SEEPROC : un modèle de processeur à chemin de données reconfigurable pour le traitement d'images embarqué

Nicolas Roudel

To cite this version:

Nicolas Roudel. SEEPROC : un modèle de processeur à chemin de données reconfigurable pour le traitement d'images embarqué. Autre. Université Blaise Pascal - Clermont-Ferrand II, 2012. Français. NNT : 2012CLF22234 . tel-00864180

HAL Id: tel-00864180
https://theses.hal.science/tel-00864180

Submitted on 20 Sep 2013

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

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

N d'ordre : D. U : 2234
E D S P I C : 561

Université Blaise Pascal - Clermont II
Commissariat à l'énergie atomique - Fontenay-aux-Roses

École doctorale
Sciences pour l'ingénieur de Clermont-Ferrand

Thèse

présentée par :

NICOLAS ROUDEL

pour obtenir le grade de

Docteur d'université

Spécialité : Vision pour la Robotique

SeePROC : Un modèle de processeur à chemin de données reconfigurable pour le traitement d'images embarqué

Thèse soutenue le 18/04/2012 devant le jury composé de

Sébastien Pillement	Président et Rapporteur
Fan Yang	Rapporteur et examinateur
Jocelyn Sérot	Directeur de thèse
François Berry	Examinateur
Laurent Eck	Examinateur

« À mon amour,
Aurélie »
« À mes filles,
Rose et Ambre »

Table des matières

1 Contexte et introduction **17**
1.1 Problématique 18
Organisation du manuscrit 20

2 Traitement d'images et architectures **23**
2.1 Vision artificielle et traitement d'images 23
2.2 Architectures embarquées pour le traitement d'images 25
2.2.1 L'approche architecture intégrée 27
2.2.1.1 les rétines avec traitements dédiés 27
2.2.1.2 Les rétines programmables 30
2.2.2 L'approche système de vision 32
2.2.2.1 Architectures à base de processeurs généralistes . 33
2.2.2.2 Architectures à base de DSP 35
2.2.2.3 Architectures à base de GPU 39
2.3 Architectures pour le traitement d'images à base de FPGAs . . . 41
2.3.1 Présentation des FPGAs 41
2.3.2 Couche de configuration 42
2.3.3 Couche d'interconnexions des FPGAs SRAM 44
2.3.4 Couche logique des FPGAs SRAM 45
2.3.5 Techniques de reconfiguration 46
2.3.5.1 Reconfiguration Statique 48
2.3.5.2 Reconfiguration Dynamique 50
2.3.5.2.1 Reconfiguration dynamique globale . . . 50
2.3.5.2.2 Reconfiguration dynamique locale 51
2.3.6 FPGA et quelques applications de traitement d'images . . 52
2.3.7 Comparaison des performances des FPGAs et des GPUs pour des applications de traitement d'images 56
2.4 Conclusion 59

3 Les processeurs à chemin de données reconfigurable **61**
3.1 Les processeurs à chemin de données reconfigurable 61
3.1.1 Les topologies d'interconnexion 63

3.2 Processeurs à chemin de données reconfigurable et leur technique de programmation . . . 67
3.2.1 Acadia . . . 68
3.2.2 ROMA . . . 69
3.2.3 ConvNet . . . 71
3.2.4 Chameleon CS2000 . . . 73
3.2.5 X.P.P . . . 76
3.2.6 IMAPCAR . . . 78
3.2.7 Systolic Ring . . . 79
3.2.8 DRIP RTR . . . 81
3.2.9 DART . . . 83
3.3 Conclusion . . . 87

4 Architecture proposée : SeeProc 89
4.1 Description fonctionnelle de SeeProc . . . 90
4.2 Architecture du chemin de données . . . 93
4.2.1 Définition et formalisation des unités de traitement . . . 94
4.2.1.1 Exemples d'opérations de voisinage réalisables avec une ALUM . . . 96
4.2.1.1.1 Convolution . . . 96
4.2.1.1.2 Corrélation . . . 97
4.2.1.1.3 Différence d'images . . . 97
4.2.1.1.4 Transformations morphologiques . . . 98
4.2.2 Implantation des ALUs matricielles . . . 100
4.2.2.1 Intégration matérielle . . . 100
4.2.2.2 Le dispositif de reconfiguration du chemin de données 102
4.2.2.3 Accès aux données . . . 103
4.2.2.4 Dimensionnement des bus d'interconnexion . . . 108
4.2.3 Conclusion . . . 109
4.3 Architecture de la partie contrôle . . . 109
4.3.1 L'unité de contrôle . . . 110
4.3.1.1 Gestion des horloges . . . 111
4.3.2 Jeu d'instructions de SeeProc . . . 112
4.3.2.1 Instructions de traitement . . . 112
4.3.2.2 Instructions pour l'interfaçage avec les éléments périphériques . . . 118
4.3.2.3 Instructions de contrôle du programme . . . 121
4.3.3 Conclusion . . . 123
4.4 Conclusion . . . 123

5 Outils de développement pour le processeur Seeproc 125
5.1 Conversion du programme assembleur . . . 127
5.2 Génération de l'architecture du processeur SeeProc . . . 131

5.2.1 Étape d'extraction d'information et de réduction matérielle 131
5.2.2 Étape de génération des éléments VHDL du SeeProc . . . 133
5.2.2.1 Génération des composants élémentaires de l'entité **SeeProc_Decoder** (Phase B) 133
5.2.2.2 Génération des composants élémentaires de l'entité **SeeProc_DPR** (Phase C) 134
5.2.2.3 Génération des composants des unités principales **SeeProc_Decoder** et **SeeProc_DPR** (Phase D et E) 137
5.2.3 Étape d'instanciation matérielle 138
5.3 Environnement de simulation 139

6 Résultats expérimentaux 143
6.1 Plateforme d'expérimentation SeeMOS 143
6.2 Étude des performances de l'architecture SeeProc implantée sur la plateforme SeeMOS 149
6.3 Étude de l'impact des procédés de minimisation 153
6.4 Étude de l'impact de la dimension des fenêtres d'intérêt 155
6.5 Applications réalisées avec le processeur SeeProc 156
6.5.1 Application 1 : Interfaçage de SeeProc avec un processeur de contrôle 157
6.5.1.1 Présentation de l'application 157
6.5.1.2 Implantation matérielle 158
6.5.2 Application 2 : Traitements multi-echelles 161
6.5.2.1 Présentation de l'application 161
6.5.2.2 Implantation matérielle 161
6.5.3 Application 3 : Extraction de points d'intérêt 165
6.5.3.1 Présentation de l'application 165
6.5.3.2 Implantation matérielle 166

7 Conclusion et Perspectives 171

A Présentation de Lex & Yacc 175

B Grammaire du langage assembleur 177

C Correspondance Alias - Valeur Hexadécimale 179

D Comparaison de la syntaxe des fichiers .mif pour Altera et .coe pour Xilinx 183

E Algorithmes de génération des fichiers VHDL du SeeProc 185
E.1 Éléments de la partie de contrôle 185
E.2 Éléments du chemin de données 187

F Présentation de SystemC **191**

G Présentation du processeur SeeCore **195**

Bibliographie **206**

Table des figures

1.1 μDrone d'exploration du CEA LIST 18
1.2 Véhicule autonome *Cycab* du LASMEA 18
1.3 Caméra intelligente BiSeeMOS 19
1.4 Caméra intelligente SeeMOS. 19

2.1 Illustration des différents types d'opérateurs du traitement d'images 25
2.2 Prévision de l'évolution du marché des capteurs d'images (source : IHS iSuppli 2011) 26
2.3 Classification des architectures pour le traitement d'images 27
2.4 Schéma synoptique de l'approche architecture intégrée. 28
2.5 Classification des rétines suivant leur mode d'intégration des traitements. 29
2.6 Illustration de la rétine développé au laboratoire LE2I. 29
2.7 Aperçu du capteur intelligent proposée par Johansson et al. . . . 31
2.8 Schéma synoptique de la rétine programmable proposée par Lin et al. 31
2.9 Schéma synoptique d'un système de vision. 32
2.10 Architecture du microcontrôleur NXP Cortex M4 33
2.11 Architecture des processeurs OMAP 34
2.12 Architecture du système de vision MeshEye 35
2.13 Architecture du DSP C64x de Texas Instruments 36
2.14 Architecture du système de vision M.Bramberger et al 37
2.15 Aperçu du système de vision M.Bramberger et al 37
2.16 Architecture du processeur média TriMedia TM3270 38
2.17 Architecture de stéréo vision proposée par Horst[67] 39
2.18 Architecture des GPU Tesla 40
2.19 Architecture des unités de traitements SIMD des GPU Tesla . . . 41
2.20 Schéma simplifié de l'architecture d'un FPGA 42
2.21 Classification des FPGAs selon leur réseau d'interconnexion. . . . 44
2.22 Schéma synoptique d'une LUT réalisant un *OU Exclusif* à 3 entrées. 46
2.23 Élément logique des FPGAs Altera Cyclone II 47
2.24 Élément logique des FPGAs Xilinx 4000 series 47
2.25 Schéma synoptique d'un élément de calcul de base des FPGAs. . . 48

2.26 Flot de développement sur FPGA. 49
2.27 Différents modes de reconfiguration d'un FPGA. 49
2.28 Reconfiguration statique d'un FPGA 50
2.29 Reconfiguration dynamique d'un FPGA 50
2.30 Résolution temporelle de l'équation $y = a \times x^2 + b \times x + c$ 52
2.31 Résolution spatialle de l'équation $y = a \times x^2 + b \times x + c$ 52
2.32 Caméra rapide du laboratoire Le2i 54
2.33 Résolution spatiale de l'algorithme d'extraction de flux optique . . 55
2.34 Extraction du flux optique suivant la méthode de Lucas et Kanade 55
2.35 Caméra intelligente BiSeeMOS 56
2.36 Performances pour un filtrage 2-D 58
2.37 Performances pour une SAD 58
2.38 Performances pour une classification par K-moyennes 58

3.1 Schéma synoptique simplifié de l'architecture d'une processeur à chemin de données reconfigurable. 62
3.2 Reconfiguration du chemin de données. 62
3.3 Bus simple. 63
3.4 Bus hiérarchique. 64
3.5 Crossbar complet. 65
3.6 Surface d'un crossbar complet. 65
3.7 Crossbar partiel. 66
3.8 Bus à anneaux. 67
3.9 Architecture du processeur Acadia. 68
3.10 Architecture du processeur ROMA. 69
3.11 Architecture du processeur ConvNet. 71
3.12 Architecture de l'ALU vectorielle du ConvNet responsable des calculs de convolution. 72
3.13 Architecture Chameleon CS2000. 73
3.14 Architecture d'un DPU. 74
3.15 Flot de développement sur l'architecture Chameleon. 75
3.16 Architecture X.P.P. 76
3.17 Architecture des éléments de traitement de l'architecture X.P.P. . 77
3.18 Flot de développement sur l'architecture X.P.P. 78
3.19 Architecture du processeur IMAPCar. 78
3.20 Architecture du processeur Systolic Ring. 79
3.21 Exemple d'algorithme pour le processeur Systolic Ring. 81
3.22 Schéma synoptique du DRIP RTR. 81
3.23 Schéma synoptique d'un élément de calcul du processeur DRIP . . 83
3.24 Schéma synoptique d'un élément de calcul minimisé du processeur DRIP 83
3.25 Architecture DART. 83
3.26 Architecture d'un *cluster* de l'architecture DART. 84

3.27 Architecture d'un DPR de l'architecture DART. 85
3.28 Flot de développement associé à l'architecture DART. 86

4.1 Schéma synoptique du processeur *SeeProc* 91
4.2 Schéma synoptique de la partie de traitement du processeur *Seeproc* 93
4.3 Schéma synoptique d'une ALU matricielle 95
4.4 Exemple de dilatation et d'érosion binaire avec 2 éléments structurants 99
4.5 Intégration matérielle des ALUMs 100
4.6 Schéma synoptique d'une ALUM intégrant la gestion dynamique de la gamme des pixels résultats 101
4.7 Schéma synoptique d'un Crossbar 102
4.8 Principe de mise en œuvre d'un SPG 105
4.9 Principe de mise en œuvre d'un SPG 105
4.10 Schéma synoptique du Seeproc_PixelGrabber pour une dimension de 3 et une largeur d'image de 7 pixels à un instant T puis de 5 pixels à un instant T+1 107
4.11 Exemple de concaténation pour l'adaptation des dimensions de bus 109
4.12 Exemple de sélection du cœur pour l'adaptation des dimensions de bus 109
4.13 Schéma synoptique de la partie de contrôle et d'ordonnancement du *Seeproc* 110
4.14 Exemple de schéma synoptique l'unité de contrôle 111

5.1 Flot de développement du *Seeproc* 126
5.2 Conversion du programme assembleur 127
5.3 Génération de l'architecture du processeur SeeProc 131
5.4 Distribution des informations pour la génération de l'architecture du processeur SeeProc 132
5.5 Schéma synoptique de la génération des éléments de base de SeeProc_Decoder 134
5.6 Schéma synoptique de la génération des éléments de base de SeeProc_DPR 137
5.7 Schéma synoptique de la génération des éléments SeeProc_DPR et SeeProc_Decoder 138
5.8 Exemple d'instanciation matérielle du SeeProc l'environnement de développement Quartus II 139
5.9 Principe d'utilisation du modèle SystemC du processeur SeeProc . 141

6.1 Prototype de la caméra intelligente SeeMOS. 144
6.2 Caméra intelligente SeeMOS. 146
6.3 Caméra intelligente SeeMOS. 146
6.4 Schéma synoptique de la caméra intelligente SeeMOS. 147

6.5 Illustration des différentes cartes composant la plateforme SeeMOS, ainsi que de sa conception modulaire. 148
6.6 Illustration de la scène de référence pour les applications. 157
6.7 Illustration du protocole de communication *handshaking*. 158
6.8 Résultats graphiques de l'application 1 160
6.9 Masques de traitements pour l'application 2. 161
6.10 Résultat graphique de l'érosion. 163
6.11 Résultat graphique du gradient horizontal. 164
6.12 Résultat graphique du filtre gaussien. 164
6.13 Décomposition de l'algorithme de Harris et Stephen. 167
6.14 Résultat graphique de l'application 3. 169

A.1 Exemple d'analyse lexicale et grammaticale d'un programme avec LEX et YACC. 176

F.1 Architecture logiciel de SystemC 191

G.1 Schéma synoptique du SeeCORE. 195
G.2 Schéma synoptique du flot d'implémentation du processeur SeeCORE 197

Liste des tableaux

2.1 Comparaison des performances entre un FGPA, un GPU et un CPU pour l'exécution d'une convolution 2D en MP/s. 57
2.2 Tableau comparatif des composants de calcul pour le traitement d'images 60

3.1 Tableau de comparaison entre diverses architecture de PCDR. . . 87

4.1 Format des instructions 93
4.2 Liste des possibilités de traitements d'une ALU matricielle. 95
4.3 Liste des opérations réalisables par **FD**, **FM** et **FR**. 96
4.4 Format des instructions 101
4.5 Exemple de possibilités de connexion au Crossbar avec un mot de contrôle de 72 bits 103
4.6 Instructions de traitement 113
4.7 Opcodes des opérations réalisables par **FD**, **FM** et **FR**. 113
4.8 Format de l'instruction LOAD 113
4.9 Exemple de l'instruction de traitement : LOAD 114
4.10 Format de l'instruction COEF 115
4.11 Format de l'instruction ROUT 116
4.12 Exemple de l'instruction de traitement : CFG IMW 117
4.13 Exemple de l'instruction de traitement : CFG RANGE 118
4.14 Instructions d'interfaçage du processeur SeeProc 118
4.15 Format des instructions WAIT, SET et RESET 119
4.16 Liste des différents paramètre de l'instruction ADDR 119
4.17 Format de l'instruction ADDR pour les paramètres START, STOP et RES 120
4.18 Format de l'instruction ADDR pour les paramètres BASE, SIZE et MODE 120
4.19 Instructions de contrôle du programme 121
4.20 Format de l'instruction JUMP 121
4.21 Liste des différentes conditions de l'instruction IFRES 121
4.22 Format de l'instruction IFRES pour les conditions BPOS, BNEG et BZERO 122

4.23 Format de l'instruction IFRES pour les conditions SUP, INF et EQU 122
4.24 Format de l'instruction TEMPO 122

5.1 Format des instructions du SeeProc 128
5.2 Génération des fichiers VHDL de la partie **SeeProc_Decoder** du processeur SeeProc 133
5.3 Génération des fichiers VHDL de la partie **SeeProc_DPR** du processeur SeeProc 135
5.4 Algorithme de création du Crossbar 136

6.1 Caractéristiques du dispositif FPGA intégré dans la SeeMOS . . . 145
6.2 Évaluation des performances de diverses architecture 149
6.3 Programme assembleur pour un filtre laplacien 3×3 150
6.4 Programme assembleur pour une convolution avec un masque de dimension 5×5 151
6.5 Programme assembleur pour une convolution avec un masque de dimension 7×7 151
6.6 Programme assembleur pour une convolution avec un masque de dimension 8×8 152
6.7 Programme 1 de test pour évaluer l'impact des procédés de minimisation 153
6.8 Programme 2 de test pour évaluer l'impact des procédés de minimisation 154
6.9 Résultats de Synthèse du programme MiN1 avec et sans minimisation 154
6.10 Résultats de Synthèse du programme MiN2 avec et sans minimisation 155
6.11 Impact de la dimension de la fenêtre d'intérêt 156
6.12 Résultats de Synthèse de l'application 1 159
6.13 Résultats de Synthèse de l'application 2 165
6.14 Extraction du gradient horizontal 168
6.15 Extraction du gradient vertical 168
6.16 Calcul des primitives 168
6.17 Résultats de Synthèse de l'application 3 170

A.1 Exemples d'expressions régulières 175

C.1 Correspondance Opcodes 179
C.2 Correspondance Flags d'entrée 179
C.3 Correspondance Flags de sortie 180
C.4 Correspondance des ALUMs et de leurs opérations 180
C.5 Correspondance des ports d'entrée du Crossbar 181
C.6 Correspondance des ports de sortie du Crossbar 182

D.1 Syntaxes fichiers MIF et COE 183

E.1 Algorithme de création de la mémoire programme 185
E.2 Algorithme de création de l'unité de contrôle 185
E.3 Algorithme de création du SeeProc_Decoder 186
E.4 Algorithme de création du compteur programme 186
E.5 Algorithme de création du module d'adressage 187
E.6 Algorithme de création du registre 187
E.7 Algorithme de création des SPG 187
E.8 Algorithme de création des ALUMs 188
E.9 Algorithme de création du SeeProc_DPR 189

Chapitre 1

Contexte et introduction

L'intérêt de la vision artificielle n'est, aujourd'hui, plus à démontrer, ne serait-ce que par l'attrait qu'elle procure en terme de richesse d'informations. Cependant cette richesse d'informations implique aussi une chaîne de traitements adaptés à la complexité inhérent à cette richesse. En effet, la quantité d'informations contenue dans le signal vidéo et la complexité des traitements requièrent des ressources de calculs colossales. Cette exigence a conduit les chercheurs et les ingénieurs à développer des machines dédiées à la perception visuelle. Le but est d'accroître la capacité de calcul des systèmes de vision à partir d'une adéquation entre la problématique de vision et la cible d'implantation matérielle. Ces machines dédiées se basent sur une organisation matérielle qui permet d'améliorer l'efficacité des unités de traitements constituant le système de vision. Pour mettre en œuvre cette optimisation, divers unités de traitements sont délocalisées le long de la chaîne de perception au sein de dispositifs dédiés. Cette délocalisation permet d'exploiter des systèmes de traitement conçus pour implémenter efficacement un algorithme spécifique.

Cet aspect s'avère encore plus crucial, voire critique, lorsque l'implémentation de ces algorithmes se fait sur des cibles embarquées et nécessite un traitement des données en temps réel. Les Fig. 1.1 et Fig. 1.2 sont de parfaits exemples démontrant l'importance, pour ces applications de navigation, d'avoir à disposition les informations nécessaires au bon moment afin d'assurer leur propre sécurité et celle de leur environnement.

A l'heure actuelle, la plupart des architectures mises en œuvre dans ces types de projets, sont souvent basées sur des ordinateurs de types PC. Ainsi, les données visuelles sont acquises par une ou plusieurs caméras et les images sont directement transmises vers le PC hôte. Cette transmission de gros volumes de données est un problème récurrent bien connue dans les systèmes de vision. Une solution permettant de répondre à ce problème, et qui constitue le cadre applicatif de cette thèse, est l'utilisation d'un système de vision embarqué de type **caméra intelli-**

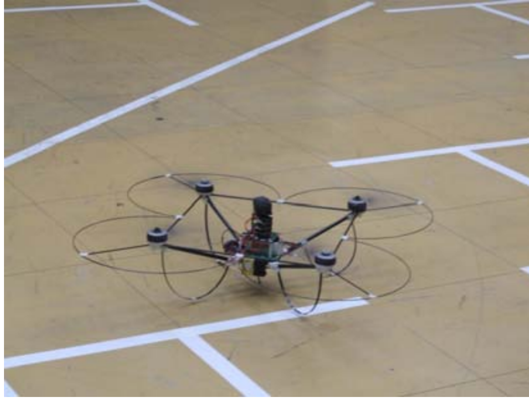


FIGURE 1.1 – μDrone d'exploration du CEA LIST

FIGURE 1.2 – Véhicule autonome *Cycab* du LASMEA

gente ou *smart camera*. L'un des objectifs des caméras intelligentes est d'offrir la possibilité d'extraire des primitives (flux optique, détection de coin, transformée de Hough, etc...) de la scène observée directement au sein du système d'acquisition. Une fois ces primitives calculées, elles sont ensuite envoyées vers le PC hôte réduisant ainsi notablement le volume de données transmis.

1.1 Problématique

Afin de répondre au besoin d'extraction de primitives au niveau du capteur, plusieurs systèmes de caméras intelligentes ont été développés au sein du LASMEA. Les Fig. 1.3 et Fig. 1.4 illustrent deux exemples de ces systèmes. Le point commun entre les différents systèmes est que leur architecture s'articule autour d'un dispositif de traitement reconfigurable de type FPGA. Ce type de composant autorise une forte parallélisation, lorsque cela est possible, des traitements. Les FPGAs permettent également de contrôler chaque élément de l'architecture afin d'optimiser la performance de chacun d'entre eux. L'inconvénient majeur des FPGAs est leur programmabilité qui nécessite de fortes compétences en électronique numérique ainsi qu'une connaissance des langages de programmation HDL.

Plusieurs travaux ont été réalisés sur ces plateformes dans le but de simplifier leur utilisation. Dans ce sens, au cours de ses travaux de thèse, F. Dias Real de Oliveira [20] a développé une interface de programmation matérielle (HPI) permettant le contrôle, de manière simple, des différents organes d'une caméra intelligente à base de FPGA.

La problématique de nos travaux est de permettre **l'implémentation, de manière simple, de traitements d'images en temps réel et en optimi-**

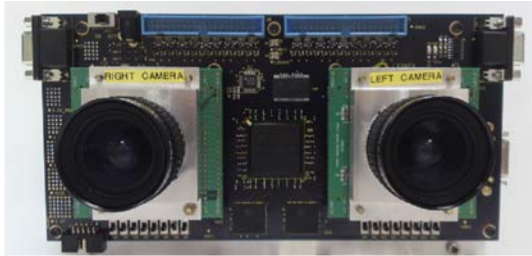


FIGURE 1.3 – Caméra intelligente BiSeeMOS

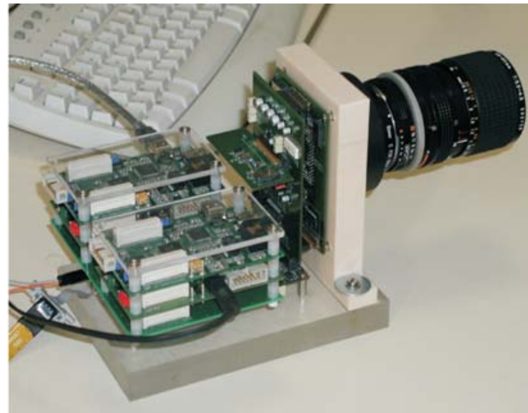


FIGURE 1.4 – Caméra intelligente SeeMOS.

sant les ressources matérielles nécessaires à leur exécution.

Cette problématique peut se scinder en deux aspects. Le premier aspect est un problème d'optimisation matérielle et consiste en la mutualisation des ressources matérielles pour avoir un chemin de données optimal. En effet, on sait que lors du déroulement d'un programme certaines opérations sont utilisées uniquement à un moment t et d'autres uniquement à un instant $t+1$. Il est donc intéressant, en terme de ressources matérielles, de trouver une factorisation matérielle de ces opérations dédiées au traitement de l'image bas niveau afin d'obtenir des blocs génériques de traitement de granularité plus importante que la granularité des blocs de traitement de base d'un FPGA. Ces blocs peuvent ainsi être configurés et reconfigurés dynamiquement selon les besoins de l'application.

Le second aspect est lié à la programmabilité intrinsèque des dispositifs FPGAs. En effet, si les FPGAs sont attractifs de par leur faible coût, leurs hautes performances ou encore leur flexibilité, le principal frein à leur utilisation, hormis leur consommation, reste leur programmation, qui se fait encore essentiellement avec des langages "bas niveau" -entendre ici niveau RTL- comme VHDL[91] ou Verilog[92]. La simplification du développement d'applications sur FPGAs réside alors en la conception d'outils permettant à un utilisateur de décrire son programme dans un langage plus haut niveau générant automatiquement les fichiers HDL nécessaires à l'exécution de son application.

La mise en œuvre de traitements **dédiés au traitement d'images** avec les notions inhérentes à ce domaine, à savoir parallélisme, reconfiguration dynamique en temps réel ou encore optimisation de l'occupation matérielle, conjugués à **une programmation intuitive** reste un challenge et motive la définition du nouveau type de processeur présenté dans ce manuscrit.

Les objectifs qui découlent des problématiques présentées précédemment peuvent être scindés en cinq points comme suit :

Facilité d'utilisation : La programmation matérielle en VHDL/Verilog nécessitant de larges compétences en électronique numérique, une interface de programmation matérielle, plus accessible aux programmeurs *logiciel* et leur rendant transparent ces aspects matériels, doit être mise au point.

Flexibilité : Selon les besoins de l'utilisateur et/ou de l'environnement, l'application doit être capable d'évoluer, de manière fonctionnelle, dynamiquement, d'où la nécessité d'une solution flexible.

Performance : Il faut que la solution proposée puisse être capable d'exécuter des applications de traitement de l'image bas niveau en temps réel.

Faible occupation des ressources matérielles : Le traitement d'image bas niveau n'étant qu'un préambule de l'application finale, il est primordial que les ressources requises pour ces traitements soient optimisées et minimales.

Portabilité : Aux vues du nombre grandissant d'architectures de vision basées sur un FPGA, il nous a paru essentiel que l'approche développée soit portable sur n'importe quel composant FPGA.

Organisation du manuscrit

La suite de ce document se décompose en six chapitres qui se répartissent comme suit :

Le chapitre 2 - Traitement d'images et architectures - s'attache à présenter des notions générales sur le domaine du traitement d'images. Il est également proposé un état de l'art sur les diverses architectures et composants de traitement capable, exclusivement ou non, d'exécuter ce type de traitement. Il est ainsi décrit des systèmes à base de processeurs généralistes (CPU), de processeurs de traitement du signal (DSP), de processeurs graphiques (GPU) et de composants reprogrammables (FPGA).

Le but de ce chapitre, au delà de l'introduction de notions générales sur le traitement d'images et les architectures associées, est de mettre en exergue que les dispositifs FPGA sont particulièrement adaptés au domaine du traitement d'images.

Le chapitre 3 - Les processeurs à chemin de données reconfigurable - présente le principe de processeurs à chemin de données reconfigurable. On explore notamment les diverses topologies d'interconnexion existantes pour terminer sur un état de l'art des processeurs à chemin de données reconfigurable.

Le but de ce chapitre est de lister les solutions existantes pour montrer un manque de solutions dédiées spécifiquement au traitement d'images. L'autre objectif de ce chapitre est de pouvoir extraire de cet existant des éléments pertinents afin de proposer une architecture pouvant répondre aux attentes présentées précédemment et de présenter les aspects méthodologiques, s'ils existent, inhérents à chacun d'entre eux

Le chapitre 4 - Architecture proposée : SeeProc - présente l'architecture du **SeeProc** développé durant cette thèse. SeeProc est un processeur à chemin de données reconfigurable comprenant notamment des unités de traitement dédiées au domaine du traitement d'images.

En premier lieu, la partie de traitement est détaillée. Construite autour d'un réseau d'interconnexion de type Crossbar, cette partie intègre un ensemble d'éléments de calculs reconfigurable dédiés au domaine du traitement d'images en temps réel.

Ensuite l'architecture de la partie contrôle est présentée. Les éléments matériels ainsi que le jeu d'instruction associé à cette partie sont exposés.

Le chapitre 5 - Outils de développement pour le processeur Seeproc - explique et détaille les outils de développement associés au processeur SeePROC. Ces outils proposent au programmeur une phase de simulation grâce à un modèle SystemC, permettant de pré-visualiser les résultats des applications développées, et une phase d'implémentation grâce à un langage assembleur et à un compilateur permettant d'obtenir les fichiers HDL optimisés et minimisés du processeur.

Le chapitre 6 - Résultats expérimentaux - présente une évaluation des performances de l'architecture proposée comparativement à des solutions existantes. Différentes applications développées sur le processeur SeePROC sont ensuite présentées afin d'évaluer l'efficacité à la fois au niveau performance des traitements mais également au niveau de la pertinence des outils de développement proposés.

Le chapitre 7 - Conclusion et perspectives - termine ce manuscrit, en explicitant les solutions apportées permettant de répondre à nos problématiques et objectifs. Enfin des perspectives de travaux futurs sont exprimées.

Chapitre 2

Traitement d'images et architectures

Ce chapitre a pour objectif d'introduire la vision artificielle dont un de ses sous-ensembles, le traitement d'images, constitue le domaine d'applications de nos travaux. Il est ensuite présenté différentes architectures, basées sur divers composants, qui permettent de réaliser ce type de traitements. On étudie notamment les architectures à base de processeurs DSP [1] ou GPU [2].

Une section est ensuite dédiée aux architectures à base de composants FPGA [3] qui ont l'avantage d'être particulièrement adaptés au domaine du traitement d'images.

2.1 Vision artificielle et traitement d'images

La vision artificielle est généralement composée d'une chaîne de traitements que l'on peut scinder en trois étapes :

Le traitement d'images (*Image Processing*) : Concerne les traitements où les données sont de type image en entrée et image en sortie. Ils s'apparentent à des filtrages (élimination de bruit,...) et/ou à des transformées (transformée de Fourier, transformée en ondelettes...).

La reconnaissance de formes (*Pattern recognition*) : Dans ce domaine d'application, les données d'entrée sont les images (résultats ou non du traitement d'images), tandis que les sortie seront des primitives pouvant être géométrique, de texture, morphologique...

1. DSP : Digital Signal Processor
2. GPU : Graphics Processing Unit
3. FPGA : Field programmable Gate Array

La vision par ordinateur (*Computer Vision*) : Ce dernier domaine comporte tous les algorithmes décisionnels que ce soit de la reconnaissance, de la classification ou bien encore des lois de commande pour agir sur des actionneurs robotiques.

Le traitement d'images traite ainsi de large quantités de données représentant une image numérique. La structure de base des données utilisées en traitement d'images est un tableau à 2 dimensions représentant la distribution spatiale de l'intensité lumineuse (en niveau de gris ou en couleur). Le volume de données manipulé, et donc le calcul à effectuer, dépend ainsi de la taille de l'image et est augmenté lors des traitements de séquences temps réel.

Classiquement, les opérateurs de traitement d'images peuvent être divisés en deux types (Fig. 2.1) :

– Les opérateurs dit "pixel" dont le résultat ne dépend que de la valeur d'entrée au même pixel comme un ajustement de l'intensité lumineuse, le passage d'un espace colorimétrique à un autre (RGB vers YUV par exemple), une différence d'image ou encore des opérations logiques (or, and, xor etc...),

– les opérateurs locaux dont le résultat en un pixel dépend du même pixel en entrée ainsi que du voisinage de ce pixel. On retrouve dans ce type d'opérations les convolutions permettant d'effectuer des filtrages d'image (gaussien, laplacien, gradients etc...), des transformations morphologiques (érosion, dilatation) ou encore des corrélations.

Ainsi, les calculs associés au traitement d'images sont répétitifs et peuvent être réalisés soit sur une partie de l'image, soit sur l'image entière. En effet, dans ces traitements les opérations sont souvent simples et s'appliquent le plus souvent sur des groupes de pixels adjacents comme par exemple la convolution ou la corrélation. Ces opérations de voisinage requièrent donc de nombreux accès mémoire et peuvent rapidement devenir de véritables goulots d'étranglement si le système matériel n'est pas correctement dimensionné. Ce sont des traitements déterministes en temps admettant une faible dépendance de données, ce qui facilite grandement leur parallélisation. Ils sont par conséquent plus en adéquation avec des processeurs de type SIMD (Single Instruction Multiple Data) de la taxonomie de Flynn[82] ou des composants reconfigurables tels que les FPGA. L'exploitation du parallélisme potentiel est alors une opportunité, voire une nécessité. De plus, ces traitements devant souvent être réalisés en temps-réel, la parallélisation apparaît alors comme une nécessité.

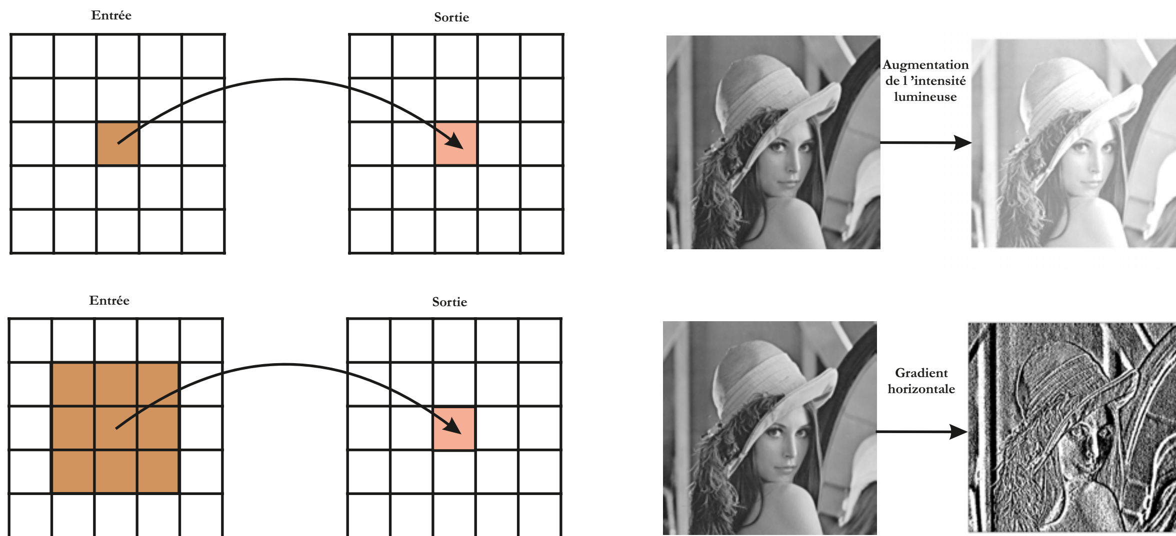


FIGURE 2.1 – Illustration des différents types d'opérateurs du traitement d'images

Par opposition à ces opérations "de masse", les phases de reconnaissance de formes ou de vision par ordinateur (qualifiées de moyen niveau et haut niveau) requièrent souvent des étapes algorithmiquement plus complexes telles que des optimisations ce qui se traduit par des opérations itératives et/ou récursives et surtout non régulières. Ces étapes se révèlent alors être des challenges calculatoires en particulier dans les contextes embarqués où il n'est souvent pas possible de disposer de la puissance de calcul classique des ordinateurs et surtout de leur programmabilité. Les traitements dit de plus haut niveau, nécessitent des algorithmes plus complexes tout en travaillant sur de plus faibles quantités de données. Des processeurs de type DSP sont alors privilégiés, ces derniers admettant une bonne programmabilité tout en gardant des optimisations matérielles pour les calculs de masse.

2.2 Architectures embarquées pour le traitement d'images

L'industrie de l'imagerie utilise principalement deux technologies pour fabriquer des capteurs d'images : la technologie CCD et la technologie CMOS. La technologie CCD (à transfert de charges) a dominé le marché des capteurs pendant plus de trente ans. Cette technologie met en œuvre des procédés de fabrication particuliers largement maîtrisés. L'atout majeur des capteurs CCD est de fournir des images de haute résolution avec un bruit très faible. Grâce aux fortes évolutions des processeurs généralistes basés sur la technologie CMOS, les

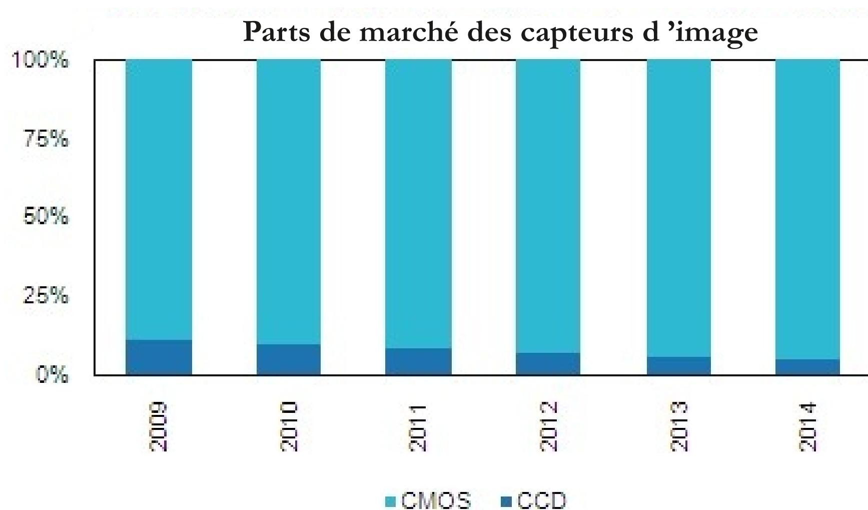


FIGURE 2.2 – Prévision de l'évolution du marché des capteurs d'images (source : IHS iSuppli 2011)

capteurs utilisant cette technologie ont vu leur performances grandir à tel point que ces capteurs concurrencent, et même dépassent en terme de parts de marché (Fig. 2.2), leur homologue CCD. Du fait des possibilités croissantes d'intégration les capteurs CMOS sont devenus incontournables dans divers domaines comme par exemple la téléphonie mobile.

Les capteurs CMOS possèdent plusieurs avantages par rapport aux capteurs CCD vis-à-vis du traitement d'images. L'un des plus remarquable est que, à l'inverse des capteurs CCD, les capteurs CMOS offrent, théoriquement, un accès aléatoire à chaque pixel de la zone photosensible. Ceci permet la sélection d'une partie de l'image en permettant d'augmenter les cadences d'acquisition. L'autre avantage majeur des capteurs CMOS consiste en la possibilité d'intégrer des traitements plus ou moins avancés soit au niveau de chaque pixel soit après la chaîne de conversion analogique/numérique.

Parmi l'état de l'art, on distingue deux solutions architecturales pour le traitement d'images. La première solution, qualifiée d'architecture intégrée ou capteur intelligent, consiste en l'intégration de traitements au sein du même composant que la zone photosensible. La seconde solution, les systèmes de vision et plus précisément les *smart cameras*, est l'association d'un imageur avec un composant de traitements déporté. La Fig. 2.3 présente ces solutions et les diverses options inhérentes à chacune.

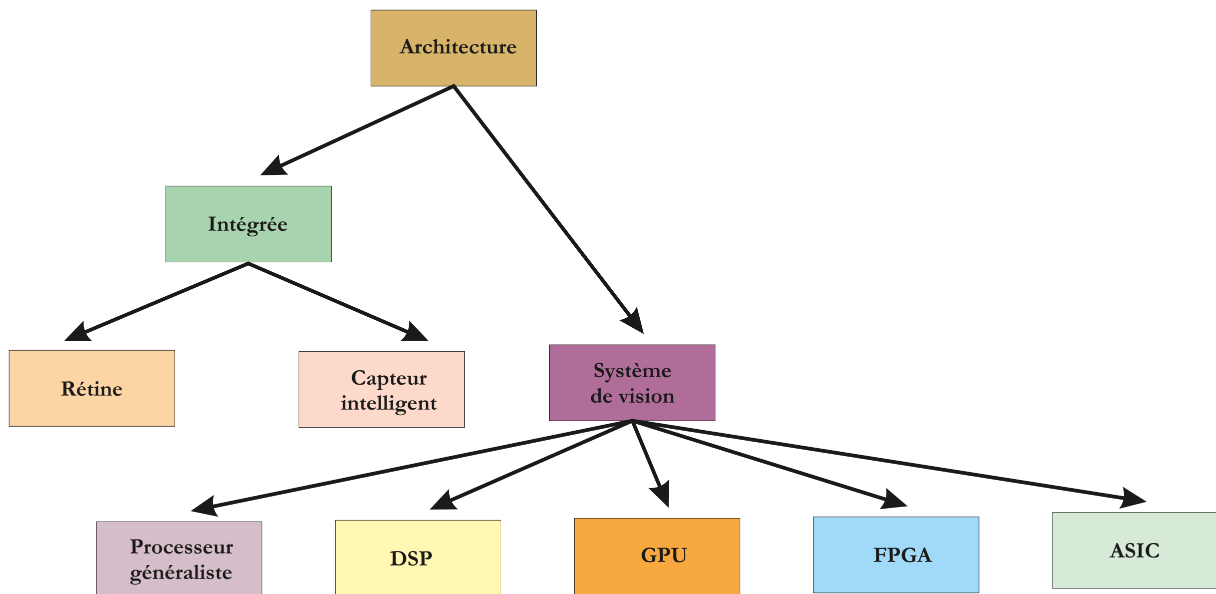


FIGURE 2.3 – Classification des architectures pour le traitement d'images

2.2.1 L'approche architecture intégrée

La Fig. 2.4 présente le schéma synoptique de l'approche architecture intégrée. Dans ce cas, une seule puce est utilisée pour à la fois réaliser l'acquisition de l'image et les traitements d'images. La partie de traitement peut être intégrée de façons différentes. Elle peut être réalisée de manière analogique sur la zone photosensible ou de manière numérique après la conversion en signal numérique du signal analogique résultant de la zone photosensible.

2.2.1.1 les rétines avec traitements dédiés

Les rétines avec traitements dédiés permettent d'optimiser la surface en silicium nécessaire à l'exécution de tâches de traitement d'images. En contrepartie, les possibilités de traitements sont figées et ne peuvent être modifiées ou enrichies.

On trouve dans l'état de l'art un grand nombre de rétines visant une large gamme de traitements d'images. Trois grandes classes de rétines peuvent être définies selon le mode d'insertion des traitements (Fig. 2.5) :

- Le mode traitement par pixel. Dans cette architecture, chaque pixel dispose de son propre circuit de traitement.
- Le mode traitement par colonne. Dans cette architecture, un circuit de traitement est disponible pour chaque colonne de la matrice photosensible.

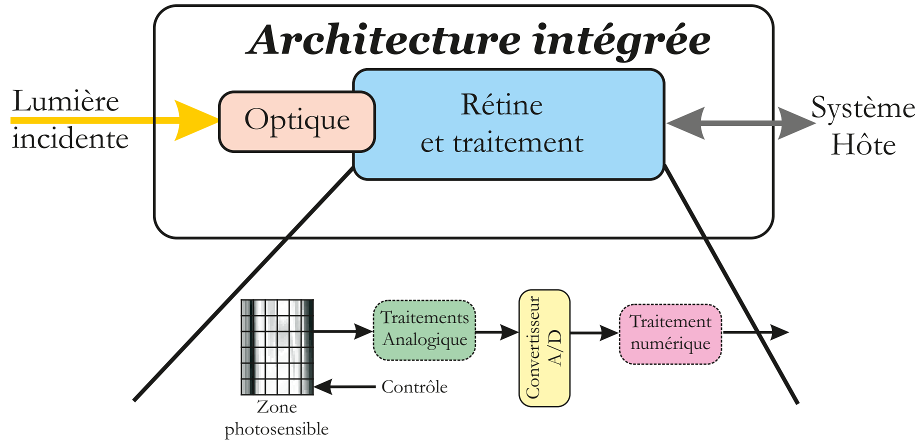


Figure 2.4 – Schéma synoptique de l'approche architecture intégrée.

– Le mode traitement par matrice entière. Dans ce dernier cas, un seul circuit de traitement est disponible pour le capteur.

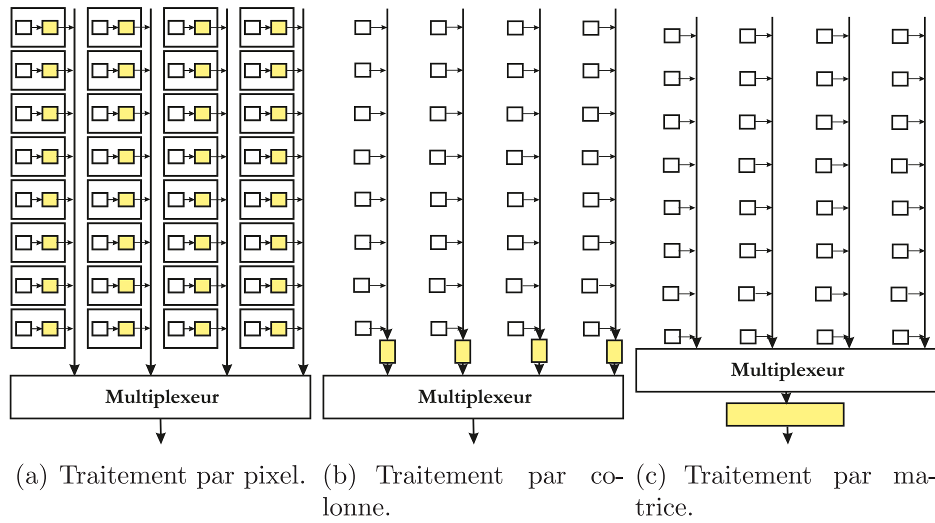
Chaque mode de traitement est adapté à un certain spectre d'algorithmes. Ainsi, les traitements par matrice sont particulièrement efficaces pour des algorithmes dont la sortie d'un pixel ne dépend que de l'entrée de ce dernier. À contrario, les traitements par pixel et colonne sont plus efficaces sur des algorithmes spatiales ou spatio-temporels.

Le point commun entre ces approches est le fait que, dans la plupart des cas, la partie de traitements est réalisée de manière analogique. En d'autres termes, les concepteurs ont ajouté de l'électronique de traitement au sein même du circuit d'acquisition de chaque pixel avant le bloc de conversion analogique/numérique.

À titre d'exemple Chen et al. [78] propose une rétine de résolution 64×64 pixels capable d'effectuer une différence d'images ou encore Ni et al. [43] qui applique une égalisation d'histogramme toujours sur une rétine de résolution 64×64 pixels. Les références [69, 50, 86] proposent des rétines dans lesquelles une partie analogique permet une détection de contours. On peut également noter les divers travaux réalisés sur ce type de rétine au sein du laboratoire LE2I de l'université de Bourgogne avec par exemple la référence [33], qui propose une rétine (Fig. 2.6) de résolution 64×64 avec une cadence d'image de 10.000 IPS [4] pouvant réaliser des traitements comme une extraction de gradients et des convolutions avec par exemple l'application de filtre de Sobel ou de Laplace.

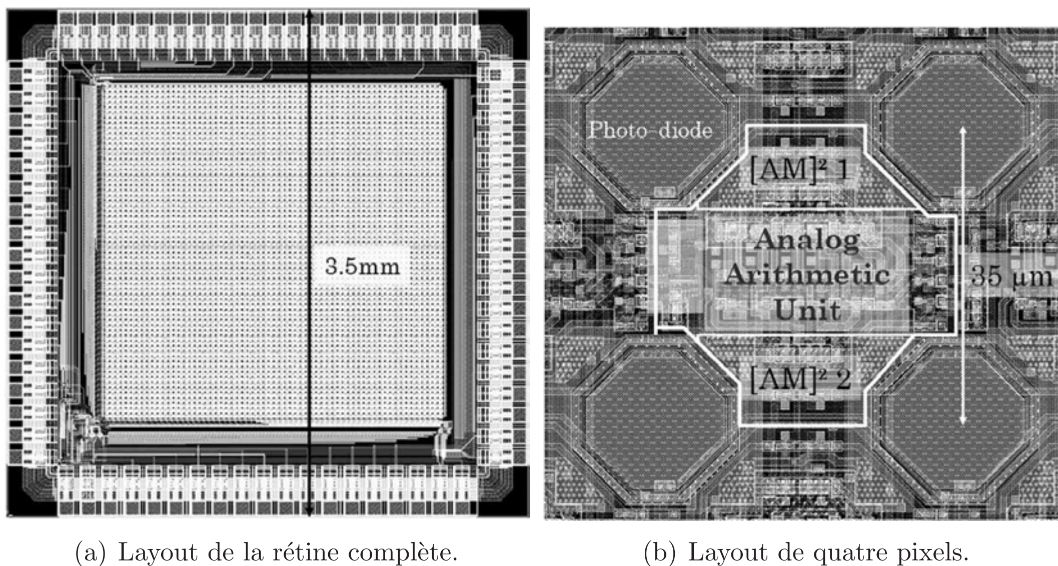
La force des rétines présentées ci-dessus est le parallélisme massif offert par

4. IPS pour Images Par Seconde



(a) Traitement par pixel. (b) Traitement par colonne. (c) Traitement par matrice.

FIGURE 2.5 – Classification des rétines suivant leur mode d'intégration des traitements.



(a) Layout de la rétine complète. (b) Layout de quatre pixels.

FIGURE 2.6 – Illustration de la rétine développé au laboratoire LE2I.

ces dernières, dans le cas du mode de traitement au niveau pixel ou colonne, ce qui est en adéquation avec le traitement d'images. Néanmoins, le fait d'avoir des traitements figés limite fortement l'utilisation de ce genre d'approche. De plus, le taux de remplissage des pixels (le *Fill factor*[5] en anglais) de ces rétines est généralement faible (de l'ordre de 10%) ce qui rend ces rétines peu sensibles et limite leur résolution.

2.2.1.2 Les rétines programmables

Une rétine programmable est un seul et unique composant dans lequel on trouve une partie photosensible, une interface de conversion analogique numérique, une partie de traitements (analogique, numérique ou les 2) et un bus de communication. Parmi les rétines programmables intégrant une partie de traitement numérique, on peut par exemple citer la rétine VCS-IV [68] qui est une matrice (64×64) de processeurs SIMD programmables pouvant réaliser divers traitements d'images tels que la détection de contours, le lissage ou le filtrage. Johansson et al.[54] proposent également un capteur intelligent de résolution 1536×524 pixels intégrant une matrice de 1536 processeurs SIMD. Ce capteur (Fig. 2.7) est capable d'effectuer divers traitements d'images tels que du filtrage gaussien ou de Sobel. Lin et al. [77] propose une rétine programmable fonctionnant à 1000 IPS avec une résolution d'image de 64×64, dont le schéma synoptique est donné en Fig. 2.8. Cette rétine permet d'effectuer des opérations de morphologie mathématique comme l'érosion et la dilatation en niveau de gris et en binaire.

Il existe également des rétines programmables dont la partie de traitement est analogique. La rétine SCAMP-3 [23] est une matrice de processeurs qui implémente une variété de traitements d'images bas niveaux à haute cadence (1000 IPS[6]). Il existe une réalisation où des processeurs de traitements d'images "MIMD IP" (Multiple Instruction Multiple Data Image Processing) [81] permettent d'exécuter à haute cadence (9600 IPS) des convolutions locales utilisant des noyaux de dimensions variables (de 3×3 à 11×11). La configuration, la dimension et les coefficients du masque sont programmables.

Enfin la rétine programmable ACE16k intègre une matrice d'éléments de traitements (PEs) mixtes (analogiques et numériques), suivant le principe des processeurs SIMD, organisée en réseaux de neurones cellulaires CNN[7]. Chaque PE permet, par exemple, le calcul d'une convolution de dimension 3×3 avec un

5. Le fill factor définit le rapport entre la surface du pixel et la surface réellement utilisée pour produire l'image soit la partie photodiode. Plus ce ration est proche de 100% plus le pixel (donc le capteur) est sensible

6. IPS pour Images Par Seconde

7. CNN pour Cellular Neural Network

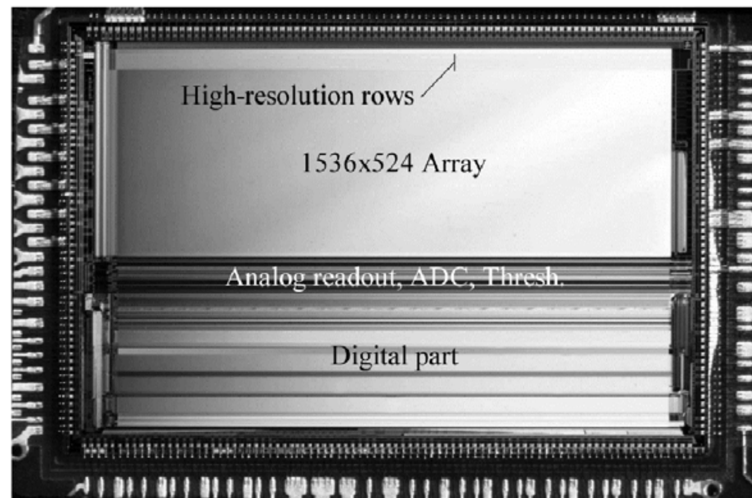


FIGURE 2.7 – Aperçu du capteur intelligent proposée par Johansson et al.

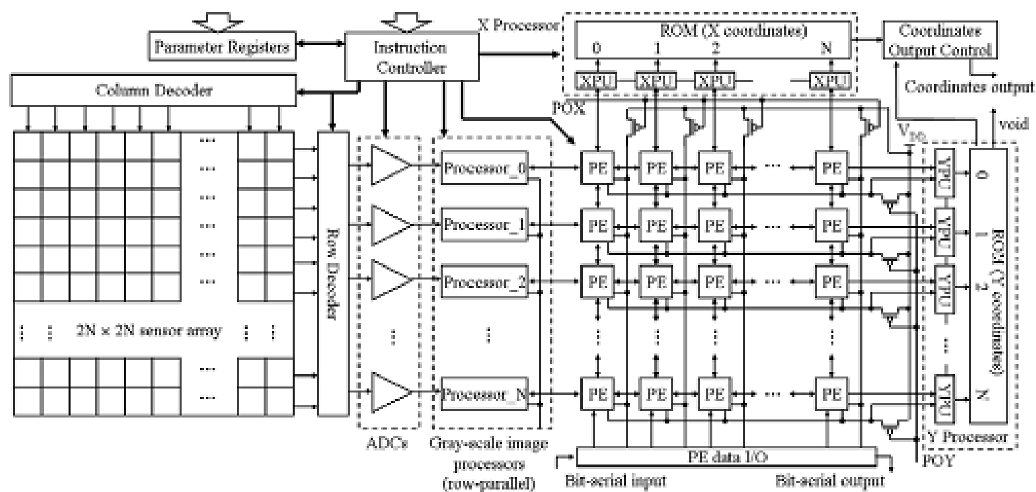


FIGURE 2.8 – Schéma synoptique de la rétine programmable proposée par Lin et al.

masque reconfigurable.

L'atout principal des capteurs intelligents est le fait, qu'à l'inverse des rétines précédemment présentées, ils peuvent intégrer un plus large spectre d'applications. L'inconvénient de ces architectures est qu'elles impliquent un grand nombre de transistors donc une grande surface de pixel ou un boîtier de grande dimension. Or même si le fait de réaliser des traitements directement sur la rétine est intéressant, la taille du pixel/boîtier est généralement trop important ce qui limite la résolution de ces capteurs ou donne lieu à des composants de dimensions imposantes.

2.2.2 L'approche système de vision

La Fig. 2.9 illustre l'architecture minimale d'un système de vision. Un système de vision est ainsi composé, au moins, d'un imageur de technologie CCD ou CMOS qui envoie des images via un bus de communication à un composant dans lequel des traitements d'images sont réalisés.

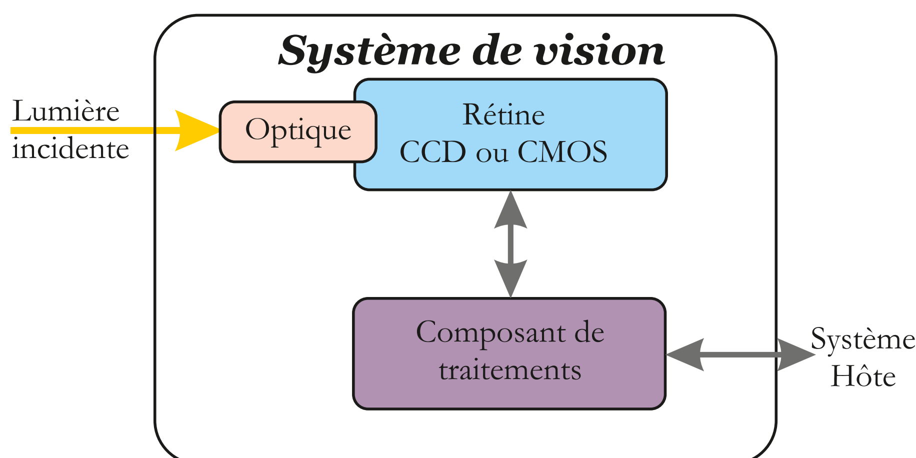


FIGURE 2.9 – Schéma synoptique d'un système de vision.

