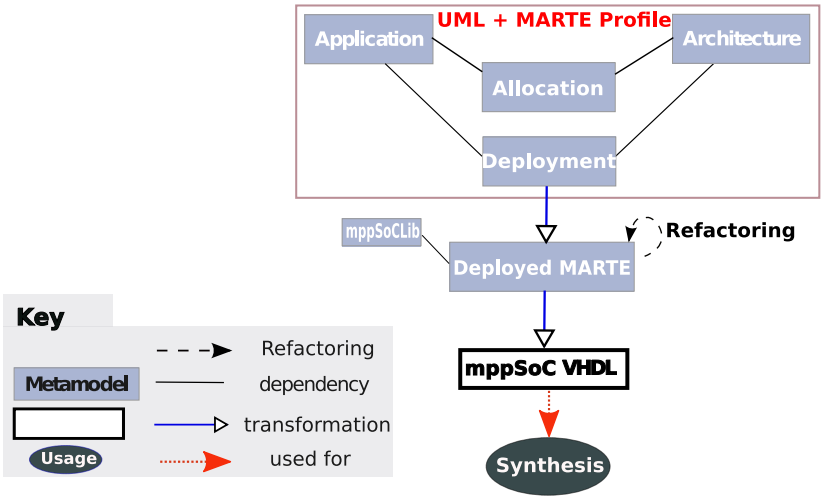



FIGURE 5.31 – *Chaîne mppSoC intégrée dans l'environnement Gaspard*

mppSoC en utilisant le profil Gaspard. Cette modélisation a été réalisée à l'aide de l'outil Papyrus. Les modèles des SoC sont adoptés par les chaînes de transformations de Gaspard jusqu'à arriver à la phase de génération de code. Les métamodèles et transformations actuelles de ces chaînes ne conviennent pas à la génération du code VHDL pour mppSoC. Nous avons alors réalisé notre propre chaîne décrite dans la section qui suit.

## 5.3 Chaîne pour la génération du code VHDL synthétisable

La chaîne de mppSoC que nous avons créé s'intègre dans le flot Gaspard tel que montré dans la figure 5.31. Le point de départ de notre chaîne de transformation est un modèle Deployed et son point d'arrivée est le code VHDL synthétisable. Nous utilisons en effet les chaînes de transformation de Gaspard pour passer du modèle UML de mppSoC jusqu'au modèle Deployed. À ce stade, nous considérons que le modèle du système mppSoC est suffisamment proche pour la génération du code VHDL. Pour cela, nous avons développé une dernière transformation pour atteindre notre objectif. Cette transformation, de type modèle-vers-texte, consiste à générer du code pour l'architecture matérielle. La génération du code de la partie matérielle dépend de la configuration de mppSoC modélisée par le concepteur. Un parcourt du modèle de l'architecture est réalisé en tirant avantage de l'outil Acceleo et une génération du code est ensuite réalisée en se basant sur la bibliothèque d'IPs mppSoCLib.

### 5.3.1 Organisation de mppSoCLib

L'architecture de mppSoC est configurable et paramétrique. Deux types d'IPs sont utilisés pour concevoir une configuration mppSoC : des IP que le concepteur doit choisir pour construire son architecture (les IPs composants) et des IP qui vont être générés automatiquement suivant ce choix pour assurer un fonctionnement juste de l'architecture (les IPs d'assemblage). Afin d'aider le concepteur, la bibliothèque mppSoCLib fournit ces deux sortes d'IPs pour alléger leur conception. Cette bibliothèque est incluse dans la bibliothèque Gas-


FIGURE 5.32 – *Organisation de mppSoCLib*

pardLib. Dans le but d'assurer une bonne utilisation de notre chaîne de transformation, les IPs sont regroupés d'une façon hiérarchique. La figure 5.32 dénote l'ensemble de dossiers construisant la bibliothèque mppSoCLib. Cette dernière contient l'ensemble des fichiers VHDL qui sont des briques de base pour la construction d'une configuration mppSoC. Le dossier "mpnoc" par exemple contient les fichiers relatifs aux réseaux d'interconnexion irréguliers. Le dossier "mppsoc-files" groupe les fichiers de configuration ; et le dossier "processor" contient les fichiers relatifs à chaque type de processeur (miniMIPS, MIPS, NIOS, OpenRisc).

Dans le paragraphe ci-après, nous présentons la transformation de modèle que nous avons appliquée pour passer d'un modèle mppSoC au code correspondant.

### 5.3.2 Transformation depuis un modèle Deployed vers du code VHDL

Un des objectifs de notre flot de conception est la génération d'un code VHDL lisible et compréhensible par les utilisateurs pour une synthèse sur des plateformes FPGA. Partant d'une configuration mppSoC modélisée par le concepteur, une transformation de ce modèle à un modèle conforme au métamodèle Deployed est ensuite réalisée. Un parcourt de ce modèle est effectué afin de grouper les paramètres de l'architecture mppSoC voulue. Le nombre de PE, la taille des mémoires, la topologie du réseau de voisinage et la méthodologie de conception sont directement déduits à partir des diagrammes modélisant l'architecture matérielle. Les autres paramètres (type du processeur, type du réseau d'interconnexion dans le mpNoC, etc.) sont déduits dans la phase de déploiement. Nous présentons ci dessous par exemple un extrait du code de notre chaîne de transformation. Cet extrait nous permet de connaître le type de processeur (getPeCodeFile) et le type de réseau d'interconnexion dans le mpNoC (getMPNOCCodeFile) à générer :

[ *query public getPECodeFile (m : Model) : CodeFile =*
*self.ownedElement->select (oclIsKindOf(CodeFile) and name =*
*'PEImpl_codefile')->asOrderedSet()->first()* ]

[*query public getMPNOCCodeFile (m : Model) : CodeFile =*
*self.ownedElement->select (oclIsKindOf(CodeFile) and name =*
*'RouterImplem_codefile')->asOrderedSet()->first()* ]

Pour générer les fichiers de configuration de mppSoC nous procédons comme suit :

1. Récupérer les paramètres nécessaires à remplir dans les fichiers de configuration. Le texte ci-après montre les deux méthodes qui permettent de connaître le nombre de lignes et le nombre de colonnes de la grille des PEs.

   [ *query public sl_nb_rows (s : AssemblyPart) : Integer =*
   *if (self.shape.size->size() = 1) then 1 else self.shape.size->at(1) endif* ]

   [*query public sl_nb_column (s : AssemblyPart) : Integer =*
   *if (self.shape.size->size() = 1) then self.shape.size->at(1) else*
   *self.shape.size->at(2) endif*]

2. Connaître le type de processeur à intégrer afin de générer le bon fichier de configuration. La ligne suivante montre le test réalisé afin de connaître s'il s'agit d'un processeur de type miniMIPS :

   [ *if m.getPECodeFile().sourceFilePath.contains('minimips')*]

3. Remplir le fichier de configuration approprié avec les paramètres.

Dans les sections précédentes, nous avons présenté l'usage de l'IDM pour résoudre la complexité de conception du système mppSoC. En partant d'une modélisation de l'architecture, une transformation de modèles permet d'obtenir le code VHDL synthétisable d'une configuration mppSoC. Nous disposons alors d'un flot de conception qui, à partir d'une modélisation en UML, génère le code VHDL.

Afin de faciliter et accélérer la conception d'une architecture SIMD, réponse à la question $Q_5$ posée à la fin du chapitre 2, nous avons développé une chaîne de génération automatique de configurations mppSoC. Cette chaîne se base sur l'utilisation de l'IDM permettant de faire face à la complexité de conception de mppSoC. En partant d'un modèle UML-MARTE d'une configuration SIMD, une transformation modèle vers texte permet de générer automatiquement le code VHDL synthétisable de l'architecture désignée. Ce code sert pour des fins de simulation ou prototypage sur des circuits reconfigurables.
La modélisation de l'architecture est indépendante de tout détail d'implémentation, dans la mesure où la saisie d'un modèle est réalisée dans un langage unifié. Notre flot de conception peut donc être utilisable même par des non spécialistes des architectures massivement parallèles sur puce.

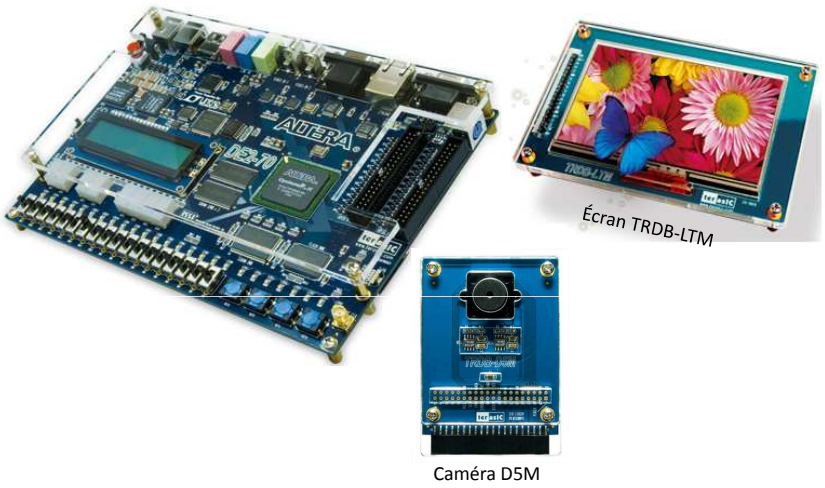



FIGURE 5.33 – *La plateforme DE2 70*

Reste maintenant la phase de test des performances du système conçu et implémenté dans un contexte réel. La section suivante s'inscrit dans ce cadre.

## 5.4 Expérimentation sur FPGA : traitement vidéo temps réel

Dans cette section, nous nous servons de la chaîne de génération de mppSoC développée afin d'implémenter et tester différentes configurations mppSoC. Le système parallèle est testé dans un contexte de traitement vidéo temps réel. Notre objectif est de valider le fonctionnement de notre système paramétrique sur la plate-forme cible. Dans une première partie, nous introduisons la plateforme de prototypage utilisée. Dans une deuxième partie, nous décrivons et discutons les différents tests expérimentaux réalisés.

### 5.4.1 Plateforme de prototypage

La plateforme de validation est la DE2 70 de Altera présentée dans la figure 5.33. Elle se compose principalement d'un FPGA Cyclone II EP2C70F896C6 et de plusieurs périphériques : des afficheurs LED, des mémoires (SRAM, flash...), des boutons poussoirs et des ports d'entrées/sorties divers (ethernet, série, JTAG...). L'FPGA Cyclone II utilisée contient 68416 éléments logiques et 250 blocs mémoires de type M4K RAM. Elle est aussi équipée avec une mémoire SDRAM de taille 32 Mbytes et une SSRAM de taille 2 Mbytes. Une caméra TRDB D5M à base d'un capteur CMOS 5 Mpixel a été aussi utilisée afin de capter les images avec une résolution de 15 images par seconde [103]. Un écran TRDB-LTM avec une résolution 480xRGBx800 sert pour l'affichage du flux d'images traité. Il fonctionne avec une fréquence de 33.2 MHz. Ses spécificités sont détaillées dans l'annexe F [104].

Les outils utilisés dans cette phase d'expérimentation sont l'outil de compilation et synthèse Quartus II V9.0 et l'outil de simulation Modelsim-Altera 6.4a. Le code VHDL généré d'une configuration mppSoC peut directement être simulé afin de tester le fonctionnement du système. Une étape de compilation et synthèse s'avère nécessaire afin de tester réellement le fonctionnement de toute l'architecture sur l'FPGA. Le fichier de configuration binaire est téléchargé du PC vers la plate-forme avec l'outil Quartus II.

### 5.4.2 Chaîne de traitement vidéo à base de mppSoC

Le flux vidéo est un flux de pixels provenant d'une image, capturée par la caméra, pixel après pixel, ligne après ligne. Il s'agit à partir de ce flux de répartir les pixels de manière adéquate dans les bancs mémoires de la SDRAM. Un pilote de caméra est implémenté en verilog et sert à stocker directement les pixels dans la SDRAM. Il est semblable à celui décrit dans [86]. Les pixels sont après lus de la mémoire et traités par le système mppSoC configuré à base de l'FPGA. Après le traitement parallèle, les pixels sont directement stockés dans une mémoire SRAM. Un afficheur LCD sert après à afficher l'image stockée sur l'écran. En effet, dans cette expérience chaque PE lit un pixel à la fois à partir de la SDRAM (définie dans le système par l'adresse 0x9008) à travers le mpNoC, et l'envoie après de la même manière à une SRAM (définie dans le système par l'adresse 0x9007). Le mpNoC assure alors le mode de communication PE-périphérique E/S : PE-SDRAM et PE-SRAM. L'afficheur LCD est directement connecté à la SRAM pour lire et afficher les pixels sur l'écran. Le pilote de l'afficheur est implémenté purement en matériel en utilisant le langage verilog. L'algorithme de traitement d'image testé est implémenté soit en assembleur, soit en C dépendamment du processeur utilisé. Le code suivant montre la programmation de l'application de capture en se basant sur l'assembleur MIPS étendu :

```
#Phase d'initialisation (initialiser les registres)
p_ori $1, $0, 0x9008
p_ori $2, $0, 0x9007
addi $8,$0,3 #mode PE-périph E/S
#Activer la capture de la camera (stocker 1 à l'add 0x9009)
ori $1, $0, 0x9009
addi $2,$0,1 #envoyer un signal d'activation
sw $2,0($1)
Loop :
#Mode PE - SDRAM
set_mode_mpnoc ($8)
#En parallèle : charger les données de la SDRAM
p_mpnoc_rec ($4,$1)
#Envoyer les données des PEs à la SRAM
p_mpnoc_send ($4,$2)
#Répéter la boucle
j Loop
```

***Programme mppSoC en assembleur MIPS étendu***

Dans cette expérimentation, la configuration mppSoC choisie contient un ACU, un nombre de PEs et un réseau mpNoC afin d'assurer les entrées/sorties parallèles. Différentes configurations sont testées en variant le nombre de PEs, l'IP processeur utilisé ainsi que la méthodologie de conception optée. En utilisant le flot de conception mppSoC, il était facile de passer d'une configuration à une autre. Avec la chaîne VHDL, nous générons automatiquement le code VHDL correspondant en quelques secondes. Le système implémenté dans ce cas (exemple avec 4 PEs) est présenté dans la figure 5.34.

Les tableaux 5.1 et 5.2 montrent les résultats de synthèse de différentes configurations mppSoC en appliquant la méthodologie de réduction et de réplication respectivement. Les

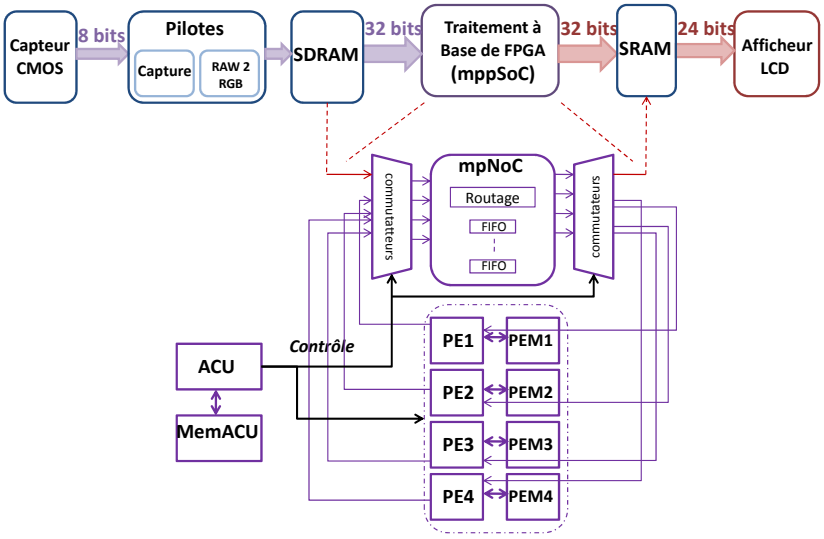



FIGURE 5.34 – *Chaîne de traitement vidéo à base de mppSoC intégrant 4 PEs*

TABLE 5.1 – Résultats de synthèse dans le cas de la réduction

| Nombre PEs | IP processeur | Ressources logiques fonctions combinatoires | registres | % | Mémoire totale ACU (bytes) | PE (bytes) | % | Conso. de puiss. (mWatts) |
|---|---|---|---|---|---|---|---|---|
| 4 | miniMIPS | 14176 | 4762 | 23 | 4096 | 4096 | 24 | 852.65 |
| 8 | miniMIPS | 21425 | 6006 | 34 | 4096 | 4096 | 33 | 940.07 |
| 16 | miniMIPS | 35673 | 8366 | 55 | 4096 | 4096 | 52 | 1296.05 |
| 32 | miniMIPS | 61130 | 13006 | 93 | 4096 | 2048 | 66 | 1969.89 |
| 4 | OpenRisc | 19674 | 7678 | 30 | 4096 | 4096 | 15 | 911.26 |
| 8 | OpenRisc | 34667 | 13534 | 53 | 4096 | 4096 | 22 | 1205.83 |
| 16 | OpenRisc | 64270 | 25160 | 98 | 4096 | 4096 | 36 | 1999.22 |

résultats sont présentés en termes d'éléments logiques, la taille mémoire utilisée (bytes) ainsi que la puissance consommée (mWatts). Toutes ces configurations contiennent un réseau mpNoC à base de crossbar. D'après ces deux tableaux, nous remarquons bien que la méthodologie de réduction permet d'intégrer un bon nombre de PEs sur une seule puce. Prenons le cas du processeur OpenRisc par exemple. Avec la méthodologie de réplication nous sommes limités à juste 8 PEs sur l'FPGA Cyclone II par contre nous avons pu avoir 16 PEs sur la même puce en appliquant la réduction. En comparant les différents processeurs utilisés, nous constatons que le processeur le moins coûteux en surface est le NIOS. Il est optimisé pour des FPGA Altera et il consomme le moins de puissance par rapport aux autres designs. Nous voyons aussi que le processeur miniMIPS est réduit en surface. En plus, il permet d'assurer un design portable sur n'importe quelle technologie FPGA. Le processeur OpenRisc reste un choix intermédiaire vu qu'il est indépendant de la technologie de l'FPGA, plus performant que le miniMIPS mais nécessite plus de ressources et consomme plus de puissance.

