Année 2014

THÈSE

présenté à

L'UFR DES SCIENCES ET TECHNIQUES
DE L'UNIVERSITÉ DE FRANCHE-COMTÉ

pour obtenir le

GRADE DE DOCTEUR
DE L'UNIVERSITÉ DE FRANCHE-COMTÉ
spécialité Sciences Pour l'Ingénieur

GÉNÉRATEUR DE COPROCESSEUR POUR LE TRAITEMENT DE DONNÉES EN FLUX (VIDÉO OU SIMILAIRE) SUR FPGA.

par

Gwenhaël GOAVEC-MEROU

Soutenance le 26 novembre 2014 devant la commission d'examen :

Directeur de thèse	M. LENCZNER	Professeur, FEMTO-SR TF, Besançon
Co-directeur de thèse	R. COUTURIER	Professeur, FEMTO-ST DISC, Belfort
Rapporteurs	G. GOGNIAT	Professeur, Université de Bretagne Sud , Lorient
	S. WEBER	Professeur, Université de Lorraine, Nancy
Examinateurs	M. PAINDAVOINE	Professeur, Université de Bourgogne, Dijon
	J.-M. FRIEDT	Ingénieur HDR, Université de Franche-Comté, Besançon
	N. COLOMBAIN	Ingénieur, Armadeus Systems, Mulhouse
	C. ELIGIO CALOSSO	chercheur, INRIM, Turin

Remerciements

Ces travaux ont été menés à l'institut FEMTO-ST dans le département temps-fréquence et plus précisément au sein de l'équipe COSYMA, puis de l'équipe OHMS. À ce titre, je tiens à remercier les directeurs de l'institut qui se sont succédés, MM. Michel De Labachelerie et Nicolas Chaillet qui m'ont acceuilli au sein de FEMTO-ST.

Mes remerciements vont également à la société Armadeus Systems et à son équipe, pour avoir permis l'encadrement et la réalisation de cette thèse CIFRE.

Je remercie les membres de mon jury qui m'ont fait l'honneur de juger mes travaux. Dans un premier temps, je remercie MM. Guy Gogniat et Serge Weber qui ont eu la lourde tâche d'être rapporteurs de ce manuscrit. Merci à vous pour vos commentaires constructifs. Je remercie particulièrement M. Claudio Eligio Calosso qui a fait le déplacement depuis Turin pour assister à ma soutenance. Je remercie également M. Michel Paindavoine pour avoir accepté de présider ce jury et M. Jean-Michel Friedt d'avoir participé à la soutenance de mes travaux.

je remercie mes directeurs de thèse MM. Michel Lenczner et Raphaël Couturier ainsi que mon encadrant M. Stéphane Domas pour leurs remarques et conseils tant sur le point théorique qu'applicatif.

Je tiens également à remercier mon second encadrant M. Nicolas Colombain ainsi que M. Fabien Marteau, de Armadeus Systems, pour leurs conseils du point de vue matériel et FPGA. Et M. Julien Boibessot pour ses remarques sur la partie logicielle (applications et pilotes).

Je remercie à nouveau M. Jean-Michel Friedt pour ses remarques et les discussions qui m'ont permis de débloquer de nombreux problèmes et de réaliser des démonstrations dans le domaine de la vidéo ainsi que du Temps-Fréquence. Mais également pour m'avoir permis de réaliser un stage sur microcontrôleur qui a été l'élément déclencheur de mon orientation actuelle.

Je remercie également MM. Enrico Rubiola et Pierre-Yves Bourgeois qui, dans le cadre du projet Oscillateur IMP, m'ont permis d'étendre mes travaux, et mes connaissances, au domaine de la métrologie Temps-Fréquence.

Plus globalement, je tiens à remercier l'ensemble des personnes du département temps-fréquence (Emile, Bruno, David, Marc, Vincent, Serge, Benoit, Yann, ...) pour l'accueil, l'aide et l'atmosphére.

Table des matières

Remerciements iii

I Introduction 1

Introduction générale 3

1 Présentation 5
1.1 Techniques d'accélération de traitements . . . 5
1.2 Calcul sur FPGA . . . 6
1.3 Méthodes de développement alternatives . . . 8
1.4 Positionnement de nos travaux . . . 9
1.5 Environnement matériel et logiciel . . . 10
1.6 Application au traitement d'images . . . 11
1.6.1 Classification par zone de traitement . . . 12
1.6.2 Types d'entrées et de sorties . . . 12
1.6.3 Types de traitements . . . 13
1.7 Application au traitement de signaux radio-fréquence . . . 14
1.7.1 Classification par zone de traitement . . . 14
1.7.2 Types d'entrées et de sorties . . . 15
1.8 Spécification de CoGen . . . 15
1.9 Bilan théorique . . . 16

II CoGen 21

2 Description de l'outil 23
2.1 Chaînes et Blocs . . . 24
2.1.1 Chaînes . . . 24
2.1.2 Blocs . . . 25
2.1.2.1 Typage des blocs . . . 26
2.1.2.2 Interfaces des blocs . . . 27
2.2 Caractéristiques temporelles . . . 27
2.2.1 Expansion des motifs . . . 30

2.2.1.1 Remplacement des variables par leur valeur 30
2.2.1.2 Génération de la liste 31
2.2.2 Retard dans la production de données 31
2.2.2.1 Exemple 31
2.3 Description des composants 31
2.3.1 Caractéristiques globales des fichiers XML 32
2.3.2 Description du matériel 33
2.3.2.1 Description d'une plate-forme 33
2.3.2.2 Description d'un FPGA 34
2.3.3 Informations communes à plusieurs types de blocs 34
2.3.3.1 Interface maître et esclave 34
2.3.3.2 Informations relatives aux caractéristiques temporelles d'un bloc 37
2.3.3.3 Utilisation des ressources 38
2.3.4 Configuration d'un bloc générique 38
2.3.4.1 Paramètres d'un bloc 38
2.3.4.2 Liste des implémentations pour un bloc donné 39
2.4 Analyse des chaînes 39
2.4.1 Analyse statique 41
2.4.1.1 Blocs admissibles 41
2.4.1.2 Cohérence des blocs 41
2.4.1.3 Cohérence des branches parallèles 41
2.4.2 Analyse temporelle et comportementale 42
2.4.2.1 Calculs des latences et traitement des branches parallèles . . 43
2.4.2.2 Validation de l'acceptance des blocs 44
2.4.2.3 Construction de la représentation du flux de sortie à partir du flux d'entrée et de l'information de sortance du bloc 47
2.5 Entrée utilisateur 50
2.6 Validation 51
2.6.1 Implémentations de l'algorithme 52
2.6.2 Résultats 55
2.6.2.1 Première chaîne 56
2.6.2.2 Seconde chaîne 57

3 Intégration et qualification de composants externes : cas des RAMs 59
3.1 Présentation du contrôleur de RAM de Xilinx 60
3.1.1 Interface de communication 60
3.1.1.1 Interfaces de lecture et d'écriture 61
3.1.1.2 Interface de commande pour l'accès physique à la RAM . . . 61
3.1.1.3 Arbitrage 61
3.1.2 Utilisation 62
3.1.2.1 Écriture 62
3.1.2.2 Lecture 63
3.1.3 Bilan de la communication avec le contrôleur de RAM 64

3.2 Estimation du comportement temporel de la RAM et du contrôleur 64
3.2.1 Caractéristiques temporelles en écriture 65
3.2.1.1 Simulation avec un seul utilisateur 65
3.2.1.2 Simulation avec deux utilisateurs 65
3.2.1.3 Cas de plus de deux utilisateurs 66
3.2.2 Caractéristiques temporelles en lecture 66
3.2.2.1 Simulation avec un seul utilisateur 66
3.2.2.2 Cas de plus d'un utilisateur 67
3.2.3 Bilan de la qualification du comportement temporel de la RAM 67
3.3 Ajout d'une couche d'abstraction entre le contrôleur et les blocs 68
3.3.1 Interface de communication et paramètres de configuration 69
3.3.1.1 Comportement du client pour l'écriture 70
3.3.1.2 Comportement du client pour la lecture 71
3.4 Bilan de l'utilisation d'une RAM externe et intégration dans CoGen 72
3.5 Conclusion 74

III Applications 77

4 Traitement de signaux 2D : application à la vidéo 79
4.1 CSI 80
4.1.1 Présentation du protocole 80
4.1.2 Implémentation 81
4.1.3 Intégration dans CoGen 81
4.1.4 Démonstration 82
4.2 Caméra Photonfocus 83
4.2.1 Implémentation 85
4.2.2 Désérialisation 85
4.2.2.1 Composant FIFO et génération du flux compatible CoGen . . 85
4.2.3 Description du bloc cameralink 86
4.2.4 Démonstration du fonctionnement de la caméra Photonfocus 87
4.3 Ajout d'un bloc de traitement dans la chaîne : filtre de seuillage 88
4.3.1 Implémentation 88
4.4 Bloc de traitements : filtre de détection de contours 90
4.4.1 Implémentation 92
4.4.1.1 Stockage des données dans les RAMs 92
4.4.1.2 Récupération des données pour utilisation 92
4.4.1.3 Intégration dans CoGen 92
4.5 Caractérisation des modes propres d'un diapason par méthode optique 93
4.5.1 Implémentation 97
4.5.1.1 Première implémentation 98
4.5.1.2 Seconde implémentation 100
4.5.2 Résultat des traitements 101
4.5.3 Analyse par CoGen 102

4.6 Conclusion 103

5 Traitement du signal 1D : application aux traitements de signaux radio-fréquence 105

5.1 Convertisseur analogique-numérique HMCAD1511 107
5.1.1 Implémentation 107
5.1.2 Contribution à CoGen 108
5.1.3 Validation basique 109
5.2 Mise en œuvre simple : diapason par RADAR 109
5.2.1 Principe de fonctionnement 109
5.2.2 Implémentation 112
5.2.3 Résultats 113
5.3 Acquisition radio-fréquence et traitement par GNURadio 114
5.3.1 Semtech SX1255 114
5.3.2 GNURadio 115
5.3.2.1 Source GNURadio 118
5.3.3 Application de démonstration 119
5.3.3.1 Principe du bloc terminal avec implémentation d'un double tampon 119
5.3.3.2 Pilote pour le transfert depuis le FPGA vers la source GNURadio 120
5.3.4 Résultats 122
5.4 Traitement du signal 122
5.4.1 Démodulation 123
5.4.1.1 Implémentation du NCO 124
5.4.1.2 Implémentation du mélangeur 125
5.4.2 Filtre à réponse impulsionnelle finie (FIR) 125
5.4.2.1 Principe 126
5.4.2.2 Problématiques 127
5.4.2.3 Implémentations 128
5.4.3 Bilan 130
5.5 Conclusion 132

Conclusion générale 135

Table des figures 137

Liste des tableaux 145

Bibliographie 147

Publications 151

1 Journaux avec commité de lecture 151
2 Conférences 151
3 Articles de vulgarisation scientifique et technique 152
4 Présentations de vulgarisation scientifique et technique 152

Première partie

Introduction

Introduction générale

Il est connu que le débit des flux vidéo ou radio-fréquence a grandement augmenté, pour atteindre, à l'heure actuelle, plusieurs centaines de Mo/s. Dans le même temps, la capacité de traitement et la vitesse des processeurs ont également suivi une courbe croissante. Toutefois, ce type de composants ne peut pas, en temps-réel, traiter des flux tels que ceux précédemment cités. Cette restriction est dûe d'une part aux bandes passantes d'acquisition limitées qui induisent la nécessité de passer par des cartes dédiées disposant de grandes capacités de stockage afin de réaliser l'acquisition en deux étapes : d'abord l'acquisition et ensuite le transfert au processeur. D'autre part, compte tenu de la complexité des processeurs, l'utilisation d'un système d'exploitation est obligatoire. Avec un tel environnement il n'est pas possible de garantir l'aspect prédictible du comportement du processeur, avec le risque majeur de perte de données. Il existe des extensions temps-réelles, mais là encore elles présentent des limites du point de vue des latences.

Une des solutions les plus adaptées pour l'acquisition d'un flux présentant un débit important et pour son traitement en temps réel est le *FPGA* (*Field Programming Gate Array*). De par sa nature massivement parallèle il est capable de traiter des volumes de données très importants avec des caractéristiques temporelles strictes. Toutefois, la mise en œuvre d'un tel composant est complexe ; elle nécessite la connaissance de langages de descriptions de type *HDL* (*Hardware Description Language*), la compréhension de l'architecture matérielle des FPGAs et plus particulièrement de celui cible, et finalement, nécessite une bonne expérience pour obtenir des résultats les plus performants possibles.

Afin de pallier cette difficulté il existe un certain nombre d'approches : certaines consistent à analyser un code écrit dans un langage de haut niveau tel que le *C*, l'interpréter et le convertir dans un langage *HDL*. La seconde approche consiste plutôt à assembler dans un langage orienté objet des instances de classes correspondant à des entités telles que les accumulateurs, signaux, additionneurs.

La dernière approche, que nous avons sélectionné, présente une approche orientée blocs. Au lieu de considérer les unités basiques mises en œuvre dans le cadre d'un FPGA, cette approche met à la disposition de l'utilisateur des blocs qui correspondent à des algorithmes. Toutefois, comme nous le verrons plus en détail, il n'est pas possible de simplement assembler des blocs pour disposer d'un ensemble de traitements sur un flux fiable et fonctionnel : selon l'approche de l'implémentation

un bloc peut présenter des limitations dans sa capacité de réception ou nécessiter des ressources matérielles disponibles uniquement sur un FPGA d'un fabricant mais pas sur les FPGAs produits par un autre. Cet ensemble de paramétres doit être pris en compte pour garantir que la chaîne de traitement aura un comportement fiable dans tous les cas.

Cette thèse *CIFRE* (*Conventions Industrielles de Formation par la REcherche*) correspond à un intérêt commun, dans l'exploitation des fonctionnalitées du FPGA, entre la société *Armadeus Systems* et l'équipe *CoSyMa* (*Composants et Systèmes Micro-Acoustiques*) du Département Temps-Fréquence de l'institut *FEMTO-ST* (*Franche-Comté Électronique, Mécanique, Thermique et Optique - Sciences et Technologies*). *Armadeus Systems* produit des cartes embarquées alliant un processeur générique dont le rôle est l'exécution d'algorithmes de haut niveau d'abstraction et la communication avec son environnement (affichage d'informations, transfert par le réseau) et un FPGA. Les deux composants sont reliés par un bus de données rapides. De cette expérience, ainsi que de la demande de nombreux clients, il est apparu le besoin de disposer d'un outil capable de simplifier la génération de chaînes de traitements. Le département Temps-fréquence quant à lui a pour objectif de déplacer certains traitements jusqu'alors analogique vers du numérique, impliquant l'utilisation de FPGA.

Nous débuterons ce manuscrit par la présentation des solutions dédiées à la mise en œuvre des FPGAs, puis nous introduirons les problématiques liées à une approche bloc et aux algorithmes de traitements vidéo et radio-fréquence. Fort de ces études il sera possible de présenter les objets et les spécifications de notre outil, nommé *CoGen* (*Coprocessor Generator*).

Le second chapitre sera dédié à la présentation des divers aspects liés aux contraintes imposées pour garantir la capacité d'assemblage, aux descriptions des blocs pour permettre l'analyse de la chaîne de traitement dans sa globalité afin de garantir sa compatibilité matérielle mais également sa capacité à réaliser l'ensemble de traitements inclus. Ce chapitre se finit sur la problématique liée à l'utilisation de composants particuliers tels que les puces de RAMs, dont le comportement est indépendant du FPGA et peut potentiellement induire des latences parfois aléatoires.

Ayant présenté *CoGen* et en en exploitant les caractéristiques présentées dans le second chapitre, nous pourrons réaliser des applications expérimentales pour valider les choix techniques de conception, dans le quatrième chapitre sur des flux vidéo et, dans le chapitre cinq, dans le cadre de traitements de signaux radio-fréquences. Ce dernier chapitre sera également l'occasion de présenter un outil dédié à la *SDR* (*Software Defined Radio*) dont les concepts de base sont proches de ceux de notre outil et qui présente un intérêt pour réaliser une seconde étape de traitements sur le flux issus du FPGA. Les perspectives porteront principalement sur un rapprochement de cet outil de traitement numérique du signal largement utilisé dans la communauté de la SDR, avec CoGen.

Chapitre 1

Présentation

1.1 Techniques d'accélération de traitements

Avec l'évolution de la vitesse des flux issus de périphériques d'acquisitions tels que les caméras industrielles et les convertisseurs analogiques-numériques, mais également avec l'augmentation de la complexité des traitements à réaliser, les durées de calculs ont augmenté. Cette évolution des contraintes en terme de transfert/stockage et de calculs entraîne, en parallèle, un besoin toujours plus grand de faire évoluer les solutions pour absorber et compenser l'augmentation de la complexité du traitement et du temps lié. Il existe plusieurs solutions ou plusieurs axes de recherches [1, 2, 3] afin de répondre à ces besoins.

La première solution concerne la distribution d'un calcul sur plusieurs cœurs ou plusieurs machines : en même temps que la vitesse des périphériques d'acquisitions a augmenté, celle des processeurs a suivi la même courbe. Les processeurs ont vu leur capacité évoluer en terme de fréquence et là où à une époque récente les processeurs n'avaient qu'un fil d'exécution, maintenant grâce à la multiplication des cœurs il est possible de réaliser des traitements parallèles. Ceci a permis de faire évoluer les solutions de traitements distribués sur plusieurs cœurs d'un même processeur, sur des grappes de processeurs telles que dans le cas des super-calculateurs ou d'ordinateurs en découpant les traitements complexes pour les distribuer, principalement dans le cadre de données matricielles ou vectorielles. Cette solution présente une évolution majeure réduisant le temps global des traitements. Toutefois, celle-ci n'est pas toujours la plus adaptée : selon la taille des données, ou les dimensions, le coût du transfert entre cœurs ou machines peut dépasser le gain apporté par une telle solution.

En parallèle de cette première approche, une seconde solution, toujours dans le même contexte, à savoir le traitement logiciel, correspond à l'exploitation des GPUs (Graphical Processor Unit). Cette solution représente une évolution rapide et est le sujet de nombreux travaux de recherches depuis quelques années [4]. L'industrie du jeu vidéo a grandement motivé les fabricants à proposer des composants capables de réaliser des traitements toujours plus complexes et toujours plus rapides,

en particulier sur des données vectorielles ou matricielles. Dans le cadre du traitement d'image, des recherches [5] ont permis de pouvoir traiter 1854 Mpixels/secondes, sur des images en 4096 × 4096 pour l'application d'un filtre médian. La limitation de cette solution réside dans la nature même du composant : celui-ci est orienté vers du calcul matriciel ou vectoriel et présente des performances dégradées pour des volumes de données petits (temps de transfert supérieur à la durée des traitements) et n'est pas adapté à un flux où les données arrivent séquentiellement.

Ces deux solutions ont un point commun : un processeur a la charge de réaliser l'acquisition. Or ce type de composant, animé par un système d'exploitation à temps partagé, peut représenter le facteur limitant dans l'acquisition d'un flux rapide. Plus globalement dans le cas d'un flux vidéo rapide ou l'acquisition d'un flux radio-fréquence par un convertisseur analogique-numérique, l'ordinateur se voit adjoindre une carte dédiée comme un framegrabber tel que le **NI_PCIe-1473R**[1] équipé d'un **Virtex-5 LX50** ou une carte disposant directement de convertisseurs telle que la carte *Alazartech* **ATS9625**[2] équipée d'un **Stratix III**. Dans ces deux cas, ce type de carte contient un FPGA (Fied Programmable Gate Array), seul composant apte à réaliser une acquisition en temps-réel, avec garantie de l'absence de pertes de données. Toutefois dans ce type de situations, le FPGA est le plus souvent cantonné à l'acquisition du flux, au stockage dans des RAMs rapides de grandes dimensions, puis au transfert au travers d'un protocole rapide vers le processeur.

Ainsi dans tous les cas présentés précédemment, le traitement est réalisé en deux étapes : la première dédiée à l'acquisition, la seconde au traitement. Les résultats des calculs sont temporellement non corrélés à l'état d'un système au moment où ceux-ci sont disponibles, et il est impossible de pouvoir réaliser, en temps-réel, un traitement avec une rétroaction sur le système étudié.

1.2 Calcul sur FPGA

Au lieu d'exploiter le FPGA dans le seul but de réaliser l'acquisition et le transfert de données brutes, il est possible de concentrer l'ensemble des traitements dans ce composant. Contrairement à un processeur, un FPGA est totalement reconfigurable, et le développeur n'est pas limité à un jeu d'instructions et a toute liberté pour y intégrer ce qu'il souhaite. Grâce à sa nature même, il est possible de réaliser des traitements en temps-réels [6, 7, 8], mais également de synthétiser un ou plusieurs cœurs de processeurs [9, 10]. Cette approche permet de conserver le même niveau d'abstraction que sur un processeur classique car les traitements se font de la même manière. Néanmoins cette technique va limiter le goulot d'étranglement induit par le transfert entre une carte d'extension et le processeur : l'acquisition et le traitement étant dans le même composant, il est possible d'accéder à des zones mémoires directement ou de réaliser une extension intégrée au processeur. Le coût temporel de la communication entre cœurs est également réduit du fait la proximité de ceux-ci. Toutefois la solution de l'intégration d'un processeur *softcore* entraîne une

1. http://sine.ni.com/nips/cds/view/p/lang/fr/nid/210038
2. http://www.alazartech.com/products/ats9625.htm

réduction de la place disponible dans le FPGA pour les traitements devant être temps-réel. Pour pallier cet inconvénient, des travaux [11] ont été réalisés dans le but d'analyser la représentation intermédiaire, produite par le compilateur, d'un code, dans un langage de programmation, pour supprimer des portions du cœur ou des instructions qui ne seraient pas nécessaires.

La seconde solution est de se passer, pour le pré-traitement, d'un processeur et de décrire l'algorithme de traitement par la logique à mettre en œuvre avec un des langages dédiés de type *HDL (Hardware Description Language)* : VHDL ou Verilog [12, 13]. Le flux après cette première série de traitements présentera un débit compatible avec les capacités d'un processeur classique [14, 15]. Au travers de tels langages, des éléments tels que des portes logiques, des accumulateurs et des mémoires vont être décrits. Un logiciel, spécifique à chaque constructeur, produit un binaire synthétisant le comportement que doit prendre le FPGA. Cette solution est sans doute celle qui permet, en se rapprochant du matériel, à l'instar de l'assembleur, de réaliser l'implémentation la plus performante. Toutefois c'est également la solution présentant la durée de développement la plus longue.

Pour un développeur habitué aux langages classiques, l'apprentissage d'un tel environnement n'est pas aussi trivial que l'apprentissage d'un nouveau langage utilisable sur un ordinateur ou équivalent. L'étape d'apprentissage du langage n'est qu'une petite partie du travail pour être en mesure de réaliser des codes aussi performants que possible. La nature du FPGA doit être prise en compte, ce qui implique une logique de développement très particulière : la contrainte de temps, l'aspect concurrent de la plupart des affectations, l'absence de structures itératives (qui sont remplacées par des machines d'états), les contraintes liées aux ressources matérielles sont autant de paramètres à prendre en compte lors de la description de l'algorithme de traitement. Il arrive que sur certains types de ressources, les possibilités offertes par un modèle d'une certaine marque ne soient pas exploitables sur d'autres modèles pour le même fabricant ou sur des FPGAs d'une marque concurrente. Cet ensemble de difficultés entraîne donc une phase d'apprentissage relativement longue pour obtenir une maîtrise correcte de cet environnement, et peut parfois être le facteur limitant pour l'adoption du FPGA.

D'un autre côté pour un développeur habitué aux FPGAs, le travail n'est pas forcément plus simple. Passer un algorithme prévu pour un processeur présente un ensemble de difficultés liées aux choix d'implémentations (parallélisme, pipeline [16] ou au contraire nécessité de mise en œuvre de machines d'états dans le cas de traitements ne permettant pas un découpage autre) ; à la nécessité d'adapter un algorithme récursif [17, 18] ; et aux contraintes telles que l'utilisation de nombres flottants [19], l'approximation par utilisation de valeur entières, et la précision des données [20]. Il est toutefois possible d'utiliser des codes existant, mais ceux-ci ne sont pas toujours génériques. Étant prévus dans le cadre d'une utilisation ou d'un projet particulier, ils doivent, la plupart du temps, être réadaptés, encapsulés ou bien un bloc d'adaptation (*glue*) doit être réalisé pour les utiliser. La validation se fait de préférence à l'aide de simulations, qui peuvent prendre du temps

et être très souvent complexes : le FPGA se comporte comme une *boîte noire*, et les solutions classiques de déverminage sont peu ou pas possibles.

1.3 Méthodes de développement alternatives

Ces difficultés inhérentes aux développements sur FPGA ont donné lieu à l'apparition de solutions dont l'objectif vise à simplifier le travail mais également à réduire le temps de développement, ceci afin de permettre de rendre ce composant plus accessible à des non spécialistes.

Dans ces solutions, on trouve plusieurs grands ensembles de voies de développements : la première, nommée High-Level Syntesis (HLS) [21, 22] consiste à générer le code HDL à partir d'un code réalisé dans un langage de haut niveau comme le C, à l'aide d'un analyseur qui peut être rapproché d'un compilateur. Le principe de cette solution consiste à utiliser une représentation intermédiaire du code sous forme d'arbre, réalisée par un compilateur. Cet arbre est ensuite analysé pour définir les traitements indépendants ou au contraire nécessitant l'achèvement d'un premier traitement, ainsi que les possibilités de factorisation de certaines ressources dont le volume est restreint, comme les multiplieurs. Une fois l'analyse finie, le code VHDL est produit. Avec une telle solution, un non spécialiste peut créer des designs, mais le code HDL produit n'est pas nécessairement optimisé sur un point de vue de performance ou donne des performances dans les mêmes grandeurs qu'un code réalisé manuellement, mais aux dépends de l'encombrement dans le FPGA [23]. Le code C, pour permettre la meilleure analyse, doit-être pensé dans l'optique de la génération du code VHDL en tenant compte des caractéristiques du FPGA, ou annoté pour guider les outils [24]. Certaines opérations itératives telles que **while** ne semblent pas toujours bien exploitées. Il est souvent nécessaire de modifier le code produit afin de l'optimiser. Ceci nécessite donc une certaine expertise en terme d'optimisation et de connaissance du langage VHDL ainsi que du FPGA.

La seconde solution consiste également à éviter à l'utilisateur l'ensemble des aspects laborieux par l'utilisation d'un langage plus classique et par la disponibilité d'un ensemble de classes qui ecapsulent les éléments de base de la programmation sur FPGA (signaux, compteurs, comparateurs, ...). C'est le cas d'outils tels que migen [25, 26] ou MyHDL [3], qui proposent un développement en python. Ou de *HIDE* [27, 28, 29]. Ce type d'outil, en plus de fournir un ensemble de briques de bases, offre également la possibilité de produire le code Verilog ou VHDL correspondant et permet d'automatiser les simulations. Le développeur peut donc, bien plus facilement, créer des classes pour un but particulier, les assembler, réaliser les simulations et finalement produire le code HDL correspondant. Ces solutions présentent un réel intérêt en vue d'accélérer le code, de simplifier la plupart des étapes de la construction d'un projet, et finalement ont l'avantage de ne pas imposer au développeur de connaissance en VHDL ou Verilog mais imposent malgré tout une connaissance des caractéristiques du FPGA et de la simulation. Les résultats sont également

3. http://www.myhdl.org/

grandement dépendants du code que le développeur à rédigé. Il semble donc que ce type de solution soit intéressante pour simplifier le travail d'une personne compétente mais n'offre pas une solution viable pour un utilisateur débutant.

La dernière solution est basée sur l'utilisation de *HDL Coder*[4], un outil commercial, dont le but est d'aider le développeur dans la génération de la chaîne de traitements par l'assemblage de blocs de base d'une manière graphique. Cette solution présente certains aspects équivalents à la solution précédente. Bien qu'il n'existe pas d'évaluations des caractéristiques de cet outil (clause du contrat d'utilisation), un examen est décrit dans [30]. Cette solution étant basée sur des blocs tels que des comparateurs ou des compteurs, les performances du résultat sont très liées à la description et donc à la connaissance du développeur vis-à-vis du matériel.

Les diverses solutions présentent l'avantage de simplifier le développement de chaînes de traitements avec un niveau de performance plus ou moins important selon la qualité et la connaissance du matériel. Toutefois ce type d'approche est un bon compromis entre la performance et le temps de développement. Il est clair que les traitements réalisés au début d'une chaîne nécessitent un niveau de performance maximum, et donc un développement proche du matériel. Par contre, plus le traitement est éloigné du périphérique, plus les contraintes sont relachées, ce qui permet d'envisager l'utilisation de blocs de traitements avec des caractéristiques plus lentes.

1.4 Positionnement de nos travaux

Partant de cet ensemble de constats, l'objectif de nos travaux est la création d'un outil logiciel, nommé CoGen, reposant sur un concept orienté assemblage de blocs [31]. Une telle approche a aussi été choisie pour *POD* (*Peripheral On Demand*) et ainsi que par le projet *White Rabbit*[5] [32] pour l'outil *hdl-make*[6]. Ces deux outils servent à l'assemblage des blocs et la création de l'ensemble des fichiers nécessaires pour la génération du binaire à destination du FPGA. Ils reposent sur un ensemble de blocs préalablement réalisés et des scripts *XML* pour la description des blocs. Notre méthodologie de travail a également été guidée par le concept du skeleton [33, 34, 35], qui ajoute, par rapport aux deux précédents outils, la possibilité, pour un même algorithme, de disposer de plusieurs implémentations plus ou moins optimisées visant à proposer une solution générique. Par rapport à ces solutions, nous avons, également, ajouté une notion de validation de l'adéquation entre les capacités de traitement des blocs et le débit reçu.

Contrairement aux solutions présentées dans les sections précédentes, où l'utilisateur doit réaliser lui même tout ou partie de l'application, et doit au moins avoir un minimum de connaissance des FPGAs, notre but est de fournir une boîte à outil sous la forme d'un ensemble d'algorithmes dont chacun est décrit par un fichier et une méthode d'assemblage associée. Chaque algorithme,

4. http://www.mathworks.fr/products/hdl-coder
5. http://www.ohwr.org/projects/white-rabbit
6. http://www.ohwr.org/projects/hdl-make

selon sa nature, peut être paramétré (taille de la matrice de convolution, de la table coefficients). Ainsi, l'utilisateur n'a à sa charge que de choisir les algorithmes et les assembler pour former une chaîne de traitement, l'outil réalisant l'ensemble des validations afin de garantir que le design HDL est exploitable dans les conditions fixées, à savoir un matériel particulier et des contraintes sur le rapport entre les performances et la consommation de blocs matériels du FPGA.

Le bénéfice pour l'utilisateur d'une telle solution est que CoGen se charge de valider l'adéquation de chaque bloc avec le matériel. Ainsi, la cohérence de l'ensemble des traitements devant être réalisés et la capacité de chaque traitement à recevoir le flux entrant est garanti. L'utilisateur n'a pas, comme c'est souvent le cas avec les solutions classiques, à se soucier des caractéristiques du matériel, et donc par effet de bord la même chaîne de traitement sera automatiquement adaptée au FPGA cible pour toujours présenter la meilleure solution possible.

La manière de développer le code d'un bloc et le choix d'implémentation sont laissés à la discrétion des développeurs qui peuvent soit utiliser un des outils précédemment cités, ou la méthode plus classique du développement manuel à l'aide d'un langage de type HDL.

Notre solution vise principalement à rendre l'utilisation des FPGAs moins difficile ; il n'y a pas une recherche directe de la simplification de développement des blocs intégrés dans l'outil mais plutôt une volonté de proposer la solution la plus adaptée à un cas donné. Toutefois, l'approche "bloc" choisie permet aux développeurs de se concentrer uniquement sur le composant HDL qu'il doit créer en faisant abstraction de l'environnement dans lequel il sera intégré ensuite.

Avant de pouvoir rédiger les spécifications de l'outil, une étude de contraintes liées au matériel mais également aux algorithmes est nécessaire. L'étude des algorithmes et la mise en œuvre pour validation va principalement porter sur deux domaines d'applications particuliers : le traitement vidéo donc à deux dimensions et le traitement du signal à une dimension et plus particulièrement des signaux radio-fréquences et de la *SDR* (*Software Defined Radio*)[36, 37, 38].

1.5 Environnement matériel et logiciel

L'ensemble des travaux sur FPGA a été réalisé sur une plate-forme Armadeus SP_VISION disposant d'un FPGA Spartan6, ainsi que de deux mémoires externes de 128Mo. Elle dispose également d'un connecteur générique utilisé pour ajouter des cartes d'adaptations dans le but de connecter des périphériques d'acquisition tels que des caméras ou des convertisseurs analogiques-numériques. Cette carte est connectée (figure 1.1) sur une seconde carte de type APF27 disposant d'un FPGA Spartan3 et d'un micro-processeur. Ce dernier sert pour la programmation du FPGA de la carte SP_VISION, et pour la communication (paramètrage des blocs, transfert des données depuis le FPGA), le transfert des données par le réseau ou à un affichage/interaction avec l'utilisateur au travers d'un écran LCD tactile.

La communication utilise le bus wishbone[39] spécifié par opencores [7]. Ce bus consiste en un maître (le processeur) et un ou plusieurs esclave(s) correspondant aux blocs se trouvant dans le FPGA. Chacun d'eux dispose d'un identifiant unique et d'une plage d'adresse utilisée pour définir des registres en lecture et écriture.

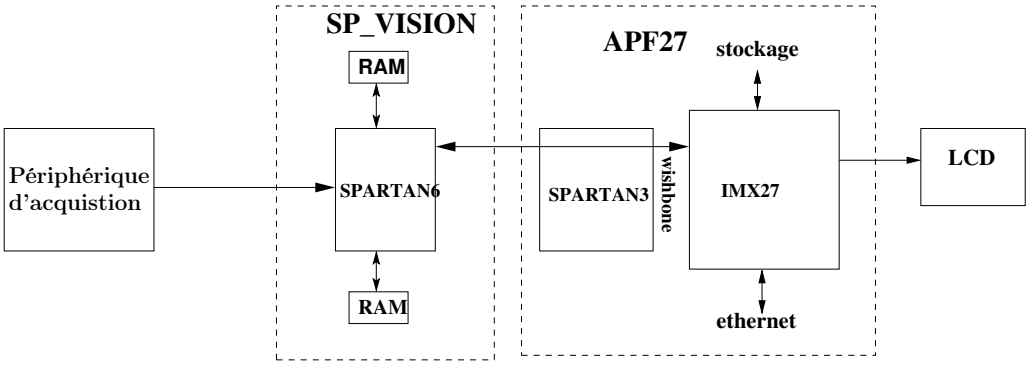


FIGURE 1.1 – Schéma global de l'environnement matériel utilisé pour ce projet.

Armadeus fournit également, pour la génération de bitstreams, un outil nommé POD (Peripheral On Demand) [40], dont le but est de construire un projet pour l'outil de synthèse, de cabler les blocs sélectionnés et de produire le fichier de contrainte et le fichier principal en charge de l'instanciation de tous les blocs.

Compte tenu de la faible bande passante entre le FPGA et le CPU en comparaison du débit de certains convertisseurs analogiques-numériques ou caméra, un maximum de traitements doit être fait dans le FPGA afin de réduire le débit. Les données une fois transférées au processeur pourront à nouveau être traitées pour des calculs ne nécessitant pas une puissance importante ou pour des algorithmes non disponibles pour le FPGA. Cette étape peut être réalisée en post-traitement, sur chaque paquet transféré depuis le FPGA ou bien avec des outils tels que GNURadio (section 5.3).

1.6 Application au traitement d'images

CoGen doit manipuler des blocs, les interconnecter et valider un ensemble de caractéristiques afin de garantir que le design pourra être synthétisé avec succès. Ce type de traitement doit, afin de proposer la solution la plus efficace, reposer sur des cas concrets en adéquation avec les algorithmes que l'outil devra manipuler, mais également avec les caractéristiques des langage HDL et des FPGAs. Il est également nécessaire que les choix réalisés présentent un faible coût en terme de consommation de ressources mais, également un temps de traitement global de la chaîne produite, le plus faible possible. Ces choix doivent également garantir que l'assemblage soit toujours possible et ne présente pas d'incohérences ni de risque de perte de données. Ceci passe donc par la nécessité de poser un certain nombre de contraintes concernant la création d'un bloc à destination de l'outil.

7. http://www.opencores.org

D'un autre côté, ces contraintes ne doivent pas être un facteur limitant pour le développeur, et ne doivent pas rendre difficile, voir impossible, l'implémentation de certains algorithmes.

C'est pourquoi, nous avons réalisé une étude et une classification d'un ensemble de filtres de traitements d'images, afin d'obtenir une vision globale des besoins et des contraintes que de tels algorithmes imposent.

1.6.1 Classification par zone de traitement

Cette première classification a pour but d'évaluer les informations que les algorithmes ont besoin de recevoir, ou de disposer, afin de réaliser le traitement. Trois grands groupes de traitements ont pu être décrit :

- le traitement ponctuel : ce type d'algorithme traite la donnée sans nécessité de connaître sa position dans l'image. Son comportement peut se rapprocher d'un traitement sur un flux quelconque non borné. Généralement, il ne dispose pas d'un état particulier, ni de la mémorisation du passé du flux. On trouve des filtres tels que les seuillages qui comparent la valeur du pixel avec un ou plusieurs seuil pour déterminer quelle sera la nouvelle donnée qui sera produite. Il existe aussi des filtres tels que le mapping dans lequel chaque niveau possible est remplacé par une valeur spécifique à l'aide d'une table de correspondance généralement stockée dans une Lookup Table (LUT).
- le traitement sur une zone : le traitement se fait sur une fenêtre de pixels à 1 ou 2 dimensions, une ligne ou une colonne. Ce type, qui englobe des filtres tels que Prewitt, Sobel ou Robert (détections de contours par convolution) (figure 1.2), le filtre médian (remplacement d'un pixel par la valeur médiane du voisinage avec utilisation d'une technique de tri) nécessitent un nombre plus important de pixels. Dans tous ces cas qui reposent sur des traitements matriciels, le filtre a besoin de mémoriser les pixels arrivés par le passé afin de pouvoir réaliser le traitement. Il doit donc connaître la largeur d'une ligne afin de permettre au développeur d'exploiter des RAMs dans un principe de tampon circulaire.
- les traitements sur l'image dans la globalité. On y trouve les rotations, les croppings, les histogrammes. Dans l'ensemble des cas, le filtre doit pouvoir réagir sur la fin de chaque ligne et la fin de l'image. Dans le cas de l'histogramme, à la fin de chaque image, l'ensemble des informations doit être propagé et la mémoire de stockage réinitialisée en vue de la réception de l'image suivante.

Il apparaît donc avec cette première classification deux grands type différents. Un premier dont le besoin se limite à la donnée sans aucune autre information et le second qui doit impérativement disposer d'informations de synchronisation ou des dimensions de l'image.

1.6.2 Types d'entrées et de sorties

Deux grandes catégories de filtres d'images ressortent de cette analyse :

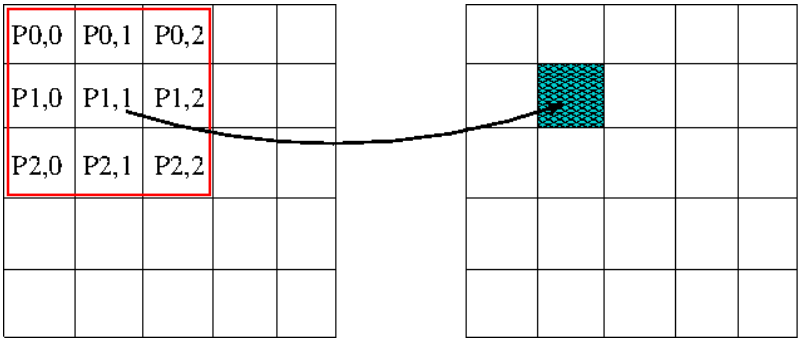


FIGURE 1.2 – Convolution par un noyau 3x3. La production d'un nouveau pixel P1,1 nécessite de disposer du pixel P2,2 de l'image source. Dans le cas du traitement sur un flux ce dernier n'arrivera qu'après une durée correspondant au temps de propagation de tous les pixels qu'une ligne, plus 1 pixel, ainsi que de potentiels temps de pauses inclus dans le flux vidéo.

- les filtres homogènes : ils reçoivent des données de type pixel et produisent des données du même type. C'est le cas des seuillages, des détections de contours et des traitements géométriques et plus généralement de toutes les modifications sur l'image ;
- les filtres hétérogènes : ils recoivent des pixels et produisent des résultats d'un autre type. C'est le cas de l'histogramme, de la FFT et des statistiques.

Ce typage est nécessaire à prendre en compte pour garantir la cohérence du traitement : appliquer un filtre sur le résultat d'un histogramme n'a pas de sens.

1.6.3 Types de traitements

Le premier type de traitements repose sur des traitements simples tels que des comparaisons comme pour les seuillages ou des accès à une mémoire comme pour le mapping : ce type de traitement ne présente aucune complexité et leur implémentation est généralement directe.

Le second représente les traitements de type matriciel (multiplication, tri, ...). Ce type de traitement est plus complexe et compte tenu des possibilités offertes par le FPGA, pour un même algorithme il est possible d'envisager diverses possibilités d'implémentations permettant un gain en performance (réduction de la latence du traitement) ou au contraire pouvant s'adapter à des modèles de petite taille ou pour un besoin d'économie (utilisation d'un sous-ensemble de blocs cablés). Selon le choix fait, le bloc présentera une capacité de traitements élevé mais ne pourra être utilisé que sur des FPGAs disposant d'un important volume de blocs cablés (multiplieur dans le cas de la multiplication matricielle) ou sera apte à être utilisé dans un environnement à faible ressources mais ne pourra accepter qu'un flux plus lent. La seconde caractéristique observable, en prenant le cas particulier d'un traitement matriciel 3×3 (figure 1.2), est que pour produire un résultat le bloc doit disposer de suffisament de données. Ainsi le calcul d'un pixel ne pourra être fait que lors de la réception de tous les pixels nécessaires, il y aura donc d'une part un

décalage temporel correspondant au temps de propagation de deux lignes et de deux pixels entre la réception d'une donnée et la production du résultat correspondant à celui-ci mais également la latence correspondant au temps de traitement entre l'arrivée du dernier pixel nécessaire et la propagation de la donnée résultat. Ce cas de figure doit être pris en compte afin de pouvoir resynchroniser des flux dans le cas où la chaîne de traitements disposerait de branches paralléles.

Il est également possible d'observer que certains algorithmes présentent des caractéristiques ou des besoins identiques. Tous les traitements matriciels par exemple, nécessitent de mémoriser des lignes entières afin de pouvoir réaliser les traitements. Il est ainsi possible, de proposer aux développeurs des solutions d'implémentation de ces parties, afin de lui permettre de constituer son implémentation par assemblage de sous blocs et donc de simplifier et réduire le temps de développement.

1.7 Application au traitement de signaux radio-fréquence

Pour les mêmes raisons qu'exposées en 1.6 et dans le but d'ouvrir les possibilités de CoGen à la gestion des flux radio-fréquences dont les caractéristiques sont équivalentes, que ce soit pour les bandes passantes ou la taille des données, à celles des flux vidéos, nous avons également réalisé des études sur les algorithmes[41] liés à ce domaine d'applications.

Cette étude reprend la structure de celle du flux vidéo.

1.7.1 Classification par zone de traitement

Hormis dans le cas des traitements liés à du radar[42] dans lequel la partie pertinente du flux, issus du convertisseur, est bornée par un début et une fin, un flux radio-fréquence ne présente pas une nature finie. C'est pourquoi seuls deux cas apparaîsent :

- le traitement ponctuel : dans ce type de cas il n'y a pas de notion de voisinage, de passé ou de futur de la donnée. C'est le cas des mélangeurs qui multiplient la donnée avec un signal de référence. Aucun stockage n'est nécessaire, la donnée pouvant être exploitée directement ;
- le traitement sur une série de données : le début de cette série est arbitraire du fait de l'absence de répères temporels ou de position comme dans le cas du traitement vidéo. Le filtrage par convolution est un exemple de ce type de traitement. Il consiste à produire une nouvelle donnée représentative d'un ensemble d'échantillons contïgus, pondérés par les coefficients d'un filtre. Les convolutions sont glissantes et présentent donc un chevauchement des données, un même échantillon sera utilisé pour plusieurs convolutions. Ceci peut entraîner la nécessité de stocker les échantillons, toutefois ce n'est pas obligatoire et dépend de l'implémentation.

En comparaison avec les traitements sur un flux vidéo, un flux radio-fréquence, dont la nature est unidimensionnelle, présente moins de contraintes sur les signaux de contrôles à fournir. Le seul cas où il est nécessaire de mettre en place un signal de contrôle semble être dans le cas d'un flux

radar, pour avertir les blocs du début ou de la fin de la rafale.

1.7.2 Types d'entrées et de sorties

Comme dans le cas de la vidéo on retrouve des traitements homogènes, comme la décimation, et d'autres hétérogènes, comme la démodulation dont l'entrée est de type réel et la sortie de type complexe.

Toutefois, on peut remarquer que certains algorithmes, tels que la Transformée de Fourier, ou son inverse, change la nature de la représentation des données : passage du domaine temporel au domaine fréquentiel, et réciproquement dans le cas de la transformée inverse. Contrairement à un histogramme, qui change également la nature de l'information, appliquer dans ce nouveau domaine des algorithmes est possible, à la condition qu'il soit adapté à celui-ci.

Cette nouvelle classification ne présente pas de contradictions vis-à-vis des études réalisées sur la vidéo, mais au contraire ajoute le besoin de préciser la nature des données au delà, de simplement, préciser le type.

1.8 Spécification de CoGen

CoGen est un outil d'assemblage de blocs en vue de l'automatisation de la production d'une chaîne de traitement sur FPGA. Les blocs proposés à l'utilisateur sont créés en amont par des développeurs et disponibles au travers d'une bibliothèque.

Cette chaîne reçoit un flux de données d'un périphérique externe, tel qu'une caméra ou un convertisseur analogique-numérique. Les données sont ensuite propagées de blocs en blocs dans l'ordre défini par l'utilisateur. À l'issue de l'ensemble des traitements, les données résultats sont transférées à l'extérieur du FPGA pour être stockées, affichées ou envoyées sur le réseau.

Plusieurs étapes sont nécessaires depuis la création de la chaîne par l'utilisateur jusqu'à la production d'un script utilisé par un assembleur de blocs pour la production du binaire final.

La première étape consiste à créer un nouveau projet en fournissant un ensemble d'informations dont les plus importantes sont l'environnement matériel par la sélection de la plate-forme cible (la carte comportant le FPGA), et au besoin par la sélection d'un modèle particulier de FPGA, s'il n'est pas celui par défaut pour la plate-forme, et quel doit être le choix d'optimisation (performance et réduction des latences ou la plus faible utilisation de ressources matérielle en vu d'une réduction de la consommation énergétique).

La seconde étape concerne la construction de la chaîne. En premier lieu en fixant les entrées sorties/sorties du projet, toujours au travers d'une liste, l'utilisateur va sélectionner le périphérique d'acquisition, ainsi que la méthode de propagation des résultats vers l'extérieur du FPGA.

Ensuite l'utilisateur va assembler des algorithmes (représentés par des blocs). L'insertion se fait en deux temps : d'abord le bloc est ajouté au projet, s'il nécessite d'être paramétré (valeur d'un

seuil pour un filtre de seuillage, coefficients d'une matrice pour un produit matriciel,...) CoGen en avertit l'utilisateur qui doit fournir ces informations, ensuite le nouveau bloc est connecté pour recevoir les données depuis un autre bloc (pouvant être le bloc correspondant au périphérique) et pour la transmission des résultats qu'il va produire.

L'étape suivante, une fois la chaîne entièrement spécifiée, est une phase d'analyse dont le but est d'une part de s'assurer de la validité de la chaîne et des blocs qui la composent, afin de garantir que la production du binaire à destination du FPGA n'échouera pas. D'autre part, elle aura pour but de vérifier l'aptitude des blocs à traiter le flux à la vitesse à laquelle chaque bloc recevra les données. Si lors de l'analyse, CoGen détermine qu'il n'est pas possible de fournir une solution viable pour la production du binaire, une erreur sera remontée à l'utilisateur.

Si la phase d'analyse finit avec succès, CoGen produit un fichier donnant l'ensemble des instructions nécessaires pour qu'un second outil, *POD*, dont le rôle est de générer un projet HDL, puisse, à l'aide de l'outil du fabricant du FPGA cible, construire le projet, assembler les blocs HDL par production d'un fichier particulier écrite en VHDL ou Verilog et lancer la production du fichier binaire.

Il apparaît dans cette description que de nombreuses questions, tant au niveau de l'environnement, de l'ordonnancement ou de la description, vont se poser avant de pouvoir construire cet outil.

1.9 Bilan théorique

Dans les sections précédentes ont étés présentés les objectifs visés pour CoGen ainsi qu'un certain nombre de caractéristiques liées aux algorithmes et au FPGA. Ces besoins ainsi que l'automatisation d'une part importante des traitements en vue d'un assemblage efficace et d'un choix le plus optimal possible de la solution présente des difficultés. Cette section recense l'ensemble des problèmes et questions qui vont se poser.

L'interconnection entre blocs présente le premier problème : un algorithme peut-être homogène (comme pour un seuillage qui reçoit des données vidéo), hétérogène (comme un histogramme) et traite un flux particulier comme de la vidéo ou générique comme un flux de nombres complexes. La conséquence est qu'il est nécessaire de garantir la cohérence entre les blocs. Le second point important dans la connection réside dans l'assemblage en lui même : deux développeurs ne vont pas nécessairement, pour un même type de flux, fournir le même ensemble des signaux de contrôle et de données pour l'entité du composant HDL. Ceci se justifie par les besoins de l'algorithme mais également par l'environnement lors du développement (nécessité de s'adapter à un flux spécifique, fournissant un ensemble de signaux). En prenant trois cas particuliers il est possible d'envisager les besoins et les difficultés induites par une telle situation : pour un seuillage la seule information nécessaire est la donnée. Ainsi à chaque cycle d'horloge, le bloc va faire correspondre la sortie avec

l'entrée en accord avec la valeur de seuil ; pour une convolution, l'implémentation va avoir besoin au minimum de la donnée, de la largeur de la ligne afin de pouvoir basculer l'utilisation des RAMs, et une information qui donne la présence d'une nouvelle donnée à stocker et donc qui démarre le nouveau calcul ; et pour finir le cropping qui a besoin d'avoir les mêmes informations que la convolution afin de pouvoir compter les pixels d'une ligne et les lignes de l'image, mais nécessite également de savoir que l'image est entièrement passée afin de se remettre dans son état initial. Avec une telle situation, si le seuillage est en début de chaîne, il est possible de considérer qu'aucune information de présence/absence de donnée ne sera propagée. Créer cette information n'est pas forcément facile, voir est impossible, et donc cet assemblage risque d'être finalement irréalisable. Il semble donc nécessaire qu'il faille fournir une solution qui permette l'assemblage sans échec possible, ni ajout d'un volume trop important de latences induites par le code de connection. Cette question en entraîne une seconde : selon le type de traitement, un algorithme doit disposer de plus ou moins d'informations. Une première approche serait de considérer que seules les informations les plus communément nécessaires seront propagées, permettant d'avoir ainsi une interface qui ne serait pas limitée à un type de flux mais au contraire générique. Ceci implique que l'implémentation doit être paramétrée par les dimensions de l'image et que le développeur devra embarquer dans son implémentation les compteurs et comparateurs nécessaires afin de calculer les informations nécessaires. La seconde option consiste à considérer que chaque flux doit disposer d'un ensemble de signaux pertinents permettant de simplifier le code du développeur qui n'aura pas la nécessité d'implémenter un ensemble de traitements pour obtenir ces informations.

Comme présenté, selon les algorithmes et contrairement à un CPU, le FPGA permet d'envisager un certain nombre d'implémentations possibles. Cette multiplicité des implémentations permet de fournir un choix riche et adapté au flux et aux caractéristiques du FPGA. Toutefois, et afin d'éviter à l'utilisateur la difficulté du choix faisant perdre de son intérêt à CoGen, cet ensemble d'implémentations ne doit être visible que par l'outil.