Cette section a pour but de présenter les diverses architectures de systèmes de vision à base de composants **programmables** que l'on trouve dans la littérature. À ce titre, nous présentons des architectures à base de processeurs généralistes, mediaprocesseurs, DSP ou encore GPU. À l'instar des rétines intelligentes câblées, on trouve également des systèmes de vision à base de composants câblés qui sont généralement des composants ASIC. À titre d'exemple, les références [19, 57] proposent des systèmes de vision à base d'ASIC pour de l'interaction avec un

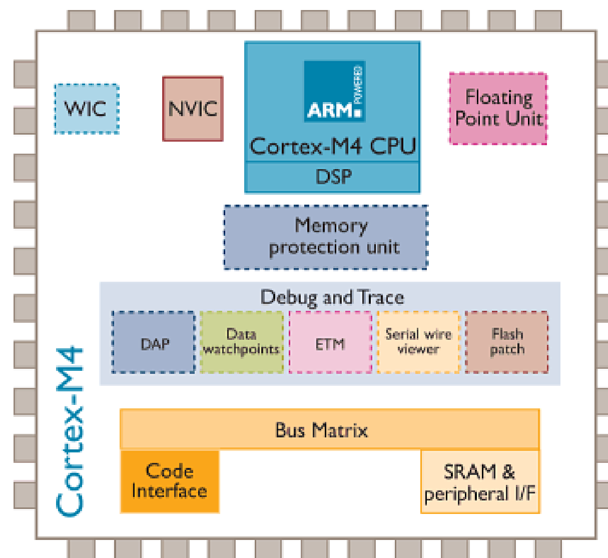


FIGURE 2.10 – Architecture du microcontrôleur NXP Cortex M4

ordinateur portable et pour de l'extraction de flot optique. De la même façon que les rétines câblées, ces système sont très performants pour un ou plusieurs traitements mais pêchent au niveau de la flexibilité et du coût de développement.

2.2.2.1 Architectures à base de processeurs généralistes

Les processeurs généralistes permettent à un concepteur d'implanter tous les algorithmes qu'il souhaite et sont les composants dont la programmation est la plus accessible parmi tous ceux présentés ci-après. Les principaux types de processeurs utilisés dans un système de vision sont des processeurs et des microcontrôleurs de type RISC. Ces composants sont parfois associés à une partie opérative de type SIMD et/ou DSP permettant d'accélérer les calculs de traitement d'images. On peut citer par exemple le microcontrôleur *Cortex-M4* développé par la société NXP [16] (Fig. 2.10) qui est basé sur un cœur de processeur ARM7TDMI (une extension des cœurs RISC) intégrant une partie DSP et une unité de calcul en codage flottant. Ceci permet d'effectuer des filtrages de façon relativement simple et optimisée. On peut également citer les processeurs OMAP [101] de Texas Instruments qui proposent, dans un seul *chip*, l'association d'un ou plusieurs cœurs de processeurs généralistes associés à une partie dédiée au traitement d'images (Fig. 2.11).

Le système de vision, et plus précisément la caméra intelligente, MeshEye [35], développée par l'Université de Stanford, est construite autour d'un microcontrôleur ATMEL AT91SAM7S. Ce microcontrôleur, à l'instar du NXP Cortex-M4, est basé sur un cœur de processeur ARM7TDMI. Ce système est utilisé pour

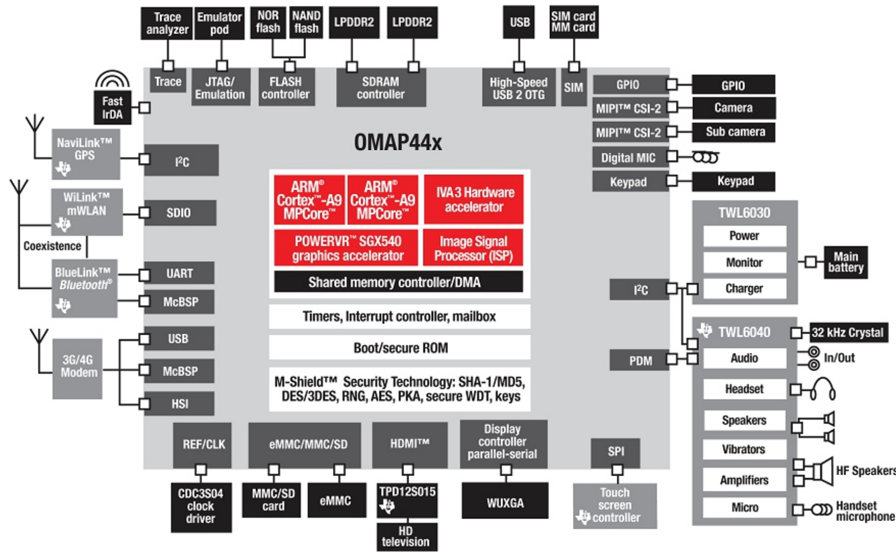


Figure 2.11 – Architecture des processeurs OMAP

de la surveillance par détection d'objet. Ce système multi-capteurs à résolution hybride permet l'optimisation du processus d'acquisition. Par exemple, un des capteurs basse résolution peut être utilisé pour la détection de mouvement dans la scène. Lorsqu'un objet mouvant est détecté, le deuxième capteur basse résolution est activé afin de réaliser un appariement stéréo. Une fois la position et la taille de l'objet mouvant estimées, une fenêtre d'intérêt contenant cet objet peut finalement être acquise par l'imageur haute résolution. Cette fenêtre d'intérêt peut ensuite être traitée par le système de vision grâce à la programmation du microcontrôleur.

Les processeurs généralistes ou GPPs [8] fonctionnent à des fréquences particulièrement élevées ce qui leur permet d'approcher les performances de processeurs de type DSP. L'ajout d'unités fonctionnelles (par exemple une deuxième unité de calcul sur les entiers, une unité de calcul flottant, des générateurs d'adresse,. . .) permet aussi d'exécuter plusieurs instructions à chaque cycle, sous réserve qu'il n'y ait pas de dépendance entre les instructions. Cette évolution a été accompagnée par l'introduction de jeux d'instructions complémentaires permettant d'adresser des unités fonctionnelles spécialisées. Ainsi Intel avec la technologie MMX [9][118], AMD avec la technologie 3DNow ![100] des Athlon ou encore AIM [10] avec la technologie Altivec sur les PowerPC[117] sont des exemples de ces extensions architecturales.

8. GPPs pour General Purpose Processors
9. MMX pour MultiMedia eXtensions
10. AIM est une alliance entre Apple, IMB et Motorola

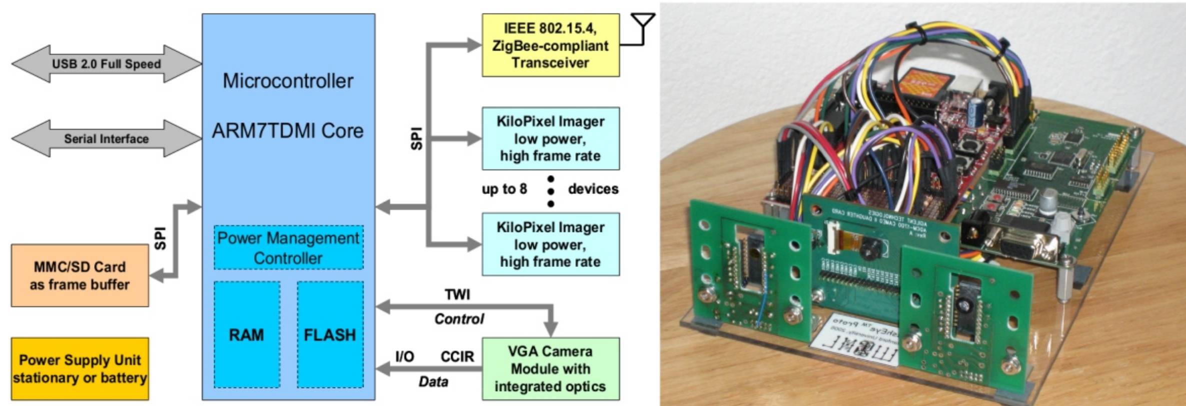


FIGURE 2.12 – Architecture du système de vision MeshEye

L'avantage principal des GPPs est sans conteste leur facilité de programmation permettant à un concepteur de développer des applications sans avoir de notion sur l'architecture du processeur. En contrepartie, ces processeurs adoptent un paradigme d'exécution temporelle en opposition avec le parallélisme nécessaire des applications de traitement d'images.

2.2.2.2 Architectures à base de DSP

Les DSP sont des processeurs dédiés au traitement du signal et vise les domaines du traitement d'images et de reconnaissance de formes. Ces processeurs sont relativement semblables aux microprocesseurs. Leur particularité réside dans le fait que ces composants sont conçus pour effectuer des calculs en temps réel et intègrent donc de nombreux opérateurs en plus du cœur calculatoire symbolisé par un opérateur de MAC opérations. Leur jeu d'instructions est de ce fait souvent plus réduit que celui d'un processeur généraliste. De plus, ces processeurs ont la possibilité de réaliser plusieurs instructions en parallèle (architecture VLIW [11]). Leur architecture est, comme celle des processeurs généralistes, figée et comprend un ensemble d'éléments qui, suivant les modèles, permettent d'exécuter des calculs sur des données codées en virgule fixe ou flottante. L'évolution technologique permet l'obtention de composants de plus en plus performants en terme de puissance de calcul. La taille des données manipulées a également augmenté passant de 16 bits à 96 bits. À titre d'exemple, la Fig. 2.13 présente l'architecture du DSP TI C64x.

11. VLIW pour Very Large Instruction Word

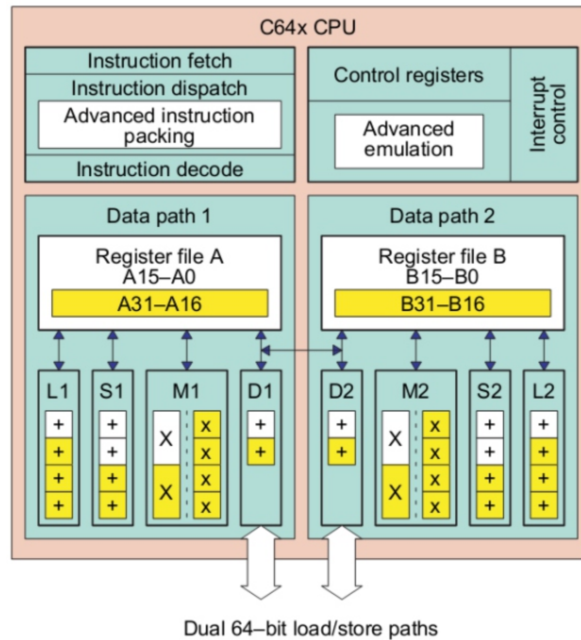


FIGURE 2.13 – Architecture du DSP C64x de Texas Instruments

Afin d'exploiter le parallélisme au niveau instruction, le C64x est un processeur VLIW 8 voies, réparties sur deux chemins de données séparés comme le montre la Fig. 2.13. Elles incluent six ALUs et deux unités de chargement et de stockage des données. De plus, des instructions SIMD sont intégrées, lui permettant de manipuler des vecteurs de données.

On peut voir dans la littérature et l'industrie que les systèmes de vision intégrants des DSPs admettent très souvent, si ce n'est systématiquement, un processeur généraliste. On peut par exemple citer les caméras NI 17xx de National Instruments [1] qui associent un GPP PowerPC pour la gestion globale du système avec un DSP de chez Texas Instruments en accélérateur de calcul.

De plus, on remarque également que, pour répondre aux contraintes temps réel des applications, l'association de plusieurs DSPs peut être nécessaire. M.Bramberger et al. ont développé un système de vision embarqué (Fig. 2.14) basé sur 2 DSP TMS320DM642 de Texas Instruments fonctionnant à une fréquence de 600MHz [47]. Un des deux DSP est en charge du contrôle du capteur d'image et de la compression vidéo. Le second permet d'appliquer des traitements permettant d'utiliser ce système dans le cadre du domaine de la surveillance.

Les processeurs média peuvent être considérés comme une classe de DSP. Ce

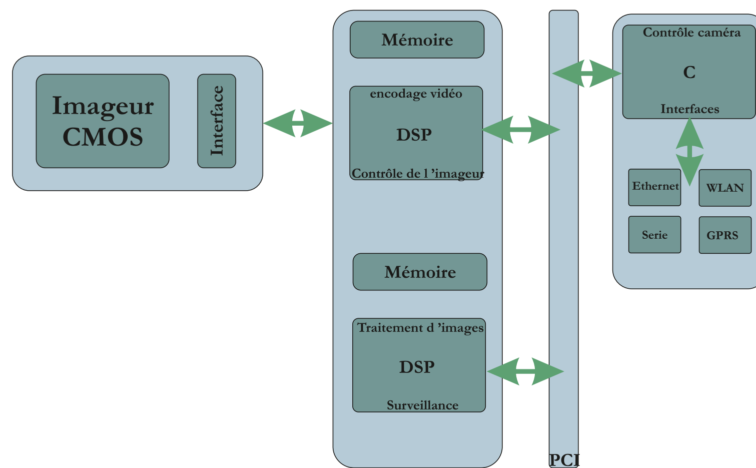


FIGURE 2.14 – Architecture du système de vision M.Bramberger et al

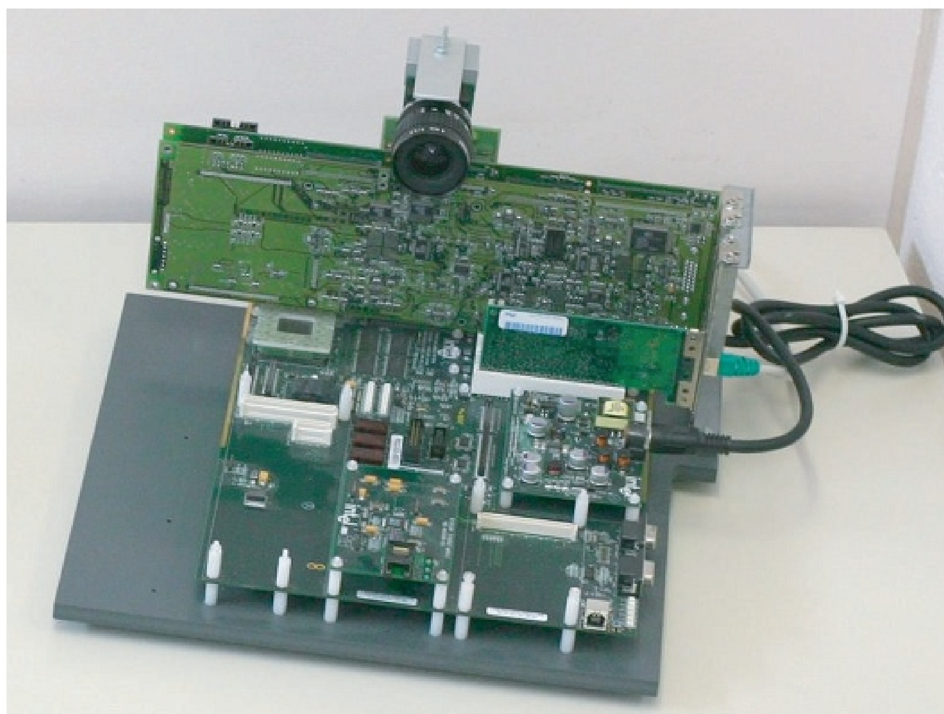


FIGURE 2.15 – Aperçu du système de vision M.Bramberger et al

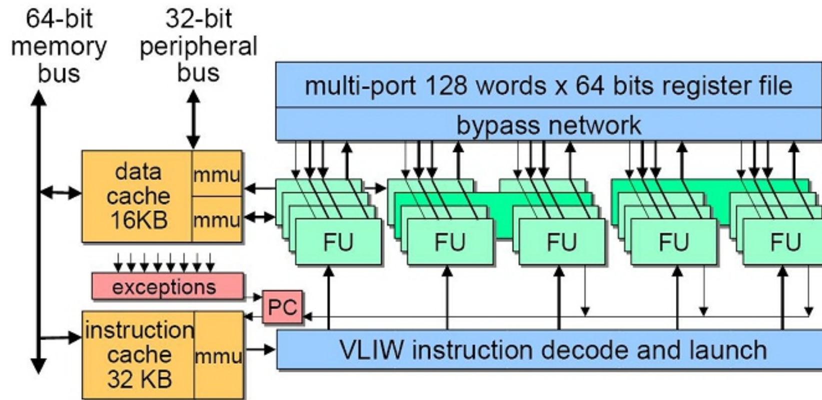


FIGURE 2.16 – Architecture du processeur média TriMedia TM3270

qui distingue les processeurs média des DSPs traditionnels est le fait que les processeurs média sont dédiés aux traitements vidéo/audio. Pour ce faire, les processeurs média intègrent généralement une partie opérative sous forme SIMD avec un nombre conséquent d'unités fonctionnelles afin d'augmenter leurs performances vis-à-vis des traitements d'images/vidéo comme par exemple le processeur média NXP TriMedia TM3270[120] (Fig. 2.16). L'architecture VLIW du Trimedia comporte 5 *slots*, signifiant que chaque instruction réalise jusqu'à 5 opérations simultanément. Les 128 registres de 64 bits chacun permettent l'exploitation du parallélisme SIMD, via des opérations vectorielles.

Le système proposé par Wolf et al. [27] est une carte PCI comportant 2 processeurs Trimedia TM1300 intégrée dans un PC hôte. Deux caméras Hi8 fournissent les images à cette carte via une connexion composite. Ce système est utilisé notamment pour de l'extraction de région ou encore du suivi de contours. Horst et al. [67] présentent un système où deux caméras DICA 321 sont utilisées pour constituer une plateforme de stéréo-vision (Fig 2.17). L'application proposée est le calcul de disparité entre deux images, afin d'obtenir une estimation de la profondeur de la scène, cependant la plateforme est conçue pour être un système généraliste de vision et traitement d'images. Les caméras DICA 321 sont équipées d'un CPLD permettant le contrôle de l'imageur et d'un processeur Trimedia pour effectuer les traitements.

En résumé, les DSPs offrent des performances, pour le traitement d'images, supérieures aux processeurs généralistes présentés précédemment. Néanmoins, nous pouvons constater que l'utilisation seule d'un DSP ne permet pas à la fois la gestion du système et le traitement d'images temps réel.

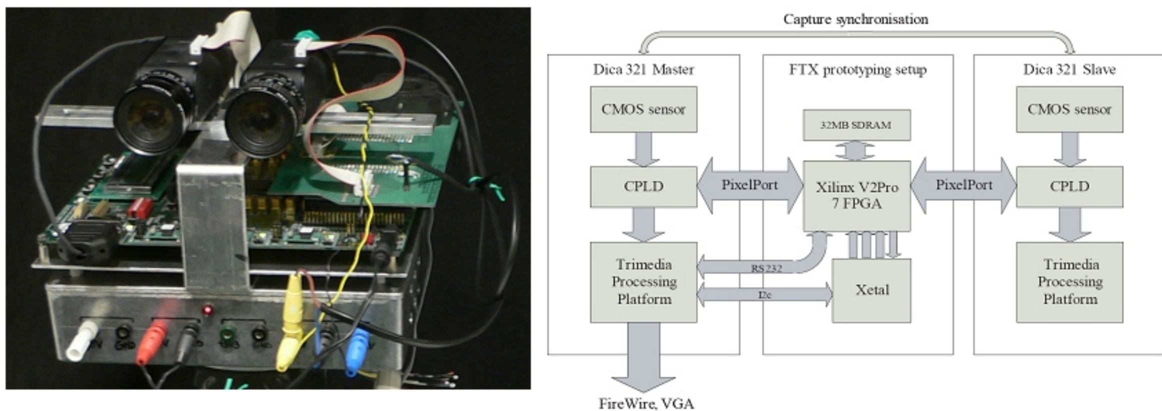


FIGURE 2.17 – Architecture de stéréo vision proposée par Horst[67]

2.2.2.3 Architectures à base de GPU

Les processeurs graphiques (GPU [12]) ont subi un essor indéniable ces dernières années à travers les besoins de rendu 3-D. Ces composants sont ainsi de plus en plus flexibles et leurs performances les rendent compatibles avec le traitement vidéo en temps réel. Leur intégration au sein de processeurs généralistes augure une poursuite de leur démocratisation. Ceci est d'autant plus vrai qu'avec l'apparition du calcul généraliste sur processeur graphique (GPGPU [13]) la pertinence de l'utilisation de GPU en dehors du rendu graphique tend à se confirmer.

Afin de donner un aperçu architectural des GPUs actuels, nous nous sommes intéressés à la gamme *Tesla* produit par *Nvidia*. Dans un article paru en janvier 2010, S. Collange [8] présente en détails l'architecture des GPU Nvidia Tesla (Fig. 2.18).

La partie opérative est composée d'un certain nombres, en fonction de la gamme, de *cluster* (jusqu'a 10) reliés aux autres éléments par un bus d'interconnexion de type *crossbar*. Chaque *cluster* contient plusieurs cœurs de traitement qui sont des unités de traitements de type SIMD (Fig. 2.19).

Ces opérateurs intègrent 8 unités (MAD [14] sur la Fig. 2.19) en charge des calculs arithmétiques généralistes. Une unité MAD est un pipeline composé d'un multiplieur puis d'un additionneur et d'une unité d'arrondi. Ces unités MAD sont capables de réaliser des calculs en virgule fixe ou en virgule flottante simple précision. L'unité FMA est utilisée pour réaliser les calculs en virgule flottante

12. GPU pour Graphics processing unit
13. http ://gpgpu.org
14. MAD pour Multiplications et Additions

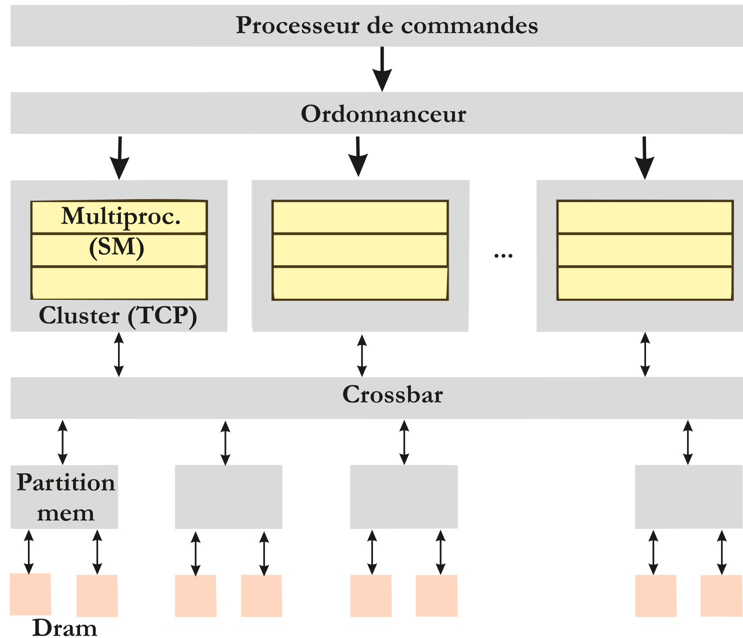


Figure 2.18 – Architecture des GPU Tesla

double précision. Enfin les unités SFU réalisent l'évaluation de fonctions élémentaires, l'interpolation d'attributs (coordonnées de texture, couleurs...) ou encore des multiplications. Elle est subdivisée en un circuit d'interpolation suivi de multiplieurs en virgule flottante.

Les GPU sont de plus en plus présents dans divers domaines. On peut ainsi noter leur utilisation pour le domaine médical [38] ou automobile [49]. Grâce à leur possibilité en terme de parallélisation, plus avancée vis-à-vis des GPP et DSP, les GPU offrent des performances supérieures pour des applications de traitement d'images. Par exemple, Umairy et al. [25] montrent que pour une opération de convolution, les performances d'un GPU sont nettement supérieures à celles d'un CPU (du double au quadruple en fonction de la dimension de la fenêtre d'intérêt).

Il est ainsi indéniable que les composants GPUs sont, parmi ceux présentés jusqu'ici, les composants offrant les meilleures performances pour les applications de traitement d'images. Ceci s'explique en grande partie par leur plus grande possibilité de parallélisation des applications. Néanmoins, les GPU sont des composants particulièrement difficiles à embarquer, qui ont une consommation énergétique importante et dont la programmation est nettement plus complexe que pour des composants du type GPP.

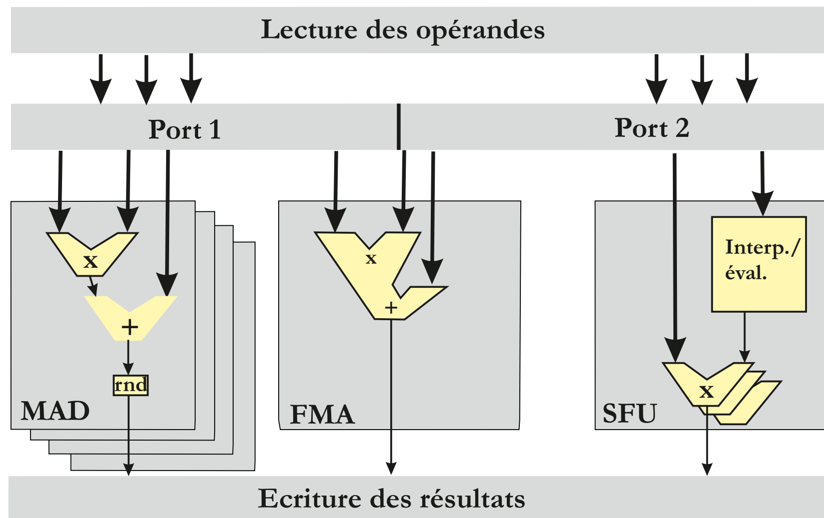


FIGURE 2.19 – Architecture des unités de traitements SIMD des GPU Tesla

2.3 Architectures pour le traitement d'images à base de FPGAs

Les architectures précédentes ont toute la particularité d'être basées sur des composants dont l'architecture est figée. Hormis les rétines, les composants présentés offrent néanmoins des possibilités de programmation permettant d'augmenter leur spectre d'applications. Dans cette section, nous nous intéressons à un type de composant dont l'architecture interne est fabriquée en fonction de l'application à exécuter : les FPGAs.

Dans un premier temps, une présentation générale de l'architecture d'un FPGA est donnée. Puis les techniques de reconfiguration sont abordées. Enfin, l'utilisation et l'intérêt des FPGAs dans le domaine du traitement d'images sont détaillés.

2.3.1 Présentation des FPGAs

Inventés par Xilinx [32] en 1985, les circuits FPGA ont connu une rapide évolution depuis plusieurs années, accompagnés d'une popularité croissante dans divers domaines allant du militaire à la recherche. Grâce à l'évolution des techniques d'intégration permettant une augmentation du nombre d'éléments logiques (LE's) et de mémoires internes disponibles par composant, l'augmentation des fréquences d'horloge et la possibilité d'exploiter massivement le parallélisme potentiel des applications, les FPGAs sont actuellement, en termes de performances,

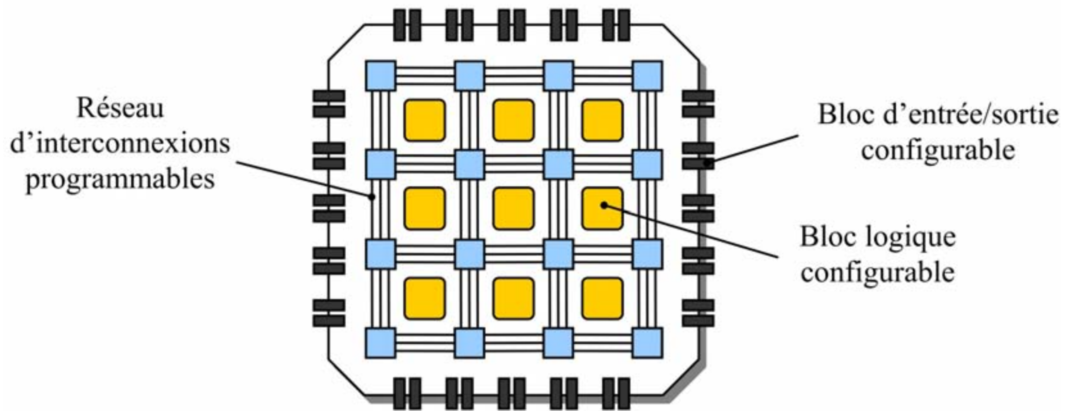


FIGURE 2.20 – Schéma simplifié de l'architecture d'un FPGA

de plus en plus proche de celles des composants ASIC. De plus, les FPGAs présentent l'avantage non-négligeable d'offrir de forte possibilités de reconfiguration, augmentant ainsi la flexibilité des systèmes basés sur ces dispositifs. Enfin, la possibilité d'implanter au sein d'un FPGA des processeurs de type *soft-core*, ainsi que des éléments fixes de type IP (Intellectual Property) viennent renforcer cette flexibilité, et constituent un attrait supplémentaire important en faveur de cette solution.

L'architecture générale des FPGAs, présentée en Fig. 2.20, est essentiellement constituée de trois éléments :

- Une matrice de blocs logiques configurables (CLB [15]) ;
- Des blocs d'entrée/sortie configurables ;
- Un réseau d'interconnexions programmables.

La programmation d'un circuit FPGA consiste à spécifier la fonctionnalité de chaque bloc logique et à organiser le réseau d'interconnexion afin de réaliser la fonction demandée.

L'architecture interne d'un FPGA peut être vue de manière hiérarchique sous la forme de trois plans virtuels superposés. Ces trois couches sont détaillées dans les sections suivantes.

2.3.2 Couche de configuration

Les FPGAs actuels sont basées sur trois technologies de configuration :

15. CLB pour Configurable Logic Bloc

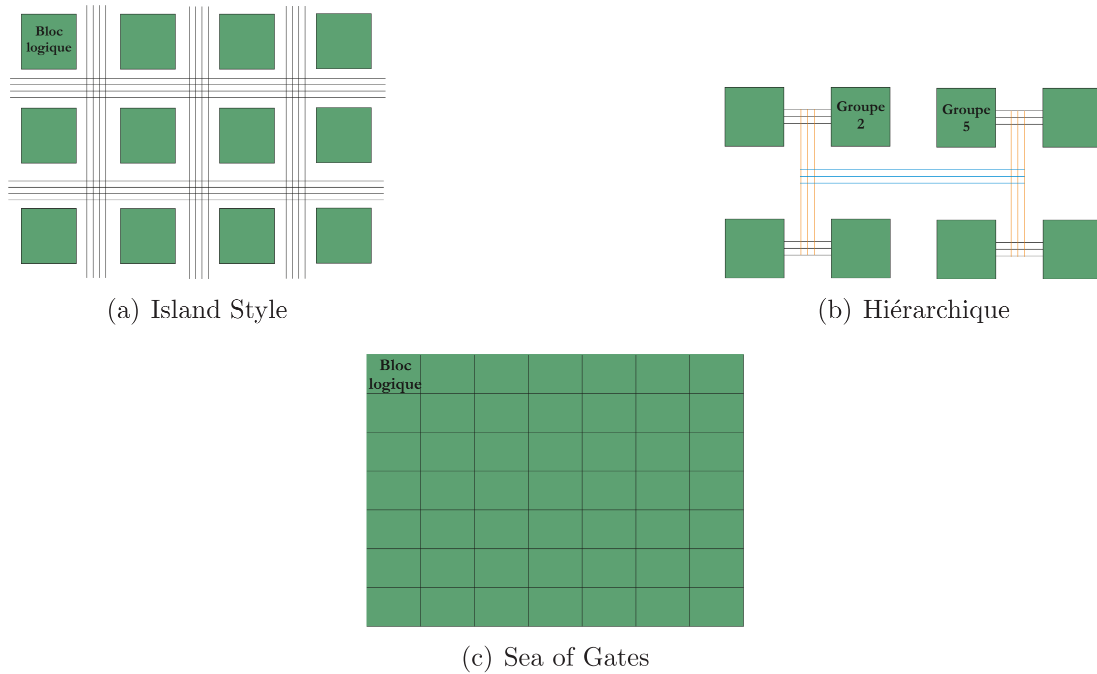
Technologie de configuration par anti-fusibles : La technologie de configuration par **anti-fusible** consiste, lors du processus de configuration, à laisser intacts les anti-fusibles (en état de haute impédance) ou bien de les détruire par l'application d'une tension (c'est ce que l'on appel le *claquage*), ceci afin de connecter deux éléments. La configuration est donc définitive et non-réversible. Par conséquent les FPGAs équipés de cette technologie ne peuvent pas être reconfigurés et s'apparentent à une PROM [16]. la gamme de FPGAs *Axcelerator* de la société *Actel* utilise cette technologie.

Technologie de configuration par mémoires SRAM : La technologie de configuration à base de **mémoires SRAM** est la technologie la plus courante car elle offre des possibilités de reconfiguration et également car elle est basée sur la technologie CMOS particulièrement bien adaptée aux procédés d'intégration. En pratique, les cellules SRAM sont utilisées pour créer le réseau d'interconnexion en connectant ou non les différentes lignes de communication. Les cellules restantes sont utilisées pour charger les LUTs [17] (tables de correspondance) qui seront généralement utilisées pour réaliser les fonctions logiques. Une SRAM étant une mémoire volatile, les composants basés sur cette technologie doivent être reconfigurés à chaque mise sous tension. Néanmoins, les données de configuration peuvent être stockées dans une mémoire externe non-volatile qui sera lue par le FPGA au démarrage.

Technologie de configuration par mémoires flash : La technologie de configuration à base de mémoire Flash offre plusieurs avantages, dont le principal est la non-volatilité. Cette caractéristique élimine le besoin de ressources externes pour stocker et charger les données de configuration inhérente à la technologie SRAM. De plus, un dispositif à base de mémoire flash peut fonctionner immédiatement après sa mise sous tension sans devoir attendre le chargement des données de configuration. Le désavantage majeur de la technologie flash réside dans le fait que le nombre de configuration est limité. En effet, les charges résiduelles présentes dans l'oxyde empêchent les dispositifs à bas de Flash d'être correctement effacés et configurés [62]. À titre d'exemple, le nombre de configurations possibles de certains dispositifs comme le Actel ProASIC [3] est estimé à seulement 500.

Le domaine du traitement d'images nécessite une flexibilité importante. En effet, ces traitements étant une étape initiale à des traitements de plus haut niveau, il est indispensable qu'ils puissent évoluer au cours d'une application. Ainsi, la technologie à base d'anti-fusibles n'est pas envisageable dans le cadre de notre étude à cause du fait qu'ils ne soient pas reconfigurables. Ce problème peut être

16. PROM pour Programmable Read Only Memory
17. LUT pour Look-Up Table



(a) Island Style

(b) Hiérarchique

(c) Sea of Gates

FIGURE 2.21 – Classification des FPGAs selon leur réseau d'interconnexion.

étendu aux FPGAs à base de mémoire Flash qui n'offrent pas un grand nombre de reconfigurations possibles.

Au niveau des technologies de configuration, seule la technologie à base de mémoires SRAM permet actuellement de répondre à la flexibilité nécessaire au traitement d'images. La suite de la présentation des FPGAs s'axe donc uniquement sur les FPGAs dotés de cette technologie.

2.3.3 Couche d'interconnexions des FPGAs SRAM

En fonction de la disposition des éléments logiques et du réseau d'interconnexion associé, un FPGA peut être classé dans une des trois catégories représentées par la Fig. 2.21.

La Topologie *Island Style* : Cette topologie consiste en un tableau à 2 dimensions de CLBs connectés grâce à des canaux de communication horizontaux et verticaux. L'entrée ou la sortie d'un CLB peut être connectée à un de ces canaux par l'intermédiaire d'une boîte de connexions composée généralement de plusieurs transistors programmables. Ces boîtes de connexions permettent ainsi de relier plusieurs CLBs ou encore de créer une connection entre les canaux de communication horizontaux et verticaux. Une grande majorité des FPGAs à base de mémoires SRAM adopte

ce type de topologie [4, 124].

La Topologie hiérarchique : Cette topologie divise l'architecture d'un FPGA en plusieurs groupes de blocs logiques. Les connexions entre CLBs d'un même groupe sont assurées par des canaux de communication représentant le plus bas niveau hiérarchique (niveau 1) de communication. La connexion entre CLBs de groupe distincts est permise par des canaux de hiérarchie supérieure. Dans l'exemple de la Fig. 2.21(b), deux niveaux de canaux permettent la connexion entre CLBs de groupe distincts. Le niveau 2, en orange sur la Fig. 2.21(b), permet d'établir la connexion entre deux groupes voisins. Le niveau 3, en bleu, assure la communication entre deux groupes non voisins. L'avantage de cette approche est qu'elle permet une estimation plus prévisible des délais de propagation et qu'elle diminue le nombre de *switch* nécessaires. La contrepartie est que les délais de propagation peuvent être plus longs que pour d'autres topologies. En effet, deux groupes de CLBs peuvent être physiquement voisins (c'est le cas des groupe 2 et 5 sur la Fig. 2.21(b)) mais ne peuvent communiquer que par l'intermédiaire de 2 voir 3 niveaux hiérarchiques de communication.

La Topologie *Sea of Gates* : Cette topologie est utilisée dans les FPGAs Xilinx XC6200 et Atmel AT94KAL Series [2, 6]. Contrairement aux deux topologies précédentes, qui adoptent le principe de séparer ressources logiques (CLBs) et de routage, l'approche *Sea of Gates*, initialement proposée dans l'architecture Triptych[31], mutualise ces ressources. Ainsi, les CLBs classiques présents dans les autres topologies sont replacés par des RLBs [18] qui sont en charge à la fois de la partie logique et de la partie de routage. Un RLB peut ainsi être configuré pour communiquer avec les RLBs voisins dans quatre directions (nord, sud, est et ouest). Des canaux de communication globaux peuvent également être intégrés afin de permettre la communication entre RLBs non voisins.

2.3.4 Couche logique des FPGAs SRAM

La fonction d'un bloc logique (CLB) au sein d'un FPGA est de fournir les éléments de calcul et de stockage élémentaire. Le composant de base d'un CLB est une table de correspondance ou *Look-Up Table* en anglais. Une LUT est un élément purement combinatoire composé de N entrées, de 2^N cellules mémoire SRAM, d'un multiplexeur 2^N vers 1 et d'une sortie (Fig. 2.22). Elle est capable de réaliser toutes les fonctions combinatoires à N entrées en configurant les cellules mémoires SRAM selon la table de vérité de la fonction voulue. À titre d'exemple, la Fig. 2.22 représente une fonction *OU Exclusif* à 3 entrées.

18. RLB pour Routing & Logic Blocks

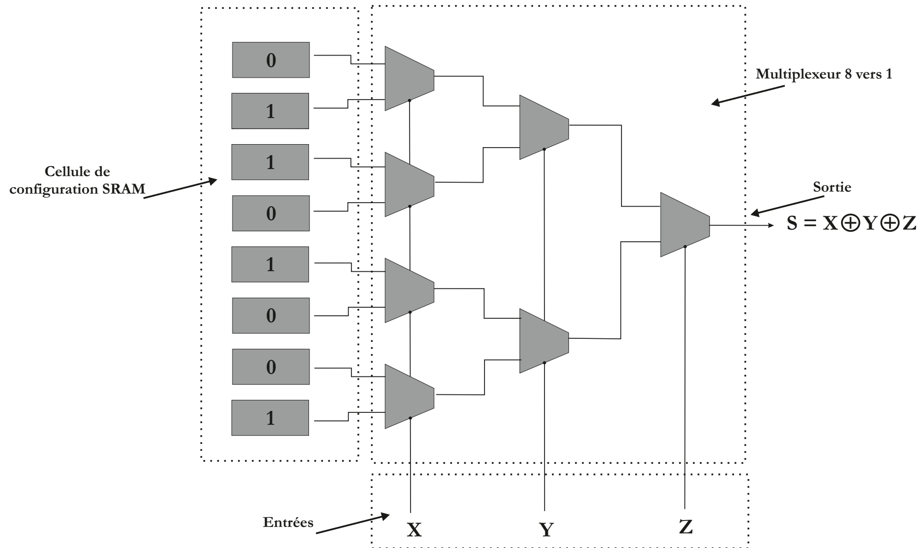


FIGURE 2.22 – Schéma synoptique d'une LUT réalisant un *OU Exclusif* à 3 entrées.

Le nombre de LUT au sein d'un CLB est variable d'un fabricant et d'une gamme de FPGAs à l'autre. L'association des plusieurs LUTs permet de réaliser des fonctions à grand nombre de variables d'entrée. Les Fig. 2.23 et Fig. 2.24 présentent le schéma synoptique d'un élément de calcul de base pour un FPGA Altera Cyclone II et un Xilinx 4000 series.

Afin d'inclure la possibilité d'effectuer des calculs séquentiels, des registres de type *flip flop* sont intégrés dans l'élément de calcul de base et un multiplexeur permet de choisir ou non l'utilisation de l'approche séquentielle. La Fig. 2.25 présente le principe d'une cellule.

À titre d'exemple, un additionneur à 2 opérandes de 32 bits, sur un FPGA Altera Cyclone III, nécessite 264 LUTs et 927 bascules D. Un multiplieur à 2 opérandes de 32 bits réservera 443 LUTs et 540 bascule D.

2.3.5 Techniques de reconfiguration

L'un des principaux atout des composants FPGAs SRAM est qu'ils sont reconfigurables "à volonté". Ainsi, le concepteur décrit une application à l'aide, généralement, d'un langage HDL (*Hardware Description Language*). Le code est ensuite synthétisé sous forme RTL [19]. Il s'en suit une étape de synthèse physique

19. RTL pour Register Transfer Logic

FIGURE 2.23 – Élément logique des FPGAs Altera Cyclone II

FIGURE 2.24 – Élément logique des FPGAs Xilinx 4000 series

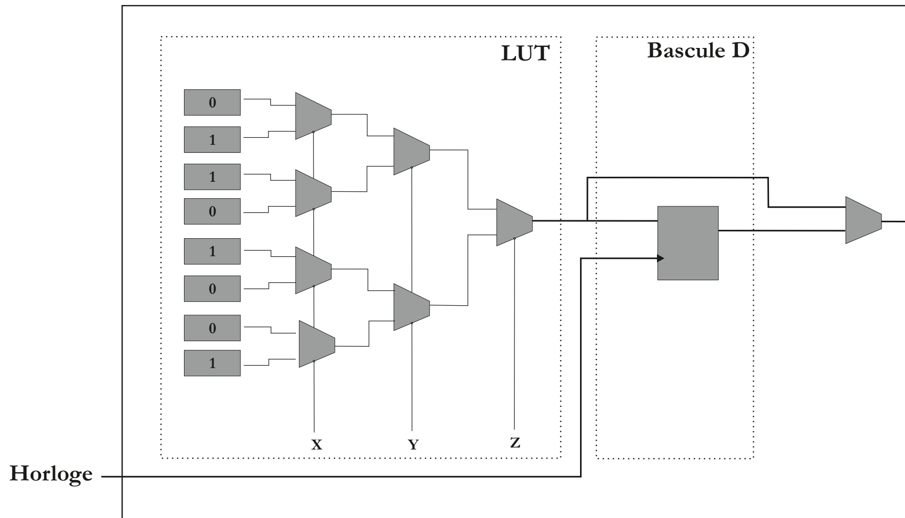


Figure 2.25 – Schéma synoptique d'un élément de calcul de base des FPGAs.

à partir du niveau RTL donnant un schéma de placement et routage. Enfin, ce schéma est chargé dans la mémoire de configuration du FPGA, via un *bitstream*[20], configurant les blocs logiques ainsi que les interconnexions entre ces blocs logiques et les I/Os (Fig. 2.26).

Il existe différents modes de reconfiguration des FPGAs. On peut les classer suivant 2 niveaux. Le premier niveau exprime l'aspect dynamique de la reconfiguration. Le second niveau indique la taille de la zone du FPGA à reconfigurer. La Fig. 2.27 présente les modes de reconfiguration détaillés par la suite.

2.3.5.1 Reconfiguration Statique

La reconfiguration statique (ou *compile-time reconfiguration* : CTR)] est utilisée lorsqu'une seule configuration est nécessaire au déroulement de l'application. Si une modification doit être apportée, il est alors inévitable d'interrompre l'application en cours comme explicité sur la Fig. 2.28.

La reconfiguration statique peut être effectuée en transférant le contenu d'une mémoire non volatile (EPROM ou Flash), dans la mémoire de configuration du FPGA. Dans ce cas, c'est le FPGA qui adresse la mémoire externe et met à jour sa mémoire de configuration. La configuration peut également être chargée dans le FPGA via une interface JTAG[21][123]. Une fois la configuration terminée, le

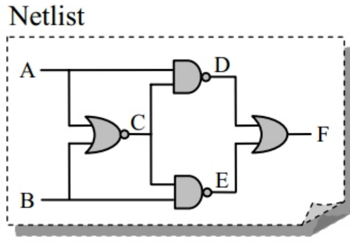
20. bistream : ensemble de bits permettant la configuration du FPGA
21. JTAG pour Joint Test Action Group

Codage avec un langage HDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity alg is
   Port ( A,B : in std_logic;
          F : out std_logic);
end alg;

