N° d'ordre : 4150 ANNÉE 2010





THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale MATISSE

présentée par

Kévin MARTIN

préparée à l'unité de recherche UMR 6074 – Irisa
Institut de Recherche en Informatique et Systèmes Aléatoires
IFSIC

Génération automatique d'extensions de jeux d'instructions de processeurs

Thèse soutenue à Rennes
le 7 septembre 2010

devant le jury composé de :

Olivier DÉFORGES
Professeur INSA Rennes / président

Daniel ETIEMBLE
Professeur Université Paris Sud 11 / rapporteur

Frédéric PÉTROT
Professeur ENSIMAG / rapporteur

Philippe COUSSY
Maître de conférences Université Bretagne Sud / examinateur

Christophe WOLINSKI
Professeur Université de Rennes 1 / directeur de thèse

François CHAROT
Chargé de recherche INRIA / co-directeur de thèse

Le temps est assassin et emporte avec lui
les rires des enfants

Mistral Gagnant
Renaud

Remerciements

Je remercie Olivier Déforges qui me fait l'honneur de présider ce jury.

Je remercie Daniel Etiemble et Frédéric Pétrot d'avoir accepté la charge de rapporteur.

Je remercie Philipppe Coussy qui a bien voulu juger ce travail.

Je remercie bien sûr mon directeur de thèse, Christope Wolinski, et mon co-directeur de thèse François Charot.

Je remercie également le professeur Kuchcinski pour sa collaboration qui a fortement contribué à la réussite de ces travaux.

Je remercie toute l'équipe projet INRIA Cairn, en particulier Antoine Floc'h, Erwan Raffin, Steven Derrien, Florent Berthelot, Georges Adouko (en espérant qu'il soutienne sa thèse un jour), Laurent Perraudeau, Charles Wagner, Patrice Quinton, Pasha et Naeem, sans oublier les lannionais Emmanuel Casseau, Daniel Ménard, Daniel Chillet, Sébastien Pillement, Arnaud Tisserand, et le chef de projet Olivier Sentieys.

Un grand merci également aux assistantes de projet qui m'ont aidé administrativement, Céline, Isabelle, Elise et Nadia.

Enfin, je remercie ma famille et mes amis qui m'ont toujours soutenu, même si ils ne comprenaient rien à mes travaux. Et surtout, un grand merci à ma femme Laouratou qui a mis au monde notre petit garçon N'maï Lamine et qui s'en est largement occupé le temps que je finisse cette thèse.

Merci à tous.

Table des matières

Introduction **1**

1 Conception d'ASIP : méthodologies et outils **13**
1.1 Conception complète d'ASIP 14
1.1.1 Exemple du langage LISA 14
1.1.2 Méthodologie 18
1.2 Conception partielle d'ASIP 21
1.2.1 Exemple 21
1.2.2 Architecture des processeurs extensibles 24
1.2.3 Méthodologie 30
1.2.4 Compilation 32
1.3 Synthèse 36

2 L'extension de jeu d'instructions **39**
2.1 Quels sont les problèmes à résoudre ? 40
2.1.1 Motifs générés vs motifs prédéfinis 40
2.1.2 Génération d'instructions 41
2.1.3 Limites architecturales et technologiques 44
2.1.4 Sélection d'instructions 46
2.1.5 Isomorphisme de graphes et de sous-graphes 47
2.1.6 Exploitation des nouvelles instructions 49
2.2 Quand les résoudre ? 50
2.2.1 À la conception 50
2.2.2 À la volée 51
2.3 Comment les résoudre ? 51
2.3.1 Recherche Tabou 52
2.3.2 Recuit simulé 52
2.3.3 Algorithme génétique 53
2.3.4 Algorithme de colonies de fourmis 54
2.3.5 Programmation dynamique 54
2.3.6 Algorithmes gloutons 55
2.3.7 Heuristiques 56