TABLE 5.2 – Résultats de synthèse dans le cas de la réplication

| Nombre PEs | IP processeur | Ressources logiques fonctions combinatoires | registres | % | Mémoire totale ACU (bytes) | PE (bytes) | % | Conso. de puiss. (mWatts) |
|---|---|---|---|---|---|---|---|---|
| 4 | miniMIPS | 21971 | 9243 | 37 | 4096 | 1024 | 11 | 1207.98 |
| 8 | miniMIPS | 42059 | 16863 | 71 | 4096 | 1024 | 18 | 1799.65 |
| 4 | OpenRisc | 18491 | 10229 | 41 | 4096 | 1024 | 13 | 1298.33 |
| 8 | OpenRisc | 40185 | 23966 | 91 | 4096 | 1024 | 22 | 1921.52 |
| 4 | NIOS | 6650 | 3353 | 10 | 8192 | 3072 | 19 | 563.27 |
| 8 | NIOS | 10180 | 4931 | 16 | 8192 | 3072 | 30 | 597.01 |
| 16 | NIOS | 18978 | 9069 | 29 | 12288 | 2048 | 59 | 866.13 |
| 32 | NIOS | 35700 | 17009 | 55 | 12288 | 1024 | 80 | 1266.01 |
| 48 | NIOS | 52678 | 24078 | 79 | 8192 | 512 | 87 | 1873.36 |

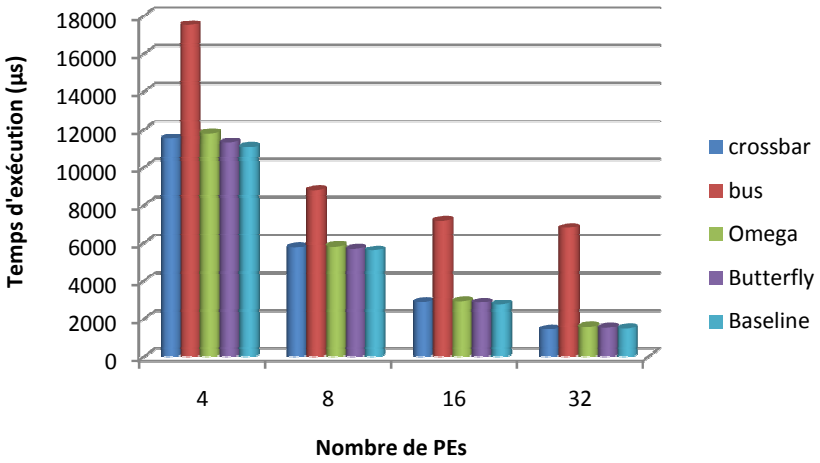



FIGURE 5.35 – *Influence du réseau d'interconnexion sur le temps d'exécution*

La mémoire présente de même un composant critique dans le contexte d'une intégration mono-puce. D'après le tableau 5.1, garder la même taille mémoire en partant d'une petite configuration à base de 4 PEs jusqu'à une configuration à base de 32 PEs, consomme pratiquement la totalité des blocs mémoires de l'FPGA. Donc le grand challenge à résoudre pour des implémentations sur une seule puce est la réduction ainsi que l'optimisation de l'utilisation mémoire.

Le réseau d'interconnexion crossbar a été choisi après effectuer une série de tests. En effet, les trois réseaux présents dans la bibliothèque mppSoCLib ont été testés dans des configurations mppSoC à base de miniMIPS réduit afin de sélectionner le réseau le plus adéquat. Pour générer ces différentes configurations, il était suffisant de modifier la phase de déploiement afin de choisir le réseau d'interconnexion. La figure 5.35 montre les temps d'exécution obtenus pour l'application de capture/affichage vidéo. Nous observons alors l'influence du choix du réseau de communication sur ces temps. Le bus paraît être inapproprié pour gérer des transferts de données parallèles. Il a en effet une bande passante limitée et ne permet pas d'accélérer des transferts de données parallèles, particulièrement

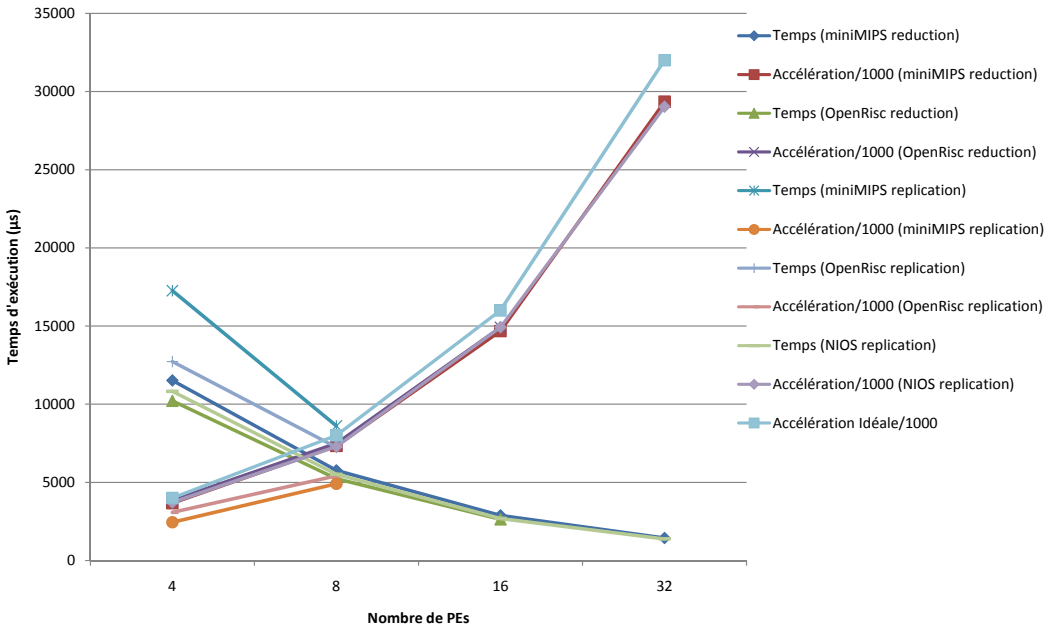



FIGURE 5.36 – *Résultats de simulation pour une image RGB de taille 800x480*

dans le cas de notre application qui gère un flot de données et pour laquelle nous devons assurer l'aspect temps réel. Un réseau multi-étages présente des performances comparables au crossbar en considérant de petites configurations mppSoC. La topologie baseline est la plus adéquate pour l'application étudiée. Cependant en ciblant des configurations avec un grand nombre de PEs, le crossbar devient le choix le plus optimal vu qu'il est le plus rapide. Par conséquent, c'est le réseau qui a été choisi.

Les temps d'exécution ainsi que les valeurs d'accélérations sont présentés dans la figure 5.36 en variant le nombre de PEs. L'accélération d'un algorithme parallèle, notée $S_p$, est le gain de temps obtenu lors de la parallélisation d'un programme séquentiel. Nous notons $t_{seq}$ le temps d'exécution séquentielle et $t_{par}$ le temps d'exécution parallèle. L'accélération est alors définie par :

$$S_p = t_{seq} \div t_{par}; avec 1 \leq S_p \leq p \tag{5.1}$$

On dit que le code est séquentiel si $S_p$=1 et "purement" parallèle si $S_p$=p (cas idéal). $t_{par}$ peut s'écrire sous la forme :

$$t_{par} = t_{seq} \div p + O_T \tag{5.2}$$

où $O_T$ est le temps supplémentaire dû au parallélisme (les surcharges).

Nous voyons l'extensibilité de mppSoC et l'amélioration de ses performances avec l'augmentation du nombre de PEs. Les temps d'exécution baissent considérablement en augmentant le degré de parallélisation de l'algorithme de capture. Le transfert de données s'effectue plus rapidement avec un nombre de PEs important. Nous remarquons que l'accélération obtenue suit l'évolution de l'accélération idéale. Ceci prouve les performances de mppSoC dans ce cas applicatif et montre aussi que cet algorithme est bien parallélisable et adapté à être exécuté en SIMD. Comme prévu, la méthodologie de réduction permet d'obtenir des

temps d'exécution plus faibles et donc des accélérations plus importantes que la méthodologie de réplication. Bien qu'elle nécessite un temps de développement assez long, elle est plus rapide en temps d'exécution. À titre d'exemple, une configuration mppSoC avec 4 PEs, à base d'un processeur miniMIPS réduit, est approximativement 1.5 fois plus rapide qu'une même configuration basée sur un processeur miniMIPS répliqué. Ce surplus de temps d'exécution dans la méthodologie de réplication s'explique par la complexité du PE induit par la charge de décodage ajoutée dans les PEs. Il existe alors un compromis temps de conception/temps d'exécution lors du choix de la méthodologie de conception du processeur.

En comparant les divers processeurs testés, l'OpenRisc montre les meilleurs performances. Ce processeur , ayant une fréquence de 100 Mhz, permet d'atteindre une accélération de l'ordre de 7.49 avec 8 PEs. Le processeur NIOS est aussi un bon processeur à intégrer. Son temps d'exécution est plus grand par rapport à l'OpenRisc dû au surcoût induit par les adaptateurs lors des transferts de données. Cependant, il reste aussi un choix convenable vu qu'on peut intégrer plus que 32 PEs sur l'FPGA Cyclone II et on peut atteindre une accélération de l'ordre de 29 avec 32 PEs.

Afin d'assurer un affichage du flux vidéo en temps réel, le traitement d'un pixel ne doit pas dépasser 30.12 ns. En testant les configurations, nous trouvons que seules celles intégrant 4 PEs à base de réplication en utilisant les processeurs miniMIPS et OpenRisc ne permettent pas d'assurer les temps réels. De ce fait, ces configurations mppSoC doivent être choisies avec un nombre plus grand de PEs (à partir de 8 PEs par exemple).

En explorant ces différents résultats expérimentaux, nous avons choisi d'étudier les deux processeurs miniMIPS (méthodologie de réduction) et NIOS dans les prochaines expérimentations. Le processeur miniMIPS est choisi pour étudier la performance d'un processeur simple et portable alors que NIOS est choisi pour étudier l'influence de l'utilisation d'un processeur softcore propriétaire dans une architecture SIMD particulière. En ciblant la plateforme DE2-70, le processeur OpenRisc ne permet pas d'obtenir une architecture parallèle intégrant un grand nombre de PEs (limitation à 16 PEs en appliquant la réduction). De plus, il consomme plus de puissance par rapport aux autres processeurs. Par conséquent, il ne constitue pas un choix pertinent dans ce cas. Il faut de même noter que les configurations à base de NIOS consomment le moins de puissance par rapport aux autres configurations. Une configuration mppSoC à base de miniMIPS réduit consomme approximativement 1.5 fois plus qu'une configuration à base de NIOS répliqué. En effet, plus la technologie est intégrée, moins elle consomme.

Les applications suivantes, exécutées sur mppSoC, constituent des benchmarks significatifs dans le traitement d'images. Elles sont extraites de la suite EEMBC (*Embedded Microprocessor Benchmark Consortium*) [32].

### 5.4.3 Application de conversion de couleur

Dans cette partie, nous allons tester un traitement simple à base de mppSoC. Une conversion de couleur du format RGB à YIQ est implémentée. Elle est expliquée dans la sous section suivante.

#### 5.4.3.1 Présentation de l'application

L'application consiste à convertir les couleurs des pixels RVB (RGB en anglais) vers le modèle YIQ. L'algorithme de compression d'image JPEG, par exemple, convertit les pixels

de l'espace RGB vers l'espace YIQ pour séparer les informations de luminance des informations de couleur. En effet, la chrominance sera plus compressée que la luminance, car l'oeil humain est plus sensible aux variations de luminance qu'aux variations de couleur. La base de couleur YIQ est aussi utilisée dans le standard NTSC (*National Television Systems Committee*) qui est un standard vidéo utilisé aux États Unis d'Amérique. L'équation de la conversion RGB vers YIQ est présentée ci dessous :

$$\begin{pmatrix} Y \\ I \\ Q \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{5.3}$$

Pour cette application, nous allons varier le nombre de PEs afin de trouver le bon nombre qui nous assure un traitement temps réel. La taille de mémoire varie de même selon les besoins de l'application. L'algorithme codé en C destiné à être exécuté sur le processeur NIOS est présenté ci dessous. Il convient à une configuration parallèle intégrant 4 PEs pour une image de taille 800x480 :

```
#include <stdio.h>
#include "nios2.h"
....
#define CMOS_Controller_0 (char*) 0x00009040
#define wrapperACU_0 (char*) 0x00004000
#define wrapperNIOS (char*) 0x00003008
....
alt_u16 WIDTH = 800 ; alt_u16 HEIGHT = 120 ;
int main()
{alt_u32 data ; ....
asm ("addi r7,r7,0 ;" : : : "r7") ; //code séquentiel
if (alt_timestamp_start() < 0) { printf ("No timestamp .\n") ; }
 else { time_start = alt_timestamp() ;
IOWR(CMOS_Controller_0,0,1) ; //Activer la caméra
set_mode_mpnoc (3) ; //Configurer le mode PE-périph E/S
 time1 = alt_timestamp() ;
asm ("add r7,r7,r0 ;" : : : "r7","r0") ; //code parallèle
NIOS2_READ_CPUID(id) ;
for(i = 0 ;i< HEIGHT-1 ; i++) {
for(j = 0 ; j < WIDTH-1 ; j++) {
p_mpnoc_rec (data, 0x709) ; //Lecture parallèle de la SDRAM
....
Y= (299*r + 587*g + 114*b)/1000 ;
 I= (596*r - 275*g - 321*b)/1000 ;
Q=(212*r - 523*g + 311*b)/1000 ;
datam = (Y + (I << 8) + (Q << 16)) & 0xffffffff ;
p_mpnoc_send (datam, 0x708) ; } } //Écriture parallèle à SRAM
asm ("addi r7,r7,0 ;" : : : "r7") ; //code séquentiel
time_elapsed = alt_timestamp() ;
temps = time_elapsed - time1 ; //calcul de temps
return 0 ; }
```

La fonction spécifique à NIOS "alt_timestamp" est utilisée afin de mesurer le temps d'exécution. Elle fournit en fait le nombre de cycles d'horloges.

Dans cet algorithme, nous n'avons pas besoin d'un réseau de voisinage. Seulement un réseau mpNoC à base de crossbar est utilisé pour les lectures et écritures de données parallèles. Chaque PE fait le traitement de conversion sur ses propres pixels. Une image est partitionnée horizontalement entre les différents PEs telle que montrée dans la figure 5.37 dans le cas de 4 PEs.

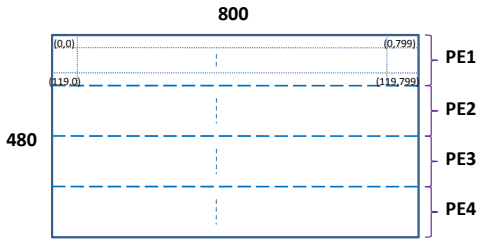



FIGURE 5.37 – *Partitionnement de pixels entre les PEs (cas de 4 PEs)*

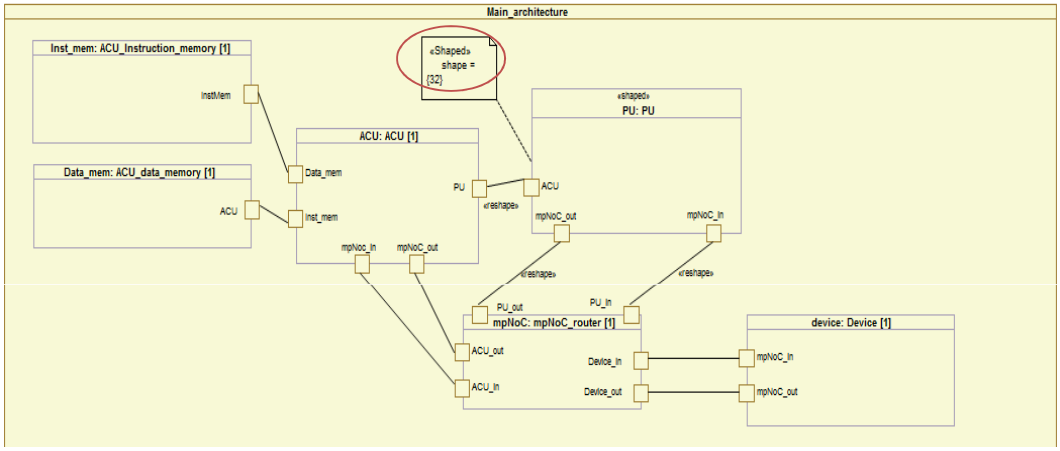



FIGURE 5.38 – *Spécification du nombre de PE*

### 5.4.3.2 Génération de mppSoC

Dans ce paragraphe, nous allons décrire la génération d'une configuration mppSoC en utilisant notre chaîne. L'architecture SIMD intègre 32 PEs. Elle est modélisée en utilisant le profil Gaspard. Chaque PE est un processeur miniMIPS réduit lié à sa propre mémoire locale (ayant une adresse de taille 5). La taille de la mémoire d'instructions de l'ACU est fixée à 4096 bytes. Cette architecture ne dispose que d'un réseau mpNoC à base de crossbar. Afin de fixer les paramètres de notre architecture, nous suivons les étapes décrites ci dessous :

1. Fixer le nombre de PE : dans notre architecture nous disposons de 32 PEs. Afin de spécifier cette valeur il suffit de fixer la valeur du tagged value "shape" du stéréotype Shaped qui est appliqué à la classe PU à 32 telle que montré dans la figure 5.38.
2. Fixer la taille des adresses mémoires : à cette étape il suffit de fixer la valeur des tagged values "adressSize" des trois classes stéréotypées "HwMemory", telle que spécifiée dans la figure 5.39.
3. Spécifier la méthodologie de conception : la méthodologie de réduction est choisie dans cette configuration. Il suffit alors d'attribuer le stéréotype "HwResource" à la classe Elementary_processor indiquant qu'il s'agit d'un processeur réduit (voir figure 5.40).
4. Indiquer le type de réseau de voisinage à intégrer : notre architecture ne contient pas de réseau de voisinage. Nous avons alors connecté chaque PU à l'ACU et au réseau mpNoC seulement indiquant l'absence du réseau de voisinage.

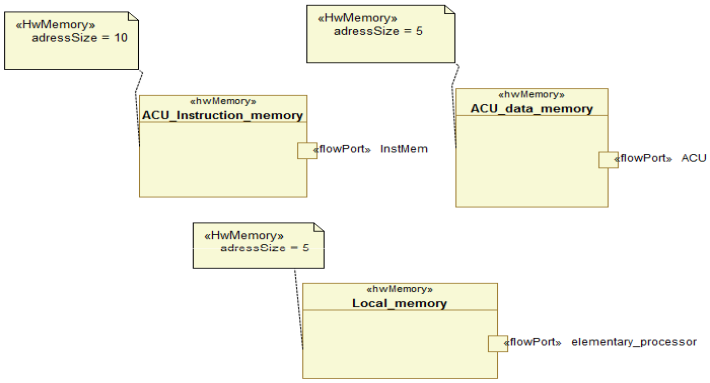



FIGURE 5.39 – *Spécification de la taille des mémoires*

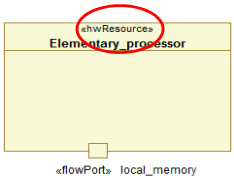



FIGURE 5.40 – *Spécification de la méthodologie de conception*

5. Spécifier le type de processeur : il faut indiquer le type du processeur dans le champ sourceFilePath du stéréotype CodeFile appliqué à l'artifact PEImpl_codeFile, comme illustrée par la figure 5.41.

6. Indiquer le type du réseau d'interconnexion dans le mpNoC : la dernière étape dans la conception de notre configuration consiste maintenant à définir le type du réseau d'interconnexion dans le mpNoC. Nous intéressons à intégrer un réseau de type crossbar. Pour cela nous avons spécifié le champ "crossbar" comme valeur pour le tagged value sourceFilePath relatif à l'Artifact RouterImplem_codefile (voir figure 5.42).

De cette manière, nous avons modélisé à haut niveau notre configuration choisie. Nous pouvons alors générer le code VHDL correspondant qui servira pour effectuer les mesures de performances présentées dans le paragraphe suivant.