Ces implémentations vont nécessiter un ensemble de traitements qui auront pour but de confirmer ou d'infirmer la possibilité de leur utilisation.

La première analyse concerne la validation temporelle et comportementale d'un bloc par rapport au flux entrant, faite de manière automatique. En effet selon le traitement et l'implémentation, le code peut nécessiter uniquement un cycle d'horloge pour des algorithmes simples, comme un seuillage, ou être mis en œuvre à l'aide d'une implémentation avec des traitements parallèles (figure 1.3) ou pipelines, comme, selon l'implémentation, la convolution. D'autres traitements, au contraire, vont nécessiter plusieurs cycles d'horloge avant de produire le résultat correspondant comme pour une érosion qui est un traitement itératif, ou pour économiser des ressources en faisant usage d'un même multiplieur pour plusieurs traitements (figure 1.4). Ainsi, si aucune vérification n'est faite entre la capacité d'un bloc et le flux, une partie des données arrivant au bloc se trouverait perdue, rendant les résultats globalement faux. Cette notion de débit maximum implique de proposer une

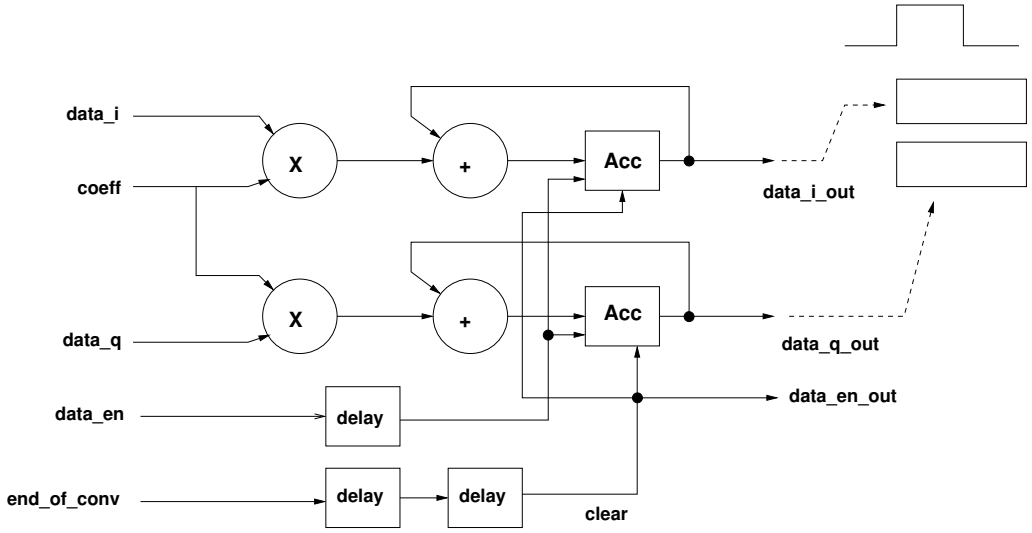


FIGURE 1.3 – Implémentation optimale d'un traitement qui reçoit deux données en parallèle, les multiplie par une troisième valeur avant de réaliser une accumulation. Du fait de l'utilisation de deux multiplieurs en parallèle, un seul cycle d'horloge est nécessaire pour réaliser les deux traitements. La capacité de réception de données d'une telle implémentation est maximisée car chaque multiplieur sera disponible au cycle suivant la réception.

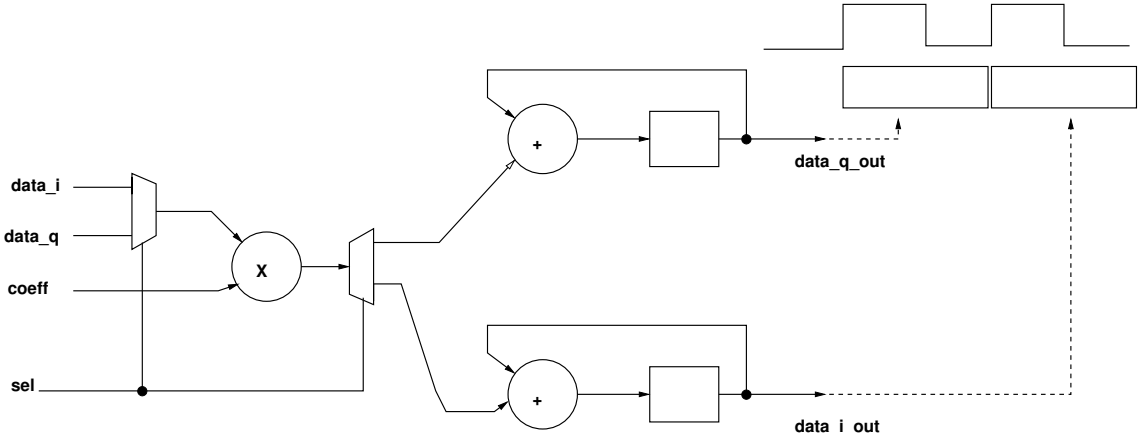


FIGURE 1.4 – Implémentation visant à économiser les multiplieur. Les données $data_i$ et $data_q$ sont séquentiellement traitées. Ainsi, le bloc multiplieur n'est pas disponible pour le lot suivant pendant deux cycles d'horloge. Cette implémentation présente une bande passante deux fois plus lente que l'implémentation de 1.3 mais pourra être privilégiée si les resources matérielles l'exigent et que le flux entrant n'est pas trop élevé.

solution la plus efficace possible et la plus pertinente pour décrire les capacités du bloc en terme de vitesse d'entrée et de représentation des données produites.

La seconde analyse concerne la concordance entre les besoins d'une implémentation et de la chaîne dans sa globalité sur un point de vue de ressources. Il est nécessaire de garantir que la génération du binaire n'échouera pas car une ressource nécessaire (blocs multiplieurs, bloc de RAM) n'est pas disponible ou que la chaîne en nécessite plus que le volume disponible sur le FPGA cible. Cette analyse doit être faite d'une manière transparente pour que CoGen soit exploitable par des utilisateurs possédant peu de compétences dans le domaine des FPGAs.

Ces caractéristiques temporelles impliquent également d'ordonnancer les traitements. En effet, et principalement dans le cas d'algorithmes mis en œuvre sous la forme d'une machine d'états finis, donc nécessitant un traitement global sur plusieurs cycles d'horloges, il est nécessaire de garantir que le traitement commencera toujours sur une nouvelle donnée : si le traitement commence trop tôt ou trop tard, la donnée pertinente se trouverait perdue.

CoGen doit pouvoir accepter des traitements dans lesquels plusieurs branches parallèles de traitements sont présentes. Ceci implique de s'assurer que les données des diverses branches arrivent de manière synchrones dans le bloc recevant les résultats. Cette vérification impose de connaître et adapter les latences des branches. La latence d'un traitement ne présente finalement, pour ce type de vérification, aucune difficulté. Par contre dans le cas de traitements matriciels qui vont produire le résultat $R_{n,m}$ suite à la réception de $P_{n+x,m+y}$, la connaissance de cette différence temporelle va se poser. Un second point important en relation avec le parallélisme réside dans la vérification de la cohérence des flux. Si l'utilisateur spécifie une branche avec une décimation par 2 et une autre branche avec une décimation par 3, CoGen doit pouvoir le détecter, avertir l'utilisateur, et s'adapter à cette situation.

Toujours concernant l'aspect temporel il apparaît également une difficulté lorsqu'il est nécessaire, pour des questions de volume de données à stocker, d'utiliser une RAM externe. Celle-ci présente des latences bien supérieures à celle des RAM internes, modifiant de ce fait d'une manière non systématique la capacité d'un bloc en terme de réception mais également la latence du-dit bloc. Les problèmes et les principes théoriques en vue de l'utilisation optimale d'un tel composant sont développés dans le chapitre 3.

L'ensemble de ses questions a montré qu'il est nécessaire que l'outil puisse avoir un ensemble d'informations caractérisant chaque bloc de la chaîne : CoGen ne peut pas analyser chaque code HDL, pour pouvoir les obtenir par lui même, il faut donc que des fichiers intermédiaires soient réalisés afin de disposer d'une description précise de chaque bloc.

Deuxième partie

CoGen

Chapitre 2

Description de l'outil

À partir du cahier des charges de l'outil mais également des diverses caractéristiques et des problématiques mises en avant dans la première partie, le travail suivant a consisté en deux grandes spécifications :

- la première a concerné la définition des blocs et plus globalement la spécification d'une structure en terme de code mais également de description particulière imposée aux développeurs afin de garantir qu'une implémentation d'un algorithme ou la réception d'un flux soit directement fonctionnelle et que CoGen soit toujours en mesure d'obtenir l'ensemble des informations pertinentes en vue des analyses à réaliser ;
- la seconde concerne l'outil à proprement parler et la nature des traitements qu'il doit réaliser afin de :
  - vérifier la validité d'un bloc vis-à-vis des caractéristiques matérielles, de son environnement (bloc précédent et bloc suivant) et du flux ;
  - vérifier que la chaîne dans sa globalité peut être mise en place dans le FPGA, cette analyse doit prendre en compte les caractéristiques matérielles et temporelles des traitements.

Ce chapitre présente, dans la première section, les choix techniques relatifs aux blocs. Ces choix concernent principalement un ensemble de contraintes imposées au développeur afin de garantir une bibliothèque homogène. Grâce à celle-ci, il est possible de vérifier la cohérence des entrées-sorties entre chaque bloc en terme de nature, ou type, des interfaces, et finalement de réaliser ces inter-connexions sans la nécessité d'ajout de code d'adaptation (glue).

Le second point présenté concerne l'aspect temporel et comportemental des blocs. Les techniques et les ressources utilisées pour la réalisation d'une implémentation ont un impact sur la capacité du bloc à recevoir et à traiter un certain flux de données. Il donc nécessaire de mettre en place une méthode de description des caractéristiques temporelles et comportementales de chaque implémentation afin de fournir à CoGen l'ensemble des informations pertinentes pour garantir l'adéquation entre une implémentation particulière et le flux reçu.

La section suivante présente la manière, pour le developpeur, de fournir à CoGen une description d'un bloc tant sur ses caractéristiques temporelles que sur les besoins de celui-ci quant aux ressources. Cette interaction se fait au travers de fichiers XML et d'un ensemble de balises dont la description est fournie.

À partir de cet ensemble d'informations, l'outil va devoir faire les analyses présentées par la suite.

Les commandes à la disposition de l'utilisateur pour ajouter, assembler et construire la chaîne de traitement seront décrites.

Certains algorithmes ou blocs de transferts vers le processeur peuvent nécessiter des composants externes, tels que des RAMs permettant le stockage d'une grande quantité de données. Ce type de composant peut présenter un aspect asynchrone vis-à-vis du FPGA. Au même titre qu'il est nécessaire de garantir l'aptitude d'un bloc à absorber un flux, il est impératif de tenir compte des caractéristiques précises (latence, bande passante, ...) de ces éléments externes afin de pouvoir conserver l'aspect prédictible du comportement global de la chaîne de traitements générée.

Ce chapitre se clôt sur un exemple concret d'une chaîne traitée par CoGen permettant de valider le comportement de cet outil.

2.1 Chaînes et Blocs

2.1.1 Chaînes

Une chaîne (figure 2.1) représente le traitement dans sa globalité, elle contient a minima une source de données et un ou plusieurs puits utilisés pour propager les résultats à l'extérieur du FPGA.

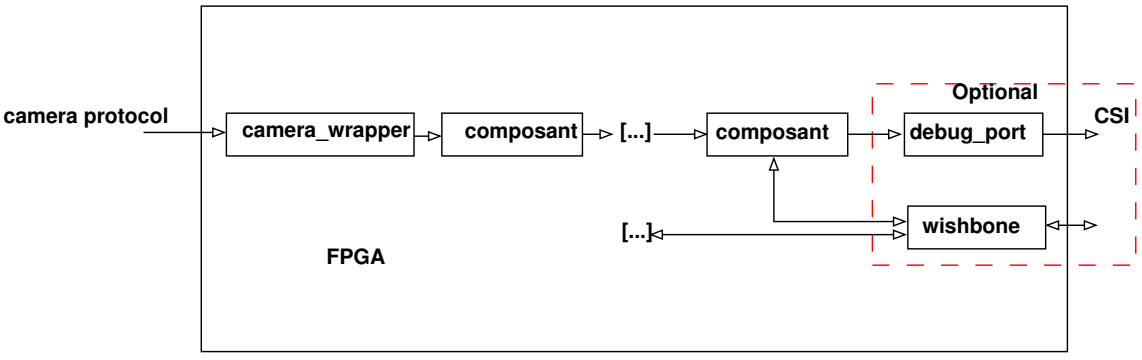


FIGURE 2.1 – Exemple de chaîne de traitements appliqué au cas particulier de l'acquisition vidéo.

Cette chaîne est représentée en mémoire comme un arbre orienté (figure 2.2) dont la source de données correspond à la racine de l'arbre et le ou les puits aux feuilles. Cet arbre peut être linéaire (figure 2.2) ou avoir plusieurs branches (figure 2.3).

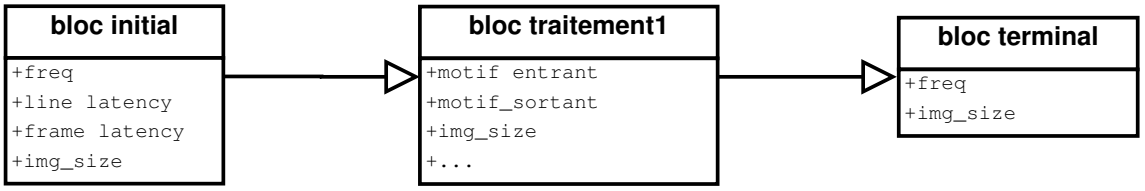
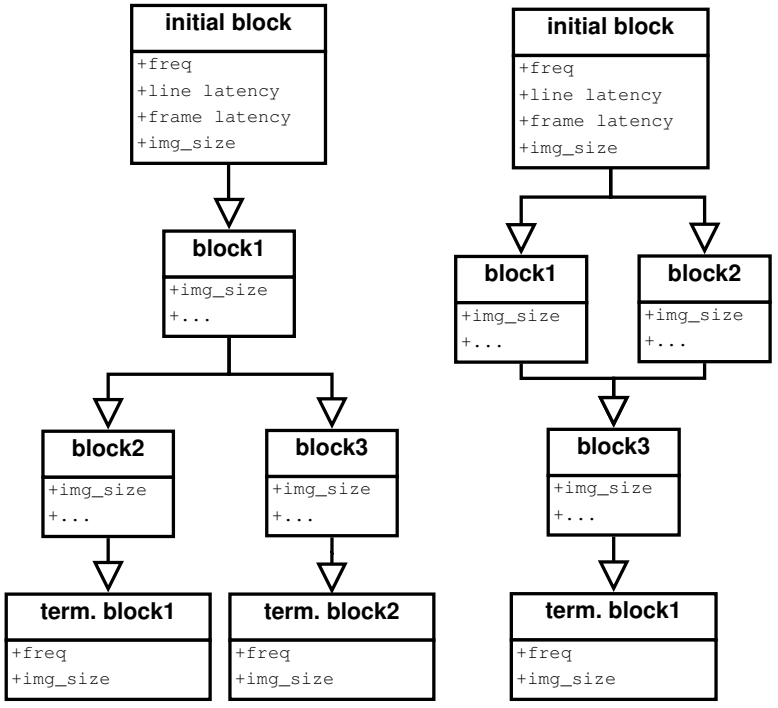


FIGURE 2.2 – Représentation d'une chaîne basique telle que stockée en mémoire.



(a) linear (b) tree

FIGURE 2.3 – Deux types de chaîne disposant de branches de traitements parallèles. 2.3(a) : branches indépendantes disposant chacunes de leur propre bloc de propagation, 2.3(b) : branches parallèles se rejoignant.

2.1.2 Blocs

Le bloc est l'unité de base manipulé à la fois par l'utilisateur lors de la constitution de la chaîne, et par CoGen, lors des diverses analyses devant être réalisées pour garantir la cohérence des données, du flux et la capacité du bloc à traiter ce dernier.

Dans la suite de cette section, nous allons présenter d'une part les contraintes que les développeurs devront respecter afin de rendre leurs implémentations compatibles avec CoGen, d'autre part les informations qu'ils devront fournir pour décrire le comportement et les caractéristiques telles que l'utilisation des ressources du FPGA. Ces informations sont déduites du code par sa connaissance ou par son analyse à l'aide de simulations. Ces descriptions, rédigées en XML, reposent sur

un ensemble de balises et d'attributs présentés dans la section 2.3.

2.1.2.1 Typage des blocs

Autant du point de vue de CoGen que de celui du développeur, nous distinguons trois catégories de blocs. Ces catégories sont nommées :

- les blocs initiaux (figure 2.1 bloc **camera_wrapper**) : ce type de bloc regroupe tous les composants dont le rôle est d'interfacer un périphérique source, tel qu'une caméra ou un convertisseur analogique-numérique, avec la chaîne. Les traitements réalisés se bornent à la conversion du flux entrant vers un flux compatible avec la chaîne, aucun autre traitement ne peut y être fait ;
- les blocs terminaux (figure 2.1 **composant**) : ils sont la réciproque des blocs initiaux, ils convertissent le flux de la chaîne vers un autre flux pour propager les données vers un microprocesseur ou vers des convertisseurs numérique-analogiques. Là encore, aucun traitement n'est effectué sur les données ;
- les blocs de traitements (figure 2.1 bloc **debug_port** et **wishbone**) : ils représentent un type de traitement correspondant à un algorithme. Ils définissent à la fois les points d'entrées et de sorties du bloc, ainsi que les caractéristiques globales (paramètres fournis par l'utilisateur). Comme présenté dans la section 1.6, un même algorithme peut présenter plusieurs solutions d'implémentation, certaines privilégiant la performance ou au contraire l'économie de ressources. Pour que CoGen puisse fournir la solution la plus adaptée en accord avec le flux et avec les ressources disponibles, un bloc de traitement peut disposer de plusieurs implémentations à la manière de greffons. Chaque description d'une implémentation donne les caractéristiques spécifiques de celle-ci. Le développeur souhaitant ajouter une nouvelle implémentation doit se conformer aux contraintes décrites dans le bloc de traitement. L'utilisateur n'a pas connaissance des diverses implémentations et ne manipule que les blocs qui décrivent l'algorithme de manière générique.

Chaque type de bloc est limité exclusivement aux traitements présentés ci-dessus. Ainsi un bloc initial ou terminal ne peut faire d'altération du contenu du flux et un bloc correspondant à un algorithme ne peut pas propager les données à l'extérieur du FPGA. Ce choix confère aux blocs un caractère générique. Par exemple, pour un bloc initial correspondant à une caméra couleur, si l'utilisateur souhaite avoir un flux uniquement en niveaux de gris, il devra ajouter un bloc de conversion, au lieu de devoir choisir (si la solution existe) le bloc initial de cette caméra avec une sortie en niveaux de gris. De la même manière, sans cette contrainte, il serait nécessaire de développer autant de blocs histogrammes, par exemple, qu'il n'y a de solutions pour propager les données en dehors du FPGA.

2.1.2.2 Interfaces des blocs

Tous les blocs ont leurs entrées-sorties (interfaces) typées (vidéo, complexe, entier, ...). A chaque type d'interface est associé un ensemble de signaux de contrôles et de données. Le bloc doit donc fournir impérativement ce jeu pour être considéré comme compatible CoGen.

Une interface comporte a minima un bus de données pour la transmission ou la réception de l'information, ainsi qu'un signal de validité (ou *enable*) dont le rôle est d'avertir le bloc qu'un traitement peut être démarré sur une nouvelle donnée. Ce signal reste à l'état haut (présence d'une nouvelle donnée) pendant exactement un cycle de l'horloge du FPGA quel que soit le débit du flux.

L'utilisation d'interfaces dédiées à la propagation d'un type de donnée et contenant un signal de validité permet de résoudre deux problèmes :

1. le signal *enable* permet de faire un ordonnancement simple et efficace entre blocs. Le bloc B ne démarre son traitement que lorsque le bloc A l'avertit de la présence d'une nouvelle information à traiter, permettant ainsi au bloc de s'adapter à la bande passante du flux entrant sans aucun paramétrage préalable ;
2. en posant la contrainte du respect de l'utilisation d'interfaces particulières, selon le type d'un bloc, il devient possible de connecter directement deux blocs entre eux sans ajout de bloc d'adaptation (*glue*). Rendre les blocs homogènes permet d'éviter une consommation inutile de ressources, un délai supplémentaire pour la connexion de deux blocs, et d'éviter de se trouver dans un cas où aucune glue n'existe et rend l'interconnexion impossible. Le seul cas où un bloc de conversion doit être ajouté est lorsque l'utilisateur souhaite propager les informations d'un type particulier en insérant, par exemple, un bloc terminal dédié à sortir un flux vidéo du FPGA. Dans ce cas, c'est à sa charge d'insérer le convertisseur adapté. Pour des blocs réalisés à partir d'outils de génération de code VHDL, le dévelopeur qui souhaite intégrer son bloc dans CoGen devra ajouter une couche d'encapsulation afin de le rendre compatible avec l'outil et répondre aux contraintes du code généré.

Pour illustrer ce choix, dans le cadre d'un bloc vidéo, l'interface correspondante (figure 2.4), présente un bus de données (*pixel_ data*), le signal *enable* (*pixel_ valid*) ainsi que deux signaux : le premier pour avertir d'une fin de ligne (*end_ of_ line*), le second pour avertir de la fin de l'image (*end_ of_ frame*). Cette interface fournit suffisamment d'informations, indépendamment du type de blocs (traitement ponctuel, sur une ligne ou sur une image dans sa globalité), pour éviter au développeur l'ajout de traitements dédiés à la connaissance de la position courante dans l'image (traitement redondant et consommateur de ressources).

2.2 Caractéristiques temporelles

La validation du bloc en rapport à l'aptitude à traiter un flux entrant ainsi que la production du flux en sortie du bloc nécessite une description comportementale de celui-ci. Trois informations

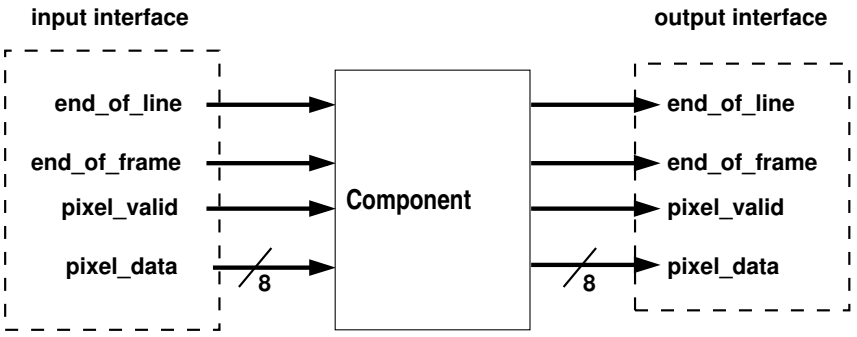


FIGURE 2.4 – Interface type pour un bloc recevant et produisant un flux vidéo.

sont nécessaires pour décrire les caractéristiques temporelles et comportementales d'un bloc : son débit maximum, appelé *acceptance*, qui caractérise la capacité à recevoir des données, sa *latence* qui est la durée entre l'entrée d'une donnée et la sortie du résultat correspondant, et sa *sortance* qui est la relation qui lie une donnée en entrée avec la propagation ou l'absence de propagation d'un résultat. Cet ensemble d'informations est utilisé pour :

- valider la capacité d'un bloc à traiter le flux entrant ;
- calculer le flux sortant en tenant compte des potentielles altérations induites par le bloc ;
- connaître les caractéristiques globales de la chaîne en analysant le flux sortant du dernier bloc ;
- garantir la cohérence de chaînes disposant de branches parallèles grâce principalement aux latences.

Toutes ces informations sont exprimées en nombre de cycles d'horloge, indépendamment de l'horloge du FPGA cible. Pour se convaincre de la pertinence de ce choix par rapport à une description faite par une fréquence ou une durée dans le cas de la latence, examinons trois cas :

1. la latence d'un bloc est induite par l'utilisation de registres synchrones, qui permettent de garantir que la durée d'un traitement ne dépasse jamais la période de l'horloge qui cadence le traitement. Décrire cette information par une durée n'est pertinent que pour une horloge particulière, le nombre de cycles d'horloge étant la seule information qui soit représentative et indépendante de la fréquence de l'horloge. L'information temporelle est obtenue par $latency = nb_cycles * \frac{1}{processing_freq}$, avec `latency` la latence exprimée par une durée, `nb_cycles` le nombre de cycles nécessaires pour produire un résultat et `processing_freq` la fréquence de l'horloge qui cadence le bloc ;
2. l'acceptance d'un bloc permet de déterminer le temps pendant lequel le bloc n'est pas en mesure de recevoir une nouvelle donnée parce qu'il est dans un état tel que celle-ci serait perdue ou qu'elle viendrait remplacer la précédente, entraînant une corruption du résultat. Ce cas de figure apparaît principalement quand le développeur, pour des raisons de réutilisation de composants internes comme les multiplieurs, réalise un ensemble de calculs avec un sous-

ensemble des composants dont il aurait réellement besoin. Ainsi, l'acceptance n'est-elle pas relative à une fréquence mais représente bien un nombre de cycles dans le traitement ;

3. l'information importante dans la sortance est la relation entre une donnée en entrée et l'état de la sortie. Par exemple, dans le cas d'un bloc réalisant une moyenne glissante sur n points, la sortance exprime le fait que pour les $n-1$ premières données, aucun résultat n'est propagé (temps pendant lequel le bloc accumule des données avant d'être en mesure de produire le premier résultat) puis que pour chaque nouvelle donnée arrivant un résultat est produit. Ce type d'information n'est pas exprimable sous la forme de fréquence.

Exprimer les caractéristiques temporelles par un nombre de cycles d'horloge n'est pas suffisant, comme le montre le point trois ci-dessus, mais aussi le cas de l'acceptance d'un bloc qui collecterait un ensemble de données avant de faire un traitement qui le rendrait indisponible. Il est nécessaire de proposer un mécanisme de description plus souple et plus riche. Dans ce travail, nous nous proposons d'exprimer ces deux informations à l'aide de motifs, des séquences de '1' et de '0'. Pour l'acceptance, le '1' signifie que le bloc est en mesure de recevoir une donnée et un '0' qu'il est occupé. Dans le cas de la sortance, le '1' signifie que la donnée correspondante de la liste d'entrée donnera lieu à la propagation d'une nouvelle donnée et le '0' qu'aucune production ne sera faite. Afin de simplifier la rédaction des acceptances et sortances, des symboles additionnels peuvent être utilisés :

- * représente une taille arbitraire dans le cas où le flux n'aurait pas une taille finie, par exemple dans le cas d'un convertisseur analogique-numérique. Dans le cas d'un flux vidéo, les dimensions de l'image seront préférées à ce symbole ;
- [] est utilisé pour décrire un sous-motif composé de caractères '0' et '1' ou d'un autre sous-motif ;
- *(n)* indique que le caractère ou le sous-motif placé juste devant est répété n fois. La valeur n peut être une valeur numérique ou une variable se trouvant au niveau de la description du bloc, de la chaîne ou de l'outil.

Ce formalisme peut être décrit en utilisant la grammaire BNF [43] (Backus-Naur Form) de cette manière :

```
REPEAT ::= (SUBPATTERN|SYMBOL) '(' ALPHANUM+ ')'
SUBPATTERN ::= '[' PATTERN ']'
PATTERN ::= SUBPATTERN | REPEAT | SYMBOL+
SYMBOL ::= ONE | ZERO
ONE ::= '1'
ZERO ::= '0'
```

Cette grammaire est convertie dans le modèle imposé par la bibliothèque utilisée pour réaliser l'expansion des motifs.

Il est à noter que pour les blocs initiaux et terminaux, l'acceptance et la sortance sont exprimées par rapport à leur propre fréquence d'horloge. Comme pour les autres blocs, CoGen va dans un

premier temps produire une expansion du motif, mais comme un changement de domaine d'horloge peut être nécessaire dans le cas où l'horloge du FPGA est utilisée pour les traitements, l'expansion est ensuite modifiée en ajoutant des temps de pause correspondant à un trou dans la transmission d'une donnée. Le nombre de "blancs", correspondant à ces pauses, qui doivent être ajoutés et leur position est déduite du *ratio* entre la fréquence du cœur du FPGA et celle du périphérique.

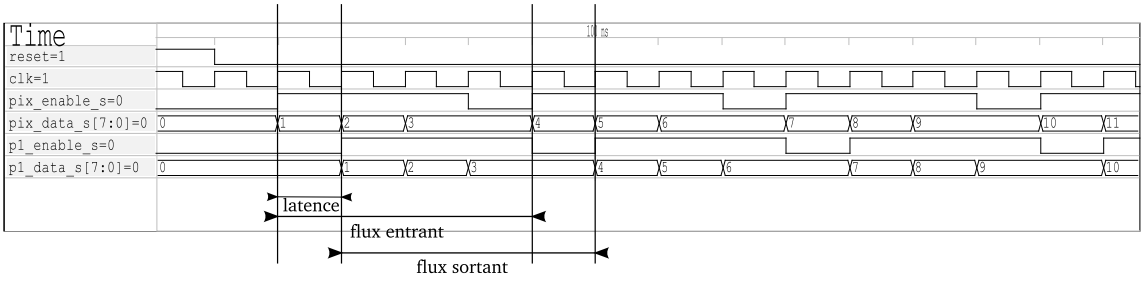


FIGURE 2.5 – Chronogramme de propagation des informations au travers d'un composant réalisant une tâche dont la durée entre l'arrivée d'une donnée et sa sortie correspond à un cycle d'horloge.

Afin d'illustrer l'écriture des motifs nous allons voir 2 exemples :

Exemple 1 : pour un bloc (figure 2.5) capable de recevoir des données et de produire un nouveau résultat à chaque cycle d'horloge avec une latence de 1 cycle, l'acceptance et la sortance sont toutes deux *1**, ce qui peut également être noté *[1]** ou *1(*)*.

Exemple 2 : en considérant un bloc de *cropping* qui supprime dans une image la ligne supérieure et inférieure ainsi que la colonne de gauche et celle de droite. Son acceptance reste la même que pour l'exemple 1. Sa sortance, par contre, est composée du sous-motif *0(IMG_ WIDTH)* pour la première et la dernière ligne, signifiant que les pixels de ces deux lignes ne sont pas propagés, d'un sous-motif *[01(IMG_ WIDTH-2)0](IMG_ HEIGHT-2)* indiquant que le premier pixel est perdu, que les *IMG_ WIDTH*-2 pixels suivant sont propagés et finalement que le dernier pixel de la ligne est supprimé. Ceci se répétant pour *WIDTH-2* lignes :

0(IMG_ WIDTH)[01(IMG_ WIDTH-2)0] (IMG_ HEIGHT-2) 0 (IMG_ WIDTH)

2.2.1 Expansion des motifs

La représentation sous la forme d'un motif n'est pas directement utilisée par CoGen : une expansion est réalisée donnant ainsi une liste de '0' et de '1', qui est exploitée dans les analyses temporelles (section 2.4.2.2 et 2.4.2.3) pour évaluer la capacité de réception mais également pour produire une nouvelle liste décrivant le forme du flux sortant du bloc. Cette expansion se fait en plusieurs étapes : en partant du motif du cropping *0(IMG_ WIDTH)[01(IMG_ WIDTH-2)0] (IMG_ HEIGHT-2) 0 (IMG_ WIDTH)*, deux étapes sont nécessaires :

2.2.1.1 Remplacement des variables par leur valeur

Avec :

- IMG_WIDTH = 3
- IMG_HEIGHT = 6

Le motif devient :
0(3)[01(3-2)0](6-2)0(3)

2.2.1.2 Génération de la liste

En utilisant un analyseur lexical tel que la bibliothèque `pyParsing`, et en utilisant les règles basées sur la grammaire BNF présentée plus tôt, la seconde étape consiste à produire la séquence de '0' et de '1' correspondant au motif. Dans le cas du motif précédent, cette séquence est :
000010010010010000

2.2.2 Retard dans la production de données

Pour certains types de traitements tel que celui présenté en 1.6.1(figure 1.2), la latence n'est pas suffisante pour exprimer le délai entre l'arrivée d'une donnée et la sortie du résultat correspondant. Il peut être également obligatoire de fournir une information relative à un retard temporel lié à la nécessité d'attendre un volume quelconque de données avant la production d'un résultat. Ce laps de temps est directement lié au flux, à la durée entre la réception de deux données consécutives et aux temps-mort potentiels.

Par exemple, dans le cas de la convolution par une matrice 3x3, pour produire le point **P1,1** le bloc doit attendre l'arrivée du point **P2,2** soit l'équivalent d'une ligne et d'un pixel. Ce retard n'est pas une latence puisque celle-ci exprime, si l'on compte à rebours, la durée entre la sortie d'un résultat et l'entrée de la donnée qui a déclenché cette production. D'autre part, il n'est pas possible d'exprimer ce retard en nombre de cycles d'horloge, car le temps requis pour l'obtention d'une ligne et d'un pixel est dépendant non seulement de la fréquence de la caméra (*pixelclock*), des temps de pause à la fin des lignes et, finalement, des potentiels traitements réalisés en amont tels qu'une décimation ou un cropping. C'est pourquoi cette information est donnée en parallèle de la latence et est exprimée par un nombre de données. CoGen, à partir du flux arrivant au bloc, pourra compter le nombre de cycle d'horloge correspondant.

2.2.2.1 Exemple

Pour un bloc ayant un retard de 5 données et avec un flux entrant en *[10100]**, soit une expansion partielle ne donnant que le nombre pertinent de données en *10100101001*, la durée pour obtenir les données sera de 11 cycles.

2.3 Description des composants

Les blocs sont caractérisés par un ensemble d'informations qui décrivent :

– les paramètres que l'utilisateur peut changer ;
– les interfaces d'entrées-sorties ;
– la nature, le type et le volume de blocs matériels requis ;
– les caractéristiques temporelles et comportementales.

Les deux premiers types d'information sont exposés à l'utilisateur pour la construction et le paramétrage de la chaîne. L'ensemble est requis pour les analyses réalisées par CoGen. CoGen ne manipulant pas directement le code HDL, ces caractéristiques sont mises à la disposition de l'outil par le biais de fichiers XML. Le choix du XML plutôt qu'un autre language de ce type se justifie surtout par le besoin d'une structuration précise et de la possibilité offerte par ce language de regrouper certaines informations dans une structure hiérarchique.

Cette arborescence est représentée sur la figure 2.6. Au premier niveau se trouve le type du bloc (initial, terminal ou traitement), au second niveau les blocs (un périphérique ou un algorithme particulier) et finalement au dernier niveau l'ensemble des implémentations liées à un bloc de niveau 2.

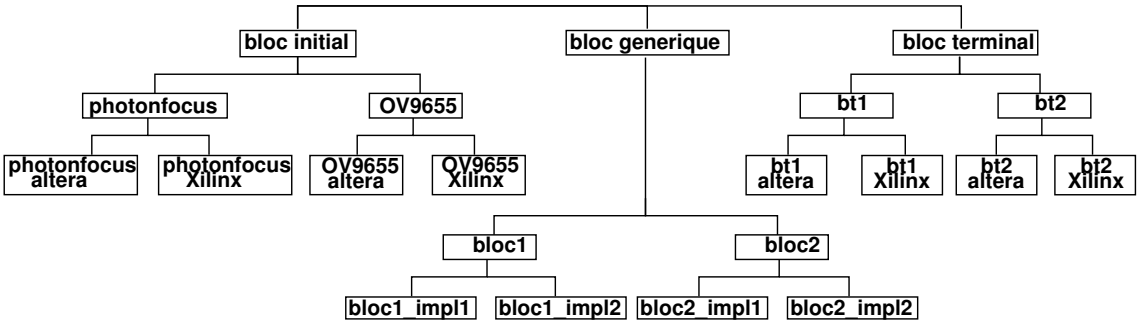


FIGURE 2.6 – Schéma de la structure hiérarchique des fichiers XML utilisés par CoGen dans le cadre des analyses.

En plus des fichiers relatifs aux blocs et à leurs implémentations, il apparaît des descriptions pour les plate-formes (cartes comportant le FPGA) qui contiennent les informations concernant la fréquence de l'horloge qui cadence le FPGA, les périphériques externes (RAMs, convertisseurs numériques-analogiques, ...) mais également le modèle du FPGA, ou la liste des FPGAs pouvant être installés. Pour chaque modèle de FPGA, un fichier recense les ressources qu'il contient (BRAM, multiplieur, ...).

2.3.1 Caractéristiques globales des fichiers XML

Chaque fichier possède un nœud racine qui identifie son type :
– `platform` pour les descriptions des cartes comportant le FPGA ;
– `fpga` pour la description d'un modèle de FPGA ;
– `initial_bloc` pour les blocs initiaux ;

- `terminal_bloc` pour les blocs terminaux ;
- `processing_bloc` pour les blocs de traitement ;
- `implementation_bloc` pour les blocs d'implémentation, ce type de fichier joue le rôle de greffon pour lier un bloc générique avec la ou les implémentation(s) de l'algorithme.

Cette balise contient un attribut `name` spécifiant le nom du bloc. Dans le cas des blocs initiaux, terminaux et des implémentations, ce nom correspond également à celui du composant POD correspondant. Elle contient également une sous-balise `description` utilisée pour la description du bloc.

Exemple pour un bloc de traitement réalisant un seuillage :

```
<?xml version="1.0" encoding="utf-8"?>
<processing_bloc name="threshold" >
  <description>Simple threshold block</description>
  ...
</processing_bloc>
```

2.3.2 Description du matériel

2.3.2.1 Description d'une plate-forme

Le fichier de description d'une plate-forme doit au minimum comporter :

- une balise `toolchain` pour donner l'outil nécessaire à la production du binaire dédié au FPGA ;
- un nœud `fpgas` comportant un attribut `main_clock` pour spécifier la fréquence du FPGA et un attribut `default_fpga` qui donne le modèle par défaut du FPGA installé sur la carte. Ce nœud peut disposer de sous-nœuds `fpga` qui fournissent la liste des FPGAs dont la plate-forme peut être équipée en lieu et place de celui par défaut.

Exemple pour une carte *SP_ VISION* de Armadeus Systems avec un FPGA par défaut de type spartan6lx100 cadencé à 100 MHz. Pour cette plate-forme les modèles lx45, lx75 ou lx150 sont également compatibles :

```
<?xml version="1.0" encoding="utf-8"?>
<platform name="sp_vision" version="0.1" >
  <description>ARMadeus Systems SP_VISION card</description>
  <toolchain>ise</toolchain>
  <fpgas main_clock="100000" default_fpga="spartan6lx100" >
    <fpga>spartan6lx45</fpga>
    <fpga>spartan6lx75</fpga>
    <fpga>spartan6lx150</fpga>
  </fpgas>
</platform>
```

2.3.2.2 Description d'un FPGA

Ce type de fichier de description fournit les caractéristiques matérielles d'un modèle particulier de FPGA. Il contient :

- un nœud `ressources`, contenant autant de sous-nœuds `ressource` que nécessaire pour spécifier les blocs câblés disponibles. Ce nœud contient les attributs :
  - `name` qui précise le nom de la ressource ;
  - `value` qui donne le volume disponible.
- un nœud `alias` utilisé lorsqu'un type de composant joue un rôle plus générique. Chaque sous-nœud `ressource` contient deux attributs : le premier `name` correspond au nom de la ressource, le second `alias` au nom du composant du FPGA.

Exemple pour un Spartan6 LX150 de Xilinx disposant entre autres de 180 blocs DSP48 (pouvant également être utilisés comme de simples multiplieurs), 268 BRAMs et 6 PLLs :

```
<?xml version="1.0" encoding="utf-8"?>
<fpga name="spartan6lx100" version="0.1" >
  <ressources>
    <ressource name="DSP48" value="180" />
    <ressource name="BRAM" value="268" />
     [...]
    <ressource name="PLL" value="6" />
  </ressources>
  <alias>
    <ressource name="multiplier" alias="DSP48" />
  </alias>
</fpga>
```

2.3.3 Informations communes à plusieurs types de blocs

2.3.3.1 Interface maître et esclave

Chaque bloc doit préciser ses interfaces de communications, entrantes ou sortantes.

À l'exception des blocs initiaux, un bloc possède au moins une interface entrante (esclave) utilisée pour la réception du flux de données issues du bloc précédent et généralement une interface sortante (maître), hormis pour les terminaux, pour la propagation des résultats du traitement du bloc.

La balise `bloc_interfaces` contient la liste de toutes les interfaces du bloc. Chaque balise `interfaces` doit fournir les informations suivantes :

- `dir` : les arguments `in` ou `out` donnent la direction de l'interface.
- `type` : `video` ou `complex` (interface contenant un bus pour la partie réelle, un pour la partie imaginaire), pour préciser le type de l'interface et donc, par extension, le jeu de signaux/bus de celle-ci.

Par exemple pour un filtre en niveau de gris recevant un pixel et produisant un résultat du même type :

```
<?xml version="1.0" encoding="utf-8"?>
<implementation_bloc name="threshold_impl1" >
  ...
  <bloc_interfaces>
    <interfaces dir="in" type="video">
      <interface name="if_in" />
    </interfaces>
    <interfaces dir="out" type="video">
      <interface name="if_out" />
    </interfaces>
  </bloc_interfaces>
  ...
</implementation_bloc>
```

Certains traitements ou algorithmes ne sont, par nature, pas typé et par extension ne fournissent pas d'attribut `type` pour leurs interfaces d'entrée et de sortie. La décimation, par exemple, peut être aussi bien appliquée sur un flux de réels issus d'un convertisseur analogique-numérique que sur un flux de complexes. Dans le cas où le bloc de haut-niveau ne fournit pas cette information, ce sont les implémentations qui doivent le faire. Ce mécanisme permet de présenter à l'utilisateur, non pas un ensemble de versions du même algorithme, mais un seul bloc. La sélection de l'implémentation présentant l'interface adéquate vis-à-vis du bloc précédent incombe à CoGen.

D'autres attributs optionnels peuvent également apparaître dans ce nœud :

- `chan` : permet, dans le cas d'un flux vidéo couleur, de guider CoGen pour la connexion automatique des interfaces. Cet attribut peut prendre les valeurs `RED`, `GREEN` ou `BLUE`.

Exemple pour un filtre couleur recevant et sortant un pixel :

```
<bloc_interfaces>
  <interfaces dir="in" type="video">
    <interface name="if_r_in" chan="RED" />
    <interface name="if_g_in" chan="GREEN" />
    <interface name="if_b_in" chan="BLUE" />
  </interfaces>
  <interfaces dir="out" type="video">
    <interface_name="if_r_out" chan="RED" />
    <interface_name="if_g_out" chan="GREEEN" />
    <interface_name="if_b_out" chan="BLUE" />
  </interfaces>
</bloc_interfaces>
```

Dans le cas où une balise `interfaces` contient plusieurs balises `interface` mais que celles-ci ne contiennent pas d'attributs `chan`, elles sont considérées comme recevant des données consécutives.

Voici un exemple pour un bloc initial (pas d'entrée) de type vidéo fournissant deux pixels consécutifs :

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<initial_bloc name="cameralink">
  ...
  <bloc_interfaces>
    <interfaces dir="in" type="video">
      <interface name="if1_in" />
      <interface name="if2_in" />
    </interfaces>
  </bloc_interfaces>
  ...
</initial_bloc>
```

Si, pour un même bloc, plusieurs balises `interfaces` sont définies avec la même direction (attributs `dir` identiques) les données sont considérées comme étant parallèles.

Par exemple, pour un filtre en niveaux de gris recevant deux pixels issus de deux traitements parallèles :

```
<?xml version="1.0" encoding="utf-8"?>
<implementation_bloc name="canny_impl1">
  ...
  <bloc_interfaces>
    <interfaces dir="in" type="video">
      <interface name="if1_in" />
    </interfaces>
    <interfaces dir="in" type="video">
      <interface name="if2_in" />
    </interfaces>
    <interfaces dir="out" type="video">
      <interface name="if_out" />
    </interfaces>
  </bloc_interfaces>
  ...
</implementation_bloc>
```

Le fait qu'un bloc dispose de plusieurs interfaces entrantes ayant la même direction pour recevoir plusieurs données en parallèle est une caractéristique de l'algorithme. Le `Laplacien DOG` de `Marr-Hildreth` (1980), qui est un filtre de détection de contours, prend en entrée le résultat de deux lissages gaussiens de tailles différentes appliqués sur la même image et réalise la différence. Il est ainsi logique que cette information soit portée par le fichier de description du bloc de traitement. Par contre, le fait qu'un bloc reçoive ou produise des données consécutives en parallèle est un cas particulier. Généralement ce type de situation se rencontre pour un bloc initial lié à un périphérique d'acquisition qui présente cette caractéristique (telle que la caméra présentée en section 4.2 et le convertisseur analogique-numérique décrit en section 5.1) et pour des implémentations liées aux mêmes périphériques ou permettant de pouvoir traiter un flux important à une fréquence d'horloge faible. C'est pourquoi, dans le cas des algorithmes, seules les implémentations peuvent spécifier des interfaces pour des données consécutives en parallèle.

Une balise `interfaces` dispose également d'un attribut `target` exclusivement utilisé pour les blocs d'implémentation. Cet attribut permet à CoGen de faire le lien entre chaque interface de l'implémentation et celles du bloc de haut-niveau.

2.3.3.2 Informations relatives aux caractéristiques temporelles d'un bloc

Les blocs initiaux, terminaux et les implémentations doivent fournir une balise `timings`. Elle contient des éléments de type `timing` qui spécifient

- `dir` : `in` ou `out`
- `freq` : notée en MHz pour l'acceptance ou la fréquence de production de données (exclusivement utilisé pour les blocs initiaux et terminaux) ;
- `pattern` : qui donne la représentation sous la forme d'un motif de l'acceptance ou de la sortance du bloc. Dans le cas des blocs initiaux, cette information est définie par rapport à la fréquence propre du périphérique.

Des attributs optionnels sont également disponibles :

- `latency` : uniquement disponible pour les balises `timing` en mode `out`. Il représente le nombre de cycles d'horloge entre l'arrivée d'une donnée et la sortie de la donnée correspondante. Cette information n'est pas disponible pour les blocs initiaux ;
- `delay` : est utilisé pour les blocs tels que les convolutions qui présentent un retard, qui n'est pas lié au temps du traitement, entre l'arrivée d'une donnée et la production du résultat correspondant (problématique présentée dans les sections 1.6.1 et 2.2.2)

La balise `timings` contient au plus un élément dont la direction est `in` et un élément dont la direction est `out` :

```
<timings>
  <timing dir="" pattern="" [freq=""] />
</timings>
```

Une implémentation doit avoir une balise `timing` par direction, chacune d'elle est décrite par un motif. Un bloc initial ne posséde qu'une seule balise au format `out`, et un bloc terminal, une seule au format `in`.

La balise `timings`, ainsi que la sous-balise `timing` et l'ensemble des attributs ne sont pas modifiables par l'utilisateur.

Exemple pour une caméra ayant un débit de pixels de 80 MHz et 8 cycles de temps de pause à la fin de chaque ligne :

```
<?xml version="1.0" encoding="utf-8"?>
<initial_bloc name="cameralink" >
  ...
  <timings>
    <timing dir="out" freq="80" pattern="[1(IMG_WIDTH)0(8)](IMG_HEIGHT)" />
  </timings>
  ...
```

```
</initial_bloc>
```

Exemple pour un composant acceptant toujours des informations, ne produisant qu'une donnée sur deux et dont la latence est de 1 cycle d'horloge :

```
<?xml version="1.0" encoding="utf-8"?>
<implementation_bloc name="threshold" >
  ...
  <timings>
    <timing dir="in" pattern="1*" />
    <timing dir="out" latency="1" pattern="[01](IMG_WIGTH*IMG_HEIGHT)" />
  </timings>
  ...
</implementation_bloc>
```

2.3.3.3 Utilisation des ressources

Les blocs initiaux, terminaux et les implémentations doivent fournir le type et le volume de ressources matérielles qu'ils utilisent.

Les balises `resource` incluses dans la balise `resources` possèdent deux attributs :

- `type` : pour le type de ressource utilisé ;
- `amount` : pour le nombre de ressources du type `type` concerné.

Exemple pour le bloc cameralink :

```
<?xml version="1.0" encoding="utf-8"?>
<initial_bloc name="cameralink" >
  ...
  <resources>
    <resource type="serdes" amount="8" />
    <resource type="PLL_ADV" amount="1" />
  </resources>
  ...
</initial_bloc>
```

2.3.4 Configuration d'un bloc générique

Certaines informations sont directement liées au bloc. Généralement ce sont des informations relatives à la spécification de l'algorithme. Les implémentations de ce bloc n'ont aucun impact sur ces informations.

2.3.4.1 Paramètres d'un bloc

Les paramètres d'un bloc doivent être fourni par l'utilisateur. Ils correspondent à des informations dont le bloc, et par extension toutes les implémentations, ont besoin pour le traitement visé.

La balise `user_config` regroupe l'ensemble des éléments que l'utilisateur doit fournir. Chaque balise `param` contient au minimum l'attribut `name` qui correspond au nom du paramètre et peut

contenir l'attribut `default` pour spécifier la valeur par défaut de ce paramètre.

```
<user_config>
  <param name="" default="" />
  ...
  <param name="" default="" />
</user_config>
```

Exemple pour le seuillage :

```
<user_config>
  <param name="threshold" default="128" />
</user_config>
```

2.3.4.2 Liste des implémentations pour un bloc donné

Un bloc peut disposer d'une ou plusieurs implémentation d'un algorithme, il en informe CoGen par une liste de répertoires.

Chaque balise `implementation` contenue dans la balise `implementations` correspond au répertoire contenant l'implémentation du bloc concerné.