2.3.8 Séparation et évaluation . . . 57
2.3.9 Programmation linéaire par nombres entiers . . . 58
2.3.10 Programmation par contraintes . . . 59
2.3.11 Comparaison des approches . . . 60
2.4 Synthèse . . . 62

3 L'identification d'instructions 65
3.1 Définitions . . . 65
3.2 Algorithme de génération de motifs . . . 66
3.2.1 Expansion d'un motif à partir d'un nœud graine . . . 66
3.2.2 Limitation du nombre de motifs : *Smart filtering* . . . 67
3.2.3 La contrainte `GraphMatch` . . . 68
3.3 Contraintes technologiques et architecturales . . . 69
3.4 Formalisation pour la programmation par contraintes . . . 70
3.4.1 Contrainte de connexité . . . 70
3.4.2 Contrainte sur le nombre d'entrées et de sorties . . . 74
3.4.3 Contrainte sur le chemin critique . . . 76
3.4.4 Contrainte de ressources . . . 77
3.4.5 Contrainte d'énergie . . . 77
3.5 Résultats d'expérimentations . . . 78
3.5.1 Le processeur cible . . . 78
3.5.2 Environnement d'exécution . . . 78
3.5.3 Benchmarks . . . 78
3.5.4 Couverture maximale . . . 79
3.5.5 Discussion . . . 80
3.6 Conclusion . . . 82

4 Sélection d'instructions et ordonnancement 83
4.1 Sélection d'instructions . . . 83
4.1.1 Définition d'une occurrence . . . 83
4.1.2 Définition du problème de sélection d'instructions . . . 84
4.1.3 Modélisation du temps d'exécution d'une occurrence . . . 90
4.1.4 Résultats d'expérimentation . . . 93
4.2 Ordonnancement . . . 96
4.2.1 Placement . . . 97
4.2.2 Ordonnanceur . . . 97
4.2.3 Ordonnancement avec recalage d'instructions . . . 99
4.2.4 Optimisation sur les transferts de données . . . 101
4.2.5 Résultats d'expérimentation . . . 104
4.3 Conclusion . . . 106

5 Génération d'architecture et adaptation du code 107
5.1 Architecture de l'extension . . . 109
5.1.1 Différents types d'instructions spécialisées . . . 109
5.1.2 Modèles d'architecture . . . 110
5.2 Allocation de registres . . . 112
5.2.1 Contrainte `Diff2` . . . 113
5.2.2 Allocation de registres appliquée au modèle A . . . 114
5.2.3 Allocation de registres appliquée au modèle B . . . 118
5.3 Génération des codes d'opération et de la description de l'architecture . . . 133
5.3.1 Génération des codes d'opérations . . . 133
5.3.2 Génération du bloc logique spécialisé . . . 133
5.3.3 Génération du chemin de donnée des instructions spécialisées . . . 134
5.3.4 Fusion du chemin de données . . . 134
5.4 Génération du code source modifié . . . 135
5.5 Génération pour la simulation SystemC . . . 135
5.5.1 SystemC . . . 135
5.5.2 SoCLib . . . 136
5.5.3 Génération des instructions spécialisées pour le modèle du NiosII . 136
5.6 Conclusion . . . 136

6 Déroulement du flot appliqué à l'algorithme GOST 139
6.1 Le flot de conception *DURASE* . . . 139
6.1.1 Gecos . . . 141
6.2 L'algorithme de chiffrement GOST . . . 143
6.3 Déroulement du flot étape par étape appliqué à l'algorithme de chiffrement GOST . . . 143
6.3.1 Paramètres d'entrée . . . 144
6.3.2 Partie frontale . . . 144
6.3.3 Création d'une extension . . . 147
6.3.4 Génération d'architecture . . . 158
6.3.5 Adaptation du code . . . 161
6.3.6 Validation par simulation avec SoCLib . . . 162
6.3.7 Résumé . . . 164
6.4 Proposition d'architecture . . . 164
6.5 Conclusion . . . 165

