UNIVERSITÉ DE MONTRÉAL

SYNTHÈSE ET DESCRIPTION DE CIRCUITS NUMÉRIQUES AU NIVEAU DES TRANSFERTS SYNCHRONISÉS PAR LES DONNÉES

MARC-ANDRÉ DAIGNEAULT
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE ÉLECTRIQUE)
DÉCEMBRE 2015

© Marc-André Daigneault, 2015.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

SYNTHÈSE ET DESCRIPTION DE CIRCUITS NUMÉRIQUES AU NIVEAU DES TRANSFERTS SYNCHRONISÉS PAR LES DONNÉES

présentée par : DAIGNEAULT Marc-André
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. SAVARIA Yvon, Ph. D., président
M. DAVID Jean Pierre, Ph. D., membre et directeur de recherche
M. MAHSEREDJIAN Jean, Ph. D., membre et codirecteur de recherche
M. BOIS Guy, Ph. D., membre
M. BLAQUIÈRE Yves, Ph. D., membre externe

DÉDICACE

"Tout le monde tient le beau pour le beau,
c'est en cela que réside sa laideur.
Tout le monde tient le bien pour le bien,
c'est en cela que réside son mal."
- Lao-tseu, Tao Tö King

"Where ignorance is our master,
there is no possibility of real peace."
- Dalaï Lama XIV

REMERCIEMENTS

Cette thèse de Doctorat représente pour moi l'accomplissement et l'aboutissement d'un long parcours académique, qui m'a parfois semblé n'avoir de fin ! Un long parcours qui hélas je dois l'admettre n'a commencé à me stimuler qu'à partir de la maîtrise. Néanmoins je ne saurais manquer de reconnaitre que l'accomplissement de cette présente étape de ma vie, que je suis sur le point de compléter, est sans contredit le fruit de l'accomplissement de toutes les étapes passées, différentes étapes qui m'ont permis d'arriver à celle-ci. Cet accomplissement n'est pas que le miens, il est également celui de tous ceux et celles qui m'ont accompagné, supporté, inspiré, et permis de grandir tout au long de ma vie. Sans eux, il m'est difficile de croire que je serais devenu la personne que je suis aujourd'hui, que j'aurais pu me rendre aussi loin dans mon parcours, et de mener à terme cette longue aventure qu'aura été pour moi les études supérieures à l'École Polytechnique de Montréal. C'est donc avec grand plaisir et humilité que je tiens ici à remercier tous ceux et celles qui ont contribué à cette grande réalisation personnelle.

J'aimerais tout d'abord remercier ma proche famille, René, Sylvie, Véronique, ainsi que Michel, Jean-Philippe, Marc-André, et Jean-François, auprès de qui j'ai eu la chance inouïe de grandir tout au long de ma vie. Vous avez sans contredit constitué un point d'ancrage, et un phare, d'une valeur inestimable pour un petit navire qui est souvent parti à la dérive. Les mots me manqueraient ici pour exprimer toute la reconnaissance que je vous dois. Je tiens également à remercier la grande famille, mes grands-parents, oncles et tantes, ainsi que mes cousins et cousines qui ont laissé en moi une empreinte indélébile de par leur chaleur, leurs valeurs, leur humour, et leur simplicité. Je ne saurais tous les mentionner tant la liste est longue. Je garderai toujours un souvenir intarissable de combien j'ai ri lors des nombreux soupers et parties de cartes les dimanches chez les Daigneault tout au long de ma jeunesse. Roger, André, et Claire, merci. Une attention toute spéciale aussi à ma marraine et mon parrain, Claudette et Fernand, pour tous les moments passés alors que vous nous gardiez de jour ma soeur et moi pendant notre petite enfance.

J'aimerais également remercier les peu nombreuses, mais combien tant exceptionnelles compagnes amoureuses avec qui j'ai eu la chance de partager des moments privilégies de ma vie. Tout particulièrement Isabelle, une femme au coeur d'une générosité rarement égalée, qui est sans aucun doute à ce jour celle qui m'a le plus fait grandir en tant qu'homme.

Je ne saurais manquer de remercier tous mes amis, de la petite école à aujourd'hui, avec qui j'ai passé une grande partie des plus beaux moments de ma vie. J'ai tant ri auprès de vous,

et je me compte si chanceux d'avoir pu grandir auprès et avec vous. En quelque sorte vous n'avez été pour moi rien de moins qu'une deuxième famille.

Je tiens également à remercier tous les collègues de laboratoire avec qui j'ai partagé l'espace unique du local M-5028 pendant toutes mes études supérieures : David, Etienne B., Patrick, Mathieu, Tarek, Jonas, Walid, Adrien, Nasreddine, Matthieu, Hêmin, et Federico, même si je n'y ai sans doute pas été présent autant que je l'aurais dû. Merci pour tous les échanges, partages, et débats que nous avons eu, et pour m'avoir fait beaucoup rire bien sur, en plus d'avoir contribué à m'offrir une ouverture toute particulière sur le monde et des réalités d'ailleurs. Il m'est difficile d'espérer retrouver un jour dans un si petit espace autant de personnes si exceptionnelles. Je tiens également à remercier les nombreux collègues et coéquipiers avec qui j'ai eu la chance de travailler tout au long de mes études, tout spécialement Yan et Etienne L. de l'équipe *Dreamwafer*, que j'ai eu la chance de rencontrer avant de débuter ma maîtrise.

Je tiens aussi à remercier tous les étudiants, de l'École Polytechnique de Montréal et de l'UQAM, à qui j'ai eu le privilège d'enseigner pour les cours et laboratoires de circuits logiques, de prototypage rapide de systèmes numériques, ainsi que de microcontrôleurs et applications, pendant toute la durée de mes études supérieures. De ceux qui m'ont donné le plus de troubles à ceux qui ont su me surprendre fois après fois alors que je mettais souvent la barre de mes évaluations assez haute. J'espère que vous ne m'en voulez pas trop, mais en face d'un tel potentiel et d'un tel talent, je ne me serais jamais pardonné de faire une évaluation trop facile. J'ai toujours cru que l'université devait offrir un milieu propice à la découverte de nos forces et faiblesses, et nous permettant de nous dépasser et de trouver nos limites. Car nos limites sont souvent beaucoup plus loin que nous le pensons, et il m'a toujours été d'une grande importance de contribuer à vous offrir un contexte vous permettant de les découvrir. Et de mémoire, il y en avait généralement toujours au moins un dans le groupe qui arrivait à me démontrer que je n'avais pas trop exagéré ! Dans tous les cas, j'espère avoir su vous faire grandir, je peux vous dire en retour que vous m'avez permis de grandir et que vous avez su, de par votre curiosité, votre détermination, et vos efforts, à m'inspirer. Une attention toute spéciale à l'intention des étudiants du dernier groupe de cours de circuits logiques à qui j'ai eu l'honneur d'enseigner pour les applaudissements à la fin des derniers cours et l'ovation de fin de trimestre, j'en suis toujours tout ému encore aujourd'hui quand je me remémore ces moments. Merci à tous.

Je tiens également à remercier mon directeur de recherche, le professeur David, pour avoir encadré et supervisé mes travaux aux études supérieures à l'École Polytechnique de Montréal. Jean Pierre a su m'inspirer, et me faire grandir non seulement sur le plan professionnel mais également personnel. Je lui serai toujours reconnaissant pour le support et son écoute des

plus précieuse dans certains des moments les plus difficiles de ma vie. Il a su m'amener plus loin, et me faire découvrir une passion pour la recherche en science appliquée dans les domaines de la conception des circuits numériques et de la science informatique, ainsi que pour l'enseignement. C'eut été un honneur pour moi de passer une des périodes les plus stimulantes de ma vie sous sa supervision.

Je tiens également à remercier nombre de professeurs qui m'ont enseigné ou avec qui j'ai eu la chance de travailler durant mon long parcours. De part leur rigueur intellectuelle, leur passion, leur professionnalisme, et leur niveau d'expertise, ils ont été pour moi de grandes sources d'inspiration. J'en oubli sans doutes quelques-uns, mais je pense notamment aux professeurs Yves Audet, Yves Blaquière, Guy Bois, Jean-Jules Brault, Jean Pierre David, Robert Guardo, Pierre Langlois, Jean-Jacques Laurin, Mario Lefebvre, Jean Mahseredjian, Christian Morin, Yvon Savaria, et Mohamad Sawan. J'en profite également pour remercier l'administration de l'École Polytechnique de Montréal et son personnel pour avoir su réunir des professeurs de si grande qualité, et de m'avoir permis d'étudier dans un environnement aussi stimulant. C'eut été pour moi un honneur de pouvoir réaliser mon parcours dans cet établissement de premier plan au niveau de la recherche scientifique en ingénierie au Canada, et de calibre international.

Je tiens également à remercier le Conseil de Recherche en Science Naturelle et de Génie (CRSNG) du gouvernement du Canada pour m'avoir octroyé une bourse d'études de 3 ans, qui a été pour moi un signal clair que je me devais de tenter cette grande aventure, et de pousser mes études plus loin.

Je tiens en tout dernier lieu à témoigner de ma gratitude envers mon employeur actuel, Synopsys Inc., pour m'avoir invité à leurs bureaux pour une entrevue cet été alors je terminais le premier jet de la rédaction de cette thèse, ainsi qu'à toute l'équipe pour l'accueil chaleureux auquel j'ai eu droit à mon arrivée suivant l'acceptation de l'offre d'emploi qui m'avait été faite. Mon directeur de recherche s'était fait rassurant vers la fin de cette thèse comme quoi j'étais attendu à la fin de mes études, mais je n'en étais pas moins des plus heureux lorsque cette affirmation eut été démontrée par un exemple réel. La réalisation de cette thèse de doctorat a été pour moi une des aventures les plus éprouvante à laquelle je me suis prêté dans ma vie, et a certes mis ma détermination à rude épreuve. Dans ce long périple où les limites sont infinies et la destination inconnue, je me suis souvent remis en question, et pendant une longue période l'horizon lointain me paraissait bien sombre. Cette aventure me fit souvent penser au récit de la traversée de l'Atlantique en kayak, seul, sans assistance et sans escale, par Mathieu Morverand, que nous avions lu une fois à l'école secondaire. C'est aujourd'hui avec grande humilité mais non sans fierté que je porte un regard derrière moi

sur ce que j'ai réalisé pendant cette étape déterminante de ma vie. Et c'est avec le plus grand enthousiasme que j'en entreprends une nouvelle, en joignant l'équipe du groupe de vérification chez Synopsys, un leader mondial dans le domaine des outils de conception et de vérification de circuits intégrés. Une nouvelle étape, rendue possible par toutes celles qui lui ont précédée.

RÉSUMÉ

Au-delà des processeurs d'instructions multi-coeurs, le monde du traitement numérique haute performance moderne est également caractérisé par l'utilisation de circuits spécifiques à un domaine d'application implémentés au moyen de circuits programmables FPGA (réseau de portes programmables in situ). Les FPGA représentent des candidats intéressants à la réalisation de calculs haute-performances pour différentes raisons. D'une part, le nombre importants de blocs de propriétés intellectuelles gravés en dur sur ces puces (processeurs, mémoires, unités de traitement de signal numérique) réduit l'écart qui les sépare des circuits intégrés dédiés en termes de ressources disponibles. Un écart qui s'explique par le haut niveau de configurabilité offert par le circuit programmable, une capacité pour laquelle un grand nombre de ressources doit être dédié sans être utilisé par le circuit programmé. Néanmoins dans un contexte où souvent plus de transistors sont disponibles qu'on puisse en utiliser, le coût associé à la configurabilité s'en trouve d'autant réduit.

De par leur capacité à être reconfigurés complètement ou partiellement, les FPGAs modernes, tout comme les processeurs d'instructions, offrent la flexibilité requise pour supporter un grand nombre d'applications. Néanmoins, contrairement aux processeurs d'instructions qui peuvent être programmés avec différents langages de programmation haut-niveau (Java, C#, C/C++, MPI, OpenMP, OpenCL), la programmation d'un FPGA requiert la spécification d'un circuit numérique, ce qui représente un obstacle majeur à leur plus grande adoption. La description de circuits numériques est généralement exprimée au moyen d'un langage concurrent pour lequel le niveau d'abstraction se situe au niveau des transferts entre registres (RTL), tels les langages VHDL et Verilog. Pour une application donnée, la réalisation d'un circuit numérique spécialisé requiert typiquement un effort de conception significativement plus grand qu'une réalisation logicielle. Il existe aujourd'hui différents outils académiques et commerciaux permettant la synthèse haut-niveau de circuits numériques en partant de descriptions C/C++/SystemC, et plus récemment OpenCL. Cependant, selon l'application considérée, ces outils ne permettent pas toujours d'obtenir des performances comparables à celles qui peuvent être obtenues avec une description RTL produite manuellement.

On s'intéresse dans ce travail à un outil de synthèse de niveau intermédiaire offrant un compromis entre les performances atteignables au moyen d'une méthode de conception RTL, ainsi que les temps de conception que permet la synthèse à haut-niveau. On considère ainsi la synthèse de circuits numériques partant d'un langage supportant la description de machines à états algorithmiques (ASM) contrôlant des connexions entre des sources et des puits avec

des interfaces de synchronisation prédéfinies. Ces interfaces sont similaires aux interfaces à flot de données *AXI4-Stream* et *Avalon-Streaming*, disposant de signaux de synchronisation de type prêt-à-envoyer/prêt-à-recevoir pour supporter la synchronisation des transferts à la source et à la destination.

Le langage d'entrée considéré est basé sur le langage de description de circuits CASM (*Channel-based Algorithmic State Machine*) existant. Le langage CASM permet la description de connexions *bloquantes*, qui bloquent le flot de contrôle d'une ASM (*Algorithmic State Machine*) tant qu'un transfert de donnée n'a pas eu lieu entre la source et le puits correspondants. Ce langage permet ainsi la description de circuits numériques comme un ensemble de processus concurrents activant dynamiquement des transferts synchronisés par les données, et ce en faisant abstraction du détail de la logique de contrôle requise pour supporter cette synchronisation. Néanmoins, l'interconnexion de sources et de puits dans différentes topologies peut induire des boucles combinatoires en termes des signaux de synchronisation des interfaces à flot de données. De telles relations cycliques sont problématiques d'une part parce qu'elles sont associées à des comportements indéterminés, et d'autre part parce qu'elles mènent à des circuits qui ne sont pas synthétisables ou qui n'ont pas le comportement désiré. La présence de telles boucles combinatoires peut s'expliquer par la présence de relations combinatoires entre les entrées et sorties de synchronisation des différents composants interconnectés. Nous proposons une approche automatisée au niveau fonctionnel (logique) capable de transformer les boucles combinatoires en circuits acycliques synthétisables. Ceci est essentiellement réalisé en ne permettant que les états stables de la boucle correspondants à un plus grand nombre de transferts de données complétés à chaque cycle. Nous proposons également d'augmenter la syntaxe de base du langage CASM avec une syntaxe supportant la description de règles d'autorisation des transferts de données, ainsi que de nouveaux opérateurs de connexions avancés.

La méthodologie de synthèse de circuits numériques présentée dans ce travail a été automatisée au moyen d'un compilateur décrit en langage Java. Pour évaluer la viabilité de la méthode de synthèse à niveau intermédiaire proposée, celle-ci a été appliquée à un certain nombre d'applications, notamment dans le domaine du traitement numérique avec des opérateur pipelinés utilisant une représentation des nombres en virgule-flottante. La description de circuits avec le langage de description au niveau des transferts synchronisés par les données présenté offre au développeur un niveau de contrôle sur l'architecture qui se rapproche du niveau RTL, tout en permettant d'abstraire différentes difficultés inhérentes à l'interconnexion d'interfaces synchronisées par les données dans différentes topologies. La méthodologie proposée permet une description plus concise des comportements désirés, et l'automatisation de différentes tâches liées à l'implémentation de la logique de contrôle vient également réduire

les sources d'erreurs possibles.

ABSTRACT

Beyond modern multi/many-cores processors, the world of computing is also caracterized by the use of dedicated circuits implemented on Field-Programmable Gate-Arrays (FPGAs). For many reasons, modern FPGAs have become interesting targets for high-performance computing applications. On one hand, their integration of considerable amounts of IP blocks (processors, memories, DSPs) has contributed to reduce the resource/performance gap that exist with Application Specific Integrated Devices (ASICs). A gap that is easily explained by the high-level of reconfigurability that these devices provide, a feature for which a considerable amount of resources (transistors) must be dedicated. Nevertheless, in a context where often more transistors are often available than it is needed or required, the impact of such a cost is less important.

The ability to reconfigure completely or partially modern FPGAs further offer the flexibility required to support multiple different applications over time, similarly to instruction processors. However, while instruction processors can be programmed with different high abstraction level software programming languages (Java, C#, C/C++, MPI, OpenMP, OpenCL), FPGA programming typically requires the specification of a hardware design, which is a major obstacle to their widespread use. The description of a hardware design is generally done at the register-transfer level (RTL), using hardware description languages (HDLs) such as VHDL and Verilog. For a given application, the design and verification of a dedicated circuit requires a significantly more important effort than a software implementation. Nowadays, numerous commercial and academic tools allow the high-level synthesis of hardware designs starting from a software description using programming languages such as C/C++/SystemC, and more recently OpenCL. Nevertheless, depending on the application considered, at current state of the art, these tools do not allow performances that matches those which can be obtained through hand-made RTL designs.

In this work, we consider an intermediate-level synthesis methodology offering a compromise between the performances and design times that can be obtained with RTL and high-level synthesis methodologies. We consider an input hardware description language that allows the description of algorithmic state machines (ASMs) handling connections between sources and sinks with predefined streaming interfaces. These interfaces are similar AXI4-Streaming and Avalon-Streaming interfaces, featuring ready-to-send/ready-to-receive synchronisation signals.

The intermediate-level hardware description language considered is based on the existing

CASM language. CASM allows the description of blocking connections, which can implicitly stall the control flow of an ASM as long as a data transfer hasn't occurred over the specified blocking connection. The CASM language allows the description of a hardware design as a collection of ASMs handling dynamic data-synchronized connections, in a way that abstracts the details of the low-level control logic required to support the adequate synchronization of data transfers. However, the interconnection of different sources and sinks in various topologies can induce combinational loops (cyclic relations) in terms of the synchronization signals of the streaming interfaces. Such combinational loops are problematic because they are associated to indeterminate behaviors, and lead to circuits that are not synthesizable, or that do not have the desired behavior. The presence of such combinational loops can be explained by the presence of combinational relations between the synchronization inputs and outputs of the interconnected devices. In this work, we propose an automated approach at the functional (logic) level that is able to transform such combinational loops into acyclic combinational functions, which can then be synthesized into digital designs. This is achieved by allowing only the stable states that correspond to the largest amount of completed data transfers, for any given clock cycle. We also propose to augment the syntax of the CASM language such as to allow the specification of data transfer authorization rules, as well as the addition of advanced connection operators.

The automated synthesis methodology proposed in this thesis has been implemented as part of a compiler developed using the Java programming language. In order to evaluate the applicability of the proposed intermediate-level synthesis methodology, the compiler has been applied to the synthesis of various applications in the field of floating-point computing. The description of hardware designs with our intermediate-level hardware description language provides the design with a level of control that is close to that of RTL, while abstracting the complexities inherent to the interconnection of streaming interfaces in different topologies. The proposed design method allows for a more concise description of the desired behavior, while automating the various tasks associated to the implementation of the low-level synchronisation logic, reducing design and verification times.

TABLE DES MATIÈRES

DÉDICACE iii

REMERCIEMENTS iv

RÉSUMÉ viii

ABSTRACT xi

TABLE DES MATIÈRES xiii

LISTE DES TABLEAUX xvi

LISTE DES FIGURES xvii

LISTE DES SIGLES ET ABRÉVIATIONS xix

CHAPITRE 1 INTRODUCTION 1
- 1.1 Problématique 2
- 1.2 Concepts fondamentaux 2
- 1.3 Objectifs de recherche 4
- 1.4 Contributions 6
- 1.5 Plan de la thèse 8

CHAPITRE 2 REVUE DE LITTÉRATURE 9
- 2.1 Introduction 9
- 2.2 Description comportementale et synthèse haut-niveau 12
  - 2.2.1 Outils de synthèse haut-niveau C/C++/SystemC 15
  - 2.2.2 Synthèse haut-niveau OpenCL 17
- 2.3 Description et synthèse de circuits numériques au-delà du niveau RTL 18
  - 2.3.1 Bluespec SystemVerilog 18
  - 2.3.2 Maxeler Compiler 19
  - 2.3.3 Langages synchrones 19
  - 2.3.4 Machines séquentielles algorithmiques 20
  - 2.3.5 Langage de niveau intermédiaire CASM 21
- 2.4 Conclusion 22

CHAPITRE 3 DESCRIPTION DE CIRCUITS AVEC LE LANGAGE CASM+ . . 24
3.1 Introduction . . . 24
3.2 Règles d'autorisation de transferts de données . . . 24
3.3 Opérateurs de connexion différée . . . 25
3.3.1 Motivation . . . 26
3.3.2 Sémantique . . . 26
3.3.3 Ordonnancement . . . 30
3.3.4 Transformation . . . 30
3.4 Opérateurs de connexion sur réseau d'interconnexions pipeliné . . . 33
3.4.1 Motivation . . . 34
3.4.2 Sémantique . . . 34
3.4.3 Ordonnancement . . . 34
3.4.4 Transformation . . . 35
3.5 Conclusion . . . 36

CHAPITRE 4 SYNTHÈSE AUTOMATISÉE DES DECRIPTIONS CASM+ 38
4.1 Introduction . . . 38
4.2 Problématique . . . 40
4.2.1 Dépendances combinatoires structurelles . . . 40
4.2.2 Relations combinatoires cycliques . . . 42
4.3 Logique de synchronisation des transferts . . . 43
4.4 Analyse de stabilité . . . 45
4.4.1 Recherche d'un ensemble de points de rétroaction . . . 45
4.4.2 Contrainte de stabilité . . . 46
4.4.3 Mono- et Multi-stabilité . . . 47
4.5 Stabilisation des relations combinatoires cycliques . . . 48
4.5.1 Méthode I . . . 50
4.5.2 Méthode II . . . 50
4.6 Génération de fonctions combinatoires acycliques . . . 51
4.7 Conclusion . . . 52

CHAPITRE 5 APPLICATION À LA CONCEPTION DE CIRCUITS 54
5.1 Introduction . . . 54
5.2 Tri de données rapide . . . 54
5.2.1 Algorithme . . . 54
5.2.2 Conception . . . 55
5.2.3 Résultats de synthèse et d'implémentation . . . 55

5.2.4 Discussion 56
5.3 Accumulateur pipeliné 57
5.3.1 Conception 57
5.3.2 Description sans opérateur de connexion différée 58
5.3.3 Description avec opérateurs de connexions différée 59
5.3.4 Résultats de synthèse et d'implémentation 59
5.3.5 Discussion 62
5.4 Produit matriciel 63
5.4.1 Algorithme 63
5.4.2 Conception 63
5.4.3 Résultats de synthèse et d'implémentation 67
5.4.4 Discussion 68
5.5 Élimination Gaussienne 71
5.5.1 Algorithme 71
5.5.2 Conception 72
5.5.3 Résultats de synthèse et d'implémentation 75
5.5.4 Discussion 78
5.6 Inversion de matrices 80
5.6.1 Algorithme 80
5.6.2 Conception 81
5.6.3 Résultats de synthèse et d'implémentation 82
5.6.4 Temps d'exécutions 83
5.6.5 Discussion 83
5.7 Conclusion 86

CHAPITRE 6 CONCLUSION 87
6.1 Synthèse des travaux 87
6.2 Comparaison avec l'état de l'art 89
6.3 Limitations de la solution proposée et améliorations futures 92

PUBLICATIONS DE L'AUTEUR 94

RÉFÉRENCES 96

LISTE DES TABLEAUX

Tableau 3.1 Attributs de connexions 25
Tableau 4.1 Table de vérité de la contrainte de stabilité de la Figure 4.9 49
Tableau 4.2 Table de vérité de la contrainte de stabilité après stabilisation 51
Tableau 4.3 Table de vérité de la fonction $add0.out.rtr$ résolue 52
Tableau 5.1 Exécution des implémentations simple et à pipeline fonctionnel. . . . 56
Tableau 5.2 Utilisation des ressources pour les deux implémentations. 57
Tableau 5.3 Résultats de synthèse de l'accumulateur sur un FPGA Virtex-5 . . . 62
Tableau 5.4 Utilisation des ressources pour le produit matriciel (Stratix-III) . . . 69
Tableau 5.5 Utilisation des ressources pour le produit matriciel (Virtex-5) 69
Tableau 5.6 Performances pour le produit matriciel sur FPGA. 69
Tableau 5.7 Comparaison d'implémentations dédiées au produit matriciel. 70
Tableau 5.8 Utilisation des ressources pour un unité de type-I (Stratix-V) 76
Tableau 5.9 Utilisation des ressources pour l'accélérateur type-I (Stratix-V) 76
Tableau 5.10 Utilisation des ressources pour l'accélérateur type-I (Virtex-5) 76
Tableau 5.11 Efficacité des calculs d'un unité de traitement(type-I) 77
Tableau 5.12 Utilisation des ressources pour un unité de type-II (Stratix-V) 77
Tableau 5.13 Utilisation des ressources pour l'accélérateur de type-II (Stratix-V) . 78
Tableau 5.14 Utilisation des ressources pour l'accélérateur de type-II (Virtex-5) . . 78
Tableau 5.15 Efficacité des calculs d'un unité de traitement (type-II) 78
Tableau 5.16 Comparaison des unités de traitement (Virtex-5) 79
Tableau 5.17 Comparaison avec un outil de synthèse C 80
Tableau 5.18 Résultats d'implémentation en simple précision (Virtex-5) 83
Tableau 5.19 Temps d'exécution (4 coeurs de traitement) 83
Tableau 5.20 Comparaison des designs dédiés pour l'inversion de matrices 84
Tableau 5.21 Comparaison contrôle vs. chemin de données 85

LISTE DES FIGURES

Figure 1.1 Interconnexion de sources et de puits synchronisés par les données. . 4
Figure 1.2 Interconnexion produisant une relation combinatoire cyclique. 6
Figure 2.1 Niveaux d'abstractions pour la conception de circuits numériques. . . 10
Figure 2.2 ASM pour l'algorithme de recherche du PGDC. 20
Figure 2.3 Protocole de synchronisation complète. 21
Figure 2.4 Description CASM pour l'algorithme de recherche du PGDC. 22
Figure 3.1 Exemple d'utilisation de règles d'autorisation. 26
Figure 3.2 Description CASM d'un produit scalaire. 27
Figure 3.3 Description CASM d'un produit scalaire avec connexion différée . . . 28
Figure 3.4 Simple utilisation de l'opérateur de connexion différée. 29
Figure 3.5 Description de boucle avec opérateur de connexion différée. 30
Figure 3.6 Description de boucles avec opérateur de connexion différée. 31
Figure 3.7 Sémantique pour les opérateurs de connexion différée I. 32
Figure 3.8 Sémantique pour les opérateurs de connexion différée II. 33
Figure 3.9 Sémantiques pour les opérateurs de connexions réseaux. 36
Figure 4.1 Flot de compilation CASM. 39
Figure 4.2 Architecture associée à une description CASM+. 40
Figure 4.3 Deux opérateurs complètement synchronisés. 41
Figure 4.4 FIFO complètement synchronisée, avec chemin de contournement. . . 42
Figure 4.5 Accumulateur basé sur un additionneur pipeliné. 43
Figure 4.6 Exemple de réseau combinatoire cyclique. 45
Figure 4.7 Transformation d'un réseau cyclique 46
Figure 4.8 Table de vérité pour une contrainte de stabilité C 48
Figure 4.9 Exemple de réseau combinatoire acyclique boucle-ouverte. 49
Figure 4.10 Exemple de réseau combinatoire acyclique résolu. 52
Figure 5.1 Algorithme *quicksort*. 55
Figure 5.2 Algorithme *partition*. 55
Figure 5.3 Représentation STL de l'architecture avec mémoire partagée. 56
Figure 5.4 Représentation STL de l'architecture proposée. 58
Figure 5.5 Comparaison des chemins de données pour l'accumulateur 58
Figure 5.6 Description CASM+ d'un accumulateur pipeliné. 60
Figure 5.7 Description d'un accumulateur pipeliné avec connexions différées. . . 61
Figure 5.8 Algorithme de produit matriciel $A \times B = C$. 63

Figure 5.9 Représentation STL de l'architecture d'un élément de traitement. . . 64
Figure 5.10 Description CASM+ de l'ASM *MACinput*. 65
Figure 5.11 Description CASM+ de l'ASM *MACoutput*. 66
Figure 5.12 Architecture intégrant de multiples unités de traitements. 67
Figure 5.13 Représentation STL d'un réseau d'interconnexions pipeliné. 67
Figure 5.14 Description CASM+ d'un réseau d'interconnexions pipeliné. 68
Figure 5.15 Algorithme d'élimination Gaussienne. 71
Figure 5.16 Illustration des accès mémoire sur deux passes consécutives. 72
Figure 5.17 Architecture d'une unité de traitement (type-I). 73
Figure 5.18 Architecture intégrant de multiples unités de traitements en série. . . 74
Figure 5.19 Architecture d'une unité de traitement (type-II). 75
Figure 5.20 Algorithme d'inversion de matrice Gauss-Jordan. 81
Figure 5.21 Optimisation des accès mémoire. 81
Figure 5.22 Architecture dédiée pour l'inversion de matrices Gauss-Jordan. 82

LISTE DES SIGLES ET ABRÉVIATIONS

FPGA	Field-Programmable Gate Array
CPU	Central Processing Unit
GPU	Graphical Processing Unit
DSP	Digital Signal Processor
EDA	Electronic Design Automation
HLS	High-Level Synthesis
RTL	Register-Transfer Level
ITRS	International Technology Roadmap for Semiconductors
CFG	Control-Flow Graph
DFG	Data-Flow Graph
CDFG	Control/Data-Flow Graph
HTG	Hierarchical Task Graph
KPN	Kahn Process Network
LLVM	Low Level Virtual Machine
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
MPI	Message Passing Interface
SCC	Strongly Connected Component
FVS	Feedback Vertex Set
PE	Processing Element
AXI	Advanced eXtensible Interface
CIM	Computation In Memory
TLM	Transaction-Level Modeling
ANSI	American National Standards Institute
ESL	Electronic System Level
VHSIC	Very High Speed Integrated Circuit
VHDL	VHSIC Hardware Description Language
SoC	System-on-Chip
ASM	Algorithmic State Machine
RAM	Random Access Memory
FIFO	First-In First-Out
BDD	Binary Decision Diagram
LPM	Library of Parameterized Modules

FLOPS	Floating-Point Operations Per Second
LUT	Look Up Table

CHAPITRE 1 INTRODUCTION

Il y a de cela une décennie (en 2005), Intel prenait un virage important en lançant la production de puces de processeurs dont les hautes-performances allaient désormais reposer sur plusieurs coeurs (*cores*). De nos jours, les puces multiprocesseurs sont omniprésentes dans nos vie, de l'unité de traitement centrale au processeur graphique d'un ordinateur de bureau, jusqu'à l'intérieur de nos téléphones cellulaires. Elles composent également le coeur des noeuds des superordinateurs les plus puissants de la planète. Cette tendance à voir augmenter le nombre de coeurs par puces s'observe toujours alors que l'on passe graduellement de puces à multiples-coeurs (*multi-cores*) vers des puces à beaucoup-de-coeurs (*many-cores*), et elle ne semble pas sur le point de s'arrêter. Au coeur de ce changement fondamental se trouve la rupture de la mise-à-l'échelle de Dennard (Robert H. Dennard) qui stipule essentiellement que la densité de puissance des transistors demeure constante avec chaque nouvelle génération de transistors [1]. Dans ce contexte, la performance des processeurs mono-coeur a cessé de croître au rythme de la loi de Moore. De plus, les puces de processeurs modernes sont confrontées au problème du silicium obscur (*dark silicon*), qui fait référence au fait que les procédés de microfabrication peuvent désormais intégrer plus de transistors qu'il est possible d'en utiliser, de sorte à ce que toute la surface d'une puce ne puisse être utilisée simultanément [2].

Au-delà des processeurs d'instructions multi-coeurs, le monde du traitement numérique haute-performance moderne est également caractérisé par l'utilisation de circuits spécifiques à un domaine d'application implémentés au moyen de circuits programmables FPGA (réseau de portes programmables *in situ*). Intel faisait d'ailleurs récemment l'acquisition du fabricant de FPGA Altera. Les FPGA représentent des candidats intéressants à la réalisation de calculs haute-performances pour différentes raisons. D'une part, le nombre importants de blocs de propriété intellectuelle gravés en dur sur ces puces (processeurs, mémoires, unités de traitement de signal numérique) réduit l'écart qui les sépare des circuits intégrés dédiés en termes de ressources disponibles [3]. Un écart qui s'explique justement par le haut niveau de configurabilité offert par le circuit programmable, une capacité pour laquelle un grand nombre de ressources doit être dédié sans être utilisé par le circuit programmé. Néanmoins dans un contexte où plus de transistors sont disponibles qu'on puisse en utiliser, le coût associé à la configurabilité s'en trouve d'autant réduit. On note également que le problème du silicium obscur ne se pose pas dans la même mesure puisqu'une grande proportion des transistors est *de facto* inactive après la programmation d'un FPGA. L'utilisation de blocs de propriété intellectuelle dédiés permet également de réduire l'écart de performance séparant les circuits intégrés dédiés des FPGAs. Récemment, des FPGA ont été utilisés par Microsoft dans

ses centres de données pour accélérer l'engin de recherche *Bing* [4]. Les différentes analyses effectuées rapportent des améliorations significatives en termes de débits et de latence.

1.1 Problématique

De par leur capacité à être reconfigurés complètement ou partiellement, les FPGA modernes, tout comme les processeurs d'instructions, offrent la flexibilité requise pour supporter un grand nombre d'applications. Néanmoins, contrairement aux processeurs d'instructions qui sont programmés avec différents langages de programmation haut-niveau (Java, C/C++, MPI, OpenMP, OpenCL), la programmation d'un FPGA requiert la spécification d'un circuit numérique, ce qui représente un obstacle majeur à leur plus grande adoption. La description de circuits numériques est généralement exprimée au moyen d'un langage concurrent pour lequel le niveau d'abstraction se situe au niveau des transferts entre registres (RTL), tels les langages VHDL et Verilog. Pour une application donnée, la réalisation d'un circuit numérique spécialisé requiert typiquement un effort significativement plus grand qu'une réalisation logicielle. Dans [5], une comparaison entre un FPGA et une unité de traitement graphique (GPU) pour l'implémentation de noyaux de calculs d'applications scientifiques rapporte un temps de développement 10× plus important pour la solution FPGA. De plus, la description de circuits au niveau RTL est une tâche souvent réservée aux experts en conception de circuits FPGA, et est difficilement accessible aux spécialistes provenant de différents domaines d'applications scientifiques. Il existe aujourd'hui différents outils académiques et commerciaux permettant la synthèse haut-niveau de circuits numériques en partant de descriptions C/C++/SystemC [6], et plus récemment OpenCL [7]. Cependant, selon l'application considérée, ces outils ne permettent pas toujours d'obtenir des performances comparables à celles qui peuvent être obtenues avec une description RTL produite manuellement.

1.2 Concepts fondamentaux

On s'intéresse dans ce travail à un outil de synthèse de niveau intermédiaire offrant un compromis entre les performances atteignables aux moyens d'une méthode de conception RTL et les temps de conception que permet la synthèse à haut-niveau. On considère ainsi la synthèse de circuits numériques partant d'un langage supportant la description de machines à états algorithmiques contrôlant des connexions entre des sources et des puits avec des interfaces de synchronisation prédéfinies. Ces interfaces sont similaires aux interfaces à flot de données *AXI4-Stream* et *Avalon-Streaming*, disposant de signaux de synchronisation de type prêt-à-envoyer/prêt-à-recevoir pour supporter la synchronisation des transferts à la source

et à la destination. Un transfert de données survient lorsque la source et le puits associés à une connexion sont tous deux prêts. Un tel niveau d'abstraction est offert par le langage de description de circuits de niveau intermédiaire CASM, proposé dans [8]. CASM permet aussi la description de connexions *bloquantes*, qui bloquent le flot de contrôle d'une ASM tant qu'un transfert de donnée n'a pas eu lieu entre la source et le puits correspondants. Conjointement, ces deux caractéristiques fondamentales du langage CASM permettent la description de circuits numériques comme un ensemble de processus concurrents activant dynamiquement des transferts synchronisés par les données, et ce en faisant abstraction du détail de la logique de contrôle requise pour supporter cette synchronisation. Le déplacement d'une donnée entre une source et un puits synchronisés par les données devient aussi simple que la spécification d'une instruction (en langage machine) de déplacement de donnée entre registres dans un processeur d'instructions.

