Compilation automatique pour les FPGAs

Jean-Baptiste Note

THÈSE DE DOCTORAT DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité INFORMATIQUE

Présentée par
JEAN-BAPTISTE NOTE

pour obtenir le titre de
DOCTEUR DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet :

COMPILATION AUTOMATIQUE POUR LES FPGAS

Soutenue le 31 octobre 2007 devant le jury composé de :

Alain	GREINER	*Président*, Université Paris VI
Jean	VUILLEMIN	*Directeur de thèse*, ÉNS
Gérard	BERRY	*rapporteur*, Esterel Technologies
Marc	POUZET	*rapporteur*, LRI
Jacques	BLANC-TALON	*examinateur*, DGA
Luc	ROBERT	*examinateur*, Realviz
Mary	SHEERAN	*examinatrice*, Chalmers University
Mark	SHAND	*examinateur*, Let It Wave

THÈSE DE DOCTORAT DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité INFORMATIQUE

Présentée par
JEAN-BAPTISTE NOTE

pour obtenir le titre de
DOCTEUR DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet :

COMPILATION AUTOMATIQUE POUR LES FPGAS

Soutenue le 31 octobre 2007 devant le jury composé de :

Alain	GREINER	*Président*, Université Paris VI
Jean	VUILLEMIN	*Directeur de thèse*, ÉNS
Gérard	BERRY	*rapporteur*, Esterel Technologies
Marc	POUZET	*rapporteur*, LRI
Jacques	BLANC-TALON	*examinateur*, DGA
Luc	ROBERT	*examinateur*, Realviz
Mary	SHEERAN	*examinatrice*, Chalmers University
Mark	SHAND	*examinateur*, Let It Wave

Remerciements

Mes premiers remerciements vont à Jean Vuillemin, pour son accueil dans l'équipe Algorithmique Matérielle de l'École Normale, pour ses conseils prodigués au cours de mes travaux, pour tout ce que j'ai appris auprès de lui. Jean a déjà beaucoup construit, et j'ai eu le privilège de construire à ma mesure avec son aide. J'en sors grandi.

Mon travail n'aurait aucune dimension applicative sans Mark Shand. Il est le concepteur de la PAM qui a servi de support à mes expérimentations. Mark est une source inépuisable et recommandable de conseils avisés grâce à son expérience et sa connaissance précise des systèmes matériels et logiciels. Son attention bienveillante et très motivante m'a beaucoup aidé à conclure ce travail.

Mon travail doit aussi beaucoup à Marc Pouzet, qui m'a accompagné et encouragé tout au long de ces quatre années de recherche et de programmation. Je remercie Gérard Berry d'avoir accepté la lourde charge de rapporteur – d'autant plus que son temps de scientifique et d'industriel est précieux. Ses suggestions précises pour la correction du manuscrit m'ont guidées, et je commence seulement à mesurer la portée de ses recommandations bibliographiques.

Je tiens à remercier Alain Greiner qui m'a fait l'honneur de présider le jury de cette thèse. J'ai eu beaucoup de plaisir de travailler, en stage de DEA, dans son laboratoire ASIM. J'y ai rencontré Alix Munier Kordon, qui m'a initié à la recherche.

Les applications liées à cette thèse ont été l'occasion de rencontrer et de travailler avec Luc Robert et Christophe Bernard. À tous les deux je tiens à exprimer ma gratitude pour m'avoir fait découvrir leur domaine. Ces rencontres ont donné un sens à mon travail. Ce n'est pas une petite satisfaction pour un ingénieur de constater que son travail trouve un prolongement dans l'industrie.

Je remercie aussi la *DGA* et l'École Normale Supérieure de m'avoir permis de travailler sur cette thèse. La première m'a soutenu financièrement, la seconde fut le lieu de ma recherche. Jacques Stern bien sûr, mais aussi Joëlle Isnard et l'ensemble du personnel administratif ont été pour moi l'incarnation chaleureuse et efficace de cette vénérable institution.

Si l'ENS est le lieu de ma recherche, l'École est aussi un lieu d'enseignement. J'ai eu le plaisir d'y enseigner quelques bases à des élèves toujours brillants et parfois motivés par les circuits digitaux. Leur rencontre a été une des joies de ces quatre années, qu'ils en soient remerciés.

Les doctorants de l'équipe voisine de cryptographie, Vivien, Nicolas, Sébastien, Malika, ont partagé avec moi repas et discussions plus ou moins scientifiques pendant quatre ans. Leur compagnie a été très agréable et motivante.

Merci enfin à toute ma famille, en particulier Cécile, de m'avoir soutenu au long de ce travail.

Version du 10 octobre 2007.

Table des matières

Table des figures

Liste des tableaux

Et pendant une éternité, il ne cessa de connaître et de ne pas comprendre.
Paul Valéry, L'Ange

1 Introduction

Le traitement automatique de l'information est aujourd'hui capital. Si la loi de Moore permet de déployer des algorithmes mathématiques de plus en plus complexes sur des processeurs de plus en plus puissants, le logiciel n'est pas toujours la meilleure solution pour réaliser un calcul. Les performances d'un processeur séquentiel sont souvent décevantes pour les métriques que sont la consommation d'énergie ou la vitesse de traitement. Les solutions *matérielles* dédiées permettent alors d'améliorer considérablement la vitesse de traitement voire de réduire la consommation électrique. La solution matérielle qui consiste à graver dans le silicium un algorithme donné demande un investissement très important en temps et en argent. Une solution intermédiaire simple peut consister à épauler un processeur standard par un co-processeur spécialisé réalisé sur un circuit reconfigurable.

1.1 Le coprocesseur PAM

La notion de Mémoire Active Programmable – Programmable Active Memory (Pam) est développée par Jean Vuillemin dès 1987 pour répondre à ce problème. Une Pam est un co-processeur générique qui s'appuie sur la technologie des circuits reprogrammables : ceux-ci permettent d'instancier un co-processeur matériel spécialisé pour l'accélération d'une application.

Dans une Pam, la grande souplesse des circuits reprogrammables est mise à profit : pour une application donnée, la programmation du circuit réalise une routine spécialisée – la partie critique du calcul – et la Pam assiste ainsi le processeur central dans l'exécution de l'algorithme.

La technologie des circuits reprogrammables permet d'une part de disposer d'un périphérique universel reconfigurable en fonction des besoins de l'utilisateur, d'autre part de disposer d'un flot de développement qui permet la mise au point rapide de l'accélérateur.

1.2 Description de circuit

La description du circuit réalisé sur le co-processeur fait classiquement appel à la famille des Langages de Description Matérielle – Hardware Description Languages (HDL). Plusieurs choix s'offrent donc à l'utilisateur pour décrire sa routine matérielle.

Les HDLs standards, Verilog [1] et VHDL [2] sont des langages de description à bas niveau des circuits. Leur expressivité reste très limitée. La description synthétisable des circuits dans de tels langages est *structurelle* – une description *comportementale*, non-synthétisable, est aussi possible. En dépit de cette distinction, les utilisateurs de tels langages doivent souvent recourir à des idiosyncrasies propres au synthétiseur pour spécifier finement les détails d'implémentation de leurs circuits sur un Field Programmable Grid Array (FPGA).

Dans le but de rendre plus accessible l'utilisation des FPGAs et d'en élargir ainsi le marché, un écosystème d'outils commerciaux permet aujourd'hui la synthèse directe de code source écrit en C – la *lingua franca* des systèmes logiciels. À l'instar des compilateurs parallélisant, qui permettent d'extraire le parallélisme sous-jacent à une description en C d'un algorithme pour exploiter les unités de traitement Single Instruction, Multiple Data (SIMD) des processeurs modernes, les synthétiseurs C peuvent extraire le parallélisme implicite d'une description séquentielle pour générer des circuits. Le principal défaut d'une telle méthodologie est que le parallélisme intrinsèque d'un circuit est caché sous la forme d'un programme séquentiel. Le concepteur dispose de peu de moyens pour spécifier les détails de son circuit, et en particulier il ne peut qu'espérer que le synthétiseur exploite tout le parallélisme disponible. L'évaluation précise de ces produits – tous commerciaux – n'a pas été conduite en détail pendant cette thèse pour des raisons de disponibilité et de coût des licenses.

Ce tableau sombre serait incomplet sans mentionner Esterel [3], qui lui apporte un peu de couleur. Esterel permet aujourd'hui la synthèse de circuits, avec une sémantique claire. En particulier, Esterel permet la description comportementale des structures de contrôle – une avancée considérable par rapport aux langages purement flot de données.

Par ailleurs Lava ([4], [5]) permet de concevoir des circuit par leur description structurelle dans un cadre fonctionnel particulièrement agréable.

Hors de ces exceptions, le monde des HDLs oscille entre deux extrêmes : un manque d'abstraction pour les HDLs standard d'une part, et des difficultés liées à la nature séquentielle du langage pour la synthèse de haut niveau depuis C d'autre part.

Les problèmes à résoudre sur le flot de conception sont pourtant nombreux, et le besoin d'abstraction fort, en raison de l'augmentation constante de la taille des circuits disponibles. Par ailleurs, aucun de ces outils n'est spécifiquement dédié à la description d'un circuit dans le contexte de son utilisation dans une Pam.

1.3 Contributions

Cette thèse introduit le Design Source Language (DSL), un langage de programmation dédié à la description et au co-développement matériel/logiciel à destination d'un coprocesseur reconfigurable. Dans ce cadre restreint, des mé-

thodes simples et bien définies sont utilisées pour tenter de répondre aux problèmes de la description de circuit.

L'originalité de ce travail repose sur plusieurs points :

- Une vision transversale complète de la chaîne de compilation. DSL permet la description, la simulation, la synthèse et l'exécution des routines programmées. La chaîne de compilation intègre aussi bien des notions de langage de programmation que des détails fins d'implémentation système.
- L'utilisation de la chaîne de compilation sur des applications concrètes, dont deux aux dimensions industrielles. L'usage de l'outil DSL adosse solidement le travail réalisé sur la chaîne de compilation.
- La mise en perspective des applications accélérées par le coprocesseur grâce à la comparaison automatique de leurs performances – en situation d'utilisation réelle – à une implémentation logicielle optimisée.
- Enfin le travail de conception réalisé sur plusieurs algorithmes pour parvenir à leur portage sur le co-processeur Pam est non-trivial. Il a produit des architectures de calcul très performantes au regard de l'état de l'art.

Le positionnement de DSL parmi les langages de description matérielle est original. Sa sémantique suit celle les langages synchrones : DSL prend le parti de ne pas abstraire la sémantique des circuits synchrones – fondamentale pour la réalisation d'un "bon circuit" – qui est clairement exposée au concepteur.

En revanche, DSL assiste le concepteur dans les tâches ingrates. Ainsi, DSL est un langage arithmétique : les variables utilisées sont de type entier, comme en C, ce qui permet une grande simplicité d'utilisation et bon niveau d'abstraction des opérations. De même, DSL automatise les transformations de haut niveau sur le circuit que sont le dimensionnement du chemin de données et la retemporisation.

Mais son comportement reste bien défini, et l'intégration d'annotations au sein même du langage permet au concepteur de circuit expérimenté d'orienter ou de préempter proprement les choix du compilateur.

La comparaison avec Esterel mérite sans doute un peu d'attention [6]. Esterel dispose d'un système de type fin et riche, là où DSL a fait le choix de la simplicité avec un typage uniforme ; en revanche DSL autorise la récursion, ce que ne permet pas Esterel. DSL est orienté flot de données, et ne propose aucune construction qui facilite les structure de contrôle – c'est au contraire un point fort d'Esterel. Esterel ne propose pas la retemporisation automatique de chemin de données. Enfin, les deux langages proposent le dimensionnement automatique et fin des types entiers pour la synthèse des opérateurs arithmétiques au bit près, même si, nous le verrons, la méthode retenue pour y parvenir diffère.

Pour finir, une interface efficace entre le circuit synchrone sur la Pam et le processeur de la station de travail est réalisée *automatiquement*.

Le résultat de ce travail est donc un système de prototypage et d'accélération matérielle automatique pour les applications décrites en DSL.

1.4 Contenu

La thèse est constituée de deux parties. La première est une description du système complet de compilation de DSL ; elle s'étend sur les chapitres 2, 3 et 4 qui reflètent l'organisation stratifiée de la chaîne de compilation d'un circuit en DSL : l'écriture, la synthèse et l'intégration système.

Le chapitre 2 se concentre sur la description de DSL en tant que *langage de programmation* : sa sémantique, sa syntaxe, et les algorithmes de transformation de circuit qui y sont intégrés. Les méthodes et algorithmes décrits sont implémentés en Jazz, mais sont généralisables à d'autres langages et indépendants de la technologie de circuit reconfigurable sous-jacente.

Le chapitre 3 contient la description détaillée de la librairie arithmétique dorsale utilisée pour synthétiser DSL sur un FPGA spécifique. Un soin particulier a été apporté à la performance des opérateurs réalisés. La méthode en usage est générale, même si le détail du langage d'implémentation et des optimisations décrites sont particulières à la technologie FPGA ciblée.

Le chapitre 4 enfin détaille l'interconnexion entre la puce FPGA et la station-hôte au travers d'un bus spécifique. L'ensemble des interfaces matérielles et routines systèmes nécessaires à une communication optimale est soigneusement mis en évidence.

La deuxième partie est constituée du long chapitre 5 ; elle met aux prises la description théorique de DSL avec la réalité de son utilisation pour décrire, simuler, synthétiser et exécuter cinq algorithmes de traitement vidéo en temps réel.

2

Conception de circuits

Dans ce chapitre nous introduisons un exemple simple pour illustrer la démarche du concepteur de circuit. À partir de la description algorithmique de haut niveau spécifiant le calcul à réaliser, le concepteur en choisit d'abord une implémentation dont il fait ensuite la traduction jusqu'au circuit final. Ces deux étapes sont clairement distinguées sur l'exemple. Notre Design Source Language (DSL) et son flot de compilation constituent un *assistant* qui épaule le concepteur au cours de la phase de traduction en circuit. La méthodologie de la conception de circuit gouverne les caractéristiques de DSL :

- DSL est un langage fonctionnel particulièrement adapté à la description et la synthèse automatique de circuits efficaces ;
- DSL unifie dans une même syntaxe spécifications de haut niveau et annotations d'implémentation ;
- DSL, guidé par les annotations manuelles, automatise certaines étapes de la conception.

Une fois les caractéristiques du flot de compilation DSL détaillées, les algorithmes sous-jacents sont décrits en détail et comparés à l'état de l'art.

2.1 Compression réversible d'un signal

2.1.1 Compression en ligne sans perte

Notre circuit d'exemple est un compresseur/décompresseur de signal linéaire. La fonction de ce circuit est de compresser/décompresser un flux de données en temps réel et *sans pertes*. Il s'agit d'un exemple d'école mais qui illustre précisément l'ensemble des problématiques qui se posent lors du passage de l'algorithme au circuit.

2.1.1.1 Principe

Le signal est échantillonné sur 8 bits pour former la suite des échantillons $(i_n)_{n\in\mathbb{N}}$ en entrée du compresseur. Le compresseur utilise un codage différentiel qui consiste à calculer la différence entre deux échantillons consécutifs i_{n-1} et i_n :

$$d_n = i_n - i_{n-1}$$

Les échantillons consécutifs sont présumés proches ($|d_n| \simeq 0$), et le codeur du compresseur tente d'exploiter ce biais :

- Si la différence d_n est suffisamment proche de zéro, un code spécial est utilisé pour la représenter. La longueur du code de sortie est alors inférieure à 8 bits : la donnée est compressée.
- Si en revanche d_n s'éloigne trop de zéro, la valeur du pixel entrant est reproduite, et un bit de contrôle supplémentaire signale l'événement dans le flux de sortie. La donnée est effectivement décompressée sur 9 bits.

Afin d'évaluer l'efficacité de la compression, le quotient de compression [7] d'un flux de données f est défini comme le rapport entre la longueur en bits de f et la longueur de son image par le compresseur $C(f)$:

$$Q(f) = \frac{|f|}{|C(f)|}$$

Si $Q(f) > 1$ le flux f est effectivement compressé ; si $Q(f) < 1$, le flux est au contraire décompressé.

2.1.1.2 Spécification du compresseur

Le circuit de compression accepte 8 bits de données par cycle en entrée. Il délivre 16 bits par cycle en sortie, plus un bit dit *bit d'enable* qui indique quand la donnée en sortie est signifiante.

Comme pour tout compresseur (hors de l'identité) le code de certaines sources est plus long que l'original : c'est la raison pour laquelle la sortie du compresseur est nécessairement plus large que son entrée. En l'occurence, dans le pire des cas, notre compresseur donne 9 bits de code pour 8 bits en entrée.

Le calcul du code de sortie se décompose en trois étapes :

- la transformée différentielle $T : \mathbb{B}^8 \to \mathbb{B} \times \mathbb{B}^8$;
- le codeur proprement dit $C : \mathbb{B} \times \mathbb{B}^8 \to \mathbb{B}^9$ code la distribution en sortie de T suivant un code de longueur variable ;
- le normalisateur par bloc $N : \mathbb{B}^9 \to \mathbb{B} \times \mathbb{B}^{16}$ réarrange les codes de longueur variable en sortie de C en blocs de longueur fixe sur 16 bits et génère l'*enable* de sortie.

Différentielle Soit $I = (i_n)_{n\in\mathbb{N}}$ la suite des valeurs des pixels présentés en entrée du compresseur aux temps $n \in \mathbb{N}$.

Le signal transformé par T est la suite $(u_n, t_n)_{n\in\mathbb{N}}$ définie par :

$$\begin{cases} d_0 = i_0 \\ d_n = i_n - i_{n-1} \\ |d_n| > 11 \Rightarrow (u_n, t_n) = (1, i_n) \\ |d_n| \leq 11 \Rightarrow (u_n, t_n) = (0, d_n) \end{cases}$$

La sortie (u_n, t_n) a donc la signification suivante :

– si $u_n = 0$, alors $t_n \in [-11, 11]$ représente effectivement la différentielle entre le pixel entrant i_n et le pixel précédent i_{n-1} ;
– si $u_n = 1$ alors t_n n'est autre que i_n, la valeur du pixel entrant.

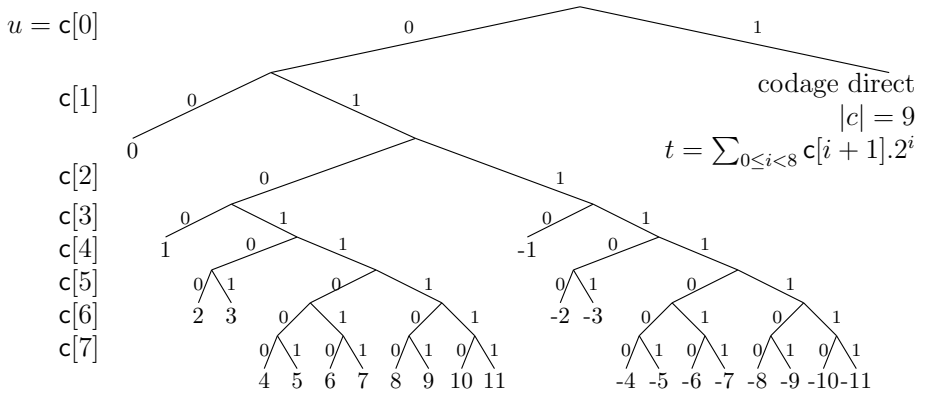


FIG. 2.1 – Arbre du code d'Élias

Codeur Le codeur reçoit la suite $(u_n, t_n)_{n\in\mathbb{N}}$ en provenance de la différentielle et la récrit suivant un code de longueur variable en $(c_n)_{n\in\mathbb{N}}$. Soit $\mathsf{c_n}$ (resp. $\mathsf{u_n}$, $\mathsf{t_n}$) le tableau des bits du code c_n (resp. u_n, t_n). Pour l'entier t_n, il s'agit des bits de sa représentation binaire en complément à deux. Le code c_n est formé de la manière suivante.

Lorsque $u_n = 1$, le code de sortie n'est autre que la valeur directe du pixel t_n, préfixée par le bit u_n qui signale la décompression :

$$\begin{array}{rcl} \mathsf{c_n}[0] & = & 1 \\ \mathsf{c_n}[i] & = & \mathsf{t_n}[i-1] \end{array} \tag{2.1}$$

Le pixel sortant est alors effectivement *décompressé* sur 9 bits. Dans la représentation de l'arbre du code en figure 2.1, $u_n = 1$ situe le code dans la partie droite de l'arbre.

Lorsque $u_n = 0$, la différentielle t_n du pixel est effectivement compressée sur 2, 4, 6 ou 8 bits grâce à une variation sur le code d'Élias [8].

Le code utilisé pour les différentielles de pixels compressées est alors un peu plus complexe :

$$\begin{array}{rcl} t = 0 & \Longrightarrow & \mathsf{c} = 00 \\ 0 < |t| \leq 1 & \Longrightarrow & \mathsf{c} = 01\mathsf{t}[1]0 \\ 1 < |t| \leq 3 & \Longrightarrow & \mathsf{c} = 01\mathsf{t}[2]10\mathsf{t}[0] \\ 3 < |t| \leq 11 & \Longrightarrow & \mathsf{c} = 01\mathsf{t}[3]11\mathsf{t}[2]\mathsf{t}[1]\mathsf{t}[0] \end{array} \tag{2.2}$$

$u = 0$ situe le code dans la branche gauche de l'arbre du code de la figure 2.1. Dans ce cas, le code c du couple $(0, t)$ est formé de la suite des bits rencontrés lors d'une descente dans la branche gauche de l'arbre vers la feuille étiquetée par t.

Le décodage d'un mot de code c vers le couple original (u, t) est obtenu par une descente dans l'arbre en empruntant les arêtes correspondant aux valeurs successives $(\mathsf{c}[i])_{0\leq i<\langle \mathsf{c}\rangle}$ des bits de c.

La mise sous forme d'arbre du pseudo-code d'Élias utilisé assure qu'il s'agit bien d'un code préfixe. Ainsi, une suite de bits constituée par la concaténation de mots du code peut être déchiffrée sans ambiguïté. Heureuse propriété, puisque c'est justement le rôle de la fonction suivante que de procéder à la concaténation des mots en sortie du codeur.

Normalisateur par blocs Les codes $(c_n)_{n\in\mathbb{N}}$ produits en sortie du codeur sont de longueur $\langle \mathsf{c_n} \rangle$ variable entre 2 et 9 bits. La fonction du normalisateur par blocs est d'assembler les mots de code de longueur variable pour constituer des blocs de longueur fixe sur 16 bits en sortie du compresseur.

Le normalisateur procède simplement en accumulant les données de la suite des codes $(c_n)_{n\in\mathbb{N}}$ jusqu'à atteindre ou dépasser 16 bits accumulés dans le tampon. Un mot complet sur 16 bits b_n est alors émis en sortie et l'émission indiquée par le bit d'activation e_n. L'accumulateur conserve les éventuels bits en excès, qui seront retirés du tampon lors de l'émission du mot suivant.

Par la suite l'opération de concaténation des mots binaires est notée $\odot$; la longueur d'un mote binaire m est notée $|m|$.

Soit $(b_n)_{n\in\mathbb{N}}$ la suite des mots en sortie du normalisateur, de longueur 16 ou zéro bits, formés par le normalisateur. La suite en sortie $(b_n)_{n\in\mathbb{N}}$ est définie de manière unique par les deux propriétés qui suivent.

La première est une propriété de conservation de l'information. La suite des blocs produits par le normalisateur est un préfixe de la suite des codes acceptés en entrée :

$$\forall n \in \mathbb{N}, \bigodot_{k=0}^{n} b_k \sqsubseteq \bigodot_{k=0}^{n} c_k$$

De manière équivalente, il existe un mot binaire de reste r_n défini de manière unique et tel que :

$$\forall n \in \mathbb{N}, \left(\bigodot_{k=0}^{n} b_k \right) \odot r_n = \bigodot_{k=0}^{n} c_k \tag{2.3}$$

La seconde propriété exprime que le normalisateur présente sur sa sortie un code b_n aussitôt que possible, de telle sorte que le reste r_n conserve toujours une longueur strictement inférieure à 16 bits :

$$\forall n \in \mathbb{N}, |l_n| - \left| \bigodot_{k=0}^{n} b_k \right| = |r_n| < 16$$

Plus spécifiquement, r_n apparaît comme le reste dans la division par 16 de la longueur accumulée des codes $|\bigodot_{k=0}^{n} c_k|$.

2.1.1.3 Résultats de compression

Les propriétés du code vues en 2.1.1.2 permettent de conclure que le quotient de (dé)compression minimal est $Q_{\min} = 8/9$. Il est atteint par exemple pour la suite alternante $i_n = 50 \times (i \bmod 2)$.

Le quotient de compression maximal est de 4, il est atteint par exemple pour la suite $(i_n)_{n\in\mathbb{N}}$ constante nulle.

Format $L \times H$	Entrée (B) $\lvert f \rvert$	Sortie (B) $\lvert C(f) \rvert$	Compression $Q(f)$
720×576	45619200	39461718	1.156
616×360	783367200	375988528	2.083
640×480	4578969600	1854740628	2.468

TAB. 2.1 – Quotient de compression pour divers flux vidéos

Le quotient de compression moyen varie suivant la nature du signal d'entrée. L'efficacité du compresseur/décompresseur a été testée sur différents flux vidéos. Chacune des images du flux testé est codée en niveaux de gris à 8 bits par pixel. Chacun des pixels de l'image fournit un échantillon en entrée du compresseur. Leur ordre dans le temps suit le balayage de l'image en *raster-scan*, c'est-à-dire l'ordre donné par le sens de lecture, de gauche à droite et de haut en bas.

En ordre raster-scan, deux pixels consécutifs dans le flux de données sont généralement (sauf au retour de ligne) des pixels proches dans l'espace. Par conséquent, on peut espérer que les hypothèses nécessaires à la compression effective du signal sont réunies.

Les quotients de compression obtenus pour différents flux vidéos sont présentés en table 2.1.

La première entrée du tableau mesure la compression sur une petite séquence vidéo au format Phase Alterned Line (PAL) tirée des séquences d'essai du Video Quality Expert Group (VQEG)[1]. Même pour cette séquence complexe, dont le désentrelaçage sans artifice donne lieu à un bruit spatial important, le quotient de compression est déjà légèrement supérieur à 1.

La deuxième entrée présente le résultat de la compression d'un extrait du dessin animé "Le Roi Lion"[2]. Le niveau de détail d'un dessin animé est beaucoup plus faible que celui d'images naturelles. Par exemple, beaucoup d'aplats de couleur presqu'uniformes sont présents. La fréquence spatiale plus basse se traduit par un quotient de compression supérieur.

La troisième et dernière entrée du tableau présente le résultat de la compression d'une vidéo d'animation[3]. Sur ces images synthétiques, le quotient de compression est encore amélioré – pour les mêmes raisons, à savoir une diminution des détails des images de la séquence.

Néanmoins ces quotients de compression sont ridicules en comparaison de ceux obtenus à l'aide de *codecs* spécialisés. De tels compresseurs adaptés au traitement de flux vidéos : d'une part ils tirent mieux partie de la structure des données, notamment de la cohérence temporelle du flux d'images, d'autre part ils éliminent les informations qui ne sont pas perceptibles (et entraînent en conséquence des pertes de données).

2.1.1.4 Spécification du décompresseur

La structure du décompresseur est symétrique à celle du compresseur : un premier étage accumule les blocs en provenance du compresseur pour les restructurer en mots du code d'Élias. Le deuxième étage est chargé de décoder

[1] http://www.its.bldrdoc.gov/vqeg/
[2] http://www.lionking.org/movies/
[3] http://www.jefflew.com/animation.html

les mots du code vers leur valeur initiale ; le dernier étage reconstitue le flux de données initial par intégration.

Ainsi le calcul du pixel décompressé se décompose en trois étapes :

- le tampon d'entrée $N^{-1} : \mathbb{B} \times \mathbb{B}^{16} \to \mathbb{B} \times \mathbb{B}^9$ est une file First-In First-Out (FIFO) qui accumule les mots de 16 bits en provenance du codeur et présente sur sa sortie, dans l'ordre d'arrivée, les mots de code complets successifs, sous un signal booléen d'activation ;
- le décodeur $C^{-1} : \mathbb{B} \times \mathbb{B}^9 \to \mathbb{B} \times \mathbb{B}^9$ décode le flux de mots codés présenté par N^{-1} ;
- la transformée inverse $T^{-1} : \mathbb{B} \times \mathbb{B}^9 \to \mathbb{B} \times \mathbb{B}^8$ enfin inverse l'étape de transformation différentielle T pour retrouver la valeur du pixel initial.

File d'entrée Le flux compressé entre dans le décompresseur par mots de 16 bits. La fonction de la file d'entrée est de séparer la suite continue des bits en mots du code. La file accumule les mots de 16 bits jusqu'à l'obtention d'un mot complet du code qui est soustrait de la file et présenté sur sa sortie.

Les mots du code sont de longueur variable. Par conséquent, le mot du code présenté en sortie est aligné sur neuf bits – qui correspondent aux mots du code les plus longs – plus un bit d'*enable*, qui permet de représenter la présence ou l'absence de mot, puisque le code ne contient pas de représentation du mot vide.

Le tampon peut recevoir jusqu'à 16 bits par cycle en entrée et évacue au maximum neuf bits par cycle en sortie. Il est par conséquent impossible de borner sa taille *a priori* pour un flux de données quelconque en entrée.

Toutefois, dans le cas où, sur un réseau synchrone, le décompresseur est relié directement au compresseur, il est possible de placer une borne supérieure de 81 bits sur la longueur du tampon. La démonstration de cette propriété est l'occasion de formaliser proprement le comportement du tampon.

Dans le cas où le décompresseur est relié directement au compresseur, le flux en entrée du tampon n'est autre que le flux $(b_n)_{n\in\mathbb{N}}$ en sortie du normalisateur. On peut par conséquent reprendre les expressions présentées en 2.1.1.2 :

$$\forall n \in \mathbb{N}, \left(\bigodot_{k=0}^{n} b_k\right) \odot r_n = \bigodot_{k=0}^{n} c_k$$

Supposons que $n \geq 8$, alors on peut distinguer les huit derniers codes du flux :

$$\forall n \in \mathbb{N}, n \geq 8 \Rightarrow \left(\bigodot_{k=0}^{n} b_k\right) \odot r_n = \left(\bigodot_{k=0}^{n-8} c_k\right) \odot \left(\bigodot_{k=n-7}^{n} c_k\right) \tag{2.4}$$

Puis on passe aux longueurs :

$$\forall n \in \mathbb{N}, n \geq 8 \Rightarrow \left|\bigodot_{k=0}^{n} b_k\right| + |r_n| = \left|\bigodot_{k=0}^{n-8} c_k\right| + \left|\bigodot_{k=n-7}^{n} c_k\right|$$

Or, $\forall n \in \mathbb{N}, |c_n| \geq 2 \Rightarrow \left|\sum_{k=n-7}^{n} c_k\right| \geq 16$, par conséquent ces huit mots ont une longueur cumulée strictement plus grande que celle du reste r_n, limité à 15 bits. Par conséquent :

$$\forall n \in \mathbb{N}, n > 7 \Rightarrow \left| \bigodot_{k=0}^{n} b_k \right| = \left| \bigodot_{k=0}^{n-8} c_k \right| + \left(\left| \bigodot_{k=n-7}^{n} c_k \right| - |r_n| \right) > \left| \bigodot_{k=0}^{n-8} c_k \right| \quad (2.5)$$

La conjonction de l'inégalité 2.5 entre les longueurs des termes $\bigodot_{k=0}^{n} b_k$ et $\bigodot_{k=0}^{n-8} c_k$, et de la relation 2.4 permet d'amener la relation préfixe suivante :

$$\forall n \in \mathbb{N}, n > 7 \Rightarrow \bigodot_{k=0}^{n-8} c_k \sqsubset \bigodot_{k=0}^{n} b_k \quad (2.6)$$

Finalement, la suite des blocs reçus par la file d'entrée du décompresseur vérifie, par combinaison à l'équation (2.3), l'inéquation suivante :

$$\forall n \in \mathbb{N}, n > 7 \Rightarrow \bigodot_{k=0}^{n-8} c_k \sqsubset \bigodot_{k=0}^{n} b_k \sqsubseteq \bigodot_{k=0}^{n} c_k \quad (2.7)$$

Elle permet d'introduire le reste r'_n qui vérifie :

$$\forall n \in \mathbb{N}, n > 7 \Rightarrow \bigodot_{k=0}^{n-8} c_k \odot r'_n \odot r_n = \bigodot_{k=0}^{n} b_k \odot r_n = \bigodot_{k=0}^{n} c_k \quad (2.8)$$

Cette relation spécifie en particulier que la concaténation des blocs jusqu'au cycle n contient au moins tous les mots de code, sans coupure, jusqu'au cycle $n-8$. Par conséquent, la file d'entrée du décompresseur placé directement en série avec le décompresseur dispose de toutes les informations nécessaires pour émettre, au cycle n, le mot de code c_{n-8}.

Une implémentation *a minima* de la file d'entrée du décompresseur contient donc en permanence le reste r'_n, et se contente de reproduire la suite c_n avec une latence de huit cycles. Or le reste r'_n vérifie :

$$\forall n \in \mathbb{N}, n > 7 \Rightarrow r'_n \sqsubseteq \bigodot_{k=n-7}^{n} c_k$$

Cette propriété garantit que la file d'attente ne contient jamais plus de huit mots de code ; par conséquent sa longueur est bien bornée par $8 * \langle \mathsf{c} \rangle_{\max} = 81$ bits.

Soit maintenant une implémentation quelconque de la file d'entrée, qui produit la suite des codes $(c'_n)_{n\in\mathbb{N}}$. Cette suite peut-être différente de la suite originale $(c_n)_{n\in\mathbb{N}}$ par insertion de mots de code de longueur nulle. S'il est difficile de spécifier explicitement et simplement un comportement précis de la file d'entrée, il est facile de lui imposer d'être meilleure que l'implémentation la plus simple à latence constante, c'est-à-dire que sa suite de sortie vérifie :

$$\forall n \in \mathbb{N}, \bigodot_{k=0}^{n-8} c_k \sqsubseteq \bigodot_{k=0}^{n} c'_k$$

Comme il est possible de construire des flux de données pour lesquels la latence de 8 cycles d'horloge entre le flux des codes originaux et le flux des codes en sortie du tampon du décompresseur est nécessairement atteinte, pour toute implémentation de la file, la borne de 81 bits de mémoire n'est en fait pas améliorable.

Décodeur Le décodeur trouve sur son entrée la suite $(c'_n)_{n\in\mathbb{N}}$ des codes en sortie de la file d'entrée et les bits d'activation $(e'_n)_{n\in\mathbb{N}}$ correspondants. Sa fonction est d'effectuer le décodage des mots du code décrit en 2.1.1.2 pour produire en sortie la suite $(u'_n, t'_n)_{n\in\mathbb{N}}$ soumise aux mêmes bits d'activation.

Elle procède simplement en descendant dans l'arbre de décodage présenté sur la figure 2.1 à l'aide des bits de c_n. Dans le cas d'un mot de code de longueur inférieure à 9 bits, les bits de bourrage sont ignorés.

Intégrateur L'intégrateur réalise la fonction inverse de la différentielle 2.1.1.2, sous la condition d'activation de l'*enable* d'entrée $(e'_n)_{n\in\mathbb{N}}$. Il dispose en entrée des signaux $(u'_n, t'_n)_{n\in\mathbb{N}}$ et $(e'_n)_{n\in\mathbb{N}}$ qui permettent de recomposer la suite $(i'_n)_{n\in\mathbb{N}}$:

$$\begin{aligned} e'_n = 0 &\implies i_n = i_n - 1 \\ e'_n = 1 \wedge u'_n = 1 &\implies i_n = t_n \\ e'_n = 1 \wedge u'_n = 0 &\implies i_n = i_{n-1} + t_n \end{aligned}$$

La suite en sortie $(i'_n)_{n\in\mathbb{N}}$ est soumise au même signal d'activation $(e'_n)_{n\in\mathbb{N}}$.

2.1.2 De l'algorithme au graphe des opérateurs

La spécification de haut niveau ne précise pas comment réaliser les calculs. C'est le rôle du concepteur de circuit que de construire, à partir de cette spécification, une architecture systolique qui réalise le calcul spécifié.

Le travail d'implémentation est ici détaillé pour le compresseur spécifié en 2.1.1.2. La structuration du calcul en trois étapes distinctes guide l'implémentation en trois modules.

2.1.2.1 Transformeur

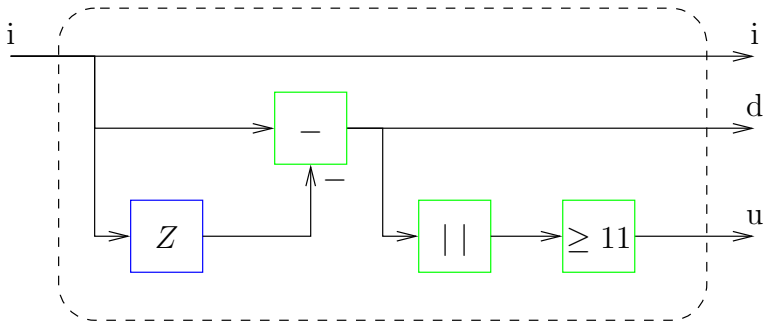


FIG. 2.2 – Transformeur du circuit de compression sans perte

Le circuit du transformeur de pixel a une variable d'entrée i qui porte au cours du temps les valeurs de la suite $(i_n)_{n\in\mathbb{N}}$. Il présente en sortie trois variables, u, d et i.

Le circuit transmet directement sur sa sortie la valeur du pixel entrant. Il calcule la valeur différentielle du pixel courant qui est exposée en sortie et

utilisée pour calculer la variable indicatrice u grâce à une valeur absolue et un comparateur.

Le circuit résultant, très simple, est présenté en figure 2.2.

2.1.2.2 Codeur d'Élias

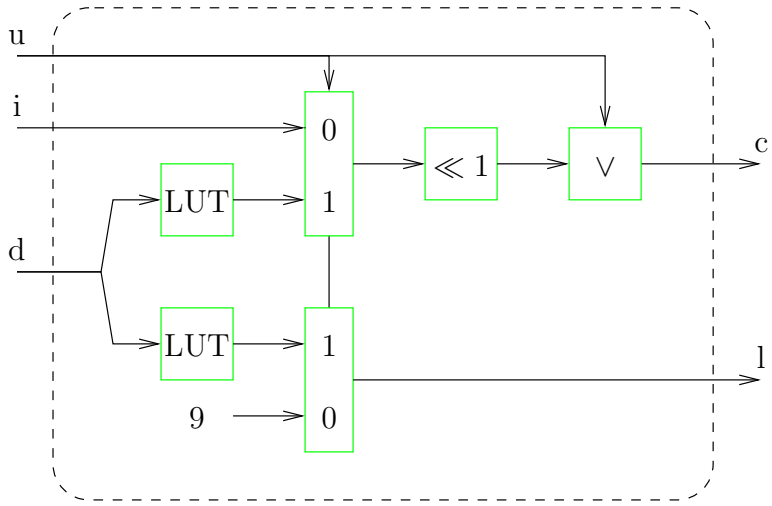


FIG. 2.3 – Codeur d'Élias du circuit de compression sans perte

Le codeur permet de réaliser le calcul effectif de la fonction de codage décrite en 2.1.1.2.

En entrée du codeur on trouve les variables u, i et d qui portent les valeurs respectives de $(u_n)_{n\in\mathbb{N}}$, $(i_n)_{n\in\mathbb{N}}$ et $(d_n)_{n\in\mathbb{N}}$.

Le code c_n ainsi que sa longueur en bits $|\mathsf{c_n}|$ sont générés en sortie, et présentés respectivement sur les variables c et l.

La variable indicatrice u détermine au plus haut niveau le code et la longueur présentés en sortie du codeur :

- Lorsque u vaut 0, c'est la valeur d qu'il convient de coder sur la branche droite de l'arbre figuré en 2.1. La représentation binaire de d est limitée à 5 bits : le calcul du code spécifique d'Élias est explicité en table pour calculer directement c_n par recherche dans un tableau à 2^5 entrées. On réalise la même opération de tabulation explicite pour le calcul de la longueur du code $|\mathsf{c_n}|$.
- Lorsque u vaut 1, on code directement la valeur du pixel entrant et la longueur du code est de 9 bits.

La sélection d'une branche de l'alternative est réalisée par un multiplexeur. La concaténation du bit u en tête du code de sortie est factorisée après le multiplexeur.

Le circuit réalisant cette fonction est présenté en figure 2.3.

2.1.2.3 Normalisateur par blocs

Le normalisateur par bloc reçoit en entrée la suite $(c_n, l_n)_{n\in\mathbb{N}}$ sur les variables c et l. Il délivre les mots de 16 bits sur sa sortie b soumise à l'indicateur de

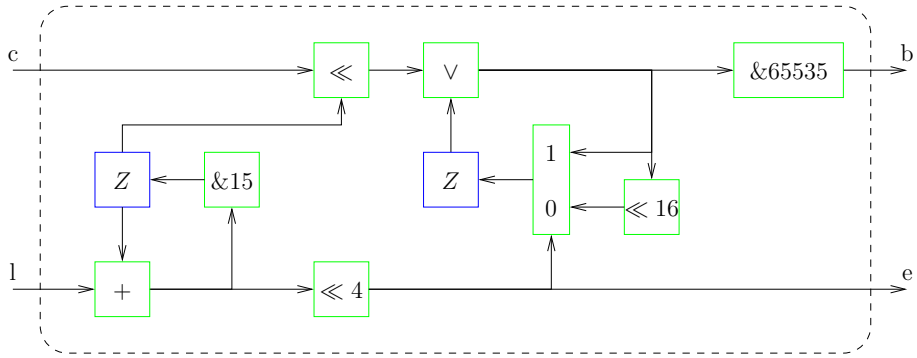


FIG. 2.4 – Normalisateur par blocs du circuit de compression sans perte

présence e.

Le normalisateur par bloc est simplement un accumulateur qui mémorise d'un cycle sur l'autre le reste de la différence r_n entre bits accumulés en entrée et bits déjà présentés en sortie dans un registre.

L'expression de la longueur de r_n comme le reste d'une division par 16 en 2.1.1.2 signifie pratiquement que la mémoire dédiée à r_n est limitée à 15 bits.

Le contrôle de l'accumulateur est réalisé à l'aide d'un registre qui contient $|r_n|$, le nombre de bits présents dans l'accumulateur. Il est incrémenté de l_n à chaque cycle, et le débordement de cette variable au-dessus de 16 bits entraîne le positionnement de l'indicateur de présence e et la délivrance d'un mot complet de 16 bits sur b

L'accumulateur réagit suivant la variable de contrôle : il ajoute en permanence au tampon les données en entrée du circuit, mais en élimine 16 bits à l'échéance de l'émission d'un mot.

2.1.2.4 Compresseur

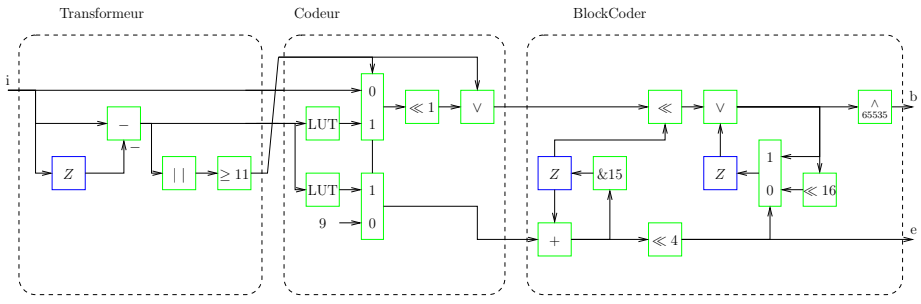


FIG. 2.5 – Circuit de compression sans perte

Le circuit de compression final est représenté en 2.5.

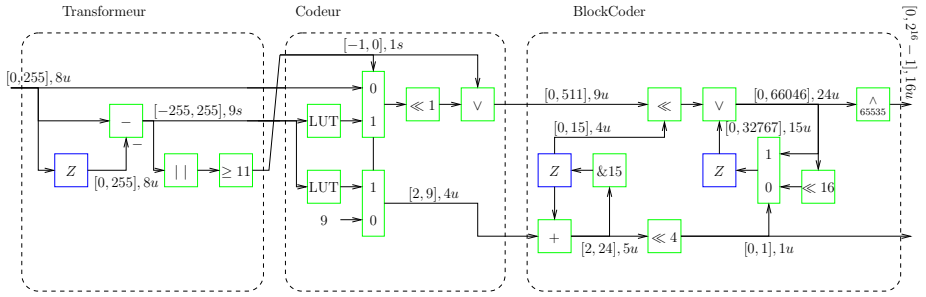


FIG. 2.6 – Bitsizing du circuit de compression sans perte

2.1.3 Du graphe au circuit

2.1.3.1 Intervalle de valeur des variables

Dans le graphe des calculs présenté en figure 2.5, les valeurs portées par les arêtes du graphe sont des entiers correspondant aux valeurs *exactes* des suites calculées par chacun des opérateurs. Or de tels entiers – dont l'intervalle de valeurs n'est pas spécifié et dont la précision est dynamique – ne sont pas facilement synthétisables en circuits. La première étape de notre travail va donc consister à limiter l'intervalle des valeurs prises par le flot des valeurs sur chacune des arêtes afin de circonscrire la précision des valeurs entières à un intervalle fini et statique.

Nous appelons cette étape de la réalisation du circuit le *bitsizing* des variables. La figure 2.6 présente l'ensemble du circuit de compression où chacune des arêtes est annotée par l'intervalle des valeurs qu'elle porte.

2.1.3.2 Mise du circuit sous enable

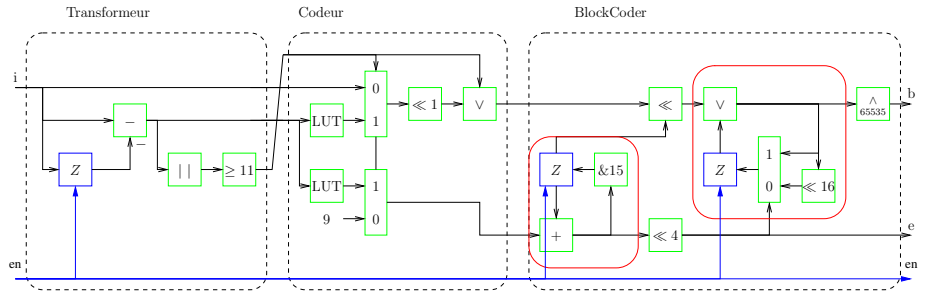


FIG. 2.7 – Mise sous *enable* du circuit de compression sans perte

L'introduction d'un signal *enable* est particulièrement important dans un langage de description de circuit. En Esterel, la construction spécifique du *weak suspend* [3] permet de réaliser un signal d'*enable*. Il y a deux raisons pour l'utilisation du signal d'*enable*.

Clock Gating La première raison est d'ordre technologique, dans le monde des ASICs et dans une moindre mesure dans la conception de circuits sur FPGA.