Conclusion 167

Publications 171

Bibliographie 173

Glossaire

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction-set Processor
CDFG	Control Data Flow Graph
CFG	Control Flow Graph
CIS	Composant Instruction Spécialisée
CPLD	Complex Programmable Logic Device
DAG	Directed Acyclic Graph
DFT	Data Flow Tree
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
GHDC	Graphe Hiérarchisé aux Dépendances Conditionnées
GPP	General Purpose Processor
HCDG	Hierarchical Conditional Dependency Graph
ILP	Integer Linear Programming
IP	Intellectual Property
ISA	Instruction Set Architecture
LE	Logic Element
LUT	Look-Up Table
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MMU	Memory Management Unit
RISC	Reduced Instruction Set Processor
RPU	Reconfigurable Processing Unit
RTL	Register Transfer Level
SHA	Secure Hash Algorithm
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
UAL	Unité Arithmétique et Logique
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLIW	Very Long Instruction Word

Introduction

Les systèmes embarqués ont aujourd'hui envahi notre quotidien par le lot de services qu'ils apportent : téléphonie mobile, navigation automobile, multimédia (photo, vidéo, musique), assistants électroniques, entre autres. Un processeur embarqué, simple et peu cher à mettre en œuvre, est la solution idéale pour de tels systèmes. Mais pour faire face aux demandes exigeantes en termes de performance, de surface de silicium et de consommation d'énergie, ces systèmes s'appuient sur des solutions matérielles adaptées aux besoins des applications. Une stratégie très répandue lors de leur conception consiste alors à associer des circuits intégrés spécialisés à un processeur généraliste à bas coût. Par exemple, de nombreux produits commerciaux associent un processeur ARM [22] avec des circuits spécialisés : téléphone mobile (Nokia N97), lecteurs multimédia (Archos 604), GPS (Mio Moov 500).

Le nombre de transistors intégrés sur un même substrat ne cesse de croître et les capacités d'intégration suivent la célèbre loi de Moore [158]. La technologie permet aujourd'hui de réunir au sein d'une même puce des composants de nature hétérogène. Ces systèmes, appelés *systèmes sur puce* (*System on chip*, SoC) incluent couramment un ou plusieurs processeurs, de la mémoire, de la logique spécialisée, des périphériques, etc. Les différentes possibilités architecturales que proposent ces systèmes offrent un espace d'exploration énorme pour les concepteurs. De plus, les fonctionnalités des produits et la complexité sous-jacente à ces systèmes augmentent considérablement. Combinée à un temps de mise sur le marché très court, la conception des SoC traverse une *crise de la complexité* [102]. La productivité des ingénieurs n'arrive pas à suivre l'évolution des systèmes et le fossé se creuse de plus en plus entre les possibilités offertes par la technologie et son utilisation effective. Ce fossé appelé *design gap*, est illustré par la figure 1.

Problématique

Afin de réduire le temps de conception, de garantir les performances souhaitées et d'assurer la qualité des nouveaux produits, la conception des SoC repose sur la réutilisation et l'assemblage de briques vérifiées et fiables, appelées IP (*Intellectual Property*) [80, 111]. Parmi ces briques, on trouve couramment des circuits dédiés (ASIC, *Application Specific Integrated Circuit*) ou des processeurs de type ASIP (*Application Specific Instruction-set*