architecture arch of alg is
signal C,D,E : std_logic;
begin
C<=A nor B;
D<=A nand C;
E<=C nand B;
F<=D or E;
end arch;
```

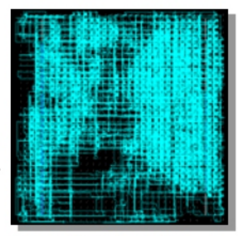


FIGURE 2.26 – Flot de développement sur FPGA.

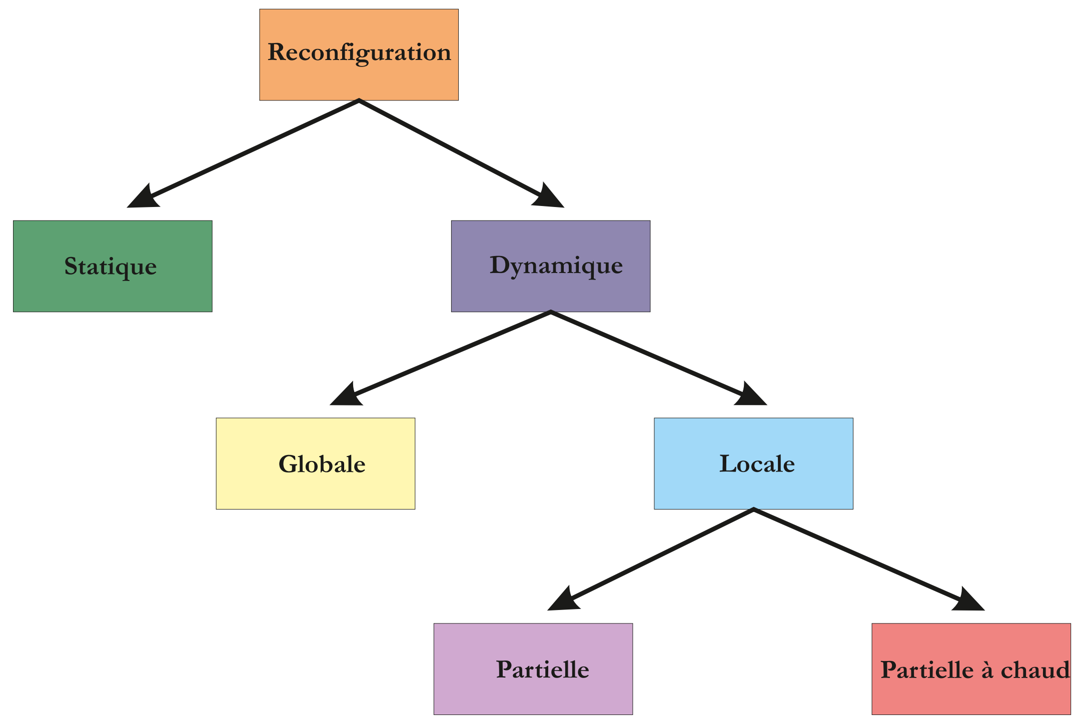


FIGURE 2.27 – Différents modes de reconfiguration d'un FPGA.

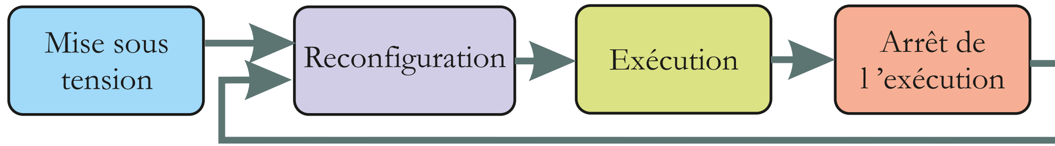


FIGURE 2.28 – Reconfiguration statique d'un FPGA

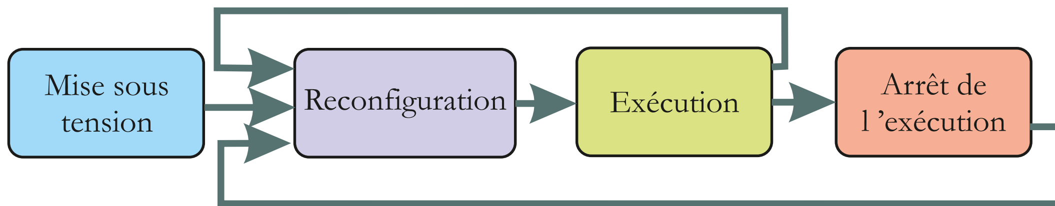


FIGURE 2.29 – Reconfiguration dynamique d'un FPGA

FPGA fonctionne jusqu'à l'arrêt du composant ou au chargement d'une nouvelle configuration.

2.3.5.2 Reconfiguration Dynamique

La reconfiguration dynamique (ou *run-time reconfiguration* : RTR). offre la possibilité de stocker, dans la mémoire de configuration du FPGA, plusieurs configurations. Pour reconfigurer le FPGA, il suffit alors de charger une des configurations disponibles. Cette technique offre ainsi une flexibilité plus importante qu'avec une reconfiguration statique. En contrepartie la reconfiguration dynamique nécessite des mécanismes plus complexes pour la gestion des *bitstreams* de reconfiguration.

La reconfiguration dynamique peut être scindée en deux catégories :
- la reconfiguration dynamique globale (*Global Run Time Reconfiguration :* ***GRTR***) présentée en Sec. 2.3.5.2.1
- la reconfiguration dynamique locale (**LRTR** pour Local Run Time Reconfiguration) détaillée Sec. 2.3.5.2.2).

2.3.5.2.1 Reconfiguration dynamique globale Dans ce mode, un ensemble de *bitstreams* de configuration est disponible et chacun d'entre eux permet d'appliquer une configuration monopolisant l'ensemble des ressources matérielles du FPGA. Ainsi, une seule configuration est active à un instant t et configure la totalité du composant. Le passage d'un plan à l'autre nécessite alors un temps de latence, durant lequel le FPGA sera inactif, de l'ordre de 5 millisecondes [66].

Afin de réduire le temps de passage d'une configuration à une autre, plusieurs travaux ont été réalisés. Ainsi, des architectures comme GARP [51] ou encore celle utilisée dans [29] ont opté pour l'utilisation de mémoire cache de reconfiguration. Cette mémoire, locale dans le but d'avoir un accès plus rapide, contient un certain nombre de configurations et permet d'en changer plus rapidement en cours d'exécution.

2.3.5.2.2 Reconfiguration dynamique locale Ici, la reconfiguration est appliquée à seulement une partie du composant. Cette approche permet de réduire les temps de reconfiguration, tout en permettant au reste du système de rester dans son état de configuration initial.

C'est en 1997 que la société Xilinx [32] commercialisa le premier FPGA permettant une reconfiguration dynamique, globale ou locale : le XC6200 [13]. Ce FPGA offrait une densité d'intégration relativement intéressante pour l'époque comparativement aux FPGAs concurrents visant également la reconfiguration dynamique. Malgré un succès auprès de la communauté scientifique, la commercialisation XC6200 fût rapidement stoppée devant le faible intérêt que lui ont porté le monde industriel, notamment à cause d'une interface de reconfiguration complexe.

La course à la densité d'intégration auraient pu avoir raison de la volonté à développer cet aspect de reconfiguration dynamique. Néanmoins, la société Xilinx a continué d'inclure la notion de reconfiguration dynamique au sein de ses composants FPGAs en développant également des outils dédiés à la mise en œuvre de la reconfiguration dynamique [14, 34]. Ce choix s'avère de nos jours être payant puisqu'avec l'intérêt grandissant pour les composants reconfigurables, d'autres manufacturiers, comme Altera[83], se sont également lancés dans ce domaine de manière plus intensive. Les références [96, 72, 45, 88, 76] présentent différents travaux réalisés sur des architectures permettant la reconfiguration dynamique.

On peut distinguer deux principaux modes de reconfiguration dynamique locale :

- La reconfiguration partielle : Ce mode permet de reconfigurer une zone du FPGA tandis que le reste du système est inactif tout en conservant ses informations de configuration. Le temps de reconfiguration dans ce cas dépend de la granularité de la zone à reconfigurer.
- La reconfiguration partielle à chaud ou active : Ce mode permet de reconfigurer sélectivement une zone du FPGA tandis que le reste du composant continue de fonctionner. Cela suppose bien sûr que la partie reconfigurée

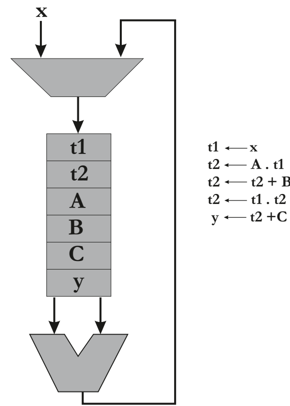


FIGURE 2.30 – Résolution temporelle de l'équation $y = a \times x^2 + b \times x + c$

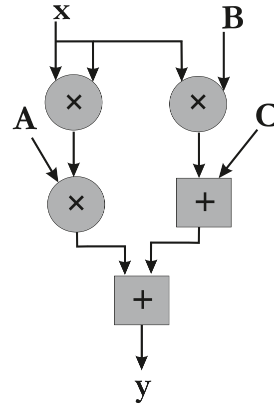


FIGURE 2.31 – Résolution spatialle de l'équation $y = a \times x^2 + b \times x + c$

n'est pas exploitée au moment de la reconfiguration.

2.3.6 FPGA et quelques applications de traitement d'images

Les FPGAs offrent une très forte flexibilité architecturale couplée à de très bonnes performances dans le cas d'applications de traitement d'images qui sont deux des aspects importants définis dans les objectifs de nos travaux. La flexibilité des FPGAs permet de créer l'architecture matérielle en fonction de l'application à exécuter. De ce fait, les performances de cette dernière s'en trouvent optimisées.

Par opposition au paradigme d'implémentation temporelle des processeurs classiques, les FPGAs offrent la possibilité d'effectuer une implémentation spatiale des applications. Par exemple, la Fig. 2.31 illustre la résolution de l'équation $y = a \times x^2 + b \times x + c$ suivant le paradigme temporel et spatial.

De part leur architectures les FPGAs apparaissent comme une solution très performante pour le traitement embarqué et notamment pour les applications de traitement d'images. Afin d'étayer ces propos, W.J. MacLean propose une évaluation de la pertinence de l'utilisation des FPGAs pour les systèmes de vision embarqués [99]. Ses conclusions peuvent se résumer ainsi :

• Points positifs :
- La flexibilité pour exploiter la nature intrinsèquement parallèle de beaucoup d'algorithmes de traitement d'images,
- un temps de développement, de test et de mise au point bien inférieur à celui des ASICs,
- un temps de reconfiguration de quelques millisecondes,
- un faible sensibilité aux perturbations par rapport aux CPUs (du fait d'une

horloge plus lente) et aux ASIC (empreinte matérielle plus petite),
- une forte protection face au *reverse engineering* (RE) grâce à l'utilisation de configurations par bit-stream cryptés,
- un coût matériel et logiciel relativement faible.

• Points négatifs :
- Le développement d'algorithmes utilisant des données codées en virgule flottante n'est pas impossible mais peut très rapidement saturer les ressources du FPGA,
- Les ressources d'un FPGA sont fixes et dans le cas où trop de ressources sont nécessaires, le concepteur est bloqué (à l'inverse des CPUs classiques où seul le temps de traitement est modifié) [22],
- Les performances des FPGAs sont moindres que celle des ASICs,
- Le développement d'application sur FPGA demande des connaissances solides en électronique numérique et en programmation en langage HDL (VHDL[91] ou Verilog[92]).

En conclusion, W.J. MacLean estime que la technologie FPGA est prometteuse pour les applications de vision embarquées principalement grâce aux avantages de leur reconfigurabilité et leur forte prédisposition à l'exploitation du parallélisme. Ceci est nuancé par les restrictions induites par la quasi obligation d'utiliser des données codées en virgule fixe. À ces considérations, s'ajoute également la nécessité de connaissances en électronique numérique afin de répondre à la difficulté du développement d'algorithmes sur FPGA.

On peut trouver dans la littérature un grand nombre de systèmes de vision intégrant un dispositif FPGA en tant que module de traitement permettant d'exécuter un large éventail d'algorithmes de traitement d'images de manière optimale.

Un système de vision a été développé par le laboratoire Le2i de l'Université de Bourgogne [48]. Ce système (Fig. 2.32), intègre un capteur CMOS haute-vitesse de 1.3 Mpixels, un circuit FPGA de la famille Virtex II de chez Xilinx et une interface USB 2.0. Le capteur d'images employé est capable d'acquérir jusqu'à 500 ips, soit une cadence de sortie de 6.55 Gbits/s. Afin de permettre la transmission d'un tel flot vidéo via le lien USB 2.0 (480 Mbits/s), des algorithmes de compression sont implémentés et exécutés au sein du circuit FPGA. Des tâches de traitement d'images sont également implémentées, comme le filtre de Sobel ou des opérations d'érosion/dilatation.

22. On peut nuancer cet argument grâce aux possibilités d'intégration de processeurs *soft-core* au sein d'un FPGA lui permettant d'exécuter un algorithme de manière temporelle.

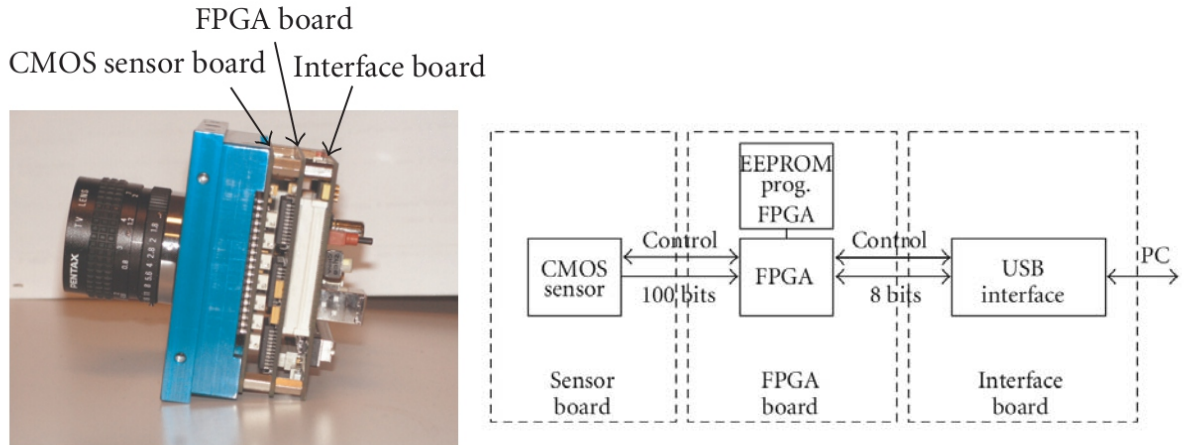


FIGURE 2.32 – Caméra rapide du laboratoire Le2i

Nous avons également, au LASMEA, deux systèmes de vision basés sur un composant FPGA. La première plateforme baptisée *SeeMOS* et présentée en détails en Sec. 6.1 intègre notamment un FPGA STRATIX I, un imageur LUPA 4000 et une carte de communication FireWire. Plusieurs applications de traitement d'images ont été réalisées sur cette plateforme. On peut citer l'implémentation de l'algorithme d'extraction de flux optique suivant la méthode de Lucas et Kanade [42] (Fig. 2.34). Cet algorithme met particulièrement en avant l'avantage de la parallélisation offert par les composants FPGAs. Par exemple, la première étape de cet algorithme consiste à calculer, pour chaque point d'une image, le gradient vertical, horizontal et temporel. Grâce à l'utilisation d'un FPGA, ces trois calculs peuvent être réalisés en parallèle là où sur un CPU classique il aurait été nécessaire de séquencer leur calcul. Ainsi, suivant notre implémentation, on obtient, pour une résolution d'image de 800 × 600, une cadence d'image de 30 IPS [23]. À titre de comparaison, Marzat et al. [79] obtiennent une cadence de 15 IPS pour une résolution d'image de 640 × 480 en utilisant un dispositif GPU.

La seconde plateforme, la caméra intelligente BiSeeMOS[40], est dédiée aux traitements stéréo. Pour ce faire la caméra BiSeeMOS (Fig. 2.35) dispose de deux capteurs d'image CMOS. La partie de traitement est basée sur un FPGA Altera Cyclone III EP3C120. Enfin, la communication est assuré par un module de communication Camera Link. À titre d'exemple, la camera BiSeeMOS permet d'appliquer une transformée de Census sur une image de résolution 1024 × 1024 à la cadence de 100 images par seconde [39].

23. IPS pour Images Par Seconde.

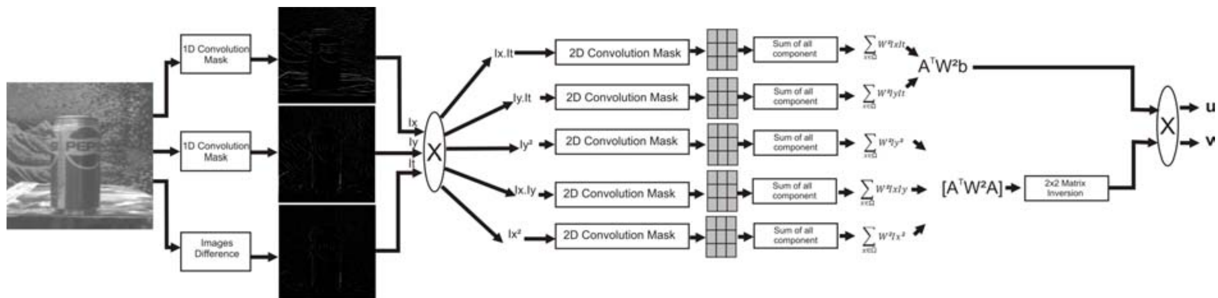


FIGURE 2.33 – Résolution spatiale de l'algorithme d'extraction de flux optique

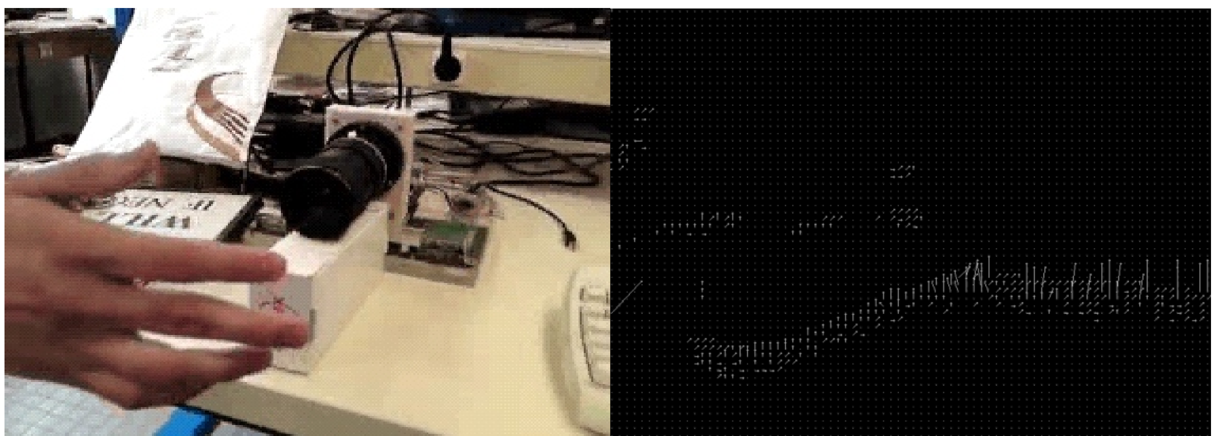


FIGURE 2.34 – Extraction du flux optique suivant la méthode de Lucas et Kanade

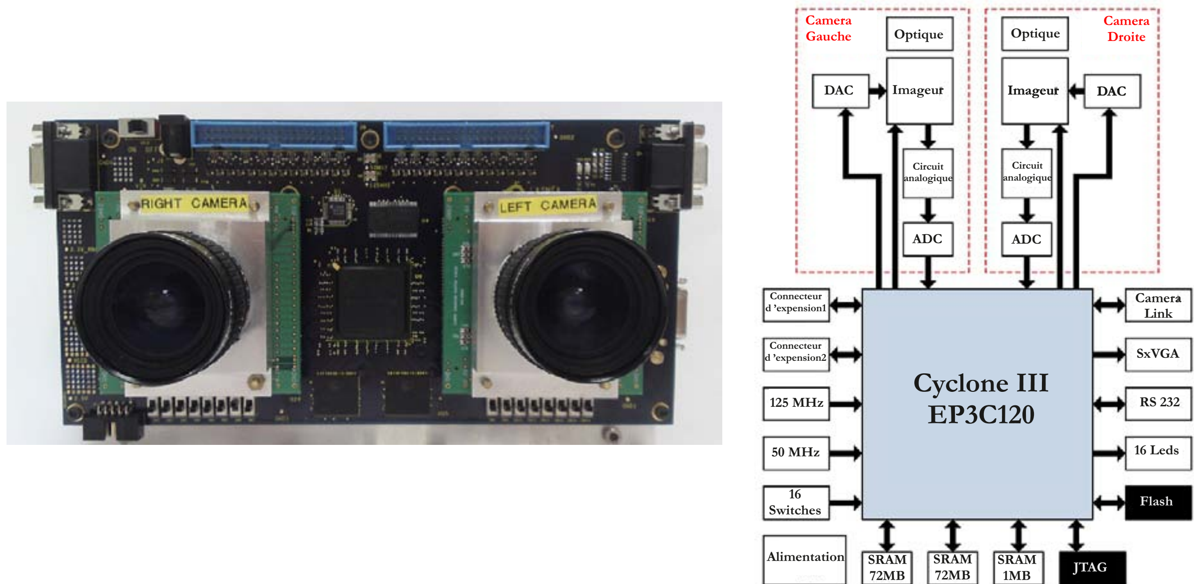


FIGURE 2.35 – Caméra intelligente BiSeeMOS

2.3.7 Comparaison des performances des FPGAs et des GPUs pour des applications de traitement d'images

Les sections précédentes ont permis de mettre en avant que seuls les composants FPGAs et GPUs admettent un niveau de parallélisation pouvant être satisfaisant dans le cadre d'applications de traitement d'images. Le but de cette section est de comparer les performances, en termes de temps de traitement ou de bande passante de sortie, de ces deux dispositifs.

Une première comparaison nous est proposée par B. Cope [9]. Dans ce papier, l'auteur évalue les performances des dispositifs FPGAs, GPU et CPU dans le cadre d'un calcul de convolution 2D. Ce type de calcul est typique du traitement d'images puisqu'il permet l'application de divers filtrages sur l'image. Les composants utilisés sont un Virtex II-Pro pour le FPGA, un GPU NVidia 6800 Ultra et un CPU Intel Pentium 4 cadencé à 3 GHz. Le Tab. 2.1 dresse les résultats obtenus par B. Cope.

Le Tab. 2.1 met en avant que, par son manque de parallélisme, le processeur généraliste est, en termes de performances, largement dépassé par les deux autres dispositifs. Il est également notable que pour des tailles réduites du masque de convolution, le dispositif GPU est le plus performant. Ceci s'explique de part le faible niveau de parallélisme requis pour ces dimensions de masques. Ainsi, grâce à son horloge interne nettement supérieure à celle d'un FPGA, le GPU

Taille du masque de convolution	FPGA	GPU	CPU
2×2	221	1070	14
3×3	202	278	9.7
5×5	112	54	5.1
7×7	90	22	2.6
9×9	73	9	1.6
11×11	23	4.7	1.2

Table 2.1 – Comparaison des performances entre un FGPA, un GPU et un CPU pour l'exécution d'une convolution 2D en MP/s.

surclasse ce dernier. Lorsque le niveau de parallélisme augmente, donc pour des tailles de masque plus importantes, le FPGA propose de meilleures performances.

Un autre article [26] propose l'implémentation d'un algorithme d'extraction de flux optique sur un FPGA (Virtex-4) et un GPU (NVIDIA GeForce 8800 GTX). Cet algorithme est décomposé en 5 étapes :

- Étape 1 : extraction de 3 gradients. Le premier gradient est un gradient temporel, soit une différence d'images successives pixel à pixel. Les deux autres sont des gradients verticaux et horizontaux extraits par l'intermédiaire d'une convolution avec un masque de taille 5×5,
- Étape 2 : pondération des gradients grâce à un masque de convolution de taille 7×7,
- Étape 3 : génération de 5 nouvelles matrices en multipliant diverses combinaisons des 3 gradients issus de l'étape 2,
- Étape 4 : application d'une convolution de taille 3×3 sur les 5 résultats de l'étape 4,
- Étape 5 : calcul des composantes verticale et horizontale du vecteur de déplacement de chaque pixel. Pour ce faire, des opérations de multiplication, de soustraction et de division sont appliquées.

L'évaluation des performances a été faite grâce au jeu d'images de synthèse *Yosemite*[21]. Ainsi pour une résolution d'images de 640×480, l'implémentation sur GPU a permis d'atteindre une cadence d'images de 150 IPS là où l'implémentation sur FPGA a donné un résultat de plus de 300 IPS.

Une troisième comparaison est présentée par S. Asano [74]. Ce papier propose une évaluation des performances des dispositifs FPGAs, GPUs et CPUs dans le cadre d'applications de traitement d'images. Les composants utilisés sont un Xilinx XC4VLX160 pour le FPGA, une XFX GeForce 280 GTX 1024MB DDR

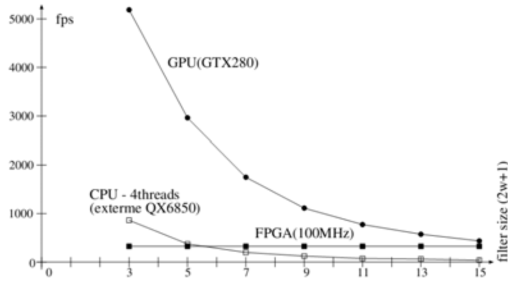


Figure 2.36 – Performances pour un filtrage 2-D

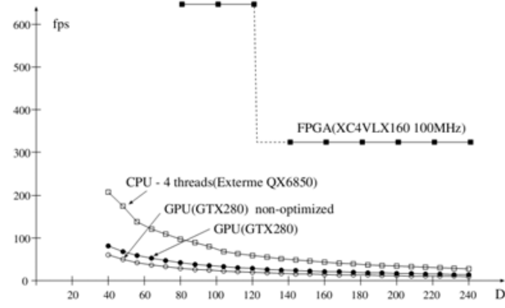


Figure 2.37 – Performances pour une SAD

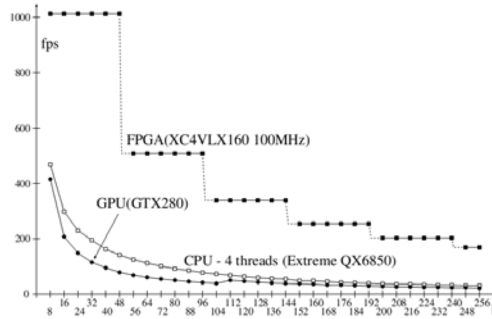


Figure 2.38 – Performances pour une classification par K-moyennes

pour le GPU et un Intel Core 2 Extrem QX6850 à 3 GHz pour le CPU. Trois traitements ont été implémentés sur ces dispositifs. Le premier traitement est un filtrage 2-Dimensions pour lequel plusieurs tailles de masque ont été appliquées et répondant à l'Eq. 2.1. Le second traitement réalise une SAD [24] dans le cadre de vision stéréo (Eq. 2.2). Enfin le dernier traitement est une classification par K-moyennes suivant l'Eq. 2.3.

$$S(x,y) = \sum_{dx=-w}^{w} \sum_{dy=-w}^{w} I(x+dx, y+dy).G(dx,dy) \tag{2.1}$$

$$SAD_{xy}(x,y,d) = \sum_{dx=-w}^{w} \sum_{dy=-w}^{w} |Ir(x+dx, y+dy) - Il(x+d+dx, y+dy)| \tag{2.2}$$

$$E = \sum_{i=1}^{K} \sum_{x \in C_i} (x - \sum_{x \in C_i} \frac{x}{|C_i|})^2 \tag{2.3}$$

24. SAD pour Sum of Absolute Difference

La Fig. 2.36 rapporte les performances concernant l'application d'un filtrage 2 dimensions. Elle montre que pour les tailles de masque testées, les performances offertes par le GPU sont les plus importantes. La taille maximale de masque testée est de 15×15 pixels, soit 225 pixels traités. Or le GPU utilisé permet le traitement de 240 données en parallèle. Du fait de l'horloge interne plus importante pour le GPU que pour le FPGA, le GPU reste meilleur que le dispositif FPGA. Néanmoins, on peut facilement considérer que le FPGA sera plus efficace pour des tailles de masque plus importantes grâce à "sa meilleure prédisposition" au parallélisme. Il est également notable que pour une taille de masque 3×3, le CPU est plus performant que le FPGA mais dès lors que la dimension augmente, le CPU devient bien moins performant que le FPGA (facteur 8 en faveur du FPGA pour une dimension 15×15.

La Fig. 2.37 montre que pour une SAD dont la taille de fenêtre d'intérêt est de 7×7 le dispositif FPGA est bien supérieur aux deux autres composants. Au delà d'une certaine valeur de D, qui définit le nombre de fenêtres de I_l à comparer, le FPGA admet un diminution de ses performances. Ce phénomène est dû à une limitation physique du FPGA en termes de blocs RAM. Néanmoins, pour $D = 240$ le GPU est moins performant d'un facteur 30.

Enfin la Fig. 2.38 confirme l'expérimentation précédente en montrant que pour une classification en 48 *clusters* le FPGA outrepasse les 2 autres composants d'un facteur de 7 à 9 grâce à son parallélisme massif et au problème d'accès aux données inhérent à l'utilisation d'un GPU.

En conclusion, pour des applications de traitement d'images, le parallélisme est un aspect plus que primordial pour l'obtention de performances élevées. De ce fait, les FPGAs possédant le parallélisme le plus massif rendent ces composants parfaitement adaptés au domaine du traitement d'images.

2.4 Conclusion

Dans ce chapitre, plusieurs types de composants de traitements ont été présentés. Le Tab. 2.2 propose une comparaison de ces composants en fonction de trois critères. Le premier critère concerne l'aspect embarquable des composants de traitements. En effet, nos travaux sont axés sur le développement d'un modèle de processeur embarqué pour le traitement d'images. Le second critère permet de classifier les composants suivant leur possibilité de parallélisme, nécessaire à l'obtention de hautes performances en traitement d'images. Enfin, un dernier critère permet d'établir la facilité de programmation inhérente au développement d'application sur chacun de ces composants.

Composant	Embarquabilité	Parallélisme	Facilité de programmation
GPP	+	−	++
DSP	*o*	-	+
Processeur Média	*o*	*o*	+
GPU	−	+	*o*
FPGA	++	++	−

Table 2.2 – Tableau comparatif des composants de calcul pour le traitement d'images

Ce tableau permet de mettre en évidence que les dispositifs de traitement FPGAs, de par leur architecture, sont particulièrement adaptés au domaine du traitement d'images.

Au niveau de la reconfiguration des FPGAs, la reconfiguration dynamique locale semble être le mode le plus adapté. En effet, il permet une optimisation matérielle des ressources nécessaires et ce à chaque instant de l'application grâce à un temps de reconfiguration très faible comparativement à la reconfiguration dynamique globale. D'un point de vue performances, cette option est donc parfaitement envisageable pour les applications de traitement d'images. Néanmoins, la reconfiguration dynamique est une technologie qui n'est pas encore mature et uniformisée. Ainsi, chaque fondeur de FPGAs dispose de ses propres méthodes de reconfiguration dynamique locale ce qui rend le portage d'un composant à un autre impossible.

S'agissant de la reconfiguration dynamique globale, étant donné le contexte de nos travaux, des véhicules en mouvement, le temps nécessaire à la reconfiguration du FPGA (de l'ordre de 5 ms pour rappel) par ce mode peut s'avérer critique pour la sécurité du véhicule et/ou de son environnement. La durée du temps de reconfiguration dynamique globale est dû au fait que la granularité des éléments logiques est très fine. Ainsi, pour passer d'une opération à une autre, un grand nombre de CLBs doivent être reconfigurés ce qui a pour conséquence un temps de reconfiguration non négligeable.

Dans le cadre de nos travaux, la reconfiguration statique a ainsi été choisie. Afin de compenser la perte de flexibilité vis-à-vis des méthodes de reconfiguration dynamique, nous avons opté pour l'implantation d'un processeur à chemin de données reconfigurable. Ce processeur nous permet de modifier dynamiquement les traitements exécutés et peut admettre une programmation plus *haut niveau* que les langages HDL. Le principe de ces processeurs est présenté dans le chapitre suivant.

Chapitre 3

Les processeurs à chemin de données reconfigurable

Dans le but de faciliter l'implémentation de traitements d'images sur FPGA et ainsi éviter le codage en langage HDL tout en conservant une flexibilité nécessaire à ce type de traitements, ce chapitre présente un modèle de calculateur dédié. Ce modèle est un processeur où le chemin de données est reconfigurable.

Dans un premier temps, le principe et l'architecture des processeurs à chemin de données reconfigurable sont présentés. On étudiera notamment les diverses topologies d'interconnexion existantes. Un état de l'art des processeurs à chemin de données reconfigurable (PCDR) est ensuite proposé.

3.1 Les processeurs à chemin de données reconfigurable

Dans un processeur, les registres, bus et opérateurs par lesquels transitent des données forment ce qu'on appelle le chemin de données, ou encore la partie opérative. Ce chemin de données est chargé d'exécuter l'instruction, lorsqu'il s'agit d'une instruction de traitement, préalablement décodée par la partie contrôle du processeur.

L'architecture d'un processeur à chemin de données reconfigurable (**PCDR**) est généralement composé d'un cœur de processeur (RISC ou CISC la plupart du temps) responsable du chargement et du décodage des instructions présentes dans une mémoire programme. Elle comporte également des unités de traitement dédiées (*PEs*[1]), reconfigurables ou non, reliées par un réseau d'interconnexion reconfigurable (Fig. 3.1).

1. PE pour Processing Element

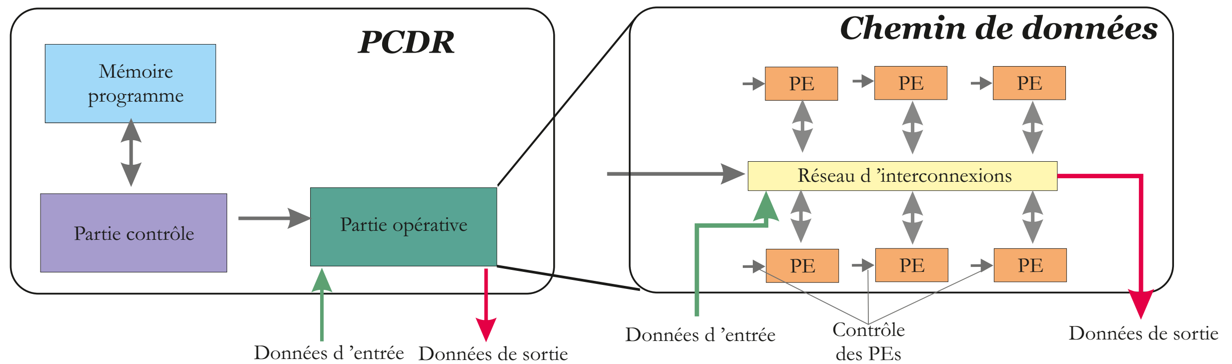


FIGURE 3.1 – Schéma synoptique simplifié de l'architecture d'une processeur à chemin de données reconfigurable.

Un PCDR a la particularité de pouvoir modifier dynamiquement l'agencement de ces unités de calcul afin de réaliser l'instruction souhaitée. La Fig. 3.2 met en avant ce principe. Un PCDR a également la possibilité de reconfigurer la fonction de chaque unité de traitement. Ainsi, la reconfiguration au sein d'un PCDR est une reconfiguration qualifiée de **fonctionnelle** [2].

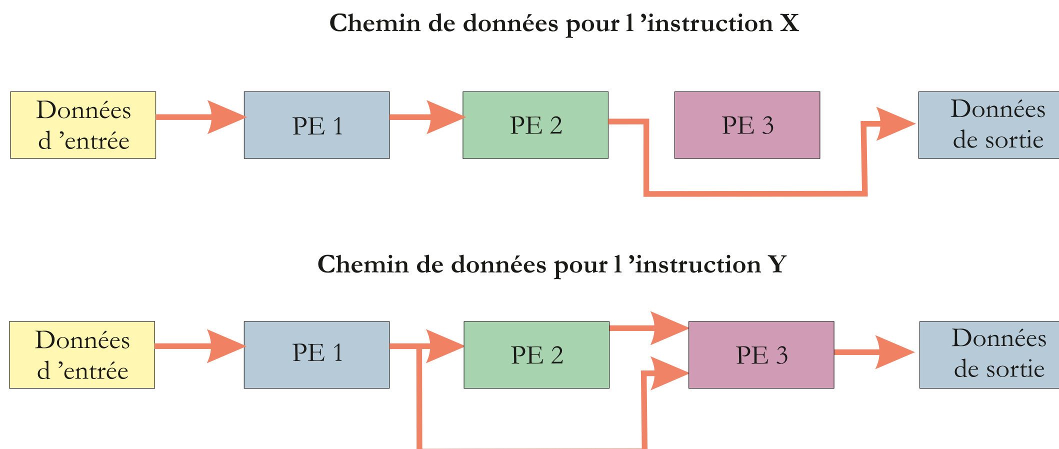


FIGURE 3.2 – Reconfiguration du chemin de données.

2. La reconfiguration des FPGAs est une reconfiguration dite **matérielle** de par le fait qu'elle modifie le comportement de l'architecture au niveau porte

3.1.1 Les topologies d'interconnexion

Comme présenté dans la section précédente, un des éléments de base des PCDR est le réseau d'interconnexion. Il nous a donc paru essentiel de parcourir les diverses topologies d'interconnexion existantes.

Les dispositifs de communication à base de bus sont variés et agissent directement sur le coût, la complexité, la consommation et les performances des architectures. La topologie la plus simpliste est le *bus simple* présentée en Fig 3.3. Ici, tous les composants du système sont reliés à un bus partagé. Cette configuration est efficace pour des architectures ne comportant que peu d'éléments. En effet, une seule communication d'un élément à un autre est permise à la fois. Les communications sont gérées par un arbitre qui va permettre, ou refuser -en cas de requêtes simultanées-, l'utilisation du bus par un couple de composants. Plusieurs bus basés sur ce principe ont été développés :

- le bus PCI,
- le bus Avalon de Altera [10],
- le bus CoreConnect de IBM [90].

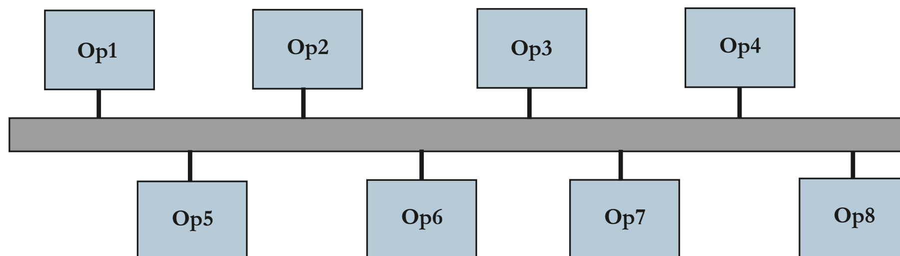


FIGURE 3.3 – Bus simple.

Une topologie permettant plusieurs transferts de données en parallèle est la topologie de *bus hiérarchique* décrite en Fig. 3.4. Les composants sont connectés à plusieurs bus qui sont interfacés les uns avec les autres grâce à des ponts (ou *Bridge* en anglais). Des transferts concurrents de données sont possibles sur chaque bus, à condition que les composants soient répartis sur les bus de telle sorte qu'il y ait une interaction minimale entre les composants de bus différents. Par exemple, en rapport avec la Fig. 3.4, si l'OP1 communique la majorité du temps avec l'OP7, il sera plus intéressant de les mettre sur le même bus. Inversement, si très peu de transactions ont lieu entre les 2, il est intéressant de les connecter sur des bus différents. Les bus pouvant avoir des fréquences d'horloge différentes, le pont peut être relativement complexe afin de manipuler les transactions inter-bus, les temporisations données (*data buffering*), les conversions

de fréquence, etc. Plusieurs SoCs commerciaux embarquent aujourd'hui un bus hiérarchique, comme le ARM PrimeXsys SoCs [105] qui est largement utilisé dans le domaine de la téléphonie mobile ou encore des GPS.

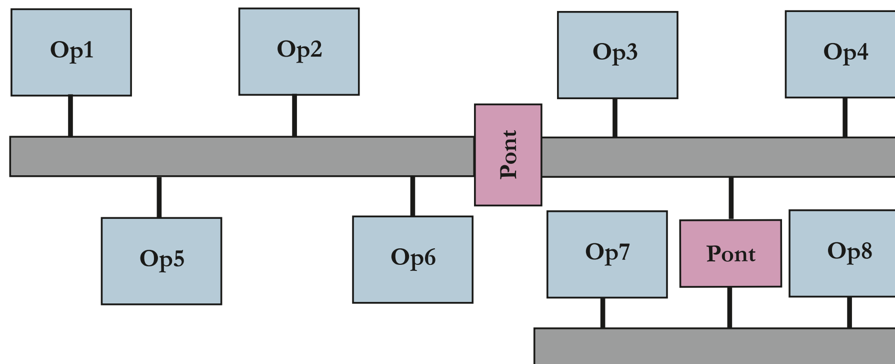


FIGURE 3.4 – Bus hiérarchique.

Pour les systèmes hautes performances qui nécessitent de nombreux transferts de données en parallèle, la topologie *Crossbar complet*, (Fig. 3.5) offre un choix très intéressant. Dans ce cas de figure, chaque élément est relié avec tous les autres éléments par l'intermédiaire de connexions directes. Le grand nombre de liaisons permet ainsi d'effectuer plusieurs transferts de données en parallèle. Plusieurs travaux [37, 59] ont montré l'utilité d'un Crossbar complet qui offre une plus grande bande passante comparativement aux topologies simple bus et bus hiérarchique. Néanmoins, plus le nombre d'éléments connectés au crossbar est grand, plus la complexité de ce dernier est important, impactant directement sur la consommation, les ressources matérielles ou encore les délais de propagation. En effet, si beaucoup d'éléments sont connectés sur le crossbar, le routage de toutes les possibilités de connections devient très complexe, allongeant ainsi les délais de propagation et monopolisant des ressources matérielles supplémentaires. La Fig. 3.6 extraite de [95], permet de mettre en évidence l'augmentation conséquente en termes de ressources lorsque le nombre de ports augmente. Un exemple commercial d'un système utilisant un crossbar complet est le multiprocesseur SoC Niagara de SUN [104].

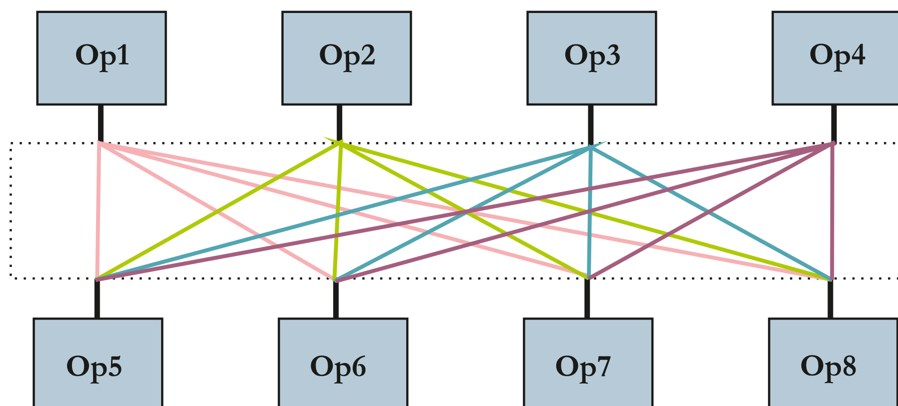


FIGURE 3.5 – Crossbar complet.

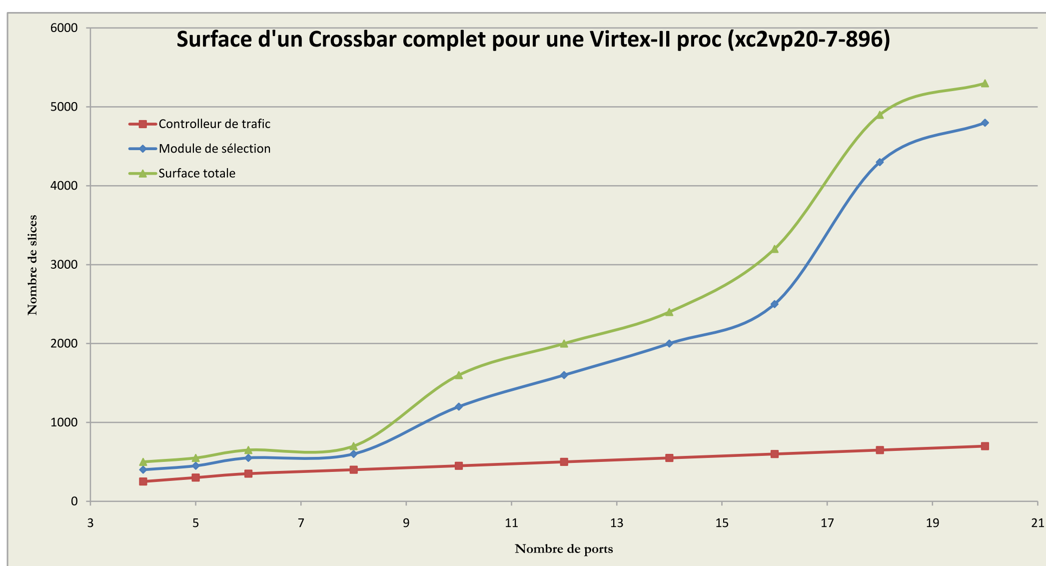


FIGURE 3.6 – Surface d'un crossbar complet.

Afin de repousser la limitation en nombre de ports du crossbar complet, il est possible d'opter pour une topologie de type *Crossbar partiel* présentée en Fig. 3.7. Il s'agit d'un réseau d'interconnexion hybride entre une topologie point à point et partagée. Ceci se traduit par une diminution du nombre de bus, de la surface, de la consommation et en encombrement des fils de communication [53, 60]. En contrepartie, le regroupement d'élément induit une baisse du parallélisme, et plus précisément du nombre de transactions simultanément possibles, et par voie de conséquence de la performance.

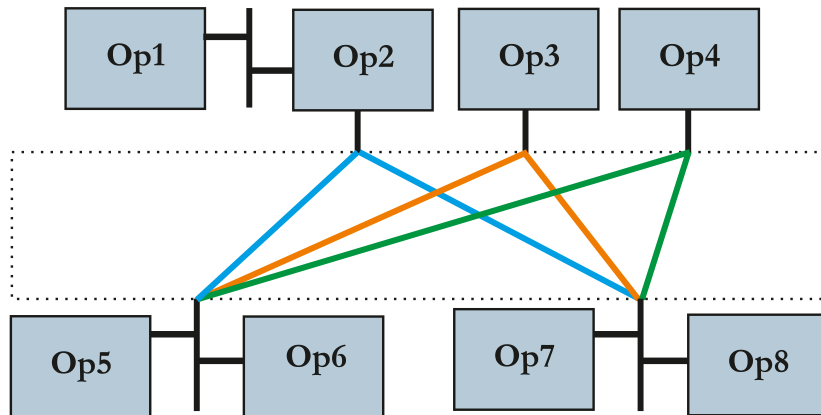


FIGURE 3.7 – Crossbar partiel.

Enfin, une autre topologie communément utilisée dans le cadre de système haute performance est la topologie de *bus en anneaux* (ou "*ring bus topology*" en anglais). Comme le montre la Fig. 3.8, les différents composants sont connectés à un ou plusieurs bus à anneaux concentriques. Les données peuvent être envoyées de la source à la destination dans le sens horaire ou anti-horaire selon, par exemple, la distance la plus courte ou encore la disponibilité du bus. Un exemple de l'utilisation de ce type de topologie peut être trouvé dans l'IBM Cell multiprocesseur [106] notamment utilisé dans la PlayStation 3. Dans ce cas, le *ring bus* a été préféré à une topologie crossbar complet principalement en raison de la plus faible surface nécessaire à son implémentation. Néanmoins, les performances de cette topologie reste inférieures à celle d'un crossbar complet, dans l'hypothèse où le nombre de ports reste dans les limitations exprimées précédemment.

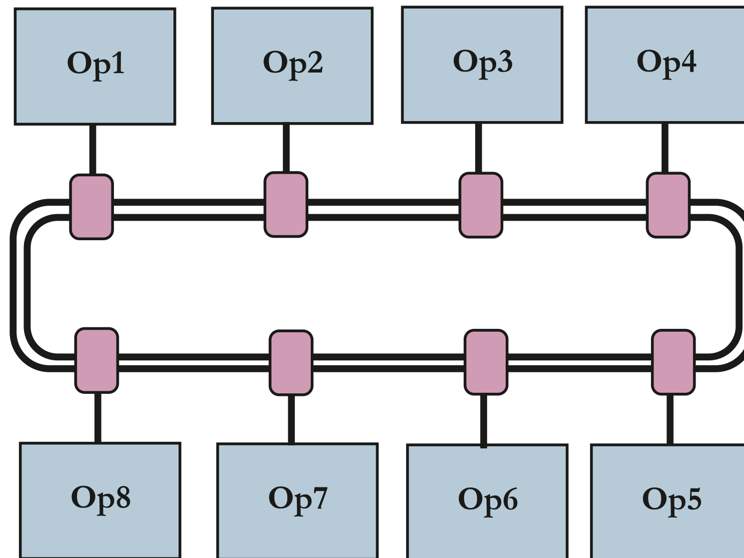


Figure 3.8 – Bus à anneaux.

3.2 Processeurs à chemin de données reconfigurable et leur technique de programmation

Cette section a pour vocation de présenter plusieurs réalisations intégrant un processeur à chemin de données reconfigurable doté d'un réseau d'interconnexions pré-câblé. Nous nous attacherons à étudier le type de topologie utilisée, la granularité des éléments de traitement employés et l'architecture mise en œuvre pour l'exploitation du chemin de données. On s'intéresse aussi à la méthodologie de développement inhérente à chacun de ces systèmes. En effet, les langages de descriptions matériel (ou HDLs pour *Hardware Description Languages*), comme le VHDL ou le Verilog, sont encore largement employés pour la programmation des FGPAs. Ces langages sont très spécifiques et contraignent les concepteurs à avoir de larges notions d'électronique numérique en plus de la connaissance de la syntaxe du langage. Ceci rend difficile le développement d'applications sur FPGAs par des concepteurs non initiés au domaine. Il est donc essentiel d'étudier les méthodologies de développement associées aux architectures à chemin de données reconfigurable afin d'évaluer la programmabilité des architectures proposées. En d'autres termes comment développer, utiliser et reconfigurer des éléments de calcul performants au sein d'un tel dispositif ?

3.2.1 Acadia

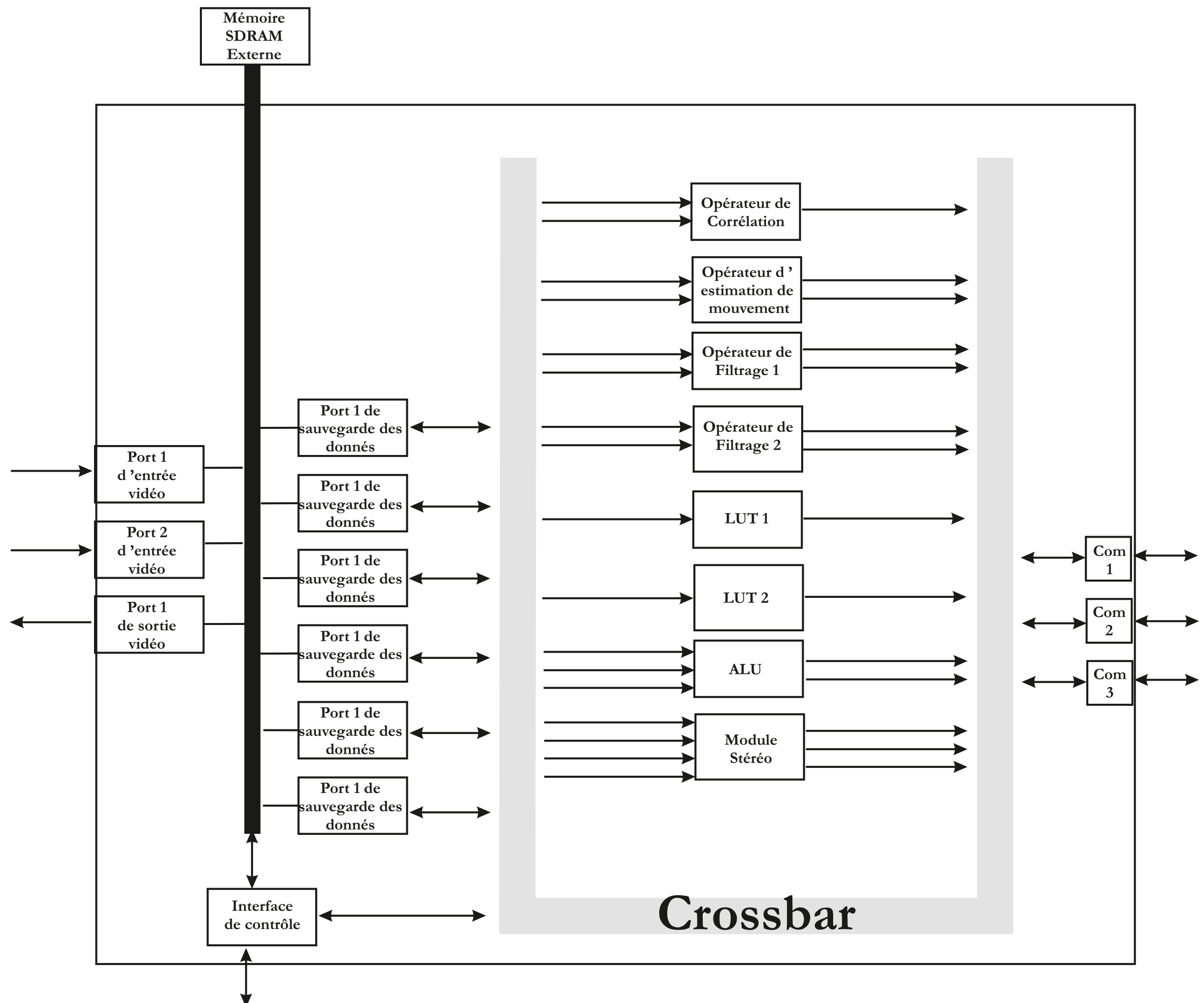


Figure 3.9 – Architecture du processeur Acadia.

Avec l'appui de la *Defense Advanced Research Projects Agency (DARPA)* américaine, la société Sarnoff a développé un processeur de vision nommé Acadia [121] (Fig. 3.9). Ce processeur offre jusqu'à 80 GOPS [3] permettant d'exécuter des traitements de vision à vitesse vidéo. La topologie d'interconnexion utilisée est un crossbar (*Cross Point Switch* sur la Fig. 3.9) sur lequel se greffe une gamme d'éléments de traitements (PE's) variée.

Le processeur Acadia permet d'effectuer des opérations de corrélation, d'esti-

3. GOPS : Giga Opérations Par Seconde

mation de mouvement ou encore de SAD [4]. De plus, l'ajout d'une ALU [5] permet d'effectuer additions, multiplications et divisions sur 1 ou 2 octets. Au niveau performance, comme dit précédemment ce processeur offre jusqu'à 80 GOPS lui permettant notamment de fournir des cartes de profondeur VGA à 30 IPS [6][73].

Le grande nombre de PE's impact inévitablement les ressources matérielles nécessaires. De plus, le processeur Acadia n'intégrant pas de méthode de minimisation de ressources, l'application de cette solution est compromise dans le cadre de notre étude. En effet, si l'on souhaite simplement appliquer un filtrage Gaussien il n'est pas nécessaire d'avoir, par exemple, un module de stéréo vision implanté.

Au niveau programmation, peu d'informations sont données dans [121]. Néanmoins, les modules de traitements ont été modélisés en langage C permettant de représenter la fonctionnalité de chaque module avec une précision au bit prés.

3.2.2 ROMA

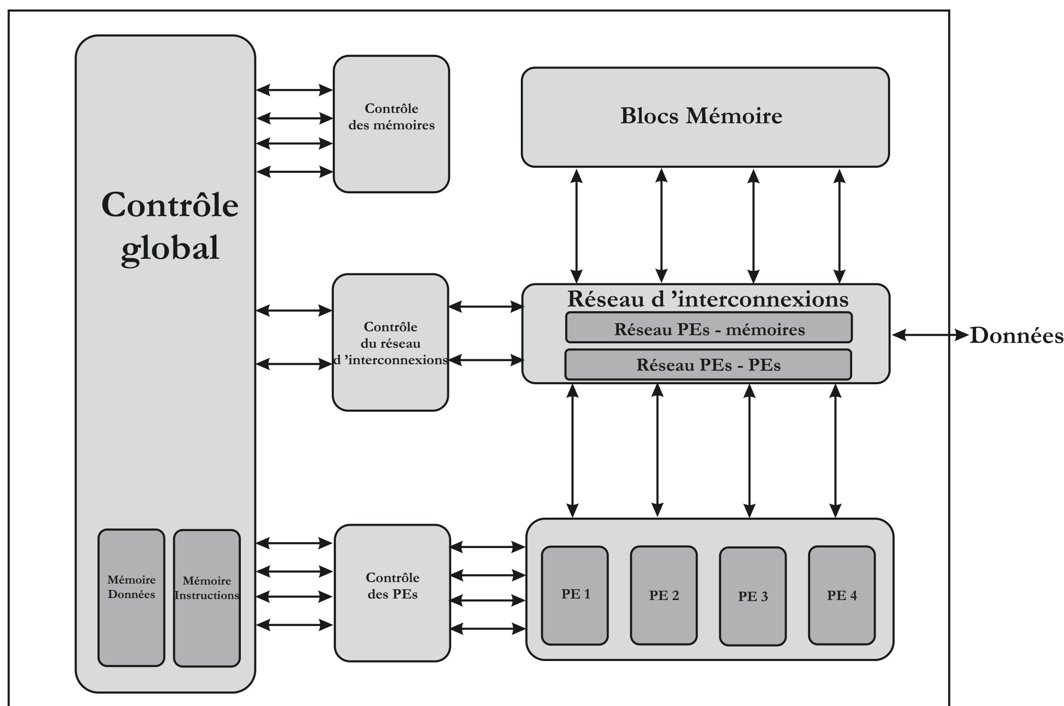


FIGURE 3.10 – Architecture du processeur ROMA.

4. SAD : Somme des différences absolues ou Sum of absolute differences
5. ALU : Arithmetic and Logical Unit
6. IPS : Image Par Seconde

ROMA [22], pour *Reconfigurable Operators for Multimedia Applications*, est issu du projet ANR "*Architectures du Futur*" réunissant des acteurs académiques et industriels. L'objectif de ce projet est la conception d'un processeur reconfigurable capable d'adapter sa structure de traitement aux motifs de calcul pour lesquels un gain significatif en termes de puissance de calcul et de consommation est attendu. Le domaine d'applications visé est le domaine vidéo.

La Fig. 3.10 présente l'architecture du processeur ROMA. Ce processeur intègre deux réseaux d'interconnexion reconfigurables. Le premier permet de connecter les opérateurs entre eux et le second de connecter les opérateurs aux mémoires. La gestion de la reconfiguration est gérée par une structure de contrôle disposant d'un jeu d'instructions dédiées.

Les opérateurs permettent d'exécuter des opérations arithmétiques (ADD, SUB, MUL, ABS, SQRT), logiques (AND, OR, XOR), de décalage ou encore des opérations d'accumulation (SAD, MAC). Un aspect intéressant concernant ces opérateurs est le fait qu'ils soient également reconfigurables permettant ainsi d'optimiser chaque étape de traitement.

Un effort tout particulier a été porté sur la méthodologie de développement du processeur ROMA. On peut distinguer 2 axes sur lesquels la méthodologie se focalise. Le premier est l'optimisation de la taille des données. Cette optimisation peut s'avérer très productive au niveau de la préservation de ressources matérielles. En effet, le besoin de dynamique est très différent entre, par exemple, une multiplication et une soustraction. Grâce à une pré-étude des opérations effectuées, la taille des données peut ainsi être minimisée et permettre une préservation des ressources matérielles ainsi qu'un potentiel gain en performance. Le second axe a pour objectif de sélectionner les parties critiques de l'application et de réaliser les transformations nécessaires à l'optimisation du placement sur l'architecture puis de générer les configurations de l'architecture.

Enfin au niveau de la programmation du processeur ROMA, le concepteur décrit son application en langage C. Cette application est ensuite transformée en un graphe HCDG [7] (*Hierarchical Conditional Dependency Graph*) grâce au compilateur GECOS[107]. La méthodologie de développement s'appuie ensuite sur ce formalisme et sur une librairie d'opérateurs pour à la fois créer le programme C du processeur de contrôle et les éléments HDLs correspondants aux opérateur.

7. Un graphe HCDG est un graphe orienté composé de nœuds et d'arêtes typés. Il sert à représenter les différentes dépendances entre les opérations d'un programme.

3.2.3 ConvNet

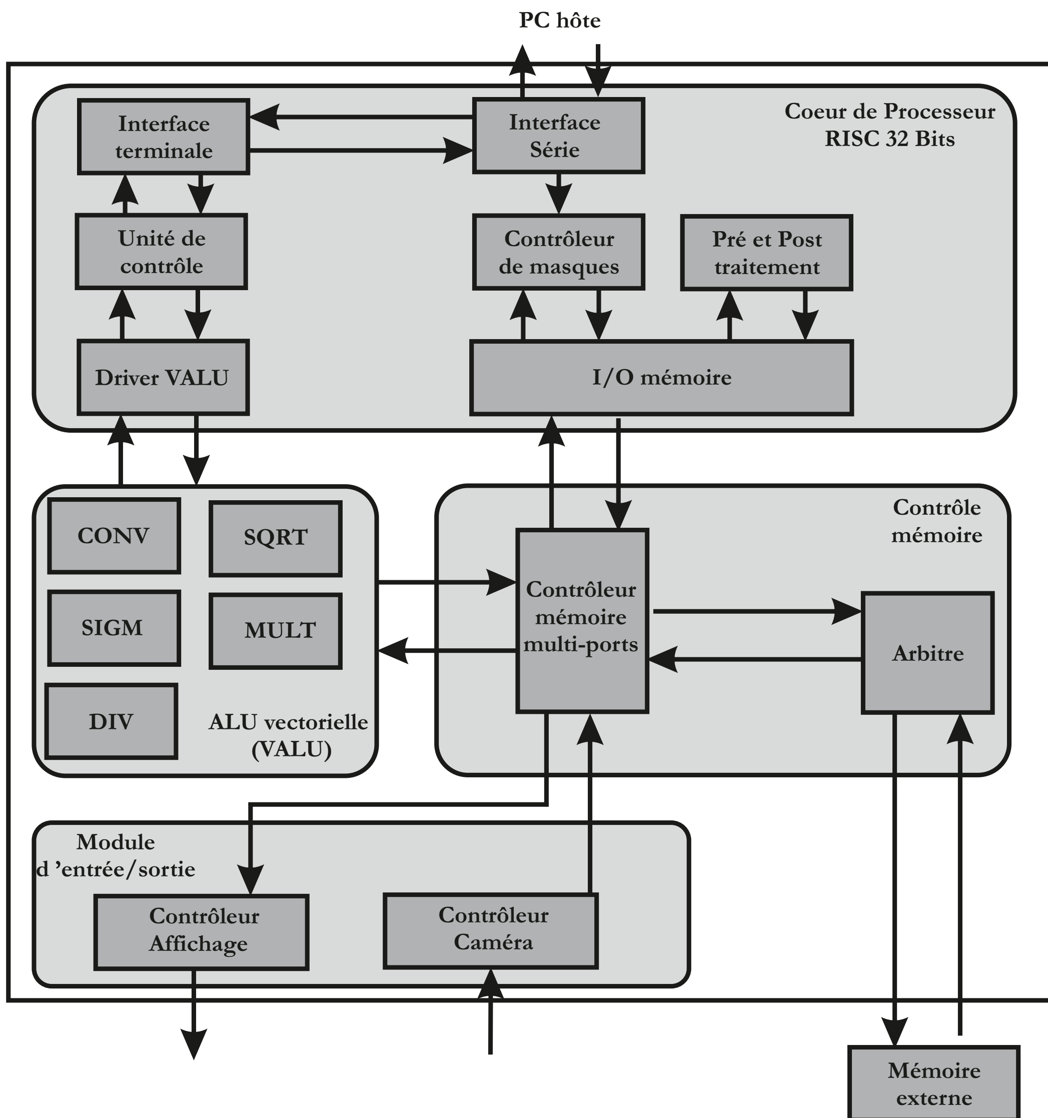


Figure 3.11 – Architecture du processeur ConvNet.

Farabet & al.[30] ont proposé l'implémentation sur FPGA d'un système de vision et de reconnaissance basé sur un réseau de convolution (*convolution network*). Ce système est programmable et peut appliquer la plupart des opérations basées sur des convolutions avec des masques de taille réduite. Le processeur ConvNet intègre un cœur de processeur RISC 32bits associé à un jeu d'instructions vectorielles pour configurer les éléments de calcul de base de l'architecture.

Le processeur ConvNET est composé d'un réseau d'interconnexion simple. Un

arbitre est en charge de la gestion de la communication entre les divers éléments de l'architecture.

Le second élément clé du processeur ConvNet est l'ALU vectorielle, ou VALU. Ce composant permet de réaliser des opérations de multiplication, division ou encore racine carrée. Mais cette VALU inclue également des opérations de convolution 2D ou de groupement et de sous-échantillonnage spatial. Les opérations sont réalisées avec des mots de 16 bits au format virgule fixe pour les coefficients et des mots de 8 bits pour les opérandes. La reconfiguration intervient au niveau de l'interconnexion entre les divers opérateurs ainsi que dans le paramètrage d'éventuels coefficients de masques. À titre d'exemple, la Fig. 3.12 détaille la partie de la VALU responsable du calcul de convolution suivant un masque de dimension 3×3. On peut noter la présence de FIFOs permettant la mise en forme matricielle d'un flot de pixels d'entrée.

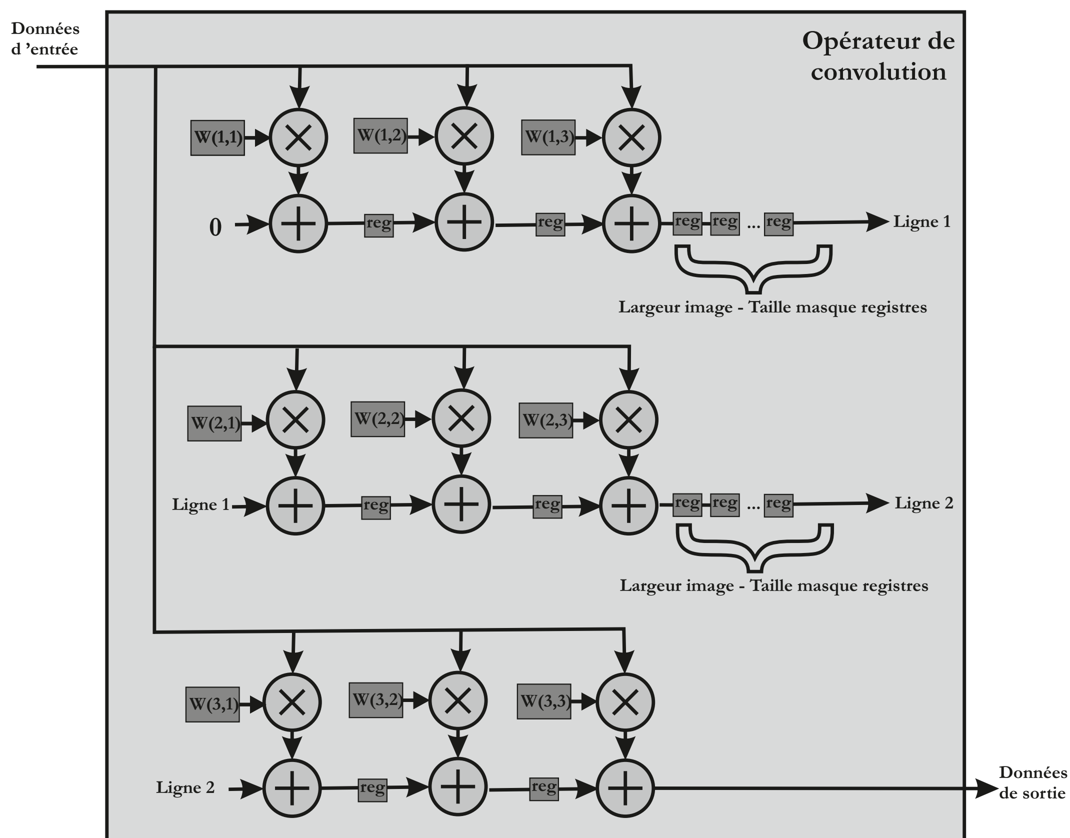


FIGURE 3.12 – Architecture de l'ALU vectorielle du ConvNet responsable des calculs de convolution.

Un exemple d'application développée sur le ConvNet, à savoir une reconnais-

sance de visage, est fourni dans [30]. On peut ainsi voir que le taux d'occupation des ressources est de 89% des ressources d'un FPGA Virtex IV.

La programmation du processeur se fait à travers le langage *Lush* qui est une extension du langage *Lisp*[63]. Le programme *Lush*, une fois compilé avec un compilateur mise au point par Farabet & al., donne le séquencement des appels sur le processeur ConvNet. Ce compilateur permet ainsi d'obtenir un code source donnant à la fois la configuration des éléments de la VALU ou encore les coefficients des masques et les interconnexions entre les divers éléments de l'architecture.

3.2.4 Chameleon CS2000

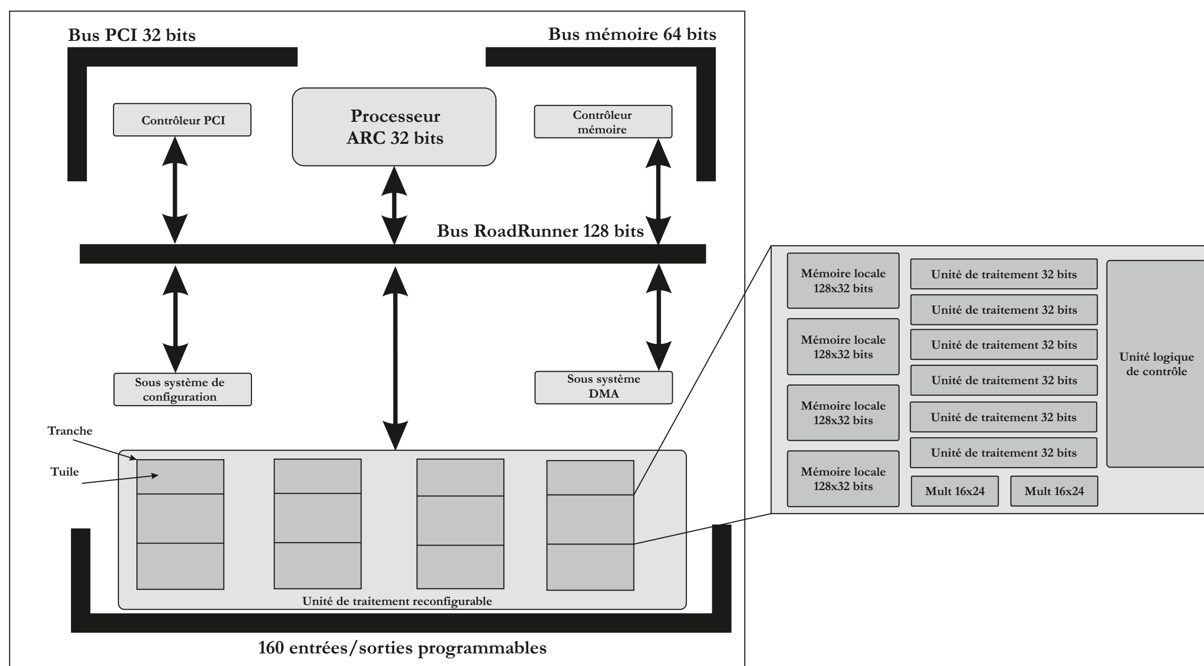


Figure 3.13 – Architecture Chameleon CS2000.

La société Chameleon Systems. Inc.[115] a développé une architecture nommée *Chameleon CS2000*, présentée en Fig. 3.13, regroupant sur un seul circuit un cœur de processeur RISC 32 bits particulier (cœur ARC), un contrôleur de mémoire complet, un contrôleur PCI, le tout connecté à une unité de traitement reconfigurable.

Cette unité est constituée, en fonction de la gamme du composant, d'un certain nombre de *Tranches* (4 dans la Fig. 3.13) intégrant 3 *Tuiles* chacun. Chaque

Tuiles regroupe quatre mémoires locales, sept unités de traitement, deux multiplieurs et de la logique de contrôle.

Les unités de traitement (Fig. 3.14) disposent d'un opérateur 32 bits multifonctions : ADD, ADD16 (2 additions 16 bits en parallèle), MAX, MAX16, MIN, MIN16, PENC (encodeur de priorité), SADD (addition jusqu'à saturation) et SADD16. Les opérandes sont sélectionnés par des multiplexeurs (24 vers 1) et passent ensuite à travers un opérateur de décalage et des masques.

La configuration de chacun de ces éléments est mémorisée dans un registre d'instruction (Fig. 3.14). Huit instructions peuvent être pré-programmées pour chaque unité de traitement. La sélection de l'instruction courante est réalisée par la logique de contrôle.

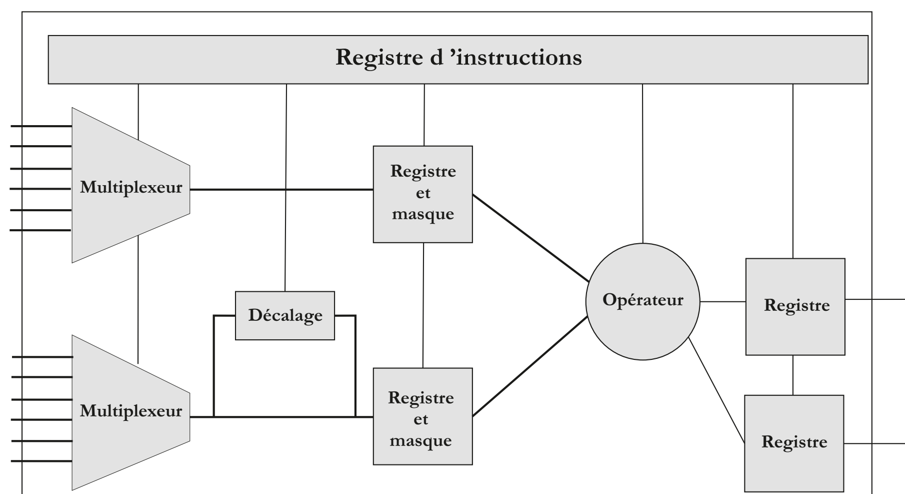


FIGURE 3.14 – Architecture d'un DPU.

Le flot de programmation du *Chameleon* est donné en Fig. 3.15. Le système *Chameleon* intègre un environnement de développement, nommé *C∼side* permettant de concevoir, mettre au point et vérifier le design du *Chameleon. C∼side* propose un compilateur C GNU optimisé pour le cœur de processeur RISC et un synthétiseur Verilog optimisé pour l'unité de traitement reconfigurable. Le code objet et le *bitstream* sont ensuite regroupés dans un exécutable. C∼side propose également un simulateur, baptisé *Chipsim*, modélisant l'ensemble de l'architecture.

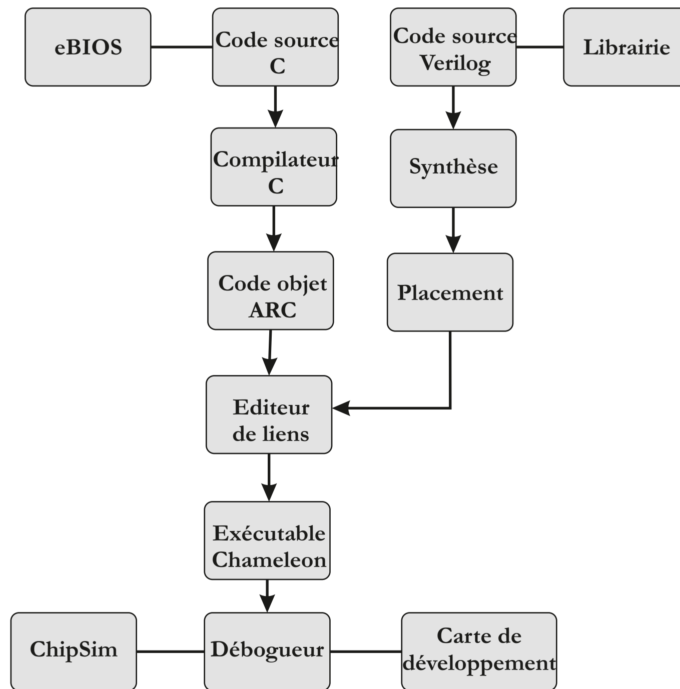


FIGURE 3.15 – Flot de développement sur l'architecture Chameleon.

3.2.5 X.P.P

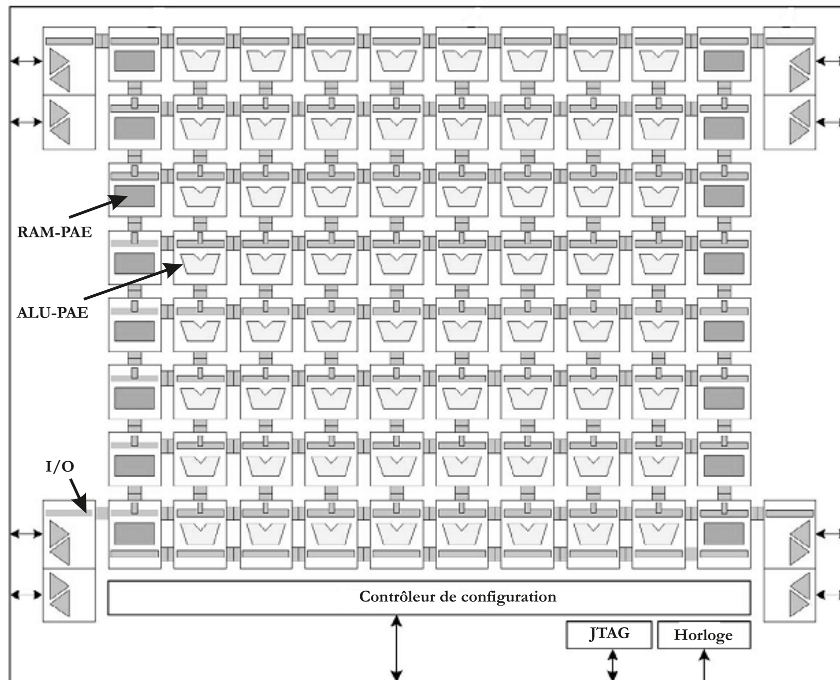


FIGURE 3.16 – Architecture X.P.P.

L'architecture X.P.P. [8], développée par la société PACT XPP Technologies, Inc.[111], est articulée autour d'une matrice d'éléments de traitement reconfigurables connectés par un réseau de communication *packet-oriented*[44].

Le routage des éléments de traitement est assuré par le contrôleur de configuration. Il charge les configurations via une interface externe depuis une S-RAM, dans son cache interne. Puis, il charge la configuration (opcodes, routage des canaux et constantes) dans la matrice.

L'architecture de la partie opérative du X.P.P. (Fig. 3.17) intègre, pour chaque élément de traitement, une ALU permettant des opérations de multiplication, addition, comparaison et décalage. On trouve également un BREG (*Back REGister*) qui autorise le routage du bas vers le haut du chemin de données et qui est aussi composé d'une ALU qui peut être utilisée pour une addition, un décalage ou une tâche de normalisation. Un FREG (*Forward REGister*) permet un routage de shaut en bas et intègre une ALU dédiée à des opérations de routage des

8. X.P.P. pour eXtrem Processing Platform

données telles que le multiplexage et la permutation. Les différents éléments de traitement sont reliés à un réseau de communication permettant des connexions point à point ainsi que point à multi-points entre les entrées/sorties des différents éléments. Jusqu'à huit canaux de données sont disponibles pour chaque direction horizontale.

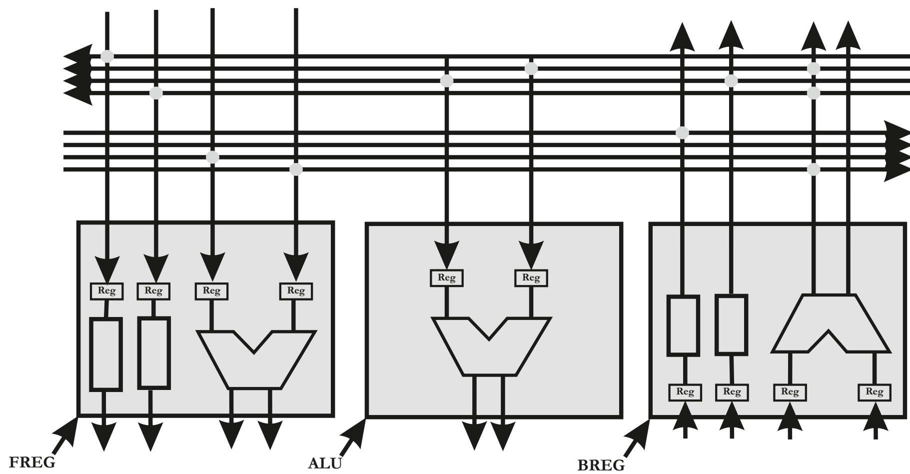


Figure 3.17 – Architecture des éléments de traitement de l'architecture X.P.P.

Afin d'exploiter les possibilités de l'architecture X.P.P., la configuration de cette dernière est décrite selon un langage propriétaire de la société PACT XPP Technologies, Inc., le *Native Mapping Language (NML)*. NML est un langage structurel qui donne accès au concepteur à toutes les caractéristiques matérielles du X.P.P. Les configurations sont définies en instanciant et en plaçant les divers éléments de traitements puis en spécifiant leurs connexions. Le flot de développement sur l'architecture X.P.P. est donné en Fig. 3.18. L'outil XMAP procède au placement des éléments de traitement et aux configurations du réseau d'interconnexion selon la description NML. Le fichier résultat (.xbin) est ensuite utilisé pour configurer l'architecture X.P.P. Le reste du flot de développement permet la mise au point, intègre un simulateur (XSIM) et un outil de visualisation (XVIS) qui analyse l'état de la matrice de XPP tout au long du processus de simulation.

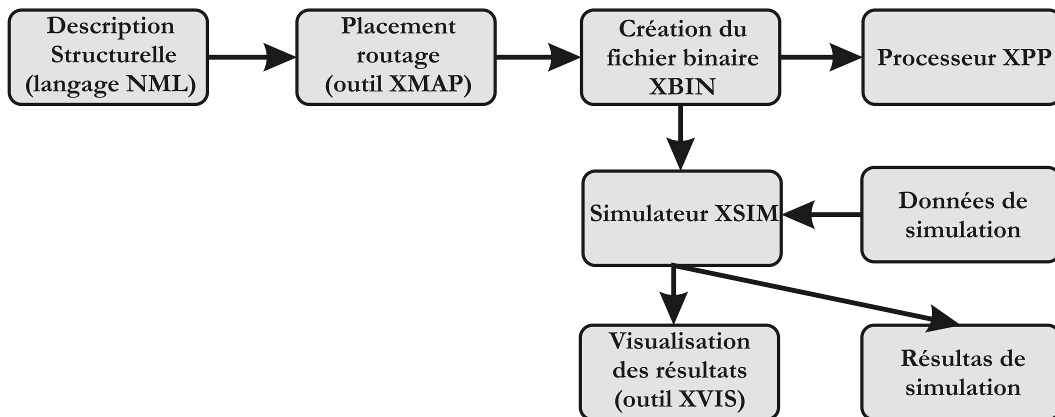


Figure 3.18 – Flot de développement sur l'architecture X.P.P.

3.2.6 IMAPCAR

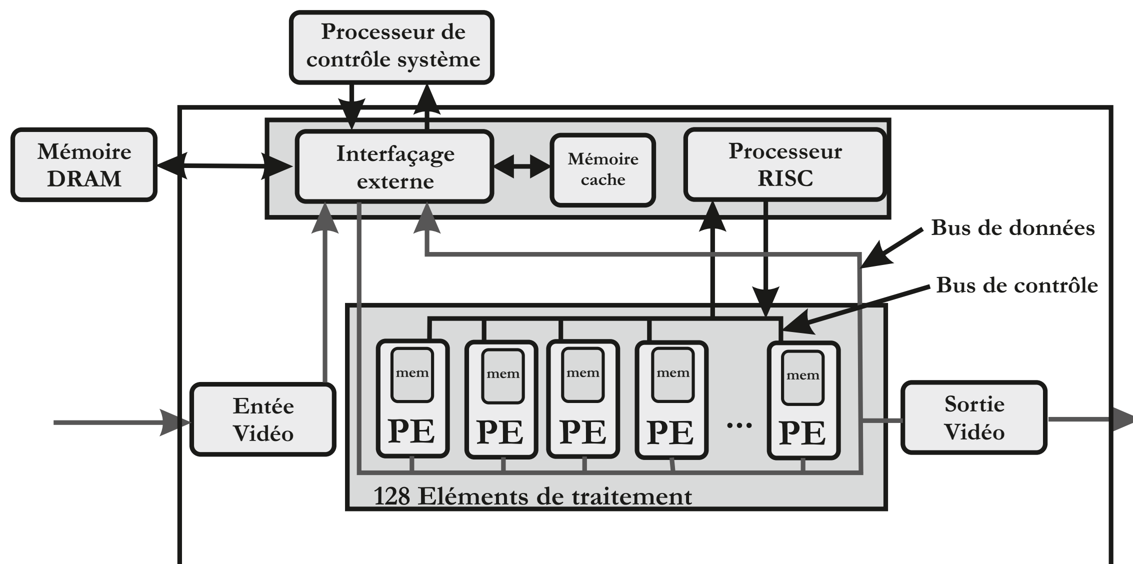


Figure 3.19 – Architecture du processeur IMAPCar.

Le processeur IMAPCar (Integrated Memory Array Processor for Car) [71], de la société NEC, est dédié au traitement d'images embarqué pour l'automobile. Ce processeur, dont l'architecture est présentée en Fig. 3.19, intègre un processeur de type RISC [28] afin de gérer l'ensemble du système. La topologie d'interconnexion utilisée dans le processeur IMAPCar est une topologie de bus en anneaux.

Grâce à ses 128 PEs, connectés sur le bus en anneaux, intégrant un chemin de données de 4 voies de 16 bits, 2 ALUs et un composant MAC de 24 bits, l'IMAP-car offre de remarquables performances de calcul allant jusqu'à 100 GOPS [9]. Un aspect original de ce processeur est qu'il traite l'image d'entrée de manière parallèle ligne par ligne. Néanmoins, l'implémentation d'une telle architecture au sein d'un FPGA requiert d'importantes ressources matérielles.

La programmation de l'IMARCAR est faite à travers une extension du langage C, appelée 1DC [24] (pour one dimensional C). 1DC intègre un nouveau type de variable, *sep* (pour des éléments séparés sur chaque PE), et six extensions au langage C tels que des opérations arithmétiques parallèles, des décalages à gauche/droite, un adressage de tableau ou encore des évaluations de statuts.

3.2.7 Systolic Ring

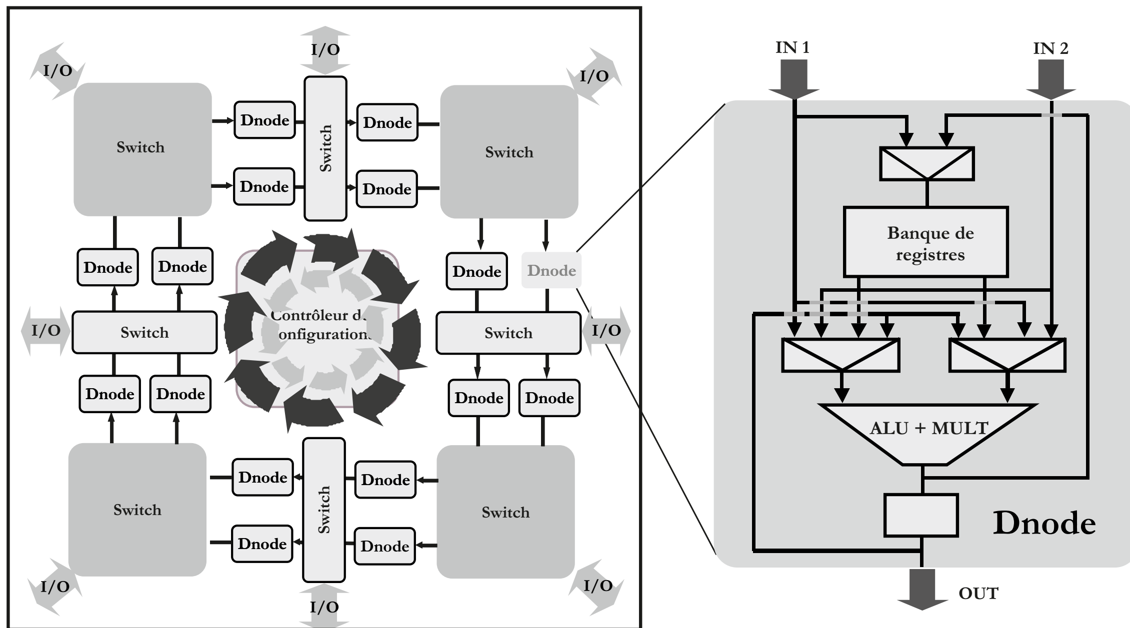


FIGURE 3.20 – Architecture du processeur Systolic Ring.

Le laboratoire LIRMM de Montpellier a développé une architecture dédiée aux traitements en flot de données [122]. Cette architecture, nommée le *Systolic Ring*[84], est composée de 2 éléments de base. Le premier est le *Dnode* et constitue l'élément de base de calcul. Le second est le *Switch* permettant d'alimenter

9. GOPS : Giga Opérations Par Seconde

les Dnodes en données. En plus des contrôleurs intégrés à ces deux éléments de base, le Systolic Ring dispose également d'un contrôleur de configuration globale. Ce contrôleur se présente sous la forme d'un processeur RISC et gère la structure reconfigurable.

Le Systolic Ring dispose de deux réseaux d'interconnexion différents. Le premier est inclus dans le Dnode et permet de réaliser diverses opérations entre les 2 entrées, la banque de registres et l'accumulateur de sortie d'un Dnode. Le second dispositif de communication est basé sur une topologie de bus en anneaux et définit le chemin de données entre les différents Dnode avec une propagation des données unidirectionnelles.

L'aspect traitement est donc symbolisé par un ensemble de Dnodes. Un Dnode est composé d'une ALU proposant toutes les fonctions logiques et arithmétiques classiques comme addition, soustraction mais aussi multiplication. De plus, est également présente une banque de registres qui sera principalement utilisée pour le stockage des résultats de calculs intermédiaires. La taille des mots manipulés choisie est de 16 bits. Enfin une opération MAC (Multiplication-ACcumulation) a été ajoutée au jeu d'instructions. Cette instruction permet ainsi l'exécution d'une somme de produits de n termes en n cycles d'horloge ce qui impacte sur l'aspect temps réel d'éventuelles applications.

Le Systolic Ring est programmé à partir d'une description en langage C. À partir du code C, un graphe de flot de données est généré sous contraintes de latence minimale. Le graphe représentant le flot de données est ensuite partitionné par un algorithme le parcourant à la recherche de motifs issus d'une bibliothèque d'opérateurs connus. Une fois ces sous-graphes identifiés, leurs paramètres propres sont extraits (latences, coefficients, etc.) puis passés à un module d'assemblage qui réalise le code final à exécuter sur le contrôleur de configuration. La Fig. 3.21 présente un exemple d'algorithme et son résultat architectural sur le Systolic Ring.

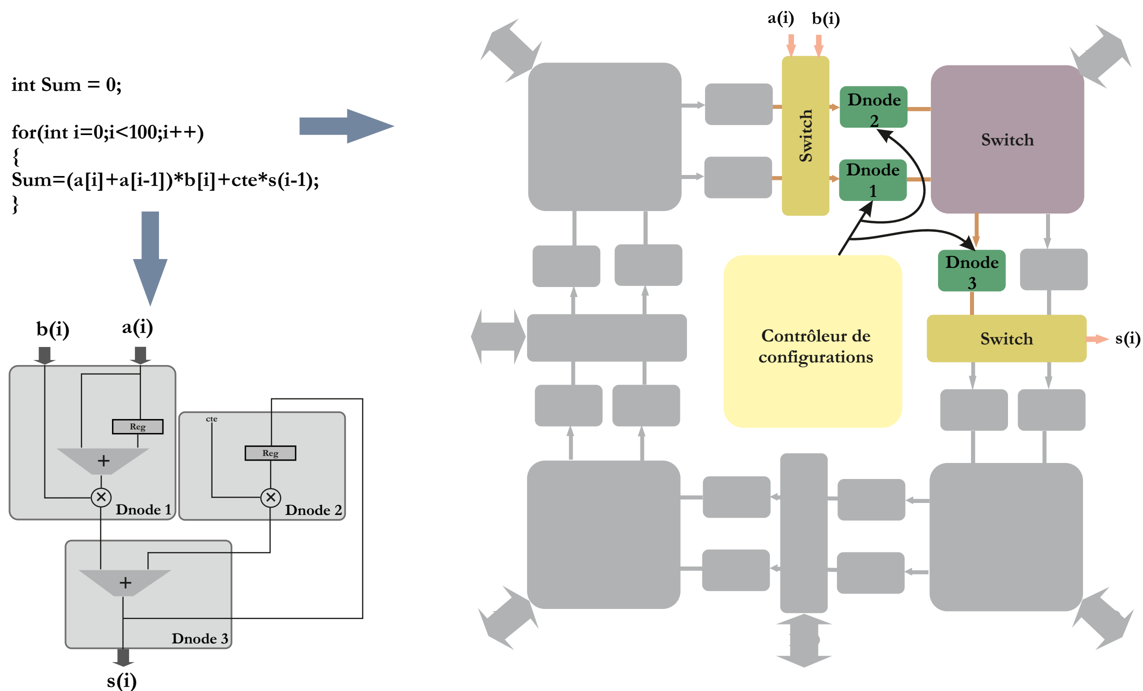


FIGURE 3.21 – Exemple d'algorithme pour le processeur Systolic Ring.

3.2.8 DRIP RTR

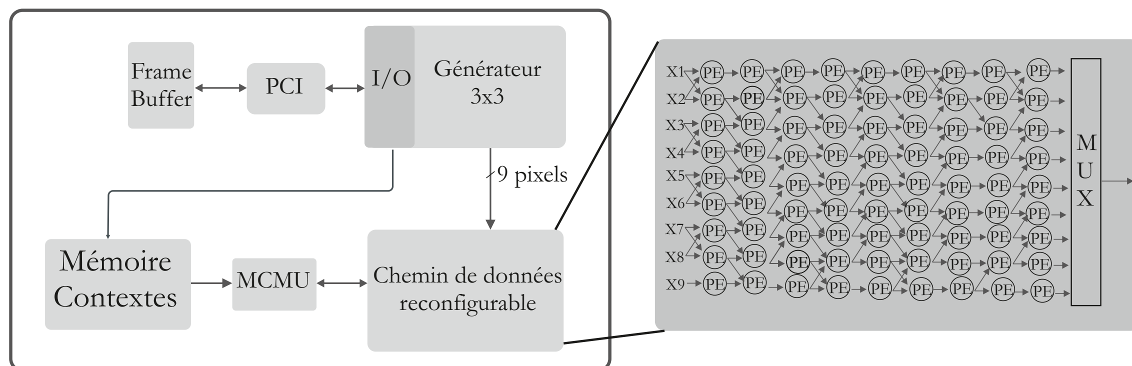


FIGURE 3.22 – Schéma synoptique du DRIP RTR.

Boschetti & al. [46] proposent une architecture à chemin de données reconfigurable dynamiquement visant des applications de traitement de l'image : Le DRIP RTR[36] (Fig. 3.22). Le chemin de données est organisé en matrice bidimensionnelle d'éléments de calcul. La reconfiguration du chemin de données est géré par la *multicontext control management unit (MCMU)* qui va communiquer

avec la mémoire de contextes où les configurations sont stockées. Deux modes de reconfiguration sont disponibles : le mode parallèle dans lequel toutes les colonnes sont reconfigurées en un seul cycle d'horloge et le mode pipeline où, à chaque cycle d'horloge, une nouvelle colonne est reconfigurée.

Le processeur d'I/O gère la communication avec le frame buffer et alimente le générateur de voisinage (voisinage 3x3 dans le cas de la Fig. 3.22). Il reçoit également les configurations du chemin de données afin de les stocker dans la mémoire de contextes.

L'aspect traitement est assuré par une matrice d'éléments de traitement (PEs) de granularité faible, Fig. 3.23. En effet, seulement deux opérations sont disponibles sur chaque PE : MAX et ADD. Néanmoins, il est possible d'affecter des poids (-1,0 ou 1) aux opérandes augmentant ainsi les capacités de traitements de ce processeur à 11 fonctions disponibles par PE. De ce fait, et malgré un taux d'occupation ressources matérielles acceptable, les possibilités de traitement offertes par le DRIP RTR sont très restreintes.

Côté méthodologie de développement, Boschetti & al ont pris en compte le fait que tous les algorithmes potentiellement applicables sur le DRIP sont seulement composés par les 11 fonctions disponibles dans chaque PE, permettant ainsi une forte réutilisation des PEs d'un algorithme à l'autre. Ainsi, le temps de reconfiguration peut être minimisé. Boschetti & al ont ainsi développé un outil, baptisé VDR pour *Visual Interface for Dynamic Reconfiguration*, afin de décrire avec un formalisme graphique de type flot de données, Data Flow Graph (DFG), divers algorithmes de traitement de l'image sur le DRIP. Cet outil exécute, au niveau pré-synthèse, une analyse de similitudes topologiques des différents algorithmes à implanter. Il en résulte un modèle VHDL optimisé de l'application développée. À titre d'exemple, un PE devant exécuter les opérations MAX(X1,X2) et MAX(X1,0) sera minimisé pour obtenir le modèle décrit en Fig. 3.24.

L'aspect de minimisation des éléments de traitement en fonction de l'application décrite par l'utilisateur est un élément intéressant de cette méthodologie. Néanmoins, le fait d'avoir jusqu'à 81 PEs à reconfigurer, ou au moins à commander, risque d'engendrer des pertes de performance non négligeable.

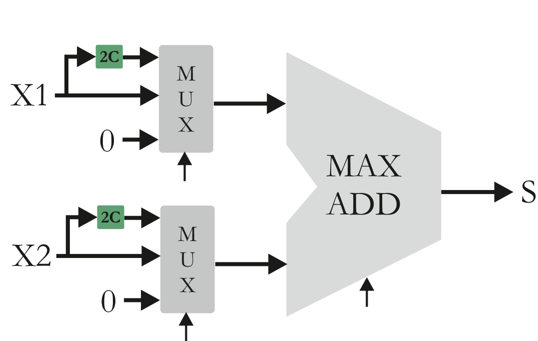


FIGURE 3.23 – Schéma synoptique d'un élément de calcul du processeur DRIP

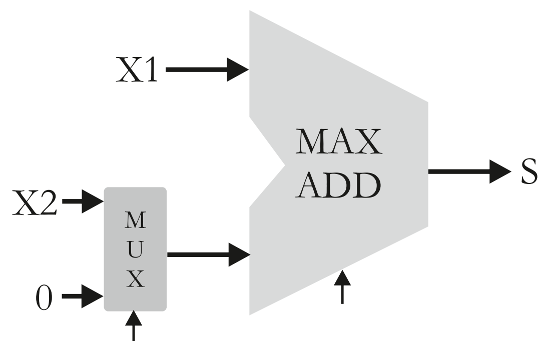


FIGURE 3.24 – Schéma synoptique d'un élément de calcul minimisé du processeur DRIP

3.2.9 DART

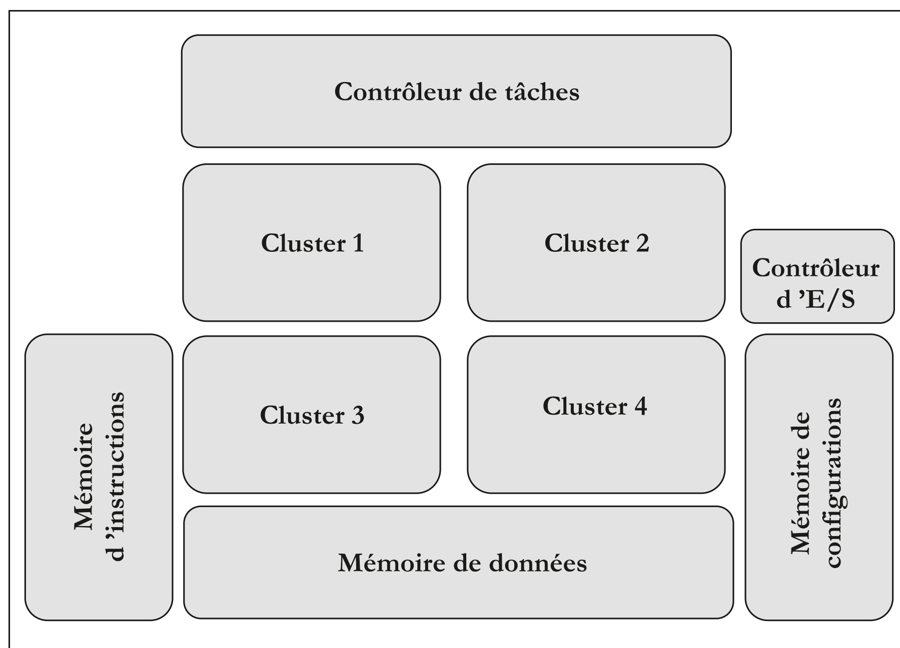


FIGURE 3.25 – Architecture DART.

L'architecture DART [17], développée à l'université de Rennes 1, est une architecture reconfigurable pour des applications mobiles. Cette architecture a été conçue de manière hiérarchique afin, notamment, de facilité la mise en place d'outils de programmation.

Le plus haut niveau hiérarchique de l'architecture DART (Fig. 3.25) est com-

posé d'un contrôleur de tâches en charge de la gestion globale de l'architecture, de ressources de mémorisation et d'un ensemble d'éléments de calculs appelés *clusters*.

Chaque *cluster* intègre 6 DPR [10] reliés par un réseau de bus segmentés, une mémoire de configuration et son contrôleur permettant de paramètrer les différents DPR et une mémoire de données et son contrôleur permettant d'alimenter les DPR (Fig. 3.26). On peut ainsi noter que chaque DRP de chaque *cluster* accèdent aux même données. Le réseau de bus permet d'agencer dynamiquement l'interconnexion des DPR et leur fournit les données à traiter. Il est ainsi possible d'effectuer des traitements indépendants dans chaque DRP ou de réaliser un pipeline de DPR permettant d'exécuter des traitements plus complexes.

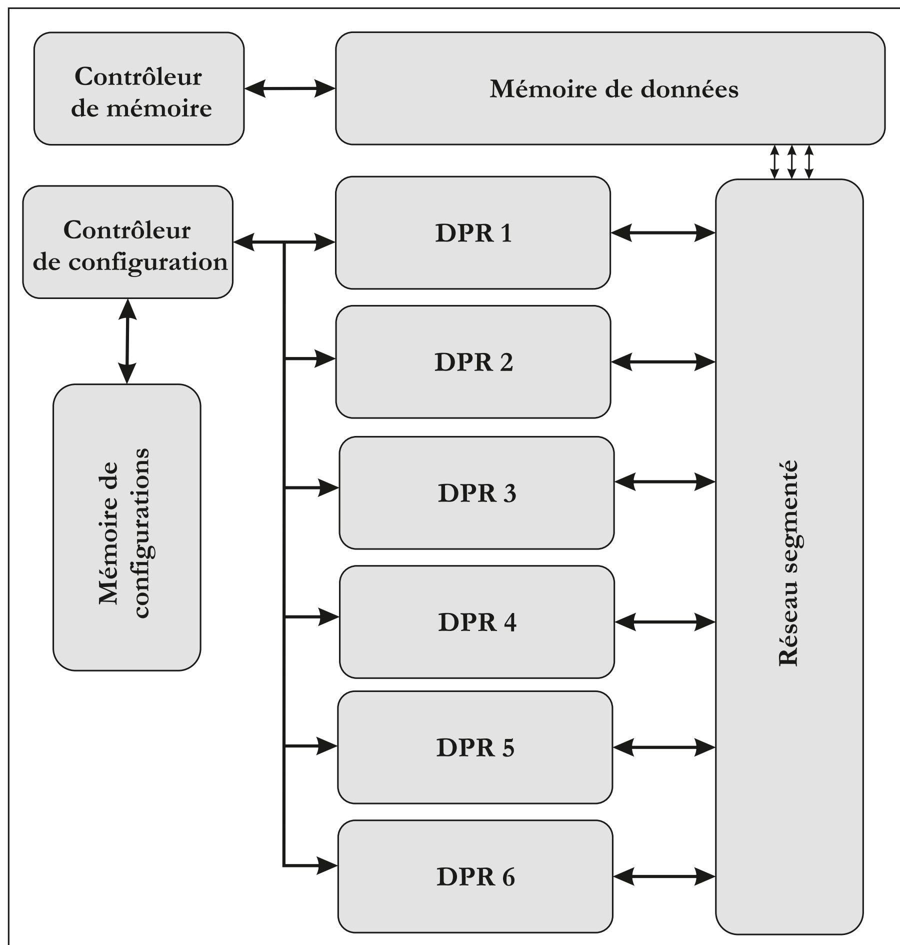


FIGURE 3.26 – Architecture d'un *cluster* de l'architecture DART.

10. DPR pour Data Path Reconfigurable

Enfin, le dernier niveau hiérarchique représente l'architecture de chaque DPR (Fig. 3.27). Cette architecture s'articule autour d'un réseau multi-bus de type crossbar. Le crossbar permet de connecter n'importe quel élément du DPR à un autre et fait la liaison avec le niveau hiérarchique précédent par l'intermédiaire des bus globaux. Chaque DRP comporte deux multiplieurs/additionneurs (MUL1 et MUL2 sur la Fig. 3.27), travaillant sur des données d'entrée de 16 bits et fournissant un résultat sur 32 bits, et deux ALUs (ALU1 et ALU2) admettant des opérandes codés sur 40 bits. Ces ALUs sont capables d'effectuer des opérations arithmétique (addition, soustraction, valeur absolue, valeur minimale, valeur maximale) et logique (et, ou, ou exclusif, non). Ces unités de traitement disposent de 4 mémoires SRAM locales contrôlées par des générateurs d'adresses (AG sur la Fig. 3.27). On peut également noter la présence de 2 registres supplémentaires permettant de créer un décalage ou de stocker temporairement des données. Enfin, il est important de signaler qu'un DPR peut fonctionner selon deux paradigmes d'exécution. Le premier, qualifié de *hardware*, offre une flexibilité totale des DPR, en d'autres termes, les opérations ou encore les interconnexions sont modifiables. Le second, qualifié de *software*, bride la flexibilité d'un DPR et ne permet de réaliser que des séquencements du type *Read-Modify-Write*. Néanmoins, la reconfiguration de ce dernier se fait en un seul cycle là où il en faut plusieurs pour le mode *hardware*.

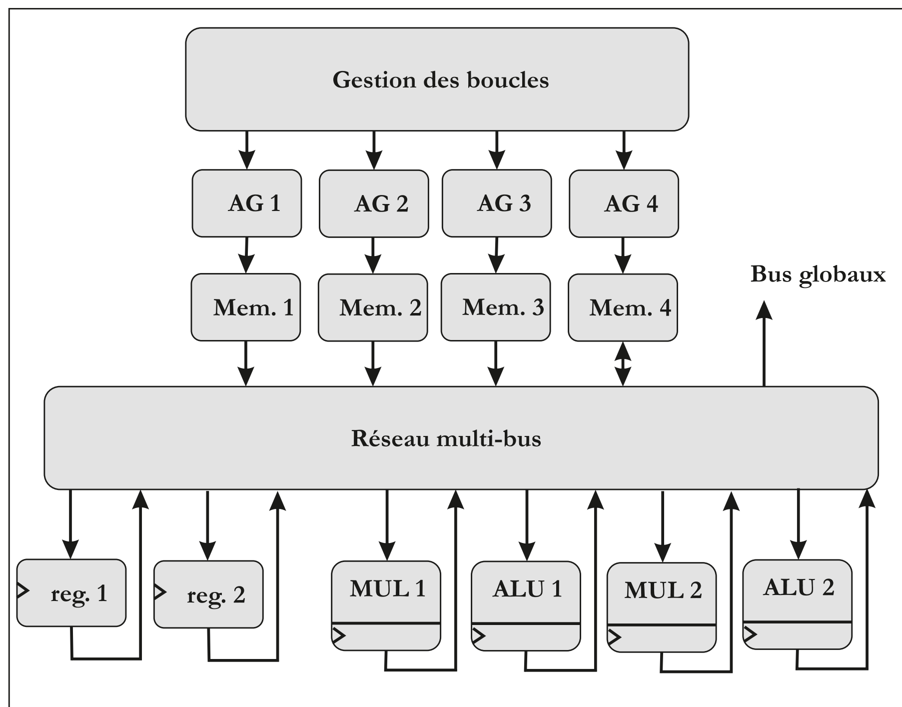


Figure 3.27 – Architecture d'un DPR de l'architecture DART.

Le flot de développement associé à l'architecture DART admet un programme en langage C comme entrée. L'environnement SUIF [97] et les modifications, apportées par les auteurs de l'architecture DART, transforme ce programme en un graphe de flot de données et de contrôle (CDFG) optimisé permettant de séparer les cœurs de boucles et le code irrégulier d'une application. Pour créer les configurations *software* l'architecture DART a été décrite en langage ARMOR [18] puis cette description est compiler avec le compilateur CALIFE [75] afin de générer les mots de configuration. Les configurations *hardware* sont créées à partir de l'outil GAUT [52] qui admet en entrée le cœur des boucles provenant de l'environnement SUIF. Enfin, le code des générateurs d'adresses est obtenu, comme pour les configurations *software*, grâce au compilateur CALIFE. Un modèle SystemC de l'architecture DART a également été développé dans l'optique de pouvoir estimer le temps de calcul et la consommation.

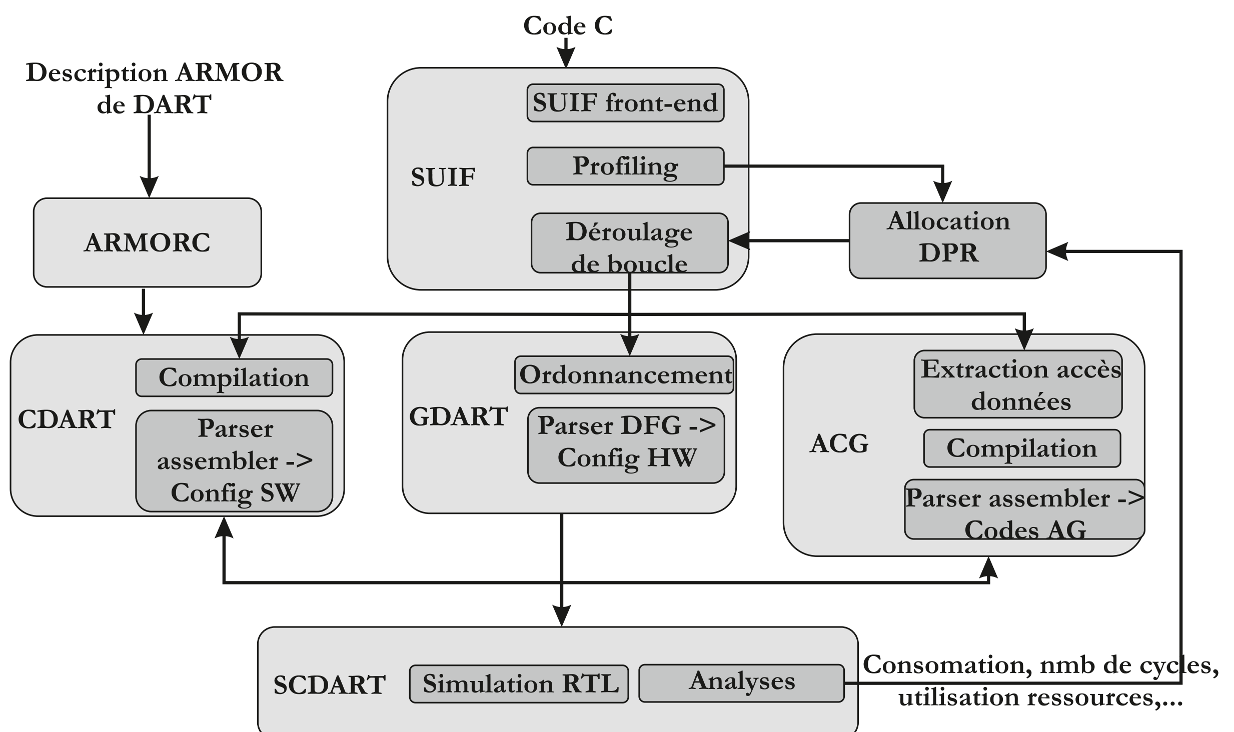


FIGURE 3.28 – Flot de développement associé à l'architecture DART.

3.3 Conclusion

Architecture	Partie contrôle	Topologie réseau	Granularité des PEs
Acadia [121]	externe	Crossbar	Multiple
ROMA [22]	Contrôleur personnalisé	n.c [11]	Opérateurs ADD, MULT, SAD
ConvNet [30]	RISC 32bits	Bus simple	ALU vectorielle
CS2000[115]	ARC (RISC 32bits)	Matrice 2D	Opérateurs ADD, MAX, MIN, PENC
XPP [111]	Contrôleur personnalisé	Crossbar	ALU
IMAPCAR [71]	RISC 32bits	Anneaux	2 ALUs et unité MAC
Systolic Ring [84]	RISC	Anneaux	ALU et unité MAC
DRIP [46]	MCMU	Matrice 2D	Opérateurs ADD, MAX
DART [17] [12]	RISC	Crossbar	2 ALUs et 2 Additionneurs/multiplieurs

Table 3.1 – Tableau de comparaison entre diverses architecture de PCDR.

Concernant la partie de contrôle, le cœur du processeur, on remarque que la plupart des architectures présentées précédemment adoptent un cœur de processeur RISC auquel sont ajoutées des instructions dédiées au contrôle de la partie opérative. L'avantage des processeurs RISC est que le temps de décodage d'une instruction est fixe et ne dépend pas de l'instruction (à l'inverse des processeurs CISC).

La topologie de type crossbar complet est la solution qui offre les meilleures performances et une possibilité de parallélisme parfaitement adaptée au domaine du traitement d'images bas niveau. Néanmoins, il convient de prendre garde aux nombres d'éléments qui seront susceptibles d'être connectés sur ce réseau afin de ne pas aboutir à un trop grand besoin en termes de ressources matérielles ce qui aurait pour conséquence une diminution importante des performances de ce composant.

Au niveau des éléments de traitement, on constate que deux approches peuvent être utilisées. La première consiste à intégrer une large gamme d'opérateurs de granularité différentes (Acadia). L'avantage de cette approche est qu'elle peut s'adapter à divers domaines d'application. Dans le cas de nos travaux, seul le traitement bas niveau de l'image est visé. De ce fait, cette approche ne correspond pas à nos besoins.

La seconde approche consiste en la duplication d'un élément de cas de base (Systolic, Imapcar, Xpp etc...). La difficulté de ce genre d'approche est de trouver l'opérateur de base ayant la granularité adéquate au domaine visé. Il serait

11. n.c. pour non communiqué
12. On considère ici uniquement l'architecture d'un DPR

donc intéressant de trouver une factorisation matérielle aux divers traitements d'image bas niveau afin d'extraire des éléments minimaux de base. Cette factorisation permettrait la mise en évidence d'une *bibliothèque* d'opérateurs élémentaires qui, en les associant, autoriserait la plupart des traitements d'image bas niveau. L'intérêt d'une telle factorisation permettrait ainsi de pouvoir optimiser les ressources matérielles associées à la partie opérative.

Ces travaux de prospection nous ont permis d'extraire un *squelette* d'architecture, à savoir un cœur de processeur RISC associé à un réseau d'interconnexion de type crossbar, en adéquation avec notre objectif de reconfigurabilité dynamique. Il est désormais important de développer un composant de traitement de base pour le traitement de l'image bas niveau. À partir de ce composant, des spécifications peuvent être apportées au *squelette* d'architecture afin de l'adapter aux objectifs fixés en introduction. Enfin la mise en place d'une méthodologie de développement permettant une programmation simple de cette architecture est également à intégrer.

Chapitre 4

Architecture proposée : SeeProc

Ce chapitre présente l'architecture du processeur à chemin reconfigurable développé, et baptisé **SeeProc**. SeeProc est un co-processeur de traitement dédié au traitement d'image bas-niveau. Le but principal de SeeProc n'est pas de contrôler l'imageur ou la carte de communication d'une caméra intelligente mais bel et bien d'effectuer des traitements d'image bas-niveau. SeeProc est donc un co-processeur.

Du chapitre précédent est extrait un ensemble de choix architecturaux en adéquation avec les objectifs présentés en introduction. À ce titre, un cœur de processeur de type RISC[28] est privilégié pour pour la partie contrôle. Le réseau d'interconnexions choisi est un réseau de type Crossbar complet. Cette topologie de réseau offre les meilleures performances en termes de bande passante mais également en terme de flexibilité. Au niveau des opérateurs de traitement, le chapitre précédent a permis de mettre en avant un manque d'élément de calcul générique dédié au domaine du traitement d'image bas niveau. En effet, les PCDRs [1], présentés en Chap. 3, intègrent soit une large gamme d'opérateurs de granularité de calculs variés (Processeur Acadia), soit un opérateur de base, de granularité plus ou moins faible, dupliqué un certain nombre de fois (Systolic Ring ou encore Chameleon CS2000). Dans le premier cas, une sous-utilisation du système en terme de ressources matérielles instanciés est fortement probable en cas d'applications de traitement de l'image bas niveau. Dans le second cas, la granularité des opérateurs impacte les performances du chemin de données. Si la granularité est trop fine, alors pour effectuer une opération de base du traitement de l'image bas niveau -lorsque celle-ci est réalisable (une convolution par exemple)- plusieurs opérateurs sont nécessaires. Ceci incluant donc plus de bus de communication voir plus de cycles d'horloge nécessaires pour réaliser l'opération qu'avec un élément de traitement de granularité optimale pour le domaine visé. Dans le cas d'une granularité trop grande, une sous-utilisation des ressources

1. PCDR pour Processeur à Chemin de Données Reconfigurable

matérielles a lieu.

Afin de contourner cette problématique, SeeProc intègre un élément de calcul générique de granularité adaptée au domaine du traitement de l'image bas niveau. On s'attache dans un premier temps à donner une description fonctionnelle de SeeProc. L'élément de calcul de base et le chemin de données complet sont ensuite décrits. Enfin, la partie de contrôle permettant d'exploiter ce chemin de données est présentée.

4.1 Description fonctionnelle de SeeProc

Afin de répondre aux objectifs exposés en introduction, le processeur SeeProc doit :
- permettre une reconfiguration rapide du chemin de données entre les éléments de calculs,
- utiliser un minimum de ressources matérielles,
- avoir une cadence de fonctionnement temps réel,
- pouvoir s'interfacer facilement avec les autres éléments de l'architecture (Capteurs, mémoires, bus de communication etc...),
- avoir un langage de programmation simple.

L'architecture de *SeeProc* s'articule autour de trois grands éléments :
- une partie de contrôle et d'ordonnancement basée sur un cœur de processeur RISC[28],
- un ensemble d'éléments de calculs,
- un dispositif d'interconnexion reconfigurable.

L'architecture du processeur à chemin de données reconfigurable SeeProc est détaillée en Fig. 4.1. Classiquement, ce processeur peut être scindé en deux parties. Le ***SeeProc_Decoder*** regroupe le cœur de processeur RISC et ses compléments formant ainsi la partie contrôle. Le ***SeeProc_DPR*** est l'association des éléments de calculs et du crossbar, en somme le chemin de données ou la partie opérative. Le processeur SeeProc peut être considéré comme un co-processeur de calculs dédié au traitement d'image bas niveau. En effet, SeeProc ne permet pas le contrôle d'une rétine ou encore d'une carte de communication. Même si, comme présenté par la suite en Sec. 4.3, il intègre un certain nombre de signaux de contrôle, son but n'est pas le contrôle des divers éléments d'une caméra intelligente.

Le chemin de données, **SeeProc_DPR** sur la Fig. 4.1, est composé d'un ensemble d'éléments génériques de calculs dédiés au domaine du traitement d'image bas niveau : c'est ce que nous appellerons des ALUMs [2] par la suite. Ce terme

2. ALUM pour ALU Matricielle

est dû au fait que chaque élément de base admet 2 opérandes d'entrée chacun sous forme d'une matrice carrée de taille fixe et définissable par le programmeur. Chaque ALUM possède également 3 sorties. La raison de la présence de 3 sorties (de 2 dimensions différentes) est que chaque ALUM est décomposée en 3 unités chaînées (**FD**, **FM** et **FR** sur la Fig. 4.1). L'architecture des ALUMs est décrite en Sec. 4.2.1. Enfin, 4 entrées de configuration (1 entrée de configuration par unité et une entrée de paramètrage pour FM) permettent de définir la fonction réalisée.

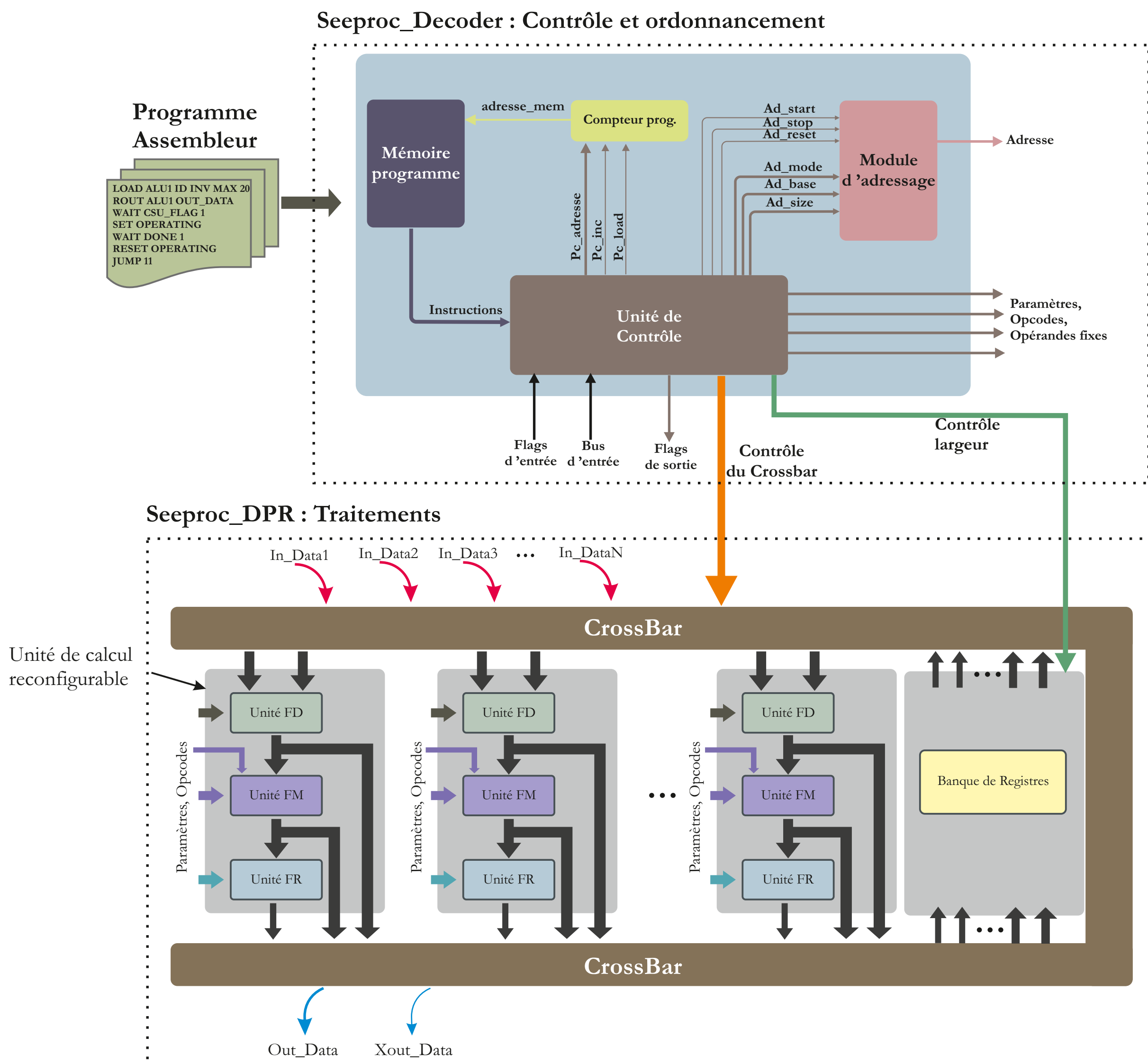


FIGURE 4.1 – Schéma synoptique du processeur *SeeProc*

Afin d'assurer une interconnexion, flexible tout en conservant une bande passante acceptable entre les diverses ALUMs, un crossbar a été intégré. Le crossbar comporte une entrée de configuration commandée par la partie contrôle permettant de définir et redéfinir *dynamiquement* les connexions entre les différents éléments de calculs, les données d'entrée et de sortie. Comme dit précédemment, chaque ALUM dispose de 3 sorties qui ont des dimensions différentes (les sorties de FD et FM sont sous forme matricielle $n \times n$ et la sortie de FR est un scalaire). Une banque de registre est également intégrée dans le chemin de données. Ainsi la sortie FR d'une ALUM peut alimenter un opérande d'une autre ALUM. De la même façon, si une ALUM admet des opérandes de taille 3×3, ses sorties FD ou FM peuvent tout à fait alimenter les opérandes d'une ALUM admettant des opérandes de taille 5×5 et vice-versa. Les processus de dimensionnement de bus sont détaillés en Sec. 4.2.2.4.

La partie contrôle, **SeeProc_Decoder** sur la Fig. 4.1, assure le décodage et l'exécution des instructions présentes en mémoire programme.

Classiquement, le **SeeProc_Decoder** intègre :

- une mémoire programme, dans laquelle est stocké le programme à exécuter en format binaire,
- un compteur programme pointant sur l'adresse, dans la mémoire programme, de l'instruction à exécuter,
- une unité de contrôle cablée responsable du décodage et de l'exécution de l'instruction.

La partie opérative de SeeProc étant spécifique, la partie contrôle admet de ce fait des parties personnalisées adaptées au contrôle du chemin de données. Ainsi si une instruction spécifique au chemin de données est décodée, l'unité **SeeProc_Decoder** mettra à jour les signaux de configuration des ALUMs ou le mot de configuration du crossbar par exemple. L'unité de contrôle dispose, à cette fin, notamment, d'un bus de sortie de 72 bits afin de contrôler directement le crossbar.

Afin de s'intégrer au mieux à son environnement - dans notre cas les divers composants d'une caméra intelligente - le **SeeProc_Decoder** dispose de signaux de contrôle propres permettant la mise en place d'un protocole de synchronisation par *handshaking*. En effet, dans l'hypothèse où un imageur délivre des images avec un temps de latence entre 2 images (ce qui est le cas le plus souvent), ce protocole permet de mettre le SeeProc en attente et ainsi de ne pas effectuer de calcul sur des données erronées. Enfin, un module d'adressage a également été inclus dans l'unité **SeeProc_Decoder**. Ce choix se justifie dans la mesure où il est commun d'avoir une ou plusieures mémoires dans une caméra intelligente. SeeProc est ainsi capable, de manière autonome, d'écrire ou lire des données présentes en mémoire.

SeeProc se programme à l'aide d'un ensemble d'instructions. Ces instructions sont écrites de manière séquentielle au sein d'un programme assembleur. Ce programme est ensuite transformé, de manière automatique, en format binaire puis est chargé dans la mémoire programme. Seeproc se charge ensuite de décoder et d'exécuter séquentiellement les instructions présentes en mémoire programme. Ces instructions peuvent être des instructions de configuration, de contrôle ou encore de test. Les instructions sont composées, comme le montre le Tab. 4.1, d'un opcode de 8 bits, et de deux opérandes de 32 bits. Les instructions sont détaillées en Sec. 4.3.2.

Opcode	Opérande 1	Opérande 2
$8bits$	$32bits$	$32bits$

TABLE 4.1 – Format des instructions

4.2 Architecture du chemin de données

Le chemin de données est construit autour d'un réseau d'interconnexion de type crossbar, d'un certain nombre d'unités de calcul génériques et reconfigurables et d'une banque de registres en série. Le schéma synoptique de cette partie, baptisée ***Seeproc_DPR*** est présenté en Fig. 4.2.

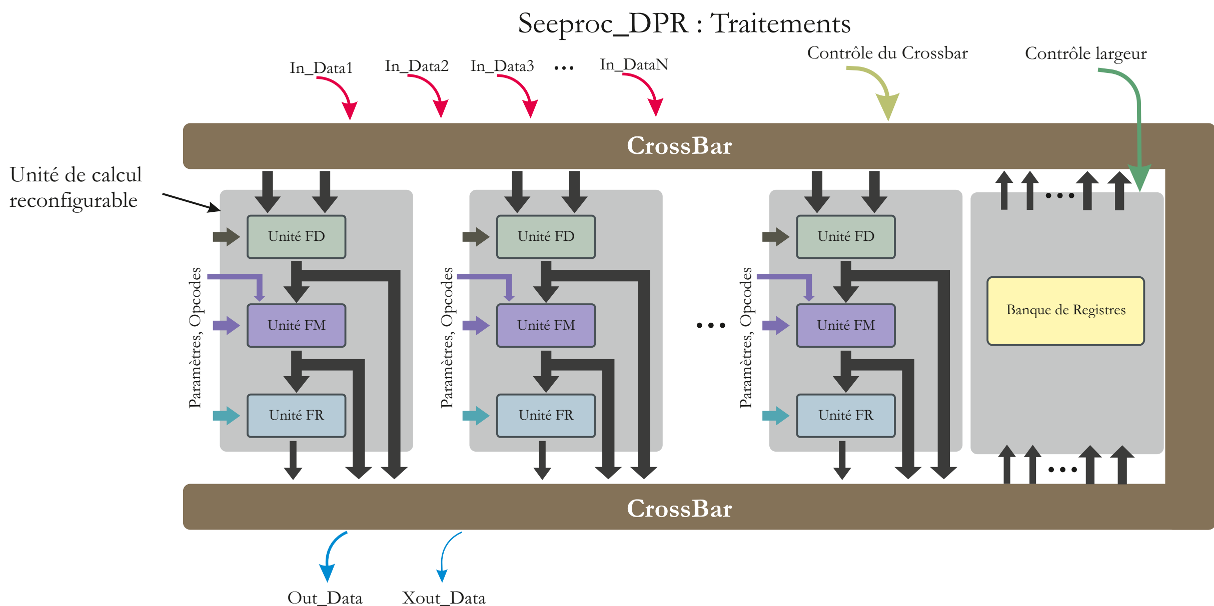


FIGURE 4.2 – Schéma synoptique de la partie de traitement du processeur *Seeproc*

On décrit dans un premier temps les éléments de traitement qui peuvent être assimilés à des unités arithmétiques et logiques travaillant sur des groupes de pixels. Puis la mise en œuvre des ces composants au sein du crossbar est expliquée.

4.2.1 Définition et formalisation des unités de traitement

À partir des travaux présentés dans [41], il a été possible de décomposer la plupart des traitements d'images bas niveau sous la forme d'une composition de 3 fonctions élémentaires. Cette décomposition est faite de la manière suivante :

$$\left\{ \begin{array}{r} Q_{(n,n)} = \mathbf{F_M}(\mathbf{F_D}(A_{(n,n)}, B_{(n,n)})) \\ x = \mathbf{F_R}(Q_{(n,n)}) \end{array} \right. \tag{4.1}$$

où $A_{(n,n)}$ et $B_{(n,n)}$ sont des opérandes d'entrée (images, motifs, masques etc...), sous forme de matrice carrée de taille $n \times n$, $Q_{(n,n)}$ est une image résultat de même forme que les opérandes d'entrée et x est un scalaire résultat.

• La première fonction $\mathbf{F_D} : (\mathbb{Z}^{n^2}, \mathbb{Z}^{n^2}) \rightarrow \mathbb{Z}^{n^2}$ est appliquée indépendamment pour chaque paire d'éléments des opérandes $A_{(n\times n)}$ et $B_{(n\times n)}$. Le résultat de la fonction $\mathbf{F_D}$ est une image $R_{(n\times n)}$ où :

$$R_{(n,n)} = \mathbf{F_D}(A_{(n,n)}, B_{(n,n)}) \tag{4.2}$$

Typiquement, la fonction $\mathbf{F_D}$ est une simple opération SIMD logique ou arithmétique.

• La seconde fonction $\mathbf{F_M} : \mathbb{Z}^{n^2} \rightarrow \mathbb{Z}^{n^2}$ admet seulement une opérande de type image. Cette fonction est appliquée de façon indépendante pour chaque élément de R, et produit le résultat de type image $Q_{(n,n)}$ telle que :

$$Q_{(n,n)} = \mathbf{F_M}(R_{(n,n)}) \tag{4.3}$$

Classiquement, la fonction $\mathbf{F_M}$ peut être une normalisation, un seuillage, un calcul de valeur absolue,...

• La dernière opération est une fonction de réduction notée $\mathbf{F_R} : (\mathbb{Z}^{n^2}) \rightarrow \mathbb{Z}$, appliquée sur les n^2 éléments de l'image ($Q_{(n,n)}$, et donnant un résultat scalaire x tel que :

$$x = \mathbf{F_R}(Q_{(n,n)}) \tag{4.4}$$

A partir de ces 3 fonctions, il est alors possible de composer un certain nombre de pré-traitements comme cela est montré dans le Tab. 4.2.1.

Opérations	F_D	F_M	F_R
Convolution	A * B	R/k	$\sum Q$
Correlation par SAD	A - B	$\|R\|$	$\sum Q$
Correlation par SSD	A - B	R^2	$\sum Q$
Filtrage Max	A * 1	R	$Max\ Q$
Différence d'images	A - B	$\|R\|$	-
Binarisation	A - B	$thold(R)$	-
Différence binaire érodée	A - B	$thold(R)$	AND Q
Dilatation binaire	A and B	R	OR Q
Erosion binaire	A or !B	R	AND Q
Dilatation en NdG	A + B	R	$Max\ Q$
Erosion en NdG	A - B	R	$Min\ Q$

TABLE 4.2 – Liste des possibilités de traitements d'une ALU matricielle.

A partir de la formalisation proposée précédemment, il a été possible de définir des éléments de calculs génériques dont le schéma synoptique est donné en Fig. 4.3 et de montrer que de tels éléments, baptisés ALUs matricielles (ALUMs), sont composés de 3 unités SIMD.

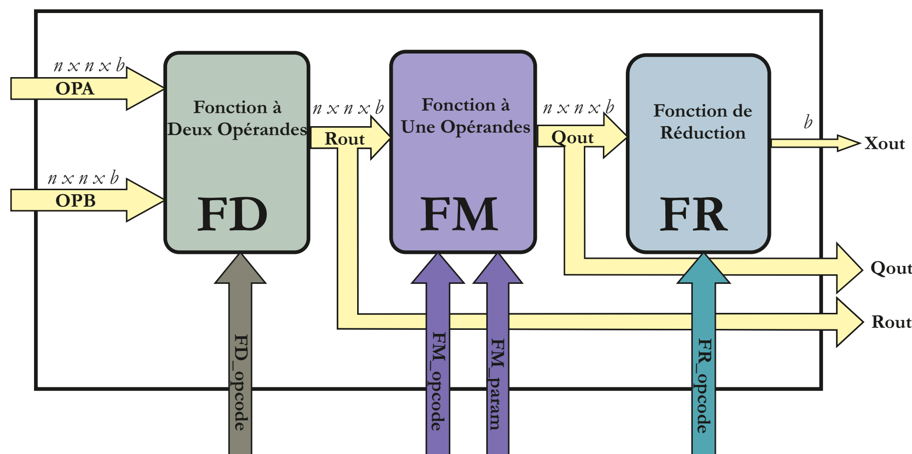


FIGURE 4.3 – Schéma synoptique d'une ALU matricielle

La première unité SIMD, **FD**, est capable d'exécuter n^2 opérations simultanément, $n \times n$ étant la taille de la fenêtre d'intérêt de l'image d'entrée. Les opérations possibles sont l'addition, la soustraction et la multiplication, plus les opérations logiques AND, OR et XOR. Il est également possible d'appliquer une opération d'identité, c'est à dire de court-circuiter l'opérande d'entrée opA vers

la sortie. La sortie de cette unité (Rout) est disponible en sortie de l'ALU. En fait, l'ALUM complète admet 3 sorties permettant d'accéder aux sorties des 3 différentes unités la composant. L'utilisation de Rout peut s'avérer utile lorsque, par exemple, on désire faire une différence d'image puis en extraire le gradient. Le format de Rout est donc de $n \times n \times b$ bits[3].

La seconde unité SIMD, **FM**, offre la possibilité d'effectuer des opérations telles qu'un calcul de valeur absolue, un seuillage, un décalage à gauche ou à droite de bits ou encore la fonction logique NOT. Pour cette unité, une entrée supplémentaire est prévue pour fixer un paramètre (seuil, décalage). De manière analogue à la précédente unité, $\mathbf{F}_M$, fournit Qout à la sortie de l'ALU. Le format de Qout est également de $n \times n \times b$ bits.

La troisième unité, **FR**, est une fonction de réduction. Les opérations disponibles sont les fonctions logiques AND et OR, la somme et l'extraction de la valeur maximale ou minimale. Ces opérations sont appliquées sur les données d'entrée en les combinant deux à deux, dans une structure d'arbre dyadique. Le format de Xout est de $1 \times b$ bits.

La liste des opérations possibles par étage est donnée par le Tab. 4.3.

Fonction FD	Fonction FM	Fonction FR
Rout = opA	Qout = Rout	Xout = Qout(pixel central)
Rout = opA + opB	Qout = -Rout	Xout = $\sum$Qout(n,n)
Rout = opA - opB	Qout = $\|Rout\|$	Xout = MAX(Qout(n,n))
Rout = opA * opB	Qout = Rout^2	Xout = MIN(Qout(n,n))
Rout = opA AND opB	Qout = SL[4](Rout,FM_param)	Xout = AND(Qout(n,n))
Rout = opA OR opB	Qout = SR[5](Rout,FM_param)	Xout = OR(Qout(n,n))
Rout = opA XOR opB	Qout = seuil(Rout,FM_param)	Xout = XOR(Qout(n,n))

TABLE 4.3 – Liste des opérations réalisables par **FD**, **FM** et **FR**.

4.2.1.1 Exemples d'opérations de voisinage réalisables avec une ALUM

4.2.1.1.1 Convolution La convolution est une des opérations de bas-niveau le plus souvent utilisée. Selon le masque de convolution employé, différents résultats

3. b représente la résolution en nombre de bits d'un pixel
1. SL signifie décalage à gauche (shift left)
2. SR signifie décalage à droite (shift right). Dans ce cas les bits introduits pour réaliser le décalage adoptent la valeur du MSB du mot d'origine afin de conserver le signe.

peuvent être obtenus (gradient vertical, horizontal, filtrage gaussien, etc...). L'équation Eq. 4.5 représente la formule générale de l'opération, où A est une fenêtre de taille $(n \times n)$ autour du pixel (i, j) de l'image d'entrée, et B est le masque de convolution, également de taille $(n \times n)$:

$$Conv(i,j) = \sum_{n=i-\frac{w}{2},m=j-\frac{w}{2}}^{i+\frac{w}{2},j+\frac{w}{2}} \frac{1}{k(n,m)} A(n,m) \times B(n,m) \tag{4.5}$$

La matrice k est une matrice de pondération, dépendant de la nature et de la taille (w) du masque utilisé. Cette matrice peut être intégrée à la matrice B. Dans notre cas, il est plus simple de l'extraire afin de permettre une décomposition plus fine des l'opération de convolution. L'opération décrite ci-dessus est appliquée pour chaque pixel de l'image d'entrée, et produit un pixel de l'image résultante (dans le cas de la formule ci-dessus, la gestion des bords n'est pas pris en compte). Il est possible de décomposer le calcul de la convolution autour du pixel (i, j) en une multiplication pour la fonction **FD**, une division (ou un décalage de k bits si k est une puissance de 2) pour **FM** et une somme pour **FR**.

4.2.1.1.2 Corrélation Un autre exemple d'opération de voisinage fréquemment utilisée est le calcul de corrélation. Cette méthode est très courante pour l'appariement de primitives, afin de détecter la présence d'un certain motif (objet ou partie d'objet par exemple) dans une zone de l'image. Le motif recherché est défini par un échantillon d'image de taille $(n \times n)$, qui est ensuite comparé à chaque portion de l'image d'entrée afin de détecter sa présence et position. Une des fonctions de corrélation les plus utilisées est la somme des différences absolues, ou SAD, décrite dans l'équation Eq. 4.6, où A est un échantillon d'image autour du pixel (i, j), et B est le motif recherché :

$$SAD(i,j) = \sum_{n=i-\frac{w}{2},m=j-\frac{w}{2}}^{i+\frac{w}{2},j+\frac{w}{2}} |A(n,m) - B(n,m)| \tag{4.6}$$

On peut décomposer une corrélation SAD par une soustraction pour **FD**, un calcul de valeur absolue pour **FM** et une somme pour FR.

4.2.1.1.3 Différence d'images D'autres exemples de traitements d'image bas niveau sont les différences d'images (Eq. 4.7) et différences d'images binarisées par seuillage (Eq. 4.8).

$$Diff(i,j) = |A(i,j) - B(i,j)| \tag{4.7}$$

$$Diffthr(i,j) = thold(A(i,j) - B(i,j)) \quad (4.8)$$

Ces opérations peuvent être appliquées pour la détection de mouvement [61]. Les opérandes A et B correspondent à deux images, ou partie d'images, consécutives d'une séquence. Dans ce cas, la fonction **FD** est une soustraction, **FM** est la fonction valeur absolue pour Diff ou la fonction seuillage pour Diffthr . Comme le résultat de ces opérations est une image, et non pas un scalaire, l'application de la fonction de réduction **FR** n'est pas nécessaire.

La fonction thold() binarise une donnée en fonction de sa valeur et d'un seuil constant et pré-défini :

$$thold(x) = (1 \quad si \quad |x| \quad > \quad seuil) \quad , \quad (0 \quad sinon) \quad (4.9)$$

Stricto sensu, la différence d'images n'est pas une opération de voisinage, car le résultat relatif à un pixel ne dépend pas de ses pixels voisins. Néanmoins, le modèle de calcul proposé peut être facilement adapté à cette opération, en court-circuitant la fonction **FR**. Cela veut dire que si un module de traitement est capable d'implémenter ce modèle de calcul, il sera également capable d'effectuer des opérations de différence d'images en routant la sortie de la fonction **FM** vers la sortie de la VALU.

4.2.1.1.4 Transformations morphologiques Des opérations de morphologie mathématiques, telles que la dilatation (Eq. 4.10 en niveau de gris) et l'érosion (Eq. 4.11 en niveau de gris), sont souvent utilisées pour la détection de contours et la segmentation d'images. Ces opérateurs peut également être décrits dans le modèle proposé, en considérant que l'opérande B est l'élément structurant de l'opération.

$$(A \oplus B)(i,j) = \max_{(i-x),(j-y) \in D_I,(x,y) \in D_B} (A(i-x, j-y) + B(x,y)) \quad (4.10)$$

où D_I est le domaine de l'image (resp. de l'élément structurant).

$$(A \ominus B)(i,j) = \min_{(i+x),(j+y) \in D_I,(x,y) \in D_B} (A(i+x, j+y) - B(x,y)) \quad (4.11)$$

où D_I (resp. D_B) est le domaine de l'image (resp. de l'élément structurant).

Ainsi pour une dilatation (resp. érosion) en niveau de gris, on aura une addition (resp. soustraction) pour **FD**, un court-circuit pour **FM** et une sélection de

maximum (resp. minimum) pour **FR**.

La Fig. 4.4 présente le principe d'une dilatation binaire et celui de l'érosion binaire avec deux éléments structurant (E.S.).

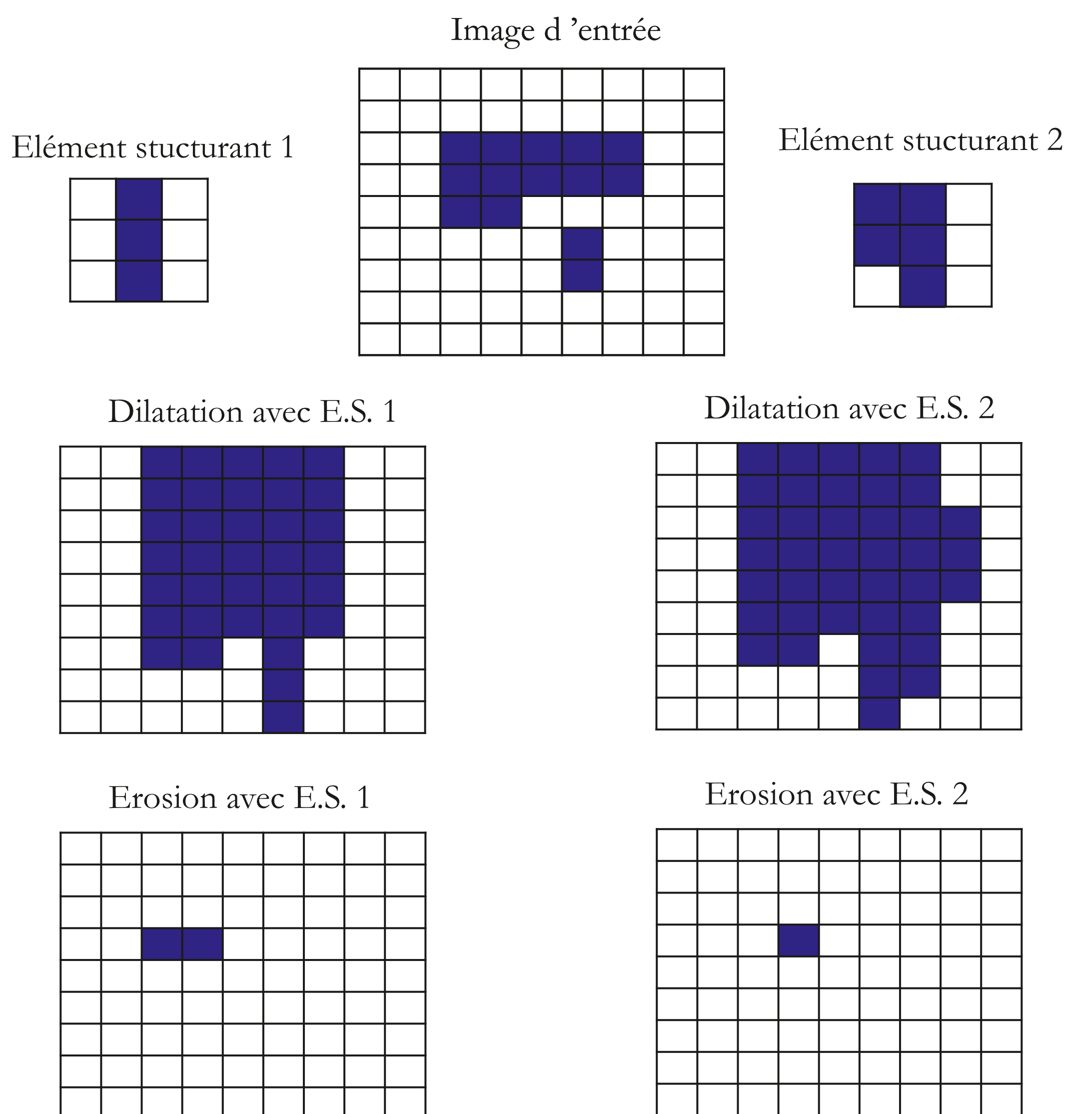


FIGURE 4.4 – Exemple de dilatation et d'érosion binaire avec 2 éléments structurants

4.2.2 Implantation des ALUs matricielles

4.2.2.1 Intégration matérielle

D'un point de vue matériel, les trois unités FD, FM et FR ont été implémentées de deux façons. Le point commun entre les 2 approches est le fait que la partie combinatoire, comme illustré sur la Fig. 4.5, est encapsulée entre des registres. La but de cette encapsulation est de synchroniser la fréquence des traitements avec la fréquence des pixels d'entrée. De plus, tous les traitements de chaque unité n'ont pas le même temps de calcul. Typiquement, la fonction **SUM** de l'unité FR a un temps de calcul plus long que la fonction **OR**. Les registres placés en sortie imposent ainsi que les résultats et les pixels d'entrée soient produits (resp. lus) à la même cadence, cette cadence étant $\frac{1}{T_{clk}}$). Ce dispositif est mis en place grâce à l'utilisation des *adaptateurs* de bus présentés dans la section suivante.

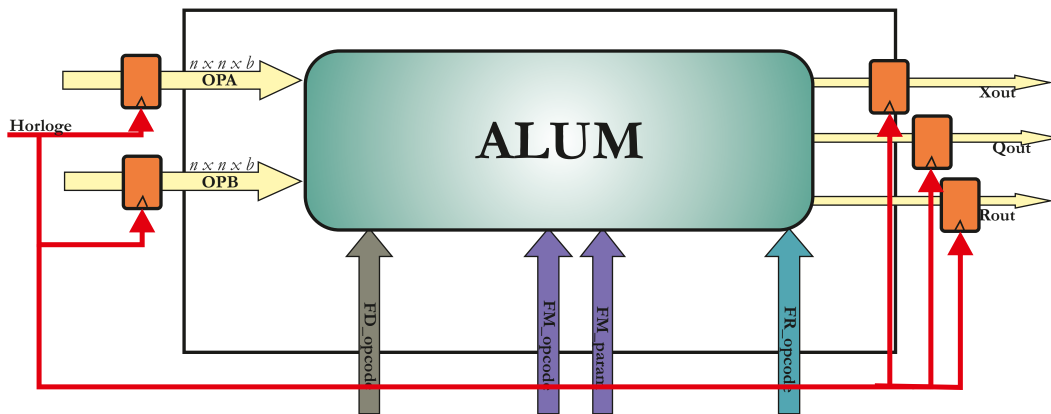


FIGURE 4.5 – Intégration matérielle des ALUMs

La première approche, appelée *méthode directe*, n'intègre pas de registres intermédiaires et est écrite pour produire un résultat tous les cycles d'horloge, sans phase de remplissage de pipeline. Ici la cadence maximale correspond à $\frac{1}{T_{dir}}$) où $T_{dir} = T_{FD} + T_{FM} + T_{FR}$.

La seconde méthode, que l'on appellera *méthode pipeline* par la suite et schématisée par la Fig. 4.6 , se distingue de par le fait que des registres sont également intégrés entre chaque unité de traitement (FD,FM et FR) toujours afin de synchroniser l'obtention des données mais cette fois-ci au niveau de la sortie de chaque unité de traitement. Il faut donc 3 cycles d'horloge par ALUM afin de remplir le pipeline de registres et obtenir des résultats valides. Mais en contrepartie, la cadence maximale est désormais de $\frac{1}{T_{pip}}$) où $T_{pip} = Max(T_{FD}, T_{FM}, T_{FR})$.

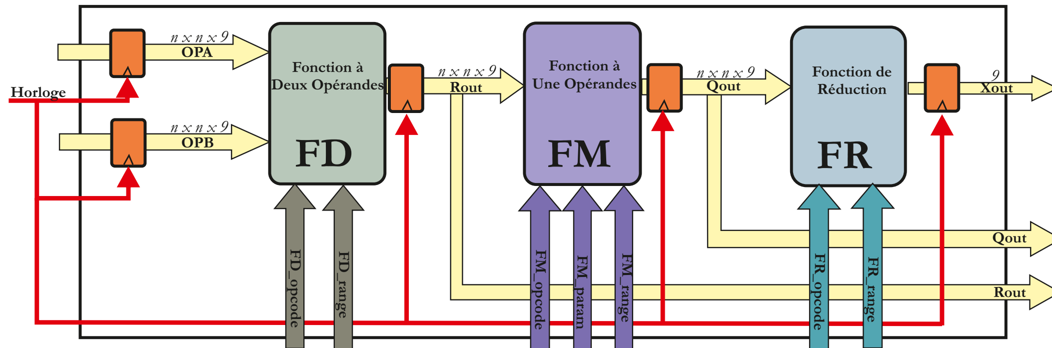


FIGURE 4.6 – Schéma synoptique d'une ALUM intégrant la gestion dynamique de la gamme des pixels résultats

Afin d'illustrer les différences de performance, un programme assembleur est compilé avec les 2 méthodes et les performances des ALUMs générées, sur un Stratix I, sont rapportées dans le Tab. 4.4.

Méthode	Fréquence de fonctionnement maximale
Direct	75,84 MHz
Pipeline	103,21 MHz

TABLE 4.4 – Format des instructions

Soit un gain de 26.5%. Pour cette raison, la *méthode pipeline* a été privilégiée et mise en place.

Dans la version actuelle de SeeProc, chaque unité de traitement (FD,FM,FR) admet en entrée des données sous la forme $n \times n \times 9$ bits. Or, on peut voir que parmi les opérations disponibles au niveau de chaque unité la présence d'opérations de multiplication, de mise au carré ou encore de somme de plusieurs résultats. Ces opérations impliquent un résultat sur 18 bits maximum. Le programmeur dispose donc d'une instruction (Sec. 4.3.2.1) lui permettant de sélectionner les 9 bits du résultat à placer en sortie. Ce choix peut être amené à évoluer au cours de l'application. Pour permettre une modification dynamique de la sélection, une entrée supplémentaire est rajoutée sur chaque unité de chaque ALUM. Ainsi, le schéma synoptique final d'une ALUM est donné la Fig. 4.6.

Concernant la résolution des données de chaque pixel d'entrée (b sur la Fig. 4.5), elle est fixée à 9 bits. Ce choix se justifie par le fait que généralement les pixels délivrés par un imageur sont codés 8 bits. Néanmoins, les opérations d'une ALUM sont prévues pour traiter des nombres signés. Ainsi, afin de conserver la résolution

pixelique d'origine, un bit placé en MSB doit être ajouté. Dans la version actuelle de SeeProc cet ajout est encore manuel et doit ainsi être réalisé en VHDL.

4.2.2.2 Le dispositif de reconfiguration du chemin de données

La partie Seeproc_DPR du processeur SeeProc intègre un dispositif de Crossbar (Fig. 4.7) permettant la reconfiguration en temps réel du chemin de données. Ce dernier est un élément déterminant de cette architecture. En effet, comme expliqué en Sec. 3.1.1, il autorise des modifications dynamiques du chemin de donnée sans avoir recours à une reconfiguration du FPGA.

Un autre aspect, déterminant dans le choix de cet élément, est le fait qu'il offre une meilleure bande passante comparativement aux topologies simple bus et bus hiérarchique. Néanmoins, comme dit précédemment en Sec. 3.1.1, il est communément admis qu'au delà de dix unités connectées le crossbar n'est plus adapté [95]. Afin de contourner cette limitation, nous proposerons, au Chap. 5, une méthodologie de développement qui intègre un processus de minimisation du nombre de ports du crossbar.

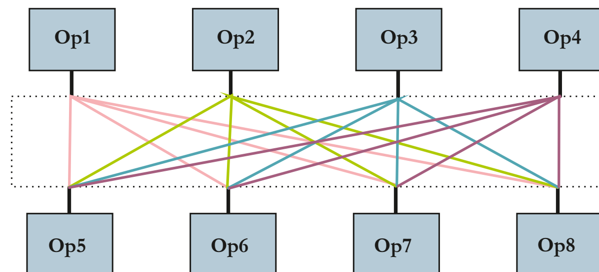


FIGURE 4.7 – Schéma synoptique d'un Crossbar

La configuration du Crossbar est réalisée à partir d'un mot de contrôle indiquant à quelle sortie chaque entrée est reliée. La largeur du mot a arbitrairement été fixée à 72 bits mais peut être augmentée ou diminuée en fonction des besoins architecturaux. Une largeur de mot de contrôle de 72 bits permet la connexion de 6 ALUs matricielles. Le nombre d'ALU maximal en fonction de la taille du mot de contrôle du Crossbar peut être calculé de la façon suivante :

– Les sorties des ALUMs sont les entrées du Crossbar,
– Chaque ALUM procède 3 sorties (Rout, Qout, Xout),
– Pour chaque ALU, il faut prévoir 2 entrées supplémentaires au niveau du Crossbar afin de fournir les opérandes A et B (In_DATAn),

– Le mot de contrôle est divisé en fonction du nombre d'entrées du Crossbar.

Le nombre d'entrées du Crossbar répond ainsi à la formule suivante :

$N_{entrees} = 3 \times N_{ALUM} + 2 \times N_{ALUM}$.

Ce nombre, une fois codé en binaire, permet de fractionner le mot de contrôle du crossbar. Le Tab. 4.5 liste de manière non exhaustive le nombre de sorties possibles en fonction du nombre d'ALU connectées au Crossbar. Ce nombre de sorties du Crossbar peut également être calculé avec la formule suivante :

$N_{sorties} = 2 \times N_{ALU} + 2$.

En effet, le Crossbar fournit les opérandes aux ALUMs, d'où le $2 * N_{ALU}$ auxquels il convient d'ajouter les deux sorties globales représentées par Xout_DATA et OUT_DATA sur la Fig. 4.2.

N_{ALUM}	$N_{entrees}$	Taille du diviseur	$N_{sorties}$	Nombre de sorties possible
2	10	4 bits	6	$72/4 = 18$
3	15	4 bits	8	$72/4 = 18$
5	25	5 bits	12	$72/5 \simeq 14$
6	30	5 bits	14	$72/5 \simeq 14$
7	35	6 bits	16	$72/6 = 12$
8	40	6 bits	18	$72/5 = 12$

Table 4.5 – Exemple de possibilités de connexion au Crossbar avec un mot de contrôle de 72 bits

D'après le Tab. 4.5, on peut voir qu'il n'est donc pas permis, par exemple, avec un mot de contrôle de 72 bits, d'exploiter plus de 6 ALUMs en parallèle. Néanmoins, grâce à la méthodologie de développement (Chap. 5), on peut contourner cette limitation puisqu'il est fort possible que les entrées/sorties de ces 6 ALUMs ne soient pas toutes utilisées.

4.2.2.3 Accès aux données

Dans le domaine du traitement d'images bas niveau, le principal goulot d'étranglement se situe en général au niveau de l'accès aux données, plus précisément entre l'imageur ou la mémoire contenant l'image et le processeur chargé de réaliser les différents traitements.

Le problème est exacerbé lorsque, comme dans le cas de Seeproc, les traitements sont réalisés sur des fenêtres. En effet, il est alors nécessaire de stocker temporairement tous les pixels de la zone d'intérêt dans, par exemple, une file de registres (FIFO).

Plusieurs travaux [108], [15] proposent des méthodes d'optimisation de stockage et d'accès aux données en mémoire pour des architectures embarquées. Seeproc pouvant être vu comme un co-processeur dans le sens où il n'est pas conçu pour contrôler directement un imageur par exemple, la gestion de la mise en forme d'un flot de pixel en matrice carrée peut être considérée comme étant à la charge du processeur principal. Néanmoins, SeeProc peut assurer la mise en forme d'un flot de pixels grâce à un composant que l'utilisateur peut choisir d'utiliser ou non. Ce mode de mise en forme est ce que l'on appellera par la suite *l'accès pixel*.

L'accès pixel est réalisé à l'aide d'un pipeline composé de registres en série et appelé **Seeproc_PixelGrabber** (**SPG**). Les Fig. 4.8 et Fig. 4.9 présentent le principe de la mise en forme réalisée par un SPG pour une image de largeur 5 pixels et un opérande d'entrée de 3×3 pixels [6] :

6. Dans sa version actuelle, le composant SPG ne gère pas les effets de bords

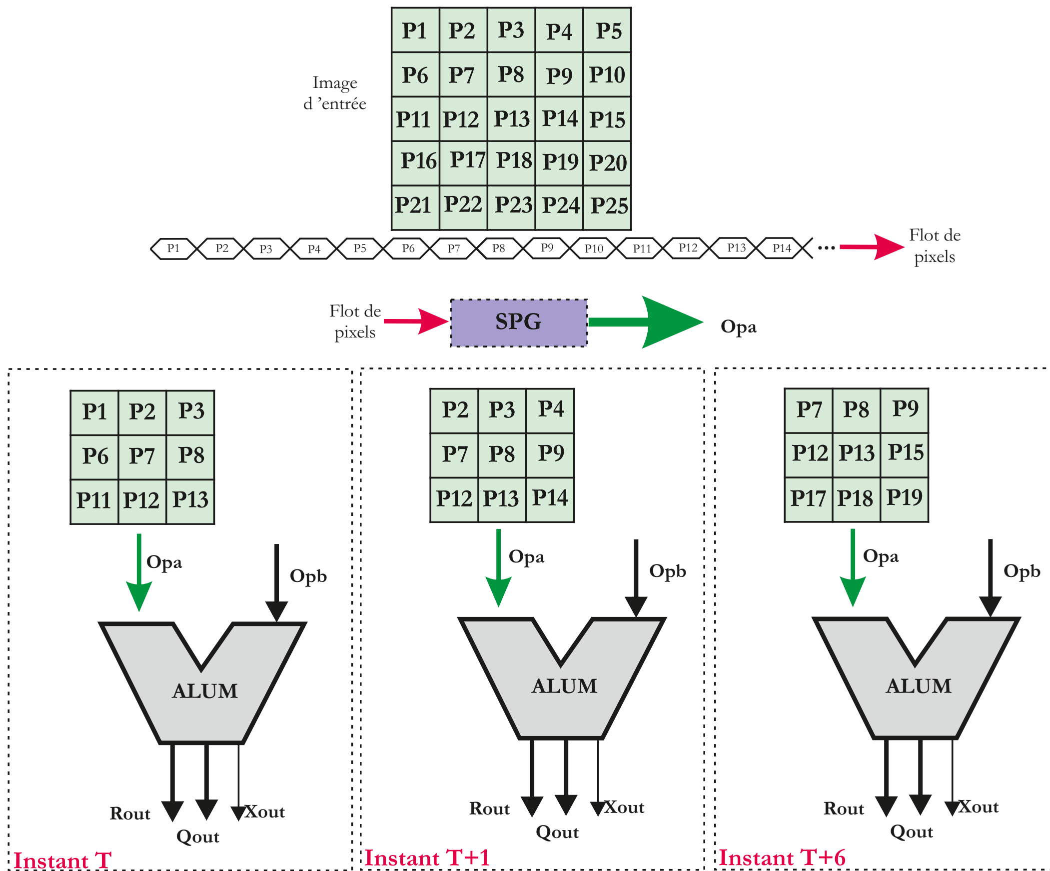


FIGURE 4.8 – Principe de mise en œuvre d'un SPG

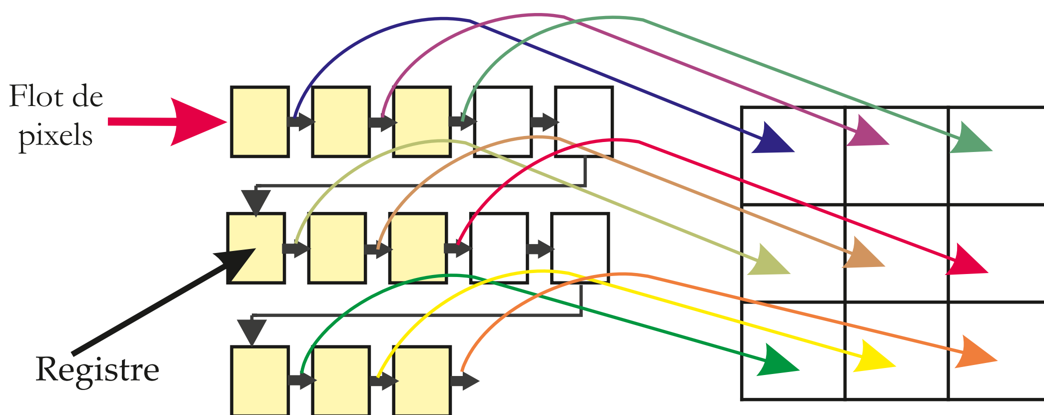


FIGURE 4.9 – Principe de mise en œuvre d'un SPG

Le nombre de registres nécessaires est dépendant de la largeur de l'image

traitée mais également de la dimension des opérandes. Il est donné par la formule suivante :

$$N_{reg} = (n-1) \times L + n \tag{4.12}$$

Où L représente la largeur de l'image traitée et n la dimension d'un côté de l'opérande.

Néanmoins, il est important de noter que chaque ALUM peut avoir une taille de fenêtre d'intérêt différente et que la largeur de l'image d'entrée peut varier au cours de l'application (par exemple, si l'on a ciblé une zone d'intérêt sur une image, il est appréciable de pouvoir focaliser un traitement uniquement sur cette zone). Pour répondre à ce problème, deux approches ont été étudiées :

- un première approche consiste à simplement créer un **SPG** pour chaque taille de fenêtre d'intérêt et pour chaque largeur d'image d'entrée. Mais, il est évident que le volume de ressources matérielles instancié et inutilisé au cours de l'application est alors important,
- la seconde approche consiste à créer un **SPG** pour chaque taille de fenêtre d'intérêt et de lui permettre de s'adapter à des changements de largeur d'image.

Le principe de la seconde méthode repose sur l'idée qu'un SPG prévu pour alimenter un opérande à partir d'une image de largeur M contient les informations pouvant alimenter un opérande, de même dimension, depuis une largeur d'image N, avec $M > N$. Afin d'illustrer le principe d'un **SPG** selon cette approche prenons l'exemple d'une ALUMs de dimension d'entrée 3×3. Cette ALUM travaille, dans un premier temps, sur une largeur d'image de 7 pixels puis dans un second temps sur une image de largeur de 5 pixels. La Fig. 4.10 représente le réseau de registres du **SPG** correspondant qui permet de fournir les données à l'ALUM pour les 2 largeurs d'image.

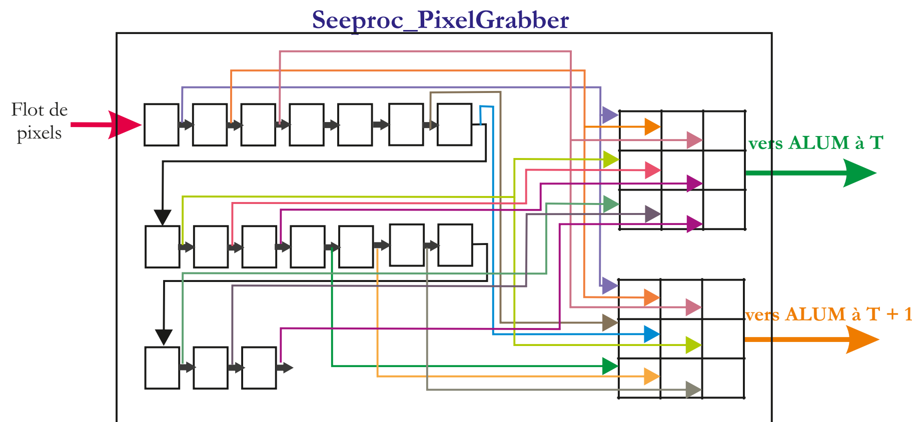


FIGURE 4.10 – Schéma synoptique du Seeproc_PixelGrabber pour une dimension de 3 et une largeur d'image de 7 pixels à un instant T puis de 5 pixels à un instant T+1

D'un point de vue matériel le SPG, est construit de la manière suivante : on crée un registre élémentaire, on le reproduit un certain nombre de fois, suivant l'Eq. 4.12, en fonction de la dimension de l'opérande à alimenter et la largeur d'image maximale puis on crée les connections entre ces divers registres. Le changement d'une configuration de réseau à une autre est activé par l'intermédiaire d'une bus de commande : **Contrôle largeur** sur la Fig. 4.2.

En pratique, on dote chaque composant SPG d'une entrée supplémentaire indiquant la largeur de l'image traitée. Cette entrée est ensuite contrôlée par la partie SeeProc_Decoder.

L'approche file de registres est particulièrement intéressante d'un point de vue temporel, car après une latence pour le remplissage du pipeline, une nouvelle fenêtre d'intérêt sera disponible à chaque cycle d'horloge. Ainsi le système s'adapte parfaitement à la vitesse de la source de données. Un autre avantage de cette méthode est que cette file de registres peut être synthétisée comme des éléments de mémoire interne du FPGA et, de ce fait, ne consommera que peu d'éléments logiques.

Néanmoins, si cette méthode semble être idéale pour des images du type S-VGA (800×600)ou XVGA(1024×768) avec des tailles de fenêtres d'intérêt réduites (ne dépassant pas 21×21 avec un FPGA Stratix I), au delà elle devient difficile à mettre en œuvre pour plusieures raisons. La première est que la consommation de mémoire interne devient très importante et peut fortement limiter le spectre d'applications susceptible d'intervenir à la suite de SeeProc. À titre d'exemple, pour une image S-VGA et une taille de fenêtre d'intérêt de

17×17 un seul SPG consomme 115.353 bits mémoire (soit 13% de la mémoire interne d'un FPGA Stratix I EP1S10). La seconde raison est liée au routage de ces registres. En effet, le grand nombre d'interconnexions diminue les performances de l'unité de traitement. Les lignes physiques de connexion se voient être de plus en plus longues ce qui augmente inévitable le temps de propagation des données. Néanmoins, avec l'évolution et les capacités des FPGAs actuels, cette approche registres reste viable.

4.2.2.4 Dimensionnement des bus d'interconnexion

Le programmeur peut affecter une taille de fenêtre d'intérêt différente à chaque ALUM. Il a donc été nécessaire de prévoir des *adaptateurs* afin de permettre la communication entre ces ALUMs. Trois cas possibles ont été identifiés :

1. une sortie Xout, donc de dimension 1, doit alimenter une entrée de dimension N,
2. une sortie Rout/Qout de dimension n doit alimenter une entrée de dimension N avec N > n,
3. une sortie Rout/Qout de dimension N doit alimenter une entrée de dimension n avec N > n.

Le premier cas peut être géré de manière analogue à l'accès pixel décrit dans la section précédente. Dans ce cas, c'est la sortie Xout de l'ALUM qui fournit le flot de pixels en entrée du SPG.

Le second cas survient lorsque la dimension de la source est inférieure, et différente de 1, de la dimension de la destination. On prend, par exemple, la sortie Rout d'une l'ALUM, qui est de dimension 3×3, pour alimenter l'opérande A d'une autre ALUM2, qui est de dimension 7×7. Dans ce cas on va effectuer une opération de remplissage (*padding* en anglais) avec des '0' logiques aux pixels non connus. La Fig. 4.11 illustre ce procédé.

Enfin, le dernier cas advient lorsque la dimension de la source est plus grande que la dimension de la destination. Dans ce cas de figure, on sélectionne le *cœur* de la source permettant de remplir la destination, en somme on effectue une troncature de la source. Une illustration de la sélection du cœur pour une source de dimension 5 et une destination de dimension 3 est donnée par la Fig. 4.12.

Toutes les combinaisons possibles sont supportées par insertion automatique d'opérateur de remplissage ou de troncature. La méthodologie de développement permet ensuite de ne générer que les éléments utiles au bon fonctionnement de l'application afin de minimiser les ressources matérielles.

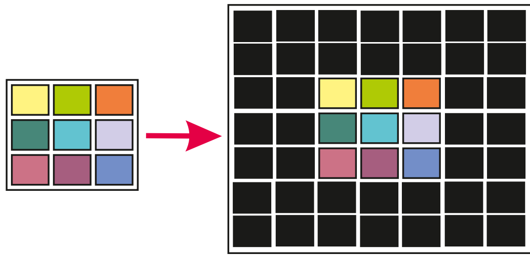


FIGURE 4.11 – Exemple de concaténation pour l'adaptation des dimensions de bus

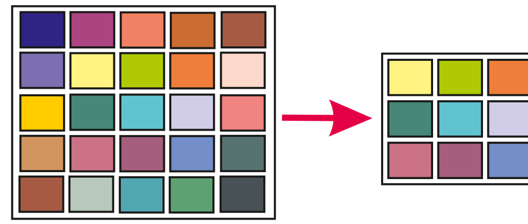


FIGURE 4.12 – Exemple de sélection du cœur pour l'adaptation des dimensions de bus

4.2.3 Conclusion

En conclusion, le chemin de données présent au sein du processeur SeeProc propose un modèle générique, basé sur une factorisation matérielle, d'unité de traitement dédiée au traitement d'image bas niveau. Afin d'utiliser pleinement les possibilités offertes par la combinaison de plusieures de ces unités, un crossbar associé à un réseau de registres en série assure toutes les communications et les dimensionnements des bus nécessaires pour le bon fonctionnement du chemin de données. Il est maintenant nécessaire de s'intéresser à la partie de contrôle permettant la configuration et reconfiguration de ce chemin de données.

4.3 Architecture de la partie contrôle

La partie contrôle, **Seeproc_Decodeur** sur la Fig. 4.1, est le cœur du processeur. Cette partie est basée sur une architecture de type RISC [28] avec des spécificités permettant le contrôle et la configuration du chemin de données présenté précédemment. Classiquement l'unité de contrôle est chargé du décodage et de l'exécution des instructions contenues dans la mémoire programme. Elle est également en charge du contrôle du programme à travers le compteur programme. Dans notre cas, elle doit en plus être capable de fournir au chemin de données les différents paramètres, codes opérations et paramètres aux ALUMs. Elle doit également générer le mot de contrôle du crossbar et gérer la configuration des SGPs. Enfin, elle est également responsable de la gestion de l'intégration du co-processeur SeeProc au sein d'un système complet et de son interfaçage avec d'autres éléments.

La partie contrôle décode et exécute des instructions composées (Tab. 4.1) d'un opcode de 8 bits, et de deux opérandes de taille égale à 32 bits. Pour ce faire elle est composée de cinq éléments, détaillés par la suite, selon la Fig. 4.13 :

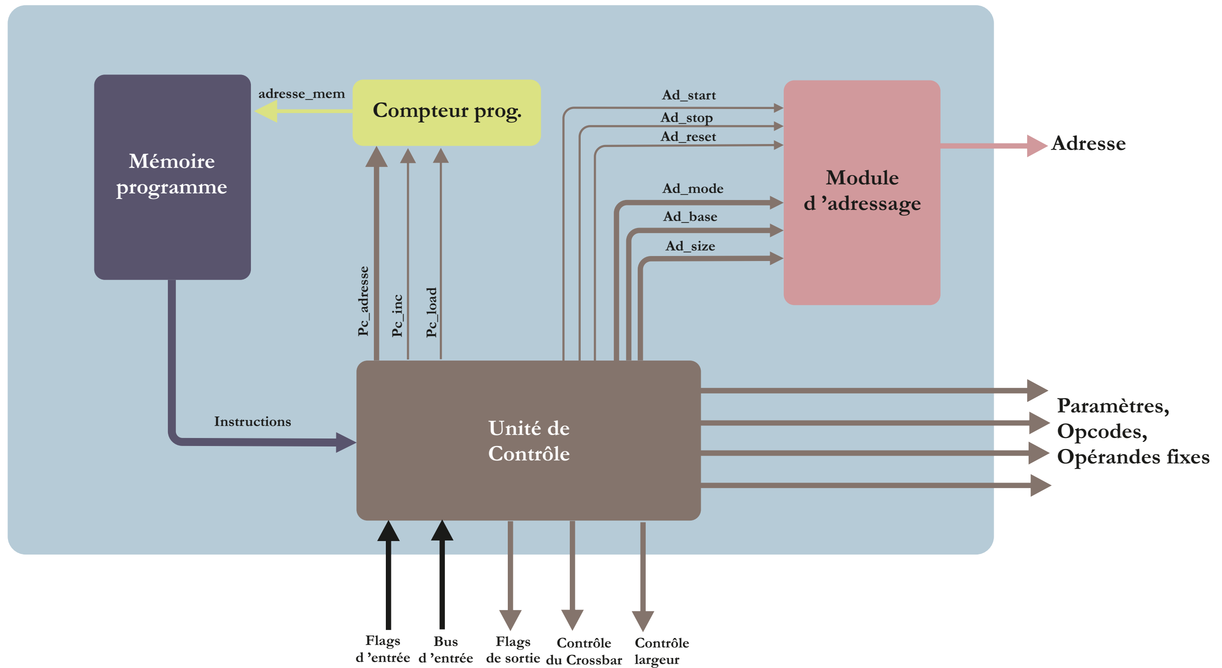


FIGURE 4.13 – Schéma synoptique de la partie de contrôle et d'ordonnancement du *Seeproc*

4.3.1 L'unité de contrôle

L'unité de contrôle (UC) est le cœur de la partie contrôle. L'UC est composée d'une machine à états finis permettant d'implanter les trois cycles classiques que sont le *fetch*, *decode* et *execute*, et ce pour chacune des instructions :

1. 1er cycle → fetch : chargement du registre d'instruction et incrément du compteur programme,
2. 2nd cycle → decode : décodage et définition du chemin de données selon l'opcode,
3. 3^{ieme} cycle → execute : exécution de l'instruction (mise à jour des flags, des opcodes, du mot de contrôle du crossbar ...etc).

L'UC est chargée de la génération du mot de contrôle pour le crossbar, de la mise à jour des flags de sortie, de la réception des flags d'entrée, de la mise à jour du compteur programme ou encore du paramètrage de l'unité d'adressage.

De plus, l'UC aura pour rôle, le contrôle et la mise à jour des codes opérations, des paramètres et des opérandes de toutes les ALUMs. L'UC aura ainsi pour sortie les trois opcodes, le paramètre FM et les trois paramètres de sélection de

gamme des résultats pour chaque ALUM (présentés en Sec. 4.2). Ces sorties seront ensuite directement reliées aux différentes entrées des ALUMs. Dans le cas où le programmeur aurait décidé de fixer un opérande, l'UC intégrera une sortie supplémentaire pouvant fournir les valeurs définies dans le programme assembleur. La Fig. 4.14 présente l'UC dans le cas de où 2 ALUMs sont utilisées dans le chemin de données et l'une d'elle, ALUM1 sur la figure, dispose d'une opérande fixe (typiquement un masque de convolution).

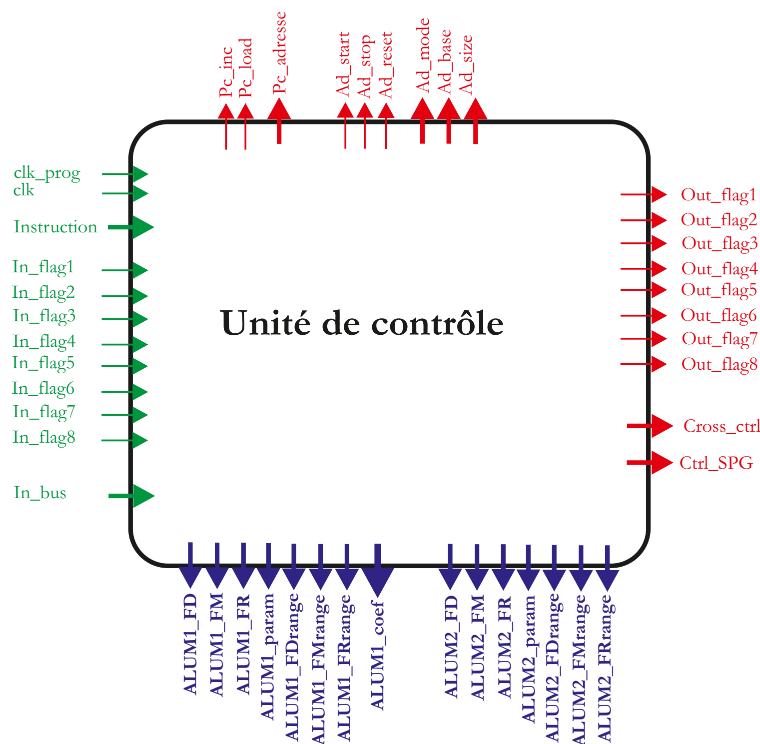


FIGURE 4.14 – Exemple de schéma synoptique l'unité de contrôle

Lors du premier cycle de chaque instruction, l'unité de contrôle est chargée avec le contenu de la mémoire programme indiqué par l'adresse stockée dans le compteur programme. Ce dernier est ensuite incrémenté, c'est l'étape de *fetch*. Puis les étapes de *decode* et *execute* sont réalisées. Les étapes de *fetch* et *decode* sont exécutées en un cycle d'horloge. L'étape *execute* nécessite un nombre de cycle d'horloge dépendant de l'instruction.

4.3.1.1 Gestion des horloges

L'UC dispose de deux entrées d'horloge. La première entrée (clk_prog sur la Fig. 4.14) est trois fois plus rapide que la seconde entrée qui correspond à l'horloge

d'acquisition des pixels. Cette première horloge permet de charger l'instruction contenue en mémoire programme en un seul cycle d'horloge "pixel". En effet, la mémoire programme est de type RAM synchrone. De ce fait 3 cycles d'horloge sont nécessaires à l'obtention de l'instruction. Ces 3 cycles sont l'envoie de l'adressage mémoire, le décodage de cette adresse puis la réception de la donnée contenue à l'adresse envoyée.

Idéalement, trois horloges permettraient une optimisation du processeur. La première horloge (clk_pix) correspondrait à la cadence d'acquisition des pixels. La seconde (clk_fde), trois fois plus rapide que clk_pix, permettrait d'effectuer les 3 cycles de *fetch*, *decode* et *execute* en un seul cycle de clk_pix (dans le cas où l'*execute* ne nécessiterait qu'un seul cycle). Enfin la troisième (clk_prog), trois fois plus rapide que clk_fde, permettrait d'effectuer le chargement de l'instruction depuis la mémoire programme en un seul cycle de clk_fde. Néanmoins, cette solution n'a pas été mise en place pour des raisons de limitation technologique des FPGAs. Par exemple, si la fréquence d'acquisition des pixels est de 60 MHz, alors on aurait clk_fde à 180 Mhz et clk_prog à 540 MHz. Or la fréquence maximale permise, avec un FPGA Stratix I EP1S60, est d'environ 390 MHz. C'est pour cette raison que seulement 2 horloges sont utilisées dans le processeur SeeProc.

4.3.2 Jeu d'instructions de SeeProc

Le jeu d'instructions permet, de manière simple, à un utilisateur de décrire l'application qu'il souhaite exécuter. Il peut ainsi configurer les opérations que les différents éléments de calcul devront réaliser, le réseau de communication entre ces éléments ainsi que l'ordre dans lequel les opérations devront être exécutées. L'ensemble des instructions décrivant une application est organisé de façon séquentielle au sein d'un programme. Comme dit précédemment, une instruction est composée d'un opcode d'un octet suivi de deux opérandes de quatre octets.

Les instructions disponibles sont au nombre de onze et sont décrites, dans les sections suivantes, par catégorie, selon leur fonction.

4.3.2.1 Instructions de traitement

Pour créer une application, l'utilisateur dispose de plusieurs instructions dédiées à la configuration des ALUMs, à l'interconnexion entre ces derniers et avec les données d'entrée, ou encore à la largeur de l'image traitée. Ces instructions sont listées dans le Tab. 4.6. Ajouté à ces instructions, l'entête du programme devra spécifier la taille de la fenêtre d'intérêt de chaque élément de calcul. Pour ce faire, l'utilisateur rajoutera la directive **ALUMn_MATRIX_SIZE XX**, où n représente le numéro du ALUM concerné et XX la valeur en nombre de pixels

d'un côté de la fenêtre d'intérêt (qui, pour rappel, est nécessairement carrée).

Instruction		Opérandes				
Mnémonique	code machine	Opérande 1	Opérande 2			
LOAD	0x01	Alum id.	FD_opocde	FM_opcode	FR_opcode	FM_param
COEF	0x06	Alum id.	Liste des coefficients de l'opérande fixe			
ROUT	0x05	Source	Destination			
CFG	0x07	Paramètre	valeur			

TABLE 4.6 – Instructions de traitement

L'instruction **LOAD** permet de paramètrer les différents opcodes et paramètres de l'ALUM désignée par l'opérande 1. L'opérande 2 spécifie les 3 opcodes et le paramètre de la seconde unité (voir Sec. 4.2.1) . Le Tab. 4.7 détaille les opcodes affectés aux différentes opérations. Ainsi, par exemple l'instruction LOAD ALUM2 1 1 2 56 configurera l'ALUM2 avec l'opcode 0 pour FD, 1 pour FM, 2 pour FR et 56 pour FM_param. L'opérande 2 est ainsi partitionné en 4 octets, chaque octet correspondant à un opcode/paramètre. Le Tab. 4.8 détaille la correspondance entre la ligne de code assembleur et le code machine obtenu.

Opcode	Fonction FD	Fonction FM	Fonction FR
0	Rout = opA	Qout = Rout	Xout = Qout(pixel central)
1	Rout = opA + opB	Qout = -Rout	Xout = $\sum$Qout(n,n)
2	Rout = opA - opB	Qout = $\|Rout\|$	Xout = MAX(Qout(n,n))
3	Rout = opA * opB	Qout = $Rout^2$	Xout = MIN(Qout(n,n))
4	Rout = opA AND opB	Qout = SL [7](Rout,FM_param)	Xout = AND(Qout(n,n))
5	Rout = opA OR opB	Qout = SR [8](Rout,FM_param)	Xout = OR(Qout(n,n))
6	Rout = opA XOR opB	Qout = seuil(Rout,FM_param)	Xout = XOR(Qout(n,n))

TABLE 4.7 – Opcodes des opérations réalisables par **FD**, **FM** et **FR**.

Opcode	Opérande 1	Opérande 2			
7 0	31 0	31 24	23 16	15 8	7 0
0x01	ALUM visée	FD_opocde	FM_opcode	FR_opcode	FM_param

TABLE 4.8 – Format de l'instruction LOAD

LOAD	ALUM3	MULT	INV	SUM	0
0x01	0x00000003	0x03	0x01	0x00	0x00

Table 4.9 – Exemple de l'instruction de traitement : LOAD

Le code machine hexadécimal complet correspondant à la ligne assembleur du Tab. 4.9 est donc *010000000303010000*. L'intérêt de ce découpage est de permettre à l'UC, lors de l'étape d'*execute*, de configurer simultanément et en un seul cycle d'horloge tous les signaux de contrôle d'une ALUM. Le partie du code VHDL de l'UC permettant l'exécution de l'instruction LOAD est le suivant :