Le signal d'*enable* permet de commander précisément l'activation des bascules d'une partie du circuit. Ainsi, lorsque l'*enable* est désactivé, les bascules du circuit sont effectivement mises en veille et la consommation électrique est réduite à sa seule composante statique. Cette technique est appellée *clock gating*. Elle est d'autant plus centrale dans les circuits actuels que la consommation électrique est une préoccupation majeure.

Contrôle de flux La deuxième raison est d'ordre méthodologique.

Dans la réalité de l'utilisation d'un circuit il est très difficile de l'alimenter avec un flux de données continu. En effet les données à destination et en provenance du circuit circulent sur des bus asynchrones (ceux utilisés au cours de cette thèse seront étudiés en détail au chapitre 4) par rapport au circuit. En outre, dans une station de travail les bus sont partagés entre plusieurs composants du système qui peuvent en demander l'accès de manière concurrente.

Pour assurer la performance des bus sur des échelles temporelles très fines, de l'ordre de la dizaine de nanosecondes pour les circuits fonctionnant à une centaine de MHz, il est nécessaire de concevoir l'ensemble du matériel et du système d'exploitation autour de cet objectif : seuls des systèmes dits *temps réels* très spécifiques sont en mesure d'assurer ce type de garanties.

Dans le cas qui est le nôtre d'une station de travail et d'un système d'exploitation généraliste, il est impossible d'assurer une telle finesse de contrôle des entrées-sorties du circuit. Il est par conséquent nécessaire de produire un circuit capable de gérer les interruptions du flux de données en entrée.

L'ajout de cette capacité à notre petit circuit est une opération très simple qui consiste à ajouter au circuit un signal d'activation général dit signal d'*enable*. L'*enable* est actif lorsque des données fraîches sont présentes en entrée, il est inactif lorsque le flux de données est interrompu.

Ce signal est figuré sur le circuit de la figure 2.7. Dans le cas du circuit de compression, le signal d'*enable* en entrée contrôle l'activation de l'ensemble des registres du circuit. Ainsi, dès que les données viennent à manquer en entrée du circuit, son état ne change pas, il est effectivement mis en veille.

2.1.3.3 Retemporisation du circuit

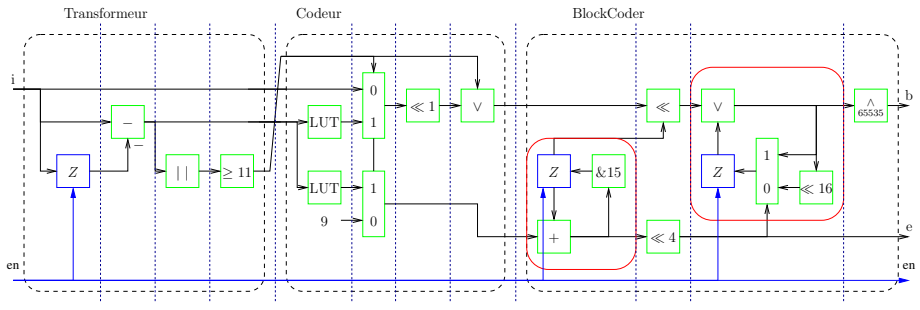


FIG. 2.8 – Retemporisation du circuit de compression sans perte

Le circuit de la figure 2.5 comporte un chemin critique ininterrompu de l'entrée i jusqu'à sa sortie b.

Pour obtenir une fréquence de fonctionnement maximale du circuit, il est nécessaire de briser ce chemin critique par l'insertion de registres. L'insertion de tels registres doit répondre à des critères très stricts de synchronisation des données [9] pour conserver la nature opératoire du circuit.

Le travail de retemporisation consiste à isoler les boucles dans le graphe puis à y insérer des *barrières de retemporisation*. L'intersection de chacune de ces barrières avec une arête du graphe de calcul donne lieu à l'insertion d'un registre de retemporisation qui vient interrompre le chemin critique.

En l'occurrence, pour notre circuit, les boucles sont au nombre de deux, situées dans la logique du normalisateur par bloc. Les barrières de resynchronisation sont insérées autant que faire se peut après chaque opérateur. Le résultat est figuré en 2.8.

2.1.3.4 Synthèse du circuit

La description du circuit sous forme de graphe fait intervenir des opérateurs entiers de haut niveau comme les additionneurs, la valeur absolue, ou encore les décaleurs de bits. La traduction de tels opérateurs en portes logiques dépend étroitement de la technologie visée. Elle est qualifiée de *technology-mapping*. C'est l'objet du chapitre 3, et elle n'est pas détaillée pour le circuit de compression.

2.1.3.5 Interfaçage du circuit

On a déjà abordé en 2.1.3.2, avec l'introduction du signal d'*enable*, un aspect de l'interfaçage du circuit avec son environnement. La réalité de la communication ne se borne pas à gérer les interruptions dans le flux de données : le circuit doit être effectivement relié avec les bus de communication de la station de travail qui utilise le co-processeur Pam. Cette étape qui fait la part belle à la programmation système est l'objet du chapitre 4.

2.1.3.6 Exécution du circuit

Une fois augmenté de l'interface avec le bus de communication, le circuit est incarné dans un fichier de configuration pour le FPGA, puis chargé à l'intérieur de celui-ci. Une routine logicielle permet alors d'envoyer simplement les données à traiter vers le FPGA et d'en récupérer les résultats.

Ainsi, l'amélioration apportée par l'exécution de notre routine sur le co-processeur FPGA peut être quantifiée. C'est l'objet du chapitre 5 que d'effectuer de telles mesure pour un ensemble de circuits variés. En particulier la section 5.1 présente les résultats d'implémentation pour le circuit de compression-décompression sans pertes.

2.2 Présentation de DSL

L'exemple du circuit de compression sans pertes est repris pour montrer comment Design Source Language (DSL) peut être utilisé pour le réaliser.

2.2.1 Spécifications de DSL

2.2.1.1 Sémantique

Notre Design Source Language (DSL) est un langage fonctionnel synchrone orienté flot de données. Il dispose de types d'ordre supérieur et de la surcharge d'opérateurs : ces caractéristiques sont directement héritées du langage Jazz [10] dans lequel le DSL est immergé.

DSL a une sémantique de flot de données synchrones définie au bit et cycle près. Chaque variable dans le code source représente le flot des valeurs calculées à chaque cycle d'horloge en réponse aux valeurs d'entrée du circuit. En DSL, les flots sont arithmétiques : la valeur de chaque variable à un instant donné est un entier de précision arithmétique arbitraire. Le système est synchrone, avec une horloge globale implicite unique. Hormis ces restrictions, DSL ressemble à Lustre [11] et à Lucid Synchrone [12].

La sémantique usuelle des opérateurs arithmétiques de DSL est étendue aux flots, à raison d'une opération arithmétique par cycle d'horloge. Des opérateurs synchrones à mémoire standard sont aussi présents, comme les registres à décalage et les blocs mémoire. Le chapitre 3 liste les primitives disponibles. En particulier, le registre synchrone Z réalise un délai unitaire entre deux cycles d'horloge. Ainsi $Z(u)$ porte la suite définie par :

$$\begin{cases} & Z(u)_0 &=& 0 \\ k > 0 \implies & Z(u)_k &=& u_{k-1}; \end{cases} \tag{2.9}$$

2.2.1.2 Types

Une force de DSL est que le même code-source peut être exécuté avec différents types en entrée. Toutes les variables d'un circuit DSL sont typées par A, le type de flot arithmétique synchrone. Les objets de type A supportent toutes les opérations arithmétiques standard du langage C[4] :

$$(+, -, *, \div, \ll, \gg, \neg, \&, |, \oplus, <, ==, >, \geq, \leq) \tag{2.10}$$

ainsi que les opérations synchrones Z et Z^n.

DSL est par ce côté comparable à Lava [4], dans lequel l'utilisation des classes de type et des *monades* [13] permettent au programmeur de définir de nouvelles interprétations pour le même code de manière très souple, en réutilisant ou en spécialisant des comportement déjà définis.

Ces propriétés sont mises à profit au sein même du compilateur. Des flots d'entiers standards y sont utilisés pour la simulation logicielle du circuit – l'exécution matérielle finale relève de la même sémantique, qui est préservée au cours des différentes étapes du compilateur.

[4] Une différence mineure est que les opérateurs de comparaison $\{<, ==, >\}$ produisent un élément de type A plutôt qu'un booléen pour des raisons d'uniformité des types

Par ailleurs d'autres types sont utilisés afin de calculer les différentes annotations automatiques ajoutées au graphe flot de données pour générer *in fine* une implémentation matérielle performante. Le compilateur fait plus spécifiquement appel aux interprétations suivantes :

- une interprétation abstraite structurelle du circuit qui permet d'accéder au graphe flot de données représentant le circuit, et de le soumettre à l'identification des composantes fortement connexes, au tri topologique et à la réécriture à destination du dorsal de compilation ;
- une interprétation en arithmétique d'intervalles afin de réaliser l'analyse d'intervalle et le *bitsizing* des variables du circuit ;
- une interprétation suivant un modèle simple de délai a priori pour estimer les paramètres critiques de la technologie-cible et guider le placement des registres de retemporisation ;

2.2.1.3 Étapes du compilateur

Le compilateur DSL procède par étapes – exactement comme nous venons de le faire pour le compresseur sans perte – afin de synthétiser un circuit à partir de son code-source de haut niveau.

1. Le code-source est converti vers une forme Assignation Statique Unique (ASU) qui correspond au graphe des opérateurs (comme présenté en 2.1.2).
2. L'intervalle des variables est automatiquement analysé. Lorsque l'analyse d'intervalle ne converge pas, l'ajout d'annotations manuelles est nécessaire pour continuer la compilation.
3. Une fois l'analyse d'intervalle réalisée, une représentation bit-à-bit des entiers est dérivée à partir de l'intervalle borné calculé, pour chaque variable et pour chaque opérateur de la forme ASU.
4. Il en résulte une description matérielle de la routine grâce à une première étape de synthèse des opérateurs (*technology mapping*).
5. Le compilateur insère ensuite des registres de retemporisation pour optimiser la vitesse d'horloge du circuit final (*retiming*).
6. Les interfaces de communication matérielles et logicielles sont générées pour réaliser de manière transparente la communication du circuit avec l'hôte du coprocesseur.
7. Le résultat des étapes précédentes est formaté pour traitement par les outils dorsaux propriétaires afin de générer le fichier de configuration final du FPGA pour l'exécution accélérée de la routine.

Chacune de ces étapes ajoute au circuit des *annotations*. Les annotations préservent la sémantique (synchrone et opératoire) du circuit : elles ne font que spécifier une propriété de la variable annotée, qui sera exploitée pour parvenir à un circuit synthétisable.

La majorité des annotations est automatiquement générée par le compilateur. Toutefois, la possibilité d'annoter manuellement le circuit est capitale : certaines annotations sont nécessaires pour aider le compilateur.

2.2.1.4 Annotations manuelles

C'est pourquoi toutes les annotations font partie intégrante de la syntaxe du langage. Une annotation manuelle explicite dans le code-source est une directive qui peut guider ou forcer le comportement du compilateur.

Certaines annotations sont requises pour aider le compilateur à réaliser une passe d'annotation automatique. C'est le cas par exemple pour l'étape de *bitsizing* qui sera étudiée plus en détail en 2.3.2.

D'autres annotations sont simplement utilisées comme des directives, afin d'imposer au compilateur un choix spécifique ; c'est le cas par exemple lors de la phase de *technology-mapping*, pour choisir telle ou telle implémentation d'un registre à décalage. Ces directives niveau source permettent au concepteur de circuit expérimenté de préciser chaque détail d'une implémentation, par exemple lorsque les choix de compilateur s'avèrent peu pertinents.

Les annotations sont liées aux différentes étapes du compilateur. Chaque type d'annotation a une signification précise dans son contexte d'opération. Elle entraîne un comportement spécifique du compilateur dans l'étape en question, et des conséquences pour toutes les étapes ultérieures du compilateur. En revanche, le comportement d'une annotation doit rester neutre, c'est-à-dire une identité, pour toutes les étapes strictement antérieures.

À chaque fois qu'une annotation spécifie un comportement qui pourrait affecter la sémantique définie par les étapes antérieures, la correction du comportement du circuit doit être vérifiée à l'exécution. L'annotation de *bitsizing* présentée en 2.3.2 fourni un exemple de ce mécanisme.

Le résultat final est que même avec des annotations nombreuses, la sémantique du code source de haut niveau est toujours préservée, au moins pour le type de référence des flots d'entiers. Une annotation dont l'effet violerait la sémantique de référence est débusquée à l'exécution.

La réalité de l'utilisation du compilateur dans les circuits de traitement vidéo étudiés au chapitre 5 montre que notre compilateur synthétise des circuits de haute performance avec seulement quelques directives dans le code-source original.

2.2.2 Description du compresseur en DSL

DSL est utilisé dans cette partie pour décrire la routine de compression. La syntaxe et l'usage de DSL sont ainsi illustrés. La description DSL suit exactement la structuration du circuit par blocs, chaque bloc étant incarné par une fonction Jazz.

2.2.2.1 Différentielle

```
// Transformée différentielle
fun Transform(i) = (u, d) {
  d = sub(i,Ze(i));      // Signal différentiel
  u = lt(C(11), abs(d)); // Indicator de décompression
}
```

FIG. 2.9 – Code source du transformateur de pixel

La description de la figure 2.2 donne lieu au code-source 2.9. La primitive de registre `Ze` sera définie plus bas ; elle comprend un appel implicite au signal d'activation du circuit.

2.2.2.2 Codeur

```
// Codeur
fun PixelCoder(u, i, d) = (c, l) {
  l  = mux(u, C(9), l8);
  cs = mux(u, i, v8);
  c  = or(u, shift(1)(sc)); // Ajout du bit préfixe

  du = and(d, C(31));       // Pixel compressé
  v8 = lut(LUTv8)(du);      // LUT(5b)->7b
  l8 = lut(LUTl8)(du);      // LUT(5b)->4b

  // LUT des codes d'Élias
  LUTv8 = [n-> (n<16) ? eliasC(n) : eliasC(n-32)];
  // LUT des longueurs
  LUTl8 = [n-> (n<16) ? eliasL(n) : eliasL(n-32)];

  // calcul des codes d'Élias
  fun eliasL(n) = cl { (cn, cl) = elias(n); }
  fun eliasC(n) = cn { (cn, cl) = elias(n); }
  fun elias(n:int) = vl {
    sign = n < 0;
    s = sign ? 1 : 0;
    a = sign ? -n : n;
    vl = (a > 11) ? (0, 0) :
         (n==0) ? (0, 2) :
         (1+2*s+4*ca, la+3);
    (ca, la) = (a==1) ? (0,1) :
               (a<4) ? (1+4*(a&1),3) :
               (3+4*(a&7),5);
  }
}
```

FIG. 2.10 – Code source du codeur

La description de la figure 2.3 donne lieu au code-source 2.10.

Le calcul des valeurs d'initialisation des Look-Up Table (LUT)s par extension pour le code d'Élias et sa longueur dans le cas d'un pixel compressé est ici réalisé très simplement grâce à l'utilisation du langage Jazz et de ses types entiers natifs.

L'intérêt de l'immersion du langage de description matériel dans un langage généraliste (c'est à dire, pour nous, de l'immersion de DSL dans Jazz), prend ici tout son sens.

```
fun BlockCoder(c, l) = (b, e) {
  var assertrv= [0,2**(blockSize-1)-1];

  // Logique de contrôle
  rl = Ze(and(nl, C(15)));  // Niveau du tampon
  nl = add(rl, e);
  e  = shift(-4)(nl);       // Bit de débordement
  eb = tobool(e);

  // Chemin de données
  rv = Ze(rc);                      // Tampon de données
  nv = or(rv, lshift(c, rl));       // Ajout du code entrant
  r  = mux(eb, shift(-blockSize)(nv), nv);
                                    // Retrait du bloc sortant
  rc = assert(r, assertrv);           // Annotation d'intervalle
  b  = and(nv, C(2**blockSize-1));// Bloc sortant
}
```

FIG. 2.11 – Code source du normalisateur par bloc

2.2.2.3 Normalisateur par bloc

La description de la figure 2.4 donne lieu au code-source 2.11.

La description du normalisateur par blocs est l'occasion de rencontrer notre première annotation manuelle de *bitsizing* agissant sur la variable `r` et définissant la variable `rc`.

L'opération d'annotation manuelle à l'aide de la fonction `assert` spécifie la taille du tampon de données, dont la longueur n'est pas calculable par notre analyse automatique (comme nous le verrons en 2.3.2). Cette annotation est très similaire à l'annotation disponible en Esterel (voir [3], chapitre 4).

Elle agit comme une identité opératoire sur les flots d'entiers qui la traversent. Toutefois, elle insère une assertion qui vérifie au cours de la simulation que le flot des valeurs calculées reste circonscrit à l'intervalle spécifié dans le code source. Par ailleurs, l'information contenue dans l'annotation est utilisée comme hypothèse supplémentaire par l'analyse automatique d'intervalle des variables du circuit.

2.2.2.4 Compresseur

Le compresseur met bout-à-bout l'ensemble des fonctions définies précédemment. Ce niveau définit aussi la primitive de registre `Ze` qui utilise le signal d'*enable* `en` explicité en entrée du circuit.

2.2.3 Compilation du compresseur avec DSL

2.2.3.1 Définition du circuit

La première opération DSL consiste à définir le circuit. Cette opération permet au compilateur de construire en mémoire la description ASU du circuit sous

```
// Compresseur sans pertes
fun Coder(i,en) = (b, oe) {
  blockSize = 16;
  (u, d) = Transform(i);
  (c, l) = PixelCoder(u, i, d);
  (b, e) = BlockCoder(c, l);
  oe = tobool(and(en, e));  // Génération de l'enable de sortie
  fun Ze(s) = reg_en(en)(s);
  /* Définition des blocs */
}
```

FIG. 2.12 – Code source du compresseur

```
fun Compresseur() = compresseur {
  // Séquence de test du compresseur
  var inP = [ 22, 12, 12, 12, 12, 12, 12, 12, 12, 0,
      15, 30, 45, 60, 75, 90, 105, 120, 135,
      150, 165, 180, 195, n-> Integer.random(16)];
  // Valeurs de l'enable d'entrée
  var inE = [n->true];

  e = Enable(inE);
  i = Input([0,255],inP);
  (b, oe) = Coder(i,e);

  compresseur = Circuit.build(1, [b], oe);
}

var circuit = Compresseur();
```

FIG. 2.13 – Définition du circuit

la forme d'un graphe explicite. La figure 2.13 présente la construction du circuit de compression. Jazz est pensé comme un langage interactif de prototypage : la suite des opérations sur le circuit est donc présentée dans cet optique, et ne présente que la ligne qui serait entrée sur une invite de commande interactive.

2.2.3.2 Opération de simulation

Le circuit DSL peut être simulé sur le jeu de données défini en entrée dans le code (2.13). La simulation du circuit au niveau arithmétique permet de conserver des vitesses de simulation correctes au regard de celles obtenues pour des simulations de bas niveau. Elle permet ainsi de prototyper rapidement le circuit.

Les jeux de données en sortie peuvent être sauvés dans un fichier pour utilisation ultérieure. En particulier, les données produites sont utilisées dans un jeu de tests automatiques qui vérifie la correction de chacune des étapes du compilateur en comparant les résultats de l'exécution du circuit après les étapes ultérieures de compilation aux données de la simulation de référence. Cette vé-

rification est faite automatiquement jusqu'à l'exécution finale du circuit sur le co-processeur FPGA. Son intérêt est particulièrement fort au cours de la mise au point du compilateur.

Une fonction simple permet d'exécuter le circuit sur un nombre spécifié de cycles et d'en sauvegarder automatiquement les résultats ; elle agit sur l'objet circuit défini en 2.2.3.1 :

```
var circuit_name = "Compresseur";
@circuit.dumpio(circuit_name, 128);
```

Lors de la simulation, les assertions de *bitsizing* présentes en tant qu'annotations des variables sont vérifiées dynamiquement à chaque cycle de simulation.

2.2.3.3 Dimensionnement du chemin de données

Le circuit supporte une opération de *bitsizing* global qui permet d'exécuter un algorithme d'annotation automatique de chacune des variables du circuit avec leur intervalle de confiance :

```
var circuit_bitsized = circuit.bitsizing(9);
```

Toutefois, le problème de la convergence de l'analyse d'intervalle est indécidable. Par conséquent, notre compilateur, à l'instar de tout programme qui essayerait de réaliser une analyse d'intervalle, peut lever une exception qui signale l'incapacité du système à obtenir la convergence de l'analyse d'intervalles dans la limite de temps spécifiée en argument.

Le concepteur du circuit doit alors assister le compilateur par l'insertion d'annotations manuelles de *bitsizing* bien choisies – une ou deux suffisent en général.

2.2.3.4 Retemporisation du circuit

Le circuit supporte une opération de *retiming* qui permet d'effectuer automatiquement le travail de retemporisation décrit en 2.1.3.3.

```
var circuit_retimed = circuit_bitsized.retime(1);
```

Son argument indique la profondeur topologique maximale admise entre les barrières de retemporisation insérées dans le circuit. En l'occurrence, l'argument de 1 insère une barrière de registre après chaque opérateur du circuit, comme cela avait été fait en 2.1.3.3.

2.2.3.5 Synthèse des opérateurs

Une fois le circuit *bitsizé* et éventuellement *retemporisé* correctement, il est possible d'en demander la réécriture dans le langage de synthèse bas niveau utilisé au cours de cette thèse, PamDC :

```
@circuit.dumppamdc(circuit_retimed)(15.0,15.0,15.0);
```

Circuit	Vitesse		Ressources		Bande passante
	ns	MHz	RAMB	Tranches	utile MiB.s^{-1}
original	20.57	48	0 (0%)	114 (1.0%)	48
retemporisé	5.39	185	0 (0%)	194 (1.8%)	185

TAB. 2.2 – Implémentation matérielle du compresseur

À partir de cette étape, le système DSL écrit en Jazz passe la main à une couche de synthèse des opérateurs de bas niveau qui a été écrite en PamDC. Les étapes ultérieures de la synthèse du circuit, qui sont très dépendantes de la technologie visée par le compilateur, seront décrites aux chapitres 3 et 4.

Le tableau 2.2 caractérise les circuits obtenus après synthèse complète. Si le détail des résultats sera expliqué dans les chapitres suivants, il est d'ores et déjà possible de constater l'efficacité de l'algorithme de retemporisation automatique sur la fréquence d'horloge du circuit final.

2.3 Implémentation de DSL

2.3.1 Graphe des opérateurs

Le code source en DSL présenté en 2.2.2 est une représentation directe du circuit sous forme de graphe de calcul présenté en 2.1.2. L'approche retenue dans DSL n'est pas celle d'une équivalence systématique entre les deux formes : le circuit écrit en DSL est simplement converti en graphe par son exécution sur un type de données structurel.

À l'issue de cette conversion, Jazz dispose en mémoire du graphe dirigé des calculs effectués par le circuit. Chaque nœud du graphe $\mathcal{G} = (V, E)$ est un opérateur arithmétique étiqueté par le type d'opération ; ses arêtes entrantes, ordonnées, en sont les opérandes, et les arêtes sortantes constituent les résultats de l'opération.

Cette représentation interne s'apparente à un déroulement du code-source initial sous forme Assignation Statique Unique (ASU) [14] pour les langages impératifs. En particulier, elle donne une existence comme arêtes à chacune des variables intermédiaires muettes du code-source original. La forme ASU est capitale pour les algorithmes ultérieurs. L'analyse d'intervalles doit s'appliquer à l'ensemble des arêtes du graphe (qu'elles soient issues de variables muettes ou explicitées dans le code-source). Quant à la retemporisation, elle opère directement sur la structure du graphe.

2.3.2 Dimensionnement du chemin de donnée

L'analyse d'intervalle est utilisée dans le compilateur DSL pour automatiser le dimensionnement du chemin de données réalisé en 2.1.3.1.

Note méthode de dimensionnement du chemin de données s'inspire fortement des méthodes d'analyse statique : elle calcule itérativement un point fixe à l'aide d'une arithmétique d'intervalles sûre, en se fondant sur les indications placées dans le code source par le programmeur. Cette approche inférentielle est notablement différente de l'approche utilisée dans Esterel, qui utilise le typage pour dimensionner le chemin de données, et essaye par ailleurs d'obtenir la preuve des assertions présentes dans le code source.

2.3.2.1 Position du problème

Le problème pour parvenir à une forme de circuit synthétisable est de déterminer pour chaque variable entière du circuit un intervalle fini qui borne l'ensemble des valeurs de la variable au cours du temps, pour toutes les exécutions possibles. Lorsqu'un tel intervalle $[i, s]$ est déterminé pour une variable, l'ensemble des valeurs peut-être codé de telle façon à être représentée par un tableau fini de $l = 1 + \lfloor \log_2(s - i) \rfloor = binsize(s - i)$ bits.

En théorie, l'analyse exacte d'une variable entière peut être réalisée par exploration systématique des états d'exécution du circuit. En pratique, l'analyse exhaustive exacte n'est pas utilisable car elle se heurte aux limites théoriques et pratiques du *model-checking* :

1. Déterminer si une variable est bornée est une propriété qui n'est *pas décidable* en toute généralité.

2. La complexité en espace du calcul augmente très vite. C'est particulièrement le cas pour les algorithmes de traitement d'image dont les tampons de données internes sont importants.
3. L'implémentation propre d'algorithmes de model-checking efficaces est complexe.

Toutes les méthodes basées sur l'exploration systématique des états du circuit sont par conséquent condamnées.

La solution retenue essaie d'adresser ces problèmes. C'est une méthode simple, approchée, et qui peut admettre l'échec de l'analyse, auquel cas l'utilisateur est invité à ajouter des informations sous la forme d'annotations manuelles pour permettre à l'analyse de se terminer.

2.3.2.2 Arithmétique d'intervalles

Afin de réaliser l'analyse d'intervalles, Jazz exécute symboliquement le circuit sur des intervalles.

DSL scrute la variation des intervalles portés par chaque variable à chaque cycle d'horloge. L'analyse se termine lorsque le programme atteint un *point fixe*, c'est-à-dire un ensemble d'intervalles stable d'un cycle sur l'autre en chaque variable du circuit.

Soit $c \in \mathbb{N}$ le cycle auquel le point fixe est atteint. Lorsque l'analyse est réussie $c < \infty$, l'intervalle rapporté pour une variable contient l'ensemble des valeurs possibles pour la variable, mais il peut être strictement plus grand.

La théorie classique de l'analyse d'intervalle [15] se préoccupe des opérations sur les réels, représentés par des intervalles d'extrémités rationnelles. Elle est ici restreinte à l'analyse d'intervalles entiers, et utilise une arithmétique qui supporte l'ensemble des opérateurs arithmétiques primitifs entiers, les opérateurs booléens bit-à-bit ainsi que les opérations synchrones.

Chaque opérateur entier est étendu aux intervalles à l'aide de quelques opérations arithmétiques usuelles réalisées sur les bornes entières de l'intervalle. L'analyse d'intervalles entiers est ainsi presqu'aussi rapide que la simulation entière du circuit : la mémoire requise est proportionelle à $\mathrm{Card}(E)$ le nombre de variables entières du circuit, et le temps d'exécution proportionnel à $c \times n$.

La définition de l'arithmétique d'intervalles est cruciale : c'est elle qui garantit la correction de la méthode. Elle répond à une propriété de sécurité, à savoir que l'intervalle calculé contient l'ensemble obtenu par l'application de l'opérateur entier aux intervalles de départ (calculé cette fois par l'extension naturelle de l'opérateur aux ensembles).

Opérateurs monotones L'arithmetique d'intervalle modèle très facilement les opérations arithmétiques *monotones* (croissantes ou décroissantes) par rapport à chacun de leurs arguments. Presque tous les opérateurs arithmétiques usuels sont monotones, l'extension de leur comportement sur les intervalles est présentée en table 2.3.

Opérateurs non-monotones D'un autre côté, les opérateurs bit-à-bit ne sont pas des fonctions monotones sur les entiers, par exemple pour l'opérateur de *et logique*. Des formules approchées doivent alors être utilisées :

$$
\begin{array}{lrcl}
Add: & [i,s]+[l,h] & = & [i+l, s+h]. \\
Sub: & [i,s]-[l,h] & = & [i-h, s-l]. \\
Not: & \neg[i,s] & = & [\neg s, \neg i]. \\
Mul: & [i,s]\times[l,h] & = & [min(il,ih,sl,sh), max(il,ih,sl,sh)]. \\
lShift: & l \geq 0 \Rightarrow [i,s] \ll [l,h] & = & [min(i \ll l, i \ll h), max(s \ll l, s \ll h)]. \\
rShift: & l \geq 0 \Rightarrow [i,s] \gg [l,h] & = & [min(i \gg l, i \gg h), max(s \gg l, s \gg h)]. \\
Div: & l > 0 \Rightarrow [i,s] \div [l,h] & = & [min(i \div h, i \div l), max(s \div h, s \div l)]. \\
Div: & h < 0 \Rightarrow [i,s] \div [l,h] & = & [min(s \div h, s \div l), max(i \div h, i \div l)].
\end{array}
$$

TAB. 2.3 – Arithmétique d'intervalles pour les opérateurs monotones

$$
\begin{array}{rcl}
i \geq 0, l \geq 0 \Rightarrow [i,s] \cap [l,h] & = & [0, min(s,h)], \\
i \geq 0, l < 0 \Rightarrow [i,s] \cap [l,h] & = & [0, s], \\
i < 0, l \geq 0 \Rightarrow [i,s] \cap [l,h] & = & [0, h], \\
s < 0, h < 0 \Rightarrow [i,s] \cap [l,h] & = & [-2^k, min(l,h)], \\
\text{else} \Rightarrow [i,s] \cap [l,h] & = & [-2^k, max(l,h)],
\end{array}
\tag{2.11}
$$

où $k = max(binsize(\neg i), binsize(\neg l))$ et $binsize(n) = \lfloor \log_2(n) \rfloor + 1$ (et valant zero lorsque n est nul) n'est autre que la longueur binaire du naturel $n \in \mathbb{N}$. La définition (2.11) est bien formée car les préconditions sont mutuellement exclusives et couvrent tous les cas. Par ailleurs la formule 2.11 coïncide avec l'intervalle enveloppe de l'ensemble exact pour les intervalles en entrée de la forme $[0,s]$ et $[0,h]$.

Toutes les autres opérations de la logique booléene sont réduites au cas du *et logique* grâce aux formules de l'algèbre de Boole. L'opérateur de multiplexage est traité naïvement par union de ses arguments :

$$
\begin{array}{lrcl}
Or: & [i,s] \cup [l,h] & = & \neg(\neg[i,s] \cap \neg[l,h]) \\
Xor: & [i,s] \oplus [l,h] & = & (\neg[i,s] \cap [l,h]) \cup \\
 & & & ([i,s] \cap \neg[l,h]) \\
Cond: & [0,1]\,?\,[i,s] : [l,h] & = & [\min(i,l), \max(s,h)]
\end{array}
$$

Il est toujours possible de rencontrer une analyse d'intervalles divergente. Par exemple, l'analyse d'intervalle entière échoue sur le code du compresseur sans annotation manuelle : des intervalles toujours plus grands sont calculés à chaque cycle.

Une solution simple est d'autoriser le concepteur de circuit à annoter le code-source avec une restriction d'intervalle explicite. Il suffit alors d'ajouter une annotation manuelle au code source pour que l'analyse d'intervalles converge :

```
rc = assert(r, assertrv);          // Annotation d'intervalle
```

Grâce à cette annotation manuelle, l'analyse d'intervalles se termine promptement sur un point fixe, pour toutes les variables du circuit. La restriction d'intervalle explicite donne au programmeur la possibilité de forcer la convergence de l'analyse d'intervalles à l'aide d'annotations manuelles.

2.3.2.3 Limitations

Syndrome linéaire Un problème de base avec l'arithmétique d'intervalle est le *syndrome linéraire* : pour $p \in [0, 255]$, l'expression $a = p - p$ devrait être évaluée à $a = [0, 0]$ et non à $a = [-255, 255]$, la valeur calculée par notre arithmétique.

Pour les nombres réels, l'*arithmétique d'intervalles affine* est une solution bien connue [16] au problème du syndrome linéraire. Un intervalle affine e y prend la forme $e = i + \epsilon l$ où $i, l \in \mathbf{R}$ et $\epsilon \in \{-1, 0, 1\}$ est une variable symbolique qui représente le point médian de l'intervalle $[i - l, i + l]$. L'arithmétique affine permet d'évaluer correctment l'expression $e - e$ en $(i - i) + \epsilon(l - l) = 0$.

Un exemple réel de ce syndrome est fourni par le circuit de diffusion de Floyd-Steinberg étudié à la section 5.2. Les multiplications de la variable $e1$ par les constantes 3, 5 et 7 y sont implémentées à l'aide de décalages et d'additions :

$$\begin{aligned} e3 &= (e1 \ll 1) + e1 \\ e5 &= (e1 \ll 2) + e1 \\ e7 &= (e1 \ll 3) - e1 \end{aligned}$$

L'analyse d'intervalles standard pour une variable $e1 \in [-7, 15]$ conduit au résultat $e7 \in [-71, 127]$. Ce résultat est correct, mais grandement surestimé. L'analyse affine en revanche conduit à l'intervalle optimal $e7 \in [-49, 105] = 7 \times [-7, 15]$.

Syndrome conditionnel Un problème toujours présent avec l'analyse affine est celui du *syndrome conditionnel* : pour une variable $p \in [0, 255]$, l'expression de la valeur absolue $a = p < 0\,?\, -p : p$ devrait être évaluée comme $a = [0, 255]$ et non pas comme $a = [-255, 255]$, la valeur calculée par l'analyse d'intervalles affine ou non.

Le syndrome conditionnel apparaît par exemple dans le tampon du renormalisateur par blocs, avec l'expression :

```
r  = mux(eb, shift(-blockSize)(nv), nv);
```

Il est nécessaire de comprendre l'interaction entre la variable de contrôle `rl` et le buffer `rv` pour interpréter correctement l'expression conditionnelle.

Algorithme de Halbwachs & Cousot La résolution du *syndrome conditionnel* est un problème très complexe. La solution la plus élaborée est celle de l'algorithme de Halbwachs et Cousot [17]. Cet algorithme conserve un contexte contenant un grand nombre d'inégalités linéraires entre variables symboliques. Toutefois, l'algorithme est difficile à implémenter correctement et en toute généralité : la méthode originale de [17] ne couvre pas toutes les opérations entières utilisées en DSL. Sa complexité est aussi un facteur potentiellement limitant pour de grands circuits.

Toutefois, la mise en oeuvre du système d'analyse statique ASTRÉE[5][18] permet d'analyser automatiquement (et correctement !) certains des circuits étudiés en exemple (voir par exemple 5.2.3.3), et d'éviter ainsi à l'utilisateur l'ajout

[5]je remercie nos voisins de l'École Normale Supérieure (ÉNS) pour nous avoir aidés à mettre en place l'expérimentation avec ASTRÉE

des annotations manuelles nécessaires avec notre analyse naïve. Par conséquent, certaines faiblesses de notre dispositif pourraient être améliorées.

En revanche, les annotations manuelles du code-source restent une nécessité pour assister à la résolution de ce problème qui n'est pas calculable. ASTRÉE n'est par exemple pas capable d'effectuer sans annotations l'analyse d'intervalles pour le circuit présentée en 5.3.

La méthode d'analyse d'intervalles utilisée en DSL conduit à une implémentation très simple en utilisant les capacités de Jazz, et intègre proprement toute la flexibilité nécessaire grâce aux annotations. Dans la pratique, seules quelques annotations sporadiques sont requises, à des points du circuit où l'attention du concepteur de circuit est la bienvenue. En ce sens, l'implémentation actuelle permet au concepteur de circuit de concentrer son attention aux endroits où elle est nécessaire.

2.3.3 Retemporisation

DSL utilise son propre algorithme de retemporisation pour automatiser l'étape d'optimisation de la fréquence d'horloge. En effet, réaliser manuellement la transformation est fastidieux et, en conséquence, source d'erreurs.

2.3.3.1 Position du problème

La retemporisation d'un circuit consiste à insérer des registres le long du chemin critique pour l'interrompre. Cette opération proprement réalisée ne fait qu'augmenter la latence du circuit sans en modifier la sémantique opératoire sous le signal d'*enable*.

La retemporisation d'un circuit est essentiellement un algorithme de graphe, par conséquent nous utilisons la représentation du circuit sous forme de graphe d'opérations obtenue en 2.3.1.

La retemporisation des circuits sans boucles ne pose pas de grand problème théorique ([19],[9]). La partie technique de l'implémentation consiste, à partir d'un graphe de circuit quelconque, à se ramener à un Graphe Acyclique Orienté – Directed Acyclic Graph (DAG). Pour parvenir à ce résultat, les boucles du graphe[6] sont d'abord isolées puis condensées.

2.3.3.2 Vers le graphe acyclique

Les deux notions de théorie des graphes utilisées sont celles de l'isolation des composantes fortement connexes pour la mise en évidence des boucles du graphe et de graphe de condensation pour la réduction des boucles du graphe. Cette dernière notion est brièvement définie :

Definition Étant donné un graphe $\mathcal{G} = (V, E)$ et une relation d'équivalence $\mathcal{R}$ sur l'ensemble de ses sommets V, le graphe de condensation de $\mathcal{G}$ par $\mathcal{R}$ est le graphe obtenu en fusionnant les sommets de $\mathcal{G}$ selon les classes d'équivalence de $\mathcal{R}$ et en supprimant les arêtes en boucles et les arêtes multiples créées par cette association.

[6] Il s'agit de boucles de rétroaction au travers d'un registre ; les boucles combinatoires sont exclues du modèle

Le graphe de condensation $\mathcal{G}/\mathcal{R}$ a autant de sommets qu'il existe de classes d'équivalence dans l'ensemble des sommets du graphe original. Ses arêtes capturent la connectivité entre ces différentes classes d'équivalence. En particulier, il est possible de trouver une partition[7] des sommets du graphe telle que le graphe de condensation est un graphe *acyclique* dont la retemporisation est aisée.

L'ensemble des relations d'équivalence qui vérifient cette propriété admet un minimum (pour la relation de comparaison usuelle des partitions d'ensemble). La relation minimum, c'est-à-dire la plus fine vérifiant la propriété, est la partition du graphe en ses composantes fortement connexes [20]. La première étape de la retemporisation consiste donc à calculer le graphe de condensation des composantes fortement connexes du graphe du circuit.

Composantes fortement connexes Il existe de nombreuses versions ([20], [21]) de l'algorithme d'extraction des composantes fortement connexes, qui est un des algorithmes de Tarjan. La version retenue pour DSL est présentée en détail dans [22].

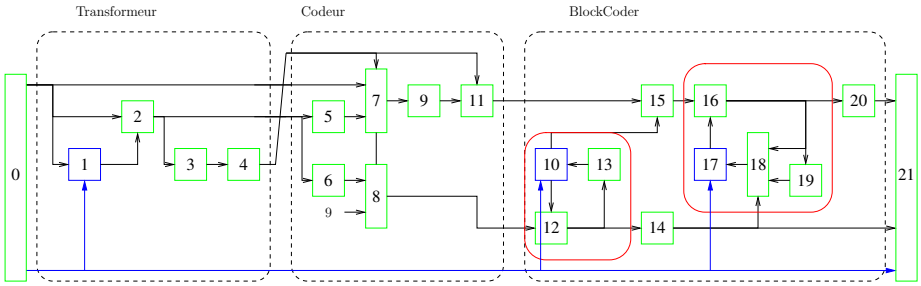


FIG. 2.14 – Numérotation du circuit de compression sans perte

Exemple À titre d'exemple, l'application de l'algorithme est présenté pour le circuit du compresseur. Les nœuds du graphe sont numérotés en figure 2.14. Deux pseudo-nœuds ont été ajoutés au graphe, le premier en entrée, le second en sortie, pour forcer la synchronisation des flux de données aux interfaces du circuit.

Dans le compresseur, les seules composantes fortement connexes non-triviales sont, d'une part l'ensemble $\{16, 17, 18, 19\}$ qui calcule le tampon de données du renormalisateur de blocs, et d'autre part l'ensemble $\{10, 12, 13\}$ qui calcule sa variable d'état et de contrôle.

Le regroupement des opérateurs des composantes fortement connexes permet de dériver un DAG à partir du circuit initial, c'est le graphe de condensation de $\mathcal{G}$ sur l'ensemble de ses composantes fortement connexes qu'on note $\overline{\mathcal{G}}$. C'est sur ce DAG que l'algorithme travaille dorénavant.

2.3.3.3 Retemporisation

À partir du graphe condensé des composantes fortement connexes $\overline{\mathcal{G}}$, l'algorithme de retemporisation proprement dit procède par étapes pour insérer les registres.

[7] à tout le moins la partition des sommets grossière $\{V\}$

1. Le délai $d(v)$ de chaque sommet v de $\overline{\mathcal{G}}$ est calculé. Pour un nœud trivial, qui correspond à un singleton du circuit original, le délai est calculé suivant la fonction implémentée et le *bitsizing* des opérandes, suivant un modèle dépendant de la technologie. Pour un nœud complexe, qui correspond à une boucle du graphe de calcul initial, il s'agit du chemin critique de la boucle.
2. Le délai critique maximal $d_{\max}$ des composantes non-triviales du graphe condensé donne la borne inférieure sur la période du circuit[8] : sans modifications complexes qui se situeraient hors du champ de DSL, le chemin critique d'une boucle n'est pas modifiable.
3. Les registres de retemporisation sont finalement insérés en tenant compte de ce délai.

Sur l'exemple du compresseur, la modification automatique des boucles mises en évidence dans le circuit pour augmenter leur fréquence maximale de fonctionnement n'est pas aisé. Elles déterminent ainsi la fréquence de fonctionnement maximale du circuit.

Insertion des registres L'insertion des registres de retemporisation sur le graphe condensé $\overline{\mathcal{G}}$ peut faire l'objet de nombreuses optimisations. C'est la simplicité qui a guidé la solution implémentée dans DSL. L'insertion des registres de retemporisation y est définie localement pour chaque composante fortement connexe v en fonction des décisions prises pour les entrées de v (à proprement parler, en fonction des sommets dont v est adjacente).

L'algorithme parcourt les sommets de $\overline{\mathcal{G}}$ en ordre topologique [9] et définit en chaque composante v les deux quantités suivantes :

- une profondeur de synchronisation entière $p(v)$ qui correspond à la latence discrète accumulée depuis l'entrée du circuit par l'insertion des registres de retemporisation ;
- un délai critique courant $c(v)$ qui correspond au délai le long du chemin critique passant par la composante v.

Ces deux quantités sont aisément calculées pour v en fonction de leur valeur pour l'ensemble $I(v)$ des sommets de $\overline{\mathcal{G}}$ dont v est adjacent. Le calcul procède par étapes :

1. Une première profondeur de synchronisation $p_i(v)$ en entrée de l'opérateur est calculée :
$$p_i(v) = \max_{u \in I(v)} c(u)$$
Il faut remarquer que pour des raisons de correction du circuit, des registres de resynchronisation seront nécessairement insérés entre v et toute composante u en entrée telle que $c(u) < p_i(v)$.
2. L'indicateur de retemporisation booléen $r(v)$ est ensuite calculé. Cet indicateur s'appuie sur $p_i(v)$ et prend en compte le délai critique maximal $d_{\max}$, pour indiquer si une barrière de registre supplémentaire est nécessaire en entrée de l'opérateur :
$$r(v) = \bigwedge_{u \in I(v)} (p_i(v) \leq p(u)) \wedge (c(i) + d(v) > d_{\max})$$

[8] ou de manière équivalente, sa fréquence maximale

[9] le programmeur se contente de définir la fonction de calcul locale, et c'est l'évaluation paresseuse de Jazz qui assure effectivement son évaluation en ordre topologique

Autrement dit, la retemporisation est nécessaire si pour une au moins des composantes d'entrées u, il n'y a pas de registre de resynchronisation entre u et v ($p_i(v) \leq p(u)$) et qu'en outre, le chemin critique cumulé entre u et v dépasse le seuil $d_{\max}$ ($c(i) + d(v) > d_{\max}$).

3. La profondeur de synchronisation en v est alors simplement définie comme :

$$\begin{cases} \neg r(v) & \Longrightarrow & p(v) = p_i(v) \\ r(v) & \Longrightarrow & p(v) = p_i(v) + 1 \end{cases}$$

Le chemin critique passant pas v quant à lui s'exprime comme la somme du délai de l'opérateur local et du maximum des délais en entrée pour les arêtes non-resynchronisées :

$$c(v) = d(v) + \max\{c(u)/u \in I(v) \ \wedge \ p(u) = p(v)\}$$

Si le calcul du positionnement des registres de resynchronisation est effectué localement, en revanche leur insertion effective est plus subtile. Prenons par exemple l'opérateur 2 de la figure 2.14 : sa sortie est utilisée par les sommets 3, 5 et 6, qui sont à des profondeurs de synchronisation différentes pour la retemporisation détaillée plus tôt en figure 2.8. Il convient pour ces trois opérateurs de partager les registres de retemporisation placés sur la sortie de l'opérateur 2 : ils sont alors au nombre de trois, contre $1 + 2 \times 3 = 7$ sans partage. Les structures paresseuses de Jazz sont à cet effet bien utiles : l'opérateur 2 dispose d'un tableau infini de registres de retemporisation en sortie :

```
retime  = [ output, k -> Z(retime[k-1]) ];
```

sur lequel les opérateurs retemporisés peuvent venir se "brancher" au besoin, et qui réalise ainsi naturellement le partage des registres de retemporisation en sortie de l'opérateur.

La simplicité des algorithmes choisis, combinée à l'expressivité de Jazz, permet de réaliser très simplement un assistant puissant pour le concepteur de circuit.

2.3.4 Au-delà des méthodologies conventionnelles

Les automatismes implémentés dans DSL sont très conventionnels. Cette thèse m'a aussi donné l'occasion de découvrir d'autres méthodologies qui n'ont pas été automatisées.

2.3.4.1 Bégaiement des registres

La première de ces méthode est l'entrelacement de plusieurs calculs indépendants au sein d'un même circuit en dupliquant son état interne. Plus précisément, la figure 2.15 présente la vue abstraite d'un circuit sous la forme d'une machine de Moore :

- $\mathcal{E}$ n'est autre que l'état interne du circuit, l'ensemble de ses primitives synchrones, c'est-à-dire de ses registres
- { n'est autre que la fonction purement combinatoire de transition de l'automate, qui calcule en fonction des entrées $\mathcal{I}$ et de l'état courant $\mathcal{E}$ à la fois la sortie $\mathcal{O}$ et l'état interne suivant de l'automate.