```
<implementations>
  <implementation dir="" />
  ..
  <implementation dir="" />
</implementations>
```

2.4 Analyse des chaînes

Une fois l'environnement spécifié par l'utilisateur (plate-forme cible, modèle de FPGA et source de données), CoGen réalise un ensemble de vérifications concernant les blocs et leurs implémentations. Ces analyses sont réalisées pendant la construction de la chaîne pour le premier groupe et une fois la chaîne entièrement construite pour le second. Le fonctionnement global est représenté figure 2.7. L'ensemble des analyses peut être groupé en deux ensembles :

- les analyses statiques (section 2.4.1) qui visent à s'assurer qu'un bloc peut être utilisé dans l'environnement cible (compatibilité matérielle) et de la cohérence de la chaîne (validation du typage des blocs entre eux) ;
- les analyses dynamiques qui sont réalisées une fois la chaîne complètement spécifiée et qui ont pour but de générer l'ensemble des solutions acceptables (chaînes uniquement constituées d'implémentations compatibles avec le matériel). Elles vérifient que chaque bloc est en mesure de traiter le flux issu du précédent, que les besoins globaux en ressources matérielles ne dépassent pas les ressources disponibles et s'assurent dans le cas d'une chaîne comportant des branches parallèles que celles-ci sont synchrones.

A l'issue de ces analyses, CoGen est conçu pour sélectionner, selon les critères fournis par l'utilisateur (réduction des latences/durée du traitement, ou économie de ressources), la solution

la plus adaptée. Cette chaîne sera ensuite décrite dans un fichier de script, donnant toutes les instructions pour la construction du projet dédié à l'outil du fondeur et à la production du binaire à destination du FPGA cible.

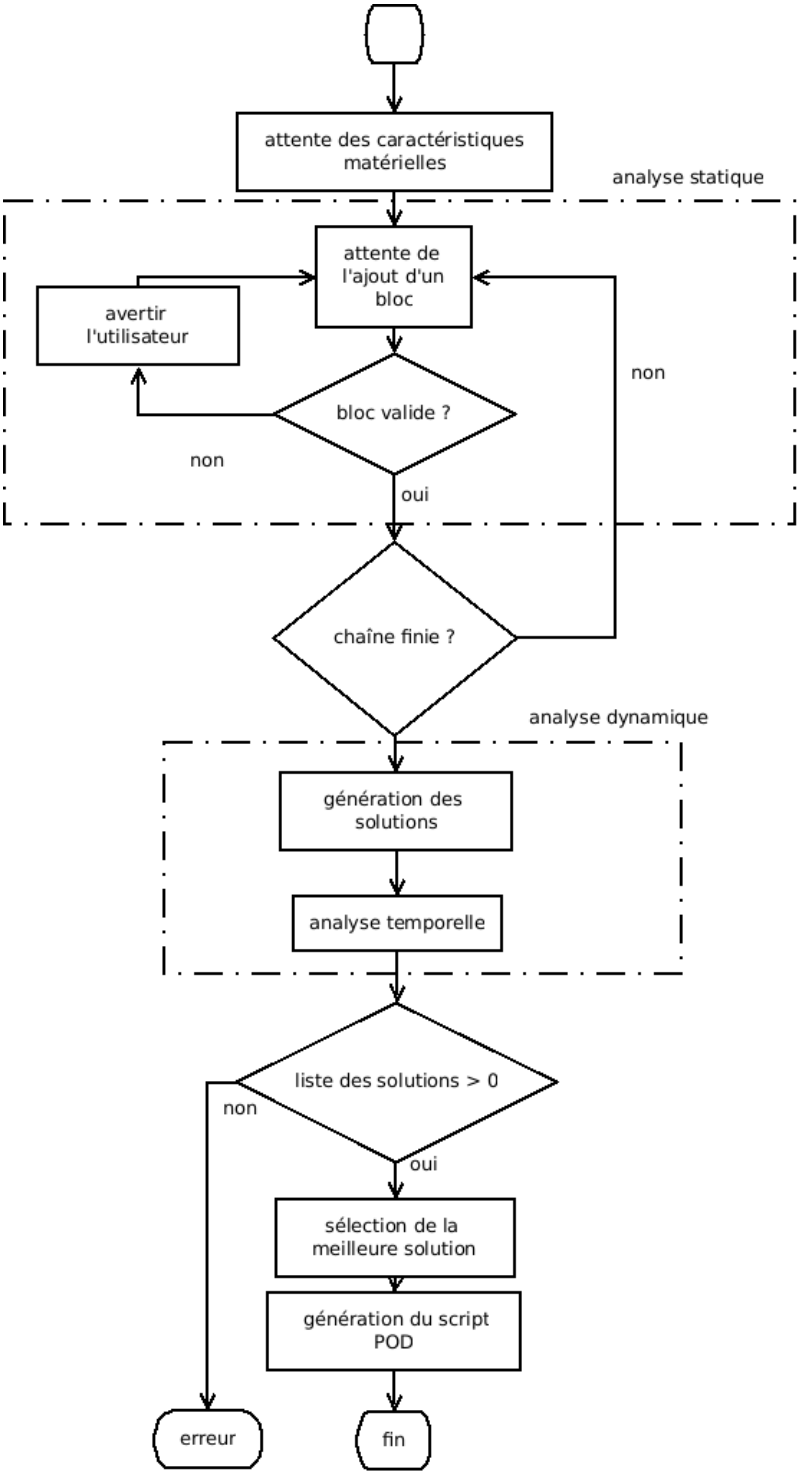


FIGURE 2.7 – Diagramme de fonctionnement global de CoGen.

2.4.1 Analyse statique

L'analyse statique se déroule pendant la construction de la chaîne et comporte deux traitements.

2.4.1.1 Blocs admissibles

Lors de l'ajout d'un bloc dans la chaîne (et avant sa connexion), l'ensemble des implémentations HDL de celui-ci sont analysées. Si une implémentation nécessite un type de ressource matérielle non disponible sur le FPGA cible, ou que le volume nécessaire dépasse le volume disponible (sans prise en compte à ce stade des besoins globaux de l'ensemble des autres blocs), l'implémentation est supprimée. Dans le cas contraire, l'implémentation est considérée comme admissible. A l'issue de l'analyse d'un bloc de traitement, s'il apparaît que celui-ci ne dispose d'aucune implémentation admissible, le bloc est considéré comme incompatible et une erreur est remontée à l'utilisateur. Cette situation peut se présenter lors de l'usage de blocs tels que ceux présentés dans les sections 2.6 et 4.5 qui font usage, pour réduire les latences, de multiplieurs de type `DSP48` inclus entre autres dans les `Spartan6`, mais absents des `Spartan3`. Un tel bloc ne peut donc être exploité sur ce dernier.

2.4.1.2 Cohérence des blocs

Une fois le bloc ajouté au projet, l'utilisateur doit l'insérer dans la chaîne. Pour chaque connexion de celui-ci avec un autre bloc, CoGen doit s'assurer que le type des interfaces des deux blocs (par comparaison de la sortie du précédent avec l'entrée de celui en cours d'insertion ou comparaison de la sortie de ce dernier avec l'entrée du suivant) sont bien du même type. Dans le cas où l'utilisateur tente de connecter deux blocs dont les interfaces diffèrent, une erreur est remontée. L'utilisateur doit soit supprimer ce bloc du projet et ajouter un bloc équivalent mais d'un type adéquat, soit ajouter un bloc de conversion pour adapter le flux.

2.4.1.3 Cohérence des branches parallèles

Lorsqu'une chaîne comporte plusieurs branches parallèles, si celles-ci se rejoignent au niveau d'un bloc, il est nécessaire de vérifier que les deux flux ont des caractéristiques communes (taille de l'image, ou facteur de décimation). Connaissant les dimensions du flux, CoGen doit en déterminer la cohérence. Dans le cas d'un traitement vidéo, ces informations passent par les variables *IMG_HEIGHT* et *IMG_WIDTH*, tandis que pour les traitements sur un flux non borné, c'est la valeur de '*' qui sert de taille. Dans tous les cas, cette ou ces variable(s) doivent être identiques pour toutes les branches. Dans le cas contraire, l'utilisateur est averti de cette différence. Toutefois ce n'est pas considéré comme une erreur, l'utilisateur pouvant volontairement faire ce choix.

2.4.2 Analyse temporelle et comportementale

Cette analyse est lancée par l'utilisateur une fois la chaîne entièrement construite. Elle sert à produire l'ensemble des combinaisons possibles d'implémentations et pour chacune d'elles, à s'assurer de la capacité des blocs à recevoir le flux du bloc précédent, à garantir la synchronisation des branches parallèles et à vérifier que les besoins en ressources ne dépassent pas le volume disponible sur le FPGA.

Afin d'éviter de multiplier les parcours des chaînes, l'ensemble des traitements se fait séquentiellement pour chaque bloc rencontré.

Le principe général est le suivant : avant le début de l'analyse, la chaîne construite par l'utilisateur est placée dans la liste des chaînes devant être analysées. CoGen prend une chaîne à traiter et va réaliser un parcours d'arbre en profondeur en partant de la racine (le bloc initial), en suivant tous les nœuds (blocs de traitements) pour arriver à la ou les feuille(s) (blocs terminaux). Pendant le parcours, pour chaque nœud, plusieurs traitements sont réalisés :

- CoGen s'assure que le bloc source et le bloc destination ont le bon nombre d'interfaces (cas de la propagation de données en parallèle). Il s'assure également que la taille des bus de données sont bien identiques (il n'est pas possible de calculer à priori cette information qui tend à évoluer selon le type du traitement de chaque bloc). Si la destination présente un nombre de données en parallèle supérieur à celui de la source, la solution est stockée pour pouvoir être modifiée et analysée si aucune solution simple n'est trouvée. En effet, il est possible d'ajouter un bloc transformant un flux séquentiel en un autre flux propageant plusieurs données consécutives en parallèle : cette solution permet d'ajouter des temps-morts dans la chaîne, relâchant les contraintes de certains blocs n'ayant pas une acceptance en 1(*) ;
- si le bloc courant dispose de plusieurs entrées issues de branches distinctes, CoGen s'assure que toutes ces branches ont été parcourues et qu'elles ont les mêmes latences. Si au moins une des branches n'a pas été parcourue, CoGen remonte dans l'arborescence pour réaliser le parcours des autres branches précédant le bloc en cours d'analyse, sinon il s'assure que toutes les latences sont identiques en rajoutant au besoin des blocs de retards pour assurer la synchronisation (section 2.4.2.1) ;
- si le bloc est de type générique, CoGen va le remplacer par un des blocs d'implémentation. Si le bloc générique dispose de plus d'une implémentation, la chaîne est dupliquée autant de fois que nécessaire et pour chaque copie, une implémentation va remplacer le bloc générique. Chacune des nouvelles chaînes est ensuite stockée dans le tableau des chaînes qui doivent être analysées ;
- CoGen réalise ensuite une étude des ressources en ajoutant au volume des ressources déjà consommées, les besoins du bloc en cours, puis s'assure qu'aucune d'elles ne dépasse le volume disponible. S'il y a dépassement, la chaîne est rejetée et le parcours de la suivante est engagé.
- l'étude suivante consiste à vérifier l'aptitude du bloc à recevoir le flux de celui (ou ceux)

qui le précéde (section 2.4.2.2). Si le bloc courant s'avère trop lent par rapport au débit d'entrée, la chaîne n'est pas directement rejetée mais mise dans une autre liste pour une autre analyse. S'il apparaît qu'aucune chaîne n'est trivialement admissible, plusieurs solutions sont envisageables :

- modifier la fréquence de l'horloge pour changer le *ratio*. En effet, si un bloc cadencé à 100 MHz produit une donnée par cycle d'horloge et que le suivant n'est en mesure de recevoir qu'une donnée tous les deux cycles, si celui-ci est cadencé à 200 MHz, il y aura bien une réception tous les deux cycles ;
- paralléliser le flux de données pour les distribuer sur plusieurs instances du même bloc ;
- altérer le flux pour adapter les cycles de propagation et les blancs du flux.

– la dernière étape est la génération du flux de sortie à partir du flux d'entrée et de la représentation de la sortance du bloc.

Si CoGen a parcouru la totalité des blocs, la chaîne est considérée comme admissible.

Si, à l'issue du parcours de toutes les chaînes à traiter, aucune chaîne n'est admissible, deux cas se présentent :

1. des chaînes mises de côté (car présentant des problèmes d'acceptance) sont disponibles pour un traitement visant à tenter par une modification du flux de rendre compatibles les blocs posant problème ;
2. soit une erreur est remontée à l'utilisateur.

2.4.2.1 Calculs des latences et traitement des branches parallèles

Le calcul de la latence globale de la chaîne ou d'un bloc a deux buts :

1. elle est utilisée pour comparer les chaînes lorsque plus d'une solution est admissible. Le critère fournit par l'utilisateur (vitesse ou économie de ressources), va permettre de réaliser le choix ;
2. une chaîne de traitement peut disposer de plusieurs branches parallèles. Si celles-ci se rejoignent au niveau d'un bloc, il est nécessaire de garantir que l'ensemble des branches sont synchrones. En se basant sur la latence (somme des latences de tous les blocs qui précédent le bloc en cours de traitement) de chaque branche, il est ainsi possible d'ajouter des blocs de délai pour équilibrer les latences.

Calcul des latences

Hormis les blocs initiaux et terminaux, toutes les implémentations ont une information de latence et, de maniére optionnelle, une information de retard décrites dans le fichier XML associé.

Par défaut, un bloc de traitement ou une implémentation qui n'a pas encore été analysé présente une latence globale de -1 : cette valeur permet de lever une ambiguïté et de déterminer facilement si le bloc a été traité ou non, cette information est utile notamment dans les branches parallèles (section 2.4.2.1).

La latence globale d'un bloc est l'accumulation des latences de tous ces prédécesseurs dans la chaîne. CoGen, lors du parcours d'une chaîne, pour un bloc donné, affecte à celui-ci la somme des latences des blocs précédents, à laquelle s'ajoute sa propre latence ainsi que le retard, si présent, après conversion en nombre de cycles d'horloge. Dans la figure 2.8(a), par exemple, le bloc **impl2a** a une latence globale de 4, correspondant à sa latence propre (2) plus la latence globale du bloc **impl1a** (2), qui elle même découle du bloc précédent.

Dans le cas où un bloc disposerait de plusieurs entrées (**impl3** figure 2.8), si au moins une des entrées de ce bloc présente une latence globale de -1, signifiant qu'il n'a pas été traité, alors CoGen ne poursuit pas le traitement, mais remonte la chaîne pour traiter les autres branches (figure 2.8(b)), avant d'être en mesure de poursuivre le parcours en profondeur (figure 2.8(c)).

Équilibrage des branches

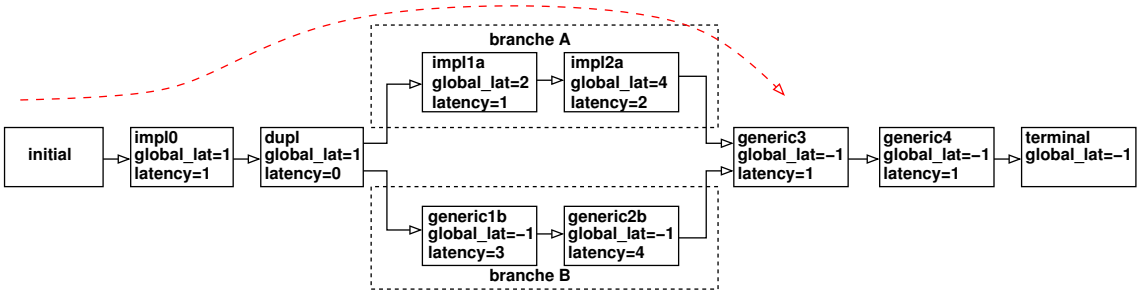
Lorsque CoGen accède à un bloc connecté à plusieurs branches et que celles-ci ont toutes été traitées (toutes les branches ayant une latence différente de -1) comme dans le cas de la figure 2.8(a), CoGen cherche le maximum de la somme des latences et des retards de chaque branche. Ce maximum devient la référence et les autres branches, si leur latence est différente du maximum, se voient affectées de blocs de délai dont la valeur est la différence entre le maximum et la latence de la branche (branche A figure 2.8(c)).

Exemple d'analyse d'une chaîne disposant de branches parallèles

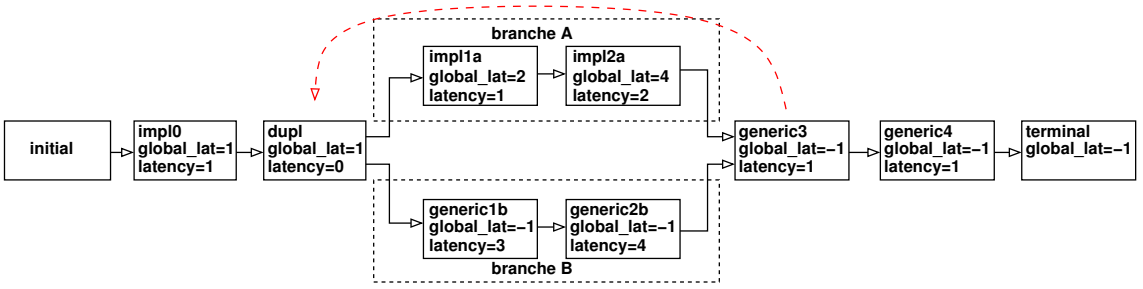
La figure présente une chaîne disposant de deux branches parallèles. CoGen, en partant du bloc initial réalise un parcours en profondeur pour couvrir l'ensemble des blocs. Lorsqu'il arrive sur le bloc **dupl**, une des deux branches est parcourue en premier (figure 2.8(a)). Lors du traitement de **impl3**, disposant de 2 entrées, il apparaît que l'ensemble des branches précédant ce bloc n'a pas été traité. En conséquence il n'est pas possible de fixer la latence globale de **impl3**, ni de continuer le parcours en profondeur. CoGen va remonter dans la hiérarchie des blocs, jusqu'à trouver une bifurcation (figure 2.8(b)), afin de compléter le traitement des blocs en aval (figure 2.8(c) brancheB). Après le traitement de **impl2b** (figure 2.8(c)), CoGen parvient à nouveau sur le bloc **impl3**. À ce moment, toutes ses entrées ont été traitées : il est maintenant possible de réaliser l'équilibrage des branches. La branche A dispose d'une latence globale de 4, alors que la branche B présente une latence de 8. CoGen ajoute donc dans la branche A un bloc de délai de $8 - 4 = 4$ cycles. Les branches étant équilibrées, il devient possible de calculer la latence globale de **impl3** et de continuer le parcours en profondeur (fig 2.8(d)) jusqu'au bloc terminal.

2.4.2.2 Validation de l'acceptance des blocs

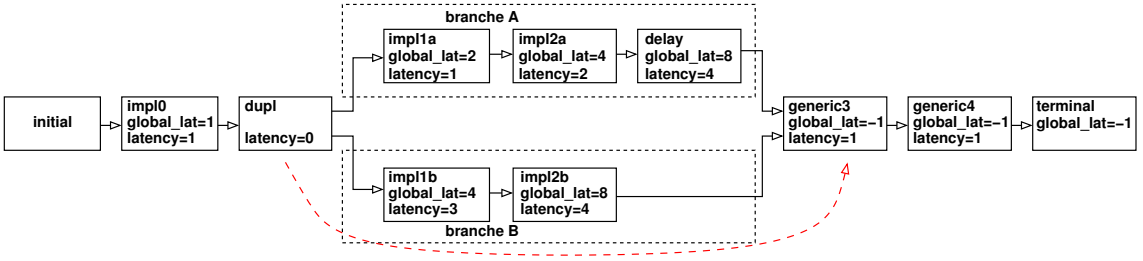
La validation de l'acceptance de chaque bloc représente l'analyse la plus importante visant à garantir l'intégrité du flux. Le modèle d'ordonnancement choisi pour CoGen repose sur un déclenchement du traitement de proche en proche. Ainsi le bloc A produit une donnée qui est reçue par le bloc B. Cette donnée reçue représente le stimulus qui va entraîner le démarrage du traite-



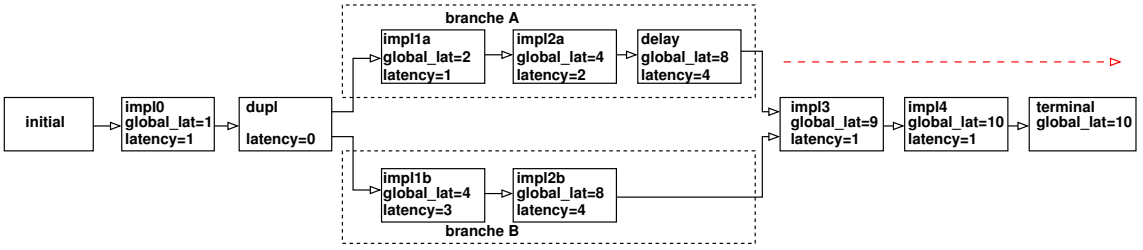
(a) CoGen a réalisé le parcours de la branche A mais n'est pas en mesure de vérifier l'équilibre des branches car la branche B n'as pas été traitée et présente donc une latence globale de −1.



(b) Ne pouvant continuer le parcours en profondeur, CoGen est reparti sur le bloc de duplication le plus proche afin de parcourir les autres branches.



(c) La branche B est maintenant traitée, CoGen est désormais en mesure de vérifier le total des latences et d'ajouter un bloc de délai si nécessaire, puis de calculer la latence globale du bloc **impl3**.



(d) Le parcours de la chaîne est entièrement finie

FIGURE 2.8 – Parcours d'une chaîne présentant des branches parallèles.

ment du bloc B. Contrairement aux techniques mises en œuvre sur processeur, telles que celle du producteur-consommateur qui repose sur des attentes et des synchronisations, dans le cas d'un flux imposé par un périphérique, il n'est pas possible de simplement retarder la propagation d'une don-

née. Si l'implémentation d'un traitement nécessite un temps d'indisponibilité trop long, les données qui pourraient être transférées seraient irrémédiablement perdues. Ainsi il est impératif de pouvoir garantir, à priori, par une connaissance du flux entrant dans chaque bloc et des caractéristiques de l'implémentation, l'aptitude de réception afin de ne pas risquer de pertes ou de corruptions de données. Comme présenté en introduction de 2.4.2, si une implémentation présente une acceptance en inadéquation avec le flux entrant, la solution est considérée comme inadaptée et est mise de côté pour être éventuellement analysée à nouveau, s'il n'y a pas de solution trivialement exploitable, après modification du format du flux.

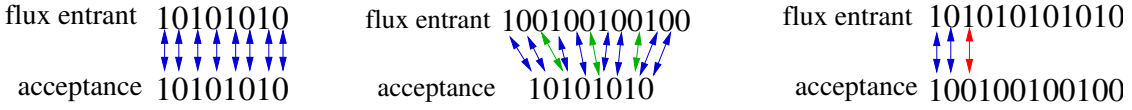
Cette analyse repose sur un principe de simulation du comportement d'un bloc vis-à-vis du flux issu du bloc précédent, permettant ainsi de tenir compte pour chaque traitement des évolutions ou altérations éventuelles appliquées en amont. Elle est faite par comparaison entre le flux entrant et l'expansion de son acceptance. Dans cette comparaison, les états successifs du bus d'entrée représentés par des caractères, tel que présenté en 2.2, sont interprétés comme des *stimuli*, un '1' signifiant l'arrivée d'une donnée, va déclencher un traitement et donc faire évoluer l'état du bloc. Dans le cas de l'absence de donnée ('0'), l'état du bloc va dépendre exclusivement de son état courant à ce pas de temps : si le bloc est en attente, il reste dans ce même état ; si le bloc est en train de réaliser son traitement alors le nouvel état va dépendre du caractère suivant dans l'expansion de son acceptance. L'algorithme 1 présente ce traitement :

À un pas de temps particulier (correspondant à un caractère dans la chaîne d'entrée), CoGen regarde l'état du bloc (correspondant au caractère à un *offset* particulier dans la liste de l'acceptance). Par cette analyse il est en mesure de savoir s'il y a concordance entre l'état de l'entrée et celui du bloc. Cette analyse peut se rapporter à la simulation du flux pour évaluer l'état du bloc en accord avec les *stimuli* représentés par les données propagées par le bloc précédent :

- si la valeur dans l'entrée est un '1' et si la valeur de l'acceptance est également un '1' (pas de temps 1 sur figures 2.9(a), 2.9(b), 2.9(c)), cela signifie que le bloc est disponible pour recevoir et traiter la donnée qui est en cours de réception. Le pas de temps (flux entrant) est avancé pour faire évoluer temporellement la simulation, ainsi que l'état du bloc (*offset* dans l'acceptance) car l'arrivée d'une donnée est le déclencheur du traitement ;
- si les valeurs dans les deux listes sont '0' (pas de temps 2 sur les figures 2.9(a), 2.9(b), 2.9(c)) cela signifie que le bloc est occupé. Là encore il y a concordance entre l'acceptance et le flux car aucune donnée ne rentre dans le bloc. Les deux index sont avancés, car contrairement à l'état d'attente, l'état actif n'est pas un état stable ;
- la liste d'entrée contient un '0' et l'acceptance est égale à '1' (pas de temps 3 sur la figure 2.9(b)), cela signifie que le bloc est en attente de la réception d'une donnée, mais comme aucune donnée n'arrive, le bloc reste inactif et seul le pas de temps est incrémenté ;
- si finalement une donnée est reçue mais que le bloc est occupé (figure 2.9(c) au pas de temps 3), alors il y a non concordance entre la forme du flux et l'acceptance du bloc : le bloc

n'est donc pas en mesure de traiter le flux. La chaîne est donc déplacée pour être traitée si nécessaire (cas de l'absence de chaîne admissible).

Si le parcours des listes se termine avec succès, le bloc est considéré comme capable de traiter le flux et le traitement suivant est réalisé.



(a) Flux identique à l'acceptance d'un bloc

(b) Flux entrant plus lent que l'acceptance

(c) Flux entrant incompatible avec l'acceptance

FIGURE 2.9 – Divers cas de comparaison entre un flux et une acceptance.

2.4.2.3 Construction de la représentation du flux de sortie à partir du flux d'entrée et de l'information de sortance du bloc

Le traitement relatif à la validation de l'acceptance d'une implémentation (2.4.2.2) nécessite de disposer de la représentation du flux de sa source. Deux cas se présentent :

1. un bloc initial : la sortie est directement disponible car fournie sous la forme d'un motif, aucun traitement autre que l'expansion du motif n'est nécessaire ;
2. un bloc de traitement : dans ce cas, il est nécessaire de générer une représentation de son flux de sortie en tenant compte de son flux d'entrée et de sa sortance.

Pour un bloc quelconque de la chaîne, le génération de la représentation du flux de sa sortie dépend du flux ainsi que de l'expansion de sa sortance. La latence du bloc ne rentre pas en ligne de compte car c'est une constante pour toutes les données qui transitent dans un bloc et ce retard n'est réellement visible que pour la première donnée.

Comme présenté dans la section 2.2, la sortance d'un bloc est exclusivement une information qui lie une donnée en sortie avec la donnée correspondante en entrée. Ce motif ne contient pas l'information temporelle qui est, elle, portée par le flux entrant dans le bloc.

Le traitement (algorithme 2) repose sur le même principe de simulation dans laquelle une itération est réalisée sur le flux entrant, afin de générer la représentation du flux sortant. Pour chaque caractère (ou pas de temps) quatre cas peuvent se présenter :

1. le caractère vaut '0'. Cela représente un temps-mort. Le bloc n'étant pas stimulé, il n'y aura pas de production d'un résultat. Un '0' est reporté dans la liste de sortie, la position dans l'expansion de la sortance n'est pas incrémentée (figure 2.10(b) pas de temps 2, 4, 6, ...) ;
2. le caractère vaut '1', ce qui représente à l'entrée d'une donnée dans le bloc. Le bloc est stimulé et donc la propagation ou l'absence de propagation est intégralement liée au traitement effectué et est donc fournie par la sortance :