```
CASE opcode is
    ...
    when conv_std_logic_vector(1,8) =>                    -- Instruction LOAD
        CASE op1 IS
            when conv_std_logic_vector(1 , 8) =>          -- Op1 designe ALUM1
                ALUM1_FD        <= op2(31 downto 24);
                ALUM1_FM        <= op2(23 downto 16);
                ALUM1_FR        <= op2(15 downto  8);
                ALUM1_FMPARAM   <= op2(7  downto  0);
            when conv_std_logic_vector(2 , 8) =>          -- Op1 designe ALUM2
                ALUM2_FD        <= op2(31 downto 24);
                ALUM2_FM        <= op2(23 downto 16);
                ALUM2_FR        <= op2(15 downto  8);
                ALUM2_FMPARAM   <= op2(7  downto  0);
            ...
        end CASE;
    ...
end CASE;
```

L'instruction **COEF** permet de définir la valeur d'un opérande de l'ALUM indiquée par l'opérande 1. Ainsi, on peut aisément définir tout type de masques à appliquer à un flux d'images ou encore un motif de recherche. Par exemple, l'instruction COEF ALUM3 1 1 1 1 -8 1 1 1 1 permet d'appliquer à une opérande de l'ALUM3 un masque de la forme suivante :

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Le format de l'instruction COEF est dépendante de la dimension des opérandes d'entrée de l'ALUM désignée par l'opérande 1. En effet, le nombre de coefficients à appliquer est de $n \times n$ où n représente la dimension d'un côté de la matrice d'intérêt. Le programme ci-dessous donne deux exemples d'utilisation de l'instruction COEF.

Exemple de l'utilisation de l'instruction COEF
ALUM1_MATRIX_SIZE 3 ALUM2_MATRIX_SIZE 5 ... ... // Autres instructions ... ... COEF ALUM1 1 1 1 1 -8 1 1 1 1 COEF ALUM2 1 1 1 1 1 2 2 2 2 2 0 0 0 0 0 -2 -2 -2 -2 -2 -1 -1 -1 -1 -1

Il est évident que tous les coefficients ne peuvent être inclus dans une seule et même instruction machine. Ainsi, pour chaque coefficient, une instruction, exécutée en un cycle d'horloge, en code machine est créée. Ces instructions répondent au format suivant :

Opcode	Opérande 1		Opérande 2
7 0	31 24	23 0	31 0
0x06	Numéro du coefficient	ALUM visée	valeur du coefficient

Table 4.10 – Format de l'instruction COEF

Correspondance entre code assembleur et code machine de l'instruction COEF	
COEF ALUM2 1 1 1 1 -8 1 1 1 1 →	06 01000002 00000001 06 02000002 00000001 06 03000002 00000001 06 04000002 00000001 06 05000002 FFFFFFF8 06 06000002 00000001 06 07000002 00000001 06 08000002 00000001 06 09000002 00000001

La partie de code de l'UC correspondante à la gestion de l'instruction COEF est la suivante :