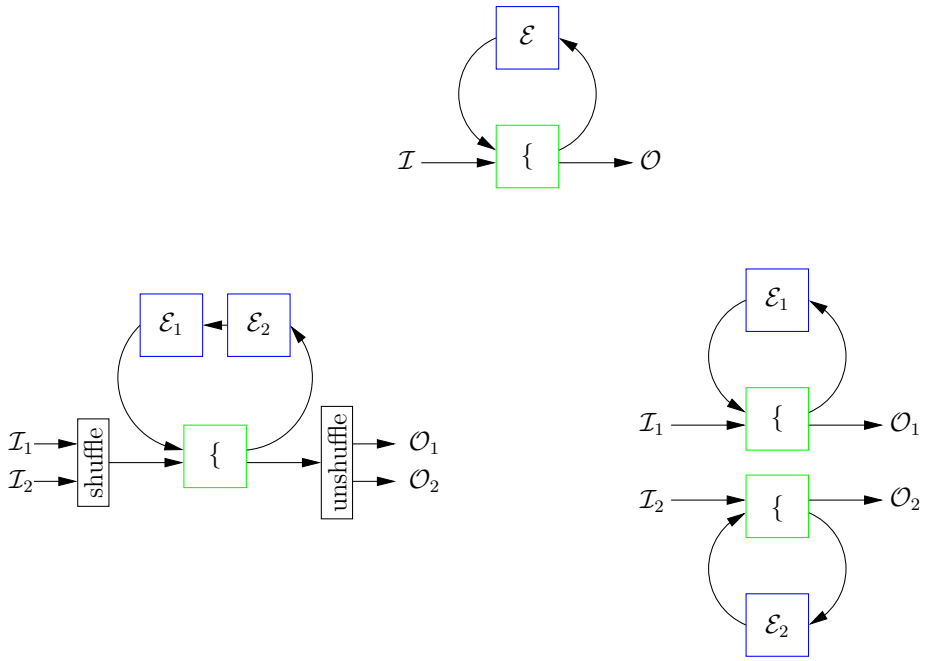


FIG. 2.15 – Bégaiement dans une machine de Moore

Partant de cette machine, on duplique son état interne $\mathcal{E}$ n fois, par exemple deux fois en $\mathcal{E}_1$ et $\mathcal{E}_2$.

On distingue dans la machine résultante deux états internes séparés qui alternent en entrée de la fonction combinatoire. Si on entrelace sur l'entree deux flux de données distincts, par la fonction *shuffle* de la figure 2.15, le circuit calcule sur sa sortie, indépendamment, deux flux de données entrelacés correspondant aux deux flux d'entrée calculés sur la machine initiale.

Par ailleurs, les registres dupliqués introduits dans la boucle de rétroaction peuvent être déplacés dans le calcul de { afin d'améliorer le chemin critique (qui, après retemporisation de l'ensemble du circuit, est situé dans la boucle de rétroaction au travers de {). Pratiquement, dans le cas d'une duplication cela signifie que la fréquence de la machine peut être théoriquement doublée. Le circuit résultant, qui peut travailler à fréquence double par rapport au circuit initial, est donc équivalent à dupliquer la machine initiale deux fois pour traiter les deux flux de donnée séparément.

Cette transformation par bégaiement est particulièrement élégante : elle ne demande aucune complexification du chemin de contrôle à l'intérieur du circuit, la complexité du contrôle étant située dans les blocs de *shuffle*.

2.3.4.2 Hiérarchie et modularité

Un aspect important de la conception moderne de systèmes complexes est l'utilisation de la modularité. La modularité est nécessaire pour réduire la complexité apparente gérée par le programmeur. Elle peut être exploitée pour réduire la complexité algorithmique gérée par la machine, c'est à dire permettre la compilation séparée sous la forme de librairies de certaines parties du code.

La modularité prend plusieurs formes : utilisation de librairies ou méthodologie de conception par objets.

Dans le cas de DSL, les algorithmes présentés dans les sections précédentes agissent sur l'ensemble de la *netliste* qui est déroulée en mémoire à partir du code source. Si le déroulement de la *netliste* est une étape nécessaire dans notre flot de conception pour générer la *netliste* bas niveau envoyée aux étapes ultérieure de compilation, diminuer le temps de compilation en modularisant les algorithmes reste un objectif louable.

Bitsizing modulaire Le problème du *bitsizing* hiérarchique est complexe : il nécessite soit de clore l'ensemble les paramètres libres du module – en ne permettant pas alors une vraie modularité – soit de laisser des paramètres libres, auquel cas il faut un moyen de décrire des contraintes symboliques dans les tailles des données aux bornes du module. Le calcul de point fixe effectué actuellement dans DSL ne semble pas adapté à ce genre de situation : il faudrait alors s'orienter plutôt vers la résolution de systèmes d'équations paramétrées avec des contraintes symboliques. Le système YICES [23] pourrait être une base pour un tel travail.

Retemporisation modulaire La modularisation de la retemporisation pose clairement le problème de savoir comment composer des modules sans connaitre *a priori* leur latence ainsi que de définir les condition de leur composition. La méthodologie des *circuits élastiques* [24] donne une réponse précise à ces questions. Toutefois, celle méthodologie est très coûteuse sur FPGA. Elle n'a pas été utilisée au cours de cette thèse.

3 Synthèse de DSL

Le chapitre précédent décrit le passage de la net-liste de haut niveau en DSL vers une représentation du circuit synchrone synthétisable grâce aux annotations de *bitsizing* portées sur chacune des variables arithmétiques entières du circuit.

Ce chapitre s'attarde sur l'étape suivante dans la réalisation de l'accélérateur matériel réalisant la fonction décrite en DSL, à savoir la translation des opérateurs sur les flots arithmétiques *bitsizés* en circuits logiques sur une technologie FPGA.

Nous décrivons la méthodologie utilisée pour la synthèse bas niveau ou *technology mapping* des opérateurs du circuit décrit en DSL. La traduction optimale dépend étroitement de la technologie cible : la plupart des FPGAs disposent de logique spécifique dédiée à la réalisation optimisée des primitives arithmétiques courantes.

Dans ce chapitre, la technologie des FPGAs est brièvement présentée en section 3.1, puis la représentation des entiers et la méthodologie d'implémentation des opérateurs sont décrites en 3.2. Enfin, à la croisée entre technologie des FPGA et théorie des opérateurs, l'implémentation de la librairie d'opérateurs de DSL optimisés pour l'architecture utilisée au cours de cette thèse – celle du Virtex II – est décrite en section 3.3.

3.1 Le circuit reconfigurable

Le Field Programmable Grid Array (FPGA) est un circuit à mi-chemin entre le matériel et le logiciel. Il s'agit d'un circuit virtuel, c'est-à-dire un circuit générique dont la fonction est définie par l'utilisateur.

Le FPGA fournit à son utilisateur les primitives suivantes :

- un ensemble de portes logiques universelles programmables
- un ensemble de registres
- une matrice d'interconnexion programmable entre ces éléments

La programmation des éléments de logique, des éléments de mémorisation et de leur interconnexion permet de réaliser au sein du FPGA tout circuit synchrone.

Cette première partie détaille la structure du FPGA utilisé au cours de cette thèse, à savoir le FPGA xc2v2000 du fournisseur Xilinx. Il s'agit du FPGA disponible sur la Pam SEPIA de chez Hewlett-Packard utilisée au cours de nos expérimentations.

3.1.1 Éléments reconfigurables

3.1.1.1 Éléments de logique combinatoire

L'élément de logique combinatoire fondamental du FPGA est une fonction logique universelle, la Look-Up Table (LUT). Une LUT est une petite mémoire accessible en lecture ou Read-Only Memory (ROM) dont le contenu est fixé lors de la configuration du circuit. Les valeurs d'une ROM décrivent par extension la table de vérité d'un opérateur logique et permettent ainsi de réaliser une fonction logique arbitraire.

Dans le cas du Virtex II, les LUTs sont des mémoires $2^4 \times 1$ bits, soit 4 bits d'adresse en entrée et un bit de résultat en sortie, qui permettent la réalisation d'une fonction logique arbitraire de $\mathbb{B}^4 \rightarrow \mathbb{B}$.

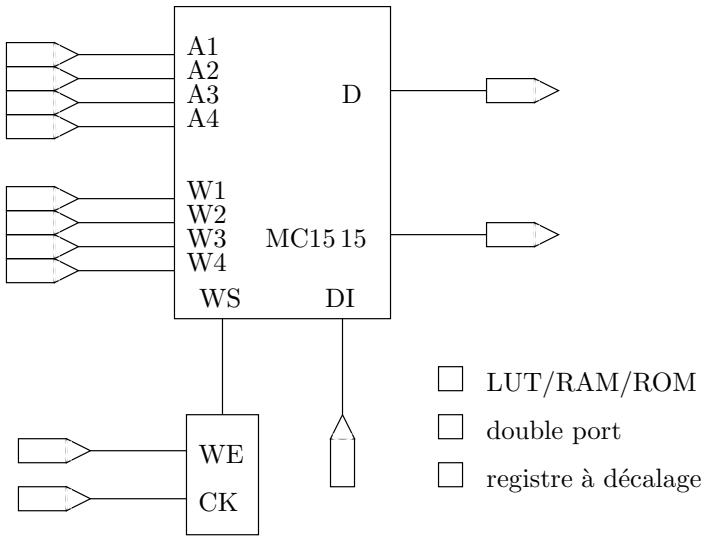


FIG. 3.1 – LUT du Virtex II

La LUT du Virtex II est présentée en figure 3.1. Ses quatre entrées sont nommées A[0 : 3] et sa sortie D ; elle contient 16 bits de mémoire interne dont le contenu est fixé lors de la configuration du FPGA. Les quatre bits d'entrée A[0 : 3] forment l'adresse de cette mémoire, et le bit D en constitue la donnée en sortie, pour réaliser une fonction logique arbitraire à quatre entrées et une sortie.

La LUT dispose de fonctions particulières additionnelles destinées à permettre l'optimisation de certaines primitives logiques courantes. Ces fonction-

nalités supplémentaires seront explorées en détail à la section 3.3 – pour leur usage lors de l'implémentation optimisée des primitives en question.

3.1.1.2 Éléments de logique synchrone

Un registre synchrone d'un bit est présent avec chaque LUT. Ce bit de mémoire permet de disposer de tous les éléments de base de la logique synchrone.

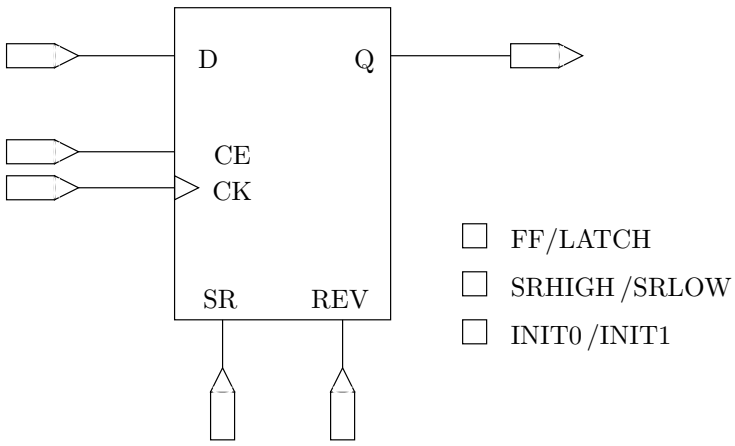


FIG. 3.2 – Registre du Virtex II

Le registre du Virtex II est présenté en figure 3.2. Dans sa configuration la plus simple, il s'agit simplement d'un registre synchrone. Son horloge est précisée par le signal entrant CK (*Clock*), la donnée entrante est échantillonnée sur le signal D (*Data*).

Le registre dispose par ailleurs de fonctions particulières courantes, comme le signal d'activation synchrone CE (*Clock Enable*), un signal de réinitialisation SR , et la possibilité de choisir la valeur initiale du registre.

3.1.1.3 Éléments d'interconnexion

Les éléments d'interconnexion programmables présents dans le FPGA permettent de relier entre elles les primitives logiques programmées dans les LUT et les registres du circuit. Leur importance est donc capitale lors de la configuration du circuit. Bien plus, leur importance sur le FPGA est considérable : l'infrastructure de routage des signaux constitue l'essentiel de la surface des puces FPGA et sa configuration représente l'essentiel (environ 90%) des informations présentes dans le fichier de configuration de la puce.

Toutefois, les fabricants de FPGA ne donnent que de vagues indications sur les possibilités d'interconnexions offertes dans leurs FPGAs ; la configuration de l'interconnexion reste donc pour l'essentiel un mystère et une partie de la technologie sur laquelle l'utilisateur de FPGA ne dispose que de peu de contrôle.

Dans la pratique pour le Virtex II, l'examen de fichiers au format Xilinx Design Language (XDL) et des représentations graphiques de la puce disponibles dans le FPGA-editor, ainsi que de certains documents Xilinx [25], peut donner une idée des ressources de routage et d'interconnexion disponibles sur le Virtex II.

Par ailleurs, le routage étant pris en charge automatiquement par les outils de compilation, il ne nous intéresse que dans le cas de l'utilisation de ressources de routage très spécifiques lors de la réalisation optimisée de certains opérateurs. Ces ressources de routage particulières sont étroitement associées à l'utilisation des ressources logiques spécialisées. Ces cas particuliers seront examinés en 3.3 ; ils sont bien documentées et les outils permettent leur utilisation de manière simple.

3.1.1.4 Éléments spécialisés

Les éléments évoqués plus haut – portes logiques, registres, et interconnexions – sont les fondements les plus généraux des circuits synchrones et permettent à eux seuls de réaliser tous les circuits synchrones imaginables.

Toutefois, la réalisation de certaines fonctions courantes avec de telles primitives serait très dispendieuse en ressources – alors qu'elles sont limitées sur une puce FPGA. Elles ne permettraient que des performances médiocres.

Afin de simultanément réduire la consommation des ressources du FPGA et d'augmenter les performances des circuits générés, le FPGA met à disposition de l'utilisateur des ressources spécialisées et optimisées pour certains usages au détriment de leur flexibilité. Ces ressources doivent réaliser un compromis entre, d'une part, leur généricité – qui permet de les utiliser dans de nombreuses circonstances – et d'autre part leur spécialisation, qui permet d'accroître les performances.

Nous décrivons pour l'instant des éléments spécialisés de très haut niveau ; les éléments spécialisés dédiés à l'implémentation des primitives arithmétiques seront explorés lors de leur utilisation en 3.3.

Mémoire dédiée Tous les FPGAs disposent de blocs de mémoire Random Access Memory (RAM) dédiée. La RAM dédiée est une mémoire très dense mais dont la bande passante est réduite par comparaison avec la mémoire disponible dans les registres synchrones.

Pour la technologie Virtex II, le bloc de RAM dispose d'une capacité de 18KiB[1] et de deux ports d'accès. Chacun des ports de cette RAM est utilisable et configurable indépendamment. Pour plus de détails, on pourra se reporter à la section "Using Block SelectRAM Memory", du chapitre 4 du guide d'utilisation du Virtex II [27].

Entrées-sorties Une dimension importante des FPGA est leur capacité à communiquer et leur souplesse à s'interfacer avec les composants extérieurs. Le FPGA dispose à cet effet de ressources d'entrée-sortie reconfigurables suivant la plupart des standards électriques.

Les entrées-sorties très versatiles des FPGAs permettent leur utilisation comme *glue logic*, à l'interface entre différents composants. Toutefois, nous ne nous intéresserons pas aux entrées-sorties bas niveau au cours de cette thèse, puisque l'infrastructure du projet SEPIA, détaillée au chapitre 4, réalise de manière transparente pour nous l'interface physique du FPGA.

[1] Les unités utilisées dans cette thèse pour les mémoires sont les unités normalisées en puissance de deux [26]

3.1.2 Structure du Virtex II

Un FPGA met à la disposition de son programmeur un certain nombre des éléments décrits plus haut. Ces éléments sont très précisément organisés : la structure des FPGA présente une grande régularité et une hiérarchie très marquée. Si tous les FPGAs répondent à une organisation rigoureuse, le détail dépend intimement du FPGA considéré.

Dans ce chapitre, la structure du Virtex II xc2v2000 – notre FPGA de prédilection – est examinée pour donner corps aux généralités de la section précédente. En particulier, les ressources effectivement disponibles sur ce FPGA sont détaillées.

3.1.2.1 Hiérarchisation du FPGA

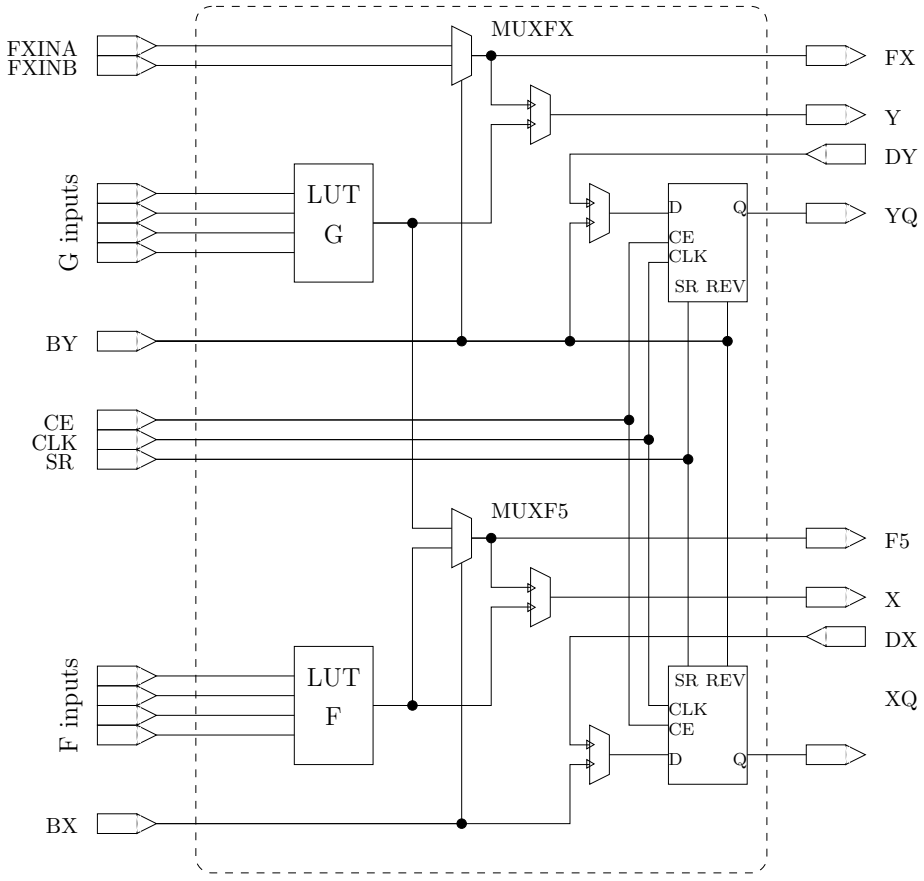


FIG. 3.3 – Architecture de la *slice* ou *tranche* du Virtex II

La tranche Les LUTs et leur registre afférent sont regroupés par deux au sein d'une *slice* en anglais dans la terminologie Xilinx – que nous traduisons par le terme *tranche* en français. Chacune de ces *tranches* comporte donc une LUT *F* et une LUT *G*, toujours suivant la terminologie Xilinx. Elle contient aussi les deux registres associés, *X* et *Y*.

La tranche dispose à son échelle de ressources particulières, éléments de routage et de logique spécialisés, dont la figure 3.3 présente une vue d'ensemble simplifiée.

Le bloc logique reconfigurable Les *tranches* sont regroupées par quatre au sein d'un Bloc Logique Configurable – Configurable Logic Block (CLB). Le CLB est figuré en 3.4. Les quatre *tranches* d'un CLB partagent les mêmes matrices d'interconnexion aux infrastructures de routage de fond de panier. Ce sont ces matrices qui contiennent les Points d'Interconnexion Programmable – Programmable Interconnect Points (pips) qui font la liaison entre les ressources globales et locales de routage.

Le CLB dispose encore, à son échelle, de ressources logiques et de routage locales.

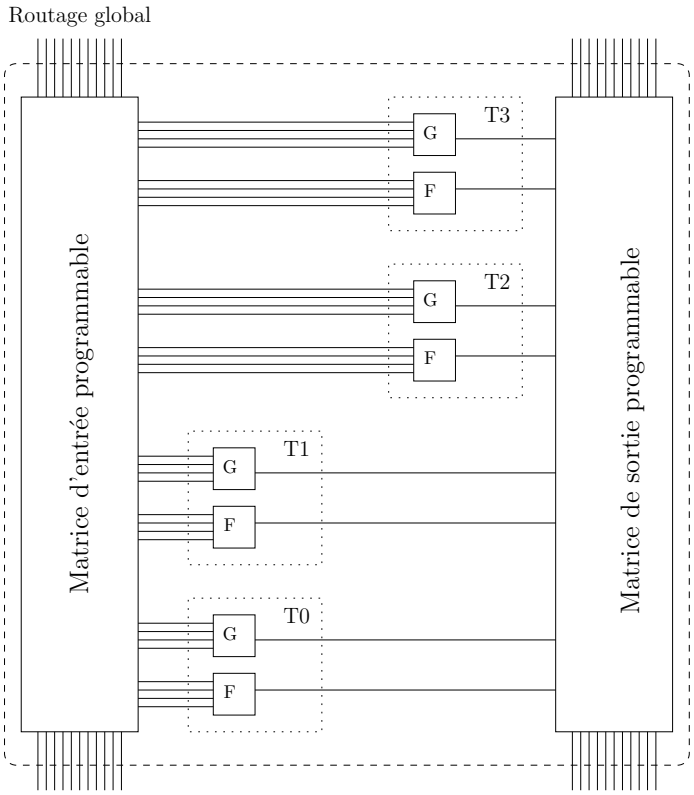


FIG. 3.4 – CLB du Virtex II

3.1.2.2 Dimensions du xc2v2000

Le Virtex II est une grille très régulière qui entrelace les CLB, les blocs RAM et les blocs d'entrées-sorties au sein de la matrice de routage du FPGA.

Pour les FPGA plus récents, la tendance semble être aujourd'hui à l'intégration d'un nombre croissant de blocs fonctionnels spécialisés en plus de ces blocs de base. Il s'agit par exemple des cœurs de processeurs PowerPC présents dans

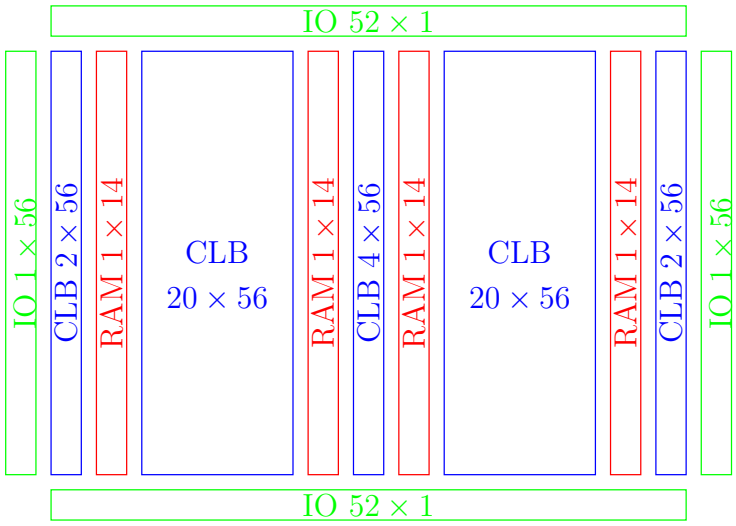


FIG. 3.5 – Grille du xc2v2000

les Virtex II Pro, de logique spécialisée pour configurer les blocs RAM en FIFOs dans les Virtex-4, voire des composantes bas niveau du bus PCI-express (PCI-e) dans certains Virtex-5.

CLB			blocs RAM	
L × C	Tranches	DistRAM	L × C	total
56 × 48	10752	336 Kib	14 × 4	1008 Kib

TAB. 3.1 – Ressources disponibles sur le xc2v2000

Les quantités de chacun de ces éléments dans le xc2v2000 utilisé au cours de cette thèse sont reportés en table 3.1 ; la figure 3.5 présente une vue d'ensemble de leur répartition géométrique.

3.1.3 Configurabilité

Le Virtex II est un FPGA assez ancien de la gamme Xilinx Virtex. La famille Virtex a été introduite en 1998, et le Virtex II est la deuxième génération de puces de cette famille. La plus récente est le Virtex-5.

Il s'agit, comme la grande majorité des FPGA actuels, d'une famille de circuits dont la mémoire de configuration est une Static Random Access Memory (SRAM) volatile. Ainsi, un *bitstream* de configuration doit être chargé dans le FPGA à chaque démarrage du circuit.

Le Virtex II dispose d'une capacité de reconfiguration partielle, qui permet de reprogrammer partiellement le FPGA pendant son fonctionnement. Cette capacité n'a pas été exploitée, mais elle peut être mise à profit pour réaliser des circuits reconfigurables dynamiquement.

Par ailleurs, le circuit dispose d'une capacité d'accès à sa propre configuration, en lecture comme en écriture, au travers du Port d'Accès à la Configuration Interne – Internal Configuration Access Port (ICAP). SEPIA permet de faire

usage de ce port pour examiner l'état interne du circuit à des fins de déverminage des circuits.

Les caractéristiques de (re)-configurabilité du Virtex II sont présentées en détail dans la documentation officielle Xilinx ([25],[27]).

3.2 Flots d'entiers et opérateurs

La question de la représentation des entiers en tableaux de bits se pose à la fois comme un problème de *typage* et comme un problème de *synthèse*. En effet la conversion d'entiers en tableaux de bits est nécessaire au sein du langage pour donner un sens aux opérateurs booléens (décalages de bits, opérateurs logiques bit-à-bit) ; elle est indispensable aussi lors de la synthèse du circuit.

Le choix fait pour DSL est de fixer une correspondance canonique [28] entre entiers et tableaux de bits. Il n'est par conséquent pas nécessaire de dissocier les deux types : le typage s'en trouve simplifié. Il suffit de définit un type unifié entier sur lequel les opérateurs de la logique booléenne disposent d'une sémantique claire. Esterel, avec son système de types plus riche, a fait le choix de dissocier les deux représentations ([3], chapitre 4). La conversion entre type entier et type tableau de bits doit alors être explicite. Mais la dissociation augmente la souplesse offerte à l'utilisateur – en particulier, elle lui offre la possibilité de définir une représentation binaire ad-hoc des entiers.

3.2.1 Représentation des entiers

Après l'analyse d'intervalle effectuée par DSL, les variables entières de précision infinie exprimées par DSL sont réduites à un intervalle de confiance borné. L'étape d'analyse statique d'intervalle indique que la borne est valide *a priori* pour représenter l'ensemble des valeurs portées par la variable au cours du temps et pour toutes les exécutions possibles du circuit.

De telles bornes permettent la représentation de la variable par un tableau fini de bits. La représentation utilisée pour les entiers est fixe ; il s'agit de la représentation standard en complément à deux des entiers 2-adiques, une représentation particulièrement adaptée à la représentation des circuits [28].

Plus précisément, lors de l'étape de *bitsizing* chaque variable v est annotée par un relatif $\langle \mathsf{v} \rangle$:

– si $\langle \mathsf{v} \rangle > 0$, alors la quantité entière représentée par la variable est non signée, sa représentation est faite par un tableau v de taille $\langle \mathsf{v} \rangle$, et la valeur $\mathbf{v} \in [0, 2^{\langle \mathsf{v} \rangle} - 1]$ portée par la variable est donnée par la formule :

$$\mathbf{v} = \sum_{i=0}^{i<\langle \mathsf{v} \rangle} \mathsf{v}[i].2^i$$

– si $\langle \mathsf{v} \rangle < 0$, alors la quantité entière représentée par la variable est signée, sa représentation est réduite à un tableau de bits v de taille $|\langle \mathsf{v} \rangle|$, mais cette fois la valeur $\mathbf{v} \in [-2^{|\langle \mathsf{v} \rangle|-1}, 2^{|\langle \mathsf{v} \rangle|-1} - 1]$ portée par la variable est donnée par la formule :

$$\mathbf{v} = -\mathsf{v}[|\langle \mathsf{v} \rangle| - 1].2^{|\langle \mathsf{v} \rangle|-1} + \sum_{i=0}^{i<|\langle \mathsf{v} \rangle|-1} \mathsf{v}[i].2^i$$

Le bit $\mathsf{v}[|\langle \mathsf{v} \rangle| - 1]$ est qualifié de bit de signe : la valeur représentée par le tableau de bits est négative lorsque ce bit est positionné à un. La valeur signée est congrue à la valeur obtenue par interprétation non-signée du tableau de bits :

$$\mathbf{v} \equiv \sum_{i=0}^{i<|\langle \mathsf{v} \rangle|} \mathsf{v}[i].2^i \bmod 2^{|\langle \mathsf{v} \rangle|}$$

autrement dit les interprétations signées et non-signées du tableau de bits sont identiques modulo $2^{|\langle \mathsf{v} \rangle|}$; l'interprétation signée n'est autre que l'unique représentant de la classe dans l'intervalle $[-2^{|\langle \mathsf{v} \rangle|-1}, 2^{|\langle \mathsf{v} \rangle|-1} - 1]$, la valeur non-signée est l'unique représentant de la même classe dans l'intervalle $[0, 2^{|\langle \mathsf{v} \rangle|} - 1]$.

Cette représentation en complément à deux permet la synthèse très aisée des opérateurs arithmétiques sur les variables grâce à son interprétation modulaire.

3.2.2 Méthodologie d'implémentation des opérateurs

Les opérateurs synthétisés de DSL sont définis par un jeu de paramètres :
- la fonction qu'ils réalisent ;
- le *bitsizing* de leurs opérandes entiers et de leur sortie.

La synthèse de l'opérateur à partir de ces paramètres suit trois exigences, la première de correction, la seconde d'économie, et la troisième, de performances.

3.2.2.1 Correction

La correction des opérateurs s'appuie essentiellement sur la simplicité de l'arithmétique en complément à deux. Cette arithmétique permet de générer des circuits très simples qui unifient sans cas particulier complexe le traitement du bit de signe par arithmétique modulaire.

Si DSL traite d'entiers en précision infinie, qui sont ensuite restreints par analyse d'intervalle, on peut procéder de même pour les opérateurs. En effet, dans cette arithmétique, l'opérateur final est simplement constitué par la restriction finie d'opérateurs infinis idéaux sur les entiers 2-adiques.

Ce traitement uniforme des opérateurs permet de générer les circuits par un code très simple et réutilisable ; en conséquence il est particulièrement facile de s'assurer de leur correction.

Concrètement, la correction de chaque opérateur est mise à l'épreuve par un jeu de tests exhaustif pour des paramètres d'implémentation variés.

3.2.2.2 Économies d'énergie

La minimisation de la consommation électrique est une préoccupation majeure dans la conception actuelle de circuits, notamment pour les circuits dans le monde de l'informatique embarquée. C'est le cas pour les Application Specific Integrated Circuit (ASIC)s mais aussi pour les FPGA.

La technique du *clock gating*, déjà évoquée en 2.1.3.2, est la méthode la plus courante pour réaliser des économies d'énergie. Son principe est de figer les registres synchrones dans une partie du circuit. Les opérateurs purement combinatoires ne sont pas concernés par ces optimisations ; les opérateurs synchrones de 3.3.3, en revanche, doivent les prendre en compte.

D'une part ces opérateurs doivent prévoir un signal d'*enable* qui permet de figer leur état interne, afin de pouvoir appliquer la méthode du *clock gating* dans les circuits DSL.

D'autre part les opérateurs peuvent être conçus pour minimiser le nombre de transitions de données lorsqu'ils sont actifs. Dans l'exemple du registre à décalage en 3.3.3.3, pour le registre à décalage naïf l'ensemble des registres bascule lorsque l'horloge est active. Au contraire l'implémentation du registre à décalage dans une RAM permet de réduire le nombre de changements d'état du circuit à quelques mots par cycle. La consommation électrique dynamique du circuit est réduite d'autant.

Cette préoccupation, si elle n'a pas été centrale lors de la réalisation de DSL, doit être prise en compte finement pour des applications industrielles.

3.2.2.3 Économies d'espace

Les unités arithmétiques et logiques entières d'un processeur travaillent sur un nombre de bits fixe, et la taille des mots manipulés par un processeur est alignée sur une puissance paire d'octets[2].

Au contraire, dans une implémentation matérielle, le principe d'économie gouverne, et chaque bit compte. Ce principe est à prendre en compte dans la réalisation des opérateurs, pour éviter la consommation de ressources logiques ou de ressources mémoire inutiles.

La méthodologie d'une implémentation économe des opérateurs consiste à ne réaliser que le calcul des bits nécessaires au résultat. Cet ajustement est précisément réalisable grâce aux annotations de *bitsizing* sur le résultat de l'opération. De même les mémoires peuvent-elles être ajustées au bit près. Dans ces deux cas, on a veillé à ce que l'usage des ressources reste minimal, au plus juste en fonction des paramètres d'implémentation.

En plus de ce principe général d'économie, l'usage de primitives logiques idoines – il s'agit alors d'optimisations spécifiques à la technologie cible – permet encore de diminuer les ressources consommées par les opérateurs.

3.2.2.4 Performance

L'opérateur théorique correct et compact doit en plus être performant. Pour un circuit, la performance est mesurée comme la fréquence maximale de fonctionnement de l'opérateur ou, de manière équivalente, la longueur du chemin critique qui le traverse.

L'usage de ressources spécifiques – qu'il s'agisse de ressources de routage ou de ressources logiques – permet d'atteindre les performances optimales.

Concrètement, on peut mesurer pour chaque opérateur sa fréquence maximale de fonctionnement en fonction de ses paramètres de synthèse ; cette mesure systématique est très fastidieuse, et les exemples d'utilisation de DSL du chapitre 5, pour lesquels la retemporisation automatique aboutit à des vitesses de fonctionnement maximales, fournissent la preuve par l'usage de la très bonne tenue des primitives réalisées dans le dorsal de synthèse DSL.

[2] voir par exemple les types d'entiers ISO `int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`

3.3 Librairie d'opérateurs

Cette section détaille l'implémentation de la librairie dorsale d'opérateurs bas niveau de DSL pour la cible Virtex II. Elle s'inspire de la présentation faite pour la librairie BigNum [29].

La description de chacun des opérateurs précise l'usage des ressources spécifiques au Virtex II pour atteindre les performances maximales et la consommation de logique minimale sur cette architecture.

Les opérateurs sont répartis en trois catégories. La première décrit les fonctions logiques simples – opérations logiques bit-à-bit et opérations logiques génériques sous la forme de ROM. La seconde décrit les opérateurs arithmétiques, additionneurs et dérivés, multiplieurs et décaleurs. Si les deux premières catégories regroupent des opérateurs purement combinatoires, la troisième regroupe tous les opérateurs synchrones, c'est-à-dire ceux contenant des registres pour lesquels une horloge doit être fournie.

3.3.1 Opérateurs logiques

La présentation des opérateurs logiques décrit un opérateur isolé et idéal. Dans la pratique, les optimisations présentes dans les outils de synthèse peuvent condenser plusieurs opérateurs unaires ou binaires bit-à-bit au sein d'une seule LUT – qui peut contenir un opérateur logique d'arité 4. Ainsi, l'opérateur n'apparaît par toujours explicitement dans le circuit synthétisé final : ces optimisations sont à prendre en compte lors de l'examen final du circuit dans lequel certaines variables DSL peuvent avoir disparu.

3.3.1.1 Négation logique

$$Neg(\langle \mathsf{o} \rangle, \langle \mathsf{a} \rangle)(a)$$

La primitive Neg génère un opérateur qui calcule la *négation logique* bit-à-bit de son opérande a *bitsizé* sur $\langle \mathsf{a} \rangle$ bits. Le résultat o est *bitsizé* sur $\langle \mathsf{o} \rangle$ bits en sortie.

Chaque bit de la sortie o est calculé soit par une LUT dont une seule entrée est utilisée, soit le cas échéant par extension du bit de signe de o.

3.3.1.2 Et logique

$$And(\langle \mathsf{o} \rangle, \langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle)(a, b)$$

La primitive And génère un opérateur qui calcule le résultat d'un *et logique* bit-à-bit entre ses opérandes a et b *bitsizés* sur respectivement $\langle \mathsf{a} \rangle$ et $\langle \mathsf{b} \rangle$ bits. Le résultat o est *bitsizé* sur $\langle \mathsf{o} \rangle$ bits en sortie.

Chaque bit $\mathsf{o}[i]$ de la sortie o est calculé par une LUT dont seules deux entrées sont alors utilisées, qui reçoivent les bits $\mathsf{a}[i]$ et $\mathsf{b}[i]$ des opérandes en entrée. Le cas échéant, l'extension du bit de signe des tableaux de bits a ou b en entrée fournit les bits nécessaires en entrée de la LUT.

3.3.1.3 Ou logique

$$Or(\langle \mathsf{o} \rangle, \langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle)(a, b)$$

La primitive *Or* génère un opérateur qui calcule le résultat d'un *ou logique* bit-à-bit entre ses opérandes a et b *bitsizés* sur respectivement $\langle \mathsf{a} \rangle$ et $\langle \mathsf{b} \rangle$ bits. Le résultat o est *bitsizé* sur $\langle \mathsf{o} \rangle$ bits en sortie.

La méthodologie présentée 3.3.1.2 est utilisée à l'identique pour le calcul des bits de sortie de l'opérateur.

3.3.1.4 Ou-exclusif logique

$$Xor(\langle \mathsf{o} \rangle, \langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle)(a, b)$$

La primitive *Xor* génère un opérateur qui calcule le résultat d'un *ou logique exclusif* bit-à-bit entre ses opérandes a et b *bitsizés* sur respectivement $\langle \mathsf{a} \rangle$ et $\langle \mathsf{b} \rangle$ bits. Le résultat o est *bitsizé* sur $\langle \mathsf{o} \rangle$ bits en sortie.

La méthodologie présentée 3.3.1.2 est utilisée à l'identique pour le calcul des bits de sortie de l'opérateur.

3.3.1.5 Fonction ROM

$$Lut(initval[])(\langle \mathsf{o} \rangle, \langle \mathsf{a} \rangle)(a)$$

La primitive *Lut* – où $\langle \mathsf{a} \rangle > 0$ – génère une ROM de $2^{\langle \mathsf{a} \rangle} \times |\langle \mathsf{o} \rangle|$ bits. Elle est initialisée par les valeurs contenues dans le tableau $initval[]$, et présente sur sa sortie o le mot à l'adresse a, $initval[\mathbf{a}]$. La LUT du Virtex II réalise naturellement toute ROM de 4 bits d'adresse. Ces LUTs peuvent être combinées de deux manières pour obtenir une ROM plus grande :

- la combinaison parallèle de ROMs contenant des mots de 1 bit étend au plus juste la taille des mots stockés pour atteindre les $|\langle \mathsf{o} \rangle|$ bits de donnée requis en sortie ;
- le multiplexage des ROMs étend l'adressage de la ROM jusqu'à atteindre les $\langle \mathsf{a} \rangle$ bits d'adresse nécessaires en entrée.

Le Virtex II dispose de multiplexeurs spécifiques qui permettent d'associer facilement les LUTs pour augmenter la profondeur des ROM. Cette combinaison réalisée à plusieurs échelles de la hiérarchie du circuit permet de regrouper les LUT de manière optimisée jusqu'à 8 bits d'adresse (256 mots).

- Au niveau de la tranche, un multiplexeur spécifique MUXF5 permet d'associer deux LUTs de $2^4 \times 1$ bits pour former une LUT de $2^5 \times 1$ bits. Ce multiplexeur ne consomme pas de LUT et son contrôle est assuré par un routage dédié qui ne consomme pas de connexion.
- Au niveau du CLB, les quatre tranches constituées en ROM $2^5 \times 1$ peuvent être associées, par deux à l'aide du MUXF6, puis par quatre grâce au MUXF7, pour atteindre 7 bits d'adresse. Un réseau un peu complexe de multiplexeurs réalise cette association, qui est figurée en 3.6 ; ici encore le routage de commande des multiplexeurs est spécifique.
- Enfin, deux CLB adjacents peuvent être regroupés grâce à un dernier multiplexeur dédié, le MUXF8, pour atteindre une ROM $2^8 \times 1$ de huit bits d'adresse.

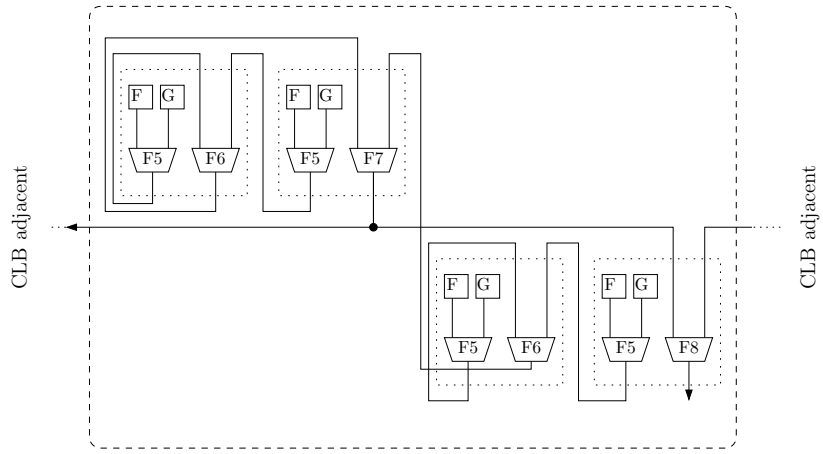


FIG. 3.6 – Multiplexeurs spécialisés du Virtex II

Au-delà de 256 mots, la combinaison des ROMs est réalisée benoîtement à l'aide de LUTs pour les multiplexeurs. Pour les ROMs très importantes, le concepteur peut choisir d'utiliser un bloc de RAM – qui sera décrit en 3.3.3.2 – mais leur technologie ne permet pas d'en faire une fonction purement combinatoire. C'est la raison pour laquelle la synthèse automatique de DSL ne fait pas usage de cette possibilité, qui est laissée à l'appréciation du concepteur du circuit.

3.3.1.6 Multiplexeur

$$Mux(\langle \mathsf{o} \rangle, \langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle)(c, a, b)$$

La primitive Mux génère un opérateur qui calcule le résultat d'un *multiplexage* entre ses opérandes a et b *bitsizés* sur respectivement $\langle \mathsf{a} \rangle$ et $\langle \mathsf{b} \rangle$ bits. Le résultat o est *bitsizé* sur $\langle \mathsf{o} \rangle$ bits en sortie. Si le bit c est positionné à un, alors $\mathbf{o} = \mathbf{a}$, sinon $\mathbf{o} = \mathbf{b}$. Le multiplexeur est construit simplement grâce à l'équation logique suivante :

$$\mathsf{o}[i] = (c \wedge \mathsf{a}[i]) \vee (\neg c \wedge \mathsf{b}[i])$$

qui est implémentée dans une LUT.

Cet opérateur de multiplexage peut être généralisé à plusieurs bits de contrôle et un tableau d'entrées pour permettre l'utilisation des MUXF5, MUXF6, MUXF7 et MUXF8 optimisés, comme pour la ROM en 3.3.1.5. Une telle extension n'a toutefois jamais été nécessaire dans les exemples traités.

3.3.2 Opérateurs arithmétiques

Contrairement aux opérateurs logiques qui font usage de ressources simples, les LUTs, et pour lesquels l'optimisation est déléguée aux outils de synthèse du vendeur Xilinx, les opérateurs arithmétiques font largement usage d'optimisations manuelles ; en conséquence de quoi ils ne sont que marginalement soumis

à optimisation de la part des outils de synthèse de Xilinx, qui respectent scrupuleusement les directives du concepteur.

3.3.2.1 Constante

$$Const(\langle \mathsf{a} \rangle, \mathbf{a})$$

La primitive *Const* génère un circuit d'arité nulle, un simple tableau de bits qui représente la valeur constante **a** suivant la convention de la section 3.2, sur $|\langle \mathsf{a} \rangle|$ bits signés si $\langle \mathsf{a} \rangle < 0$, non-signés si $\langle \mathsf{a} \rangle > 0$, en complément à deux.

Cette primitive a un coût nul en termes de logique – les constantes sont propagées et intégrées au sein des LUT qui les utilisent.

3.3.2.2 Additionneur

$$Add(\langle \mathsf{o} \rangle, \langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle)(a, b)$$

La primitive *Add* génère un additionneur qui calcule la somme des opérandes a et b *bitsizés* sur respectivement $\langle \mathsf{a} \rangle$ et $\langle \mathsf{b} \rangle$ bits. Le résultat o est *bitsizé* sur $\langle \mathsf{o} \rangle$ bits en sortie.

Unité de calcul Le calcul d'un bit de sortie se fait grâce à l'utilisation d'un additionneur complet qui additionne ses trois bits d'entrée a, b et c pour produire un bit de somme s et un bit de retenue r en sortie.

L'équation qui le gouverne est bien connue :

$$a + b + c = 2.s + r$$

Logique dédiée La tranche Xilinx comporte des portes logiques spécialisées dont l'utilisation permet d'épauler une LUT pour réaliser un additionneur complet. La logique additionnelle est constituée d'un multiplexeur et d'une porte xor disposés aux côtés de la LUT. La figure 3.7 illustre ce mode de configuration de la tranche, qui réalise ainsi le calcul du bit de sortie $\mathsf{o}[i]$:

$$\begin{aligned} \mathrm{F}_1 &= \mathsf{a}[i] \\ \mathrm{F}_2 &= \mathsf{b}[i] \\ \mathrm{Fout} &= \mathrm{F}_1 \oplus \mathrm{F}_2 \\ \mathrm{Y} &= \mathrm{XORCY}(\mathrm{Fout}, \mathrm{CIN}) \\ \mathrm{COUT} &= \mathrm{MUXCY}(\mathrm{Fout}, \mathrm{CIN}, \mathrm{F}_2) \\ \mathsf{o}[i] &= \mathrm{Y} \end{aligned}$$

Cette logique additionnelle permet d'implémenter des additionneurs très compacts – une seule LUT est consommée par bit de sortie, quand une implémentation naïve en demanderait deux.

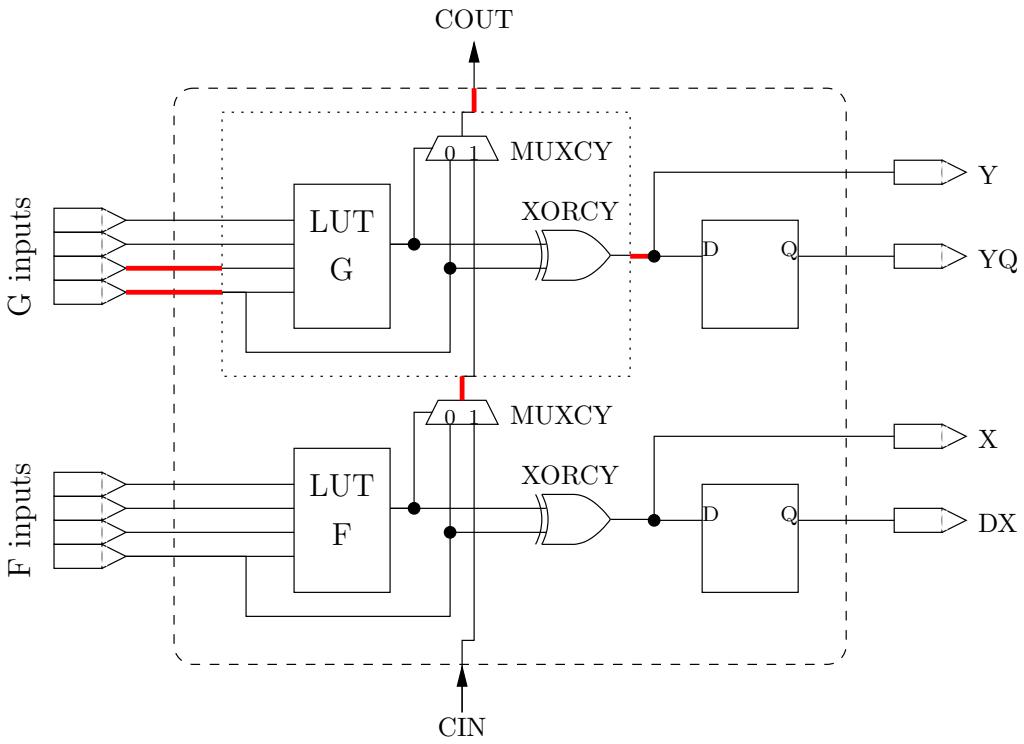


FIG. 3.7 – Logique d'additionneur spécialisée pour le Virtex II

Routage dédié Par ailleurs, le chemin qui propage la retenue sortante COUT vers le signal CIN de l'additionneur complet suivant est un "chemin de retenue rapide" selon la terminologie Xilinx. Il s'agit d'une ressource de routage dédiée qui n'est coupée par aucun pips, et qui permet par conséquent des délais de propagation très courts.