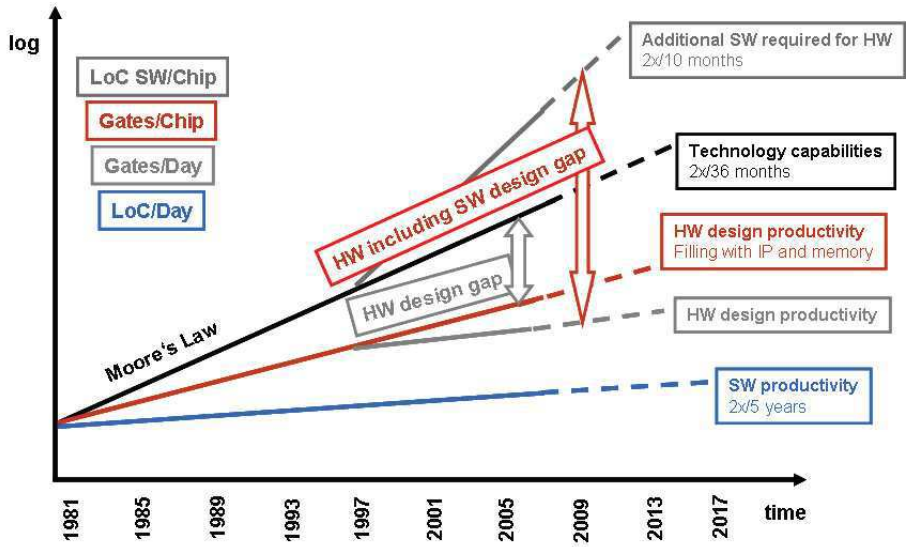


FIG. 1 – Difficultés de conception (tiré de ITRS 2007 [11])

Processor) dont le rôle est de réaliser une fonctionnalité précise. Un ASIP est un processeur spécialisé pour une application ou un domaine d'applications. Un exemple typique d'ASIP est le processeur DSP (*Digital Signal Processor*). L'utilisation des processeurs spécialisés de type DSP permet de répondre aux exigences du domaine du traitement du signal tout en restant dans un cadre de programmabilité pour bénéficier du principal avantage d'un processeur, sa flexibilité. La figure 2 montre les résultats d'évaluation des performances de différents processeurs réalisée par BDTI [10] (*Berkeley Design Technology, Inc*). Les résultats sont obtenus pour des applications classiques du traitement du signal (transformées de Fourier, filtres, etc.) selon une métrique qui prend en compte les caractéristiques de chaque processeur (temps d'exécution, utilisation de la mémoire, consommation d'énergie) [32]. Il apparaît clairement que pour le domaine d'applications du traitement du signal, les DSP (TI C64x ou Freescale MSC8144) sont plus performants que les processeurs à usage général (Intel Pentium III, MIPS, ARM).

Ces différences de performances s'expliquent par l'architecture des DSP qui est spécialement conçue pour le traitement du signal. Une des particularités architecturales du DSP est la présence d'une unité fonctionnelle, appelée *MAC* (*Multiply-ACcumulate*), qui permet de réaliser une multiplication suivie d'une addition en un seul cycle, alors que ces deux opérations successives nécessitent généralement plusieurs cycles pour être exécutées par un processeur généraliste. La présence d'un *MAC* dans un DSP est née de l'observation que la multiplication suivie d'une addition est une séquence d'instructions très courante dans les applications du traitement du signal.

Un ASIP est une brique de base intéressante pour la conception des SoC, mais aussi pour la conception des MPSoC (*Multi Processor System on Chip*), que ce soit dans un contexte de plate-forme multi-processeur hétérogène (figure 3), où chaque processeur est dédié à une fonctionnalité précise, ou bien dans un contexte de plate-forme multi-processeur