```
Input: input_list : représentation du flux entrant
Input: pattern_list : acceptance du bloc en cours de traitement
Output: boolean : bloc compatible/non compatible
Data: length : integer
Data: offsetInput : integer
Data: offsetPattern : integer
begin
    length ← size(input_list)
    offsetPattern ← 0
    offsetInput ← 0
    for offsetInput ← 0 to length do
        if input_list[offsetInput] = "1" then
            if pattern_list[offsetPattern] = "1" then
                offsetPattern ⟵ offsetPattern + 1
            else
                return FAIL
            end
        else
            if pattern_list[offsetPattern] = "0" then
                offsetPattern ← offsetPattern + 1
            end
        end
    end
    return SUCCESS
end
```

Algorithme 1: Algorithme de validation de l'acceptance d'un bloc en fonction du flux entrant.

- le caractère dans la sortance au pas courant vaut '0', la donnée n'est pas propagée, un '0' est reporté dans le flux sortant (figure 2.10(a) pas de temps 2, 4, 6, 8) ;
- le caractère vaut '1', une information sort et un '1' est inscrit dans le flux sortant (figure 2.10(a) pas de temps 1, 3, 5, 7).

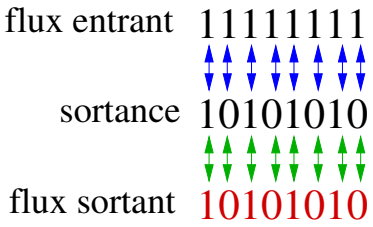
Dans ces deux cas, la position courante dans la sortance est avancée.

Ce traitement est réalisé sur toute la longueur de la chaîne du flux entrant.

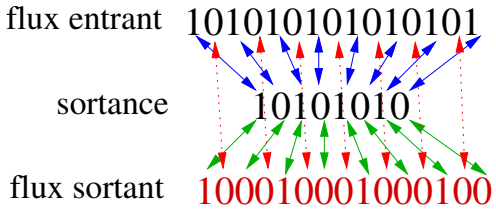
```
Input: input_list : flux entrant dans le bloc
Input: pattern_list : acceptance du bloc
Input: input_size : taille du flux
Output: Array : représentation du flux sortant du bloc
Data: indexinput : integer
Data: indexPattern : integer
Data: i : integer
Data: output_list : array 0 .. length(input_list)
begin
    indexPattern ← 0
    output_list ← ""
    for i ← 0 to input_size do
        if intput_list[i] = "1" then
            output_list ← concat(output_list, pattern_list[indexPattern])
        else
            output_list ← concat(output_list, "0")
        end
    end
    return output_list
end
```

Algorithme 2: Algorithme de génération du flux sortant basé sur le flux entrant et sur la sortance d'un bloc en cours d'analyse.

L'absence de notion temporelle dans la description de la sortance est induite par le fait que cette notion est déduite du flux entrant. Par exemple, en prenant le cas d'un bloc capable de recevoir une donnée tous les deux cycles, réalisant un traitement ayant pour effet de décimer le flux d'un facteur 2, et pour une taille d'analyse arbitraire de N, pour que le flux entrant soit jugé compatibles il doit avoir au moins la même représentation que l'acceptance soit $2 * N$ éléments. La sortance, quant à elle, contient N éléments (soit une alternance de '1' et de '0') car elle ne prend pas en compte l'acceptance du bloc. Ainsi, grâce au traitement ci-dessus, chaque temps-mort en entrée se retrouvera dans le flux en sortie, générant un flux de $2 * N$ caractères, avec un motif répété de '1000' correspondant exactement à la séquence attendue, dans laquelle le premier 1 correspond à la production d'une donnée, le premier '0' au temps mort du pas de temps 2, le second '0' à la non propagation de la donnée suivante et finalement le troisième '0' à l'absence de donnée reçue au pas de temps 4, tel que représenté sur la figure 2.10(b).



(a) Génération du flux de sortie pour un bloc recevant des données à chaque cycle, mais n'en produisant que tous les deux cycles



(b) Génération du flux de sortie pour un bloc recevant des données à tous les deux cycles et qui réalise une décimation par deux.

FIGURE 2.10 – Deux cas de génération du flux de sortie avec dans le premier cas une décimation sur un flux constant et dans le second, la même décimation mais sur un flux présentant des temps morts.

2.5 Entrée utilisateur

Actuellement, CoGen est en mesure d'accepter des informations via deux entrées distinctes :

- la première solution est un *shell* interactif où l'utilisateur saisit les instructions pour spécifier la plate-forme cible et le modèle du FPGA, ajouter un bloc, le connecter ou lancer les étapes d'analyses ou de production du script à destination de POD pour la génération du binaire à charger dans le FPGA. Pour accéder à ce mode, l'utilisateur doit entrer la commande `cogen` sans arguments.
- La seconde solution consiste à utiliser un fichier script qui contient l'ensemble des instructions pour ajouter, assembler et produire la chaîne. Ces instructions sont les mêmes que celles utilisées en mode interactif : `cogen -s fichier_script.cog`

Ce jeu d'instructions est assez réduit et comporte les commandes suivantes :

- create *projectname* : pour la création d'un nouveau projet CoGen ;
- selectplatform *platform* : pour choisir la plate-forme cible (apf27, apf51, sp_vision) ;
- selectfpga *fpga* : pour choisir le type de FPGA, si celui-ci n'est pas le modèle par défaut. Cette commande est optionnelle ;
- selectinitialblock *initialblockname [instancename]* : pour ajouter un bloc initial ;
- selectterminalblock *terminalblockname [instancename]* : pour ajouter un bloc terminal ;
- addblock *blockname [instancename]* : pour ajouter un bloc de traitement du flux. Si un nom n'est pas fourni en argument de cette commande, le nom de l'instance sera automatiquement généré et correspondra au nom du bloc postfixé par un numéro (00, 01, ...) incrémenté si plusieurs instances d'un même bloc sont présentes dans le projet. Qu'un nom soit fourni ou généré, c'est celui-ci qui sera utilisé par la suite pour les étapes de connexions et de configuration ;
- connectblock *instancename_block_source.if_out instancename_block_dest.if_in* : sert à re-

lier deux instances, la première fournissant le flux à la seconde ;

- setgeneric *instancename_ block.generic value* : cette instruction permet de changer la valeur par défaut d'un paramètre de l'instance ;
- generatesolution : cette commande est utilisée une fois la chaîne entièrement spécifiée afin de produire l'ensemble des chaînes compatibles ;
- generatedesign : pour produire le script utilisé par POD pour la génération du design.

Exemple de commandes pour la création d'une chaîne (représentée figure 2.11) recevant un flux d'une caméra photonfocus (section4.2), propageant les données au moyen d'un flux vidéo, et appliquant un filtre median puis un *cropping*, à destination du Spartan6 LX100 équipant une carte SP_VISION :

- create testUser
- selectplatform sp_vision
- selectfpga spartan6lx100
- selectinitialblock photonfocus camera
- selectblock median
- selectblock cropping
- selectterminalblock debug_port csi
- connectblock camera.if_out median00.if_in
- connectblock median00.if_out cropping00.if_in
- connectblock cropping00.if_out csi.if_in
- generatesolution
- generatedesign

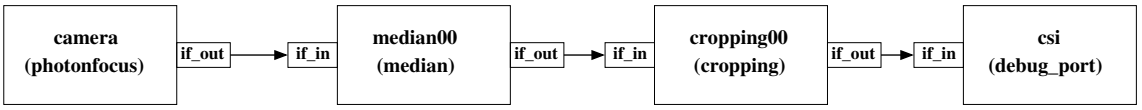


FIGURE 2.11 – Exemple de chaîne de traitements. Elle comporte un bloc initial (camera), deux blocs de traitements (median00 et cropping00) et un bloc terminal (csi).

2.6 Validation

Pour illustrer et valider les opérations réalisées par CoGen, nous définissons deux chaînes qu'il devra analyser. Le but est de vérifier que CoGen prend bien en compte les aspects matériels, le flux entrant, et que le choix de la solution construite est en adéquation avec les attentes.

La première chaîne (figure 2.12) est composée d'un bloc initial, d'un algorithme de détection de contours et d'un bloc terminal pour propager le flux vers le processeur. La seconde chaîne (figure 2.13) inclut également, en amont de ce bloc de détection de contours, une moyenne/décimation sur

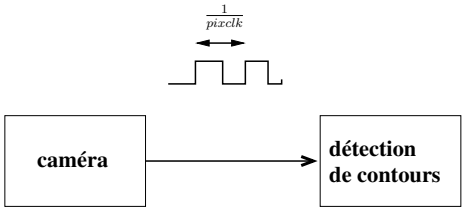


FIGURE 2.12 – Première chaîne proposée pour validation. Elle comporte un bloc initial servant à la réception de données de type vidéo et un bloc qui réalise une détection de contours.

deux pixels consécutifs, dont le but est d'altérer le flux entrant dans le bloc de traitement, et ainsi permettre d'avoir un flux plus lent que dans la première chaîne.

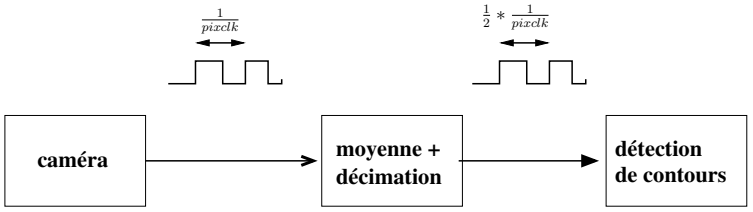


FIGURE 2.13 – Seconde chaîne de traitements. Elle comporte en plus des blocs de la première chaîne un bloc dédié à la décimation. Ainsi en sortie de ce nouveau bloc, la fréquence du flux initial est divisée par 2.

Ces deux chaînes seront analysées dans le cas de deux caméras distinctes, la première fournissant une *pixelclock* de 100 MHz et la seconde de 50 MHz. Toutes deux sont en niveaux de gris, produisant des données codées sur 8 bits et le bloc terminal produit un nouveau flux également en niveaux de gris 8 bits.

Deux FPGAs différents seront également utilisés pour ces tests pour contrôler la bonne prise en compte du matériel :

- un Spartan 6 LX45 avec 58 DSP48 (bloc qui inclus la fonction de multiplication) ;
- un Spartan 3 A200 avec 16 multiplieurs mais pas de DSP48.

Dans les deux cas, la fréquence des FPGAs est de 100 MHz.

Les résultats sont présentés dans la section 2.6.2 après une brêve présentation des algorithmes de convolution.

2.6.1 Implémentations de l'algorithme

Pour les besoins de cet ensemble de tests, les implémentations sont décrites exclusivement par leurs entrées-sorties, leurs latences et par le nombre et le type des multiplieurs requis.

Du fait de sa simplicité, le bloc moyenneur ne dispose que d'une seule implémentation. Sa latence est de un cycle d'horloge, son acceptance est *1(IMG_ WIDTH*IMG_ HEIGHT)* et sa sor-

tance *[01](IMG_WIDTH*IMG_HEIGHT/2)* où *IMG_WIDTH* et *IMG_HEIGHT* représentent la largeur et la hauteur de l'image en pixels.

La détection de contours est implémentée par une convolution qui utilise deux noyaux 3x3 pour calculer les dérivées dans les directions verticale et horizontale. Ce type d'algorithme, du fait de l'utilisation massive de multiplications, est un bon candidat pour disposer de plusieurs implémentations visant à :

- tirer parti au mieux des ressources disponibles ;
- être plus générique.
- être le plus économe en ressources.

Ces implémentations présentent également des approches donnant la priorité

- à la performance aux dépens des ressources disponibles
- ou au contraire à chercher l'économie de ressources mais induisant une baisse des performances.

Bien qu'il soit possible d'en envisager plus, nous n'allons considérer que quatre implémentations :

1. une dont le but est de minimiser au maximum les latences en faisant le meilleur usage des ressources disponibles pour une catégorie de FPGA ;
2. une seconde également réalisée dans le but de réduire les latences mais totalement indépendante du type et du modèle de FPGA vis-à-vis des ressources ;
3. une présentant un compromis entre la réduction des latences et la réduction de l'utilisation de ressources ;
4. finalement une dernière dont le but est de consommer le moins de multiplieurs possible.

L'implémentation 1 (figure 2.14) nécessite 18 blocs multiplieurs de type DSP48, uniquement disponibles sur les Spartan6. Cette implémentation présente les latences les plus faibles. En faisant usage des registres en entrée et en sortie des blocs DSP48 ainsi que du chaînage de ceux-ci, il est possible de recevoir et de produire une donnée à chaque cycle d'horloge à partir du moment où le pipeline est chargé. La latence est de deux cycles et l'acceptance et la sortance sont représentées par `1(IMG_WIDTH*IMG_HEIGHT)` et `1(IMG_WIDTH*IMG_HEIGHT)`. Cette implémentation ne peut-être utilisée avec un Spartan3 du fait de l'absence des DSP48

L'implémentation 2 (figure 2.15) nécessite 18 multiplieurs décrits par l'opérateur générique '*'. Elle peut ainsi être utilisée avec n'importe quel type de FPGA. Cette implémentation est basée sur la technique du pipeline, avec deux étages, le premier réalise l'ensemble des multiplications, le second les additions. Pour garantir le respect des temps de propagations, le résultat du second étage est mémorisé dans un registre synchrone avant la propagation, ce qui ajoute un cycle de plus à la latence, elle devient de 3 cycles. Les motifs restent identiques à la première implémentation : `1(IMG_WIDTH*IMG_HEIGHT)`.

L'implémentation 3 (figure 2.16) réduit l'utilisation de multiplieurs : seulement 9 sont utilisés pour calculer séquentiellement les dérivées dans les directions horizontale et verticale. Les additions

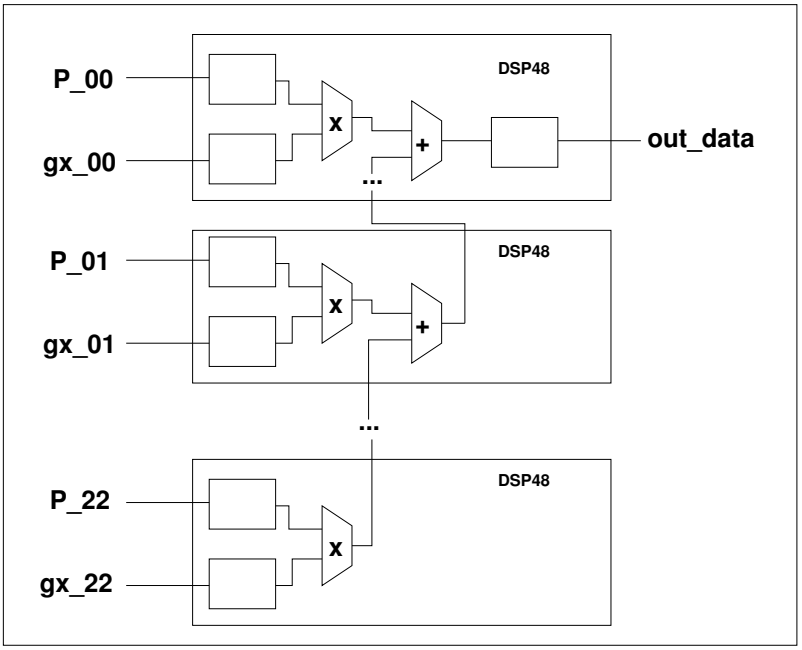


FIGURE 2.14 – Schéma du calcul de la dérivée horizontale avec une utilisation optimale des blocs DSP48.

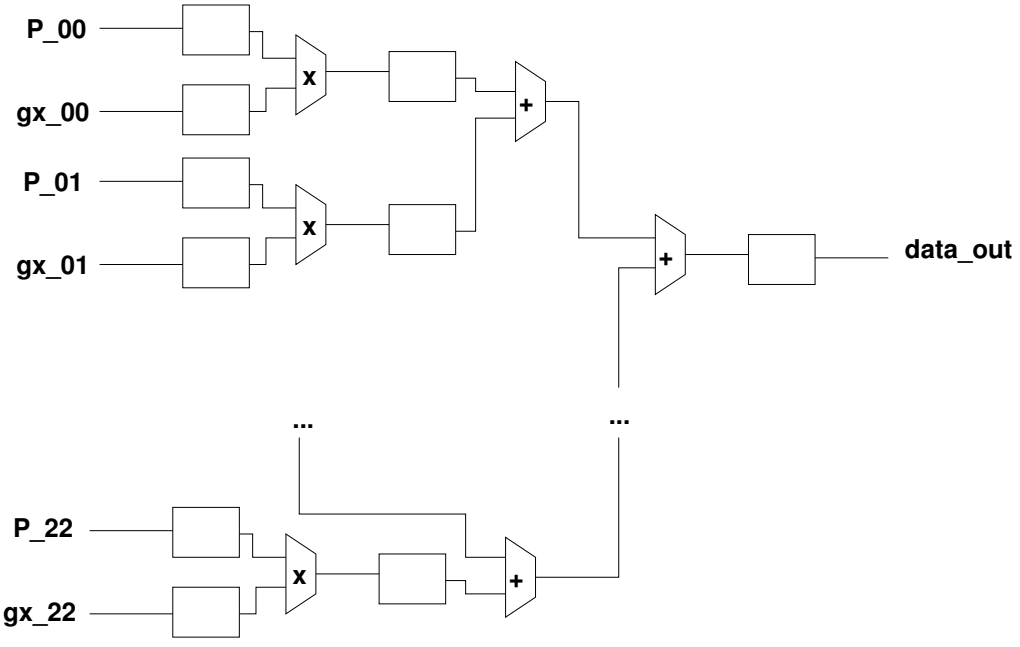


FIGURE 2.15 – Schéma du calcul de la dérivée horizontale avec utilisation au premier niveau de multiplieurs et au second de registres pour les additions.

relatives à la dérivée en x sont réalisées en parallèle des multiplications pour la dérivée en y. Un dernier cycle d'horloge est nécessaire pour réaliser la somme des deux dérivées avant le stockage et la transmission. Du fait du sous-ensemble de multiplieurs, le débit du flux entrant est divisé

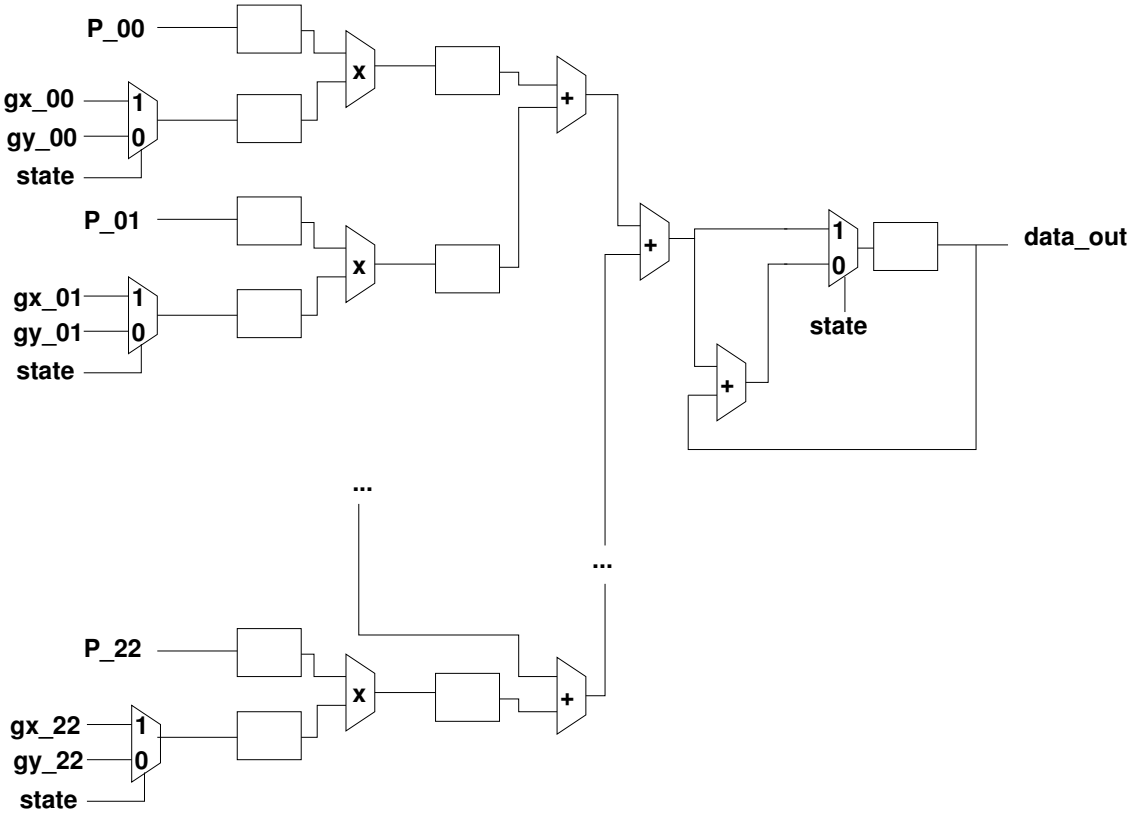


FIGURE 2.16 – Afin de réduire le volume de multiplieurs nécessaire, des multiplexeurs sont utilisés pour charger séquentiellement les coefficients pour la dérivée en x et en y, ainsi que pour réaliser la dernière addition.

par 2 et la latence est incrémentée d'un cycle. La latence finale est de 4 cycles et les motifs sont `[10](IMG_WIDTH*IMG_HEIGHT)` pour l'acceptance et reste `1(IMG_WIDTH*IMG_HEIGHT)` pour la sortance.

La dernière implémentation (figure 2.17) minimise les ressources de calculs par l'utilisation d'un unique multiplieur. En conséquence, cette implémentation nécessite 18 cycles d'horloge pour réaliser l'ensemble des multiplications nécessaires. Un cycle supplémentaire est également requis pour réaliser la dernière addition en sortie du multiplieur et propager le résultat obtenu. Ainsi cette implémentation n'est pas en mesure de recevoir de nouveaux pixels dans un temps inférieur à 18 cycles. De ce fait, la latence est de 19 cycles, l'acceptance est `[10(17)](IMG_WIDTH*IMG_HEIGHT)` et la sortance `1(IMG_WIDTH*IMG_HEIGHT)`.

2.6.2 Résultats

Nous présentons maintenant les résultats obtenus avec les deux chaînes et pour les deux horloges de cadencement des blocs de traitements.

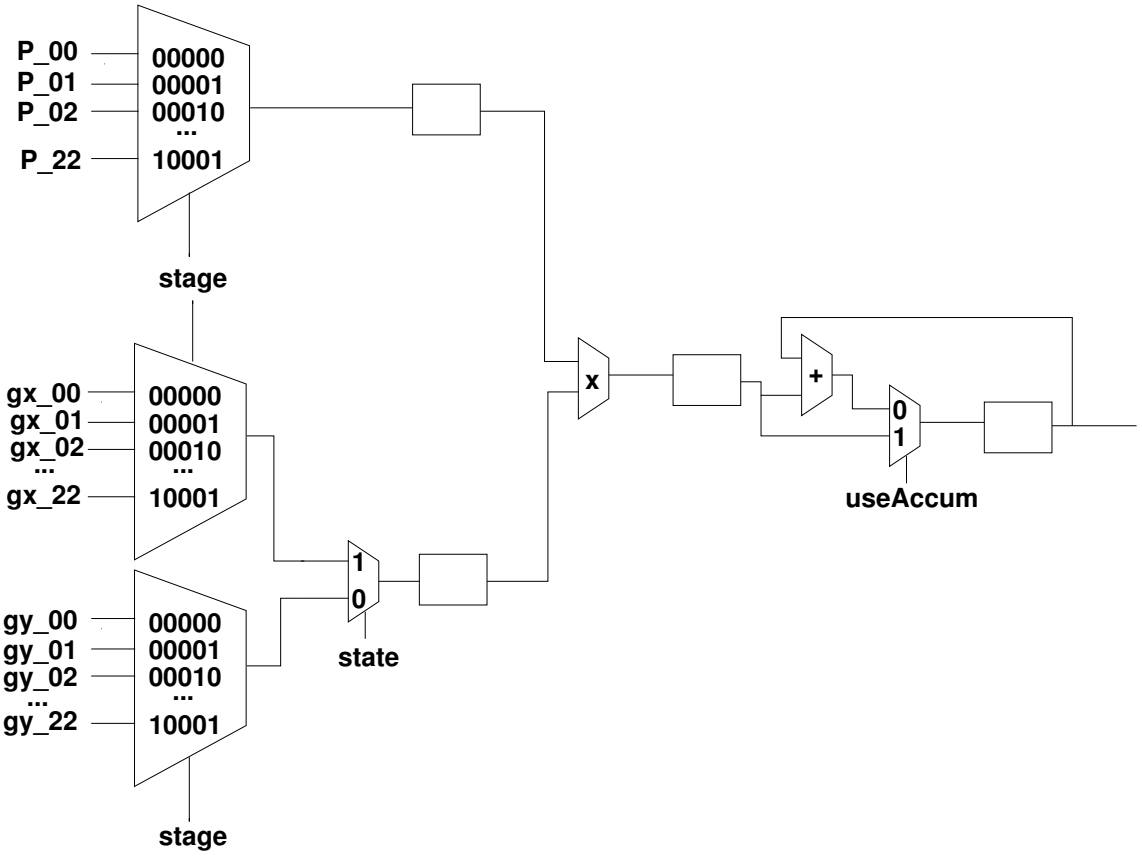


FIGURE 2.17 – Afin d'obtenir une implémentation ne nécessitant qu'un seul multiplieur, chaque entrée de celui-ci est multiplexée pour charger séquentiellement les coefficients et les pixels.

version	acceptance	sortance	latence
1	1(IMG_WIDTH*IMG_HEIGHT)	1(IMG_WIDTH*IMG_HEIGHT)	2
2	1(IMG_WIDTH*IMG_HEIGHT)	1(IMG_WIDTH*IMG_HEIGHT)	3
3	[10](IMG_WIDTH*IMG_HEIGHT)	1(IMG_WIDTH*IMG_HEIGHT)	4
4	[10(17)](IMG_WIDTH*IMG_HEIGHT)	1(IMG_WIDTH*IMG_HEIGHT)	19

TABLE 2.1 – Récapitulatif des acceptances, sortances et latences pour les 4 implémentations proposées.

2.6.2.1 Première chaîne

Le premier test est réalisé avec la caméra dont la *pixelclock* est de 100 MHz. Dans le cas du Spartan6, seules les implémentations 1 et 2 sont conservées. La raison du rejet des autres implémentations est liée à leurs acceptances trop faibles. Les deux implémentations requièrent le même volume de ressources, la première est sélectionnée du fait de ses latences réduites.

Dans le cas du Spartan3, CoGen n'est pas en mesure de trouver une solution acceptable, car

comme dans le cas du Spartan6, seules les 2 premières disposent d'une acceptance suffisamment importante pour absorber le flux de pixels, toutefois la première nécessite un type de ressource non disponible sur un Spartan3 (les DSP48) et la seconde requièrt plus de multiplieurs que disponibles dans ce FPGA. Ainsi, l'utilisateur est informé de l'impossibilité de trouver une solution acceptable.

Pour le second test, la caméra disposant d'une *pixelclock* à 50 MHz est utilisée. Dans le cas du Spartan6, la seule différence est que la solution 3 est acceptable puisque le flux entrant est en adéquation avec son acceptance. Selon le choix de l'utilisateur, en ce qui concerne le critère de choix : performance ou économie de ressources, soit la solution 1 est sélectionnée du fait de ses performances, soit la solution 3 du fait de son aspect économe en ressources.

Pour le Spartan3, comme la solution 3 est adaptée au flux entrant et qu'elle consomme un volume de multiplieurs inférieur aux possibilités de ce FPGA, elle devient la seule solution admissible et est donc sélectionnée.

2.6.2.2 Seconde chaîne

Pour les deux périodes d'horloges, les solutions restent les mêmes que pour la première chaîne dans le cas de l'horloge à 50 MHz. Ceci est dû au bloc de décimation qui produit un flux dont le débit est divisé par 2. Ainsi, le flux devient 50 Méchantillons/s dans le cas de l'horloge à 100 MHz et 25 Méchantillons/s pour la seconde, rendant dans tous les cas l'implémentation 3 compatible.

Ces résultats présentent plusieurs caractéristiques importantes de CoGen. D'une part, CoGen produit les résultats attendus. Ensuite, les caractéristiques du modèle du FPGA cible n'ont pas besoin d'être connus par l'utilisateur. CoGen gère automatiquement ce type d'informations. Finalement, en se basant sur l'acceptance et la sortance de chaque bloc, CoGen est capable de prendre en compte l'évolution du flux au long de la chaîne, tel que présenté dans la série de tests utilisant un bloc de décimation.

Les deux chaînes de traitements présentées sont relativement simples dans le but de faciliter la présentation. Toutefois, des chaînes de traitements plus complexes peuvent également être traitées. Telle que des chaînes avec des branches parallèles ou des cascades de filtres pour lesquels des implémentations plus lentes peuvent être nécessaire aprés le premier niveau de filtrage.

Chapitre 3

Intégration et qualification de composants externes : cas des RAMs

L'ensemble des traitements présentés dans les chapitres précédents repose sur un mécanisme synchrone garanti par l'utilisation du signal `enable` dont chaque bloc doit disposer. En exploitant les informations d'acceptances et de sortances, il devient possible de garantir de manière fiable la validité de la chaîne et des traitements. Toutefois, il est parfois nécessaire de faire usage de composants externes au FPGA, comme les RAMs, dans le cas où le besoin en stockage devient important et ne peut être réalisé à l'aide des blocs de RAMs contenus dans le FPGA.

La première difficulté avec ce type de composant repose sur l'aspect asynchrone de leur fonctionnement : il est donc nécessaire de pouvoir le qualifier de manière précise afin d'intégrer dans CoGen une politique de gestion, voire ajouter une logique particulière pour apporter un aspect prédictible aux délais d'accès, absents par défaut, permettant de tenir compte des latences dans les analyses réalisées et dans les altérations temporelles induites par leur fonctionnement.

Le second problème concerne les latences induites par les RAMs : contrairement à une BRAM dont les accès prennent toujours un cycle d'horloge, sans phase préliminaire d'envoi de configuration, une RAM externe est contrôlée au travers d'un protocole spécifique qui impose de fournir un ensemble de commandes avant le transfert effectif de la ou des donnée(s). Par ailleurs, un composant de ce type est composé de plusieurs **Columns** et **Rows**. Si deux transactions doivent accéder à des **Rows** différentes, un ensemble de commandes supplémentaire est nécessaire pour réaliser l'ouverture de la nouvelle **Row**. Pour finir, une DRAM nécessite des cycles de rafraîchissement du contenu sous peine de perdre l'ensemble des données.

Nous allons présenter, dans la suite de ce chapitre, les travaux réalisés autour des RAMs externes disponibles sur la carte SP_VISION, comportant un Spartan6 Xilinx, commercialisée par la société Armadeus Systems. Ces RAMs sont des **LPDDR** du modèle **MT46H16M16LFBF-6**[44] de chez Micron Technology cadencées à 166 MHz et disposant d'un bus de données 16bits.

Ce travail a pour but de réaliser une première évaluation des caractéristiques temporelles et comportementales de ce type de composant et de proposer une approche afin de simplifier les méthodes d'accès. Il devra être étendu à d'autre type de mémoire afin de valider les conclusions de ce cas d'étude.

Cette présentation est réalisée en trois parties :

- l'étude du contrôleur de RAM, fourni par le fondeur du FPGA, nécessaire à la communication avec le composant ;
- la caractérisation du comportement temporel de la puce disponible sur la carte grâce à une phase de simulation, prenant en compte un certain nombre de cas d'utilisation (exploitation par un seul client, accès concurrent de plusieurs clients en lecture et écriture, stockage de données une par une et non par lot). Cette caractérisation a pour but d'évaluer la meilleure manière d'intégrer à l'outil la prise en compte de l'utilisation d'un composant externe asynchrone en vue de garantir la fiabilité de la chaîne tout en tentant de limiter la baisse des performances globales.
- la réalisation d'un bloc intermédiaire entre une implémentation et le contrôleur permettant de simplifier l'utilisation de la RAM du point de vue du développeur et de disposer d'une interface homogène et paramétrable. Ce bloc est également utilisé pour implanter la politique garantissant les aspects temporels nécessaires à la préservation du comportement prédictible de la chaîne de traitements ;

3.1 Présentation du contrôleur de RAM de Xilinx

Xilinx fournit un contrôleur de RAM qui permet de s'affranchir du développement et de la connaissance physique d'un tel composant. Ce bloc offre également :

- la possibilité de fournir un accès à plusieurs utilisateurs, de manière concurrente, au travers de ports ; l'accès repose sur un mécanisme d'arbitrage ;
- la gestion automatique des phases de rafraîchissement de la RAM ;
- une abstraction de la nature de la RAM, ce qui permet de réaliser des développements en s'affranchissant des comportements de la mémoire.

Avant d'être en mesure de simuler la RAM pour la qualifier, il est nécessaire de comprendre le mécanisme de fonctionnement du contrôleur.

3.1.1 Interface de communication

Un port est composé de deux ou trois interfaces selon qu'il est en mode bidirectionnel (lecture/écriture) ou dans un mode unidirectionnel (lecture ou écriture).

3.1.1.1 Interfaces de lecture et d'écriture

Dans tous les cas, l'écriture et la lecture ne sont pas réalisées physiquement pour chaque mot mais au travers d'une FIFO (First In First Out) comportant au plus 64 mots. Il est bien entendu possible de réaliser des transferts d'un seul mot, mais toujours par l'intermédiaire de la FIFO.

Le contenu des FIFOs est physiquement écrit dans la RAM dans un second temps, ceci ayant pour but de limiter les ralentissements induits par l'accès physique au composant.

Les interfaces de lecture et d'écriture sont composées de 8 signaux :

- un signal d'horloge ;
- un signal d'activation pour l'empilement (ou le dépilement en mode lecture) ;
- un bus de données ;
- deux signaux pour informer que la FIFO est pleine ou vide ;
- le nombre de mots contenus dans la FIFO ;
- un signal d'erreur signalant le débordement (pour l'interface d'écriture) ou une pénurie de données (pour l'interface de lecture) si le `enable` est activé alors que la FIFO est déjà pleine (respectivement vide).

3.1.1.2 Interface de commande pour l'accès physique à la RAM

Le transfert effectif des données ne se fait pas automatiquement, c'est au code utilisateur de gérer le moment où les données doivent être physiquement transférées ou à quel moment la FIFO doit être remplie par un volume particulier de données contenues dans la RAM.

Cette interface dispose également d'une FIFO plus petite qui fait le tampon dans le cas où une transaction est en cours entre le contrôleur et la RAM (accès concurrent). Cette interface est commune pour la lecture et l'écriture, et contient un ensemble de 7 signaux :

- un signal d'horloge
- un signal d'activation qui à la fois donne l'ordre d'empiler la commande et déclenche le traitement dans le contrôleur ;
- un bus qui fournit le type de la commande (principalement ordre d'écriture ou de lecture) ;
- un bus donnant la taille du transfert en lecture en écriture dans/depuis la RAM ;
- l'adresse dans la RAM où écrire ou lire les données ;
- et finalement deux signaux donnant l'état (vide ou plein) de la FIFO de commande.

3.1.1.3 Arbitrage

Le contrôleur dispose également d'un mécanisme d'arbitrage qui repose sur un ensemble de 12 **slots**. Chacun d'eux contient le numéro de client de plus forte priorité. Le mode de fonctionnement par défaut permet de faire évoluer la priorité des ports à chaque cycle d'horloge. Nous avons remplacé ce type d'évolution dynamique des priorités par une priorité statique pour chaque port. En effet, sachant que la priorité fluctue dans le temps, à partir d'un temps initial qui correspond

à la réinitialisation du contrôleur, il est impossible de pouvoir obtenir une simulation garantissant le comportement réel car l'arrivée de la première donnée n'est pas corrélée à cette remise à 0.

3.1.2 Utilisation

3.1.2.1 Écriture

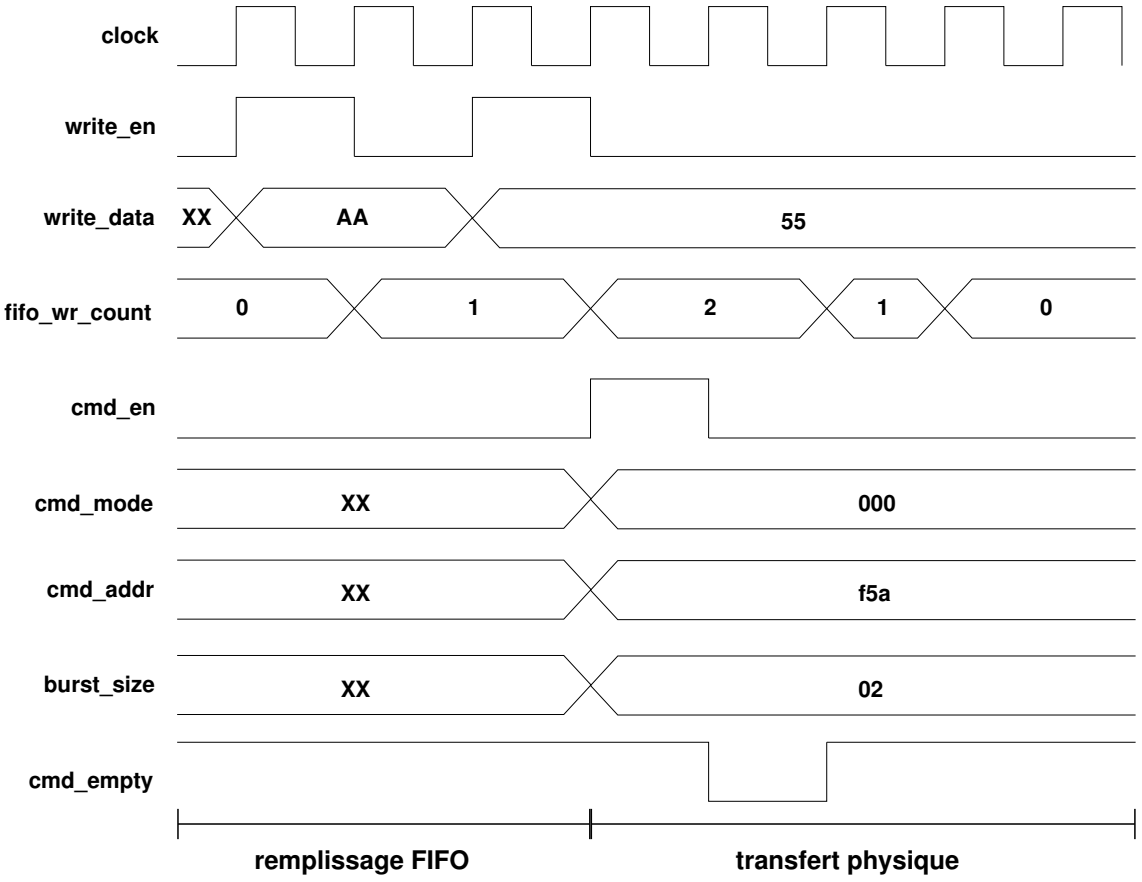


FIGURE 3.1 – Chronogramme de principe du mécanisme d'écriture dans la RAM pour un volume de transfert de 2 mots. Partie gauche : transfert des données vers la FIFO ; partie droite : envoi de la commande d'écriture physique dans la RAM.

La phase d'écriture de données est sans doute le mode le plus simple et consiste en cette série d'actions (représentées dans le chronogramme 3.1) :

- stockage dans la FIFO (partie gauche du chronogramme 3.1) : pour chaque nouvelle donnée, l'information est présentée sur le bus de données (`write_data`) et le signal de contrôle `write_en` est passé, pendant un cycle d'horloge, à l'état haut. Un cycle d'horloge après l'ordre d'écriture, le nombre d'éléments contenus dans la FIFO est mis à jour (`fifo_wr_count`) ;
- transfert des données dans le composant de RAM (partie droite de 3.1) : le nombre de données à transférer (valeur entre 1 et 64) et l'adresse sont présentés sur les bus correspondants

(`burst_size` et `cmd_addr`), le type de commande est fixé au mode écriture (valeur 000 dans `cmd_mode`) et le signal `cmd_en` est activé. Aucune information en retour n'est fournie concernant le début réel et la fin de la transaction. La seule manière d'évaluer si le traitement a eu lieu est de se baser sur l'état du signal `cmd_empty` qui passe à 0 lorsqu'une requête est envoyée et repasse à l'état haut quand elle a été prise en compte, et sur la valeur de `fifo_wr_count` qui est décrémenté dès que le transfert est initié.

Le nombre de données à envoyer dans la RAM peut être inférieur à la taille de la FIFO d'écriture, il est ainsi possible de continuer le stockage pendant le transfert des données, garantissant d'éviter la perte de données.

3.1.2.2 Lecture

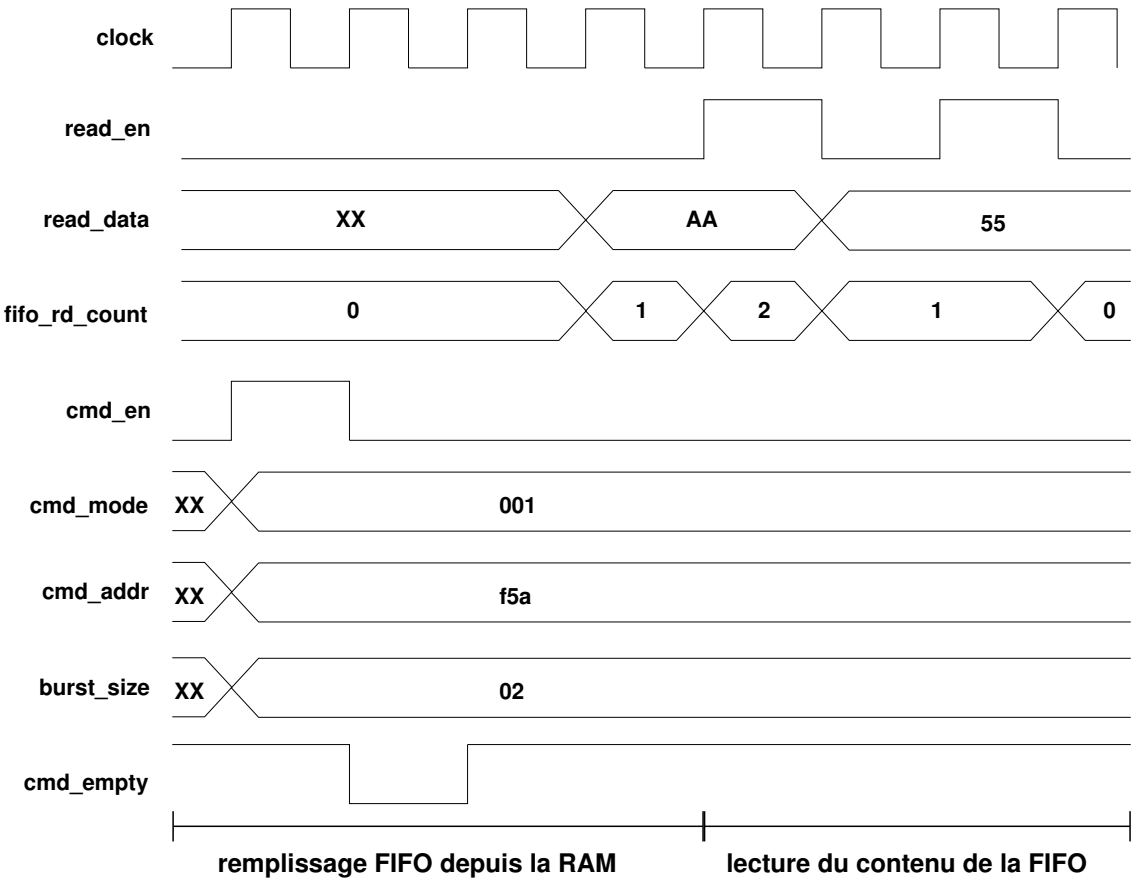


FIGURE 3.2 – Chronogramme de principe du mécanisme de lecture depuis la RAM pour un volume de transfert de 2 mots. Partie gauche : transfert physique depuis la RAM dans la FIFO de lecture ; partie droite : récupération des données contenues dans la FIFO.

Le principe est globalement le même que pour l'écriture et comporte également deux étapes (présentées dans le chronogramme 3.2) :

– transfert des données de la RAM dans la FIFO (partie gauche de la figure 3.2) : à l'instar du

transfert pour l'écriture, l'adresse de base et le nombre d'éléments sont fournis. La différence réside principalement dans la valeur 001 pour `cmd_mode` ;

– lecture des données depuis la FIFO (partie droite de 3.2) : dès lors que la FIFO n'est plus vide, il est possible de disposer directement des données. Par défaut, la valeur en haut de la *pile* est disponible au travers de `read_data`. Le fait de passer le signal `read_en` à l'état haut a pour effet de décrémenter le compteur `fifo_rd_count` et de mettre la prochaine donnée sur le bus.

Comme pour l'écriture, c'est le signal `cmd_empty` qui permet de vérifier que la commande a été prise en compte, et il faut attendre tant que `fifo_rd_count` est égal à '0' ou que le signal `fifo_rd_empty` (non représenté) est à '1'.

3.1.3 Bilan de la communication avec le contrôleur de RAM

Plusieurs points sont à remarquer :

– la relative complexité d'accès ;

– les données peuvent être rassemblées en un `burst` pour éviter les lenteurs d'accès au composant physique, condition sine qua non pour obtenir les meilleurs performances ;

– des `burst` d'une taille inférieure à la capacité de la FIFO peuvent être réalisés, permettant d'éviter les débordements d'écriture et de pouvoir compléter la FIFO avant qu'elle ne soit vide en mode lecture. Grâce à cette solution, il peut théoriquement être possible de rendre l'accès transparent, principalement en mode écriture. En mode lecture, une latence initiale est toujours nécessaire lors du remplissage initial.

3.2 Estimation du comportement temporel de la RAM et du contrôleur

Connaissant le protocole de communication avec le contrôleur, un ensemble de simulations a été réalisé pour caractériser le comportement temporel du composant de RAM et du contrôleur. Ceci afin de pouvoir dresser un tableau des durées des étapes du chronogramme 3.3

Deux ensembles de tests ont été réalisés :

– une série relative aux temps d'écriture avec un ou plusieurs ports utilisés ;

– une seconde série concernant la lecture.

Dans les deux cas, les tests ont étés réalisés sur une même zone de la mémoire (nommée **Row**) puis sur des zones différentes. Ces deux cas de figures sont importants car le changement de **Row** implique une étape d'ouverture qui a pour conséquence une augmentation du nombre de cycles d'horloges nécessaires pour compléter la transmission.

Les tests ont étés réalisés avec les ports cadencés à 200 MHz, le contrôleur étant cadencé à 266 MHz (2× RAM_clock).

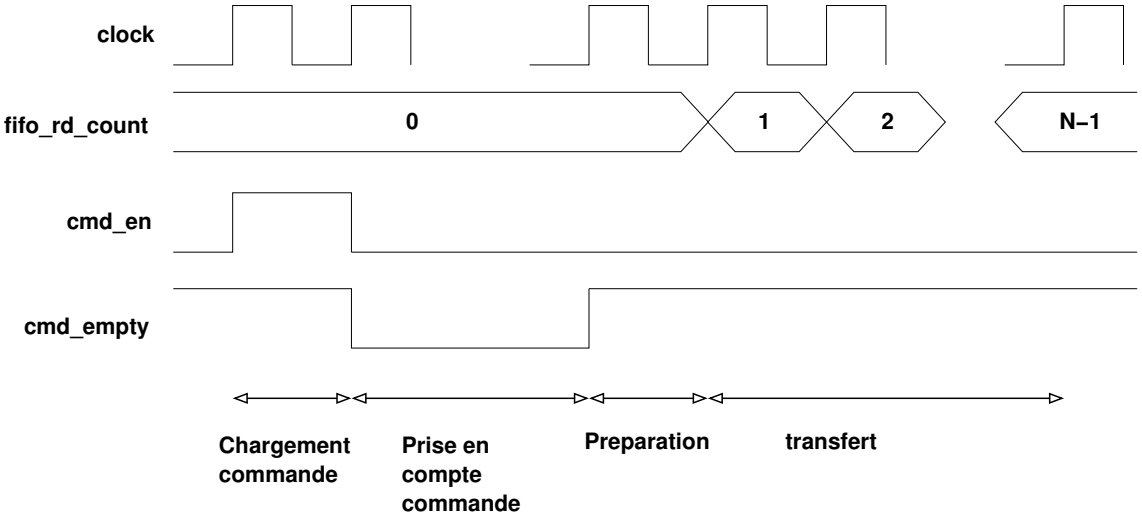


FIGURE 3.3 – Chronogramme de l'évolution des signaux selon les étapes réalisées lors de la réception d'une commande de lecture.

3.2.1 Caractéristiques temporelles en écriture

3.2.1.1 Simulation avec un seul utilisateur

Les premières simulations ont eu pour but d'évaluer les caractéristiques temporelles des transactions pour un seul port avec des bursts de 2, 32 et 64 mots en réalisant l'écriture à chaque fois au même endroit puis en changeant de **Row** entre deux transactions. Les résultats sont reportés dans le tableau 3.1. Dans ce tableau ainsi que dans les suivants, l'augmentation du temps induit par une phase de rafraîchissement n'est pas mentionnée. Elle correspond à un allongement de 20 ns du temps de préparation.

changement de **Row**	chargement commande	prise en compte	préparation	transfert		
				2 mots	32 mots	64 mots
non	5ns	30ns	50ns	5ns	190ns	375ns
oui	5ns	30ns	70ns	5ns	190ns	375ns

TABLE 3.1 – Caractéristiques temporelles de l'ensemble contrôleur et RAM pour des transferts de 2, 32 et 64 mots, sans ou avec changement de zone.

3.2.1.2 Simulation avec deux utilisateurs

La série suivante de simulations repose sur le même principe mais avec deux ports. Les commandes d'écritures sont envoyées en même temps pour les deux utilisateurs, générant ainsi un conflit. Les résultats sont reportés dans le tableau 3.2. Les caractéristiques temporelles pour le

port ayant la plus grande priorité (port 1) ne sont pas reportées car elles sont identiques au cas précédent. Pour la même raison, le temps de transfert des données n'est pas indiqué.

changement de **Row**	chargement commande	durée de prise en compte	préparation		
			2 mots	32 mots	64 mots
non	5ns	55ns	55ns	240ns	425ns
oui	5ns	55ns	125ns	310ns	495ns

TABLE 3.2 – Caractéristiques temporelles pour le second port avec un accès de type conflit, pour des transferts de 2, 32 et 64 mots, sans ou avec changement de zone.

La durée de préparation pour le second port peut-être décomposé ainsi :

- durée de la préparation du port1 à laquelle on soustrait la différence des durées de prise en compte pour chacun des deux ports ;
- durée de transfert du port 1 ;
- latence de 25 ns entre la fin du transfert courant et le début du transfert du port suivant. Cette latence passe à 75ns sans doute dû au changement de **Row**. Cette durée est plus longue que dans le cas d'un seul port sans que l'on puisse l'expliquer précisément, le code du contrôleur n'étant pas disponible.

Un second point à remarquer : la durée de prise en compte de la commande de transfert par le contrôleur est identique pour le port 2 quelque soit la **Row** visée, ce qui est logique car le traitement lié au port2 est retardé tant que le traitement lié au port 1 n'est pas fini.

3.2.1.3 Cas de plus de deux utilisateurs

Dans le cas où il est fait usage de 3 ports ou plus, le comportement est similaire au précédent, et repose sur la durée du port ayant la priorité juste supérieure au port considéré.

3.2.2 Caractéristiques temporelles en lecture

3.2.2.1 Simulation avec un seul utilisateur

Le principe de cette série de simulations est globalement identique à celui présenté précédemment. La latence induite par le rafraichîssement est identique à celle en lecture.

changement de **Row**	chargement commande	durée de la prise en compte	préparation	transfert		
				2 mots	32 mots	64 mots
non	5ns	25ns	100ns	5ns	190ns	375ns
oui	5ns	25ns	140ns	5ns	190ns	375ns

TABLE 3.3 – Caractéristiques temporelles pour des transferts de 2, 32 et 64 mots, sans ou avec changement de zone, en mode lecture.

Cette série de tests (tableau 3.3) indique que le temps de transfert est indépendant du fait que la requête soit une écriture ou une lecture.

3.2.2.2 Cas de plus d'un utilisateur

Dans le cas d'un conflit, les caractéristiques temporelles reposent sur les mêmes principes que dans le cas de l'écriture.

3.2.3 Bilan de la qualification du comportement temporel de la RAM

Plusieurs points importants sont à retenir :

- le changement de **Row** a un impact important sur les temps d'accès, il faut donc tenter de concentrer au maximum les blocs dans une même **Row** afin de minimiser ce changement ;
- les accès en conflit sont complexes à gérer car il faut tenir compte de l'utilisation ou non de **Row** différentes, et de la taille de chaque *burst*. Éviter les conflits ajoute également un paramètre supplémentaire car il est nécessaire de savoir quand la requête de transfert a été émise pour connaître le moment où le contrôleur passera au traitement suivant ;
- les cycles de rafraîchissement ont un impact sur les temps d'accès. Toutefois il n'y a pas de règle ou de loi pour dire a priori quand ils se produiront par rapport à la date d'arrivée de la première donnée ;
- le temps de transfert entre le contrôleur et la RAM, une fois la commande lancée, n'est pas dépendant du nombre de ports utilisés ;

L'ensemble de ces caractéristiques doit être pris en compte par CoGen de telle sorte qu'il soit possible à tout moment de connaître l'état de la RAM et le temps nécessaire avant qu'elle soit à nouveau disponible. C'est également grâce à ce mécanisme qu'il est possible d'ordonnancer efficacement les traitements, évitant la perte de données ou le blocage d'accès par un bloc dû à un conflit.

3.3 Ajout d'une couche d'abstraction entre le contrôleur et les blocs

Le choix d'ajouter une couche d'abstraction, nommée `client_ram`, entre le contrôleur (figure 3.4) et un bloc de traitements à trois buts :

- d'une part, simplifier le travail du développeur en cachant la complexité d'utilisation de ce type de composant. Le choix a été fait de tenter de s'approcher au mieux de la structure d'un bloc de RAM matériel disponible dans les FPGA. Le développeur n'a donc plus à gérer toute la logique nécessaire pour la communication avec les FIFOs et les commandes pour les transactions avec le composant de RAM ;
- d'autre part, le contrôleur présenté dans la section 3.1 est spécifique à `Xilinx`, il n'est pas garanti qu'un autre fondeur propose un contrôleur avec les mêmes caractéristiques. Une implémentation d'un traitement quelconque ne faisant usage d'aucun bloc matériel particulier tels que les PLLs, BRAMs et DSP48, peut être considérée comme générique vis-à-vis d'une famille de FPGA et d'un constructeur. Cette même implémentation peut donc être utilisée sur n'importe quel matériel. Toutefois, si elle fait usage d'une RAM externe, le fait qu'il soit nécessaire de fournir une interface de communication compatible avec le contrôleur de RAM fourni par Xilinx fait que, par effet de bord, cette implémentation devient spécifique à un constructeur, matériel et famille de FPGA. Il devra donc exister plusieurs versions afin de s'adapter à la cible, non pas pour proposer des solutions plus ou moins économes, mais uniquement pour ré-implémenter la politique de communication qui pourrait imposer de repenser partiellement ou totalement la logique d'implémentation de la solution. Ceci est d'une part contre-productif et finalement non souhaitable. Le fait d'ajouter une couche d'abstraction permettra de supprimer cette dépendance et ainsi de conserver au bloc son aspect générique.
- finalement, laisser au développeur le soin de gérer les interactions avec le contrôleur ne permettrait pas de mettre en œuvre une politique de gestion efficace apportant un aspect déterministe dans les transactions avec le contrôleur et par extension avec la RAM. Dans ce type d'approche, CoGen ne serait pas en mesure de savoir précisément à quel moment la FIFO est vidée ou pleine, ni finalement de connaître exactement à quel instant l'accès physique est réalisé. Ce cas de figure empêcherait, par extension, de garantir la validité du traitement réalisé par la chaîne.

Il est également à noter qu'il pourrait être envisageable de considérer que grâce à l'interface proche de celle d'une RAM interne, CoGen pourrait, afin de simplifier la chaîne, si les besoins en place ne dépassent pas les capacités d'une BRAM, privilégier ce type en lieu et place des RAMs externes.

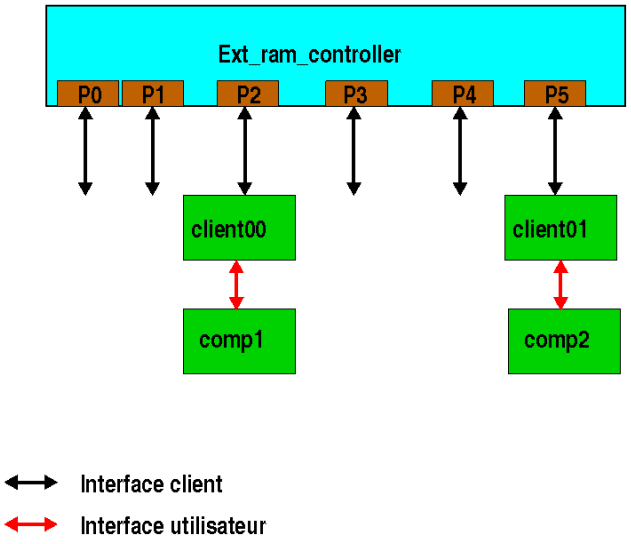


FIGURE 3.4 – Schéma de connexion de composants au contrôleur de RAM au travers du composant client. Deux clients sont connectés, le premier au port 2, le second au port 5. Deux blocs sont connectés aux clients.

3.3.1 Interface de communication et paramètres de configuration

L'interface entre un bloc utilisateur et un client de RAM (figure 3.5) contient les bus et signaux suivants :

- deux bus d'adresses, un pour la lecture et un pour l'écriture ;
- deux bus de données, un depuis l'implémentation vers le client pour l'écriture et le second dans la direction opposée ;
- deux signaux d'activation, pour réaliser une requête de lecture ou d'écriture ;
- et finalement deux signaux d'acquittement.

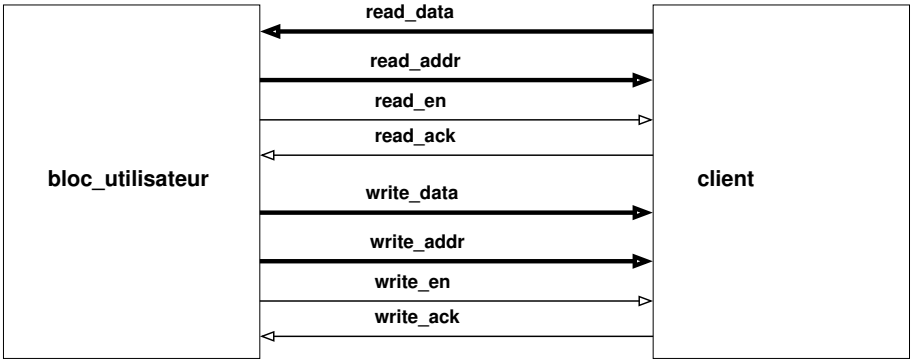


FIGURE 3.5 – Schéma de l'interface de communication entre un bloc utilisateur et un client de RAM.

Comparées à l'interface d'une RAM interne, les seules différences se situent au niveau du signal d'activation de lecture utilisé pour dépiler la FIFO et des signaux d'acquittement utilisés afin de synchroniser les traitements. Ces derniers sont nécessaires en cas d'attente, comme dans le cas de la lecture qui impose que la FIFO contienne au moins un élément avant de pouvoir le fournir au bloc utilisateur.

Concernant la configuration, le client dispose de plusieurs paramètres permettant de fixer certains points de son comportement :

- un paramètre permettant de spécifier le seuil, exprimé en nombre de données écrites dans la FIFO. Une fois ce seuil atteint, le client déclenche une écriture physique dans la RAM ;
- un paramètre spécifiant si le client doit réaliser des requêtes de lecture avant que la FIFO ne soit vide, ceci afin de permettre un principe de flux tendu (pas d'attente de l'utilisateur induite par la nécessité de devoir remplir à nouveau la FIFO) ;
- un paramètre qui spécifie le seuil bas du contenu de la FIFO à partir duquel le client devra réaliser une nouvelle requête de lecture d'une taille égale à *FIFO_SIZE - THRESHOLD_READ*, où *FIFO_SIZE* correspond à taille de la FIFO en nombre d'éléments et *THRESHOLD_READ* au seuil fixé par le paramètre. Ce paramètre n'est pris en compte que si le précédent est fixé à *TRUE*. Dans le cas contraire, une requête de *FIFO_SIZE* éléments est réalisée lorsque que la FIFO est vide.

3.3.1.1 Comportement du client pour l'écriture

Le principe du fonctionnement du client pour l'écriture (représenté sur la figure 3.6) est le suivant : lors de la réception d'une requête d'écriture (état haut du signal d'activation), le client vérifie si son compteur interne est à 0 ou si la FIFO est non vide :

- dans le cas où la FIFO est vide (compteur interne valant 0) il mémorise l'adresse, car c'est celle-ci qui sera utilisée comme adresse de base lors du transfert réel dans la mémoire ;
- si le nombre d'écritures a atteint le seuil de vidage (flush), le client génère un ordre d'écriture physique au niveau du contrôleur et réinitialise le compteur.

Dans tous les cas, incluant le cas où $0 < COUNTER < THRESHOLD_HIGH$, le client mémorise la donnée, incrémente son compteur, et hormis dans le cas où la FIFO est pleine, lève pendant un cycle d'horloge son signal d'acquittement. L'adresse fournie par l'utilisateur n'est prise en compte que si le compteur vaut 0 (état initial ou requête suivant une transaction d'écriture).

Du point de vue de l'utilisateur, une écriture consiste à mémoriser la donnée et l'adresse sur les bus correspondants et à lever pendant un cycle d'horloge son signal de validation. La prise en compte de l'acquittement par l'utilisateur est optionnelle selon l'implémentation et les besoins du traitement.

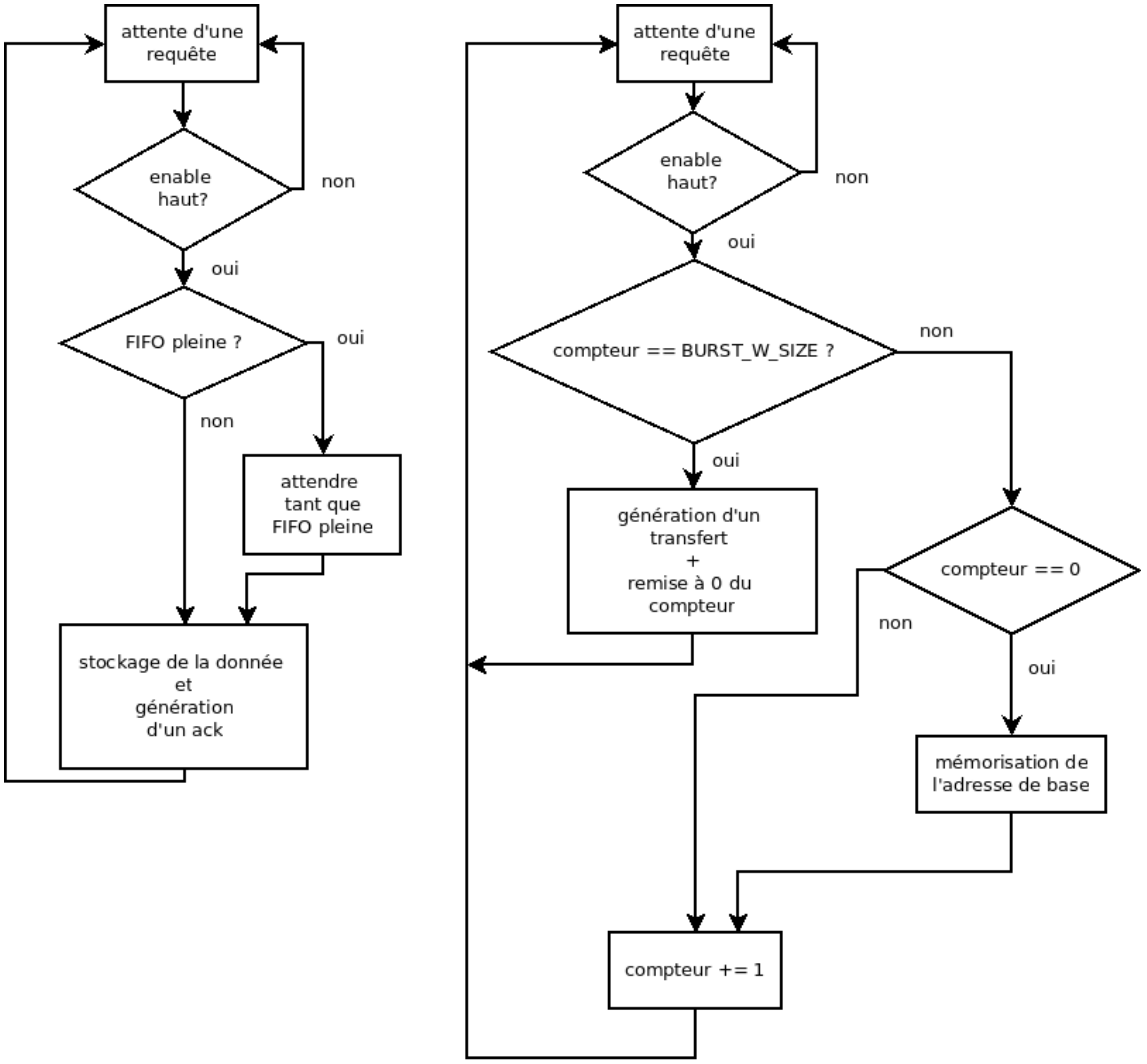


FIGURE 3.6 – Diagramme de flux de la logique d'écriture. À gauche, processus en charge des interactions avec l'utilisateur. À droite, processus en charge de la gestion du transfert vers la RAM, du stockage de l'adresse de base et de la mise à jour du compteur d'écritures réalisées par l'utilisateur.

3.3.1.2 Comportement du client pour la lecture

Le principe du comportement du client de RAM lors d'une lecture (figure 3.7) est le suivant : quand le client détecte l'état haut du signal d'activation, il vérifie le nombre de données disponibles ;

- la FIFO est vide, il fait une requête au niveau du contrôleur pour obtenir une série de données correspondant au nombre maximum de données que peut contenir la FIFO. Il attend ensuite que le signal `empty` passe à l'état bas afin de pouvoir dépiler une donnée, la stocker sur le bus et acquitter la commande ;
- la FIFO n'est pas vide : si le client est configuré pour faire un pré-remplissage et que le nombre d'éléments disponibles a atteint le seuil bas, il fait une requête de lecture pour compléter le

contenu de la FIFO en demandant un volume de données correspondant à la différence entre la capacité de la FIFO et le nombre d'éléments encore disponibles. En parallèle, il dépile un élément, le met sur le bus et avertit l'utilisateur ;

– dans les autres cas, il se contente de dépiler une donnée, de la mémoriser sur le bus et d'avertir l'utilisateur

Le fonctionnement du bloc utilisateur, dans le cas de la lecture, est globalement le même que pour l'écriture. À un instant donné il place l'adresse sur le bus correspondant, lève le signal de `enable` et mémorise à la réception de l'acquittement la donnée portée par le bus de données.

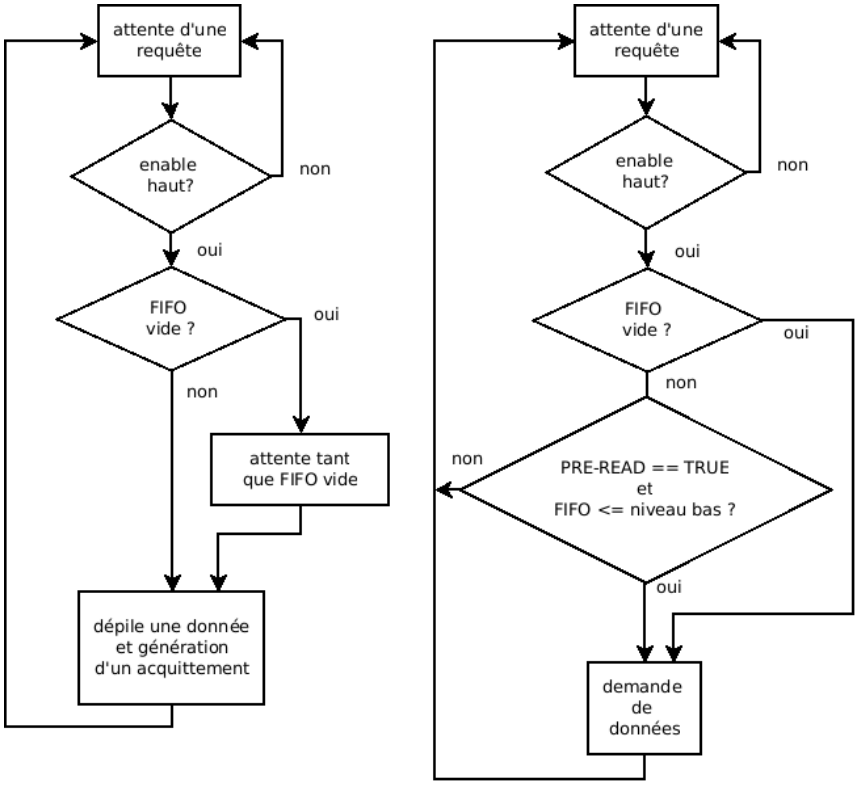


FIGURE 3.7 – Diagramme de flux de la logique de lecture. À gauche, processus en charge des interactions avec l'utilisateur. À droite, processus en charge du remplissage de la FIFO lorsqu'elle est vide ou qu'elle a atteint le seuil bas.

3.4 Bilan de l'utilisation d'une RAM externe et intégration dans CoGen

À l'heure actuelle, le support des RAMs externes n'a pas été intégré dans CoGen. Ces composants n'ont été mis en œuvre que dans quelques cas particuliers où il est possible de garantir les latences, et jamais en accès concurrent.

Deux cas principaux ont été mis en œuvre :

- stockage d'une image dans sa globalité pour produire un nouveau flux en adéquation avec les capacités du bus vidéo du processeur, après décimation du nombre d'images par seconde. Le mécanisme utilise les deux RAMs disponibles qui sont à tour de rôle utilisées en écriture ou en lecture, afin d'éviter les conflits ;
- stockage d'un gros volume d'échantillons issus d'un convertisseur analogique-numérique. Dans ce cas le processeur demande le stockage d'un ensemble de données, et réalise la lecture une fois la mémoire remplie. En réalisant le transfert vers la mémoire sur un sous-ensemble de la taille de la FIFO, il est possible de garantir l'absence de perte de données.

Avant d'être en mesure d'intégrer ce support, plusieurs travaux restent à faire :

- réaliser le même type de simulations sur d'autres modèles de RAM pour vérifier les caractéristiques mises en avant dans la section 3.2 ;
- intégrer dans les fichiers descriptifs XMLs les délais entre la réception d'une donnée et l'accès à la RAM, ainsi que des informations au travers de variables pour l'altération des motifs d'acceptances et de latences en accord avec les limitations induites par la RAM ;
- mettre en place un mécanisme permettant de connaître à chaque pas de temps, lors de l'analyse, l'état du contrôleur et de la RAM ;
- ajouter un mécanisme permettant de fixer la taille des *bursts* en accord avec la fréquence d'utilisation et la charge globale de la RAM.

Les problèmes liés aux retards dus aux phases de rafraîchissement du composant peuvent être évités par une prise en compte du pire cas. Il est toutefois nécessaire pour cela de modifier le client de RAM de telle sorte qu'il n'acquitte la requête en cas de lecture ou d'écriture physique qu'après le temps normal plus le délai induit par cette phase, qui ne sera appliqué que si un rafraîchissement n'a pas eu lieu. Pour ce faire, il est nécessaire d'avoir un repère temporel tel que la durée entre le front montant du signal **fifo_empty** et le début du remplissage ou du transfert du contenu. Ceci est relativement complexe pour plusieurs raisons :

- si le port est seul à réaliser des accès, la durée est directement calculable. Si plus d'un port font une requête au même cycle, la durée va dépendre de la priorité du port, du nombre de ports et de la taille des *bursts* des ports de plus forte priorité. Si finalement la requête est émise alors que le contrôleur est déjà occupé, la durée va dépendre de la date de démarrage de la transaction en cours et des potentiels autres ports qui pourraient être en attente ;
- le second point concerne l'accès à des **Rows** différentes, car il est nécessaire de prendre également en compte cette durée qui dépend des mêmes cas de figures que ceux présentés dans le premier point.

Cette problématique est théoriquement limitée au cas de la lecture lors du remplissage initial de la FIFO, car grâce à la possibilité de réaliser un pré-chargement ou un transfert du contenu, les délais ne sont plus un facteur limitant pour l'acceptance des blocs utilisateurs.

Le problème d'ouverture de **Row** peut être partiellement limité en tentant de concentrer l'ensemble des blocs faisant usage de la mémoire dans une zone ne nécessitant pas de changement. Dans le cas où ce n'est pas possible, l'organisation des blocs de telle sorte de limiter le nombre de changement serait préférable, ceci afin d'éviter des cas où chaque requête nécessiterait une ouverture.

Concernant l'estimation de l'état de la RAM, qui est sans doute l'un des points les plus critiques, mais obligatoire pour évaluer le bon fonctionnement de l'ensemble de la chaîne à chaque pas de temps, il est nécessaire de prendre en compte l'ensemble des blocs faisant usage de la RAM.

Plusieurs traitements devront avoir lieu :

- une chaîne de caractères, du même type que dans le cas des acceptances et sortances tenant compte des latences des blocs, doit être générée. Ceci afin de pouvoir estimer si des conflits apparaîssent ;
- pour chaque port il faut pouvoir connaître l'état des FIFOs, grâce à cela il sera possible de déterminer quand auront lieu les accès physiques ;
- se basant sur ces deux informations, il sera possible d'évaluer la taille des *bursts* afin de garantir l'absence de perte de données et l'accès équitable de tous les blocs connectés au contrôleur.

Toutefois ce principe nécessite d'être raffiné car le risque le plus important concerne les rétroactions : si un bloc quelque part dans la chaîne réalise une requête et qu'un second en amont, alors que la RAM n'est pas disponible, demande à son tour un transfert, il est possible que le retard induit par l'indisponibilité de la mémoire puisse modifier tout l'ordonnancement de la chaîne. Une solution basique semble être de reboucler sur l'ensemble des blocs à chaque nouveau pas de temps. Cette solution implique également une nouvelle évaluation a minima de toutes les sortances, voir des acceptances. Là encore, l'utilisation de sous ensembles des FIFOs peut simplifier le problème mais nécessite la mise en place d'un modèle pour en avoir la garantie.

L'estimation de la taille des *burst* est finalement un point important car s'ils sont petits la durée du transfert deviendra négligeable par rapport aux préparations de la transmission, risquant une perte de données et une impossibilité pour tout autre bloc d'y avoir accès. À l'opposé, dans le cas d'une taille grande, le temps global de la transmission auquel s'ajoute les potentiels cycles de rafraîchissement et d'ouverture risquent de provoquer des débordements de la FIFO de stockage pour une écriture et une carence de donnée en lecture.

3.5 Conclusion

Pour l'heure nous avons retardé l'intégration des RAMs externes dans CoGen et nous nous sommes principalement focalisés sur des traitements qui ne nécessitent pas de stockage de données ou dont les besoins sont en adéquation avec les BRAMs disponibles, tels que ceux présentés dans

les deux chapitres suivants.

Cette limitation induit l'exclusion, temporaire, de certaines classes d'algorithmes de traitements d'images tels que les rotations (section 1.6) qui imposent un stockage complet de l'image et donc nécessitent des capacités de stockages bien supérieures à celles des BRAMs. Pour le traitement de flux de données scalaires (chapitre 5) cette limitation ne présente aucun impact.

Dans les chapitres suivants certaines chaînes ont nécessité l'exploitation des RAMs externes mais que dans des cas où, tel que précisé dans la section 3.4, nous avions la garantie que cet usage n'avait aucun impact sur les caractéristiques temporelles des blocs.

Troisième partie

Applications

Chapitre 4

Traitement de signaux 2D : application à la vidéo

Les premiers travaux, sur lesquels ont reposé une partie des spécifications de CoGen, correspondent à du traitement vidéo.

Plusieurs travaux vont être présentés dans la suite de ce chapitre :

- dans le but d'être en mesure de valider le fonctionnement d'un périphérique vidéo d'acquisition, la première étape a consisté en l'implémentation d'un bloc terminal dédié à la conversion du flux de la chaîne de traitement vers le protocole **CSI** (CMOS Sensor Interface), utilisé par de nombreux capteurs CMOS, afin de pouvoir transmettre, au processeur au travers du périphérique d'entrée compatible, un flux vidéo pour affichage sur l'écran disponible sur la carte APF27 ;
- une fois le transfert vidéo vers le processeur disponible, il est devenu possible d'implémenter un bloc de conversion du flux **cameralink** (protocole vidéo basé sur un transfert série de données à haute vitesse, au travers de lignes différentielles) vers un flux compatible avec la chaîne de traitement. C'est le sujet de la seconde section ;
- ayant interfacé une caméra rapide avec le FPGA afin de fournir un flux réel, il nous a été possible de réaliser des traitements en temps-réel, tels qu'un seuillage, traitement basique mais important pour évaluer d'une manière rapide la complexité induite par les contraintes mises en place au niveau des interfaces entre blocs. Dans un second temps, l'implémentation d'un algorithme de détection de contours, illustrant un cas plus complexe car nécessitant une notion de localisation dans l'image (gestion de la fin de ligne et de la fin de l'image principalement) ;
- finalement, ce chapitre est clôt sur une mise en application pratique du bloc de réception et des principes de base consistant en la caractérisation, en temps-réel, des modes propres de vibration d'un diapason par une méthode optique.

4.1 CSI

Le bus CSI (*CMOS Sensor Interface*), ou debug port dans les documentations de la carte SP_VISION, est un protocole synchrone maître-esclave entre une caméra CMOS et un CPU. La carte SP_VISION dispose d'un connecteur dédié, permettant de générer un flux à destination du processeur iMX27, en se comportant comme un capteur. L'implémentation réalisée a servi d'une part à valider le fonctionnement de la caméra, et d'autre part à transférer des flux de données importants, dans le sens FPGA-CPU, avec une absence de surcharge du processeur. Dans le cas de l'iMX27, la fréquence maximum du bus est de 66 MHz. Selon la configuration du processeur (16bits ou 24bits par pixels) il est possible de transférer 33Mo/s en 16bits ou 22Mo/s en mode 24bits.

4.1.1 Présentation du protocole

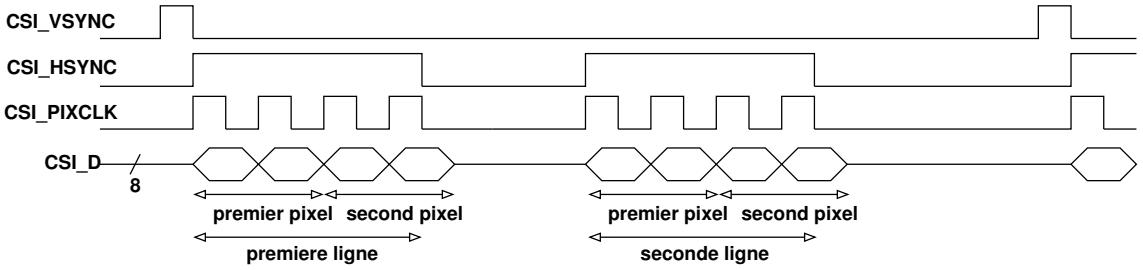


FIGURE 4.1 – Chronogramme de transfert d'une image de 2 lignes par 2 colonnes.

Le bus CSI est composé de 3 signaux de contrôle et du bus de données présenté sur la figure 4.1 :

- CSI_PIXCLK : signal d'horloge pour le cadencement du transfert. Dans le cas de l'iMX27 de l'APF27 qui doit recevoir exclusivement des images couleurs, et selon le mode sélectionné, deux ou trois cycles d'horloges peuvent être nécessaires pour transférer un pixel ;
- CSI_VSYNC : synchronisation verticale. Il passe à l'état haut pour signifier le début d'une nouvelle image et reste à l'état bas pendant le reste du transfert de l'image.
- CSI_HSYNC : synchronisation horizontale. Reste à l'état haut pendant le transfert des pixels d'une ligne, repasse à l'état bas entre chaque ligne.
- CSI_D : bus de données sur 8 bits.

Au démarrage de la transmission d'une image, le signal CSI_VSYNC passe à l'état haut pendant une durée déterminée, puis repasse à l'état bas. Ensuite, le signal CSI_HSYNC passe à son tour à l'état haut. Dans le même temps, le signal d'horloge est généré avec CSI_PIXCLK et les pixels sont propagés sur le bus. Une fois le dernier pixel de la ligne transféré, CSI_HSYNC et CSI_PIXCLK passent à nouveau à l'état bas, et les données sur CSI_D deviennent invalides. Après ce temps de pause, le même mécanisme que pour la première ligne se reproduit et ceci pour toutes les lignes de

l'image.

4.1.2 Implémentation

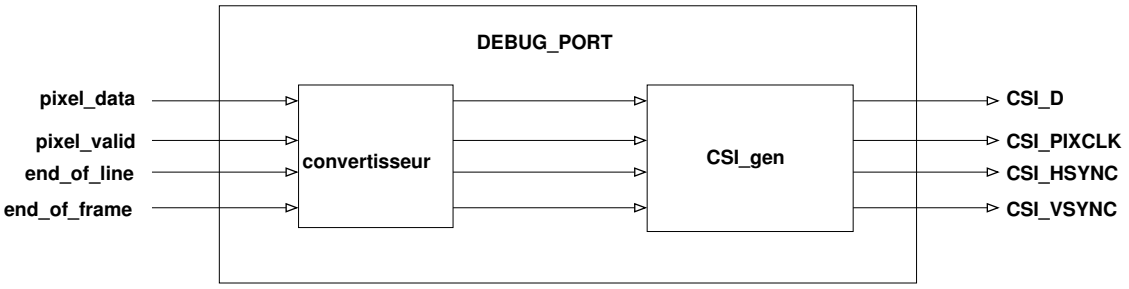


FIGURE 4.2 – Schéma du bloc terminal chargé de convertir un flux vidéo compatible CoGen vers un flux vidéo au format CSI. À gauche : conversion du format des données de la chaîne vers un format RGB565 ou RGB888. À droite : bloc dédié à la gestion des signaux du bus CSI.

Le bloc terminal en charge de la conversion depuis le format du flux de la chaîne de traitements vers le format d'un flux CSI (figure 4.2) est décomposé en deux entités :

1. une première entité qui a deux rôles :
   - changer le format du pixel : pour un flux en niveaux de gris codés sur 8 bits et une configuration en RGB565 sur 16 bits, la valeur de la donnée entrante sera dupliquée et tronquée pour obtenir deux copies : l'une sur 5 bits et l'autre sur 6. Dans le cas du format RGB888 la même donnée sera seulement dupliquée. Pour un flux couleur entrant, le même principe sera appliqué mais utilisant le canal adapté. Le résultat sera ensuite propagé en deux cycles d'horloge ;
   - le second rôle est de produire un signal de début de nouvelle transmission ainsi que l'avis de début et de fin de ligne.
2. la seconde entité a pour but de mettre en forme les signaux de contrôle en respectant les contraintes temporelles du protocole (durée de l'état haut du signal CSI_VSYNC, délais entre le front descendant de ce dernier et le front montant de CSI_HSYNC, temps de pause entre chaque ligne) avec application de délais sur les données pour les transmission.

4.1.3 Intégration dans CoGen

Dans le cas d'une chaîne traitant des données en niveaux de gris, le fichier de description XML contient :