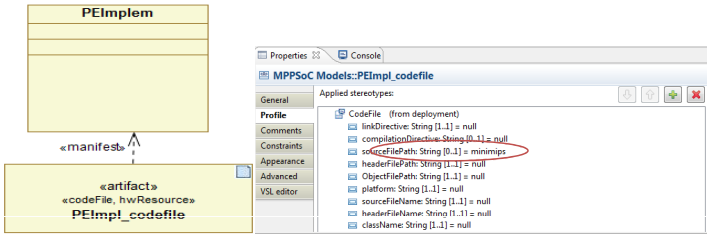



FIGURE 5.41 – *Spécification du type de processeur*

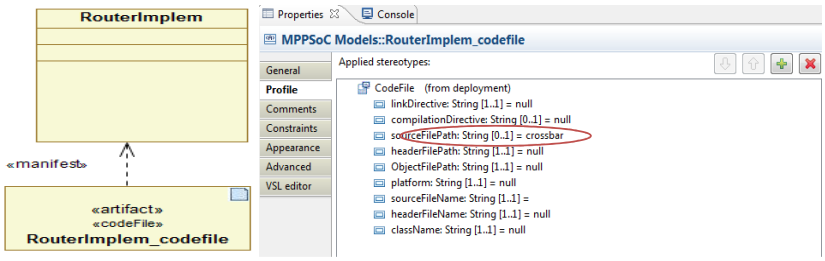



FIGURE 5.42 – *Spécification du réseau d'interconnexion de mpNoC*

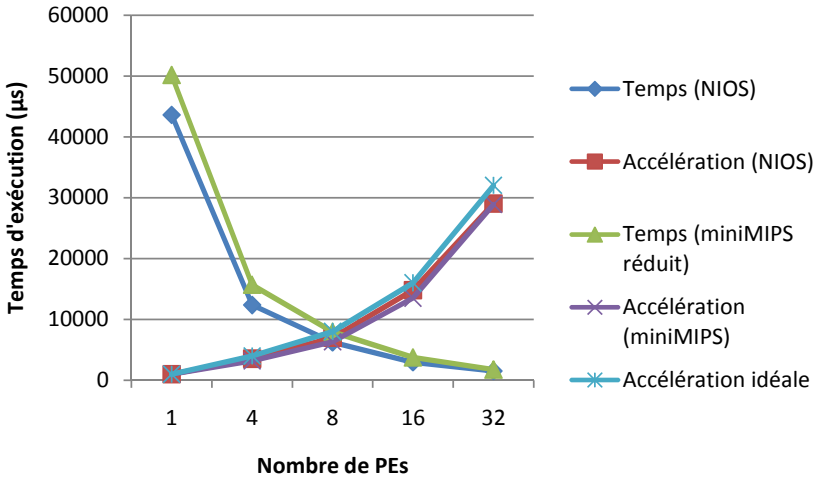



FIGURE 5.43 – *Résultats de simulation de la conversion RGB à YIQ*

### 5.4.3.3 Mesure de performances

La figure 5.43 montre l'évolution des temps d'exécution et accélérations en fonction du nombre de PEs et du processeur utilisé. Dans la figure 5.43, le nombre 1 de PE indique que la configuration SIMD est constituée seulement de l'ACU pour mesurer le temps d'exécution séquentielle. Pour un nombre N>1, la configuration est constituée de l'ACU et N PEs. Nous remarquons la réduction du temps d'exécution avec l'augmentation du nombre de PEs. Par rapport à la première application, nous avons juste ajouté des opérations de calcul ; les communications à travers le mpNoC sont les mêmes. De ce fait, nous observons que l'accélération est presque la même et suit aussi l'évolution de l'accélération idéale. La meilleure accélération est obtenue avec le processeur NIOS, de l'ordre de 29 en intégrant 32 PEs dans l'architecture. Nous voyons alors que le processeur NIOS présente les meilleurs résultats. Du fait qu'il est plus performant que le miniMIPS (plus rapide en calcul), les temps d'exécution obtenus avec le NIOS sont réduits d'un facteur 0.78 par rapport au miniMIPS. Il sera alors l'IP processeur choisi pour la prochaine application. Pour assurer l'aspect temps réel, nous avons besoin d'intégrer des configurations mppSoC intégrant 8 PEs ou plus.

D'après ces résultats, nous validons la facilité de tester différentes configurations mppSoC en utilisant l'outil de génération développé. Cela nous facilite l'exploration et nous guide dans le choix de la configuration la plus adaptée à une application donnée. Nous montrons de même qu'une architecture SIMD est bien adaptée pour exécuter des applica-

tions traitant un flot de données. Intégrer plus de PEs dans l'architecture permet d'accélérer l'exécution surtout en l'absence de communications entre les processeurs. Dans la prochaine sous section, nous testons une application de convolution et nous étudions l'influence de l'ajout de communications de voisinage sur la performance de mppSoC.

### 5.4.4 Application de convolution

À ce niveau, nous allons appliqué un filtre Laplacien sur l'image captée. Cette application est mieux expliquée dans ce qui suit.

#### 5.4.4.1 Présentation de l'application

La convolution par un masque est réalisée en remplaçant les composantes de chaque pixel par une combinaison linéaire des composantes pixels voisins. Les coefficients de la combinaison linéaire sont les coefficients du masque. Parmi les filtres qui servent pour la détection de contours, nous notons le filtre Laplacien. Les contours correspondent en général à des changements brusques de propriétés physiques ou géométriques de la scène. Le filtre Laplacien est un filtre de convolution particulier utilisé pour mettre en valeur les détails qui ont une variation rapide de luminosité. Le Laplacien est donc idéal pour rendre visible les contours des objets, d'où son utilisation dans la reconnaissance de formes dans des applications militaires, puis civiles. Le masque utilisé dans ce cas (matrice 3x3) est le suivant :

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} \tag{5.4}$$

Dans ce cas, deux versions algorithmiques ont été implémentées. La première ne contient pas de communication de voisinage dans la quelle les traitements sur les bords ne sont pas traités. La deuxième version inclut les communications de voisinage. Nous choisissons alors d'intégrer le réseau tore. En effet, la topologie 2D est la mieux adaptée pour le traitement d'images. Le réseau tore présente le meilleur choix vu que les PEs sur les bords de la grille ont besoin d'échanger des données.

#### 5.4.4.2 Génération de mppSoC

Dans ce paragraphe, nous allons décrire la génération d'une configuration mppSoC à base de NIOS en utilisant notre chaîne. L'architecture SIMD décrite intègre 32 PEs, un réseau de voisinage configuré en tore et un réseau mpNoC à base de crossbar. Les différences entre cette configuration et celle décrite dans la sous section 5.4.3.2 est le type du processeur utilisé, la méthodologie de conception, la taille des mémoires ainsi que l'intégration d'un réseau de voisinage. Dans ce cas, la mémoire de données est de taille 1024 bytes et la mémoire d'instructions a une taille de 32768 bytes. Les étapes de modélisation de cette configuration sont les suivantes :

1. Spécifier le nombre de processeurs élémentaires : les PEs, au nombre de 32, sont arrangés dans une grille 2D. Nous avons alors spécifié le shape du stéréotype Shaped appliqué à la classe PU à {4,8} indiquant que notre grille de processeurs englobe 4 lignes et 8 colonnes, comme illustrée dans la figure 5.44.

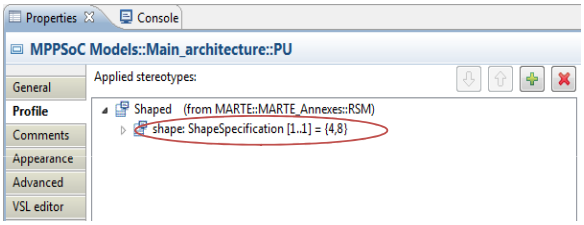



FIGURE 5.44 – *Spécification du nombre de processeurs élémentaires*

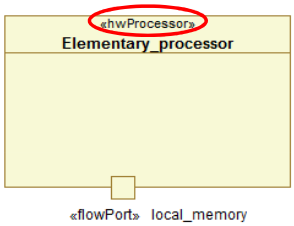



FIGURE 5.45 – *Modélisation du processeur dans le cas de réplication*

2. Fixer la taille des adresses mémoires : en fixant la valeur des "tagged values" adressSize des trois classes stéréotypées "HwMemory".
3. Spécifier la méthodologie de conception : dans ce cas il faut attribuer le stéréotype "HwProcessor" à la classe Elementary_processor indiquant qu'il s'agit d'un processeur répliqué (complet), comme le montre la figure 5.45.
4. Indiquer le type de réseau de voisinage à intégrer : notre architecture dispose d'un réseau de voisinage de type tore. Nous avons alors défini une architecture composée d'une répétition de PU sur deux dimensions. Chaque PU est connecté à ses voisins au nord, au sud, à l'est et à l'ouest. Le nombre de ports servant à modéliser le réseau de voisinage est égal à 4. Une connexion de type InterRepetition existe entre les ports North/South et West/East et la valeur du tagged value isModulo est fixée à true car les PU dans les bords sont connectés à ceux existants dans les bords opposés.
5. Spécifier le type de processeur : il suffit d'indiquer le type du processeur (dans notre cas : NIOS) dans le champ sourceFilePath du stéréotype CodeFile appliqué à l'artifact PEImpl_codeFile (voir figure 5.46).

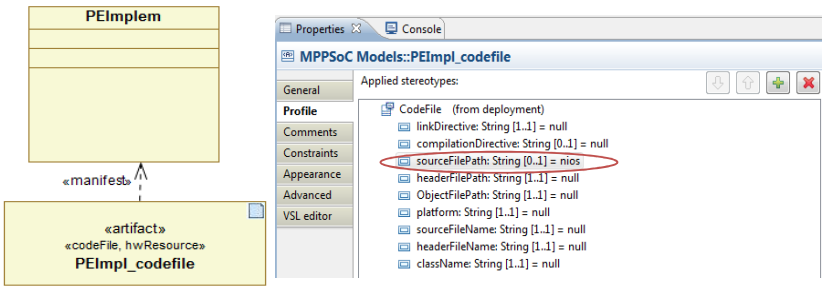



FIGURE 5.46 – *Choix du type de processeur*

TABLE 5.3 – Résultats de synthèse

| Nombre PEs | Ressources logiques fonctions combinatoires | registres | % | Conso. de puiss. (mWatts) |
|---|---|---|---|---|
| 32 | 61130 | 13006 | 93 | 1919.46 |

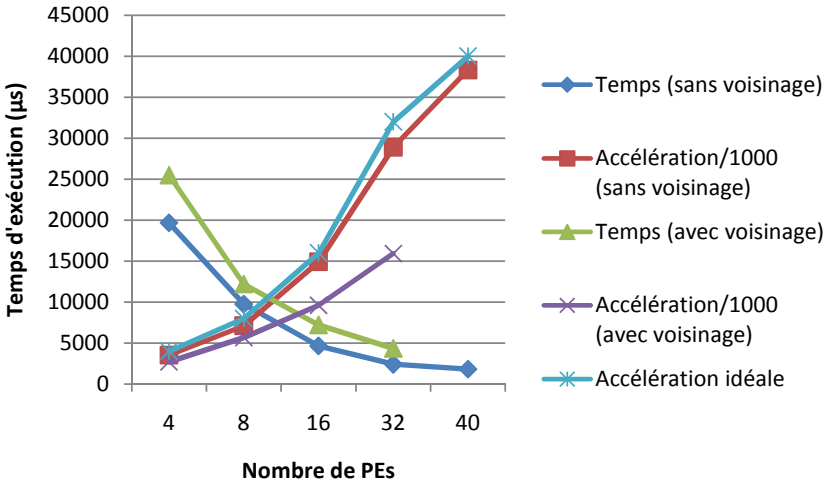



FIGURE 5.47 – *Résultats de simulation de la convolution*

6. Définir le type du réseau d'interconnexion dans le mpNoC : nous avons alors spécifié le champ "crossbar" comme valeur pour le tagged value sourceFilePath relatif à l'Artifact RouterImplem_codefile.

L'étape suivante consiste à générer automatiquement le code VHDL. Après une phase de compilation et synthèse sur l'FPGA Cyclone II, nous avons pu effectuer les mesures de performances détaillées dans ce qui suit.

#### 5.4.4.3 Mesure de performances

Le tableau 5.3 montre les résultats de synthèse en intégrant 32 PE à base de NIOS et un réseau de voisinage configuré avec la topologie tore. Nous voyons que cette configuration est la plus grande configuration possible sur l'FPGA Cyclone II.

La figure 5.47 montre l'évolution des temps d'exécution et accélérations en fonction du nombre de PEs et de la version de l'application. Comme prévu, l'algorithme sans voisinage est plus rapide que l'autre version ajoutant les communications de voisinage. Ces communications retardent l'exécution de l'ordre de 1.5 %. Le temps d'exécution augmente considérablement avec l'augmentation de ces communications (avec un nombre plus grand de PEs). Ceci est bien illustré en voyant l'évolution de l'accélération avec voisinage. En effet, sans inclure les communications, l'accélération suit la courbe idéale et nous pouvons atteindre par exemple une accélération de l'ordre de 28.8 avec 32 NIOS et qui augmente avec le nombre de PEs. Cependant, cette valeur chute à 15.92 en considérant les communications entre les PEs. Malgré coûteuses, les communications inter-PE ne dégradent pas trop les performances de mppSoC. Nous constatons de même que 8 PEs sont suffisants pour assurer

TABLE 5.4 – Comparaison des temps d'exécution

| Système | Application | Taille image | Temps d'exécution (ms) |
|---|---|---|---|
| SIMD (32PEs) | convolution | 512x512 | 3 |
| 2D systolique | convolution | 512x512 | 5 |

une convolution en temps réel (25.34 ns /pixel) sans considérer les traitements des pixels situés entre les PEs. Dans ce dernier cas, il faut intégrer plus que 8 PEs. Une configuration avec 16 PEs est largement suffisante et produit chaque pixel en un temps égal à 18.74 ns.

D'après ces différentes expérimentations, nous remarquons la performance de mppSoC à exécuter rapidement des algorithmes data parallèles mais aussi à répondre aux exigences de différentes applications nécessitant de plus des communications. Nous constatons clairement que le réseau mpNoC intégré dans mppSoC est très performant à gérer des transferts de données data parallèles sans dégrader les performances. Ces communications qui étaient depuis longtemps un problème majeur dans les architectures SIMD anciennes est bien résolu dans notre système. Néanmoins, il s'avère nécessaire de comparer mppSoC avec d'autres systèmes afin d'étudier mieux ses caractéristiques et performances ; ce qui fera l'objet de la section suivante.

## 5.5 Comparaison de mppSoC avec d'autres systèmes

Cette section est dédiée à comparer les performances obtenues avec mppSoC par rapport à d'autres systèmes. Vu que les configurations à base de NIOS présentent les meilleurs performances, elles ont été retenues pour les différentes comparaisons présentées dans ce qui suit. Différentes architectures SIMD ont été récemment proposées [63, 14, 115, 91, 95]. Cependant, elles présentent des solutions propriétaires se basant sur une logique conçue à la main et n'utilisant pas les IPs disponibles. Dans la plupart des cas, ces architectures SIMD sont destinées pour une application bien précise. Notre système mppSoC se base sur un design moins coûteux en utilisant les IPs disponibles spécifiquement les IP processeurs et il est flexible et programmable pouvant s'adapter à différentes applications de TSS.

Comparé à d'autres implémentations, mppSoC présente de bonnes performances. Le tableau 5.4 compare entre mppSoC et une architecture 2D systolique à base de FPGA [107], en terme de temps d'exécution pour l'application de convolution. La configuration SIMD utilise des processeurs NIOS répliqués et intègre un réseau mpNoC à base de crossbar et un réseau de voisinage en topologie tore. Comparée à l'architecture systolique, l'architecture SIMD achève de meilleurs résultats bien qu'elle intègre un plus petit nombre de PEs (32 comparé à 49 utilisés dans l'architecture systolique). Un autre point très important à considérer est la manière d'implémentation des systèmes cités. En effet, l'architecture systolique est conçue à la main et nécessite un long temps de développement. Par opposition, notre configuration mppSoC a été générée rapidement en utilisant l'outil de génération de mppSoC. Elle se base tout de même sur l'IP processeur disponible NIOS et sur une méthodologie de réplication simple réduisant les coûts de développement. L'aspect programmable rend le système flexible pouvant s'adapter à différentes applications.

D'autres solutions ont été proposées afin d'augmenter les performances du processeur.

TABLE 5.5 – Comparaison entre configurations SIMD et accélérations C2H

| | Algorithme | Accélération | Freq. (MHz) |
|---|---|---|---|
| SIMD | RGB à YIQ | 29x | 88.47 |
| C2H | RGB à YIQ | 39.9x | 110 |
| SIMD | Convolution | 15.9x | 88.47 |
| C2H | Convolution | 13.3x | 95 |

TABLE 5.6 – Comparaison entre configuration mppSoC et autres systèmes embarqués

| | Architecture | Processeur | Fréquence MHz | Puissance W | Performance Iter./cycle |
|---|---|---|---|---|---|
| VIRAM | vectoriel | VIRAM | 200 | 2.0 | 8.5 |
| PowerPC | RISC | MPC7455 | 1000 | 21.3 | 7 |
| mppSoC à base de NIOS (32 PEs) | SIMD | NIOSII | 88.47 | 1.26 | 12 |

À titre d'exemple, notons le logiciel de compilation de matériel C2H fourni par Altera, décrit dans [4], qui permet d'accélérer des boucles écrites en C. Le tableau 5.5 compare entre des benchmarks compilés avec C2H [45] et d'autres exécutés sur des configurations mppSoC se composant de 32 PE-NIOS. Nous remarquons clairement qu'une solution SIMD programmable est aussi efficace qu'une solution purement matérielle pouvant fournir de comparables résultats avec en plus l'avantage de flexibilité.

Le tableau 5.6 compare entre mppSoC à base de NIOS et l'architecture vectorielle VIRAM et le processeur super-scalaire PowerPC [58]. À première vue nous constatons que le système SIMD présente la plus faible fréquence mais aussi la moindre consommation de puissance dû à sa simplicité. La performance est mesurée en nombre d'itérations par cycle pour l'algorithme de conversion RGB à YIQ. Notre système a une performance plus grande que VIRAM et PowerPC par un facteur de 1.5x et 1.7x respectivement. Pour des benchmarks purement data parallèles, nous voyons clairement qu'une architecture SIMD est plus performante qu'une architecture vectorielle ou super-scalaire. Du fait de son extensibilité, le système mppSoC peut encore fournir de meilleures résultats en augmentant le nombre de PEs. Bien que le système mppSoC n'est ni optimisé pour une moindre consommation de puissance, ni pour une grande fréquence d'horloge, il présente avec 32 PEs une consommation de puissance minimale par rapport aux autres architectures tel que montré dans le tableau 5.6. Optimiser plus l'architecture de mppSoC permet alors d'obtenir encore de meilleurs performances. En comparant la manière de programmation, notre système mppSoC à base de NIOS est facile à programmer puisque nous n'avons pas changé le jeu d'instruction ni le compilateur du processeur. Alors que dans le cas de VIRAM par exemple, le compilateur a été modifié afin de tenir compte des instructions vectorielles ; ce qui est très difficile à développer.

Nous comparons finalement notre système mppSoC avec une implémentation purement logicielle sur un processeur Pentium 4 à 1.6 GHz et à deux implémentations matérielles optimisées pour calculer la convolution [85]. Le tableau 5.7 compare entre ces différentes implémentations en termes de temps d'exécution et de dissipation d'énergie exprimée en mWatt/MHz. L'application considérée est une convolution 3x3 sur une image de pixels 8-

TABLE 5.7 – Comparaison entre configuration mppSoC et autres systèmes embarqués

| | Pentium 4 | [85] | [84] | mppSoC |
|---|---|---|---|---|
| Dissipation d'énergie (mW/MHz) | 51.25 | 102.1 | 19 | 14.31 |
| Temps d'exécution (ms) | 26 | 4.6 | 13 | 5.1 |

bits et de taille 1024x1024. Le processeur Pentium consomme une puissance de l'ordre de 82 Watts. La configuration mppSoC choisie est celle décrite dans la sous section 5.4.4.2. Dans cette comparaison, nous n'avons pas pu comparer les ressources logiques consommées sur l'FPGA entre mppSoC et les implémentations matérielles vu que ces dernières se basent sur des FPGA Xilinx. Nous remarquons que mppSoC dissipe le moins d'énergie comparé aux autres systèmes. Une architecture SIMD est aussi 5.41 fois plus rapide par rapport à une implémentation purement logicielle sur un processeur ayant une fréquence de 1.6 GHz, et légèrement lente (1.1 fois plus lente) par rapport à une implémentation matérielle optimisée [85] (mppSoC est plus performante qu'une implémentation matérielle non optimisée [84]). Ceci prouve la performance de mppSoC. Bien que le système SIMD est un peu lent qu'un système matériel câblé, le temps de développement pour une application utilisant mppSoC est considérablement plus bas puisque les concepteurs n'auront qu'à programmer leur application sans se préoccuper des détails de bas niveau de l'architecture. L'approche SIMD comporte également beaucoup d'avantages étant donné que mppSoC peut être utilisé pour différentes applications de TSS, ce qui résulte potentiellement à économiser les efforts importants de développement. Enfin, en utilisant mppSoC, nous pouvons adapter la configuration SIMD aux besoins de l'application et intégrer les éléments nécessaires pour l'accélération de l'exécution. Tout ceci se base sur une chaîne de génération réduisant encore le temps de conception et facilitant la tâche aux concepteurs.