```
CASE opcode is
    ...
    when conv_std_logic_vector(6,8) => -- Instruction COEF
        CASE op1(3 downto 0) is -- Numero de l ALUM visee
            when conv_std_logic_vector(1,4) => -- Detection de l ALUM1
                ind1 := conv_integer(op1(31 downto 24)); -- Numero du ↩
                    coefficient vise
                COEF_ALUM1(ind1*9-1 downto ind1*9-9) <= op2(8 downto 0); -- ↩
                    Affectation de la valeur au coefficient
            when conv_std_logic_vector(2,4) => -- Detection de l ALUM2
                ind1 := conv_integer(op1(31 downto 24)); -- Numero du ↩
                    coefficient vise
                COEF_ALUM1(ind1*9-1 downto ind1*9-9) <= op2(8 downto 0); -- ↩
                    Affectation de la valeur au coefficient
            ...
        end CASE;
    end CASE;
    ...
end CASE;
```

L'instruction **ROUT** permet de définir le chemin de données en contrôlant les différentes entrées/sorties du crossbar. Ainsi, l'instruction ROUT ALUM1 ALUM2_OPA connectera la sortie de l'ALUM1 à l'opérande A de l'ALUM2. Le format de l'instruction est le suivant :

Opcode	Opérande 1	Opérande 2
7 0	31 0	31 0
0x05	Num. port d'entrée du crossbar (08x)	Num. port de sortie du crossbar (08x)

Table 4.11 – Format de l'instruction ROUT

Le mot de contrôle est, dans la version actuelle du SeeProc, de largeur 72 bits. Il est partitionné en 14 ensembles de 5 bits[9], comme le rappelle le Tab. 4.5. Le contenu d'un ensemble indique le port de sortie du crossbar auquel est connecté l'entrée correspondante à l'ensemble. Par exemple si les 5 MSBs, représentant la sortie Xout_Data sur le schéma en Fig. 4.1, du mot de contrôle du crossbar valent *"10011"* alors la sortie X de l'ALUM3 est interconnectée avec la sortir Xout_Data. Un tableau de correspondance des ports d'entrées et de sorties est donné en Annexe. C.

Pour finir avec les instructions dédiées à la partie opérative, l'utilisateur dispose également d'une instruction de configuration **CFG**. Cette instruction admet en opérande 1 le paramètre à configurer et en opérande 2 la valeur à affecter à ce paramètre. Dans la version actuelle du SeeProc, deux paramètres sont configurables. Le premier paramètre, **IMW**, indique la largeur de l'image traitée. Ainsi,

9. De ce fait les 2 MSB ne sont pas utilisés

l'instruction CFG IMW 512 permet de spécifier que l'image d'entrée est de largeur 512 pixels. Les éventuels changements de ce paramètre sont gérés matériellement grâce au bus *contrôle largeur* présent sur la Fig. 4.13 qui permet de paramètrer les différents SPGs. L'intérêt d'une telle instruction est qu'elle permet de modifier en temps réel la largeur de l'image traitée. Le format en code machine de l'opérande 2 ne représente pas la valeur décimale écrite dans le code assembleur. Elle représente le numéro de changement de largeur d'image dans le programme assembleur. Par exemple, si dans un programme deux instructions CFG IMW 125 et CFG IMW 200 sont successivement présentes, alors l'opérande 2 en code machine de la première sera 0x00000000 et la seconde 0x00000001. Grâce aux outils de développement présentés au chapitre suivant, le SPG sait que lorsqu'il aura son bus d'entrée **Contrôle largeur** égale à 1, alors la largeur de l'image à traiter est de 200. L'intérêt d'une telle conversion est que la taille du bus **Contrôle largeur** aurait été bien plus grande s'il avait fallu indiquer directement la largeur de l'image. Dans la version actuelle du SeeProc, la dimension du bus **Contrôle largeur** est de 4 bits, permettant ainsi 16 changements de largeur d'image possible au sein d'une application.

Opcode	Opérande 1	Opérande 2
7 0	31 0	31 0
0x07	0x00000000	Numéro du changement

Table 4.12 – Exemple de l'instruction de traitement : CFG IMW

Le second paramètre modifiable grâce à l'instruction CFG permet de sélectionner les 9 bits, parmi les 18 disponibles, d'un résultat intermédiaire d'une ALUM qui seront transmis à l'unité suivante. Le format de l'instruction CFG avec ce paramètre, **RANGE**, est donnée en Tab. 4.13. Le programmeur spécifie l'ALUM visée, puis le nombre de décalage vers la droite (SR) pour FD, FM et FR. Afin de conserver le signe de résultat, le MSB est conservé et est concaténé avec les 8 bits indiqués par l'utilisateur. Par exemple l'instruction CFG RANGE ALUM1 4 2 0, aura pour effet de sélectionner le bit 17 concaténé avec les bits 13 à 6 [10] du résultat de FD , le bit 17 concaténé avec les bits 15 à 8 pour FM et 17 à 9 pour FR. En cas d'absence de l'instruction CFG RANGE, les 9 bits sélectionnés sont par défaut les 9 MSBs, soit les bits 17 à 9.

10. Le MSB est le bit 17 et le LSB le bit 0

Opcode	Opérande 1	Opérande 2			
7 0	31 0	31 24	23 16	15 8	7 0
0x07	0x00000001	ALUM visé	SR pour FD	SR pour FM	SR pour FR

Table 4.13 – Exemple de l'instruction de traitement : CFG RANGE

4.3.2.2 Instructions pour l'interfaçage avec les éléments périphériques

Dans le but d'interfacer SeePROC avec des éléments périphériques, 4 instructions ont été prévues. Comme dit précédemment, SeeProc est un co-processeur de traitement. Il doit donc pouvoir s'intégrer aisément au sein d'une architecture complète de type caméra intelligente. Ces instructions permettent la mise en œuvre de protocoles d'interfaçage, tel que le *handshaking*. Les instructions d'interfaçage sont listées dans le Tab. 4.14 ci-dessous :

Mnémonique (code machie)		
SET (0x02)	Flag_out ID	
RESET (0x03)	Flag_out ID	
WAIT (0x00)	Flag_in ID	Valeur
ADDR (0x09)	Paramètre	Valeur

Table 4.14 – Instructions d'interfaçage du processeur SeeProc

L'instruction **WAIT** permet de vérifier si un flag, spécifié en opérande 1, est égal à la valeur indiquée par l'opérande 2. Un flag correspond à un signal sur 1 ou plusieurs bits. Dans le cas où la valeur du flag est différente de l'opérande 2, alors le programme est mis en pause jusqu'à obtention de cette dernière.

Les instructions **SET** et **RESET** sont utilisées pour modifier la valeur du flag de sortie indiqué en opérande 1. L'instruction SET permet d'affecter un niveau logique haut au flag désigné par l'opérande 1. De même manière, l'instruction RESET affectera un niveau logique bas.

Dans la version actuelle, le processeur SeeProc dispose de 8 flags d'entrée (7 correspondent à des signaux de 1 bit et 1 correspondant à un signal codé sur 32 bits) et 8 flags de sortie correspondant à des signaux de 1 bit. Grâce aux outils de développement, l'utilisateur peut aisément changer le nom attribué (*alias*) à chacun de ces flags afin de facilité l'écriture du programme assembleur. À titre d'exemple, l'Annexe C liste, entre autre, les alias attribués à certains flags utilisés par la suite.

	Opcode	Opérande 1	Opérande 2
	7 0	31 0	31 0
WAIT	0x00	Id du flag ou du bus testé	Valeur attendue
SET	0x02	Id du flag manipulé	0x00000000
RESET	0x03	Id du flag manipulé	0x00000000

TABLE 4.15 – Format des instructions WAIT, SET et RESET

Afin d'illustrer l'intérêt de telles instructions, prenons l'exemple suivant : Le SeeProc est autorisé, par un élément périphérique quelqu'il soit, à fonctionner uniquement lorsque le flag d'entrée, nommé *STARTPROC*, passe à un niveau logique haut. SeeProc doit ensuite indiquer à cet élément, via un flag de sortie, qu'il est en phase de traitement. La gestion de ce protocole se fera de la manière suivante :