La Figure 1.1 illustre de manière abstraite le niveau d'abstraction considéré, comprenant un contrôleur de connexions (une machine à états), ainsi qu'un ensemble de sources et de puits synchronisés par les données interconnectés par un réseau d'interconnexion point-à-point. Le contrôleur est responsable d'activer dynamiquement des connexions entre les sources et puits présents. Lorsqu'une connexion non-bloquante est activée, un canal synchronisé par les données est établi entre la source et le puits associés à cette dernière. Lorsqu'un canal est établi, les données peuvent circuler librement de la source à la destination, à raison d'une donnée par cycle d'horloge pour lequel la source et le puits sont tous deux prêts à envoyer et recevoir, respectivement. Dans le cas d'une connexion bloquante, le canal est établi jusqu'à ce que le transfert d'une donnée ait lieu, après quoi le canal disparait (i.e. : la connexion est automatiquement désactivée).

Les sources et les puits supportent différents protocoles de synchronisation (*FS*,*HS*, et *NS*). Le protocole *FS* (*Full-Synchronized*) est associé à une interface disposant de deux signaux de synchronisation, *rts* (*ready-to-send*) et *rtr* (*ready-to-receive*). Le signal *rts* indique si la source est prête à envoyer une donnée, tandis que le signal *rtr* indique si le puits est prêt à recevoir une donnée. Le protocole *HS* (*Half-Synchronized*) est associé à une interface ne disposant que d'un signal *rts*. Pour ce protocole, la synchronisation se fait à la source uniquement. La source ne se soucie pas que la donnée soit reçue (i.e. : la contre-pression n'est pas prise en compte, on considère que le puits est toujours prêt à recevoir). Dans le cas du protocole *NS* (*Not-Synchronized*), une donnée est envoyée à chaque cycle d'horloge (i.e. : on considère que la source et le puits sont toujours prêts à envoyer et recevoir des données). Notons qu'il est possible d'interconnecter des sources et des puits qui ont des protocoles de synchronisation différents. Dans de tels cas, si une sortie de synchronisation (*rts/rtr*) ne peut pas être pairée à une entrée de synchronisation (*rts/rtr*) correspondante, cette dernière est considérée/laissée

flottante. De même, si une entrée de synchronisation (*rts/rtr*) ne peut pas être pairée à une sortie de synchronisation (*rts/rtr*) correspondante, cette dernière est assignée à la valeur 1 (vrai).

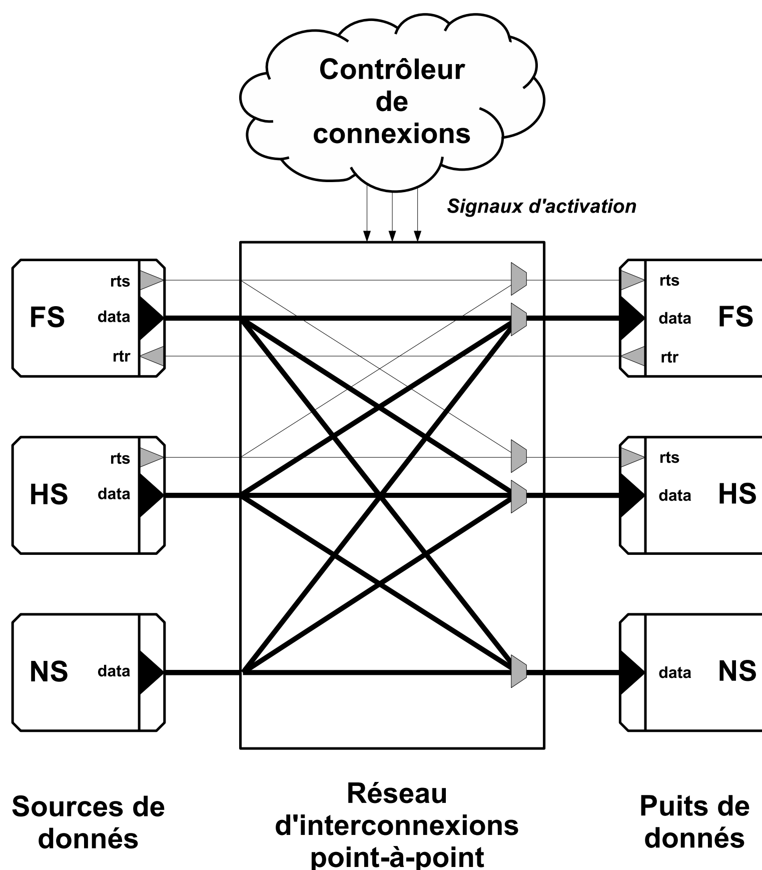


Figure 1.1 Interconnexion de sources et de puits avec des interfaces synchronisées par les données supportant différents protocoles (FS,HS,NS). Les triangles représentent les entrées et sorties, et les trapèzes des multiplexeurs.

1.3 Objectifs de recherche

Bien que le language de niveau intermédiaire CASM contribue à simplifier la description de circuits numériques, l'interconnexion de sources et de puits dans différentes topologies peut induire des boucles combinatoires en termes des signaux de synchronisation des interfaces à flot de données. De plus, ces boucles combinatoires ne sont pas nécessairement équivalentes à des réseaux acycliques de fonctions combinatoires. De telles relations cycliques sont problématiques, d'une part parce qu'elles sont associées à des comportements indéterminés, et d'autre part, parce qu'elles mènent à des circuits qui ne sont pas synthétisables ou qui n'ont

pas le comportement désiré. La présence de telles boucles combinatoires peut s'expliquer par la présence de relations combinatoires entre les entrées et sorties de synchronisation des différents composants interconnectés. Par exemple, une FIFO *complètement synchronisée* est prête à recevoir une nouvelle entrée si elle n'est pas pleine (dépendance à un état interne), ou si elle est lue au même cycle (dépendance avec une entrée de synchronisation). La Figure 1.2 donne une illustration simple de cette problématique, en considérant le modèle illustré à la Figure 1.1. Dans cette illustration, d'une part l'entrée rtr de la source FS peut dépendre de la sortie rtr du puit FS, et d'autre part la présence d'une dépendance combinatoire entre la sortie rtr du puit FS et l'entrée rtr de la source FS vient produire une relation combinatoire cyclique. Afin de résourdre le problème, nous proposons une approche automatisée au niveau fonctionnel (logique) capable de transformer les boucles combinatoires en circuits acycliques synthétisables. Ceci est essentiellement réalisé en ne permettant que les états stables de la boucle correspondants à un plus grand nombre de transferts de données complétés à chaque cycle. Cette approche fonctionnelle vient s'inscrire dans l'objectif de faciliter l'accès et de simplifier la description de circuits numériques. De même, elle permet d'abstraire un problème important associé à l'implémentation bas-niveau de la logique de contrôle pour la synchronisation des transferts de données. Nous proposons également une syntaxe permettant de spécifier des règles pour la synchronisation et l'ordonnancement des transferts de données sur un groupe de connexions activées. Cette nouvelle syntaxe offre une séparation au niveau de la description du contrôle en termes des connexions à activer vis-à-vis de comment les transferts doivent s'exécuter au sein des connexions activées. Lorsque des règles d'autorisation forment des relations cycliques, celles-ci sont automatiquement traduites en un réseau acyclique de fonctions logiques de contrôle qui maximise le nombre de transferts de données.

Nous proposons également de nouveaux opérateurs de connexions avancés développés dans l'objectif de simplifier la description de circuits qui interconnectent des opérateurs pipelinés. Une des difficultés inhérentes à l'utilisation d'opérateurs pipelinés est la présence d'un décalage temporel entre le moment auquel les opérandes sont acceptées par l'opérateur et celui pour lequel l'opérateur produit un résultat à sa sortie. Il s'ensuit que la spécification algorithmique d'architectures intégrant des opérateurs pipelinés est significativement plus complexe qu'il en est pour la spécification d'architectures intégrant des opérateurs non-pipelinés. L'opérateur de connexion différée présenté dans cette thèse contribue à abstraire cette complexité, la ramenant à un niveau qui s'apparente à celle de la spécification algorithmique d'architectures intégrant des opérateurs non-pipelinés. D'autre part, l'interconnexion d'un grand nombre de sources et puits au moyen d'un réseau combinatoire d'interconnexions point-à-point peut contribuer à augmenter significativement le chemin critique de l'architecture (au niveau de la logique d'aiguillage du chemin de données), ce qui peut affecter négativement

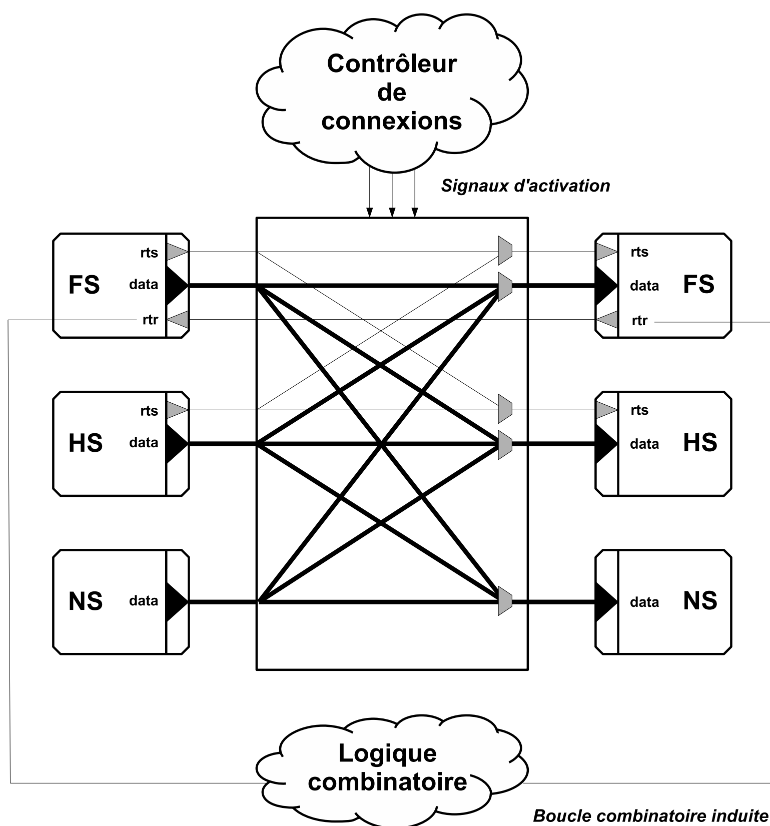


Figure 1.2 Interconnexion de sources et de puits induisant une relation combinatoire cyclique.

les performances du circuit spécifié. Pour contrecarrer ce problème, il est typiquement requis de faire appel à un ou plusieurs réseaux d'interconnexions pipelinés. Or le passage d'un réseau d'interconnexion combinatoire point-à-point vers un réseau d'interconnexion pipeliné entraîne une modification non-négligeable de la description algorithmique. Il faut notamment gérer l'adressage adéquat des destinations accessibles via le réseau d'interconnexions. Les opérateurs de connexions sur un réseau pipeliné que nous proposons dans ce travail permettent d'abstraire la gestion d'un tel adressage en inférant automatiquement un réseau d'interconnexion pipeliné synchronisé par les données. La syntaxe est telle que la description de connexions sur un réseau d'interconnexion pipeliné est similaire à la description de connexions point-à-point non-pipelinées.

1.4 Contributions

La méthodologie de synthèse de circuits numériques présentée dans ce travail a été automatisée au moyen d'un compilateur décrit en langage Java. Pour évaluer la viabilité de la méthode de synthèse à niveau intermédiaire proposée, celle-ci a été appliquée à un certain

nombre d'applications, notamment dans le domaine du traitement numérique avec des opérateurs pipelinés utilisant une représentation des nombres en virgule-flottante. On considère des circuits réalisant des algorithmes avec différents niveaux de complexité au niveau du flot de contrôle. Les applications considérées incluent le tri de données *Quicksort*, l'accumulation pipelinée en représentation virgule-flottante, le produit matriciel dense, l'élimination Gaussienne, et l'inversion de matrices. Une comparaison de certains circuits numériques produits avec notre approche de synthèse de niveau intermédiaire avec d'autres implémentations similaires décrites au niveau RTL et/ou au niveau C montrent que l'approche proposée permet un compromis intéressant entre les 2 approches, en termes de qualité d'implémentation et de temps de développement. La description de circuits avec le langage de description au niveau des transferts synchronisés par les données présenté offre au développeur un niveau de contrôle sur l'architecture qui se rapproche du niveau RTL, tout en permettant d'abstraire différentes difficultés inhérentes à l'interconnexion d'interfaces synchronisées par les données dans différentes topologies. La méthodologie proposée permet une description plus concise des comportements désirés, et l'automatisation de différentes tâches liées à l'implémentation de la logique de contrôle vient également réduire les sources d'erreurs possibles.

De manière sommaire, cette thèse présente les contributions suivantes au niveau de la description et de la synthèse automatisée de circuits au niveau des transferts synchronisés par les données :

1. Une méthode de résolution automatisée des boucles combinatoires en termes des signaux de synchronisation des interfaces synchronisées par les données. Cette résolution est exécutée de manière à produire un réseau acyclique de fonctions maximisant le nombre de transferts de données.
2. Le langage permet la description de règles contrôlant l'autorisation des transferts de données au sein des connexions activées. Les règles formant des relations cycliques en termes des signaux de synchronisation des sources et puits sont résolues de manière automatisée.
3. Un opérateur de connexion avancé de type *différé*, mis au point pour abstraire un niveau de complexité associé à la spécification de circuits intégrants des opérateurs pipelinés, et un mécanisme de traduction associé vers des transferts de base (bloquants/non-bloquants).
4. Un opérateur de connexion avancé de type *réseau d'interconnexion pipeliné*, mis au point pour abstraire la complexité liée à l'utilisation d'un réseau d'interconnexion pipeliné pour interconnecter de grands nombres de sources et de puits de données.
5. Un compilateur permettant d'automatiser la synthèse de niveau intermédiaire des

descriptions de circuits décrits au niveau des transferts synchronisés par les données. Le compilateur produit une description au niveau RTL décrite en langage VHDL synthétisable qui peut être traitée avec les outils existants.

6. L'application de la méthode de synthèse de niveau intermédiaire à différentes applications au niveau du traitement numérique avec des opérateurs pipelinés utilisant une représentation des nombres de type virgule-flottante, telles la sommation pipelinée, le produit matriciel dense, l'élimination Gaussienne, et l'inversion de matrice. Ces applications sont comparées avec des implémentations produites avec des approches RTL et de synthèse haut-niveau en langage C, de manière à évaluer le compromis offert par l'approche de niveau intermédiaire en termes de performances et de temps de conception.

1.5 Plan de la thèse

La suite de cet ouvrage est organisée comme suit : Le chapitre 2 présente un survol de l'état de l'art en matière de description et synthèse à un niveau d'abstraction allant au-delà de celui offert par les langages de descriptions au niveau des transferts entre les registres (RTL). Le chapitre 3 présente les ajouts apportés au langage CASM existant, ce qui inclut la descriptions de règles d'autorisation des transferts de données, et des opérateurs de connexions avancés. Le chapitre 4 présente la méthode au niveau fonctionnel permettant la résolution automatique des boucles combinatoires en termes des signaux de synchronisation des interfaces synchronisées par les données. Le chapitre 5 vient ensuite présenter l'application de la méthode de synthèse de niveau intermédiaire à la conception de divers circuits numériques de différentes complexités. Le chapitre 6 vient conclure cette thèse.

CHAPITRE 2 REVUE DE LITTÉRATURE

2.1 Introduction

Le domaine de l'automatisation de la conception des circuit numériques (EDA) a vu le jour dans les années suivant l'arrivé des circuits intégrés (1958). L'évolution des outils de conceptions assistée par ordinateurs est étroitement liée à l'évolution des circuits intégrés : les nouvelles générations d'outils de conception rendent possibles le développement de circuits plus complexes, qui permettent à leur tour de supporter de nouvelles générations d'outils de conception pour des circuits de plus grandes complexités. Après tout la conjecture de Moore, qui s'applique encore aujourd'hui, stipule que la densité de transistors double tous les deux ans, si bien que les circuits intégrés modernes peuvent intégrer plus d'un milliard de transistors. Il va sans dire que ce taux de croissance exponentiel demeure une source de pression inouïe et sans repos sur l'évolution des outils de conception responsables d'assister et d'automatiser le développement des circuits intégrés. Afin de composer avec cette augmentation incessante de la complexité, différents niveaux d'abstractions (plus élevés) ont été graduellement ajoutés/superposés au flot de conception des circuits numériques intégrés. Une description à un niveau d'abstraction plus élevé contient moins de détails par rapport à l'implémentation finale, mais est plus efficace pour exprimer la fonctionnalité désirée. Le flot de conception assistée par ordinateurs permet d'automatiser le raffinement d'une spécification à un niveau d'abstraction donné jusqu'à une implémentation physique finale.

La Figure 2.1 illustre les 4 principaux niveaux de conceptions du processus de synthèse automatisé des circuits numériques intégrés. À l'état de l'art, le *niveau système* représente le plus haut-niveau d'abstraction. À ce niveau d'abstraction, un système complexe est décrit en termes de processus communicants au moyen de langages tels SystemC et SystemVerilog, qui permettent de supporter différents degrés de raffinements au niveau de la spécification des communications et du traitement (*untimed, partially-timed, cycle accurate*). Il est ainsi possible de raffiner successivement la spécification exécutable du système à concevoir en partant d'un programme C/C++ dépourvu de toute notion de concurrence, jusqu'à une spécification pour laquelle les transferts et les opérations sont spécifiées avec un modèle d'exécution concurrente, exprimé au niveau du cycle d'horloge. Un tel niveau de raffinement se rapproche à peu de choses près du niveau de conception suivant, le *niveau des transferts entre registres* (RTL). Au niveau RTL, un circuit est entièrement spécifié au niveau du cycle d'horloge en termes de transferts de données entre des registres, et en termes d'opérations arithmétiques et logiques (combinatoires) à réaliser sur ces données. Au *niveau logique*, l'implémentation se précise

alors que le circuit intégré est représenté par un réseau de portes logiques (ET/NON-ET, OU/NON-OU, et inverseurs). Au niveau des portes, le comportement observé se rapproche davantage de l'implémentation physique finale du circuit. Ce niveau permet également différentes optimisations indépendantes de la technologie du réseau de portes logiques (optimisations 2-niveaux, multi-niveaux, etc...). Le plus bas niveau d'abstraction, le *niveau physique*, offre une représentation du circuit au niveau des ressources physiques qui composent le circuit. Dans un flot de conception dédié, ces ressources sont essentiellement des transistors interconnectés au moyen de traces métalliques, tandis que dans un flot de conception FPGA il s'agit plutôt d'éléments configurables du circuit programmable (blocs de logiques configurables et matrices de routage).

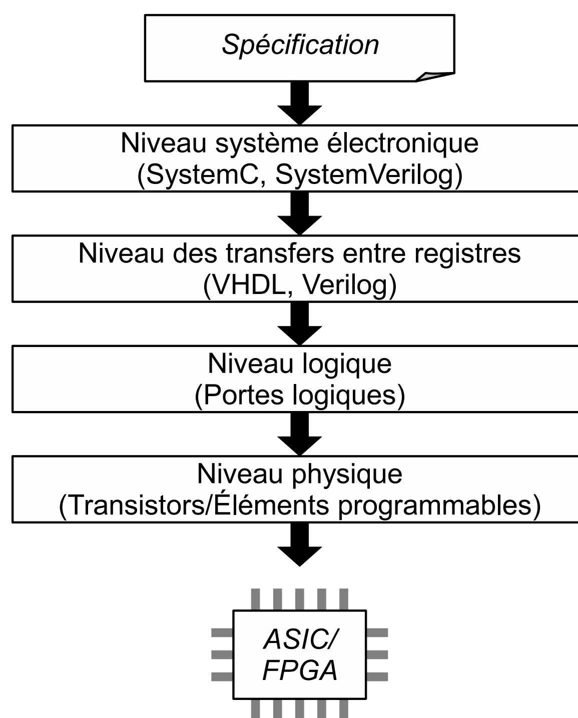


Figure 2.1 Niveaux d'abstractions pour la conception de circuits numériques.

L'automatisation du processus de raffinement d'une spécification produite à un plus haut niveau d'abstraction vers une implémentation physique bas-niveau est très avantageuse sur le plan de la productivité associée au développement de circuits numériques intégrés. De manière analogue au monde de la conception logicielle, la spécification d'une application à un niveau d'abstraction plus élevé permet une plus grande portabilité, et rend possible une optimisation automatisée de la spécification en ciblant différents objectifs de conception (surface, vitesse, et consommation de puissance). En masquant les détails d'implémentation des couches d'abstractions inférieures, un niveau d'abstraction supérieur rend également possible

un développement plus agile, et plus tolérant aux changements de spécifications tardifs. En effet, il est plus simple d'apporter un changement à un algorithme lorsque la spécification du circuit correspondant est produite à un niveau algorithmique plutôt qu'à un niveau physique. De plus, en déléguant la prise en charge des détails d'implémentation bas-niveaux à des outils de conception automatisée, on réduit significativement les sources d'erreurs possibles en plus de faciliter leur identification et leur correction, ce qui contribue également à réduire l'effort de développement d'un circuit intégré.

À l'état de l'art, bien qu'il soit possible de produire un circuit numérique depuis une spécification algorithmique au moyen des outils de synthèse haut-niveau, le développement de circuit numériques spécialisés est typiquement réalisé au niveau RTL. Les outils de synthèse haut-niveau permettent de produire rapidement un circuit spécialisé pour un algorithme donnée, mais de manière générale les implémentations résultantes ne sont pas comparables aux implémentations produites manuellement au niveau RTL. Dans le rapport sur la feuille de route des semi-conducteurs (ITRS) [9], cette incapacité des outils de synthèse haut-niveau à remplacer l'approche RTL comme niveau de description de circuit est formulée ainsi :

> *For instance, although behavioral synthesis is essential to system-level design, efficient behavioral synthesis is not yet realized today, despite having been a research topic for more than a decade, and despite recent advances driven by C- and SystemC-based synthesis and transaction level modeling (TLM) technologies.*

Cette situation représente un obstacle majeur à l'automatisation de la synthèse d'une spécification au niveau système ou algorithmique en une spécification au niveau RTL. La description de circuits spécialisés au niveau RTL requiert l'expertise des concepteurs de circuits dédiés, et n'est généralement pas accessible aux développeurs d'applications scientifiques ne disposant pas d'une telle expertise. De plus, le niveau d'abstraction RTL introduit dans les années 90 n'est plus adéquat pour composer avec les niveaux de complexité rendus possibles par les procédés de fabrications modernes, ce qui a une impact négatif sur la productivité associée au développement des circuits intégrés.

Ce chapitre présente une revue de l'état de l'art en matière de description et de synthèse de circuits numériques au-delà du niveau RTL. La section 2.2 porte un regard sur les plus récents avancements dans le vaste domaine de la synthèse comportementale haut-niveau. La section 2.3 fait un survol de différentes méthodes de description et de synthèse de circuits dédiés au-delà du niveau RTL.

2.2 Description comportementale et synthèse haut-niveau

Les premiers travaux de recherches dans de domaine de la synthèse comportementale datent des années 70, et ont réellement pris de l'importance à partir des années 80. Une analyse présentée dans [10] propose un survol historiques des différentes générations d'outils de synthèse à haut-niveau. On y rapporte notamment les principales raisons derrière les échecs des générations précédentes et les succès de la dernière génération. La synthèse comportementale haut-niveau fait généralement référence à la synthèse de programmes décrits en langage C ou un de ces dérivés (C++/SystemC/OpenCL). Cela s'explique notamment par le fait que les langages C/C++ ont longtemps été et demeurent encore aujourd'hui parmi les langages de programmations les plus connus et utilisés au sein de la communauté.

Le flot de conception de synthèse haut-niveau est caractérisé par 4 étapes importantes permettant la transformation d'un programme en description d'un circuit numérique :

1. *Capture du programme* : Le programme est capturé en une représentation interne. Les représentations internes courantes sont les graphes de flot de contrôle et de données (CFG, DFG, CDFG) et les graphes hiérarchiques de tâches (HTG). Différentes optimisations sont réalisées sur la représentation interne (propagation de constantes, élimination de code inutile, ...).
2. *Allocation* : Allocation de ressources (opérateurs et éléments de mémoire) permettant de supporter l'exécution des différentes opérations du programme. Différentes ressources peuvent supporter des opérations identiques.
3. *Ordonnancement* : Ordonnancement des opérations du programme de manière à optimiser le temps d'exécution ou la quantité de ressources utilisées. Optimisation des boucles itératives. Analyse des dépendances de données qui existent entre les différentes opérations du programme.
4. *Association des ressources et synthèse du contrôle* : Association des opérations à des ressources du circuit. La synthèse du contrôle implique la génération des circuits d'aiguillage (mux) de données et de séquenceurs supportant l'exécution des cédules associées aux différentes opérations.

Conjointement, les étapes d'allocation, d'ordonnancement, d'association des ressources, et de synthèse du contrôle réalisent la synthèse d'un circuit automatiquement optimisé afin de satisfaire différents objectifs en termes d'utilisation de ressources et de temps d'exécution. Pour une couverture plus exhaustive, [11] présente une introduction au flot de synthèse haut-niveau, tandis que [12] offre une couverture plus en profondeur de la théorie et des algorithmes supportant les optimisations architecturales des étapes 2 à 4.

À l'état de l'art, un effort de recherche important est toujours investi vers l'amélioration des résultats de synthèse haut-niveau des circuits numériques. Dans [13, 14, 15, 16], un riche ensemble de transformations visant la parallélisation des opérations est proposé. Les travaux dans [13, 17] explorent des transformations de code spéculatives. Dans [14], une technique de décalage de boucle (*loop shifting*) est présentée, permettant de décaler les opérations au-delà des frontières de chaque itération. Les travaux dans [18, 19] traitent de la synthèse comportementale de descriptions contenant des opérations sur des nombres utilisant une représentation à virgule flottante. Le travail dans [20, 21] s'intéresse à la synthèse de descriptions C contenant des pointeurs et des structures de données complexes. Le travail dans [22] s'intéresse à la synthèse comportementale ciblant des systèmes reconfigurables dynamiquement. Dans [23], une approche visant la génération d'architectures multi modes, intégré à l'outil de synthèse comportementale *GAUT*, est proposée.

L'analyse des accès mémoire par des pointeurs peut être utilisée afin de paralléliser davantage les opérations, et de générer des hiérarchies mémoires distribuées supportant une meilleure réutilisation des données. Dans [24], une approche permettant de partitionner les calculs et les tableaux vers différentes mémoires est présentée. Par analyse des empreintes des accès mémoires au travers des boucles, il est possible de réaliser des regroupements de partitions de tableaux dans différentes mémoires et de générer plusieurs unités de calculs de type contrôleur et chemin de données. L'approche proposée pour la génération de systèmes à mémoires distribuées, en plus de permettre des améliorations des temps d'exécution, permet également une réduction intéressante dans les tests comparatifs effectués avec un outil commercial partant de code C non transformé. Dans [25], une autre approche exposant au compilateur l'interaction entre le système mémoire et les unités de calculs est présentée. La méthodologie s'intéresse particulièrement aux accès à des tableaux dans des boucles imbriquées. Ainsi, en présence de réutilisation de données, le compilateur peut prévoir des mémoires tampons (embarqués) pour contenir des partitions de tableaux ce qui permet de réduire considérablement les temps d'accès. Le compilateur est également en mesure d'explorer la possibilité d'utiliser plusieurs mémoires parallèles afin de permettre une plus grande bande passante vers les unités de calcul. Dans [26], une approche consistant à utiliser des mémoires avec unités de calculs intégrés (CIM) dans la synthèse comportementale est proposée. L'idée derrière l'utilisation de telles mémoires est qu'elles offrent une meilleure bande passante que le système d'interconnexions (bus) à l'unité de calcul intégré et qu'elles permettent de réduire les communications au travers de ce même système d'interconnexions. Les résultats obtenus font état d'amélioration d'un facteur 2x en termes de performances et de réduction de la mesure énergie-délais pour des applications intenses en accès mémoires. Dans [27], une méthode (évalué dans le cadre de travail offert par l'outil *Cyber*) permettant de générer des architectures à mémoires

distribuées hétérogènes, ciblant les applications intensives en accès mémoires est proposée.

A l'ère de l'électronique mobile, les techniques d'optimisations pour la synthèse à haut-niveau de circuits numériques à faibles consommations de puissance font également l'objet de recherches actives. Dans [28], une approche basée sur l'utilisation de verrous (*latch*) est présentée. Les verrous possèdent plusieurs avantages en termes de surface, de puissance et de vitesse, mais ne peuvent pas être lus et écrits simultanément. La méthodologie proposée montre que l'utilisation d'une seule horloge est possible en modifiant légèrement le temps de vie des variables. Cette méthode est ensuite comparée à une approche basée sur l'utilisation d'horloge activable (*clock gating*), et des réductions de puissances allant de 39% à 65% sont obtenues avec des surfaces similaires. D'autres travaux [29] considèrent également le verrou comme élément mémoire de base pour la synthèse à haut niveau. En plus de réduire la puissance, les verrous sont ici également utilisés afin de réduire (potentiellement) la latence. Les algorithmes ont été intégrés au cadre de travail HLS-1, et les résultats de comparaison avec un outil de synthèse haut-niveau "conventionnel" rapportent des réductions de latence en moyenne de 18,2% en plus de réductions de surface et de puissance de 9,2% et 18,2% respectivement. Dans [30], le problème d'effets thermiques dans les circuits intégrés est considéré. La méthodologie consiste en un recuit simulé en 2 étapes combinant la minimisation de puissance et de la température du circuit. La méthodologie réussit systématiquement à trouver des implémentations avec des températures de pointe réduites. Les résultats, obtenus par comparaison avec un compilateur haut-niveau minimisant la puissance, font état d'une diminution de la température de pointe de l'ordre de 12% à 16% pour un surcoût de surface moyen de 15%. Les travaux présentés dans [31], poursuivant un objectif similaire, les étapes d'ordonnancement et de liaison (*scheduling* et *binding*) utilisent le retour de simulations thermiques afin de produire des circuits pour lesquels la température moyenne est réduite. Les résultats font état d'une réduction moyenne d'environ 7° C.

Avec chaque nouvelle génération de procédés de microfabrication, il est observé que le nombre de fautes dans les circuits intégrés augmente significativement. Dans ce contexte, les chercheurs se sont intéressés à tenir compte de cette nouvelle réalité dans le processus de synthèse comportementale. Dans [32], une approche permettant de tolérer des défauts au moyen de tissus reconfigurables est proposée. Brièvement, il s'agit de générer un ensemble de circuits réalisant une fonction, de sorte que par reconfiguration de ce circuit, on maximise les chances d'avoir une configuration fonctionnelle en présence de défauts. Dans [33], on propose d'utiliser de verrous transparents afin de tolérer le variations dues aux procédés lors de la synthèse comportementale. L'idée est de remplacer les bascules par des bistables transparents offrant la possibilité de propager du temps de jeu/lousse (*slack*) au niveau du chemin de données. Le travail présente une définition d'une mesure du ratio de design physiques respectant les

synchronismes initialement imposés, et la méthodologie rend possible une amélioration de ce ratio de l'ordre de 27%. Dans [34], ont traite du sujet de l'utilisation d'approches statistique afin de tenir compte des variations statistiques lors de la synthèse comportementale, et afin de rechercher des solutions maximisant la proportion de circuits physiques atteignant des objectifs de puissance ou de fréquence d'horloge.

2.2.1 Outils de synthèse haut-niveau C/C++/SystemC

Des décennies de recherche dans le domaine de la synthèse haut-niveau ont permis à de nombreux outils académiques et commerciaux de voir le jour. Cette section propose un survol des principaux outils de synthèse haut-niveau à l'état de l'art. Une revue plus exhaustive des différents outils et des possibilités qu'ils offrent à l'état de l'art est présenté dans [6].

Catapult (Mentor)

L'outil de synthèse à haut niveau *Catapult C* de Mentor en date de 2010 avait été nommé 3 ans de suite par *Gary Smith EDA* comme étant le leader du marché des outils de synthèse haut-niveau [35]. Traditionnellement, le langage ANSI C++ était la seule entrée possible de ce dernier, mais il supporte depuis des années le langage SystemC pour la modélisation de systèmes. L'outil de Mentor se démarque à ce niveau notamment avec le support d'un style de modélisation compatible avec l'approche TLM2.0. Catapult est présentement utilisé par des entreprises comme Qualcomm, STMicroelectronics, et AMD. Catapult C supporte un large sous-ensemble de la norme ANSI C++, mais il ne supporte pas l'allocation dynamique de mémoire et les pointeurs doivent pointer sur des structures statiquement définies comme des tableaux. Les appels de fonctions peuvent être mis en ligne (*inline*) selon la hiérarchie désirée, et les arguments de fonctions sont utilisés pour inférer les ports d'entrées-sortie [36]. Le compilateur est capable de pipeliner l'exécution de modules (fonctions) pouvant s'exécuter de façon concurrente et pipelinée. L'outil inclut également les types de données *bit-accurate* de la bibliothèque *Algorithmic C* [37], offrant des temps d'exécutions pouvant être de jusqu'à 100x meilleurs, ce qui réduit les temps de simulation. L'allocation de ressources et l'ordonnancement des opérations de l'outil sont dirigés par le choix technologique spécifié, ce qui augmente le potentiel de réutilisation et permet de recibler le design vers différentes technologies tout en produisant une implémentation optimisée.

Vivado HLS (Xilinx)

Afin d'enrichir son environnement de développement intégré de circuits FPGA, le fabricant Xilinx faisait l'acquisition de l'entreprise AutoESL et de son compilateur de synthèse haut-niveau *AutoPilot*. AutoPilot supporte un sous-ensemble des langages C/C++ et SystemC. Dans [38], l'implémentation d'un décodeur de profil simple MPEG-4 avec AutoPilot est discutée. Il est observé que le compilateur génère un peu plus de 10x lignes de VHDL par ligne de code source C pour chacun des modules utilisés. Le travail dans [39] présente l'implémentation d'un décodeur sphérique avec l'outil Autopilot HLS. Dans [40], l'outil Vivado HLS est appliqué à la synthèse d'algorithmes intensifs en calculs réalisant une technique d'apprentissage machine. Une comparaison avec des implémentations RTL écrites à la main montre que dans un cas des performances comparables peuvent être atteintes, alors que dans l'autre cas un différentiel significatif existe, de 30× pour un code non-optimisé à 4× pour un code optimisé.

Cynthesizer (Cadence)

L'outil de synthèse à haut niveau *Cynthesizer* de Cadence (acquisition de Forte Design Systems par Cadence en 2014) supporte les descriptions SystemC et C++. Un large sous-ensemble de SystemC/C++ est supporté pour la synthèse, sauf l'allocation dynamique de mémoire, l'arithmétique des pointeurs, le *déréférencement* de pointeurs pointant vers des régions non allouées statiquement (tableaux), et l'utilisation de méthodes virtuelles [41]. Bien que Cynthesizer supporte des descriptions de comportements purement séquentielles, l'outil permet également l'application de directives afin de guider les étapes d'ordonnancement et l'allocation d'éléments de mémoire. Dans la littérature, Cynthesizer a été intégré au sein de l'outil de conception ESL *SystemCoDesigner* dans une collaboration universitaire avec l'entreprise *Forte Design Systems* [42].

CyberWorkBench (NEC)

L'outil *CyberWorkBench* a été développé au courant des 20 dernières années dans les laboratoires de l'entreprise NEC. Cet outil supporte comme entrée le langage C. Un des arguments pour ce choix est que ce langage est typiquement utilisé pour décrire les applications dans les systèmes sur puces (SoC) contenant des processeurs embarqués. De plus, la simulation en C permet des temps d'exécution jusqu'à 100x meilleurs que ce qu'offre la simulation RTL. Un aspect intéressant de CWB est son outil de vérification formelle fortement couplé au synthétiseur [43]. La synthèse comportementale tient également compte de la technologie ciblée

par l'usager. Dans [44], l'outil CyberWorkBench est présenté brièvement dans un article rapportant des expériences avec l'utilisation de ce dernier. On y retrouve notamment le cas de conception d'une carte d'interface réseau (*NIC - Network Interface Card*) pour une grappe d'ordinateurs WindowsNT. La synthèse comportementale a alors été utilisée étant donnée la complexité du contrôle trop importante pour une description RTL, mais étant donnée la vitesse de transferts à 1,25Gbps, une approche de type *firmware* ne pouvant pas être utilisée non plus. La description résultante contient 23000 lignes de code, représentant plus de 20 processus communicants, pour un total de plus de 1000 états.