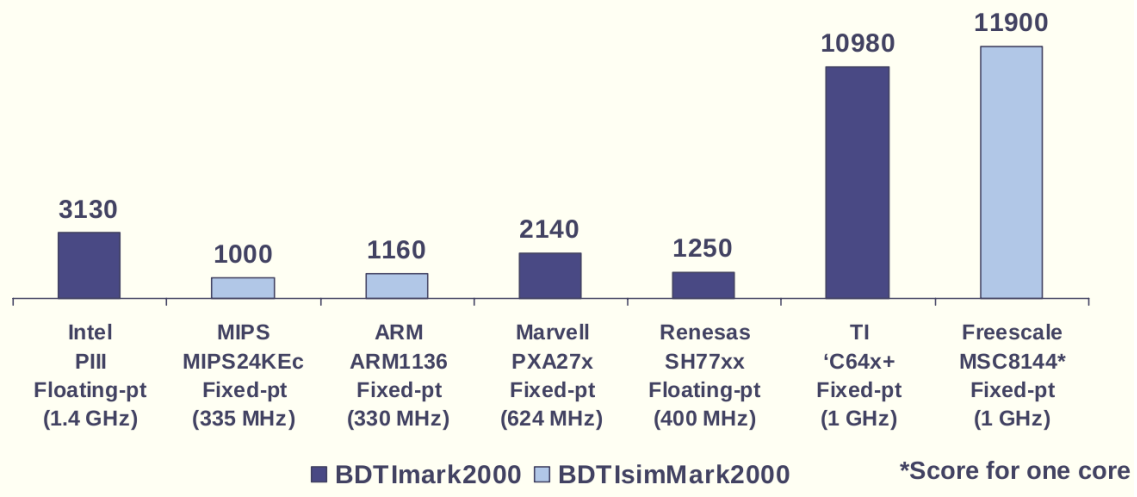


FIG. 2 – Évaluation des performances de différents processeur par BDTI [34], (*Higher is Faster*)

homogène, où le même processeur spécialisé est répliqué n fois. La conception d'un ASIP est un problème dont la première difficulté est de choisir parmi ses nombreuses déclinaisons architecturales.

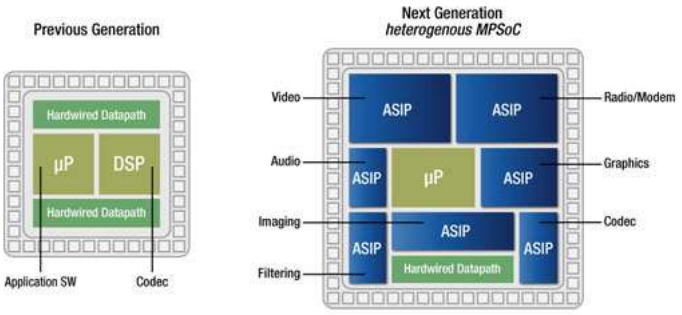


FIG. 3 – Évolution des SoC selon Retarget [176]

Les ASIP couvrent le large spectre des solutions architecturales disponibles qui vont du processeur à usage général au circuit dédié, comme illustré par la figure 4. Parmi ces solutions, l'ASIP a une place intéressante. L'ASIP peut être dédié à un domaine d'applications, comme l'est le DSP pour le traitement du signal. L'ASIP peut aussi être entièrement dédié à un jeu d'algorithmes donné. La conception d'un processeur entièrement spécialisé vise à définir un jeu d'instructions (ISA, *Instruction Set Architecture*) complètement dédié à une application ou à un domaine d'applications [107, 108, 140]. Dans le cas d'une spécialisation complète, le processeur n'embarque que le jeu d'instructions nécessaire à l'exécution d'une ou de plusieurs applications et permet de gagner en surface et en fréquence [130, 214]. Le processeur et son compilateur associé sont conçus grâce à des langages de description

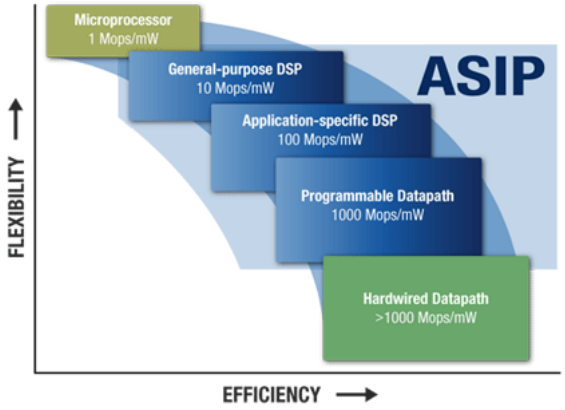


FIG. 4 – Place des ASIP dans l'espace des solutions architecturales

d'ISA de haut niveau. Le compilateur CHESS de Target [147] basé sur le langage de description d'architecture nML [71], et LISATek de Coware basé sur LISA [104] sont des exemples commerciaux dans ce domaine. L'inconvénient majeur de cette technique est que l'architecture du jeu d'instructions doit être repensée pour chaque nouveau processeur.