```
WAIT STARTPROC 1 // Attente de la requete de l'element peripherique
SET OPERATING // Requete recue et debut du traitement
    ...
    ... // Autres instructions
    ...
    ...
RESET OPERATING  // Fin du traitement
JUMP 12 // Retour a la ligne "WAIT STARTPROC 1"
```

La dernière instruction, **ADDR**, permet d'utiliser un générateur d'adresse intégré dans le module d'adressage de la Fig. 4.13. L'intérêt d'une telle instruction est de donner une certaine autonomie à SeeProc vis-à-vis de la lecture et du stockage de données en mémoire. SeeProc peut ainsi aller lire des données stockées dans une mémoire, les traiter puis sauvegarder le résultat en mémoire. Pour ce faire, l'instruction ADDR admet plusieurs paramètres, énumérés dans le Tab. 4.16 :

Paramètre	**Fonction**	
START	Démarre le compteur	
STOP	Met en pause le compteur	
RES	Remet le compteur à l'adresse de base	
BASE	Valeur	Définit l'adresse de début
SIZE	Valeur	Définit le nombre d'adresse à générer
MODE	Valeur	Définit le mode d'adressage

TABLE 4.16 – Liste des différents paramètre de l'instruction ADDR

Les deux opérandes BASE et SIZE permettent de définir la plage d'adresses à générer. Par exemple si l'on a ADDR BASE 20 et ADDR SIZE 1280 alors la plage d'adresses sera de 20 à 1300, 1300 correspondant à la somme des valeurs de BASE et SIZE. Le paramètre MODE permet de spécifier un mode d'adressage particulier. Dans sa version actuelle SeeProc permet d'effectuer un adressage normal et sous-échantillonné. L'incrément du sous-échantillonnage peut être modifié à tout moment de l'application. Les Tab. 4.17 et Tab. 4.17 donnent le format de l'instruction ADDR.

Opcode		Opérande 1		Opérande 2	
7	0	31	0	31	0
0x09		Paramètre traité		0x00000000	

TABLE 4.17 – Format de l'instruction ADDR pour les paramètres START, STOP et RES

Opcode		Opérande 1		Opérande 2	
7	0	31	0	31	0
0x09		Paramètre traité		Valeur	

TABLE 4.18 – Format de l'instruction ADDR pour les paramètres BASE, SIZE et MODE

Il est à noter que la gestion du signal de sélection Écriture/Lecture d'une mémoire n'est pas pris en compte par l'instruction ADDR. Néanmoins, il suffit de contrôler un des différents flag disponibles avec les instructions SET et RESET. Le programme ci-dessous montre un exemple de l'utilisation des différentes instructions d'adressage. Dans ce programme, lorsque le flag *STARTPROC* sera égal à 1, le processeur déclenchera un compteur allant de 1 à 307200 par pas de 1.

```
// Chargement des parametres
ADDR BASE 1
ADDR SIZE 307200 // 640*480
ADDR MODE NORMAL
     ...
     ... // Autres instructions
     ...
     ...
WAIT STARTPROC 1
ADDR START
WAIT STOPPROC 1
ADDR STOP
ADDR RESET
```

4.3.2.3 Instructions de contrôle du programme

Le contrôle du programme est assuré par 3 instructions présentées dans le Tab. 4.19.

Mnémonique (code machine)			
JUMP (0x04)	numéro de ligne		
TEMPO (0x08)	Valeur		
IFRES (0x0A)	Conditions	Valeur	numéro de ligne

Table 4.19 – Instructions de contrôle du programme

Les sauts inconditionnels sont possibles grâce à l'instruction **JUMP**. Ils peuvent être utilisés afin de réaliser une boucle infinie. L'instruction JUMP admet un opérande indiquant le numéro de la ligne du programme assembleur à laquelle il souhaite effectuer son branchement. L'outil assembleur (Chap. 5) fera automatique la conversion entre la ligne de code assembleur et l'adresse mémoire correspondante.

Opcode	Opérande 1	Opérande 2
7 0	31 0	31 0
0x04	Numéro de ligne	0x00000000

Table 4.20 – Format de l'instruction JUMP

Les sauts conditionnels sont réalisés grâce à l'instruction **IFRES**. Cette instruction est suivie d'un code de condition permettant de tester plusieurs paramètres sur le bus d'entrée de l'UC. Si la condition est vérifiée alors le branchement est réalisé sinon le programme se poursuit. Cette condition est une comparaison avec le bus d'entrée de l'unité de contrôle. Le Tab. 4.21 liste les différentes conditions disponibles pour réaliser un branchement conditionnel.

Condition (code machine)	**Branchement si**	
BPOS (0x00000000)	Valeur du bus d'entrée positive	
BNEG (0x00000001)	Valeur du bus d'entrée négative	
BZERO (0x00000002)	Valeur du bus d'entrée nulle	
SUP (0x00000003)	Valeur	Données du bus d'entrée supérieure à valeur
INF (0x00000004)	Valeur	Données du bus d'entrée inférieure à valeur
EQU (0x00000005)	Valeur	Données du bus d'entrée égale à valeur

Table 4.21 – Liste des différentes conditions de l'instruction IFRES

Opcode		Opérande 1		Opérande 2	
7	0	31	0	31	0
0x0A		Condition testée		Numéro de ligne	

TABLE 4.22 – Format de l'instruction IFRES pour les conditions BPOS, BNEG et BZERO

Opcode		Opérande 1				Opérande 2	
7	0	31	24	23	0	31	0
0x0A		Numéro de ligne		Condition testée		Valeur	

TABLE 4.23 – Format de l'instruction IFRES pour les conditions SUP, INF et EQU

L'instruction **TEMPO** permet de mettre en pause le programme pendant le nombre de cycles d'horloge indiqué en opérande 1. Cette instruction permet, par exemple, à l'utilisateur de changer la configuration d'une ALUM au bout de 100 images traitées sans avoir besoin d'activer un flag. La partie VHDL du code de l'UC gérant l'instruction TEMPO est donnée ci-dessous et son format dans le Tab. 4.24 :

```
CASE opcode is
    ...
    WHEN conv_std_logic_vector(8, 8) => -- Instruction TEMPO
        ind := conv_integer(op1); -- Conversion en entier de la valeur ↩
            contenue dans l'operande1 / ind est initialise a 1
        if i < ind then -- Tant que i inf a ind
            i <= i+1; -- on incremente i
            state <= "11"; -- on reste dans l'etape d'execute
        else
            i <= 1; --
        end if;
    ...
    end CASE;
    ...
end CASE;
```

Opcode		Opérande 1		Opérande 2	
7	0	31	0	31	0
0x08		Nombre de cycles d'horloge		0x00000000	

TABLE 4.24 – Format de l'instruction TEMPO

4.3.3 Conclusion

La partie contrôle du processeur SeeProc a été conçue pour exploiter au maximum les capacités du chemin de données. Pour ce faire, des instructions dédiées à ce dernier ont été intégrées à un jeu d'instruction classique des processeurs de type RISC. On peut notamment configurer les ALUMS, modifier les interconnexions entre les éléments du chemin de données de manière dynamique ou encore adapter la partie opérative à des changements de largeur de l'image d'entrée en temps réel.

4.4 Conclusion

Ce chapitre a permis de détailler l'architecture complète du processeur à chemin de données reconfigurable SeeProc. Parmi les originalités de ce processeur, on peut noter les éléments de traitement, les ALUMs, dont la granularité est en adéquation avec les traitements bas niveau de l'image. Afin de contrôler ces éléments, une partie contrôle personnalisée a également été mise en œuvre. Cette partie, articulée autour d'un cœur de processeur RISC, dispose d'un jeu d'instruction permettant, en plus du contrôle classique du programme, la gestion des reconfiguration des éléments de traitement ainsi que l'agencement de ces unités.

Il reste désormais à rendre la programmation du processeur SeeProc accessible à des programmeurs non initiés aux langages HDL. Il est également nécessaire d'évaluer si des possibilités de minimisation architecturale en fonction de l'application développée sont envisageables. Ce travail est présenté dans le chapitre suivant intitulé Méthodologie de développement sur le processeur SeeProc.

Chapitre 5

Outils de développement pour le processeur Seeproc

Le chapitre précédent décrit l'architecture matérielle du processeur SeeProc. Un des objectifs fixés en introduction est de rendre sa programmation intuitive pour des programmeurs non familiers des architectures et des langages de développement matériels. Un autre aspect important de nos travaux est l'optimisation des ressources matérielles. Dans cette optique un ensemble d'outils ont été développés ayant pour but de :

- faciliter la programmation,
- donner accès aux programmeurs aux possibilités de personnalisation du processeur,
- minimiser, en termes de ressources matérielles, les éléments VHDLs générés,
- proposer un environnement de simulation afin de faciliter le prototypage d'applications.

Le flot de développement complet d'une application sur le processeur Seeproc est présenté en Fig. 5.1. Il est scindé en 3 parties qui sont traitées dans les sections suivantes. La première partie permet de réaliser la conversion du code assembleur en code machine. La seconde partie permet de générer une description optimisée - en VHDL - de l'architecture du processeur. Enfin, la troisième partie, propose à l'utilisateur un environnement de simulation lui permettant la mise au point de son application.

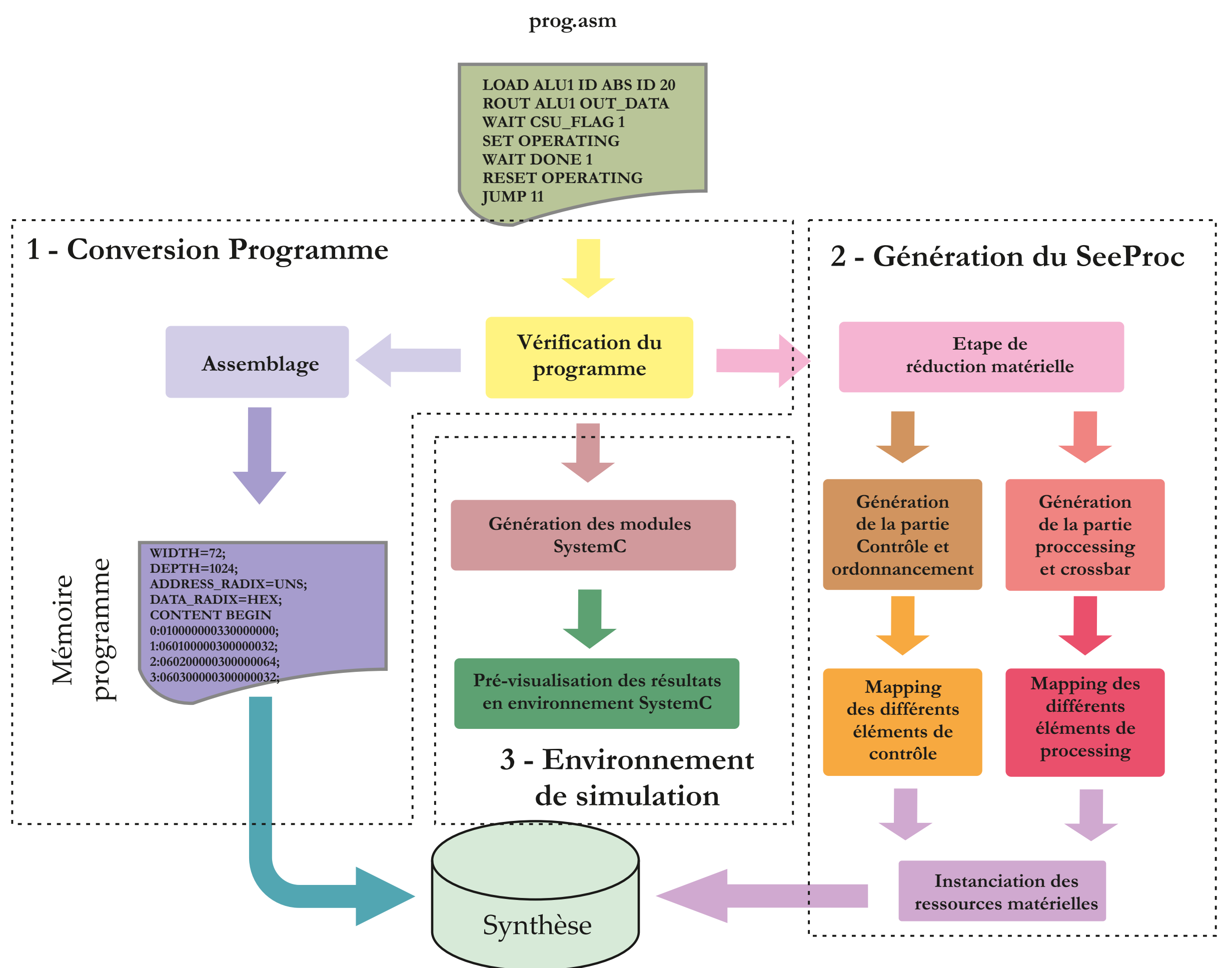


FIGURE 5.1 – Flot de développement du *Seeproc*

5.1 Conversion du programme assembleur

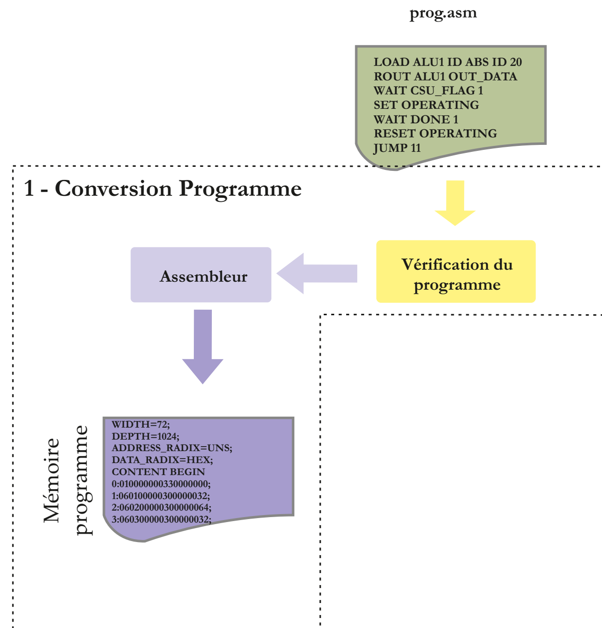


FIGURE 5.2 – Conversion du programme assembleur

Présenté dans la Sec. 4.3.2, et rappelé dans le Tab. 5.1 [1], le jeu d'instructions est composé de 11 instructions au total. Un assembleur (Fig. 5.2) dédié est en charge de la conversion des lignes textuelles du programme en code machine, chaque instruction assembleur correspondant à une instruction en code machine (hormis pour l'instruction COEF). La mémoire programme du processeur SeeProc est initialisée avec ce code machine. Le décodage des instructions est ensuite pris en charge par l'unité de contrôle. L'outil assembleur est décrit en langage C à l'aide des outils d'analyse lexicale et grammaticale Lex et Yacc [94] présentés en Annexe A. Il supporte certaines fonctionnalités permettant d'alléger la programmation et d'améliorer la lisibilité du programme. La Fig. 5.2 présente les éléments logiciels utilisés pour cette partie de la méthodologie.

Outre la traduction des instructions en code machine, l'assembleur se charge de la vérification de la syntaxe du programme et de la gestion des différents *alias* qui permettent de manipuler symboliquement (via un nom et non une adresse) les différents flags, registres ou encore codes opérations de l'architecture. Une liste des *alias* est donnée en Annexe C.

1. les indications du type (08x) indiquent le nombre de bits héxadécimaux de codage

Mnémonique	Opcode	Opérande 1	Opérande 2
WAIT	0x00	Id du flag ou du bus testé (08x)	Valeur attendue (08x)
LOAD	0x01	ALUM visée (08x)	FD (02x) - FM (02x) - FR (02x) - FMparam (02x)
SET	0x02	Id du flag manipulé (08x)	0x00000000
RESET	0x03	Id du flag manipulé (08x)	0x00000000
JUMP	0x04	Num. de ligne (08x)	0x00000000
ROUT	0x05	Num. port d'entrée du crossbar (08x)	Num. port de sortie du crossbar (08x)
COEF	0x06	Num. du coef.(02x) - ALUM visée (06x)	valeur du coefficient (08x)
CFG	0x07	Paramètre traité (08x)	Valeur (08x)
TEMPO	0x08	Nmb de cycles d'horloge (08x)	0x00000000
ADDR	0x09	Paramètre traité (08x)	0x00000000 Valeur (08x)
IFRES	0x0A	Condition testée (08x) Numéro de ligne (02x) - Condition testée (06x)	Numéro de ligne (08x) Valeur (08x)

Table 5.1 – Format des instructions du SeeProc

Au niveau de la syntaxe du programme assembleur, outre les instructions formant le programme proprement dit, le programme doit nécessairement comporter un entête (*header*). Cet entête doit indiquer le nombre d'ALUM utilisées dans le programme et les dimensions de ces dernières. Enfin, pour signifier à l'outil assembleur de début des instructions, une ligne de commentaire finissant par le mot *instructions* est placée avant le début des instructions. Un exemple d'entête est donné dans le programme ci-dessous :

```
// Chargement des parametres
ALUM_NUM 2
ALUM1_MATRIX_SIZE 3
ALUM1_MATRIX_SIZE 5

// Debut des instructions
...
... // instructions
...
```

L'entête est analysé par l'outil assembleur afin d'initialiser 2 paramètres, NALU indiquant le nombre d'ALUM présent dans l'application et DIM spécifiant la dimension de chacune d'entre elles, utilisées dans les autres outils du flot de développement afin notamment de gérer la généricité de l'architecture.

L'outil assembleur permet, à partir du programme assembleur, de créer un fichier au format MIF, permettant d'instancier la mémoire programme du processeur SeeProc. Le format MIF est un format compatible avec les composants Altera. Néanmoins, il est très proche du format COE correspondant aux composants Xilinx. Ainsi, il est rapide de modifier le fichier mif.c afin d'obtenir la syntaxe voulue. L'Annexe. D présente un comparatif syntaxique des deux for-

mats[2].

Le cas de l'instruction COEF est particulier. Lorsque cette instruction est détectée, le programme stocke le numéro de ligne correspondant ainsi que la dimension de l'ALUM visée. La raison de ce traitement est due au fait qu'une seule ligne assembleur de l'instruction COEF correspond à plusieures lignes d'instruction en code machine. Il est en effet nécessaire de connaître le nombre de ligne en code machine supplémentaires introduites par les éventuelles instructions COEF afin de faire correspondre de manière automatique les numéros de ligne des instructions JUMP et IFRES. Prenons l'exemple du programme assembleur suivant :

```
// Chargement des parametres
ALUM_NUM 1
ALUM1_MATRIX_SIZE 3

// Debut des instructions
0 LOAD ALUM1 MULT ID SUM 0
1 COEF ALUM1 1 1 0 1 0 −1 0 −1 −1
2 CFG RANGE ALUM1 4 0 2
3 ROUT IN_DATA_1 ALUM1_OPA
4 ROUT IN_DATA_2 ALUM1_OPB
5 ROUT ALUM1_XOUT X_OUT_DATA
6 CFG IMW 512
7 WAIT STARTPROC 1
8 CFG IMW 256
9 SET OPERATING
10 WAIT STOPPROC 1
11 RESET OPERATING
12 JUMP 7
```

Sans la gestion du nombre de lignes supplémentaires introduit par l'instruction COEF, le fichier mif serait le suivant :

2. Un ajout simple permettant le passage automatique d'un format à l'autre serait l'ajout d'un paramètre dans l'entête afin de spécifier la cible matérielle (TARGET Altera par exemple).

```
WIDTH=72; DEPTH=1024;

ADDRESS_RADIX=UNS; DATA_RADIX=HEX;

CONTENT BEGIN
    0   :   010000000130100000;
    1   :   060100000100000001;
    2   :   060200000100000001;
    3   :   060300000100000000;
    4   :   060400000100000001;
    5   :   060500000100000000;
    6   :   0606000001ffffffff;
    7   :   060700000100000000;
    8   :   0608000001ffffffff;
    9   :   0609000001ffffffff;
    10  :   070000000101040002;
    11  :   050000000100000002;
    12  :   050000000200000003;
    13  :   050000000d00000000;
    14  :   070000000000000000;
    15  :   000000000100000001;
    16  :   070000000000000001;
    17  :   020000000000000000;
    18  :   000000000200000001;
    19  :   030000000000000000;
    20  :   040000000700000000;
    [21..1023]  :   ffffffffffffffffff;
END;
```

On peut voir que l'adresse 7 du fichier mif ne correspond pas à la ligne 7 du programme assembleur. Cela introduit dans le cas de ce programme une erreur au niveau de la largeur de l'image d'entrée. Il est donc nécessaire de gérer une conversion correcte du numéro des lignes. Dans l'exemple ci-dessus, l'outil assembleur détecte une instruction COEF ayant pour cible une ALUM de dimension 3×3. Il faut donc ajouter 8 lignes au code mif pour obtenir le bon branchement (ligne 15).

Une autre instruction est convertie de manière spécifique. Il s'agit de l'instruction CFG suivie de IMW. En effet, comme précédemment expliqué, en Sec. 4.3.2.1, la valeur correspondant à la largeur de l'image n'est pas présente dans le code machine. A chaque fois qu'une instruction CFG IMW xx est détectée et que la valeur (xx) n'a pas été utilisée dans les instructions précédentes, un compteur est incrémenté et fait office de second opérande en code machine. Lors de l'étape suivante, la génération du SeeProc, le fichier responsable de la création des SPGs prend en compte le nombre de largeur d'image différente et génère les SPGs en fonction.

5.2 Génération de l'architecture du processeur SeeProc

La seconde partie du flot de conception a pour rôle la génération des éléments constituant l'architecture matérielle du processeur SeeProc. Cette partie est illustrée en Fig. 5.3. Elle est décomposée en 3 phases. La première phase, à travers l'extraction d'informations depuis le programme assembleur, permet de minimiser les ressources matérielles nécessaires à l'exécution de ce dernier. La seconde phase est la génération à proprement parler - en VHDL - des éléments de SeeProc en se basant sur les informations extraites lors de la première phase. Enfin la troisième phase consiste en l'instanciation matérielle des éléments obtenus.

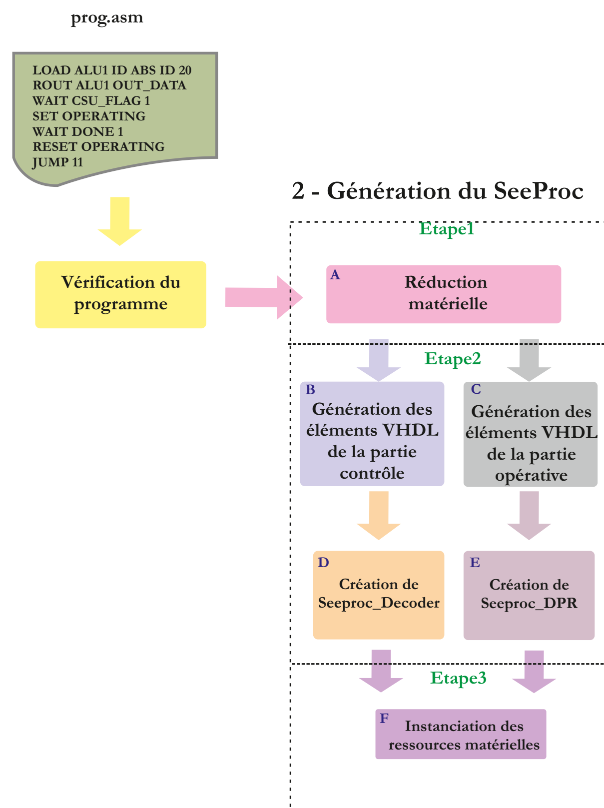


FIGURE 5.3 – Génération de l'architecture du processeur SeeProc

5.2.1 Étape d'extraction d'information et de réduction matérielle

La première phase de la génération de l'architecture du processeur SeeProc consiste en l'extraction de caractéristiques du fichier assembleur. Cette phase

vise à réduire l'ensemble des ressources matérielles requises. Pour cela, deux programmes (MiniUC.c et MiniDPR.c) spécialisés parcourent le programme assembleur et extraient des informations nécessaires à la minimisation de chaque élément de l'architecture finale. Les informations, et leur variable associée, extraites sont :

- Le nombre d'ALUM utilisées Nalu,
- la dimension des opérandes de chaque ALUM Dim,
- les opérations que chaque ALUM effectue au cours de l'application MinAlum,
- la présence ou non d'opérande fixe pour chaque ALUM Coef,
- la relation entres les différents ports d'entrée/sortie du crossbar MinCross,
- l'utilisation ou non d'instruction d'adressage MinAddr,
- la présence d'un ou plusieurs changements de la largeur d'image traitée Imw,
- Le nombre d'instructions présent dans le programme assembleur NumInst.

Une variable supplémentaire est utilisée par l'ensemble des fichiers de génération VHDL. Cette variable (prefix) affecte un préfixe au nom de tous les éléments VHDL générés. Cette variable est entrée par l'utilisateur lors de la compilation de son programme assembleur. L'intérêt d'une telle instruction est que, dans l'hypothèse où le programmeur décide de générer 2 SeeProc différents et de les utiliser simultanément, ce préfixe est indispensable afin d'éviter des erreurs de compilation à cause de fichier VHDL ayant le même nom mais pas le même contenu.

Les informations extraites sont ensuite distribuées dans les différents programmes responsables de la génération (en VHDL) des divers éléments de l'architecture du processeur SeeProc. La Fig. 5.4 illustre la distribution des informations suivant la partie de SeeProc à générer.

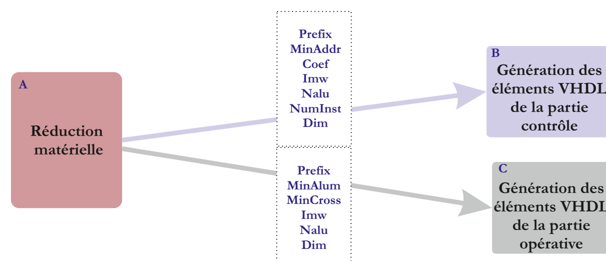


Figure 5.4 – Distribution des informations pour la génération de l'architecture du processeur SeeProc

5.2.2 Étape de génération des éléments VHDL du SeeProc

L'étape précédente a permis d'extraire un certain nombre de caractéristiques en fonction du code assembleur à compiler. L'étape suivante est l'étape de génération du code VHDL décrivant l'architecture du processeur SeeProc. Dans un premier temps, les composants de base des deux entités que sont le **SeeProc_DPR** et le **SeeProc_Decoder** sont générés de façon minimale en termes de ressources matérielles (phases B et C sur la Fig. 5.3) en fonction des caractéristiques extraites lors de la première étape. Ces éléments sont ensuite instanciés et routés afin de former les deux entités du SeeProc (phases D et E sur la Fig. 5.3).

5.2.2.1 Génération des composants élémentaires de l'entité SeeProc_Decoder (Phase B)

Chaque élément VHDL est paramétré par un ensemble de caractéristiques spécifiques. Le Tab. 5.2 liste les programmes, le composant VHDL qu'ils créent et les paramètres utilisés.

Programme C	Élément créé	Paramètres
Mem_prog	Mémoire programme	prefix - NumInst
Compteur	Module d'adressage	prefix - MinAddr
CU	Unité de contrôle	prefix - Nalu - Dim - Coef - Imw - MinAddr
PC	Compteur programme	prefix

Table 5.2 – Génération des fichiers VHDL de la partie **SeeProc_Decoder** du processeur SeeProc

La création du composant matériel **Mémoire programme** est dépendante, en plus du prefix, du nombre de lignes du code machine généré (variable NumInst. Ceci permet de spécifier la profondeur minimale et suffisante de la mémoire programme nécessaire à l'exécution du programme machine.

Le module d'adressage est créé si et seulement si la variable MinAddr indique la présence d'instructions d'adressage dans le programme assembleur. Si aucune de ces instructions n'est détectée alors le module d'adressage ne sera pas créé.

La génération de l'unité de contrôle est faite en fonction de plusieurs caractéristiques. Les paramètres Nalu, Dim et Coef indique au programme le nombre de sorties, permettant le contrôle des différentes ALUM, sont nécessaires. De

manière analogue au module d'adressage, si la variable MinAddr indique que le programme contient des instructions d'adressage, alors les sorties de l'unité de contrôle permettant de gérer ce module sont créées. Il en est de même avec la variable Imw qui autorise ou non la création du bus de contrôle *Contrôleur largeur* présent sur la Fig. 4.1.

Enfin, le composant **Compteur programme** ne dépend que de la variable prefix. L'Annexe E donne les algorithmes des programmes présentés ci-dessus et la Fig. 5.5 donne un schéma synoptique de cette étape.

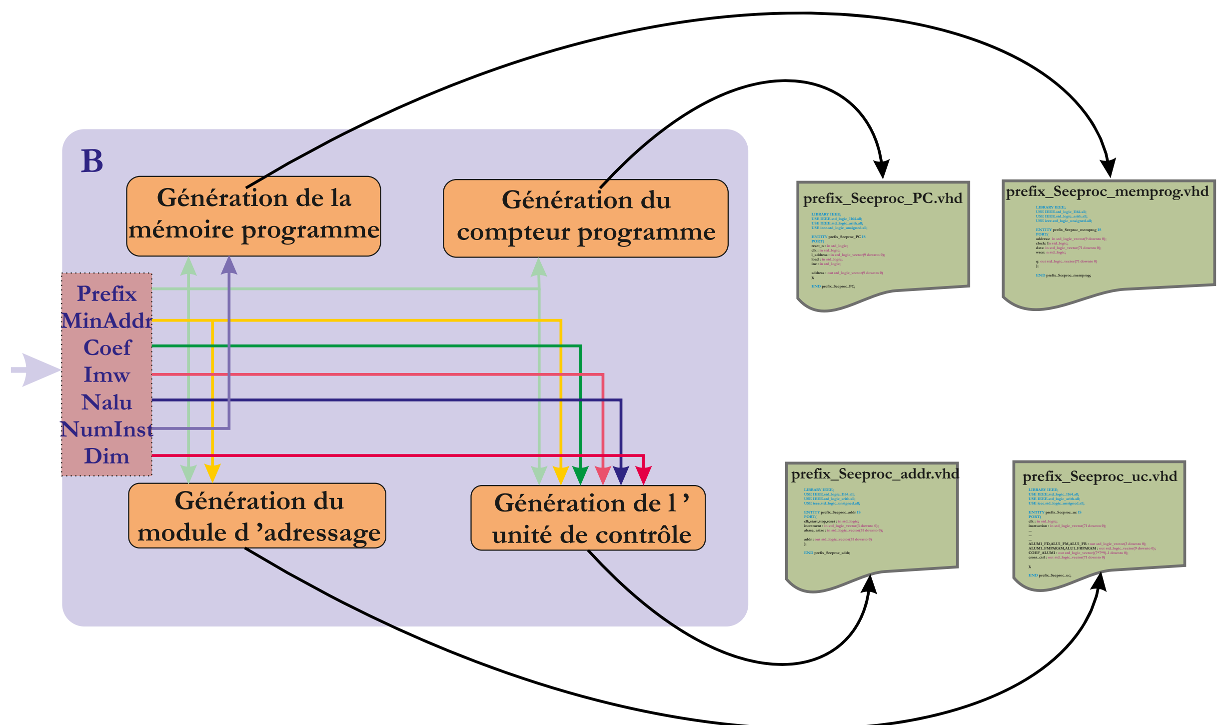


FIGURE 5.5 – Schéma synoptique de la génération des éléments de base de SeeProc_Decoder

5.2.2.2 Génération des composants élémentaires de l'entité SeeProc_DPR (Phase C)

La partie **SeeProc_DPR** de l'architecture du processeur SeeProc est construite à partir de 4 programmes listés dans le Tab. 5.3.

Programme C	**Élément VHDL créé**	**Paramètres**
ALUM	ALUM	prefix - Nalu - Dim - Minalum
SPG	SeeProc PixelGrabber	prefix - Dim - Imw
Crossbar	Crossbar	prefix - Nalu - Dim - Imw - Mincross
reg	Registre	prefix

TABLE 5.3 – Génération des fichiers VHDL de la partie **SeeProc_DPR** du processeur SeeProc

Le premier programme, ALUM, gère la création de la ou des ALUMs nécessaires au fonctionnement du programme assembleur. Le nombre d'ALUMs à créer est défini par le paramètre Nalu. La dimension des opérandes des ALUMs est contenue dans le paramètre Dim. Enfin, la liste des opérations réalisées par ALUMs est indiquée par le paramètre MinALUM. Ce dernier paramètre permet de ne pas créer inutilement des opérations qui ne sont jamais utilisées au cours de l'application. On minimise ainsi les ressources matérielles requises par ALUM.

Le programme SPG est en charge de la création des différents **SeeProc_PixelGrabber**. La génération est dépendante des paramètres Dim et Imw (en plus du prefix). L'algorithme utilisé est décrit en Annexe E.

La création du Crossbar est réalisée, en plus du prefix, en fonction des paramètres :

- Nalu : permet de définir le nombre d'entrées/sorties du crossbar,
- Dim : permet de caractériser les différentes entrées/sorties du crossbar,
- Imw : permet de gérer les modifications de dimensions entre les différentes connections,
- MinCross : permet de ne créer que les connections appliquées dans le programme assembleur.

Le programme ci-dessous donne l'algorithme du programme en charge de la création du composant Crossbar.

```
Debut

    I − Ouvrir/creer un fichier nomme "prefix_seeproc_crossbar.vhd".

    II − Extraire la dimension maximale, dimmax, depuis le param\`{e}tre dim

    III − Ecrire les ports d entree/sortie du crossbar en fonction des
    param\`{e}tres dim et nalu.

    IV − Appeler les composants SPGs en fonction de la param\`{e}tre dim.

    V − Tester l existance de connections depuis les ALUM'n'_XOUT_Data
    vers OUT_DATA pour gerer le dimensionnement de 1 vers dimmax.
        Si existance alors
        | Creation de la connection en fonction de imw

    VI − Tester l existance de connections depuis les ALUM'n'_ROUT et
    ALUM'n'_QOUT vers OUT_DATA pour gerer le dimensionnement de dim[n]
    vers dimmax.
        Si existance alors
        | Creation de la connection  en fonction de imw

    VII − Tester l existance de connections depuis les ALUM'n'_Xout vers
    ALUM'm'_OPA et  ALUM'm'_OPB pour gerer le dimensionnement de 1 vers
    dim[m].
        Si existance alors
        | Creation de la connection  en fonction de imw

    VIII − Tester l existance de connections depuis les ALUM'n'_Rout et
    ALUM'n'_Qout vers ALUM'm'_OPA et ALUM'm'_OPB pour gerer le
    dimensionnement de dim[n] vers dim[m].
        Si existance alors
        | Creation de la connection  en fonction de imw

    IX − Tester les autres possibilites de connections (celles dont les
    dimensions entre entree et sortie sont les meme)
        Si existance alors
        | Creation de la connection independamment de imw