Ces différentes comparaisons présentées confirment les avantages d'un système SIMD programmable implémenté à base de FPGA. Nous avons aussi montré que l'utilisation des processeurs soft-core est efficace pour une telle architecture particulière.

## 5.6 Conclusion

Ce chapitre a décrit le flot de conception mppSoC et a présenté la chaîne de génération VHDL développée. Une étude de cas de l'usage de l'outil de conception de mppSoC pour la modélisation et génération de configurations SIMD a été exposée. Cette étude a été menée avec des applications de traitement d'images. Nous avons alors vu que grâce à l'outil développé, nous pouvons facilement générer une configuration au choix. Cet outil facilite de même l'exploration de mppSoC pour une application donnée. En effet, il suffit de modifier le modèle UML de mppSoC et automatiquement le code VHDL synthétisable peut être régénéré. La modélisation d'une configuration mppSoC est indépendante de tout détail d'implémentation, dans la mesure où la saisie d'un modèle est réalisée dans un langage unifié. Notre flot de conception est donc utilisable même par des non-spécialistes des architectures parallèles sur puce. Enfin, nous avons comparé les performances de mppSoC avec d'autres systèmes et architectures à base de FPGA. Des facteurs d'accélération importants ont été mesurés et une bonne performance a été démontrée. Notre système mppSoC comble

un large écart entre les architectures à usage générale et les architectures spécialisées dédiées aux applications de traitement d'images.

# Chapitre 6

# Conclusion et perspectives

6.1 Bilan . . . . . 146
6.2 Perspectives . . . . . 147

## 6.1 Bilan

Les travaux présentés dans ce document s'intéressent au développement d'un système SIMD sur puce flexible et générique, nommé mppSoC (*massively parallel processing System on Chip*), pouvant s'adapter aux applications de traitement de signal intensif systématique. Il résulte de ces travaux la proposition d'un flot de conception automatisé. Sa mise en œuvre a nécessité des contributions dans différents domaines tels que l'ingénierie dirigée par les modèles, la conception logicielle/matérielle, l'intégration d'IPs et la validation d'architectures sur FPGA. Afin de réduire le temps de mise sur le marché et faciliter la conception des architectures massivement parallèles sur puce, nous avons choisi la conception par assemblage d'IPs. Cette manière de conception permet de résoudre le problème de l'augmentation constante de la complexité de conception. De nos travaux de thèse, il résulte la proposition d'une chaîne de génération automatisée et dirigée par les modèles. Cette chaîne permet la génération, à partir d'un modèle UML haut niveau d'une configuration mppSoC, du code VHDL synthétisable destiné à être directement simulé par les outils de simulation ou compilé et prototypé sur FPGA à l'aide des outils de synthèse et compilation.

Nos travaux ont commencé par l'étude de la problématique de la complexité croissante de la conception de SoC et au compromis performance/flexibilité à résoudre. Ce contexte touche particulièrement les architectures SIMD, sujet de notre travail. À travers cette étude, nous avons constaté que la plupart des architectures SIMD sont destinées pour une application bien précise ; ajoutons de plus leurs spécialisations et difficultés de conception. Ciblant un large éventail d'applications data parallèles, une architecture SIMD flexible et programmable s'avère nécessaire. Toutefois, une manière de conception efficace et rapide se révèle très utile.

Pour résoudre cette problématique, les premières contributions présentées dans cette thèse ont consisté à définir un modèle mppSoC générique et flexible. Les paramètres et caractéristiques de mppSoC ont été justifiés. Le modèle proposé peut s'adapter facilement aux exigences de différentes applications, particulièrement dans le domaine de traitement d'images. Le modèle mppSoC peut être configuré selon l'application.

Par la suite, nous nous sommes intéressés à développer une démarche de conception rapide et modulaire pour mppSoC. Cette conception se base alors sur un assemblage de composants ou IPs. Des IPs standard peuvent être utilisés, d'autres IPs ont été conçus pour mppSoC et peuvent être réutilisés selon les besoins. Au terme de ce travail, une bibliothèque mppSoCLib a été mise en place. Elle est très utile pour implémenter directement une configuration mppSoC et permet d'alléger la tâche de conception des composants aux concepteurs.

Après ces deux premières étapes, nous avons présenté dans la troisième étape de notre thèse, la mise en place d'une chaîne de génération de configuration mppSoC qui a été intégrée dans l'environnement GASPARD. Cette chaîne a pour objectif de générer automatiquement le code VHDL synthétisable d'un modèle haut niveau d'une configuration mppSoC. Nous avons développé une transformation de type modèle-vers-texte qui permet de générer le code VHDL représentant la configuration mppSoC modélisée. Ce code peut alors être simulé ou implémenté sur FPGA en se servant des outils de simulation et synthèse disponibles. Cet outil est très bénéfique et permet de réduire considérablement le temps de conception de mppSoC avec une augmentation de la productivité.

Enfin, dans la dernière partie de notre thèse, nous avons illustré l'usage de la chaîne de génération mppSoC afin de choisir la configuration adéquate à une application donnée. Les

configurations mppSoC générées ont été testées dans un contexte applicatif réel de traitement d'images à base de FPGA. Cela a permis de valider l'outil développé ainsi que les configurations mppSoC implémentées. Cette étude de cas a aussi été l'occasion de mesurer les performances de mppSoC et déduire l'efficacité du système vis à vis des applications testées. Une comparaison menée entre mppSoC et d'autres systèmes a montré les performances suffisantes et l'efficacité de mppSoC.

## 6.2 Perspectives

Le travail effectué dans cette thèse peut être poursuivi suivant de nombreuses directions, nous en présentons ici quelques-unes :

### Exploration des configurations mppSoC

Dans le dernier chapitre de la thèse, nous avons exposé la chaîne de génération de mppSoC à partir d'une modélisation haut niveau. Cette chaîne permet de générer du code VHDL synthétisable qui peut être simulé ou implémenté sur FPGA. À travers une mesure de performances nous déduisons alors l'efficacité de la configuration modélisée. Si nous voulons tester une autre configuration, nous n'avons qu'à modifier le modèle de haut niveau. En effet, de simples modifications dans le modèle de départ suivies d'une phase de génération automatique du code permettent de tester différentes alternatives. À titre d'exemple, pour changer le nombre de processeurs, il suffit de modifier le paramètre correspondant dans le modèle de départ. Actuellement toutes ces spécifications sont ajoutées manuellement par le concepteur. Une perspective de ce travail est la définition d'un outil d'exploration automatique autour de la génération de la configuration mppSoC la plus adéquate pour une application donnée. Il permet alors de parcourir l'espace de solutions à un haut niveau d'abstraction et effectuer des simulations de performance afin de choisir la meilleure configuration. Cette étape nécessite la spécification des critères de performances à tenir en compte lors du choix du système final (temps d'exécution de l'application, consommation d'énergie, ressources logiques, taille mémoire, etc.).

### Vers un modèle d'exécution multi-SIMD

Le modèle mppSoC que nous avons proposé est un modèle purement SIMD homogène, inspiré des systèmes SIMD classiques à savoir la MasPar. L'aspect totalement synchrone du système pose des problèmes surtout dans l'intégration de configurations assez grandes comportant un grand nombre de PEs. Il serait intéressant d'étendre le modèle pour supporter l'exécution multi-SIMD. Un modèle multi-SIMD est plus efficace et peut supporter une large classe d'applications. Dans ses travaux, Wittaya Chantamas a présenté un modèle de programmation parallèle appelé MASC (*(Multiple ASsociative Computing*) qui repose sur un modèle multi-SIMD [19]. Cependant, ce modèle n'est pas abordé dans le contexte des systèmes sur puce. Il serait alors opportun de revoir l'efficacité d'un tel modèle bénéficiant des évolutions technologiques des SoC.

Nous pouvons voir le système comme un système localement synchrone (au sein du groupe SIMD) globalement asynchrone (entre les réseaux SIMD). Cette approche multi-SIMD offre une autonomie d'opération et rend le système plus général pas uniquement

dédié aux applications régulières. Un système multi-SIMD est un système hétérogène. Potentiellement, cette approche peut être à même d'assurer une efficacité énergétique [111].

La consommation de puissance est un autre critère très important à considérer pour les systèmes performants sur puce. En effet, le système mppSoC conçu est assez simple et n'est ni optimisé pour une moindre consommation de puissance, ni optimisé pour une grande fréquence d'horloge. Des techniques d'optimisation peuvent être alors ajoutées dans le code VHDL afin de maximiser les performances de mppSoC. À tire d'exemple, la consommation de puissance du système peut être réduite si on pense à intégrer un module de gestion de puissance qui permet de désactiver les PEs qui sont en état inactif lors de l'exécution d'une instruction parallèle. Ce contexte est particulièrement valable dans le cas des instructions conditionnelles (*if-then-else*). Les communications de voisinage doivent être de même accélérées. Nous pouvons penser à piloter ces communications par un module matériel de contrôle au lieu d'être guidées par les instructions. La reconfiguration dynamique est aussi un autre aspect à étudier pour adapter les communications suivant l'exécution de l'application.

# Publications personnelles

1. M. Baklouti, S. Le Beux, Ph. Marquet, M. Abid and JL. Dekeyser, *Implémentation par Assemblage d'IP d'Architectures Massivement Parallèles*, in : Proc. of Journées Scientifiques des jeunes chercheurs en Génie Électrique et Informatique (GEI), Sousse, Tunisie, 2008.
2. M. Bakouti, S. Le Beux, Ph. Marquet, M. Abid and JL. Dekeyser, *Implementation of a Simple Massively Parallel Processor System on FPGA : Case Study*, in : Proc. of the 1st International Conference on Embedded Systems and Critical Applications (ICESCA), Tunis, Tunisia, 2008.
3. M. Bakouti, Ph. Marquet, M. Abid and JL. Dekeyser, *A design and an implementation of a parallel based SIMD architecture for SoC on FPGA*, in : Proc. of Conference on Design and Architectures for Signal and Image processing (DASIP), Bruxelles, Belgium, 2008.
4. M. Bakouti, Ph. Marquet, M. Abid and JL. Dekeyser, *The impact of Interconnection Networks in a Massively Parallel FPGA Architecture on a Parallel Reduction Algorithm*, in : Proc. of the 3rd IEEE International Design and Test Workshop (IDT), Monastir, Tunisia, 2008.
5. M. Bakouti, Ph. Marquet, M. Abid and JL. Dekeyser, *Study and Integration of a Parametric Neighbouring Interconnection Network in a Massively Parallel Architecture on FPGA*, in : Proc. of the 7th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA), Rabat, Morocco, May 2009.
6. R. Tligue, Y. Aydi, M. Baklouti, M. Abid and JL. Dekeyser, *The design methodology and the implementation of MPSOC based on Delta MINs on FPGA*, in : Proc. of the 21th IEEE International Conference on Microelectronics (ICM), Marrakech, Morocco, December 2009.
7. A. Helmi, M. Baklouti and M. Abid, *SoC based Parallel Architecture : Case of Study*, in : Proc. of the 10th International Conference on Sciences and Techniques of Automatic control and computer engineering (STA'09), Hammamet, Tunisia, December 2009.
8. M. Bakouti, Ph. Marquet, M. Abid and JL. Dekeyser, *Reconfigurable Communication Networks in a Parametric SIMD Parallel System on Chip*, in : Proc. of the 6th International Symposium on Applied reconfigurable Computing (ARC), Bangkok, Thailand, March 2010.
9. M. Bakouti, *Parametric Framework for Massively Parallel System on Chip*, in : EuroDocInfo10, Valenciennes, France, January 2010.
10. M. Bakouti, Y. Aydi, Ph. Marquet, M. Abid and JL. Dekeyser, *Scalable mpNoC for Massively Parallel Systems - Design and Implementation on FPGA*, in : Journal of Systems Architecture (JSA), 56(2010), 2010, pp. 278-292.

11. M. Bakouti, Ph. Marquet, M. Abid and JL. Dekeyser, *IP based configurable SIMD massively parallel SoC*, in : PhD Forum of 20th International Conference on Field Programmable Logic and Applications (FPL), Milano, Italy, August 2010.
12. Y. Aydi, M. Bakouti, Ph. Marquet, M. Abid and JL. Dekeyser, *A Design Methodology of MIN-Based Network for MPPSoC on Reconfigurable Architecture*, Chap. x, IGI-Global : Reconfigurable Embedded Control Systems : Applications for Flexibility and Agility. To appear.

# Bibliographie


[1] F. Abbes, E. Casseau, M. Abid, P. Coussy, and J.B. Legoff. IP integration methodology for SoC design. In *International Conference on Microelectronics ICM*, pages 343–346, 2004.

[2] A. Abbo, R. Kleihorst, V. Choudhary, L. Sevat, P. Wielage, S. Mouy, B. Vermeulen, and M. Heijligers. XETAL-II : A 107 GOPS, 600 mW Massively Parallel Processor for Video Scene Analysis. *IEEE Journal of Solid-State Circuits*, 43(1) :192–201, Jan 2008.

[3] Altera. Avalon Interface Specifications. Document version 1.2, April 2009. http://www.altera.com/literature/manual/mnl_avalon_spec.pdf.

[4] Altera. NIOS II C2H Compiler User Guide. 2006. www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf.

[5] M. Auguin. Co-Conception de Systèmes Spécialisés sur Composant. In *École Thématique sur les Systèmes Enfouis*, Nov 2000. http ://www.ens-lyon.fr/ARCHI05/ecole2000/cours/auguin.pdf.

[6] M. Baklouti, Y. Aydi, Ph. Marquet, M. Abid, and JL. Dekeyser. Scalable mpNoC for Massively Parallel Systems - Design and Implementation on FPGA. *Journal of Systems Architecture (JSA)*, 56 :278–292, 2010.

[7] M. Baklouti, Ph. Marquet, M. Abid, and JL. Dekeyser. A design and an implementation of a parallel based SIMD architecture for SoC on FPGA. In *Proc. of Conference on Design and Architectures for Signal and Image processing (DASIP)*, Bruxelles, Belgium, 2008.

[8] M. Baklouti, Ph. Marquet, JL. Dekeyser, and M. Abid. Reconfigurable Communication Networks in a Parametric SIMD Parallel System on Chip. In *the $6^{th}$ international Symposium on Applied Reconfigurable Computing (ARC'10)*, Bangkok, Thailand, March 2010.

[9] M. Bakouti, Ph. Marquet, M. Abid, and JL. Dekeyser. Study and Integration of a Parametric Neighbouring Interconnection Network in a Massively Parallel Architecture on FPGA. In *Proc. of the 7th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA)*, Rabat, Morocco, May 2009.

[10] J. L. Basille. SYMPATI's project. In *IEEE International Conference on Systems Engineering*, pages 515–518, Kobe, Japan, 1992.

[11] L. Bengtsson, K. Nilsson, and B. Svensson. A High-Performance Embedded Massively Parallel Processing System. In *First International Conference on Massively Parallel Computing Systems*, Ischia, Italy, 1994.

[12] R. A. Bergamaschi and W. R. Lee. Designing Systems-on-Chip Using Cores. In *Proc. of the 37$^{th}$ Design Automation Conference*, 2000.

[13] T. Blank. The MasPar MP-1 architecture. In *Proc. of the IEEE Compcon Spring*, pages 20–24, 1990.

[14] Ph. Bonnot, F. Lemonnier, G. Edelin, G. Gaillat, O. Ruch, and P. Gauget. Definition and SIMD implementation of a multi-processing architecture approach on FPGA. In *Proc. of Design Automation and Test in Europe DATE08*, 2008.

[15] P. Boulet. Contributions aux environnements de programmation pour le calcul intensif. Habilitation à diriger des recherches, Université des Sciences et de Technologie de Lille, Décembre 2002.

[16] P. Boulet. Array-OL Revisited, Multidimensional Intensive Signal Processing Specification. Technical Report RR-6113, February 2007.

[17] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. Peter Graf. A Massively Parallel FPGA-based Coprocessor for Support Vector Machines. In *Proc. of the 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pages 115–122, 2009.

[18] The Cell project at IBM Research. http ://www.research.ibm.com/cell/. 2010.

[19] W. Chantamas. *A multiple associative computing model to support the execution of data parallel branches using the manager-worker paradigm*. PhD thesis, Kent State University, 2009.

[20] C. Chavet. *Synthèse automatique d'interfaces de communication matérielles pour la conception d'applications du domaine du traitement du signal*. PhD thesis, Université de Bretagne Sud, 2007.

[21] Y-K. Chen. Multimedia Signal Processing on Platforms with Multiple Cores. In *IEEE International Conference on Multimedia & Expo (ICME)*, Singapore, 2010.

[22] C.C. Cheng and al. iVisual : An Intelligent Visual Sensor SoC With 2790fps CMOS Image Sensor and 205 GOPS/W Vision Processor. *IEEE Journal of Solid-State Circuits*, 44(1) :127–135, 2009.

[23] P. Chou, R. Ortega, K. Hines, K. Partridge, and G. Borriello. IPCHINOOK : An Integrated IP-based Design Framework for Distributed Embedded Systems. In *Proc. of the 36$^{th}$ Design Automation Conference*, pages 44–49, 1999.

[24] J-J. Clar. *Développement d'applications parallèles pour un système multiprocesseur expérimental*. PhD thesis, Université du Québec à Chicoutimi, 2002.

[25] F. Clermidy, D. Varreau, and D. Lattard. A NoC-Based Communication Framework for Seamless IP Integration in Complex Systems. In *IPSoC*, 2005.

[26] T. Collette, C. Gamrat, D. Juvin, J-F. Larue, L. Letellier, M. Peythieux, R. Schmit, and M. Viala. Symphonie calculateur massivement parallèle : modélisation et réalisation. *Traitement du Signal*, 14(6) :637–644, 1997.

[27] Thinking Machines Corporation. Connection Machine Model CM-2 Technical Summary. Technical report, 1990.

[28] DaRT. Graphical Array Specification for Parallel and Distributed Computing (GASPARD-2). 2010. http://www.gaspard2.org/.

[29] JL. Dekeyser, S. Le Beux, and Ph. Marquet. Une approche modèle pour la conception conjointe de systèmes embarqués hautes performances dédiés au transport. In *Workshop International : Logistique & Transport*, 2007.

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

[31] S. Duquennoy, S. Le Beux, Ph. Marquet, S. Meftali, and JL. Dekeyser. MpNoC design : modeling and simulation. In *Proc. of the* $15^{th}$ *IP Based SoC Design Conference, IP/SOC*, 2006.

[32] EEMBC. The Embedded Microprocessor Benchmark Consortium. http://www.eembc.org/home.php.

[33] D. Gajski et al. *High-Level Synthesis : Introduction to Chip and System Design*. Kluwer Academic Publishers, 1991.