Pour réduire la complexité de ce travail de conception et le temps de mise sur le marché, l'approche défendue dans la dernière décennie est de spécialiser partiellement un processeur. Elle consiste à partir d'un processeur existant, vérifié et optimisé, classiquement un RISC (*Reduced Instruction Set Processor*), et à coupler un module matériel au chemin de données du processeur pour étendre son jeu d'instructions [25, 30, 55, 121]. C'est l'extension de jeu d'instructions. Le processeur conserve sa généricité initiale tout en proposant des instructions spécialisées, mises en œuvre dans le module matériel par de la logique spécialisée. Dans le cas d'une spécialisation partielle, le processeur possède un jeu d'instructions de base (sous-entendu, avec son architecture), capable d'exécuter tout type d'applications. Utiliser un processeur existant permet de gagner sur les coûts d'ingénierie non récurrents et de concevoir rapidement un processeur spécialisé. De nombreux succès commerciaux témoignent de l'efficacité de cette technique : Xtensa de Tensilica [93, 146], ARC 700 [199], NIOSII d'Altera [17], MIPS Pro Series [155]. L'inconvénient de cette approche est que les instructions spécialisées doivent respecter les contraintes qu'impose l'architecture. Celles-ci diffèrent selon la nature du couplage de l'extension.

La figure 5 illustre l'exemple du processeur NIOSII d'Altera où l'extension est très fortement couplée au chemin de données du processeur. L'extension, en parallèle avec l'UAL (Unité Arithmétique et Logique) du processeur, agit comme une unité fonctionnelle spécialisée. Dans ce cas, le nombre d'opérandes en entrée et en sortie de l'extension doit être le même que pour une instruction du processeur. Pour le NIOSII, les instructions possèdent 2 entrées et 1 sortie. De plus, le chemin critique de l'instruction doit être en adéquation avec la fréquence du processeur. Enfin, le surcoût matériel et la consommation d'énergie doivent être pris en considération.

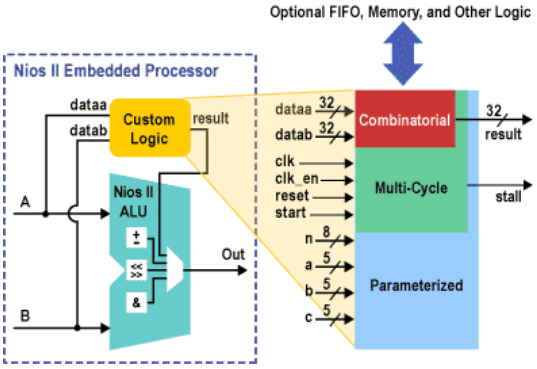


FIG. 5 – Exemple du processeur NIOSII

La figure 6 présente le processeur S6000 de Stretch [185] dont l'architecture est basée sur un Xtensa de Tensilica. Dans le S6000, les instructions spécialisées sont mises en œuvre dans le module appelé ISEF (*Instruction Set Extension Fabric*). L'ISEF est fortement couplé au processeur. Il possède sa propre file de registres, peut accéder à la mémoire locale du système et agit à la manière d'un co-processeur.