Synphony C Compiler (Synopsys)

En 2010, Synopsys faisait l'acquisition du développeur d'outil de synthèse haut-niveau Synfora pour renforcer sa position dans le domaine des outils de conceptions et de vérification de systèmes numériques. Synfora est l'entreprise derrière la suite *PICO* (*Program-In Chip-Out*), présenté dans [45]. Le langage d'entrée est un sous-ensemble de la norme ANSI C, mais cet outil a ceci de particulier que le processus de synthèse a recours à une *template* architecturale, nommée pipeline d'éléments de calculs vectoriels (*PPA - parallel processing array*). Le modèle d'exécution parallèle est un réseau de processus séquentiels de Kahn (KPN), bien adapté pour les applications qui travaillent avec des flux de données. Dans [46], une méthode permettant de spécifier à haut-niveau et de supporter les chemins multi-cycles (au sein du chemin de données) avec l'outil HLS Synphony C est présentée.

LegUp Compiler (Toronto University)

L'outil de synthèse haut-niveau *LegUp*, développé à l'Université de Toronto, est basé sur le projet d'infrastructure de compilateur LLVM (Low Level Virtual Machine). Le compilateur supporte différents modes, pouvant produire une solution entièrement matérielle (circuit spécifique à l'application), ou bien une solution hybride logicielle-matérielle intégrant un processeur d'instructions. Le travail dans [47] présente l'outil et propose une comparaison avec l'outil HLS eXCite (Y Explorations) en utilisant le benchmark CHStone.

2.2.2 Synthèse haut-niveau OpenCL

Parmi les plus récents développements dans le domaine de la synthèse haut-niveau se trouve l'utilisation du langage OpenCL comme description d'entrée. OpenCL est une structure logicielle à standard ouvert, spécifiant son propre langage, pour la programmation d'unité de traitements dans un environnement de calcul hétérogène pouvant intégrer des unités de trai-

tement central (CPU), des unités de traitement graphique (GPU), des réseaux de portes programmables in situ (FPGA), et autres. Le langage OpenCL supporte un modèle d'exécution permettant d'exprimer le parallélisme au niveau des données et au niveau des tâches. Il est largement utilisé pour la programmation d'applications scientifiques sur GPU, et plusieurs bibliothèques sont disponibles. Il est donc intéressant de considérer ce langage pour la programmation FPGA. En exprimant un niveau de parallélisme explicitement au niveau de la description logicielle, il est possible de réduire l'effort de parallélisation que doit réaliser l'outil de synthèse haut-niveau. Le travail dans [7] présente l'application du compilateur OpenCL HLS d'Altera à un algorithme de filtrage de documents. Dans [48], un compilateur OpenCL HLS est proposé, basé sur l'utilisation de l'outil de synthèse C/C++ AutoPilot (maintenant Vivado HLS).

2.3 Description et synthèse de circuits numériques au-delà du niveau RTL

En partant d'une description logicielle C/C++, la synthèse comportementale haut-niveau permet d'abstraire presqu'en totalité la complexité associée à l'expression du parallélisme et la description du circuit spécialisé. En contrepartie, à l'état de l'art, les outils de synthèse haut-niveau ne parviennent généralement pas à produire des circuits numériques pour lesquels les performances peuvent réellement rivaliser avec des implémentations RTL décrites manuellement. Face à cette situation, différentes approches sont proposées dans la littérature pour réduire la complexité associée à la description de circuits numériques au niveau RTL.

2.3.1 Bluespec SystemVerilog

Bluespec SystemVerilog est un langage de description de circuits centré sur les opérations. Bluespec permet la spécification de comportements au moyens d'actions atomiques gardées par des règles [49, 50]. La méthode de compilation, qui s'enracine dans la théorie des systèmes de réécriture de termes (*Term Rewriting Systems*, [51, 52]), permet de gérer la concurrence liée à différentes règles de réécriture de registres. Chacune de ces règles est décrite comme si elle était la seule à opérer à tout moment. Il est également possible de spécifier l'ordre dans lequel l'effet d'une règle est perçu par une autre règle lorsqu'elles sont activées simultanément. L'approche permet également un raffinement temporel graduel, les règles pouvant être raffinées en sous-règles plus fines pour supporter une plus grande concurrence à l'exécution. L'ordonnancement des règles peut également être dirigé par l'usager, sujet qui est discuté dans [53]. Dans [54, 55], le processeur BlueSPARC et son implémentation avec BlueSpec SystemVerilog (BSV) sont présentés. Le processeur BlueSparc est un processeur multi-fils pouvant supporter jusqu'à 16 fils, pouvant exécuter des applications commerciales destinées au pro-

cesseur UltraSPARC III. Dans [56], un processeur pipeliné permettant d'exécuter du code natif Java, implémenté avec BlueSpec, est également présenté. Dans [57], une comparaison entre Bluespec et un outil de synthèse C/C++ pour la synthèse d'un décodeur Reed-Solomon est présentée, rapportant que l'approche Bluespec permet de meilleurs résultats de synthèse tout en permettant une amélioration de la productivité.

2.3.2 Maxeler Compiler

Le compilateur Maxeler permet d'implémenter des engins de calculs sur des plateformes FPGA propriétaires. Ces plateformes peuvent être interconnectées entre-elles et avec un hôte de type ordinateur personnel au moyen de liens PCI-Express [58, 59]. Ce type d'architecture permet d'exécuter les régions non-critiques d'une application sur un unité de traitement central (CPU) et d'exécuter les régions critiques sur des processeurs spécialisés (FPGA). Le compilateur prend en entrée la description d'un graphe de flot de données (DFG), spécifiée au moyen d'un méta-programme basé sur le langage Java. Dans [60], l'approche est appliquée à l'implémentation d'un processeur spécialisé pour un algorithme de *Reverse Time Migration* (RTM). L'algorithme de *Reverse Time Migration* est un algorithme très intensif en calculs dans le domaine de la géophysique. Une comparaison entre l'implémentation de type processeur à flot de donnée spécialisé produite avec le compilateur Maxeler et une implémentation logicielle sur des processeurs multi-coeurs d'Intel rapporte une efficacité 10× plus élevée en termes de consommation de puissance. Dans [61], l'approche est appliquée à l'accélération des calculs dans le domaine de la mécanique moléculaire. L'engin de calcul à flot de données produit permet une amélioration des temps de calculs de l'ordre de 29× en comparaison avec un processeur d'Intel à 12 coeurs. En comparaison avec un processeur graphique basée sur l'architecture Fermi, des améliorations de temps d'exécution entre 2.5× et 4× sont rapportées.

2.3.3 Langages synchrones

Les langages synchrones Lustre et Esterel sont aussi utilisés pour la description et la synthèse de circuits numériques. Les langages synchrones sont basés sur un modèle mathématique facilitant l'analyse formelle des circuits et systèmes. Ils sont basés sur un modèle d'exécution concurrente et sont bien adaptés à la description de systèmes réactifs. Chacun de ces langages propose également des sémantiques claires pour la composition parallèle de deux processus concurrents. La composition parallèle de machines de Mealy peut mener à des relations associées des boucles combinatoires. Dans Lustre, de telles boucles combinatoires ne sont pas permises. Dans Esterel, les boucles combinatoires sont supportées pour autant que les rela-

tions associées soient équivalentes à des fonctions logique combinatoire. Une telle équivalence requiert que la relation cyclique ne permette pas plus d'un seul point-fixe à tout moment. La revue dans [62] présente une introduction aux langages synchrones et offre une présentation plus détaillée des langages Lustre et Esterel.

2.3.4 Machines séquentielles algorithmiques

La méthode de conception ASM (Algorithmic State Machine) a été proposée par Tom Osborne dans les années 60 à l'Université de Californie à Berkeley. Dans les années 70, HP avait largement adopté l'approche pour la conception des premiers calculateurs de bureau et microprocesseurs. En 1973, Chris Clare de HP Labs publie un livre sur la méthode ASM [63]. Les ASM permettent de capturer au moyen d'un graphe les conditions et les affectations concurrentes d'un algorithme. La Figure 2.2 illustre un exemple d'ASM pour l'algorithme de recherche du plus grand diviseur commun. Les rectangles représentent des états, les rectangles aux coins arrondis représentent des affectations concurrentes, les losanges représentent des conditions, et les arcs dirigés représentent le flot de contrôle. Dans [64] une variation de la notation ASM classique est proposée pour faciliter la capture de machines plus complexes. Dans [65], un langage de description d'ASM est proposé. Le langage comprend des directives permettant de spécifier des régions parallèles et séquentielles, des boucles de répétitions, et autres. Pour les régions séquentielles, une analyse de dépendances de données est réalisée afin de permettre l'optimisation de l'ordonnancement des opérations. L'ordonnancement est suivi par l'allocation et l'association des ressources.

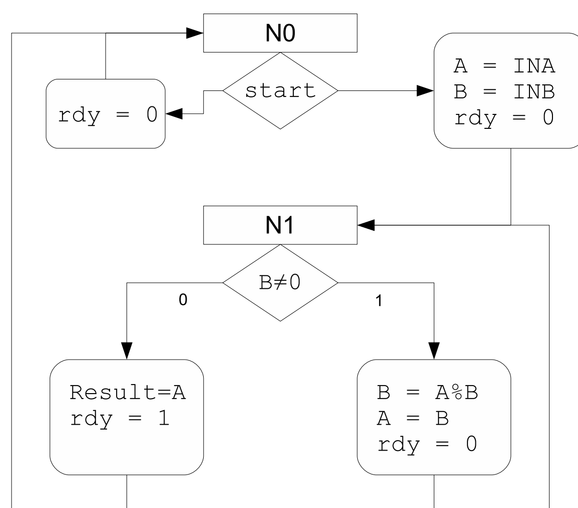


Figure 2.2 ASM pour l'algorithme de recherche du PGDC.

2.3.5 Langage de niveau intermédiaire CASM

Le langage de niveau intermédiaire CASM a été proposé dans [66, 67] pour supporter la description d'ASM contrôlant des connexions entre des sources et des puits synchronisés par les données. Cette approche élève également le niveau d'abstraction offert par la méthode ASM en supportant la description de connexions bloquantes (=) et non-bloquantes ($* =$). Une connexion bloquante vient bloquer le flot de contrôle d'une ASM jusqu'à ce qu'un transfert de donnée ait lieu. Les sources et les puits possèdent des interfaces prédéfinies de type flot de données. Ces interfaces possèdent des signaux de synchronisation de type prêt-à-envoyer/prêt-à-recevoir. Un transfert de donnée a lieu sur une connexion lorsque la source et le puits sont tous deux prêts. Ce type de protocole de synchronisation est déjà supporté par les interfaces AXI-Stream et Avalon Streaming, et est utilisé par de nombreux modules de propriété intellectuelle. La Figure 2.3 illustre l'opération d'un protocole de synchronisation complète. Un transfert de donnée a lieu à chaque cycle d'horloge pour lequel les signaux de synchronisations RTS et RTR sont tous deux vrais. Le langage CASM supporte également les appels et retours d'états, au moyen d'une pile d'état, ce qui permet la description de fonctions récursives.

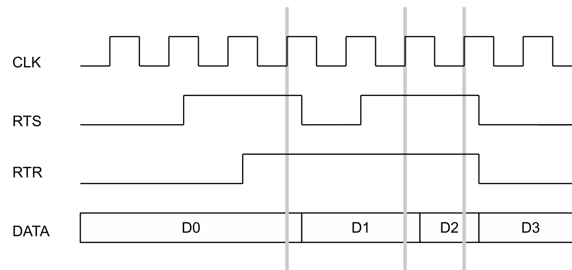


Figure 2.3 Protocole de synchronisation complète.

La Figure 2.4 présente une description CASM de l'algorithme de recherche du plus grand diviseur commun. Les entrées et sorties *INA*,*INB*,et *RESULT* ont des interfaces complètement synchronisées. Dans l'état *N0*, l'ASM attend la réception des deux arguments de la fonction. Dans l'état *N1*, lorsque le test sur la valeur contenue dans le registre *B* est vrai, l'opération $A\%B$ est réalisée, autrement le résultat contenu dans le registre *A* est envoyé à la sortie. Le retour à l'état *N0* a lieu après que le résultat soit envoyé à la sortie. Dans [68], le langage CASM a été utilisé pour la description d'un opérateur de multiplication sur des grands entiers basé sur la transformée de Fourier rapide (FFT).

```
input INA[32],INB[32] {protocol="FS"};
output RESULT[32] {protocol="FS"};
register A[32],B[32];
device mod {type="modulo_op",width=32};

ASM gcd {

    N0:
        A = INA;
        B = INB;
        goto N1;
    N1:
        if( B != 0 )
            mod.dina = A;
            mod.dinb = B;
            B = mod.dout;
            A = B;
            return N1;
        else
            RESULT = A;
            goto N0;
        end;
}
```

Figure 2.4 Description CASM pour l'algorithme de recherche du PGDC.

2.4 Conclusion

L'évolution des outils d'automatisation de la conception des circuits numériques est étroitement liée à celle des circuits numériques intégrés. Afin de composer avec l'augmentation de complexité des circuits intégrés, différents niveaux d'abstraction de conception ont été introduits au flot de conception. Partant d'un niveau de description physique, le niveau logique, le niveau transfert de registres, et le niveau système sont venus s'ajouter au flot. Les outils de conception assistée par ordinateur permettent d'automatiser la traduction/synthèse des descriptions d'un niveau d'abstraction supérieur vers un niveau inférieur. À l'état de l'art, la synthèse automatisée des descriptions au niveau système (C/C++/SystemC/SystemVerilog) vers le niveau RTL passe par la synthèse comportementale haut-niveau. Après des décennies de recherche dans le domaine de la synthèse haut-niveau, les outils académiques et commerciaux permettent aujourd'hui de produire rapidement des circuits numériques partant de descriptions algorithmiques C/C++/SystemC, mais les résultats de synthèse ne sont généralement pas comparables aux résultats atteints avec des descriptions RTL produites manuellement [9]. Or le niveau RTL est mal adapté pour gérer la complexité offerte par les circuits intégrés modernes. Dans des travaux plus récents, on observe la synthèse haut-niveau se tourner vers le langage de traitement parallèle OpenCL [7, 48]. D'autre approches visent également à élever le niveau d'abstraction au-delà du niveau RTL tout en permettant l'expression de processus concurrents et/ou la description d'un circuit, telles les approches Bluespec et Maxeler, ainsi que des approches basées sur la méthode de conception de ma-

chine séquentielles algorithmiques (ASM). Bien que de telles approches n'offrent pas le même niveau d'abstraction qu'une approche de synthèse haut-niveau partant d'une description algorithmique séquentielle, ainsi que la même simplicité et vitesse de développement, en offrant une plus grande liberté de conception elles permettent d'obtenir un meilleur compromis en termes de qualité d'implémentation par rapport au résultats atteignables avec une descriptions RTL produite manuellement. À ce jour, la question de savoir dans quelle mesure les langages C/C++ sont appropriés pour la description de circuits numériques demeure ouverte, et est toujours sujet de différents débats. Parmi les questions ouvertes, citons la recherche qui vise à déterminer la forme que prendrait le prochain langage de description de circuits numériques, et quelles abstractions devraient-il contenir ? Comment serait-il possible de traduire un tel langage automatiquement en une description RTL ? Nous pouvons postuler que ces questions continueront de stimuler l'esprit des chercheurs de la communauté tant qu'une solution viable de remplacement à la description au niveau RTL n'aura pas été établie et reconnue, car la description de circuits au niveau RTL entraîne un coût de productivité toujours plus important avec chaque nouvelle génération de circuits intégrés.

CHAPITRE 3 DESCRIPTION DE CIRCUITS AVEC LE LANGAGE CASM+

3.1 Introduction

Dans ce chapitre, nous présentons les différents ajouts apportés au langage CASM. Le premier ajout important est la possibilité de spécifier des règles d'autorisation des transferts de données. Les boucles combinatoires induites par ces règles d'autorisation seront résolues automatiquement par le compilateur afin de maximiser le nombre de transferts de données qui sont réalisés à chaque cycle d'horloge. Les deuxième et troisième ajouts consistent en des opérateurs de connexions avancés, spécialisés pour la description d'architectures pipelinés. Pour chacun de ces opérateurs, nous présentons les sémantiques associées, et nous proposons une méthode de traduction vers une description n'utilisant que des opérateurs de base du langage CASM (bloquants et non-bloquants).

3.2 Règles d'autorisation de transferts de données

Les règles d'autorisation permettent de contraindre les transferts de données associés à des connexions entre des sources et des puits avec des interfaces synchronisées par les données. En contraignant l'autorisation des transferts de données au moyen de règles, il est possible de décrire des comportements de synchronisation, de priorité, ou d'ordonnancement des transferts sur des connexions actives. L'activation des connexions est déterminée par les énoncés conditionnels ($if, else, switch, case$) traditionnels, tandis que l'autorisation des transferts est contrainte par les règles d'autorisation. Ces règles sont spécifiées au moyen de règles d'implications de la forme $t_i => guard_k(t_i)$, ou t_i représente l'identifiant d'une connexion, et $guard_k(t_i)$ représente une règle s'appliquant aux transferts de données sur cette connexion. Ensemble, les règles d'autorisation sont combinées pour former l'équation d'autorisation d'une connexion tel qu'exprimée par l'Équation 3.1, où $t_i.active$ est un signal indiquant si la connexion t_i est active.

$$t_i.authorize = t_i.active \wedge (\bigwedge_{k=1}^{K} guard_k(t_i)) \quad (3.1)$$

Les règles d'autorisation des transferts de données peuvent être spécifiées en fonction de différents attributs associés à chaque connexion. La Table 3.1 résume l'ensemble des attributs disponibles. L'attribut *active* spécifie que la connexion est active. L'attribut *available* (dis-

ponible) spécifie que la source et le puits d'une connexion sont tous deux prêts, en fonction des signaux de synchronisation des interfaces à flot de données. L'attribut rtf spécifie si la connexion est à la fois active et disponible. L'attribut $fire$ spécifie qu'un transfert de données a lieu entre la source et le puits de la connexion associée. Dans le cas d'une connexion bloquante, l'attribut *done* (fait) spécifie que la connexion correspondante a alloué un transfert de donnée depuis son activation. L'attribut *complete* (complété) correspond à la disjonction des attributs $fire$ et *done*. Lorsque les règles d'autorisation forment des relations cycliques en termes des signaux de synchronisation des interfaces à flot de données ou des attributs d'autorisations, le compilateur peut résoudre ces dépendances cycliques en un réseau acyclique de fonctions logiques qui maximise le nombre de transferts de données à chaque cycle d'horloge. Plus particulièrement, les dépendances cycliques avec un seul ou plusieurs point-fixes sont supportées.

Tableau 3.1 Attributs de connexions

Attribut	Description
active	Signal d'activation généré par une ASM.
available	La source et le puits d'une connexion sont prêts.
rtf	La connexion est prête à permettre un transfert ($active \wedge available$).
authorize	Conjonction des règles associées à une connexion.
fire	Un transfert de donnée a lieu sur la connexion ($rtf \wedge authorize$).
done	Une connexion bloquante a permis le transfert d'une donnée.
complete	Disjonction des attributs *fire* et *done* ($done \vee fire$).

La Figure 3.1 donne un exemple illustrant l'utilisation de règles pour contraindre l'autorisation de transferts de données sur un ensemble de connexions actives. Les règles (lignes 3 et 4) s'appliquant aux connexions $t1$ et $t2$ spécifient que l'autorisation de transferts sur celles-ci doit être simultanée. La connexion $t1$ attend qu'un transfert ait lieu sur $t2$ pour procéder au transfert, et inversement la connexion $t2$ attend qu'un transfert ait lieu sur $t1$ pour procéder au transfert. Bien que cela induise une boucle avec deux états possibles, l'approche de synthèse proposée dans cette thèse permet la résolution de celle-ci en un circuit de contrôle acyclique et de telles règles sont permises. La règle (ligne 7) sur la connexion $t4$ spécifie une contrainte d'ordonnancement requérant que $t4$ puisse autoriser un transfert de donnée uniquement après que la connexion $t3$ en ait autorisé un. En effet, il faut avoir $t3.done$ pour que $t4$ puisse procéder au transfert d'une donnée.

3.3 Opérateurs de connexion différée

Cette section présente les opérateurs de connexion différée proposés dans cette thèse. L'opérateur de connexion différée est conçu spécialement pour réduire la complexité associée à la

```
1:  {t1} x = a;
2:  {t2} y *= b;
3:  t1 => t2.fire;
4:  t2 => t1.fire;
5:  {t3} u = c;
6:  {t4} v = c;
7:  t4 => t3.done;
```

Figure 3.1 Exemple d'utilisation de règles pour contraindre l'autorisation de transferts de données sur un ensemble de connexions actives.

description de machines séquentielles algorithmiques interconnectant des sources et des puits d'opérateurs pipelinés. Dans un premier temps, les opérateurs et leur sémantique sont introduits au moyen de différents exemples. Dans un second temps, le processus de traduction automatisée de ces opérateurs par notre compilateur est présenté. Ce processus implique notamment de remplacer les opérateurs de connexion différée par des opérateurs de connexions de base du langage CASM.

3.3.1 Motivation

La description d'ASM interconnectant des opérateurs pipelinés est sensiblement plus complexe que lorsque des composant combinatoires ou non-pipelinés sont utilisés. La Figure 3.2 illustre un exemple simple dans lequel une ASM réalise un produit scalaire entre deux vecteurs à 3 dimensions. La description présentée du produit scalaire permet de saisir facilement le comportement désiré. En assumant que l'opérateur de multiplication est combinatoire, la boucle itérative peut s'exécuter à raison d'une itération par cycle d'horloge. Cependant la situation est bien différente lorsque l'on considère que l'opérateur de multiplication est pipeliné. Dans ce cas, la boucle itérative s'exécute à raison d'une itération à tous les L_{MUL} cycles, où L_{MUL} est la latence de l'opérateur de multiplication pipeliné. Bien qu'il soit possible de transformer la description de la boucle illustrée afin de supporter un débit d'une itération par cycle, une telle transformation alourdit significativement la description algorithmique du comportement désiré. L'opérateur de connexion différée permet de décrire un tel comportement pipeliné, mais avec une syntaxe similaire à celle qui serait utilisée en présence d'opérateurs combinatoires.

3.3.2 Sémantique

L'opérateur de connexion différée permet de gérer plus facilement la description d'ASMs contrôlant des sources et des puits issus de différents domaines temporels d'un chemin de données pipeliné. Le langage supporte des opérateurs de connexion différée de type bloquant

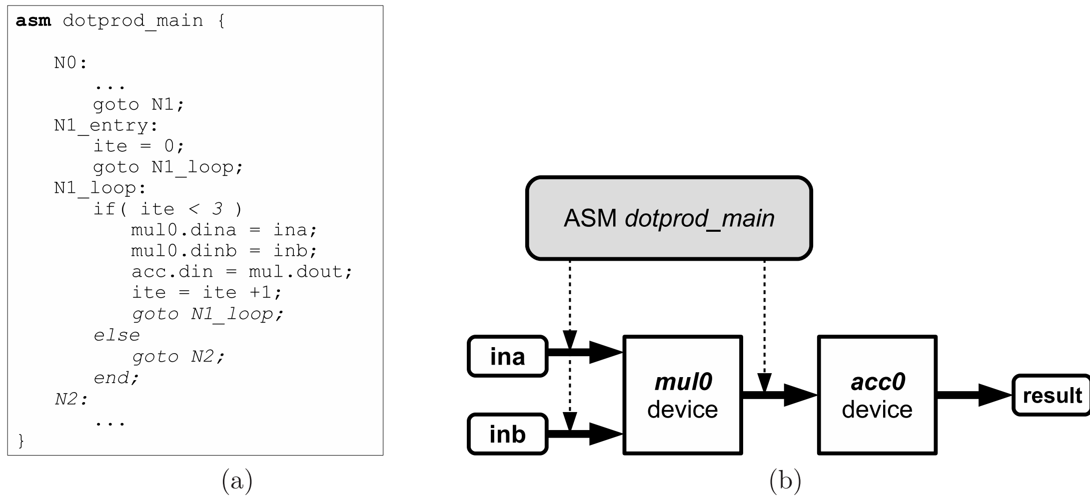


Figure 3.2 Produit scalaire sur deux vecteurs à 3 dimensions. a) Description CASM. b) Architecture cible.

et non-bloquant, dénotés par les symboles **?=** et **?*=** respectivement. Sémantiquement, une connexion différée entre une source A et un puits X dans une ASM asm_0 se traduit par l'émission d'une requête, à une ASM implicite asm_0^*, qui permet de réaliser une connexion entre la source A et le puits X. Les requêtes de connexion différées sont mémorisées dans des mémoires tampons (FIFO) jusqu'à temps qu'elles puissent être exécutées par l'ASM implicite asm_0^*. Une requête de connexion différée est exécutée en allouant un transfert de donnée sur la connexion correspondante. Pour un opérateur de connexion différée bloquante, la requête faite par l'ASM asm_0 est bloquante. Lorsqu'une mémoire tampon impliquée dans une requête bloquante est pleine, le flot de contrôle de l'ASM asm_0 est bloqué jusqu'à ce qu'un espace se libère. Pour un opérateur de connexion différée non-bloquante, la requête correspondante n'est pas bloquante. Dans ce cas, lorsque la mémoire tampon est pleine, le flot de contrôle de l'ASM asm_0 n'est pas bloqué et la requête n'est pas émise. La Figure 3.3 donne une illustration de l'utilisation de l'opérateur de connexion différée bloquante dans l'ASM *dotprod_main* de la Figure 3.2. Dans cette nouvelle description, la connexion bloquante qui envoit le résultat de l'opérateur de multiplication vers l'accumulateur est remplacée par une connexion différée bloquante. Cet opérateur de connexion se traduit par une requête de connexion qui peut être émise au même moment que les opérandes sont envoyées à l'entrée de l'opérateur de multiplication. Cette requête est envoyée vers une ASM implicite *dp_slv* qui se chargera de réaliser la connexion entre l'opérateur de multiplication et l'accumulateur, dès qu'il sera possible de réaliser le transfert d'une donnée.

La Figure 3.4 illustre l'application de l'opérateur de connexion différée pour gérer des transferts de données en utilisant un opérateur pipeliné à un seul opérande. Dans cet exemple,

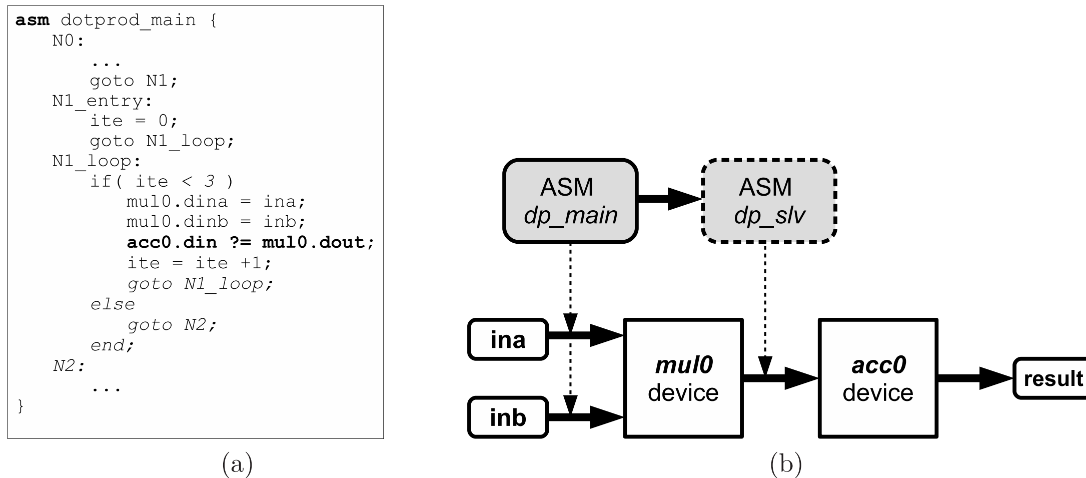


Figure 3.3 Produit scalaire sur deux vecteurs à 3 dimensions utilisant un opérateur de connexion différée. a) Description CASM. b) Architecture cible avec ASM implicite.

le port de lecture d'une mémoire RAM à double-port et de latence non-nulle est utilisé. En utilisant l'opérateur de connexion bloquante de base, il est possible de spécifier simplement de multiples opérations de lecture tel qu'illustré en Figure 3.4a. Cependant, tout comme dans l'exemple de la Figure 3.2, ce style de description entraîne une sous-utilisation des ressources pipelinées. En effet, il faut attendre que la valeur soit lue à l'état $N0$ ($X = mem.rddata$) avant de pouvoir passer à l'état $N1$ qui commande une deuxième lecture ($mem.rdaddr = 1$). En assumant une latence de 1 cycle, le style de description de la Figure 3.4b permet de pipeliner efficacement les deux opérations de lecture. En effet, dans ce cas pendant que la première valeur est lue à la sortie de la mémoire ($X = mem.rddata$), une deuxième lecture est commandée ($mem.rdaddr = 1$). En contrepartie, la description est alourdie du fait que la spécification de chaque opération de lecture, composée de deux connexions, est maintenant séparée dans différents états consécutifs. Pour conserver son efficacité, cette description devra être révisée si la latence de la mémoire est augmentée. La Figure 3.4c montre comment l'utilisation de l'opérateur de connexion différée vient simplifier la description de transferts issus de différents domaines temporels d'un pipeline. La description est pratiquement identique à celle de la Figure 3.4a, mais elle permet d'obtenir la même efficacité que la description de la Figure 3.4b. De plus, la description est insensible à la latence de l'opérateur utilisé. La Figure 3.4d montre comment cet opérateur permet de décrire simplement des macros (macro utilisé au même sens qu'en langage C) spécifiant plusieurs connexions dans différents étages d'un chemin de données pipeliné.

La Figure 3.5 illustre un deuxième exemple d'application de l'opérateur de connexion différée. L'exemple consiste en une boucle itérative réalisant N opérations au moyen d'un opérateur

```
N0:
    mem.rdaddr = 0
    X = mem.rddata;
    goto N1;

N1:
    mem.rdaddr = 1;
    Y = mem.rddata;
    goto N2;
N2:
    ...
```

(a)

```
N0:
    mem.rdaddr = 0
    goto N1;
N1:
    mem.rdaddr = 1;
    X = mem.rddata;
    goto N1a;
N1a:
    Y = mem.rddata;
    goto N2;
N2:
    ...
```

(b)

```
N0:
    mem.rdaddr = 0
    X ?= mem.rddata;
    goto N1;
N1:
    mem.rdaddr = 1;
    Y ?= mem.rddata;
    goto N2;
N2:
    ...
```

(c)

```
N0:
    Memread(mem,0,X)
    goto N1;
N1:
    Memread(mem,1,Y)
    goto N2;
N2:
    ...
```

(d)

Figure 3.4 Application de l'opérateur de connexion différée pour la description d'une opération de lecture sur une mémoire pipelinée.

pipeliné à un seul opérande dénoté $op1$ et de latence L_1. La boucle itérative décrite dans la Figure 3.5a est pipelinée par décomposition en trois phases différentes. Dans un premier temps, il faut remplir le pipeline de l'opérateur $op1$, dans un deuxième temps il faut continuer de remplir le pipeline de l'opérateur $op1$ et récupérer les résultats à sa sortie, puis finalement, dans un troisième temps il faut uniquement récupérer les derniers résultats à sa sortie. De manière générale, considérant une macro-opération constituée de P opérateurs pipelinés consécutifs et en assumant $N > P$, la structure conditionnelle de la boucle doit être décomposée en $2 \times P + 1$ cas différents, ce qui contribue à obfusquer la description de l'algorithme à réaliser. Dans la description de la Figure 3.5b, l'utilisation de l'opérateur de connexion différée permet de ramener la description de la boucle au niveau de simplicité associé à l'utilisation d'opérateurs combinatoires. La Figure 3.5c montre comment il devient possible de spécifier plus clairement et simplement le comportement de l'ASM au moyen de macros.

La Figure 3.6 illustre un troisième exemple d'application de l'opérateur de connexion différée. Dans cet exemple, la description spécifie l'exécution de deux boucles itératives consécutives. La première boucle $BOUCLE1$ réalise N opérations au moyen d'un opérateur pipeliné à un seul opérande dénoté $op1$ et de latence L_1. La deuxième boucle $BOUCLE2$ réalise N opérations au moyen d'un opérateur pipeliné à un seul opérande dénoté $op2$ et de latence L_2. On considère $L_2 > L_1$. Le problème illustré dans la description à la Figure 3.6a est qu'il

```
L0:
   if(count < LAT-1)
      op1.din = A0;
      count++
      goto L0;
   elsif(count < N-LAT)
      op1.din = A0;
      X0 = op1.dout;
      count++
      goto L0;
   elsif(count < N)
      X0 = op1.dout;
      count++
      goto L0;
   else
      goto N1;
   end;

N1:
   ...
```

(a)

```
L0:
   if( count < N )
      op1.din = A0;
      X0 ?= op1.dout;
      count++
      goto L0;
   else
      goto N1;
   end;

N1:
   ...
```

(b)

```
L0:
   if( count < N )
      DO_OP(op1,X0,A0);
      count++
      goto L0;
   else
      goto N1;
   end;

N1:
   ...
```

(c)

Figure 3.5 Utilisation de l'opérateur de connexion différée pour la description de boucles itératives faisant appel à des opérateurs pipelinés.

faut maintenant démarrer les opérations ($op2$) de la boucle $BOUCLE2$ à même le corps de la description associée à la boucle $BOUCLE1$. Autrement, la boucle $BOUCLE2$ serait démarrée avec L_1 cycles de retard, augmentant la latence liée à l'exécution du design. De plus la description est sensible à la latence des opérateurs et n'aura pas le même comportement si $L_2 < L_1$. En comparaison, la description avec l'opérateur de connexion différée est plus concise et permet d'abstraire la latence des opérateurs. Il n'est alors pas requis de spécifier les opérations de la boucle $BOUCLE2$ à même la description de la boucle $BOUCLE1$ occasionné par le chevauchement de l'exécution des deux boucles.

3.3.3 Ordonnancement

Le mécanisme d'émission et de traitement des requêtes de connexion différée est responsable de maintenir la cohérence entre l'ordre d'exécution des requêtes d'accès aux sources et aux puits et l'ordre d'exécution de ces requêtes. Des dépendances d'ordonnancement existent entre des connexions différées ayant des sources et/ou des puits identiques. Par exemple, si une requête pour $x\,?{=}b$ est faite, suivi d'une requête pour $y\,?{=}b$, le mécanisme de traitement des requêtes doit s'assurer de réaliser la connexion $x{=}b$ avant de réaliser la connexion $y{=}b$. Il n'est pas possible de faire simultanément des requêtes d'accès à des sources/puits identiques.

3.3.4 Transformation

Cette section présente le processus de transformation proposé d'une description contenant des opérateurs de connexion différées bloquants et non-bloquants ($? =$ et $?* =$) en une description

```
BOUCLE1:
   if(count < LAT1-1)
      op1.din = A0;
      count++
      goto BOUCLE1;
   elsif(count < N-LAT1)
      op1.din = A0;
      X0 = op1.dout;
      count++
      goto BOUCLE1;
   elsif(count < N)
      X0 = op1.dout;
      op2.din = A0;
      count++
      goto BOUCLE1;
   else
      goto BOUCLE2;
   end;

BOUCLE2:
   if(count < (LAT2-LAT1)-1)
      op2.din = A0;
      count++;
      goto BOUCLE2;
   elsif(count < 2*N-LAT2)
      op2.din = A0;
      X0 = op2.dout;
      count++;
      goto BOUCLE2;
   elsif( count < 2*N )
      X0 = op2.dout;
      count++;
      goto BOUCLE2;
   else
      goto N1;
   end;

N1:
   ...
```

(a)

```
BOUCLE1:
   if( count < N )
      op1.din = A0;
      X0 ?= op1.dout;
      count++
      goto BOUCLE1;
   else
      goto BOUCLE2;
   end;

BOUCLE2:
   if( count < 2*N )
      op2.din = A0;
      X0 ?= op2.dout;
      count++
      goto BOUCLE2;
   else
      goto N1;
   end;

N1:
   ...
```

(b)

```
BOUCLE1:
   if( count < N )
      DO_OP(op1,X0,A0);
      count++
      goto BOUCLE1;
   else
      goto BOUCLE2;
   end;

BOUCLE2:
   if( count < 2*N )
      DO_OP(op2,X0,A0);
      count++
      goto BOUCLE2;
   else
      goto N1;
   end;
N1:
   ...
```

(c)

Figure 3.6 Utilisation de l'opérateur de connexion différée pour la description de deux boucles itératives consécutives faisant appel à des opérateurs pipelinés.

ne contenant que des transferts de base ($=$ et $*=$). Le processus de transformations implique de remplacer les connexions différées de l'ASM source par des requêtes de connexion dans des tampons mémoire de type FIFO. Ces tampons mémoires sont ensuite lus par une autre ASM implicite, qui exécute les connexions différées entre les source et puits requis.