```
<?xml version="1.0" encoding="utf-8"?>
<terminal_block name="debug_port" >
  <description> data flow to CSI convert</description>
  <timings>
```

```
        <timing dir="in" freq="33" pattern="1*"  />
  </timings>
  <block_interfaces>
    <interfaces dir="in" type="gray">
      <interface name="if_in" size="8" />
    </interfaces>
  </bloc_interfaces>
</terminal_block>
```

Il précise que sa capacité maximale de traitement est de 33 MHz, qu'il ne dispose que d'une entrée, de type niveaux de gris, et que la taille du bus est de 8 bits. Du fait de la simplicité du traitement, aucun bloc matériel n'est requis.

4.1.4 Démonstration

Pour valider le bon fonctionnement de notre implémentation du protocole CSI, nous avons réalisé l'application représentée sur la figure 4.3.

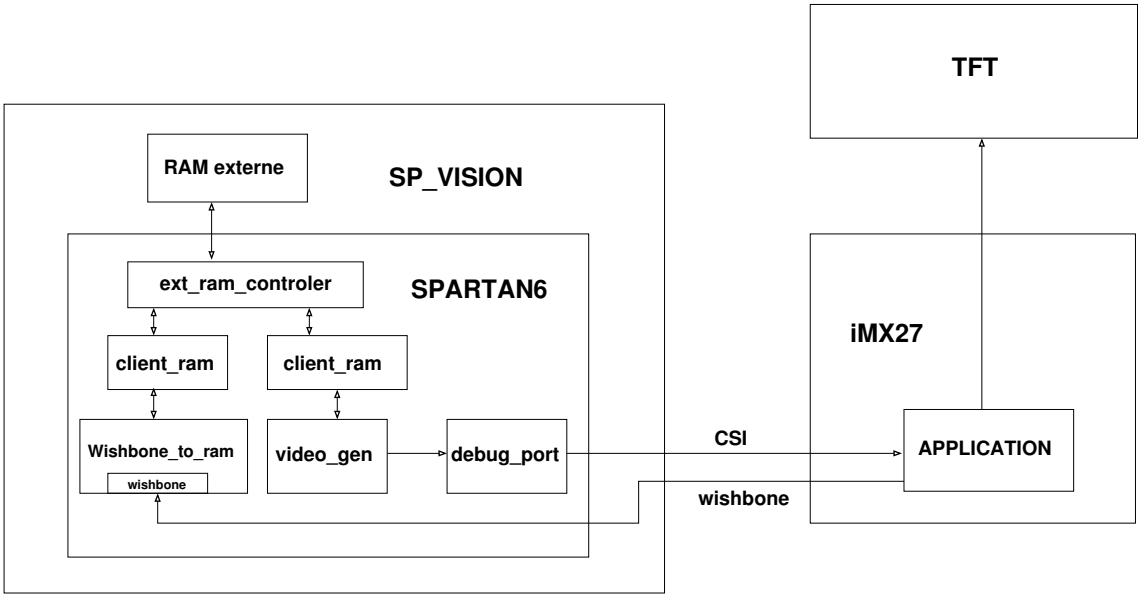


FIGURE 4.3 – Schéma de l'application HDL de test du fonctionnement de l'implémentation du protocole CSI.

Cette application consiste en la génération d'un flux vidéo compatible CoGen à partir d'une image en niveau de gris, préalablement stockée depuis le processeur de la carte APF27, par le bus wishbone, dans une des RAMs externes connectée au FPGA de la carte SP_VISION. Le bloc **video_gen** a la charge de produire en boucle un flux vidéo utilisé comme source pour le bloc terminal **debug_port**. Ce flux est ensuite converti et transmis au processeur sur son entrée CSI. Les données sont récupérées par une application pour un affichage sur l'écran de la carte. Le résultat obtenu est présenté sur la figure 4.4, qui nous permet de constater que l'image transférée en RAM

est effectivement affichée sur l'écran LCD, validant l'implémentation du protocole CSI.

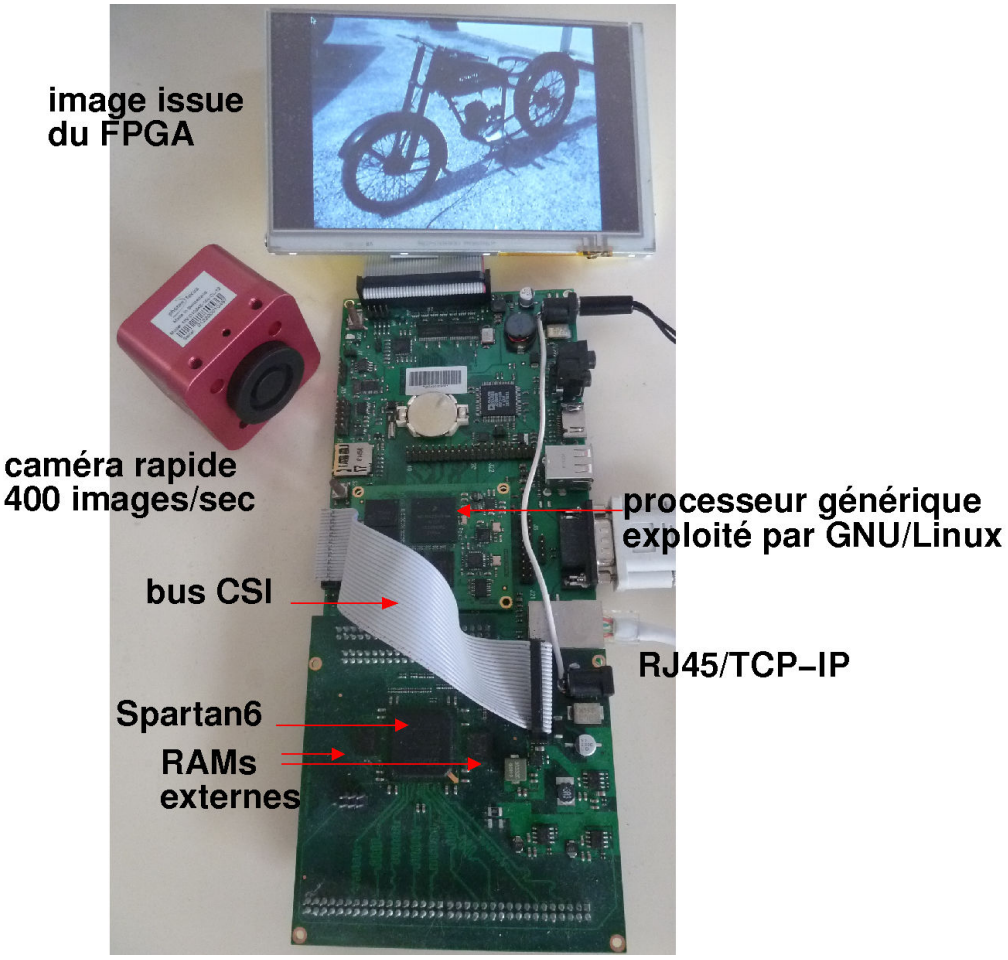


FIGURE 4.4 – Test du fonctionnement de l'implémentation HDL du protocole **CSI** : transmission d'une image, préalablement stockée dans une RAM externe, pour affichage.

4.2 Caméra Photonfocus

Ayant validé le bon fonctionnement du transfert vidéo, nous allons maintenant remplacer le flux artificiel par une source réelle.

La caméra *Photonfocus MVD1024E160* (figure 4.4 en haut à gauche) est un périphérique d'acquisition vidéo en niveau de gris. Le flux de données fourni est en LVDS (Low Voltage Differential Signals) présenté en 4.5. L'interface entre le périphérique et le FPGA est composée de :

- un signal d'horloge présentant une fréquence de 80 MHz (1x CLK) ;
- 4 paires différentielles pour le transfert des données à 7× la fréquence du signal d'horloge. Ces signaux sont utilisés, dans le cas de cette caméra, pour transférer la valeur de deux pixels consécutifs (R[0 :7] pour le premier, G[0 :7] pour le second, B[0 :7] n'est pas utilisé),

les signaux de synchronisation horizontale (**LVAL**) et verticale (**FVAL**) ainsi qu'un signal précisant si la donnée est valide ou non (**DVAL**), qui dans la pratique est toujours à l'état haut. Chaque paire fournit 7 bits.

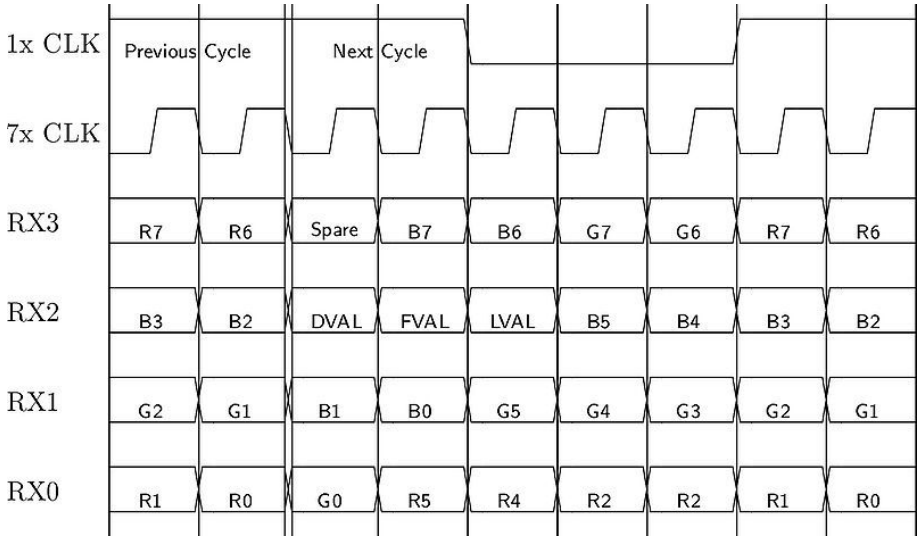


FIGURE 4.5 – Structure et représentation des bits transférés par une caméra au moyen du protocole cameralink. Référence : http://www.thefullwiki.org/Camera_Link.

Du point de vue temporel, tel que présenté sur la figure 4.6, la transmission d'une image est découpée en plusieurs étapes :

- un temps d'exposition, par défaut de 20ms, suivit d'un temps de pause,
- ensuite le signal **FVAL** passe à l'état haut. Un nouveau temps de pause est présent ;
- le signal **LVAL** et les données commencent à être propagées.
- A chaque fin de ligne, après le passage de **LVAL** à l'état bas un nouveau temps de pause est observé avant le transfert de la ligne suivante ;
- A la fin de la transmission de l'image un ultime temps de pause est imposé.

Le temps de pause est toujours une durée de $10 \times 10ns$.

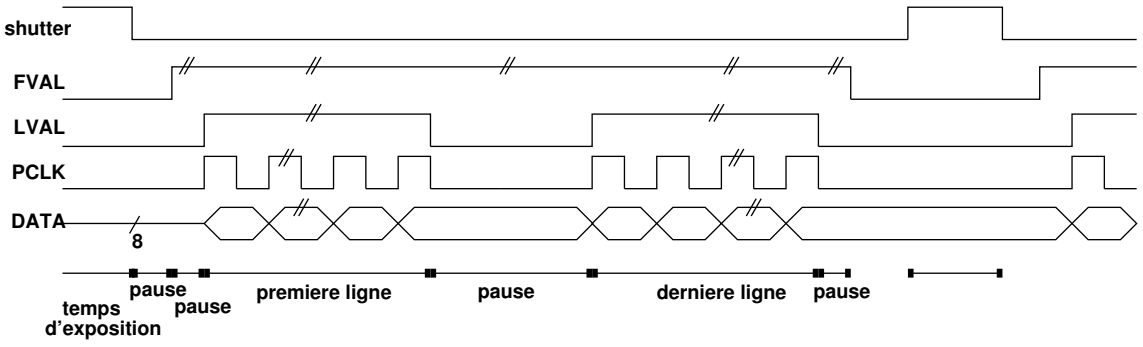


FIGURE 4.6 – Chronogramme du flux cameralink après désérialisation des données transmises.

4.2.1 Implémentation

Le bloc initial est composé de 3 sous-éléments :

- un composant qui convertit le flux série en un flux parallèle grâce à un ensemble de composants matériels disponibles dans le Spartan6 ;
- une FIFO utilisée pour réaliser le changement de domaine d'horloge entre la caméra et le FPGA. Cette étape est nécessaire pour éviter de perdre des données ;
- le dernier bloc transforme le flux depuis le format cameralink vers un flux vidéo compatible avec CoGen et propage les données à l'extérieur du bloc initial.

4.2.2 Désérialisation

La conversion du flux série LVDS, dans un Spartan6, vers un flux parallèle nécessite l'utilisation d'un ensemble de blocs matériels pour :

- la génération d'une horloge sept fois plus rapide que celle fournie par la caméra. Ceci est obtenu à l'aide de blocs **PLL_ADV** ;
- la détection d'un motif particulier entre l'horloge rapide et l'horloge lente pour la synchronisation des blocs de désérialisation. Ce motif correspond généralement à deux cycles avant le front descendant (rapport entre les deux signaux d'horloges sur la figure 4.5) du signal lent, ou en représentation binaire : **1100011** ;
- la conversion du flux série vers le flux parallèle (la désérialisation) à proprement parler reposant sur l'utilisation de blocs **serdes** cascadés et pouvant chacun faire une conversion 1 vers 4, c'est-à-dire que sur un flux série (un bit en entrée par cycle d'horloge rapide) ils vont fournir, au bout de quatre cycles de cette même horloge, quatre données en parallèle.

Les données, une fois extraites de la trame, sont ensuite transférées à la FIFO pour stockage et exploitation depuis le bloc en charge de la conversion de flux.

4.2.2.1 Composant FIFO et génération du flux compatible CoGen

Le flux cameralink, une fois reconstruit (figure 4.6), n'est pas directement exploitable par CoGen, il est donc nécessaire de réaliser la conversion de ce format vers le format compatible avec CoGen.

Contrairement au protocole CSI, une des difficultés est que le signal **FVAL** (signal de synchronisation verticale) est à l'état haut pendant tout le transfert de l'image : il n'est donc pas possible de simplement tester l'état de ce signal pour détecter le début de la transmission. Il est nécessaire, à l'initialisation du bloc et avant le début de la propagation, d'attendre son passage par '0' signifiant que l'image en cours est entièrement transmise, puis d'attendre le front montant suivant. Avec cette condition initiale, il est possible de garantir que le bloc sera synchronisé avec la caméra et ne transférera pas une image partielle.

Le signal **LVAL** peut servir, modulo un retard des données par rapport à ce signal, à gérer le signal sortant de fin de ligne sans avoir à connaître a priori la largeur de l'image. Au contraire, le signal **FVAL** ne repasse à l'état bas qu'après un temps de pause, une fois l'arrivée du dernier pixel de l'image. Compte tenu de cette durée relativement longue, il n'est pas possible de simplement décaler temporellement les pixels. Un compteur de lignes a dû être mis en œuvre.

4.2.3 Description du bloc cameralink

Le bloc est décrit par le fichier xml présenté en 4.7.

```
<?xml version="1.0" encoding="utf-8"?>
<initial_block name="cameralink" >
  ...
  <timings>
        <timing dir="out" freq="80" exposure_time="10000"
      pattern="0(EXPOSURE_CYCLES)0(8)[1(img_height)0(8)](img_width)"  />
  </timings>
  <bloc_interfaces>
    <interfaces dir="out" type="gray">
      <interface name="if1_in" />
      <interface name="if2_in" />
    </interfaces>
  </bloc_interfaces>
  ...
</initial_block>
```

FIGURE 4.7 – Fichier de description XML du bloc cameralink.

Dans cette description, la balise **pattern** est décrite sur la base de temps de la caméra. La balise **exposure_time** et la variable **EXPOSURE_CYCLES** sont des mots-clés liés : dans le cas de cette caméra à 80 MHz, **EXPOSURE_CYCLES** vaut 800.

Sachant que les composants sont cadencés par l'horloge du FPGA, il est nécessaire de réaliser un changement de domaine d'horloge. Du fait de la dérive entre les deux horloges, le motif après expansion doit être modifié afin de faire ressortir cette information. Dans le cas d'un FPGA à 100 MHz, il apparaît (figure 4.8) des cycles où aucune donnée ne sera vue dans le domaine d'horloge du FPGA. La période de ces "blancs" est quantifiable, et représente un blanc toutes les 4 données. De ce fait, dans l'expansion, un caractère '0' sera ajouté tous les 4 caractères, donnant ainsi une répétition de ce motif tous les 5 cycles, puisque le rapport vaut $\frac{100}{80} = \frac{5}{4}$. Par exemple, le début de cette chaîne deviendra 11110111101... au lieu de 111111111...

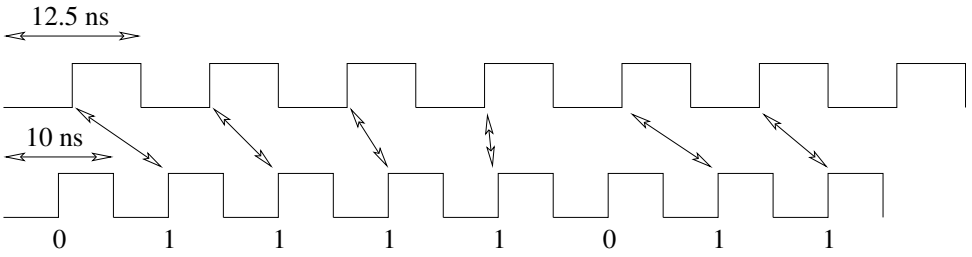


FIGURE 4.8 – Chronogramme de la prise en compte du changement de domaine d'horloge entre une horloge à 80 MHz (12.5 ns de période) et une horloge à 100 MHz (10 ns de période). Nous pouvons constater que tous les 5 cycles de l'horloge à 100 MHz un "blanc" apparaît, induit par la dérive d'un facteur $\frac{100}{80} = \frac{5}{4}$.

4.2.4 Démonstration du fonctionnement de la caméra Photonfocus

Cette première démonstration (schéma 4.9), la plus simple possible, consiste à transmettre les données de la caméra, sans aucun traitement, au bloc *debug_port* et par extension au processeur iMX27 pour affichage (figure 4.10).

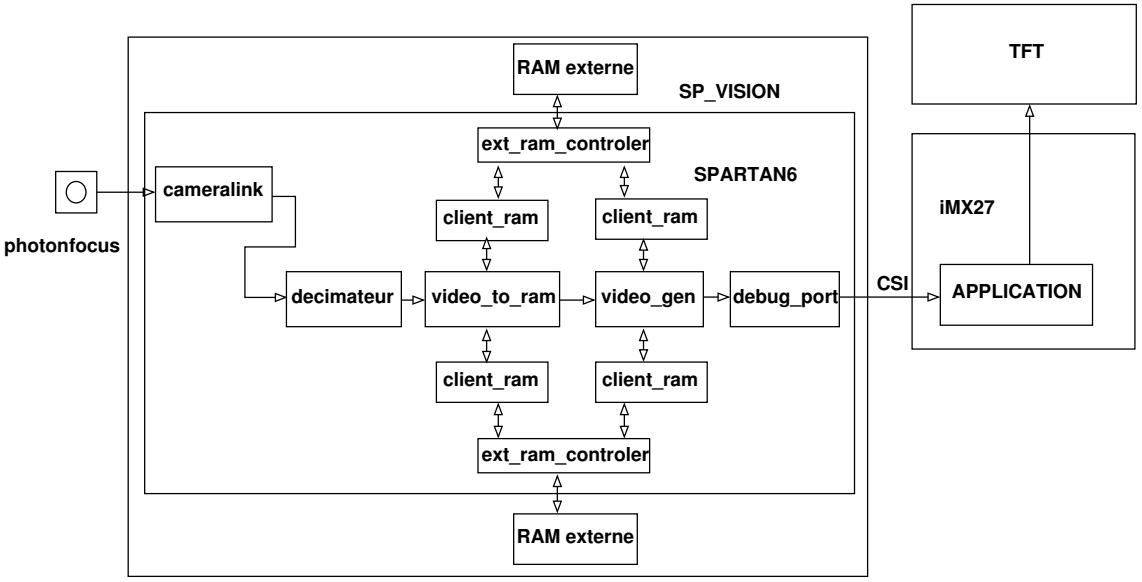


FIGURE 4.9 – Test de la caméra Photonfocus : schéma de l'application FPGA.

Sachant que le périphérique CSI du processeur ne peut pas dépasser 33 Mpixels/s, une première étape consiste à stocker l'ensemble des pixels dans une RAM externe, seule capable d'absorber un tel volume de données. Un nouveau flux compatible avec le *debug_port* est ensuite généré, à l'instar de la démonstration de test du bloc terminal. Deux points sont à remarquer :

– le flux à la sortie de la caméra est cadencé par une horloge à 80 MHz pour deux pixels, soit un total de 160 Mpixels/s. Ce débit est bien supérieur à ce que le bus CSI est capable

d'absorber : un bloc de décimation a donc été ajouté en amont du stockage ;

– pour éviter tous les problèmes liés aux latences des RAMs externes, et garantir que leur comportement, grâce aux FIFOs, sera équivalent à celui d'une BRAM, les deux RAMs sont utilisées à tour de rôle, l'une en écriture pendant que l'autre est lue et réciproquement (mécanisme de ping-pong). Une interface particulière est également mise en place entre le bloc d'acquisition et le générateur afin de synchroniser la lecture avec l'écriture.



FIGURE 4.10 – Test de la caméra Photonfocus : transmission d'une image, reçue depuis la caméra.

4.3 Ajout d'un bloc de traitement dans la chaîne : filtre de seuillage

Ce type de filtre fait partie des plus simples dans le cadre du traitement vidéo : dans le cas d'une image en niveau de gris, la valeur du pixel est comparée à un seuil. Si la valeur est inférieure au seuil, la nouvelle valeur devient 0, si elle est supérieure ou égale au seuil, le résultat devient $2^n - 1$ avec n le nombre de bits du pixel.

4.3.1 Implémentation

L'implémentation présentée sur la figure 4.11 est relativement simple : elle correspond à une comparaison de la valeur à un seuil. Le résultat est ensuite utilisé comme bit de sélection pour un démultiplexeur dont les deux entrées sont fixées statiquement au moment de la synthèse. Le résultat,

ainsi que l'ensemble des signaux de contrôles, sont ensuite mémorisés de manière synchrone.

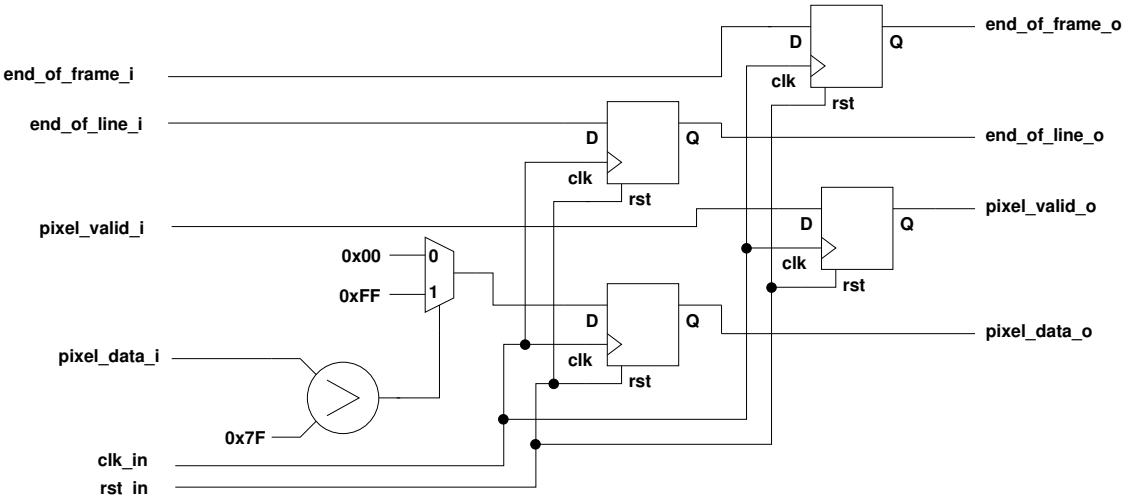


FIGURE 4.11 – Schéma du bloc d'implémentation d'un seuillage simple.

Cette dernière étape peut sembler inutile compte tenu de la relative simplicité d'un tel code. Toutefois, par mesure de sécurité, il est préférable, ne pouvant faire d'hypothèses sur le bloc précédent et suivant, d'ajouter une mémorisation afin de s'affranchir de tous les problèmes de temps de propagation des signaux, et dans le cas d'implémentations plus complexes, de garantir la cohérence des données. Cette étape ajoute 1 cycle d'horloge de latence. Si cette latence est un mal nécessaire, il n'en reste pas moins que c'est un délai qui est négligeable : pour une horloge à 100 MHz cette durée correspond à 10 ns. Une caméra telle que la photonfocus, dans sa configuration par défaut, nécessite pour une image la somme de :

- 10 ms pour le temps d'exposition ;
- $(1024 \times \frac{1}{80MHz}) \times 1024$ pour le transfert de l'ensemble des pixels de l'image ;
- $(10ns \times 10) \times 1024$ pour le temps de pause à la fin de chaque ligne.

Le total de 23ms est significativement plus grand que les 10ns de latence dûes à l'étape de mémorisation.

Pour un tel bloc, le fichier du bloc de traitement et de l'implémentation correspondent aux figures 4.12 et 4.13

Cette première application, dont le résultat est présenté sur la figure 4.14, montre bien qu'il est possible, sans augmentation de la complexité du développement, de réaliser des blocs compatibles CoGen. Les signaux de contrôle, qui ne sont pas exploités dans cet exemple, doivent être propagés en respectant les mêmes délais que les données pour éviter une désynchronisation.

```
<?xml version="1.0" encoding="utf-8"?>
<processing_block name="threshold" >
  <description>Simple threshold block</description>
  <user_config>
    <param name="threshold" default="128" />
  </user_config>
  <implementations>
    <implementation>th_1</implementation>
  </implementations>
  <interfaces>
    <interface name="if_in" type="video" dir="IN" />
    <interface name="if_out" type="video" dir="OUT" />
  </interfaces>
</processing_block>
```

FIGURE 4.12 – Fichier de description du bloc de traitement pour un seuil simple. L'utilisateur peut changer le paramètre du seuil qui est fixé par défaut à 128.

```
<?xml version="1.0" encoding="utf-8"?>
<implementation_bloc name="th_1" >
  <description>Simple threshold implementation</description>
  <timings>
    <timing dir="in" pattern="1(IMG_WIDTH*IMG_HEIGHT)" />
    <timing dir="out" pattern="1(IMG_HEIGHT*IMG_WIDTH)" latency="1" />
  </timings>
</implementation_bloc>
```

FIGURE 4.13 – Fichier de description de l'implémentation d'un seuillage simple. Ce bloc est capable d'accepter des données à chaque cycle d'horloge et ne modifie pas le flux.

4.4 Bloc de traitements : filtre de détection de contours

L'implémentation d'un seuillage (section 4.3), appliqué au flux, a permis de réaliser une première mise en pratique réelle des concepts présentés précédemment. Toutefois, cette démonstration, de par sa simplicité, ne permet pas de tirer profit de l'ensemble des informations fournies au travers de signaux de contrôle, pour implémenter un traitement complexe sans obligation systématique d'ajout d'une logique dédiée à connaître la position courante dans l'image.

L'algorithme de ce filtre de détection de contours, dont le résultat visuel est représenté sur la figure 4.15, ainsi que les diverses solutions d'implémentation relatives aux multiplieurs, ayant été présentées dans la section 2.6 du chapitre 2, nous allons nous focaliser sur la partie gestion et organisation de la mémorisation des données pour leur utilisation ultérieure.

Dans la figure 1.2, la production du nouveau pixel $P_{1,1}$, résultat de la convolution par une matrice 3×3, nécessite de disposer d'un ensemble de pixels. Ce traitement ne peut être réalisé

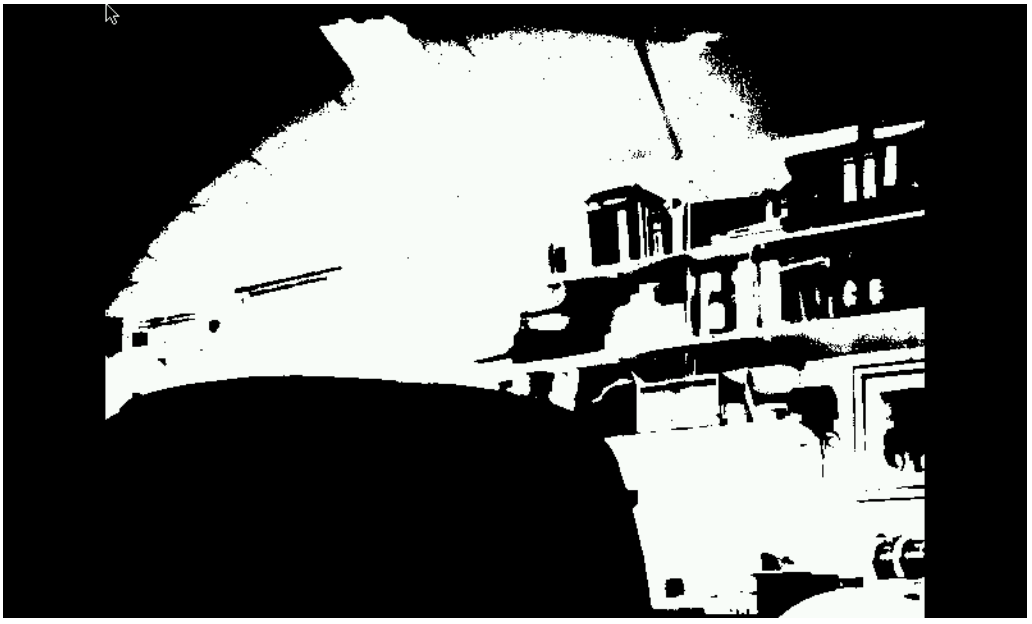
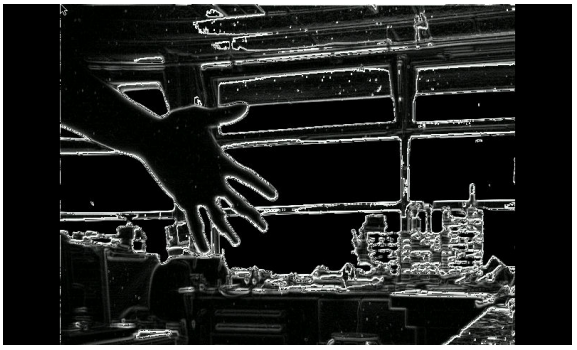


FIGURE 4.14 – Résultat de l'application d'un filtre de seuillage en temps-réel sur le flux issu d'un capteur vidéo.



(a) Scéne filmée sans traitement



(b) Même scéne que 4.15(a) mais avec application d'un filtre de Sobel directement sur le flux issu de la caméra. Dans cette image, une main, au premier plan, est présente afin de montrer, plus finement, le résultat de la détection de contours.

FIGURE 4.15 – Images issues de la caméra Photonfocus : 4.15(a) image non traitée ; 4.15(b) résultat du filtrage de l'image appliquée directement sur le flux avant transfert au processeur.

avant d'avoir réceptionné le pixel $P_{2,2}$. Une étape de mémorisation des pixels doit être réalisée pour disposer de ceux-ci lorsque le besoin apparaît. À chaque fin de réception d'une ligne, et pour le traitement de la ligne suivante, il est également nécessaire de réassigner les données stockées : les pixels de la ligne du haut ne sont plus nécessaires, le contenu de la ligne du milieu va être utilisé pour les calculs des $P_{y-1,}$ et la ligne du bas va servir pour les $P_{y,}$.

4.4.1 Implémentation

Comme présenté, il est nécessaire de mémoriser, pour en disposer par la suite, l'ensemble des données des trois lignes. Plus exactement les pixels des lignes y et $y-1$ sont obtenus depuis des BRAMs, pour la dernière ligne $(y+1)$ les données doivent être à la fois mémorisées dans une BRAM, pour leur usage ultérieur (traitement sur les deux lignes suivantes), et stockées dans un simple registre à décalage disposant de **KERNEL_SIZE** cases dédiées aux traitements immédiats de la ligne du bas de la matrice de convolution.

4.4.1.1 Stockage des données dans les RAMs

La partie stockage (figure 4.16) repose sur l'utilisation des trois BRAMs en mode **dual port** (accès possible par deux *processes* en parallèle) avec un mécanisme de tampon circulaire.

Deux RAMs sont utilisées pour la lecture, la troisième pour le stockage. Le rôle de chaque RAM évolue pour chaque nouvelle ligne : pour une ligne y quelconque, une de ces RAMs reçoit les données, à la ligne suivante $(y+1)$, la même RAM sert de source pour les calculs de la ligne du milieu du noyau et finalement, à la ligne $y+2$ cette RAM est utilisée pour la ligne du haut du noyau. Pour la ligne suivante $(y+3)$, elle est à nouveau utilisée pour le stockage.

Pour réaliser le stockage, le signal d'activation pour l'écriture est connecté à un multiplexeur (bas gauche de la figure 4.16) afin de pouvoir sélectionner, en écriture, seulement une des trois RAMs. Les autres signaux (adresse et données) sont simplement connectés à toutes les BRAMs. Le signal de sélection du multiplexeur évolue lors du passage à l'état haut du signal de fin de ligne, permettant ainsi de basculer séquentiellement d'une RAM à une autre.

4.4.1.2 Récupération des données pour utilisation

Ce traitement est présenté en partie droite de la figure 4.16. Le bus d'adresse est commun d'une part à toutes les RAMs, et d'autre part à la lecture et à l'écriture. Pour l'organisation des données, comme la relation entre une RAM et un des registres à décalage évolue avec le temps et le passage d'une ligne à l'autre, une série de multiplexeurs est connectée sur la sortie des RAMs et piloté par le même compteur que les démultiplexeurs liés au stockage. Deux multiplexeurs sont nécessaires, car comme présenté précédemment, les données nécessaires pour le calcul de la ligne $y+1$ du noyau sont directement issues de l'entrée du bloc.

4.4.1.3 Intégration dans CoGen

Il existe plusieurs versions de ce bloc, telles que présentées dans la section 2.6. Ici, nous allons uniquement présenter la première version qui fait usage de blocs DSP48, avec un noyau de convolution de 3×3.

La description, présentée en 4.17, indique que l'implémentation nécessite 9 DSP48 et 3 RAMs. Le bloc est capable de recevoir des données à chaque cycle, ne modifie pas les dimensions du flux

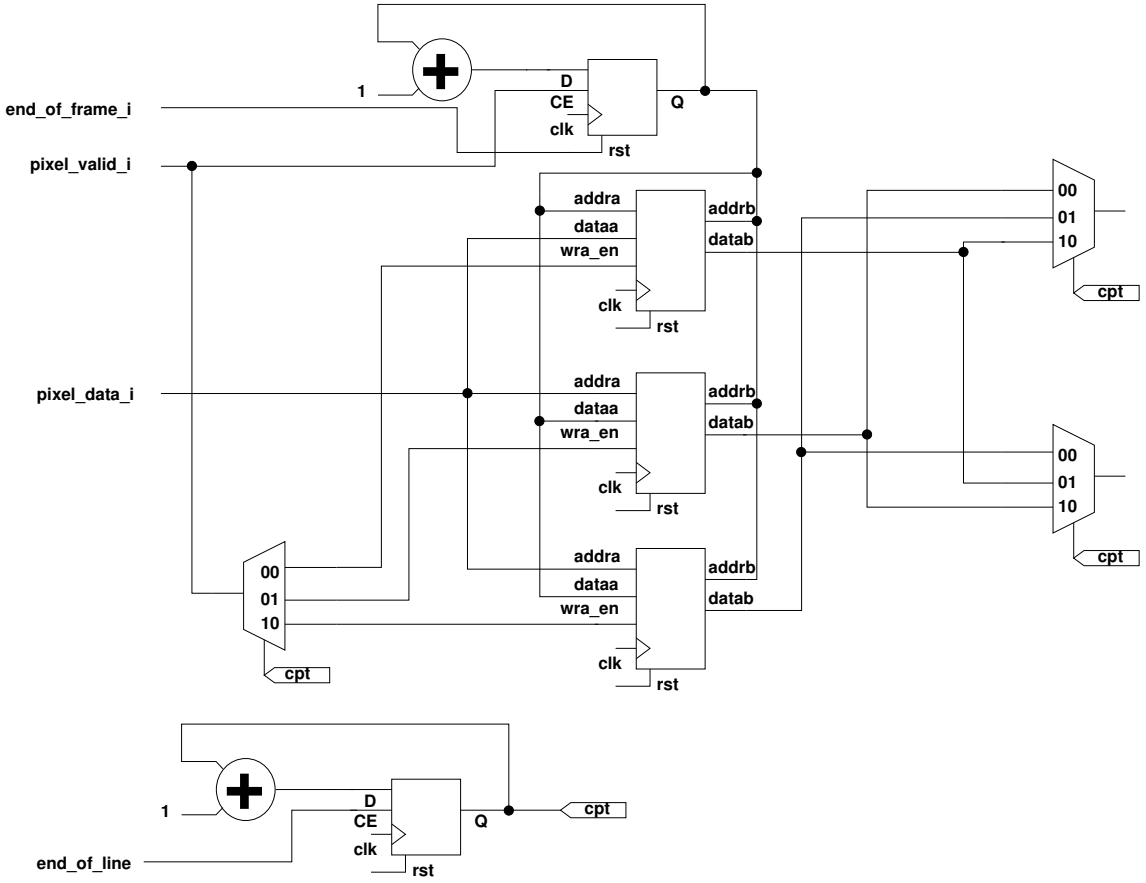


FIGURE 4.16 – Schéma de l'implémentation du stockage des données dans des BRAMs avec un principe de tampon rotatif pour l'écriture (partie gauche) et la lecture (partie droite). La sortie de l'accumulateur en bas à gauche (signal **cpt**) est utilisé comme contrôle pour l'écriture dans une RAM particulière et pour le lien entre chaque RAM et les multiplications. Sa valeur évolue à chaque fin de ligne.

et nécessite, en plus de la latence propre au traitement, un délai équivalent à 2 lignes complètes et 2 pixels.

4.5 Caractérisation des modes propres d'un diapason par méthode optique

En utilisant la méthodologie décrite précédemment, nous nous concentrons sur une application pratique, à savoir la mesure de déplacement en temps réel d'un diapason de musique par traitement d'image (figure 4.18). La mesure quantitative d'un déplacement avec une résolution sub-pixel est basée sur la mesure de phase par corrélation entre un modèle de référence et un motif d'image (figure 4.19) attaché à l'objet en mouvement [45, 46]. Afin de souligner la vitesse de traitement

```
<?xml version="1.0" encoding="utf-8"?>
<terminal_block name="edge_1" >
  <description> 3x3 fast edge detector</description>
  <timings>
        <timing dir="in" pattern="1(IMG_WIDTH*IMG_HEIGHT)"  />
        <timing dir="out" latency="2"
      delay="2*(IMG_WIDTH*IMG_HEIGHT)+2" pattern="1(IMG_WIDTH*IMG_HEIGHT)"  />
  </timings>
  <resources>
    <resource type="DSP48" amount="18" />
    <resource type="BRAM" amount="3" />
  <resources>
  <block_interfaces>
    <interfaces dir="in" type="gray">
      <interface name="if_in" size="8" />
    </interfaces>
    <interfaces dir="out" type="gray">
      <interface name="if_out" size="8" />
    </interfaces>
  </bloc_interfaces>
</terminal_block>
```

FIGURE 4.17 – Description XML d'un des blocs d'implémentations de l'algorithme de détection de contours.

élevée, nous démontrons ce concept sur un diapason vibrant à 440 Hz, puis sur la caractérisation de la réponse impulsionnelle de ce même objet pour en identifier les modes de vibration jusqu'à 12 kHz.

L'algorithme de traitement du signal, à mettre en œuvre sur le FPGA, vise à identifier la phase de la composante de Fourier du motif périodique imprimé sur une cible attachée à l'objet en mouvement. Étant donné que le grossissement de l'image de la cible, ainsi que les paramètres géométriques du modèle, sont connus, le coefficient de Fourier associé à une seule longueur d'onde est nécessaire. Ainsi, la fonction d'analyse de l'échantillon de longueur finie L (largeur de l'image) est donnée par

$$analyse = \underbrace{\exp\left(-\left(\frac{x}{4p\sqrt{2}}\right)^2\right)}_{\text{fenêtrage}} \times \underbrace{\exp\left(2i\pi \times \frac{x}{p} + \varphi_0\right)}_{\text{fonction de projection}}$$

avec $x = [1..L] - \overline{x}$ l'ensemble des pixels d'une ligne, p la période du motif (en pixel) et $\overline{x}$ l'indice du point milieu de la ligne. L'enveloppe gaussienne réduit les influences des artéfacts associés à la transformée de Fourier pour une série d'échantillons de longueur finie. Ayant identifié la fonction d'analyse utilisée pour un motif de référence, une seule ligne de l'image $mesure_k$ ($k \in [1..L]$)

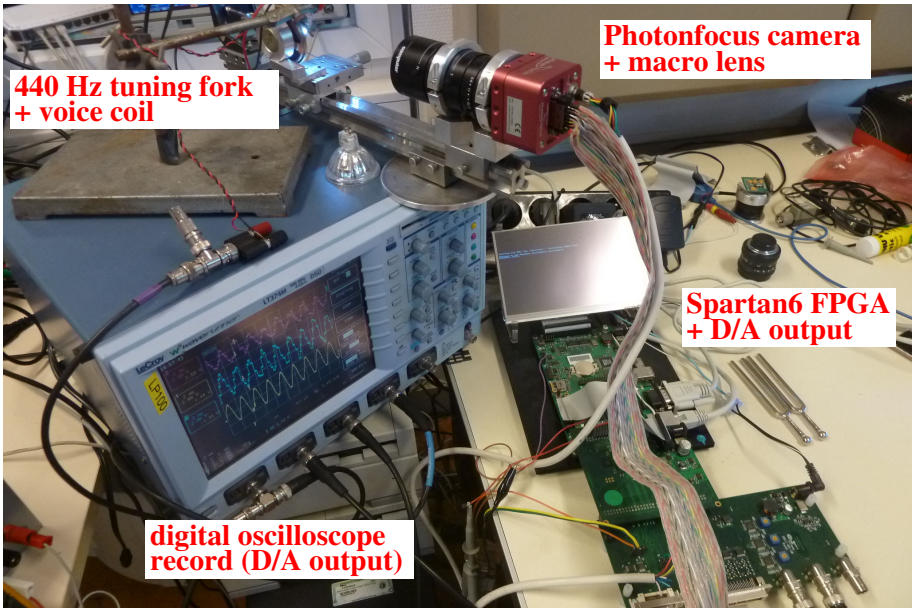


FIGURE 4.18 – Montage expérimental pour la mesure de vibration d'un diapason musical par traitement d'images. Le haut-parleur proche du diapason est utilisé pour exciter le diapason à la fréquence de 442 Hz (proche de la valeur nominale de 440 Hz, due à une modification du diapason pour le collage d'une mire à son extrémité.

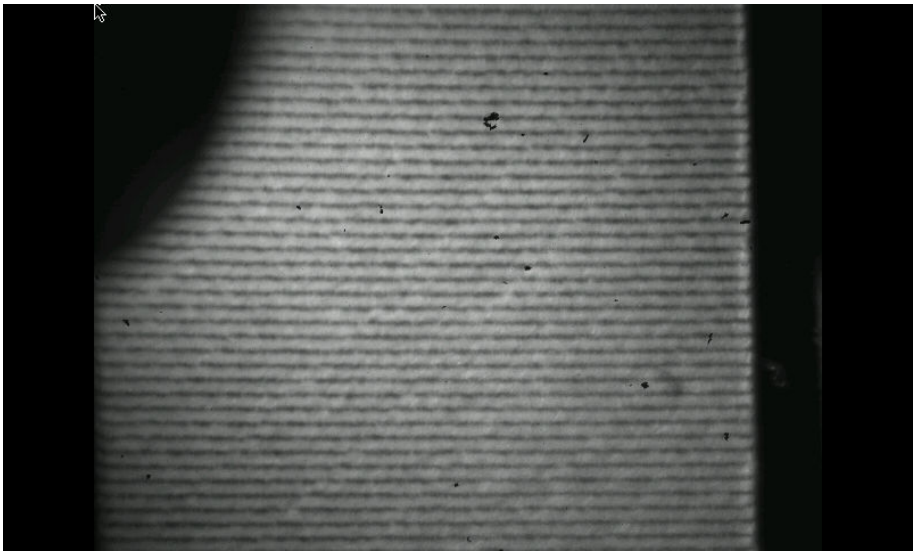


FIGURE 4.19 – Cible disposant d'un réseau périodique – un motif imprimé d'une résolution de 300 dpi – collée sur le bout d'une des branches d'un diapason musical à 440 Hz et observée par une caméra pour une détection de mouvement.

présentant le motif cible fournit une information de phase φ grâce à l'inter-corrélation : $\varphi = arg\left(\Sigma_{k=[1..L]} mesure_k \times analyse_k\right)$.

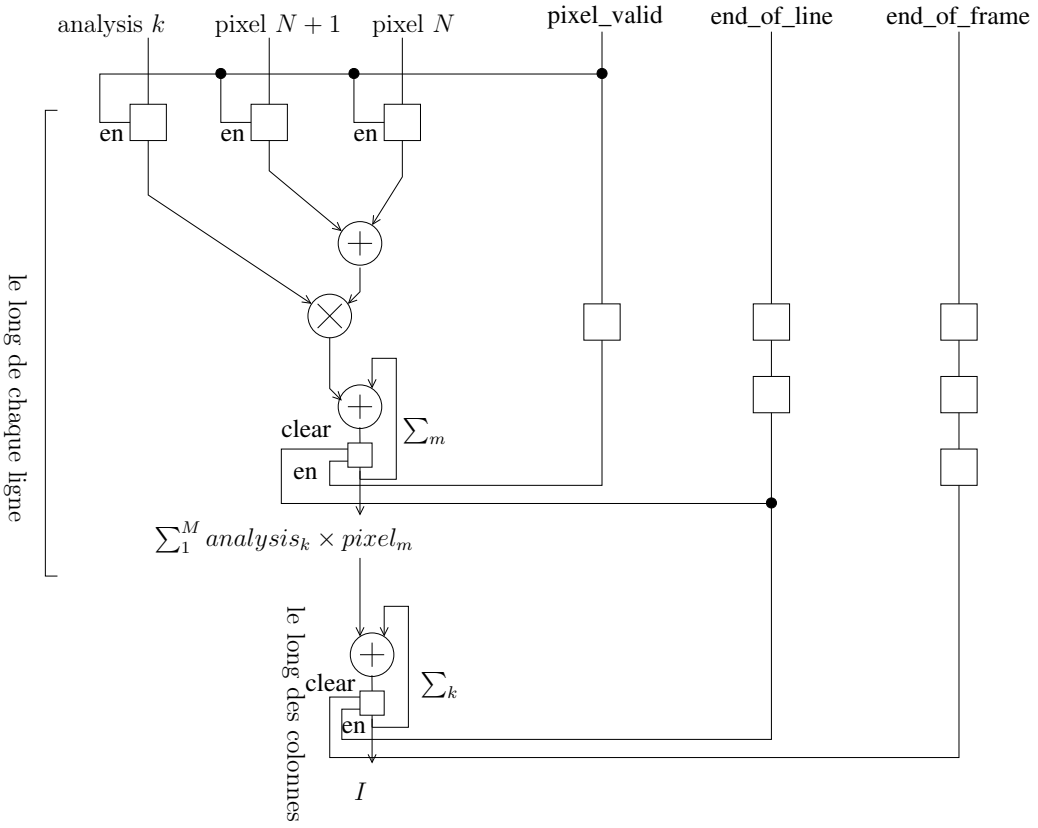


FIGURE 4.20 – Schéma du bloc de traitement vidéo : comme deux pixels consécutifs sont transférés par la caméra à chaque cycle d'horloge, la somme des deux pixels consécutifs est réalisé par un premier additionneur. La multiplication par la fonction d'analyse est ensuite réalisée, puis le résultat est accumulé pour calculer l'inter-corrélation le long de la ligne. Le calcul est effectué en parallèle sur la partie réelle (composante I) et sur la partie imaginaire (composante Q) avec seul le changement de la fonction d'analyse de $\cos(2\pi x/p)$ en $\sin(2\pi x/p)$.

Comme le DSP48, bloc matériel contenant un multiplieur et un accumulateur, est optimisé pour traiter des données sur 18 bits, la fonction d'analyse est multipliée par 32765 pour être un entier signé sur 16 bits. Dans notre cas, $L = 1024$, $p = 32$ et $\overline{x} = 512$. φ_0 est une constante ajustée pour éviter les rotations de 2π lorsque le calcul du déplacement s'éffectue près des bornes de l'intervalle $[0 : 2\pi]$

La source d'images est une caméra Photonfocus MVD1024E160 communiquant au travers du protocole Cameralink. Pour nos traitements, nous avons pu baisser le temps d'exposition (attribut `frame_exposure` du bloc 4.7) à la valeur de 10 μs grâce à l'utilisation d'une source lumineuse intense à base de LEDs.

Le débit réel est de 5285 images, exclusivement limité par la caméra, puisque tous les traitements présentés figure 4.21, sont implémentés par l'utilisation d'un pipeline décrit dans la figure 4.20. Cette implémentation est capable de recevoir deux pixels consécutifs (deux balises de type