[34] M. Dulong et T. M. Bernard. Recherche d'une exploitation énergétique optimale des ressources de calcul dans un système de vision sur puce. In *Colloque GRETSI*, Sep 2009.

[35] J. Eyre and J. Bier. The Evolution of DSP Processors. In *IEEE Signal Processing Magazine*, 2000.

[36] P. Feautrier. Parallélisation Automatique. Technical report, September 2008.

[37] A. L. Fisher. Scan Line Array Processors for Image Computation. In *13th annual international symposium on Computer architecture*, pages 338–345, Tokyo, Japan, 1986.

[38] M. J. Flynn. Very high-speed computing systems. *Proc. of the IEEE*, 54(12) :1901–1909, December 1966.

[39] T. Fountain. *Processor Arrays Architecture and Applications*. Academic Press, 1987.

[40] T. J. Fountain, K. N. Matthews, and M. J. B. Duff. The CLIP7A Image Processor. *IEEE Trans. on pattern analysis and machine intelligence*, 10(3) :310–319, 1988.

[41] V. Garg and D. E. Schimmel. CCSIMD : a Concurrent Communication and Computation Framework for SIMD Machines. *Parallel Computer Routing and Communication*, 1417 :55–64, 1998.

[42] V. Garg and D. E. Schimmel. Hiding Communication Latency in Data Parallel Applications. In *Parallel and Distributed Processing Symposium*, pages 18–23, 1998.

[43] J-L. Gaudiot. The Walls of Computer Design, (abstract). *Lecture Notes in Computer Science*, 4330 :1, 2006.

[44] D. Ginhac, J. Dubois, M. Paindavoine, and B. Heyrman. An SIMD Programmable Vision Chip with High-Speed Focal Plane Image Processing. *EURASIP Journal on Embedded Systems*, 2008(Article ID 961315), 2008.

[45] J. Hamblen. SoC, PSoC, & SoPC Design. 2010. http://users.ece.gatech.edu/~hamblen/489X/ppt/SOPC.

[46] F. Hanning, H. Dutta, A. Kupriyanov, J. Teich, R. Schaer, S. Siegel, R. Merker, R. Keryell, B. Pottier, and O. Sentieys. Co-Design of Massively Parallel Embedded Processor Architectures. In *Proc. of Reconfigurable Communication-centric Systems on Chip, ReCoSoC'05*, 2005.

[47] W. Daniel Hillis. *The Connection Machine*. MIT Press, 1985.

[48] R. Hoare, S. Tung, and K. Werger. An 88-Way Multiprocessor within an FPGA with Customizable Instructions. In *Proc. of the* $18^{th}$ *International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

[49] M. Hohenauer, F. Engel, R. Leupers, G. Ascheid, and H. Meyr. A SIMD Optimization Framework for Retargetable Compilers. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(1), 2009.

[50] H.Singh, G.Lu, M.Lee, E. Filho, and R. Maestre. Design and Implementation of the MorphoSys Reconfigurable Computing Processor. *VLSI signal Processing systems for signal, image and video Technology*, 24 :147–164, 2000.

[51] ITRS. International Technology Roadmap for Semiconductors, 2009 Edition Design.

[52] ITRS. International Technology Roadmap for Semiconductors, 2009 update system drivers. http://www.itrs.net/links/2009ITRS/2009Chapters_2009Tables/2009_SysDrivers.pdf.

[53] J. Jacobs, L. Engelen, J. Kuper, and G.J.M. Smit. Image Quantisation on a Massively Parallel Embedded Processor. In *Symposium on Systems, Architectures, Modeling, and Simulation, SAMOS*, pages 139–148, 2007.

[54] R. Keryell. D'un Petit Ordinateur Massivement Parallèle : Synthèse du Projet POMP. *Techniques et Sciences Informatiques*, 12(6) :715–743, 1993.

[55] B. Khailany, W.J. Dally, U.J. Kapasi, P. Mattson, J. Namkoong, J.D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine : media processing with streams. *Micro, IEEE*, 21(2) :35–46, 2001.

[56] K. Kim and al. A 125 GOPS 583 mW Network-on-Chip Based Parallel Processor With Bio-Inspired Visual Attention Engine. *IEEE Journal of Solid-State Circuits*, 44(1) :136–147, 2009.

[57] 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, Dec 2006.

[58] C. Kozyrakis and D. Patterson. Vector Vs. Super-scalar and VLIW Architectures for Embedded Multimedia Benchmarks. In *Proc. of 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 283–293, 2002.

[59] S. Kyo and S. Okazaki. IMAPCAR : A 100 GOPS In-Vehicle Vision Processor Based on 128 Ring Connected Four-Way VLIW Processing Elements. *Journal of Signal Processing Systems*, 2008.

[60] S. Kyo, S. Okazaki, and I. Kuroda. An Extended C Language and a SIMD Compiler for Efficient Implementation of Image filters on Media Extended Micro-processors. In *Proc. of Advanced Concepts for Intelligent Vision Systems, Acivs*, 2003.

[61] S. Le Beux, Ph. Marquet, O. Labbani, and JL. Dekeyser. FPGA Implementation of Embedded Cruise Control and Anti-Collision Radar. In *9th Euromicro conference on digital system design, DSD*, Dubrovnik, Croatia, August 2006.

[62] A. Legrand and Y. Robert. *Algorithmique Parallèle – Cours et exercices corrigés*. Dunod, 2003.

[63] S.Y.C. Li, G.C.K. Cheuk, K.H. Lee, and Ph.H.W. Leong. FPGA-based SIMD Processor. In *Proc. of the* $11^{th}$ *Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM*, 2003.

[64] X. Li and O. Hammami. An Automatic Design Flow for Data Parallel and Pipelined Signal Processing Applications on Embedded Multiprocessor with NoC : Application to Cryptography. *International Journal of Reconfigurable Computing*, 2009.

[65] G-H. Lin, Y-N. Wen, X-L. Wu, S-J. Chen, and Y-H. Hu. Design of a SIMD Multimedia SoC Platform. In *Proc. of the IEEE International SOC Conference (SOCC)*, 2007.

[66] E. Lindholm and S. Oberman. The NVIDIA GeForce 8800 GPU, 2007. http://www.hotchips.org/archives/hc19/2_Mon/HC19.02/HC19.02.01.pdf.

[67] J. Makino, K. Hiraki, and M. Inaba. GRAPE-DR : 2-Pflops massively-parallel computer with 512-core, 512-Gflops processor chips for scientific computing. In *the ACM/IEEE Conference on Supercomputing (SC)*, pages 1–11, 2007.

[68] G. Malige and Ph. Vandewiele. Architecture parallèle pour le traitement de l'image. Technical report, 2004.

[69] R. Manduchi, G. M. Cortelazzo, and G. A. Mian. Multistage sampling structure conversion of video signals. *IEEE Transactions on circuits and systems for video technology*, 3(5) :325–340, 1993.

[70] UML MARTE. The UML Profile for MARTE : Modeling and Analysis of Real-Time and Embedded Systems. http://www.omgmarte.org/.

[71] E. Martin, H. Dubois, O. Sentieys, and J. L. Philippe. Définition de mesures objectives de performances pour la mise en œuvre parallèle d'algorithmes de traitement d'image. In *Actes de Colloques GRETSI, Groupe d'Etudes du Traitement du Signal et des Images*, 1991.

[72] M. Mehrara, T. Jablin, D. Upton, D. August, K. Hazelwood, and S. Mahlke. Multicore Compilation Strategies and Challenges : An overview of parallelism and compiler technology. *IEEE Signal Processing Magazine*, pages 55–63, 2009.

[73] G. E. Moore. Cramming more components onto integrated circuits. *Proc. of the IEEE*, 86(1) :82–85, 1998.

[74] NCR. *Processor Gapp – Geometric Arithmetic Parallel Processor*. NCR Corporation, 1984.

[75] J. Nickolls. The Design of the MasPar MP-1, A Cost-Effective Massively Parallel Computer. In *Proc. of the IEEE Compcon Spring*, 1990.

[76] OCP, I.P. : Open core protocol specification 2.0, 2003. http://www.ocpip.org/.

[77] OMG. MDA Guide Version 1.0.1, 2003.

[78] OMG. UML 2.0 OCL Specification, 2003. http://www.lri.fr/~wolff/teach-material/2008-09/IFIPS-VnV/UML2.0OCL-specification.pdf.

[79] OMG. UML Profile for MARTE : Modeling and Analysis of Real-Time Embedded Systems, version 1.0, 2009. http://www.omg.org/spec/MARTE/1.0/PDF/.

[80] OpenCores. http://opencores.org/.

[81] OpenCores. miniMIPS : :Overview. http://opencores.org/project,minimips.

[82] OpenCores. OR1200 OpenRISC processor. http://opencores.org/openrisc,or1200.

[83] J. H. Patel. Performance of processor-memory interconnections for multiprocessors. *IEEE. Transactions Computer*, 1981.

[84] S. Perri, M. Lanuzza, P. Corsonello, and G. Cocorullo. SIMD 2-D convolver for fast FPGA-based image and video processors. In *Proc. of the Military and Aerospace Programmable Logic Devices (MAPLD) International Conference*, 2003.

[85] S. Perri, M. Lanuzza, P. Corsonello, and G. Cocorullo. A high-performance fully reconfigurable FPGA-based 2D convolution processor. *J. Microprocessors and Microsystems*, 29 :381–391, 2005.

[86] M. Petouris, A. Kalantzopoulos, and E. Zigouris. An FPGA-based Digital Camera System Controlled from an LCD Touch Panel. In *Proc. of the 9th International Symposium on Signals, Circuits & Systems, ISSCS*, 2009.

[87] S. Piskorski, L. Lacassagne, and D. Etiemble. Instructions SIMD flottantes 16 bits pour réduire la consommation dans les processeurs embarqués à jeux d'instructions spécialisables. In *SYMPA 06*, 2006.

[88] S. Piskorski, L. Lacassagne, and D. Etiemble. IPLG : Un outil pour la fusion d'opérateurs en Traitement d'Images. In *SympA'13*, September 2009.

[89] W. Raab, J. Berthold, U. Hachmann, H. Eisenreich, and J-U. Schluessler. Low Power Design of the X-GOLD SDR 20 Baseband Processor. In *Proc. of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 792–793, 2010.

[90] U. Ramacher. Software-defined radio prospects for multistandard mobile phones. *Computer*, 40(10) :62–69, 2007.

[91] R.L. Rosas, A. de Luca, and F.B. Santillan. SIMD Architecture for Image Segmentation using Sobel Operators Implemented in FPGA Technology. In *Proc. of the* $2^{nd}$ *International Conference on Electrical and Electronics Engineering ((ICEEE '05)*, 2005.

[92] K. Sankaralingam, S.W. Keckler, W.R. Mark, and D. Burger. Universal Mechanisms for Data-Parallel Architectures. In *Proc. of the* $36^{th}$ *International Symposium on Microarchitecture (MICRO-36)*, 2003.

[93] M. Sayed, W. Badawy, and G. Jullien. Towards an H.264/AVC HW/SW Integrated Solution : An Efficient VBSME Architecture. *IEEE Transactions on Circuits and Systems*, 55(9) :912–916, September 2008.

[94] D. H. Schaefer. *The Massively Parallel Processor*, chapter History of the MPP, pages 1–5. The MIT Press, j. l. potter, editor edition.

[95] F. Schurz and D. Fey. A Programmable Parallel Processor Architecture in FPGA for Image Processing Sensors. In *Integrated Design and Process Technology, IDPT*, 2007.

[96] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee : a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3) :1–15, 2008.

[97] SHARC Processor Benchmarks. http://www.analog.com.

[98] J.H. Siegel, W.C. Nation, C.P. Kruskal, and L.M. Napolitano. Using the Multistage Cube Network Topology in Parallel Supercomputers. *Proc. IEEE*, 12(77) :1932–1953, 1989.

[99] The SPIRIT Consortium : Enabling Innovative IP Re-Use And Design Automation, 2007. http://www.spiritconsortium.org/.

[100] H. H. Taylor, D. Chin, and A.W. Jessup. An MPEG Encoder Implementation on the Princeton Engine Video Supercomputer. In *Data Compression Conference*, pages 420–429, 1993.

[101] Sandbridge Technologies. Sandblaster DSP. http://www.sandbridgetech.com/sandblaster_dsp.php.

[102] Teraflops Research Chip. http://techresearch.intel.com/articles/Tera-Scale/1449.htm.2010.

[103] Terasic. TRDB-D5M Terasic D5M Hardware Specification, 2008.

[104] Terasic. TRDB-LTM 4.3 Inch Digital Touch Panel Development Kit, 2007.

[105] R. Tligue, Y. Aydi, M. Baklouti, M. Abid, and JL. Dekeyser. The design methodology and the implementation of MPSOC based on delta MINs on FPGA. In *21th IEEE International Conference on Microelectronics, ICM'09*, Marrakech, Morocco, Dec 2009.

[106] J.G. Tong, I.D.L. Anderson, and M.A.S. Khalid. Soft-Core Processors for Embedded Systems. In *the* $18^{th}$ *International Conference on Microelectronics (ICM06)*, pages 170–173, 2006.

[107] C. Torres-Huitzil and M. Arias-Estrada. Real-time image processing with a compact FPGA-based systolic architecture. *Real-Time Imaging*, 10 :177–187, 2004.

[108] Virtual Socket Interface Alliance. http://www.vsi.org.2010.

[109] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C-C. Miao, J.F. Brown III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5) :15–31, 2007.

[110] M. Wilder. Ardbeg Vector Processor. http://www.fer.hr/_download/repository/Ardbeg-Vector-Processor-Oct-2008.pdf, 2008.

[111] D. H. Woo and H.-H. S. Lee. Extending Amdahl's law for energy-efficient computing in the many-core era. *IEEE Comput.*, 41(12) :24–31, 2008.

[112] D.H. Woo, J.B. Fryman, A.D. Knies, M. Eng, and H-H.S. Lee. POD : A parallel-on-die architecture. In $1^{st}$ *Annual Workshop on High Performance Embedded Computing, Lexington*, 2007.

[113] D.H. Woo, J.B. Fryman, A.D. Knies, and H-H.S. Lee. Chameleon : Virtualizing Idle Acceleration Cores of A Heterogeneous Multicore Processor for Caching and Prefetching. *ACM Transactions on Architecture and Code Optimization*, 7(1), 2010.

[114] P. Wu, A.E. Eichenberger, A. Wang, and P. Zhao. An Integrated Simdization Framework Using Virtual Vectors. In *Proc. of the* $19^{th}$ *annual ACM international conference on Supercomputing.*, pages 169–178, 2005.

[115] X. Xizhen and S. G. Ziavras. H-SIMD machine : configurable parallel computing for matrix multiplication. In *International Conference on Computer Design : VLSI in Computers and Processors*, pages 671–676, 2005.

[116] P. Yiannacouras, J.G. Steffan, and J. Rose. VESPA : portable, scalable, and flexible FPGA-based vector processors. In *Proc. of the 2008 international conference on Compilers, architectures and synthesis for embedded systems, CASES'08*, pages 61–70, 2008.

[117] B. Zerrouk. SMAL-X 31 une architecture parallèle de granularité fine pour le traitement d'images et l'émulation neuro-mimétique. Technical Report 1146, December 1989.