Le chemin de retenue dédié permet de réaliser des additionneurs extrêmement rapides – en dépit d'une implémentation naïve avec chemin critique linéaire en fonction du nombre de bits de sortie $|\langle \mathsf{o} \rangle|$. Il n'a jamais, dans aucun des circuits présentés au chapitre 5, été nécessaire d'utiliser des implémentations plus complexes des additionneurs, dont le chemin critique serait théoriquement meilleur, mais qui ne sont pertinentes que pour l'addition de très grands nombres.

3.3.2.3 Moins unaire

$$Minus(\langle \mathsf{o} \rangle, \langle \mathsf{a} \rangle)(a)$$

La primitive *Minus* génère un moins unaire qui calcule l'opposé de son opérande a *bitsizé* sur $\langle \mathsf{a} \rangle$ bits. Le résultat o est *bitsizé* sur $\langle \mathsf{o} \rangle$ bits en sortie.

Le circuit de négation utilise la relation suivante en arithmétique de complément à deux :

$$-\mathbf{a} = \neg\mathbf{a} + 1$$

et s'appuie sur l'additionneur complet pour réaliser la fonction, grâce à une configuration un peu différente de la LUT :

$$\begin{aligned} \mathrm{F}_1 &= \mathsf{b}[i] \\ \mathrm{Fout} &= \neg\,\mathrm{F}_1 \\ \mathrm{Y} &= \mathrm{XORCY}(\mathrm{Fout}, \mathrm{CIN}) \\ \mathrm{COUT} &= \mathrm{MUXCY}(\mathrm{Fout}, \mathrm{CIN}, \mathrm{F}_2) \\ \mathsf{o}[i] &= \mathrm{Y} \end{aligned}$$

L'incrément de 1 est réalisé par exemple en injectant une retenue entrante initiale dans le circuit.

3.3.2.4 Soustracteur

$$Sub(\langle \mathsf{o} \rangle, \langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle)(a, b)$$

La primitive *Sub* génère un soustracteur qui calcule la différence des opérandes a et b *bitsizés* sur respectivement $\langle \mathsf{a} \rangle$ et $\langle \mathsf{b} \rangle$ bits. Le résultat o est *bitsizé* sur $\langle \mathsf{o} \rangle$ bits en sortie.

Sa réalisation s'appuie sur la formule :

$$\mathbf{a} - \mathbf{b} = \mathbf{a} + \neg\mathbf{b} + 1$$

L'additionneur complet est légèrement modifié – le second opérande est nié – et une retenue entrante initiale est injectée sur le chemin de retenue rapide pour réaliser l'incrément.

$$\begin{aligned} \mathrm{F}_1 &= \mathsf{a}[i] \\ \mathrm{F}_2 &= \mathsf{b}[i] \\ \mathrm{Fout} &= \mathrm{F}_1 \oplus \neg\,\mathrm{F}_2 \\ \mathrm{Y} &= \mathrm{XORCY}(\mathrm{Fout}, \mathrm{CIN}) \\ \mathrm{COUT} &= \mathrm{MUXCY}(\mathrm{Fout}, \mathrm{CIN}, \mathrm{F}_1) \\ \mathsf{o}[i] &= \mathrm{Y} \end{aligned}$$

3.3.2.5 Additionneur-soustracteur

$$AddSub(\langle \mathsf{o} \rangle, \langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle)(s, a, b)$$

La primitive *AddSub* génère un additionneur/soustracteur qui calcule la différence ou la somme des deux opérandes a et b *bitsizés* sur respectivement $\langle \mathsf{a} \rangle$ et $\langle \mathsf{b} \rangle$ bits, en fonction du bit de décision s : si s est nul, la somme est calculée, si s vaut 1, la différence est calculée. Le résultat o est *bitsizé* sur $\langle \mathsf{o} \rangle$ bits en sortie.

Les paragraphes précédents ont mis en évidence comment calculer une somme ou une différence. Il est aisé de rendre le circuit plus souple en ajoutant dans l'additionneur complet le bit de décision s :

$$
\begin{aligned}
\mathrm{F}_1 &= \mathsf{a}[i] \\
\mathrm{F}_2 &= \mathsf{b}[i] \\
\mathrm{F}_3 &= s \\
\mathrm{Fout} &= (\mathrm{F}_3 \wedge (\mathrm{F}_1 \oplus \neg\, \mathrm{F}_2)) \vee (\neg\, \mathrm{F}_3 \wedge (\mathrm{F}_1 \oplus \mathrm{F}_2)) \\
\mathrm{Y} &= \mathrm{XORCY}(\mathrm{Fout}, \mathrm{CIN}) \\
\mathrm{COUT} &= \mathrm{MUXCY}(\mathrm{Fout}, \mathrm{CIN}, \mathrm{F}_1) \\
\mathsf{o}[i] &= \mathrm{Y}
\end{aligned}
$$

Par ailleurs la retenue entrante initiale n'est autre que s : ainsi le circuit réalise au choix la fonction d'addition ($s = 0$) ou de soustraction ($s = 1$).

3.3.2.6 Comparateur

$$CompareLt(\langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle)(a, b)$$

La primitive *AddSub* génère un comparateur qui calcule un bit résultat de la comparaison $\mathbf{a} < \mathbf{b}$: si l'assertion est vérifiée, alors le bit de sortie $\mathbf{o}$ est positionné à un ; si au contraire $\mathbf{b} \leq \mathbf{a}$ alors $\mathbf{o} = 0$.

Le comparateur s'appuie simplement sur la relation :

$$\mathbf{a} < \mathbf{b} \iff \mathbf{a} - \mathbf{b} < 0$$

et calcule le signe de la différence de ses deux opérandes grâce à un circuit de soustraction. Il bénéficie ainsi de la compacité et de la performance liées à l'utilisation du chemin de retenue rapide.

3.3.2.7 Égalité

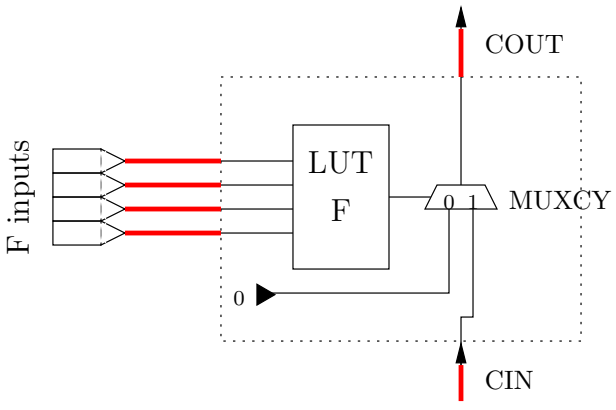


FIG. 3.8 – Conjonction rapide par le chemin de retenue du Virtex II

$$Eq(\langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle)(a, b)$$

La primitive *Eq* génère un comparateur qui calcule un bit résultat de la comparaison $\mathbf{a} = \mathbf{b}$.

L'opérateur d'égalité présente un exemple d'utilisation du chemin de retenue rapide pour propager autre chose qu'une retenue arithmétique traditionnelle.

L'opération à réaliser ici est un *et logique* large sur le résultat de la comparaison bit à bit des opérandes **a** et **b** :

$$\mathbf{a} = \mathbf{b} \Leftrightarrow \bigwedge_{i=0}^{i<\max(|\langle \mathsf{a} \rangle|,|\langle \mathsf{b} \rangle|)} \mathsf{a}[i] = \mathsf{b}[i]$$

Pour réaliser cette opération, on utilise le multiplexeur spécialisé d'une manière particulière pour calculer sur la retenue sortante COUT un *et logique* entre la sortie Fout de la LUT et la retenue entrante CIN :

$$\text{Fout} \wedge \text{CIN} = \text{MUXCY}(\text{Fout}, \text{CIN}, 0)$$

La configuration de la figure 3.8 réalise ce "et rapide".

La LUT est configurée pour calculer le résultat de la comparaison de deux bits de chaque opérande :

$$\begin{aligned} \text{F}_1 &= \mathsf{a}[i] \\ \text{F}_2 &= \mathsf{b}[i] \\ \text{F}_3 &= \mathsf{a}[i+1] \\ \text{F}_3 &= \mathsf{b}[i+1] \\ \text{Fout} &= \neg(\text{F}_1 \oplus \text{F}_2) \wedge \neg(\text{F}_3 \oplus \text{F}_3) = \neg((\text{F}_1 \oplus \text{F}_2) \vee (\text{F}_3 \oplus \text{F}_3)) \end{aligned}$$

L'utilisation du chemin de retenue permet de réaliser une conjonction logique large et rapide. À l'instar de l'additionneur, une implémentation plus intelligente de la conjonction par un arbre de profondeur logarithmique est *asymptotiquement* plus efficace. En pratique, la propagation le long du chemin de retenue rapide donne de meilleurs résultats pour l'implémentation naïve de profondeur linéaire – au moins pour les tailles d'opérandes qui nous intéressent.

3.3.2.8 Multiplieurs

$$Mult(\langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle)(a, b)$$

La primitive *Mult* génère un multiplieur qui calcule le produit de ses deux opérandes a et b *bitsizés* sur respectivement $\langle \mathsf{a} \rangle$ et $\langle \mathsf{b} \rangle$ bits. Le résultat o est *bitsizé* sur $\langle \mathsf{o} \rangle$ bits en sortie.

Multiplieurs dédiés Le Virtex II contient des multiplieurs sur 18×18 bits signés en entrée pour 36 bits en sortie. Ils sont en nombre limité et accolés à chaque bloc de RAM avec lesquels ils partagent des ressources de routage, ce qui rend leur utilisation concurrente délicate. Sur le xc2v2000, ils sont au nombre de 56, autant que de blocs RAM. Ils peuvent être pipelinés pour augmenter leur performance.

Multiplieur standard Pour obtenir des multiplieurs plus larges et/ou se soustraire aux contraintes d'utilisation des multiplieurs dédiés, nous avons implémenté une version ad-hoc de multiplieurs sur la tranche du Virtex II.

L'algorithme retenu est ici encore l'algorithme naïf de décalage et somme. Dans cet algorithme qui imite l'algorithme de multiplication à la main, le i$^{\text{ème}}$ bit $\mathsf{a}[i]$ du multiplicande $\mathbf{a}$ est utilisé comme bit de contrôle de l'addition de $\mathbf{b} \ll i$ à la somme partielle $\mathbf{p_i}$ pour obtenir la somme partielle suivante $\mathbf{p_{i+1}}$[3] :

$$\begin{aligned} p_0 &= 0 \\ p_{k+1} &= p_k + \mathsf{a}[k].(\mathbf{b} \ll k) \\ \Rightarrow p_{|\langle \mathsf{a} \rangle|} &= \mathbf{b} \times \mathbf{a} \end{aligned} \tag{3.1}$$

Le calcul d'un bit de sortie se fait grâce à l'utilisation de l'additionneur complet en 3.3.2.2 modifié pour accepter un bit de contrôle m qui valide l'un des bits d'entrée ; l'équation qui le gouverne est la suivante :

$$a + b.m + c = 2.s + r$$

La tranche Xilinx dispose précisément de la logique nécessaire pour modifier l'additionneur complet optimisé afin d'y ajouter le bit de contrôle m – grâce à une porte *et* dédiée située entre deux entrées de la LUT à destination de l'entrée 0 du multiplexeur MUXCY.

Le circuit de la figure 3.9 réalise ainsi les équations logiques suivantes :

$$\begin{aligned} \mathrm{F}_1 &= \mathsf{b}[i] \\ \mathrm{F}_2 &= \mathsf{a}[k] \\ \mathrm{F}_3 &= \mathsf{p_k}[i] \\ \mathrm{Fout} &= \mathrm{F}_3 \oplus (\mathrm{F}_1 \wedge \mathrm{F}_2) \\ \mathrm{Y} &= \mathrm{XORCY}(\mathrm{Fout}, \mathrm{CIN}) \\ \mathrm{COUT} &= \mathrm{MUXCY}(\mathrm{Fout}, \mathrm{CIN}, \mathrm{F}_1 \wedge \mathrm{F}_2) \\ \mathsf{p_{k+1}}[i] &= \mathrm{Y} \end{aligned}$$

Ces éléments de base sont combinés le long du chemin de retenue rapide pour former une addition conditionnelle complète d'un décalage du multiplicateur b :

$$p_{k+1} = (p_k + \mathsf{a}[k] \times (\mathbf{b} \ll k))$$

Enfin ces additionneurs sont combinés en série, la suite définie en (3.1) est déroulée dans l'espace pour obtenir le résultat final de la multiplication.

3.3.2.9 Décalage de bits constant

$$Shift(\langle \mathsf{o} \rangle, \langle \mathsf{a} \rangle, \mathbf{d})(a)$$

La primitive $Shift$ génère un circuit de décalage de $\mathbf{d}$ bits de son opérande a. Si $\mathbf{d} > 0$, les bits de a sont décalés vers la gauche, si $\mathbf{d} < 0$, ses bits sont

[3] dans le cas d'un multiplicande signé, le bit de signe du multiplicande $\mathsf{a}[|\langle \mathsf{a} \rangle| - 1]$ contrôle la soustraction au résultat de $\mathbf{b} \ll (|\langle \mathsf{a} \rangle| - 1)$

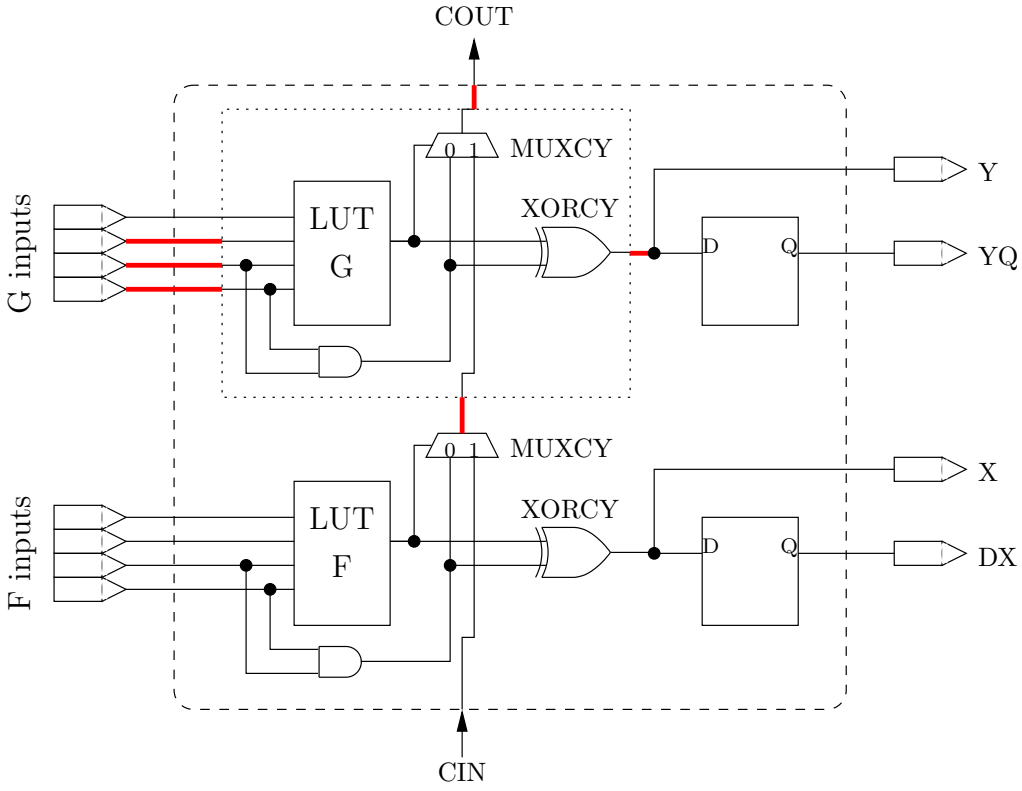


FIG. 3.9 – Logique de multiplication-accumulation spécialisée du Virtex II

décalés vers la droite. En décalage vers la gauche, des bits nuls sont ajoutés ; pour le décalage vers la droite, l'extension de signe naturelle est utilisée.

Cette primitive a un coût nul en termes de logique – le décalage fixe n'est qu'une opération de routage.

3.3.2.10 Décalage de bits à gauche

$$ShiftL(\langle \mathsf{o} \rangle, \langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle)(a, b)$$

La primitive $ShiftL$ – avec $\langle \mathsf{b} \rangle > 0$ – génère un opérateur de décalage à gauche qui calcule le résultat du décalage de l'opérande **a** de **b** bits vers la gauche c'est-à-dire $\mathbf{o} = \mathbf{a} \ll \mathbf{b}$. Le résultat o est *bitsizé* sur $\langle \mathsf{o} \rangle$ bits en sortie.

Cet opérateur est implémenté par une suite de $\langle \mathsf{b} \rangle$ décalages conditionnels contrôlés par les bits du tableau b. Plus précisément, le bit $\mathsf{b}[k]$ contrôle le décalage de la quantité partiellement décalée s_k de 2^k bits vers la gauche. Cette opération est réalisée par un simple multiplexeur :

$$\begin{aligned} s_0 &= \mathbf{a} \\ s_{k+1} &= \mathrm{mux}(\mathsf{b}[k], s_k \ll 2^k, s_k) \\ \Rightarrow s_{\langle \mathsf{b} \rangle} &= \mathbf{a} \ll \mathbf{b} \end{aligned}$$

Chaque bit de l'opérande b se traduit par une couche de multiplexeurs. Afin de conserver un routage et une géométrie de l'opérateur les plus simples possibles, il n'a pas été fait usage des multiplexeurs larges décrits en 3.3.1.5, qui permettraient de réduire la profondeur de l'opérateur.

3.3.2.11 Décalage de bits à droite

$$ShiftR(\langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle)(a, b)$$

La primitive $ShiftR$ – $\langle \mathsf{b} \rangle > 0$ – génère un opérateur de décalage à droite qui calcule le résultat du décalage **a** de **b** (nécessairement positif) bits vers la droite c'est-à-dire $\mathbf{o} = \mathbf{a} \gg \mathbf{b}$. Le résultat o est *bitsizé* sur $\langle \mathsf{o} \rangle$ bits en sortie.

L'implémentation est strictement identique au décalage à gauche, grâce à la formule :

$$\begin{aligned} s_0 &= \mathbf{a} \\ s_{k+1} &= \mathrm{mux}(\mathsf{b}[k], s_k \gg 2^k, s_k) \\ \Rightarrow s_{\langle \mathsf{b} \rangle} &= \mathbf{a} \gg \mathbf{b} \end{aligned}$$

3.3.3 Opérateurs synchrones

Les opérateurs suivants sont les opérateurs primitifs qui contiennent des registres. Ils sont implicitement synchronisés sur l'unique horloge du circuit DSL qui les contient.

3.3.3.1 Registres

$$Reg(\langle \mathsf{o} \rangle, \langle \mathsf{a} \rangle, \langle \mathsf{e} \rangle)(e, a)$$

La primitive Reg génère une barrière de registre entre son entrée a et sa sortie o soumise au signal d'activation arithmétique e. L'horloge des registres est implicite – DSL ne décrit que des circuits à horloge unique.

Plus précisément, chaque bit $\mathsf{o}[i]$ en sortie est calculé selon la formule suivante :

$$\mathsf{o}[i] = \mathrm{Z}(\mathrm{mux}(\mathsf{e}[i], \mathsf{a}[i], \mathsf{o}[i])) \tag{3.2}$$

Le signal d'activation e est donc un signal arithmétique : chacun de ses bits contrôle individuellement l'activation des registres de la primitive. Le cas le plus simple, $\langle \mathsf{e} \rangle = -1$, correspond à un signal d'activation binaire traditionnel. Un seul bit, qui est distribué automatiquement par extension du bit de signe, contrôle l'ensemble des registres.

La synthèse de cette primitive sur les ressources du Virtex II est très aisée puisque le registre du Virtex II dispose d'un signal d'activation intégré. Le multiplexeur de (3.2) est intégré à la ressource figurée en 3.2.

Registres de resynchronisation La tranche du Virtex II est particulièrement adaptée à l'insertion de registres de resynchronisation dans le flot de données. En effet, comme l'a montré la figure 3.3, un registre est situé après chacune des LUTs de la tranche. Ainsi le positionnement de registres de resynchronisation à la suite d'un opérateur – arithmétique ou logique – possède deux qualités essentielles :

- d'une part, le registre est idéalement positionné directement derrière un bit calculé par l'opérateur, ce qui minimise le temps de routage entre bit calculé et registre de resynchronisation ;
- d'autre part, la tranche dans laquelle le registre inséré prend place est déjà occupée par un opérateur logique : son ajout n'entraîne par conséquent aucun sur-coût, ce que mettra en évidence l'analyse des résultats de la synthèse des circuits resynchronisés du chapitre 5.

3.3.3.2 Mémoires

$$Mem(mode_1, mode_2)(\langle \mathsf{o}_1 \rangle, \langle \mathsf{d}_1 \rangle, \langle \mathsf{a}_1 \rangle)(\langle \mathsf{o}_2 \rangle, \langle \mathsf{d}_2 \rangle, \langle \mathsf{a}_2 \rangle)(e_1, we_1, d_1, a_1)(e_2, we_2, d_2, a_2)$$

La primitive Mem génère une mémoire double-port synchrone en lecture et en écriture. Chacun des deux ports dispose de deux signaux de contrôle et de deux signaux de données :

1. un signal de contrôle booléen d'activation global e (*enable*),
2. un signal booléen d'activation spécifique d'écriture we (*write enable*),
3. une entrée de données qui porte l'adresse **a** à laquelle effectuer l'opération de lecture/écriture,
4. une entrée de données d qui porte le cas échéant la valeur écrite à l'adresse a.

La primitive Mem possède deux sorties qui portent le résultat de la lecture des données sur chacun des deux ports. Dans le modèle simplifié de DSL, les deux ports ont la même horloge implicite, qui est celle du circuit. En revanche, les tailles d'adressage $\langle \mathsf{a}_1 \rangle$ et $\langle \mathsf{a}_2 \rangle$ peuvent être différentes sur les deux ports, ce qui permet de lire différemment le même tableau de données.

La logique d'activation de la RAM est simple. Lorsque le signal d'activation du port e est nul, aucune modification, ni de l'état interne de la mémoire, ni de l'état de la sortie o n'est enregistrée. Lorsque le bit e est positionné, le bit we est pris en compte. Si $we = 1$, alors l'état interne de la mémoire à l'adresse a est modifié pour prendre la valeur d. Le comportement de la sortie o en fonction de we est quant à lui dicté par le paramètre *mode* qui spécifie trois comportements différents :

- En mode *lecture prioritaire*, lors d'une écriture ($e = 1$, $we = 1$) la valeur mem[**a**] présente en RAM à l'adresse **a** est lue avant d'être remplacée par **d**, et présentée sur la sortie o au cycle suivant.
- En mode *écriture prioritaire* la valeur présente en RAM à l'adresse a est ignorée lors d'une écriture à cette adresse : la sortie o porte au cycle suivant la valeur **d**, et la mémoire se comporte alors simplement comme un registre pour sa valeur d'entrée **d** lors d'une écriture.

– En mode *exclusif*, lecture et écriture sont exclusifs, c'est-à-dire que lors d'une écriture ($e = 1$, $we = 1$) la valeur présentée sur la sortie o n'est pas mise à jour au cycle suivant.

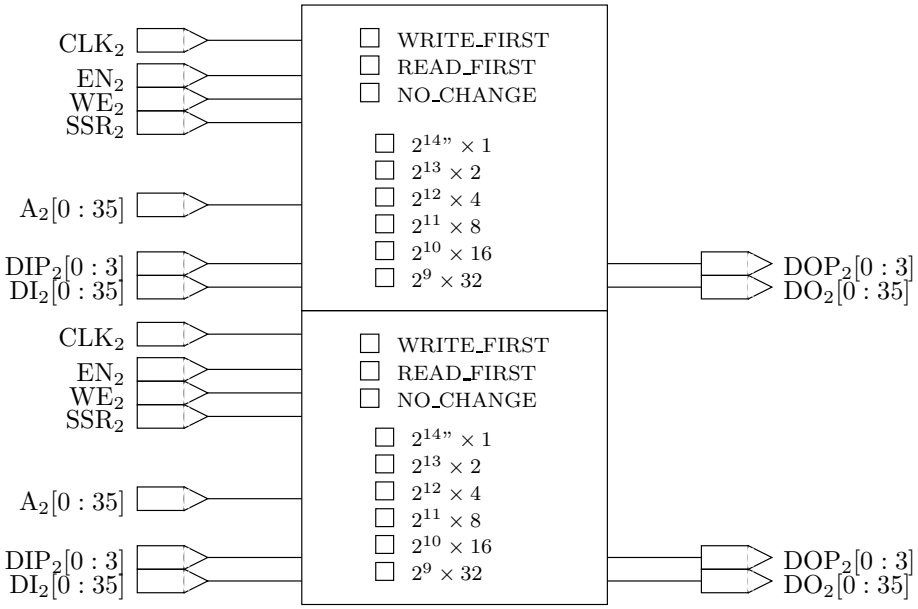


FIG. 3.10 – Bloc RAM du Virtex II

Bloc RAM du Virtex II La primitive de mémoire DSL est très précisément modelée sur la primitive de bloc RAM disponible sur le Virtex II (figure 3.10). Cette primitive est très souple et complexe ; aussi le lecteur intéressé est invité à en consulter les détails directement dans les manuels Xilinx [27]. Les blocs de RAM du Virtex II sont de 16KiB, au nombre de 56 sur le xc2v2000. Ils peuvent être configurés – indépendamment sur chacun des ports – avec toutes les configurations entre $2^{14} \times 1$ et $2^9 \times 32$ bits.

La technologie Altera dispose de primitives très proches, et notre primitive de RAM, malgré sa spécificité, n'est pas un obstacle à l'usage d'autres technologies.

3.3.3.3 Registres à décalage

$$ShiftReg(p, \langle \mathsf{o} \rangle, \langle \mathsf{d} \rangle)(e, d)$$

La primitive $ShiftReg$ génère un registre à décalage de profondeur p sur les données présentées en d soumises au signal d'activation booléen e.

Registres à décalage distribués Pour les registres à décalage de taille moyenne, le Virtex II permet d'utiliser les bits de configuration des LUTs de la tranche comme mémoire RAM. Cette RAM est qualifiée de Mémoire Distribuée – Distributed RAM (DistRAM). Elle dispose encore plus spécifiquement d'un mode de configuration en registre à décalage de largeur 1 bit et de profondeur fixée entre 1 et 16 bits.

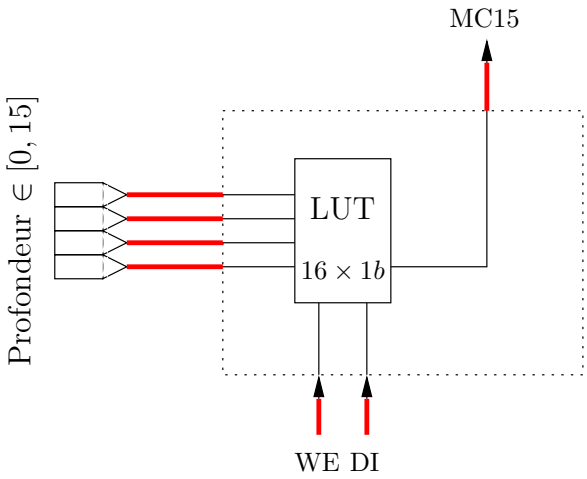


FIG. 3.11 – Configuration de la LUT du Virtex II en registre à décalage

Dans ce mode de configuration, figuré en 3.11, un bit de configuration de LUT précise son usage en registre à décalage, et les fils suivants sont alors actifs :

- l'entrée WS (*Write Set*) en constitue le bit de contrôle d'avancement ;
- l'entrée DI en est le bit de donnée entrante ;
- la sortie MC15 en est le bit de sortie à la profondeur 16 ;
- les entrées F_1,F_2,F_3 et F_3 de la LUT précisent la profondeur de la sortie D ;
- la sortie D en est le bit de sortie de profondeur réglable spécifiée par les entrées de la LUT.

Cette configuration ne fait pas intervenir d'horloge, c'est le bit de contrôle d'avancement WS qui joue ce rôle. WS est un signal calculé par le truchement d'un élément de logique spécialisé qui combine l'horloge CK et le bit d'écriture WE.

Un chemin de routage dédié permet de chaîner les LUTs configurées en registres à décalage pour réaliser des registres à décalage de taille conséquente sans encombrer les ressources de routage générales, comme figuré en 3.12, au sein de la tranche. Le chemin dédié relie la sortie MC15 d'une LUT à l'entrée DI de la LUT située immédiatement en-dessous. Ce chemin spécifique est prolongé au-delà même des frontières du CLB.

Le dernier étage d'un registre à décalage implémenté en DistRAM est constitué de la sortie d'une LUT configurée en registre à décalage sur sa fenêtre réglable D suivie d'un registre synchrone terminal qui permet de couper le chemin critique en sortie de la LUT.

Registres à décalage en RAM Pour les registres très profonds ou très larges, la mémoire distribuée peut se révéler très coûteuse en ressources logiques. Dans ces cas, l'utilisation du port d'un bloc RAM en mode *lecture prioritaire* et d'un compteur d'adresse (voir 3.3.3.4) permet d'obtenir à peu de frais un registre à décalage. Le second port de la mémoire est par ailleurs libre pour ouvrir une fenêtre sur le registre à décalage.

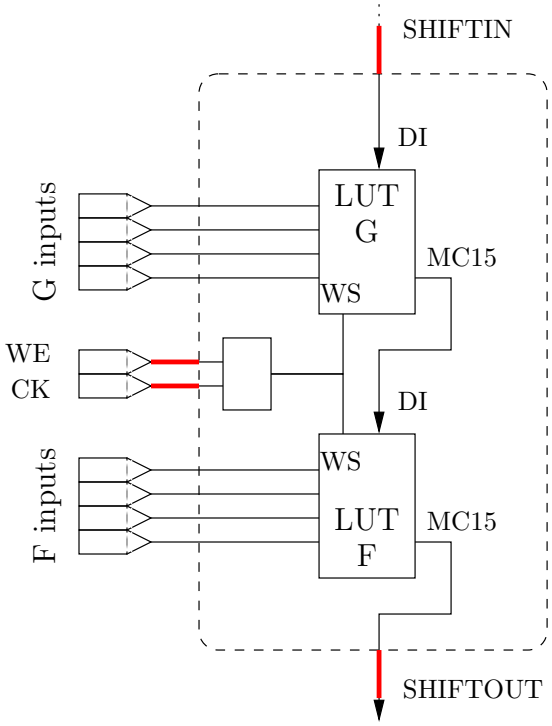


FIG. 3.12 – Combinaison des deux LUTs de la tranche du Virtex II en registre à décalage

Puisque DSL fournit à son utilisateur la possibilité de construire un tel registre à décalage à l'aide des primitives de haut niveau, le registre à décalage en RAM n'a pas de primitive bas niveau dédiée : c'est une primitive écrite directement en DSL.

3.3.3.4 Compteurs

$$Count(l, h)(\langle \mathsf{o} \rangle)(e)$$

La primitive *Count* génère un compteur synchrone de valeur initiale l, comptant dans l'intervalle $[l, h]$ et soumis au signal d'activation booléen e.

La synthèse d'un tel opérateur utilise des primitives connues : additionneur pour l'incrément, égalité pour la détection de la borne supérieure, et registre pour le stockage de l'état interne. Le schéma général est figuré en 3.13.

Ces primitives sont légèrement adaptées pour gagner en compacité et en performance.

- Le détecteur d'égalité utilisé ici, dont l'argument de comparaison est fixe, peut comparer quatre bits par LUT, il est donc deux fois plus compact et deux fois plus rapide que le comparateur générique décrit en 3.3.2.7.
- L'additionneur est placé juste devant les registres qui stockent la valeur courante du compteur. Ainsi le temps de propagation entre incrémenteur

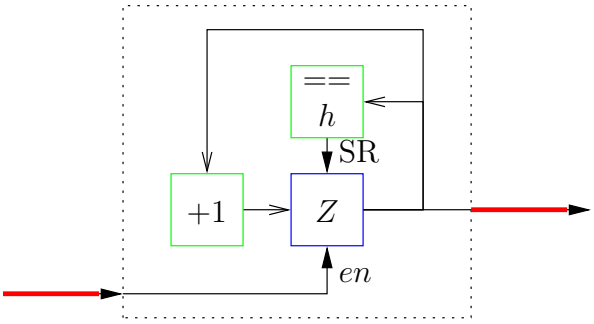


FIG. 3.13 – Compteur rapide implanté sur le Virtex II

et registre est minimal, ce qui est crucial puisque le chemin critique du compteur est précisement celui de l'incrémenteur.

– Le registre qui stocke l'état interne est initialisé à la valeur l qui est aussi sa valeur de réinitialisation. Le registre du Virtex II (figure 3.2) dispose de l'option de configuration INIT0 /INIT1 qui permet d'ajuster sa valeur initiale et d'un signal de retour à la valeur initiale SR (*Set - Reset*) qui permet de l'y réinitialiser. Cette fonctionnalité est mise en œuvre pour stocker la valeur initiale l du compteur et retourner dans cet état une fois atteinte la borne supérieure h.

Ces optimisations rendent inutile le pipelinage du signal de remise à zéro du compteur.

4

Environnement du DUT

Les circuits décrits par le Design Source Language (DSL) du chapitre 2 ont été adaptés pour une architecture reconfigurable particulière au chapitre 3. Ce chapitre traite de la technologie qui permet l'exécution effective du circuit sur un coprocesseur matériel.

Le coprocesseur utilisé est issu de la famille des Pam [30]. L'utilisation d'un coprocesseur Pam nécessite un canal de communication rapide entre la mémoire de l'hôte – qui contient les données de travail – et la Pam – qui implémente le circuit opérant sur les données. La communication emprunte les bus matériels de l'hôte et fait par conséquent intervenir à la fois des éléments matériels d'interface sur le bus, et des éléments logiciels du système d'exploitation.

Les routines tant matérielles que logicielles représentent un investissement important en termes de complexité de développement. Elles sont par ailleurs critiques pour la performance globale de l'application. Par conséquent, elles sont partagées entre tous les circuits décrits en DSL et une mesure fine de leur efficacité est conduite.

Nous décrivons dans ce chapitre l'environnement de communication du Design Under Test (DUT), ou comment à partir de la spécification de haut niveau écrite en DSL, le système de compilation génère une interface de communication normalisée entre le circuit exécuté sur le FPGA et la mémoire de l'hôte. Une telle interface comporte une composante matérielle qui enveloppe le DUT sur le FPGA et une composante logicielle sous la forme de librairies exécutées sur la machine-hôte. L'ensemble constitue un système d'interfaçage automatique qui réalise sur le bus asynchrone, de manière transparente, la sémantique de réseaux de Kahn synchrones de DSL [31, 32].

Le chapitre termine sur un descriptif de la chaîne de compilation utilisée pour générer le fichier de configuration du coprocesseur FPGA une fois les routines de communication ajoutées au DUT.

4.1 La plate-forme SEPIA

SEPIA [33] est la plate-forme matérielle utilisée à l'ÉNS comme support pour DSL. C'est un projet des laboratoires Hewlett-Packard, développé en particulier par Mark Shand, dans la lignée des travaux sur Peripheral Component Interconnect (PCI) Pamette [34].

SEPIA est une carte PCI munie de deux FPGA Xilinx Virtex II de type xc2v2000. Le premier de ces FPGAs est un FPGA système directement relié au bus PCI : il est qualifié de PCI interface FPGA (PIF), et sa reconfiguration dynamique est impossible. Le PIF pilote un deuxième FPGA qualifié de FPGA utilisateur qui peut être reprogrammé à volonté et en toute sécurité : le FPGA utilisateur est utilisé pour réaliser les circuits décrits en DSL. DSL fait usage de l'infrastructure de communication entre le FPGA utilisateur et le processeur de la station de travail au travers du PIF et du bus PCI.

La section 4.1.1 décrit l'architecture bas niveau de la communication au niveau du bus PCI et ses mécanismes logiciels.

La section 4.1.2 traite des interfaces fournies par SEPIA, c'est-à-dire des mécanismes présents sur la carte pour réaliser des communications efficaces sur le bus PCI et les composantes logicielles afférentes.

Les concepteurs de SEPIA ont fait le choix de la souplesse en autorisant les accès bas niveau aux ressources de communication à partir du processus de calcul en mode utilisateur. La section 4.1.3 détaille les interfaces fournies en mode utilisateur, que ce soit en logiciel dans le processus de calcul ou en matériel dans le FPGA utilisateur.

4.1.1 Description matérielle

4.1.1.1 Interconnections sur le bus PCI

Le bus PCI fonctionne grâce à un espace d'adressage dans lequel chaque périphérique présent sur le bus se voit allouer une plage lors de la phase de configuration.

Une fois la phase de configuration effectuée, le bus PCI est un bus partagé où chaque périphérique est chargé de décoder les adresses qui y circulent et le cas échéant, de répondre aux requêtes dont il est destinataire, lorsque l'adresse de la transaction est située dans son propre espace d'adressage.

La figure 4.1 présente l'interconnection générale entre le processeur, la mémoire centrale et la carte SEPIA.

Les deux entités qui nous intéressent sur le bus PCI sont la carte SEPIA elle-même et le pont PCI. Le pont PCI est un concentrateur qui permet aux périphériques PCI d'accéder aux ressources de la machine situées en-dehors du bus lui-même et inversement, au processeur central d'accéder au bus PCI.

Le moteur PCI de SEPIA est l'entité logique du PIF chargée de l'interface entre le bus et la carte SEPIA. Il supporte les transferts de données à raison de 64 bits par cycle à une fréquence de 66MHz ; la bande passante théorique du bus est par conséquent de 512 MiB.s^{-1}. L'espace adressable de SEPIA est en revanche réduit à 32 bits ; cette limitation de SEPIA est prise en compte par le Basic Input/Output System (BIOS) de la machine lors de la phase de configuration de l'espace d'adressage PCI.

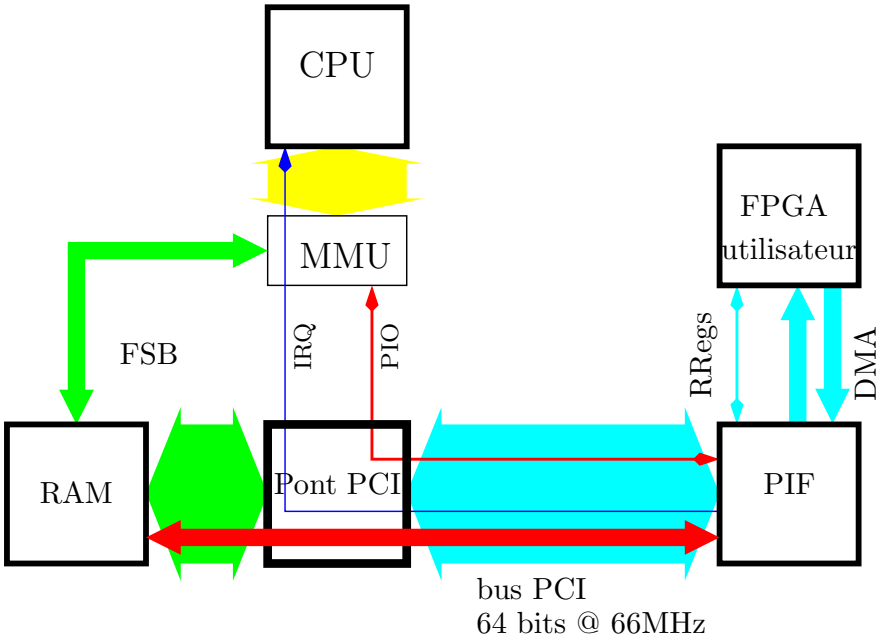


FIG. 4.1 – Interconnection du coprocesseur SEPIA

Lors d'un dialogue entre la carte SEPIA et le reste du système, deux modes de communication sont distingués, en fonction du chemin qu'empruntent les données entre les composants de la figure 4.1.

Accès PIO Dans le premier mécanisme dit de Programmable Input/Output (PIO), le processeur central (CPU) accède explicitement au coprocesseur par une transaction dont il est l'initiateur.

Dans cette méthode d'entrée-sortie, le CPU intervient explicitement dans chaque transaction, lecture comme écriture. Le pont PCI réalise la transaction physique sur le bus pour le compte du processeur et, dans le cas d'une lecture par exemple, renvoie vers le processeur le résultat de la requête.

Vu du processeur, les requêtes vers le bus PCI sont beaucoup plus lentes que celles, par exemple, à destination de la mémoire centrale. La méthode de PIO est par conséquent très gourmande en temps CPU et très peu efficace en termes de bande passante.

Accès DMA La méthode la plus efficace pour réaliser des entrées-sorties est de permettre à la carte PCI d'effectuer les requêtes de lecture-écriture des données en mémoire centrale de manière autonome, au travers du pont PCI et sans intervention du processeur central : c'est ce qu'on appelle la méthode d'Accès Direct à la Mémoire (DMA).

La carte SEPIA dispose en effet de la capacité d'initier directement des transactions sur le bus PCI – elle peut donc aller lire ou écrire en mémoire centrale en s'adressant directement au contrôleur DMA du pont PCI, qui fait suivre les requêtes en lecture comme en écriture vers la mémoire.

Le rôle du processeur se limite alors à envoyer vers la carte SEPIA, en mode PIO, les informations de contrôle nécessaires pour effectuer le transfert de données, sans intervenir directement lors du transfert lui-même.

Mais si le processeur n'intervient pas au cours d'une transaction DMA, il n'a a priori aucun moyen d'en apprécier la progression, sauf à consulter régulièrement l'état de la carte. Les mécanismes d'attente active sont généralement consommateurs inutiles de ressources : le bus PCI fournit les moyens d'une communication événementielle grâce aux *interruptions*. En l'occurence une *interruption* adressée au CPU, la carte SEPIA lui signale explicitement la fin d'un transfert.

4.1.1.2 Adresses

La mémoire centrale de la station de travail est une ressource partagée à laquelle il est possible d'accéder à partir du processeur et du bus PCI. Or les adresses manipulées par ces composants sont différentes, ce qui implique une correspondance entre trois familles d'adresses distinctes.

Adresses physiques Les adresses physiques sont les adresses utilisées sur le Front Side Bus (FSB) pour accéder à la mémoire physique. Elles sont codées sur 32 ou 64 bits et couvrent linéairement la mémoire installée dans la machine.

Adresses bus Si les bus PCI et FSB sont distincts, la mémoire physique est toutefois accessible depuis le bus PCI au travers du Contrôleur DMA – DMA Controller (DMAC). Le plus souvent, les adresses attribuées sur le bus à la mémoire physique sont égales aux adresses physiques de la mémoire utilisées sur le FSB. Toutefois, certaines architectures permettent d'utiliser une IOMMU, un circuit spécialisé qui peut rediriger certaines adresses bus vers des adresses physiques différentes.

Dans le cas de SEPIA, la limitation à 32 bits d'adresses du moteur PCI signifie que la mémoire physique accessible au coprocesseur est limitée. Comme ces 32 bits d'adresse doivent au moins couvrir la plage dédiée à la carte elle-même ainsi que celle dédiée à la mémoire physique, la mémoire physique adressable par SEPIA est effectivement limitée à 2GiB. L'utilisation d'une IOMMU pourrait permettre de contourner la limitation, toutefois cette fonctionnalité n'est pas présente sur toutes les architectures x86[1].

Adresses virtuelles Les adresses mémoire manipulées par les processus de calcul exécutés en mode utilisateur par le CPU sont des adresses virtuelles. Les accès mémoire vers de telles adresses sont réalisés par un circuit dédié, l'Unité de Gestion de la Mémoire – Memory Management Unit (MMU). Le système d'exploitation gère pour chaque processus une structure appelée Virtual Memory Area (VMA) qui décrit la correspondance entre adresses virtuelles et adresses physiques ou adresses du bus. La VMA est exploitée par la MMU pour résoudre automatiquement l'accès à une adresse virtuelle en un accès vers la mémoire physique à une adresse physique donnée, ou encore en une transaction au niveau d'un bus, par exemple vers le bus PCI en mode PIO.

[1] une IOMMU est disponible sur les architectures Opteron, mais elle n'a pas été exploitée par DSL

4.1.1.3 Vue d'ensemble

Au terme de cette présentation, il est possible de détailler les échanges figurés sur la figure 4.1. Processeur, mémoire et pont PCI échangent des adresses physiques sur le FSB. Le processeur peut communiquer directement avec le coprocesseur en mode PIO au travers du pont PCI. Le coprocesseur peut communiquer directement avec la mémoire grâce au DMAC du pont PCI.

Le paragraphe suivant est dédié à l'étude de l'interface logicielle de la communication avec SEPIA. L'interface matérielle bas niveau est décrite en détail dans la documentation de SEPIA [33].

4.1.2 Interface logicielle en mode noyau

L'interface logicielle pour l'accès à un périphérique sur bus PCI fait appel à de nombreuses notions de programmation système, en particulier la notion de mémoire virtuelle.

Le principe et les applications de la mémoire virtuelle sont mis en évidence, puis exploités dans le cadre particulier de l'utilisation du coprocesseur SEPIA.

4.1.2.1 Virtual Memory Area (VMA)

Les architectures modernes utilisent un système de mémoire fondé sur deux principes :

- l'espace d'adressage d'un processus est virtualisé, comme esquissé en 4.1.1.2 ;
- l'espace d'adressage est paginé : la VMA est divisée en unités atomiques homogènes appelées pages. Sur architecture x86, la taille de la page est de 4KiB. Chacune des pages de la VMA possède un jeu de permissions exploitées par la MMU lors de l'accès par le processus à une adresse virtuelle.

Ces deux aspects sont gérés par la MMU et permettent une grande flexibilité dans la gestion de la mémoire par les systèmes d'exploitation.

L'exemple le plus simple des opportunités offertes par l'utilisation d'une MMU est celui de l'allocation mémoire. Lorsqu'un processus demande un bloc de mémoire supplémentaire au système d'exploitation, celui-ci alloue le nombre de pages mémoires correspondant et met en place les permissions nécessaires dans la VMA. En cas de fragmentation de la mémoire physique, ces pages peuvent être non-contiguës. La mémoire allouée peut néanmoins apparaître comme un bloc continu d'adresses virtuelles grâce à l'entremise de la MMU qui réalise la translation des adresses virtuelles vers les adresses physiques utilisées par le contrôleur RAM.