Les Figures 3.7 et 3.8 illustrent au moyen d'un exemple simple le processus de traduction des descriptions contenant des opérateurs de connexion différée. Dans la Figure 3.7a, la description d'une ASM spécifiant l'utilisation de connexion différée est présentée. La Figure 3.7b présente cette description après la transformations des connexions différée en requêtes de connexion. Pour chaque ensemble de sources et de puits connectés, un identifiant unique sera attribué à chaque source, et de même pour chaque puits. De même, pour chaque ensemble de sources et de puits connectés, il y aura jusqu'à une FIFO de requête pour chaque source et puits de l'ensemble. Une requête de connexion différée p_j ?=s_i est complétée en écrivant à la fois l'identifiant de la source s_i dans la FIFO de requête du puits p_j, et l'identifiant du puits p_j dans la FIFO de requête de la source s_i. Une règle d'autorisation est également ajoutée afin de synchroniser ces deux transferts. Pour une connexion différée bloquante la requête associée est bloquante ($=$), tandis que pour une connexion différée non-bloquante la requête

associée est non-bloquante ($*=$).

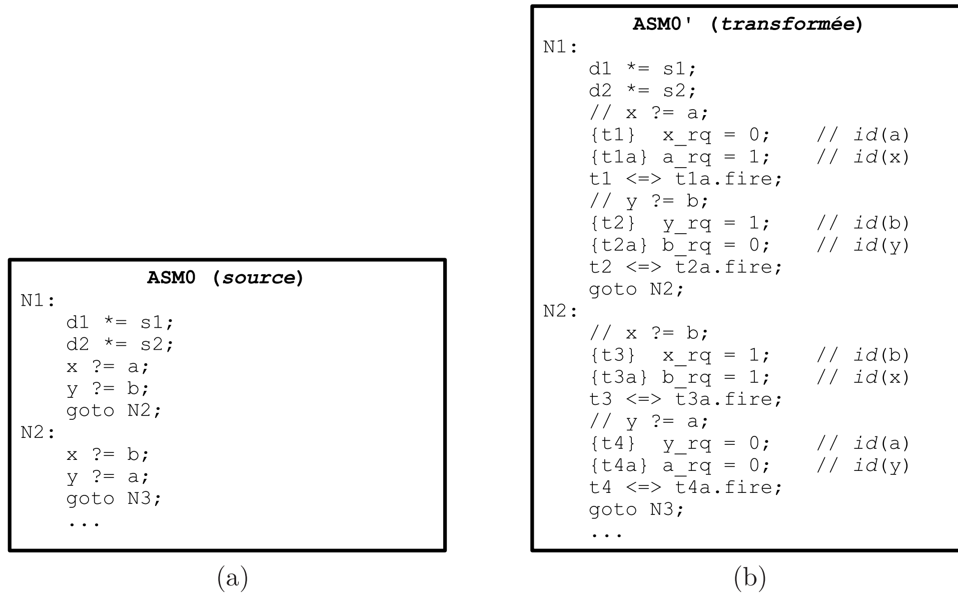


Figure 3.7 Transformation d'une ASM incluant des opérateurs de connexion différée, en une ASM qui émet des requêtes de connexions dans des FIFO. a) Exemple d'ASM source. b) ASM transformée.

La figure 3.8a illustre l'architecture cible pour une ASM réalisant des connexions différées. Cette architecture inclue l'ASM source modifiée, des FIFOs pour mémoriser les requêtes, et l'ASM implicite responsable d'exécuter les requêtes de connexion différée. De manière générale, pour chaque ensemble de sources et de puits connectés, il y aura jusqu'à une FIFO de requête pour chaque source et puits de l'ensemble. Cependant, lorsque pour un ensemble de sources et de puits connectés, une source peut envoyer à un seul puits ou qu'un puits peut recevoir d'une seule source, alors la source ou le puits respectivement n'ont pas besoin de FIFO de requête. Dans le cas ou un ensemble de sources et de puits connectés ne contient qu'une seule source ou un seul puits, une seule FIFO de requête est requise pour l'ensemble. Dans le cas particulier ou un ensemble de sources et de puits connectés ne contient qu'une seule source et qu'un seul puits, une seule FIFO de réservation est requise pour l'ensemble.

La figure 3.8b présente la description associée à l'ASM implicite responsable d'exécuter les requêtes de connexion différée. Une requête de connexion différée $p_j\,?{=}s_i$ peut être exécutée lorsque la sortie de la FIFO de requête du puits p_j correspond à l'identifiant de la source s_i, et que la sortie de la FIFO de requête de la source s_i correspond à l'identifiant du puits p_j. Les identifiants seront lus des FIFO correspondantes de manière synchronisée avec l'exécution d'un transfert de donnée sur la connexion $p_j\,?{=}s_i$. La description de l'ASM implicite responsable d'exécuter les requêtes de connexions différées est décrite comme une composi-

tion d'autant d'ASMs qu'il y a de FIFO de requête dans le mécanisme de traitement des connexions différées.

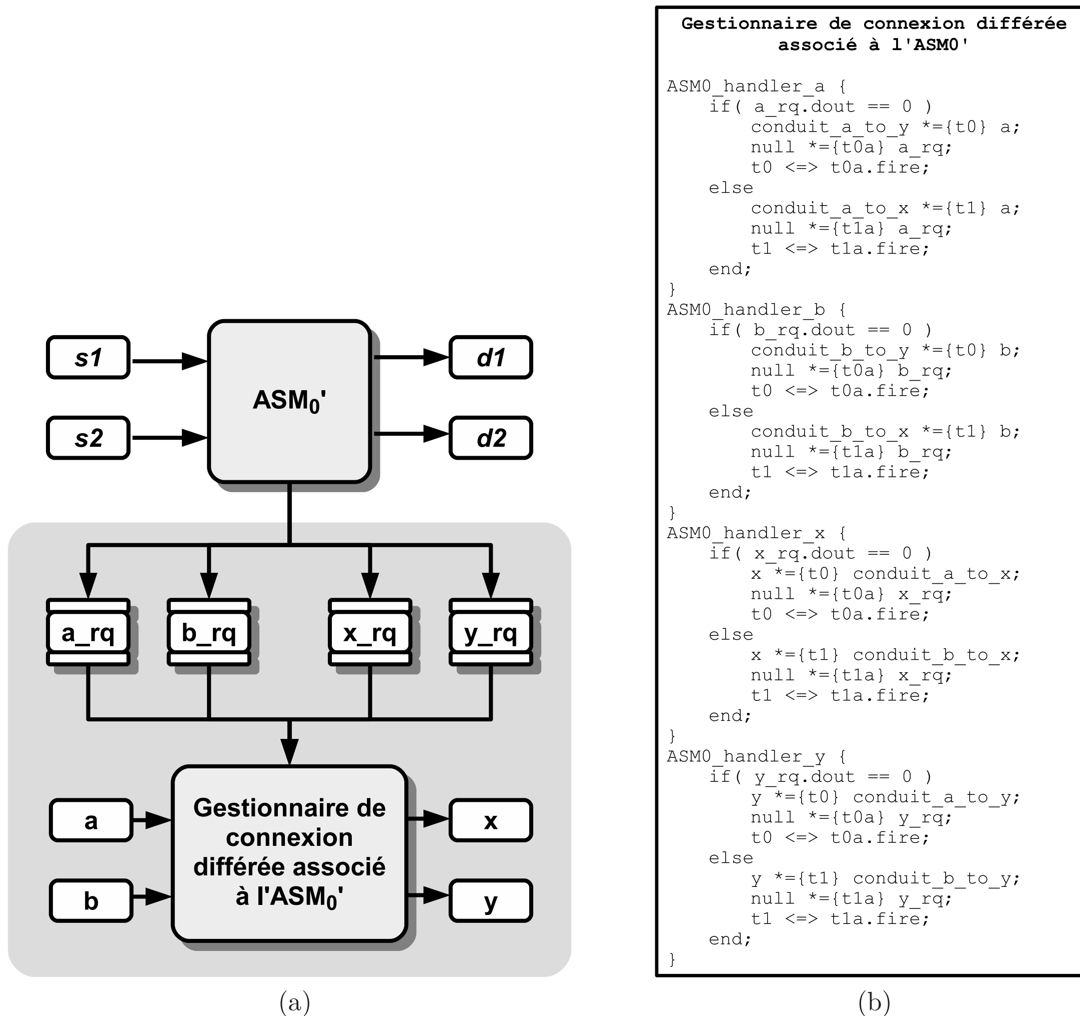


Figure 3.8 Sémantique pour les opérateurs de connexion différée. a) Circuit correspondant à l'ASM source de la Figure 3.7, illustrant l'ASM source, les FIFO de requête, ainsi que l'ASM implicite. b) Description de l'ASM implicite, responsable de traiter les requêtes de connexion, par une collection d'ASM.

3.4 Opérateurs de connexion sur réseau d'interconnexions pipeliné

Cette section présente les opérateurs de connexion sur réseau d'interconnexions pipeliné proposés dans cette thèse. Les opérateurs de connexion sur réseau d'interconnexions pipeliné sont conçus pour réduire la complexité associée à l'interconnexion d'un nombre important de sources et de puits au moyen de réseau d'interconnexion pipelinés.

3.4.1 Motivation

L'interconnexion d'un nombre important de sources et de puits dans différentes topologies au moyen d'un réseau d'interconnexion purement combinatoire (0 cycles de latence) peut avoir un impact négatif sur la fréquence maximale d'opération de l'implémentation finale. Il est possible de spécifier manuellement l'utilisation d'un réseau d'interconnexion pipeliné, mais cela vient alourdir la description par rapport à l'utilisation de connexions point-à-point de base. En effet, pour connecter une source s_i vers un puit p_j en passant par un réseau d'interconnexion, il faut connecter la source s_i au réseau et spécifier une destination cible en utilisant un identifiant associé au puits p_j. Les opérateurs de connexion sur réseau d'interconnexion pipeliné viennent abstraire ce niveau de complexité en permettant la spécification des connexions sur réseau pipeliné avec une syntaxe similaire à celle utilisée pour spécifier des connexions point-à-point de base ($=, * =$).

3.4.2 Sémantique

Les opérateurs $| = |$ et $|* = |$ correspondent à des connexions bloquantes et non-bloquantes respectivement, sur un réseau d'interconnexion pipeliné implicite. Pour chaque ASM contenant des opérateurs de connexion sur réseau pipeliné, un réseau sera inféré, et l'ensemble des puits est séparé en deux ensembles disjoints afin de départager les puits qui sont accessibles via réseau de ceux qui ne le sont pas. Pour chaque source qui est connectée à au moins un puits accessible via le réseau pipeliné, un port d'entrée sera présent sur le réseau pipeliné. Chaque port est composé de deux canaux complètement synchronisés, un pour les données à envoyer et l'autre pour les destinations des données à envoyer. Une connexion $p_j|{=}|s_i$ sur réseau pipeliné est ainsi traduite en une requête d'envoi sur le port d'entrée associé à s_i du réseau implicite en spécifiant l'adresse de destination correspondant à p_i. Cette requête est bloquante dans le cas d'un opérateur bloquant $| = |$, et est non-bloquante dans le cas d'un opérateur non-bloquant $|* = |$.

3.4.3 Ordonnancement

Dans la présente implémentation, le réseau d'interconnexions pipeliné implicite garantit une complétion en ordre des requêtes d'envois pour chaque paire de source et de puits possible. Ainsi, si on envoie une séquence de données $I = i_0, i_1, i_2, ...$ d'une source s_i vers un puits p_j, la séquence $i_0, i_1, i_2, ...$ reçue vérifiera toujours $t_r(i_0) < t_r(i_1) < t_r(i_2) < ...$, ou $t_r(x)$ correspond au temps de réception d'une donnée x. Si au même moment un ensemble de données $J = j_0, j_1, j_2, ...$ est envoyé d'une source $s_{k \neq i}$ vers le puits p_j, la séquence des données

reçues respectera $t_r(i_0) < t_r(i_1) < t_r(i_2) < ...$ et $t_r(j_0) < t_r(j_1) < t_r(j_2) < ....$

3.4.4 Transformation

La Figure 3.9 illustre au moyen d'un exemple simple le processus de traduction d'une description contenant des opérateurs de connexions sur réseau pipeliné en une description contenant uniquement des opérateurs de base ($=, *=$). La Figure 3.9a donne la description d'une ASM réalisant quatre connexions sur réseau pipeliné. Dans cette description, les puits x et y sont accessibles via un réseau pipeliné, tandis que le puits z est accessible via un réseau point-à-point entièrement combinatoire. La Figure 3.9b illustre l'architecture cible intégrant l'ASM source transformée, ainsi que le réseau d'interconnexion pipeliné et ces ports d'entrées et sorties. Il y a deux ports d'entrée dans ce réseau, chacun composé d'un canal de données et d'un canal d'adresse (de destination), pour les sources a et b qui peuvent envoyer des données sur le réseau. La Figure 3.9c illustre la description de l'ASM après le remplacement des opérateurs de connexion sur réseau par des requêtes d'envois sur le réseau. Les canaux des ports d'entrée du réseau sont nommés en suivant la syntaxe *ncdata_<srcname>* et *ncaddr_<srcname>*. Des règles sont également ajoutées pour assurer la synchronisation des données et des adresses associées aux requêtes d'envois. Les adresses adéquates pour réaliser les connexions spécifiées sont générées automatiquement par le processus de transformation automatisé. Pour chaque ensemble de sources et de puits connectés, un identifiant (adresse) unique sera attribué à chaque puits. Le compilateur est capable de produire une implémentation pour laquelle un nombre minimal de bits est utilisé pour encoder les adresses de destination. Présentement, deux types de réseaux sont supportés : le simple bus partagé et le réseau complètement connecté. Le type de réseau et différents paramètres de configurations peuvent être spécifiés comme propriétés d'une ASM. Dans l'implémentation courante, un seul type de réseau peut être spécifié par ASM. Les paramètres de configuration incluent la taille des tampons FIFO d'entrée et sortie du réseau, ainsi que la taille maximale des multiplexeurs et démultiplexeurs entre chaque étage de pipeline.

Réseau à bus simple partagé

En spécifiant l'utilisation d'un réseau à bus simple partagé, il est possible de réduire à un minimum l'utilisation des ressources logiques consacrées à l'interconnexion des sources et des puits synchronisés par les données. En contrepartie ce réseau offre une bande-passante limitée à un transfert par cycle d'horloge. Pour chaque ensemble de sources et de puits connectés, un simple bus partagé est inféré, doté d'un seul multiplexeur pipeliné ayant chacune des sources de l'ensemble à son entrée.

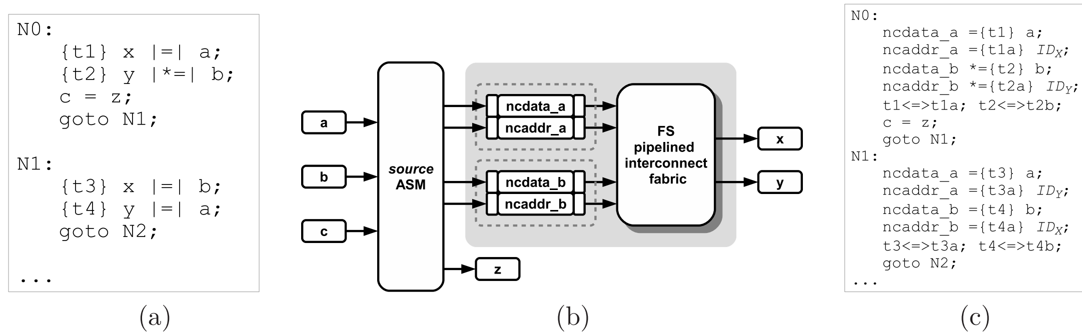


Figure 3.9 Sémantiques matérielles pour les opérateurs de connexions sur réseau pipeliné. a) Exemple de code source. b) Représentation matérielle. c) Code source transformé.

Réseau complètement connecté

En spécifiant l'utilisation d'un réseau complètement connecté, il est possible d'obtenir la bande-passante la plus élevée possible pour un ensemble de sources et de puits interconnectés. En contrepartie, ce réseau requiert une quantité plus importante de ressources logiques pour son implémentation. Pour chaque ensemble de sources et de puits connectés, il y aura un multiplexeur pour chacun des puits présents. Le nombre d'entrées de chaque multiplexeur correspond au nombre de sources connectées au puits correspondant.

3.5 Conclusion

En simplifiant la description des transferts de données entre des sources et des puits avec interfaces à flot de données au moyen de connexions bloquantes et non-bloquantes, le langage CASM offre un niveau d'abstraction intéressant pour la synthèse de circuits numériques. Afin d'élever davantage ce niveau d'abstraction au niveau des transferts synchronisés par les données, ce chapitre a fait la présentation de différents ajouts sémantiques et syntaxiques de notre variante CASM+ du langage CASM original.

Le langage CASM+ permet la spécification de *règles d'autorisation* des transferts de données sur des connexions actives. Ces règles peuvent former des relations combinatoires cycliques en termes des attributs et signaux de synchronisations des connexions impliquées. Néanmoins, la méthode de synthèse automatisée qui est présenté au Chapitre 4 permet de résoudre de telles relations cycliques en un réseau acyclique de fonctions logiques maximisant le taux de transferts. En particulier, l'approche proposée permet de supporter les relations combinatoires cycliques ayant un ou plusieurs point stables.

Afin de simplifier et de faciliter la description d'algorithmes utilisant des opérateurs pipelinés,

un opérateur de connexion de type *différé* a été introduit. Cet opérateur de connexion avancé permet d'abstraire le niveau de complexité associé à la description du contrôle requis pour une utilisation efficace des opérateurs pipelinés. Au moyen de cet opérateur de connexion différée, il est possible de spécifier des opérations ayant lieu dans différents domaines temporels avec une syntaxe similaire à celle utilisée pour interconnecter des opérateurs combinatoires au moyen des opérateurs de connexion de base.

Un opérateur de connexion de type *réseau* d'interconnexion pipeliné a également été introduit. Cet opérateur permet de spécifier des connexions sur un réseau d'interconnexion pipeliné complètement synchronisé par les données avec une syntaxe similaire à celle utilisée pour spécifier des connexions point-à-point avec les opérateurs de connexion de base. Le réseau d'interconnexion pipeliné est généré automatiquement par le compilateur.

CHAPITRE 4 SYNTHÈSE AUTOMATISÉE DES DECRIPTIONS CASM+

La description de connexions entre des composants aux interfaces de type flot de données dans différentes topologies peut induire des relations combinatoires cycliques dans la logique de contrôle du chemin de données. Plus particulièrement, ces relations combinatoires cycliques peuvent admettre un ou plus d'un point stable à la fois. La gestion manuelle de telles relations combinatoires cycliques contrevient à l'esprit de la description au niveau des transferts synchronisés par les données, qui vise à réduire le niveau de complexité associé à la spécification de la logique de contrôle. Dans ce chapitre, une approche de niveau fonctionnel est proposée pour automatiser le processus de résolution des relations combinatoires cycliques en termes des signaux de synchronisations et d'autorisation des transferts de données. Les relations cycliques sont traduites en un réseau acyclique de fonctions logiques maximisant le nombre de transferts réalisés à chaque cycle d'horloge. La méthode de synthèse automatisée a été intégrée à notre compilateur CASM+.

4.1 Introduction

Le compilateur CASM+ réalisé dans le cadre de ce travail de recherche a été développé avec le langage de programmation Java. La Figure 4.1 illustre les principales étapes du flot de compilation. L'étage d'entrée du compilateur est responsable d'analyser la description CASM+ et d'en produire un arbre syntaxique abstrait. Cet arbre est produit à l'aide du générateur de *parser* (outil d'analyse de texte) SableCC. L'arbre syntaxique produit à partir de la description CASM+ est ensuite analysé afin de produire une représentation intermédiaire hiérarchique. Il est alors possible de réaliser les transformations des opérateurs de connexion de type différé et réseau, présenté au Chapitre 3, pour obtenir une représentation contenant uniquement des opérateurs (bloquant et non-bloquant) de base. Les ASMs sont ensuite analysées afin d'élaborer les différentes équations logiques liées aux différents attributs des connexions, et aux signaux de synchronisation des interfaces à flot de données. Les équations logiques sont représentées par des arbres de décision binaire (BDD) au sein du compilateur. La bibliothèque *JavaBDD* est utilisée à cet effet, permettant de s'interfacer à différentes implémentations, dont la bibliothèque C BuDDy. Après cette étape, l'identification et la résolution des dépendances combinatoires cycliques sont réalisées, produisant un réseau acyclique de fonctions logiques. Cette étape fait appel aux outils d'analyse de graphes. Pour cette fin, la bibliothèque *JGraphT* est utilisée par le compilateur. Ensuite, une description au niveau RTL en langage VHDL est produite, en vue d'être synthétisée à l'aide des outils

de synthèse RTL existants.

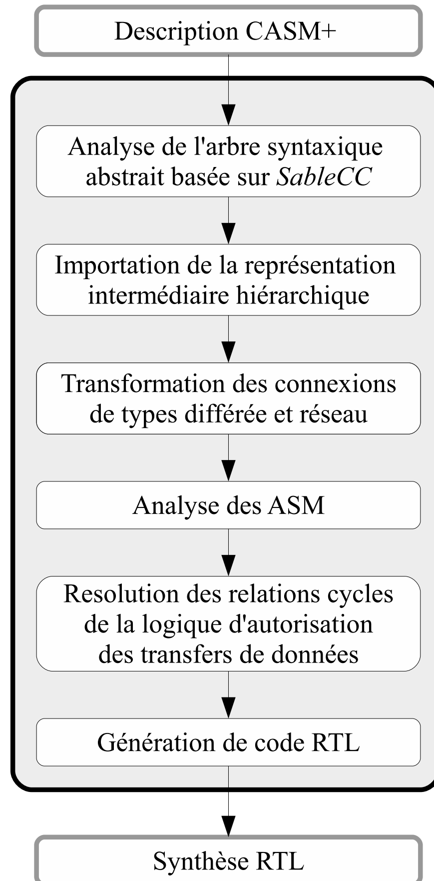


Figure 4.1 Flot de compilation CASM.

L'architecture générique d'un circuit associée à une description CASM+ est illustrée dans la Figure 4.2. L'architecture comprend un ensemble d'ASM concurrentes, ainsi que les sources et les puits synchronisés par les données qui sont interconnectés. Le bloc de logique d'autorisation des transferts est responsable de piloter les *entrées de synchronisation* des sources et des puits, en fonction des connexions activées et des *sorties de synchronisation* des sources et des puits. Le bloc de logique d'autorisation des transferts pilote également un réseau d'interconnexions point-à-point (combinatoire) composé de multiplexeurs permettant d'aiguiller les données des sources aux puits.

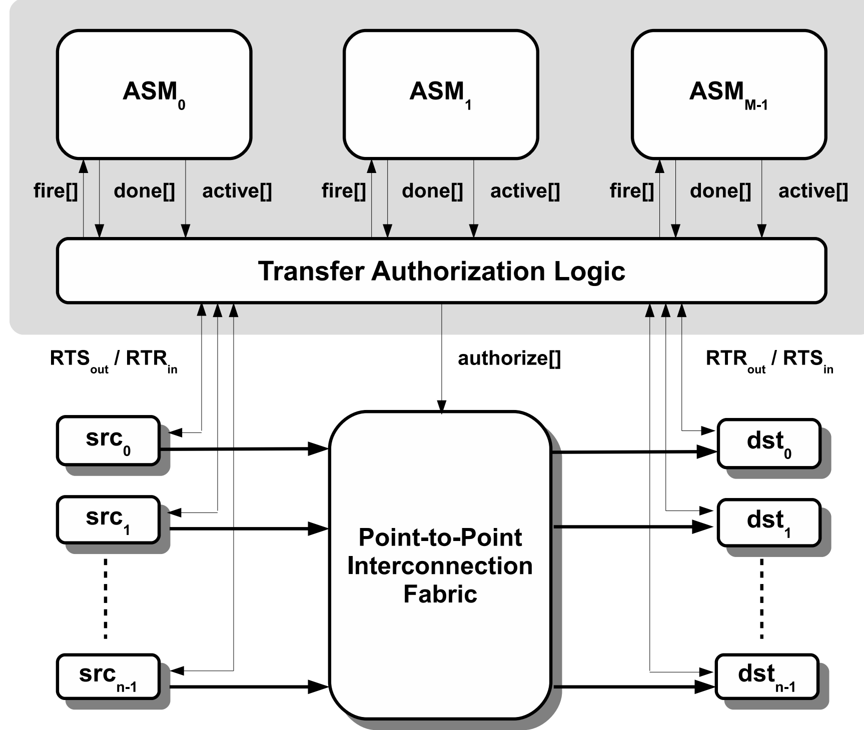


Figure 4.2 Architecture associée à une description CASM+.

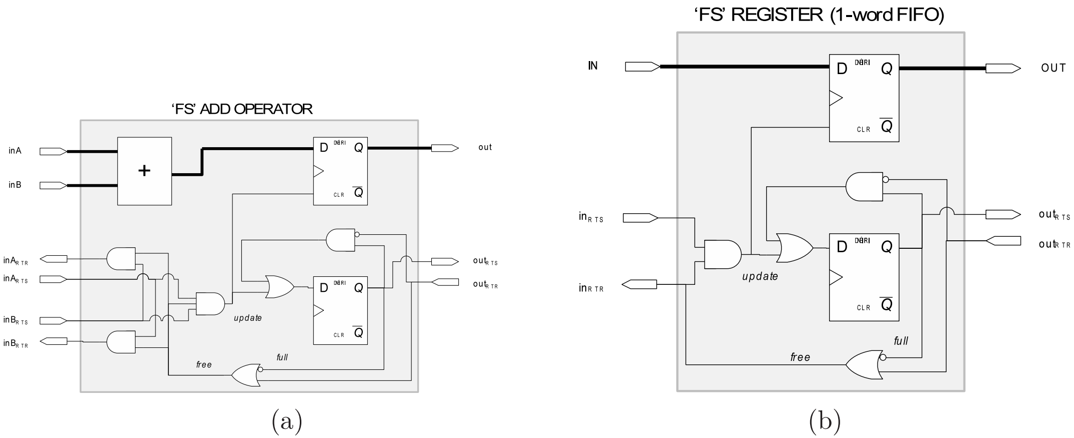
4.2 Problématique

4.2.1 Dépendances combinatoires structurelles

L'interconnexion de sources et de puits synchronisés par les données dans différentes topologies peut mener à des dépendances combinatoires cycliques. Ces dépendances cycliques peuvent être causées par la présence de fonctions combinatoires liant les entrées et sorties de synchronisation des différents composants interconnectés, ou par la spécification de certaines règles d'autorisation des transferts de données. La Figure 4.3 illustre les circuits d'un opérateur d'addition pipeliné et d'un registre complètement synchronisé (FIFO de 1 mot). On observe dans les deux circuits la présence de fonctions combinatoires liant les entrées de synchronisation (*rtr*) et les sorties de synchronisation (*rtr*). Ces fonctions combinatoires sont résumées par les Équations 4.1, 4.2, et 4.3.

$$add.inA_{rtr} = (\overline{add.out_{rts}} \vee add.out_{rtr}) \wedge add.inB_{rts} \tag{4.1}$$

$$add.inB_{rtr} = (\overline{add.out_{rts}} \vee add.out_{rtr}) \wedge add.inA_{rts} \tag{4.2}$$



(a) (b)

Figure 4.3 Deux opérateurs complètement synchronisés. Un additionneur à latence de 1 cycle (a) et un registre synchronisé (b).

$$reg.in_{rtr} = \overline{reg.out_{rts}} \vee reg.out_{rtr} \tag{4.3}$$

Dépendances fonctions d'un état interne

Les dépendances combinatoires structurelles liant les entrées et sorties de synchronisation des différents composants interconnectés considérées dans cette thèse peuvent également dépendre d'états internes à ces composants. La Figure 4.4 illustre une FIFO de 1 mot complètement synchronisée (FS), avec un chemin de contournement (*bypass*). Le chemin de contournement est rendu possible par le multiplexeur connecté à la sortie *out*, qui permet à l'entrée *in* d'être reliée directement à la sortie par un chemin combinatoire. Pour ce type de composant, lorsqu'un mot est présent dans la FIFO, cette dernière se comporte exactement comme le registre complètement synchronisé (FIFO de 1 mot) de la Figure 4.3b. Cependant, lorsque la FIFO est vide, cette dernière se comporte comme un conduit (fil) complètement synchronisé pour lequel l'entrée est directement connectée à la sortie. Pour un conduit complètement synchronisé, on observe les relations $conduit.out_{rts} = conduit.in_{rts}$, $conduit.in_{rtr} = conduit.out_{rtr}$, et $conduit.out = conduit.in$. Il s'ensuit que les relations liant les entrées et sorties de ce composant dépendent de l'état interne $full$, qui n'est pas équivalent à la sortie rts dans ce cas. La relation combinatoire liant la sortie rts de la FIFO de 1 mot avec chemin de contournement dépend de son entrée rts ainsi que de son état interne $full$. De même, la sortie rtr dépend de l'entrée rtr ainsi que de la valeur de l'état interne $full$.

La méthode de synthèse automatisée présentée dans cette section permet de supporter des

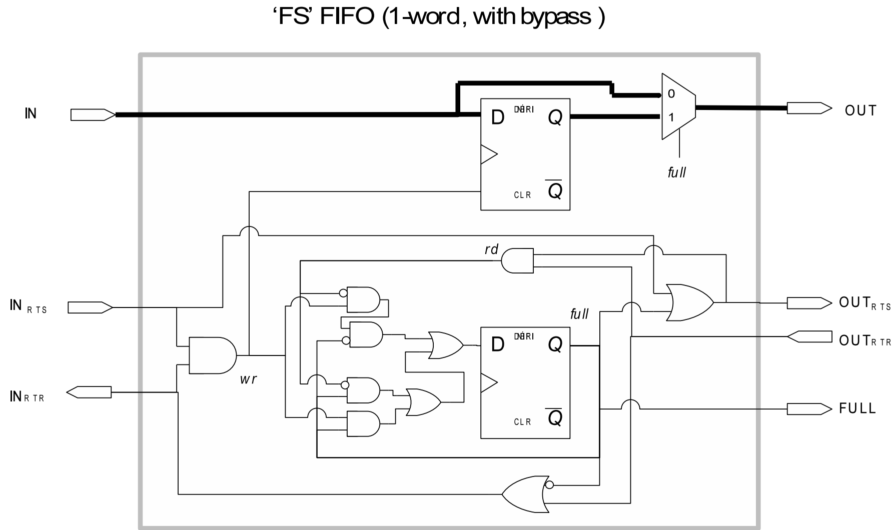


Figure 4.4 FIFO complètement synchronisée, avec chemin de contournement.

composants pour lesquels les dépendances combinatoires liant les entrées et sorties de synchronisation dépendent d'un état interne. En effet, il suffit alors que cet état interne soit observable (par le circuit de contrôle) depuis l'extérieur du composant. Dans l'exemple de la FIFO de 1 mot avec chemin de contournement illustré à la Figure 4.4, cela est réalisé par la présence du port de sortie $FULL$.

4.2.2 Relations combinatoires cycliques

Le circuit illustré à la Figure 4.5 représente le chemin de données d'un accumulateur utilisant un additionneur pipeliné. On assume que l'additionneur et la FIFO A correspondent aux circuits de la Figure 4.3. On assume que l'entrée *source* et la sortie *result* de l'accumulateur ont des interfaces complètement synchronisées. Le chemin de données présente une rétroaction de la sortie de l'additionneur vers l'entrée de la FIFO A ou vers l'entrée inB de l'additionneur. Selon que la rétroaction ait lieu vers l'un ou l'autre de ces entrées, les relations combinatoires cycliques des Équations 4.4 et 4.5 sont formées, respectivement. L'Équation 4.4 est obtenue en substituant $add.out_{rtr} = A.in_{rtr}$ dans l'Équation 4.1, où l'expression $A.in_{rtr}$ est définie par l'Équation 4.3. L'Équation 4.5 est obtenue directement en substituant $add.out_{rtr} = add.inB_{rtr}$ dans l'Équation 4.2.

Un problème fondamental avec ces relations cycliques est qu'elles peuvent supporter plus d'un état stable à la fois. Pour la relation cyclique de l'Équation 4.4, lorsque le vecteur d'entrée $(A.out_{rts}, add.out_{rts}, add.inB_{rts})$ est égal à $(1, 1, 1)$, la relation est satisfaite à la fois par $add.inA_{rtr} = 0$ et $add.inA_{rtr} = 1$. De même, pour la relation cyclique de l'Équation 4.5

lorsque le vecteur d'entrée $(add.out_{rts}, add.inA_{rts})$ est égal à $(1, 1)$, la contrainte est satisfaite pour $add.inB_{rtr} = 0$ et $add.inB_{rtr} = 1$. Dans ce contexte, il est préférable pour chacun de ces cas de restreindre les points stables à $add.inA_{rtr} = 1$ et $add.inB_{rtr} = 1$ respectivement, car ceux-ci permettent un plus grand nombre de transferts de données.

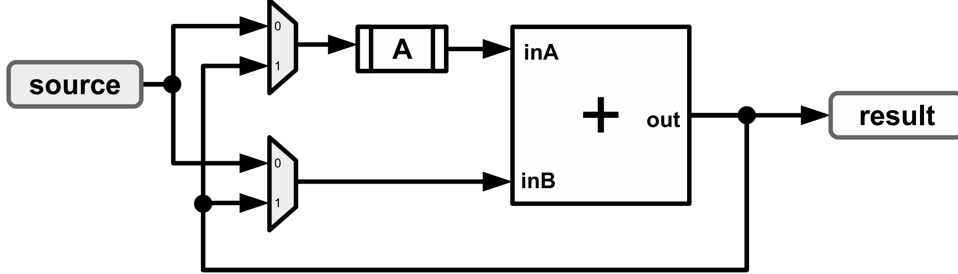


Figure 4.5 Accumulateur basé sur un additionneur pipeliné.

$$A.in_{rtr} = \overline{A.out_{rts}} \vee ((\overline{add.out_{rts}} \vee A.in_{rtr}) \wedge add.inB_{rts}) \tag{4.4}$$

$$add.inB_{rtr} = (\overline{add.out_{rts}} \vee add.inB_{rtr}) \wedge add.inA_{rts} \tag{4.5}$$

Bien que certains outils de synthèse RTL soient capables de supporter la présence de relations combinatoires cycliques, cela s'applique lorsque celles-ci sont équivalentes à des fonctions logiques. Le cas des relations cycliques des Équations 4.4 et 4.5 est plus problématique car il n'y a pas d'équivalence à un réseau acyclique de fonctions logiques. Afin de rendre le réseau cyclique de fonctions logiques équivalent à un réseau acyclique de fonction logiques, il faut s'assurer que pour chaque vecteur d'entrée un et un seul point stable soit permis. Afin de retrouver le point stable correspondant à l'intention du concepteur, nous proposons de sélectionner celui qui permet le plus de transferts de données à chaque cycle d'horloge.

4.3 Logique de synchronisation des transferts

La logique de synchronisation des transferts consiste en un réseau de fonctions logiques liant les signaux de synchronisation de sorties aux signaux de synchronisation d'entrées des sources et puits interconnectés, en fonction des connexions activées et complétées. Ce réseau contient toutes les fonctions logiques associées aux attributs des énoncés de connexion présentés au Chapitre 3. Afin de réduire le nombre de fonctions dans le réseau, il est possible de réécrire ce dernier en termes d'attributs d'instances de connexion. Une instance de connexion est spécifiée par une source et un puits, ainsi que par un identifiant. Plusieurs énoncés de connexion entre une source si et un puits pi sont ainsi associés à une et une seule instance de connexion

$(conn(s_i \rightarrow p_i)$. Puisque le nombre de connexions possibles entre N sources et M puits est inférieur ou égal au nombre d'énoncées de connexions possibles, la réécriture du réseau en termes d'instances de connexion, plutôt qu'en termes d'énoncés de connexions, permet de réduire considérablement le nombre de fonctions logiques (noeuds) qu'il contient.

Considérant une instance de connexion c_i associée à un ensemble d'énoncés de connexion $t_1, t_2, ..., t_n$, l'Équation 4.6 permet de ré-exprimer le réseau en termes d'attributs d'*instances* de connexion, partant de la description des attributs d'*énoncés* de connexion. L'Équation 4.9 illustre la transformation inverse. Puisque la transformation de l'Équation 4.6 permet de réduire le nombre de noeuds dans le graphe associé au réseau de fonctions, cette dernière est appliquée préalablement à la résolution des relations combinatoires cycliques. L'ensemble des fonctions associées aux attributs des énoncés de connexions peut être retrouvé par application de la transformation inverse de l'Équation 4.9, une fois la résolution des relations combinatoires cycliques complétée.