# Annexe A

# Implémentation VHDL de miniMIPS

### Extension d'instructions

```
1
2  -- Le type Record contenant le micro-code d'une instruction
3     type micro_instr_type is
4     record
5         op_mode : op_mode_type;     -- Mode de l'operation
6         op_code : bus6;             -- Code de l'operation
7         bra : std_logic;            -- instruction de branchement
8         link : std_logic;           -- Branchement a une adresse: l'adresse
9         -- de retour est stocke dans un registre
10        code_ual : alu_ctrl_type;   -- Code de l'operation pour l'ALU
11        op_mem : std_logic;         -- Operation memoire demandee
12        r_w : std_logic;            -- Selection Read/Write en memoire
13        mode : std_logic;           -- calcul de l'adresse a partir du PC
14        --courant ('1') ou l'operande 1 de l'ALU ('0')
15        mem_size: mem_size_type;    -- Taille de l'acces memoire
16        off_sel : off_sel_type;     -- Offset: PC(31..28) & Adresse & 00 || 0
17        --|| sgn\_ext(Imm) & 00 || sgn_ext(Imm)
18        exc_cause : exc_type;       -- Exception inconditionnelle a generer
19        cop_org1 : std_logic;       -- Registre Source 1: registre general
20        --si 0, registre du coprocesseur si 1
21        cop_org2 : std_logic;       -- Registre Source 2: registre general
22        --si 0, registre du coprocesseur si 1
23        cs_imm1 : std_logic;        -- Utilisation de l'operande immediat 1
24        --au lieu du banc de registre
25        cs_imm2 : std_logic;        -- Utilisation de l'operande immediat 2
26        --au lieu du banc de registre
27        imm1_sel : std_logic;       -- Origine de l'operande immediat 1, 0:0,
28        --1: shamt
29        imm2_sel : std_logic;       -- Origine de l'operande immediat 2,
30        --0: unsigned imm,1: signed imm
31        sl_reg_use2: std_logic;     -- Si le PE doit utiliser son registre
32        --comme op2
33        level : level_type;         -- Disponibilite de la donnee pour le
34        --bypass
35        ecr_reg : std_logic;        -- Ecriture du resultat dans un registre
36        bank_des : std_logic;       -- Selection du banc de registre:
```

```
37          --GPR si 0, coprocesseur si 1
38          scr_sel1 : rscr1_type;      -- Adresse du registre Source de op2:
39          --Rt, Rd
40          scr_sel2 : rscr2_type;      -- Adresse du registre Source de op2:
41          --Rt, Rd
42          des_sel  : rdest_type;      -- Destination du registre d'adresse:
43          --Rt, Rd, $31, $0
44          op_vector : std_logic;      -- 0 instruction normale,
45          --1 instruction vectorielle a executer par le PE
46      end record;
47
48      type micro_code_type is array (natural range <>) of micro_instr_type;
49      constant micro_code : micro_code_type :=
50      (
51  (OP_SPECIALP,"010100",'0','0', OP_SETB  ,'0','0','0',"000",OFS_PCRL,
52  IT_NOEXC,'0','1', '1','0', '1','0',0,LVL_EX , '1','1',D_RS,D_ST,D_ST,'1'),
53  --PSETB
54  (OP_SPECIALP,"011101",'0','0', OP_NOTB  ,'0','0','0',"000",OFS_PCRL,
55  IT_NOEXC,'0','1', '1','0', '0','0',0,LVL_EX , '1','1',D_RS,D_ST,D_ST,'1'),
56  --PNOTB
57  (OP_SPECIALP,"011100",'0','0', OP_ANDB  ,'0','0','0',"000",OFS_PCRL,
58  IT_NOEXC,'0','1', '0','0', '0','0',0,LVL_EX , '1','1',D_RS,D_ST,D_ST,'1'),
59  --PANDB
60  (OP_SPECIALP,"010101",'0','0', OP_MFB   ,'0','0','0',"000",OFS_PCRL,
61  IT_NOEXC,'0','1', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_ST,D_RT,'1'),
62  --PMFB
63  (OP_SPECIALP,"010110",'0','0', OP_MTB   ,'0','0','0',"000",OFS_PCRL,
64  IT_NOEXC,'0','1', '0','0', '0','0',0,LVL_EX , '1','1',D_RS,D_ST,D_ST,'1'),
65  --PMTB
66  (OP_SPECIALP,"011110",'0','0', OP_MB    ,'0','0','0',"000",OFS_PCRL,
67  IT_NOEXC,'0','1', '1','0', '1','0',0,LVL_EX , '1','1',D_RS,D_ST,D_ST,'1'),
68  --PMB
69  (OP_SPECIAL ,"100000",'0','0', OP_ADD   ,'0','0','0',"000",OFS_PCRL,
70  IT_NOEXC,'0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),
71  --ADD
72  (OP_NORMAL  ,"001000",'0','0', OP_ADD   ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
73  '0','0', '0','1', '0','1',Z,LVL_EX , '1','0',D_RS,D_RT,D_RT,'0'),--ADDI
74  (OP_NORMAL  ,"001001",'0','0', OP_ADDU  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
75  '0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_RT,'0'),--ADDIU
76  (OP_SPECIAL ,"100001",'0','0', OP_ADDU  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
77  '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--ADDU
78  (OP_SPECIAL ,"100100",'0','0', OP_AND   ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
79  '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--AND
80  (OP_NORMAL  ,"001100",'0','0', OP_AND   ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
81  '0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_RT,'0'),--ANDI
82  (OP_NORMAL  ,"000100",'1','0', OP_EQU   ,'0','0','1',"000",OFS_SESH,IT_NOEXC,
83  '0','0', '0','0', '0','0',0,LVL_DI , '0','0',D_RS,D_RT,D_RT,'0'),--BEQ
84  (OP_REGIMM  ,"000001",'1','0', OP_LPOS  ,'0','0','1',"000",OFS_SESH,IT_NOEXC,
85  '0','0', '0','1', '0','0',Z,LVL_DI , '0','0',D_RS,D_RT,D_RT,'0'),--BGEZ
86  (OP_REGIMM  ,"010001",'1','1', OP_LPOS  ,'0','0','1',"000",OFS_SESH,IT_NOEXC,
87  '0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_31,'0'),--BGEZAL
88  (OP_NORMAL  ,"000111",'1','0', OP_SPOS  ,'0','0','1',"000",OFS_SESH,IT_NOEXC,
89  '0','0', '0','1', '0','0',Z,LVL_DI , '0','0',D_RS,D_RT,D_RT,'0'),--BGTZ
```

```
(OP_NORMAL  ,"000110",'1','0', OP_LNEG ,'0','0','1',"000",OFS_SESH,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_DI , '0','0',D_RS,D_RT,D_RT,'0'),--BLEZ
(OP_REGIMM  ,"000000",'1','0', OP_SNEG ,'0','0','1',"000",OFS_SESH,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_DI , '0','0',D_RS,D_RT,D_RT,'0'),--BLTZ
(OP_REGIMM  ,"010000",'1','1', OP_SNEG ,'0','0','1',"000",OFS_SESH,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_31,'0'),--BLTZAL
(OP_NORMAL  ,"000101",'1','0', OP_NEQU ,'0','0','1',"000",OFS_SESH,IT_NOEXC,
'0','0', '0','0', '0','0',O,LVL_DI , '0','0',D_RS,D_RT,D_RT,'0'),--BNE
(OP_SPECIAL ,"001101",'0','0', OP_OUI  ,'0','0','0',"000",OFS_PCRL,IT_BREAK,
'0','0', '0','1', '0','0',Z,LVL_DI , '0','0',D_RS,D_RT,D_RT,'0'),--BREAK
(OP_SPECIAL ,"011010",'0','0', OP_DIV  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '0','0', '0','0',O,LVL_EX , '0','0',D_RS,D_RT,D_RT,'0'),--DIV
(OP_SPECIAL ,"011011",'0','0', OP_DIVU ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '0','0', '0','0',O,LVL_EX , '0','0',D_RS,D_RT,D_RT,'0'),--DIVU
(OP_NORMAL  ,"000010",'1','0', OP_OUI  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_DI , '0','0',D_RS,D_RT,D_RT,'0'),--J
(OP_NORMAL  ,"000011",'1','1', OP_OUI  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_31,'0'),--JAL
(OP_SPECIAL ,"001001",'1','1', OP_OUI  ,'0','0','0',"000",OFS_NULL,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--JALR
(OP_SPECIAL ,"001000",'1','0', OP_OUI  ,'0','0','0',"000",OFS_NULL,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_DI , '0','0',D_RS,D_RT,D_RT,'0'),--JR
(OP_NORMAL  ,"001111",'0','0', OP_LUI  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_RT,'0'),--LUI
(OP_NORMAL  ,"100000",'0','0', OP_OUI  ,'1','0','0',"000",OFS_SEXT,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_MEM, '1','0',D_RS,D_RT,D_RT,'0'),--LB
(OP_NORMAL  ,"110110",'0','0', OP_OUI  ,'1','0','0',"000",OFS_SLAV,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_MEM, '1','0',D_RS,D_RT,D_RT,'0'),--S_LBP
(OP_NORMAL  ,"111110",'0','0', OP_OP2  ,'1','1','0',"000",OFS_SLAV,IT_NOEXC,
'0','0', '0','0', '0','0',O,LVL_DI , '0','0',D_RS,D_RT,D_RT,'0'),--S_SBP
(OP_NORMAL  ,"110111",'0','0', OP_OUI  ,'1','0','0',"010",OFS_SLAV,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_MEM, '1','0',D_RS,D_RT,D_RT,'0'),--S_LWP
(OP_NORMAL  ,"111111",'0','0', OP_OP2  ,'1','1','0',"010",OFS_SLAV,IT_NOEXC,
'0','0', '0','0', '0','0',O,LVL_DI , '0','0',D_RS,D_RT,D_RT,'0'),--S_SWP
(OP_NORMAL  ,"100100",'0','0', OP_OUI  ,'1','0','0',"100",OFS_SEXT,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_MEM, '1','0',D_RS,D_RT,D_RT,'0'),--LBU
(OP_NORMAL  ,"100001",'0','0', OP_OUI  ,'1','0','0',"001",OFS_SEXT,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_MEM, '1','0',D_RS,D_RT,D_RT,'0'),--LH
(OP_NORMAL  ,"100101",'0','0', OP_OUI  ,'1','0','0',"101",OFS_SEXT,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_MEM, '1','0',D_RS,D_RT,D_RT,'0'),--LHU
(OP_NORMAL  ,"100011",'0','0', OP_OUI  ,'1','0','0',"010",OFS_SEXT,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_MEM, '1','0',D_RS,D_RT,D_RT,'0'),--LW
(OP_COP0    ,"000000",'0','0', OP_OP2  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','1', '1','0', '0','0',O,LVL_DI , '1','0',D_RS,D_RD,D_RT,'0'),--MFC0
(OP_SPECIAL ,"010000",'0','0', OP_MFHI ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--MFHI
(OP_SPECIAL ,"010010",'0','0', OP_MFLO ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--MFLO
(OP_COP0    ,"000100",'0','0', OP_OP2  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '1','0', '0','0',O,LVL_DI , '1','1',D_RS,D_RT,D_RD,'0'),--MTC0
(OP_SPECIAL ,"010001",'0','0', OP_MTHI ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_DI , '0','0',D_RS,D_RT,D_RT,'0'),--MTHI
(OP_SPECIAL ,"010011",'0','0', OP_MTLO ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
```

```
143 '0','0', '0','1', '0','0',Z,LVL_DI , '0','0',D_RS,D_RT,D_RT,'0'),--MTLO
144 (OP_SPECIAL ,"011000",'0','0', OP_MULT ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
145 '0','0', '0','0', '0','0',0,LVL_EX , '0','0',D_RS,D_RT,D_RT,'0'),--MULT
146 (OP_SPECIAL ,"011001",'0','0', OP_MULTU,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
147 '0','0', '0','0', '0','0',0,LVL_EX , '0','0',D_RS,D_RT,D_RT,'0'),--MULTU
148 (OP_SPECIAL ,"100111",'0','0', OP_NOR  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
149 '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--NOR
150 (OP_SPECIAL ,"100101",'0','0', OP_OR   ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
151 '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--OR
152 (OP_NORMAL  ,"001101",'0','0', OP_OR   ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
153 '0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_RT,'0'),--ORI
154 (OP_NORMAL  ,"101000",'0','0', OP_OP2  ,'1','1','0',"000",OFS_SEXT,IT_NOEXC,
155 '0','0', '0','0', '0','0',0,LVL_DI , '0','0',D_RS,D_RT,D_RT,'0'),--SB
156 (OP_NORMAL  ,"101001",'0','0', OP_OP2  ,'1','1','0',"001",OFS_SEXT,IT_NOEXC,
157 '0','0', '0','0', '0','0',0,LVL_DI , '0','0',D_RS,D_RT,D_RT,'0'),--SH
158 (OP_SPECIAL ,"000000",'0','0', OP_SLL  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
159 '0','0', '1','0', '1','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--SLL
160 (OP_SPECIAL ,"000100",'0','0', OP_SLL  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
161 '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--SLLV
162 (OP_SPECIAL ,"101010",'0','0', OP_SLT  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
163 '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--SLT
164 (OP_NORMAL  ,"001010",'0','0', OP_SLT  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
165 '0','0', '0','1', '0','1',Z,LVL_EX , '1','0',D_RS,D_RT,D_RT,'0'),--SLTI
166 (OP_NORMAL  ,"001011",'0','0', OP_SLTU ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
167 '0','0', '0','1', '0','1',Z,LVL_EX , '1','0',D_RS,D_RT,D_RT,'0'),--SLTIU
168 (OP_SPECIAL ,"101011",'0','0', OP_SLTU ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
169 '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--SLTU
170 (OP_SPECIAL ,"000011",'0','0', OP_SRA  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
171 '0','0', '1','0', '1','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--SRA
172 (OP_SPECIAL ,"000111",'0','0', OP_SRA  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
173 '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--SRAV
174 (OP_SPECIAL ,"000010",'0','0', OP_SRL  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
175 '0','0', '1','0', '1','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--SRL
176 (OP_SPECIAL ,"000110",'0','0', OP_SRL  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
177 '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--SRLV
178 (OP_SPECIAL ,"100010",'0','0', OP_SUB  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
179 '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--SUB
180 (OP_SPECIAL ,"100011",'0','0', OP_SUBU ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
181 '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--SUBU
182 (OP_NORMAL  ,"101011",'0','0', OP_OP2  ,'1','1','0',"010",OFS_SEXT,IT_NOEXC,
183 '0','0', '0','0', '0','0',0,LVL_DI , '0','0',D_RS,D_RT,D_RT,'0'),--SW
184 (OP_SPECIAL ,"001100",'0','0', OP_OUI  ,'0','0','0',"000",OFS_PCRL,IT_SCALL,
185 '0','0', '0','1', '0','0',Z,LVL_DI , '0','0',D_RS,D_RT,D_RT,'0'),--SYSC
186 (OP_SPECIAL ,"100110",'0','0', OP_XOR  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
187 '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'0'),--XOR
188 (OP_NORMAL  ,"001110",'0','0', OP_XOR  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
189 '0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_RT,'0'),--XORI
190 (OP_SPECIALP,"100000",'0','0', OP_ADD  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
191 '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--PADD
192 (OP_SPECIALP,"100001",'0','0', OP_ADDU ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
193 '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--PADDU
194 (OP_NORMAL  ,"011000",'0','0', OP_ADD  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
195 '0','0', '0','1', '0','1',Z,LVL_EX , '1','0',D_RS,D_RT,D_RT,'1'),--PADDI
```

```
196 (OP_NORMAL  ,"011001",'0','0', OP_ADDU ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
197 '0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_RT,'1'),--PADDIU
198 (OP_SPECIALP,"100100",'0','0', OP_AND  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
199 '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--PAND
200 (OP_NORMAL  ,"101100",'0','0', OP_AND  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
201 '0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_RT,'1'),--PANDI
202 (OP_SPECIALP,"011010",'0','0', OP_DIV  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
203 '0','0', '0','0', '0','0',0,LVL_EX , '0','0',D_RS,D_RT,D_RT,'1'),--PDIV
204 (OP_SPECIALP,"011011",'0','0', OP_DIVU ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
205 '0','0', '0','0', '0','0',0,LVL_EX , '0','0',D_RS,D_RT,D_RT,'1'),--PDIVU
206 (OP_PMEM    ,"000000",'0','0', OP_OUI  ,'1','0','0',"000",OFS_SEXT,IT_NOEXC,
207 '0','0', '0','1', '0','0',Z,LVL_MEM, '1','0',D_RS,D_RT,D_RT,'1'),--PLB
208 (OP_PMEM    ,"000011",'0','0', OP_OUI  ,'1','0','0',"100",OFS_SEXT,IT_NOEXC,
209 '0','0', '0','1', '0','0',Z,LVL_MEM, '1','0',D_RS,D_RT,D_RT,'1'),--PLBU
210 (OP_PMEM    ,"000001",'0','0', OP_OUI  ,'1','0','0',"001",OFS_SEXT,IT_NOEXC,
211 '0','0', '0','1', '0','0',Z,LVL_MEM, '1','0',D_RS,D_RT,D_RT,'1'),--PLH
212 (OP_PMEM    ,"000111",'0','0', OP_OUI  ,'1','0','0',"101",OFS_SEXT,IT_NOEXC,
213 '0','0', '0','1', '0','0',Z,LVL_MEM, '1','0',D_RS,D_RT,D_RT,'1'),--PLHU
214 (OP_NORMAL  ,"111100",'0','0', OP_LUI  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
215 '0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_RT,'1'),--PLUI
216 (OP_PMEM    ,"000010",'0','0', OP_OUI  ,'1','0','0',"010",OFS_SEXT,IT_NOEXC,
217 '0','0', '0','1', '0','0',Z,LVL_MEM, '1','0',D_RS,D_RT,D_RT,'1'),--PLW
218 (OP_SPECIALP,"101000",'0','0', OP_OP2  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
219 '0','1', '1','0', '0','0',0,LVL_DI , '1','0',D_RS,D_RD,D_RT,'1'),--PMFC0
220 (OP_SPECIALP,"010010",'0','0', OP_MFLO ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
221 '0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--PMFLO
222 (OP_SPECIALP,"010000",'0','0', OP_MFHI ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
223 '0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--PMFHI
224 (OP_SPECIALP,"101001",'0','0', OP_OP2  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
225 '0','0', '1','0', '0','0',0,LVL_DI , '1','1',D_RS,D_RT,D_RD,'1'),--PMTC0
226 (OP_SPECIALP,"010001",'0','0', OP_MTHI ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
227 '0','0', '0','1', '0','0',Z,LVL_DI , '0','0',D_RS,D_RT,D_RT,'1'),--PMTHI
228 (OP_SPECIALP,"010011",'0','0', OP_MTLO ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
229 '0','0', '0','1', '0','0',Z,LVL_DI , '0','0',D_RS,D_RT,D_RT,'1'),--PMTLO
230 (OP_SPECIALP,"011000",'0','0', OP_MULT ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
231 '0','0', '0','0', '0','0',0,LVL_EX , '0','0',D_RS,D_RT,D_RT,'1'),--PMULT
232 (OP_SPECIALP,"011001",'0','0', OP_MULTU,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
233 '0','0', '0','0', '0','0',0,LVL_EX , '0','0',D_RS,D_RT,D_RT,'1'),--PMULTU
234 (OP_SPECIALP,"100111",'0','0', OP_NOR  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
235 '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--PNOR
236 (OP_SPECIALP,"100101",'0','0', OP_OR   ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
237 '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--POR
238 (OP_NORMAL  ,"101101",'0','0', OP_OR   ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
239 '0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_RT,'1'),--PORI
240 (OP_PMEM    ,"000100",'0','0', OP_OP2  ,'1','1','0',"000",OFS_SEXT,IT_NOEXC,
241 '0','0', '0','1', '0','0',0,LVL_DI , '0','0',D_RS,D_RT,D_RT,'1'),--PSB
242 (OP_PMEM    ,"000101",'0','0', OP_OP2  ,'1','1','0',"001",OFS_SEXT,IT_NOEXC,
243 '0','0', '0','1', '0','0',0,LVL_DI , '0','0',D_RS,D_RT,D_RT,'1'),--PSH
244 (OP_SPECIALP,"101010",'0','0', OP_SLT  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC
245 '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--PSLT
246 (OP_SPECIALP,"101011",'0','0', OP_SLTU ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
247 '0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--PSLTU
248 (OP_NORMAL  ,"011010",'0','0', OP_SLT  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
```

```
'0','0', '0','1', '0','1',Z,LVL_EX , '1','0',D_RS,D_RT,D_RT,'1'),--PSLTI
(OP_NORMAL  ,"011011",'0','0', OP_SLTU ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '0','1', '0','1',Z,LVL_EX , '1','0',D_RS,D_RT,D_RT,'1'),--PSLTIU
(OP_PMEM    ,"000110",'0','0', OP_OP2  ,'1','1','0',"010",OFS_SEXT,IT_NOEXC,
'0','0', '0','1', '0','0',0,LVL_DI , '0','0',D_RS,D_RT,D_RT,'1'),--PSW
(OP_SPECIALP,"000000",'0','0', OP_SLL  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '1','0', '1','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--PSLL
(OP_SPECIALP,"000100",'0','0', OP_SLL  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--PSLLV
(OP_SPECIALP,"000011",'0','0', OP_SRA  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '1','0', '1','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--PSRA
(OP_SPECIALP,"000111",'0','0', OP_SRA  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--PSRAV
(OP_SPECIALP,"000010",'0','0', OP_SRL  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '1','0', '1','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--PSRL
(OP_SPECIALP,"000110",'0','0', OP_SRL  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--PSRLV
(OP_SPECIALP,"100010",'0','0', OP_SUB  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--PSUB
(OP_SPECIALP,"100011",'0','0', OP_SUBU ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--PSUBU
(OP_SPECIALP,"100110",'0','0', OP_XOR  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '0','0', '0','0',0,LVL_EX , '1','0',D_RS,D_RT,D_RD,'1'),--PXOR
(OP_NORMAL  ,"110100",'0','0', OP_XOR  ,'0','0','0',"000",OFS_PCRL,IT_NOEXC,
'0','0', '0','1', '0','0',Z,LVL_EX , '1','0',D_RS,D_RT,D_RT,'1') --PXORI
);
```