Par ailleurs, les systèmes d'exploitation modernes utilisent extensivement l'indirection offerte par la MMU pour effectuer certaines optimisations :

- L'allocation paresseuse permet au système d'exploitation de n'allouer une page en mémoire physique qu'au moment de son utilisation effective en écriture plutôt qu'au moment de la requête d'allocation – la demande d'allocation est honorée par l'insertion dans la VMA d'une page générique nulle protégée en écriture. Lors d'un accès en lecture, le processus lit correctement les valeurs – nulles – présentes sur la page. Lors d'un accès en écriture, la MMU déclenche une interruption pour violation d'accès. Le système d'exploitation reprend la main et substitue à la page générique

une page de données réservée au processus et accessible en écriture. Le tout est réalisé de manière transparente pour le processus.
- La mémoire virtuelle disque permet au système de libérer une partie de la mémoire physique déjà allouée à un processus – afin de la rendre disponible pour d'autres processus – en l'écrivant sur un périphérique de mémoire de masse. Le système d'exploitation réalise cette opération en substituant à la page libérée une page générique protégée en lecture et en écriture. Au cas où le processus ferait un accès ultérieur à cette page, la violation d'accès résultante permet au système d'exploitation d'intervenir pour restaurer les données originelles à partir de la copie présente en mémoire de masse.
- Enfin, des zones de la VMA permettent de réaliser des accès transparents et avec une interface universelle vers des ressources d'entrée-sortie variées : c'est la notion de redirection mémoire (*memory mapping*). Par exemple, l'accès à une zone mémoire redirigée sur un fichier masque les accès au disque dur. D'autres redirections sont possibles, vers des périphériques par l'entremise d'un pilote, ou directement vers des bus de données, comme le bus PCI.

L'accès au coprocesseur SEPIA fait la part belle au mécanisme de VMA. Les paragraphes suivants présentent l'aspect logiciel des modes d'accès au coprocesseur vus en 4.1.1.1, en particulier comment ceux-ci interagissent à bas niveau avec la VMA. Si les concepts décrits sont identiques pour tous les systèmes d'exploitation, l'interface logicielle bas niveau a été spéficiquement réalisée et utilisée sur le système d'exploitation Linux. L'ouvrage [35] fournit une excellente référence sur les Application Programmer Interface (API)s propres à ce système.

4.1.2.2 PIO

Pour réaliser les accès en PIO à SEPIA, le pilote Linux de la carte SEPIA effectue une redirection mémoire dans la VMA du processus vers une partie de l'espace d'adresses de SEPIA sur le bus PCI.

Un accès par le processus dans l'intervalle d'adresses virtuelles redirigé déclenche automatiquement un accès de même type sur le bus PCI à l'adresse correspondante – c'est-à-dire vers le coprocesseur SEPIA.

Par exemple, l'écriture d'une donnée dans la mémoire provoque en réalité une transaction PCI en mode PIO d'écriture sur le bus, avec la donnée initiale, qui est *in fine* reçue et interprétée par SEPIA. Inversement, la lecture d'une donnée déclenche une lecture sur le bus PCI, et la valeur transmise en retour par le coprocesseur est livrée au processus de calcul. C'est l'un des sens que l'on peut attribuer à la notion de *Mémoire Active Programmable* : la mémoire vue du processus de calcul est le résultat d'un traitement dynamique par SEPIA, et peut dont être qualifiée d'*active*.

Les accès PIO sont naturellement réalisés par ce mécanisme. Ce mode de communication est utilisé pour faire transiter des informations de contrôle de la carte : état des interruptions, du moteur de DMA, et bien peu pour faire circuler effectivement des données, qui peuvent être traitées bien plus rapidement par l'utilisation du moteur DMA.

4.1.2.3 DMA

Le mécanisme d'une transaction DMA est complexe : le processeur fournit à SEPIA les informations de contrôle sur la localisation des données à traiter en mémoire (tampon de lecture et tampon d'écriture) et lui instruit d'engager la transaction DMA. Le processeur peut alors vaquer à d'autres activités : une fois la transaction traitée, SEPIA en signale activement la fin par une interruption au processeur qui peut réagir à l'événement.

Translation d'adresse Le coprocesseur sur bus PCI s'adresse au contrôleur DMA à l'aide d'adresses transmises sur le bus correspondant aux adresses physiques des pages de mémoire à traiter.

Or SEPIA n'a aucun accès aux informations de VMA du processus de calcul, ni même aux informations de redirection d'une éventuelle IO MMU (IOMMU). Lorsque le processus de calcul soumet à SEPIA l'adresse d'un tampon de données à traiter par DMA, le CPU doit par conséquent utiliser les adresses natives au bus PCI.

Le programme qui s'exécute sur le CPU doit effectuer explicitement la traduction des adresses virtuelles qu'il manipule vers les adresses physiques de la mémoire, puis des adresses physiques vers les adresses bus correspondantes sur le bus PCI. Ces adresses bus sont finalement transmises comme données en mode PIO vers le moteur DMA de la carte PCI.

Descripteur DMA L'opération de traduction des adresses virtuelles en adresses bus doit être répétée pour l'ensemble des pages qui constituent un tampon de données, c'est-à-dire avec une granularité égale à 4KiB.

Or, pour des volumes importants d'information, le nombre de pages impliqué devient vite important, et la transmission en mode PIO des adresses physiques constitue alors un goulot d'étranglement.

La donnée qui est transmise au moteur de DMA de SEPIA n'est donc pas directement l'adresse bus d'une page de mémoire, mais l'adresse d'un tableau d'adresses bus qui décrit l'ensemble des pages constitutives d'un tampon de données : c'est une forme de *scatter-gather*. La structure complexe décrivant les pages d'un tampon est appelée descripteur de DMA.

Pour chaque transaction DMA le processus de calcul construit un descripteur dont il soumet l'adresse par PIO au moteur DMA de SEPIA. Celui-ci charge et interprète les données contenues à l'adresse du descripteur. Le descripteur peut, dans le cas de tampons importants, recouvrir plusieurs pages : un mécanisme de liste chaînée entre les pages constituant le descripteur permet au moteur DMA de poursuivre son travail sans intervention du CPU.

Réception de l'interruption Lorsque la fin du descripteur est atteinte, le moteur DMA déclenche une interruption. L'interruption entraîne l'exécution immédiate par le CPU d'une routine spécifique à la carte SEPIA, qui peut réagir en conséquence.

Le détail des transactions est beaucoup plus complexe. Le mécanisme des interruptions par exemple est un mode de communication entre processus asynchrones – d'une part la carte PCI, d'autre part le processeur – et requiert les mêmes précautions que l'utilisation d'un synchronisateur matériel entre deux domaines d'horloge [36].

4.1.3 Interfaces en mode utilisateur

4.1.3.1 Interface logicielle

Les détails de fonctionnement décrits au paragraphe précédent supposent que la partie logicielle dispose d'un niveau de privilège élevé pour effectuer les manipulations bas niveau de translation d'adresses ou d'accès aux interruptions du système. La description ne tient pas compte des restrictions placées sur les processus exécutés par les utilisateurs. Or il est peu souhaitable d'exécuter un processus de calcul avec des privilèges élevés – la machine est alors à la merci d'une erreur de programmation. C'est pourquoi une dimension importante de SEPIA est qu'elle fournit à ses utilisateurs des mécanismes permettant d'utiliser les ressources de la carte en toute sécurité à partir de processus normaux.

L'architecture SEPIA résoud ce problème en ajoutant au pilote de la carte en mode noyau des appels systèmes qui permettent au processus utilisant SEPIA d'utiliser le pilote pour effectuer les tâches nécessaires au bon fonctionnement d'une transaction DMA. Ainsi donc, le pilote n'est utilisé que pour fournir un nombre limité de fonctionnalités, et laisse au processus la liberté du détail des transactions avec SEPIA.

Accès PIO Les accès PIO vers la carte SEPIA sont directement permis grâce au système de redirection mémoire : à la demande d'un processus utilisateur, le pilote SEPIA étend la VMA du processus par une zone qui est redirigée vers les registres de contrôle de la carte.

Tampons de données La VMA entraîne de gros problèmes pour l'utilisation des fonctions DMA de SEPIA à partir d'un processus non-privilégié. En effet le processus n'est jamais sûr qu'une zone de mémoire allouée corresponde effectivement à une zone de mémoire physique. Quand bien même la mémoire physique serait allouée à un moment donné, rien ne garantit que les pages physiques sous-jacentes demeurent valides !

Avant de pouvoir donner l'ordre au coprocesseur de travailler sur un jeu de données, le processus de calcul doit être en capacité de s'assurer que les tampons de données alloués pour le coprocesseur en lecture et en écriture correspondent effectivement à des pages physiques, et que cette mémoire physique est verrouillée – qu'elle ne va pas, par exemple, être libérée pour d'autres et mise sur disque.

Le pilote SEPIA fournit donc au processus utilisateur la possibilité de verrouiller l'assignation physique d'une zone mémoire de la VMA.

Descripteur du tampon de données Une fois les pages verrouillées en mémoire physique, le processus a besoin de récupérer les adresses bus liées aux pages verrouillées pour la construction du descripteur de DMA (4.1.2.3).

Le pilote SEPIA fournit donc au processus utilisateur la possibilité de récupérer l'adresse bus d'une adresse virtuelle de sa VMA.

4.1.3.2 Interface matérielle

Le pendant des interfaces logicielles en mode utilisateur est constitué des interfaces matérielles fournies par le PIF au FPGA utilisateur reprogrammable.

L'objectif est identique : offrir à l'utilisateur une interface souple et puissante, mais sécurisée pour prévenir des erreurs de programmation qui pourraient mettre en cause l'intégrité du système.

Accès PIO SEPIA fournit dans le FPGA utilisateur une interface qualifiée de RRegs, qui permet de relayer à l'interieur du FPGA des transactions en mode PIO sur le bus PCI. Certaines adresses PCI de SEPIA sont réservées à cet effet. Toute transaction PIO à leur endroit est retransmise au niveau d'une interface mémoire asynchrone dans le FPGA utilisateur.

L'interface comporte un bus de données entrant, un bus d'adresse entrant, deux bits de contrôle indiquant l'arrivée d'une requête et sa nature, lecture ou écriture. La réponse à la requête est signalée par l'utilisateur à l'aide d'un bit de contrôle, et le cas échéant, les bits du bus de données sortant sont échantillonnés au même cycle pour fournir la réponse à la requête.

L'interface permet une communication asynchrone, et en effet l'horloge de l'interface en mode utilisateur est choisie indépendamment de l'horloge utilisée par le moteur PCI.

Accès DMA SEPIA fournit aussi une interface DMA très simple. Cette interface est synchrone, et son horloge est fixée à 125 MHz.

Sur le canal de DMA entrante, qui est activé lorsque le moteur de DMA lit des données en provenance de la mémoire, la largeur du chemin de données est de 64 bits en lecture, plus un bit d'*enable* en lecture. Un bit de contrôle en écriture permet à l'utilisateur d'interrompre le flux de données entrant. Lorsque ce signal est levé, la transaction DMA atomique courante (de la taille d'une page, soit 4KiB) s'achève, mais le canal entrant DMA est ensuite désactivé jusqu'à la retombée du signal.

L'interface du canal de DMA sortant est constituée d'un bus de données en écriture de 64 bits de large, d'un bit d'*enable* en écriture, sur lequels les données à écrire en mémoire sont présentées. Pour réguler le flux de données, l'interface fournit un bit de contrôle en lecture qui signale que les tampons de sortie du moteur DMA sont presque pleins. L'utilisateur doit réagir à ce signal en interrompant le flux de données en sortie.

4.2 Interconnexion du DUT

Nous venons de détailler l'infrastructure offerte par SEPIA. La combinaison d'une interface logicielle et d'une interface matérielle en mode utilisateur offre une grande flexibilité à l'expérimentateur pour la conception rapide de circuits performants. Dans le cadre de DSL, nous utilisons ces interfaces pour réaliser la sémantique de réseaux de Kahn synchrones décrite par DSL.

En effet les circuits DSL décrivent le traitement de données entre l'entrée et la sortie synchrones du circuit. Le code-source DSL spécifie comment, à chaque coup d'horloge, le circuit accéléré lit sur son entrée une donnée en provenance de la mémoire et y écrit la donnée calculée en sortie.

Ainsi donc la description du circuit en DSL porte implicitement la marque d'une communication entre le FPGA et la mémoire de la station-hôte qui contient les données sur lesquelles le circuit opère. La présente section décrit comment le circuit pur synthétisé au chapitre 3, qualifié de DUT, est interfacé avec SEPIA pour réaliser de manière transparente la communication asynchrone avec la mémoire de l'hôte au travers du bus.

4.2.1 Normes d'interconnections

Le problème de l'interconnection entre systèmes synchrones est particulièrement crucial pour faire communiquer des systèmes fonctionnant à des fréquences différentes – ou générant des événements avec des périodicités variées – ou encore des systèmes l'objet de normalisation.hétérogènes dans le cadre d'une cosimulation logicielle / matérielle.

Dans le cadre de DSL, ce problème est repoussé hors du langage : la description du matériel assurant la liaison entre le circuit et la station-hôte au travers du bus PCI n'est pas faite en DSL.

Une solution pour modéliser les interactions entre systèmes synchrones sur horloges hétérogènes dans un cadre strictement synchrone existent [37]. Cette méthode impose des conditions sur les horloges et les événements aux interfaces des circuits et introduit un calcul d'horloge élégant. Toutefois l'interface PCI de SEPIA ne répond pas aux contraintes imposée par cette solution.

Une solution plus adaptée aurait pu consister à nous placer dans le cadre de solutions normalisées existantes, comme Standard Co-Emulation Modeling Interface (SCE-MI) [38]. Nous n'avions pas connaissance de ces normes au moment de la réalisation de l'interface de communication de DSL.

4.2.2 Objectifs de l'environnement du DUT

Les interfaces de SEPIA ne sont pas bien adaptées à la simplicité du modèle synchrone de DSL :

- d'une part elles sont trop rigides, et mal adaptées à des performances optimales pour des circuits variés ;
- d'autre part elles sont trop complexes pour notre modèle synchrone simple.

Pour répondre à ces problèmes, DSL génère automatiquement une couche d'adaptation autour du DUT pour adapter au mieux les interfaces rigides de SEPIA au circuit synthétisé. Cette construction comprend deux volets :

- d'une part, la réalisation d'interfaces matérielles adaptées autour du DUT sur le FPGA utilisateur ;

– d'autre part, la contrepartie logicielle sous la forme d'une API simplifiée d'appel au coprocesseur.

4.2.2.1 Le matériel qui entoure le DUT

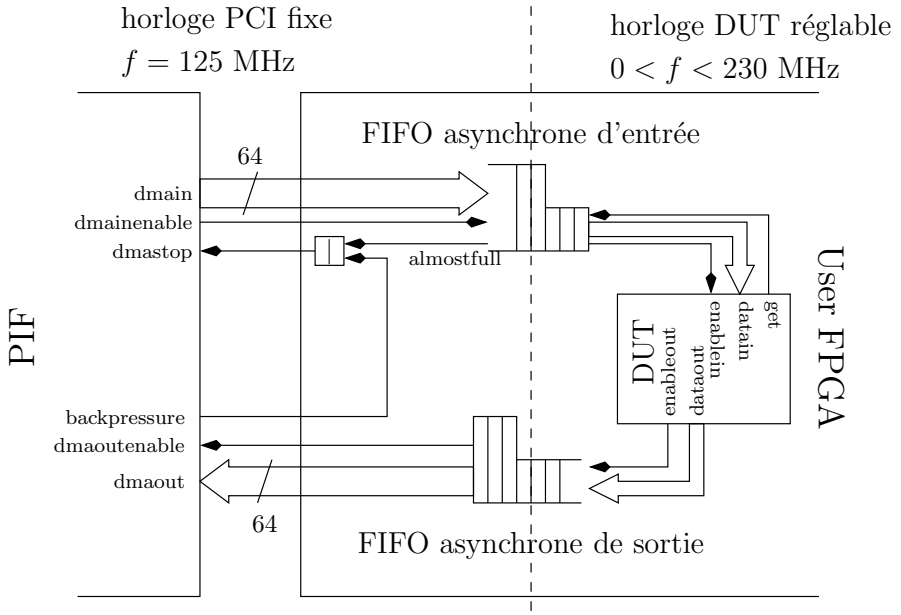


FIG. 4.2 – Interconnection DMA dans le FPGA utilisateur

L'interface matérielle qui entoure le DUT est présentée en figure 4.2. Le moteur DMA de SEPIA est utilisé pour amener les données à vitesse maximale en entrée du circuit, et inversement écrire les résultats calculés en sortie du circuit vers la mémoire centrale.

Adaptation d'horloge et de largeur de données La fréquence nominale de fonctionnement du DUT varie en fonction des performances obtenues lors de la synthèse du circuit. Par ailleurs, la largeur des données en entrée du circuit est variable.

Il est donc impossible de se satisfaire de l'horloge fixe à 125 MHz et du bus de données de 64 bits fournis par SEPIA comme interface en entrée comme en sortie du moteur DMA.

Des tampons FIFO asynchrones qui utilisent les blocs RAM double-port du Virtex II sont insérés et permettent d'adapter l'interface sur ces deux points :

– Chaque FIFO dispose de deux horloges distinctes en entrée et en sortie : le DUT dispose ainsi de sa propre horloge, découplée de l'horloge du moteur DMA. L'horloge du DUT utilise un générateur qui permet même de la régler dynamiquement depuis le programme de calcul.
– La largeur du chemin de données aux bornes du circuit est automatiquement alignée à 8, 16, 32 ou 64 bits – la largeur est identique en entrée et en sortie. Cette flexibilité permet de ne pas gaspiller de bande passante avec des bits d'alignement superflus.

La réalisation de ces FIFOs fait appel à toute la flexibilité des RAM double-port de la technologie Xilinx, en particulier le fait que les horloges et la configuration de la mémoire sont distinctes sur chacun des deux ports.

Contrôle de flux du DUT Le contrôle de flux a été simplifié au maximum au niveau du DUT. En entrée, le DUT dispose d'un signal *get* qui permet au circuit d'activer la réception de données en provenance de la FIFO d'entrée. Si celle-ci dispose de données, elle lève le signal d'*enable* en entrée du circuit. La plupart des circuits DSL ne fait pas usage du signal *get* qui est alors implicitement levé en continu. Certains circuits utilisent pourtant cette capacité de contôle, comme le circuit final d'estimation de mouvement de la section 5.3, qui ne reçoit une donnée que tous les onze cycles.

Le flux de données en sortie du DUT est directement écrit sur la FIFO de sortie – sans rétroaction directe de la FIFO sur le circuit. Certains circuits, tels les décompresseurs, peuvent éventuellement produire des débordements de tampon et sont donc impossibles à exécuter correctement *via* l'interface automatique de DSL.

Contrôle de flux DMA Le moteur DMA de SEPIA fournit sur l'interface d'entrée un bit de contrôle qui permet d'éviter que le moteur DMA en lecture n'engage de nouvelles transactions. Pratiquement, cela signifie qu'une fois le drapeau levé, le flux de données DMA entrant termine l'unité de 4KiB courante et s'arrête.

À l'inverse, le moteur DMA en sortie fournit un drapeau d'information qui signale lorsque les tampons du moteur de DMA sortant approchent de la saturation – il ne reste plus que 8KiB d'espace disponible.

Le signal "presque plein" du moteur de DMA sortant est combiné avec le signal "presque plein" de la FIFO asynchrone en entrée du DUT pour générer le signal de contrôle du moteur DMA entrant : le flux est ainsi très naturellement et très simplement régulé. Toutefois, comme nous l'avons vu, cette régulation simple ne permet pas d'exécuter correctement des circuits dont les débits en entrée et en sortie sont variables – cette limitation n'a pas été sensiblement contraignante au cours de la thèse, mais constitue un problème à résoudre pour obtenir un environnement d'exécution correct en toute généralité.

4.2.2.2 Le logiciel pour communiquer avec le DUT

L'API d'appel au coprocesseur est très simple. L'utilisateur fait uniquement appel à une fonction de traitement qui spécifie un tampon d'entrée contenant les données à traiter – correctement alignées – un tampon de sortie et leur longueur commune. La fonction d'appel au coprocesseur se charge de construire les descripteurs DMA des tampons, de les soumettre à SEPIA, et ne retourne qu'une fois la fin du traitement effectuée. L'utilisateur peut alors examiner le tampon de sortie du coprocesseur pour récupérer les résultats.

Cette communication est donc synchrone : le processeur attend la fin de la transaction avant de reprendre l'exécution du programme. Pour les applications simples, cette approche est pratique ; toutefois certaines applications ne peuvent s'en contenter. Le circuit de la section 5.3 fournit encore un exemple où il est nécessaire de prendre le contrôle des transferts à bas niveau pour atteindre des performances optimales.

Largeur de l'identité bits	Vitesse ns	MHz	Bande passante MiB.s^{-1}
8	4.56	218	218
16	4.55	219	439
32	4.57	218	874
64	4.65	214	1718

TAB. 4.1 – Compilation des identités

4.2.3 Mesure de performances

L'efficacité de l'ensemble du système d'accélération matérielle dépend en particulier de l'efficacité des routines logicielles et matérielles ajoutées dans ce chapitre. Des mesures de performances précises permettent de garantir que les circuits produits par DSL ne souffrent pas artificiellement de dégradations de performances dues à ces routines.

4.2.3.1 Méthodologie

Afin de mesurer les performances de l'environnement du DUT, les circuits les plus simples sont synthétisés et leurs performances mesurées. Les circuits choisis sont les identités sur 8, 16, 32, 64 bits de large en entrée et en sortie.

Ces circuits opèrent sur un tampon de données de taille fixée, rempli par un motif aléatoire. Le temps de traitement du tampon de données par le coprocesseur est mesuré, et le tampon de données résultat est comparé au tampon d'entrée pour vérifier le bon comportement du circuit.

4.2.3.2 Performance de la synthèse

Les routines matérielles décrites en 4.2.2.1 ajoutent sur l'horloge du DUT des opérateurs logiques liés au contrôle des tampons FIFOs asynchrones. Il faut donc s'assurer que l'implémentation des FIFOs utilisés n'introduit pas de chemin critique qui limiterait artificiellement les performances de la synthèse du DUT.

L'examen des résultats de synthèse des circuits-identités simples permet d'évaluer l'incidence des éléments matériels de l'environnement du DUT sur les chemins critiques du circuit synthétisé.

Le tableau 4.1 indique les fréquences nominales obtenues après synthèse des identités en situation dans leur environnement de communication avec SEPIA. Une analyse détaillée permet de préciser que, si la FIFO en sortie du DUT ne présente pas de problème particulier, le chemin critique est situé dans le logique de contrôle de la FIFO d'entrée.

En termes de fréquence absolue, les résultats sont néanmoins très bons. Bien plus, le croisement de ces résultats avec les mesures de bande passante présentées en 4.11 positionne la fréquence maximale introduite par les FIFOs au-dessus de la fréquence de saturation du bus PCI – pour tous les DUT possibles. Ainsi donc les FIFOs n'introduisent pas de limitation effective lors de l'utilisation réelle des circuits générés par DSL.

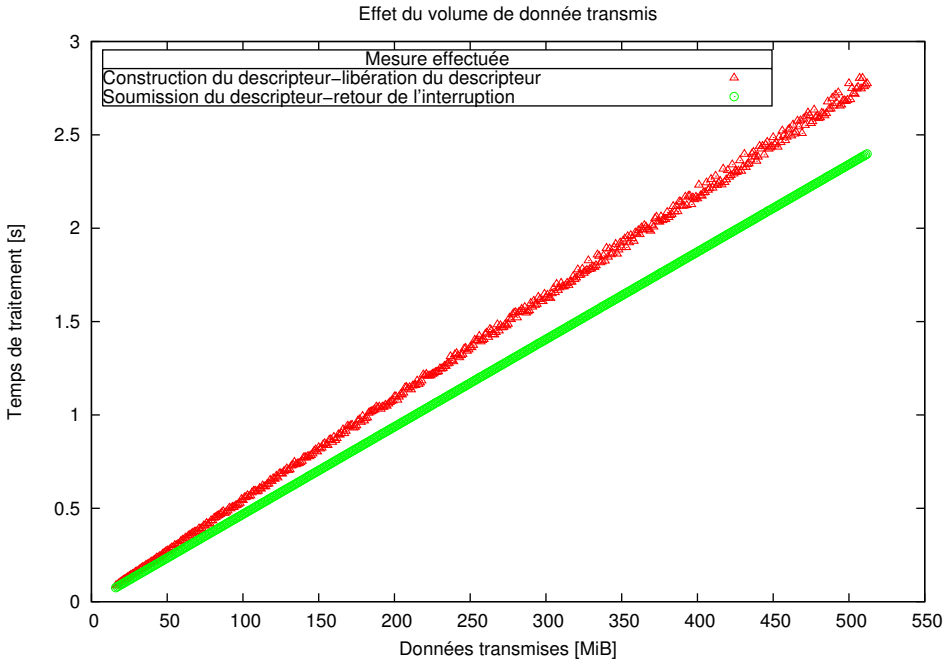


FIG. 4.3 – Temps de traitement en fonction de la taille du tampon d'entrée

4.2.3.3 Que mesurer ?

La mesure d'un "temps de traitement par le coprocesseur" pose la question essentielle de savoir ce que nous allons précisément mesurer. Dans ce cas précis, plusieurs possibilités s'offrent à nous.

Une transaction DMA peut en effet être, du point de vue de l'application, décomposée en deux temps :

- Une phase logicielle au cours de laquelle le processeur est actif dans la construction du descripteur DMA des tampons de données en entrée et en sortie. Le coprocesseur est alors passif.
- Une phase matérielle au cours de laquelle le processeur, inactif, attend que le coprocesseur signale la fin du traitement des données.

Dans notre dispositif expérimental, quatre temps sont enregistrés :

1. le temps courant juste avant la construction du descripteur par le logiciel ;
2. le temps à la fin de la construction du descripteur, et avant la soumission du descripteur au coprocesseur ;
3. le temps lorsque le signal de fin traitement parvient à l'application.
4. le temps après libération du descripteur de DMA

Ces quatre références permettent en particulier de calculer deux temps. Premièrement, un temps de transaction total, qui inclut la construction du descripteur et le traitement par le coprocesseur. Cette mesure correspond à l'appel par le programme de la fonction d'API simplifiée pour l'appel au coprocesseur.

Deuxièmement, un temps de transaction purement matériel, qui n'inclut que le temps de traitement par le coprocesseur, à l'exclusion des opérations logicielles

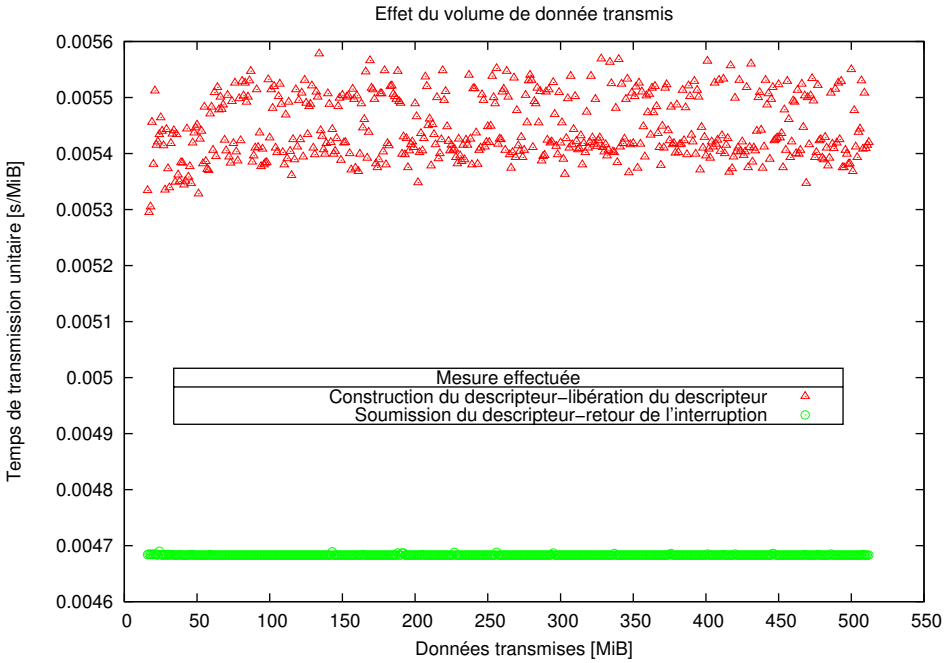


FIG. 4.4 – Temps de traitement unitaire en fonction de la taille du tampon d'entrée

de traitement du descripteur[2].

La figure 4.3 présente les deux mesures distinctes, pour des volumes de données traitées croissants. La figure 4.4 présente le temps mesuré ramené à une taille unitaire de données. Ces courbes indiquent deux tendances :

- le temps consacré à l'élaboration du descripteur est non-négligeable et représente toujours environ 15% du temps de traitement total – les algorithmes employés sont linéaires en fonction de la taille des tampons de données ;
- le temps logiciel d'élaboration du descripteur est soumis à d'importantes fluctuations, alors que le temps de traitement matériel est très régulier.

La première mesure correspond à l'usage de la fonction d'API de haut niveau d'utilisation du coprocesseur. En cas de recherche de performances maximales, lors de l'usage répété du coprocesseur, il est possible de sacrifier la simplicité aux performances et d'utiliser pour la communication avec le coprocesseur des tampons de données statiques, dont le descripteur peut être construit une fois puis réutilisé par la suite. La factorisation de l'étape d'élaboration du descripteur permet alors de gagner en performances et d'éviter les variations importantes du temps de traitement liées à la charge logicielle par ailleurs. Cette méthode est utilisée extensivement dans l'élaboration du circuit d'estimation de mouvement en 5.3.

Pour la suite des mesures, les deux mesures sont indiquées, mais les commentaires concernent le temps de traitement matériel, plus régulier.

[2]le temps de latence de l'interruption est négligé au regard des durées totales mesurées. [39] fournit plus de détails.

4.2.3.4 Source de temps et taille des données

Un problème récurrent [39] pour la mesure du temps en logiciel est la précision des fonctions UNIX de mesure de temps. Il est difficile d'obtenir une résolution supérieure au centième de seconde[3] ; or cette résolution n'est pas suffisante.

Le problème est contourné par l'utilisation d'un compteur matériel : il s'agit d'un compteur interne au coprocesseur réglé sur une fréquence fixe, dont la valeur peut être lue par accès PIO (Programmable Input/Output) au coprocesseur. Une telle source de temps permet d'augmenter considérablement la résolution des temps mesurés – le compteur choisi itère à 125 MHz.

La figure 4.4 permet de constater l'efficacité de l'utilisation d'un compteur matériel pour lequel des résultats très précis sont obtenus même pour des tailles de tampon faibles.

Au vu des résultats enregistrés, la taille de tampon de données utilisés est fixée à 256MiB, et les mesures de temps sont faites à l'aide du compteur matériel : la quantité de données et la qualité de la mesure du temps semblent alors suffisantes pour être significatives.

4.2.3.5 Première implémentation

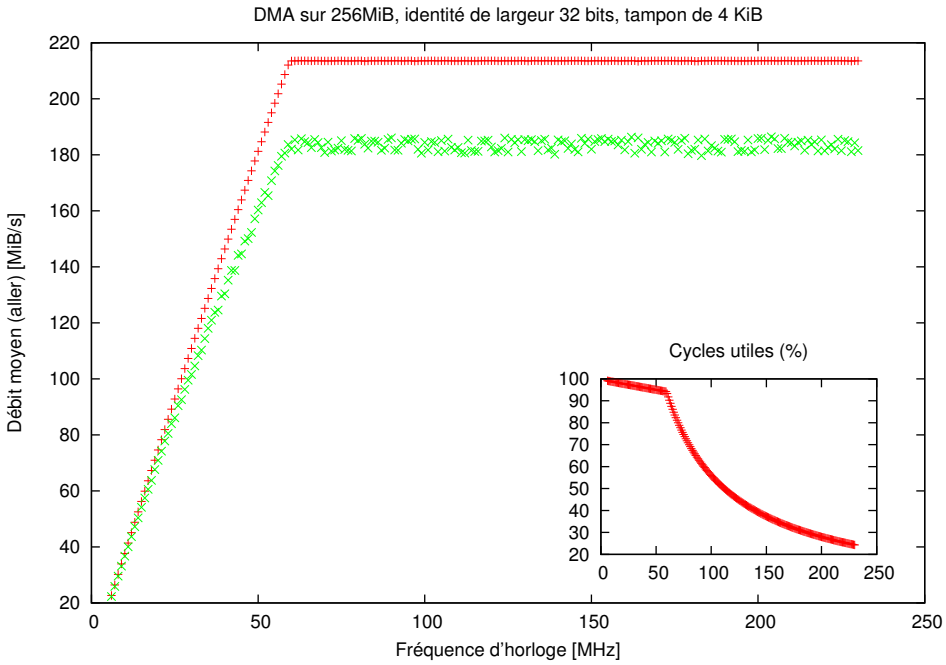


FIG. 4.5 – Vitesse de traitement moyenne mesurée pour une identité de 32 bits pour des valeurs croissantes de la fréquence du DUT

La figure 4.5 présente les mesures de vitesse obtenues en traitant un tampon

[3] hors de l'utilisation de fonctions particulières, non-portables, ou de l'accès au Time Stamp Counter (TSC) des processeurs

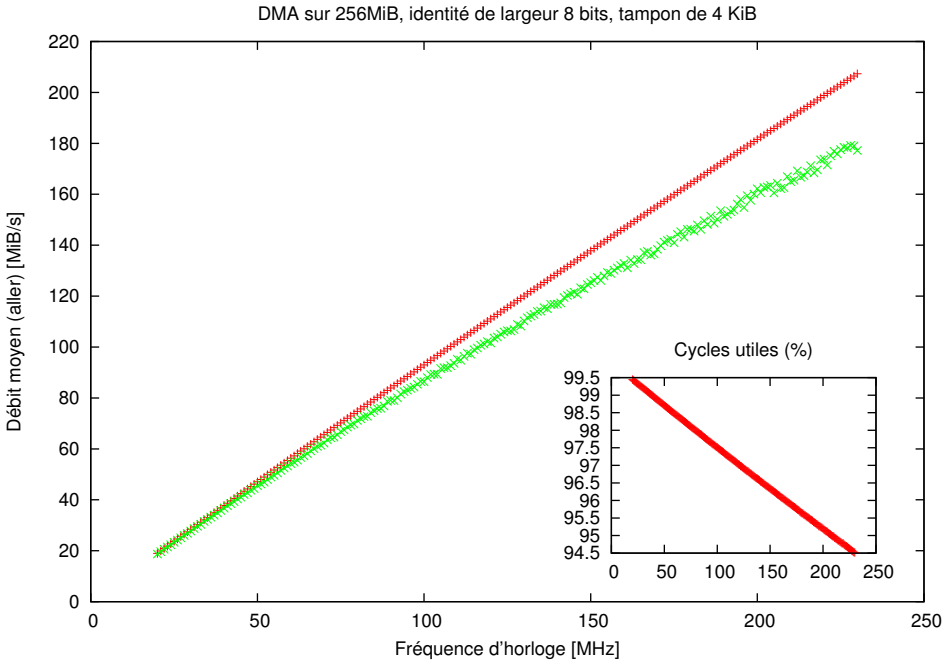


FIG. 4.6 – Vitesse de traitement moyenne mesurée pour une identité de 8 bits pour des valeurs croissantes de la fréquence du DUT

de taille 256MiB. Le DUT est une identité sur 32 bits. Sa fréquence reportée en abscisses, est réglable et croît de 10 à 230 MHz. Le temps de traitement mesuré est utilisé pour calculer la valeur du débit de données moyen en entrée du circuit, exprimé en MiB.s^{-1} et reporté en ordonnées.

Deux situations se distinguent lorsque la fréquence augmente :

1. pour les fréquences inférieures à une fréquence limite f_l, le débit moyen apparaît comme une fonction quasi-linéaire de la fréquence interne du circuit ;
2. pour les fréquences au-dessus de la fréquence limite f_l, le coprocesseur entre en phase de saturation. Le débit moyen atteint un maximum et le temps total de traitement un minimum incompressible. Il correspond au fait que la bande passante maximale est atteinte sur le bus de données PCI, qui ne peut pas fournir les données au circuit suffisamment vite.

Lors de la première phase, c'est la fréquence du circuit qui limite la vitesse de traitement du coprocesseur matériel ; lors de la seconde phase, c'est la bande passante du bus PCI.

Si ces résultats sont conformes à nos attentes, une seconde courbe qui estime le pourcentage de cycles actifs du DUT lors du traitement des données est tracée en encadré dans le graphe 4.5 pour vérifier que l'environnement du DUT a bien un comportement optimal dans la phase linéaire.

Cette seconde courbe laisse apparaître un défaut de notre dispositif expérimental : le ratio de cycles utiles du circuit, s'il oscille entre 95% et 85%, décroît fortement pendant la phase linéaire jusqu'à la fréquence de saturation. Cette

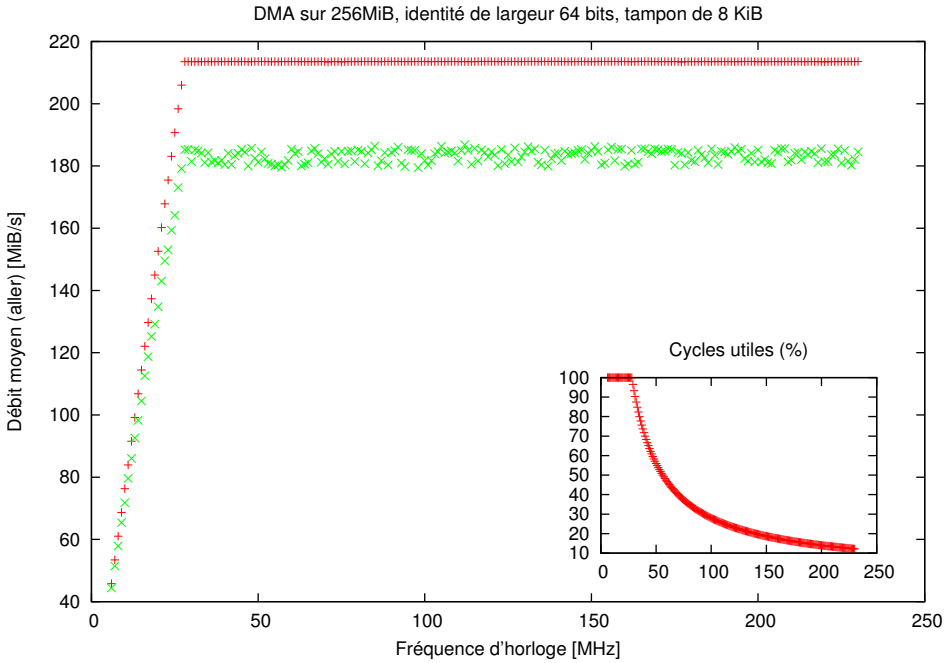


FIG. 4.7 – Vitesse des transferts DMA pour 64 bits de large

situation est gênante pour un circuit sur 8 bits de large, puisqu'alors la bande passante maximale du bus ne peut être atteinte, même pour des fréquences extrêmement élevées du circuit, comme l'indique la courbe 4.6.

4.2.3.6 Taille des tampons FIFO

La déficience mise en évidence lors des mesures précédentes est due à un effet non-linéaire de l'environnement du DUT sur le moteur DMA. En effet, dans les courbes tracées en 4.2.3.5, la FIFO en entrée du DUT a une capacité de 4KiB de données correspondant à deux blocs RAM Xilinx. Cette quantité est un minimum qui correspond à la taille des transferts DMA sur le bus PCI, et dans cette configuration la FIFO d'entrée ne peut contenir qu'une seule de ces trames DMA. Par conséquent le contrôle de flux en entrée lève le signal de demande d'interruption du moteur DMA aussitôt que les premières données parviennent dans la FIFO, et n'autorise un nouveau transfert de trame DMA en entrée du circuit qu'une fois la FIFO entièrement vide. Le circuit est alors confronté à des temps d'attente liés à des effets non-linéaires du contrôle de flux – en particulier lorsque le moteur DMA transfère les résultats produits par le circuit dans la mémoire centrale.

Le flux de données en entrée du circuit est par conséquent régulièrement interrompu. La méthode pour rectifier ce défaut est d'augmenter la taille de la FIFO d'entrée à 8KiB : elle peut ainsi absorber jusqu'à deux trames DMA consécutives, ce qui constitue une réserve de données suffisante pour que le circuit reste actif sans interruption même lors du transfert des données en sortie du circuit vers la mémoire centrale.

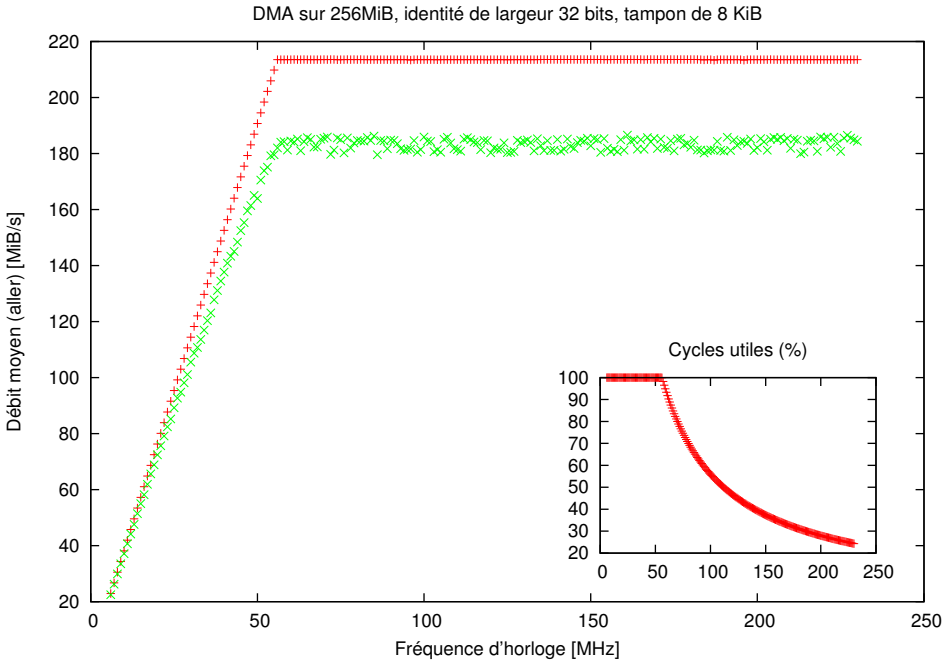


FIG. 4.8 – Vitesse des transferts DMA pour 32 bits de large

Avec ce nouvel appareillage, les flux mesurés sont effectivement optimaux pour toutes les largeurs de l'identité (4.7, 4.8, 4.9, 4.10). Les cycles utiles atteignent 100% lors de la phase linéaire et la fréquence seuil de saturation du bus f_l est abaissée. En particulier, l'identité sur 8 bits de large peut maintenant atteindre la fréquence de saturation du bus (4.10).

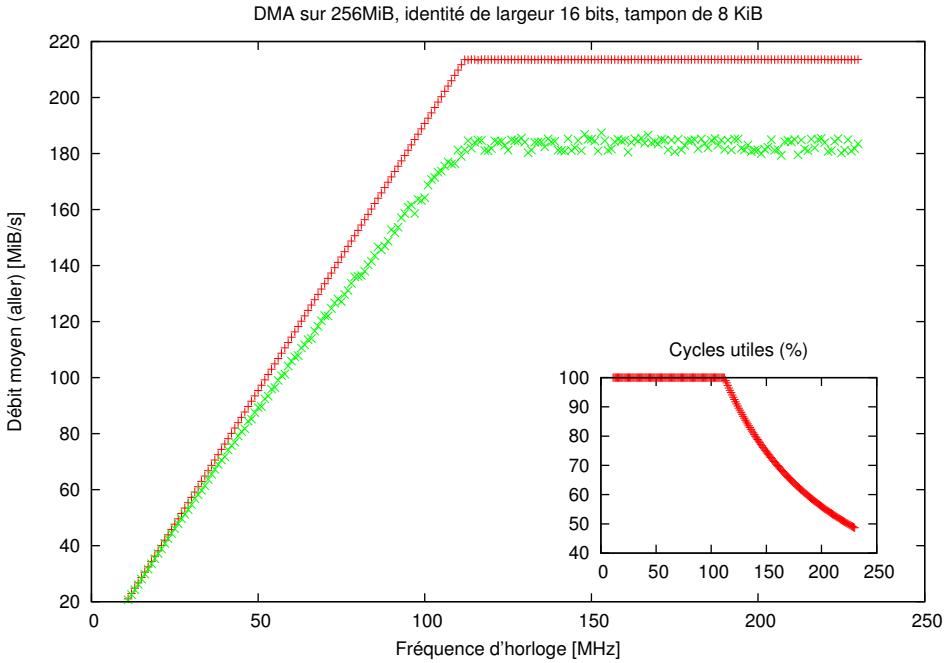


FIG. 4.9 – Vitesse des transferts DMA pour 16 bits de large

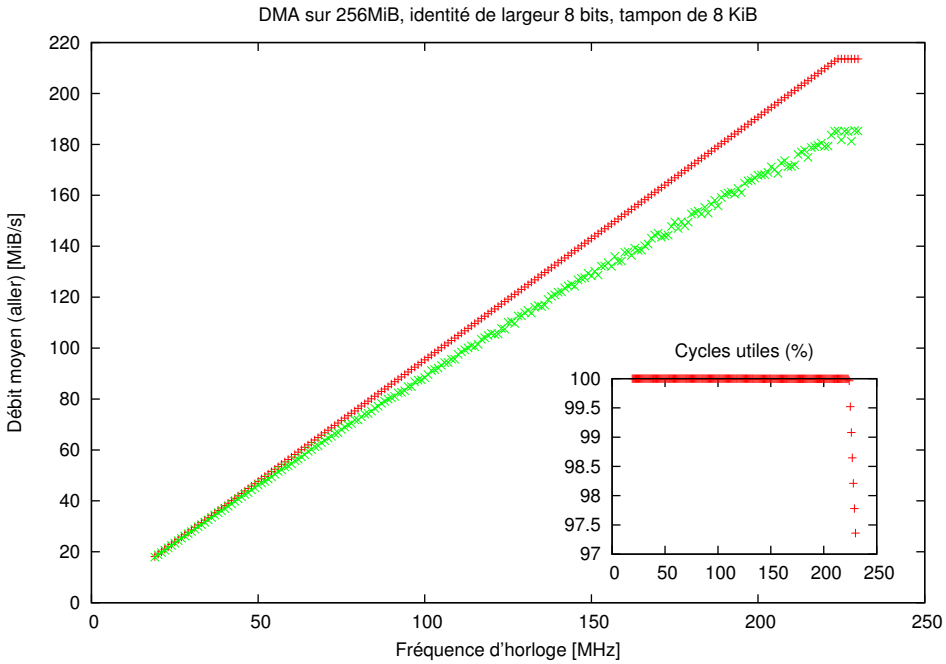


FIG. 4.10 – Vitesse des transferts DMA pour 8 bits de large

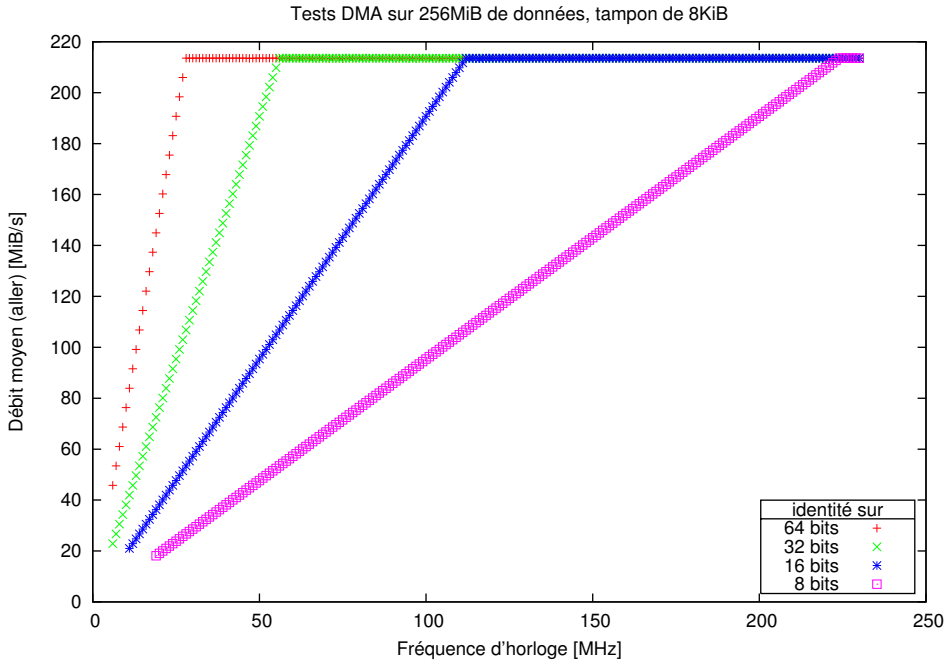


FIG. 4.11 – Performance des transferts DMA sur SEPIA

4.3 Chaîne de compilation

La chaîne de compilation part d'une description du circuit en DSL, l'analyse (chapitre 2), la synthétise (chapitre 3) et y ajoute les routines matérielles nécessaires à son interaction avec SEPIA dans ce chapitre. Elle fait pour toutes ces étapes appel à de nombreux outils. Jazz a été cité au chapitre 2 ; les phases de synthèse et d'ajout des routines matérielles font appel à PamDC, enfin la génération du fichier de configuration final du FPGA utilise les outils propriétaires Xilinx. Cette partie s'attarde – brièvement ! – sur ces deux outils annexes mais capitaux.