Par exemple, considérant les énoncés de connexions $\{t1\}x$ `*=` a et $\{t2\}x$ `*=` a de deux états différents d'une ASM donnée, il est possible d'exprimer $conn(a \rightarrow x).fire = t1.fire \vee t2.fire$. Inversément, partant de l'expression de $conn(a \rightarrow x).fire$, il est possible d'exprimer $t1.fire = t1.active \wedge conn(a \rightarrow x).fire$ et $t2.fire = t2.active \wedge conn(a \rightarrow x).fire$.

$$ci. < attribute >= t1. < attribute > \vee t2. < attribute > \vee ... \vee tn. < attribute > \tag{4.6}$$

$$t1. < attribute >= t1.active \wedge ci. < attribute > \tag{4.7}$$

$$t2. < attribute >= t2.active \wedge ci. < attribute > \tag{4.8}$$

$$tn. < attribute >= tn.active \wedge ci. < attribute > \tag{4.9}$$

Afin d'illustrer les différentes étapes de la méthode de résolution des relations cycliques proposée, l'exemple de l'accumulateur déjà présenté à la Figure 4.5 sera utilisé. Plus particulièrement, seule la configuration formée des connexions $add.inA$ `*=` A, $add.inB$ `*=` $add.out$, et A `*=` $source$ est considérée. La première étape de la méthode de résolution proposée consiste à produire le réseau de fonctions logiques, liant les signaux de synchronisation de sorties aux signaux de synchronisation d'entrées des sources et puits interconnectés. La Figure 4.6 illustre un sous-graphe du réseau de fonctions logiques pour l'accumulateur dans la configuration considérée. Le sous-graphe contient un ensemble de points fortement connexes composé

des signaux de synchronisation $add.out.rtr$ et $add.inB.rtr$. Les rectangles réprésentent des entrées primaires pour le sous-graphe considéré.

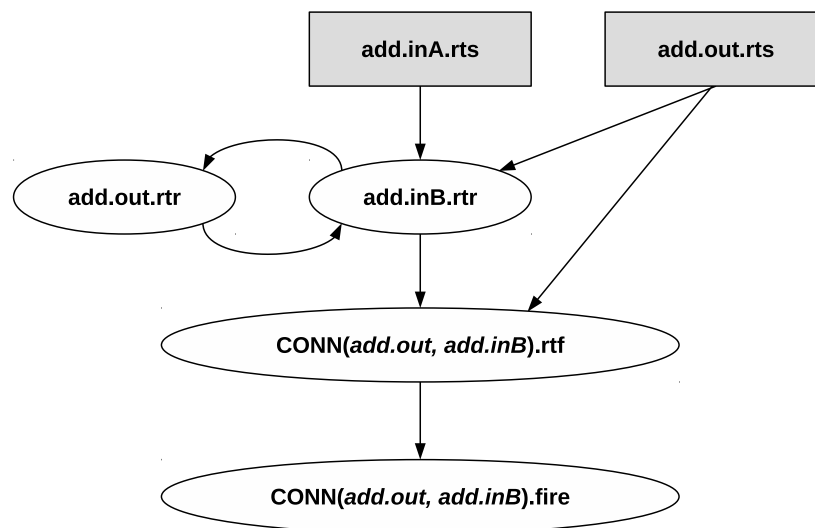


Figure 4.6 Exemple de réseau combinatoire cyclique.

4.4 Analyse de stabilité

L'analyse de stabilité permet d'identifier les combinaisons d'entrées des relations combinatoires cycliques pour lesquelles plusieurs états stables sont possibles. Au niveau du graphe associé au réseau de fonctions logiques, les relations combinatoires cycliques forment des ensembles de points (fonctions) fortement connexes (*Strongly Connected Component*, (SCC)). Un groupe de points fortement connexes dans un graphe orienté est un groupe pour lequel entre chaque paire de points (u, v), il existe un chemin de u à v. Après identification de tous les ensembles de points fortement connexes, ceux-ci sont triés topologiquement. Le tri topologique permet de garantir que lors de la résolution de chaque SCC, les noeuds en amont du SCC correspondent à des fonctions combinatoires (acycliques). Cette approche ouvre la porte à d'éventuelles simplifications des fonctions logiques à l'entrée du SCC considéré. La méthode d'analyse et de résolution est réalisée itérativement sur chacun des ensembles de points fortement connexes. À chaque itération, un ensemble de points fortement connexes est résolu afin d'être remplacé par un ensemble de points formant un graphe acyclique.

4.4.1 Recherche d'un ensemble de points de rétroaction

L'analyse de chaque SCC, basée sur les travaux présentés par [69], requiert de trouver un ensemble de points de rétroaction (Feedback Vertex Set), qui rendent acyclique le SCC lors-

qu'enlevés de ce dernier. Au moyen d'un ensemble de points de rétroaction, chaque SCC est transformé en un réseau acyclique en déconnectant les sorties de chaque noeuds de cet ensemble. La Figure 4.7 illustre une telle transformation, considérant un SCC avec un vecteur d'entrées i, et des vecteurs de sorties $\mathbf{X}$ et $\mathbf{Y}$, ou $\mathbf{X}$ représente un ensemble de points de rétroaction. Après avoir déconnecté les sorties du vecteur $\mathbf{X}$, le vecteur d'entrées (de rétroaction) est renommé $\mathbf{x}$.

4.4.2 Contrainte de stabilité

Il a été démontré dans [69] que les assignations stables de $\mathbf{x}$ sont celles qui vérifient la contrainte $C(\mathbf{i}, \mathbf{x})$ donnée par l'Équation 4.10, où $\mathbf{X} = X_K X_{K-1}..X_1 X_0$ et $\mathbf{x} = x_K x_{K-1}..x_1 x_0$ (avec K points de rétroaction). De manière intuitive, l'Équation 4.10 exprime que pour qu'une assignation de $\mathbf{x}$ soit stable, sa valeur doit être équivalente à celle de $\mathbf{X}$. En effet, avant de déconnecter les sorties de $\mathbf{X}$, on observait $\mathbf{X} = \mathbf{x}$, puisque chaque sortie X_k est reliée à l'entrée x_k directement par un fil.

Au moyen de cette contrainte, il est possible de déterminer si une assignation $\mathbf{x_k}$ de $\mathbf{x}$ est stable pour au moins une combinaison du vecteur d'entrée i en vérifiant l'inégalité de l'Équation 4.11. De même, l'ensemble des assignations stables de $\mathbf{x}$, $S(\mathbf{x})$, est donnée par la quantification existentielle de C sur les variables du vecteur $\mathbf{i}$. L'ensemble des assignations stables est exprimé au moyen de l'Équation 4.12. L'Équation 4.12 indique ainsi toutes les assignations du vecteur $\mathbf{x}$ pour lesquelles il existe au moins une assignation du vecteur d'entrée $\mathbf{i}$ qui satisfait la contrainte C.

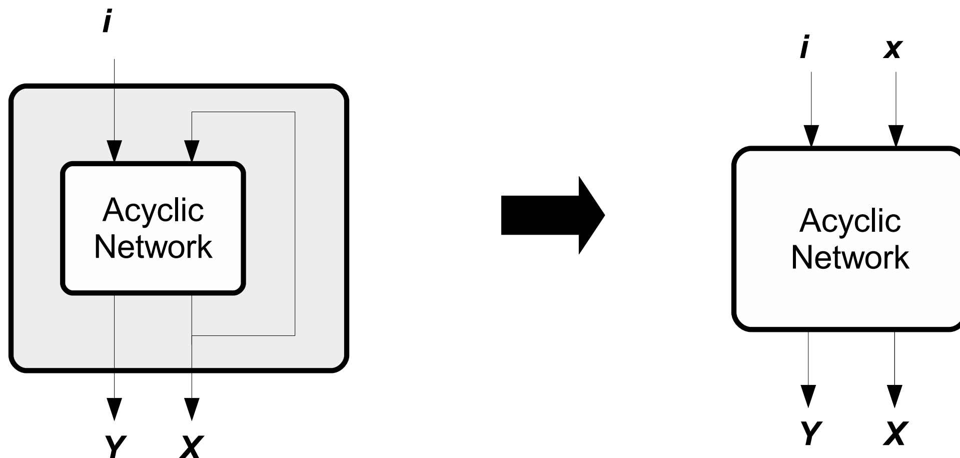


Figure 4.7 Transformation d'un réseau cyclique en un réseau acyclique boucle-ouverte.

$$C(\mathbf{i},\mathbf{x}) = \bigwedge_{k=1}^{K} (X_k(\mathbf{i},\mathbf{x}) \Leftrightarrow x_k) \tag{4.10}$$

$$\exists(\mathbf{i})(C|_{\mathbf{x}=\mathbf{x_k}}) \neq 0 \tag{4.11}$$

$$S(\mathbf{x}) = \exists(\mathbf{i})(C) \tag{4.12}$$

4.4.3 Mono- et Multi-stabilité

Inversement, il est possible de déterminer l'ensemble des assignations de **i** qui supportent au moins un état stable au moyen de l'Équation 4.13. L'ensemble des assignations de **i** qui supportent au moins un état stable est donné par la quantification existentielle de C sur les variables de **x**. Le complément de cette expression permet également d'obtenir l'ensemble des assignations de **i** qui ne supportent aucun un état stable. Les relations cycliques ayant des assignations de **i** qui ne supportent aucun un état stable ne sont pas permises, elles sont associées à des comportement instables. Il est possible d'obtenir l'ensemble des assignations de **i** qui supportent un et un seul état stable au moyen de l'Équation 4.15. Cet ensemble est obtenu par construction en faisant la disjonction de tous les cas de figures possibles ne permettant qu'un et un seul état stable à la fois. À partir de l'expression de l'ensemble des assignations de **i** qui supportent un et un seul un état stable, il est possible de dériver l'ensemble des assignations de **i** qui supportent plus d'un état stable au moyen de l'Équation 4.16.

$$stable(\mathbf{i}) = \exists(\mathbf{x})(C) \tag{4.13}$$

$$unstable(\mathbf{i}) = \overline{\exists(\mathbf{x})(C)} \tag{4.14}$$

$$\begin{aligned} monostable(\mathbf{i}) = {} & (C|_{\mathbf{x}=\mathbf{s_0}} \wedge \overline{C|_{\mathbf{x}=\mathbf{s_1}}} \wedge ... \wedge \overline{C|_{\mathbf{x}=\mathbf{s_{n-1}}}}) \vee \\ & (\overline{C|_{\mathbf{x}=\mathbf{s_0}}} \wedge C|_{\mathbf{x}=\mathbf{s_1}} \wedge ... \wedge \overline{C|_{\mathbf{x}=\mathbf{s_{n-1}}}}) \vee \\ & ... \vee \\ & (\overline{C|_{\mathbf{x}=\mathbf{s_0}}} \wedge \overline{C|_{\mathbf{x}=\mathbf{s_1}}} \wedge ... \wedge C|_{\mathbf{x}=\mathbf{s_{n-1}}}) \end{aligned} \tag{4.15}$$

$$multistable(\mathbf{i}) = stable(\mathbf{i}) \wedge \overline{monostable(\mathbf{i})} \quad (4.16)$$

La Figure 4.8 illustre la table de vérité associée à une contrainte $C(\mathbf{i}, \mathbf{x})$ qui présente les trois cas de figures mentionnés (instable, monostable, multi-stable). La contrainte contient 2-bits d'entrée ($\mathbf{i}$) et 1 seul point de rétroaction ($\mathbf{x}$). Lorsque le vecteur d'entrées $\mathbf{i}$ est assigné à $(0, 0)$ un comportement instable est détecté, il n'y a aucune valeur de $\mathbf{x}$ qui permette de satisfaire la contrainte C. Lorsque le vecteur d'entrées $\mathbf{i}$ est assigné à $(0, 1)$ ou $(1, 0)$, un comportement monostable est observé. Dans chaque cas, il y a une et une seule assignation de $\mathbf{x}$ qui permette de satisfaire la contrainte C. Finalement lorsque le vecteur d'entrée $\mathbf{i}$ est assigné à $(1, 1)$ un comportement multi-stable est observé. Dans ce cas, il existe deux assignations possibles de $\mathbf{x}$ qui permettent de satisfaire la contrainte.

i	*x*	*C*	
00	0	0	instable(**i**) = $\overline{i_1 + i_0}$
00	1	0	
01	0	0	monostable(**i**) = i_1 xor i_0
01	1	1	
10	0	1	
10	1	0	
11	0	1	multi-stable(**i**) = $i_1 \, i_0$
11	1	1	

Figure 4.8 Table de vérité pour une contrainte de stabilité C avec des entrées primaires $i_1 i_0$ et de rétroaction x_0, illustrant les 3 cas possibles (instable, monostable, multi-stable).

Revenant à l'exemple de la Figure 4.6, en choisissant le point *add.out.rtr* comme ensemble de point de rétroaction, la Figure 4.9 illustre le graphe acyclique obtenu en déconnectant la sortie du point de rétroaction *add.out.rtr*. Le graphe acyclique possède une nouvelle entrée $add.result.rtr.fvs$ pour les cibles originales du point *add.out.rtr* dans le réseau. Le Tableau 4.1 illustre la table de vérité correspondant à la contrainte de stabilité associée (Équation 4.10) au graphe boucle-ouverte de la Figure 4.9. On observe que lorsque les entrée $add0.a.rts$ et $add0.out.rts$ sont toutes deux à 1, la contrainte présente un comportement multi-stable en termes du signal de rétroaction $add0.out.rtr.fvs$.

4.5 Stabilisation des relations combinatoires cycliques

Le processus de stabilisation des relations combinatoires cycliques transforme une contrainte de stabilité en vue de la rendre monostable. Pour chaque combinaison d'entrées $\mathbf{i_k}$ permettant

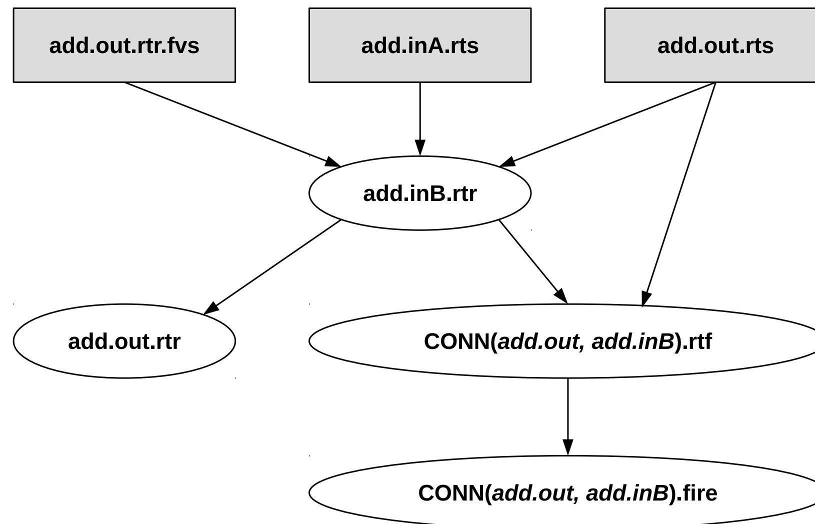


Figure 4.9 Exemple de réseau combinatoire acyclique boucle-ouverte.

Tableau 4.1 Table de vérité de la contrainte de stabilité de la Figure 4.9

Entrées primaires		Entrées de rétroaction	Contrainte de stabilité
add.inA.rts	add.out.rts	add.out.rtr.fvs	add.out.rtr <=> add.out.rtr.fvs
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

un comportement multi-stable, il est requis de contraindre la contrainte pour ne permettre qu'un et un seul point stable. On assume que la contrainte de stabilité initiale ne présente que des comportements monostables et multi-stables, les comportements instables ne sont pas supportés. Pour réaliser la sélection automatique de l'état monostable à conserver pour chaque combinaison d'entrées permettant un comportement multi-stable, l'état associé à un plus grand nombre de transferts de données est conservé. Pour chaque état stable, les attributs $fire$ des transferts de données associés aux signaux de synchronisation et d'autorisation impliqués dans le réseau cyclique sont utilisés pour évaluer le nombre de transferts autorisés.

Dans un premier temps, la contrainte de stabilité est augmentée pour y inclure l'évaluation des attributs $fire$ des connexions impliquées, tel qu'exprimé par l'Équation 4.17. Pour ajouter un attribut $c_j.fire$ à la contrainte, on exprime que pour satisfaire la contrainte augmentée il faut également que la valeur de $c_j.fire$ soit équivalente à l'expression logique de cet attribut,

soit $c_j.FIRE$, exprimé en fonction des entrées $\mathbf{i}$ et $\mathbf{x}$.

$$C' = C \wedge (\bigwedge_{j=1}^{J} (c_j.fire <=> c_j.FIRE)) \tag{4.17}$$

4.5.1 Méthode I

A ce point, il est possible de construire la contrainte monostable en itérant sur l'ensemble des combinaisons d'entrées associées à des comportements multi-stables (Équation 4.16). Pour chacune de ces combinaisons d'entrées $\mathbf{i_k}$, en considérant que $\mathbf{s_i}$ est un état stable allouant un nombre maximal de transferts de données, la contrainte peut être rendue monostable en appliquant la transformation de l'Équation 4.18. L'Équation 4.18 exprime que lorsque le vecteur d'entrée $\mathbf{i}$ est évalué à $\mathbf{i_k}$, seul l'état stable $\mathbf{s_i}$ est permis (premier terme de la disjonction). Toutefois, lorsque le vecteur d'entrée $\mathbf{i}$ n'est pas égal à $\mathbf{i_k}$, la contrainte C demeure identique (deuxième terme de la disjonction).

$$C' = (\mathbf{i_k} \wedge \mathbf{s_i}) \vee (\overline{\mathbf{i_k}} \wedge C) \tag{4.18}$$

4.5.2 Méthode II

L'approche itérative basée sur l'Équation 4.18 permet de stabiliser correctement les contraintes, néanmoins plusieurs combinaisons d'entrées peuvent activer des ensembles d'états stables identiques. Une autre approche possible consiste à identifier tous les états multi-stables possible et de les trier selon un ordre décroissant de nombre de transferts alloués. Par la suite, en itérant sur l'ensemble des états multi-stable possibles, il est possible de stabiliser la contrainte en appliquant pour chaque état multi-stable $\mathbf{s_i}$ la transformation spécifiée par l'Équation 4.19. La transformation de l'Équation 4.19 spécifie que les combinaisons d'entrées permettant l'état stable $\mathbf{s_i}$ ne pourront permettre que cet état stable et aucun autre (premier terme de la disjonction), tandis que la contrainte n'est pas modifiée sur l'ensemble des combinaisons d'entrées i ne supportant pas l'état stable $\mathbf{s_i}$ (deuxième terme de la disjonction).

$$C' = ((C|_{\mathbf{x}=\mathbf{s_i}}) \wedge \mathbf{s_i}) \vee ((\overline{C|_{\mathbf{x}=\mathbf{s_i}}}) \wedge C) \tag{4.19}$$

Le Tableau 4.2 illustre la table de vérité de la contrainte de stabilité augmentée associée au graphe boucle-ouverte de la Figure 4.9, après stabilisation. On observe que pour chaque assignation du vecteur d'entrée $\mathbf{i}$ (formé des deux entrées *add.inA.rts* et *add.out.rts*), il n'y a qu'une et une seule assignation du vecteur de rétroaction $\mathbf{x}$ (*add.out.rtr.fvs*) qui vient

satisfaire la contrainte de stabilité.

Tableau 4.2 Table de vérité de la contrainte de stabilité après stabilisation

Entrées primaires		Entrées de rétroaction	transferts exécutés	Contrainte de stabilité
add.inA.rts	add.out.rts	add.out.rtr.fvs	conn(add.out− > add.inB).fire	add.out.rtr <=> add.out.rtr.fvs
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	0	0
1	0	0	0	0
1	0	1	0	1
1	1	0	0	**0**
1	1	1	1	1

4.6 Génération de fonctions combinatoires acycliques

Une fois la contrainte de stabilité rendue monostable, la génération d'un réseau acyclique de fonction logiques combinatoires est réalisée en remplaçant l'expression des fonctions logiques des points de rétroaction à l'aide de l'Équation 4.20. L'équation 4.20 permet d'obtenir directement une expression de la fonction d'un noeud de rétroaction X_k en termes des entrées primaires ($\mathbf{i}$) du réseau cyclique associé à la contrainte. L'Équation 4.20 exprime la quantification existentielle sur les variable du vecteur de rétroaction $\mathbf{x}$ de la contrainte stabilisée, après simplification de cette dernière en assignant la variable x_k à 1. Il s'en suit que X_k correspond à la somme (disjonction) des assignations de $\mathbf{i}$ pour lesquelles la contrainte C est satisfaite lorsque $x_k = 1$.

$$X_k(\mathbf{i}) = \exists(\mathbf{x})(C|_{x_k=1}) \tag{4.20}$$

Partant de la table de vérité illustrée au Tableau 4.2, correspondant à la contrainte stabilisée associée au réseau boucle-ouverte de la Figure 4.9, l'application de l'Équation 4.20 permet d'obtenir *add.out.rtr* = *add.inA.rts*. La table de vérité correspondante est illustrée par le Tableau 4.3. Il est alors possible de remplacer l'expression de *add.out.rtr* dans le graphe cyclique de la Figure 4.6 par cette nouvelle expression, de sorte à obtenir un réseau acyclique. Le graphe du réseau acyclique résultant est illustré à la Figure 4.10.

Tableau 4.3 Table de vérité de la fonction $add0.out.rtr$ résolue

Entrées primaires		Entrées de rétroaction	Contrainte de stabilité
add.inA.rts	add.out.rts	add.out.rtr	add.out.rtr <=> add.out.rtr.fvs
0	0	0	1
0	1	0	1
1	0	1	1
1	1	1	1

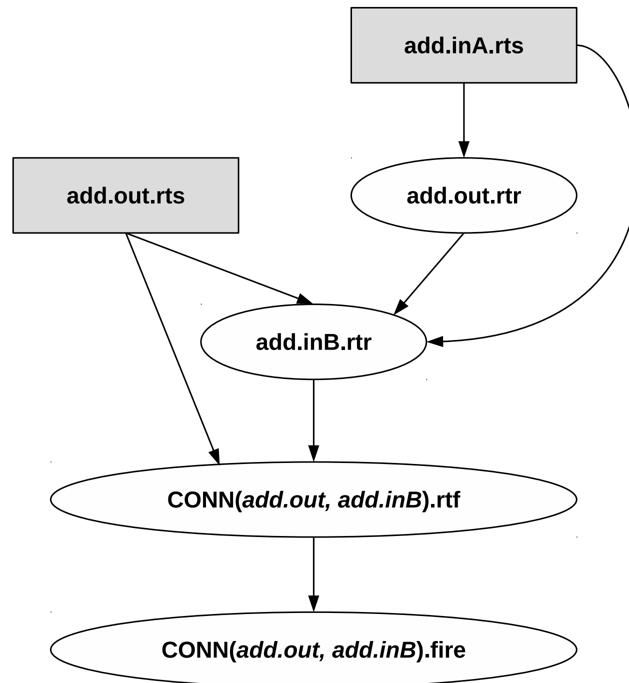


Figure 4.10 Exemple de réseau combinatoire acyclique résolu.

4.7 Conclusion

La description de circuits au niveau des transferts synchronisés par les données est bien adaptée à la gestion d'interconnexions entre des sources et des puits avec des interfaces de type flot de données. Néanmoins, l'interconnexion de sources et de puits dans différentes topologies peut mener à des relations combinatoires cycliques en termes des signaux de synchronisation des interfaces à flot de données. Ces relations combinatoires sont problématiques car elles peuvent supporter plus d'un état stable à la fois, et ne sont pas équivalentes à des réseaux acycliques de fonctions logiques. Cette problématique a été illustrée au moyen d'un exemple simple présentant un accumulateur basé sur un additionneur pipeliné complètement synchronisé. Lorsque plus d'un état stable à la fois sont supportés par le réseau cyclique, le choix de cet état stable en fonction des signaux de synchronisation et d'autorisation vient affecter le

nombre de transfert autorisé par le circuit de contrôle.

L'approche de synthèse automatisée présentée dans ce chapitre propose de transformer le réseau combinatoire cyclique afin qu'il ne permette qu'un et un seul état stable à la fois. La résolution des états multi-stables est réalisée en sélectionnant automatiquement l'état stable qui permet d'accomplir un nombre maximal de transferts de données. Les différentes étapes de cette méthode de résolution des relations combinatoires cycliques ont été intégrées au compilateur CASM+, décrit en langage Java, permettant d'automatiser complètement le processus.

CHAPITRE 5 APPLICATION À LA CONCEPTION DE CIRCUITS

5.1 Introduction

Ce chapitre présente l'application de la méthode de conception au niveau des transferts synchronisés par les données à l'implémentation de circuits numériques dédiés. Différentes applications du domaine du calcul scientifique utilisant des opérateurs pipelinés avec une représentation des nombres de type virgule-flottante sont considérées, telles la sommation pipelinée, le produit matriciel dense, l'élimination Gaussienne, et l'inversion de matrices. Ces applications sont comparées avec des implémentations similaires produites avec des approches RTL et/ou de synthèse haut-niveau en langage C. Le tri de données au moyen de l'algorithme récursif *Quicksort* est également considéré. Ensembles, ces applications permettent d'évaluer le compromis offert par l'approche de niveau intermédiaire proposée en termes de performances et de temps de conception.

5.2 Tri de données rapide

Le tri de données est une opération fondamentale dans le domaine du traitement de données. Deux architectures sont proposées. La première architecture est basée sur une seule ASM interconnectant une mémoire à simple port (2 cycles de latence) et un comparateur pipeliné (2 cycles de latence). La deuxième architecture est basée sur 2 ASM concurrentes, formant un pipeline fonctionnel pour cacher la latence du comparateur, et ainsi de réduire le temps d'exécution de l'implémentation finale. Ces travaux ont fait l'objet d'une première publication dans [70].

5.2.1 Algorithme

Le tri rapide *Quicksort* est un algorithme récursif très répandu pour sa simplicité et son temps d'exécution moyen ($O(n \times log(n))$). La Figure 5.1 présente une description de l'algorithme récursif. Dans un premier temps, un pivot est choisi de manière arbitraire. La valeur contenue à l'indice mémoire de ce pivot est ensuite utilisée pour partitionner la plage mémoire (*gauche* à *droite*) en un ensemble de valeurs inférieures ou égales et un ensemble de valeurs supérieures. La position de la valeur utilisée comme pivot après partitionnement est retourné par la routine *partition*. On ré-exécute ensuite la routine *quicksort* sur chacun des deux ensembles résultant du partitionnement courant de la plage mémoire (*gauche* à *droite*). L'algorithme de partitionnement utilisé est illustré à la Figure 5.2.

```
1: quicksort(mem, left, right)
2: if left < right then
3:     pivot = (left + right) >> 1 ;
4:     pivot = partition(mem, left, right, pivot) ;
5:     quicksort(mem, left, pivot-1) ;
6:     quicksort(mem, pivot+1, right) ;
7: end if
```

Figure 5.1 Algorithme *quicksort*.

```
 1: partition(mem, left, right, pivotIndex)
 2: storeIndex = left ;
 3: pivotValue = mem[pivotIndex] ;
 4: swap mem[pivotIndex] and mem[right] ;
 5: for i = left → right − 1 do
 6:     r1 = mem[i] ;
 7:     if r1 < pivotValue then
 8:         mem[i] = mem[storeIndex] ;
 9:         mem[storeIndex] = r1 ;
10:         storeIndex++ ;
11:     end if
12: end for
13: swap mem[pivotIndex] and mem[right] ;
14: return  storeIndex ;
```

Figure 5.2 Algorithme *partition*.

5.2.2 Conception

La représentation de niveau intermédiaire pour l'architecture basée sur deux ASM et un comparateur pipeliné est illustrée à la Figure 5.3. L'architecture interconnecte des registres, un comparateur pipeliné, ainsi que des mémoires de type FIFO, LIFO, et RAM. L'algorithme est décomposé en deux processus concurrent (*Quicksort* et *SwapUnit*). L'ASM *Quicksort* est responsable d'initier les comparaisons requise par la routine de partitionnement, tandis que l'ASM *SwapUnit* est responsable de réaliser les permutations requises en fonction des résultats de comparaison. Trois ASMs supplémentaires sont présentes pour arbitrer les accès à la mémoire double-port partagée par les ASMs *Quicksort* et *SwapUnit*.

5.2.3 Résultats de synthèse et d'implémentation

Les résultats de synthèse et d'implémentation présentés dans cette section ont été obtenus avec l'outil de synthèse Quartus II d'Altera en ciblant un FPGA de type Stratix-III. La fonctionnalité de l'implémentation RTL produite par notre compilateur a été validée par simulation RTL. Le comparateur pipeliné ainsi que les mémoires RAM ont été réalisés à

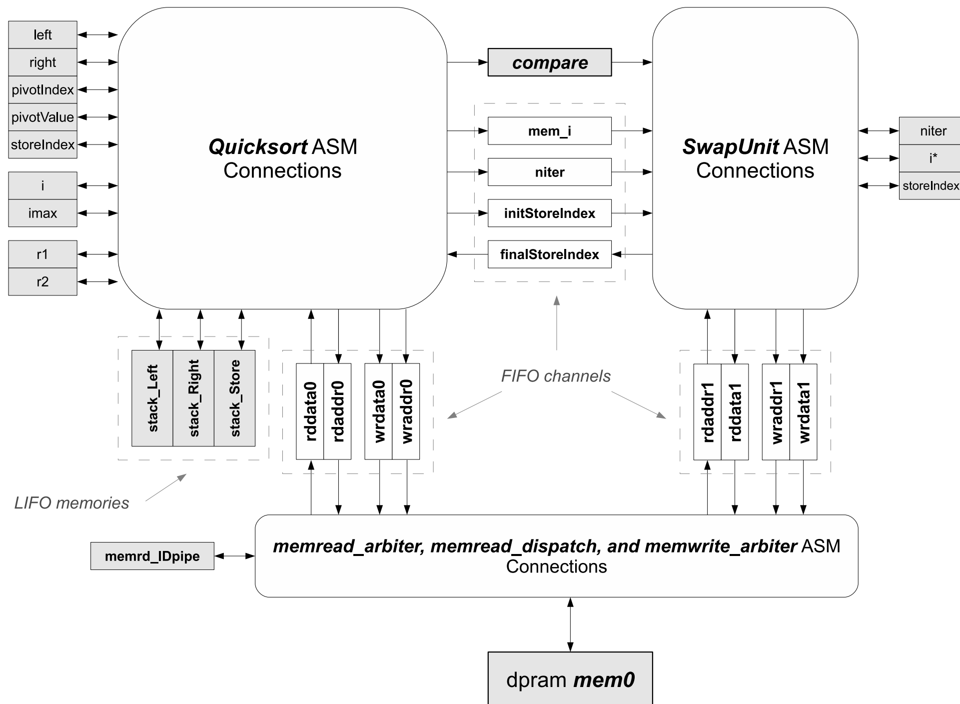


Figure 5.3 Représentation STL de l'architecture avec mémoire partagée.

partir de composants de la librairie LPM d'Altera.

Le Tableau 5.1 résume les temps d'exécutions et fréquences maximales pour les deux implémentations matérielles de l'algorithme *Quicksort*. Les fréquences maximales des deux implémentations sont presqu'identiques, à 325 MHz et 318 MHz. Les temps d'exécution ont été obtenus pour une séquence décroissante de 256 valeurs. Pour l'architecture à ASM simple 16223 cycles sont requis, contre 12796 pour l'architecture pipelinée. Le Tableau 5.2 présente une analyse des ressources utilisées par les différents modules des deux implémentations produites.

Tableau 5.1 Exécution des implémentations simple et à pipeline fonctionnel.

Implémentation	Fréquence d'horloge maximale (MHz)	Cycles d'exécution
simple	325	16223
pipeline fonctionnel	318	12796

5.2.4 Discussion

L'application de l'approche de niveau intermédiaire à la description de circuits dédiés réalisant l'algorithme de tri de données rapide permet d'apprécier comment celle-ci permet de simplifier

Tableau 5.2 Utilisation des ressources pour les deux implémentations.

Implémentation	Simple		Pipelinée	
	ALUTs	Registres	ALUTs	Registres
- Logique d'autorisation	43	0	69	0
- Réseau d'interconnexions	147	0	294	0
- ASM *QuickSort*	164	86	155	71
- ASM *SwapUnit*	–	–	27	10
- Composants interconnectés	184	270	403	965
Total	538	356	948	1046

la descriptions d'architectures pipelinées et la gestion de transferts entre des sources et des puits avec des interfaces à flot de données. Bien que les implémentations produites n'offrent pas un niveau de parallélisme élevé, elles permettent néanmoins d'évaluer le compilateur sur des architectures intégrant différents types de mémoires ainsi que des opérateurs pipelinés. Bien que l'architecture pipelinée requiert près de deux fois plus de ressources combinatoires et trois fois plus de registres, on observe que la partie contrôle (ASMs et logique d'autorisation des transferts) requiert un nombre similaire de ressources de part et d'autres. La logique de contrôle de l'architecture pipelinée requiert environ 10% plus de ressources que l'architecture de base.

5.3 Accumulateur pipeliné

Cette section présente l'application de la méthode de synthèse à niveau intermédiaire à la conception d'un accumulateur basé sur l'utilisation d'un additionneur pipeliné. L'accumulateur est un opérateur important qui intervient dans plusieurs applications de calculs scientifiques. Dans un premier temps, une implémentation est produite sans faire usage de l'opérateur de connexion différée [71],[72]. Dans un second temps, la même architecture est décrite à nouveau, mais en faisant cette fois usage de l'opérateur de connexion différée. Les deux implémentations sont comparées à une implémentation RTL d'une architecture presqu'identique.

5.3.1 Conception

L'architecture de l'accumulateur proposé est basée sur la méthode de réduction DB (*Delayed Buffering*) proposée dans [73]. La méthode de réduction DB requiert l'ajout d'un registre tampon et réalise les opérations d'addition dès que possible, entre les opérandes provenant de l'entrée de l'accumulateur, la sortie de l'additionneur, et la sortie du registre tampon. La

Figure 5.4 illustre l'architecture de l'implémentation proposée, basée sur 2 ASMs concurrentes qui gèrent l'interconnexion des sources et puits. L'ASM principale *main* est responsable d'alimenter l'additionneur et le registre tampon A en fonction de la disponibilité des données provenant des sources *source*, fb, et A. L'ASM $forward$ est responsable de rediriger les résultats à la sortie de l'additionneur vers l'étage d'entrée via le conduit de rétroaction fb, ou bien vers le port de sortie *result* de l'accumulateur.

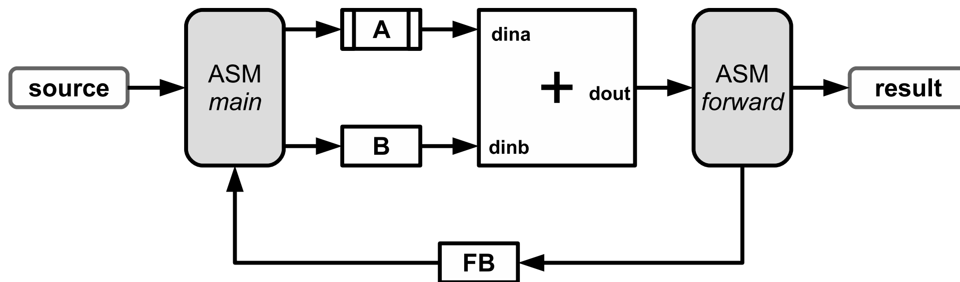


Figure 5.4 Représentation STL de l'architecture proposée.

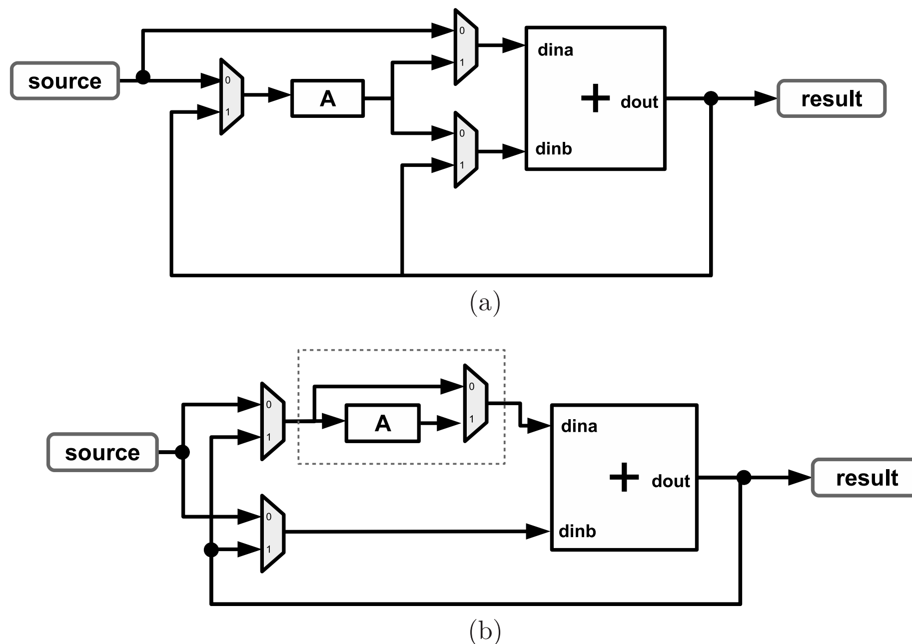


Figure 5.5 Comparaison des chemins de données pour l'accumulateur. (a) Chemin de données présenté dans [73]. (b) Chemin de données associé à l'architecture de la Figure 5.4.