```
<?xml version="1.0" encoding="utf-8"?>
<implementation_bloc name="tuning_fork" >
    ...
    <timings>
        <timing dir="out" pattern="0(img_height*img_width)01"  />
    <timing dir="in" pattern="1*" />
    </timings>
    <bloc_interfaces>
        <interfaces dir="in" type="gray">
            <interface name="if1_in" />
            <interface name="if2_in" />
        </interfaces>
        <interfaces dir="out" type="gray">
            <interface name="real_out" />
            <interface name="im_in" />
        </interfaces>
    </bloc_interfaces>
    ...
</implementation_bloc>
```

FIGURE 4.21 – Description XML du bloc d'implémentation relatif au calcul pour le diapason.

interface en mode **in**) à chaque cycle d'horloge (acceptance en $1*$) et de produire un résultat de type complexe (deux balises **interface** en mode **out**) deux cycles après l'arrivée du dernier lot de pixels de la ligne (sortance valant $0(IMG_HEIGHT \times IMG_WIDTH)01$ avec IMG_HEIGHT la hauteur de l'image, IMG_WIDTH la largeur). Toutefois, cette analyse présente une limitation significative due à l'orientation de la caméra puisque chaque fin de ligne implique un délai de 100 ns qui, pour une image de 1024 lignes, représente 102.4 μs.

Dans le but d'éviter les limitations associées au temps de transfert des informations de déplacement vers un CPU, les données sont sorties vers deux convertisseurs numériques-analogiques, fournissant une tension proportionnelle aux composantes I et Q de l'inter-corrélation. Les Figures 4.22 et 4.23 présentent l'affichage sur un oscilloscope du résultat du calcul lors de la vibration du diapason à 440 Hz, excité magnétiquement par un haut-parleur [47]. Les données I et Q obtenues par l'oscilloscope sont enregistrées et l'extraction de la phase est réalisée en post-traitement, comme étant $\varphi = \arctan(\frac{Q}{I})$, évitant l'implémentation de la fonction arctan() dans le FPGA. Dans le cas de l'utilisation de telles informations dans une boucle d'asservissement, le traitement sur les coefficients I et Q est souvent suffisant et l'étape de calcul de l'arctan s'avére une réelle difficulté.

4.5.1 Implémentation

Compte tenu de la nature particulière du flux fourni par la caméra Photonfocus, deux implémentations ont étés produites. Dans les deux cas plusieurs points sont communs :

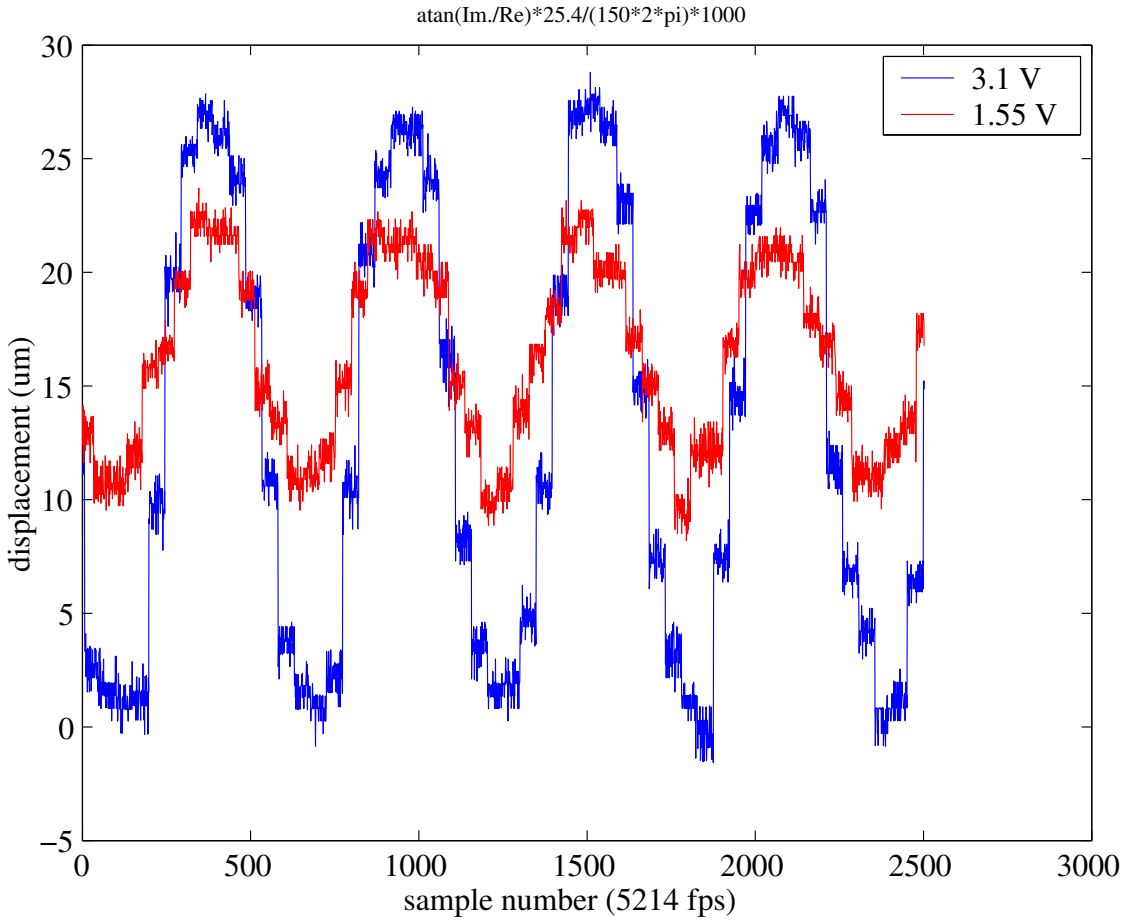


FIGURE 4.22 – Capture sur un oscilloscope numérique de la sortie des convertisseurs numérique-analogiques, représentant les composantes I et Q du calcul de la inter-corrélation, convertis en angle, par le calcul de arctan(Q/I) en post-traitement. La fréquence de rafraîchissement est de 5.2 kHz, bien supérieure à la fréquence d'oscillation forcée de 442 Hz.

1. les constantes utilisées pour les multiplications par les coefficients d'analyse sont chargées dans deux RAMs, l'une pour la partie réelle, l'autre pour la partie imaginaire. Le chargement est fait depuis le processeur une fois le FPGA programmé ;
2. les multiplications sont de type complexe, toutefois les pixels sont des réels. Il est donc possible de supprimer les deux multiplications utilisant la partie imaginaire du pixel afin d'économiser des ressources.

4.5.1.1 Première implémentation

Sachant que la caméra produit deux pixels consécutifs, que chaque DSP48 dispose d'un additionneur avant l'étape de multiplication et en vue d'économiser le nombre requis de multiplieurs, la première implémentation a consisté à tourner la caméra de 90°. Ainsi, chaque coefficient est utilisé pour une ligne entière.

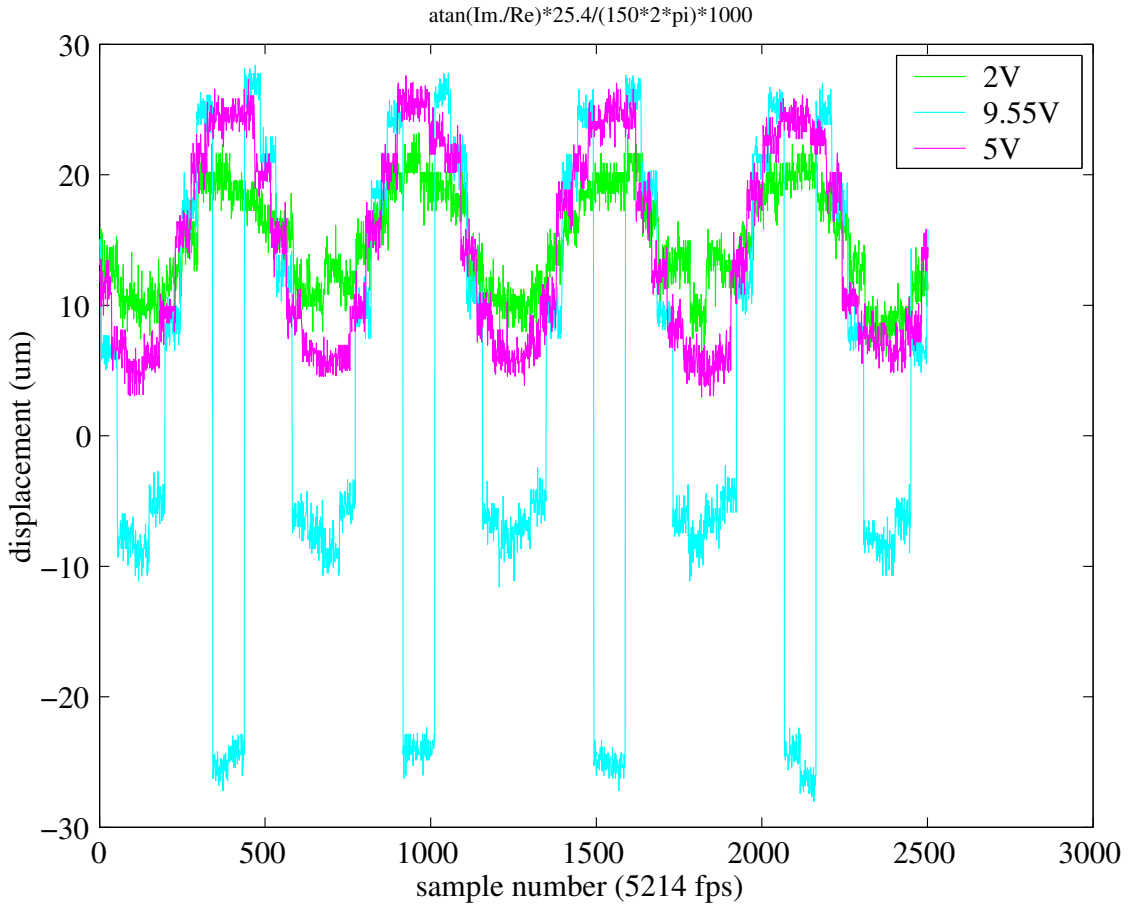


FIGURE 4.23 – Phase calculée par $\arctan(Q/I)$ et mesurée pour diverses tensions d'excitation du haut-parleur. Pour la tension la plus importante, la rotation de phase de 2π est visible (courbe bleue). Régler l'offset de phase, φ_0, maximise la plage de mesure en conservant la rotation de phase dans la gamme $[\pi/2, \pi/2]$.

Le schéma de l'implémentation est représenté en figure 4.20

Le DSP48 dispose de trois lignes de données en entrée, d'un additionneur en amont de la multiplication et d'un additionneur en sortie de la multiplication qui peut-être utilisé en tant qu'accumulateur (rebouclage de la sortie du dernier registre sur une des entrées de l'additionneur).

Ainsi le principe, pour tous les pixels d'une ligne est le suivant :

- les deux pixels et le coefficient sont verrouillés dans les registres en entrée du DSP48 à l'aide du signal *pixel_valid*,
- ces deux valeurs sont propagées dans le pré-additionneur, puis dans le multiplieur et finalement dans le post-additionneur.
- le signal *pixel_valid*, retardé de 1 cycle d'horloge, permet de remettre à jour le contenu du registre de sortie.

A l'issue d'une ligne, le signal *end_of_line* est utilisé pour incrémenter l'offset des RAMs, qui aura pour effet de mettre à disposition le coefficient suivant. Ce même signal, retardé de deux

cycles, est utilisé pour, à la fois réinitialiser l'accumulateur contenu dans le bloc DSP48, et pour mettre à jour le registre d'un second accumulateur qui somme les résultats par ligne.

Finalement, la réception d'un niveau haut sur le signal *end_of_frame* aura pour but de remettre à '0' l'offset de la RAM (priorité sur le signal *end_of_line*), de propager la valeur complexe du résultat, avec utilisation du bon retard, et de remettre à '0' le second accumulateur.

Pour simplifier la description, nous n'avons parlé que d'un seul traitement, mais dans l'implémentation, le traitement des parties réelles et imaginaires sont faites en parallèle de manière strictement identique.

Cette implémentation nécessite donc deux blocs DSP48s et deux RAMs dual-ports.

Cette première implémentation est sans doute la plus économique mais présente un nombre limité de traitement d'images par seconde, à cause des caractéristiques de la caméra. En effet, à la fin de chaque ligne, un temps de pause de 100 ns est imposé. Pour nos tests, nous avons configuré la caméra pour produire des images de 12×1024 avec un temps de pause de 10 μs, ceci afin de moyenner sur 12 estimations de phase et d'améliorer le rapport signal à bruit. Nous constatons que le temps de propagation des pixels d'une ligne est inférieur au temps de pause : une ligne passe en $\frac{12}{2} \times *12.5\ ns + 100ns = 175\ ns$ avec $\frac{3}{5}$ du temps induit par le temps de pause et $\frac{12}{2}$ le fait que pour une ligne de douze pixels de large, seulement 6 transmissions sont nécessaires. Cette implémentation permet de traiter 5285 images/seconde de 12×1024, soit un débit de 61.9 Mo/seconde.

4.5.1.2 Seconde implémentation

La première implémentation est certes intéressante, mais les temps de pauses induits par un grand ensemble de lignes n'est pas satisfaisant. Nous avons donc pris le parti de concevoir une seconde implémentation, cette fois-ci en ne pivotant pas la caméra.

Le principe de cette solution (figure 4.24), ne diffère de la première implémentation qu'en trois points : chaque pixel d'une ligne doit être multiplié par un coefficient particulier. La caméra fournit $pixel_n$ et $pixel_{n+1}$, ainsi faut-il pouvoir au même instant disposer des deux coefficients $coeff_n$ et $coeff_{n+1}$. Deux solutions sont possibles : soit enregistrer les coefficients pairs dans une RAM et les coefficients impairs dans une seconde, mais la consommation de RAM serait multipliée par deux ; soit modifier la configuration des RAMs afin qu'elles répondent aux besoins. Dans la première implémentation, les RAM étaient configurées en mode double port, avec pour chaque port la même dimension de bus (16 bits). Dans cette nouvelle version, la RAM est configurée pour que le port d'écriture soit toujours en 16 bits et ainsi permette de ne pas modifier l'interface avec le processeur, mais le port en lecture est configuré en 32 bits, soit la taille de deux coefficients successifs. Le bus de sortie est ensuite scindé en deux pour alimenter le traitement de $pixel_n$ et $pixel_{n+1}$. Comme il est nécessaire d'obtenir les coefficients par colonne et non plus par ligne, l'offset est incrémenté à chaque nouveau pixel et non plus à la fin de la chaque ligne. Finalement, les traitements sont dupliqués pour traiter en parallèle le pixel N et le pixel $N + 1$.

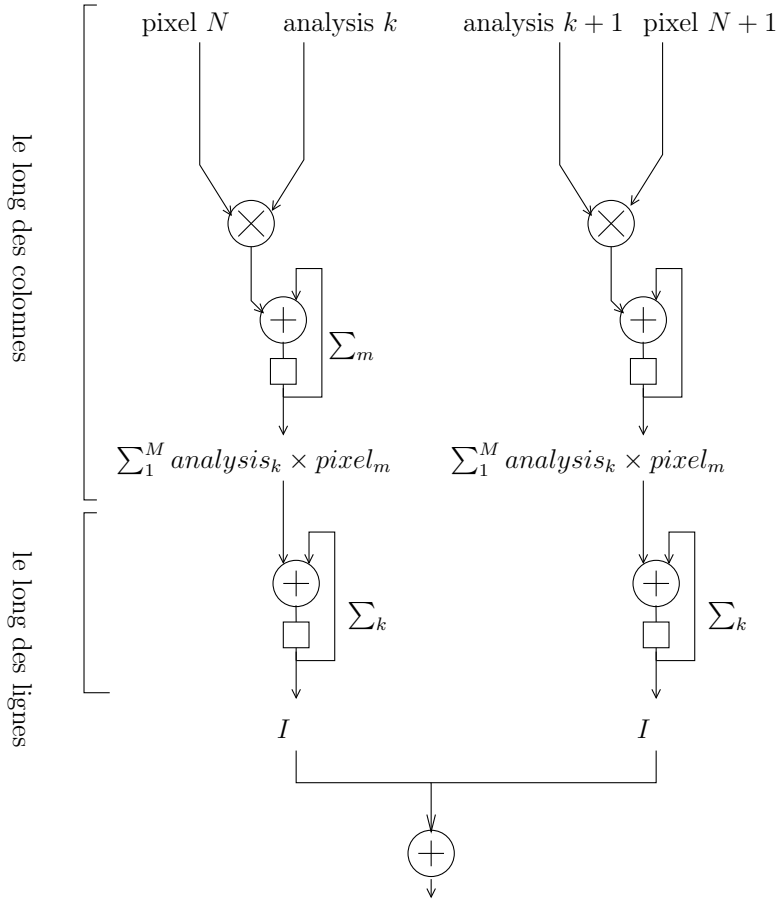


FIGURE 4.24 – Seconde implémentation : contrairement à la première version, le pré-additionneur n'est pas utilisé, le même traitement est réalisé pour le pixel N et $N+1$, et une dernière addition est opérée pour la fusion des deux résultats partiels. Les signaux de contrôle ne sont pas représentés sur cette figure car identiques à ceux de la figure 4.20.

Cette implémentation nécessite quatre blocs DSP48s, et deux RAMs dual-ports, le reste des caractéristiques, présentées dans la description XML figure 4.21, restent identiques.

Sur les images, de la même dimension que précédemment, le temps de traitement devient $(((\frac{1024}{2} \times 12.5\ ns) + 100) \times 12) + 10000\ ns = 88000\ ns$ soit 11364 images/seconde, incluant les 10 μs de temps d'exposition qui n'occupent que 15% du temps de traitement.

Nous avons pu également tester avec une image de dimension 1024×4, permettant ainsi d'obtenir un débit de 27778 images/seconde, soit 108.5 Mo/seconde, au détriment du rapport signal à bruit d'un facteur $\sqrt{3}$.

4.5.2 Résultat des traitements

Afin de pouvoir identifier les différents modes harmoniques du diapason, nous avons réalisé une seconde application dans laquelle la sortie sur des convertisseurs numérique-analogiques a été

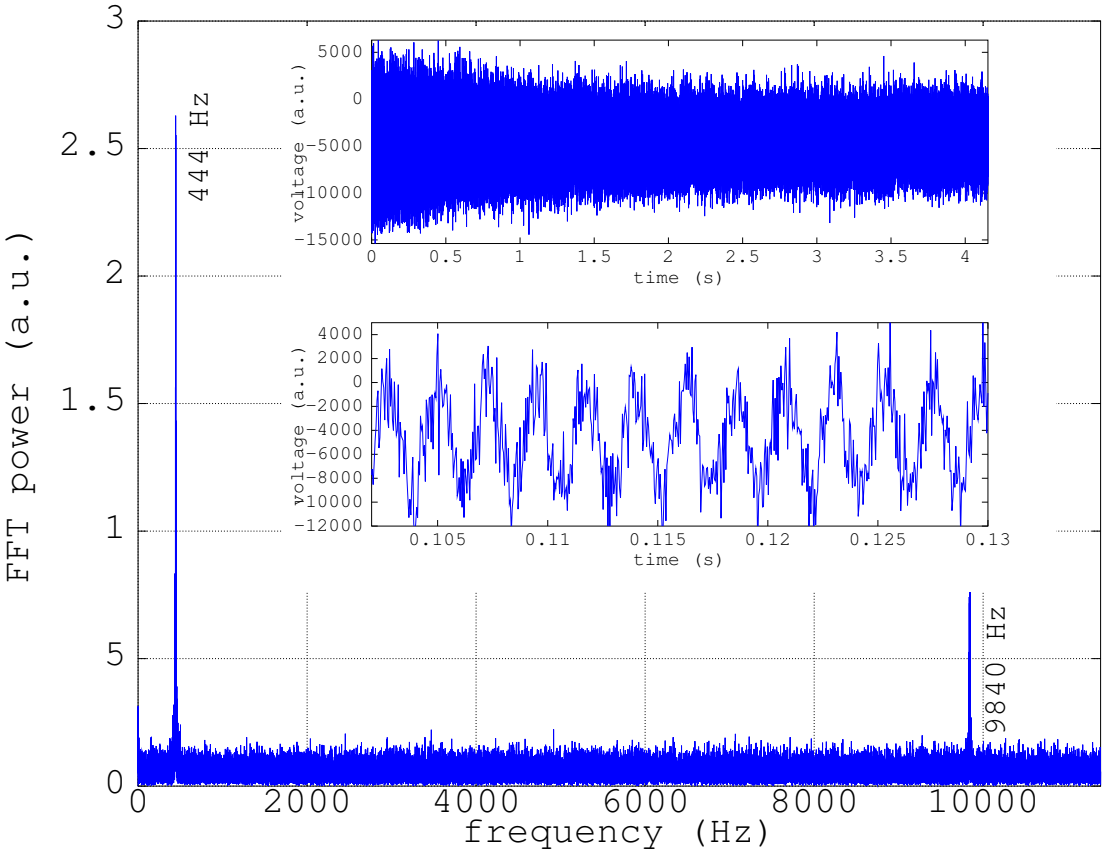


FIGURE 4.25 – Résonance d'un diapason par méthode optique jusqu'à 12 kHz. On observe deux modes : le premier à 440 Hz et le second à 9840 Hz.

remplacée par un stockage dans une RAM externe pour un transfert de données plus important et un post-traitement sur ordinateur. Nous avons ainsi pu stocker plusieurs minutes d'acquisition. Pour cette démonstration, l'excitation en régime permanent a été remplacée par une excitation impulsionnelle Les résultats sont présentés sur la figure 4.25, issue de la tranformée de Fourier (bas) de la réponse impulsionnelle acquise en fonction du temps (haut).

4.5.3 Analyse par CoGen

Afin de valider, avec ce traitement, les analyses faites par CoGen, nous avons, dans le cas de la seconde solution d'orientation, ajouté une troisième variante : basée sur la seconde, elle réalise le traitement des deux pixels séquentiellement.Cette nouvelle implémentation présente l'avantage d'avoir la même consommation de ressources que la première version. Du fait de l'utilisation d'un même multiplieur pendant deux cycles consécutifs, son acceptance est divisée par 2 devenant ainsi [10]($IMG_HEIGHT * IMG_WIDTH$).

Dans le cas de la caméra Photonfocus MVD1024E160 cadencée à 80 MHz et avec un FPGA à 100 MHz, seule les deux premières solutions sont acceptables, la troisième ne permet pas d'avoir un

débit suffisant en réception (maximum 50 MHz). Toutefois, en prenant un autre modèle de caméra (MVD1024E80) qui présente les mêmes caractéristiques hormis la *pixel clock* qui est de 40 MHz, la dernière implémentation devient également compatible.

Il serait possible de considérer d'autres solutions, pour s'adapter à une gamme plus importante de FPGAs par l'utilisation de multiplieurs simples en lieu et place des DSP48, en décrivant un accumulateur pour remplacer celui disponible en interne. Il serait également possible de placer une machine d'état pour faire usage d'un nombre plus faible de multiplieurs, mais aux dépends de la bande passante maximale du bloc.

4.6 Conclusion

Dans ce chapitre nous avons présenté un ensemble de cas de figure d'implémentations de blocs de traitements vidéos, de sources ou de transferts. Ces développements tiennent compte, d'une part des caractéristiques qu'un bloc doit respecter pour être considéré compatible avec CoGen, et d'autre part des caractéristiques du matériel afin de proposer des solutions performantes, car tirant profit de certaines ressources d'un FPGA particulier, ou plus génériques permettant de s'adapter à la majorité des cibles.

Nous avons constaté que le jeu de signaux et de bus permet de simplifier significativement la logique des développements d'implémentations. Dans le cas de la détection de contours ou de celui du diapason, la plupart des traitements sont uniquement basés sur les signaux de contrôle, avec introduction ou pas de retard. Dans certains cas la gestion des adresses peut ne pas avoir besoin de logiques particulières, le signal de présence d'un pixel permettant l'incrémentation et le signal de fin de ligne ou d'images la réinitialisation.

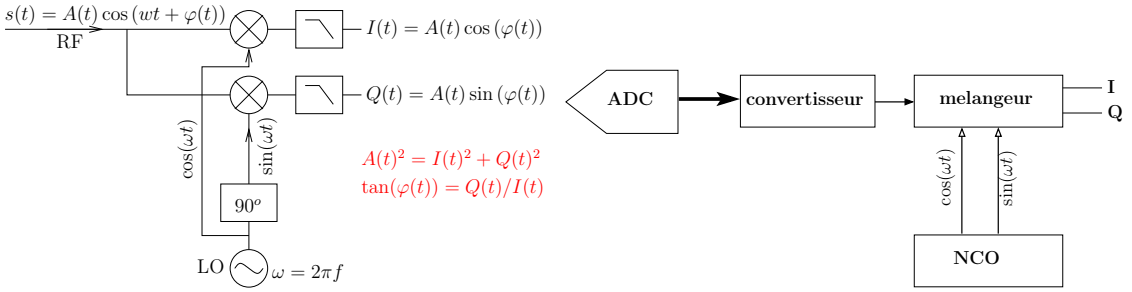
Chapitre 5

Traitement du signal 1D : application aux traitements de signaux radio-fréquence

Un flux d'image est un candidat naturel à la dualité FPGA/CPU pour combiner traitements temps-réel (FPGA) et algorithmes complexes (CPU). Un autre flux unidimensionnel de données présente des bandes passantes et des complexités similaires : les signaux radio-fréquences (RF) à vocation d'être traités par logiciel, ou SDR (Software Defined Radio). Les algorithmes de ce type de traitement comprennent typiquement une première étape simple (mélange par une transposition de fréquence, filtre et décimation) adaptée au FPGA, et une étape complexe (démodulation, PLL, détection d'enveloppe ou de phase) plus simple à implémenter sur CPU. Ce type d'algorithmes exploite pleinement la complémentarité CPU/FPGA. Nous allons donc nous efforcer de porter la chaîne de traitement issue de CoGen à une source, le convertisseur analogique-numérique rapide. Ce flux de données pré-traité par le FPGA est transféré au CPU qui exécute un système d'exploitation (GNU/Linux) qui se charge de transférer les données de l'espace noyau à l'espace utilisateur pour être exploité par un environnement libre de traitement du signal : GNURadio. La hiérarchie des niveaux d'abstraction couvre donc le FPGA, le noyau Linux, l'espace utilisateur Linux et finalement GNURadio.

Sans être, comme dans le cas de la vidéo, un flux matriciel, les flux radio-fréquences sont de type vectoriel ou complexe définis par un terme en phase (flux I pour *in phase*) et un terme en quadrature (flux Q). Les traitements qui vont nous intéresser nécessitent donc un convertisseur analogique-numérique avec deux voies pour obtenir le flux I/Q démodulé analogiquement (figure 5.1(a)) ou l'utilisation d'un mélangeur dans le FPGA (figure 5.1(b)).

Le flux de données radio-fréquence ainsi que les algorithmes liés sont le sujet de ce chapitre qui va présenter :



(a) Schéma de la démodulation d'un signal RF par un oscillateur local (LO). Le signal RF est mélangé au signal LO pour produire la composante I et avec le signal décalé de 90° pour obtenir la composante Q.

(b) Schéma de démodulation d'un signal RF par méthode numérique. Le bloc NCO (*Numerically Controlled Oscillator*) fournit à la fois le signal LO direct et le même signal déphasé remplaçant l'oscillateur local et le déphaseur. Le bloc mélangeur, qui prend en entrée le signal RF échantillonné et les deux signaux de référence, génère à la fois les signaux I et Q.

FIGURE 5.1 – Deux solutions pour la démodulation d'un signal RF : 5.1(a) en analogique, 5.1(b) en numérique.

- l'implémentation et l'utilisation du convertisseur analogique-numérique (*ADC*) *Hittite* **HMCAD1511** capable d'échantillonner jusqu'à une fréquence de 1 Géchantillons/s avec une résolution de 8 bits par échantillon. La mise en œuvre de ce composant qui s'apparente à la source d'images Photonfocus vue dans le chapitre précédent, va faire l'objet d'une démonstration qui reprend la caractérisation des modes propres d'un diapason mais par méthode radar ;
- le remplacement de l'ensemble des composants analogiques, en amont du FPGA, par un radio-modem (modulateur-démodulateur à ondes radio) capable de réaliser ces pré-traitements d'une manière plus condensée. Ces travaux ont, également, servis à évaluer la complexité de création d'une source pour une application de radio définie par logiciel (*SDR*) nommée *GNU-Radio*. Cette seconde démonstration consiste en l'acquisition d'un flux modulé en fréquence (FM) pour un traitement logiciel (démodulation du signal et affichage ou transfert vers une carte son) ;
- la dernière partie de ce chapitre concerne la création de blocs, tels qu'un mélangeur, NCO et filtre passe bas, afin de remplacer, dans le FPGA, l'ensemble des blocs analogiques des premiers travaux et pouvoir, à partir d'un convertisseur analogique-numérique, disposer des mêmes fonctions de base que celles fournies par un radio-modem. Ce travail a pour but de réaliser des chaînes de traitements radio-fréquences reconfigurables selon le besoin de l'application. L'objectif est, comme en traitement d'image, de réduire le débit des données pour le rendre accessible au CPU.

5.1 Convertisseur analogique-numérique HMCAD1511

Le HMCAD1511 est un ADC 8 bits rapide pouvant échantillonner 1, 2 ou 4 voies avec des fréquences maximales de 1 GHz, 500 MHz et 250 MHz respectivement. Les données sont propagées au travers de 8 paires différentielles LVDS fournissant selon le mode 8, 4 ou 2 données en parallèle par voie. Il fournit également une horloge qui est 8, 4 ou 2 fois plus lente que l'horloge qui le cadence, fournissant donc au maximum une fréquence de 125 MHz.

Le code du bloc initial dédié est basé sur un code fourni par Hittite, adapté aux caractéristiques de CoGen.

5.1.1 Implémentation

Comme pour la caméra Photonfocus (section 4.2), le *HMCAD1511* transfère les données en LVDS. Les signaux, présentés sur la figure 5.2, sont :

- DxyA/B : 8 paires différentielles, chacune portant les 8bits d'une donnée ;
- LCLK : signal d'horloge dont la fréquence est la même que celle de l'échantillonnage. Elle est utilisée pour la désérialisation des données ;
- FCLK : une horloge lente dont la fréquence est selon le mode 2, 4 ou 8 fois plus lente que la fréquence d'échantillonnage. Cette horloge est utilisée pour la synchronisation puisqu'à chaque front montant de cette horloge, un ensemble de données est disponible pour la propagation ;

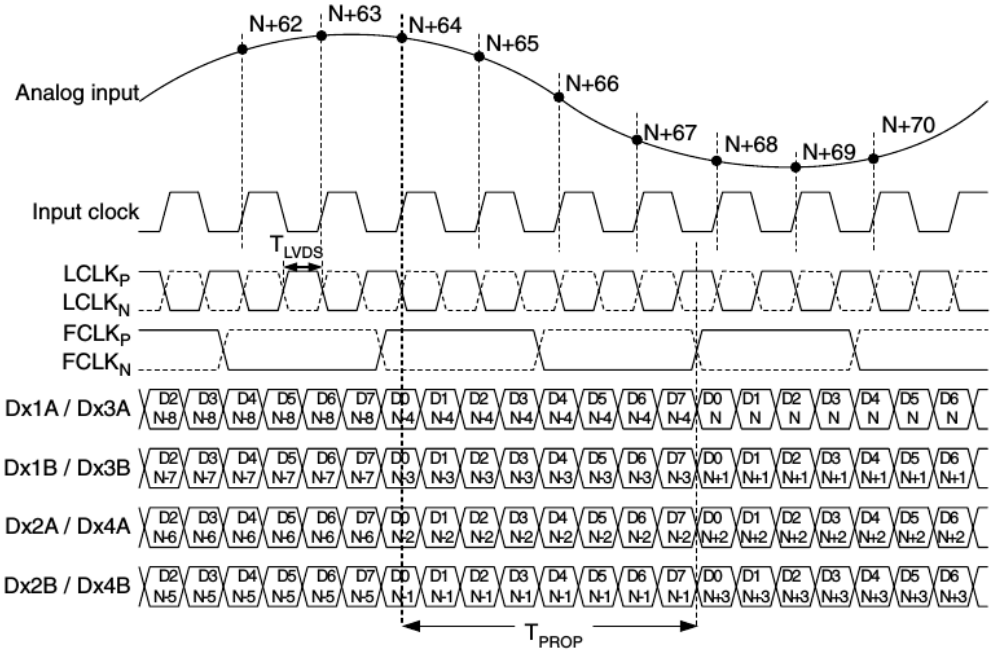


FIGURE 5.2 – Chronogramme des signaux entre le convertisseur-analogique numérique et le FPGA dans le cas du mode 2 voies. Les paires Dx1A, Dx1B, Dx2A, Dx2B fournissent les données pour la voie 1 et les paires Dx3A, Dx3B, Dx4A, Dx4B pour la voie 2. Source : documentation hittite.

Contrairement à la caméra Photonfocus, les données sont en **DDR** (Dual data rate), ce qui signifie qu'à chaque front d'horloge (montant et descendant) un nouveau bit est disponible sur chaque paire.

Le bloc est sensiblement identique à celui de la caméra, à deux exceptions près :

1. la partie désérialisation a dû être modifiée pour tenir compte de l'utilisation des deux fronts de l'horloge LCLK et un second étage a été réalisé basé sur le signal FCLK pour la re-synchronisation des données ;
2. La FIFO, pour réaliser le changement de domaine d'horloge, peut être contournée pour fournir directement les données cadencées par l'horloge du convertisseur, et le bloc fournit une interface de type horloge utilisée par les blocs de traitements si la FIFO n'est pas utilisée.

5.1.2 Contribution à CoGen

Selon le mode et la fréquence d'échantillonnage, les données reçues n'ont pas la même signification et la fréquence de propagation peut varier, ainsi plusieurs points ont été mis en avant et ont permis d'ajouter des caractéristiques à CoGen :

- l'utilisateur doit fournir le mode de fonctionnement et la fréquence d'échantillonnage au moment de la construction de la chaîne. Les paramètres de configuration de l'ADC deviennent constants une fois le code généré, ceci afin de garantir que les traitements restent justes (affectation des lignes différentielles). L'ensemble des analyses repose sur la fréquence fournie par l'utilisateur, mais en fonctionnement il est de la responsabilité de l'utilisateur de respecter cette valeur, CoGen n'ayant aucun pouvoir sur la source de cadencement ;
- dans le cas de la caméra, l'horloge fournie avait une fréquence inférieure à celle du FPGA. Dans le cas de la *SP_ VISION* par exemple, si le convertisseur est cadencé au maximum de sa capacité, la fréquence dépasse celle de l'horloge du cœur. Dans ce type de situation, un changement de domaine d'horloge n'est pas envisageable sous peine de pertes des données. Deux solutions étaient envisageables :
  - imposer une décimation du flux. Cette solution présente également une perte de données, voire des effets de bords, mais sachant que ce comportement est connu, elle peut être acceptable. Toutefois cette solution est contraire au principe d'un bloc initial qui ne doit pas modifier le flux ;
  - activer ou non, selon le cas, le changement de domaines d'horloge. Avec cette solution il est possible de garantir que les données ne seront pas perdues et le changement peut-être réalisé en aval si le flux est décimé. Cette seconde solution implique que les blocs initiaux doivent fournir une interface d'horloge maître, et que les blocs de traitements doivent, si nécessaire, disposer d'une même interface mais de type esclave. L'aspect optionnel, dans le cas des blocs de traitements, se justifie par le fait que, par défaut, tous les blocs disposent déjà d'une entrée horloge et qu'il est donc possible de connecter l'horloge du convertisseur

en lieu et place de celle du FPGA. Les seuls blocs ayant réellement besoin de deux interfaces sont ceux qui ont une interface de communication avec le processeur. La communication avec le CPU devant être synchrone, elle doit donc utiliser l'horloge du FPGA, fournie par le CPU et utilisée pour toutes les transactions.

La seconde solution a été retenue pour la simple raison qu'elle était respectueuse des règles de CoGen et n'imposait pas, à l'utilisateur, une décimation qui pourrait impliquer des problèmes dans les traitements et les résultats.

Compte tenu des trois modes de fonctionnements, trois versions de ce bloc initial sont disponibles (la description en mode 2 voies est présentée sur la figure 5.3), toutefois les trois pointent sur le même bloc implémentation puisque le comportement nécessite juste un paramétrage adapté.

5.1.3 Validation basique

Afin de valider le bon fonctionnement du convertisseur et de la désérialisation, une première application a été réalisée : celle-ci ne contient que deux éléments

- le bloc initial correspondant au convertisseur ;
- un bloc de stockage/transmission dont le rôle est de mémoriser dans une BRAM, sur ordre du processeur un ensemble de données contiguës, puis lorsque que cette première étape est finie, d'en avertir le processeur pour qu'il réalise le transfert pour un post-traitement.

Les résultats obtenus sont représentés sur la figure 5.4 avec à gauche le signal temporel et à droite le résultat fréquentiel, transformée de Fourier du jeu de données. Sachant que le signal est de 10 MHz, qu'il est échantillonné à 500 MHz, nous voyons bien sur la vignette de la figure 5.4 une période de $\frac{500}{10} = 50$ points par période, ce qui prouve que nous ne perdons pas de données.

5.2 Mise en œuvre simple : diapason par RADAR

Ayant validé les principes de CoGen dans le cas de flux à une dimension et le fonctionnement du convertisseur (section 5.1), nous avons souhaité réaliser une application plus ambitieuse. Celle-ci reprend le but de l'application présentée en 4.5, à savoir observer le déplacement d'un bras de diapason musical, mais en lieu et place de la mire et d'une caméra, nous avons utilisé un capteur à onde élastique de surface dont le retard de l'écho est fonction de la contrainte, soudé sur une des branches du diapason (présenté en vignette de la figure 5.5).

5.2.1 Principe de fonctionnement

L'interrogation du capteur est réalisée sans fil par une liaison radio-fréquence, au travers d'une antenne connectée à celui-ci. Le mécanisme d'interrogation dispose d'une seconde antenne. L'interrogation d'un tel composant se fait en deux temps :