4.3.1 Le langage PamDC

PamDC [40] est une librairie C++ qui permet la description hiérarchique de circuits en C++. En PamDC les circuits sont des objets dont une fonction d'instanciation permet de générer une net-liste sur laquelle plusieurs opérations peuvent être réalisées :

- simulation du circuit correspondant,
- génération d'un fichier de description du circuit au format Electronic Design Interchange Format (EDIF).

PamDC est spécialisée pour la technologie Xilinx : elle permet la description du circuit FPGA Virtex II qui est la base de ce travail. En outre, PamDC expose de nombreux éléments de logique spécialisés tels que décrits et utilisés au chapitre 3.

Cette librairie est utilisée comme dorsal pour DSL. C'est à l'aide de cette librairie que l'implémentation effective des primitives du chapitre 3 et l'ajout de l'environnement de communication de ce chapitre sont réalisés. La synthèse d'un circuit écrit en DSL consiste à générer un fichier C++ décrivant le circuit comme un object PamDC. Ce circuit est inséré dans la description complète de la puce Virtex II, le fichier EDIF correspondant à la puce est généré, puis l'ensemble est envoyé pour traitement aux outils Xilinx.

4.3.2 La chaîne de compilation Xilinx

Le placement / routage de la net-liste ainsi que la génération du bitstream de configuration final pour la programmation du FPGA ne sont possibles à l'heure actuelle qu'en utilisant les outils propriétaires des vendeurs de FPGA. En effet, l'architecture interne des puces FPGA tout comme le format de fichier du *bitstream* qui permet de les programmer sont des secrets industriels bien gardés.

Dans le cas de l'architecture Xilinx utilisée dans cette thèse, le passage de la description du circuit sous forme de net-liste au bitstream qui programme la net-liste est un processus long qui fait intervenir de nombreux programmes.

La figure 4.12 montre le processus qui permet d'obtenir le bistream qui sera chargé dans le FPGA (fichier .bit) à partir du fichier de description au format EDIF. Le processus est réalisé par la chaîne de compilation, par étapes dans l'ordre suivant.

Traduction de la net-liste Le fichier EDIF généré par PamDC est converti dans le format de net-liste propre à Xilinx, le format Native Generic Database

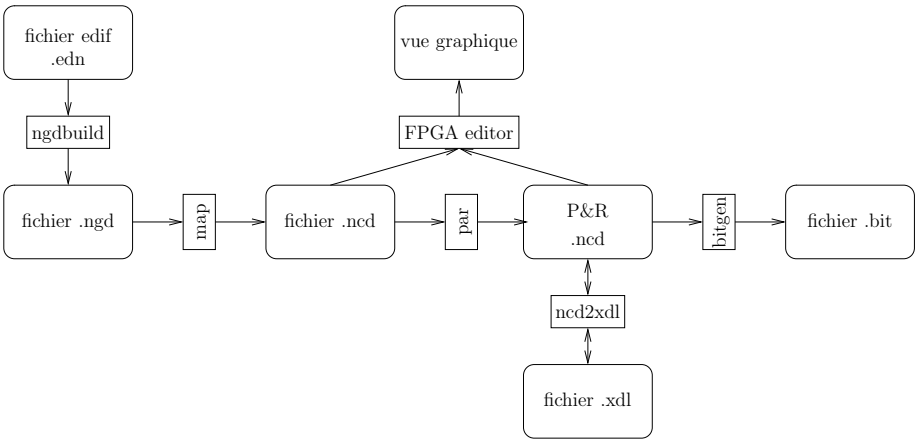


FIG. 4.12 – Outils de compilation Xilinx

(NGD), par le programme `ngdbuild`. Cet outil peut convertir un grand nombre de formats de net-liste en entrée vers le format intermédiaire commun NGD.

Projection de la net-liste Sur la base de la net-liste au format NGD, le programme `map` effectue la projection de la net-liste de haut niveau vers les primitives natives propres au circuit Xilinx ciblé pour la synthèse. Cette association est décrite dans un fichier Native Circuit Description (NCD). À cette occasion, une première évaluation de la longueur du chemin critique du circuit est effectuée. Il s'agit d'une borne inférieure du chemin critique effectif final : elle ne prend en compte que les délais liés dus aux temps de propagation dans les opérateurs logiques – pas encore ceux liés à la propagation des signaux électriques dans les fils, qui ne seront connus que dans les étapes ultérieures.

Une fois cette étape réalisée, le processus de compilation dispose de toutes les informations sur les primitives logiques qui seront effectivement utilisées par le circuit final ; en particulier le taux d'occupation de la logique peut être évalué.

Placement-routage L'étape suivante consiste à localiser ces primitives sur des positions physiques du FPGA – c'est le placement – puis à configurer les interconnexions du FPGA afin de relier les primitives logiques – c'est le routage.

Alors le modèle du circuit permet d'obtenir une évaluation précise du chemin critique du circuit, qui est comparée à la contrainte de temps de propagation spécifiée par l'utilisateur.

L'optimisation du placement-routage sous contrainte est l'étape la plus gourmande en temps de calcul. Les algorithmes de recuit simulé employés pour effectuer le placement-routage sont non-déterministes. Ainsi le résultat du placement-routage est-il toujours différent pour un même circuit DSL initial, qui ne contraint pas le placement des opérateurs arithmétiques eux-mêmes.

Le programme `par` effectue le placement-routage, et le résultat de cette opération est une série d'annotations ajoutées au fichier NCD initial, qui précisent la position de chacun des opérateurs bas niveau ainsi que le routage de chacun des signaux du circuit.

Assemblage du bitstream Une fois que le placement-routage s'est correctement déroulé, le fichier NCD contient les annotations de placement et de routage qui décrivement entièrement l'état interne du FPGA. Le rôle de `bitgen`, ultime programme de la chaîne de compilation Xilinx, est donc simplement de traduire un sous-ensemble de l'information contenue dans le fichier NCD vers le format propriétaire de configuration de l'état interne du FPGA.

Der Teufel steckt im Detail.

5 Applications fonctionnelles

L'intérêt de DSL dépasse le cas d'école présenté dans le chapitre 2. C'est la raison pour laquelle plusieurs exemples d'algorithmes candidats au portage sur une Pam FPGA sont maintenant étudiés.

Dans la section 5.1 le circuit de compression du chapitre 2 est repris et complexifié pour l'adapter à des situations d'utilisation réalistes. La section 5.2 y ajoute deux algorithmes de tramage utilisés dans les têtes d'impression des imprimantes numériques, qui mettent en évidence la concision et la facilité d'utilisation de DSL. La section 5.3 présente un exemple complexe d'estimation de mouvement dans un flux vidéo ; il s'agit de l'exemple d'utilisation de DSL le plus conséquent, qui permet de mettre en évidence une stratégie de codéveloppement logiciel/matériel avec DSL. Enfin la section 5.4 termine sur un exemple d'algorithme d'extraction de points singuliers dans une image. Son implémentation sur FPGA n'est toutefois pas pertinente au regard des résultats obtenus en logiciel.

Ces algorithmes peuvent être dans l'ensemble considérés comme des algorithmes de traitement d'images : les performances de toutes les implémentations présentées sont par conséquent évaluées sous un gabarit commun, à savoir le traitement d'un flux vidéo au format PAL progressif[1].

5.1 Compression d'images

5.1.1 Problèmes de l'algorithme initial

La définition du compresseur/décompresseur d'image en ligne du chapitre 2 suppose que le flux de données en entrée possède deux caractéristiques idéales :

– le flux est *continu*, c'est-à-dire que les données en entrée du circuit sont signifiantes à chaque cycle d'horloge ;

[1] Un tel flux vidéo est constitué d'images de 720×576 pixels à la fréquence nominale de 25 images par seconde.

– le flux est *infini*, c'est-à-dire que le flot d'échantillons ne se termine jamais.

Ces deux caractéristiques du flux d'entrée permettent de garantir une propriété fondamentale du circuit de *codec* : la latence entre le flux de données en entrée et le flux de données – identique sous le signal d'activation – en sortie est bornée. En pratique, la latence croît jusqu'à son maximum de huit cycles pour ne plus varier par la suite.

Toutefois ces hypothèses ne sont pas réalistes : non seulement le flux d'entrée n'est pas continu, mais surtout il n'est pas infini ! Prenons par exemple l'utilisation du *codec* dans le cadre d'un flux DMA comme présenté au chapitre 4 : le *codec* est alors une identité complexe qui s'applique sur un tampon de données de taille finie pour produire un tampon de données identique en sortie.

Dans ces conditions d'utilisation réelle du *codec*, la spécification présentée en 2.1.1.2 pose un problème majeur lié au transformateur de bloc. En effet ce composant mémorise d'un cycle sur l'autre le reste r_n correspondant aux bits de code en quantité insuffisante pour former un bloc de 16 bits. Le décompresseur reste en attente de cette information qui lui est nécessaire pour générer la fin du tampon de données.

Ainsi donc, lorsque le flux en entrée du compresseur s'arrête, par exemple lorsque le moteur DMA a fini de transmettre le tampon de données, le compresseur retient indéfiniment ces r_n bits en l'absence d'autres données en entrée. Du point de vue de l'utilisateur le flux en sortie du décompresseur est incomplet, et il n'en verra pas la fin dans un temps fini. Dans le cadre d'une transaction DMA au travers du *codec*, celle-ci ne terminera jamais.

Il est par conséquent nécessaire de complexifier l'implémentation afin de prendre en compte ces deux paramètres :

– le flux en entrée du circuit n'est sans doute pas continu, et il faut adjoindre aux données un signal d'*enable* ;
– le flux en entrée du circuit est certainement fini, et il faut permettre à l'utilisateur de terminer un flux en purgeant l'ensemble des bits contenus dans le transformateur de bloc du compresseur.

5.1.2 Spécification d'une fonction de purge

L'opération de purge (*flush*) du codeur doit permettre de terminer un flux de manière propre, c'est à dire que les deux blocs du compresseur et du décompresseur doivent pouvoir à la fois reconnaître la fin d'un flux et rester dans un état cohérent. Par état cohérent, on entend un état qui permette d'utiliser à nouveau le circuit pour traiter un nouveau tampon de données, par exemple une nouvelle image, ou un nouveau tampon de DMA.

Par ailleurs la signalisation de la fin du flux au niveau du compresseur devrait répondre à des contraintes d'économie et de simplicité :

– en termes de mémoire, le tampon interne du décompresseur doit rester de taille finie, même en cas de purges répétées ;
– en termes de bande passante, la signalisation de la fin du flux en entrée du compresseur doit être bien pensée pour ne pas ajouter un bit de plus entre le compresseur et le décompresseur.

La solution suivante permet de répondre à l'ensemble de ces contraintes en s'appuyant sur le signal d'*enable* qui est maintenant explicité en entrée du compresseur.

– Le signal de purge est généré automatiquement sur un front descendant du signal d'*enable* en entrée du circuit :

$$f_n = e_{n-1} \& \neg e_n$$

ainsi donc il n'y a pas de signal supplémentaire en entrée du compresseur pour signaler la fin du flux et le compresseur se verra automatiquement purgé en cas d'interruption du flux de données en entrée.
– Le signal de purge est propagé du compresseur au décompresseur au sein même du flux de données grâce à un code spécial réservé dans l'arbre de code de la figure 2.1 ; ainsi donc la bande passante entre compresseur et décompresseur n'est pas augmentée.

Enfin la spécification décrite dans les paragraphes suivants assure la cohérence du *codec* après la purge.

Conséquences de l'ajout de la purge La spécification conduit à des propriétés plus fortes que les réquisitions initiales : le circuit garantit à son utilisateur, en toutes circonstances, de restituer le flux initial en un temps fini : le *codec* est donc une identité de latence finie pour tous les flux, continus ou pas.

Néanmoins la spécification n'est pas sans incidence sur la qualité de la compression. En premier lieu, réserver un mot code à la commande de purge dans l'arbre d'Élias est fait au détriment de la qualité de compression pour la valeur auparavant associée à ce mot code. Toutefois l'effet sur le taux de compression du passage d'une valeur du code de 8 bits (code d'Élias de la différentielle) à 9 bits (codage direct de la valeur du pixel) pour un unique élément reste marginal.

Par ailleurs les codes de *flush* et bits de bourrage associés ponctuent le flux compressé en fonction des variations de l'*enable* d'entrée. Dans le cas de l'utilisation du *codec* pour traiter un tampon de DMA par exemple, les retours à zéro de l'*enable* d'entrée sont des événements très rares – au maximum 1 événement pour 4KiB de données en entrée.

5.1.2.1 Compresseur

L'ensemble du compresseur est revu par rapport à 2.1.1.2 en ajoutant un signal d'*enable* explicite en entrée et en ajoutant la génération et le support d'un signal de purge à partir de l'*enable*.

La suite des trois étapes de transformation, codage et compression est maintenant transformée pour générer et prendre en compte ces signaux :

– la transformée $T : \mathbb{B}^8 \times \mathbb{B} \to \mathbb{B} \times \mathbb{B}^8 \times \mathbb{B}$ accepte en entrée un signal supplémentaire d'*enable* et génère sur un bit supplémentaire en sortie le signal de *flush* ;
– le codeur $C : \mathbb{B} \times \mathbb{B}^8 \times \mathbb{B} \times \mathbb{B} \to \mathbb{B}^9 \times \mathbb{B} \times \mathbb{B}$ prend en compte les deux signaux de contrôle supplémentaires correspondant à l'*enable* et au *flush*, ce dernier étant propagé jusqu'à la sortie du codeur ;
– le normalisateur par bloc $N : \mathbb{B}^9 \times \mathbb{B} \times \mathbb{B} \to \mathbb{B} \times \mathbb{B}^{16}$ prend lui aussi en compte les signaux de *flush* et d'*enable* additionnels sur son entrée.

Différentielle et signal de purge Nous choisissons de sortir la valeur de différentielle -11 de l'arbre d'Élias afin de réserver son code pour signifier la commande de *flush* en sortie du compresseur. La différentielle initiale est par conséquent amendée de la manière suivante :

– Un pixel conduisant à une valeur différentielle de -11 est dorénavant décompressé, et sa valeur codée directement sur 9 bits ;
– le signal de *flush* est généré lorsque l'*enable* passe de l'état actif à l'état inactif.

Soient $(i_n)_{n\in\mathbb{N}}$ la suite des valeurs des pixels et $(e_n)_{n\in\mathbb{N}}$ la suite des valeurs du signal d'*enable* présenté en entrée du compresseur aux temps $n \in \mathbb{N}$, on définit (u_n, t_n, f_n) les sorties du transformeur comme :

$$\begin{array}{l}
\left\{\begin{array}{lcl} j_{-1} = 0 & & \\ e_n = 1 & \Longrightarrow & j_n = i_n \\ e_n = 0 & \Longrightarrow & j_n = j_{n-1} \end{array}\right. \\
d_n = j_n - j_{n-1} \\
u_n = (|d_n| > 11) \vee (d_n = 11) \\
\left\{\begin{array}{lcl} u_n = 1 & \Longrightarrow & t_n = i_n \\ u_n = 0 & \Longrightarrow & t_n = d_n \end{array}\right. \\
f_n = e_{n-1} \& \neg e_n ;
\end{array}$$

5.1.2.2 Codeur

Le codeur est identique à la description faite en 2.1.1.2, à ceci près que la nouvelle entrée $(f_n)_{n\in\mathbb{N}}$ commande dorénavant la production par le codeur du code réservé de *flush*, c'est-à-dire $Elias(-11) = 01111110$ de longueur 8 bits.

Au niveau de l'implémentation, afin de prendre en compte les états du circuit où le signal d'*enable* est a zéro, la donnée de longueur de code l_n transmise au bloc coder est mise à zéro si l'*enable* e_n est nul :

$$e_n = 0 \implies l_n = 0$$

Pour le reste, le calcul de (c_n, l_n) est identique à celui décrit pour le codeur original en 2.1.2.2.

5.1.2.3 Normalisateur par bloc

L'effet du signal de purge f_n sur le normalisateur par bloc est plus complexe que pour les deux autres blocs fonctionnels ; ce signal force la production d'un mot de code supplémentaire en sortie.

Plus précisément, le signal $f_n = 1$ est accompagné du mot de code $c_n = Elias(-11)$, de longueur 8 bits en sortie du codeur et provoque dans le normalisateur l'ajout du code de *flush* proprement dit et du nombre de bits de bourrage nécessaire pour aligner la longueur cumulée des codes précédents – y compris le code de *flush* – sur un multiple de 16 bits.

Plus précisément, deux cas peuvent se présenter lors de l'arrivée d'un signal de purge et du mot de purge associé.

– Aucun mot n'aurait été émis au cycle n. Alors l'effet du signal de purge est de forcer l'émission d'un mot en sortie du compresseur *immédiatement*, accompagné de $16 - |r_n|$ bits de bourrage.
– Un mot est naturellement émis au temps n en raison de l'arrivée du mot de *flush*. Le tampon contient au cycle suivant $|r_{n+1}||$ bits de reste du mot de purge. Alors la logique force l'émission d'un mot de code contenant les bits restant r_{n+1} accompagnés de $16 - |r_{n+1}|$ bits de bourrage au cycle suivant.

Circuit	Vitesse		Ressources		Bande passante
	ns	MHz	RAMB	Tranches	utile MiB.s^{-1}
original	20.29	49	0 (0%)	136 (1.2%)	49
retemporisé	5.79	172	0 (0%)	295 (2.7%)	172

TAB. 5.1 – Implémentation matérielle du compresseur

Circuit	Vitesse		Ressources		Bande passante
	ns	MHz	RAMB	Tranches	utile MiB.s^{-1}
original	38.34	26	0 (0%)	755 (7.0%)	26
retemporisé	24.02	41	0 (0%)	966 (8.9%)	41

TAB. 5.2 – Implémentation matérielle du *codec*

Cette disjonction des cas permet de montrer que le tampon interne au normalisateur ne change pas de taille : la quantité de données signifiantes à mémoriser d'un cycle sur l'autre est inchangée, la valeur des bits de bourrage ayant peu d'importance.

Il est d'ores et déjà important de constater que ce processus est inversible. En effet le nombre de bits de bourrage ajoutés n'est fonction que de l'état antérieur du circuit, et peut donc être prédit avec exactitude par le décodeur au moment de la reconnaissance du signal de *flush*.

5.1.2.4 Décompresseur

L'ensemble du décompresseur n'est pas beaucoup modifié par rapport à la spécification initiale. Seul le couple tampon d'entrée/décodeur est modifié pour reconnaître le mot codant la purge et mettre à jour son état interne en conséquence : il s'agit simplement de supprimer du tampon à la fois le mot de code de purge et les bits de bourrage associés, dont on vient de voir que leur nombre est prédictible.

Tampon d'entrée et décodeur L'ajout du signal de purge entraîne la mise en place d'un décodeur spécifique chargé de signaler la présence du mot de purge dans le flux ainsi que d'un module qui permet de calculer le nombre de bits de bourrage qui le suivent.

Ce module est une machine à état qui calcule la valeur du reste r'_n – c'est-à-dire de l'état interne du normalisateur par bloc au moment où le code c_n a été émis en sortie du codeur. Ainsi donc, le décodeur peut prévoir le nombre de bits de bourrage qui ont été ajoutés par le normalisateur par bloc pour supprimer du tampon d'entrée ces bits non-significatifs.

5.1.3 Résultats d'implémentation

5.1.3.1 Compresseur

Le tableau 5.1 résume les résultats d'implémentation pour le compresseur seul. Le compresseur est un circuit sans boucle de rétroaction ; ainsi donc les vitesses obtenues après retemporisation automatique du circuit sont excellentes.

Version	Images	Durée du traitement		Vitesse	
		totale	par image	$Im.s^{-1}$	$MiB.s^{-1}$
memcpy	300	.273s	.911ms	1097.5	455.1
logiciel	300	4.081s	13.604ms	73.5	30.4
Pam	300	7.267s	24.224ms	41.2	17.1
Pam, retemporisé	300	4.196s	13.987ms	71.4	29.6

TAB. 5.3 – Performances de l'implémentation du compresseur/décompresseur

Par rapport à la synthèse du compresseur sans purge (tableau 2.2), l'augmentation de logique est minimale.

5.1.3.2 Compresseur-décompresseur

Le seul effet sur le circuit de compression / décompression des modifications liées à l'ajout de la purge, outre l'augmentation de la surface du circuit, est d'allonger la boucle critique au sein du décodeur dans le décompresseur. Cette même boucle critique limite les performances de la retemporisation automatique du circuit, qui n'atteint pas les 50 MHz. L'optimisation de la boucle est hors de portée de notre méthode de retemporisation automatique.

5.1.3.3 Performances

Les performances sont évaluées sur l'identité que constitue le *codec* dans son ensemble. En effet, l'architecture d'accès DMA aux circuits se prête mal à l'évaluation des performances de circuits à débit variable comme le sont les compresseurs ou décompresseurs seuls.

La tableau 5.3 présente les performances de l'algorithme mesurées sur un flux d'images PAL. La première entrée du tableau montre la version logicielle d'une identité : la fonction système optimisée *memcpy* est utilisée comme référence. Le deuxième entrée présente la performance de l'implémentation logicielle de l'algorithme de compression en ligne. Enfin, les deux entrées suivantes montrent les performances de la version matérielle du *codec*, en premier lieu la version brute du circuit, en deuxième lieu la version retemporisée.

Pour ce premier exemple de circuit, l'usage du coprocesseur ne permet pas un gain de vitese par rapport à la version logicielle pure. Toutefois, la charge processeur enregistrée lors de l'exécution de la routine est beaucoup moins importante lorsque SEPIA est utilisée.

5.2 Tramage numérique

Notre compilateur est maintenant utilisé pour générer l'implémentation matérielle de deux algorithmes de tramage numérique. Le tramage est l'art, en imprimerie, de reproduire des images en tons continus à l'aide des procédés d'impression discrets en quadrichromie (couleur) ou en monochromie (noir et blanc). Son incarnation moderne, le tramage numérique, consiste à reproduire des images en tons continus (typiquement 24 bits par pixel) sur des périphériques ou quadrichromes (imprimantes couleurs Cyan, Magenta, Jaune, Noir (CMJN)) ou monochromes (imprimantes noir et blanc).

Les imprimantes à jet d'encre ou à aiguilles doivent exécuter de tels algorithmes à une vitesse suffisante pour atteindre l'impression temps réel. La vitesse d'exécution requise dépend de la résolution et de la vitesse d'impression ciblées. Nous utilisons ici la terminologie et des exemples tirés du monde des imprimantes à jet d'encre noir et blanc. Les méthodes de tramage sont très similaires pour les imprimantes laser et à aiguilles, qu'elles soient couleur ou monochromes.

5.2.1 Algorithme

L'image-source est une image monochrome codée en 256 niveaux de gris (8 bits par pixel). La largeur d'image est fixée à 720 pixels. Cette dimension, qui correspond à la largeur des images au format PAL, sera identique pour l'ensemble des algorithmes de traitement d'image présentés dans cette thèse.

La tête d'impression parcourt l'image imprimée en ordre raster-scan[2], à la vitesse d'une colonne de points d'impression par cycle d'horloge. La sortie binaire i du circuit de tramage contrôle numériquement chacune des buses de la tête d'impression. En conséquence, une goutte d'encre est déposée ($i = 1$) ou pas ($i = 0$) sur la position courante. La valeur unitaire de la goutte d'encre correspond à une teinte saturée de valeur 256.

Afin de prendre la décision de déposer la goutte d'encre, la valeur en ton continu p de chaque pixel en entrée est ajoutée à l'erreur d diffusée depuis les pixels précédents :

$$e = p + d$$

La quantité $e/256$ représente la quantité fractionnaire d'encre qu'une imprimante hypothétique à 256 niveaux de gris devrait idéalement déposer à la position courante. L'erreur diffusée vers les pixels suivants, $r = e - 256i$ est la différence entre la quantité d'encre effectivement déposée et cette quantité théorique.

La diffusion de l'erreur commise du pixel courant vers les pixels voisins dans l'image permet de conserver localement la quantité d'encre totale.

Le tramage numérique est par conséquent réalisé en deux étapes distinctes, qui sont unies au sein d'une boucle qui permet la diffusion de l'erreur d'un cycle sur l'autre :

- *dropInk* est la partie du circuit chargée de la prise de décision du contrôle de la sortie digitale i et de la buse terminale ;
- *diffuseError* diffuse l'erreur r commise à la position courante vers les pixels adjacents.

[2] l'ordre boustrophédonique est aussi possible

5.2.1.1 Dépôt de la goutte d'encre

La décision binaire i d'envoyer ou pas une goutte d'encre résulte de la comparaison de la quantité d'encre totale e au seuil h :

$$i = t \geq h$$

Deux versions de $dropInk$ sont étudiées :

– Dans ce premier code naïf, le seuil est constant et égal à la quantité nominale d'encre par goutte, $h = 256$.

$$\begin{array}{l} fun \quad i \quad = \quad \mathbf{dropInk_{RD}(e)} \quad \{ \\ \quad // \ ink \ drop \ if \ e \geq 256 \ ; \ i \in \{0,1\} \\ \quad i \quad = \quad e \gg 8; \\ \} \end{array} \tag{5.1}$$

– Dans la version avancée, une table de seuillage th (disons de taille 16) est utilisée pour calculer une valeur pseudo-aléatoire du seuil en prenant comme index les bits de poids faible de la quantité à seuiller :

$$h = th[t\&15] = th[t[0..3]]$$

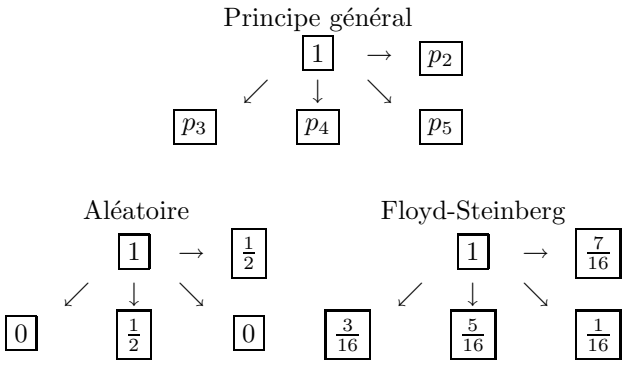
Ce seuillage élimine très efficacement les effets de Moiré présents dans la version précédente lors du rendu d'aplats (zones de couleur uniforme).

$$\begin{array}{l} var \quad th \quad = \quad [10, 15, 11, 8, 12, 14, 12, 13, 15, 11, 14, \\ \qquad\qquad 15, 9, 13]; \\ \quad // \ Threshold \ Table \ th[0..15] \in [8, 15] \end{array}$$

$$\begin{array}{l} fun \quad i \quad = \quad \mathbf{dropInk_{FS}(e)} \quad \{ \\ \quad h \quad = \quad th[e\&15]; \\ \quad // \ threshold \ value \ ; \ 8 \leq h < 16 \\ \quad i \quad = \quad (e \gg 4) > h; \\ \quad // \ ink \ drop \ if \ e > 16h \ ; \ i \in \{0,1\} \\ \} \end{array} \tag{5.2}$$

5.2.1.2 Diffusion de l'erreur

Lorsque la valeur nominale de $256i$ unités d'encre par goutte n'est pas égale à e, il reste une erreur $r = e - 256 \times i$. Cette erreur est divisée et diffusée sans perte vers les pixels avoisinants. Une telle répartition permet de préserver la quantité d'encre localement.



Dans les trois situations mises en évidence ci-dessus, l'erreur r calculée à la position $\text{P}[r, c]$ est diffusée vers les quatre pixels avoisinants $\text{P}[r, c+1]$, $\text{P}[r+1, c-1]$, $\text{P}[r+1, c]$, $\text{P}[r+1, c+1]$, en proportion de la fraction $p_k \times r$ pour $k \in [0..3]$, et de telle sorte que $\sum_{0 \leq k < 4} p_k = 1$.

5.2.1.3 Méthodologie et résultats

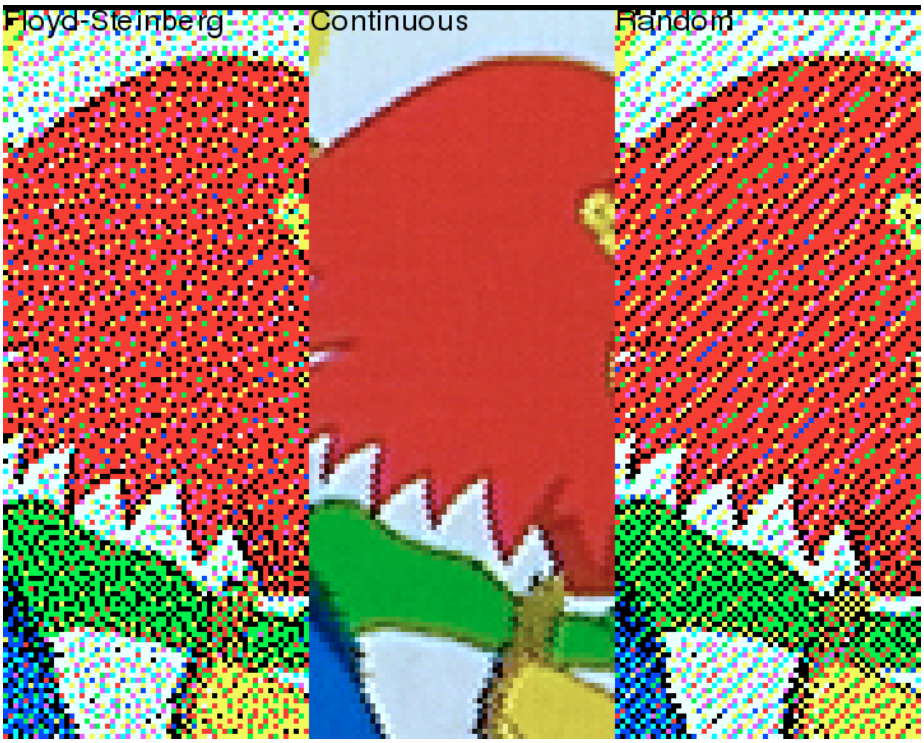


FIG. 5.1 – Comparaison des diffusions aléatoire et de Floyd-Steinberg

Les algorithmes de *diffusion aléatoire* et de la *diffusion de Floyd-Steinberg* sont présentés et synthétisés sur le coprocesseur matériel SEPIA.

Dans la réalité de l'utilisation du tramage numérique sur les images couleurs en CMJN, il existe des interactions entre les quatre canaux de couleurs, qui ne sont pas traités de manière indépendante.

Par ailleurs, CMJN est un système de synthèse additif des couleurs. Or les formats d'image informatiques standards sont en général orientés vers l'affichage des images sur écran, et utilisent par conséquent le système de représentation des images sur trois canaux de couleur en synthèse additive, le système Rouge, Vert, Bleu (RVB). À ces trois canaux chromatiques s'ajoute potentiellement le canal de transparence dit canal α.

Afin de simuler l'utilisation des algorithmes de tramage dans les imprimantes couleur, le choix s'est porté sur l'utilisation d'images en représentation Rouge, Vert, Bleu, Alpha (RVBA). Ainsi donc, l'application de l'algorithme de tramage

Version	Images	Durée du traitement		Vitesse	
		totale	par image	$Im.s^{-1}$	$MiB.s^{-1}$
logiciel	300	10.153s	33.843ms	29.5	49.0
SIMD	300	3.565s	11.885ms	84.1	139.5
Pam	300	5.021s	16.737ms	59.7	99.1
Pam retemporisé	300	4.944s	16.481ms	60.6	100.6

TAB. 5.4 – Performances de l'implémentation de la diffusion aléatoire

nécessite le traitement de 4 canaux, comme pour le CMJN, mais contrairement à ce dernier, on réalise une approximation en traitant chacun de ces canaux de manière indépendante.

La figure 5.1 présente le résultat de l'application des algorithmes de diffusion sur une image-test.

L'algorithme de diffusion aléatoire se comporte dans le domaine fréquentiel comme un filtre aléatoire, et l'image résultante comporte des hautes fréquences qui conduisent à des effets de Moiré correspondant à un bruit blanc décelable pour l'œil. L'algorithme de diffusion de Floyd-Steinberg élimine ces effets désagréables ; il se comporte comme un filtre passe-bas dans le domaine fréquentiel, éliminant ainsi les hautes fréquences perceptibles dans le domaine spatial. Le bruit blanc sur une surface uniforme se transforme en bruit bleu, que l'œil ne perçoit pas.

5.2.2 Diffusion aléatoire

5.2.2.1 Implémentation logicielle

L'implémentation logicielle de référence de l'algorithme de diffusion aléatoire est très simple, et ses performances sont raisonnables, comme le montre la première entrée de la table 5.4. Elles permettent immédiatement de traiter en temps réel des flux vidéo PAL.

```
int DitherRd      (const unsigned char px) {
static int        zd, za[L], idx = 0 ;
int e     =       px + zd,
  i       =       e ≫ 8,
  a       =       (e ≫ 1) & 127,
  d       =       a + (e & 1) + za[idx] ;
za[idx]   =       a ;
  zd      =       d ;
  idx     =       (idx + 1) % (L) ;
return            i ; }
```
(5.3)

5.2.2.2 Implémentation logicielle SIMD

Les processeurs modernes disposent de jeux d'instructions dits "multimédia", qui sont techniquement des instructions SIMD : une seule instruction exécute simultanément la même opération arithmétique sur un tableau d'opérandes. Ces jeux d'instruction permettent d'exploiter un parallélisme fin au niveau du traitement des données dans un programme.

Circuit	Vitesse		Ressources		Bande passante
	ns	MHz	RAMB	Tranches	utile MiB.s^{-1}
original	7.90	126	0 (0%)	743 (6.9%)	506
retemporisé	7.87	127	0 (0%)	773 (7.1%)	508

TAB. 5.5 – Implémentation matérielle de la diffusion aléatoire

Si ces jeux d'instruction ont une histoire ancienne, qui remonte aux processeurs vectoriels, le plus connu de ces jeux d'instruction est l'ensemble Matrix Math Extensions (MMX™) introduit par Intel comme extension de l'architecture x86.

L'utilisation d'un tel jeu d'instruction est tout-à-fait adaptée au traitement en parallèle des quatre canaux de couleur présents dans l'image originale. Nous utilisons à cette fin le jeu d'instruction Streaming SIMD Extensions 2 (SSE2™), qui permet de traiter simultanément quatre entiers de 32 bits qui sont compactés dans un registre de 128 bits.

La transformation du code initial en code SSE2™ est quasiment directe grâce à l'utilisation des fonctions et types spécialisés fournis par le compilateur GNU Compiler Collection (GCC). Le remplacement consiste, d'une part, à remplacer les types entiers de base par les types SSE2™, et d'autre part, à remplacer les fonctions standard sur les entiers en notation infixe par des appels à la fonction préfixe qui représente l'instruction réalisant l'opération arithmétique sur le vecteur d'entiers.

Le code suivant est directement issu du code 5.3 par une telle transformation :

```
static const __m128i mask_127 = _mm_set1_epi32(127);
static const __m128i mask_1   = _mm_set1_epi32(1);

__m128i DitherRd              (__m128i px) {
static __m128i                zd, za[L];
static int                    idx = 0;
__m128i e                     = _mm_add_epi32(px, zd),
i                             = _mm_srli_epi32(e, 8),
ed                            = _mm_srli_epi32(e, 1),
a                             = _mm_and_si128(ed, mask_127),
em                            = _mm_and_si128(e, mask_1),
d                             = _mm_add3_epi32(a, em, za[idx]);
za[idx]                       = a;
zd                            = d;
idx                           = (idx + 1) % (L);
return                        i; }
```

Les résultats obtenus (tableau 5.4) sont parfaitement alignés sur ce que nous pouvions attendre théoriquement : en situation réelle, l'utilisation du jeu d'instruction SSE2™ permet de tripler la vitesse de traitement des images par comparaison avec le traitement séquentiel des quatre canaux.

5.2.2.3 Implémentation matérielle

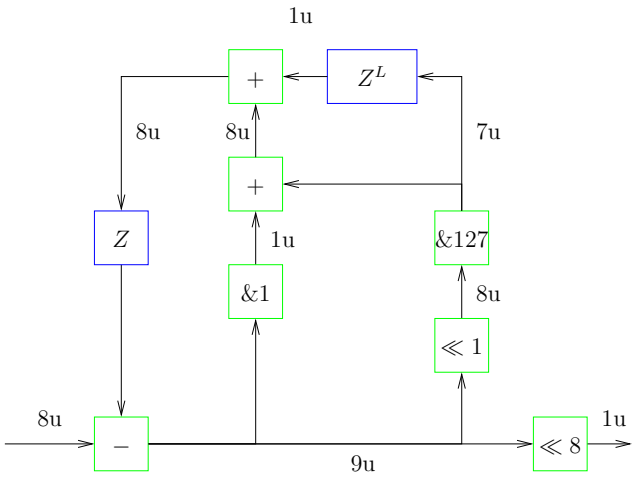


FIG. 5.2 – Schéma du circuit de diffusion aléatoire

Résultats de la synthèse La synthèse de la diffusion aléatoire ne présente aucune difficulté majeure : l'analyse d'intervalles converge au bout de 3 itérations sur le code initial. Le code source a été annoté pour conduire à l'utilisation de la mémoire distribuée pour la synthèse des registres à décalage, suivant 3.3.3.3, ce qui explique l'absence de blocs de RAM dans les ressources utilisées (table 5.5). Structurellement, le circuit, figuré en 5.2, est essentiellement constitué d'une grande boucle, donc d'une seule composante fortement connexe où réside le chemin critique du circuit. Par conséquent, la retemporisation automatique du circuit n'a aucun effet sur la fréquence de fonctionnement maximale du circuit. Cela n'a pas beaucoup d'importance, puisqu'à la fréquence nominale atteinte de 126 MHz pour 32 bits de données en entrée du circuit, la bande passante du bus PCI est déjà saturée à l'exécution.

Performances La mesure effective de la vitesse de traitement des images par l'accélérateur matériel n'atteint toutefois pas la limite mesurée à plus de 200MiB.s^{-1} au chapitre 4. L'explication est simple, puisque l'implémentation choisie utilise la librairie de traitement synchrone des données. Pour chaque image à traiter, le processeur prépare le tampon de DMA, en construit le descripteur, le soumet pour traitement à SEPIA, puis attend l'interruption signalant la fin du traitement avant de libérer descripteur et tampon. La Pam n'est pas active lors du travail du processeur aux autres tâches, et inversement le processeur ne fournit aucun travail lorsqu'il attend les résultats de la Pam.

Toutefois le traitement en temps réel du flux vidéo PAL est ici atteint sans plus de complications. La performance de la routine exécutée sur la Pam dépasse la vitesse de l'implémentation logicielle naïve, mais reste en-dessous des performances obtenues avec le SIMD – il faudrait utiliser une communication asynchrone pour parvenir à dépasser ce résultat.

Version	Images	Durée du traitement		Vitesse	
		totale	par image	$Im.s^{-1}$	$MiB.s^{-1}$
logiciel	300	12.778s	42.593ms	23.4	38.9
SIMD	300	6.253s	20.845ms	47.9	79.5
Pam	300	5.051s	16.837ms	59.3	98.5
Pam retemporisé	300	5.014s	16.714ms	59.8	99.2

TAB. 5.6 – Performances des implémentations de la diffusion de Floyd-Steinberg

5.2.3 Diffusion de Floyd-Steinberg

5.2.3.1 Implémentation logicielle

Ici encore, l'implémentation logicielle de référence ne présente pas de difficulté majeure. La vitesse de traitement présentée en table 5.6 diminue cependant significativement par rapport à l'algorithme de diffusion aléatoire.

```
                static int
thresholds[16]  =  { 10, 15, 11, 8, 10, 15, 13, 12,
                     14, 8, 14, 12, 15, 9, 13, 9 };
           int floyd
                   (const unsigned char px) {
           static int
                   zc, ze, zret, zl[L-2];
           static int
                   i = 0;
           int te  =  px + zret,
               e0  =  te & 15,
               eq  =  te >> 4,
               di  =  (thresholds[e0] < eq),
               e1  =  di ? eq-16 : eq,
               ce  =  zl[i];
             zret  =  ce + e0 + 7*e1;
            zl[i]  =  3*e1 + zc;
               zc  =  5*e1 + ze;
               ze  =  e1;
                i  =  (i + 1) % (L-1);
           return
                   di; }
```
(5.4)

5.2.3.2 Implémentation logicielle `SIMD`

La même méthodologie que celle décrite pour la diffusion aléatoire permet de convertir la majorité du code C de diffusion de Floyd-Steinberg en une version exploitant le jeu d'instruction SSE2™. Une partie de l'algorithme résiste cependant : il s'agit de l'accès au tableau de seuillage, qui consiste simplement en un accès mémoire mais ne peut pas s'exprimer de manière simple comme une fonction arithmétique. Par conséquent, l'implémentation doit sortir du flot SIMD pour aller effectuer les quatre accès au tableau de seuil. Il s'agit d'extraire du vecteur d'entiers contenu dans un registre XMM chacun des entiers, d'effectuer les accès correspondants vers le tableau, puis de réassembler le résultat dans un nouveau registre XMM.

L'augmentation de performances est donc beaucoup moins conséquente, puisque la mesure finale de vitesse indique une vitesse seulement deux fois supérieure à l'implémentation logicielle sérialisée de référence.

Circuit	Vitesse		Ressources		Bande passante
	ns	MHz	RAMB	Tranches	utile MiB.s^{-1}
original	14.79	67	4 (7.1%)	305 (2.8%)	270
retemporisé	14.96	66	4 (7.1%)	385 (3.5%)	267

TAB. 5.7 – Implémentation matérielle de la diffusion de Floyd-Steinberg

5.2.3.3 Implémentation matérielle

Résultats de la synthèse La synthèse de la diffusion de Floyd-Steinberg est plutôt simple : l'analyse d'intervalles converge après l'ajout d'une seule instruction de *bitsizing* dans la boucle principale du circuit [41].

Par comparaison, l'utilisation du programme ASTRÉE [18] permet d'analyser automatiquement l'intervalle de confiance pour toutes les variables de la version C du programme de Floyd-Steinberg, sans aucune annotation manuelle.

Le code source a été cette fois-ci annoté pour forcer l'utilisation de blocs RAM pour la synthèse des registres à décalage, ce qui explique l'utilisation de 4 blocs de RAM (table 5.7). La structure générale du circuit est identique à la diffusion aléatoire, avec la même incidence sur la retemporisation, qui est inutile. La fréquence nominale du circuit tombe à 68MHz, en raison d'une plus grande complexité dans la boucle principale.

Performances Les performances mesurées sont plus favorables à la Pam : le gain de performances obtenu par l'utilisation des instructions SIMD en logiciel n'est cette fois pas suffisant pour atteindre les performances du co-processeur matériel.

5.3 Estimation de mouvement

Nous présentons dans cette partie l'implémentation efficace de l'algorithme d'estimation de mouvement *PixelMatch*. *PixelMatch* calcule un vecteur de mouvement pour chaque pixel entre deux trames vidéo consécutives. Ce calcul est utilisé pour le suivi de mouvement dans un flux vidéo ; il est très proche des calculs d'estimation de mouvement par blocs utilisés en compression vidéo, par exemple dans la norme Motion Picture Expert Group (MPEG), qui calcule un vecteur de mouvement par bloc de pixels.

Nous commençons par présenter la spécification mathématique, c'est-à-dire la spécification du calcul réalisé par l'algorithme, puis décrivons une série de transformations du code de haut niveau original, afin de parvenir à une description intermédiaire optimisée.

À partir de ce code intermédiaire, nous synthétisons grâce à DSL un circuit matériel permettant le traitement de flux vidéo PAL en temps réel sur la Pam SEPIA. L'implémentation logicielle de ce même code intermédiaire permet d'écrire un programme aux performances très supérieures à la spécification originale. Le traitement logiciel de séquences vidéos en temps réel reste toutefois difficile à moins d'utiliser des machines très coûteuses.

5.3.1 Description de l'algorithme

PixelMatch estime le mouvement de chaque pixel entre deux images consécutives I_1 et I_2 dans un flux vidéo de résolution $W \times H$. Pour chaque pixel dans I_1, un voisinage est exploré dans I_2 afin d'y trouver le pixel le plus proche, c'est-à-dire celui qui a la meilleure corrélation avec le pixel original. Le vecteur de mouvement entre le pixel original et le pixel le mieux corrélé dans I_2, ainsi que la corrélation associée, sont enregistrés comme résultat du calcul pour le pixel. La fonction de corrélation utilisée est la Somme des valeurs Absolues des Différences (SAD) entre les pixels de I_1 et ceux de I_2. La SAD est prise sur un voisinage carré autour du pixel de référence.