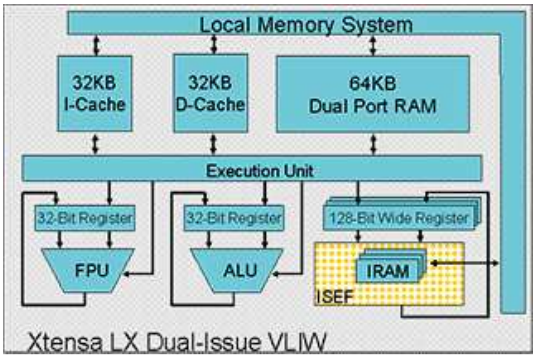


FIG. 6 – Exemple du processeur Stretch S6000 [185]

Les ASIP peuvent se présenter sur différents supports. Il est bien sûr possible de fondre un ASIP sur du silicium comme tout processeur, mais ils peuvent également être implantés dans des FPGA (*Field Programmable Gate Array*). En effet, grâce à leurs capacités d'intégration croissantes, les FPGA d'aujourd'hui intègrent directement des blocs mémoires, des blocs DSP, et permettent de synthétiser une architecture matérielle complexe, et même un processeur sur un composant. Les processeurs implantables dans ces circuits peuvent être, soit des macro-cellules spécialisées, dites processeur *Hard-core* (PowerPC chez Xilinx [211], ARM chez Altera [17]), soit des processeurs synthétisables, dits processeur *Soft-core* (MicroBlaze chez Xilinx, NIOSII chez Altera). Ces *soft-core* sont configurables à travers plusieurs paramètres (profondeur du pipeline, taille du cache, présence ou non de multiplieurs, etc.). Un processeur *soft-core* est réalisé à partir de la logique programmable présente sur la puce. Ainsi, un concepteur peut mettre autant de proces-

seurs *soft-core* que la puce le permet. À titre illustratif, un processeur NiosII dans sa déclinaison dite « rapide » occupe seulement 1% de la surface d'un FPGA de la famille Stratix II (composant à 180 000 cellules logiques). À l'inverse, le concepteur ne choisit pas le nombre de processeurs *hard-core* présents dans un FPGA. L'intérêt majeur des FPGA est la possibilité de faire du prototypage rapide et la proximité des cœurs de processeurs avec de la logique programmable rend possible la spécialisation de ces processeurs pour concevoir un ASIP.

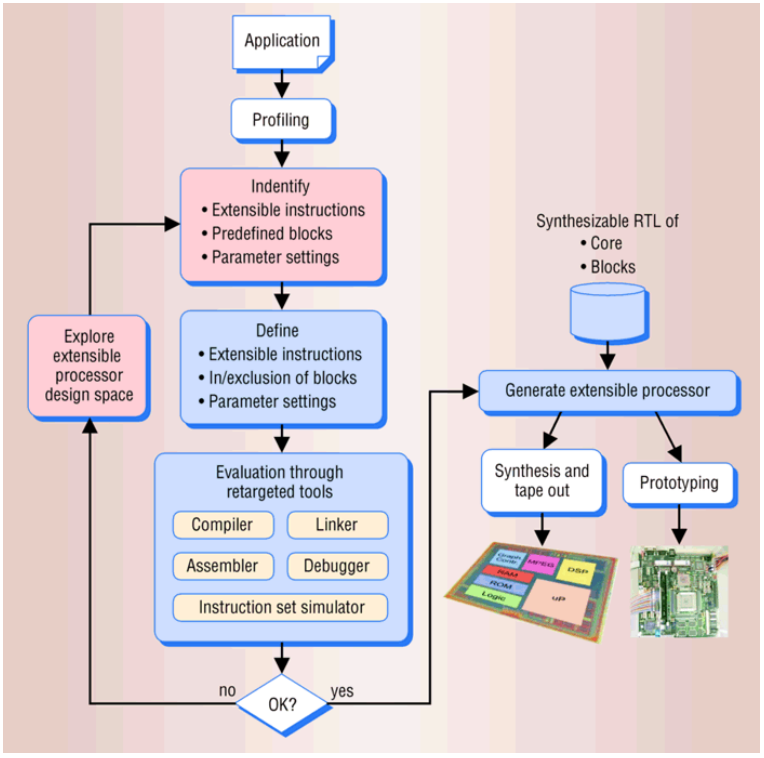


Fig. 7 – Flot générique de conception d'ASIP proposé par Henkel [102]