## Décodage des instructions

```

-- Definir les sorties synchrones
    process (clock)
    begin
        if clock='1' and clock'event then
            if reset='1' then
                DI_bra <= '0';
                DI_link <= '0';
                DI_op1 <= (others => '0');
                DI_op2 <= (others => '0');
                DI_code_ual <= OP_OUI;
                DI_offset <= (others => '0');
                DI_adr_reg_dest <= (others => '0');
                DI_ecr_reg <= '0';
                DI_mode <= '0';
                DI_op_mem <= '0';
                DI_r_w <= '0';
                DI_mem_size <= SIZE_BYTE;
                DI_adr <= (others => '0');
                DI_exc_cause <= IT_NOEXC;
                DI_level <= LVL_DI;
                DI_it_ok <= '0';
                DI_rd    <= "00000";
                state <= 0;
```

```
25
26                  DI_SL_op <= (others => '0');
27                  DI_SL_code_ual <= OP_OUI;
28                  DI_SL_adr_reg_dest <= (others => '0');
29                  DI_SL_adr_reg_scr1 <= (others => '0');
30                  DI_SL_adr_reg_scr2 <= (others => '0');
31                  DI_SL_adr_reg_use1 <= '0';
32                  DI_SL_adr_reg_use2 <= '0';
33                  DI_SL_ecr_reg <= '0';
34                  DI_SL_op_mem <= '0';
35                  DI_SL_r_w <= '0';
36                  DI_SL_mem_size <= SIZE_BYTE;
37                  DI_SL_it_ok <= '0';
38              elsif stop_all='0' then
39                  if clear='1' or stop_di='1' then
40                      -- Nop instruction
41                      DI_bra <= '0';
42                      DI_link <= '0';
43                      DI_op1 <= (others => '0');
44                      DI_op2 <= (others => '0');
45                      DI_code_ual <= OP_OUI;
46                      DI_offset <= (others => '0');
47                      DI_adr_reg_dest <= (others => '0');
48                      DI_ecr_reg <= '0';
49                      DI_mode <= '0';
50                      DI_op_mem <= '0';
51                      DI_r_w <= '0';
52                      DI_mem_size <= SIZE_BYTE;
53                      DI_adr <= (others => '0');
54                      DI_exc_cause <= IT_NOEXC;
55                      DI_level <= LVL_DI;
56                      DI_rd    <= "00000";
57                      if clear='1' then
58                        DI_it_ok <= '0';
59                        DI_SL_it_ok <= '0';
60                      else
61                        DI_it_ok <= EI_it_ok;
62                        DI_SL_it_ok <= EI_it_ok;
63                      end if;
64                      state <= state;
65
66                      DI_SL_op <= (others => '0');
67                      DI_SL_code_ual <= OP_OUI;
68                      DI_SL_adr_reg_dest <= (others => '0');
69                      DI_SL_adr_reg_scr1 <= (others => '0');
70                      DI_SL_adr_reg_scr2 <= (others => '0');
71                      DI_SL_adr_reg_use1 <= '0';
72                      DI_SL_adr_reg_use2 <= '0';
73                      DI_SL_ecr_reg <= '0';
74                      DI_SL_op_mem <= '0';
75                      DI_SL_r_w <= '0';
76                      DI_SL_mem_size <= SIZE_BYTE;
77                  elsif vector = '0' then -- Noraml step, not-vector instruction
```

```
78                      --Master
79                      DI_bra <= PRE_bra;
80                      DI_link <= PRE_link;
81                      DI_op1 <= PRE_op1;
82                      DI_op2 <= PRE_op2;
83                      DI_code_ual <= PRE_code_ual;
84                      DI_offset <= PRE_offset;
85                      DI_adr_reg_dest <= PRE_adr_reg_dest;
86                      DI_ecr_reg <= PRE_ecr_reg;
87                      DI_mode <= PRE_mode;
88                      DI_op_mem <= PRE_op_mem;
89                      DI_r_w <= PRE_r_w;
90                      DI_mem_size <= PRE_mem_size;
91                      DI_adr <= EI_adr;
92                      DI_exc_cause <= PRE_exc_cause;
93                      DI_level <= PRE_level;
94                      DI_it_ok <= EI_it_ok;
95                      DI_rd    <= PRE_bit_sel;
96                      state <= next_state;
97
98                      --Slave NOP
99                      DI_SL_op <= (others => '0');
100                     DI_SL_code_ual <= OP_OUI;
101                     DI_SL_adr_reg_dest <= (others => '0');
102                     DI_SL_adr_reg_scr1 <= (others => '0');
103                     DI_SL_adr_reg_scr2 <= (others => '0');
104                     DI_SL_adr_reg_use1 <= '0';
105                     DI_SL_adr_reg_use2 <= '0';
106                     DI_SL_ecr_reg <= '0';
107                     DI_SL_op_mem <= '0';
108                     DI_SL_r_w <= '0';
109                     DI_SL_mem_size <= SIZE_BYTE;
110                     DI_SL_it_ok <= '0';
111                 else --vector instruction
112                     --Master NOP
113                     DI_bra <= '0';
114                     DI_link <= '0';
115                     DI_op1 <= (others => '0');
116                     DI_op2 <= (others => '0');
117                     DI_code_ual <= OP_OUI;
118                     DI_offset <= (others => '0');
119                     -- Necessary as hazard detection is done by Master
120                     DI_adr_reg_dest <= PRE_adr_reg_dest;--(others => '0');
121                     DI_ecr_reg <= '0';
122                     DI_mode <= '0';
123                     DI_op_mem <= '0';
124                     DI_r_w <= '0';
125                     DI_mem_size <= SIZE_BYTE;
126                     DI_adr <= EI_adr;
127                     DI_exc_cause <= IT_NOEXC;
128                     DI_rd    <= PRE_bit_sel;
129                     -- Necessary as hazard detection is done by Master
130                     DI_level <= PRE_level;--LVL_DI;
```

```
131                     if clear='1' then
132                       DI_it_ok <= '0';
133                       DI_SL_it_ok <= '0';
134                     else
135                       DI_it_ok <= EI_it_ok;
136                       DI_SL_it_ok <= EI_it_ok;
137                     end if;
138                     state <= next_state;
139 
140                     --Slave
141                     DI_SL_op <= PRE_sl_op;
142                     DI_SL_code_ual <= PRE_code_ual;
143                     DI_SL_adr_reg_dest <= '0' & PRE_adr_reg_dest(5 downto 0);
144                     DI_SL_adr_reg_scr1 <= '0' & PRE_adr_reg_scr1(5 downto 0);
145                     DI_SL_adr_reg_scr2 <= '0' & PRE_adr_reg_scr2(5 downto 0);
146                     DI_SL_adr_reg_use1 <= PRE_adr_reg_use1;
147                     DI_SL_adr_reg_use2 <= PRE_adr_reg_use2;
148                     DI_SL_ecr_reg <= PRE_ecr_reg;
149                     DI_SL_op_mem <= PRE_op_mem;
150                     DI_SL_r_w <= PRE_r_w;
151                     DI_SL_mem_size <= PRE_mem_size;
152                     DI_SL_it_ok <= EI_it_ok;
153                 end if;
154             end if;
155         end if;
156     end process;
```

# Annexe B

# Fichiers de compilation OR32

### Makefile

```
main.or32: boot.o main.o
        or32-elf-gcc -g -T orp.ld boot.o main.o support/uart.o
        support/uart_basic.o -o main.or32

main.o: main.c
        or32-elf-gcc -g -c main.c -o main.o

boot.o: boot.S
        or32-elf-gcc -g -c -DIC=0 -DDC=0 boot.S -o boot.o

clean:
        rm -f *.o main.or32 main.bin main.vmem
```

### Boot.s

```

/* Support defines */
#define STACK_SIZE          0x01000

#define MAX_SPRS_PER_GRP_BITS (11)
#define SPRGROUP_DC     (3<< MAX_SPRS_PER_GRP_BITS)
#define SPRGROUP_IC     (4<< MAX_SPRS_PER_GRP_BITS)
#define SPR_DCBIR       (SPRGROUP_DC + 3)
#define SPR_ICBIR       (SPRGROUP_IC + 2)

#define SPR_SR_ICE      0x00000010  /* Instruction Cache Enable */
#define SPR_SR_DCE      0x00000008  /* Data Cache Enable */

#define SPR_SR_SM       0x00000001  /* Supervisor Mode */


        .section .stack
        .space STACK_SIZE
_stack:

        .section .reset, "ax"
```

```
22
23          .org    0x100
24  _reset_vector:
25          l.nop
26          l.nop
27          l.addi  r2,r0,0x0
28          l.addi  r3,r0,0x0
29          l.addi  r4,r0,0x0
30          l.addi  r5,r0,0x0
31          l.addi  r6,r0,0x0
32          l.addi  r7,r0,0x0
33          l.addi  r8,r0,0x0
34          l.addi  r9,r0,0x0
35          l.addi  r10,r0,0x0
36          l.addi  r11,r0,0x0
37          l.addi  r12,r0,0x0
38          l.addi  r13,r0,0x0
39          l.addi  r14,r0,0x0
40          l.addi  r15,r0,0x0
41          l.addi  r16,r0,0x0
42          l.addi  r17,r0,0x0
43          l.addi  r18,r0,0x0
44          l.addi  r19,r0,0x0
45          l.addi  r20,r0,0x0
46          l.addi  r21,r0,0x0
47          l.addi  r22,r0,0x0
48          l.addi  r23,r0,0x0
49          l.addi  r24,r0,0x0
50          l.addi  r25,r0,0x0
51          l.addi  r26,r0,0x0
52          l.addi  r27,r0,0x0
53          l.addi  r28,r0,0x0
54          l.addi  r29,r0,0x0
55          l.addi  r30,r0,0x0
56          l.addi  r31,r0,0x0
57
58          l.movhi r3,hi(_start)
59          l.ori   r3,r3,lo(_start)
60          l.jr    r3
61          l.nop
62
63          .section .text
64
65  _start:
66
67  .if IC | DC
68          /* Flush IC and/or DC */
69          l.addi  r10,r0,0
70          l.addi  r11,r0,0
71          l.addi  r12,r0,0
72  .if IC
73          l.addi  r11,r0,IC_SIZE
74  .endif
```

```
.if DC
        l.addi  r12,r0,DC_SIZE
.endif
        l.sfleu r12,r11
        l.bf    loop
        l.nop
        l.add   r11,r0,r12
loop:
.if IC
        l.mtspr r0,r10,SPR_ICBIR
.endif
.if DC
        l.mtspr r0,r10,SPR_DCBIR
.endif
        l.sfne  r10,r11
        l.bf    loop
        l.addi  r10,r10,16

        /* Enable IC and/or DC */
        l.addi  r10,r0,(SPR_SR_SM)
.if IC
        l.ori   r10,r10,(SPR_SR_ICE)
.endif
.if DC
        l.ori   r10,r10,(SPR_SR_DCE)
.endif
        l.mtspr r0,r10,SPR_SR
        l.nop
        l.nop
        l.nop
        l.nop
        l.nop
.endif

        /* Set stack pointer */
        l.movhi r1,hi(_stack)
        l.ori   r1,r1,lo(_stack)

        /* Jump to main */
        l.movhi r2,hi(_main)
        l.ori   r2,r2,lo(_main)
        l.jr    r2
        l.nop
```

## Ram.ld

```
MEMORY
        {
        reset   : ORIGIN = 0x00000000, LENGTH = 0x00000200
        vectors : ORIGIN = 0x00000200, LENGTH = 0x00001000
    ram     : ORIGIN = 0x00001200, LENGTH = 0x00006E00
/*0x8000 total*/
        }
```

```
7
8  SECTIONS
9  {
10          .reset :
11          {
12          *(.reset)
13          } > reset
14
15
16          .vectors :
17          {
18          _vec_start = .;
19          *(.vectors)
20          _vec_end = .;
21          } > vectors
22
23          .text :
24          {
25          *(.text)
26          } > ram
27
28        .rodata :
29          {
30          *(.rodata)
31          *(.rodata.*)
32          } > ram
33
34       .icm :
35          {
36          _icm_start = .;
37          *(.icm)
38          _icm_end = .;
39          } > ram
40
41       .data :
42          {
43          _dst_beg = .;
44          *(.data)
45          _dst_end = .;
46          } > ram
47
48        .bss :
49          {
50          *(.bss)
51          } > ram
52
53        .stack (NOLOAD) :
54          {
55          *(.stack)
56          _src_addr = .;
57          } > ram
58
59 }
```

TABLE C.1 – Cas OPCOD = SPECIAL, décodage de FUNC

| | | INS 2 :0 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| INS 5 :3 | 000 | SLL | | SRL | SRA | SLLV | | SRLV | SRAV |
| | 001 | JR | JALR | | | SYSCALL | BREAK | | |
| | 010 | MFHI | MTHI | MFLO | MTLO | | | | |
| | 011 | MULT | MULTU | DIV | DIVU | | | | |
| | 100 | ADD | ADDU | SUB | SUBU | AND | OR | XOR | NOR |
| | 101 | | | SLT | SLTU | | | | |
| | 110 | | | | | | | | |
| | 111 | | | | | | | | |

# Annexe C

# Codage des instructions MIPS32

### Cas OPCOD = SPECIAL, décodage de FUNC

Le code SPECIAL est particulier, c'est un code d'indirection. Il indique en fait qu'on se trouve dans le cas d'une instruction de type R. Il faut alors analyser le code FUNC (les 6 bits de poids faibles) pour décoder l'instruction. Ce code est utilisé par les instructions qui ne nécessitent pas d'immédiat, mais plutôt plusieurs registres. Il permet de laisser des valeurs pour OPCOD disponibles. La table C.1 indique les valeurs possibles du code FUNC.

### Cas OPCOD = SPECIALP, décodage de FUNC

Nous avons ajouté le code SPECIALP. Il fonctionne de la même manière que SPECIAL. Il s'agit en fait d'un autre code d'indirection. Ce code est utilisé pour les instructions dédiées aux PEs et de type R. Comme le code SPECIAL, il permet d'économiser de la place dans le tableau des valeurs de l'OPCOD, et ainsi d'utiliser plus d'instructions (mais interdit l'utilisation d'un immédiat sur 16 bits). La table C.2 indique les valeurs possibles du code FUNC dans ce cas.

### Cas OPCOD = P_MEM

L'OPCOD P_MEM est un code d'indirection qui contient l'ensemble des instructions d'accès mémoire des PEs. Ces instructions comportent un immédiat sur 16 bits, dont les

TABLE C.2 – Cas OPCOD = SPECIALP, décodage de FUNC

| | | INS 2 :0 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| INS 5 :3 | 000 | P_SLL | | P_SRL | P_SRA | P_SLLV | | P_SRLV | P_SRAV |
| | 001 | | | | | | | | |
| | 010 | P_MFHI | P_MTHI | P_MFLO | P_MTLO | P_SETB | P_FB | P_MTB | P_BRD |
| | 011 | P_MULT | P_MULTU | P_DIV | P_DIVU | P_ANDB | P_NOTB | P_MB | |
| | 100 | P_ADD | P_ADDU | P_SUB | P_SUBU | P_AND | P_OR | P_XOR | P_NOR |
| | 101 | P_MFC0 | P_MTC0 | P_SLT | P_SLTU | | | | |
| | 110 | | | | | | | | |
| | 111 | | | | | | | | |

TABLE C.3 – Cas OPCOD = P_MEM

| | | INS 14 :13 | | | |
|---|---|---|---|---|---|
| | | 00 | 01 | 10 | 11 |
| INS 15 | 0 | P_LB | P_LH | P_LW | P_LBU |
| | 1 | P_SB | P_SH | P_SW | P_LHU |

TABLE C.4 – Cas OPCOD = BCOND

| | | INS 16 | |
|---|---|---|---|
| | | 0 | 1 |
| INS 20 | 0 | BLTZ | BGEZ |
| | 1 | BLTZAL | BGEZAL |

TABLE C.5 – Cas OPCOD = COPRO

| | | INS 23 | |
|---|---|---|---|
| | | 0 | 1 |
| INS 25 | 0 | MFC0 | MTC0 |
| | 1 | RFE | |

3 bits de poids forts indiquent la fonction (lecture/écriture et taille des données). L'adresse est calculée comme la somme de l'immédiat et du registre Rs. Ainsi, la diminution de la taille de l'immédiat à 13 bits ne pose pas de problèmes. Le tableau C.3 montre la signification des différentes valeurs que peuvent prendre les 3 bits de poids forts.

### Autres valeurs de OPCOD

D'autres cas particuliers sont décrits dans les tableaux C.4 et C.5.

TABLE C.6 – Extension du jeu d'instructions

| Syntaxe | Description de l'instruction | Type |
|---|---|---|
| **S_LBP** Rt, I(Rs) | **ACU Load Byte PE** : ARt <- PE_M[i](Adr)<br>l'ACU charge un byte de la mémoire du PE<br>la mémoire cible : i = (ARs + I)/MAX_LOC_SIZE,<br>l'adresse : Adr=(ARs + I)%MAX_LOC_SIZE | I [1] |
| **S_LWP** Rt, I(Rs) | **ACU Load Word PE** : ARt <- PE_M[i](Adr)<br>l'ACU charge un mot de la mémoire du PE | I [1] |
| **S_SBP** Rt, I(Rs) | **ACU Store Byte PE** : PE_M[i](Adr) <- ARt<br>l'ACU stocke un byte dans la mémoire du PE<br>la mémoire cible : i = (ARs + I)/MAX_LOC_SIZE,<br>l'adresse : Adr=(ARs + I)%MAX_LOC_SIZE | I [1] |
| **S_SWP** Rt, I(Rs) | **ACU Store Word PE** : PE_M[i](Adr) <- ARt<br>l'ACU stocke un mot dans la mémoire du PE | I [1] |
| **P_MFCO** Rt, Rd | **PE Move From Control Coprocessor** : PRt <- Copro[Rd]<br>le registre Rd du coprocesseur est déplacé dans<br>le registre entier PRt | R |
| **P_MTCO** Rt, Rd | **PE Move To Control Coprocessor** : Copro[Rd] <- PRt<br>le registre entier PRt est déplacé dans<br>le registre Rd du coprocesseur | R |
| **P_SETB** Rd, Sh | **PE SET Bit** : PBd <- Sh<br>le bit Rd du registre STATUS est initialisé à Sh | R [2] |
| **P_MB** Rd, Sh | **PE Move Bit** : PBd <- PBh<br>le bit Sh du registre STATUS est déplacé au bit Rd<br>du registre STATUS | R [2] |
| **P_MFB** Rt, Rd | **PE Move From Bit** : PRt <- PBd<br>le bit Rd du registre STATUS est déplacé dans<br>le registre PRt | R [2] |
| **P_MTB** Rs, Rd | **PE Move To Bit** : PBd <- PRs<br>le registre PRs est déplacé dans le bit Rd<br>du registre STATUS | R [2] |
| **P_ANDB** Rs, Rd | **PE AND Bit** : PBd <- PBd and PRs<br>AND logique entre le bit Rd du registre STATUS et PRs | R [2] |
| **P_NOTB Rd** | **PE NOT Bit** : PBd <- not PBd<br>le bit Rd du registre STATUS est inversé | R [2] |
| **P_BRD** Rt, Rs | **PE BRoaDcast** : PRt <- SRs<br>le registre SRs de l'ACU est déplacé dans<br>le registre PRt de chaque PE | R |

### Nouvelles instructions ajoutées

Le jeu d'instructions MIPSI a été étendu. Les nouvelles instructions ajoutées sont décrites dans le tableau C.6.

1. 13 bits seulement sont utilisés pour la valeur immédiate
2. Exécuté seulement si le bit E est à 0

# Annexe D

# Description des outils de compilation du processeur miniMIPS

### Génération

*Generation* est le premier outil à utiliser pour la génération d'un fichier exécutable. Il s'utilise ainsi :

$$generation \ < dest\_s\_src > < dest\_p > < src > \tag{D.1}$$

Il permet, à partir d'un fichier source assembleur étendu src, d'en séparer le code MIPSI du code étendu. Après exécution, dest_s_src contient le code assembleur correspondant aux parties compatibles MIPSI de src. Sa seule différence avec src est qu'à la place des instructions additionnelles de l'assembleur étendu, on a placé des instructions NOP. Le deuxième fichier généré est dest_p. Il ne contient que les instructions spécifiques au nouveau jeu étendu. De plus, il est directement créé sous sa forme binaire, c'est-à-dire que toutes les instructions sont traduites en un mot de 4 octets et stockées à la suite dans le fichier. Toutes ces instructions sont placées les unes à la suite des autres. Lorsque, du coté du code séquentiel, on trouve une instruction NOP, on place aussi un NOP dans dest_p (sinon on ne pourrait savoir que ce NOP est à conserver et pas à remplacer). Pour la génération de ce second fichier, *generation* fournit le travail d'un assembleur allégé. Cette technique nous permet d'obtenir un fichier source en assembleur MIPSI. On peut alors l'assembler avec *sde-as*. On obtient deux binaires : le premier est un fichier exécutable au format elf ne contenant pas les instructions étendues, le second est un fichier contenant les instructions étendues du programme. Il n'est pas au format elf, il ne contient en fait que le segment de code.

### Fusion

Fusion permet d'obtenir le fichier binaire exécutable final. Il s'utilise ainsi :

$$fusion < dest > < src\_s > < src\_p > \tag{D.2}$$

src_s et src_p sont les deux fichiers binaires obtenus précédemment (par *generation* et sde-as). dest est le fichier binaire final. Il est exécutable par le simulateur MPPSoC. La fusion est en fait assez simple à réaliser. On effectue une copie du fichier src_s où les instructions NOP sont remplacées par une instruction de src_p. Le fichier src_p est donc lu séquentiellement.