Ainsi donc la spécification de *PixelMatch* contient-elle trois boucles de recherche. L'analyse détaillée des trois boucles, de leurs opérations internes, de leurs besoins en mémoire et bande passante nous permet de mettre en évidence un agencement optimal du parcours des deux images d'entrées.

Le circuit matériel final, qui permet d'atteindre le traitement en temps réel de flux vidéos au format PAL, est obtenu par déroulement *a minima* dans l'espace et dans le temps d'une unité de calcul optimisée.

5.3.1.1 Spécification

Soient I_1 et I_2 deux images de résolution $W \times H$. Ces deux images sont consécutives dans le flux vidéo. L'objectif est de produire une estimation du mouvement de chaque pixel de l'image de référence I_1 vers l'image de destination I_2. À cette fin, étant donné un pixel de I_1, on se propose d'en trouver le *pixel-image*, son jumeau dans I_2.

Corrélation La valeur $\mathcal{E}_{\vec{d}}(p_0)$ quantifie la ressemblance entre le pixel situé à la position p_0 dans I_1 et le pixel candidat situé à la position translatée $p_0 + \vec{d}$ dans I_2. La fonction classique de somme des valeurs absolues des différences

(SAD) calculée sur une région autour de ces deux positions est utilisée pour évaluer $\mathcal{E}_{\vec{d}}(p_0)$. La sommation de la différence des valeurs absolues sur le voisinage des pixels constitue la première de nos boucles imbriquées, c'est la boucle d'intégration :

$$\mathcal{E}_{\vec{d}}(p_0) = \sum_{|\vec{\delta}|\leqslant\beta} |I_1(p_0 + \vec{\delta}) - I_2(p_0 + \vec{d} + \vec{\delta})| \tag{5.5}$$

La mesure de distance entre pixels utilisée ici est la distance de *Manhattan*. Par conséquent la valeur absolue des différences est intégrée sur une région carrée dont le rayon β n'est autre que le demi-côté. β est un paramètre ajustable de l'algorithme *PixelMatch*.

Vecteur de mouvement Pour chaque position p_0 dans I_1, une *corrélation minimale* est calculée parmi un ensemble limité de vecteurs de déplacement. Ce minimum désigne la position jumelle de p_0 dans I_2 et le vecteur de déplacement associé. Le calcul du minimum constitue la deuxième de nos boucles imbriquées, la boucle de minimisation par recherche exhaustive dans l'espace des vecteurs de mouvement :

$$\mathcal{E}_{\vec{d}_{\text{opt}}}(p_0) = \min_{|\vec{d}|\leqslant\alpha} \mathcal{E}_{\vec{d}}(p_0) \tag{5.6a}$$

$$\vec{d}_{\text{opt}}(p_0) = \arg\min_{|\vec{d}|\leqslant\alpha} \mathcal{E}_{\vec{d}}(p_0) \tag{5.6b}$$

Plus précisément, la recherche de minimum est effectuée sur une région carrée dont le demi-côté α est le deuxième paramètre de l'algorithme *PixelMatch*. Ce paramètre est choisi de telle sorte que $\alpha \leqslant \beta$ afin d'éviter les correspondances inappropriées – l'inégalité permet d'assurer que le calcul de la corrélation $\mathcal{E}_{\vec{d}}(p_0)$ inclus le pixel p_0 pour l'ensemble des valeurs de $\vec{d}$.

Images Le vecteur de mouvement et la *corrélation minimale associée* sont calculés pour *chaque* position de I_1. Itérer sur l'ensemble des positions dans l'image constitue notre dernière boucle : pour des images PAL cette boucle a une fréquence de 12 MHz.

Comparaison à MPEG-2 Dans le *block-matching* tel qu'utilisé dans la compression vidéo MPEG [42] un vecteur de mouvement est calculé pour un bloc de pixels de taille 16×16. Par ailleurs, des approches multi-échelles sont en général utilisées pour réduire l'espace de recherche à explorer, au détriment de la qualité du vecteur de mouvement. *PixelMatch*, qui calcule un vecteur de mouvement par pixel par recherche exhaustive est donc *a priori* un problème beaucoup plus complexe.

Première description de l'algorithme En définitive, nous aboutissons à une première implémentation naïve correspondant à la spécification :

Entrées : images I_1 et I_2 de taille $W \times H$
Données : table des corrélations $\mathcal{E}[\,][\,]$
Sorties : table des corrélations optimales $\mathcal{E}_{\text{opt}}[\,]$
Sorties : table des vecteurs de mouvements $\vec{d}_{\text{opt}}\,[\,]$

```
InitOptimum ;              /* Initialisation de E_opt[] et d_opt[] */
InitData ;                          /* Initialisation de E[][] */
```

```
1 forall |d| ⩽ α do
2     foreach p ∈ I_1 do
3         forall |δ| ⩽ β do
              E[d][p] += E_d(p + δ);
          UpdateOptimum(p, d, E[d][p]);
```

Algorithme 1 : Implémentation naïve de *PixelMatch*

Dans cette description, la boucle (1) est la boucle de recherche exhaustive qui itère sur l'ensemble des vecteurs de mouvement ; la boucle (2) itère en ordre raster-scan sur l'ensemble des pixels de l'image et appelle la boucle (3) qui calcule la SAD en itérant sur l'ensemble des pixels de la zone d'intégration. L'appel à la routine `UpdateOptimum` met à jour, le cas échéant, le tableau de la corrélation optimale $\mathcal{E}_{\text{opt}}[\,]$ et du vecteur de déplacement $\vec{d}_{\text{opt}}\,[\,]$ à l'index du pixel courant p :

Entrées : p, $\vec{d}$, $\mathcal{E}[\vec{d}][p]$
Données : table des corrélations optimales courantes $\mathcal{E}_{\text{opt}}[\,]$
Données : table des vecteurs de mouvements courants $\vec{d}_{\text{opt}}\,[\,]$

```
if E[d][p] ⩽ E_opt[p] then
    d_opt[p] ← d;
    E_opt[p] ← E[d][p];
```

Algorithme 2 : routine `UpdateOptimum`

Définitions auxiliaires L'image translatée de I_2 par le vecteur de mouvement $\vec{d}$ est définie par :

$$I_2 \circ T_{\vec{d}}\,(p) = I_2\,(p + \vec{d}\,) \tag{5.7}$$

L'énergie $E_{\vec{d}}$ évaluée à la position p, pour le vecteur de déplacement $\vec{d}$ est définie comme la valeur absolue de la différence entre les deux valeurs des pixels des I_1 et $I_2 \circ T_{\vec{d}}$ à la position p :

$$E_{\vec{d}}\,(p) = |I_1\,(p) - I_2 \circ T_{\vec{d}}\,(p)| \tag{5.8}$$

Par ailleurs il nous faudra décomposer le calcul de l'intégrale sur la surface carrée de (5.5) en bandes verticales. À cette fin la corrélation verticale $\mathcal{E}^v_{\vec{d}}$ évaluée à la position p et pour le vecteur de déplacement $\vec{d}$ est définie comme :

$$\mathcal{E}^v_{\vec{d}}\,(p) = \sum_{\delta \in \{0\} \times [-\beta, \beta]} E_{\vec{d}}\,(p + \delta\,) \tag{5.9}$$

5.3.1.2 Calcul efficace de la SAD

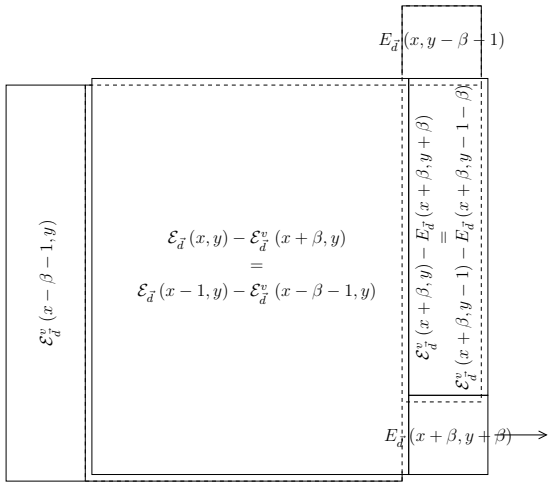


FIG. 5.3 – Partage des sous-expressions communes dans le calcul de corrélation

Supposons le vecteur de déplacement fixé à $\vec{d}$. Beaucoup de termes peuvent alors être partagés entre les différentes incarnations de la formule (5.5) pour les valeurs successives, en ordre raster-scan, de la position p dans I_1.

Regardons par exemple la figure 5.3 ; la SAD sur la région qui entoure la position (x, y) peut être calculée à partir de la somme autour de la position $(x-1, y)$ par la soustraction de la somme sur la bande verticale qui quitte la surface d'intégration et l'ajout de la somme sur la bande verticale entrante. La formule 5.10 précise ce calcul incrémental :

$$\begin{aligned}\mathcal{E}_{\vec{d}}(x,y) = \mathcal{E}_{\vec{d}}(x-1,y) - \mathcal{E}^v_{\vec{d}}(x-\beta-1,y) \\ + \mathcal{E}^v_{\vec{d}}(x+\beta,y)\end{aligned} \tag{5.10}$$

L'intégrale sur la bande verticale $\mathcal{E}^v_{\vec{d}}(x+\beta, y)$ elle-même peut être calculée avec un minimum d'opérations à partir de la bande verticale qui la surplombe. Il suffit d'y ajouter l'énergie à la position entrante dans la bande, et d'en soustraire l'énergie de la position qui quitte la bande verticale, pour obtenir la formule :

$$\begin{aligned}\mathcal{E}^v_{\vec{d}}(x+\beta,y) = \mathcal{E}^v_{\vec{d}}(x+\beta, y-1) \\ - E_{\vec{d}}(x+\beta, y-\beta-1) + E_{\vec{d}}(x+\beta, y+\beta)\end{aligned} \tag{5.11}$$

Ces équations permettent d'éviter les calculs redondants : les sommes partielles $\mathcal{E}_{\vec{d}}$, $\mathcal{E}^v_{\vec{d}}$, $E_{\vec{d}}$ sont mises en mémoire afin d'être réutilisées pour le calcul de la corrélation dans les positions suivantes. Il s'agit d'un compromis logique / mémoire qui permet d'éliminer la plus grosse partie du calcul de la formule (5.5).

La boucle la plus profonde de l'algorithme 1 est alors remplacée par un appel à une fonction de *calcul incrémental* de la corrélation. La fonction `IncrCorrelation` s'appuie sur une mémoire statique interne qui permet de stocker les résultats

intermédiaires du calcul d'un appel de fonction sur l'autre, afin de calculer la fonction de corrélation de manière incrémentale d'une position à la suivante.

Grâce à ce premier compromis, la boucle (3) de l'algorithme naïf 1 est ainsi remplacée :

Entrées : images I_1 et I_2 de taille $W \times H$
Données : table des corrélations $\mathcal{E}[\,][\,]$
Sorties : table des corrélations optimales $\mathcal{E}_{\text{opt}}[\,]$
Sorties : table des vecteurs de mouvements $\vec{d}_{\text{opt}}\,[\,]$

```
InitOptimum ;              /* Initialisation de E_opt[] et d_opt[] */
InitData ;                          /* Initialisation de E[][] */
```

1 **forall** $|\vec{d}| \leqslant \alpha$ **do**
2 **foreach** $p \in I_1$ **do**
 $\mathcal{E}[\vec{d}][p\,] \leftarrow$ `IncrCorrelation` $_{\vec{d}}\,(I_1, I_2)$;
 `UpdateOptimum`$(p\,, \vec{d}, \mathcal{E}[\vec{d}][p\,])$;

Algorithme 3 : Implémentation *PixelMatch* avec calcul incrémental de la corrélation

Sans présumer de l'implémentation effective de la nouvelle routine de calcul `IncrCorrelation` sur laquelle la section 5.3.2 se concentrera, les boucles de l'algorithme 3 sont encore sujettes à optimisation.

5.3.1.3 Optimisation des boucles

L'algorithme 3 présente le calcul de manière très inefficace. En effet, chacune des valeurs de la corrélation calculée est stockée dans le tableau à double indexation – sur les pixels et les vecteurs de déplacement – $\mathcal{E}[\,][\,]$. Cette présentation a pourtant un avantage : elle permet de réordonner à loisir les deux boucles 2 et 1 qui sont *commutatives*.

Réduction de la mémoire Le premier travail à effectuer est de réduire la mémoire nécessaire sur la présentation 3. Cette réduction a lieu à deux endroits :

- La table des corrélations $\mathcal{E}[\,][\,]$ peut être réduite à une variable locale
- La fonction de corrélation incrémentale `IncrCorrelation` n'a besoin que d'un seul contexte de mémoire interne, au lieu des $(2\alpha + 1)^2$ précédents, puisqu'elle n'est jamais appelée simultanément pour les valeurs différentes de $\vec{d}$.

La version suivante est la résultante de ces simplifications :

Entrées : images I_1 et I_2 de taille $W \times H$
Sorties : table des corrélations optimales $\mathcal{E}_{\text{opt}}[\,]$
Sorties : table des vecteurs de mouvements $\vec{d}_{\text{opt}}\,[\,]$

```
InitOptimum ;              /* Initialisation de E_opt[] et d_opt[] */
```

1 **forall** $|\vec{d}| \leqslant \alpha$ **do**
2 **foreach** $p \in I_1$ **do**
 $\mathcal{E}_{\vec{d}} \leftarrow$ `IncrCorrelation` (I_1, I_2);
 `UpdateOptimum`$(p\,, \vec{d}, \mathcal{E}_{\vec{d}}\,)$;

Algorithme 4 : Implémentation optimisée de *PixelMatch*

Ordre des boucles L'ordre des boucles présenté dans l'algorithme 4 nécessite une passe sur les deux images pour chaque valeur de $\vec{d}$. Au cours de chacune de ces passes, la routine `UpdateOptimum` met à jour les tableaux des optimums courants $\mathcal{E}_{\text{opt}}[\,]$ et $\vec{d}_{\text{opt}}\,[\,]$. Il est donc nécessaire d'utiliser une mémoire dédiée pour ces tableaux. La lecture, et la mise à jour éventuelle de ces tableaux pour chaque nouvelle valeur de la corrélation consomme une bande passante importante.

La localisation spatiale des données serait bien mieux exploitée en calculant l'ensemble $\mathcal{E}_p$ des valeurs de la corrélation à la position p pour tous les vecteurs de mouvement explorés. Alors toutes les données nécessaire au calcul du minimum sont présentes dans la même passe sur les images, sans qu'il soit besoin de les mémoriser d'une passe sur l'autre. La commutativité permet d'inverser directement l'ordre des boucles de l'algorithme 3, pour amener le calcul de minimum de la formule 5.6a cœur des boucles. La nouvelle expression de cet algorithme optimisé en consommation mémoire comme en bande passante devient, après réduction des tableaux en variables locales :

Entrées : images I_1 et I_2 de taille $W \times H$
Sorties : tableau des corrélations et vecteurs de mouvements

```
1 foreach p ∈ I1 do
      E_{d_opt} ← maxint;
2     forall |d| ⩽ α do
          E_d ← IncrCorrelation_d (I1, I2);
          if E_d ⩽ E_{d_opt} then
              d_opt ← d;
              E_{d_opt} ← E_d ;
      Écriture(p, E_{d_opt}, d_opt ) ;
```

Algorithme 5 : *PixelMatch* avec bande passante optimisée

Dans cette nouvelle variation, les quantités $\mathcal{E}_{\vec{d}_{\text{opt}}}$ et $\vec{d}_{\text{opt}}$ sont réduites à des variables locales sur la pile. En revanche le calcul simultané des corrélations incrémentales pour les $(2\alpha+1)^2$ vecteurs de déplacement nécessite de dupliquer autant de fois la mémoire statique interne à la fonction $\texttt{IncrCorrelation}_{\vec{d}}$.

Il reste donc à implémenter correctement la boîte noire qu'est pour l'instant `IncrCorrelation`.

5.3.2 Implémentations du calcul de corrélation incrémental

Au cœur de l'algorithme 5 se trouve la fonction `IncrCorrelation` de calcul incrémental de la corrélation. Nous explorons maintenant les différents compromis entre mémoire et logique (voire entre mémoire et nombre d'instructions, dans le cas du logiciel) possibles pour l'implémentation de cette fonction.

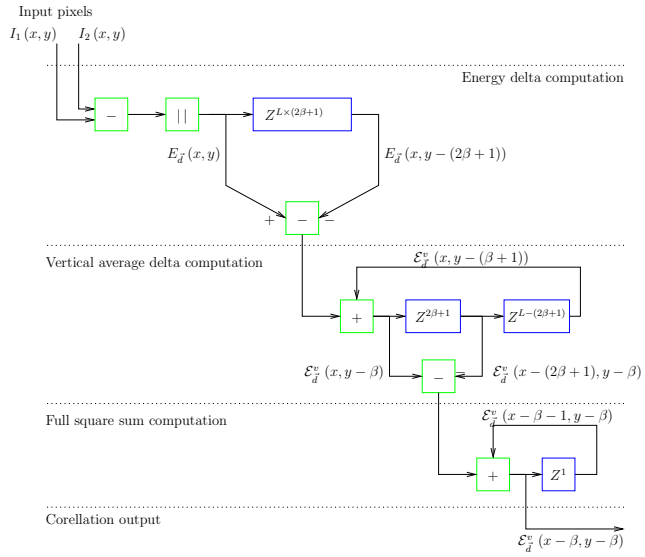


FIG. 5.4 – Calcul de la corrélation à logique minimale

5.3.2.1 Logique minimale

Les deux formules (5.10) et (5.11) vues en 5.3.1.2 permettent de réduire le calcul au minimum de sa complexité :

$$\begin{aligned}\mathcal{E}_{\vec{d}}(x-\beta,y-\beta) &= \mathcal{E}_{\vec{d}}(x-\beta-1,y-\beta)\\ &\quad -\mathcal{E}^v_{\vec{d}}(x-2\beta-1,y-\beta)+\mathcal{E}^v_{\vec{d}}(x,y-\beta-1)\\ &\quad -E_{\vec{d}}(x,y-2\beta-1)\\ &\quad +|I_1(x,y)-I_2\circ T_{\vec{d}}(x,y)|\end{aligned} \tag{5.12}$$

Il suffit alors d'un additionneur, de deux soustracteurs et d'une valeur absolue pour calculer $\mathcal{E}_{\vec{d}}$. Le coût d'une telle réduction de complexité se trouve dans les registres à décalage nécessaires pour le stockage des valeurs $\mathcal{E}_{\vec{d}}$, $\mathcal{E}^v_{\vec{d}}$ et $E_{\vec{d}}$ sur une fenêtre glissante de largeur adéquate.

Le circuit de la figure 5.4 incarne ce calcul. Sur son entrée, les pixels des deux images défilent en ordre raster-scan, et il calcule dans le même ordre en sortie les valeurs de $\mathcal{E}_{\vec{d}}$. Les besoins en mémoire de cette version du circuit – qui dépendent de la largeur des fenêtres de mémorisation – sont explicités en table 5.8. Le registre à décalage contenant $E_{\vec{d}}$ est très profond, et c'est sans doute là le principal inconvénient de ce circuit.

5.3.2.2 Dupliquer le calcul de l'énergie

Pour remédier à cet inconvénient, il suffit de ne pas mémoriser le terme $E_{\vec{d}}$, mais plutôt de le recalculer au besoin avec des données en provenance directe des images I_1 et I_2. La formule suivante incarne ce nouveau calcul :

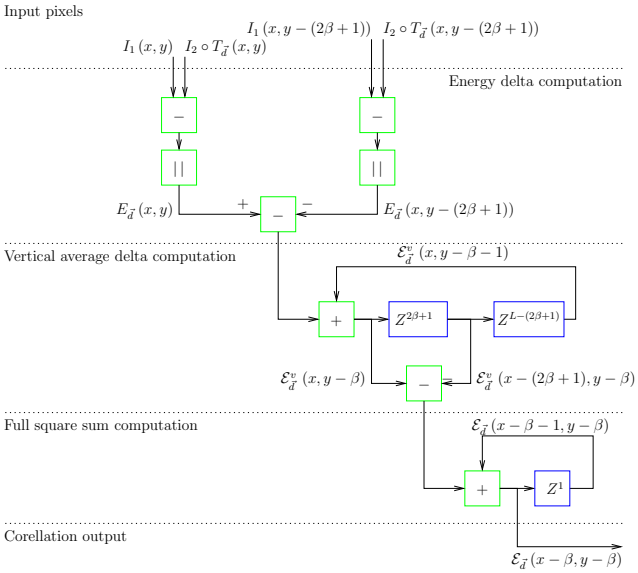


FIG. 5.5 – Calcul de la corrélation à l'équilibre logique/mémoire

$$\begin{aligned}\mathcal{E}_{\vec{d}}(x-\beta,y-\beta) &= \mathcal{E}_{\vec{d}}(x-\beta-1,y-\beta)\\ &- \mathcal{E}^v_{\vec{d}}(x-2\beta-1,y-\beta) + \mathcal{E}^v_{\vec{d}}(x,y-\beta-1)\\ &- |I_1(x,y-2\beta-1) - I_2 \circ T_{\vec{d}}(x,y-2\beta-1)|\\ &+ |I_1(x,y) - I_2 \circ T_{\vec{d}}(x,y)|\end{aligned} \tag{5.13}$$

La méthode permet d'échanger la mémoire du tampon des valeurs de $E_{\vec{d}}$ contre la logique qui permet de les calculer directement. La bande passante en provenance des images I_1 et I_2 est également accrue.

Le circuit de la figure 5.5 réalise ce calcul. Les besoins en mémoire de ce circuit sont bien moindres que précédemment, comme le montre la table 5.8 – le registre à décalage $E_{\vec{d}}$ a simplement disparu.

5.3.2.3 Dupliquer l'intégrale verticale

Le registre à décalage le plus important est maintenant celui qui mémorise les termes $\mathcal{E}^v_{\vec{d}}$. Il est à nouveau possible de s'en débarasser en recalculant le terme $\mathcal{E}^v_{\vec{d}}(x,y-\beta-1)$ de la formule (5.13). Ici encore, ce calcul est fait à partir de données en provenance directe de I_1 et de I_2, comme le montre la formule suivante :

$$\begin{aligned}\mathcal{E}_{\vec{d}}(x-\beta,y-\beta) &= \mathcal{E}_{\vec{d}}(x-\beta-1,y-\beta)\\ &- \mathcal{E}^v_{\vec{d}}(x-2\beta-1,y-\beta)\\ &+ \sum_{0\leqslant \mathrm{d_y} \leqslant 2\beta} |I_1(x,y-\mathrm{d_y}) - I_2 \circ T_{\vec{d}}(x,y-\mathrm{d_y})|\end{aligned} \tag{5.14}$$

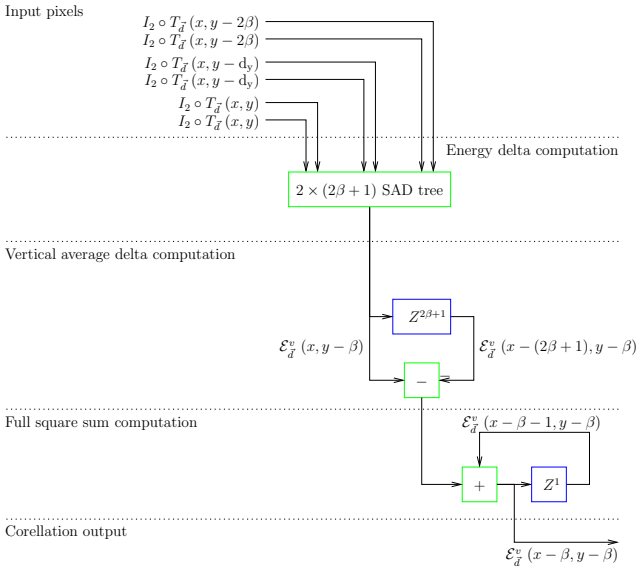


FIG. 5.6 – Calcul de la corrélation à mémoire minimale

	Logique minimale	Intermédiaire	Mémoire minimale
$\mathcal{E}_{\vec{d}}$ SR	$(n + 2\log_2(2\beta+1)) \times 1$		
$\mathcal{E}^v_{\vec{d}}$ SR	$(n + \log_2(2\beta+1)) \times W$		$(n + \log_2(2\beta+1)) \times (2\beta+1)$
$E_{\vec{d}}$ SR	$n \times W.(2\beta+1)$	0	
Mémoire	76,047 b	8,463 b	1,346 b
Logique	66 LUTs	83 LUTs	148 LUTs

TAB. 5.8 – Compromis logique/mémoire pour `IncrCorrelation`

Cette nouvelle mouture du circuit élimine la majeure partie de la fenêtre glissante contenant $\mathcal{E}^v_{\vec{d}}$ – la mémoire interne à l'opération `IncrCorrelation` est même rendue indépendante de W, la largeur de l'image.

Ce gain en mémoire s'accompagne d'une augmentation considérable dans la consommation de logique, de bande passante, et dans la complexité de câblage du module. En effet, le circuit contient désormais un arbre de SAD de $(2\alpha+1)$ éléments en entrée, et doit récupérer $(2\beta+1)$ valeurs de pixels en provenance de chacune des images I_1 et I_2 (figure 5.6).

5.3.3 Implémentations

Sur les bases établies dans les paragraphes précédents, et à l'aide des expressions de haut niveau des algorithmes, les différentes implémentations sont maintenant conduites. Les paramètres d'implémentation de l'algorithme sont d'abord précisés.

Puis les résultats sont présentés dans la perspective d'utilisation de la Pam comme accélérateur matériel dans le cadre d'un co-développement matériel/logiciel.

Le portage de l'algorithme sur FPGA consiste essentiellement à effectuer un partitionnement optimal de l'application entre la partie de l'algorithme 5 exécutée sur le processeur de la station de travail et la partie exécutée sur la Pam.

À partir de l'algorithme 5 une première implémentation logicielle de référence est réalisée ; elle est ensuite optimisée. Puisque le temps réel n'est pas atteint, la fonction de corrélation incrémentale, au cœur de l'algorithme 5 est portée sur FPGA dans la section 5.3.3.3. Cette première manière d'utiliser le coprocesseur SEPIA ne permet toutefois pas à la Pam d'augmenter suffisamment les performances de l'algorithme. Il faut alors dérouler partiellement la boucle interne de l'algorithme dans le FPGA, ce qui conduit à l'implémentation de la section 5.3.3.4. C'est alors la bande passante entre SEPIA et la mémoire de la station de travail qui limite la vitesse de traitement. Ce problème est résolu avec l'implémentation de la section 5.3.3.5 qui permet de dérouler la totalité de la boucle interne dans le FPGA.

5.3.3.1 Paramètres de l'algorithme

L'objectif de la réalisation FPGA de l'algorithme de *PixelMatch* est de parvenir au traitement en temps réel de flux vidéos en Standard Definition (SD). Le terme de SD recouvre à la fois les flux au standard National Television System Committee (NTSC), aux États-Unis, et les flux à la norme PAL, partout ailleurs.

La bande passante des deux standards est à peu près identique, et légèrement plus élevée pour la sous-norme de PAL de qualité la plus élevée ; nous choisissons donc d'utiliser les chiffres du standard PAL 4/3. Les images sont au format 720×576 à une fréquence de rafraîchissement de 25 images / seconde[3], soit une bande passante brute de $10,368\,e6$ pixels par seconde.

Les paramètres algorithmiques β et α sont choisis égaux à 5. Ainsi donc le carré d'intégration de la corrélation est d'une surface de 121 pixels ; et la recherche de la corrélation optimale se fait entre 121 vecteurs de mouvement différents. À cette résolution, le flux d'entrée est de $10,368.10^6$ pixels par seconde, et le flux de sortie est constitué de $10,368.10^6$ vecteurs de mouvements par seconde, qui sont sélectionnés sur une base de $1,254\,528.10^9$ corrélations calculées.

5.3.3.2 Implémentation logicielle

Choix de l'implémentation La description du code de haut niveau de l'algorithme 5 est directement traduit en C une fois l'implémentation logicielle correcte de `IncrCorrelation` choisie.

Les microprocesseurs modernes sont des machines séquentielles une fois exploité le parallélisme au niveau instruction des architectures superscalaires modernes. Dans une *machine séquentielle* l'ensemble de l'algorithme est naturellement déroulé dans le temps, par opposition à la *machine parallèle* où l'expression naturelle est le déroulement dans l'espace.

Par conséquent, on pourrait croire que le seul paramètre à optimiser lors d'une réalisation logicielle est le nombre d'instructions exécutées. Toutefois, la hiérarchie des niveaux de cache dans un processeur peut avoir un effet énorme

[3]en mode progressif. En mode entrelacé, deux demi-trames sont émises à une fréquence double de 50 Hz, pour une bande passante qui reste identique

Version	Fils	Images	Durée du traitement		Vitesse	
			totale	par image	Im.s^{-1}	MiB.s^{-1}
référence	1	300	1427.166s	4757.222ms	.2	.1
optimisée	1	300	145.870s	486.236ms	2.0	1.7
	2	300	71.693s	238.979ms	4.1	3.4
	4	300	36.159s	120.532ms	8.2	6.8
	6	300	36.545s	121.817ms	8.2	6.8

TAB. 5.9 – Performances de l'implémentation logicielle de l'algorithme

Circuit	Vitesse		Ressources		Bande passante
	ns	MHz	RAMB	Tranches	utile MiB.s^{-1}
original	16.02	62	5 (8.9%)	132 (1.2%)	124
retemporisé	4.88	204	5 (8.9%)	405 (3.7%)	409

TAB. 5.10 – Implémentation d'un noyau de calcul `IncrCorrelation` sur FPGA

sur la durée d'exécution des instructions, à tel point qu'il peut être plus efficace de dupliquer certains calculs si la conséquence en est une meilleure utilisation de la mémoire cache. C'est pourquoi un arbitrage doit être fait entre, d'une part, le *nombre d'instructions* et, de l'autre, l'*utilisation mémoire* et la *localisation des données* utilisées dans l'algorithme.

La discussion de la section 5.3.2 nous a montré l'ensemble des compromis entre le nombre d'instructions et la mémoire disponible pour la fonction `IncrCorrelation`.

C'est l'architecture de la partie 5.3.2.2 qui est le meilleur candidat pour une implémentation logicielle. Les données mémoïsées de l'architecture 5.3.2.1, plus gourmande en mémoire, ne tiennent pas dans le cache ; à l'inverse il y a trop de calculs dans l'architecture de 5.3.2.3.

Performances Le code est directement dérivé de la description de haut niveau, compilé avec GCC 4.0 et exécuté sur un cœur Opteron 275 à 2.2GHz. Le profiling à l'aide de cachegrind [43] montre un cache-miss de moins de 1% dans le cœur de la boucle.

Les processeurs modernes, multi-coeurs et SMP, peuvent parvenir à un déroulement dans l'espace de l'exécution grâce au parallélisme au niveau processus. Une implémentation permet de distribuer en parallèle l'exécution du code sur les quatre cœurs de la station de travail pour obtenir une vitesse d'exécution effective au quart du temps réel pour les flux au format PAL.

Version	Images	Durée du traitement		Vitesse	
		totale	par image	Im.s^{-1}	MiB.s^{-1}
Pam	300	299.510s	998.367ms	1.0	.8
Pam retemporisé	300	300.444s	1001.482ms	.9	.8

TAB. 5.11 – Performances de *PixelMatch*, un noyau de calcul

5.3.3.3 Implémentation FPGA de `IncrCorrelation`

Si l'implémentation logicielle optimisée apporte un gain de performances significatif par rapport à l'implémentation de référence, elle reste très en-dessous d'une vitesse de traitement en temps réel – à moins d'utiliser une machine à 16 cœurs, particulièrement chère et consommatrice en électricité.

Nous allons donc utiliser la Pam comme coprocesseur d'accélération pour l'algorithme de *PixelMatch* afin de parvenir au traitement en temps réel des flux vidéo PAL.

L'utilisation d'une Pam comme coprocesseur d'accélération pose la question du partitionnement logiciel / matériel de l'algorithme.

La partie de l'algorithme la plus intensive en calculs est la routine de calcul incrémental de la corrélation, `IncrCorrelation`. C'est donc tout naturellement que la première version combinée logiciel/matériel de l'algorithme consiste à déporter cette routine sur la Pam. La description suivante est utilisée comme base du premier essai de portage de l'algorithme sur SEPIA :

Entrées : images I_1 et I_2 de taille $W \times H$
Sorties : tableau des corrélations et vecteurs de mouvements

`InitOptimum` ; `/* Initialisation de` $\mathcal{E}_{\text{opt}}[\,]$ `et` $\vec{d}_{\text{opt}}[\,]$ `*/`
1 **forall** $|\vec{d}| \leqslant \alpha$ **do**
2 **foreach** $p \in I_1$ **do**
$\mathcal{E}_{\vec{d}} \leftarrow$ `IncrCorrelation` (I_1, I_2);
$\mathcal{E}[\vec{d}][p] \leftarrow \mathcal{E}_{\vec{d}}$;
3 **foreach** $p \in I_1$ **do**
`UpdateOptimum`$(p, \vec{d}, \mathcal{E}[\vec{d}][p])$;

Algorithme 6 : Première séparation logicielle/matérielle de *PixelMatch*

Dans le partitionnement de l'algorithme 6, la boucle 2 réalise le calcul de la corrélation sur l'ensemble de l'image pour un vecteur de déplacement $\vec{d}$ donné. Son cœur est implémenté comme un circuit sur la Pam, et son exécution correspond au passage des images I_1 et I_2 au travers du circuit. La mise à jour des tableaux des optimums courants $\mathcal{E}_{\text{opt}}[\,]$ et $\vec{d}_{\text{opt}}[\,]$ est réalisée en logiciel dans la boucle 3. L'ensemble du calcul de la corrélation pour un déplacement et de la mise à jour du tableau d'optimum doit être répété pour chacun des $(2\alpha+1)^2$ vecteurs de déplacement, c'est-à-dire 121 fois pour chaque paire d'images. La fréquence théorique minimale du circuit nécessaire pour atteindre le temps réel est donc de :

$$W \times H \times f \times (2\alpha+1)^2 \approx 1,2\,GHz$$

Le choix de l'implémentation de `IncrCorrelation` pour cette version de l'algorithme s'est porté sur la version à logique minimale et mémoire maximale décrite en 5.3.2.1, ce qui explique la consommation importante de blocs RAM pour ce circuit – ces RAMs contiennent les plus longs des registres à décalage. Au contraire, la consommation logique du circuit synthétisé est très faible, comme le montre la table 5.10. La version du circuit retimée automatiquement par DSL permet presque d'atteindre la vitesse maximale du Virtex II. Toutefois, cette vitesse maximale est loin d'atteindre la vitesse théorique nécessaire pour le traitement en temps réel de flux vidéo PAL : il faut faire mieux.

Circuit	Vitesse		Ressources		Bande passante utile MiB.s^{-1}
	ns	MHz	RAMB	Tranches	
original	79.34	12	30 (53.5%)	2420 (22.5%)	50
retemporisé	5.14	194	30 (53.5%)	5097 (47.4%)	778

TAB. 5.12 – Implémentation de $2\alpha + 1$ noyaux de calcul sur FPGA

Version	Images	Durée du traitement		Vitesse	
		totale	par image	Im.s^{-1}	MiB.s^{-1}
Pam	300	150.970s	503.236ms	1.9	1.6
Pam retemporisé	300	61.899s	206.331ms	4.8	4.0

TAB. 5.13 – Performances, $2\alpha + 1$ noyaux de calcul

5.3.3.4 Déroulement dans l'espace de `IncrCorrelation`

Les architectures matérielles permettent de multiplier la puissance de calcul d'un circuit en le duplicant autant de fois. *PixelMatch* est un très bon exemple pour mettre en œuvre cette méthode de parallélisation par déroulement spatial : plusieurs instances de $\texttt{IncrCorrelation}_{\vec{d}}$ peuvent calculer simultanément les valeurs de la corrélation pour différentes valeurs du vecteur de mouvement $\vec{d}$.

La manière la plus naturelle d'effectuer ce déroulement est de calculer successivement dans le temps les corrélations pour l'ensemble des vecteurs de déplacement à $\mathrm{d_{y_0}}$ fixé :

$$\{\vec{d} = (\mathrm{d_x}, \mathrm{d_{y_0}})/\mathrm{d_x} \in [-\beta, \beta]\}$$

La formulation suivante de l'algorithme permet ainsi de dérouler $2\alpha + 1$ fois

Version	Images	Durée du traitement		Vitesse	
		totale	par image	Im.s^{-1}	MiB.s^{-1}
Pam	300	147.821s	492.738ms	2.0	1.6
Pam retemporisé	300	58.763s	195.877ms	5.1	4.2

TAB. 5.14 – Performances, optimisation de $2\alpha + 1$ noyaux de calcul

le module `IncrCorrelation` dans l'espace :

Entrées : images I_1 et I_2 de taille $W \times H$
Sorties : tableau des corrélations et vecteurs de mouvements

```
   InitOptimum ;                    /* Initialisation de ℰopt[] et d⃗opt[] */
1  forall |d⃗y| ⩽ α do
2      foreach p ∈ I1 do
3          forall |d⃗x| ⩽ α do
               ℰd⃗ ← IncrCorrelation d⃗x (I1, I2);
               if ℰd⃗ ⩽ ℰd⃗opt then
                   d⃗opt ← d⃗;
                   ℰd⃗opt ← ℰd⃗ ;
               ℰ[dy][p] ← ℰd⃗opt ;
4      foreach p ∈ I1 do
           UpdateOptimum(p, dx, ℰ[dx][p]);
```

Algorithme 7 : Deuxième séparation logicielle/matérielle de *PixelMatch*

Le choix de l'implémentation de `IncrCorrelation` pour cette version de l'algorithme s'est porté sur la version à mémoire minimale. Ceci explique que la consommation de blocs RAM, présentée en 5.12 reste très inférieure à celle d'une hypothétique duplication en $2\alpha + 1$ exemplaires du circuit présentée en 5.10 – et ceci malgré la présence en tête du circuit de deux registres à décalage pour stocker les valeurs de I_2 et I_1. Quant à la consommation de logique elle est à l'inverse bien plus que multipliée par $2\alpha+1 = 11$, en raison des arbres de calculs de la SAD.

La retemporisation du circuit permet ici encore d'atteindre des vitesses de pointe sur le circuit, au point de pouvoir théoriquement atteindre le traitement en temps réel des flux PAL.

Dans le partitionnement de l'algorithme 7, la boucle 2 réalise le calcul de la corrélation sur l'ensemble de l'image pour $2\alpha + 1$ vecteurs de déplacement. Son cœur est implémenté comme un circuit sur la Pam, et son exécution correspond au passage des images I_1 et I_2 au travers du circuit. La mise à jour des tableaux des optimums courants $\mathcal{E}_{\text{opt}}[\,]$ et $\vec{d}_{\text{opt}}[\,]$ est réalisée en logiciel dans la boucle 4. L'ensemble du calcul de la corrélation pour un déplacement et de la mise à jour du tableau d'optimum doit être répété pour chacun des $2\alpha + 1$ vecteurs de déplacement, c'est-à-dire 11 fois pour chaque paire d'images. La fréquence théorique minimale du circuit nécessaire pour atteindre le temps réel est donc de :

$$W \times H \times f \times (2\alpha + 1) \approx 115\text{MHz}$$

qui est largement en-dessous de la fréquence nominale obtenue par retemporisation du circuit. La largeur d'entrée de ce circuit est de 32 bits ; à cette fréquence et pour cette largeur de circuit, la bande passante du bus PCI est saturée, et le circuit se comporte comme un circuit de fréquence 50MHz. C'est le facteur limitant de la vitesse de notre application, qu'il nous faut maintenant améliorer.

Le tableau 5.13 présente les résultats d'une première implémentation avec entrées-sorties synchrones, et un seul fil d'exécution. À l'instar des circuits pré-

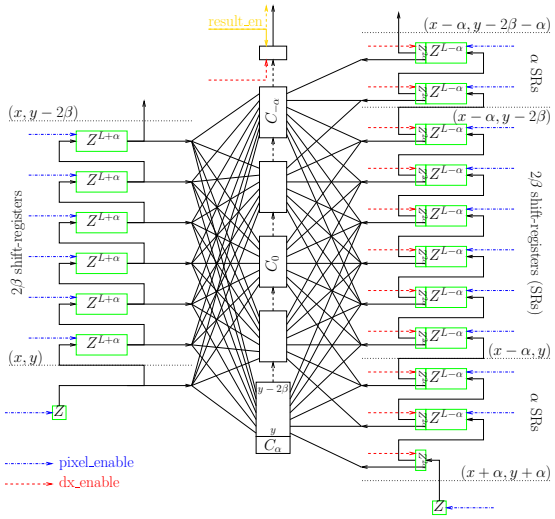


FIG. 5.7 – Schéma du circuit pour les paramètres $\alpha = 2, \beta = 3$

cédents, par exemple en 5.2.2.3, il existe une différence importante entre la consommation de bande passante théorique et la consommation effectivement mesurée.

Le tableau 5.14 permet de quantifier l'amélioration apportée par l'utilisation de multiples fils d'exécution concurrents. Cette technique permet très naturellement d'utiliser l'ordonnanceur du système d'exploitation pour atteindre les performances maximales attendues d'une implémentation avec entrées-sorties asynchrones. La mesure montre très clairement une amélioration des performances de l'ordre de 10%.

L'implémentation est délicate. Chaque fil d'exécution dispose d'un buffer de DMA alloué de manière statique dès son démarrage, qui est réutilisé pour chacun des appels au coprocesseur SEPIA. Cette méthode permet de limiter les appels à l'allocateur mémoire. Le descripteur DMA est lui aussi alloué au démarrage du fil, afin de limiter les appels systèmes nécessaires à sa construction (voir 4.2.3.3). Enfin, l'accès au coprocesseur est sérialisé au grâce à un mutex. Le chemin protégé par le mutex comporte simplement la soumission du descripteur et l'attente de l'interruption de fin de traitement.

5.3.3.5 Déroulement espace/temps du noyau de calcul

Nous pouvons pousser plus loin la logique de déroulement dans l'espace de l'application ; le circuit qui déroule totalement $(2\alpha + 1)^2$ fois dans l'espace le module `IncrCorrelation` reçoit en entrée à chaque cycle un pixel de chacune des images et produit une estimation de mouvement.

Toutefois, l'analyse de l'algorithme précédent montre que seule la diminution de bande passante associée à ce déroulement est utile : la fréquence de fonctionnement du circuit est déjà suffisante pour atteindre le traitement en temps réel.

Cette architecture qui duplique de la logique est par conséquent tout-à-fait

Circuit	Vitesse		Ressources		Bande passante
	ns	MHz	RAMB	Tranches	utile MiB.s^{-1}
original	79.71	12	30 (53.5%)	6488 (60.3%)	2
retemporisé	5.33	187	30 (53.5%)	7719 (71.7%)	34

TAB. 5.15 – Implémentation de $2\alpha+1$ noyaux de calcul sur FPGA et déroulement interne

Version	Images	Durée du traitement		Vitesse	
		totale	par image	Im.s^{-1}	MiB.s^{-1}
Pam	300	163.633s	545.446ms	1.8	1.5
Pam retemporisé	300	9.072s	30.242ms	33.0	27.4

TAB. 5.16 – Performances, $2\alpha + 1$ noyaux de calcul et déroulement interne

adaptée au *repliement de l'espace dans le temps*. Cette méthodologie est décrite théoriquement dans [44], et a déjà été utilisée à de nombreuses reprises dans l'architecture de circuits digitaux (par exemple dans [45]). Elle nous permet en l'occurrence d'échanger une réduction de la logique utilisée contre un allongement de la durée du calcul. La mémoire requise reste quant à elle identique.

La logique contenue dans les $(2\alpha+1)^2$ instances du module peut être repliée k fois de telle sorte que chaque instance de la logique calcule séquentiellement dans le temps la corrélation pour k valeurs différentes du déplacement. Dans notre cas, un bon équilibre pour le partage de logique est d'équilibrer le déploiement dans le temps et dans l'espace : la boucle de calcul de $(2\alpha + 1)^2$ corrélations à chaque position est déroulée $2\alpha+1$ fois dans l'espace et $2\alpha+1$ fois dans le temps. C'est aussi la configuration pour laquelle la logique de contrôle nécessaire pour le repliement est la plus simple :

Entrées : images I_1 et I_2 de taille $W \times H$
Sorties : tableau des corrélations et vecteurs de mouvements

```
1 foreach p ∈ I1 do
2     forall |d⃗x| ⩽ α do
3         forall |d⃗y| ⩽ α do
              E_d⃗opt ← maxint;
              d⃗ ← (dx, dy);
              Edx ← IncrCorrelation_d⃗x (E_d⃗ , I1, I2);
              if Edx ⩽ E_d⃗opt then
                  d⃗opt ← d⃗;
                  E_d⃗opt ← Edx ;
      E[d⃗][p] ← E_d⃗opt ;
```

Algorithme 8 : Version finale de *PixelMatch*

Dans cette version de l'architecture, une unité de calcul que l'on note $C_{\mathrm{d_{y_0}}}$ calcule à leur tour les valeurs $(\mathcal{E}_{(\mathrm{d_x},\mathrm{d_{y_0}})})_{\mathrm{d_x}\in[-\beta,\beta]}$ de la corrélation, puis le circuit se déplace vers le pixel suivant. Le circuit est constitué des unités de calcul $(C_{\mathrm{d_{y_0}}})_{\mathrm{d_{y_0}}\in[-\beta,\beta]}$, qui correspondent au dépliement dans l'espace de la boucle 3.

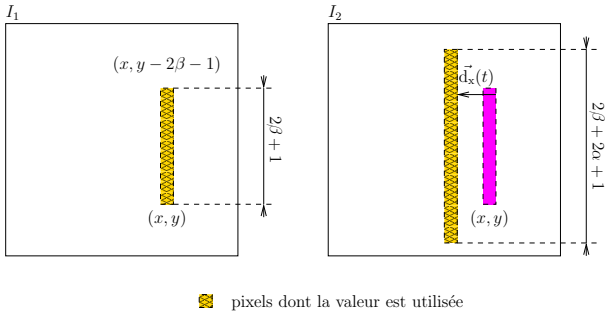


FIG. 5.8 – Accès aux données de I_1 et I_2 pour les unités de calcul $(C_{d_y})_{d_y \in [-\alpha,\alpha]}$

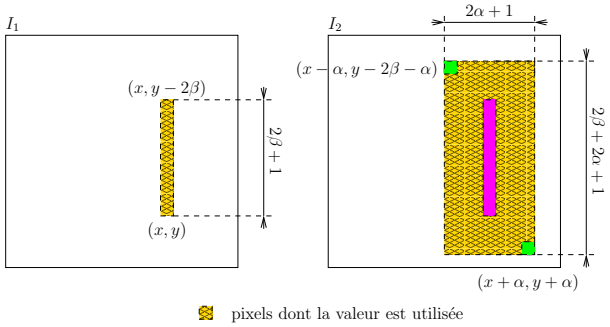


FIG. 5.9 – Accès aux données de I_1 et I_2 pour les $(2\alpha+1)^2$ corrélations

Enfin, la réutilisation de ces unités par multiplexage dans le temps des valeurs du déplacement horizontal correspond au déroulement dans le temps de la boucle 2. La figure 5.7 présente le schéma d'un circuit réalisé pour des paramètres α et β plus petits.