Fin
```

TABLE 5.4 – Algorithme de création du Crossbar

Enfin, le programme Reg génère le composant **Registre**, qui est l'élément de base pour le composant SPG. Il ne dépend que du paramètre prefix. Les algorithmes associés à chaque programme sont donnés en Annexe. E. La Fig. 5.6 présente le schéma synoptique de cette phase.

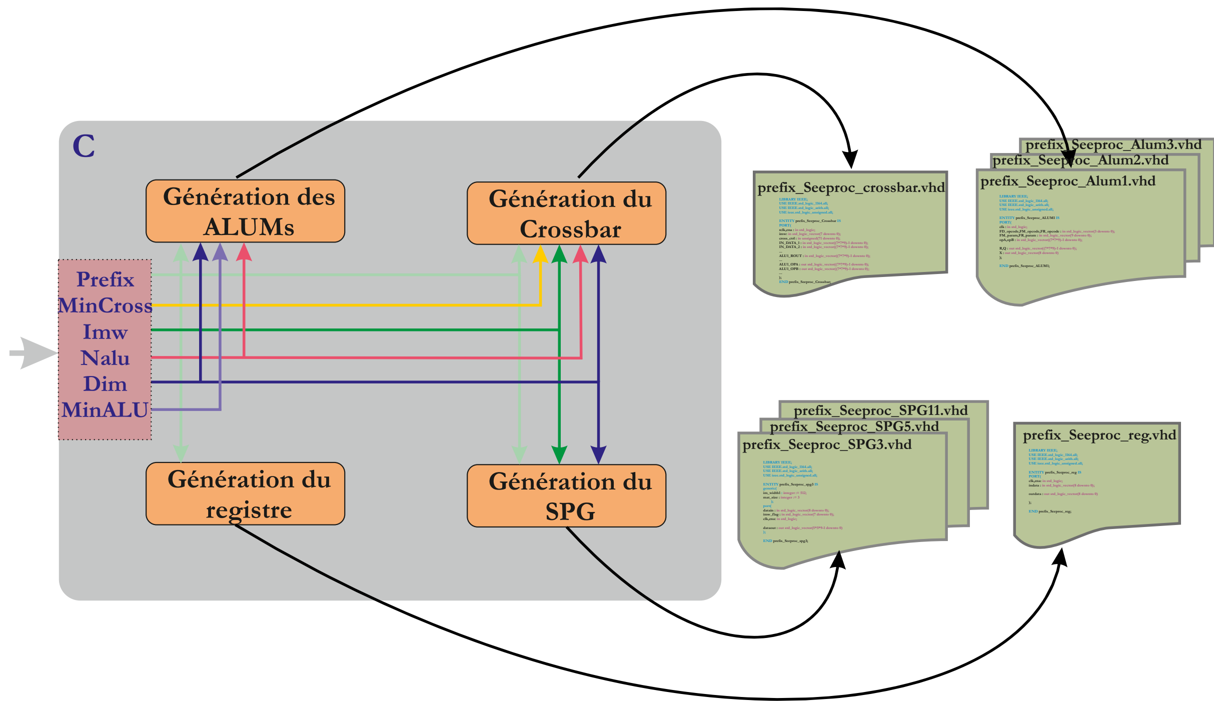


FIGURE 5.6 – Schéma synoptique de la génération des éléments de base de SeeProc_DPR

5.2.2.3 Génération des composants des unités principales SeeProc_Decoder et SeeProc_DPR (Phase D et E)

Les dernières phases (D et E sur la Fig. 5.3) de la génération des éléments matériels de l'architecture du processeur SeeProc sont la création du composant SeeProc_Decoder (phase D) et du composant SeeProc_DPR (phase E). Dans les 2 cas, un programme va router les éléments d'architecture de base générés dans les phases précédentes afin d'obtenir un élément de hiérarchie supérieure. L'intérêt de ces générations est de simplifier l'instanciation matérielle qui vient en dernière étape. Il est en effet plus simple pour un utilisateur de router 2 éléments (en l'occurrence SeeProc_Decoder et SeeProc_DPR) que d'en router 8. Les 2 programmes responsables de ces générations prennent ainsi en caractéristique d'entrée, les caractéristiques d'entrée de l'ensemble des éléments à intégrer dans le composant de hiérarchie supérieure. La Fig. 5.7 expose les schémas synoptiques de ces 2 programmes.

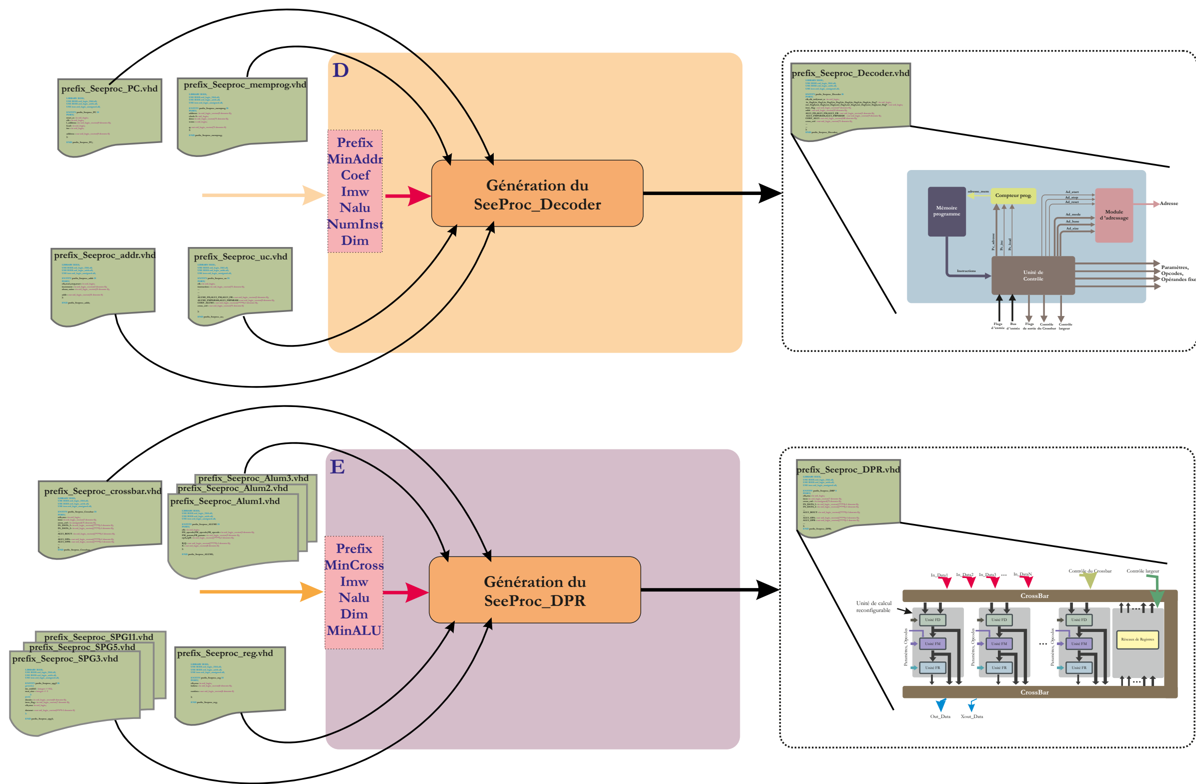


FIGURE 5.7 – Schéma synoptique de la génération des éléments SeeProc_DPR et SeeProc_Decoder

5.2.3 Étape d'instanciation matérielle

Une fois les éléments VHDL générés, l'étape d'instanciation matérielle va clore le flot de développement du SeeProc. Cette étape n'est pas automatique et est, de ce fait, à la charge du programmeur. Néanmoins elle reste relativement simple à réaliser. En fonction du type de FGPA sur lequel le SeeProc doit être implanté, l'utilisateur exécute l'environnement de développement associé. À partir des deux éléments VHDL de plus haute hiérarchie obtenus lors de l'étape précédente, à savoir la partie **SeeProc_Decoder** et **SeeProc_DPR** de SeeProc, l'utilisateur peut soit créer les symboles correspondant pour ensuite les connecter graphiquement (*mapping*), soit créer un fichier vhdl et effectuer un routage textuel qui reste nettement moins simple que la première approche.

Une solution plus simple consiste, à partir des composants **SeeProc_Decoder** et **SeeProc_DPR**, de générer un seul élément de hiérarchie maximale afin d'éviter au concepteur de devoir réaliser un routage à la main. Dans sa version actuelle, le processeur SeeProc n'intègre pas cette solution. Ceci se justifie par le fait que le processeur SeeProc est un support de recherche et qu'il est appréciable d'avoir

la possibilité d'effectuer des tests sur les signaux transitant d'un composant à l'autre de façon simple.

Dans le cadre de nos travaux, nous avons implanté l'architecture du processeur SeeProc sur cible FPGA de marque ALTERA. L'environnement de développement associé est Quartus II [119]. La Fig. 5.8 illustre cette étape.

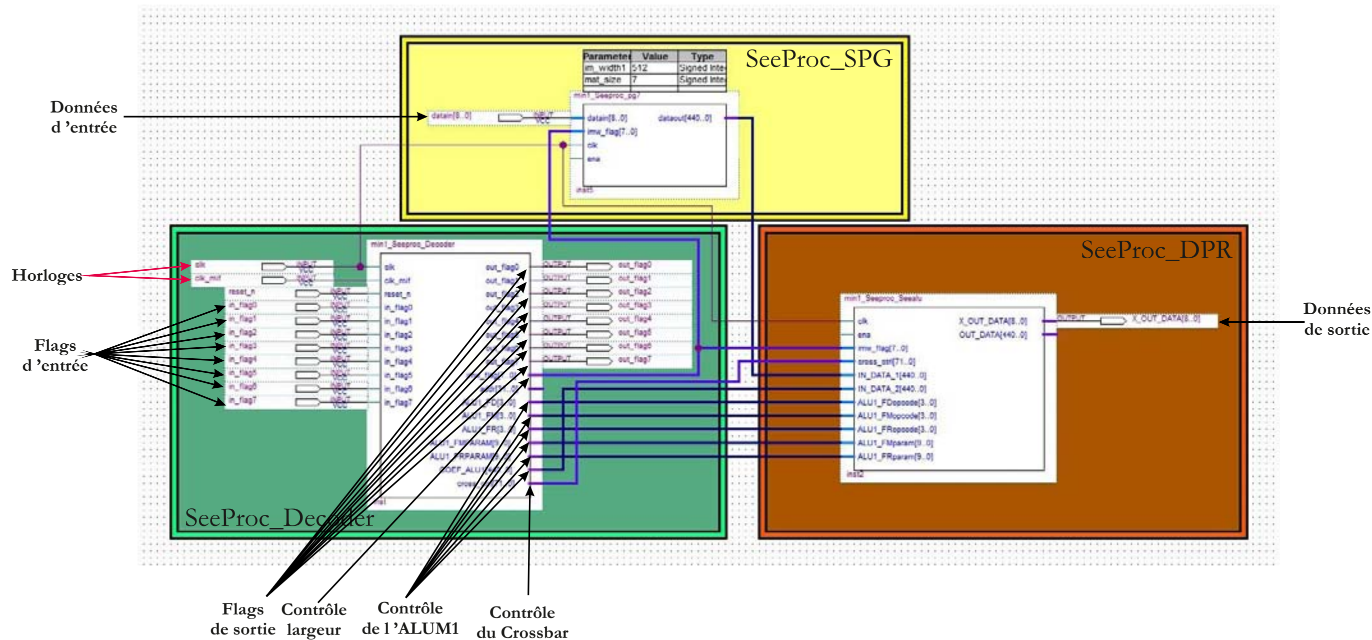


FIGURE 5.8 – Exemple d'instanciation matérielle du SeeProc l'environnement de développement Quartus II

La Fig. 5.8 ne représente que les éléments de SeeProc. Pour rappel, SeeProc est un co-processeur, il est donc nécessaire de connecter les différents IOs de SeeProc à un processeur maître permettant l'utilisation de SeeProc.

5.3 Environnement de simulation

La dernière partie du flot de développement sur le processeur SeeProc consiste en un environnement de simulation utilisant un langage de modélisation au niveau système(*SLDL* [3]).

Le but de cette étape est de permettre à l'utilisateur d'évaluer le résultat de son programme assembleur sans avoir à générer et à implanter l'architecture

3. SLDL pour Sytem Level Design Language

correspondante sur un FPGA cible. Il peut ainsi plus rapidement effectuer des modifications afin d'ajuster son programme assembleur en fonction de l'application qu'il souhaite exécuter.

Les langages SLDL permettent *la modélisation de systèmes numériques matériels et logiciels à l'aide d'un seul et unique langage. Ils permettent ainsi de modéliser des systèmes matériels, logiciels ou encore mixtes. L'objectif de la modélisation d'un système est d'aboutir à sa simulation afin de vérifier l'exactitude du comportement attendu*[85].

Deux langages ont été envisagés dans le cadre de nos travaux : SpecC[112] et SystemC[93]. Le langage SpecC est un sur-ensemble du langage C. Il intègre en plus des fonctionnalités lui permettant de modéliser des systèmes matériels. Néanmoins, ce langage n'a eu que peu de succès ce qui justifie le fait que SystemC ait été privilégié. Le langage SystemC est pour sa part fondé sur le langage C++[113]. Un programme SystemC est donc un programme C++. SystemC est implanté sous forme d'une bibliothèque définissant un ensemble de classe C++ permettant de modéliser un système de manière hiérarchisée en séparant les aspects fonctionnels et les aspects de communication. SystemC intègre également un noyau de simulation et des outils de traçage. SystemC a également l'avantage d'être *open-source*. Une présentation plus approfondie de SystemC est disponible en Annexe F.

L'architecture de SeeProc a ainsi été modélisée en langage SystemC. En pratique, un programme spécialisé parcourt le fichier assembleur et génère les divers composants en SystemC. Cette approche est analogue à celle de la création des fichier VHDL. Le programme admet en paramètre les paramètres nalu, imw et dim vues précédemment. Ces paramètres permettent la création des ALUMs avec les dimensions d'opérandes spécifiées dans le fichier assembleur, les SPGs selon les largeurs d'image et les dimensions des ALUMs. Les minimisations matérielles ne sont pas prises en compte dans le modèle SystemC puisqu'il n'y a pas de limitation en termes ressources matérielles au niveau de ce modèle. Le contenu de la mémoire programme est simulé par sous programme du modèle SystemC.

Le SeeProc étant un co-processeur, les instructions d'interfaçage (WAIT, SET, RESET et ADDR) sont simulées et générées via la console de commande. Ainsi, lors d'une instruction SET, RESET ou ADDR, les changements de valeur des signaux affectés par ces instructions sont transmis à l'utilisateur par l'intermédiaire de *cout* dans la console. L'instruction WAIT demande à l'utilisateur, par l'intermédiaire de *cin*, la valeur du signal désigné par l'opérande 1 de l'instruction WAIT.

L'option de réaliser un *testbench* peut également être envisagée mais dans ce

cas deux charges sont imposées au concepteur. La première réside dans la création à proprement parler du *testbench*. La seconde est que les changements, inhérents aux modifications de la valeur des flags, s'opèrent de manière figée selon l'ordonnancement décrit dans le *testbench*. Avec la première option, l'utilisateur n'a pas à concevoir de *testbench* et est libre d'appliquer les changements quand il le souhaite.

Enfin, il est également nécessaire d'alimenter le modèle SystemC en données, et de pouvoir visualiser les résultats en sortie du modèle. Pour fournir les données d'entrée, l'utilisateur dispose d'un module SystemC qui lit une image au format pgm et qui à chaque coup d'horloge, envoie les données formatées selon la dimension des opérandes à alimenter sur les bus In_Data_n. Les résultats peuvent ensuite être visualisés sous 2 formes : une forme graphique avec l'utilisation de la bibliothèque *Cimg*[98] ou sous forme de chronogrammes avec la fonction *trace* intégrée à SystemC et présentée en Annexe. F. Le flot complet d'utilisation du modèle SystemC du SeeProc est résumé dans la Fig. 5.9.

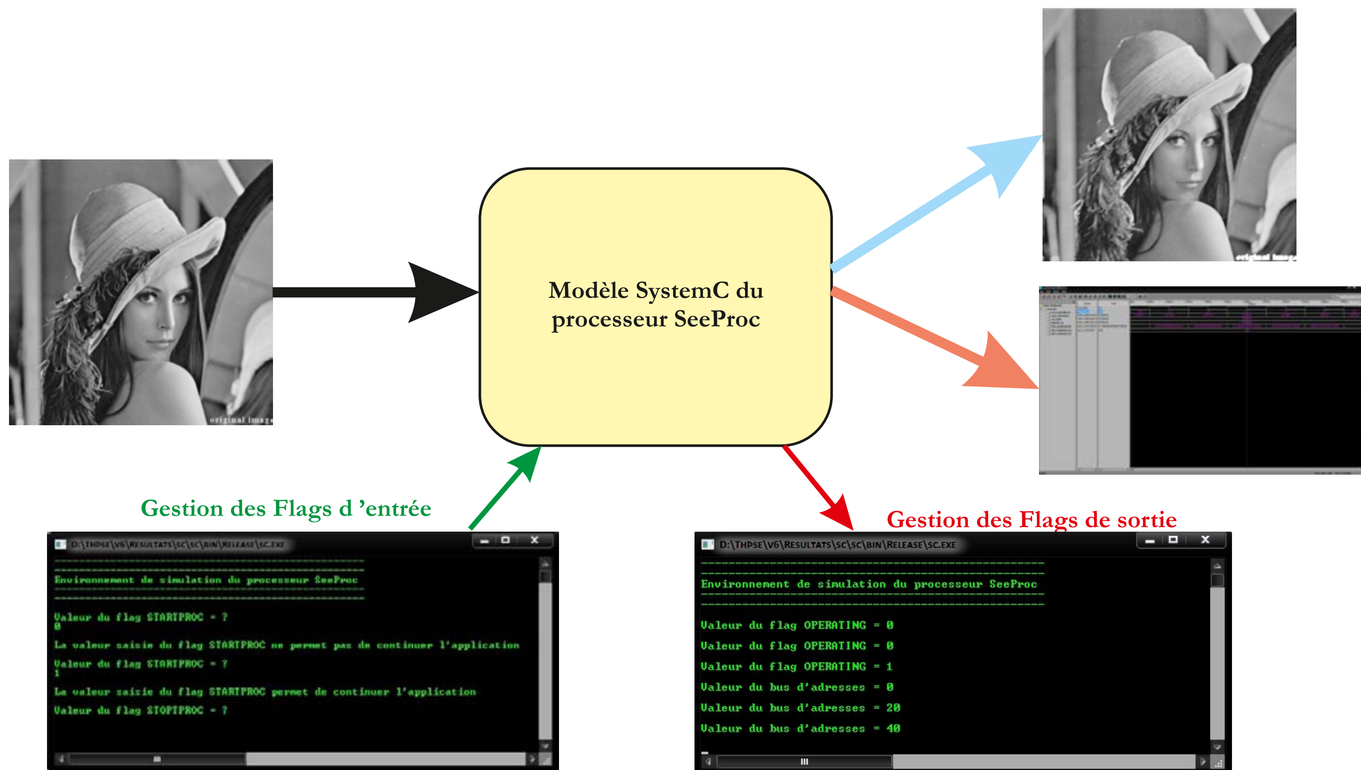


FIGURE 5.9 – Principe d'utilisation du modèle SystemC du processeur SeeProc

Chapitre 6

Résultats expérimentaux

Afin d'évaluer les performances du SeeProc et de valider la méthodologie de développement proposée, une plateforme d'expérimentation a été utilisée. Cette plateforme, la caméra intelligente SeeMOS, sera présentée en Sec. 6.1. Ensuite, en Sec. 6.2 les performances du SeeProc seront comparées aux résultats proposés dans [55]. Les impacts des divers processus de minimisation seront présentés en Sec. 6.3. Enfin, plusieurs applications complètes développées sur SeeProc seront présentées en Sec. 6.5.

6.1 Plateforme d'expérimentation SeeMOS

Le projet SeeMOS est un projet de recherche mené au LASMEA. Son principal objectif est de proposer une plateforme de recherche dédiée à la vision active. Ce projet a consisté dans un premier temps dans le développement d'un prototype de caméra intelligente basée sur un imageur CMOS et un dispositif FPGA [7]. Ce prototype est illustré en Fig. 6.1.

La caméra SeeMOS est une architecture matérielle modulaire de traitement embarqué. La version actuelle de la caméra (Fig. 6.2 et Fig. 6.3) est composée de :

- un capteur d'images CMOS,
- une unité inertielle,
- un dispositif FPGA,
- 5 blocs de mémoires SRAM indépendants,
- un carte de *co-processing* basée sur un DSP,
- une interface de communication Firewire.

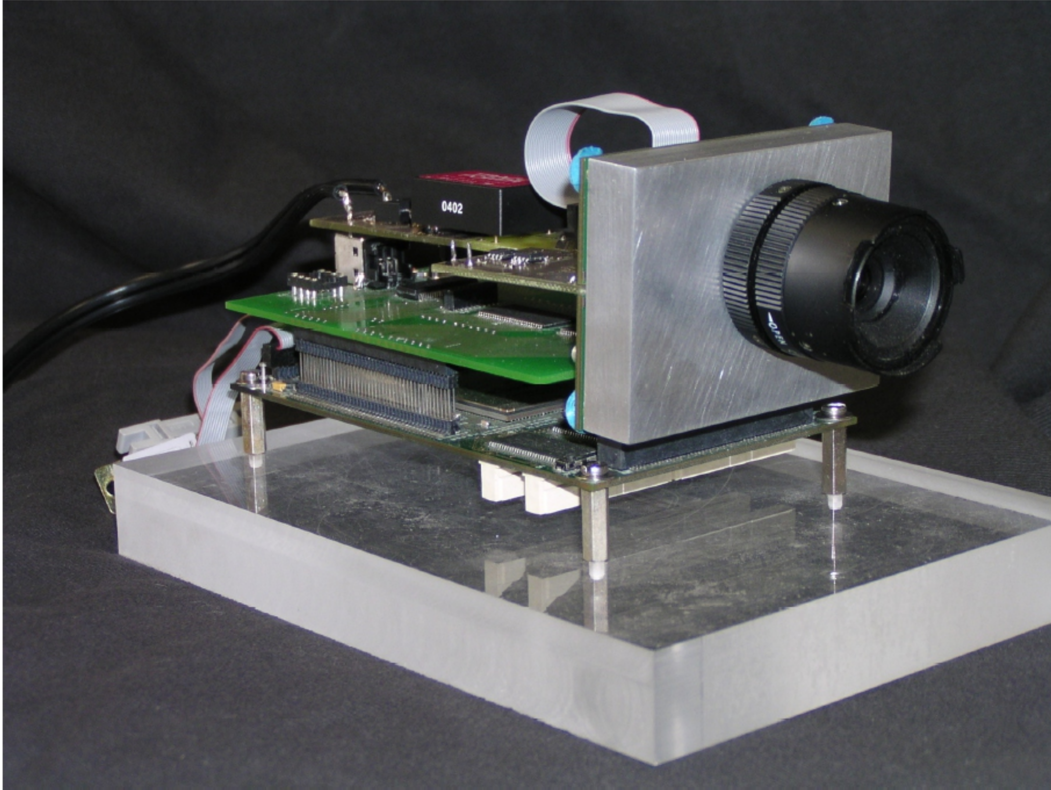


FIGURE 6.1 – Prototype de la caméra intelligente SeeMOS.

Le capteur CMOS est un modèle LUPA-4000 de chez Cypress Semiconductor. Il s'agit d'un capteur d'images monochrome et de résolution 4 Mpixels (2048x2048). Il est capable d'acquérir jusqu'à 66 Mpixels par seconde, chaque pixel étant codé sur 10 bits. L' acquisition est faite en mode "global shutter". La fréquence d'acquisition permet l'obtention d'une cadence d'image de plus de 200 fps [1] pour une résolution VGA (640x480).

Le choix d'un imageur CMOS se justifie principalement par ses capacités d'adressage aléatoire. Ceci s'avère extrêmement utile pour des applications telles que le suivi d'objets, avec la possibilité d'acquérir uniquement une fenêtre d'intérêt contenant l'objet à suivre. L'adressage aléatoire permet donc d'allier, au sein d'un même dispositif, la haute-résolution du capteur à une haute cadence d'acquisition si nécessaire, par l'adressage uniquement d'une petite portion de la matrice photosensible. La conception de la caméra SeeMOS permet d'exploiter pleinement l'adressage aléatoire, obtenant par exemple des cadences de 1000 fps pour des images de résolution 140x140 pixels.

1. fps : Frame per Second, ou image par seconde

L'unité inertielle est composée de 3 accéléromètres, alignés sur les axes X, Y , et Z du capteur, et de 3 gyroscopes. Les données inertielles permettent l'estimation des mouvements 3D de la caméra (ego-motion), ainsi que de son orientation et de sa position.

L'ensemble des composants de la plateforme sont connectés au dispositif FPGA (Fig. 6.4). Ce dernier, de la famille ALTERA Stratix modèle EP1S60F1020C7, joue un rôle central dans le système, étant responsable de l'interconnexion, du contrôle et de la synchronisation des modules sensoriels (imageur + unité inertielle), des RAMs externes, ainsi que des cartes de communication (interface Firewire) et de co-processing (carte DSP). Les principales caractéristiques du FPGA utilisé sont détaillées dans le Tab. 6.1.

Modèle	**Altera Stratix EP1S60F1020F7**
LEs (Logic Elements)	57.120
M512 RAM blocks (32 x 18 bits)	574
M4K RAM blocks (128 x 36 bits)	292
M-RAM blocks (4k x 144 bits)	6
Embedded multipliers (9 x 9 bits)	144
Package	1.020-Pine FineLine BGA
User I/O pins	773
Pitch (mm)	1,00
Area (mm^2)	1.089
Length x width (mm x mm)	33 x 33
Speed grade	7

Table 6.1 – Caractéristiques du dispositif FPGA intégré dans la SeeMOS

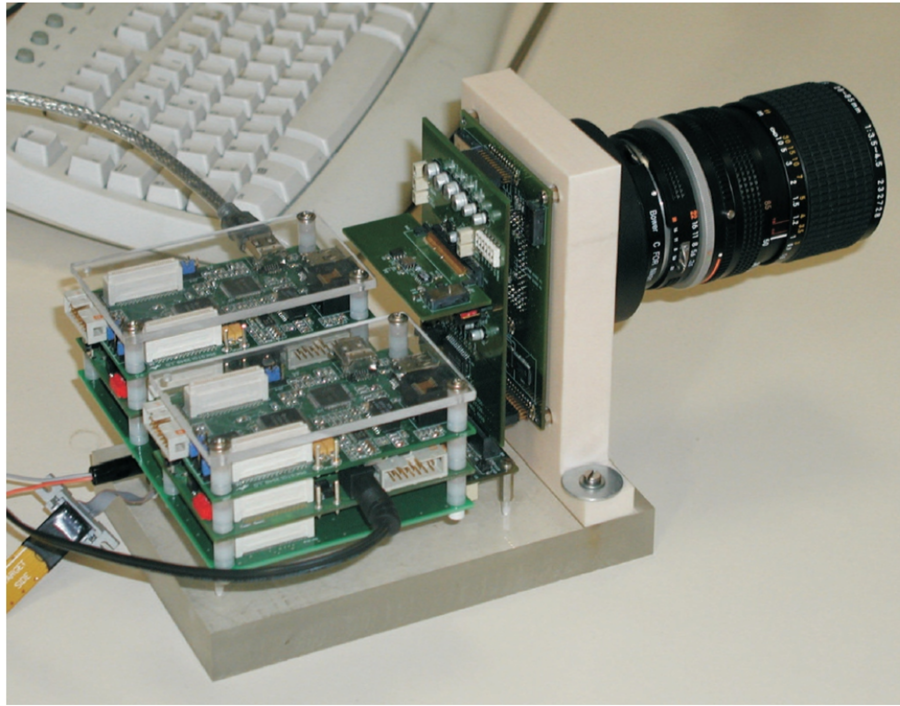


Figure 6.2 – Caméra intelligente SeeMOS.

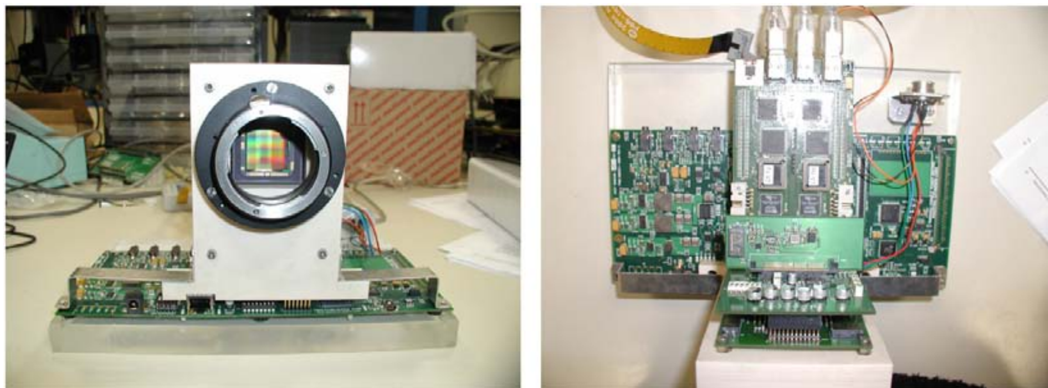


Figure 6.3 – Caméra intelligente SeeMOS.

La carte FPGA intègre également 5 blocs de mémoire RAM statique. Chaque bloc a une capacité de stockage de 2 Mo, divisés en 1M mots de 16 bits. Chacun d'entre eux dispose de son propre bus d'adresse et données. Ceci permet des accès concurrents aux différents blocs, constituant une caractéristique essentielle afin d'exploiter pleinement les possibilités de parallélisme offertes par le circuit FPGA. Un connecteur situé sur le dessous de la carte permet l'ajout d'un

bloc supplémentaire de mémoire SDRAM (SODIMM 144 broches, 133/100 MHz), d'une capacité de jusqu'à 64 Mo.

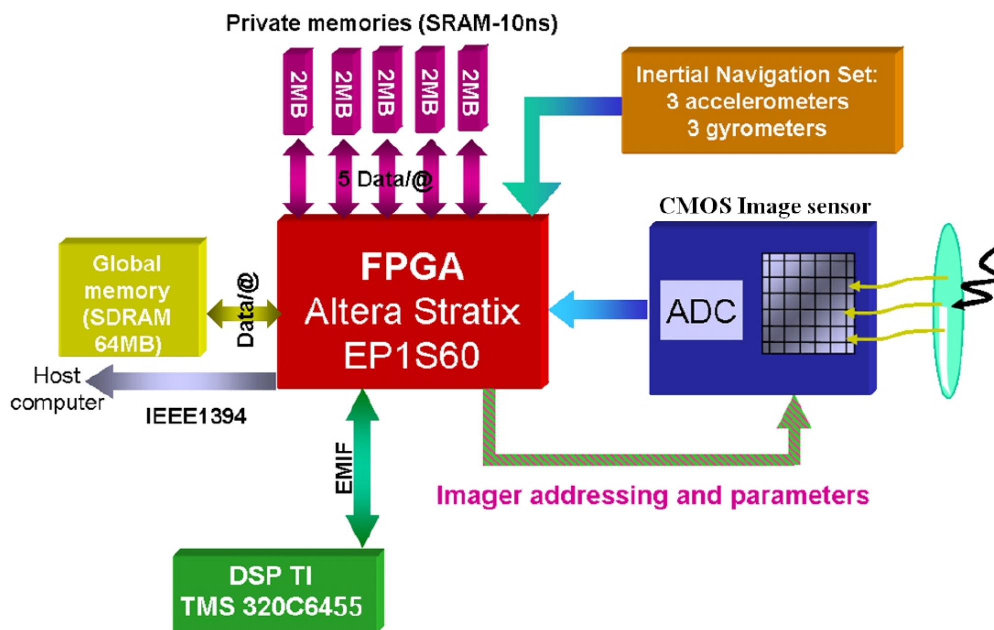


FIGURE 6.4 – Schéma synoptique de la caméra intelligente SeeMOS.

Une carte DSK (DSP Starter Kit) de chez Texas Instruments a également été intégrée à la plateforme. Cette carte dispose d'un dispositif DSP modèle TMS320C6455-1000, basé sur l'architecture VLIW VelociTI. Il s'agit d'un DSP virgule fixe, fonctionnant à une fréquence d'horloge de 1GHz. Il possède un cache L1 de 64Ko (32Ko programme, 32Ko données), et un cache L2 de 2Mo.

L'interface entre la caméra et le système hôte est assurée par un module Firewire (IEEE 1394), délivrant un débit descendant (caméra vers PC) de 20 Mo/s, et un débit ascendant (PC vers caméra) de 10 Mo/s.

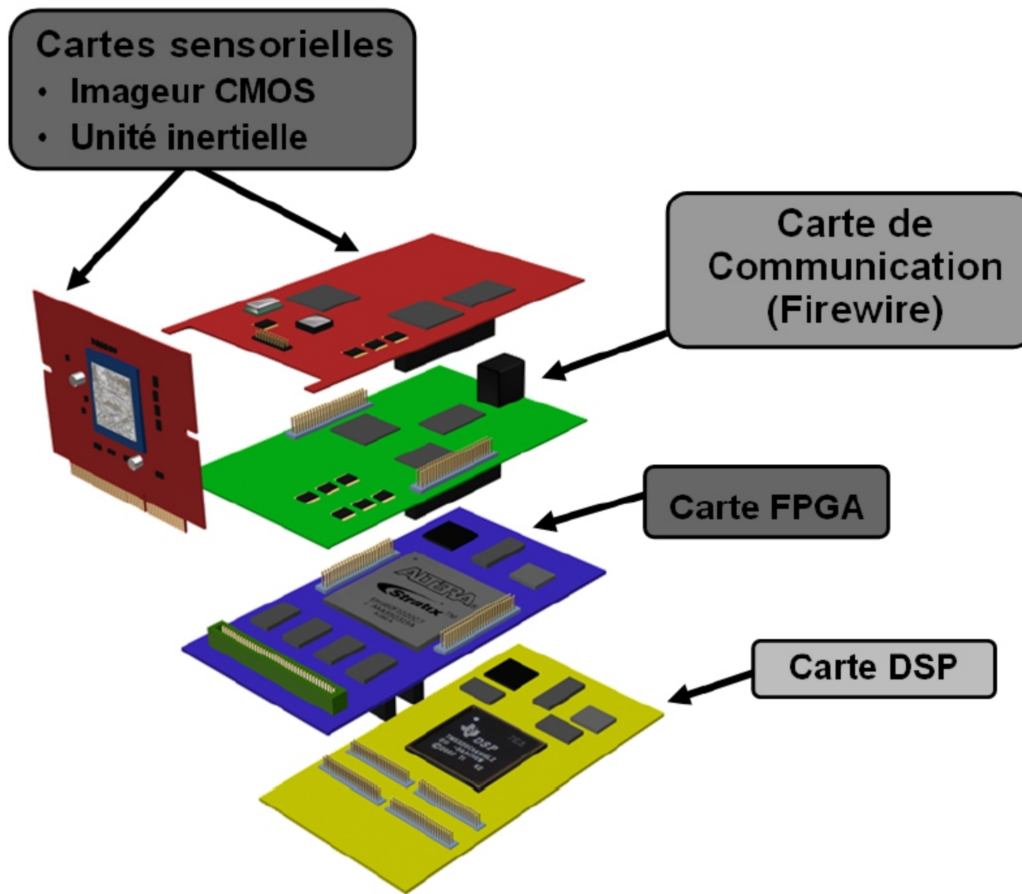


Figure 6.5 – Illustration des différentes cartes composant la plateforme SeeMOS, ainsi que de sa conception modulaire.

Une de forces majeures de la plateforme matérielle SeeMOS provient de sa modularité, les différents éléments matériels étant intégrés dans différentes cartes (Fig. 6.5). Ceci facilite fortement l'évolution de la plateforme, les cartes pouvant être aisément remplacées. On peut donc procéder à une modification ou upgrade d'une partie du système, sans pour autant le re-concevoir dans son intégralité. Grâce à cette caractéristique la plateforme a connu des évolutions constantes, comme en témoignent les figures Fig. 6.1, Fig. 6.2 et Fig. 6.3. D'autres évolutions sont actuellement en vue, notamment l'intégration d'une carte de co-processing DSP dédiée (à la place du DSK Texas), et le passage vers une nouvelle génération de composant FPGA (Stratix III ou Stratix IV).

6.2 Étude des performances de l'architecture SeeProc implantée sur la plateforme SeeMOS

Afin d'évaluer les performances du SeeProc en terme de rapidité de calcul, nous avons, dans un premier temps, programmé ce dernier avec plusieurs programmes assembleur présentés ci-après. Ces programmes ont été écris dans l'optique de comparer ensuite les performances du SeeProc avec d'autres architectures exécutant des algorithmes similaires. Le Tab. 6.2 est extrait de [55] et recense diverses architectures et leur performance en terme de rapidité de traitement.

	Système	**Application**	**Res. Image**	**Temps**	
1	TMS320C80 [64]	5x5 Convolution Gaussienne	512x512	40	ms
2	TMS320C80 [64]	3x3 Dilatation niv. gris	512x512	32.7	ms
3	Splash2 [56]	3x3 Filtre médian	512x512	27	ms
4	PDSP16488 40MHz [109]	8x8 Convolution	512x512	6.56	ms
5	PPIP [116]	5x5 Convolution Gaussienne	256x256	730	μs
6	LSI Logic's L64240[12]	8x8 Convolution	512x512	13.11	ms
7	Blue wave sys. C6200 [114]	3x3 Convolution	512x512	7.2	ms
8	Alacron's AI-860 [5]	8x8 Convolution	512x512	66.1	ms
9	UWGSP5 [80]	3x3 Convolution	512x512	19	ms
10	IMAP vision [70]	3x3 Convolution	256x256	650	μs
11	IMAP vision [70]	3x3 Dilatation niv. gris	256x256	310	μs
12	DECChip 21064 [87]	5x5 Convolution	512x512	220	ms
13	MAP1000 200MHz [65]	7x7 Convolution	512x512	7.9	ms
14	VP24000/10	8x8 template matching	512x512	40	ms
15	Pentium III 500 MHz [11]	8x8 Convolution	512x512	56.5	ms
16	Torres-Huitzil & al. [55]	7x7 windows-based operator	512x512	8.35	ms

Table 6.2 – Évaluation des performances de diverses architecture

Le premier programme applique une filtrage Laplacien 3×3 sur une image d'entrée de résolution 512×512 pixels. Le masque de convolution appliqué est de la forme :

-1	-1	-1
-1	8	-1
-1	-1	-1

Cette application nous permet de comparer les performances du SeeProc avec les systèmes 3, 7, 9 et 10 Le code assembleur correspondant est donné par le

Tab. 6.3.

```
// Programme1
ALUM_NUM 1
ALUM1_MATRIX_SIZE 3

// Debut des instructions
CFG IMW 512
LOAD ALUM1 MULT ID SUM 0
COEF ALUM1 −1 −1 −1 −1 8 −1 −1 −1 −1
ROUT IN_DATA_1 ALUM1_OPA
ROUT IN_DATA_2 ALUM1_OPB
ROUT ALUM1_XOUT X_OUT_DATA
```

TABLE 6.3 – Programme assembleur pour un filtre laplacien 3×3

Résultats de Synthèse : Programme 1			
	Utilisés	**Total**	**Pourcentage**
Éléments logiques	987	57.120	2%
Bits mémoire interne	10.149	5.215.104	$< 1\%$
Éléments DSP bloc 9-bit	8	144	6%
Vitesse Maximale	108,4 MHz soit 2,41 ms de traitement pour une image		

Le second programme assembleur développé applique un masque de convolution de taille 5×5 sur des images de résolution 512×512. Ce programme permet de positionner le SeeProc, en terme de vitesse de calcul, par rapport aux systèmes 1,5 et 12. Le masque de convolution appliqué est le suivant [2] :

-1	-1	-1	-1	-1
-2	-2	-2	-2	-2
0	0	0	0	0
2	2	2	2	2
1	1	1	1	1

2. Les valeurs du masque de convolution appliqué n'ont pas d'importance dans le cas de l'évaluation des performances. En effet, quelque soit le masque de convolution appliqué, la fréquence maximale de fonctionnement est la même.

```
// Programme 2
ALUM_NUM 1
ALU1M_MATRIX_SIZE 5

// Debut des instructions
CFG IMW 512
LOAD ALUM1 MULT ID SUM 0
COEF ALUM1 −1 −1 −1 −1 −1 −2 −2 −2 −2 −2 0 0 0 0 0 2 2 2 2 2 1 1 1 1 1
ROUT IN_DATA_1 ALUM1_OPA
ROUT IN_DATA_2 ALUM1_OPB
ROUT ALUM1_XOUT X_OUT_DATA
```

TABLE 6.4 – Programme assembleur pour une convolution avec un masque de dimension 5×5

Résultats de Synthèse : Programme 2			
	Utilisés	**Total**	**Pourcentage**
Éléments logiques	2.033	57.120	4%
Bits mémoire interne	20.293	5.215.104	$< 1\%$
Éléments DSP bloc 9-bit	24	144	17%
Vitesse Maximale	67,83 MHz soit 3,86 ms de traitement pour une image		

Le troisième programme assembleur développé applique un masque de convolution de taille 7×7 sur des images de résolution 512×512 permettant de comparer le SeeProc avec les systèmes 13 et 16.

```
// Programme 3
ALUM_NUM 1
ALUM1_MATRIX_SIZE 7

// Debut des instructions
CFG IMW 512
LOAD ALUM1 MULT ID SUM 0
COEF ALUM1 −1 −1 −1 −1 −1 −1 −1 −2 −2 −2 −2 −2 −2 −2 −4 −4 −4 −4 −4 −4 −4 0 0 ↩
    0 0 0 0 0 4 4 4 4 4 4 4 2 2 2 2 2 2 2 1 1 1 1 1 1 1
ROUT IN_DATA_1 ALUM1_OPA
ROUT IN_DATA_2 ALUM1_OPB
ROUT ALUM1_XOUT X_OUT_DATA
```

TABLE 6.5 – Programme assembleur pour une convolution avec un masque de dimension 7×7

Résultats de Synthèse : Programme 3			
	Utilisés	**Total**	**Pourcentage**
Éléments logiques	3.776	57.120	7 %
Bits mémoire interne	30.901	5.215.104	1%
Éléments DSP bloc 9-bit	44	144	33%
Vitesse Maximale	36,6 MHz soit 7,16 ms de traitement pour une image		

Enfin le dernier programme assembleur développé applique un masque de convolution de taille 8×8 sur des images de résolution 512×512 permettant de comparer le SeeProc avec les systèmes 6, 8 et 15.

```
// Programme 4
ALUM_NUM 1
ALUM1_MATRIX_SIZE 8

// Debut des instructions
CFG IMW 512
LOAD ALUM1 MULT ID SUM 0
COEF ALUM1 −1 −1 −1 −1 −1 −1 −1 −1 −2 −2 −2 −2 −2 −2 −2 −2 −4 −4 −4 −4 −4 −4 ↩
    −4 −4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 4 4 4 4 4 4 4 2 2 2 2 2 2 2 2 1 1 ↩
    1 1 1 1 1 1
ROUT IN_DATA_1 ALUM1_OPA
ROUT IN_DATA_2 ALUM1_OPB
ROUT ALUM1_XOUT X_OUT_DATA
```

TABLE 6.6 – Programme assembleur pour une convolution avec un masque de dimension 8×8

Résultats de Synthèse : Programme 4			
	Utilisés	**Total**	**Pourcentage**
Éléments logiques	4.933	57.120	9 %
Bits mémoire interne	36.449	5.215.104	1%
Éléments DSP bloc 9-bit	63	144	44%
Vitesse Maximale	25,46 MHz soit 10,02 ms de traitement pour une image		

D'un point de vue comparatif, le SeeProc obtient des performances relativement bonnes vis-à-vis des architectures présentées dans le Tab. 6.2. De plus, on peut voir que le taux d'occupation du FPGA est faible avec les applications de bases appliquées ici. Il est également notable une certaine variation au niveau des vitesses de traitement. Les causes de ces variations sont détaillées dans la Sec. 6.4.

6.3 Étude de l'impact des procédés de minimisation

Afin d'évaluer l'impact des procédés de minimisation mis en place, deux programmes ont été compilés avec et sans processus de minimisation. Le premier programme (MiN1) n'est composée que d'une seule ALUM tandis que le second (MiN2) en utilise 6. Ces programmes sont compilés puis synthétisés sous l'environnement Quartus II distribué par Altera. Les résultats sont disponible en Tab. 6.9 et Tab. 6.10.

```
// Programme MiN1
ALUM_NUM 1
ALU1M_MATRIX_SIZE 3

// Debut des instructions
CFG IMW 512
LOAD ALUM1 MULT ID SUM 0
COEF ALUM1 −1 −1 −1 −1 8 −1 −1 −1 −1
ROUT IN_DATA_1 ALUM1_OPA
ROUT IN_DATA_2 ALUM1_OPB
ROUT ALUM1_XOUT X_OUT_DATA
```

Table 6.7 – Programme 1 de test pour évaluer l'impact des procédés de minimisation

```
// Programme MiN2
ALUM_NUM 6
ALUM1_MATRIX_SIZE 3
ALUM2_MATRIX_SIZE 3
ALUM3_MATRIX_SIZE 3
ALUM4_MATRIX_SIZE 3
ALUM5_MATRIX_SIZE 3
ALUM6_MATRIX_SIZE 3

// Debut des instructions
CFG IMW 512
LOAD ALUM1 MULT ID SUM 0 0
COEF ALUM1 -1 -1 -1 -1 8 -1 -1 -1 -1
LOAD ALUM2 OR ABS AND 0 0
LOAD ALUM3 ADD INV MAX 0 0
COEF ALUM3 128 128 128 50 50 50 0 0 0
LOAD ALUM4 XOR THR MIN 20 0
LOAD ALUM5 ID SL ID 2 0
LOAD ALUM6 ID INV ID 0 0
ROUT IN_DATA_1 ALUM1_OPA
ROUT IN_DATA_2 ALUM1_OPB
ROUT IN_DATA_3 ALUM3_OPA
ROUT IN_DATA_4 ALUM3_OPB
ROUT ALUM1_XOUT ALUM2_OPA
ROUT ALUM3_XOUT ALUM2_OPB
ROUT ALUM2_XOUT ALUM4_OPA
ROUT IN_DATA_5 ALUM4_OPB
ROUT ALUM4_XOUT ALUM5_OPA
ROUT ALUM5_QOUT ALUM6_OPA
ROUT ALUM6_XOUT X_OUT_DATA
```

TABLE 6.8 – Programme 2 de test pour évaluer l'impact des procédés de minimisation

Résultats de Synthèse du programme MiN1			
	Sans minimisation	**Avec minimisation**	**Gain**
Éléments logiques	3.594	987	72,5%
Bits mémoire interne	76.656	10.149	86,7%
Éléments DSP bloc 9-bit	18	8	55,5%
Vitesse Maximale	79,06 MHz	108,4 MHz	27%

TABLE 6.9 – Résultats de Synthèse du programme MiN1 avec et sans minimisation

Résultats de Synthèse du programme MiN2			
	Sans minimisation	**Avec minimisation**	**Gain**
Éléments logiques	27.621	4.003	85,5%
Bits mémoire interne	122.376	43.700	64,2%
Éléments DSP bloc 9-bit	108	8	92,5%
Vitesse Maximale	56,35 MHz	75,7 MHz	25,5%

Table 6.10 – Résultats de Synthèse du programme MiN2 avec et sans minimisation

Les résultats proposés dans les Tab. 6.9 et Tab. 6.10 montrent les avantages, tant en termes de ressources matérielles que de performance, des procédés de minimisation mis en place dans la méthodologie de développement du SeeProc. Au niveau de la diminution des ressources matérielles, celle ci s'explique par le fait que seules les opérations nécessaires sont compilées. De plus, sans procédés de minimisation, toutes les connections possibles entres les éléments de traitements sont instanciées au niveau du crossbar, d'où les ressources supplémentaires, entre autres, au niveau des éléments de mémoires interne. Pour ce qui est des performances, à savoir la fréquence maximale de l'horloge, l'augmentation est notamment du au fait que les connexions du crossbar sont minimisées. Ainsi, le routage matériel est mieux optimisé, car il y a moins de connexions à réalisées, et, par voix de conséquence, le chemin critique est diminué, augmentant ainsi la fréquence maximale du système.

6.4 Étude de l'impact de la dimension des fenêtres d'intérêt

Afin de vérifier l'impact d'un changement de la dimension de la fenêtre d'intérêt, le résultats de synthèse des programmes 1,2 et 3 de la section Sec. 6.2 sont repris dans le Tab. 6.11. Ces programmes appliquent tous les 3 une opération de convolution, permettant ainsi d'évaluer l'impact, en terme de fréquence d'horloge maximale, de la dimension des fenêtres d'intérêt.

On peut ainsi noter une diminution de la vitesse maximale de fonctionnement quand on augmente la dimension de la fenêtre d'intérêt. Ce phénomène est principalement du au fait que le routage matériel des, par exemple, 7 lignes de la fenêtre d'intérêt augmente le chemin critique et diminue la fréquence maximale de fonctionnement.

Il est également notable que la vitesse de traitement est très fortement dépendante

FPGA	Stratix EPIS60F1020C7		
ALUM1_Matrix_Size	3	5	7
Nombre de LE's	987	2.033	3.776
Bits mémoire	10.149	20.293	30.901
Blocs DSP	8	24	44
Vitesse max.	108,4 MHz	67,83 MHz	36,6 MHz

Table 6.11 – Impact de la dimension de la fenêtre d'intérêt

de la fonction de réduction **FR**. En effet, si l'on change l'opération SUM de **FR** par l'opération OR, par exemple, la fréquence maximale d'horloge augmente à 116,96 MHz pour une dimension d'opérande 7×7. Ceci s'explique de par le fait que les opérations de **FR** sont appliquées sur les données d'entrée en les combinant deux à deux, dans une structure d'arbre dyadique. Pour une opérande d'entrée de dimension $(n * n)$, $2 * log_2 n$ étages internes sont nécessaires. Lorsqu'il s'agit d'opérations logiques (AND, OR, XOR ...etc), le chemin critique est moins impacté que lors d'opérations arithmétiques ou de tests (SUM, MAX, MIN).

Enfin, le FPGA de la plateforme SeeMOS est, pour rappel, un Stratix I. Cette gamme de FPGAs est ancienne, pour ne pas dire obsolète. Si l'on compile le programme 3 avec un FPGA Stratix III (EP3SL340H1152C2), la fréquence maximale de fonctionnement atteint 147,36 MHz.

6.5 Applications réalisées avec le processeur SeeProc

Afin de vérifier l'efficacité du flot de développement et les performances du processeur SeeProc dans le cas d'applications complètes, deux applications sont présentées. La première montre l'efficacité de l'interfaçage du co-processeur SeeProc avec un processeur de hiérarchie supérieure. La seconde application met en avant la flexibilité de l'architecture proposée en appliquant divers traitements de diverses dimensions sur des images de diverses résolutions. Enfin la dernière application est l'extraction de points d'intérêt suivant la méthode proposée par Harris et Stephen[58].

Ces applications sont réalisées sur des scènes réelle dont un exemple d'image est donné ci dessous :

FIGURE 6.6 – Illustration de la scène de référence pour les applications.

6.5.1 Application 1 : Interfaçage de SeeProc avec un processeur de contrôle

6.5.1.1 Présentation de l'application

La première application consiste à vérifier l'intégration du co-processeur SeeProc en tant qu'élément de traitement au sein d'un processeur de hiérarchie supérieure. Le processeur choisit est le SeeCore développé par F. Dias de Olivera [20]. Une présentation de ce processeur est donnée en Annexe G.

Le protocole de communication utilisé dans le processeur SeeCore est un protocole de type *handshaking* illustré en Fig. 6.7. Le traitement appliqué est une différence d'images binarisée suivant l'Eq. 6.1.

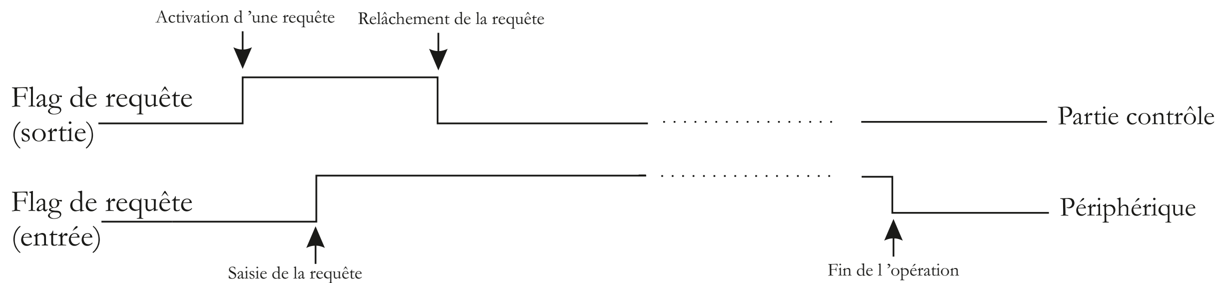


FIGURE 6.7 – Illustration du protocole de communication *handshaking*.

$$R(i,j) = thold(A(i,j) - B(i,j)) \tag{6.1}$$

Où la fonction *thold* répond à l'algorithme suivant :

```
Si A(i,j) − B(i,j) < seuil alors
    R(i,j) = 0
Sinon
    R(i,j) = 1
fin
```

6.5.1.2 Implantation matérielle

Le programme assembleur correspondant à cette application est le suivant :

```
// Chargement des parametres
ALUM_NUM 1
ALUM1_MATRIX_SIZE 1

// Debut des instructions
CFG IMW 600
SET OPERATING
SET nWe1
SET nWe2
LOAD ALUM1 SUB THR ID 15
CFG ALUM1 RANGE 9 9 9
ROUT IN_DATA_1 ALUM1_OPA
ROUT IN_DATA_2 ALUM1_OPB
ROUT ALUM1_XOUT X_OUT_DATA
ADDR BASE 0
ADDR SIZE 479999          //800*600 − 1
ADDR MODE NORMAL
WAIT CSU_FLAG 1
SET OPERATING
RESET nWe2                //Controle de la memoire contenant le resultat
ADDR START
WAIT INBUS 479999
ADDR RESET
RESET OPERATING
JUMP 12
```

Dans le cas de l'application 1, seule une ALUM est nécessaire pour réaliser le traitement souhaité. Le traitement est appliqué sur chaque couple [3] de pixel et ne dépend d'aucun pixel avoisinant. De ce fait, on justifie les paramètres présents en entête du programme. La résolution des images d'entrée étant de 800×600 pixels le paramètre **IMW** est fixé à 600.

Afin de gérer le protocole de *handshaking* deux *flags* sont utilisés. Le *flag* d'entrée **CSU_FLAG** permet ainsi de recevoir la requête envoyée par le processeur SeeCore tandis que le *flag* de sortie **OPERATING** permet d'indiquer à ce dernier que la requête a été reçue et que le traitement est en cours.

Le SeeCore fournit les 2 images successives dans 2 mémoires externes et autorise l'écriture du résultat dans une troisième. Afin de contrôler ces mémoires, les *flags* de sortie **nWe1** et **nWe2** sont utilisés. nWe1 configure les 2 mémoires contenant les images en mode lecture. nWe2 est chargé de la gestion de la mémoire où est stocké le résultat du traitement. Le module d'adressage du co-processeur SeeProc est alors en charge de l'adressage de ces mémoires via les instructions **ADDR**.

Le Tab. 6.12 présente les résultats de synthèse associé à l'application 1. Il est notable la faible utilisation en terme de ressources, 4%, dont seulement 1% sont imputables u processeur SeeProc. Pour une résolution d'image de 800×600 la cadence d'image est de 41.6 IPS [4]. Cette cadence est bridée par le dispositif de communication de la plateforme SeeMOS qui ne permet l'envoi d'un octet toutes les $50\eta s$ (soit à une fréquence de 20 MHz). La Fig. 6.8 illustre les résultats visuels de l'application 1.

Résultats de Synthèse de l'application 1		
Ressources	**Utilisées**	**Disponibles**
Éléments logiques	2.473 (4%)	57.120
Bits mémoire interne	75.088 (1%)	5.215.104
Éléments DSP 9-bit	0	144

TABLE 6.12 – Résultats de Synthèse de l'application 1

3. Un couple de pixel est défini par le pixel de coordonnées (x,y) de l'image I associé au pixel de mêmes coordonnées de l'image I+1.

4. IPS pour Images Par Seconde

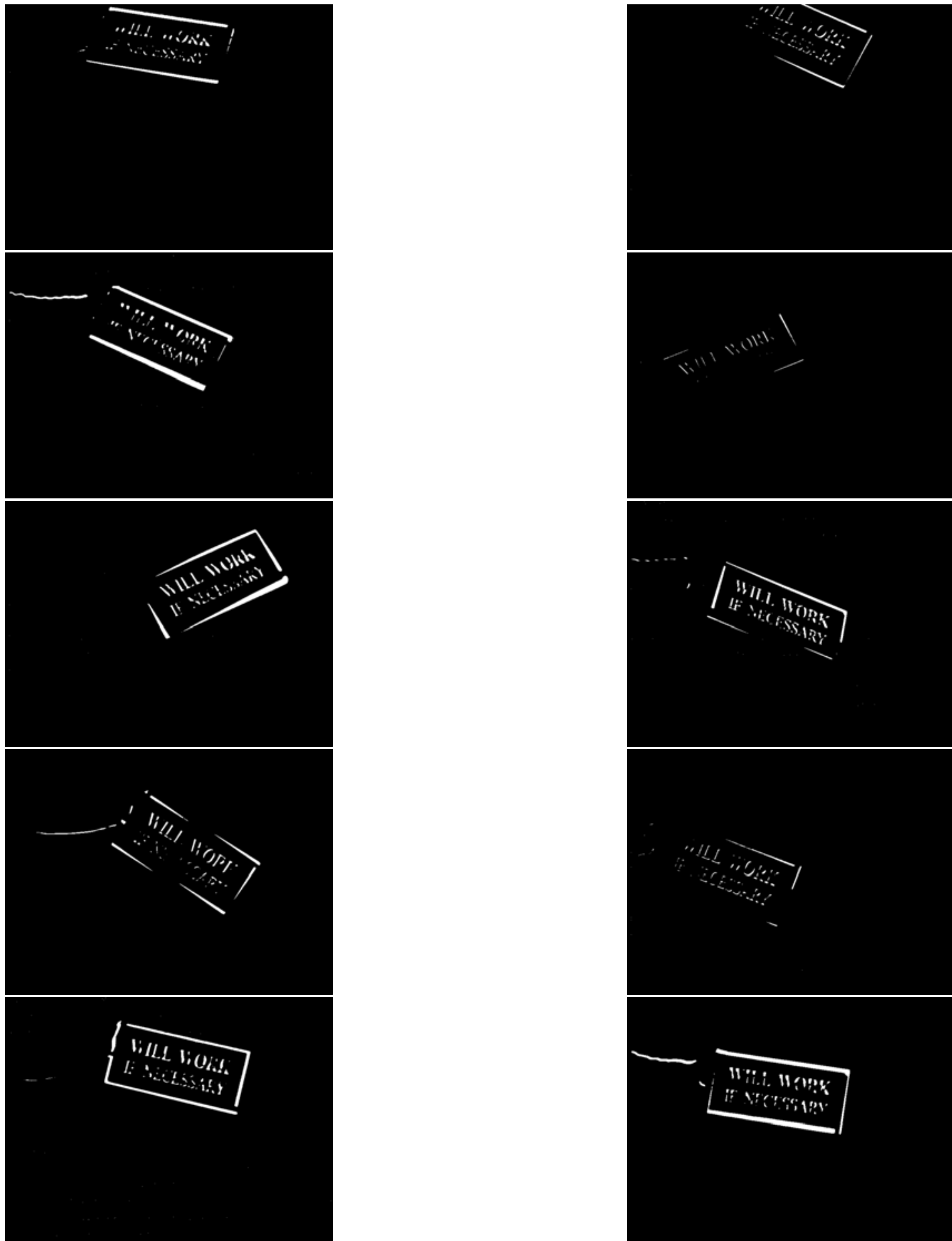


Figure 6.8 – Résultats graphiques de l'application 1

6.5.2 Application 2 : Traitements multi-echelles

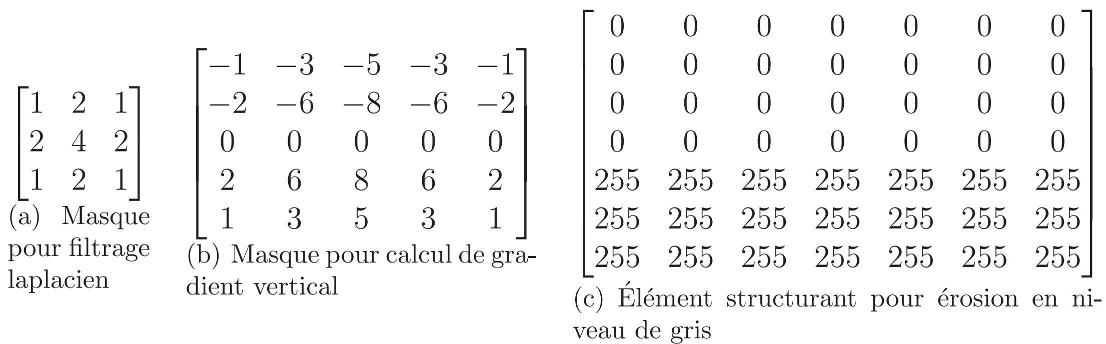
6.5.2.1 Présentation de l'application

Afin de vérifier la flexibilité de l'architecture SeeProc, trois traitements sont réalisés de manière séquentielle sur des images de diverses résolutions. Cette application permet ainsi de vérifier l'efficacité du chemin de données reconfigurable ainsi que des composants responsables de l'alimentation en données. L'application est divisible en trois étapes :

1. La première partie de l'application applique une érosion en niveau de gris avec un élément structurant de dimension 7×7 pixels sur un flux d'images de résolution 512×512 pixels (Fig. 6.9(c)).

2. La seconde partie de l'application extrait un gradient vertical sur une zone d'intérêt de dimension 5×5 sur un flux d'images de résolution 256×256 pixels (Fig. 6.9(b)).

3. La troisième étape applique réalise un filtre gaussien de dimension 3×3 pixels sur un flux d'images de résolution 128×128 pixels (Fig. 6.9(a)).



(a) Masque pour filtrage laplacien

(b) Masque pour calcul de gradient vertical

(c) Élément structurant pour érosion en niveau de gris

FIGURE 6.9 – Masques de traitements pour l'application 2.

6.5.2.2 Implantation matérielle

Le programme assembleur correspondant à cette application est le suivant :

```
// Chargement des parametres
ALUM_NUM 3
ALUM1_MATRIX_SIZE 3
ALUM2_MATRIX_SIZE 5
ALUM3_MATRIX_SIZE 7

// Debut des instructions
LOAD ALUM1 MULT ID SUM 0
CFG RANGE ALUM1 7 10 10
COEF ALUM1 1 2 1 2 4 2 1 2 1
LOAD ALUM2 MULT ID SUM 0
CFG RANGE ALUM2 8 10 10
COEF ALUM2 −1 −3 −5 −3 −1 −2 −6 −8 −6 −2 0 0 0 0 0 2 6 8 6 2 1 3 5 3 1
LOAD ALUM3 SUB ID MIN 0
CFG RANGE ALUM3 10 10 10
COEF ALUM3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255
ROUT IN_DATA_1 ALUM1_OPA
ROUT IN_DATA_2 ALUM1_OPB
ROUT IN_DATA_3 ALUM2_OPA
ROUT IN_DATA_4 ALUM2_OPB
ROUT IN_DATA_5 ALUM3_OPA
ROUT IN_DATA_6 ALUM3_OPB
ROUT ALUM3_XOUT X_OUT_DATA
CFG IMW 512
RESET OPERATING
RESET DONE
WAIT CSU_FLAG 1
CFG IMW 256
SET OPERATING
RESET DONE
ROUT ALUM2_XOUT X_OUT_DATA
WAIT CSU_FLAG 0
CFG IMW 128
RESET OPERATING
SET DONE
ROUT ALUM1_XOUT X_OUT_DATA
```

Les trois traitements proposés travaillant sur des dimensions de fenêtre d'intérêt différentes, trois ALUMs sont utilisées. L'ALUM1 est en charge du filtrage laplacien, l'ALUM2 de l'extraction d'un gradient vertical et l'ALUM3 de l'érosion en niveau de gris. C'est ainsi l'ALUM3 qui opère en premier lieu sur une image de résolution 512×512 pixels.

Afin de signifier au contrôleur de l'imageur la résolution d'image à acquérir, deux flags de sorties,**OPERATING** et **DONE**, sont utilisés par le processeur SeeProc de la manière suivante :

- Lorsque OPERATING = DONE = '0' alors on acquiert une image de résolution 512×512 pixels.
- Lorsque OPERATING = '1' et DONE = '0' alors on acquiert une image de résolution 256×256 pixels.
- Lorsque OPERATING = '0' et DONE = '1' alors on acquiert un image de

résolution 128×128 pixels.

Le passage d'une résolution à une autre est géré par le flag d'entrée **CSU_FLAG**. Cet indicateur est contrôlé par un compteur externe qui compte 300 images de la résolution courante avant d'inverser le signal CSU_FLAG et ainsi de spécifier un changement de résolution.

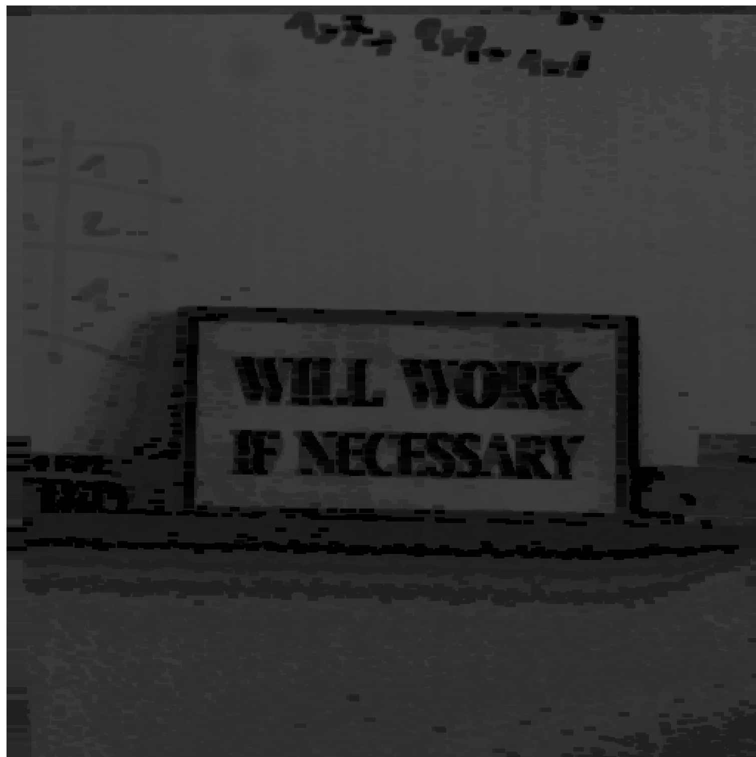


FIGURE 6.10 – Résultat graphique de l'érosion.

FIGURE 6.11 – Résultat graphique du gradient horizontal.

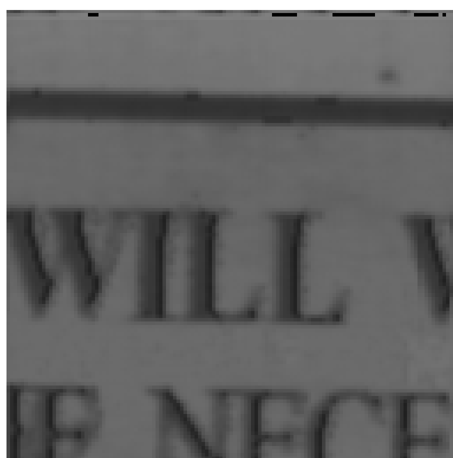


FIGURE 6.12 – Résultat graphique du filtre gaussien.

Le Tab. 6.13 expose les résultats de synthèse de l'application. Les taux d'utilisation des éléments logiques est cette fois de 37%. Ceci s'explique de par le fait que trois ALUMs sont utilisées et qu'elles admettent des opérandes de dimensions différentes. Ainsi, un SPG différent est nécessaire pour chacune d'entre elles, augmentant ainsi le nombre d'éléments logiques utilisés. L'application d'opérations de multiplication impact également le nombre d'éléments DSP. Enfin, on constate

que la mémoire interne utilisée reste faible.

Il est également notable la fréquence maximale du processeur SeeProc qui avoisine les 4 MHz. Ceci est du à la présence de l'opération MIN effectuée au niveau de l'ALUM3 qui impact fortement sur le chemin critique. De plus, la dimension des opérandes de l'ALUM3 sont de 7×7 ce qui augmente également le chemin critique. Néanmoins, cette vitesse est à pondérer dans le sens où le FPGA utilisé dans ces applications est un FPGA très ancien et actuellement obsolète.

Résultats de Synthèse de l'application 2		
Ressources	**Utilisées**	**Disponibles**
Éléments logiques	20.850 (37%)	57.120
Bits mémoire interne	56.136 (1%)	5.215.104
Éléments DSP 9-bit	34 (24%)	144
Fréquence maximale du SeeProc		4.16 MHz

Table 6.13 – Résultats de Synthèse de l'application 2

6.5.3 Application 3 : Extraction de points d'intérêt

6.5.3.1 Présentation de l'application

La troisième application développée est l'exécution de l'algorithme de détection de points d'intérêt d'*Harris et Stephen* [58]. Cet algorithme, fréquemment utilisé dans le domaine de la vision pour la navigation, permet d'extraire les primitives d'une image. Les primitives détectées peuvent être de deux sortes : un contour ou un coin. C'est ce second type de primitive qui est communément appelé point d'intérêt de *Harris*.

L'algorithme de *Harris&Stephen* caractérise l'auto-corrélation d'une image multipliée par une fonction de lissage (Gaussienne). On va ainsi calculer les valeurs propres de la la matrice de Harris présentée en Equ. 6.2. Ces valeurs propres renseignent sur les caractéristiques de l'image. Si les deux valeurs propres ont des valeurs proches et élevées, alors on est en présence d'un coin. Si seulement une des deux valeurs propres est élevée alors il s'agit d'un contour. Sinon, on est dans une région dite homogène.

$$M = \begin{pmatrix} A & C \\ C & B \end{pmatrix} \quad (6.2)$$

Avec

$$A = \frac{\delta I^2}{\delta x} \otimes w \quad B = \frac{\delta I^2}{\delta y} \otimes w \quad C = (\frac{\delta I}{\delta x}\frac{\delta I}{\delta y}) \otimes w$$

Où w représente le masque de convolution gausien. Il est ensuite possible d'extraire les valeurs propres de M (Equ. 6.3 et Equ. 6.4).

$$\lambda_1 = \frac{1}{2}(A + B - \sqrt{A^2 + 4C^2 - 2AB + B^2}) \tag{6.3}$$

$$\lambda_2 = \frac{1}{2}(A + B + \sqrt{A^2 + 4C^2 - 2AB + B^2}) \tag{6.4}$$

Le calcul de ces deux valeurs propres étant peu aisé, notamment au niveau matériel, *Harris&Stephen* ont proposé l'opérateur exprimé en Equ. 6.5 afin de facilité la caractérisation du point traité :

$$R = Det(M) - k * Trace(M)^2 \tag{6.5}$$

avec : $Det(M) = AB - C^2$, $Trace(M) = A + B$ et $k \in [0.04, 0.06]$ [5]

Ainsi, les points d'intérêt, autrement dit les coins, sont détectés lorsque R > 0.

6.5.3.2 Implantation matérielle

L'algorithme peut être décomposé suivant la Fig. 6.13. Dans le cadre de cette application, le processeur SeeProc est en charge du calcul des primitives intermédiaires A,B et C. La suite du calcul est réalisée par des opérateurs externes même si le processeur SeeProc aurait pu les prendre en charge. Ce choix se justifie de par le fait que les primitives A,B et C sont des scalaires et qu'il est dommage, ne serait-ce qu'en termes de ressources matérielles, d'utiliser un processeur pour effectuer des additions et multiplications scalaires.

5. k est une valeur déterminée empiriquement. Qui s'appuie seulement sur l'expérience

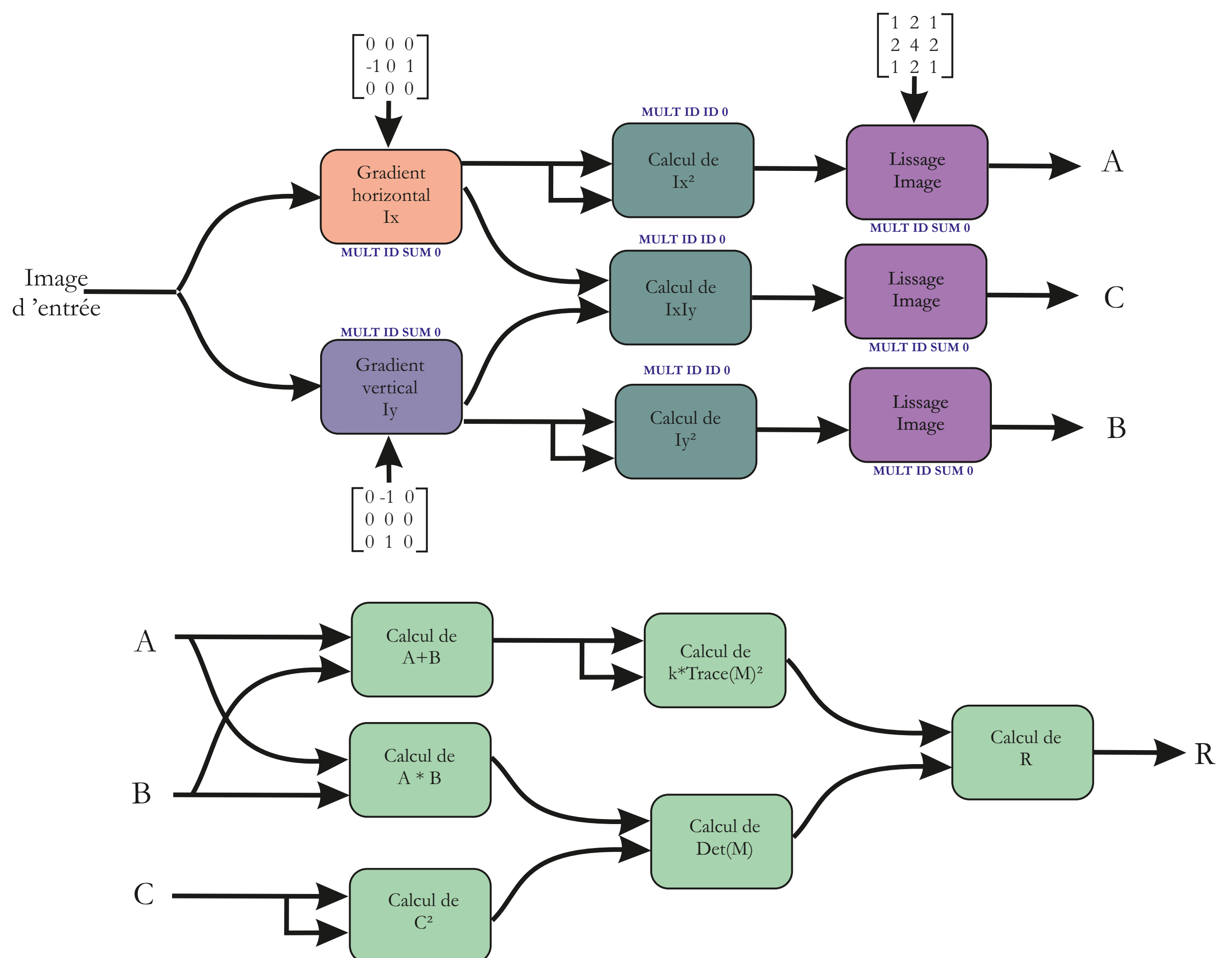


Figure 6.13 – Décomposition de l'algorithme de Harris et Stephen.

Dans sa version actuelle, le processeur SeeProc n'admet qu'une seule sortie scalaire. De ce fait, pour cette application plusieurs instanciations sont nécessaires. Ainsi, cinq processeurs SeeProc sont instanciés pour le calcul des primitives A, B et C. Les deux premiers processeurs permettent le calcul des gradients verticaux et horizontaux (Tab. 6.14 et Tab. 6.15). Les trois suivants sont identiques et calculent chacun une des trois primitives (Tab. 6.16).