5.3.2 Description sans opérateur de connexion différée

La Figure 5.6 illustre la description CASM+ d'un accumulateur pipeliné basé sur la méthode de réduction DB. La description fait uniquement appel aux opérateurs de connexion de base

(=,∗ =). L'ASM principale *main* est responsable d'alimenter l'additionneur et le registre tampon A en fonction de la disponibilité des données provenant des sources *source*, fb, et A. Dans l'état $N0$, les connexions $t1, t2, t3, t4$ représentent les différents cas de figures possibles. Des règles sont associées à ces connexions pour spécifier que chaque source (*source* et fb) peut seulement être envoyée vers un seul puits à la fois. À chaque fois que l'additionneur accepte une paire d'opérandes, un identifiant sera inscrit dans un registre à décalage de latence identique à celle de l'additionneur. L'identifiant sert à indiquer vers quelle destination envoyer le résultat à la sortie de l'additionneur. De plus, le registre *tokencount* est utilisé afin de maintenir le compte du nombre de données contenues dans l'accumulateur. Lorsque l'accumulateur reçoit la dernière donnée d'un vecteur à réduire, l'ASM accepte cette donnée puis passe à l'état $N1$. Dans l'état $N1$, les données à la source ne sont plus acceptées, et les résultats à la sortie de l'additionneur sont réacheminés vers l'étage d'entrée jusqu'à ce qu'il n'y ait plus qu'une seule donnée dans l'accumulateur. Cette dernière est acheminée vers le port de sortie *result*. L'ASM $forward$ est uniquement responsable de diriger les résultats à la sortie de l'additionneur vers l'une ou l'autre des deux destinations possibles, tel qu'indiqué à la sortie du registre à décalage $myAdderID$.

5.3.3 Description avec opérateurs de connexions différée

La Figure 5.7 illustre la description CASM+ d'un accumulateur pipeliné basé sur la méthode de réduction DB en faisant usage des opérateurs de connexion différée. La description est presqu'identique à celle basée uniquement sur les opérateurs de base qui vient d'être présentée. Cependant, la gestion des transferts de données dans différents domaines temporels est spécifiée à un niveau d'abstraction plus élevé au moyen d'une seule ASM. Le mécanisme permettant de diriger les résultats à la sortie de l'additionneur est implicite, et il n'y a plus besoin de spécifier le registre à décalage $myAdderID$. Plutôt que d'inscrire dans ce dernier la destination désirée au moment où les opérandes sont acceptées par l'additionneur, l'opérateur de connexion différée est utilisé permettant de spécifier explicitement les connexions désirées (source/puits plutôt qu'un identifiant).

5.3.4 Résultats de synthèse et d'implémentation

Afin de permettre une comparaison avec une implémentation RTL de la littérature ([73]), les résultats de synthèse et d'implémentation de l'accumulateur pipeliné ont été obtenus pour un FPGA de type Virtex-5 de Xilinx. Le Tableau 5.3 présente l'ensemble des résultats obtenus pour différentes implémentations de l'accumulateur proposé. Les résultats de synthèse pour l'accumulateur décrit uniquement avec des opérateurs de connexion de base ont été présentés

```
define DATAWIDTH=32;
signed input   source{protocol = "FS"}[DATAWIDTH];
signed input   last{protocol = "MAIN"}[1];
signed output  result{protocol = "FS"}[DATAWIDTH];
signed post    fb{protocol = "FS"}[DATAWIDTH];
signed queue   fifoA{size = 1, type = "fifo_wbypass"}[DATAWIDTH];
signed post    fifoB{protocol = "FS"}[DATAWIDTH];
device myAdder {filename = "fp_add_sp.asm", dwidth = DATAWIDTH, latency = 11};
signed register tokenCount[5];

always alw0 {
     myAdder.a *={ta} fifoA;
     myAdder.b *={tb} fifoB;
     ta <=> tb.transfer.fire;
}
asm front {
     switch (state)
     case N0:
          tokenCount *= 0;
          goto N1;
     case N1:
          fifoA *={t1} source;
          fifoA *={t2} fb;
          fifoB *={t3} source;
          fifoB *={t4} fb;
          t1 => (not t3.transfer.fire) and (not t2.transfer.fire);
          t3 => (not t1.transfer.fire) and (not t4.transfer.fire);
          t2 => (not t4.transfer.fire) and (not t1.transfer.fire);
          t4 => (not t2.transfer.fire) and (not t3.transfer.fire);
          tokenCount *={tinc} tokenCount + 1;
          tokenCount *={tdec} tokenCount - 1;
          tinc => !alw0.ta.transfer.fire & (t1.transfer.fire or t3.transfer.fire);
          tdec => alw0.ta.transfer.fire & !(t1.transfer.fire or t3.transfer.fire);
          if( last == 1 )
               myAdderID *={t1c} 0;
               t1c => alw0.ta.transfer.fire;
               goto N2;
          else
               myAdderID *={t1d} 0;
               t1d => alw0.ta.transfer.fire;
               goto N1;
          end;
     case N2:
          fifoA *={t6} fb;
          fifoB *={t7} fb;
          t6 => not t7.transfer.fire;
          t7 => not t6.transfer.fire;
          if( tokenCount == 2 )
               myAdderID ={t2c} 1;
               t2c => alw0.ta.transfer.fire;
               goto N0;
          else
               myAdderID *={t2d} 0;
               t2d => alw0.ta.transfer.fire;
               tokenCount *={t2dec} tokenCount - 1;
               t2dec => alw0.ta.transfer.fire;
               goto N2;
          end;
     end;
}

asm forward {
     if( myAdderID == 0 )
          fb *={t0} myAdder.dout;
          null *={t0id} myAdderID.dout;
     else
          result *={t1} myAdder.dout;
          null *={t1id} myAdderID.dout;
     end;
}
```

Figure 5.6 Description CASM+ d'un accumulateur pipeliné.

```
define DATAWIDTH=32;
signed input   source{protocol = "FS"}[DATAWIDTH];
signed input   last{protocol = "MAIN"}[1];
signed output  result{protocol = "FS"}[DATAWIDTH];
signed post    fb{protocol = "FS"}[DATAWIDTH];
signed queue   fifoA{size = 1, type = "fifo_wbypass"}[DATAWIDTH];
signed post    fifoB{protocol = "FS"}[DATAWIDTH];
device myAdder {filename = "fp_add_sp.asm", dwidth = DATAWIDTH, latency = 11};
signed register tokenCount[5];

always alw0 {
     myAdder.a *={ta} fifoA;
     myAdder.b *={tb} fifoB;
     ta <=> tb.transfer.fire;
}
asm front {
     switch (state)
     case N0:
          tokenCount *= 0;
          goto N1;
     case N1:
          fifoA *={t1} source;
          fifoA *={t2} fb;
          fifoB *={t3} source;
          fifoB *={t4} fb;
          t1 => (not t3.transfer.fire) and (not t2.transfer.fire);
          t3 => (not t1.transfer.fire) and (not t4.transfer.fire);
          t2 => (not t4.transfer.fire) and (not t1.transfer.fire);
          t4 => (not t2.transfer.fire) and (not t3.transfer.fire);
          tokenCount *={tinc} tokenCount + 1;
          tokenCount *={tdec} tokenCount - 1;
          tinc => !alw0.ta.transfer.fire & (t1.transfer.fire or t3.transfer.fire);
          tdec => alw0.ta.transfer.fire & !(t1.transfer.fire or t3.transfer.fire);
          if( last == 1 )
               fb ?*={t1c} myAdder.result;
               t1c => alw0.ta.transfer.fire;
               goto N2;
          else
               fb ?*={t1d} myAdder.result;
               t1d => alw0.ta.transfer.fire;
               goto N1;
          end;
     case N2:
          fifoA *={t6} fb;
          fifoB *={t7} fb;
          t6 => not t7.transfer.fire;
          t7 => not t6.transfer.fire;
          if( tokenCount == 2 )
               result ?={t2c} myAdder.result;
               t2c => alw0.ta.transfer.fire;
               goto N0;
          else
               fb ?*={t2d} myAdder.result;
               t2d => alw0.ta.transfer.fire;
               tokenCount *={t2dec} tokenCount - 1;
               t2dec => alw0.ta.transfer.fire;
               goto N2;
          end;
     end;
}
```

Figure 5.7 Description d’un accumulateur pipeliné avec connexions différées.

dans [72]. En considérant des ports d’entrée et de sortie avec des interfaces complètement synchronisées (FS), cette implémentation requiert 731 LUTs et 644 registres, et permet une fréquence d’horloge maximale de 315 MHz. Pour la même architecture décrite en faisant usage d’opérateurs de connexion différée, l’implémentation requiert 673 LUTs et 631 registres, et supporte également une fréquence d’horloge maximale de 315 MHz. Le fait de supporter

la contre-pression à la sortie de l'accumulateur contribue à augmenter la complexité de la logique de contrôle. En spécifiant une interface demi-synchronisée (HS) pour le port de sortie, la logique de contrôle ne dépend plus du signal d'entrée $myadder.dout.rtr$, ce qui permet une logique de contrôle simplifiée. En assumant que le port de sortie est toujours prêt à recevoir, l'implémentation utilise une quantité équivalente de ressources mais permet maintenant une fréquence d'horloge maximale de 357 MHz. La logique de contrôle associée à la synchronisation des données peut être simplifiée davantage si l'on considère que la source est toujours prête à envoyer une donnée (interface NS). Dans ce cas, l'implémentation résultante requiert toujours une quantité similaire de ressources, mais permet une fréquence d'horloge maximale de 368 MHz.

Tableau 5.3 Résultats de synthèse de l'accumulateur sur un FPGA Virtex-5

Implémentation	Caractéristiques			
	LUTs	Registres	DSPs	Fmax (MHz)
RTL (*design de référence*) - Tai et al. [73]	501	615	0	363
STL (*sans connexions différées + sortie FS*) - Daigneault and David [72]	731	644	0	315
STL (*avec connexions différées + sortie FS*) - [Ce travail]	673	631	0	315
STL (*avec connexions différées + sortie HS*) - [Ce travail]	687	632	0	357
STL (*avec connexions différées + entrée NS*) - [Ce travail]	662	632	0	368

5.3.5 Discussion

En comparaison, les deux implémentations de l'architecture proposée avec et sans opérateurs de connexion différée, considérant des interfaces complètement synchronisées, permettent d'obtenir des fréquences d'opérations maximales comparables. De même, la quantité de logique requise par les deux approches est presqu'équivalente. Il s'ensuit que les résultats montrent que l'utilisation des opérateurs de connexion différée est faite sans pénalité de performance au niveau de l'implémentation. En contrepartie, ils permettent de simplifier la description de l'accumulateur en faisant abstraction du mécanisme de redirection des données à la sortie de l'additionneur.

Les implémentations dotées d'interfaces complètement synchronisées à l'entrée et à la sortie de l'accumulateur supportent une fréquence d'horloge maximale de 315 MHz contre 363 MHz pour l'implémentation présentée dans [73]. Néanmoins, l'implémentation présentée dans [73] ne supporte pas de contre-pression à la sortie de l'accumulateur. En considérant que la

sortie de l'accumulateur est toujours prête à recevoir, des fréquences d'horloges maximales presqu'identiques sont obtenues. Ainsi, en spécifiant que le port de sortie de l'accumulateur utilise une interface demie-synchronisée HS, la fréquence d'horloge maximale obtenue grimpe à 357 MHz, une différence de 2% avec le design RTL présenté dans [73]. La fréquence d'horloge maximale grimpe davantage, jusqu'à 368 MHz, en considérant également que la source est toujours prête à envoyer une donnée.

5.4 Produit matriciel

Cette section présente l'application de la méthode de synthèse à niveau intermédiaire à la conception d'un circuit dédié pour le produit matriciel dense, ayant fait l'objet de publications dans [74] et [75].

5.4.1 Algorithme

Le produit matriciel est une opération fondamentale de l'algèbre linéaire et intervient dans de nombreux algorithmes de traitement et calculs scientifiques. Pour une multiplication de matrices de taille $n \times n$, n^2 produits scalaires doivent être réalisés, un pour chaque paire $(ligne, colone)$ possible. Il s'ensuit une complexité algorithmique $O(n^3)$. La Figure 5.8 illustre l'algorithme de produit matriciel $A \times B = C$.

```
 1: matmul(A, B, n)
 2: for i = 0 → n − 1 do
 3:     for j = 0 → n − 1 do
 4:         C[i][j] = 0 ;
 5:         for k = 0 → n − 1 do
 6:             C[i][j] += A[i][k]*B[k][j] ;
 7:         end for
 8:     end for
 9: end for
10: return C ;
```

Figure 5.8 Algorithme de produit matriciel $A \times B = C$.

5.4.2 Conception

On considère la conception d'un circuit dédié pour le produit matriciel en assumant que les matrices sont contenues dans une mémoire externe avec une bande-passante d'une donnée par cycle. Afin d'obtenir une réutilisation de données intéressante, des mémoires sur puce sont intégrées au circuit. L'architecture du circuit de calcul repose sur l'exécution parallèle

de N éléments de calcul spécialisés pour le produit scalaire. Chacun de ces éléments de calcul spécialisés contient une mémoire à double ports pouvant contenir une rangé complète de la matrice A. Après avoir chargé N rangées consécutives de A, les colonnes de la matrice B sont envoyées à tous les éléments de calculs. En sortie, les N rangées de C correspondantes sont obtenues.

La Figure 5.9 illustre l'architecture d'un élément de calcul spécialisé pour le produit scalaire, intégrant un multiplieur et un accumulateur simple précision (virgule flottante) pipelinés, ainsi que deux mémoires RAM sur puce pouvant contenir chacune une rangée complète de A ou de C. Des mémoires de type FIFO (de petites tailles) sont utilisées aux interfaces et au sein du chemin de données. La FIFO *last* est utilisée pour indiquer si la valeur à la sortie du multiplieur est la dernière de l'accumulation associée au produit scalaire.

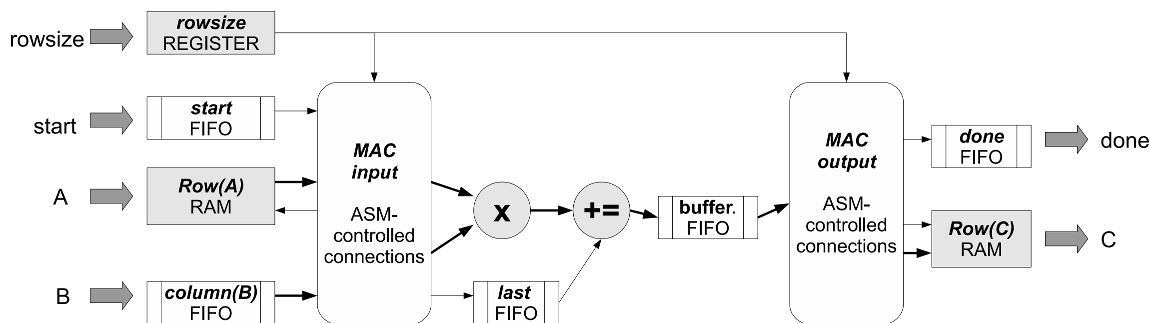


Figure 5.9 Représentation STL de l'architecture d'un élément de traitement.

La Figure 5.10 illustre la description CASM+ de l'ASM responsable d'alimenter le multiplieur d'un élément de calcul spécialisé pour le produit scalaire. L'ASM $MACinput$ réalise deux boucles imbriquées de taille n ($rowsize$). Pour chaque colonne de B reçue, la rangée de A contenue en mémoire est lue complètement. La sortie de la mémoire $memA$ est connectée en permanence à l'entrée $mul.a$ du multiplieur. La sortie de la FIFO $columnB$ recevant les éléments de la colonne de B est connectée en permanence à l'entrée $mul.b$ du multiplieur. L'état $N2$ est associé à la boucle d'itération interne de l'algorithme, et est responsable d'indiquer si la paire d'opérandes à multiplier correspond à la dernière pour une rangée et une colonne données.

La Figure 5.11 illustre la description de l'ASM $MACoutput$ qui gère l'écriture des résultats à la sortie de l'accumulateur pipeliné vers la mémoire RAM sur puce associée à une rangée de C. L'algorithme présente une boucle itérative de n itérations, pour l'écriture de chacun des n résultats à la sortie de l'accumulateur à chaque fois que toutes les n colonnes de B sont lues complètement. Le processus de type *always* décrit un ensemble de connexions non-bloquantes activées en permanence.

```
1:  device mul {type="fp_mul", protocol = "FS"}[32] ;
2:  device acc {type="user_acc"} ;
3:  device memA {type="dp_ram", ... }[32] ;
4:  queue columnB {type="fifo",size=8}[32] ;
5:  queue last,start {type="fifo",size=8}[1] ;
6:  register i,j,rowsize[11] ;
7:  asm MACinput {
8:      {ta} mul.a *= memA.rddata ;
9:      {tb} mul.b *= columnB ;
10:     ta <=> tb.fire ;
11:     {tc} acc.din *= mul.result ;
12:     {td} acc.last *= last ;
13:     tc <=> td.fire ;
14:     switch( state )
15:     case N0 :
16:       {t01} null = start ;
17:       {t02} j *= 0 ;
18:       goto N1 ;
19:     case N1 :
20:       if( j < rowsize )
21:           {t11} i *= 0 ;
22:          goto N2 ;
23:       else ;
24:          goto N0 ;
25:       end ;
26:     case N2 :
27:       if( i < rowsize )
28:          if( i == (rowsize-1) )
29:              {t21} memA.rdaddr *= i0 ;
30:              {t22} i = i + 1 ;
31:              {t23} last = 1 ;
32:              t21 <=> t22.fire ;
33:          else ;
34:              {t24} memA.rdaddr *= i0 ;
35:              {t25} i = i + 1 ;
36:              {t26} last = 0 ;
37:              t24 <=> t25.fire ;
38:          end ;
39:          goto N2 ;
40:       else ;
41:          {t27} j *= j + 1 ;
42:          goto N1 ;
43:       end ;
44:     end ; }
```

Figure 5.10 Description CASM+ de l'ASM $MACinput$.

La Figure 5.12 illustre l'architecture du circuit spécialisé pour le produit matriciel dense. L'architecture interconnecte les entrées et sorties de N éléments de calculs de produit scalaire au moyen d'un réseau d'interconnexions pipeliné complètement synchronisés. Le réseau d'interconnexion pipeliné à l'étage d'entrée permet de diriger vers chacun des éléments de

```
1:  always {
2:      acc.din *={tc} mul.R ;
3:      acc.last *={td} lastQueue ;
4:      outputbuffer *={te} acc.dout ;
5:      rowSizeRegister *=tf rowSize ;
6:      tc <=> td.fire ;
7:  }
8:  asm MACoutput {
9:      switch( state )
10:     case N0 :
11:       i1 *={t01} 0 ;
12:       goto N1 ;
13:     case N1 :
14:       if( i1 < rowSizeRegister )
15:           memC.wraddr *={t21} i1 ;
16:           rowC.din *={t22} outputBuff ;
17:           i1 ={t23} i1 + 1 ;
18:           t21 <=> t22.fire ;
19:           t21 <=> t23.fire ;
20:         goto N1 ;
21:       else ;
22:         goto N2 ;
23:       end ;
24:     case N2 :
25:         done ={t31} 1 ;
26:         goto N0 ;
27:     end ;
28: }
```

Figure 5.11 Description CASM+ de l'ASM *MACoutput*.

calculs les données et arguments qui leur sont propres (rangées de A), ainsi que de diffuser simultanément vers tous les éléments de calculs les colonnes de la matrice B. Le réseau à l'étage de sortie est responsable d'acheminer les valeurs lues des mémoires contenant les rangées de C. La pertinence d'utiliser un réseau pipeliné est de supporter des fréquences d'horloges maximales intéressantes lorsque des quantités appréciables d'éléments de calculs sont considérées. La Figure 5.13 donne une représentation au niveau des ASMs de l'architecture du réseau utilisé pour envoyer des données vers un des différents éléments de calculs. Un réseau considérant 4 éléments de calculs spécialisés est illustré. La description CASM+ associée à un élément d'aiguillage particulier, *sw_0_0*, est donnée à la Figure 5.14. L'ASM responsable de l'aiguillage réalise un test sur la valeur du champ de sélection de rangées pour déterminer vers quelle FIFO envoyer les données.

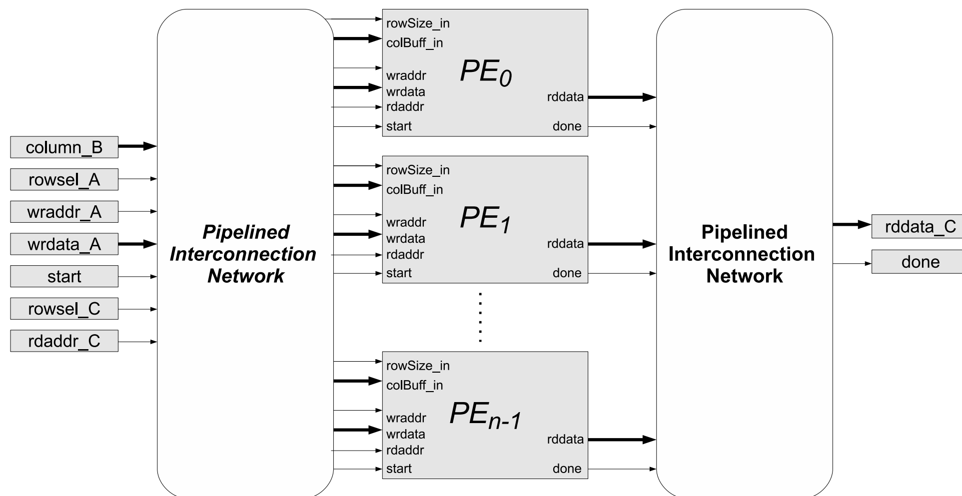


Figure 5.12 Architecture intégrant de multiples unités de traitements.

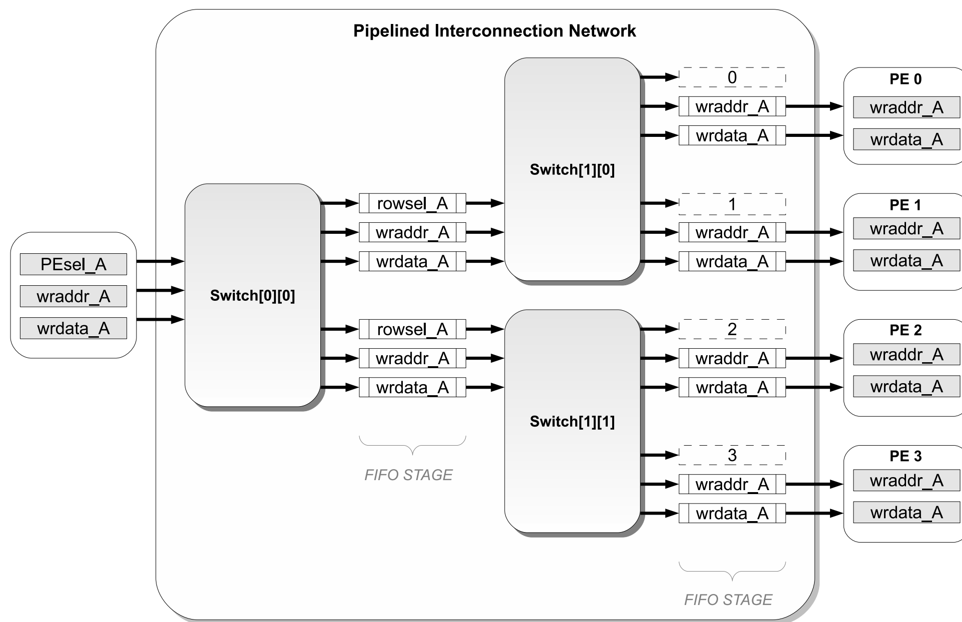


Figure 5.13 Représentation STL d'un réseau d'interconnexions pipeliné.

5.4.3 Résultats de synthèse et d'implémentation

Les résultats de synthèse et d'implémentation de l'architecture spécialisée pour le produit matriciel sont obtenus pour les FPGAs de type Stratix-III d'Altera et Virtex-5 de Xilinx. Les opérateurs arithmétiques utilisent une représentation en virgule flottante fonctionnent avec une précision simple. Le Tableau 5.4 présente les ressources utilisées en termes de LUTs, registres, DSP, et mémoires RAM, pour l'architecture avec un nombre d'éléments de calculs allant de 1 à 56 ciblant un FPGA de type Stratix-III. La quantité de ressources utilisées pour

```
1: queue A_rowselq_0_3 {type="fifo", size=1}[8];
2: queue A_wraddrq_0_3 {type="fifo", size=1}[10];
3: queue A_wrdataq_0_3 {type="fifo", size=1}[32];
4: queue A_rowselq_0_1 {type="fifo", size=1}[8];
5: // ...
6:
7: asm {
8: if( A_rowselq_0_3.dout.main < 2 )
9:   {t1} A_rowselq_0_1 *= A_rowselq_0_3;
10:  {t2} A_wraddrq_0_1 *= A_wraddrq_0_3;
11:  {t3} A_wrdataq_0_1 *= A_wrdataq_0_3;
12:  t1 <=> t2.fire;
13:  t1 <=> t3.fire;
14: else
15:  {t4} A_rowselq_2_3 *= A_rowselq_0_3;
16:  {t5} A_wraddrq_2_3 *= A_wraddrq_0_3;
17:  {t6} A_wrdataq_2_3 *= A_wrdataq_0_3;
18:  t4 <=> t5.fire;
19:  t4 <=> t6.fire;
20: end;
21: goto N0;
22: }
```

Figure 5.14 Description CASM+ d'un réseau d'interconnexions pipeliné.

un seul élément de calcul s'élève à 1,3k LUTs, 2,3k registres, 4 blocs DSP, et 65,5kbits de mémoire RAM. Le Tableau 5.5 présente les ressources utilisées pour l'architecture avec un nombre d'éléments de calculs allant de 1 à 56 ciblant un FPGA de type Virtex-5. La quantité de ressources utilisées pour un seul élément de calcul s'élève à 1,3k LUTs, 1,2k registres, 4 blocs DSP, et 2 blocs RAM.

Le Tableau 5.6 présente les fréquences d'horloges maximales et le nombre d'opérations par secondes pour les différentes implémentations de l'architecture sur le Stratix-III et le Virtex-5. Il est observé que la fréquence maximale d'horloge n'est pas significativement affectée par l'augmentation du nombre d'éléments de calculs de produit scalaire pour n variant de 16 à 56. Sur le Stratix-III, le nombre d'opérations par secondes de pointe varie de 7.68 GFLOPS à 27 GFLOPS pour n variant de 16 à 56. Sur le Virtex-5, le nombre d'opérations par secondes de pointe est similaire variant de 7.55 GFLOPS à 25.54 GFLOPS.

5.4.4 Discussion

Dans la littérature, plusieurs travaux considèrent l'implémentation FPGA de l'algorithme de multiplication matricielle dense. Le Tableau 5.7 résume différentes implémentations FPGA du produit matriciel à l'état de l'art, permettant une comparaison avec les résultats obtenus pour l'implémentation proposée dans ce travail. Toutes les implémentations mentionnées,

Tableau 5.4 Utilisation des ressources pour le produit matriciel (Stratix-III)

Élements de calcul (PEs)	LUTs	Ressources FPGA Registres	DSPs (blocs)	RAMs (bits)
1	1.3 (1%)	2.3 (1%)	4 (1%)	65.5k (<1%)
16	27.4k (13%)	43.8k (22%)	64 (8%)	1.1M (7%)
24	41.1k (20%)	65.3k (32%)	96 (13%)	1.6M (11%)
32	56.7k (28%)	79.9k (39%)	128 (17%)	2.1M (14%)
40	66.7k (33%)	113.1k (56%)	160 (21%)	2.7M (18%)
56	93.1k (46%)	157.9k (78%)	224 (21%)	3.7M (25%)

Tableau 5.5 Utilisation des ressources pour le produit matriciel (Virtex-5)

Éléments de calcul (PEs)	LUTs	Ressources FPGA Registres	DSPs (blocs)	RAMs (blocs)
1	1.3k (<1%)	1.2 (<1%)	4 (<1%)	2 (<1%)
16	31.7k (21%)	26.2k (17%)	64 (6%)	33 (6%)
24	57.1k (38%)	49.5k (33%)	96 (9%)	49 (9%)
32	77.4k (51%)	65.6k (43%)	128 (12%)	65 (12%)
40	96.2k (64%)	82.2k (54%)	160 (15%)	81 (15%)
56	132.4k (88%)	114.7k (76%)	224 (21%)	113 (21%)

Tableau 5.6 Performances pour le produit matriciel sur FPGA.

Éléments de calcul (PEs)	Stratix-III		Virtex-5	
	Fmax	GFLOPS	Fmax	GFLOPS
1	287	0.57	253	0.51
16	240	7.68	236	7.55
24	241	11.57	240	11.52
32	230	14.72	225	14.4
40	225	18	224	17.92
56	241	27	228	25.54

à l'exception de [76] et [77], considèrent une représentation en virgule-flottante à simple précision. On remarque que les implémentations FPGA du produit matriciel sur des FPGAs Virtex-5 et Stratix-III permettent d'atteindre des performances comprises entre 10 et 30 GFLOPS.

En comparaison avec l'implémentation sur un Virtex-5 rapportée dans [78], l'implémentation proposée dans ce travail avec 32 éléments de calculs supporte une fréquence d'opération maximale supérieure de 12%. De même, le taux d'opérations par seconde est aussi supérieur, avec 14,4 GFLOPS contre 12,8 GFLOPS. En comparaison avec l'implémentation sur un Stratix-III rapportée dans [79], l'implémentation proposée dans ce travail avec 40 éléments de calculs supporte également une fréquence d'opération maximale supérieure de 12%. Le taux d'opé-

rations par seconde est de 18 GFLOPS contre 16 GFLOPS. L'architecture rapportée dans [77] offre des performances significativement supérieures, intégrant jusqu'à 40 éléments de calculs opérant à 372 MHz, et fonctionnant en double précision. Dans ce travail, l'algorithme de produit matriciel utilisé est différent, basé un produit vectoriel plutôt que sur produit scalaire. Une telle formulation de l'algorithme permet à chaque élément de calcul d'exécuter plusieurs accumulations indépendantes entrelacées. L'avantage majeur de cette approche est que l'entrelacement de multiples accumulations indépendantes permet de cacher la latence de l'additionneur. Le circuit d'accumulation se ramène ainsi à celui d'un additionneur qui offre une fréquence d'horloge maximale plus élevée. Cela contraste avec l'accumulateur basé sur la méthode de réduction DB utilisé dans l'architecture proposée, qui permet d'accumuler une nouvelle donnée d'un même vecteur de réduction à chaque cycle d'horloge. Néanmoins, l'utilisation d'un tel accumulateur permet une description simple de l'algorithme de produit matriciel basé sur une formulation de type produit scalaire.

Tableau 5.7 Comparaison d'implémentations dédiées au produit matriciel.

Travail	Niveau d'abstraction	Nombre de PE	Fmax	GFLOPS	Type de FPGA
Zhuo and Prasanna [76]	RTL	8	155	2.48	Virtex-II-50
Govindu et al. [80]	RTL	–	240	19.6	Virtex-II-125
Kumar et al. [77]	RTL	19	373	14.2	Virtex-V-95
Kumar et al. [77]	RTL	40	372	29.8	Virtex-V-240
Jiang et al. [78]	RTL	32	200	12.8	Virtex-V
Holanda et al. [79]	RTL	40	200	16	Stratix-III-260
Daigneault and David [75]	STL	32	225	14.4	Virtex-V-240
Daigneault and David [75]	STL	40	225	18	Stratix-III-260
Daigneault and David [75]	STL	56	228	25.5	Virtex-V-240
Daigneault and David [74]	STL	56	240	26.9	Stratix-III-260

Bien que l'approche de synthèse et de description à niveau intermédiaire proposée ne remplace pas la capacité du concepteur à traduire un algorithme en une architecture pipelinée et parallèle, elle permet à ce dernier d'exprimer rapidement, clairement et surement l'architecture qui est désirée. Le compilateur CASM+ développé gère la génération de la logique de contrôle bas-niveau supportant la synchronisation des transferts de données sur des connexions entre des sources et des puits avec des interfaces à flot de données. Tout en offrant un niveau de conception supérieur au niveau RTL, les résultats présentés pour l'implémentation d'une architecture dédiée pour le produit matriciel montre que des résultats comparables avec ceux rapportés à l'état de l'art peuvent être obtenus.

5.5 Élimination Gaussienne

Cette section présente l'application de la méthode de synthèse à niveau intermédiaire à la conception de circuits dédiés pour l'algorithme d'élimination Gaussienne. Deux architectures sont présentées, supportant deux modes de pivotement différents. Ces deux architectures considèrent que la matrice qui représente le système d'équation est contenue dans une mémoire externe avec une bande passante limitée, et sont basées sur la mise en série de plusieurs répliques identiques d'un élément de traitement spécialisé. L'implémentation de cet élément est comparée à une implémentation RTL d'un élément de traitement similaire. L'implémentation est également comparée à des implémentations de l'algorithme obtenues au moyen d'un outil de synthèse à haut niveau commercial.

5.5.1 Algorithme

L'algorithme d'élimination Gaussienne permet de réaliser la résolution de systèmes d'équations linéaires et l'inversion de matrices. Dans cette section, l'élimination est appliquée à la résolution de systèmes d'équations linéaires. La Figure 5.15 illustre une description de l'algorithme considéré. La matrice augmentée de taille $m \times n$ associée au système d'équation est contenue dans un tableau A à deux dimensions. Pour chaque passe de réduction k, une ligne de pivot non-nul doit être sélectionnée. Selon le mode de pivotement considéré, l'indice de la première ligne de pivot non-nul, ou l'indice de la ligne ayant le pivot maximal (en valeur absolue) est retourné. Par la suite, la ligne de pivot est permutée avec la ligne k du système, avant d'être normalisée. Après, chaque ligne i ($i > k$) en dessous de la ligne de pivot est transformée selon une combinaison linéaire sur les éléments j ($j >= k$) des lignes i et k.

```
1:  for k = 1 to m do
2:      i_pivot = getPivotRowIndex(k)
3:      swap(i_pivot,k)
4:      normalize(k)
5:      for i = k+1 to m do
6:          for j = k+1 to n do
7:              A[i,j] = A[i,j]-A[k,j]*A[i,k]
8:          end for
9:          A[i,k] = 0
10:     end for
11: end for
```

Figure 5.15 Algorithme d'élimination Gaussienne.

5.5.2 Conception

Afin de produire une implémentation permettant un haut niveau de parallélisme dans un contexte où le système entier ne peut être contenu dans la mémoire RAM sur puce du FPGA, l'architecture proposée est associée au *pipelinage* de la boucle d'itération externe (en k) de l'algorithme de la Figure 5.15. Chaque élément de traitement réalise une itération du corps de la boucle d'itération externe. L'architecture proposée permet de mettre en chaîne plusieurs de ces éléments de traitement afin de permettre l'exécution parallèle (pipelinée) de plusieurs itérations consécutives de la boucle d'itération externe. La Figure 5.16 illustre un exemple considérant une matrice augmenté de dimension 4×5 et une implémentation intégrant 2 éléments de calculs. L'élimination Gaussienne du système requiert deux passes de lecture/écriture du système complet ou en partie. Lors de la première passe, le système est lu entièrement, et à la sortie du circuit de calcul deux colonnes consécutives ont été réduites. Lors de la deuxième passe, seulement un sous-ensemble de la matrice A doit être chargée. La portion du système A qui est lue/écrite en mémoire à chaque passe est indiquée par un rectangle pointillé.