```
<?xml version="1.0" encoding="utf-8"?>
<initial_block name="hmcad1511 dual mode" >
  ...
  <timings>
        <timing dir="out" freq="FREQ_SAMP/4"
      pattern="1(*)"  />
  </timings>
  <user_config>
    <param name="FREQ_SAMP" default="500" />
  </user_config>
  <resources>
    <resource type="PLL_ADV" amount="1" />
    <resource type="SERDES" amount="18" />
    <resource type="IODELAY" amount="18" />
  </resources>
  <bloc_interfaces>
    <interfaces dir="out" name="voieA" type="real">
      <interface name="data1_out" />
      <interface name="data2_out" />
      <interface name="data3_out" />
      <interface name="data4_out" />
    </interfaces>
    <interfaces dir="out" name="voieB" type="real">
      <interface name="data5_out" />
      <interface name="data6_out" />
      <interface name="data7_out" />
      <interface name="data8_out" />
    </interfaces>
  </bloc_interfaces>
  ...
</initial_block>
```

FIGURE 5.3 – Fichier de configuration du bloc HMCAD1511 pour une configuration deux voies. L'utilisateur peut, lors de la construction de la chaîne fournir la fréquence d'échantillonnage (balise *param*), la fréquence fournie pour la sortance est calculée sur la base de l'information fournie par l'utilisateur et est divisée par 4 en accord avec les caractéristiques de ce mode. Deux balises *interfaces* contiennent chacune quatre balises *interface* correspondant à 4 des 8 sorties de l'ADC fournies, là encore, en respectant les caractéristiques du convertisseur.

1. dans un premier temps, le capteur est activé par l'envoi d'un signal à 2.4 GHz. La durée de cette charge est d'environ 50 ns ;
2. dans un second temps, après l'arrêt de l'émission, le capteur va se décharger. Le signal émis contient plusieurs échos dont le retard est directement lié à la contrainte appliquée.

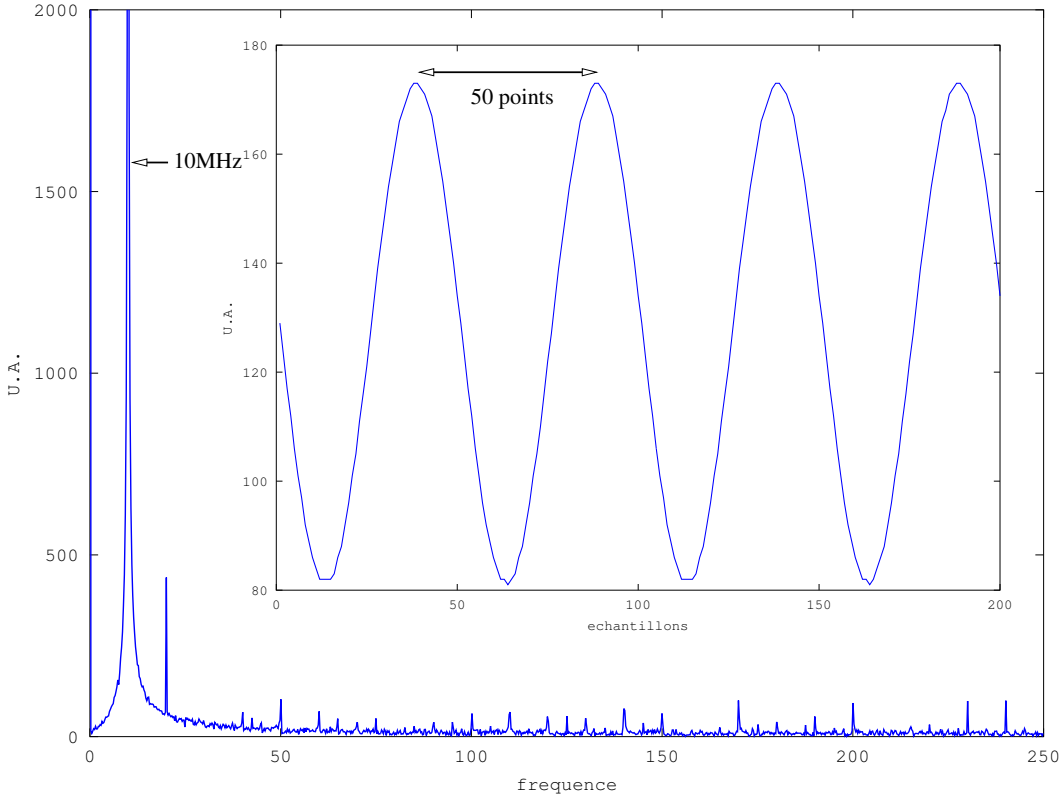


FIGURE 5.4 – Résultat de l'acquisition d'un signal sinusoïdal d'une fréquence de 10 MHz avec une fréquence d'échantillonnage de 500 MHz. Bas : représentation fréquentielle du signal échantillonné. En vignette, représentation temporelle.

La réponse du capteur est démodulée pour obtenir un flux I/Q de 100 MHz de bande passante ; ces deux informations sont ensuite numérisées par le *HMCAD1511* à une fréquence de 500 MHz.

Cette application (figure 5.5) est principalement analogique. La partie numérique effectue les traitements suivants (figure 5.6) :

- la gestion du switch *ZASWA-2-50DR* utilisé pour basculer entre le mode émission et le mode réception du RADAR impulsionnel avec une durée de transition de 50 ns ;
- la conversion du flux de données ;
- la propagation d'un sous ensemble de données. Le flux continu de l'ADC n'est pas pertinent dans cette application : seul un certain nombre de données contigües doit être conservé. Les données importantes débutent après le passage du switch en mode réception et finissent au bout d'un temps correspondant à la durée de décharge du capteur ;
- le stockage de la totalité de ses données ou de certaines particulières ;
- finalement, le transfert vers le processeur pour un stockage sur disque ou un affichage.

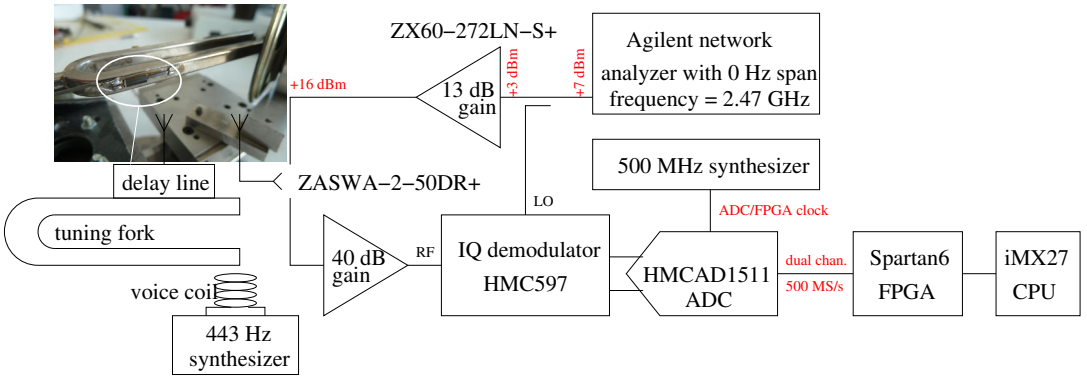


FIGURE 5.5 – Schéma complet de l'application d'interrogation de capteur.

5.2.2 Implémentation

Compte tenu de l'aspect rafale des données reçues, qui doivent être synchrones avec le début de la décharge de la ligne à retard et connaître la fin de celle-ci, un nouveau type d'interface a été ajouté à CoGen. Cette interface, qui est une variante de l'interface de type complexe (deux signaux de données ainsi qu'un signal *enable*), ajoute un signal de contrôle de fin de rafale.

La partie numérique de la chaîne, représentée sur la figure 5.6, contient 5 blocs :

- le bloc initial de conversion du flux issu du convertisseur analogique-numérique ;
- un bloc qui gère le switch pour le mettre en émission pendant une durée de 50 ns, puis le mettre en réception et transférer les données du convertisseur. Ce bloc est configurable, lors de la construction de la chaîne, pour fixer la durée entre deux phases d'acquisitions. À la fin d'une rafale, en même temps que la propagation de la dernière donnée, ce bloc met à l'état haut le signal de fin de transfert ;
- un bloc qui stocke directement les données qu'il reçoit. Ce bloc est une variante utilisée dans le cas de l'interrogation par méthode optique (caméra) à la différence qu'il stocke des données de type complexes et qu'il se base sur la fin de transfert pour garantir le stockage/transfert d'une rafale dans sa globalité et ainsi éviter que les données reçues par le processeur ne correspondent à la fin d'une rafale puis au début de la suivante ;
- un bloc qui compte le nombre d'échantillons pour n'en conserver qu'un sur l'ensemble et le transférer pour stockage. Ce mode est représentatif du cas où l'on ne s'intéresse pas à l'ensemble de la réponse du capteur mais uniquement aux caractéristiques d'un des échos représentatifs du champ de contrainte observé par la jauge. Sa sortie est une interface de type complexe classique ne comportant que deux bus (un pour le I et l'autre pour le Q) ainsi qu'un *enable*. L'absence du signal de fin s'explique par la nature même du flux. En effet contrairement au bloc précédent, ce n'est plus l'ensemble de la décharge qui est à étudier mais bien les évolutions successives d'une résonance particulière ;
- finalement un bloc terminal dont le principe est équivalent à celui de la méthode optique

mais pour le stockage d'un flux I/Q sans notion de début ou de fin de transfert.

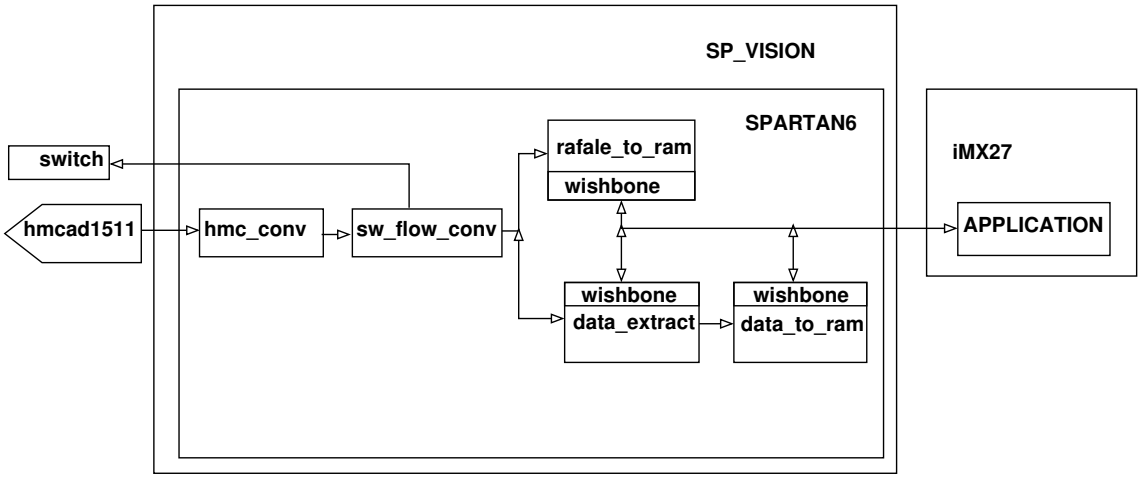


FIGURE 5.6 – Schéma de l'application FPGA dans sa globalité, comportant un bloc initial (HMCAD1511), un bloc dont le rôle est de piloter le switch pour charger la ligne à retard puis de transférer au(x) suivant(s) l'ensemble des points correspondant à la décharge du capteur, un bloc terminal pour le stockage de la rafale, un bloc dédié à l'extraction d'un point particulier et finalement un second bloc terminal utilisé pour le stockage d'un nombre configuré de points correspondant à un point particulier de la décharge.

Grâce aux deux blocs terminaux présents, cette application présente deux modes de fonctionnement :

1. un premier mode qui stocke l'ensemble des données de la décharge du capteur permettant ainsi de sélectionner une résonance particulière (figure 5.7 partie gauche) ;
2. un second mode qui va prendre un échantillon particulier après le passage du switch en réception : le délai est sélectionné à partir de l'information fournie depuis le processeur à partir de la représentation du premier mode (figure 5.7 partie droite). Ce second mode permet de voir précisément l'évolution temporelle de la contrainte appliquée au capteur.

5.2.3 Résultats

Après stockage des résultats dans le mode d'acquisition d'un point par décharge (second mode) et post-traitement, nous obtenons la figure 5.8, représentative de la réponse impulsionnelle du diapason dans les domaines temporel (en vignettes) et spectral.

En comparaison de la méthode optique, nous obtenons les deux résonances (à 440 Hz et 10143 Hz), une à 40714 Hz absente dans le cas de la méthode optique du fait du débit limité de la caméra mais nous pouvons également voir une résonance à 5465 Hz, absente de la courbe 4.25 : nous n'avons pu trouver de réponse à cette différence mais faisons l'hypothèse d'une déplacement du bras du diapason colinéaire aux franges (et donc indétectable par inter-corrélation).

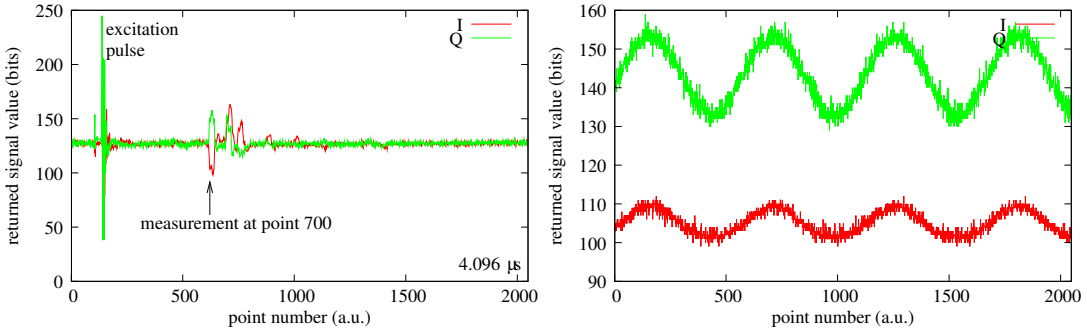


FIGURE 5.7 – Courbes observées : à gauche, représentation temporelle de la décharge du capteur avec ses 8 résonances. À droite, affichage d'une suite d'échantillons correspondant au point 700 ($\frac{1}{500MHz} * 700 = 1.400\mu s$) après la charge du capteur.

Cette application nous permet d'obtenir une interrogation à 250 kHz et donc une bande passante de la mesure de la contrainte de 125 kHz, uniquement limité par le temps de décharge du capteur ($4\mu s$).

5.3 Acquisition radio-fréquence et traitement par GNURadio

La première application a permis de valider les principes de développement liés à CoGen sur un flux issus d'un ADC. Mais cette démonstration est massivement orientée vers l'analogique avec un certain encombrement et manque de flexibilité, dû aux éléments en amont du convertisseur. Pour simplifier l'ensemble, dans le cas du traitement radio-fréquence nous nous sommes intéressés à un radio-modem de la marque *Semtech* : le SX1255.

L'intégration de ce composant a été également l'occasion d'évaluer la complexité et la possibilité d'interfacer une chaîne de traitements généré par CoGen avec un post-traitement réalisé par GNURadio.

5.3.1 Semtech SX1255

Ce composant, dont le fonctionnement interne est représenté sur la figure 5.9, embarque directement la démodulation du signal, le filtrage, ainsi que la conversion analogique-numérique, puis transfère les données au travers du protocole **i2s** (*Inter-IC Sound* ou *Integrated Interchip Sound*).

Il est *full-duplex* (possibilité d'émettre en même temps que de recevoir). Il est capable d'émettre/recevoir des fréquences comprises entre 400 et 500 MHz avec une fréquence d'échantillonnage maximum de 36 MHz. Dans le cas de la réception, tel que présenté sur la figure 5.9, le signal est d'abord démodulé à l'aide d'une PLL fractionnaire et d'un mélangeur, puis le signal est filtré avant d'être converti en format numérique, puis décimé et transféré vers le périphérique connecté. Le flux

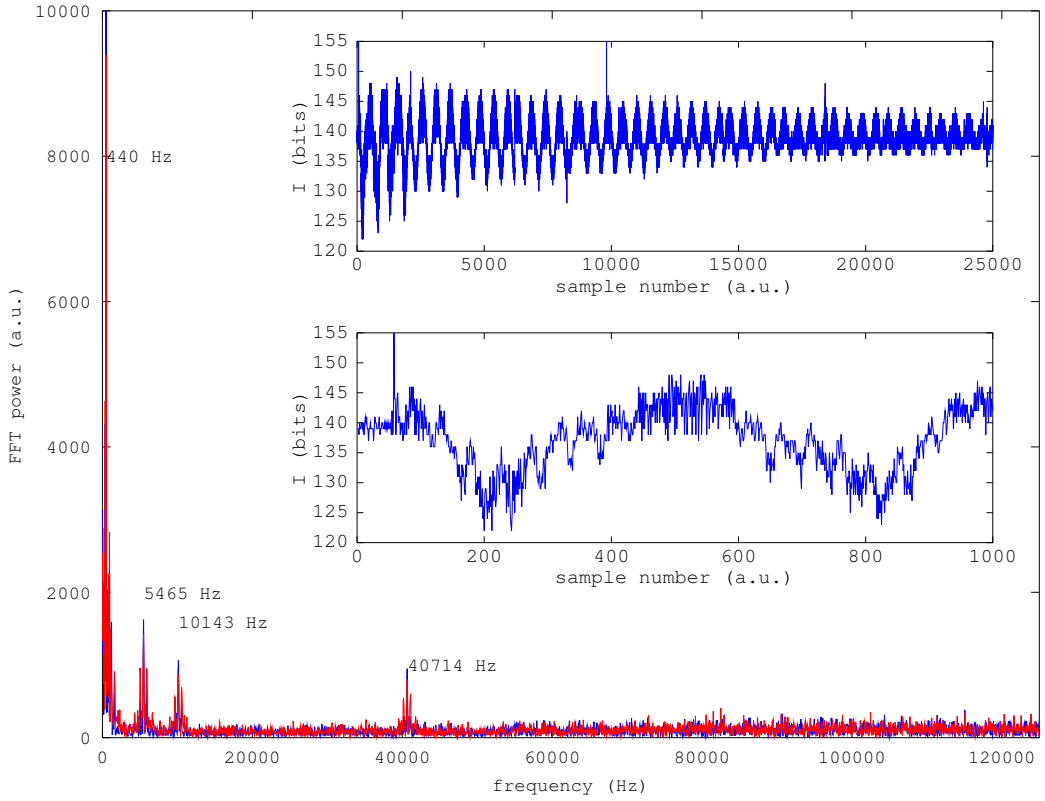


FIGURE 5.8 – Réponse impulsionnelle (vignettes) et représentation fréquentielle des modes propres du diapason, caractérisé par une jauge de contrainte soudée, par méthode RADAR. Quatre modes sont visibles : 440 Hz, 5465 Hz, 10143 Hz, 40714 Hz. Nous y retrouvons les résonances à 440 Hz et 10143 Hz, celle à 40714 Hz ne pouvant pas être observée en méthode optique du fait de la fréquence d'échantillonnage trop basse. Nous ignorons la raison de l'absence de celle à 5465 Hz dans la méthode optique.

fourni, au format *I2s* (figure 5.10), comporte directement les données I/Q en parallèle avec un débit maximum de 4 MHz pour des données codées sur 9bits. Sa configuration (fréquence de la porteuse pour l'émission et la réception, configuration de la fréquence de l'horloge et de la décimation) est réalisée en *SPI*.

Le schéma 5.11 présente le bloc correspondant, et la figure 5.12 présente la description du bloc. L'implémentation pour l'heure ne comporte que la partie réception.

5.3.2 GNURadio

GNURadio [8] propose à la fois un ensemble de bibliothèques de blocs de traitements du signal (démodulation FM ou AM, FFT, affichage dans le domaine temporel ou fréquentiel, sortie son), mais également une application graphique, nommée *gnuradio-companion*, pour la construction de la

8. http://www.gnuradio.org

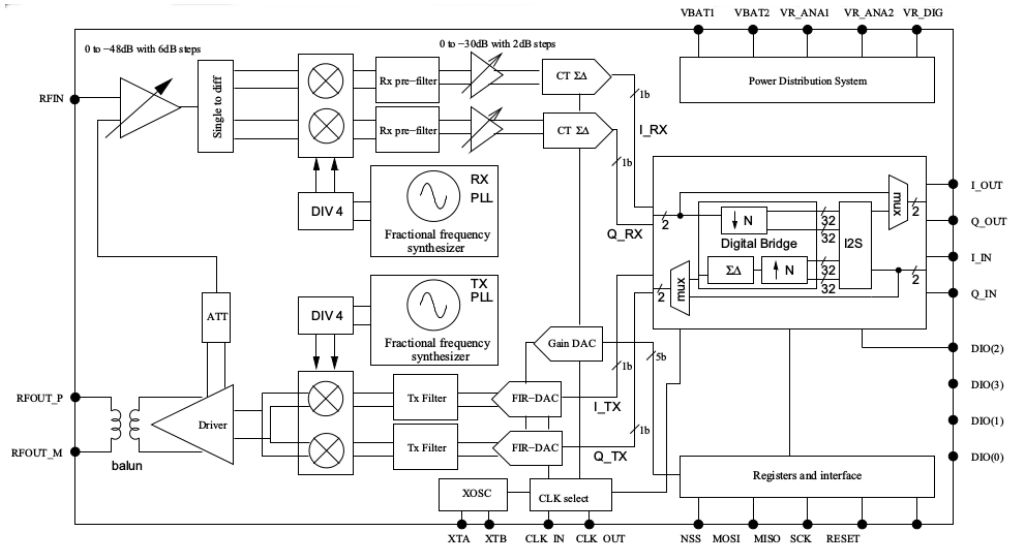


FIGURE 5.9 – Schéma interne du SX1255. Sur la partie gauche la modulation/démodulation de signaux radio-fréquences. Au centre, la conversion analogique-numérique pour la réception et la conversion numérique-analogique pour l'émission. À droite, la décimation et la conversion en un flux **i2s** pour la réception ou la conversion du flux **i2s** pour l'émission. Source : Semtech.

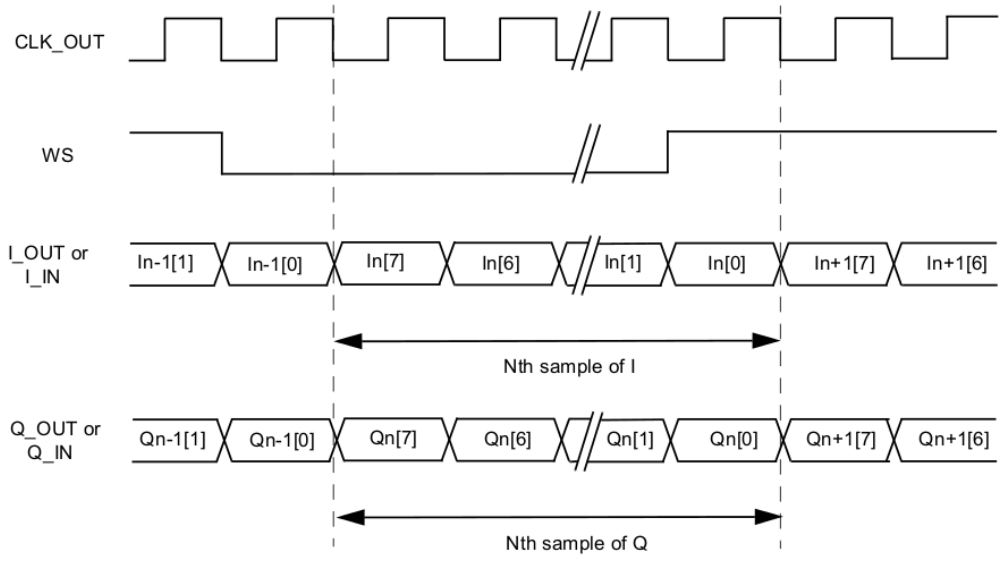


FIGURE 5.10 – Chronogramme du protocole série *i2s* utilisé pour la transfert de données. Un signal d'horloge (CLK_OUT) cadence la lecture/écriture sur le front montant, respectivement descendant. le signal *WS* est utilisé comme délimiteur entre deux données consécutives. Source : Semtech.

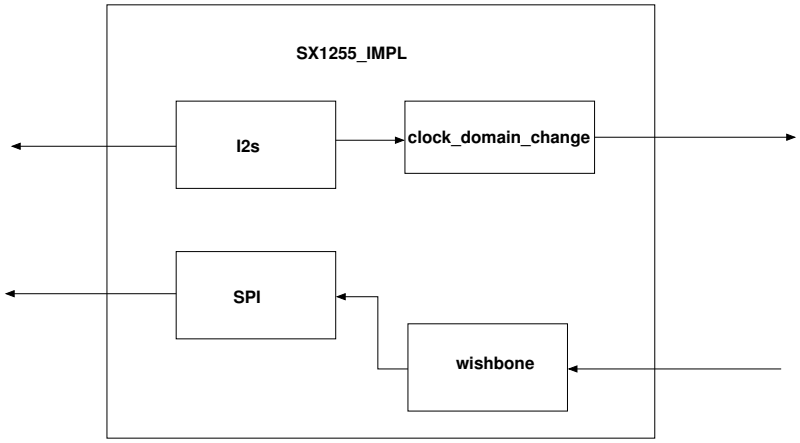


FIGURE 5.11 – Schéma du bloc d'implémentation du SX1255. Celui-ci comporte un sous-bloc de conversion *i2s* pour la réception des données, un sous-bloc qui réalise le changement de domaine d'horloge entre celle du radio-modem et l'horloge du FPGA, une implémentation du protocole *SPI* pour la configuration faite, au travers du bus wishbone, depuis le processeur.

```
<?xml version="1.0" encoding="utf-8"?>
<initial_block name="sx1255" >
  <user_config>
    <param name="FREQ_CLK" default="4" />
    <param name="DATA_OUT_SIZE" default="9" />
  </user_config>
  <timings>
        <timing dir="out" freq="FREQ_CLK" pattern="1(*)"  />
  </timings>
  <bloc_interfaces>
    <interfaces dir="out" name="data" type="complex">
      <interface name="data_out" size="DATA_OUT_SIZE" />
    </interfaces>
  </bloc_interfaces>
</initial_block>
```

FIGURE 5.12 – Fichier de description du bloc SX1255. Le bloc est paramétrable pour la fréquence du bus *i2s* et la taille des données reçues. Du fait de la simplicité du protocole de transfert, ce bloc ne consomme pas de ressources matérielles particulières.

chaîne de traitements par assemblage de blocs (exemple d'application sur la figure 5.13). À l'origine principalement utilisé avec des émetteur-récepteurs USRP de *ETUS Research*[9], son utilisation s'est démocratisée avec la possibilité d'exploiter une clé USB pour la réception de la télévision numérique terrestre. C'est à cette occasion que nous nous sommes intéressésà ce logiciel et avons eu

9. http://home.ettus.com

la possibilité de réaliser des blocs de traitements (*ACARS*) complétant les fonctionnalités d'origine [48]. L'aspect open-source du logiciel encourage en effet à compléter les fonctionnalités fournies par ses propres sources ou bloc de traitements. Lors de ces travaux, la ressemblance dans les concepts de bases (connexion de blocs, vérification de la cohérence des interfaces) nous a motivé à évaluer la possibilité de pouvoir exploiter cet outil pour la partie logicielle des chaînes de traitements, en complémentarité avec CoGen.

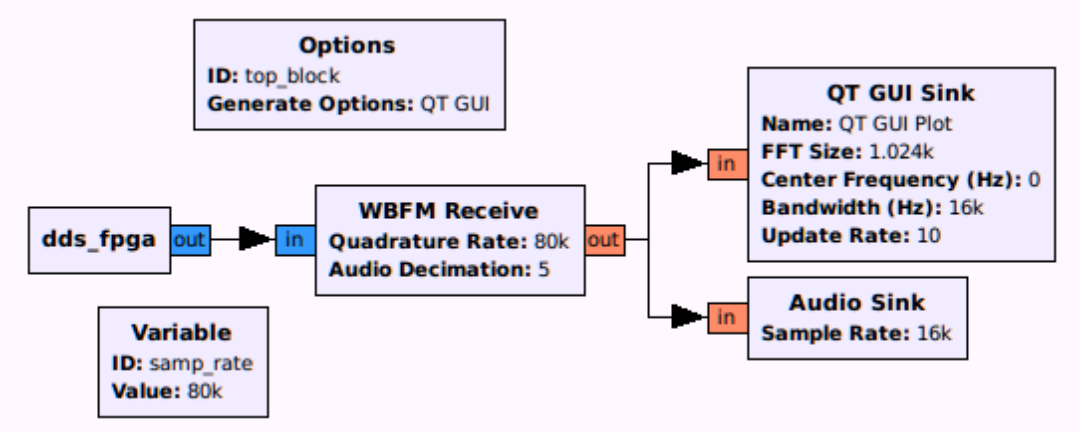


FIGURE 5.13 – Exemple de chaîne de traitements GNURadio. Une source (équivalent d'un bloc initial CoGen) fournit un flux de données de type complexe à un bloc de démodulation FM (équivalent d'un bloc de traitement). Ce dernier propage ses résultats à deux blocs de type sink (équivalent aux blocs terminaux), dont le premier est dédié à l'affichage et le second au transfert vers une carte son.

En effet, CoGen, ainsi que la plupart des logiciels de ce type, est conçu pour la génération de chaînes de traitements sur FPGA : de ce fait, l'ensemble des traitements réalisés sur le processeur, tel que l'affichage, le stockage ou le transfert doit être réalisé manuellement. Le projet GNURadio propose, pour palier cette lacune, une solution intéressante permettant de réaliser ce genre de traitements avec une approche similaire à celle CoGen.

5.3.2.1 Source GNURadio

Le travail que nous avons réalisé du point de vue GNURadio a consisté en la création d'une source, capable de transmettre à une chaîne logicielle, produite au travers de *gnuradio-companion*, le flux de données issues du FPGA. Ce travail a été simplifié par la présence systématique d'un pilote au niveau du noyau réalisé dans la phase de *codesign* lors des créations de blocs terminaux dédiés au transfert d'un flux vers le processeur.

Son implémentation comporte deux fonctions principales :

1. un processus léger (*thread*) qui, en boucle, demande des données au pilote. Lorsque le pilote est en mesure de fournir les données, elles sont stockées dans un tampon circulaire. Ce type

de tampon est utilisé pour garantir au niveau de la chaîne de traitement que des données seront toujours disponibles et ainsi éviter une carence ;

2. une fonction particulière dont la présence est obligatoire pour que GNURadio puisse fournir/demander des données, selon le type du bloc, et dont le but est de transférer le nombre d'éléments demandé lors de l'appel de la fonction. Cette fonction va donc consommer au rythme imposé les données obtenues depuis le FPGA au travers du pilote.

5.3.3 Application de démonstration

L'application visée pour cette démonstration consiste à réceptionner un flux radio-fréquence modulé en fréquence (FM), produit par un émetteur FM *RADIOMETRIX TX2-433-160-5V* à 433.9 MHz connecté à la sortie d'une carte son. La partie FPGA aura la charge de réaliser la conversion du flux i2s, une décimation pour obtenir un flux compatible avec les capacités de transfert du processeur et stocker/transférer ces résultats (figure 5.14 partie gauche).

La partie logicielle devra fournir à la demande de GNURadio un ensemble de données qui sera passé dans un bloc de démodulation FM puis transféré vers la carte son de la plate-forme mais dont le résultat sera également affiché sur l'écran (figure 5.14 partie droite).

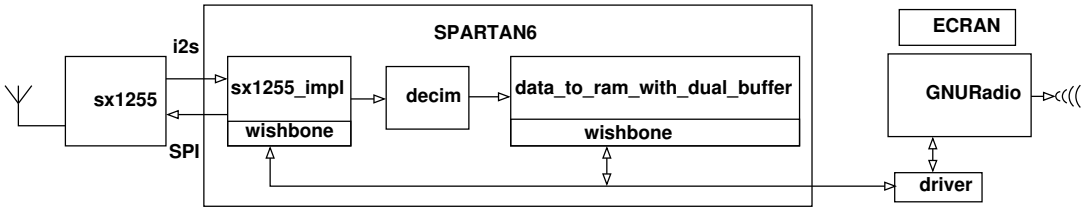


FIGURE 5.14 – Schéma global de l'application visée. À gauche, le traitement dans le FPGA (conversion du flux, décimation, stockage/transfert). À droite, partie logicielle.

Afin de réaliser la démodulation, il est impératif de garantir la continuité du flux, sans quoi le son serait saccadé voire inaudible. Cette application a donc pour objectif de démontrer qu'aucun échantillon n'est perdu au cours des transferts.

5.3.3.1 Principe du bloc terminal avec implémentation d'un double tampon

Le but de ce double tampon est de permettre un stockage en continu des données. À chaque fin de tampon, le bloc avertit le processeur pour la récupération des données. Pendant que celui-ci réalise cette action, le bloc est en mesure de continuer le stockage dans le second tampon, sans risque d'écraser les données avant leurs transferts.

Le principe de l'implémentation de ce bloc est présenté sur la figure 5.15. Son fonctionnement est le suivant :

- un compteur est incrémenté à chaque nouvelle donnée par le signal enable. Quand ce compteur atteint la valeur maximale, il est remis automatiquement à 0 ;
- dans le même temps la valeur du compteur est utilisée comme adresse pour le stockage dans la RAM, la donnée entrante est directement connectée sur ce bloc ainsi que le enable qui sert d'ordre d'écriture.
- l'adresse est également comparée à deux valeurs : l'une correspond à la moitié de la capacité de la mémoire, la seconde à sa capacité totale. Si l'une des comparaisons est évaluée comme vraie, le *OU* logique génère pendant un cycle d'horloge un état haut sur un signal de sortie connecté à une broche du processeur et permet de produire une interruption. Ces deux comparaisons sont également utilisées au niveau de l'esclave *wishbone* pour mettre à jour le statut du bloc, utilisé par le processeur pour connaître la partie du tampon à lire ;
- l'esclave *wishbone* reçoit les commandes du processeur, met à jour l'adresse de lecture et transmet la donnée correspondante.

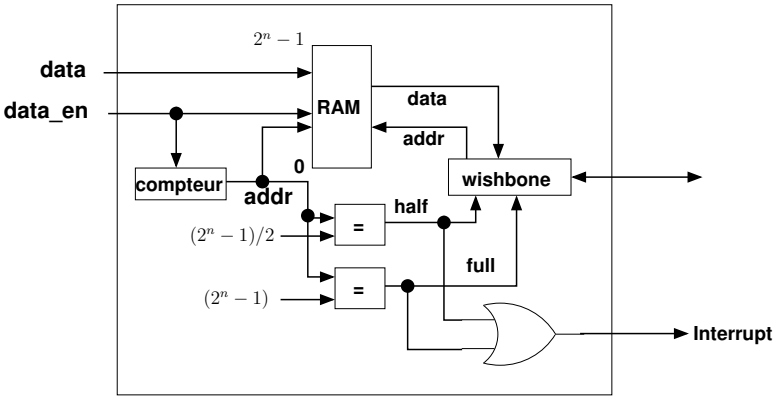


FIGURE 5.15 – Implémentation du bloc terminal pour le transfert des données, avec utilisation du principe du double tampon. Le compteur est utilisé pour incrémenter l'adresse de la RAM, il est également utilisé pour déterminer à quel moment une interruption doit être levée et mettre à jour le statut du bloc. Le bloc wishbone gère l'ensemble des transactions de lecture et de statut.

5.3.3.2 Pilote pour le transfert depuis le FPGA vers la source GNURadio

Ce type de pilote est assez classique. Il crée un pseudo fichier, utilisé par l'application en espace utilisateur afin de fournir les données à celle-ci. Compte tenu du besoin potentiel d'un flux continu sans perte de donnée, ce pilote ne se contente pas d'aller chercher des données à la volée. C'est le bloc terminal qui impose quand le bloc doit aller chercher les données avec un principe de double tampon.

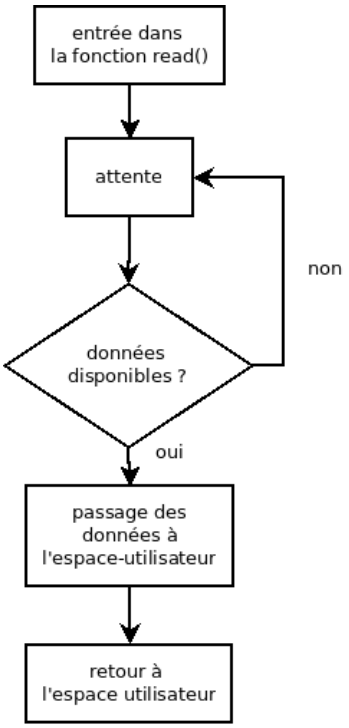
Ainsi le principe de fonctionnement, dont le diagramme est présenté sur la figure 5.16, est le suivant :

- une ligne dédiée est mise à l'état haut pour avertir le processeur de la présence d'un tampon

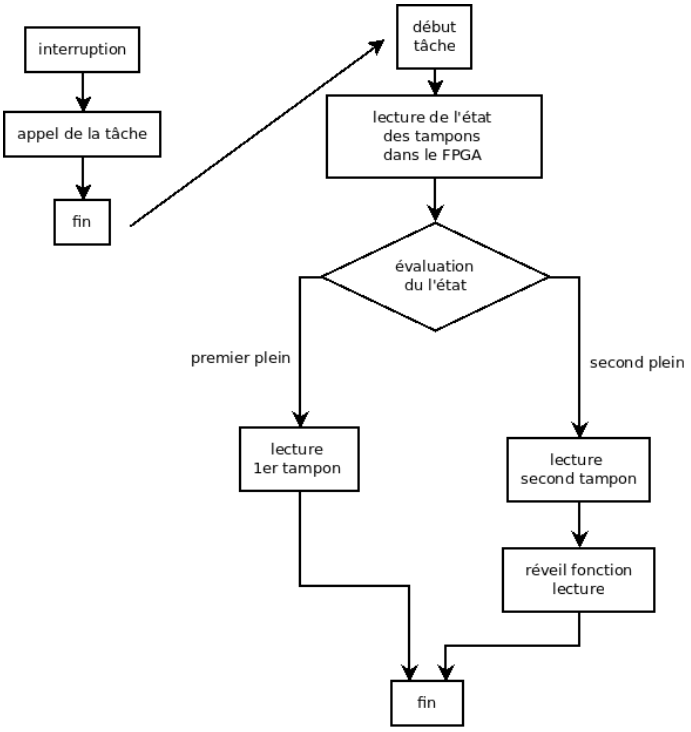
à transférer (partie gauche figure 5.16(b)).

- le pilote va sonder le FPGA pour savoir quel tampon doit être lu et par extension dans quelle zone de son propre tampon les données doivent être stockées ;
- le transfert est initié ;
- lorsque les deux tampons sont transférés, la tâche en charge du transfert réveille la fonction de lecture depuis l'espace utilisateur pour que l'application puisse copier et traiter le flux.

Ce mécanisme est la condition sine qua non pour garantir un flux continu. Si le pilote initiait le stockage/transfert, un certain nombre de données seraient perdues pendant le temps du transfert et avant la prochaine requête.



(a) Diagramme de fonctionnement de la fonction de lecture. L'exécution de cette fonction est mise en veille jusqu'à la mise à disposition d'un tableau de données complet. Une fois le tableau disponible, la fonction se charge de le copier dans l'espace utilisateur et finit son exécution



(b) Diagramme de principe de la gestion de l'avis de mise à disposition d'un tampon complet. Un événement matériel déclenche l'exécution d'une fonction particulière. Celle-ci déclenche à son tour l'exécution d'une tâche, qui, dans un premier temps, vérifie l'état du bloc dans le FPGA, puis va transférer les données. Si l'état est *full* alors la tâche débloque la fonction de lecture.

FIGURE 5.16 – Diagramme des deux parties principales utilisées, d'une part, pour la lecture depuis l'espace utilisateur et d'autre part pour la récupération des données depuis le FPGA.

5.3.4 Résultats

À partir du montage expérimental (figure 5.17), nous avons été en mesure de pouvoir d'une part observer le signal démodulé (partie droite de la figure 5.17) mais également, après connexion, d'un haut-parleur sur la sortie son de la carte APF51, nous avons pu démontrer notre capacité à produire un son de même qualité qu'un poste de radio.

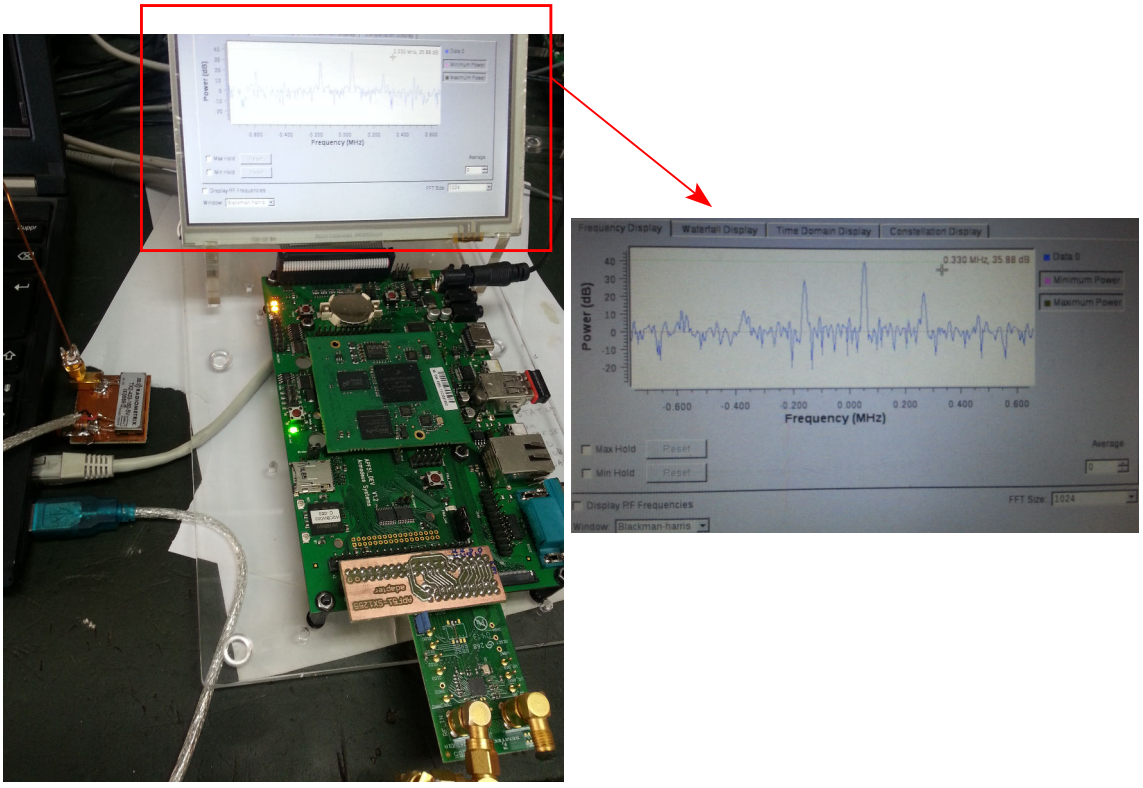


FIGURE 5.17 – Montage expérimental et détail de l'affichage du signal démodulé. À gauche : au premier plan le radio-modem SX1255, sur le côté gauche, connecté à la sortie son d'un ordinateur, l'émetteur FM. Au dernier plan, l'écran pour l'affichage du signal démodulé. À Droite : Représentation fréquentielle du signal démodulé dans le sink GNURadio.

5.4 Traitement du signal

Les applications réalisées précédemment présentent plusieurs inconvénients :

- l'analyse des modes du diapason met en œuvre un ensemble de composants analogiques discrets (switch, générateur de fréquence 2.4 GHz, démodulateur I/Q, atténuateur) qui nécessitent une bonne connaissance du domaine. De plus le montage est relativement complexe et encombrant ;
- l'utilisation du SX1255 présente l'avantage de simplifier la partie analogique de la chaîne en embarquant dans une puce la démodulation du signal, le filtrage et la conversion, mais

présente également l'inconvénient d'avoir une plage de fréquence de fonctionnement limitée.

Il n'est donc pas possible, avec une telle approche, de pouvoir proposer une plate-forme reconfigurable pour des analyses variées, allant de la réception et le décodage de données audio, ou par exemple, des signaux fournis par les avions comme les protocoles ACARS (130.025 MHz-136.900 MHz) et ADS-B (1090 MHz), par les navires comme le AIS (161,975 MHz et 162,025 MHz), ou dans le cas de la métrologie de l'analyse de signaux.

Dans l'optique de limiter la partie analogique de la chaîne de traitement afin d'obtenir une plate-forme adaptée à tous les traitements radio-fréquence temps-réel et de disposer d'un outil entièrement reconfigurable, nous avons décidé de réaliser les implémentations de l'ensemble des blocs nécessaires, en HDL, dans le formalisme CoGen.

Les blocs principaux nécessaires pour disposer, a minima, des mêmes fonctionnalités du SX1255 sont :

- un bloc de démodulation du signal entrant nécessaire pour obtenir l'information de phase (I) et de quadrature (Q) dans le cas de l'analyse de signaux par rapport à un signal de référence (porteuse) dont une image locale est synthétisée par un oscillateur numérique (**NCO**, *Numerically Controlled Oscillator*) ;
- un bloc pour l'implémentation d'un filtre passe-bas dont le but est d'éliminer les composantes fréquentielles supérieures à la porteuse.

La suite de cette section va présenter les divers blocs (NCO, mélangeur et filtres). Pour la simplification du discours et des schémas, et bien que l'ensemble des travaux ait initialement été réalisé sur un HMCAD1511 en mode dual, les explications seront présentées pour un convertisseur ne propageant qu'une seule donnée par cycle d'horloge.

5.4.1 Démodulation

La première étape, que l'on retrouve dans la première démonstration au travers du démodulateur ou dans la seconde, contenu dans le SX1255 au travers d'un mélangeur entre le signal et un signal de référence, est la démodulation.

En traitement analogique, la démodulation d'un signal est réalisée par le mélange entre le signal analysé (RF) et un signal de référence produit par un oscillateur local (LO) pouvant être un générateur de fréquence comme pour le cas du diapason ou une PLL (*Phase-Locked loop*) dans le cas du SX1255. Le mélange est fait d'une part entre le signal analysé et le signal de référence pour obtenir la composante I et le signal de référence déphasé de 90° pour la composante Q.

En traitement numérique, pour la génération du signal de référence, deux choix sont communément utilisés, le premier pour produire une sinusoïde, le second pour un signal carré :

1. l'utilisation d'une ROM contenant une sinusoïde synthétisée a priori. Le déphasage est réalisé par lecture de deux adresses, dont la seconde est incrémentée d'une valeur correspondant à un quart de période, en même temps afin de décaler les deux sorties de 90° (1/4 de période).

Cette solution présente les résultats les plus proches du traitement analogique mais impose une utilisation plus ou moins importante de mémoire ;

2. une approximation de la sinusoïde par un signal carré. Cette solution présente une plus grande simplicité d'implémentation et ne nécessite pas d'utilisation de mémoire. La nature binaire du signal produit permet de limiter le signal en sortie du bloc à 1 seul bit avec '0' pour l'état bas (ou négatif) et '1' pour l'état (ou positif). Toutefois elle présente l'inconvénient de produire des raies aux harmoniques impaires. Ces raies peuvent être atténuéessupprimées par filtrage.

Pour la partie mélangeur, selon le type du signal de référence, deux multiplications sont nécessaires (une pour le I et une pour le Q), si une sinusoïde est utilisée. Dans le cas d'un signal carré, un simple comparateur pour propager la donnée entrante ou son complément à deux suffit à effectuer l'opération recherchée.

Dans notre cas nous avons décidé d'évaluer l'exploitation d'un NCO pour la production d'un signal carré.

5.4.1.1 Implémentation du NCO

Ce type de composant, dont le schéma est présenté sur la figure 5.18, est assez simple : un compteur est utilisé pour aller de la valeur 0 à une valeur maximale, correspondant à la période du signal devant être généré, fixée lors de la construction de la chaîne. Une fois cette valeur atteinte le compteur est réinitialisé.

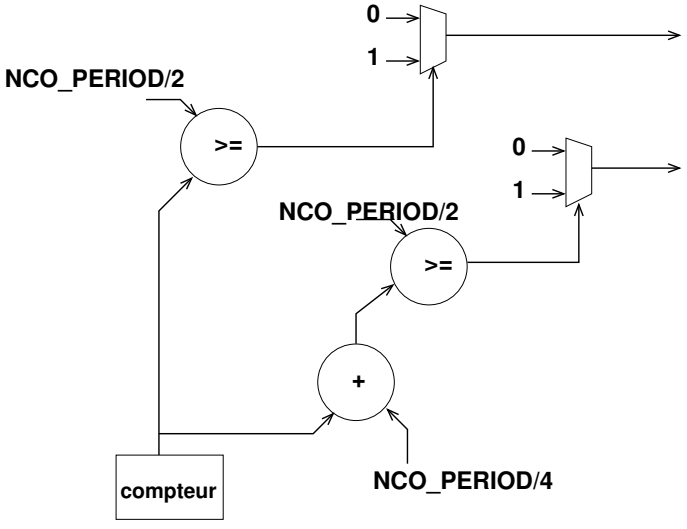


FIGURE 5.18 – Schéma du bloc de génération d'un signal carré et du même signal déphasé de 90°.

À partir de la valeur courante du compteur, une première comparaison est réalisée pour produire la composante. Si le compteur est inférieur à la demi période, la valeur en sortie vaut 1 sinon 0. Pour le second signal (composante Q), déphasé de 90°, la valeur du compteur orignal est incrémentée

d'un quart de période. Ainsi la même comparaison est appliquée pour produire le signal de sortie correspondant.

5.4.1.2 Implémentation du mélangeur

Compte tenu de la nature du flux fourni par le NCO, à savoir 1 bit codant la valeur positive ou négative, ce bloc est également très simple. Son principe de fonctionnement consiste à tester la valeur des signaux I/Q. Si celle-ci est égale à '0', représentant une valeur négative dans le cas d'un sinus ou cosinus, le complément à deux de la valeur reçue depuis l'ADC est propagée ; dans le cas où le signal vaut '1' alors la valeur entrante est directement propagée.

L'ensemble NCO/mélangeur est intégré dans un même bloc dont la description est présentée dans la figure 5.19.