```
// Chargement des parametres
ALUM_NUM 1
ALUM1_MATRIX_SIZE 3

// Debut des instructions
CFG IMW 512
LOAD ALUM1 MULT ID SUM 0
CFG RANGE ALUM1 10 10 10
COEF ALUM1 0 0 0 −1 0 1 0 0 0
ROUT IN_DATA_1 ALUM1_OPA
ROUT IN_DATA_2 ALUM1_OPB
ROUT ALUM1_XOUT X_OUT_DATA
```

Table 6.14 – Extraction du gradient horizontal

```
// Chargement des parametres
ALUM_NUM 1
ALUM1_MATRIX_SIZE 3

// Debut des instructions
CFG IMW 512
LOAD ALUM1 MULT ID SUM 0
CFG RANGE ALUM1 10 10 10
COEF ALUM1 0 −1 0 0 0 0 0 1 0
ROUT IN_DATA_1 ALUM1_OPA
ROUT IN_DATA_2 ALUM1_OPB
ROUT ALUM1_XOUT X_OUT_DATA
```

Table 6.15 – Extraction du gradient vertical

```
// Chargement des parametres
ALUM_NUM 2
ALUM1_MATRIX_SIZE 3
ALUM2_MATRIX_SIZE 3

// Debut des instructions
CFG IMW 512
LOAD ALUM1 MULT ID ID 0
CFG RANGE ALUM1 8 10 10
LOAD ALUM2 MULT ID SUM 0
COEF ALUM2 1 2 1 2 4 2 1 2 1
CFG RANGE ALUM2 7 10 10
ROUT IN_DATA_1 ALUM1_OPA
ROUT IN_DATA_2 ALUM1_OPB
ROUT ALUM1_ROUT ALUM2_OPA
ROUT IN_DATA_3 ALUM2_OPB
ROUT ALUM2_XOUT X_OUT_DATA
```

Table 6.16 – Calcul des primitives

Figure 6.14 – Résultat graphique de l'application 3.

Le Tab. 6.17 donne les résultats de synthèse de cette application. On peut voir que 14% des ressources, en terme d'éléments logiques, du FPGAs sont utilisées. Dû au nombre important de multiplication de l'algorithme, 35% des éléments DSP sont utilisés. Enfin la fréquence maximale obtenue est de 46.67 MHz, ce qui en d'autre terme permet d'obtenir théoriquement une cadence d'images de 175 IPS pour une résolution d'image de 512×512.

Résultats de Synthèse de l'application 3		
Ressources	**Utilisées**	**Disponibles**
Éléments logiques	8.220 (14%)	57.120
Bits mémoire interne	47.796 (1%)	5.215.104
Éléments DSP 9-bit	51 (35%)	144
Fréquence maximale du SeeProc		46.67 MHz

Table 6.17 – Résultats de Synthèse de l'application 3

Chapitre 7

Conclusion et Perspectives

Les travaux présentés dans ce manuscrit ont été réalisés au sein du groupe GRAVIR (GRoupe d'Automatique : VIsion et Robotique) au LASMEA (Laboratoire des Sciences et Matériaux pour l'Électronique et d'Automatique) de Clermont-Ferrand en collaboration avec le Commissariat Européen Atomique (CEA) de Fontenay-aux-Roses. Ils s'inscrivent dans la thématique "Systèmes de Perception".

Le développement de l'architecture présentée dans ce manuscrit a été régie par plusieurs objectifs qui sont rappelés ci-dessous :

Facilité d'utilisation : La programmation matérielle en VHDL/Verilog nécessitant de larges compétences en électronique numérique, une interface de programmation matérielle, plus accessible aux programmeurs *logiciel* et leur rendant transparent ces aspects matériels, doit être mise au point.

Flexibilité : Selon les besoin de l'utilisateur et/ou de l'environnement, l'application doit être capable d'évoluer, fonctionnellement parlant, dynamiquement, d'où la nécessité d'une solution flexible.

Performance : Il faut que la solution proposée puisse être capable d'exécuter des applications de traitement d'images temps réel.

Faible occupation des ressources matérielles : Le traitement d'image bas niveau n'étant qu'un préambule de l'application finale, il est primordial que les ressources requises pour ces traitements soient optimisées.

Portabilité : Au vu du nombre grandissant d'architecture de vision basées sur un FPGA, il nous a paru essentiel que l'approche développée soit portable sur n'importe quel composant FPGA.

Les solutions apportées à ces objectifs sont les suivantes :

Concernant la **Facilité d'utilisation**, la mise au point d'un jeu d'instructions dédiée au processeur SeeProc associé à un outil assembleur permet de manière

simple à un concepteur orienté *logiciel* de décrire une application sur le processeur SeeProc. Afin de permettre un prototypage rapide d'application, un modèle complet, utilisant le langage SystemC, est également à disposition du concepteur. Ce modèle permet d'avoir une estimation aussi proche du résultat final que possible et d'offrir la possibilité d'effectuer des mises au points rapidement.

La **Flexibilité** permise par le SeeProc s'appuie sur une architecture de type processeur à chemin de données reconfigurable (Sec. 3.1). Ce type de processeur permet la reconfiguration dynamique, dans le cas du SeeProc en contrôlant un réseau d'interconnexion de type Crossbar complet pré-câblé, de l'agencement entre les différents éléments de traitement. Il permet également une reconfiguration fonctionnelle des unités de traitement.

La définition d'un système ayant une **Faible occupation en termes de ressources matérielles** est permise par l'intermédiaire de processus de minimisation matérielles intégrés dans les outils de développement associée au processeur SeeProc. Ces processus permettent, en fonction du code assembleur decrivant l'application, d'extraire un certain nombre de paramètres utilisés ensuite lors de la génération des éléments VHDL afin de minimiser, en termes de ressources matérielles, chacun de ces éléments.

Afin d'avoir un processeur ayant des **Performances** optimales dans le cadre d'applications de traitement d'images, nous avons proposé un opérateur dédié à ces applications. Cet opérateur, baptisé ALUM, permet d'effectuer efficacement la plupart des traitements d'images grâces à trois unités SIMD. L'aspect performance est aussi intrinsèquement lié à l'aspect précédent. En effet, grâce à la minimisation des ressources, nous avons pu observer un gain non négligeable en termes de performance.

Enfin, la **Portabilité** du processeur est assurée par les outils de développement associés au SeeProc. En effet, au sein de ces outils, un étape de génération des différents éléments du processeur SeeProc a été intégrée afin de fournir tous ces éléments en langage VHDL. Ainsi, le processeur SeeProc peut être instancié sur n'importe quelle famille de FPGA.

Les travaux présentés dans ce manuscrit laissent de nombreuses voies ouvertes pour des améliorations et développements futurs. Une de ces voies est le développement d'un compilateur capable de générer, à partir d'un langage haut niveau, textuel ou graphique, le code assembleur permettant de programmer l'architecture SeeProc. Ceci dans le but d'appréhender plus facilement la programmation du processeur SeeProc.

D'un point de vue architectural, les applications, présentées dans le chapitre

précédent, ont permis de mettre en avant certaines limitations dans l'architecture actuelle. En premier lieu, on a pu voir que que le fait de ne disposer que d'une seule sortie scalaire par ALUM oblige parfois le concepteur à instancier plusieurs processeurs pour répondre à ses besoins. Une autre amélioration intéressante serait d'avoir la possibilité de paramètrer la dimension des bus de chaque ALUM. En effet, actuellement les bus intra-ALUM sont limités à un taille de 9 bits, ceci pouvant impacter sur la précision des résultats obtenus.

Annexe A

Présentation de Lex & Yacc

LEX (LEXical parser) & **YACC** (Yet Another Compiler Compiler) sont des outils qui engendrent des programmes d'analyse de texte. Les programmes générés offrent des fonctionnalités de reconnaissance, de structuration, de traduction d'un texte écrit dans un langage donné. Leurs implémentations GNU se nomment respectivement Flex [103] et Bison [102].

LEX permet de générer des séquences de lexèmes à partir de la séquence de caractères présente en entrée. Ces lexèmes sont définis sous la forme d'expressions régulières. Le Tab. A.1 présente divers exemples d'expressions régulières (E.R.). Ensuite, à chaque lexème reconnu, une valeur (un token) est associé et sera utilisée par l'analyseur syntaxique. Par analogie, on peut considérer que l'outil LEX permet de découper le texte en mots.

E.R.	Recherche...
abc	la chaîne "abc"
a*	les chaînes formées de 0 ou plusieurs "a" ; par ex : ∅,a,aaa,...
a+	les chaînes formées de 1 ou plusieurs "a" ; par ex : a,aaa,...
a(bc) ?	la chaîne "a" suivie éventuellement de la chaîne "bc"
a—(bc)	la chaîne "a" ou la chaîne "bc"
a.c	toute chaîne dont le premier caractère est un "a" et le dernier un "c"
[a-z]	tous les caractères en minuscules
[ˆ ab]	tous les caractères sauf "a" et "b"
.	n'importe quel caractère

Table A.1 – Exemples d'expressions régulières

YACC va générer des analyseurs syntaxiques. Un analyseur syntaxique permet de déclencher des actions quand des structures grammaticales sont reconnues.

YACC accepte la description d'une grammaire et vérifie que la séquence de tokens produites par LEX est conforme à cette grammaire. YACC transforme la description d'une grammaire en un automate à pile qui :

- lit les tokens produits par LEX ;
- à chaque nouveau token décide si une règle peut-être appliquée ou s'il faut garder le token en réserve ;
- décide en finale si la séquence complète de tokens satisfait la règle principale.

La Fig. A.1 propose un exemple d'analyse lexicale et grammaticale utilisant les outils LEX et YACC [110].

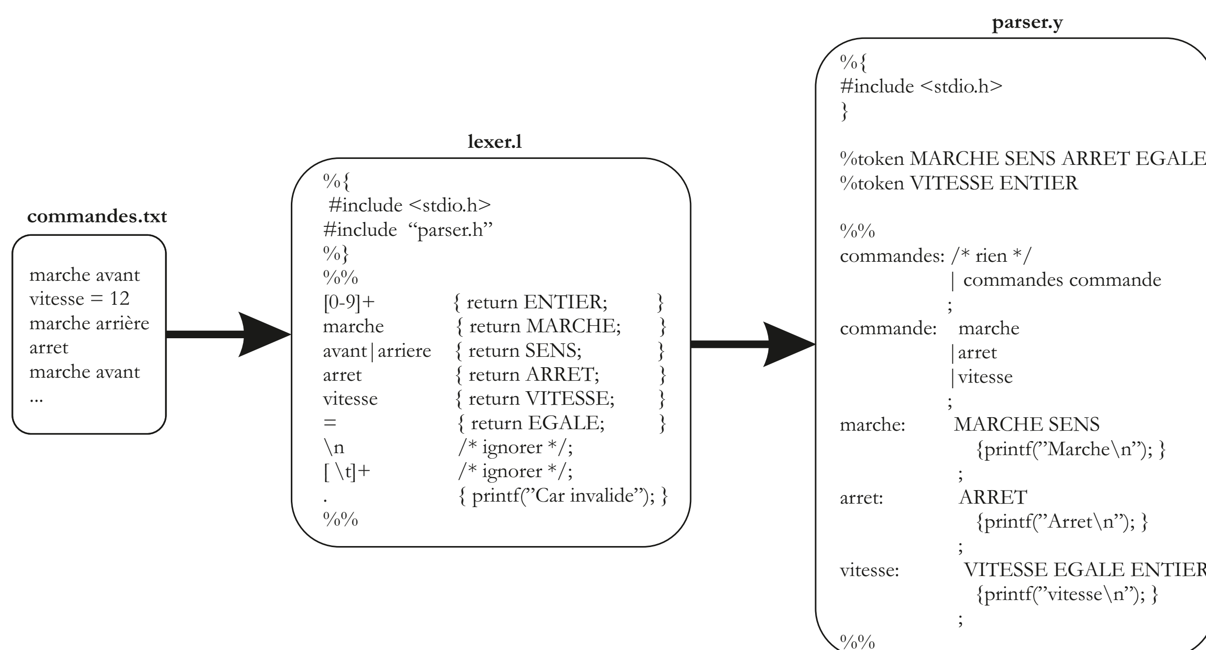


FIGURE A.1 – Exemple d'analyse lexicale et grammaticale d'un programme avec LEX et YACC.

Annexe B

Grammaire du langage assembleur

```
/* Regles */

programme:
         entete ligne
          ;

entete:
         alum_num matrix_size
        ;

alum_num:
          ALUM_NUM VAL
        ;

matrix_size:
          matrix_size matrix_sizenum
        ;

matrix_sizenum:
          ALUM1_MATRIX_SIZE VAL
        | ALUM2_MATRIX_SIZE VAL
        | ALUM3_MATRIX_SIZE VAL
        | ALUM4_MATRIX_SIZE VAL
        | ALUM5_MATRIX_SIZE VAL
        | ALUM6_MATRIX_SIZE VAL
        ;

ligne:
        | ligne instruction
      ;
instruction:
        WAIT INFLAG VAL
      | LOAD NUM_ALU OPCODE OPCODE OPCODE VAL
      | SET OUTFLAG
      | RESET OUTFLAG
      | JUMP VAL
      | ROUT SOURCE DESTINATION
      | COEF NUM_ALU coeff
      | CFG IMW VAL
      | CFG RANGE NUM_ALU VAL VAL VAL
      | TEMPO VAL
      | ADDR ADDROPT1       // ADDROPT1: instructions Start/Stop/Res
      | ADDR ADDROPT2 VAL   // ADDROPT2: instructions Base/Size
      | ADDR MODE ADDROPT3  // ADDROPT3: mode d'incrementation
      | IFRES IFOPT1 VAL    // IFOPT1: instruction Bpos/Bneg/Bzero
      | IFRES VAL IFOPT2 VAL// IFOPT2: instruction Sup/Inf/Equ
    ;

coeff:
      | coeff VAL
    ;
```

Annexe C

Correspondance Alias - Valeur Hexadécimale

Les tableaux ci-dessous donnent la correspondance entre les mnémoniques utilisés lors de l'écriture des instructions assembleur et les valeurs hexadécimales correspondantes en code machine

Mnémonique	Opcode en codage hexadécimal
WAIT	0
LOAD	1
SET	2
RESET	3
JUMP	4
ROUT	5
COEF	6
CFG	7
TEMPO	8
ADDR	9
IFRES	A

Table C.1 – Correspondance Opcodes

Mnémonique	Opcode en codage hexadécimal
CSU_FLAG	0
STARTPROC	1
STOPPROC	2
IN_BUS	9

Table C.2 – Correspondance Flags d'entrée

Mnémonique	Opcode en codage hexadécimal
OPERATING	0
DONE	1
nWe1	2
nWe2	3

Table C.3 – Correspondance Flags de sortie

Mnémonique	Opcode en codage hexadécimal
ALUM1	1
ALUM2	2
ALUM3	3
ALUM4	4
ALUM5	5
ALUM6	6
ID	0
ADD	1
SUB	2
MULT	3
AND	4
OR	5
XOR	6
ID	0
INV	1
ABS	2
SQR	3
SL	4
SR	5
THR	6
ID	0
SUM	1
MAX	2
MIN	3
AND	4
OR	5
XOR	5

Table C.4 – Correspondance des ALUMs et de leurs opérations

Mnémonique	Opcode en codage hexadécimal
IN_DATA_1	1
IN_DATA_2	2
IN_DATA_3	3
IN_DATA_4	4
IN_DATA_5	5
IN_DATA_6	6
IN_DATA_7	7
IN_DATA_8	8
IN_DATA_9	9
IN_DATA_10	A
IN_DATA_11	B
IN_DATA_12	C
ALUM1_XOUT	D
ALUM1_ROUT	E
ALUM1_QOUT	F
ALUM2_XOUT	10
ALUM2_ROUT	11
ALUM2_QOUT	12
ALUM3_XOUT	13
ALUM3_ROUT	14
ALUM3_QOUT	15
ALUM4_XOUT	16
ALUM4_ROUT	17
ALUM4_QOUT	18
ALUM5_XOUT	19
ALUM5_ROUT	1A
ALUM5_QOUT	1B
ALUM6_XOUT	1C
ALUM6_ROUT	1D
ALUM6_QOUT	1E

Table C.5 – Correspondance des ports d'entrée du Crossbar

Mnémonique	Opcode en codage hexadécimal
X_OUT_DATA	0
OUT_DATA	1
ALUM1_OPA	2
ALUM1_OPB	3
ALUM2_OPA	4
ALUM2_OPB	5
ALUM3_OPA	6
ALUM3_OPB	7
ALUM4_OPA	8
ALUM4_OPB	9
ALUM5_OPA	A
ALUM5_OPB	B
ALUM6_OPA	C
ALUM6_OPB	D

Table C.6 – Correspondance des ports de sortie du Crossbar

Annexe D

Comparaison de la syntaxe des fichiers .mif pour Altera et .coe pour Xilinx

```
FORMAT MIF                    |           FORMAT COE
                              |
WIDTH=16;                     |          DEPTH=16;
ADDRESS_RADIX=UNS;            |          memory_initialization_radix=16;
DATA_RADIX=HEX;               |          memory_initialization_vector=
                              |
CONTENT BEGIN                 |
    0 :   ff;                 |          ff,
    1 :   ab;                 |          ab,
    2 :   f0;                 |          f0,
    3 :   11;                 |          11,
    4 :   11;                 |          11,
    5 :   00;                 |          00,
    6 :   01;                 |          01,
    7 :   aa;                 |          aa,
    8 :   bb;                 |          bb,
    9 :   cc;                 |          cc;
    [10..15]: 00;             |
END;                          |
```

TABLE D.1 – Syntaxes fichiers MIF et COE

Annexe E

Algorithmes de génération des fichiers VHDL du SeeProc

E.1 Éléments de la partie de contrôle

```
Parametre utilise: prefix, numlig
Debut

       I - Ouvrir/creer un fichier nomme "prefix_Seeproc_Memoire_programme.vhd↩
           "

       II - Ecrire le code VHDL de la memoire programme en
       fonction de numlig
Fin
```

Table E.1 – Algorithme de création de la mémoire programme

```
Parametre utilise: prefix, nalu, dim minadrr, imw et coef
Debut
       I - Ouvrir/creer un fichier nomme "prefix_Seeproc_unite_controle.vhd"

       II - Ecrire les ports d entree/sortie en fonction des
       param\`{e}tres nalu, dim, minaddr, imw et coef

       III - Ecrire la machine d etat pour les etapes de fetch et
       decode

       IV - Ecrire l etape execute en fonction des param\`{e}tres nalu, dim, ↩
           minaddr, imw et coef
Fin
```

Table E.2 – Algorithme de création de l'unité de contrôle

```
Parametre utilise: prefix, nalu, dim, minaddr et coef Debut
       I - Ouvrir/creer un fichier nomme "prefix_Seeproc_Decoder.vhd"

       II - Ecrire les ports d entree/sortie en fonction des
       param\`{e}tre nalu, dim, minaddr et coef

       III - Ecrire la declaration composant "prefix_Seeproc_Memoire_programme↩
           "

       IV - Ecrire la declaration du composant "prefix_Seeproc_Program_counter↩
           "

       V - Ecrire la declaration du composant
       "prefix_Seeproc_Unite_controle" en fonction des param\`{e}tres nalu, ↩
           dim,
       coef et minaddr

       VI - Si minaddr vaut 1 alors
             |  Ecrire la declaration du composant "prefix_Seeproc_compteur"

       VII - Ecrire les signaux internes pour le routage des divers
       elements en fonction des param\`{e}tres nalu, dim, coef et minaddr

       VIII - Ecrire le routage du composant "prefix_Seeproc_Memoire_programme↩
           "

       XI - Ecrire le routage du composant "prefix_Seeproc_Program_counter"

       X - Ecrire le routage du composant "prefix_Seeproc_Unite_controle" en ↩
           fonction des param\`{e}tres nalu, dim,
       coef et minaddr

       XI - Si minaddr vaut 1 alors
             | Ecrire le routage du composant "prefix_Seeproc_compteur"
Fin
```

TABLE E.3 – Algorithme de création du SeeProc_Decoder

```
Parametre utilise: prefix
Debut

       I - Ouvrir/creer un fichier nomme "prefix_Seeproc_Program_counter.vhd"

       II - Ecrire le code VHDL du compteur programme
Fin
```

TABLE E.4 – Algorithme de création du compteur programme

```
Parametre utilise: prefix, minaddr
Debut
    Si minaddr vaut 1 alors
      |  I - Ouvrir/creer un fichier nomme "prefix_Seeproc_compteur.vhd"
      |
      |  II - Ecrire le programme VHDL du compteur programme
Fin
```

TABLE E.5 – Algorithme de création du module d'adressage

E.2 Éléments du chemin de données

```
Parametre utilise: prefix
Debut

       I - Ouvrir/creer un fichier nomme "prefix_Seeproc_reg.vhd"

       II - Ecrire le code VHDL du registre

Fin
```

TABLE E.6 – Algorithme de création du registre

```
Parametre utilise: prefix, nalu, dim, et imw
Debut

    Faire nalu fois

       I - Ouvrir/creer un fichier nomme "prefix_Seeproc_PG'j'.vhd"
       avec j = dim[nalu]

       II - Ecrire les port d entree/sortie du composant
       "prefix_Seeproc_PG'j'" en fonction des param\`{e}tres dim[j] et
       imw

       III - Ecrire la declaration composant "prefix_Seeproc_reg"

       IV - Ecrire les signaux internes de routage en fonction de
       dim[j] et de imw

       V - Ecrire le reseau de registre en fonction de dim[j] et de
       imw

Fin
```

TABLE E.7 – Algorithme de création des SPG

```
Parametre utilise: prefix, dim, nalu et minALUM
Debut
Faire nalu fois
       I - Ouvrir/creer un fichier nomme "prefix_seeproc_ALUM'j'.vhd"
        avec j indice d iteration.

       II - Ecrire les ports d entree/sortie de l ALUM'j' en fonction
        de la param\`{e}tre dim[j].

       III - Ecrire le "process" FD.
        Tester la presence des differentes operations de FD.
        Si presence alors
        |    Ecrire de l operation FD correspondante.


       IV - Ecrire le "process" FM.
        Tester la presence des differentes operations de FM.
        Si presence alors
        |    Ecrire de l operation FM correspondante.

       V - Ecrire le "process" FR.
        Tester la presence des differentes operations de FR.
        Si presence alors
        |   Ecrire de l operation FR correspondante.
Fin
```

TABLE E.8 – Algorithme de création des ALUMs

```
Parametre utilise: prefix, minCross, imw, nalu, dim et MinALU
Debut
       I - Ouvrir/creer un fichier nomme "prefix_Seeproc_DPR.vhd"

       II - Extraire la dimension maximale, dimmax, depuis le param\`{e}tre ↩
           dim

       II - Ecrire les port d entree/sortie du composant
       "prefix_Seeproc_DPR" en fonction des param\`{e}tres dim, dimmax et
       nalu

       IV - Ecrire la declaration composant
       "prefix_Seeproc_Crossbar" en fonction des param\`{e}tres nalu et
       dim

       V - Faire nalu fois
            | Ecrire la declaration du composant "prefix_Seeproc_ALUM'j'" en ↩
              fonction de j = indice d iteration et de dim[j]

       VI - Ecrire les signaux interne de routage en fonction de
       dimmax, nalu et dim

       VII - Ecrire le routage du composant "prefix_Seeproc_Crossbar"

       XI - Faire nalu fois
            | Ecrire le routage des composants
            "prefix_Seeproc_ALUM'j'" en fonction de j = indice d
            iteration
Fin
```

Table E.9 – Algorithme de création du SeeProc_DPR

Annexe F

Présentation de SystemC

SystemC est une bibliothèque du langage de programmation C++ [113] permettant de modéliser des systèmes complexes tels que des SoCs. Un des principaux avantages de SystemC est de pouvoir modéliser un système à différents niveaux d'abstraction. La version 2.1 de SystemC est définie par un standard IEEE [89] et une version *open source* est disponible sur le site de l'OSCI [1]. L'architecture logiciel de SystemC est présentée en Fig. F.1.

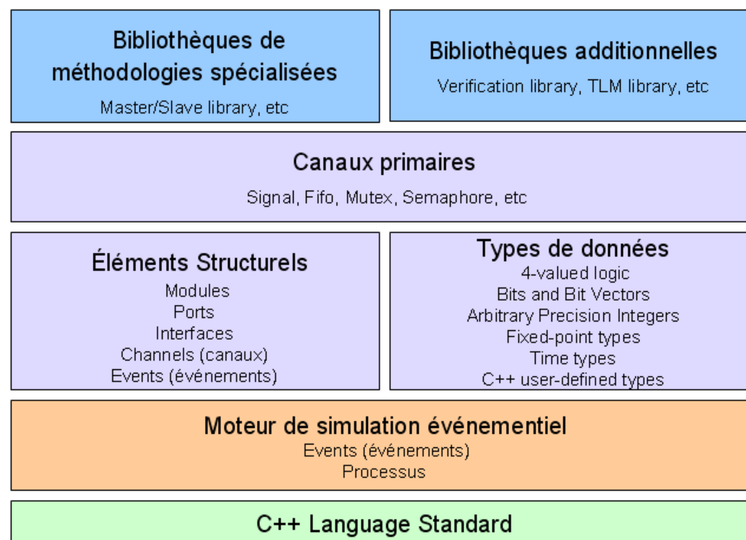


FIGURE F.1 – Architecture logiciel de SystemC

Le langage C++ offre une vitesse de simulation importante et permet une réutilisation grâce à l'aspect objet de sa programmation. SystemC apporte en plus la possibilité d'exécuter en parallèle des processus ainsi qu'une librairie pour

1. Open SystemC Initiative : http ://www. systemc.org.

la modélisation matérielle. Un modèle SystemC est donc un programme C++ qui peut être compilé afin d'obtenir un modèle exécutable simulant son comportement. Ainsi, la prise en main par un nouvel utilisateur est relativement accessible et l'utilisation d'outils existants en C++ possible.

Un composant est décrit à l'aide de la classe *sc_module*. Cette classe permet de définir les ports d'e/s ainsi que le comportement du composant. Les ports d'e/s sont également définis au sein d'une classe, *sc_port*, réunissant divers types de port dont les plus utilisés sont *sc_signal* et *sc_fifo*. Pour décrire le comportement d'un composant, trois types de processus sont disponibles :

Sc_Method :
- Fonctions appelées à chaque notification d'un événement dans leur liste de sensibilité,
- Exécutées en entier à chaque appel,
- La liste de sensibilité est construite avec la fonction *sensitive()*.

Sc_Thread :
- Lancés une fois au début de la simulation,
- Ce sont des boucles infinies qui sont mise en veille par la fonction *wait()*,
- Ils sont réveillés par l'intermédiaire de leur liste de sensibilité.

Sc_CThread : Semblables à Sc_Thread mais uniquement sensible à un front d'horloge, ce qui les rend très bien adaptés aux systèmes synchrones.

Exemple de Sc_method

```
#include "systemc.h"

SC_MODULE(and3)
{
sc_in<bool> A;
sc_in<bool> B;
sc_in<bool> C;
sc_out<bool> D;

void compute_and()
{
D = A & B & C;
}

SC_CTOR(and3)
{
SC_METHOD(compute_and);
sensitive << A;
sensitive << B;
sensitive << C;
};
};
```

Exemple de Sc_thread

```
#include "systemc.h"

SC_MODULE(tri)
{
sc_in<bool> appel_pieton;
sc_in<bool> clk;
sc_out<sc_uint<2>> feux;
void compute_feu()
{
while(true){
feux=0; //feux vert
wait(appel_pieton)

for(int i=0;i<50;i++)
wait();
feux=1; //feux orange

for(int i=0;i<50;i++)
wait();
feux=2; //feux rouge

for(int i=0;i<5000;i++)
wait();
}
}

SC_CTOR(tri)
{
SC_THREAD(compute_feu);
sensitive << clk.pos();
};
};
```

Exemple de Sc_cthread

```
#include "systemc.h"

SC_MODULE(tri)
{
sc_in<bool> clk;
sc_out<sc_uint<2> > feux;
void compute_feu()
{
while(true){
feux=0; //feux vert
wait(1000)

feux=1; //feux orange
wait(50);

feux=2; //feux rouge
wait(1000);
}
}

SC_CTOR(tri)
{
SC_CTHREAD(compute_feu);
sensitive << clk.pos();
};
};
```

La bibliothèque SystemC intègre également un simulateur événementiel rapide. La classe *Sc_time* permettant de mesurer le temps est composée d'une partie numérique et d'une partie spécifiant son unité. Ainsi, la ligne de code *sc_time ctrl(10, SC_US) ;* permet de définir un signal *ctrl* de période 10μs. La classe *Sc_clock* permet de définir un signal d'horloge en spécifiant le nom associé à ce signal, la période, le rapport cyclique ou encore l'instant de départ. Enfin, la classe *Sc_start* permet à l'utilisateur de démarrer la simulation ainsi que d'en définir la durée.

Afin d'enregistrer des simulations, via des chronogrammes, SystemC dispose de la classe *Sc_trace* qui permet de générer des fichiers au format VCD. Le programme ci-dessus explique comment fonctionne cette classe :

Exemple de création de chronogrammes avec SystemC

```
sc_trace_file *Chrono = sc_create_vcd_trace_file(”Exemple_Chrono) ;
sc_trace(Chrono, clk50, ”Horloge”) ;
sc_trace(Chrono, A, ”Entree A”) ;
sc_trace(Chrono, B, ”Entree B”) ;
sc_trace(Chrono, C, ”Entree C”) ;
sc_trace(Chrono, D, ”Resultat D”) ;

sc_start(50, SC_US) ;

sc_close_vcd_trace_file(Chrono) ;
```

Annexe G

Présentation du processeur SeeCore

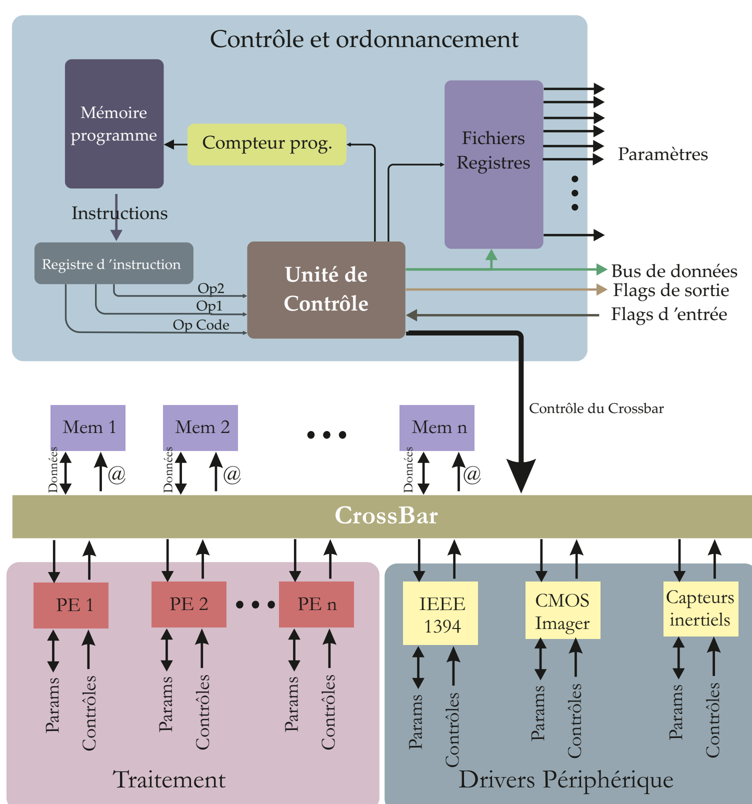


FIGURE G.1 – Schéma synoptique du SeeCORE.

Dans le cadre de ses travaux de thèse, F. Dias [20] a développé un processeur dédié à la gestion des divers composant d'une caméra intelligente à base de FPGAs. Ce processeur, baptisé SeeCORE, est présenté en Fig. G.1. Le but de ce

processeur est de donner à un utilisateur non avertit une certaine transparence des aspects matériels lors de l'utilisation d'une caméra intelligente.

Ce processeur est construit autour d'un cœur de processeur RISC avec certaines fonctionnalités supplémentaires afin de répondre à des exigences de gestion en parallèle des divers modules d'une caméra intelligente et de permettre une gestion souple du parcours des données entre les capteurs, les traitements et l'interface de communication.

Écris en VHDL, des pilotes périphériques sont utilisés pour contrôler, via les instructions du processeur RISC, les composants d'une caméra intelligente comme par exemple la rétine, les capteurs inertiels, les banques mémoire ou encore le dispositif de communication. Ces pilotes communiquent via un réseau d'interconnexion point à point et un protocole de communication de type *handshaking*.

La topologie utilisée au sein du SeeCORE est un crossbar complet. Comme explicité en Sec. 3.1.1, ce dispositif de communication permet l'exploitation simultanée de plusieurs blocs mémoires, ainsi que de redéfinir le chemin de données de façon dynamique, sans faire appel à une reconfiguration du FPGA. Dans le cadre du processeur SeeCORE, le crossbar permet d'implantation des stratégies telle que le *memory swapping*.

La méthodologie de développement proposée s'appuie sur quatre éléments :
- Un langage de programmation du type assembleur, reposant sur un jeu d'instructions réduit,
- Un outil assembleur, capable de transformer le code textuel du programme assembleur en code machine binaire,
- Un modèle virtuel d'une architecture de processeur RISC, capable de décoder et exécuter les instructions assembleur. Ce modèle est écrit en langage SpecC[112],
- Une version de ce même processeur, décrit en langage VHDL.

Cette méthodologie comporte 2 niveau de développement comme le montre la Fig. G.2. Le premier niveau de développement est une phase *off-line* : une fois en possession du code binaire de l'application, généré par l'outil assembleur depuis un programme écrit avec l'assembleur personnalisé, celui-ci peut être exécuté en simulation par la version virtuelle du processeur écrite en SpecC. Le modèle SpecC est ensuite compilé. Le résultat obtenu est une version exécutable du modèle virtuel du processeur, capable d'exécuter les instructions du programme assembleur. En fonction des résultats de simulation, le programmeur pourra déboguer ou optimiser son programme.

La seconde phase de développement est le niveau *temps réel* : après validation du code assembleur par l'utilisateur grâce au modèle SpecC, l'outil assembleur génère un fichier *bit-stream*. Ce fichier va permettre de configurer la mémoire programme, à la compilation, de la version VHDL du processeur et ainsi permettre l'exécution temps-réel de l'application.

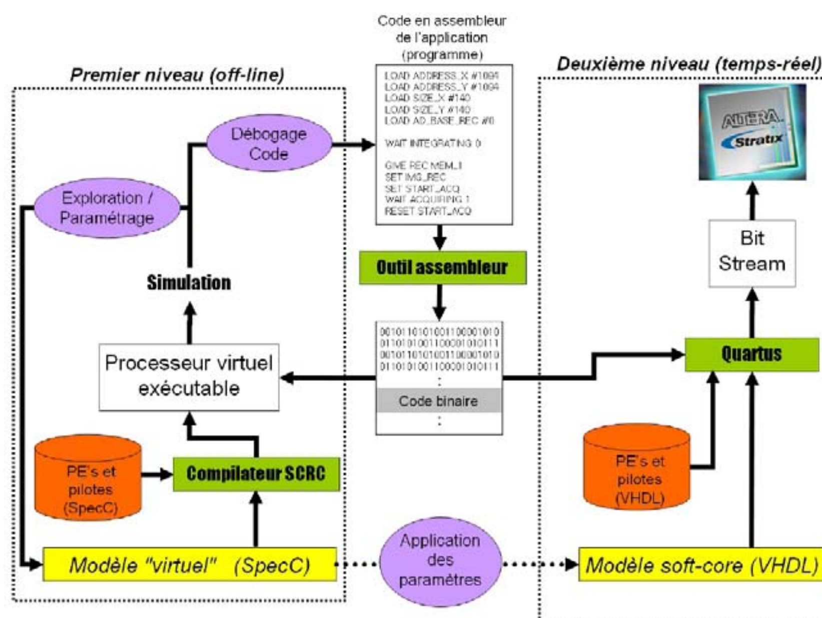


FIGURE G.2 – Schéma synoptique du flot d'implémentation du processeur SeeCORE

Bibliographie

[1] NI 17xx Smart Cameras National Instruments. *http ://www.ni.com/pdf/products/us/cat_ni_1742.pdf.*

[2] Accelerator Series FPGAsACT3 Family Actel Corporation. *www.ee.pdx.edu/˜mperkows/temp/May13/PE014.XC6200Ahmad.pdf*, 1997.

[3] ProASIC3 flash family FPGAs Actel Corporation. *http ://www.actel.com/documents/PA3_DS.pdf*, 2005.

[4] Stratix III device handbook Actel Corporation. *http ://www.altera.com/literature/hb/stx3/stratix3_handbook.pdf*, 2006.

[5] Alacron. *http ://www.alacron.com.*

[6] FPGA AT94KAL Series Atmel Corporation. *www.atmel.com/atmel/acrobat/doc1138.pdf*, 2008.

[7] P. Chalimbaud. *Conception d'une plateforme d'implémentation matérielle dédiée aux systèmes de vision active basés sur un imageur CMOS.* PhD thesis, Universite Blaise-Pascal, 2004.

[8] S. Collange. Analyse de l'architecture gpu tesla. 2010.

[9] B. Cope. Implementation of 2d convolution on fpga, gpu and cpu. *Technical report, Department of Electrical and Electronic Engineering, Imperial College, London*, 2006.

[10] Altera Corporation. *Avalon Interface Specifications*, 2011.

[11] Intel Corporation. *http ://www.intel.com.*

[12] LSI Logic Corporation. *LSI, L64240 Multibit Filter (MFIR).*

[13] Xilinx Corporation. *XC6200 Field Programmable Gate Arrays*, 1996.

[14] Xilinx Corporation. *Two flows for partial reconfiguration : Module based or small bit manipulations*, application note : virtex, virtex-ii families edition, 2002.

[15] M. Darouich. *REEFS : Une architecture reconfigurable pour la stéréovision embarquée en contexte temps-réel.* PhD thesis, UNIVERSITÉ DE RENNES 1, 2010.

[16] NXP Microcontroller Corter M4 Datasheet. *http ://ics.nxp.com/products/lpc4000/datasheet/lpc4310.lpc4320.lpc4330.lpc4350.pdf*, 2011.

[17] R. David. *Architecture reconfigurable dynamiquement pour applications mobiles*. PhD thesis, Universite de Rennes 1, 2003.

[18] Armor Langage de description darchitecture. *http ://www.irisa.fr/cosi/SEMINAIRE/transparents/Armor.pdf*.

[19] P. de la Hamette et G. Tröster. Fingermouse - architecture of an asic-based mobile stereovision smart camera. *IEEE International Symposium on Wearable Computers*, 2006.

[20] F. Dias Real de Oliveira. *Conception d'une méthodologie d'implémentation d'applications de vision dans une plateforme hétérogène de type Smart Camera*. PhD thesis, Universite Blaise-Pascal, 2010.

[21] Séquence d'images Yosemite. *http ://www.cs.brown.edu/˜black/images.html*.

[22] ANR Architectures du futur. *https ://roma.irisa.fr/*.

[23] P. Dudek. Implementation of simd vision chip with 128x128 array of analogue processing elements. *The International Symposium on Circuits and Systems (ISCAS)*, 2005.

[24] S. Kyo et al. Efficient implementation of image processing algorithms on linear processor arrays using the data parallel language 1dc. *IAPR Workshop on Machine Vision and Applications*, 1996.

[25] S.A.H. Al Umairy et A.S. van Amesfoort et I.D. Setija et M.C. van Beurden et H.J. Sips. On the use of small 2d convolutions on gpus. *A4MMC 2010 - 1st Workshop on Applications for Multi and Many Core Processors*, 2010.

[26] J. Chase et B. Nelson et J. Bodily et Z. Wei et D. Lee. Real-time optical flow calculations on fpga and gpu architectures : A comparison study. *International Symposium on Field-Programmable Custom Computing Machines*, 2008.

[27] W. Wolf et B. Ozer et T. Lu. Smart cameras as embedded systems. *IEEE Computer*, 2002.

[28] J.C. Heudin et C. Panetto. *Les Architectures RISC - Théorie et pratique des ordinateurs à jeu d'instructions réduits*. Editions Dunod, 1990.

[29] R. Enzler et C. Plessl et M. Platzner. Co-simulation of a hybrid multi-context architecture. *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2003.

[30] C. Farabet et C. Poulet et Y. LeCun. An fpga-based stream processor for embedded real-time vision with convolutional networks. *Fifth IEEE Workshop on Embedded Computer Vision (ECV'09)*, 2009.

[31] G.Borriello et C.Ebeling et S.Hauck et S.Burns. The triptych fpga architecture. *IEEE Trans. on VLSI Systems*, 1995.

[32] Fpga et cpld xilinx. Site web. http ://www.xilinx.com.

[33] J. Dubois et D. Ginhac et M. Paindavoine et B. Heyrman. A 10 000 fps cmos sensor with massively parallel image processing. *IEEE Journal of solid-state circuit*, 2008.

[34] S.A. Guccione et D. Levi et P. Sundararajan. Jbits : A java-based interface for reconfigurable computing. *In Proceedings of the 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference*, 2000.

[35] S. Hengstler et D. Prashanth et S. Fong et H. Aghajan. Mesheye : A hybrid-resolution smart camera mote for applications in distributed intelligent surveillance. *In International Symposium on Information Processing in Sensor Networks (IPSN)*, 2007.

[36] A. M. Adario et E. L. Roehe et S. Bampi. Dynamically reconfigurable architecture for image processor applications. *36th Design Automation Conference*, 1999.

[37] V. Lahtinen et E. Salminen et K. Kuusilinna et T. Hamalainen. Comparison of synthesized bus and crossbar interconnection architectures. *Proceedings of the 2003 international symposium on circuits and systems*, 2003.

[38] F. Röbler et E. Tejada et T. Fangmeier et T. Ertl et M. Knauff. Gpu-based multi-volume rendering for the visualization of functional brain images. *IN PROCEEDINGS OF SIMVIS 2006*, 2006.

[39] F. Pelissier et F. Berry. Design of a real-time embedded stereo smart camera. *Advanced Concepts for Intelligent Vision Systems*, 2010.

[40] F. Pelissier et F. Berry. Phd forum : Biseemos : A fast embedded stereo smart camera. *Distributed Smart Cameras (ICDSC), 2011 Fifth ACM/IEEE International Conference on*, 2011.

[41] F. Dias Real et F. Berry et F. Marmoiton et J. Serot. A configurable window-based processing element for image processing. *Smart Cameras IAPR Machnie, Vision and Application (MVA'07)*, 2007.

[42] N. Roudel et F. Berry et J. Sérot et L. Eck. Hardware implementation of a real time lucas and kanade optical flow. *Conference on Design and Architectures for Signal and Image Processing*, 2009.

[43] Y. Ni et F. Devos et E. Vaillant. Histogram-equalization based adaptive image sensor for real-time vision. *IEEE Journal of Solid-State Circuits*, 1997.

[44] V. Baumgarte et F. May et A. Nückel et M. Vorbach et M. Weinhardt. Pact xpp - a self-reconfigurable data processing architecture. *The Journal of Supercomputin*, 2003.

[45] T.J. Todman et G.A. constantinides et S.J.E. Wilton et W. Luk et P.Y.K. Cheung. Reconfigurable computing : architectures and design methods. *IEEE proceedings of Comput. Digit. Tech*, 2005.

[46] M. R. Boschetti et I. S. Silva et S. Bampi. A run-time reconfigurable datapath architecture for image processing application. *Design, Automation and Test in Europe Conference and Exhibition*, 2004.

[47] M. Bramberger et J. Brunner et B. Rinner et H. Schwabach. Real-time video analysis on an embedded smart camera for traffic surveillance. *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2004.

[48] R. Mosqueron et J. Dubois et M. Paindavoine. High-speed smart camera with high resolution. *EURASIP Journal on Embedded Systems*, 2007.

[49] S.N. Sinha et J. Frahm et M. Pollefeys et Y. Genc. Gpu-based video feature tracking and matching. *In Workshop on Edge Computing Using New Commodity Architectures*, 2006.

[50] J. Kim et J. Kong et S. Suh et M. Lee et J. Shin et H.B. Park et C.A. Choi. A low power analogue cmos vision chip for edge detection using electronic switches. *ETRI Journal*, 2005.

[51] J.R. Hauser et J. Wawrzybek. Garp : A mips processor with a reconfigurable coprocessor. *5th IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[52] O. Sentieys et J.P. Diguet et J.L. Philippe. Gaut : A high level synthesis tool dedicated to real time signal processing applications. *In European Design Automation Conference*, 2000.

[53] S. Muralix et L. Beniniz et G. De Micheli. An application-specific design methodology for on-chip crossbar generation. *Computer-Aided Design of Integrated Circuits and Systems*, 2007.

[54] R. Johansson et L. Lindgren et J. Melander et G. Möller. A multi-resolution 100 gops 4 gpixels/s programmable cmos image sensor for machine vision. *IEEE WORKSHOP ON CHARGE COUPLED DEVICES AND ADVANCED IMAGE SENSORS*, 2003.

[55] C. Torres-Huitzil et M. Arias-Estrada. Fpga-based configurable systolic architecture for window-based image processing. *EURASIP Journal on Applied Signal Processing*, 2005.

[56] S.C. Woo et M. Ohara et E. Torrie et J. Pal Singh et A. Gupta. The splash-2 programs : Characterization and methodological considerations. *22nd International Symposium on Computer Architecture*, 1995.

[57] T. Röwekamp et M. Platzner et Liliane Peters. Specialized architectures for optical flow computation : A performance comparison of asic, dsp, and

multidsp. *8th International Conference on Signal Processing Applications & Technology*, 1997.

[58] C Harris et M Stephens. A combined corner and edge detector. *Alvey vision conference*, 1988.

[59] Y. Zhang et M.J. Irwin. Power and performance comparison of crossbars and buses as on-chip interconnect structures. *Asilomar Conference on Signal, Systems and Computers*, 1999.

[60] S. Pasricha et N. Dutt et M. Ben-Romdhane. Constraint-driven bus matrix synthesis for mpsoc. *Asia and South Pacific Conference on Design Automation*, 2006.

[61] F. Dias et P. Chalimbaud et F. Berry et J. Serot et F. Marmoiton. Embedded early vision systems : implementation proposal and hardware architecture. *In Cognitive Systems with Interactive Sensors (COGIS)*, 2006.

[62] P. Pavan et R. Bez et P. Olivo et E. Zanoni. Flash memory cells-an overview. *Proceedings of the IEEE, vol. 85*, 1997.

[63] J. McCarthy et R. Brayton et D. Edwards et al. *Modélisation et simulation multi-niveaux avec le langage SystemC*, 1960.

[64] K. Guttag et R. J. Gove et J. R. Van Ake. A single chip multiprocessor for multimedia : the mvp. *IEEE Computer Graphics Application*, 1992.

[65] C. Basoglu et R. J. Gove et K. Kojima et J. O'Donnell. A single-chip processor for media applications : the map1000. *International Journal of Imaging Systems and Technology*, 1999.

[66] S. Donthi et R.L. Haggard. A survey of dynamically reconfigurable fpga devices. *Proceedings of the 35th Southeastern Symposium on System Theory*, 2003.

[67] J.V.D. Horst et R.V. Leeuwen et H. Broers et R. Kleihorst et P. Jonker. A real-time stereo smartcam, using fpga, simd and vliw. 2006.

[68] T. Komuro et S. Kagami et M. Ishikawa. A dynamically reconfigurable simd processor for a vision chip. *IEEE Journal of Solid-State Circuits*, 2004.

[69] J. Kong et S. Kim et D. Sung et J. Shin. A 160x120 light-adaptative cmos vision chip for edge detection based on a retinal structure using a saturating resistive network. *ETRI Journal*, 2007.

[70] Y. Fujita et S. Kyo et N. Yamashita et S. Okazaki. A 10 gips simd processor for pc-based real-time vision applications : Architecture, algorithm implementation and language support. *Computer Architectures for Machine Perception*, 1997.

[71] S. Kyo et 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.

[72] A. Shoa et S. Shirani. Run-time reconfigurable system for digital signal processing applications : A survey. *Journal of VLSI Signal Processing*, 2005.

[73] P. Chang et T. Camus et R. Mandelbaum. Stereo-based vision system for automotive imminent collision detection. *In Intelligent Vehicles Symposium, 2004, IEEE*, 2004.

[74] S. Asano et T. Maruyama et Y. Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. *The International Conference on Field Programmable Logic and Applications (FPL)*, 2009.

[75] F. Charot et V. Messe. A flexible code generation framework for the design of application specific programmable processors. *In International workshop on Hardware/Software Codesign*, 1999.

[76] R. Tessier et W. Burleson. Reconfigurable computing and digital signal processing : a survey. *J. VLSI Signal Process*, 2001.

[77] Q. Lin et W. Miao et W. Zhang et Q. Fu et N. Wu. A 1,000 frames/s programmable vision chip with variable resolution and row-pixel-mixed parallel image processors. *Sensors 2009*, 2009.

[78] S. Chen et W. Tang et E. Culurciello. A 64x64 pixels uwb wireless temporal-differnce digital image sensor. *International Symposium on Circuits and Systems (ISCAS)*, 2010.

[79] J. Marzat et Y. Dumortier et A. Ducrot. Real-time dense and accurate parallel optical flow using cuda. *International Conferences in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2009.

[80] W. Lee et Y. Kim et R. J. Gove et C. J. Read. Mediastation 5000 : integrating video and audio. *IEEE Multimedia*, 1994.

[81] R. Etienne-Cummings et Z. Kalayjian et D. Cai. A programmable focal-plane mimd image processor chip. *IEEE Journal of Solid-State Circuits*, 2001.

[82] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput., Vol. C-21*, 1972.

[83] cpld et asic altera. Site web. Fpga. http ://www.altera.com.

[84] J. Galy G. Sassatelli et L. Torres, G. Cambon. Architectures reconfigurables dynamiquement pour les systèmes embarqués. *GRETSI, Groupe dEtudes du Traitement du Signal et des Images*, 2001.

[85] B. Le GAL. *Modélisation et simulation multi-niveaux avec le langage SystemC.*

[86] J. Garcia-Lamont. Analogue cmos prototype vision chip with prewitt edge processing. *Analog Integrated Circuits and Signal Processing*, 2011.

[87] DECchip 21064 Evaluation Board User's Guide. *http ://www.compaq.com/cpq-alphaserver/technology/literature/eb64ug.pdf*.

[88] R. Hartenstein. A decade of reconfigurable computing : a visionary retrospective. *Proceedings of the conference on Design, automation and test in Europe*, 2001.

[89] Open SystemC Initiative : http ://www.systemc.org/. *SystemC v2.1 Language Reference Manual (IEEE Std 1666-2005)*.

[90] IBM. *IBM CoreConnect bus cores*, 2006.

[91] IEEE. *IEEE Standard VHDL Language Reference Manual*, 1076-2000 edition, 2000.

[92] IEEE. *IEEE Standard VERILOG Language Reference Manual*, 1364-2001 edition, 2001.

[93] Open SystemC Initiative. *http ://www.systemc.org*.

[94] T. Mason et D. Brown J. R. Levine. *Lex & yacc*. O'Reilly & Associates, Inc, 1992.

[95] S. Wong et S. Vassiliadis JY. Hur et T. Stevanov. Systematic customization of on-chip crossbar interconnects. *Springer Berlin / Heidelberg*, 2007.

[96] J. Lallet. *Mozaïc : plate-forme générique de modélisation et de conception d'architectures reconfigurables dynamiquement*. PhD thesis, UNIVERSITÉ DE RENNES 1, 2008.

[97] S.-W Liao. *SUIF Explorer : an Interactive and Interprocedural Parallelizer*. PhD thesis, Computer Systems Laboratory, Stanford University, 2000.

[98] The Cimg library. *http ://cimg.sourceforge.net/*.

[99] W.J. MacLean. An evaluation of the suitability of fpgas for embedded vision systems. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPRW'05)*, 2005.

[100] 3DNow! Technology Manual. *http ://support.amd.com/us/Embedded_TechDocs/21928.pdf*, 2000.

[101] OMAP 4 mobile applications platform. *http ://www.ti.com/lit/ml/swpt034b/swpt034b.pdf*, 2011.

[102] Site officiel de Bison. *http ://www.gnu.org/software/bison/*.

[103] Site officiel de Flex. *http ://flex.sourceforge.net/*.

[104] S. Phillips. Victoriafalls : Scaling highly-threaded processor cores. *Proceedins HotChips*, 2007.

[105] ARM PrimeXsys Platform. *http ://www.arm.com*.

[106] IBM Cell Project. *http ://www.research.ibm.com/cell*.

[107] The Gecos (Generic Compiler Suite) project. *http ://gecos.gforge.inria.fr/doku.php*, 2004.

[108] M. Raulet. *Optimisations Mémoire dans la Méthodologie AAA pour Code Embarqué sur Architectures Parallèles.* PhD thesis, INSA de Rennes, 2006.

[109] Zarlink Semiconductor. *http ://www.zarlink.com/zarlink/hs/82_PDSP16488AMA.htm.*

[110] J. Sérot. *Cours : Une petite introduction à Lex et YAcc*, 2011.

[111] Inc. site web PACT XPP Technologies. *http ://www.pactxpp.com/.*

[112] The specc system. Site web. *http ://www.cecs.uci.edu/˜specc/.*

[113] B. Stroustrup. *The C++ programming language.* Special Edition. Addison-Wesley Longman Publishing Co., 1997.

[114] Blue Wave Systems. *http ://www.bluewavesystems.com/.*

[115] Chameleon Systems Europe : Chameleon systems europe. *http ://www.chameleonsystems.com/*, 2004.

[116] Z.A. Talib. A real-time 256x256 pixel parallel image processing system. Master's thesis, Massachusetts Institute of Technology, 1997.

[117] Motorolas AltiVec Technology. *http ://www.iele.polsl.pl/elenota/Motorola/altivecwp.pdf*, 1997.

[118] PENTIUM PROCESSOR WITH MMX TECHNOLOGY. *http ://download.intel.com/design/archives/processors/mmx/docs/24318504.pdf*, 1997.

[119] Introduction to the Quartus II Software. *http ://www.altera.com/literature/manual/archives/intro_to_quartus2.pdf*, 2010.

[120] Jan-Willem van de Waerdt. *The TM3270 Media-processor.* PhD thesis, 2006.

[121] G. van der Wal et M. Hansen et M. Piacentino. The acadia vision processor. *International Workshop on Computer Architecture for Machine Perception*, 2000.

[122] A.H. Veen. Dataflow machine architecture. *ACM Computing Surveys (CSUR)*, 1986.

[123] JTAG white paper. *http ://files.sjtag.org/WhitePaper/SJTAG_white_paper_0.4.pdf*, 2005.

[124] Virtex-4 family overview Xilinx. *http ://direct.xilinx.com/bvdocs/publications/ds112.pdf*, 2005.

Résumé

Les travaux présentés dans ce manuscrit proposent une architecture de processeur à chemin de données reconfigurable (PCDR) dédiée aux traitements d'images bas niveau. Afin de répondre aux exigences de ce domaine de traitements, le processeur, baptisé *SeeProc* et basé sur une architecture RISC, intègre dans son chemin de données des unités de calcul spécifiquement dédiées au traitement de données pixeliques sous forme matricielle. Ces unités pouvent être configurées en nombre et en fonctionnalité en fonction de l'application visée. La topologie d'interconnexion du chemin de données est assurée dynamiquement via un dispositif de type crossbar. De plus, pour rendre la programmation de SeeProc accessible à des utilisateurs n'ayant pas de notions d'électronique numérique, un langage assembleur dédié et une méthodologie d'optimisation ont été développés.

Mots-clefs : Vision par ordinateur, smart camera, FPGA, traitement d'images, SOPC, processeur à chemin de données reconfigurable

Abstract

The work presented in this manuscript suggest an architecture of a reconfigurable datapath processor (RDP) dedicated to low-level image processing. To meet the requirements of this field, the processor, called *SeeProc* and based on a RISC architecture, includes in its datapath customs processing elements specifically dedicated to the computation of image data in matrix form. These units can be configured in number and functionality depending on the application. The datapath interconnection topology is provided dynamically using a crossbar device. In addition, to make the programming accessible to users with no knowledge of electronics digital, a dedicated assembly language and an optimization methodology have been developed.

Keywords : Computer vision, smart camera, FPGA, image processing, SOPC, reconfigurable datapath processor