Il nous faut dupliquer $(2\alpha+1)^2$ fois la mémoire interne au bloc de calcul `IncrCorrelation` pour mener de front le calcul simultanné des $(2\alpha+1)^2$ vecteurs de déplacement. Or la mémoire, ressource rare sur un FPGA, impose une contrainte forte sur l'architecture du circuit, puisque c'est dans ce cas la ressource la moins abondante. Le circuit de corrélation de la section 5.3.2.3, dont l'utilisation mémoire est minimale, est le seul a pouvoir être ainsi dupliqué sur le xc2v2000.

Le partage de logique a un coût en termes de logique de contrôle, puisqu'alors il est nécessaire d'amener vers les unités de calcul à chaque coup d'horloge les valeurs appropriées en provenance des tampons qui mémorisent les données de I_1 et I_2.

Le FPGA contient ainsi des registres à décalage contenant les données de I_1 et I_2 sur une fenêtre glissante. Ces tampons sont utilisés comme une cache d'où sont lues les données nécessaires à l'alimentation des $(C_{d_{y_0}})_{d_{y_0} \in [-\beta,\beta]}$. L'augmentation de bande passante en provenance des deux images est ainsi limitée puisque beaucoup des accès aux données à partir des $2\alpha+1$ modules se recouvrent pour aboutir au motif d'accès figuré en 5.9.

Grâce à cette cache, les données en entrée du design sont les deux flux très simples des deux images, le buffer se chargeant de la complexité liée à la distribution des données appropriées vers chaque module.

Tous les $2\alpha + 1$ coups d'horloge, le circuit capture un nouveau pixel dans les tampons de I_1 et I_2. Puis au cours des $2\alpha+1$ coups d'horloge suivants, la fenêtre d'accès de la figure 5.8 glisse vers la droite suivant la valeur de d_x. L'ensemble des pixels de I_2 et I_1 auxquels accède le circuit au cours du calcul d'un vecteur de mouvement est présenté dans la figure 5.9. Cette figure nous donne le gabarit des registres à décalage nécessaires en tête du circuit pour mémoriser les valeurs des images.

Ce circuit atteint l'optimum examiné à la section 5.3.1.3, qui calcule le vecteur de mouvement final avec un minimum de mémoire et de bande passante. Par la réduction de la consommation en bande passante externe obtenue, cette implémentation de l'algorithme de calcul permet de traiter en temps réel des flux vidéo PAL, comme le montre le tableau de performances 5.16. Nous sommes parvenus à ce résultat avec un circuit final utilisant au plus juste les ressources FPGA – tant en logique, qu'en mémoire et bande passante.

5.4 Détection de points de Harris

5.4.1 Description algorithmique

Nous décrivons dans cette partie une variante de l'algorithme de recherche de points caractéristiques de Harris [46] (*Harris feature points*) dans une image.

Ces points sont des points particuliers extraits dans chaque image d'un flux vidéo auxquels est associé un *descripteur*. Un descripteur est une quantité scalaire ou vectorielle calculée sur un voisinage du point singulier dans l'image dont la caractéristique est de posséder une relative invariance aux mouvements usuels de caméra : translation, rotation, zoom. Cette propriété permet d'associer deux à deux les points de Harris d'une image sur l'autre grâce à une mesure de distance entre deux descripteurs : c'est la phase d'appariement (*matching*). Ainsi utilisés comme repères dans le flux vidéo, les points caractéristiques de Harris permettent de calculer, par exemple, le mouvement de la caméra, ou bien de suivre un objet dans une scène filmée (respectivement *camera tracking* et *object tracking*).

La section 5.4.1.2 décrit l'algorithme de recherche des points caractéristiques de Harris. L'algorithme est d'abord décrit comme une suite de filtres à appliquer à l'image. La section suivante décrit une méthodologie simple qui permet de convertir efficacement la suite de filtres en une implémentation matérielle à mémoire réduite. Le prototype logiciel réalisé à l'aide de cette approche est aussi très performant. Cette méthodologie peut être utilisée pour la grande variété des algorithmes de traitement d'images qui sont constitués d'une succession de filtres.

5.4.1.1 Description de l'algorithme

L'algorithme de base d'extraction des points de Harris consiste à isoler dans une image les points qui se distinguent de leur voisinage. À cette fin, un produit scalaire $M(x, y)$ est construit en chaque pixel (x, y). Cette matrice symétrique permet d'évaluer la dissimilarité d'un pixel avec son voisinage. Soit donc (x, y) un point de l'image, W un voisinage du point et x une fonction de convolution, la matrice symétrique $M(x, y)$ est donnée par :

$$M(x,y) = \iint_W \begin{bmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial I}{\partial x} & \frac{\partial I}{\partial y} \end{bmatrix} w(x,y)\, dx\, dy \tag{5.15}$$

Un point caractéristique est un point où cette fonction peut garantir une grande dissimilarité du pixel avec ses voisins dans toutes les directions. La garantie existe lorsque la matrice symétrique M a des valeurs propres importantes, ou plus simplement lorsque la fonction de réponse suivante est élevée :

$$\mathcal{H} = detM - k.(traceM)^2$$

Le paramètre k est fixé à 0.04, comme Harris le recommande [46]. Le scalaire ainsi obtenu en chaque point est le critère de Harris. Il est filtré par un maximum local pour obtenir les points distingués de Harris.

5.4.1.2 Paramètres et filtres

L'implémentation réelle de l'extracteur de Harris ajoute quelques filtres préliminaires au calcul décrit ci-dessus. La notation de produit matriciel $\otimes$ signifie

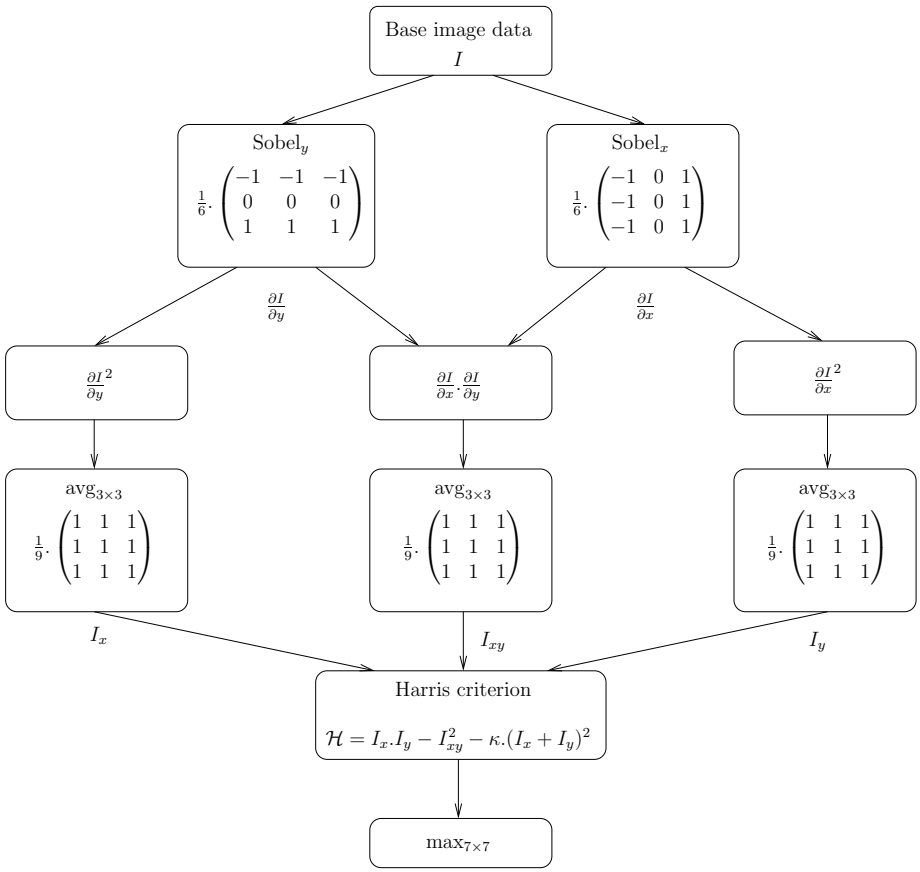


FIG. 5.10 – Chaîne des filtres de l'extraction des points de Harris

ici le produit de l'image avec un noyau de convolution – comme il est habituel en traitement d'image.

En premier lieu, les quantités $\frac{\partial I}{\partial y}$ et $\frac{\partial I}{\partial y}$ sont approximées par des fonctions de Sobel :

$$\begin{aligned}\frac{\partial I}{\partial x}(x,y) &\simeq \frac{1}{6}.\begin{pmatrix}-1&0&1\\-1&0&1\\-1&0&1\end{pmatrix} \otimes I(x,y)\\ \frac{\partial I}{\partial y}(x,y) &\simeq \frac{1}{6}.\begin{pmatrix}-1&-1&-1\\0&0&0\\1&1&1\end{pmatrix} \otimes I(x,y)\end{aligned}$$

Ces fonctions lissent la différentielle suivant sa direction orthogonale.

La fonction de convolution $w(x,y)$ de l'expression 5.15 est choisie simplement comme une indicatrice 3×3 :

$$M(x,y) = \frac{1}{9}.\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix} \otimes \begin{bmatrix}\frac{\partial I}{\partial x}^2 & \frac{\partial I}{\partial y}\frac{\partial I}{\partial x}\\ \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} & \frac{\partial I}{\partial y}^2\end{bmatrix}(x,y) = \begin{bmatrix}I_x^2 & I_yI_x\\ I_xI_y & I_y^2\end{bmatrix}(x,y)$$

Enfin, le critère de Harris $\mathcal{H}$ est filtré par un maximum local sur un carré de taille 7×7 : seuls les maxima locaux dans un carré de 7 pixels de côté centré autour du point sont considérés comme intéressants. Ce dernier filtre n'est pas un filtre de convolution classique.

L'algorithme est en définitive décomposé en la suite des filtres décrite dans la figure 5.10.

5.4.1.3 Optimisation des filtres de convolution

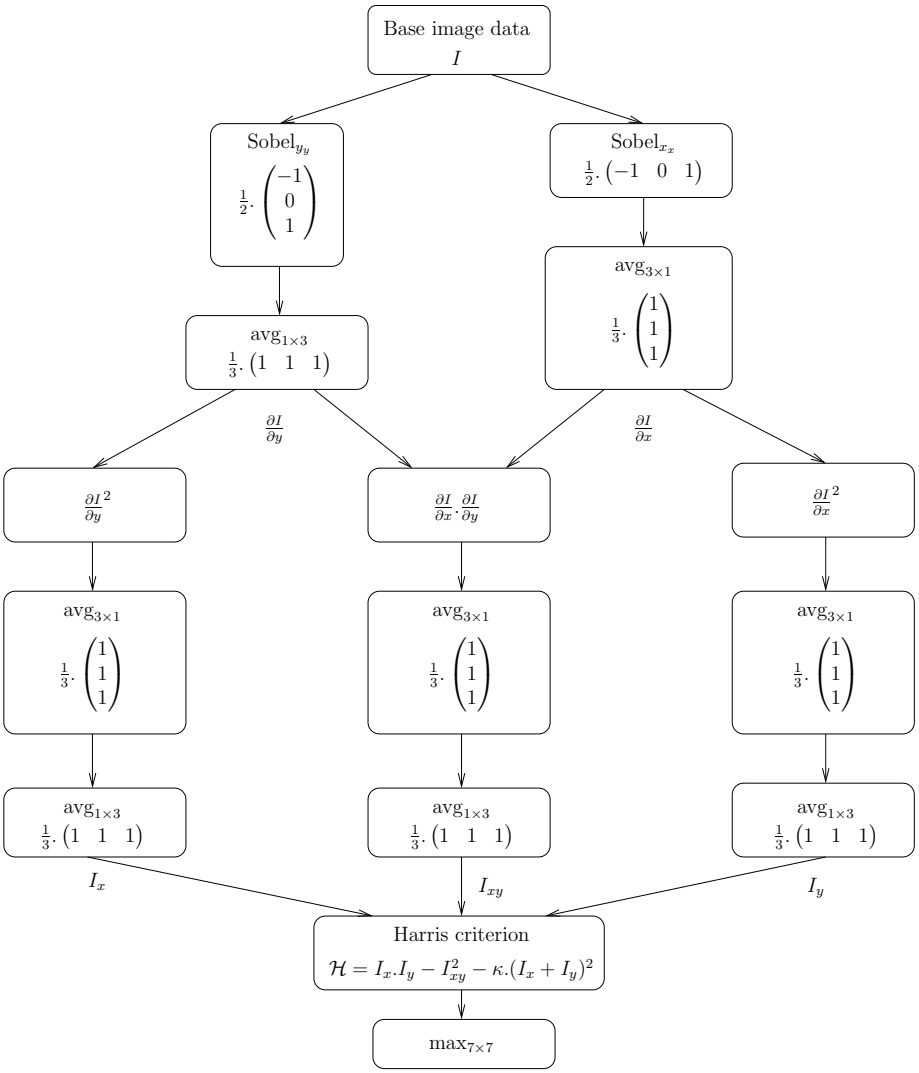


FIG. 5.11 – Chaîne des filtres utilisés après séparation des filtres

La notion de *filtre séparable* permet de réduire la complexité de l'application d'un filtre à une image. Souvent la matrice d'une convolution f peut s'exprimer comme un produit matriciel :

$$M(f) = M(f_1) \times M(f_2)$$

Dans ce cas l'application du filtre initial peut être décomposée en l'application successive des deux filtres du produit :

$$M \otimes I(x, y) = M(f_1) \otimes (M(f_2) \otimes I)(x, y)$$

Le filtre est *séparable* lorsque les matrices de décomposition $M(f_1)$ et $M(f_2)$ sont respectivement une matrice colonne et une matrice ligne – le filtre est alors séparé en une composante purement verticale et une composante purement horizontale.

À titre d'exemple le filtre de moyennage utilisé lors du calcul de M peut s'écrire comme le produit suivant :

$$\frac{1}{9}.\begin{pmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{pmatrix} = \frac{1}{3}.\begin{pmatrix}1\\ 1\\ 1\end{pmatrix} \times \frac{1}{3}.\begin{pmatrix}1 & 1 & 1\end{pmatrix}$$

Le filtre 3×3 coûte $3 \times 3 = 9$ additions lorsque l'opérateur est écrit sous sa forme initiale contre $2 \times 3 = 6$ additions lorsqu'est mise à profit la séparabilité.

D'une manière générale, la séparabilité d'un filtre permet de linéariser une complexité quadratique.

La première étape d'une implémentation efficace de l'algorithme d'extraction des points de Harris est d'utiliser la séparabilité de tous les filtres de convolution de 5.10. L'application de ces optimisations à notre algorithme de Harris aboutit à la nouvelle chaîne de filtres figurée en 5.11.

5.4.1.4 Synthèse d'un filtre

La chaîne de filtres de la figure 5.11 est une description mathématique. Elle n'indique rien sur la manière de calculer effectivement chacun des filtres décrits. Nous décrivons maintenant la méthodologie d'implémentation.

Implémentation d'un filtre Un filtre est implémenté comme un opérateur sur un flot de pixels qui décrit l'image en raster-scan. Dans ces conditions, un filtre utilise un registre à décalage pour mémoriser le flux de données en entrée sur une fenêtre glissante. Par exemple, le filtre représenté par le noyau de convolution suivant :

$$\begin{pmatrix}1 & 1 & 1\end{pmatrix} \tag{5.16}$$

contient deux registres à décalage de profondeur unitaire, est implémenté en DSL sous la forme :

```
fun filter(p0) =  (f) {
  p1 = Z(p0);
  p2 = Z(p1);
  f = p0 + p1 + p2;
}
```

et en C avec un tampon circulaire :

```
int filter(int px) {
   /* Ring buffer */
   static int rbuf[2];
   /* Control variable for the ring buffer */
   static unsigned i = 0;

   int val = px + rbuf[0] + rbuf[1];
   rbuf[i%2] = px;
   i++;
}
```

Lorsque la largeur de l'image est spécifiée, disons à une longueur L, le filtre représenté par le noyau de convolution suivant :

$$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \tag{5.17}$$

contient deux registres à décalage de profondeur L, et s'implémente de manière identique.

Caractéristiques de l'implémentation Les implémentations retenues pour chacun de filtres sont caractérisées par deux quantités :

- la mémoire requise dans l'implémentation du filtre ;
- le délai introduit par le filtre.

La mémoire requise est celle les registres à décalage.

Le delai introduit par le filtre correspond au décalage entre la position du pixel dont la valeur entre dans le circuit et la position du pixel à laquelle la valeur du filtre est calculée en sortie.

Par exemple sur les deux implémentations du filtre 5.16, au cycle où le pixel de coordonnées (x, y) entre dans le filtre, la valeur calculée en sortie du filtre est la valeur de la convolution pour le pixel plus ancien $(x-1, y)$. Ce filtre introduit donc un délai $(1, 0)$ entre son entrée et sa sortie.

De même le filtre 5.17, introduit-il un délai d'une ligne entre son entrée et sa sortie, de valeur $(0, 1)$.

Le principe d'économie guide l'implémentation : le délai introduit par le filtre est minimal ; il en est de même pour la mémoire requise dans les registres à décalage – la valeur en sortie du filtre est calculée aussitôt que possible et aucune information superflue au calcul n'est conservée.

5.4.1.5 Composition des filtres

Nous disponsons d'une méthode pour implémenter les filtres individuels sur un flot de données. Mais ces filtres individuels doivent être composés suivant la figure 5.11 pour extraire les points de Harris.

Le problème à résoudre lors de la composition des filtres est celui de la synchronisation des flux de données entre les filtres – en raison des délais évoqués au paragraphe précédent. Il nous faut par conséquent une méthodologie de composition des filtres dans un tel graphe, d'une forme un peu plus générale que le simple série-parallèle. Deux types de composition des filtres sont étudiés : leur

Version	Images	Durée du traitement		Vitesse	
		totale	par image	Im.s^{-1}	MiB.s^{-1}
référence	300	691.136s	2303.789ms	.4	.1
filtres	300	11.526s	38.422ms	26.0	10.7
tampons optimisés 1	300	9.927s	33.091ms	30.2	12.5
tampons optimisés 2	300	9.650s	32.167ms	31.0	12.8
entiers	300	8.676s	28.920ms	34.5	14.3
flottants SIMD	300	4.996s	16.653ms	60.0	24.9

TAB. 5.17 – Performance de code de Harris

mise en série et la synchronisation de branches parallèles en entrée d'un nœud d'arité multiple (*fan-in*).

Composition série Lorsque deux filtres sont composés en série, les délais des filtres d'accumulent. Par exemple, le délai total pour l'implémentation du filtre Sobel_x séparé est la somme du delai de Sobel_{x_x} et de $\text{avg}_{3\times1}$, c'est-à-dire $(1,0)+(0,1)=(1,1)$ – le délai attendu d'après la forme originale du filtre Sobel_x.

Synchronisation parallèle L'arrivée de flux de données en provenance de branches parallèles sur un opérateur commun impose une contrainte de synchronisation des flux de donnée. Par exemple l'opération de calcul de la fonction de réponse $\mathcal{H}$ combine le résultat des trois chaînes de filtres calculant I_x, I_{xy} et I_y en parallèle. Le sommet du graphe qui calcule la fonction de réponse agit comme point de synchronisation entre les trois branches, c'est-à-dire que les quantités I_x, I_{xy} et I_y en sortie doivent être calculées à la même position dans l'image, au même cycle d'horloge.

À cette fin, les délais cumulés dans chacune des branches parallèles doivent être identiques au point de synchronisation. Si ils ne le sont pas, de simples registres à décalage permettent la resynchronisation, à la manière de la méthode employée dans la retemporisation en 2.3.3.3.

En l'occurrence, le délai total depuis la source des données Idans chacune des branches calculant I_x, I_{xy} et I_y est identique et égal à $(2,2)$. Le délai final de l'extracteur qui prend en compte le filtre $\max_{7\times7}$ final de délai $(3,3)$, est de $(5,5)$.

5.4.2 Présentation des résultats

Le développement d'une version FPGA de l'algorithme d'extraction des points de Harris suit la méthodologie déjà utilisée en 5.3, à savoir le développement d'une version logicielle de l'algorithme, puis la déportation des parties critiques du calcul du logiciel vers le coprocesseur Pam. À chacune de ces étapes, on vérifie soigneusement et automatiquement que le résultat du calcul est strictement identique à une version de référence.

Le code source de ces différentes versions est disponible en [47]. Les vitesses rapportées sont celles effectivement mesurées depuis la lecture de l'image sur le disque jusqu'à la production d'une image-résultat en noir et blanc.

5.4.2.1 Version logicielle de référence

La version logicielle de référence calcule sans détours[4] le critère de Harris pour chacun des points de l'image. Sa vitesse exprimée dans la première ligne du tableau 5.17 est exécrable – ce à quoi nous pouvions nous attendre, au vu des nombreux calculs dupliqués.

5.4.2.2 Version logicielle de simulation matérielle

L'étape suivante consiste à réaliser une version logicielle en suivant la méthodologie matérielle, à l'aide de la description de filtres suivant la figure 5.11 et la méthodologie d'implémentation des filtres en C décrite en 5.4.1.4. Les fonctions qui calculent chacun des filtres sont composées naturellement en C suivant la méthodologie décrite en 5.4.1.5.

Les résultats obtenus pour cette implémentation sont très encourageants, puisque l'extraction est faite en temps réel, à plus de 26 images par secondes.

5.4.2.3 Élimination du contrôle commun

La version précédente de l'algorithme fait intervenir dans chacune des fonctions décrivant un filtre une variable d'état qui contrôle le registre à décalage du filtre.

Il faut remarquer que la plupart de ces variables d'état sont synchrones, c'est-à-dire qu'elles prennent la même valeur dans la même itération de la boucle, parce que les registres à décalage sont identiques.

Le partage des variables identiques amène un gain de vitesse appréciable, puisque la vitesse d'extraction passe alors à 30 images par seconde.

5.4.2.4 Élimination des opérations modulaires

Le calcul des indices dans les tampons circulaires fait appel à des opérations modulaires sur la variable de contrôle. Ces opérations modulaires peuvent être remplacées par des opérations conditionnelles. Le gain de vitesse est alors minime mais significatif, puisque l'extraction atteint maintenant 31 images par secondes.

5.4.2.5 Utilisation de l'ALU

Si les unités flottantes des processeurs modernes sont performantes, les unités de calcul entier restent bien supérieures en vitesse. La conversion de l'algorithme des flottants vers les entiers, qui est ici entièrement réalisable, prouve ce point : la version entière de l'algorithme gagne beaucoup en vitesse, pour atteindre 35 images par secondes.

En ce qui concerne l'implémentation FPGA, la conversion vers l'arithmétique entière est la condition *sine qua non* d'une implémentation compacte et performante.

[4] Early optimization is the root of all evil.

5.4.2.6 Utilisation des instructions SIMD

L'utilisation des instructions SIMD disponibles sur notre processeur Opteron permet de mettre en parallèle quatre opérations arithmétiques sur des flottants de longueur 32 bits. Les flottants sont choisis en raison d'un jeu d'instruction SIMD plus complet et plus souple[5] que son équivalent entier.

La méthode suit l'implémentation utilisée pour les algorithmes de diffusion en 5.2.2.2, à savoir que quatre images sont traitées en parallèle dans le chemin de données. Les résultats pour cette implémentation sont excellents et permettent d'obtenir une vitesse de traitement trois fois supérieure au temps réel.

Il faut noter que la partie la plus difficile à optimiser pour obtenir de bons résultats avec les instructions SIMD est le filtre $\max_{7\times7}$ en fin de traitement, qui doit être récrit pour ne pas utiliser de branchement. Nous renvoyons le lecteur au code-source de l'implémentation [47] pour les détails de cette optimisation.

Au vu des résultats obtenus avec l'implémentation logicielle de l'extracteur de points de Harris, il n'apparaît pas opportun de passer à l'utilisation du co-processeur FPGA.

[5] en particulier, les flottants dispensent de se préoccuper des variations de largeur du chemin de données entier, entre 16 et 64 bits

6 Conclusion et perspectives

La thèse proposait de réaliser un langage de description et de synthèse efficace de routines matérielles pour leur usage sur une Pam. Cet objectif est maintenant évalué à l'aune des résultats obtenus sur les circuits réalisés dans le chapitre 5.

6.1 Évaluation de DSL

6.1.1 Résultats

Tous les circuits réalisés atteignent des performances qui touchent aux limites de l'architecture ciblée en termes de fréquence de fonctionnement et aux limites de ressource du FPGA d'implantation. Leur expression en DSL reste néanmoins claire et concise.

L'exécution des routines est presqu'optimale – elle sature la bande passante du bus PCI.

Ainsi, tant sur le plan de la description de la routine, que sur celui de sa synthèse ou encore celui de son intégration système, DSL répond efficacement aux problèmes posés par la conception de circuit pour une Pam.

6.1.2 Limites

Toutefois, les limites de notre approche sont nombreuses, même si les exemples choisis ne les mettent pas nécessairement en évidence.

6.1.2.1 Langage de description

DSL et son approche synchrone sont très adaptés à la description de circuits orientés flot de données, mais beaucoup moins à la description de machines à états complexes.

Concernant la retemporisation, le modèle de délai utilisé est un modèle discret simpliste, qui gagnerait certainement à refléter plus finement la réalité. Par ailleurs, DSL ne prend pas en compte la retemporisation interne des opérateurs arithmétiques, qui est pourtant simple pour les composantes triviales du graphe de condensation.

Le dernier problème, et non des moindres, est que les bons conseils de simplicité, concision, et clarté – prônés pour la description des circuits ne sont que partiellement appliqués à la programmation de DSL lui-même.

6.1.2.2 Intégration système

Le modèle des entrées-sorties du circuit choisi est très contraignant et ne permet pas, par exemple, d'implanter proprement des circuits dont le volume de données en entrée et en sortie n'est pas une quantité fixe, tel le compresseur de la section 5.1 isolé de son décompresseur.

Par ailleurs, l'API simplifiée de communication avec le co-processeur ne permet pas d'utiliser simplement des entrées-sorties asynchrones vers le coprocesseur : la programmation bas niveau est encore nécessaire à cette fin.

Le coprocesseur SEPIA souffre enfin de problèmes d'intégration système plus généraux.

En premier lieu, il est impossible d'utiliser SEPIA comme un vrai coprocesseur : le temps partagé est impossible avec un FPGA. Une solution bien adaptée à l'esprit des solutions matérielles serait de permettre le partage de l'espace sur le FPGA de manière transparente ; malheureusement les outils des vendeurs ne permettent pas encore ce type de manipulations avec une souplesse suffisante.

Enfin, les accès en mode utilisateur au moteur de DMA de SEPIA ne sont pas sécurisés. Par conséquent, l'utilisateur de SEPIA dispose des moyens d'accéder en lecture comme en écriture aux données de tout processus en cours d'exécution sur la machine. Étant données les caractéristiques du bus PCI, la correction de ce problème demanderait de déplacer en mode noyau les accès privilégiés au moteur DMA. Réaliser la fiabilisation des transactions DMA en conservant la même flexibilité pour l'utilisateur et le même niveau de performances est une opération complexe.

6.2 Perspectives

L'utilisation d'une Pam comme co-processeur pour le calcul haute performance est aujourd'hui confrontée à de nombreux concurrents.

En effet l'avantage d'un circuit sur une implémentation séquentielle réside essentiellement dans l'exploitation du haut degré de parallélisme permis par les architectures systoliques.

Or plusieurs facteurs concourrent aujourd'hui à offrir au programmeur la possibilité d'exploiter le parallélisme sur des machines autrefois purement séquentielles. Tous les supports de calcul sont soumis à ce mouvement général d'exposition explicite du parallélisme et offrent par conséquent à leur tour des possibilités de parallélisme avancées.

6.2.1 Évolution du calcul haute performance

6.2.1.1 Processeurs

Dans le monde des processeurs, le parallélisme est déjà présent. Il est souvent soigneusement dissimulé derrière une Instruction Set Architecture (ISA) purement séquentielle, mais l'architecture qui la réalise est faite d'unités d'exécutions multiples qui peuvent exploiter le parallélisme implicite du flot d'instructions à l'avenant.

Dans cette approche, le parallélisme est caché, et exploité de manière opportuniste lors de l'exécution d'un programme. Les limites de cette approche sont atteintes. D'une part la logique de contrôle nécessaire à l'exploitation du parallélisme est extrêmement complexe ; d'autre part le parallélisme présent dans un programme séquentiel est très limité. Les processeurs évoluent en conséquence en exposant de plus en plus leurs capacités de parallélisme qui doivent être utilisées explicitement.

Les jeux d'instructions SIMD exposent un parallélisme fin au niveau instruction.

Les machines Very Long Instruction Word (VLIW) poussent la logique plus loin en confiant au compilateur – plutôt qu'à la logique de contrôle des processeurs – le soin d'exploiter le parallélisme inter-instructions.

Les architectures architectures multi-cœur (Intel, AMD) ou multi-thread (architecture Niagara de Sun), voire multi-Synergistic Processing Unit (SPU) (IBM et Sony avec le Cell Broadband Engine Architecture (CBEA)) sont les derniers avatars de cette tendance lourde. À ce niveau, le programmeur doit explicitement prendre en compte le parallélisme pour en bénéficier.

Bien exploités, et nous en avons eu de nombreux exemples dans le chapitre 5, le multi-cœur et les instructions SIMD permettent des gains de vitesse importants.

6.2.1.2 GPU

Un Graphical Processing Unit (GPU) est aujourd'hui composé de plusieurs centaines d'unités spécialisées, de petits microprocesseurs qui fonctionnent en parallèle sur des tableaux de données communs.

Bien loin de se cantonner au seul calcul d'images, la puissance de calcul présente dans les cartes graphiques est aujourd'hui mise à disposition pour l'informatique haute performance [48]. En particulier, la société NVidia propose de programmer très simplement ses GPUs en C au travers de la librairie CUDA.

6.2.1.3 FPGA

Les FPGAs sont aussi soumis à des mutations. Ils utilisent les transistors disponibles en nombre toujours plus grand pour intégrer des éléments spécialisés d'interface ou de calcul, voire des cœurs de processeurs entiers.

6.2.2 Outils de programmation

Les évolutions des différentes plate-formes de calcul entraînent une évolution des méthodologies de développement.

Pour le logiciel, le temps de l'exploitation sans efforts des transistors apportés par la loi de Moore, lorsque les programmes bénéficiaient automatiquement de réductions exponentielles de leur temps d'exécution, est révolu : l'exploitation du parallélisme est maintenant explicite.

Parallèlement dans le monde des FPGAs l'exploitation des ressources spécialisées des architectures actuelles demande la réécriture des circuits pour en tirer pleinement parti.

Toutefois les outils de développement ne suivent que lentement l'évolution des méthodologies. Pourtant, les mutations dans de monde du développement logiciel et dans le monde du développement matériel sont l'occasion d'une convergence bienvenue entre les deux mondes : d'une part le logiciel doit prendre en compte le parallélisme, d'autre part le FPGA doit intégrer des éléments spécialisés, à la manière d'un jeu d'instruction spécifique.

Or, autant exécuter séquentiellement un programme décrit sous sa forme parallèle est facile – c'est ce que font par exemple tous les logiciels d'émulation de matériel – autant l'extraction du parallélisme et la génération d'architectures à partir d'une description séquentielle d'un algorithme est un problème difficile.

Ces deux points militent pour les deux caractéristiques à la base de DSL : abstraction des opérations et formulation explicite du parallélisme. Un langage commun qui permettrait d'exploiter uniformément les architectures parallèles les plus diverses reste à inventer, mais il suivra sans doute ces deux directions.

6.3 Évaluation personnelle

Cette thèse a été l'occasion pour moi d'apprendre, de formaliser et d'appliquer de nombreuses méthodologies de développement logiciel et matériel sur des architectures variées, non seulement dans le monde du FPGA, mais encore dans le monde du logiciel. J'ai pu apprécier le lien fondamental entre ces deux mondes.

DSL est la base de cet apprentissage ; les applications réalisées en sont le fruit. J'ai finalement appris le métier de la conception de circuit.

Acronymes

DAG	Graphe Acyclique Orienté – Directed Acyclic Graph
API	Application Programmer Interface
ISA	Instruction Set Architecture
VLIW	Very Long Instruction Word
MPEG	Motion Picture Expert Group
VQEG	Video Quality Expert Group
SPU	Synergistic Processing Unit
CBEA	Cell Broadband Engine Architecture
FSB	Front Side Bus
codec	Codeur - Décodeur
ASIC	Application Specific Integrated Circuit
GPU	Graphical Processing Unit
FPGA	Field Programmable Grid Array
EDIF	Electronic Design Interchange Format
NCD	Netlist Circuit Description
NGD	Native Generic Database
XDL	Xilinx Design Language
HDL	Langages de Description Matérielle – Hardware Description Languages
CMJN	Cyan, Magenta, Jaune, Noir
RVBA	Rouge, Vert, Bleu, Alpha
RVB	Rouge, Vert, Bleu
DSL	Design Source Language
B.P.	Bande Passante
SAD	Somme des valeurs Absolues des Différences
SIMD	Single Instruction, Multiple Data
SD	Standard Definition
HD	High Definition
PAL	Phase Alterned Line
NTSC	National Television System Committee
DMA	Accès Direct à la Mémoire
PIO	Programmable Input/Output

IO Input/Output
Pam Mémoire Active Programmable – Programmable Active Memory
DUT Design Under Test
LUT Look-Up Table
Loc Line of code
RAM Random Access Memory
DDR Double Data Rate
RAMB RAM Block
SRAM Static Random Access Memory
ROM Read-Only Memory
DistRAM Mémoire Distribuée – Distributed RAM
ASU Assignation Statique Unique
CPU processeur central
TSC Time Stamp Counter
BIOS Basic Input/Output System
PCI Peripheral Component Interconnect
PCI-e PCI-express
PIF PCI interface FPGA
DMAC Contrôleur DMA – DMA Controller
pips Points d'Interconnexion Programmable – Programmable Interconnect Points
IRQ Interrup ReQuest
XDL Xilinx Design Language
NCD Native Circuit Description
CLB Bloc Logique Configurable – Configurable Logic Block
FIFO First-In First-Out
ÉNS École Normale Supérieure
DI Département informatique
DI Département informatique de l'ÉNS
MMU Unité de Gestion de la Mémoire – Memory Management Unit
IOMMU IO MMU
MMX™ Matrix Math Extensions
SSE2™ Streaming SIMD Extensions 2
GNU GNU's Not Unix
GCC GNU Compiler Collection
VMA Virtual Memory Area
ICAP Port d'Accès à la Configuration Interne – Internal Configuration Access Port
SCE-MI Standard Co-Emulation Modeling Interface

Bibliographie

[1] *IEEE Std 1364-2001 : IEEE Standard Hardware Description Language based on the Verilog Hardware Description Language*, IEEE Standards Board.

[2] *IEEE Std 1076-2000 : IEEE Standard VHDL Language Reference Manual*, IEEE Standards Board, VHDL Analysis and Standardization Group.

[3] *The Esterel v7 Reference Manual*, Esterel Technologies, Novembre 2005.

[4] P. Bjesse, K. Claessen, M. Sheeran, et S. Singh, "Lava : hardware design in haskell," *Proceedings of the third ACM SIGPLAN*, 1998.

[5] K. Claessen et M. Sheeran, *The Lava programming language*, 2005. [En ligne]. Disponible : http://www.cs.chalmers.se/~koen/Lava/index.html

[6] G. Berry, "Esterel and jazz : Two synchronous languages for circuit design," Septembre 1999, p. 1–1. [En ligne]. Disponible : http://www.springerlink.com/content/kf0nxvlxk0lfny2c

[7] Wikipédia, "Taux de compression — wikipédia, l'encyclopédie libre," 2007. [En ligne]. Disponible : http://fr.wikipedia.org/w/index.php?title=Taux_de_compression&oldid=16823704

[8] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Transactions on Information Theory*, vol. 21, p. 194–203, Mars 1975.

[9] F. T. Leighton, *Introduction to parallel algorithms and architectures : array, trees, hypercubes*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1992.

[10] G. Berry, P. Bertin, F. Bourdoncle, A. Frey, et J. Vuillemin, "A strongly typed object language for circuit synthesis." 1996. [En ligne]. Disponible : http://www.exalead.com/jazz/

[11] N. Halbwachs, P. Caspi, P. Raymond, et D. Pilaud, "The synchronous data-flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, p. 1305–1320, Septembre 1991. [En ligne]. Disponible : http://citeseer.ist.psu.edu/halbwachs91synchronous.html

[12] M. Pouzet, *Lucid Synchrone, version 3. Tutorial and reference manual*, Université Paris-Sud, LRI, Avril 2006. [En ligne]. Disponible : http://www.lri.fr/~pouzet/lucid-synchrone

[13] P. Wadler, "Monads for functional programming," dans *First International Spring School on Advanced Functional Programming Techniques*. London, UK : Springer-Verlag, 1995, p. 24–52. [En ligne]. Disponible : citeseer.ist.psu.edu/wadler95monads.html

[14] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, et F. Zadek, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. on Programming Languages and Systems*, vol. A 13(4), p. 451–490, Mars 1991.

[15] R. E. Moore, *Interval Analysis*. Prentice-Hall, Englewood Cliffs New-Jersey, 1966.

[16] N. S. Nedialkov, V. Kreinovich, et S. A. Starks, "Interval arithmetic, affine arithmetic, taylor series methods : Why, what next ?" 2003. [En ligne]. Disponible : http://citeseer.ist.psu.edu/nedialkov03interval.html

[17] P. Cousot et N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," dans *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Tucson, Arizona : ACM Press, New York, NY, 1978, p. 84–97. [En ligne]. Disponible : http://citeseer.csail.mit.edu/cousot78automatic.html

[18] P. Cousot *et al.*, "The ASTRÉE static analyzer," *Ecole Normale Supérieure*, 2005. [En ligne]. Disponible : http://www.astree.ens.fr/

[19] C. E. Leiserson et J. B. Saxe, "Retiming synchronous circuitry," Palo Alto, CA, Tech. Rep., Août 1986. [En ligne]. Disponible : ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-013.pdf

[20] A. V. Aho, J. E. Hopcroft, et J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, Massachusetts : Addison-Wesley Publishing Company, 1974.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, et C. Stein, *Introduction to algorithms*, 1992.

[22] E. Nuutila, "Efficient transitive closure computation in large digraphs," *Acta Polytechnica Scandinavia : Math. Comput. Eng.*, vol. 74, p. 1–124, 1995. [En ligne]. Disponible : http://www.cs.hut.fi/~enu/ps/thesis-ch3.ps

[23] B. Dutertre et L. de Moura, "The yices smt solver." [En ligne]. Disponible : http://yices.csl.sri.com/tool-paper.pdf

[24] J. Cortadella, M. Kishinevsky, et B. Grundmann, "Self : Specification and design of a synchronous elastic architecture for dsm systems," Tech. Rep., 2006. [En ligne]. Disponible : http://www.lsi.upc.edu/~jordicf/gavina/BIB/reports/self_tr.pdf

[25] Xilinx, "Virtex-II DataSheet DS031 (version 3.4)," Mars 2005. [En ligne]. Disponible : http://www.xilinx.com/bvdocs/publications/ds031.pdf

[26] Wikipédia, "Binary prefix — wikipedia, the free encyclopedia," 2007, [Online ; accessed 13-September-2007]. [En ligne]. Disponible : http://en.wikipedia.org/w/index.php?title=Binary_prefix&oldid=157240879

[27] Xilinx, "Virtex-2 Platform FPGA User Guide (UG002 version 2.0)," Mars 2005. [En ligne]. Disponible : http://www.xilinx.com/bvdocs/userguides/ug002.pdf

[28] J. Vuillemin, "On circuits and numbers," *IEEE Transactions on Computers*, vol. 43, no. 8, p. 868–879, 1994. [En ligne]. Disponible : citeseer.ist.psu.edu/162544.html

[29] J. Hervé, F. Morain, D. Salesin, B. Serpette, J. Vuillemin, et P. Zimmermann, "Bignum : un module portable et efficace pour une arithmetique a precision arbitraire," INRIA, Tech. Rep. [En ligne]. Disponible : http://www.inria.fr/rrrt/rr-1016.html

[30] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, et P. Boucard, "Programmable active memories : the coming of age," *IEEE Trans. on VLSI*, vol. 4, NO.1, p. 56–69, Mars 1996.

[31] G. Kahn, "The semantics of a simple language for parallel programming," *Information Processing 74 : Proceedings of the IFIP Congress 74*, vol. North-Holland, p. 471–475, 1974.

[32] G. Kahn et D. B. MacQueen, "Coroutines and networks of parallel processes," *Information Processing 77 : Proceedings of the IFIP Congress 77*, vol. North-Holland, p. 993–998, 1977.

[33] L. Moll, A. Heirich, et M. Shand, "Sepia : scalable 3D compositing using PCI Pamette," dans *IEEE Symposium on FPGAs for Custom Computing Machines*, K. L. Pocek et J. Arnold, Eds. Los Alamitos, CA : IEEE Computer Society Press, 1999, p. 146–155. [En ligne]. Disponible : citeseer.ist.psu.edu/475281.html

[34] M. Shand, "Programmation de la carte pci pamette," *École Normale Supérieure*, 2005. [En ligne]. Disponible : http://www.di.ens.fr/AlgorithmiqueMaterielle.html

[35] J. Corbet, A. Rubini, et G. Kroah-Hartman, *LINUX Device Drivers*. O'Reilly Media, 2005. [En ligne]. Disponible : http://lwn.net/Kernel/LDD3/

[36] R. Ginosar, "Fourteen ways to fool your synchronizer," dans *ASYNC '03 : Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems*. Washington, DC, USA : IEEE Computer Society, 2003, p. 89. [En ligne]. Disponible : http://www.ee.technion.ac.il/people/ran/papers/Sync_Errors_Feb03.pdf

[37] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, et M. Pouzet, "*N*-Synchronous Kahn Networks : a Relaxed Model of Synchrony for Real-Time Systems," dans *ACM International Conference on Principles of Programming Languages (POPL'06)*, Charleston, South Carolina, USA, January 2006. [En ligne]. Disponible : http://www.lri.fr/~pouzet/bib/popl06.pdf

[38] *Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual*, Mai 2003. [En ligne]. Disponible : http://www.eda.org/itc/scemi.pdf

[39] L. Moll et M. Shand, "Systems performance measurement on PCI pamette," dans *IEEE Symposium on FPGAs for Custom Computing Machines*, K. L. Pocek et J. Arnold, Eds. Los Alamitos, CA : IEEE Computer Society Press, 1997, p. 125–133. [En ligne]. Disponible : http://citeseer.ist.psu.edu/1690.html

[40] H. Touati et M. Shand, *PamDC : a C++ Library for the Simulation and Generation of Xilinx FPGA Designs*, 1999. [En ligne]. Disponible : http://research.compaq.com/SRC/pamette/

[41] J.-B. Note et J. Vuillemin, "Towards automatically compiling efficien fpga hardware," dans *Proceedings of the International Workshop on Hardware Design and Functional Languages*, Mars 2007, p. 115–124.

[42] J.-C. Tuan, T.-S. Chang, et C.-W. Jen, "On the data reuse and memory bandwidth analysis for full-search block-matching vlsi architecture," *IEEE Trans. Circuits Syst. Video Techn.*, vol. 12, no. 1, p. 61–72, 2002.

[43] N. Nethercote et J. Seward, "Valgrind : A program supervision framework," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, 2003.

[44] J. Hopcroft, W. Paul, et L. Valiant, "On time versus space," *J. ACM*, vol. 24, no. 2, p. 332–337, 1977.

[45] C. D. Thompson, "Area-time complexity for vlsi," dans *STOC '79 : Proceedings of the eleventh annual ACM symposium on Theory of computing*. New York, NY, USA : ACM Press, 1979, p. 81–88.

[46] C. Harris et M. Stephens, "A combined corner and edge detector," dans *Proceedings of The Fourth Alvey Vision Conference*, Manchester, 1988, p. 147–151. [En ligne]. Disponible : http://www.csse.uwa.edu.au/~pk/research/matlabfns/Spatial/Docs/Harris/

[47] J.-B. Note. (2006) Implémentation de référence de l'extracteur de harris. [En ligne]. Disponible : http://gxaafoot.homelinux.org/cgi-bin/archzoom.cgi/jean-baptiste.note@m4x.org--libre/harris

[48] Wikipédia, "Gpgpu — wikipedia, the free encyclopedia," 2007. [En ligne]. Disponible : http://en.wikipedia.org/w/index.php?title=GPGPU&oldid=132247297

Résumé

Cette thèse explore les possibilités algorithmiques offertes par la synthèse de haut niveau de circuits dans le cadre de la *logique synchrone* et à destination d'une Mémoire Active Programmable. Une chaîne de compilation expérimentale permettant de générer automatiquement un circuit reconfigurable à partir d'une spécification de haut niveau y est présentée.

Le langage de haut niveau est DSL (Design Source Language). DSL est basé sur le langage fonctionnel Jazz. DSL permet de décrire tout type de circuit dans le modèle de la logique synchrone, d'en faire la simulation et la synthèse, puis de l'exécuter sur une Mémoire Active Programmable.

Le compilateur procède par étapes successives pour synthétiser un circuit à partir de son code-source de haut niveau. Chacune des étapes de la compilation génère des *annotations* qui précisent les propriétés du circuit jusqu'à une forme synthétisable. Les annotations sont pour la plupart ajoutées automatiquement par le compilateur mais sont partie intégrante de la syntaxe de DSL et peuvent ainsi être précisées par le concepteur.

DSL prend en charge la génération automatique de l'ensemble des routines systèmes qui permettent au circuit de communiquer avec son hôte.

Ce système de prototypage et d'accélération matérielle automatique sur PAM est testé sur des circuits variés, comme des algorithmes de tramage, d'estimation de mouvement et de détection des points de Harris.

Mots-clef : PAM, Mémoire Active Programmable, langage synchrone, conception de circuits, calcul haute performance, compression sans perte, tramage, estimation de mouvement.

Abstract

This dissertation explores the algorithmic opportunities offered by the high-level synthesis of *digital synchronous circuits* targeting Programmable Active Memories. We present the implementation of an experimental compiler toolchain which automatically compiles a reconfigurable circuit from its high-level specification.

The circuit description is written in DSL (Design Source Language (DSL)) which is rooted in the Jazz functional programming language. DSL can describe any *digital synchronous circuit*. DSL can then simulate, synthetize and execute the hardware routine on a Programmable Active Memory.

The compiler proceeds in stages from the DSL source-code in order synthetize the circuit from its high-level specification. Each compiler stage generates *annotations* which specify circuit properties down to a synthetizable form. Most annotations are generated automatically. Manual annotations are part of the syntax of DSL, and can be used by an experienced designer to specify circuit details through source guidance.

DSL automatically generates the hardware and software interfaces needed for speedy communication between the circuit and its host.

The whole system is thus a fully automatic hardware accelerator for software applications described in DSL. It has been tested on a strong set of algorithms, including a lossless codec, digital dithering algorithms, motion estimation and Harris feature point extraction.

Keywords: PAM, Programmable Active Memory, synchronous language, circuit design, high-performance computing, lossless compression, dithering, motion estimation.