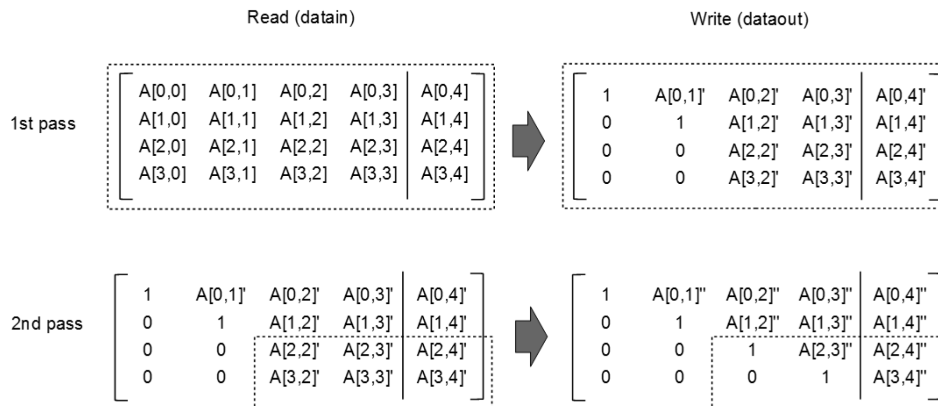


Figure 5.16 Illustration des accès mémoire sur deux passes consécutives.

Design-I

La Figure 5.17 illustre le chemin de données abstrait au niveau des ASMs de l'élément de traitement permettant de traiter en flot les éléments de la matrice A. L'élément de traitement est composé de trois ASMs concurrentes, intègre 3 opérateurs pipelinés travaillant en représentation virgule-flottante avec précision simple, ainsi qu'une FIFO basée sur une mémoire RAM permettant de contenir une ligne entière de la matrice augmentée A. L'élément de traitement reçoit le système (complet ou partiel) contenu dans A ligne par ligne. La première

ligne reçue n'ayant pas de pivot nul est retenue comme ligne de pivot. Tant qu'une ligne de pivot non-nul n'est pas trouvée, les lignes de pivot nul reçues sont envoyées directement vers la sortie, comme si elles avaient été réduites. Lorsqu'une première ligne de pivot non-nul est finalement détectée, la valeur du pivot est mémorisée dans un registre et la ligne entière est mémorisée dans la FIFO *ith_line*. Pour chacune des lignes j suivantes, une combinaison linéaire de la ligne j avec la ligne de pivot donne la nouvelle ligne j à produire en sortie. La ligne de pivot sera envoyée à la sortie à la toute fin, après toutes les autres.

L'élément de traitement illustré à la Figure 5.17 a été intégré au sein d'un coprocesseur doté d'interfaces mémoires de type maître et esclave. L'architecture du coprocesseur est illustrée à la Figure 5.18. Les multiples instances de l'élément de traitement sont reliées en chaîne. L'interface esclave permet de charger les arguments requis par les différentes instances de l'élément de traitement, ainsi que de commander l'exécution de l'algorithme. L'interface maître permet au coprocesseur d'accéder directement à la mémoire externe lors de l'exécution de l'algorithme. Le fonctionnement du coprocesseur a été validé au sein d'un système sur puce basé sur un processeur d'instruction Nios-II d'Altera et utilisant une mémoire SDRAM externe. Ce premier design ainsi que son implémentation FPGA ont fait l'objet d'une publication dans [81].

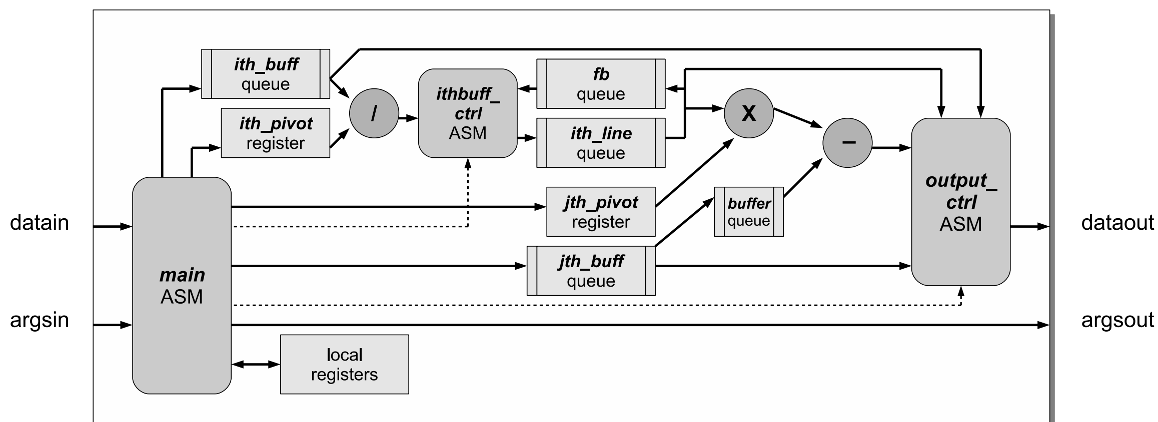


Figure 5.17 Architecture d'une unité de traitement (type-I).

Design-II

La deuxième architecture pour l'élimination Gaussienne réalise le pivotement en cherchant le pivot maximal (en valeur absolue) sur une colonne donnée. Contrairement à la première architecture, celle-ci reçoit la matrice A colonne par colonne, plutôt que ligne par ligne. L'architecture intégrant plusieurs éléments de traitements offre un débit atteignant jusqu'à 1

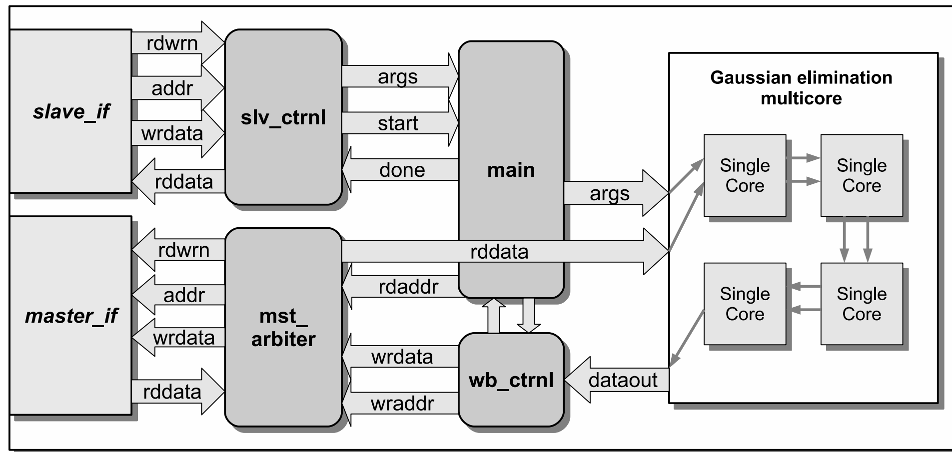


Figure 5.18 Architecture intégrant de multiples unités de traitements en série.

donnée par cycle. Considérant N éléments, à chaque fois que le système A (ou sous-système de A) est lu complètement, N colonnes consécutives de la matrice A sont réduites (i.e. : N itérations consécutives en terme de k dans l'algorithme de la Figure 5.15).

La Figure 5.19 illustre le chemin de données abstrait au niveau des ASMs de l'élément de traitement permettant de traiter en flot de données les éléments de la matrice A. Les entrées et sorties complètement synchronisées *datain* et *dataout* permettent de convoyer les colonnes de la matrice A. Les entrées et sorties complètement synchronisées $argsin$ et $argsout$ permettent de convoyer les arguments et paramètres d'opération. L'ASM *loader* est responsable d'identifier le pivot maximal et l'index de la ligne correspondante. Cette identification ne s'applique qu'à une seule colonne. Toutes les colonnes sont convoyées au fur et à mesure vers l'ASM *swapper* via des FIFOs. La FIFO col_buff permet de mémoriser une colonne entière de la matrice A. Cette mémoire permet de tamponner les valeurs de la colonne k pendant la recherche du pivot maximal. L'ASM *swapper* est responsable de réaliser la permutation des lignes, en réalisant une permutation de deux éléments sur chaque colonne. Les colonnes après permutation sont dirigées vers l'ASM *main*, qui est responsable de réaliser les combinaisons linéaires sur les différents éléments de chaque colonne. Pour les éléments de chaque colonne au-dessus de la ligne pivot, aucune opération n'est nécessaire. Pour un élément de la ligne pivot, une simple normalisation par rapport au pivot de la ligne (p_i) a lieu. Pour les éléments j en-dessous de la ligne de pivot, la combinaison linéaire $e'_j = e_j - (p_j/p_i) \times e_i$ est appliquée, où e_j représente l'élément j d'une colonne, p_j le pivot de la ligne j, e_i l'élément i d'une colonne, et p_i est le pivot de la ligne i.

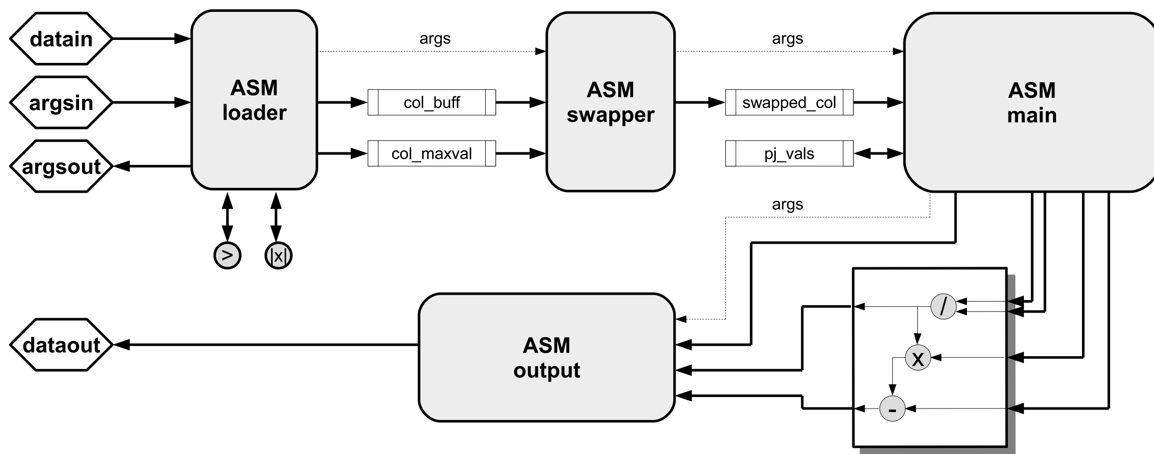


Figure 5.19 Architecture d'une unité de traitement (type-II).

5.5.3 Résultats de synthèse et d'implémentation

Design-I

Le Tableau 5.8 détaille les ressources utilisées par les différents modules de l'élément de calcul spécialisé de la première implémentation. Les opérateurs de soustraction, de multiplication, et de division représentent environ la moitié des éléments logiques et le tiers des registres de l'élément de calcul. Au niveau du contrôle, la logique d'autorisation des transferts et les ASMs représentent 12% des éléments logiques et 4% des registres utilisés. Presque la totalité de la mémoire RAM sur puce est utilisée par la FIFO pouvant contenir une ligne complète de la matrice A. Environ 10% de la mémoire sur puce est utilisée par l'opérateur de division en virgule flottante à précision simple. Selon un modèle considérant des délais courts (*fast timing model*), l'élément a une fréquence d'horloge maximale de 464 MHz sur un FPGA Stratix-V.

Le Tableau 5.9 résume les résultats de synthèse et d'implémentation de l'architecture intégrant de 8 à 64 éléments de calculs spécialisés ciblant un FPGA Stratix-V. De 8 à 48 éléments, la fréquence d'horloge diminue de 410 MHz à 389 MHz. Une baisse de 389 MHz à 361 Mhz est observée entre 48 et 64 éléments. Le Tableau 5.10 résume les résultats de synthèse et d'implémentation de l'architecture intégrant 1 et 32 éléments de calculs spécialisés en ciblant un FPGA Virtex-5. On observe une diminution de 5% au niveau de la fréquence d'horloge maximale pour une augmentation de 1 à 32 éléments de calculs, soit de 263 MHz à 250 MHz.

Le Tableau 5.11 résume les résultats d'efficacité au niveau des calculs. Pour une efficacité de 100%, l'élément de calcul peut accepter une nouvelle donnée provenant de la matrice à chaque cycle d'horloge. Dans les implémentations produites, les en-têtes de boucles itératives et autres préviennent l'obtention d'une telle efficacité, néanmoins des efficacité entre 70,6%

et 97,5% ont été mesurées pour des dimensions de 16 à 256.

Tableau 5.8 Utilisation des ressources pour un unité de type-I (Stratix-V)

Composant	LC Comb.	LC Reg.	DSPs (blocs)	RAMs (bits)
User Registers	0	256	0	0
User Queues	196	530	0	0
User FIFO-ram	141	109	0	36.8k
FP-sub	601	768	0	165
FP-mul	156	457	1	99
FP-div	206	913	5	4.8k
Interconnect	193	0	0	0
Authorization logic	110	0	0	0
Main ASM	87	54	0	0
Ith-line buffer ASM	18	17	0	0
Output control ASM	34	25	0	0
Others	353	29	0	363
Total	2095	3158	6	42.3k

Tableau 5.9 Utilisation des ressources pour l'accélérateur type-I (Stratix-V)

Éléments	Caractéristiques				
de calcul	ALUTs	Registres	DSPs (blocs)	RAMs (Mbits)	Fmax (MHz)
8	13k	25k	52	0.3	410
16	24k	48k	100	0.7	410
32	49k	95k	196	1.4	400
48	73k	140k	292	2.0	389
64	98k	187k	388	2.7	361

Tableau 5.10 Utilisation des ressources pour l'accélérateur type-I (Virtex-5)

Éléments	Caractéristiques				
de calcul	ALUTs	Registres	DSPs (blocs)	RAMs (blocs)	Fmax (MHz)
1	3.1k	3.4k	4	1	263
32	99k	110k	314	32	250

Design-II

Le Tableau 5.12 détaille les ressources utilisées par les différents modules de l'élément de calcul spécialisé de la première implémentation. Les opérateurs de soustraction, de multiplication, et de division représentent environ 25% des éléments logiques et 37% des registres de l'élément de calcul spécialisé. Au niveau du contrôle, la logique d'autorisation des transferts et les

Tableau 5.11 Efficacité des calculs d'un unité de traitement(type-I)

Nb. rangées de la matrice (N)	16	32	64	128	256
Efficacité (%)	70.6	82.8	90.7	95.0	97.5

ASMs représentent 11% des éléments logiques et 3% des registres utilisés. Presque la totalité de la mémoire RAM sur puce est utilisée par les deux FIFOs pouvant contenir une colonne complète de la matrice A. Selon modèle considérant des délais courts (*fast timing model*), l'élément a une fréquence d'horloge maximale de 436 MHz sur un FPGA Stratix-V.

Le Tableau 5.13 résume les résultats de synthèse obtenus sur un FPGA Stratix-V d'Altera. Les fréquences d'horloges rapportées sont obtenues avec des modèles considérant des délais longs/courts (*slow/fast timing models*). En considérant le modèle de délais lents, une diminution de l'ordre de 10% de la fréquence maximale est observée pour un passage de 8 à 48 unités de traitement parallèles. Cette diminution est de 4% pour un passage de 8 à 32 unités de traitement parallèles. Les résultats de synthèse sur un Virtex-5 sont résumés dans Le Tableau 5.14. La fréquence maximale pour une seule unité de traitement est de 225 MHz. Un coprocesseur intégrant 32 de ces coeurs de calculs atteint 210 MHz, pour une diminution de la fréquence maximale de l'ordre de 7%. Le Tableau 5.15 rapporte l'efficacité des calculs en termes des taux d'utilisation des opérateurs pipelinés. Pour des matrices comptant de 16 à 256 rangées, l'efficacité mesurée est de 79,7% à 98,6%.

Tableau 5.12 Utilisation des ressources pour un unité de type-II (Stratix-V)

Composant	LC Comb.	LC Reg.	DSPs (blocs)	RAMs (bits)
User Registers	370	800	0	0
User Queues	674	1239	0	0
User FIFO-ram	288	178	0	67.5k
FP-sub	635	655	0	294
FP-mul	174	424	1	139
FP-div	256	725	5	5k
FP-compare	109	2	0	0
Interconnect	331	0	0	0
Authorization logic	305	0	0	0
Loader ASM	96	58	0	0
Main ASM	46	38	0	0
Swapper ASM	45	35	0	0
Sequencer ASM	24	13	0	0
Others	-	-	0	0
Total	4639	4829	6	73k

Tableau 5.13 Utilisation des ressources pour l'accélérateur de type-II (Stratix-V)

Éléments de calcul	Caractéristiques				
	ALMs	Registres	DSPs (blocs)	RAMs (Mbits)	Fmax (MHz)
8	25.7k	37.9k	48	0.6	291/427
16	51.8k	75.8k	96	1.2	278/401
32	102.3k	151.6k	192	2.4	281/401
48	147.8k	224.8k	288	3.6	262/364

Tableau 5.14 Utilisation des ressources pour l'accélérateur de type-II (Virtex-5)

Éléments de calcul	Caractéristiques				
	LUTs	Registres	DSPs (blocs)	RAMs (blocs)	Fmax (MHz)
1	5.4k	5.4k	5	3	225
32	172k	170k	160	97	210

5.5.4 Discussion

D'autres implémentations FPGA de circuits dédiés à l'algorithme d'élimination Gaussienne sont présentes dans la littérature à l'état de l'art [82, 83, 84]. Toutes ces implémentations décrites au niveau RTL considèrent des matrices pouvant être entièrement mémorisées dans la mémoire RAM sur puce du FPGA. Cela permet d'exploiter une bande-passante mémoire plus grande au moyen d'une architecture SIMD dotée d'un seul contrôleur pour l'ensemble des opérateurs parallèles. Cela contraste avec les implémentations proposées dans ce travail qui requièrent plusieurs contrôleurs indépendants afin de pipeliner les opérations dans un contexte où la bande-passante mémoire est limitée. Dans [82], un circuit spécialisé pour l'inversion de matrice au moyen de l'algorithme Gauss-Jordan est présentée. L'architecture intègre des opérateurs spécialisés en précision double, et supporte une fréquence d'horloge maximale de 270 MHz. En considérant un seul élément de calcul, celle-ci est similaire à la première proposée dans ce travail. Le Tableau 5.16 compare les résultats de synthèse et d'implémentation pour ces architectures avec un seul élément de calcul. Bien que les opérateurs des deux implémentations soient significativement différents, on observe de part et d'autre des fréquences d'horloge maximales équivalentes. Puisque la fréquence d'opération maximale est liée à la logique de contrôle supportant le chemin de données, ces résultats

Tableau 5.15 Efficacité des calculs d'un unité de traitement (type-II)

Nb. rangées de la matrice (N)	16	32	64	128	256
Efficacité (%)	79.7	90.9	95.0	97.3	98.6

montrent que l'approche de synthèse à niveau intermédiaire proposée permet de générer un circuit de contrôle de qualité comparable à une implémentation RTL à l'état de l'art.

Tableau 5.16 Comparaison des unités de traitement (Virtex-5)

Implémentation simple coeur	Caractéristiques				
	LUTs	Registres	DSPs (blocs)	RAMs (blocs)	Fmax (MHz)
Duarte et al. [82]	4k	3.9k	41	-	270
Design-I	3.1k	3.4k	4	1	263
Design-II	5.4k	5.4k	5	3	225

Dans un deuxième temps, le deuxième design (Design-II) proposé dans ce travail a été comparée avec une implémentation matérielle de l'algorithme au moyen de l'outil de synthèse C commercial Vivado HLS. Comme point de départ, l'algorithme d'élimination Gaussienne de référence a été employé, conjointement avec la spécification de différentes directives particulières au compilateur. L'interface de l'implémentation matérielle ciblée dispose d'un port maître dans un système adressable (*memory-mapped*). La première optimisation appliquée consiste à spécifier un pipeline au niveau de la boucle imbriquée intérieure (*ref-alg-opt1*). La deuxième optimisation consiste à spécifier un tampon mémoire pour mémoriser la colonne de pivot qui est réutilisée à chaque itération de la boucle extérieure à la boucle pipelinée la plus intérieure. Il a également été spécifié explicitement au compilateur que les itérations de la boucle la plus intérieure sont indépendantes (*ref-alg-opt2*). Par la suite, une description C de notre unité de traitement (sérialisable) à flot de données a été produite. L'implémentation matérielle cible des interfaces de type flot de données. La description C de cet unité fait appel à 3 fonctions pipelinées (max/swap/reduce) correspondant aux principales ASMs de l'architecture illustrée à la Figure 5.19 (*st-sc-alg*). Le design a également été optimisé en spécifiant que les boucles intérieures de chaque sous-fonction doivent être pipelinées (*st-sc-alg-opt1*).

Le Tableau 5.17 résume les résultats de synthèse obtenus en ciblant un FPGA de type Virtex-7 pour les différentes descriptions C produites. Au niveau de l'utilisation de LUTs et de registres, on remarque que l'architecture à flot de données en requiert légèrement plus. Au niveau de l'algorithme de référence, les directives d'optimisations permettent d'obtenir une accélération d'un facteur 24×, et se traduisent par une utilisation plus importante des blocs DSPs, ainsi que par l'utilisation de 4 blocs RAM pour la réutilisation des données. La fréquence d'horloge maximale rapportée est de 235 MHz. Pour le design avec des interfaces à flot de données, les optimisations spécifiées au moyen de directives ont permis d'obtenir une augmentation d'un facteur $6,7\times$ au niveau du débit supporté. Néanmoins, la version optimisée obtenue est limitée à 1 donnée acceptée à tous les 5 cycles. Cela contraste avec nos implémentations CASM+ (casm-design-I/casm-design-II) qui atteignent des bandes passantes à

près d'une donnée par cycle en entrée. D'autre part, bien que l'implémentation *ref-alg-opt2* atteigne des niveaux d'efficacité comparables à ceux obtenus avec l'approche au niveau des transferts synchronisés par les données, il n'a pas été possible de paralléliser davantage les calculs de l'algorithme pour les implémentations générées avec l'outil de synthèse C commercial, afin d'obtenir des taux d'opérations par seconde comparables à notre architecture multi-coeurs.

Tableau 5.17 Comparaison avec un outil de synthèse C

Implémentation de l'algorithme	Caractéristiques					Performances	
	LUTs	Registres	DSPs (blocs)	RAMs (blocs)	Fmax (MHz)	Temps d'exécution ($\times 10^9$ cycles)	Débit (données/cycles)
ref-alg	3238	2680	9	0	235	26.9	-
ref-alg-opt1	3306	2624	9	0	150	16.1	-
ref-alg-opt2	3473	2788	13	4	235	1.1	-
st-sc-alg	4047	2903	5	2	236	-	0.03
st-sc-alg-opt1	4912	3668	9	2	179	-	0.2
casm-type-I	3.1k	3.4k	4	1	263	-	1
casm-type-II	5.4k	5.4k	5	3	225	-	1

5.6 Inversion de matrices

Cette section présente l'application de la méthode de synthèse à niveau intermédiaire à la conception d'un circuit dédié pour l'inversion de matrices au moyen de l'algorithme d'élimination Gauss-Jordan. L'implémentation produite est comparée à une implémentation RTL à l'état de l'art [82].

5.6.1 Algorithme

L'algorithme d'élimination Gauss-Jordan considéré dans ce travail est illustré à la Figure 5.20. Cette algorithme réalise de façon entrelacée l'exécution des étapes d'élimination avant et arrière (élimination sur les lignes en dessous et au dessus du pivot, respectivement). Pour chaque colonne k, l'indice du pivot maximal est trouvé, et une permutation avec la ligne d'indice k a lieu au besoin. La rangée k est ensuite normalisée, puis utilisée pour réaliser les éliminations des rangées au-dessus, et en-dessous de la rangée k.

Optimisation mémoire

L'implémentation de l'algorithme d'élimination Gauss-Jordan utilisé fait appel à l'optimisation mémoire proposée dans [85], qui requiert seulement la mémorisation de la moitié de la

```
1:  for column k= 1 to m do
2:      i_pivot = selectPivotRowIndex(k)
3:      swap(i_pivot,k)
4:      normalize(k)
5:      for i = 1 to m do
6:          for j = k to n do
7:              A[i,j] = A[i,j]-A[i,k]*A[k,j]
8:          end for
9:      end for
10: end for
```

Figure 5.20 Algorithme d'inversion de matrice Gauss-Jordan.

matrice augmentée. De plus, cette optimisation permet également de réduire d'un facteur deux le nombre de lectures et d'écritures à chaque itération de la boucle extérieure (k) de l'algorithme. La Figure 5.21 donne une illustration de cette optimisation. Considérant une matrice augmentée 3×6, la région encadrée par un trait pointillé représente le contenu de la mémoire 3×3 à chaque itération de la boucle extérieure de l'algorithme d'élimination. On observe que les colonnes qui ne sont pas mémorisées sont implicites en fonction de la valeur de la variable d'itération k.

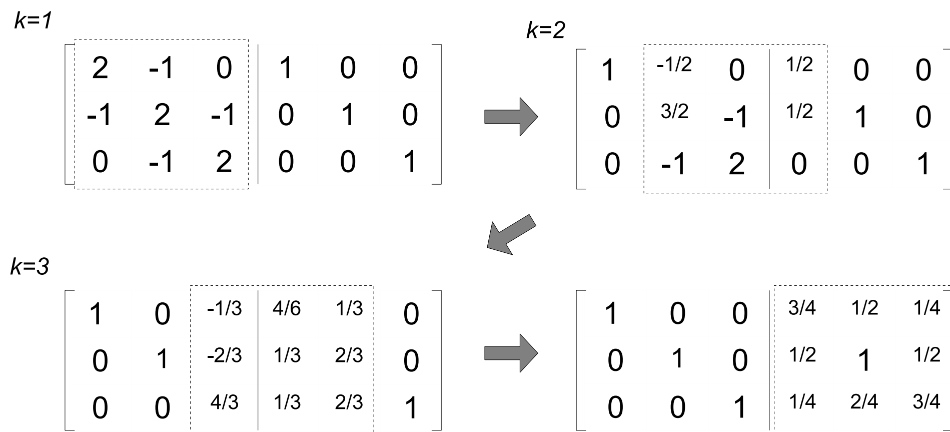


Figure 5.21 Optimisation des accès mémoire.

5.6.2 Conception

L'architecture de l'implémentation proposée pour l'inversion de matrices avec l'algorithme d'élimination Gauss-Jordan est basée sur celle qui est proposée dans [82]. Cette architecture intègre un unité de recherche du pivot maximal, ainsi qu'un unité de traitement des rangées permettant de réaliser l'élimination et la normalisation des rangées. L'unité de traitement peut être parallélisée selon une approche SIMD, pour laquelle n éléments consécutifs d'une

rangée sont traités en parallèle. La Figure 5.22 illustre cette architecture au niveau des ASMs. Les tâches de recherche du pivot maximal et d'élimination des rangées sont pipelinées, de sorte que la recherche du pivot pour l'itération $k+1$ se fait de manière concurrente avec les opérations d'élimination de l'itération k. La FIFO *ithrowcache* permet de mémoriser entièrement la ligne de pivot.

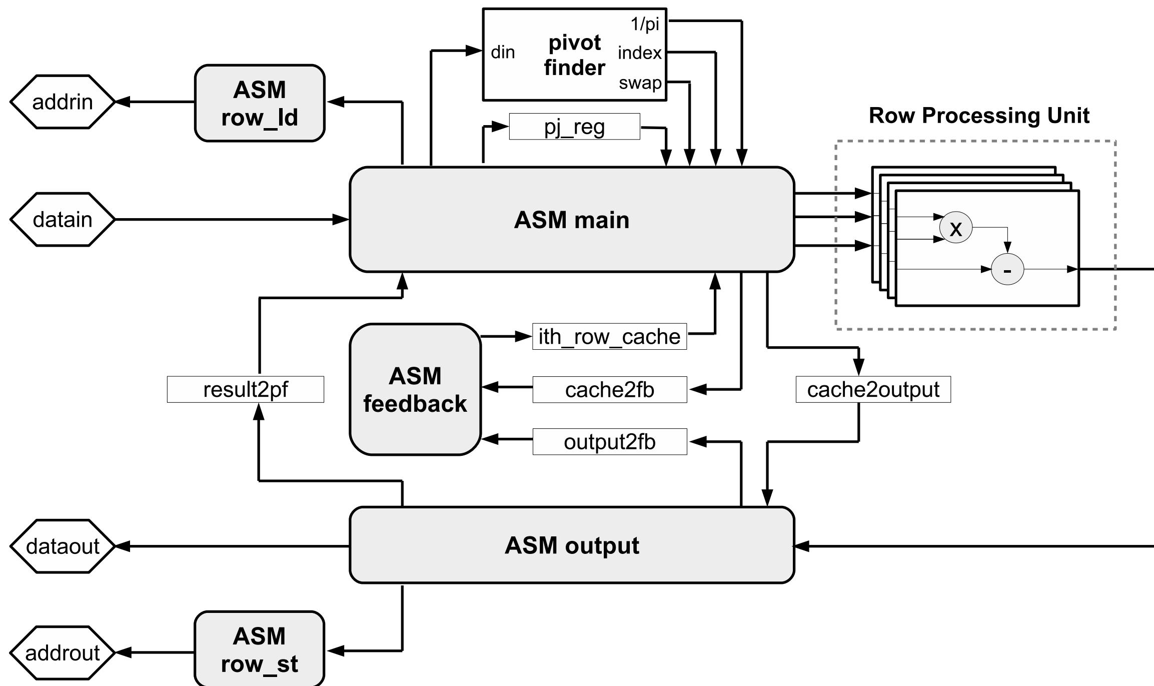


Figure 5.22 Architecture dédiée pour l'inversion de matrices Gauss-Jordan.

5.6.3 Résultats de synthèse et d'implémentation

La synthèse de l'architecture dédiée pour l'inversion de matrices a été réalisée avec l'outil ISE de Xilinx, ciblant un FPGA de type Virtex-5. En considérant une implémentation simple précision avec 4 unités de traitement, Le Tableau 5.18 résume les résultats obtenus en termes de ressources utilisées et de fréquence d'horloge maximale pour l'unité de recherche du pivot, l'unité de traitement des rangées, ainsi que le design dans son ensemble. Au total, le design requiert 5.9k LUTs, 6.6k registres, 20 DSP, et 2 blocs de mémoire RAM. La fréquence d'horloge maximale atteint 251 MHz. L'unité de recherche du pivot représente 1.5k LUTs et 1.8k registres, principalement dédiées au diviseur responsable de produire la réciproque du pivot. L'unité de traitement des rangées pour sa part représente 3.4k LUTs, 3.5k registres, et 20 DSPs dédiés aux multiplieurs et soustracteurs.

Tableau 5.18 Résultats d'implémentation en simple précision (Virtex-5)

Composant	Caractéristiques				
	LUTs	Registres	DSPs (DSP48E)	RAMs (36k BRAM)	Fmax (MHz)
Pivot finder	1.5k	1.8k	0	0	255
RPU	3.4k	3.5k	20	0	268
Core (x4)	5.9k	6.6k	20	2	251

5.6.4 Temps d'exécutions

Le Tableau 5.19 présente les temps d'exécutions obtenus pour des matrices de dimensions 16×16 à 256×256 considérant 4 unités de traitement parallèles et une fréquence d'horloge de 250 MHz. L'implémentation produite supporte une performance de pointe atteignant $2 \times N$ opérations en virgule flottante par cycle d'horloge, ou N représente de nombre d'unités parallèles dans l'unité de traitement des rangées. Pour une matrice de dimension 256×256, l'efficacité d'utilisation des opérateurs (multiplication et soustraction) observée atteint 94%.

Tableau 5.19 Temps d'exécution (4 coeurs de traitement)

Dimensions de la matrice	Cycles d'exécution	Temps d'exécution (ms)
16x16	3826	0.015
32x32	15808	0.063
64x64	89602	0.36
128x128	599963	2.40
256x256	4495168	17.9

5.6.5 Discussion

On retrouve dans la littérature différentes implémentations FPGA de circuits dédiés pour l'inversion de matrices denses au moyen de l'algorithme d'élimination Gauss-Jordan [82, 84]. Ces implémentations ont été obtenues au moyen d'une méthode de conception et de description au niveau RTL. Dans cette section, on considère la comparaison de notre design avec celui proposé dans [82]. Les deux circuits sont basés sur la même architecture et réalisent le même algorithme d'élimination optimisé pour l'utilisation mémoire.

Le Tableau 5.20 présente une comparaison sommaire de différentes implémentations de l'architecture, en termes de ressources utilisées et de performances obtenues. Bien que les architectures réalisées soient presqu'identiques, les opérateurs arithmétiques en virgule flottante utilisés dans [82] ne sont pas des opérateurs provenant de librairies standards. Plus particulièrement, ces opérateurs sont conçus spécialement pour faire une utilisation plus importante des blocs DSPs. Cela contraste avec nos implémentations qui utilisent des opérateurs stan-

dards générés avec l'outil *CoreGen* de Xilinx. Étant donné que ces opérateurs consomment significativement plus de ressources en termes de LUTs et de registres, des implémentations en simple et double précisions sont considérées. En termes de LUTs et de registres utilisés, nos implémentations en simple précision requièrent une quantité comparable aux implémentations proposées dans [82]. En double précision nos implémentations en requièrent de 2 à 3 fois plus. Bien que l'opérateur de division (double précision) utilisé dans nos implémentations soit limité à une fréquence d'opération de 258 MHz, on observe en double précision une baisse de fréquence appréciable à 219 MHz et 206 MHz pour des implémentations à 1 et 4 éléments de traitements parallèles. Par rapport aux implémentations proposées dans [82], ces fréquences d'horloges maximales plus basses peuvent s'expliquer par la plus grande quantité de ressources logiques et de registres utilisés. En contrepartie, pour les implémentations obtenues en simple précision, des fréquences d'horloges de 269 MHz et 251 MHz sont obtenues pour des implémentations à 1 et 4 coeurs. Des fréquences comparables à celle de 270 MHz obtenue pour l'implémentation à 1 coeur rapportée dans [82].

Les temps d'exécutions pour des matrices de tailles 128×128 et 256×256 rapportés dans [82] ont été obtenus en considérant une fréquence d'horloge de 250 MHz. Pour nos implémentations en simple précision, la même fréquence d'horloge est ainsi considérée. Toutefois, nos implémentations en double précision ne supportant pas une telle fréquence d'horloge, leurs fréquences d'horloge maximales respectives (219 MHz et 206 Mhz) sont considérées. On observe des temps d'exécutions comparables, bien que nos implémentations requièrent des temps légèrement supérieurs.

Tableau 5.20 Comparaison des designs dédiés pour l'inversion de matrices

	Duarte et al. [82]		Ce travail			
Configuration						
Précision	double	double	simple	simple	simple	simple
Type d'opérateurs	spécialisés	spécialisés	IP Xilinx	IP Xilinx	IP Xilinx	IP Xilinx
N_{PE}	1	4	1	4	1	4
Ressources utilisées						
LUTs	4k	10.6k	3.1k	5.9k	8.5k	18.6k
Registres	3.9k	10.3k	3.4k	6.6k	10.7k	20.8k
DSPs	41	77	5	20	14	56
RAMs	variable	variable	variable	variable	variable	variable
F_{max}	270	-	269	251	219	206
Temps d'exécution (s)						
128x128	0.008	0.002	0.009	0.002	0.010	0.003
256x256	0.067	0.017	0.070	0.018	0.080	0.022

Le Tableau 5.21 s'intéresse aux ressources utilisées au niveau du chemin de données et de son circuit de contrôle pour l'implémentation simple coeur proposée dans [82], et notre implémentation 4 coeurs en simple précision. Au niveau du chemin de données, on observe pour le

module $pivotfinder$ que l'utilisation d'un opérateur de division standard en simple précision entraîne une utilisation plus importante de ressources logiques et de registres. L'utilisation de l'opérateur spécialisé de division requiert toutefois significativement plus de blocs DSPs. Au niveau du contrôleur pour le chemin de données de l'architecture considérée, l'implémentation proposée dans [82] requiert environ 0.9k LUTs et 0.8k registres. Dans notre implémentation, en considérant les différentes ASMs illustrées à la Figure 5.22 et la logique d'autorisation des transferts de données, environ 0.8k LUTs et 0.1k registres sont requis. La centaine de registres utilisés est associée aux registres servant à mémoriser l'état de chaque ASM ainsi que le status de complétion (attribut *done*) des connexions bloquantes. Les autres registres spécifiées explicitement par l'usager dans la description sont considérés comme des éléments du chemin de données, ce qui explique le nombre significativement plus faible de registres utilisés par notre contrôleur. Au niveau des fonctions logiques combinatoires, les contrôleurs utilisent des quantités comparables de LUTs.