```
<?xml version="1.0" encoding="utf-8"?>
<initial_block name="demod" >
  <user_config>
    <param name="NCO_PERIOD" default="40" />
  </user_config>
  <timings>
        <timing dir="out" pattern="1(*)"  />
  </timings>
  <bloc_interfaces>
    <interfaces dir="in" name="data_in" type="real">
      <interface name="data_in" />
    </interfaces>
    <interfaces dir="out" name="data_out" type="complex">
      <interface name="data_out" />
    </interfaces>
  </bloc_interfaces>
</initial_block>
```

FIGURE 5.19 – Description XML du bloc de démodulation. Celui-ci intègre le NCO et le mélangeur, prend en entrée un flux issu d'un ADC et propage un nouveau flux de type complexe issu du mélange entre le signal RF et l'oscillateur local dont la période est fournie en paramètre du bloc.

5.4.2 Filtre à réponse impulsionnelle finie (FIR)

Le filtre passe bas, dont nous avons réalisé plusieurs implémentations afin de disposer de solutions adaptées à des taux de transferts importants ou au contraire lents, est un Filtre à réponse impulsionnelle finie (FIR *Finite Impulse Response*)[49]. Il est au cœur de la majorité des traitements rapides requis de la part du FPGA (anti-repliement, passe-bas, passe-bande).

Ce filtre a pour but de supprimer les composantes fréquentielles au dessus d'une valeur particulière, afin d'éviter dans le cas de la décimation un repliement des raies dans la bande d'analyse.

5.4.2.1 Principe

Un *FIR* se caractérise par des sorties dépendantes des coefficients du filtres et des valeurs passées de la donnée entrante uniquement. Les sorties sont calculées par convolution. Les sorties dépendent des mêmes coefficients et, partiellement, des mêmes entrées (chevauchement) mais chaque sortie n'est pas liée à une autre.

Ce filtre dont la formule est : $y_n = \sum_{k=0..m} b_k x_{n-k}$ avec y_n une sortie, b_k les coefficients du filtre, x_n les entrées, m le nombre de coefficients avec $m \leq n$, est classiquement implémenté, pour un traitement logiciel, par un algorithme équivalent à l'algorithme 3. Dans celui-ci, la première sortie correspond à y_{m-1}. Ceci se justifie par le retard du filtre. Il est en effet nécessaire de disposer d'au moins m données passées pour calculer y.

Input: coeff_fiter : tableau des coefficients du filtre
Input: data_in : données échantillonnées
Output: data_out : sorties du filtre
Data: m : integer
Data: data_length : integer
Data: k : integer
Data: n : integer
Data: tmp : complex
begin
  $m \leftarrow size(coeff_filter)$
  $data_length \leftarrow size(data_in)$
  $k \leftarrow 0$
  $n \leftarrow 0$
  $tmp \leftarrow 0$
  for $n \leftarrow m-1$ **to** $data_length$ **do**
    for $k \leftarrow 0$ **to** m **do**
      $tmp \longleftarrow coeff_filter[k] * data_in[n-k]$
      $data_out[n] \longleftarrow data_out[n] + tmp$
    end
  end
end

Algorithme 3: Principe d'implémentation d'un FIR dans le cas d'un traitement logiciel.

5.4.2.2 Problématiques

Partant du principe de base de cet algorithme, plusieurs points sont importants à prendre en compte en vue de l'implémentation sur FPGA :

- l'algorithme 3 présente le cas classique d'utilisation sur un ordinateur : le filtrage est réalisé en post-traitement après avoir acquis toutes les données en mémoire. La boucle itérative du premier niveau permet de déplacer la fenêtre de convolution pour produire l'ensemble des $y(n)$ et pour chaque pas, la seconde itération réalise le traitement à proprement parler. Dans le cas d'un FPGA, une telle approche n'est pas souhaitable car le temps de traitement de l'ensemble de données mémorisées risque de dépasser le temps d'acquisition. Compte tenu de la nature du flux et du fait qu'un FPGA est naturellement massivement parallèle, il est donc préférable d'exploiter directement les données au lieu de les stocker. Entre chaque convolution, un retard dans le démarrage est appliqué afin de retrouver le principe de la première itération. Toutefois, cette approche implique de traiter en parallèle un sous ensemble de convolutions. Le nombre de traitements parallèles dépend de la taille des coefficients puisqu'une fois les m données reçues, le bloc est théoriquement disponible pour traiter un nouveau jeu de données. Par extension cette solution va consommer un volume important de blocs multiplieurs. En effet, chaque convolution nécessite 2 multiplieurs pour réaliser la multiplication sur des complexes (dans notre cas les coefficients des filtres sont des réels, leur partie imaginaire est égale à 0 donc le traitement n'est pas nécessaire). Ainsi pour un filtrage avec 128 coefficients, 128 convolutions parallèles doivent être utilisées et donc 256 multiplieurs seraient nécessaires, dépassant les capacités du plus gros FPGA de la gamme Spartan6. La solution à ce problème réside dans le fait qu'une décimation est appliquée en aval du filtrage : ainsi, une partie des résultats ne sont pas exploités et sachant que les convolutions sont toutes indépendantes, il est possible de ne réaliser que les traitements pertinents. Pour le même nombre de coefficients, si une décimation par 4 est appliquée, il est possible de réduire le nombre de multiplieurs à 64 ;
- dans ce type de traitement, la sortie repose sur les entrées passées. Ainsi le calcul de y_n dépend des valeurs $x[n-m..n]$ avec m la taille de la table de coefficients et n l'index de la sortie. Là encore le fait de stocker m données avant de débuter le traitement n'est ni souhaitable ni adapté. La solution mise en œuvre consiste à inverser la table de coefficients ou à utiliser des index pour les coefficients de $m..0$ au lieu de $0..m$. Grâce à cette approche il est ainsi possible d'exploiter directement la donnée ;
- Le dernier point à lever concerne l'ordonnancement des diverses convolutions : l'approche naïve consiste à considérer que lors qu'une convolution a appliqué les m coefficients, le résultat est propagé, et un nouveau traitement démarre. Toutefois, cette option présente l'inconvénient de limiter le facteur de décimation en accord avec le nombre de coefficients. Pour $m = 128$ et un facteur de décimation de 16, le délai entre deux convolutions est de 8, mais si le facteur de

décimation est 10 alors le délai passe à 12.8, et donc la sortie aura un comportement erroné. La solution consiste à considérer que le premier bloc de convolution est initialement démarré par l'arrivée de la première donnée. Ensuite, une fois que celui-ci aura produit un premier résultat, il sera piloté par le dernier bloc de convolution.

5.4.2.3 Implémentations

Comme dans le cas du filtre de détection de contours, cet algorithme est un bon candidat pour disposer de nombreuses implémentations [50] allant d'une acceptance en $1*$ mais avec un besoin très important en terme de multiplieurs, à une implémentation avec une acceptance très faible mais un nombre de multiplieurs également faible. Dans ce cas encore, le fait de disposer de blocs ayant une capacité de réception faible n'est pas un inconvénient car ils peuvent être mieux adaptés pour des convertisseurs lents ou des flux déjà décimés.

Bloc de haut niveau

Compte tenu de la nécessité de réaliser le traitement directement sur le flux, le plus haut niveau de ce bloc (figure 5.20) va contenir autant de sous-blocs dédiés au calcul de la convolution que nécessaire. Pour limiter la consommation de BRAMs, une seule LUT est utilisée au lieu de une par bloc de convolution. Les coefficients sont donc transférés de blocs en blocs avec un retard correspondant au besoin, décalage correspondant au facteur de décimation.

Un compteur est également utilisé pour piloter la BRAM et produire un signal de fin de traitement quand il atteint $m - 1$. Ce signal étant utilisé pour avertir le bloc que le traitement courant est fini et que le résultat doit être propagé.

Finalement, comme les convolutions produisent les données avec un délai relatif au facteur de décimation, par rapport à la convolution précédente, un démultiplexeur est ajouté pour propager le nouveau résultat vers l'extérieur du bloc.

Convolutions

Un bloc de convolution (figure 5.21) reçoit la donnée, le *enable*, la valeur d'un coefficient et un signal qui signale la fin du traitement courant.

Ensuite un premier étage de délais est présent pour retarder les coefficients par rapport aux données afin d'appliquer la décimation demandée. Au dernier étage et selon les cas d'implémentation, soit un DSP48 est utilisé pour réaliser à la fois la multiplication et l'accumulation, soit cette étape est réalisée par un multiplieur classique, et dans ce cas la sortie du multiplieur est connectée à un accumulateur. Dans tous les cas, le *enable* est retardé d'un cycle d'horloge pour permettre le vérouillage de l'accumulateur, et le signal de fin de traitement est lui retardé de deux cycles d'horloge pour permettre de propager la donnée et en parallèle de réinitialiser l'accumulateur.

Autres implémentations possibles

L'implémentation présentée, dans le détail, dans la section précédente est la version disposant de la meilleure acceptance, permettant de pouvoir fonctionner dans toutes les situations. Ceci

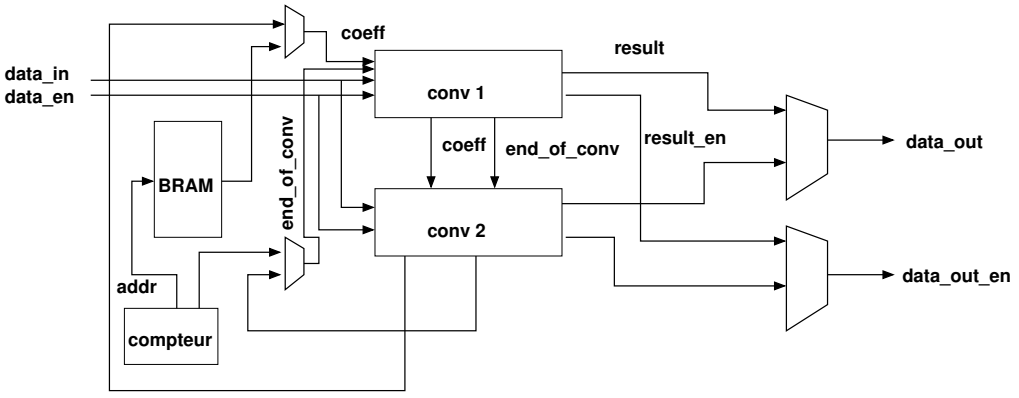


FIGURE 5.20 – Schéma du bloc de haut niveau pour l'implémentation d'un FIR, avec deux convolutions réalisées en parallèle. Les signaux *coeff* et *end_ of_ conversion* pour l'entrée de la première convolution sont issus de la RAM et du compteur pour le premier traitement puis sont, ensuite, pilotés par le dernier bloc de convolution.

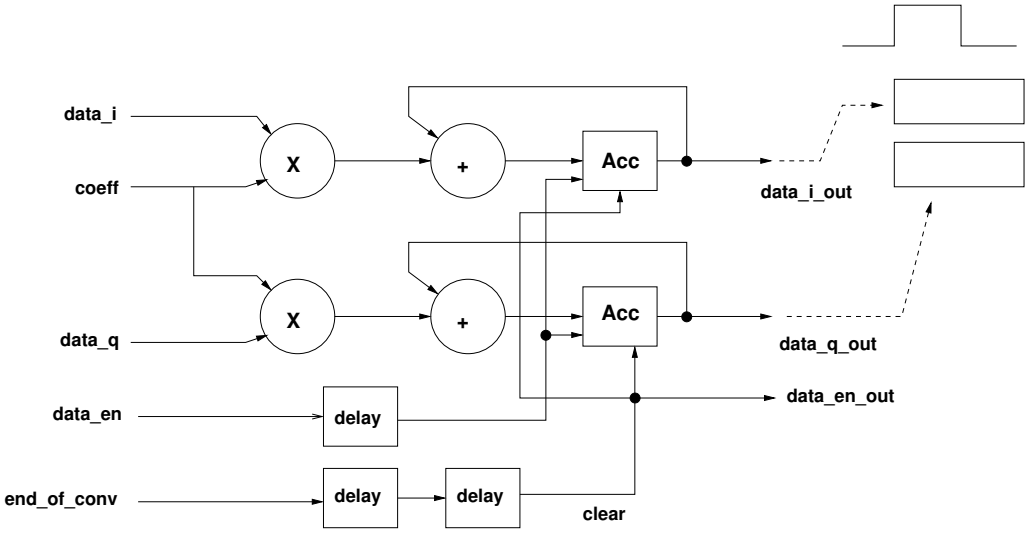


FIGURE 5.21 – Schéma du bloc d'implémentation du traitement de la convolution. Il comporte trois étages : un premier qui réalise la multiplication, de manière asynchrone le résultat est propagé à un additionneur qui correspond au second étage et réalise la somme entre le contenu de l'accumulateur et le résultat issu du multiplieur. Enfin, le résultat est stocké dans un registre synchrone, piloté par le signal *enable* décalé d'un cycle d'horloge. Le signal de fin de traitement est utilisé pour réinitialiser le contenu du registre mais également comme nouveau *enable*.

s'obtient aux dépends des ressources. Dans le cas d'un convertisseur tel que le HMCAD1511 ou d'autres comme le *LTC2158* qui peut être cadencé jusqu'à 310 MHz, seule cette implémentation est acceptable. Mais pour des convertisseurs plus lents ou si le filtrage/décimation est cascadé, il est possible de sauver des ressources grâce à des implémentations certes plus lentes mais plus

économes et finalement plus adaptées aux besoins.

Une autre solution (fig 5.22), qui reprend le bloc de haut niveau précédent mais modifie légèrement le bloc de la convolution, consiste à réaliser la multiplication de la valeur complexe par le coefficient en deux cycles d'horloges. Ainsi, pour le même nombre de coefficients et la même décimation, il est possible de diviser par deux la consommation de multiplieurs. La description de cette solution est identique à la précédente hormis la consommation de ressources et l'acceptance qui passe en **[10]***.

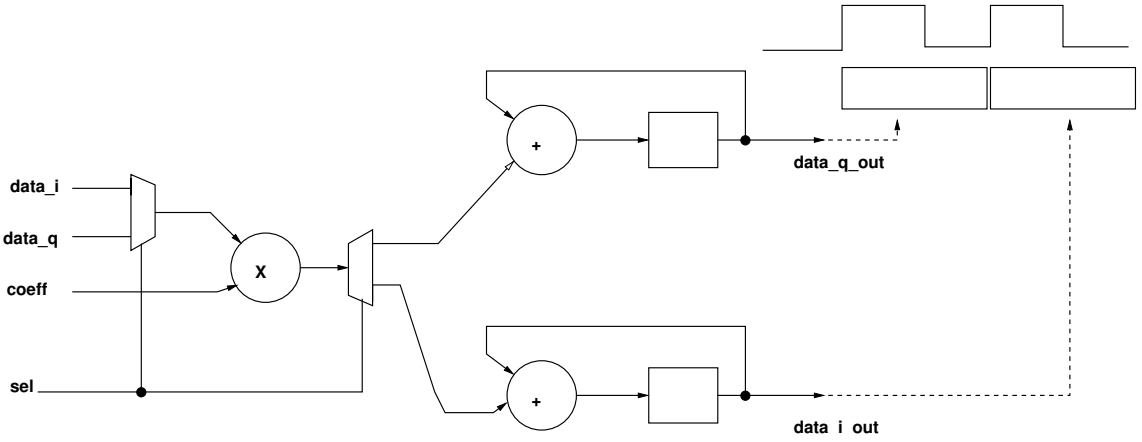


FIGURE 5.22 – Exemple simplifié de convolution qui, pour économiser les multiplieurs, utilise le même composant, séquentiellement, pour le calcul de $coeff * data_i$ puis $coeff * data_q$.

Il est bien entendu possible de pousser encore plus loin l'économie de ressources en exploitant un même multiplieur pour plus qu'une convolution. Ainsi, par exemple, il est possible de traiter deux convolutions consécutives en réalisant séquentiellement sur deux cycles la multiplication pour la première, puis sur les deux cycles suivants la multiplication de l'autre convolution, un ensemble de multiplexeurs étant ajoutés pour transférer le résultat vers l'accumulateur correspondant. Un tel bloc verrait à nouveau sa consommation divisée par deux et son acceptance passer en **1000***.

5.4.3 Bilan

Afin de vérifier le bon fonctionnement de la chaîne de traitement, nous injectons un signal à 10 MHz à l'entrée du système. À partir des données brutes issues de l'ADC, nous effectuons une simulation de tous les traitements, à savoir la démodulation (à 10 MHz) à partir d'un NCO carré, le filtrage par convolution permettant d'accéder aux données I et Q, la décimation par 16, le calcul de la phase et enfin l'évaluation de la densité spectrale de puissance du signal. La validité de chaque étape implémentée en VHDL est testée par rapport à une implémentation de référence sous Matlab. La comparaison entre le post-traitement, sur ordinateur personnel, des données brutes acquises par le FPGA, et le traitement complet effectué sur FPGA, est proposée sur la Fig. 5.23. Ce calcul est effectué sur un échantillon de 32895 données (32768 points, plus 127 correspondant à l'ordre du

filtre permettant, après décimation, de calculer le spectre sur 2048 points) : le plancher de bruit ainsi que la courbe proche de la porteuse sont en parfait accord entre les deux calculs.

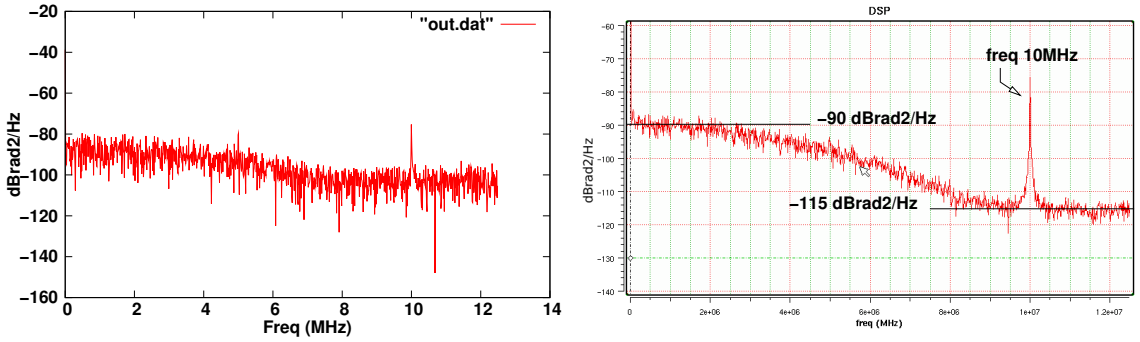


FIGURE 5.23 – Comparaison entre post-traitement sur ordinateur basé sur des données brutes (gauche) et traitement complet de la chaîne ADC-FPGA-i.MX (droite). Les résultats concordent parfaitement, avec un plancher à -115 dBrad^2/Hz et un bruit de phase de -90 dBrad^2/Hz à quelques kHz de la porteuse, et valident l'implémentation des l'algorithmes en blocs VHDL. La capture d'écran de la partie droite représente de plus le moyennage de 10 spectres.

La partie droite de la figure 5.23 représente une capture d'écran de l'APF 27, résultat de l'implémentation ADC-FPGA-IMX de toute la chaîne de traitement. Après filtrage, le flux I/Q est transféré vers le processeur pour calcul de la phase et du spectre. Nous y avons également inclus une fonction de moyennage, permettant de filtrer efficacement le bruit non corrélé.

Avec cet ensemble d'implémentations, il nous a été possible de nous affranchir de l'étage analogique de démodulation RF pour réaliser le même type de traitements. Ce type de développement constitue la pierre angulaire de nombreuses applications, notamment dans le cadre de la métrologie du temps et des fréquences où l'évaluation de l'instabilité d'une source de fréquence ultrastable (i.e. à 10^{-15}) est primordiale.

Pour s'en convaincre, nous avons volontairement calibré une source de fréquence à un niveau de bruit de $-113,5$ dBrad^2/Hz qui constitue notre signal sous test. Ceci est représenté par la ligne horizontale (verte) de la figure 5.24. À partir des données brutes issues de l'ADC, nous reconstruisons la densité spectrale de puissance de bruit du signal, représentée par la courbe (bleue) dont le plancher (vers les hautes fréquences) est quelque peu supérieur à celui de l'autre courbe et dû à un calcul effectué en précision fixe.

L'autre courbe (rouge) représente le calcul du spectre de bruit à partir des données I/Q disponibles après démodulation, filtrage et décimation dans le FPGA.

On observe l'atténuation du filtre s'opérer à partir de 4 MHz. Dans ce type de graphique, ceci n'est pas gênant lorsque l'information utile se situe plus près de la porteuse.

Il est à noter que dans ce type de montage, la limite de mesure est directement liée au bruit

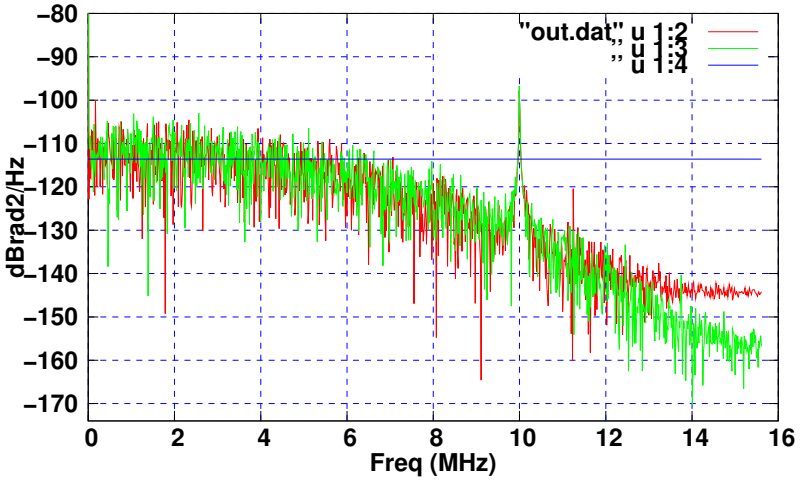


FIGURE 5.24 – Comparaison du plancher d'une source de bruit calibrée (vert) et de la densité spectrale de bruit issue du traitement du signal radio-fréquence échantillonné ; en bleu, traitement totalement logiciel réalisé sur les données brutes ; en rouge démodulation, filtrage et décimation réalisées dans le FPGA.

propre de quantification de l'ADC [51, 52, 53, 54], évalué théoriquement par la formule $\frac{V_q^2}{12} \cdot \frac{1}{BW}$ avec $V_q = \frac{V_{ref}}{2^N}$ avec N le nombre de bits de l'ADC, V_{ref} l'amplitude maximale d'entrée et BW la fréquence d'échantillonnage divisée par deux. Pour $N = 8$ ce plancher vaut -135 dBrad^2/Hz en accord avec les valeurs observées expérimentalement. En métrologie Temps-Fréquence, ce dispositif simple n'est pas suffisant pour atteindre les performances ultimes (-174 dBrad^2/Hz correspondant au plancher de bruit thermique), mais en constitue la brique élémentaire d'ores et déjà étudiée à l'Institut FEMTO-ST.

5.5 Conclusion

Ce chapitre a montré qu'il était possible, dans le cadre de CoGen, d'ajouter et d'exploiter des blocs de traitements de signaux à une dimension, sans avoir à faire de modifications dans les spécifications de l'outil.

Nous avons démontré notre capacité à utiliser un ADC rapide pour sonder un capteur. Puis nous avons validé, à la fois, la possibilité d'utiliser un radio-modem pour remplacer l'ensemble des composants analogiques de mise en forme du signal RF par un seul composant incluant toutes les fonctionnalités de transposition de fréquence et de conversion analogique-numérique. Par ailleurs, nous avons étendu les capacités de traitement du FPGA en démontrant notre capacité à interfacer une chaîne de traitement générée par CoGen avec un outil tel que GNURadio, permettant ainsi de reposer la partie logicielle sur un ensemble de blocs déjà existant et décrit dans un niveau d'abstraction plus élevé. Finalement, nous avons réalisé un ensemble de blocs de traitements destinés au FPGA, permettant de limiter l'utilisation de composants discrets, tel que le démodulateur, de

gagner en flexibilité et de disposer d'une plate-forme reconfigurable.

Concernant GNURadio, nous avons démontré notre capacité à réaliser une source compatible avec cet outil, correspondante à un bloc terminal CoGen. Pouvoir exploiter GNURadio, et profiter des bibliothèques existantes, pour la partie logicielle va permettre de simplifier également, du côté du processeur, les traitements. Toutefois, les blocs qui constituent la chaîne de traitements dans le FPGA, ainsi que le périphérique d'acquisition, peuvent nécessiter des configurations (coefficients d'un filtre, réglage du gain). Cet ensemble d'informations doit être paramétré depuis GNURadio. L'approche qui consiste à intégrer l'ensemble de ces informations dans le bloc source, qui serait considéré comme une abstraction totale du FPGA, n'est pas souhaitable car il devrait y avoir autant de combinaisons que de chaînes de traitements. La seconde approche, consistant à recréer, dans GNURadio, la chaîne contenue dans le FPGA, avec des blocs dont le rôle serait de fournir au FPGA les paramètres et de copier leur(s) entrée(s) dans leur(s) sortie(s) n'est pas souhaitable non plus pour deux raisons : d'une part ce serait une dupplication de la construction, d'autre part la gestion de blocs en plus dans la chaîne logicielle serait une augmentation des traitements réalisés par le cœur de GNURadio. Notre point de vue est donc de considérer des blocs, à l'instar des *variables* (sans entrée, ni sortie) dont le rôle serait, au lancement de l'application ou pendant son l'execution, de transférer au FPGA les paramètres ou informations nécessaires. Ainsi, au même titre qu'il y a une étape de création d'un driver dédié à un bloc dans le FPGA, il devra y avoir la création d'un bloc pour la GNURadio ; et lors de la construction d'une chaîne, CoGen sera en mesure de constituer la base d'un projet GNURadio, comportant le bloc source ainsi que les blocs de configurations.

Conclusion générale

L'objectif principal de cette thèse était la création d'un outil de génération de chaînes de traitements, sur FPGA, avec pour but principal de simplifier le travail de l'utilisateur en faisant reposer l'assemblage sur des blocs déjà disponibles, validés et qualifiés.

Cet outil, pour réaliser des traitements fiables, doit tenir compte d'une part des contraintes matérielles des FPGAs et des langages HDL et, d'autre part, des caractéristiques globales des algorithmes. C'est pourquoi, afin de réaliser une spécification de l'outil, nous avons étudié les problématiques des algorithmes de traitements d'images mais également ceux applicables sur des flux radio-fréquences et nous avons synthétisé les contraintes de ces deux gammes d'applications.

Ces analyses ont permis de considérer une solution basée sur des interfaces standardisées, que tous les blocs d'un même type doivent utiliser, afin de garantir que les interconnexions soient toujours possibles sans ajouter une consommation de ressources du FPGA et un ajout de délais inutiles. En considérant les divers algorithmes, il nous a semblé judicieux de permettre pour un même algorithme de disposer de plusieurs implémentations, non visibles par l'utilisateur mais qui peut permettre à l'outil d'évaluer un ensemble de combinaisons pour ne conserver que celles qui seraient compatibles avec l'architecture cible et qui ne nécessiteraient pas un nombre supérieur de ressources à la capacité maximale de la cible. Le dernier point important concernait la capacité de traitement d'un bloc. Selon le type de l'implémentation, un bloc peut être capable de recevoir une nouvelle donnée à chaque cycle d'horloge ou au contraire nécessiter plusieurs cycles pour compléter le traitement. Pour permettre à CoGen d'être en mesure de réaliser un ensemble d'analyses basés sur la possibilité de connecter les blocs entre eux, la compatibilité matérielle et la capacité de recevoir le flux issus du bloc précédent, nous avons mis en place des balises XML, format de description choisi pour décrire l'ensemble des caractéristiques du bloc. La création du fichier de description est à la charge du développeur lors de l'intégration de son bloc dans CoGen. Un dernier point important, bien que non encore intégré, concernait les RAMs externes dont le comportement n'est pas synchrone de celui du FPGA. Un tel composant présente des latences qui peuvent altérer la réprésentation du flux et doit donc être le mieux connu pour mettre en place une politique efficace destinée à limiter voir supprimer l'aspect non prédictible de son comportement.

Ayant mis en place les bases de l'outil, tant au niveau des caractéristiques minimales des blocs, qu'au niveau des informations à fournir, nous avons pu évaluer la difficulté et les avantages in-

duits par cette méthodologie sur du traitement d'images puis sur du traitement de signaux radio-fréquences. Nous avons pu constater que la contrainte d'interfaces particulières n'était en rien une limitation ni n'apportait d'inconfort dans le développement. Le cas de la radio-fréquence a également permis de réaliser une première tentative pour lier une chaîne de traitement temps-réel produite par CoGen avec l'environnement logiciel GNURadio pour réaliser la suite du traitement et profiter de ses blocs déjà disponibles. Il a été, par ailleurs, prouvé que le concept de l'ajout de blocs dans GNURadio, pour configurer les divers blocs dans le FPGA, est fonctionnel.

Un des points qu'il reste à ajouter concerne la prise en charge de composants ayant les mêmes caractéristiques que les cartes *Armadeus Systems*, mais incluant dans la même puce à la fois un FPGA et un cœur de CPU minimal, comme le *Zynq* de *Xilinx* et le *Cyclone V SOC* de *Altera*. Ce type de composant nécessite de prendre en compte le fait que certaines fonctions ne sont pas intégrées dans le CPU mais passent par le FPGA. Cette caractéristique implique donc que CoGen puisse ajouter des composants ne faisant pas partie de la chaîne de traitement mais ayant des caractéristiques de consommations de ressources.

Le second point concerne le protocole de communication entre le processeur et les blocs dans le FPGA : sur une carte *Armadeus* le protocole est le *wishbone*, dans le cas d'un *Zynq* le protocole est de type *AXI*. Il est donc nécessaire de considérer, dans le choix des blocs, que le protocole est un critère discriminant à prendre en compte lors de l'évaluation de la compatibilité d'un bloc avec l'environnement cible.

Le dernier point concerne directement les outils de synthèse imposés par les fondeurs. Ceux-ci sont en constante évolution. Les outils tels que CoGen, POD, MiGen, doivent suivre ces évolutions. Avec l'arrivée de *Vivado* pour les nouvelles générations de FPGAs de *Xilinx*, il semble, a priori, être nécessaire de respecter une certaine structure pour les blocs HDL. La question est de savoir s'il est malgré tout possible de produire des projets avec un outil générique comme POD et avec une structuration non spécifique à un outil, ou si au contraire il sera nécessaire de proposer pour ces cas particuliers une version adaptée et d'ajouter à CoGen la capacité de directement communiquer avec l'outil de synthèse au lieu de passer par POD.

Table des figures

1.1 Schéma global de l'environnement matériel utilisé pour ce projet. 11
1.2 Convolution par un noyau 3x3. La production d'un nouveau pixel P1,1 nécessite de disposer du pixel P2,2 de l'image source. Dans le cas du traitement sur un flux ce dernier n'arrivera qu'après une durée correspondant au temps de propagation de tous les pixels qu'une ligne, plus 1 pixel, ainsi que de potentiels temps de pauses inclus dans le flux vidéo. 13
1.3 Implémentation optimale d'un traitement qui reçoit deux données en parallèle, les multiplie par une troisième valeur avant de réaliser une accumulation. Du fait de l'utilisation de deux multiplieurs en parallèle, un seul cycle d'horloge est nécessaire pour réaliser les deux traitements. La capacité de réception de données d'une telle implémentation est maximisée car chaque multiplieur sera disponible au cycle suivant la réception. 18
1.4 Implémentation visant à économiser les multiplieur. Les données $data_i$ et $data_q$ sont séquentiellement traitées. Ainsi, le bloc multiplieur n'est pas disponible pour le lot suivant pendant deux cycles d'horloge. Cette implémentation présente une bande passante deux fois plus lente que l'implémentation de 1.3 mais pourra être privilégiée si les resources matérielles l'exigent et que le flux entrant n'est pas trop élevé. . . . 18

2.1 Exemple de chaîne de traitements appliqué au cas particulier de l'acquisition vidéo. 24
2.2 Représentation d'une chaîne basique telle que stockée en mémoire. 25
2.3 Deux types de chaîne disposant de branches de traitements parallèles. 2.3(a) : branches indépendantes disposant chacunes de leur propre bloc de propagation, 2.3(b) : branches parallèles se rejoignant. 25
2.4 Interface type pour un bloc recevant et produisant un flux vidéo. 28
2.5 Chronogramme de propagation des informations au travers d'un composant réalisant une tâche dont la durée entre l'arrivée d'une donnée et sa sortie correspond à un cycle d'horloge. 30
2.6 Schéma de la structure hiérarchique des fichiers XML utilisés par CoGen dans le cadre des analyses. 32

2.7 Diagramme de fonctionnement global de CoGen. 40
2.8 Parcours d'une chaîne présentant des branches parallèles. 45
2.9 Divers cas de comparaison entre un flux et une acceptance. 47
2.10 Deux cas de génération du flux de sortie avec dans le premier cas une décimation sur un flux constant et dans le second, la même décimation mais sur un flux présentant des temps morts. 50
2.11 Exemple de chaîne de traitements. Elle comporte un bloc initial (camera), deux blocs de traitements (median00 et cropping00) et un bloc terminal (csi). 51
2.12 Première chaîne proposée pour validation. Elle comporte un bloc initial servant à la réception de données de type vidéo et un bloc qui réalise une détection de contours. 52
2.13 Seconde chaîne de traitements. Elle comporte en plus des blocs de la première chaîne un bloc dédié à la décimation. Ainsi en sortie de ce nouveau bloc, la fréquence du flux initial est divisée par 2. 52
2.14 Schéma du calcul de la dérivée horizontale avec une utilisation optimale des blocs DSP48. 54
2.15 Schéma du calcul de la dérivée horizontale avec utilisation au premier niveau de multiplieurs et au second de registres pour les additions. 54
2.16 Afin de réduire le volume de multiplieurs nécessaire, des multiplexeurs sont utilisés pour charger séquentiellement les coefficients pour la dérivée en x et en y, ainsi que pour réaliser la dernière addition. 55
2.17 Afin d'obtenir une implémentation ne nécessitant qu'un seul multiplieur, chaque entrée de celui-ci est multiplexée pour charger séquentiellement les coefficients et les pixels. 56

3.1 Chronogramme de principe du mécanisme d'écriture dans la RAM pour un volume de transfert de 2 mots. Partie gauche : transfert des données vers la FIFO ; partie droite : envoi de la commande d'écriture physique dans la RAM. 62
3.2 Chronogramme de principe du mécanisme de lecture depuis la RAM pour un volume de transfert de 2 mots. Partie gauche : transfert physique depuis la RAM dans la FIFO de lecture ; partie droite : récupération des données contenues dans la FIFO. 63
3.3 Chronogramme de l'évolution des signaux selon les étapes réalisées lors de la réception d'une commande de lecture. 65
3.4 Schéma de connexion de composants au contrôleur de RAM au travers du composant client. Deux clients sont connectés, le premier au port 2, le second au port 5. Deux blocs sont connectés aux clients. 69
3.5 Schéma de l'interface de communication entre un bloc utilisateur et un client de RAM. 69

3.6 Diagramme de flux de la logique d'écriture. À gauche, processus en charge des interactions avec l'utilisateur. À droite, processus en charge de la gestion du transfert vers la RAM, du stockage de l'adresse de base et de la mise à jour du compteur d'écritures réalisées par l'utilisateur. 71

3.7 Diagramme de flux de la logique de lecture. À gauche, processus en charge des interactions avec l'utilisateur. À droite, processus en charge du remplissage de la FIFO lorsqu'elle est vide ou qu'elle a atteint le seuil bas. 72

4.1 Chronogramme de transfert d'une image de 2 lignes par 2 colonnes. 80

4.2 Schéma du bloc terminal chargé de convertir un flux vidéo compatible CoGen vers un flux vidéo au format CSI. À gauche : conversion du format des données de la chaîne vers un format RGB565 ou RGB888. À droite : bloc dédié à la gestion des signaux du bus CSI. 81

4.3 Schéma de l'application HDL de test du fonctionnement de l'implémentation du protocole CSI. 82

4.4 Test du fonctionnement de l'implémentation HDL du protocole **CSI** : transmission d'une image, préalablement stockée dans une RAM externe, pour affichage. 83

4.5 Structure et représentation des bits transférés par une caméra au moyen du protocole cameralink. Référence : http://www.thefullwiki.org/Camera_Link. 84

4.6 Chronogramme du flux cameralink après désérialisation des données transmises. . . 84

4.7 Fichier de description XML du bloc cameralink. 86

4.8 Chronogramme de la prise en compte du changement de domaine d'horloge entre une horloge à 80 MHz (12.5 ns de période) et une horloge à 100 MHz (10 ns de période). Nous pouvons constater que tous les 5 cycles de l'horloge à 100 MHz un "blanc" apparaît, induit par la dérive d'un facteur $\frac{100}{80} = \frac{5}{4}$. 87

4.9 Test de la caméra Photonfocus : schéma de l'application FPGA. 87

4.10 Test de la caméra Photonfocus : transmission d'une image, reçue depuis la caméra. 88

4.11 Schéma du bloc d'implémentation d'un seuillage simple. 89

4.12 Fichier de description du bloc de traitement pour un seuil simple. L'utilisateur peut changer le paramètre du seuil qui est fixé par défaut à 128. 90

4.13 Fichier de description de l'implémentation d'un seuillage simple. Ce bloc est capable d'accepter des données à chaque cycle d'horloge et ne modifie pas le flux. 90

4.14 Résultat de l'application d'un filtre de seuillage en temps-réel sur le flux issu d'un capteur vidéo. 91

4.15 Images issues de la caméra Photonfocus : 4.15(a) image non traitée ; 4.15(b) résultat du filtrage de l'image appliquée directement sur le flux avant transfert au processeur. 91

4.16 Schéma de l'implémentation du stockage des données dans des BRAMs avec un principe de tampon rotatif pour l'écriture (partie gauche) et la lecture (partie droite). La sortie de l'accumulateur en bas à gauche (signal **cpt**) est utilisé comme contrôle pour l'écriture dans une RAM particulière et pour le lien entre chaque RAM et les multiplications. Sa valeur évolue à chaque fin de ligne. 93

4.17 Description XML d'un des blocs d'implémentations de l'algorithme de détection de contours. 94

4.18 Montage expérimental pour la mesure de vibration d'un diapason musical par traitement d'images. Le haut-parleur proche du diapason est utilisé pour exciter le diapason à la fréquence de 442 Hz (proche de la valeur nominale de 440 Hz, due à une modification du diapason pour le collage d'une mire à son extrémité. 95

4.19 Cible disposant d'un réseau périodique – un motif imprimé d'une résolution de 300 dpi – collée sur le bout d'une des branches d'un diapason musical à 440 Hz et observée par une caméra pour une détection de mouvement. 95

4.20 Schéma du bloc de traitement vidéo : comme deux pixels consécutifs sont transférés par la caméra à chaque cycle d'horloge, la somme des deux pixels consécutifs est réalisé par un premier additionneur. La multiplication par la fonction d'analyse est ensuite réalisée, puis le résultat est accumulé pour calculer l'inter-corrélation le long de la ligne. Le calcul est effectué en parallèle sur la partie réelle (composante I) et sur la partie imaginaire (composante Q) avec seul le changement de la fonction d'analyse de $\cos(2\pi x/p)$ en $\sin(2\pi x/p)$. 96

4.21 Description XML du bloc d'implémentation relatif au calcul pour le diapason. . . . 97

4.22 Capture sur un oscilloscope numérique de la sortie des convertisseurs numérique-analogiques, représentant les composantes I et Q du calcul de la inter-corrélation, convertis en angle, par le calcul de $\arctan(Q/I)$ en post-traitement. La fréquence de rafraîchissement est de 5.2 kHz, bien supérieure à la fréquence d'oscillation forcée de 442 Hz. 98

4.23 Phase calculée par $\arctan(Q/I)$ et mesurée pour diverses tensions d'excitation du haut-parleur. Pour la tension la plus importante, la rotation de phase de 2π est visible (courbe bleue). Régler l'offset de phase, φ_0, maximise la plage de mesure en conservant la rotation de phase dans la gamme $[\pi/2, \pi/2]$. 99

4.24 Seconde implémentation : contrairement à la première version, le pré-additionneur n'est pas utilisé, le même traitement est réalisé pour le pixel N et $N+1$, et une dernière addition est opérée pour la fusion des deux résultats partiels. Les signaux de contrôle ne sont pas représentés sur cette figure car identiques à ceux de la figure 4.20. 101

4.25 Résonance d'un diapason par méthode optique jusqu'à 12 kHz. On observe deux modes : le premier à 440 Hz et le second à 9840 Hz. 102

5.1 Deux solutions pour la démodulation d'un signal RF : 5.1(a) en analogique, 5.1(b) en numérique. 106

5.2 Chronogramme des signaux entre le convertisseur-analogique numérique et le FPGA dans le cas du mode 2 voies. Les paires Dx1A, Dx1B, Dx2A, Dx2B fournissent les données pour la voie 1 et les paires Dx3A, Dx3B, Dx4A, Dx4B pour la voie 2. Source : documentation hittite. 107

5.3 Fichier de configuration du bloc HMCAD1511 pour une configuration deux voies. L'utilisateur peut, lors de la construction de la chaîne fournir la fréquence d'échantillonnage (balise *param*), la fréquence fournie pour la sortance est calculée sur la base de l'information fournie par l'utilisateur et est divisée par 4 en accord avec les caractéristiques de ce mode. Deux balises *interfaces* contiennent chacune quatre balises *interface* correspondant à 4 des 8 sorties de l'ADC fournies, là encore, en respectant les caractéristiques du convertisseur. 110

5.4 Résultat de l'acquisition d'un signal sinusoïdal d'une fréquence de 10 MHz avec une fréquence d'échantillonnage de 500 MHz. Bas : représentation fréquentielle du signal échantillonné. En vignette, représentation temporelle. 111

5.5 Schéma complet de l'application d'interrogation de capteur. 112

5.6 Schéma de l'application FPGA dans sa globalité, comportant un bloc initial (HMCAD1511), un bloc dont le rôle est de piloter le switch pour charger la ligne à retard puis de transférer au(x) suivant(s) l'ensemble des points correspondant à la décharge du capteur, un bloc terminal pour le stockage de la rafale, un bloc dédié à l'extraction d'un point particulier et finalement un second bloc terminal utilisé pour le stockage d'un nombre configuré de points correspondant à un point particulier de la décharge. 113

5.7 Courbes observées : à gauche, représentation temporelle de la décharge du capteur avec ses 8 résonances. À droite, affichage d'une suite d'échantillons correspondant au point 700 ($\frac{1}{500MHz} * 700 = 1.400\mu s$) après la charge du capteur. 114

5.8 Réponse impulsionnelle (vignettes) et représentation fréquentielle des modes propres du diapason, caractérisé par une jauge de contrainte soudée, par méthode RADAR. Quatre modes sont visibles : 440 Hz, 5465 Hz, 10143 Hz, 40714 Hz. Nous y retrouvons les résonances à 440 Hz et 10143 Hz, celle à 40714 Hz ne pouvant pas être observée en méthode optique du fait de la fréquence d'échantillonnage trop basse. Nous ignorons la raison de l'absence de celle à 5465 Hz dans la méthode optique. 115

5.9 Schéma interne du SX1255. Sur la partie gauche la modulation/démodulation de signaux radio-fréquences. Au centre, la conversion analogique-numérique pour la réception et la conversion numérique-analogique pour l'émission. À droite, la décimation et la conversion en un flux **i2s** pour la réception ou la conversion du flux **i2s** pour l'émission. Source : Semtech. 116

5.10 Chronogramme du protocole série *i2s* utilisé pour la transfert de données. Un signal d'horloge (CLK_OUT) cadence la lecture/écriture sur le front montant, respectivement descendant. le signal *WS* est utilisé comme délimiteur entre deux données consécutives. Source : Semtech. 116

5.11 Schéma du bloc d'implémentation du SX1255. Celui-ci comporte un sous-bloc de conversion *i2s* pour la réception des données, un sous-bloc qui réalise le changement de domaine d'horloge entre celle du radio-modem et l'horloge du FPGA, une implémentation du protocole *SPI* pour la configuration faite, au travers du bus wishbone, depuis le processeur. 117

5.12 Fichier de description du bloc SX1255. Le bloc est paramétrable pour la fréquence du bus *i2s* et la taille des données reçues. Du fait de la simplicité du protocole de transfert, ce bloc ne consomme pas de ressources matérielles particulières. 117

5.13 Exemple de chaîne de traitements GNURadio. Une source (équivalent d'un bloc initial CoGen) fournit un flux de données de type complexe à un bloc de démodulation FM (équivalent d'un bloc de traitement). Ce dernier propage ses résultats à deux blocs de type sink (équivalent aux blocs terminaux), dont le premier est dédié à l'affichage et le second au transfert vers une carte son. 118

5.14 Schéma global de l'application visée. À gauche, le traitement dans le FPGA (conversion du flux, décimation, stockage/transfert). À droite, partie logicielle. 119

5.15 Implémentation du bloc terminal pour le transfert des données, avec utilisation du principe du double tampon. Le compteur est utilisé pour incrémenter l'adresse de la RAM, il est également utilisé pour déterminer à quel moment une interruption doit être levée et mettre à jour le statut du bloc. Le bloc wishbone gère l'ensemble des transactions de lecture et de statut. 120

5.16 Diagramme des deux parties principales utilisées, d'une part, pour la lecture depuis l'espace utilisateur et d'autre part pour la récupération des données depuis le FPGA. 121

5.17 Montage expérimental et détail de l'affichage du signal démodulé. À gauche : au premier plan le radio-modem SX1255, sur le côté gauche, connecté à la sortie son d'un ordinateur, l'émetteur FM. Au dernier plan, l'écran pour l'affichage du signal démodulé. À Droite : Représentation fréquentielle du signal démodulé dans le sink GNURadio. 122

5.18 Schéma du bloc de génération d'un signal carré et du même signal déphasé de 90°. 124

5.19 Description XML du bloc de démodulation. Celui-ci intègre le NCO et le mélangeur, prend en entrée un flux issu d'un ADC et propage un nouveau flux de type complexe issu du mélange entre le signal RF et l'oscillateur local dont la période est fournie en paramètre du bloc. 125

5.20 Schéma du bloc de haut niveau pour l'implémentation d'un FIR, avec deux convolutions réalisées en parallèle. Les signaux *coeff* et *end_of_conversion* pour l'entrée de la première convolution sont issus de la RAM et du compteur pour le premier traitement puis sont, ensuite, pilotés par le dernier bloc de convolution. 129

5.21 Schéma du bloc d'implémentation du traitement de la convolution. Il comporte trois étages : un premier qui réalise la multiplication, de manière asynchrone le résultat est propagé à un additionneur qui correspond au second étage et réalise la somme entre le contenu de l'accumulateur et le résultat issu du multiplieur. Enfin, le résultat est stocké dans un registre synchrone, piloté par le signal *enable* décalé d'un cycle d'horloge. Le signal de fin de traitement est utilisé pour réinitialiser le contenu du registre mais également comme nouveau *enable*. 129

5.22 Exemple simplifié de convolution qui, pour économiser les multiplieurs, utilise le même composant, séquentiellement, pour le calcul de $coeff * data_i$ puis $coeff * data_q$. 130

5.23 Comparaison entre post-traitement sur ordinateur basé sur des données brutes (gauche) et traitement complet de la chaîne ADC-FPGA-i.MX (droite). Les résultats concordent parfaitement, avec un plancher à -115 dBrad^2/Hz et un bruit de phase de -90 dBrad^2/Hz à quelques kHz de la porteuse, et valident l'implémentation des l'algorithmes en blocs VHDL. La capture d'écran de la partie droite représente de plus le moyennage de 10 spectres. 131

5.24 Comparaison du plancher d'une source de bruit calibrée (vert) et de la densité spectrale de bruit issue du traitement du signal radio-fréquence échantillonné ; en bleu, traitement totalement logiciel réalisé sur les données brutes ; en rouge démodulation, filtrage et décimation réalisées dans le FPGA. 132

Liste des tableaux

2.1 Récapitulatif des acceptances, sortances et latences pour les 4 implémentations proposées. 56

3.1 Caractéristiques temporelles de l'ensemble contrôleur et RAM pour des transferts de 2, 32 et 64 mots, sans ou avec changement de zone. 65

3.2 Caractéristiques temporelles pour le second port avec un accès de type conflit, pour des transferts de 2, 32 et 64 mots, sans ou avec changement de zone. 66

3.3 Caractéristiques temporelles pour des transferts de 2, 32 et 64 mots, sans ou avec changement de zone, en mode lecture. 67

Bibliographie

[1] D. H. Jones, A. Powell, C.-S. Bouganis et P. Y. K. Cheung : Gpu versus fpga for high productivity computing. *In Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, FPL '10, pages 119–124, Washington, DC, USA, 2010. IEEE Computer Society.

[2] R. Kalarot et J. Morris : Comparison of fpga and gpu implementations of real-time stereo vision. *In Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*, pages 9–15, June 2010.

[3] Cristian Grozea, Zorana Bankovic et Pavel Laskov : Fpga vs. multi-core cpus vs. gpus : Hands-on experience with a sorting application. *In* R. Keller, D. Kramer et J.-P. Weiss, éditeurs : *Facing the Multicore-Challenge*, volume 6310 de *Lecture Notes in Computer Science*, pages 105–117. Springer Berlin Heidelberg, 2010.

[4] R. Couturier : *Designing Scientific Applications on GPUs*. Chapman & Hall/CRC Numerical Analysis and Scientific Computing Series, 2013.

[5] G. Perrot : *Algorithmes rapides pour le traitement des images bruitées sur GPU*. Thèse de doctorat, Université de Franche-Comté, Besançon, France, 2014.

[6] S. M. Karabernou et F. Terranti : Real-time fpga implementation of hough transform using gradient and cordic algorithm. *Image Vision Comput.*, 23(11):1009–1017, octobre 2005.

[7] H. T. Ngo, M. Zhang, L. Tao et V. K. Asari : Design of a high performance architecture for real-time enhancement of video stream captured in extremely low lighting environment. *Microprocess. Microsyst.*, 33(4):273–280, juin 2009.

[8] H. T. Ngo, V. K. Asari, M. Z. Zhang et L. Tao : Design of a systolic-pipelined architecture for real-time enhancement of color video stream based on an illuminance-reflectance model. *Integr. VLSI J.*, 41(4):474–488, juillet 2008.

[9] J. Yu, G. Lemieux et C. Eagleston : Vector processing as a soft-core cpu accelerator. *In Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, FPGA '08, pages 222–232, New York, NY, USA, 2008. ACM.

[10] P. Corsonello, S. Perri, P. Zicari et G. Cocorullo : Microprocessor-based fpga im-

plementation of spiht image compression subsystems. *Microprocessors and Microsystems*, 29(6):299–305, 2005.

[11] V. Brost, F. Yang, M. Paindavoine et N. Farrugia : Multiple modular very long instruction word processors based on field programmable gate arrays. *Journal of Electronic Imaging*, 16, 2007.

[12] P. P. Chu : *FPGA Prototyping by VHDL Examples : Xilinx Spartan-3 Version*. Wiley, 2008.

[13] S. Kilts : *ADVANCED FPGA DESIGN, Architecture, implementation, and Optimization*. Wiley Interscience/IEEE PRESS, 2007.

[14] M. S. Prieto et A. R. Allen : A hybrid system for embedded machine vision using fpgas and neural networks. *Mach. Vision Appl.*, 20(6):379–394, octobre 2009.

[15] S. Hussmann et T. H. Ho : A high-speed subpixel edge detector implementation inside a fpga. *Real-Time Imaging*, 9(5):361–368, 2003.

[16] N. Sudha : An area-efficient pipelined array architecture for euclidean distance transformation and its fpga implementation. *In VLSI Design, 2004. Proceedings. 17th International Conference on*, pages 689–692, 2004.

[17] D. Mihhailov, V. Sklyarov, I Skliarova et A Sudnitson : Parallel fpga-based implementation of recursive sorting algorithms. *In Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, pages 121–126, Dec 2010.

[18] Valery Sklyarov : Fpga-based implementation of recursive algorithms. *Microprocessors and Microsystems*, 28(5–6):197 – 211, 2004. Special Issue on FPGAs : Applications and Designs.

[19] L. Zhuo et V. K. Prasanna : Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems. *IEEE Trans. Parallel Distrib. Syst.*, 18(4):433–448, avril 2007.

[20] J. Sun, G. D. Peterson et O. O. Storaasli : High-performance mixed-precision linear solver for fpgas. *IEEE Trans. Comput.*, 57(12):1614–1623, décembre 2008.

[21] D. Chen, J. Cong et P. Pan : Fpga design automation : A survey. *Foundations and Trends in Electronic Design Automation*, 1(3):195–330, November 2006.

[22] R. A. Walker et R. Camposano : *A survey of high-level synthesis systems*. Kluwer Academic Publishers, 1991.

[23] Z. Guo, W. Najjar et B. Buyukkurt : Efficient hardware code generation for fpgas. *ACM Trans. Archit. Code Optim.*, 5(1):6 :1–6 :26, mai 2008.

[24] J. Cong, B. Lui, S. Neuendorffer, J. Noguera, K. Vissers et Z. Zhang : High-level synthesis for fpgas : From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions*, 30(4):473–491, March 2011.

[25] S. Bourdeauducq : Migen, une “boîte à outils” en python. *GNU/Linux magazine France*, 149:15–18, May 2012.

[26] S. Bourdeauducq : *An introduction to Migen.* milkymist, 2013.

[27] K. Benkrid, A. Benkrid et S. Belkacemi : Efficient fpga hardware development : A multi-language approach. *J. Syst. Archit.*, 53(4):184–209, avril 2007.

[28] S. Belkacemi, K. Benkrid et D. Crookes : Hide : a logic based hardware intelligent description environment. *In Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on*, pages 174–180, Dec 2002.

[29] K. Benkrid, D. Crookes, J. Smith et A Benkrid : High level programming for real time fpga based video processing. *In Acoustics, Speech, and Signal Processing, 2000. ICASSP '00. Proceedings. 2000 IEEE International Conference on*, volume 6, pages 3227–3230 vol.6, 2000.

[30] A. U. Irtürk : *GUSTO : General architecture design Utility and Synthesis Tool for Optimization.* Thèse de doctorat, University of California, San Diego, 2009.

[31] D. Crookes, K. Alotaibi, A. Bouridane, P. Donachy et A. Benkrid : An environment for generating fpga architectures for image algebra-based algorithms. *In Image Processing, 1998. ICIP 98. Proceedings. 1998 International Conference on*, pages 990–994 vol.3, Oct 1998.

[32] J. Serrano, M. Cattin, E. Gousiou, E. van der Bij, T. Wostowski, G. Daniluk et M. Lipiński : The white rabbit project. *In Proceedings of the The second International Beam Instrumentation Conference (IBIC)*, pages 948–954, 2013.

[33] K. Benkrid, D. Crookes et A. Benkrid : Towards a general framework for fpga based image processing using hardware skeletons. *Parallel Computing Journal (ParCo), special issue on parallel image and video processing, Elsevier Science*, 28/7-8(1):1141–1154, August 2002.

[34] K. Benkrid et D. Crookes : From application descriptions to hardware in seconds : A logic-based approach to bridging the gap. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(4):420–436, April 2004.

[35] K. Benkrid, D. Crookes, J. Smith et A. Benkrid : High level programming for fpga based image and video processing using hardware skeletons. *In Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, pages 219–226, March 2001.

[36] S. J. Olivieri : *Modular FPGA-Based Software Defined Radio for CubeSats.* Thèse de doctorat, Faculty of the Worcester Polytechnic Institute, 2011.

[37] D. Wright : *FIELD PROGRAMMABLE GATE ARRAY (FPGA) BASED SOFTWARE DEFINED RADIO (SDR) DESIGN.* Thèse de doctorat, Naval Postgraduate School, MONTEREY, CALIFORNIA, 2009.

[38] R. H. L. Stroop : *Enhancing GNU Radio for Run-Time Assembly of FPGA-Based Accelerators.* Thèse de doctorat, Thesis submitted to the Faculty of the Virginia Polytechnic Institute, 2012.

[39] OpenCores. *Wishbone B4 – WISHBONE System-on-Chip (SoC)Interconnection Architecturefor Portable IP Cores*, 2010.

[40] F. Marteau : Pod tutorial. http://www.armadeus.com/wiki/index.php?title=POD_Tutorial.

[41] S. W. Smith : *Digital Signal Processing, A Practical Guide for Engineers and Scientists*. Newnes, 2002.

[42] R. Cariou : *Le traitement du signal RADAR, Détection et interprétation de l'écho radar*. DUNOD, 2012.

[43] K. D. Lee : *Programming Languages, An Active Learning Approach*. Springer Science + Business Media, LLC, 2008.

[44] Micron Technology, Inc. *Mobile Low-Power DDR SDRAM*, 2010.

[45] B. Pan, K. Qian, H. Xie et Anand Asundi : Two-dimensional digital image correlation for in-plane displacement and strain measurement : a review. *Meas. Sci. Technol.*, 20:062001, 2009.

[46] F. Hild et S. Roux : Digital image correlation : from displacement measurement to identification of elastic properties - a review. *Strain*, 42:69–80, 2006.

[47] P. Sandoz, J.-M. Friedt, É. Carry, B.Trolard et J. Garzon Reyes : Frequency domain characterization of tuning-fork mechanical vibrations by vision and digital image processing. *American Journal of Physics*, 77(1):20–26, January 2009.

[48] J-M. Friedt et G. Goavec-Merou : La réception radiofréquence définie par logiciel (software defined radio - sdr). *GNU/Linux magazine France*, 153:04–33, October 2012.

[49] D. Schlichthärle : *Digital Filters - Basics and Design, 2nd ed.* Springer Science, 2011.

[50] U. Meyer-Baese : *Digital Signal Processing with Field Programmable Gate Arrays*. Springer Science, 2001.

[51] W. R. Bennett : Spectra of quantized signals. *Bell System Technical Journal*, 27:446–472, JULY 1948.

[52] B. M. Oliver, J. R. Pierc et C. E. Shannon : The philosophy of pcm. *In Proceedings of the IRE*, volume 36, page 1324–1331, November 1948.

[53] H. Gish et J. N. Pierce : Asymptotically efficient quantizing. *IEEE Transactions on Information Theory*, IT-14(5):676–683, September 1968.

[54] R. M. Gray et D. L. Neuhoff : Quantization. *IEEE Transactions on Information Theory*, IT-44(6):2325–2383, October 1998.

Publications

1 Journaux avec commité de lecture

1. G. Goavec-Merou, N. Chretien, J.-M Friedt, P. Sandoz, G. Martin, M. Lenczner et S. Ballandras : Fast contactless vibrating structure characterization using real time field programmable gate array-based digital signal processing : Demonstrations with a passive wireless acoustic delay line probe and vision. *Review of Scientific Instruments*, 85(1):015109–015109–10, Jan 2014.

2 Conférences

1. R. Couturier, S. Domas, G. Goavec-Merou, M. Favre, M. Lenczner et A Meister : A cost effective AFM setup, combining interferometry and FPGA. *In Thermal, Mechanical and Multi-Physics Simulation and Experiments in Microelectronics and Microsystems (EuroSimE), 2013 14th International Conference on*, pages 1–6, April 2013.
2. G. Goavec-Merou, K. Breshi, G. Martin, S. Ballandras, J. Bernard, C. Droit et J.M. Friedt : Multipurpose use of radiofrequency sources for probing passive wireless sensors and routing digital messages in a wireless sensor network. *In Green Computing and Communications (GreenCom), 2012 IEEE International Conference on*, pages 675–680, Nov 2012.
3. C. Droit, J.-M.Friedt, G. Goavec-Merou, G. Martin, S. Ballandras, K. Breschi Jimenez-Ramirez, J. Bernard et H. Guyennet : Radiofrequency transceiver for probing SAW sensors and communicating through a wireless sensor network. *In The Sixth International Conference on Sensor Technologies and Applications (SENSORCOMM 2012 - August 19 - 24, 2012 - Rome, Italy)*, August 2012.
4. R. Couturier, S. Domas, G. Goavec-Merou, M. Favre, M. Lenczner et A Meister : A new approach based on a least square method for real-time estimation of cantilever array deflections with an FPGA. *In Design, Control and Software Implementation for Distributed MEMS (dMEMS), 2012 Second Workshop on*, pages 30–37, April 2012.

5. Y. Yakoubi, M. Lenczner, G. Goavec-Merou, R. Couturier et J.-M. Friedt : Diffusive Realization of a Lyapunov Equation Solution, and its FPGA Implementation. *In IFAG Conference*, pages 5477–5488, Milano, Italie, 2011.
6. G. Goavec-Merou, Y. Yakoubi, R. Couturier, M. Lenczner, J-M Friedt et F. Wang : FPGA implementation of diffusive realization for a distributed control operator. *In Hardware and Software Implementation and Control of Distributed MEMS (DMEMS), 2010 First Workshop on*, pages 50–55, June 2010.
7. B. Francois, G. Martin, P. Grosclaude, M. Lamothe, G. Goavec-Merou, S. Ballandras et J.-M. Friedt : Fabrication and characterization of a SAW-oscillator-based sensing system including an integrated reciprocal counter and a wireless Zigbee transmission system. *In Ultrasonics Symposium (IUS), 2010 IEEE*, pages 1290–1293, Oct 2010.

3 Articles de vulgarisation scientifique et technique

1. J.-M Friedt et G. Goavec-Mérou : La réception radiofréquence définie par logiciel (software defined radio – SDR). *GNU/Linux Magazine France*, 153:4–33, octobre 2012.
2. J.-M Friedt et G. Goavec-Mérou : Traitement du signal sur système embarqué – application au RADAR à onde continue. *GNU/Linux Magazine France*, 149, mai 2012.
3. G. Goavec-Mérou et J.-M Friedt : Le microcontrôleur stm32 : un coeur ARM cortex-m3. *GNU/Linux Magazine France*, 148, avril 2012.
4. G. Goavec-Mérou et J.-M Friedt : Développement pour iPod touch sous GNU/Linux : application à la communication par liaison bluetooth. *GNU/Linux Magazine France Hors Série*, 51, December-January 2010-2011.
5. G. Goavec-Mérou et J.-M Friedt : Etude d'un système d'exploitation pour microcontrôleur faible consommation (TI msp430) : pilote pour le stockage de masse au format FAT sur carte SD. *GNU/Linux Magazine France Hors Série*, 47, apr-May 2010.
6. J.-M Friedt et G. Goavec-Mérou : Interfaces matérielles et OS libres pour Nintendo DS : Dslinux et RTEMS. *GNU/Linux Magazine France Hors Série*, 43, August 2009.

4 Présentations de vulgarisation scientifique et technique

1. J.-M Friedt et G. Goavec-Mérou : GNURadio as a general purpose digital signal processing environment. *In FOSDEM 2014*, janvier 2014.
2. N. Chrétien, M. Lamothe, G. Goavec-Mérou et J.-M Friedt : Development of libraries for radiofrequency instrumentation using FPGAs. *In RMLL2010*, juillet 2010.
3. G. Goavec-Mérou et J.-M Friedt : Developing for Apple iPod under GNU/Linux. *In RMLL2010*, juillet 2010.

4. G. Goavec-Mérou et J.-M Friedt : Étude d'un système d'exploitation pour microcontrôleur msp430 : développement d'un driver FAT pour TinyOS-2.x. *In RMLL2009*, juillet 2009.
5. J.-M Friedt et G. Goavec-Mérou : Systèmes d'exploitation libres pour Nintendo DS : uclinux et RTEMS. *In RMLL2009*, juillet 2009.
6. G. Goavec-Mérou et J. Boibessot : Xenomai sur cible ARM9 i.MX et mesure non intrusive des performances du système. *In RMLL2009*, juillet 2009.

Résumé

L'utilisation de matrice de portes logiques reconfigurables (FPGA) est une des seules solutions pour traiter des flux de plusieurs 100 MÉchantillons/seconde en temps-réel. Toutefois, ce type de composant présente une grande difficulté de mise en œuvre : au delà d'un type langage spécifique, c'est tout un environnement matériel et une certaine expérience qui sont requis pour obtenir les traitements les plus efficaces. Afin de contourner cette difficulté, de nombreux travaux ont été réalisés dans le but de proposer des solutions qui, partant d'un code écrit dans un langage de haut-niveau, vont produire un code dans un langage dédié aux FPGAs. Nos travaux, suivant l'approche d'assemblage de blocs et en suivant la méthode du *skeleton*, ont visé à mettre en place un logiciel, nommé *CoGen*, permettant, à partir de codes déjà développés et validés, de construire des chaînes de traitements en tenant compte des caractéristiques du FPGA cible, du débit entrant et sortant de chaque bloc pour garantir l'obtention d'une solution la plus adaptée possible aux besoins et contraintes. Les implémentations des blocs de traitements sont soit générés automatiquement soit manuellement. Les entrées-sorties de chaque bloc doivent respecter une norme pour être exploitable dans l'outil. Le développeur doit fournir une description concernant les ressources nécessaires et les limitations du débit de données pouvant être traitées. CoGen fournit à l'utilisateur moins expérimenté une méthode d'assemblage de ces blocs garantissant le synchronisme et cohérence des flux de données ainsi que la capacité à synthétiser le code sur les ressources matérielles accessibles. Cette méthodologie de travail est appliquée à des traitements sur des flux vidéos (seuillage, détection de contours et analyse des modes propres d'un diapason) et sur des flux radio-fréquences (interrogation d'un capteur sans-fils par méthode RADAR, réception d'un flux modulé en fréquence, et finalement implémentation de blocs de bases pour déporter le maximum de traitements en numérique).

Mots-clés : FPGA, temps-réel, traitements vidéo, traitements radio-fréquences, Software Defined Radio (SDR).

Abstract

Using Field Programmable Gate Arrays (FPGA) is one of the very few solution for real time processing data flows of several hundreds of Msamples/second. However, using such components is technically challenging beyond the need to become familiar with a new kind of dedicated description language and ways of describing algorithms, understanding the hardware behaviour is mandatory for implementing efficient processing solutions. In order to circumvent these difficulties, past researches have focused on providing solutions which, starting from a description of an algorithm in a high-abstraction level language, generetes a description appropriate for FPGA configuration. Our contribution, following the strategy of block assembly based on the *skeleton* method, aimed at providing a software environment called CoGen for assembling various implementations of readily available and validated processing blocks. The resulting processing chain is optimized by including FPGA hardware characteristics, and input and output bandwidths of each block in order to provide solution fitting best the requirements and constraints. Each processing block implementation is either generated automatically or manually, but must comply with some constraints in order to be usable by our tool. In addition, each block developer must provide a standardized description of the block including required resources and data processing bandwidth limitations. CoGen then provides to the less experienced user the means to assemble these blocks ensuring synchronism and consistency of data flow as well as the ability to synthesize the processing chain in the available hardware resources. This working method has been applied to video data flow processing (threshold, contour detection and tuning fork eigenmodes analysis) and on radiofrequency data flow (wireless interrogation of sensors through a RADAR system, software processing of a frequency modulated stream, software defined radio).

Keywords : FPGA, real time, video processing, radiofrequency datastream processing, Software Defined Radio (SDR).