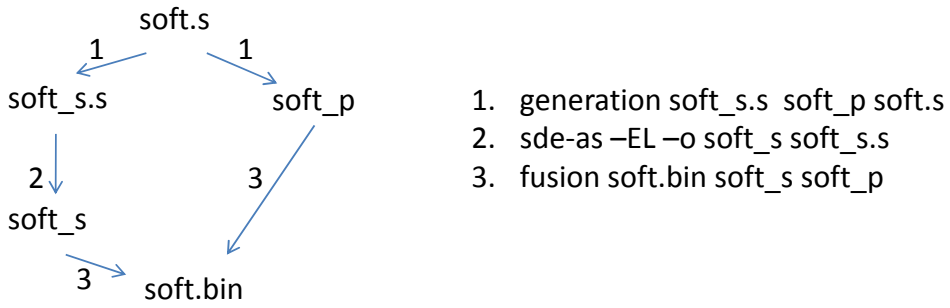



FIGURE D.1 – *Étapes d'assemblage*

Le nombre d'instructions contenues dans src_p est égal au nombre d'instructions NOP dans le fichier src_s.

### mppsocassembly

*mppsocassembly* est un simple script-shell permettant directement de générer un exécutable étendu à partir d'un fichier source en assembleur étendu. Il s'utilise ainsi :

$$mppsocassembly \ < dest > < src > \tag{D.3}$$

Il compile les outils *generation* et *fusion* si nécessaire. Ensuite, il utilise *generation*, sde-as et *fusion*. La figure D.1 résume les différentes étapes de l'assemblage :

$$mppsocassembly \quad soft.bin \quad soft.s \tag{D.4}$$

### elftext2mif

*elftext2mif* est un script-shell permettant de générer un fichier en format MIF (Memory Initialisation File) à partir d'un fichier binaire généré. Le code source de ce script est le suivant :

```
1  #!/bin/sh
2  # take in input an elf file and have as output a mif file
3  echo "WIDTH=32;"
4  echo "DEPTH=1024;"
5  echo
6  echo "ADDRESS_RADIX=UNS;"
7  echo "DATA_RADIX=HEX;"
8  echo
9  echo "CONTENT BEGIN"
10
11 for file; do
12
13 sde-objdump -D $file | cut -f 2 | tail +7 | grep -v : | grep -v ^$ | \
14 pr -T -n -N 0 | sed "s/^[[:blank:]]\+\([[:digit:]]\+\)[[:blank:]]\+\
15 ([[:xdigit:]]\+\)/\t\1\t:\2\;/g"
16
17 done
18 echo "END;"
```

# Annexe E

# IPs Mémoires de la bibliothèque mppSoC

### Mémoire portable

```
1
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.ALL;
4  USE ieee.numeric_std.ALL;
5
6  ENTITY ram IS
7          GENERIC
8          (
9                  ADDRESS_WIDTH   : integer := 10;
10                 DATA_WIDTH      : integer := 32
11         );
12         PORT
13         (
14         clock           : IN  std_logic;
15         data            : IN  std_logic_vector(DATA_WIDTH - 1 DOWNTO 0);
16         write_address   : IN  std_logic_vector(ADDRESS_WIDTH - 1 DOWNTO 0);
17         read_address    : IN  std_logic_vector(ADDRESS_WIDTH - 1 DOWNTO 0);
18         we              : IN  std_logic;
19         q               : OUT std_logic_vector(DATA_WIDTH - 1 DOWNTO 0)
20         );
21 END ram;
22
23 ARCHITECTURE rtl OF ram IS
24 TYPE RAM IS ARRAY(0 TO 2 ** ADDRESS_WIDTH - 1) OF
25 std_logic_vector(DATA_WIDTH - 1 DOWNTO 0);
26
27         SIGNAL ram_block : RAM;
28 BEGIN
29         PROCESS (clock)
30         BEGIN
31                 IF (clock'event AND clock = '1') THEN
32                         IF (we = '1') THEN
33                             ram_block(to_integer(unsigned(write_address)))
```

```
                            <= data;
                        END IF;

                        q <= ram_block(to_integer(unsigned(read_address)));
                END IF;
        END PROCESS;
END rtl;
```

### Mémoire Altera

```

-- megafunction wizard: %RAM: 1-PORT%
-- GENERATION: STANDARD
-- VERSION: WM1.0
-- MODULE: altsyncram

-- ============================================================
-- File Name: ram.vhd
-- Megafunction Name(s):
--                      altsyncram
--
-- Simulation Library Files(s):
--                      altera_mf
-- ============================================================
-- ************************************************************
-- THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
-- ************************************************************


LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY altera_mf;
USE altera_mf.all;

ENTITY ram IS
        PORT
        (
                address : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
                clken           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
                wren            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
        );
END ram;


ARCHITECTURE SYN OF ram IS

        SIGNAL sub_wire0        : STD_LOGIC_VECTOR (31 DOWNTO 0);


```

```
44
45          COMPONENT altsyncram
46          GENERIC (
47                  address_aclr_a          : STRING;
48                  indata_aclr_a           : STRING;
49                  init_file               : STRING;
50                  intended_device_family  : STRING;
51                  lpm_hint                : STRING;
52                  lpm_type                : STRING;
53                  numwords_a              : NATURAL;
54                  operation_mode          : STRING;
55                  outdata_aclr_a          : STRING;
56                  outdata_reg_a           : STRING;
57                  power_up_uninitialized          : STRING;
58                  widthad_a               : NATURAL;
59                  width_a         : NATURAL;
60                  width_byteena_a         : NATURAL;
61                  wrcontrol_aclr_a                : STRING
62          );
63          PORT (
64                          clocken0        : IN STD_LOGIC ;
65                          wren_a  : IN STD_LOGIC ;
66                          clock0  : IN STD_LOGIC ;
67                          address_a       : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
68                          q_a     : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
69                          data_a  : IN STD_LOGIC_VECTOR (31 DOWNTO 0)
70          );
71          END COMPONENT;
72
73  BEGIN
74          q    <= sub_wire0(31 DOWNTO 0);
75
76          altsyncram_component : altsyncram
77          GENERIC MAP (
78                  address_aclr_a => "NONE",
79                  indata_aclr_a => "NONE",
80                  init_file => "mem.hex",
81                  intended_device_family => "Stratix",
82                  lpm_hint => "ENABLE_RUNTIME_MOD=NO",
83                  lpm_type => "altsyncram",
84                  numwords_a => 1024,
85                  operation_mode => "SINGLE_PORT",
86                  outdata_aclr_a => "NONE",
87                  outdata_reg_a => "UNREGISTERED",
88                  power_up_uninitialized => "FALSE",
89                  widthad_a => 10,
90                  width_a => 32,
91                  width_byteena_a => 1,
92                  wrcontrol_aclr_a => "NONE"
93          )
94          PORT MAP (
95                  clocken0 => clken,
96                  wren_a => wren,
```

```
97                  clock0 => clock,
98                  address_a => address,
99                  data_a => data,
100                 q_a => sub_wire0
101         );
102
103
104
105 END SYN;
106
107 -- ============================================================
108 -- CNX file retrieval info
109 -- ============================================================
110 -- Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
111 -- Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
112 -- Retrieval info: PRIVATE: AclrByte NUMERIC "0"
113 -- Retrieval info: PRIVATE: AclrData NUMERIC "0"
114 -- Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
115 -- Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
116 -- Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
117 -- Retrieval info: PRIVATE: BlankMemory NUMERIC "0"
118 -- Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "1"
119 -- Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
120 -- Retrieval info: PRIVATE: Clken NUMERIC "1"
121 -- Retrieval info: PRIVATE: DataBusSeparated NUMERIC "1"
122 -- Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
123 -- Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
124 -- Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
125 -- Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Stratix"
126 -- Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
127 -- Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
128 -- Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
129 -- Retrieval info: PRIVATE: MIFfilename STRING "data_ACU.hex"
130 -- Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "17161"
131 -- Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
132 -- Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_A NUMERIC "3"
133 -- Retrieval info: PRIVATE: RegAddr NUMERIC "1"
134 -- Retrieval info: PRIVATE: RegData NUMERIC "1"
135 -- Retrieval info: PRIVATE: RegOutput NUMERIC "0"
136 -- Retrieval info: PRIVATE: SingleClock NUMERIC "1"
137 -- Retrieval info: PRIVATE: UseDQRAM NUMERIC "1"
138 -- Retrieval info: PRIVATE: WRCONTROL_ACLR_A NUMERIC "0"
139 -- Retrieval info: PRIVATE: WidthAddr NUMERIC "15"
140 -- Retrieval info: PRIVATE: WidthData NUMERIC "32"
141 -- Retrieval info: PRIVATE: rden NUMERIC "0"
142 -- Retrieval info: CONSTANT: ADDRESS_ACLR_A STRING "NONE"
143 -- Retrieval info: CONSTANT: INDATA_ACLR_A STRING "NONE"
144 -- Retrieval info: CONSTANT: INIT_FILE STRING "data_ACU.hex"
145 -- Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Stratix"
146 -- Retrieval info: CONSTANT: LPM_HINT STRING "ENABLE_RUNTIME_MOD=NO"
147 -- Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
148 -- Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "17161"
149 -- Retrieval info: CONSTANT: OPERATION_MODE STRING "SINGLE_PORT"
```

```
150 -- Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
151 -- Retrieval info: CONSTANT: OUTDATA_REG_A STRING "UNREGISTERED"
152 -- Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED STRING "FALSE"
153 -- Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "15"
154 -- Retrieval info: CONSTANT: WIDTH_A NUMERIC "32"
155 -- Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
156 -- Retrieval info: CONSTANT: WRCONTROL_ACLR_A STRING "NONE"
157 -- Retrieval info: USED_PORT: address 0 0 15 0 INPUT NODEFVAL address[14..0]
158 -- Retrieval info: USED_PORT: clken 0 0 0 0 INPUT NODEFVAL clken
159 -- Retrieval info: USED_PORT: clock 0 0 0 0 INPUT NODEFVAL clock
160 -- Retrieval info: USED_PORT: data 0 0 32 0 INPUT NODEFVAL data[31..0]
161 -- Retrieval info: USED_PORT: q 0 0 32 0 OUTPUT NODEFVAL q[31..0]
162 -- Retrieval info: USED_PORT: wren 0 0 0 0 INPUT NODEFVAL wren
163 -- Retrieval info: CONNECT: @address_a 0 0 15 0 address 0 0 15 0
164 -- Retrieval info: CONNECT: q 0 0 32 0 @q_a 0 0 32 0
165 -- Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
166 -- Retrieval info: CONNECT: @clocken0 0 0 0 0 clken 0 0 0 0
167 -- Retrieval info: CONNECT: @data_a 0 0 32 0 data 0 0 32 0
168 -- Retrieval info: CONNECT: @wren_a 0 0 0 0 wren 0 0 0 0
169 -- Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
170 -- Retrieval info: GEN_FILE: TYPE_NORMAL ram.vhd TRUE
171 -- Retrieval info: GEN_FILE: TYPE_NORMAL ram.inc FALSE
172 -- Retrieval info: GEN_FILE: TYPE_NORMAL ram.cmp FALSE
173 -- Retrieval info: GEN_FILE: TYPE_NORMAL ram.bsf FALSE
174 -- Retrieval info: GEN_FILE: TYPE_NORMAL ram_inst.vhd FALSE
175 -- Retrieval info: LIB_FILE: altera_mf
```

# Annexe F

# Écran LCD TRDB-LTM

### Diagramme de temps horizontal

La figure F.1 montre l'évolution des signaux de la dimension horizontale de l'écran dans le temps.

La figure F.2 montre les paramètres de l'écran LCD dans la direction horizontale.

La figure F.3 montre l'évolution des signaux de la dimension verticale de l'écran dans le temps.

La figure F.4 montre les paramètres de l'écran LCD dans la direction verticale.

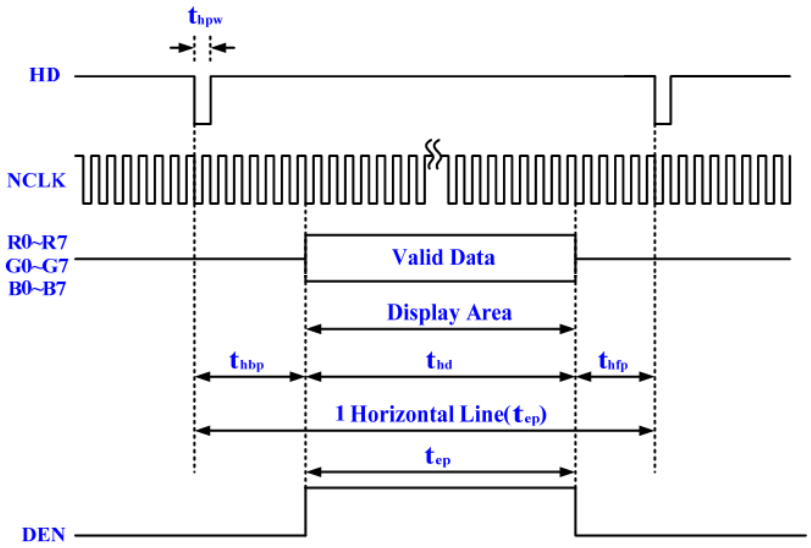



FIGURE F.1 – *Spécification des signaux de l'écran dans la direction horizontale*

| Parameter | | Symbol | Panel Resolution | | | Unit |
|---|---|---|---|---|---|---|
| | | | 800xRGBx480 | 480xRGBx272 | 400xRGBx240 | |
| NCLK Frequency | | FNCLK | 33.2 | 9 | 8.3 | MHz |
| Horizontal valid data | | $t_{hd}$ | 800 | 480 | 400 | NCLK |
| 1 Horizontal Line | | $t_h$ | 1056 | 525 | 528 | NCLK |
| HSYNC Pulse Width | Min. | $t_{hpw}$ | 1 | | | NCLK |
| | Typ. | | - | | | |
| | Max. | | - | | | |
| Hsync back porch | | $t_{hbp}$ | 216 | 43 | 108 | NCLK |
| Hsync front porch | | $t_{hfp}$ | 40 | 2 | 20 | NCLK |
| DEN Enable Time | | $t_{ep}$ | 800 | 480 | 400 | NCLK |

FIGURE F.2 – *Paramètres des signaux de l'écran dans la direction horizontale*

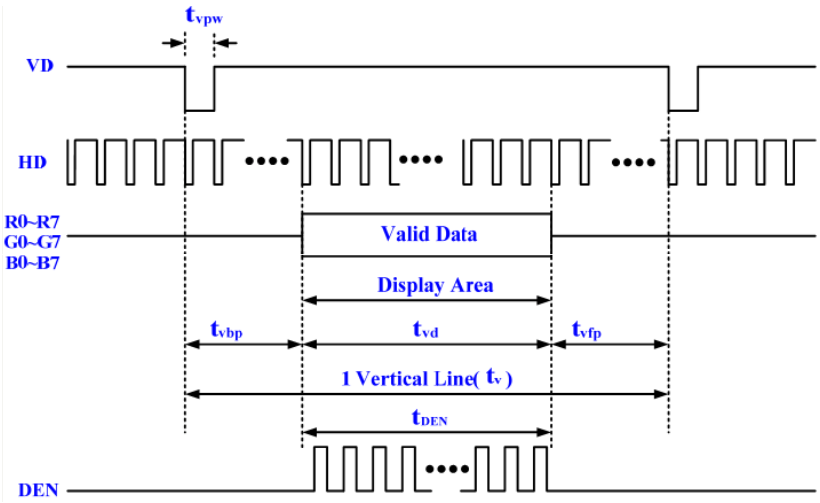



FIGURE F.3 – *Spécification des signaux de l'écran dans la direction verticale*

| Parameter | | Symbol | Panel Resolution | | | Unit |
|---|---|---|---|---|---|---|
| | | | 800xRGBx480 | 480xRGBx272 | 400xRGBx240 | |
| Vertical valid data | | $t_{vd}$ | 480 | 272 | 240 | H |
| Vertical period | | $t_v$ | 525 | 286 | 262 | H |
| VSYNC Pulse Width | Min. | $t_{vpw}$ | 1 | | | H |
| | Typ. | | - | | | |
| | Max. | | - | | | |
| Vertical back porch | | $t_{vbp}$ | 35 | 12 | 20 | H |
| Vertical front porch | | $t_{vfp}$ | 10 | 2 | 2 | H |
| Vertical blanking | | $t_{vb}$ | 45 | 14 | 22 | H |
| DEN Enable Time | | $T_{DEN}$ | 480 | 272 | 240 | H |

FIGURE F.4 – *Paramètres des signaux de l'écran dans la direction verticale*

## Méthode de conception rapide d'architecture massivement parallèle sur puce : de la modélisation à l'expérimentation sur FPGA

**Résumé :** Les travaux présentés dans cette thèse s'inscrivent dans le cadre des recherches menés sur la conception et implémentation des systèmes sur puce à hautes performances afin d'accélérer et faciliter la conception ainsi que la mise en œuvre des applications de traitement systématique à parallélisme de données massif. Nous définissons dans ce travail un système SIMD massivement parallèle sur puce nommé mppSoC : massively parallel processing System on Chip. Ce système est générique et paramétrique pour s'adapter à l'application. Nous proposons une démarche de conception rapide et modulaire pour mppSoC. Cette conception se base sur un assemblage de composants ou IPs. À cette fin, une bibliothèque mppSoCLib est mise en place. Le concepteur pourra directement choisir les composants nécessaires et définir les paramètres du système afin de construire une configuration SIMD répondant à ses besoins. Une chaîne de génération automatisée a été développée. Cette chaîne permet la génération automatique du code VHDL d'une configuration mppSoC modélisée à haut niveau d'abstraction (UML). Le code VHDL produit est directement simulable et synthétisable sur FPGA. Cette chaîne autorise la définition à un haut niveau d'abstraction d'une configuration adéquate à une application donnée. À partir de la simulation du code généré automatiquement, nous pouvons modifier la configuration dans une démarche d'exploration pour le moment semi-automatique. Nous validons mppSoC dans un contexte applicatif réel de traitement vidéo à base de FPGA. Dans ce même contexte, une comparaison entre mppSoC et d'autres systèmes montre les performances suffisantes et l'efficacité de mppSoC.

**Mots clés :** Assemblage d'IPs, FPGA, ingénierie dirigée par les modèles, mppSoC, parallélisme, SIMD, systèmes-sur-puce, traitement intensif de données.

## A rapid design method of a massively parallel System on Chip: from modeling to FPGA implementation

**Abstract:** The main purpose of this PhD is to contribute to the design and implementation of high-performance Systems on Chip to accelerate and facilitate the design and execution of systematic data parallel applications. A massively parallel SIMD processing System-on-Chip named mppSoC is defined. This system is generic, parametric in order to be adapted to the application requirements. We propose a rapid and modular design method based on IP assembling to construct an mppSoC configuration. To this end, an IP library, mppSoCLib, is implemented. The designer can select the necessary components and define the parameters to implement the SIMD configuration satisfying his needs. An automated generation chain was developed. It allows the automatic generation of the corresponding VHDL code of an mppSoC configuration modeled at high abstraction level model (in UML). The generated code is simulable and synthetizable on FPGA. The developed chain allows the definition at a high abstraction level of an mppSoC configuration adequate for a given application. Based on the simulation of the automatically generated code, we can modify the SIMD configuration in a semi-automatic exploration process. We validate mppSoC in a real video application based on FPGA. In this same context, a comparison between mppSoC and other embedded systems shows the sufficient performance and effectiveness of mppSoC.

**Keywords:** IP assembling, FPGA, model-driven engineering, mppSoC, parallelism, SIMD, systems-on-chip, intensive signal processing.