Tableau 5.21 Comparaison contrôle vs. chemin de données

	Duarte et al. [82]				Ce travail			
Composants	LUTs	Registres	DSPs	Fmax	LUTs	Registres	DSPs	Fmax
Pivot finder	0.6k	0.7k	29	270	1.5k	1.8k	0	255
Row Processing Unit	0.9k	1.1k	12	281	3.4k	3.5k	20	268
FP pre-/post-processor	1.6k	1.3k	0	310	-	-	-	-
Chemin de données	3.1k	3.1k	41	-	5.1k	6.5k	20	-
Logique de contrôle	0.9k	0.8k	0	-	0.8k	0.1k	0	-
Total	4.0k	3.9k	41	270	5.9k	6.6k	20	251

En regard des résultats obtenus par l'application de la méthode de synthèse de niveau intermédiaire à la conception d'un circuit spécialisé pour l'inversion de matrices, on observe que l'approche permet d'obtenir des résultats d'implémentations comparables à ceux obtenus avec une approche au niveau RTL, tout en permettant de bénéficier des avantages d'un niveau d'abstraction supérieur. Au niveau des transferts synchronisés par les données, le langage CASM+ permet de simplifier la descriptions de circuits au moyen d'ASMs concurrentes gérant l'interconnexion de sources et de puits avec des interfaces synchronisés par les données. Le langage CASM+ permet d'exprimer des opérations concurrentes et un contrôle précis au cycle d'horloge, mais une portion considérable de la logique de contrôle supportant la synchronisation adéquate des transferts de données est implicite et est générée automatiquement par le compilateur. Néanmoins, les implémentations de circuits spécialisés pour l'inversion de matrices produites montrent comment l'approche de niveau intermédiaire permet d'obtenir un contrôle efficace du chemin de données associé à l'architecture présentée à la Figure 5.22.

5.7 Conclusion

Dans ce chapitre, la méthode de description et de synthèse à niveau intermédiaire basée sur le langage CASM+ a été appliquée à la conception de différents circuits spécialisés intégrant des opérateurs et composants pipelinés. Le circuit spécialisé pour le tri rapide Quicksort présenté permettait d'évaluer le compilateur sur une première application requérant l'interconnexion de différents types de composants pipelinés, notamment des mémoires de type RAM. Les autres circuits spécialisés présentés ciblent des algorithmes de calculs avec des nombres en représentation virgule flottante. Dans un premier temps, un accumulateur pipeliné est présenté, reproduisant un algorithme de réduction présenté à l'état de l'art. Dans un second temps, des circuits spécialisés de plus grandes envergures sont considérés, associés à des algorithmes de produit matriciel, d'élimination Gaussienne, et d'inversion de matrices. Les comparaisons entre nos implémentations FPGA et celles retrouvées à l'état de l'art montrent comment l'approche de niveau intermédiaire permet d'obtenir des performances et qualités de designs comparables, tout en offrant les avantages inhérents à une approche de plus haut niveau par rapport à l'utilisation d'une approche au niveau RTL. Dans le cas de l'architecture spécialisée pour l'élimination Gaussienne dans un contexte de bande-passante mémoire limitée, nos expérimentations avec un outil de synthèse haut-niveau C/C++ commercial n'ont pas permis de produire (dans des temps équivalents) une implémentation offrant des performances comparables.

L'application de la méthode de description et de synthèse à niveau intermédiaire proposée à la conception de différents circuits spécialisés, de différents niveaux de complexités en terme de contrôle, permet également de valider sur des applications réelles le fonctionnement du compilateur, ainsi que sa méthode automatisée de résolution des relations combinatoires cycliques. Dans tous les cas observés, les descriptions RTL générées automatiquement par le compilateur manifestent le comportement désiré par rapport à la description CASM+ spécifiée. Une grande partie de la logique de contrôle supportant les transferts de données entre des sources et des puits aux interfaces à type flot de données prédéfinies est implicite et générée automatiquement par le compilateur.

CHAPITRE 6 CONCLUSION

6.1 Synthèse des travaux

Les procédés de micro-fabrication modernes permettent des niveaux d'intégration supportant plus d'un milliard de transistors par puce. Néanmoins, les coûts non-récurrents associés à la fabrication de circuits spécialisés au moyen de ces procédés sont tels que leur développement n'est justifiable que pour de très grands volumes de production. En revanche, les FPGA modernes permettent également de bénéficier des avancées au rythme de la loi de Moore en matière de densité d'intégration, mais avec des coûts non-récurrents nettement moins importants. De par leur capacité à être reprogrammés au moyen d'un fichier binaire, les FPGA s'apparentent d'une certaine façon à un processeur d'instruction pouvant supporter une vaste panoplie d'applications (de programmes) différentes. Étant donné leur potentiel intéressant en termes de temps d'exécution (puissance de calcul) et de consommation énergétique dans différentes applications, Intel a déjà annoncé leur intégration dans les puces de processeurs Xeon. Néanmoins, l'adoption des FPGA est confronté à un obstacle majeur, à savoir que leur programmation requiert la spécification d'un circuit numérique au moyen d'un langage de description de matériel (HDL).

Dans ce contexte, cette thèse s'est intéressée au sujet de la synthèse de niveau intermédiaire partant du langage de description de circuits CASM. Le langage CASM, introduit dans les années 2000, permet la description de machines séquentielles algorithmiques (ASM) gérant des connexions bloquantes et non-bloquantes entre des sources et des puits avec des interfaces prédéfinies de type flot de données. Sur la base du langage CASM, nous avons proposé une syntaxe supplémentaire permettant d'associer des règles à des énoncés de connexion. Les règles viennent contraindre l'autorisation des transferts de données sur les connexions actives, et peuvent sous-tendre des relations combinatoires cycliques en termes des signaux de synchronisation implicites des interfaces à flot de données. Deux opérateurs de connexion avancés ont également été proposés. L'opérateur de connexion différée a été proposé dans l'objectif d'abstraire le niveau de complexité associé à l'interconnexion d'opérateurs et de composants pipelinés. Avec cet opérateur, il est possible de décrire/spécifier au même moment un ensemble de connexion (associés à une opération complexe) issus de différents domaines temporels (étages) d'un pipeline. Les connexions différées sont traduites en requêtes de connexion qui seront exécutées (allouant le transfert d'une donnée) dès que possible. La logique de contrôle responsable de gérer et de traiter les requêtes de connexion différée est implicite et générée automatiquement par le compilateur. L'opérateur de connexion sur ré-

seau d'interconnexion pipeliné a été proposé afin de simplifier la description de connexions sur un réseau d'interconnexion pipeliné, tout en permettant la génération automatique de ce dernier selon différents objectifs (débit vs. surface). Avec cet opérateur, la spécification d'une connexion entre une source et un puit via un réseau d'interconnexion pipeliné se fait aussi simplement que dans le cas d'une connexion point-à-point standard. Les réseaux de types complètement connectés et bus partagés sont supportés.

Bien que le niveau d'abstraction proposé par le langage CASM soit bien adapté à la description de circuits qui interconnectent des sources et des puits avec des interfaces à type flot de données, l'interconnexion de sources et de puits dans différentes topologies peut induire des boucles combinatoires en termes des signaux de synchronisation des interfaces. Il en va de même pour les règles d'autorisation des transferts de données qui peuvent sous-tendre des relations combinatoires cycliques. La présence de relations cycliques est problématique d'une part parce qu'elle peut être associée à un comportement indéterminé, et d'autre part parce qu'elle mène à des circuits qui ne sont pas synthétisables ou pour lesquels l'analyse des temps de propagation est plus délicate. Notamment, ces relations combinatoires cycliques au niveau de la logique de synchronisation ne sont typiquement pas équivalentes à un réseau acyclique de fonctions logiques combinatoires. Afin de répondre à cette problématique, la thèse présenté propose une approche automatisée, travaillant à un niveau fonctionnel (logique), capable de transformer ces boucles combinatoires en circuits acycliques synthétisables. La résolution des comportements multi-stables en comportements monostables est faite de sorte à maximiser le nombre de transferts de données réalisés à chaque cycle d'horloge. La méthode de résolution proposée a été intégré à notre compilateur CASM+ développé en langage Java et permet d'abstraire tout le détail associé à la résolution des relations cycliques combinatoires en circuits combinatoires acycliques au niveau des signaux de synchronisation.

Dans l'optique de valider l'approche de synthèse automatisée proposée, le compilateur CASM+ développé dans le cadre de cette thèse a été appliqué à la conception de circuits spécialisés de différentes envergures, et de différents niveaux de complexité. Les circuits spécialisés présentés réalisent des algorithmes tels le tri rapide, le produit de matrices denses, l'élimination Gaussienne, ainsi que l'inversion de matrices denses. Le design d'un accumulateur basé sur un additionneur pipeliné est également présenté. Les comparaisons de ces circuits numériques avec d'autres implémentations similaires décrites au niveau RTL et/ou au niveau C montrent que l'approche proposée permet un compromis intéressant en termes de qualité d'implémentation, et de temps de développement. D'une part, la description de circuits interconnectant des sources et des puits avec des interfaces de type flot de données avec le langage CASM+ permet une description plus concise des comportements désirés. D'autre part, l'automatisation de nombreuses tâches liées à l'implémentation de la logique de contrôle vient également

réduire les sources d'erreurs possibles.

6.2 Comparaison avec l'état de l'art

À l'état de l'art, différentes méthodologies de conception de circuits numériques offrant un niveau d'abstraction supérieur au niveau RTL ont été proposées. Les outils de synthèse haut-niveau permettent notamment de supporter la synthèse automatisée de circuits numériques partant d'une spécification exécutable, généralement décrite au moyen des langages de programmation C/C++, SystemC, et plus récemment OpenCL. Sur la base d'une description algorithmique composée d'un ou de plusieurs processus séquentiels, les outils de synthèse haut-niveau ont la capacité de réordonnancer les opérations, en vue de paralléliser leur exécution de manière à satisfaire un budget de ressources disponibles et différentes contraintes de performance spécifiées par le concepteur (temps d'exécution, fréquence d'horloge, et consommation de puissance). Alors que la conception de circuits numériques au niveau RTL est généralement réservée aux spécialistes dans ce domaine, la synthèse haut-niveau permet au plus néophyte en électronique de synthétiser un circuit numérique partant d'une description logicielle. Toutefois, malgré des décennies de recherches intensives dans ce domaine, les résultats obtenus au moyen des outils de synthèse haut-niveau ne sont généralement pas suffisamment intéressant pour permettre à cette dernière de remplacer la méthodologie de conception au niveau RTL. De plus, bien que cette approche soit accessible au néophyte en électronique, en pratique l'obtention des meilleurs résultats requiert une bonne compréhension des implications associées à la manière de décrire un algorithme sur l'architecture matérielle qui sera synthétisée par l'outil. Bien qu'il soit possible de guider l'outil de synthèse haut-niveau au moyen de directives qui s'ajoutent à l'algorithme de départ, même pour un spécialiste en conception de circuits numériques il n'est pas toujours évident de comprendre pourquoi le compilateur n'est pas capable de produire l'architecture désirée (avec les performances escomptées). Le concepteur se trouve alors dans une impasse, puisqu'il n'est généralement pas envisageable d'aller modifier manuellement l'implémentation RTL produite par le compilateur. D'une part le spécialiste a souvent l'impression qu'il ne dispose pas d'un niveau de contrôle suffisant sur l'architecture qu'il cherche à produire, et d'autre part le néophyte peut difficilement produire des implémentations de qualité à la hauteur de ce dont ces outils sont capables.

En regard de la méthode de synthèse haut-niveau, qui propose un grand pas en avant en matière d'élévation du niveau d'abstraction au niveau de la conception de circuits numériques par rapport à la méthode de conception RTL, le niveau de conception avancé dans cette thèse propose certes un pas moins grand, mais d'autant plus sur. La description de

circuits numériques avec les langages CASM/CASM+ offre au concepteur un contrôle précis au niveau de l'architecture à produire et de la concurrence des opérations à exécuter (précision au cycle d'horloge). Néanmoins un niveau de complexité important, associé à la synchronisation des transferts sur des canaux (connexions) synchronisés par les données, est abstrait de la description. Au moyen de sémantiques clairement définies, le compilateur est en mesure de synthétiser automatiquement la logique de contrôle responsable d'assurer la synchronisation adéquate des transferts de donnés sur les connexions spécifiées par le concepteur. Bien que cette approche ne permette pas des temps de conceptions aussi rapides qu'une approche de synthèse haut-niveau C/C++, elle contribue tout de même de réduire de manière non-négligeable les temps de conceptions et de vérification associés à l'utilisation d'une méthodologie RTL. D'autre part, pour l'ensemble des applications considérées dans cette thèse, des résultats comparables à ceux obtenus avec une approche RTL ont été observés au niveau des performances obtenues (débit/latence), ainsi qu'en termes d'utilisation de ressources.

La description d'architectures dédiés au moyen d'interconnexions entre des sources et des puits dotés d'interfaces de type flot de donnés (*streaming interfaces*) n'est pas une approche unique aux langages CASM et CASM+ considérés dans cette thèse. Les interfaces de type flot de donnés sont couramment utilisées pour interconnecter des blocs de propriétés intellectuelles disponibles dans les librairies standard. Dans le monde des FPGAs, les interfaces AXI4-Stream et Avalon Streaming sont d'ailleurs largement supportées. Ces interfaces sont caractérisées par la présence de signaux de synchronisation de type *prêt-à-envoyer* et *prêt-à-recevoir*. Toutefois, les flots de conception supportés par les outils de synthèse modernes (*Vivado* de Xilinx, *Quartus II* d'Altera) se limitent typiquement à la spécification de canaux statiques. C'est à dire que pour chaque paire source-puits est reliée de manière permanente au moyens de fils. Bien que cela soit adéquat pour la description d'architecture à flot de donnés, cela est également une limitation importante pour la descriptions d'architectures où le flot de contrôle représente une partie importante de l'algorithme (la tâche/fonction) à réaliser. Cela contraste avec l'approche présentée dans cette thèse, pour laquelle les canaux synchronisés par les données sont activés et désactivés dynamiquement (à l'exécution) par des machines à états concurrentes.

En présences de relations combinatoires liant les entrées et sorties de synchronisation des différents blocs de propriétés intellectuelles interconnectés, l'interconnexion des sources et puits dans certaines topologies peut induire des relations combinatoires cycliques. De telles relations cycliques sont problématiques d'une part parce qu'elles sont associées à des comportements indéterminés, et d'autre part parce qu'elles mènent à des circuits qui ne sont pas synthétisables ou qui n'ont pas le comportement désiré. La résolution manuelle de telles boucles combinatoires est un réel problème, même pour les spécialistes en conception de cir-

cuits numériques les plus chevronnés. Afin d'adresser le problème, nous avons proposé une approche automatisée au niveau fonctionnel (logique) capable de transformer ces boucles combinatoires en circuits acycliques synthétisables, produisant un circuit de contrôle correct par construction. Ceci est réalisé en ne permettant que les états stables de la boucle correspondants à un plus grand nombre de transferts de données complétés à chaque cycle. Cette approche permet de simplifier le processus de description de circuits numériques. De même, elle permet d'abstraire un problème important associé à l'implémentation bas-niveau de la logique de contrôle supportant la synchronisation des transferts de données. À l'état de l'art, il n'existe pas d'autres approches de synthèse permettant d'automatiser la résolution de ce problème fondamental. De plus, puisque l'utilisation exclusive de canaux statiques pour interconnecter des sources et des puits à interfaces de type flot de donnés n'est qu'un cas particulier du modèle considéré dans cette thèse (supportant l'instanciation dynamique de tels canaux), la base théorique et la méthode de résolution automatisée des relations combinatoires cycliques proposée sont directement applicables aux outils de synthèse commerciaux utilisés dans l'industrie.

Les travaux de recherche présentés dans cette thèse ont fait l'objet de différentes publications scientifiques. Dans [70], une première version du compilateur a été appliquée à la conception et à la synthèse automatisée de circuits dédiés à l'exécution de l'algorithme de tri de données rapide *Quicksort*. Cette première application a permis d'évaluer la capacité du compilateur à produire un circuit de contrôle efficace partant de descriptions de circuits intégrants différents opérateurs et mémoires pipelinés. Dans [71] et [72], la méthode de description et de synthèse au niveau des transferts synchronisés par les données a été appliquée à la conception d'accumulateurs pipelinés en format virgule flottante. Un accumulateur basé sur l'architecture DB (*delayed buffering*) proposée à l'état de l'art a été conçu, permettant de comparer les résultats obtenus avec l'approche de niveau intermédiaire proposée à ceux obtenus au moyen d'une approche de description et de synthèse au niveau RTL. Dans [74] et [75], la méthode de description et de synthèse au niveau des transferts synchronisés par les données a été appliquée à la conception d'un design de plus grande envergure, soit un coprocesseur pour réaliser le produit matriciel en format virgule flottante. Les résultats de synthèse et d'implémentation FPGA obtenus ont également pu être comparés à ceux obtenus dans la littérature au moyen d'une approche RTL traditonnelle. Il a été possible d'observer que l'approche proposée permet d'obtenir des performances comparables tout en supportant la description d'architectures parallèles à un niveau d'abstraction plus élevé. Dans [81], la description et la synthèse au niveau des transferts synchronisés par les données est appliquée à la conception d'une architecture dédiée à l'élimination Gaussienne. Contrairement à l'algorithme de produit matriciel, l'exécution de l'algorithme d'élimination Gaussienne est sujette à un flot

de contrôle qui dépend des données, ce qui entraîne un circuit de contrôle plus complexe. Dans un contexte de bande-passante mémoire limitée, nos expérimentations avec un outil de synthèse haut-niveau C/C++ commercial n'ont pas permis de produire (dans des temps équivalents) une implémentation offrant des performances comparables à celles obtenues avec notre approche au niveau des transferts synchronisés par les données. Cette thèse présente également l'application de la méthode de description et de synthèse proposée à la conception d'une architecture dédiée à l'inversion de matrices proposée à l'état de l'art. L'algorithme d'inversion de matrices Gauss-Jordan considéré est également sujet à un flot de contrôle qui dépend des données. Une comparaison entre nos implémentations FPGA et celles retrouvées à l'état de l'art montrent comment l'approche de niveau intermédiaire permet d'obtenir des performances et qualités de designs comparables, tout en offrant les avantages inhérents à une approche de plus haut niveau par rapport à l'utilisation d'une approche au niveau RTL.

6.3 Limitations de la solution proposée et améliorations futures

L'approche de synthèse de niveau intermédiaire proposée dans cette thèse élève le niveau d'abstraction RTL offert par les langages de description de circuits VHDL et Verilog. Néanmoins, bien que le langage CASM+ permette de simplifier la description de circuits interconnectant des sources et des puits avec des interfaces à flot de données, pour le moment l'approche de synthèse de niveau intermédiaire proposée ne supporte pas l'exploration architecturale automatisée (allocation et association des ressources) et le réordonnancement des opérations que l'on retrouve dans les outils de synthèse haut-niveau C/C++. Tout en conservant la possibilité de spécifier complètement l'architecture désirée, il serait intéressant dans des travaux futurs de considérer d'abstraire les opérateurs pipelinés et les mémoires afin de permettre au compilateur d'automatiser la recherche d'une architecture intéressante en respect des objectifs ciblés (latence, débit, ressources, consommation de puissance). De même il serait possible de spécifier des variables qui serait allouées sur différents types d'éléments mémoires (registres, dossier de registres, mémoires RAM) également de manière automatisée en fonction de leurs temps de vie.

D'autre part, bien que le modèle de programmation basé sur des processus séquentiels communicants permette une fine description du parallélisme au sein d'une architecture à flot de données, il serait également intéressant dans des travaux futurs d'intégrer au langage CASM+ une syntaxe et des sémantiques permettant d'exprimer des algorithmes (ASMs) selon des modèles de programmation parallèle de type mémoire partagée et/ou mémoire distribuée. Cela permettrait dans une certaine mesure d'inférer automatiquement le(les) fichier(s) de description responsables d'interconnecter les multiples mémoires et unités de traitement d'une

architecture (pouvant être mise-à-l'échelle) à multi-coeurs spécialisés. D'autre part, une telle approche permettrait également de simplifier davantage le problème associé à la description d'algorithmes et d'architectures parallèles dédiées.

PUBLICATIONS DE L'AUTEUR

Les travaux de recherche présentés dans cette thèse ont fait l'objet de différentes publications scientifiques :

Articles de revue

- DAIGNEAULT, M.-A., and DAVID, J.P. (2016). Automated Synthesis of Streaming Transfer Level Hardware Designs. *ACM Transactions on Reconfigurable Technology* (TRETS). (*en voie d'etre soumis*)

- DAIGNEAULT, M.-A., and DAVID, J.P. (2014). Fast description and synthesis of control-dominant circuits. *Computers & Electrical Engineering* (CEENG).

Articles de conférence

- DAIGNEAULT, M.-A., and DAVID, J.P. (2015). Intermediate-Level Synthesis of a Gauss-Jordan Elimination Linear Solver. *2015 IEEE International Parallel and Distributed Processing Symposium Workshop* (IPDPSW).

- DAIGNEAULT, M.-A., and DAVID, J.P. (2013). High-Level Description and Synthesis of Floating-Point Accumulators on FPGA. *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines* (FCCM).

- DAIGNEAULT, M.-A., and DAVID, J.P. (2013). Hardware Description and Synthesis of Control-intensive Reconfigurable Dataflow Architectures (Abstract Only). *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (FPGA).

- DAIGNEAULT, M.-A., and DAVID, J.P. (2012). Synchronized-transfer-level design methodology applied to hardware matrix multiplication. *2012 International Conference on Reconfigurable Computing and FPGAs* (ReConFig).

— DAIGNEAULT, M.-A., and DAVID, J.P. (2012). Raising the abstraction level of HDL for control-dominant applications. *2012 22nd International Conference on Field Programmable Logic and Applications* (FPL).

RÉFÉRENCES

[1] M. Bohr, “A 30 year retrospective on dennard’s mosfet scaling paper,” *Solid-State Circuits Society Newsletter, IEEE*, vol. 12, no. 1, pp. 11–13, Winter 2007.

[2] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, June 2011, pp. 365–376.

[3] I. Kuon and J. Rose, “Measuring the gap between FPGAs and ASICs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, Feb. 2007.

[4] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Xiao, and D. Burger, “A reconfigurable fabric for accelerating large-scale datacenter services,” in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, June 2014, pp. 13–24.

[5] J. Bodily, B. Nelson, Z. Wei, D.-J. Lee, and J. Chase, “A comparison study on implementing optical flow and digital communications on fpgas and gpus,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 2, pp. 1–22, May 2010.

[6] W. Meeus, K. Van Beeck, T. Goedem, J. Meel, and D. Stroobandt, “An overview of today’s high-level synthesis tools,” *DESIGN AUTOMATION FOR EMBEDDED SYSTEMS*, 2013.

[7] D. Chen and D. Singh, “Invited paper : Using opencl to evaluate the efficiency of cpus, gpus and fpgas for information filtering,” in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, Aug 2012, pp. 5–12.

[8] J.-P. David and E. Bergeron, “An Intermediate Level HDL for System Level Design.” in *FDL’04*, 2004, pp. 526–536.

[9] ITRS, “International Technology Roadmap for Semiconductors,” Tech. Rep.

[10] G. Martin and G. Smith, “High-Level Synthesis : Past, Present, and Future,” *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18–25, Jul. 2009.

[11] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, “An Introduction to High-Level Synthesis,” *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 8–17, Jul. 2009.

[12] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, 1st ed. McGraw-Hill Higher Education, 1994.

[13] S. Gupta, N. Savoiu, S. Kim, N. Dutt, R. Gupta, and A. Nicolau, "Speculation techniques for high level synthesis of control intensive designs," *Proceedings of the 38th conference on Design automation - DAC '01*, pp. 269–272, 2001.

[14] S. Gupta, N. Dutt, R. Gupta, and a. Nicolau, "Loop shifting and compaction for the high-level synthesis of designs with complex control flow," *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, pp. 114–119, 2004.

[15] S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau, "Coordinated parallelizing compiler optimizations and high-level synthesis," *ACM Transactions on Design Automation of Electronic Systems*, vol. 9, no. 4, pp. 441–470, Oct. 2004.

[16] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau, *SPARK : A parallelizing approach to the high-level synthesis of digital circuits*, 1st ed. Kluwer Academic Publishers, 2004.

[17] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Dynamic conditional branch balancing during the high-level synthesis of control-intensive designs," *2003 Design, Automation and Test in Europe Conference and Exhibition*, pp. 270–275, 2003.

[18] Z. Baidas, A. Brown, and A. Williams, "Floating-point behavioral synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 7, pp. 828–839, Jul. 2002.

[19] J. Tripp, K. Peterson, C. Ahrens, J. Poznanovic, and M. Gokhale, "Trident : an FPGA compiler framework for floating-point algorithms," in *Field Programmable Logic and Applications, 2005. International Conference on*. IEEE, 2005, pp. 317–322.

[20] L. Semeria, K. Sato, and G. De Micheli, "Resolution of dynamic memory allocation and pointers for the behavioral synthesis from C," *Proceedings Design, Automation and Test in Europe Conference and Exhibition 2000 (Cat. No. PR00537)*, pp. 312–319, 2000.

[21] L. Semeria, K. Sato, and G. De Micheli, "Synthesis of hardware models in C with pointers and complex data structures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 6, pp. 743–756, 2001.

[22] M. Meribout and M. Motomura, "A combined approach to high-level synthesis for dynamically reconfigurable systems," *IEEE Transactions on Computers*, vol. 53, no. 12, pp. 1508–1522, Dec. 2004.

[23] C. Andriamisaina, P. Coussy, E. Casseau, and C. Chavet, "High-level synthesis for designing multimode architectures," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 29, no. 11, pp. 1736–1749, 2010.

[24] S. Ravi, a. Raghunathan, and N. Jha, "Generation of distributed logic-memory architectures through high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 11, pp. 1694–1711, Nov. 2005.

[25] Q. Liu, G. Constantinides, K. Masselos, and P. Cheung, "Compiling C-like Languages to FPGA Hardware : Some Novel Approaches Targeting Data Memory Organization," *The Computer Journal*, vol. 00, no. 0, 2009.

[26] C. Huang, S. Ravi, a. Raghunathan, and N. Jha, "Use of Computation-Unit Integrated Memories in High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1969–1989, Oct. 2006.

[27] C. Huang, S. Ravi, A. Raghunathan, and N. K. Jha, "Generation of Heterogeneous Distributed Architectures for Memory-Intensive Applications Through High-Level Synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 11, pp. 1191–1204, Nov. 2007.

[28] W. Yang, I.-c. Park, and C.-m. Kyung, "Low-power high-level synthesis using latches," *Proceedings of the ASP-DAC 2001. Asia and South Pacific Design Automation Conference 2001 (Cat. No.01EX455)*, pp. 462–465, 2001.

[29] S. Paik, I. Shin, T. Kim, and Y. Shin, "HLS-l : A High-Level Synthesis Framework for Latch-Based Architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 5, pp. 657–670, May 2010.

[30] V. Krishnan and S. Katkoori, "TABS : Temperature-Aware Layout-Driven Behavioral Synthesis," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 18, no. 99, p. 1, 2010.

[31] R. Mukherjee and S. Memik, "An integrated approach to thermal management in high-level synthesis," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, no. 11, pp. 1165–1174, Nov. 2006.

[32] C. He and M. F. Jacome, "Defect-Aware High-Level Synthesis Targeted at Reconfigurable Nanofabrics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 5, pp. 817–833, May 2007.

[33] Y. Chen and Y. Xie, "Tolerating process variations in high-level synthesis using transparent latches," *2009 Asia and South Pacific Design Automation Conference*, pp. 73–78, Jan. 2009.

[34] Y. Xie and Y. Chen, "Statistical High-Level Synthesis under Process Variability," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 78–87, Jul. 2009.

[35] Mentor, "Catapult C synthesis," *Website : http ://www. mentor. com*, 2010.

[36] T. Bollaert, "Catapult synthesis : A practical introduction to interactive c synthesis," in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Springer Netherlands, 2008, pp. 29–52.

[37] M. Graphics. (2011) Algorithmic c datatypes. [Online]. Available : http://www.mentor.com/esl/catapult/algorithmic

[38] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "Autopilot : A platform-based esl synthesis system," in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Springer Netherlands, 2008, pp. 99–112.

[39] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas : From prototyping to deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, April 2011.

[40] F. Winterstein, S. Bayliss, and G. Constantinides, "High-level synthesis of dynamic data structures : A case study using vivado hls," in *Field-Programmable Technology (FPT), 2013 International Conference on*, Dec 2013, pp. 362–365.

[41] M. Meredith, "High-Level SystemC Synthesis with Forte's Cynthesizer High-Level Synthesis," in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Springer Netherlands, 2008, pp. 75–97.

[42] J. Keinert, M. Streubehr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith, "SystemCoDesigner - an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications," *ACM Transactions on Design Automation of Electronic Systems*, vol. 14, no. 1, pp. 1–23, Jan. 2009.

[43] K. Wakabayashi and B. C. Schafer, " " All-in-C" Behavioral Synthesis and Verification with CyberWorkBench High-Level Synthesis," in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Springer Netherlands, 2008, pp. 113–127.

[44] K. Wakabayashi, "C-based synthesis experiences with a behavior synthesizer,"cyber"," in *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings.* IEEE, 2002, pp. 390–393.

[45] S. Aditya and V. Kathail, "Algorithmic synthesis using pico," in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Springer Netherlands, 2008, pp. 53–74.

[46] C. Karfa and S. Jain, "On multi-cycle path support in model based high-level synthesis," in *Students' Technology Symposium (TechSym), 2014 IEEE*, Feb 2014, pp. 253–258.

[47] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup : An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, pp. 24 :1–24 :27, Sep. 2013.

[48] K. Shagrithaya, K. Kepa, and P. Athanas, "Enabling development of opencl applications on fpga platforms," in *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, June 2013, pp. 26–30.

[49] J. Hoe, "Synthesis of operation-centric hardware descriptions," in *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design.* IEEE Press, 2000, pp. 511–519.

[50] N. Dave, Arvind, and M. Pellauer, "Scheduling as Rule Composition," *2007 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007)*, pp. 51–60, May 2007.

[51] F. Baader and T. Nipkow, *Term rewriting and all that.* New York, NY, USA : Cambridge University Press, 1998.

[52] N. Marti-Oliet, "Rewriting logic : roadmap and bibliography," *Theoretical Computer Science*, vol. 285, no. 2, pp. 121–154, Aug. 2002.

[53] D. Rosenband, "Hardware synthesis from guarded atomic actions with performance specifications," *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, pp. 784–791, 2005.

[54] E. S. Chung and J. C. Hoe, "Implementing a high-performance multithreaded microprocessor : A case study in high-level design and validation," *2009 7th IEEE/ACM International Conference on Formal Methods and Models for Co-Design*, pp. 98–107, Jul. 2009.

[55] E. Chung and J. Hoe, "High-Level Design and Validation of the BlueSPARC Multithreaded Processor," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 29, no. 10, pp. 1459–1470, 2010.

[56] F. Gruian and M. Westmijze, "BluEJAMM : A Bluespec Embedded Java Architecture with Memory Management," *Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007)*, pp. 459–466, Sep. 2007.

[57] A. Agarwal, M. C. Ng, and Arvind, "A comparative evaluation of high-level hardware synthesis using reed solomon decoder," *Embedded Systems Letters, IEEE*, vol. 2, no. 3, pp. 72–76, Sept 2010.

[58] O. Lindtjorn, R. Clapp, O. Pell, H. Fu, M. Flynn, and H. Fu, "Beyond traditional microprocessors for geoscience high-performance computing applications," *Micro, IEEE*, vol. 31, no. 2, pp. 41–49, 2011.

[59] M. J. Flynn, O. Mencer, V. Milutinovic, G. Rakocevic, P. Stenstrom, R. Trobec, and M. Valero, "Moving from petaflops to petadata," *Commun. ACM*, vol. 56, no. 5, pp. 39–42, May 2013.

[60] H. Fu, L. Gan, R. Clapp, H. Ruan, O. Pell, O. Mencer, M. Flynn, X. Huang, and G. Yang, "Scaling reverse time migration performance through reconfigurable dataflow engines," *Micro, IEEE*, vol. 34, no. 1, pp. 30–40, Jan 2014.

[61] F. Pratas, D. Oriato, O. Pell, R. Mata, and L. Sousa, "Accelerating the computation of induced dipoles for molecular mechanics with dataflow engines," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, April 2013, pp. 177–180.

[62] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, Jan 2003.

[63] C. Clare, *Designing logic systems using state machines*, ser. McGraw-Hill series in electrical engineering. McGraw-Hill, 1973.

[64] S. de Pablo, S. Caceres, J. Cebrian, and M. Berrocal, "A proposal for asm++ diagrams," in *Design and Diagnostics of Electronic Circuits and Systems, 2007. DDECS '07. IEEE*, April 2007, pp. 1–4.

[65] R. Sinha and H. Patel, "Abstract state machines as an intermediate representation for high-level synthesis," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, March 2011, pp. 1–6.

[66] J.-P. David and E. Bergeron, "An Intermediate Level HDL for System Level Design." in *FDL'04*, 2004, pp. 526–536.

[67] E. Bergeron, X. Saint-Mleux, M. Feeley, and J. P. David, "High level synthesis for data-driven applications," in *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping*, ser. RSP '05, 2005, pp. 54–60.

[68] K. Kalach and J. David, "Hardware implementation of large number multiplication by fft with modular arithmetic," in *IEEE-NEWCAS Conference, 2005. The 3rd International*, june 2005, pp. 267 – 270.

[69] J.-H. Jiang, A. Mishchenko, and R. Brayton, "On breakable cyclic definitions," in *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, 2004, pp. 411–418.

[70] M.-A. Daigneault and J. David, "Raising the abstraction level of hdl for control-dominant applications," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, Aug 2012, pp. 515–518.

[71] M.-A. Daigneault and J. P. David, "Hardware description and synthesis of control-intensive reconfigurable dataflow architectures (abstract only)," in *Proceedings of the*

ACM/SIGDA International Symposium on Field Programmable Gate Arrays, ser. FPGA '13. New York, NY, USA : ACM, 2013, pp. 274–275.

[72] M.-A. Daigneault and J. David, "High-level description and synthesis of floating-point accumulators on fpga," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, April 2013, pp. 206–209.

[73] Y.-G. Tai, C.-T. D. Lo, and K. Psarris, "Accelerating matrix operations with improved deeply pipelined vector reduction," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 2, pp. 202 –210, feb. 2012.

[74] M.-A. Daigneault and J. David, "Synchronized-transfer-level design methodology applied to hardware matrix multiplication," in *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, Dec 2012, pp. 1–7.

[75] M.-A. Daigneault and J. P. David, "Fast description and synthesis of control-dominant circuits," *Computers & Electrical Engineering*, vol. 40, no. 4, pp. 1199 – 1214, 2014.

[76] L. Zhuo and V. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, no. 4, pp. 433 –448, april 2007.

[77] V. Kumar, S. Joshi, S. Patkar, and H. Narayanan, "Fpga based high performance double-precision matrix multiplication," in *VLSI Design, 2009 22nd International Conference on*, jan. 2009, pp. 341 –346.

[78] J. Jiang, V. Mirian, K. P. Tang, P. Chow, and Z. Xing, "Matrix multiplication based on scalable macro-pipelined fpga accelerator architecture," in *Reconfigurable Computing and FPGAs, 2009. ReConFig '09. International Conference on*, dec. 2009, pp. 48 –53.

[79] B. Holanda, R. Pimentel, J. Barbosa, R. Camarotti, A. Silva-Filho, L. Joao, V. Souza, J. Ferraz, and M. Lima, "An fpga-based accelerator to speed-up matrix multiplication of floating point operations," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, may 2011, pp. 306 –309.

[80] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna, "Analysis of high-performance floating-point arithmetic on fpgas," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, april 2004, p. 149.

[81] M.-A. Daigneault and J. David, "Intermediate-level synthesis of a gauss-jordan elimination linear solver," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, May 2015, pp. 176–181.

[82] R. Duarte, H. Neto, and M. Vestias, "Double-precision gauss-jordan algorithm with partial pivoting on fpgas," in *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, Aug 2009, pp. 273–280.

[83] J. Arias-Garcia, C. Llanos, M. Ayala-Rincon, and R. Jacobi, "Fpga implementation of large-scale matrix inversion using single, double and custom floating-point precision," in *Programmable Logic (SPL), 2012 VIII Southern Conference on*, March 2012, pp. 1–6.

[84] S. Moussa, A. Razik, A. Dahmane, and H. Hamam, "Fpga implementation of floating-point complex matrix inversion based on gauss-jordan elimination," in *Electrical and Computer Engineering (CCECE), 2013 26th Annual IEEE Canadian Conference on*, May 2013, pp. 1–4.

[85] G. de Matos and H. Neto, "Memory optimized architecture for efficient gauss-jordan matrix inversion," in *Programmable Logic, 2007. SPL '07. 2007 3rd Southern Conference on*, Feb 2007, pp. 33–38.