La conception des ASIP respecte en général un processus de spécialisation assez générique. La figure 7 montre un flot de conception simplifié proposé dans [102]. Le flot prend en entrée une application décrite dans un langage de haut niveau, typiquement en langage C, et génère en sortie un processeur spécialisé. Une première étape de profilage permet de révéler les portions de code les plus fréquemment exécutées pour lesquelles des instructions spécialisées permettraient d'améliorer les performances. Ensuite, l'étape d'identification d'instructions a pour objectif de trouver toutes les instructions candidates à une spécialisation. Puis, l'étape de sélection d'instructions cherche à conserver quelques instructions spécialisées parmi l'ensemble des instructions candidates. Il s'agit de déterminer quelle partie du programme sera exécutée par les instructions de base du processeur et quelle partie exploitera des instructions spécialisées. Ce problème est le problème de partitionnement logiciel/matériel au niveau instruction. Enfin, une phase d'évaluation permet de vérifier fonctionnellement le processeur et d'estimer les performances. Si les perfor-

mances sont conformes aux attentes, alors le processeur peut être généré ; sinon, il faut explorer une nouvelle fois l'espace de conception pour trouver une solution convenable.

Ce processus de spécialisation de processeur est majoritairement réalisé par le concepteur. Le génie humain propose des résultats de grande qualité mais les besoins en termes de performance, de mise sur le marché, combinés à la complexité des systèmes amènent à considérer des flots de conception automatisés [57, 62, 151]. L'effort requis pour étendre un jeu d'instructions peut même s'avérer plus coûteux en temps et en argent que la conception d'un ASIC [59]. L'espace de conception est tellement vaste qu'il est nécessaire de disposer d'outils et de méthodologies de conception appropriés. L'automatisation est donc un enjeu clé dans la spécialisation de processeurs, et en particulier dans l'extension de jeu d'instructions.

Méthodologie proposée et contributions de la thèse

Cette thèse s'inscrit dans cet objectif de disposer de la méthodologie et des outils associés de génération automatique d'extensions spécialisées pour des processeurs de type ASIP, et de compilation des applications sur ces nouvelles architectures. L'utilisation de la théorie des graphes est l'approche dominante dans les recherches sur l'automatisation de l'extension de jeu d'instructions, elle fournit un cadre analytique idéal [82]. La représentation du flot de contrôle ou de données d'une application sous forme de graphe est assez intuitive : les nœuds représentent les opérations et les liens représentent les dépendances. Les instructions spécialisées sont alors représentées par des sous-graphes, appelés motifs.

La figure 8 montre le flot de conception simplifié proposé dans cette thèse à travers un exemple, une portion de code issue de l'algorithme *SHA* (*Secure Hash Algorithm*) du benchmark MiBench [96]. Une partie frontale permet d'extraire le flot de données et de le représenter sous forme de graphe. La méthodologie proposée est appliquée sur des graphes flot de données, qu'ils représentent l'application complète ou bien une partie de celle-ci, choisie à l'issue d'une étape de profilage. Comme le montre la figure 8 (cadre pointillé), nous nous focalisons dans la thèse sur les étapes suivantes :

- génération d'instructions (c'est-à-dire génération de motifs),
- sélection d'instructions (c'est-à-dire sélection de motifs) et ordonnancement,
- allocation de registres,
- génération d'architecture,
- génération de code adapté à la nouvelle architecture.

L'extension de jeu d'instructions est étudiée depuis une bonne décennie et la littérature est riche en la matière [82]. Malgré cela, les nombreux problèmes que soulève l'extension de jeu d'instructions sont encore ouverts car ils sont complexes (voir intraitables) et nécessitent pour la plupart des temps de résolution très longs. En effet, le flot de données étant représenté sous forme de graphe, la génération d'instructions s'apparente au problème de génération de sous-graphes. Or, le nombre de sous-graphes est exponentiel

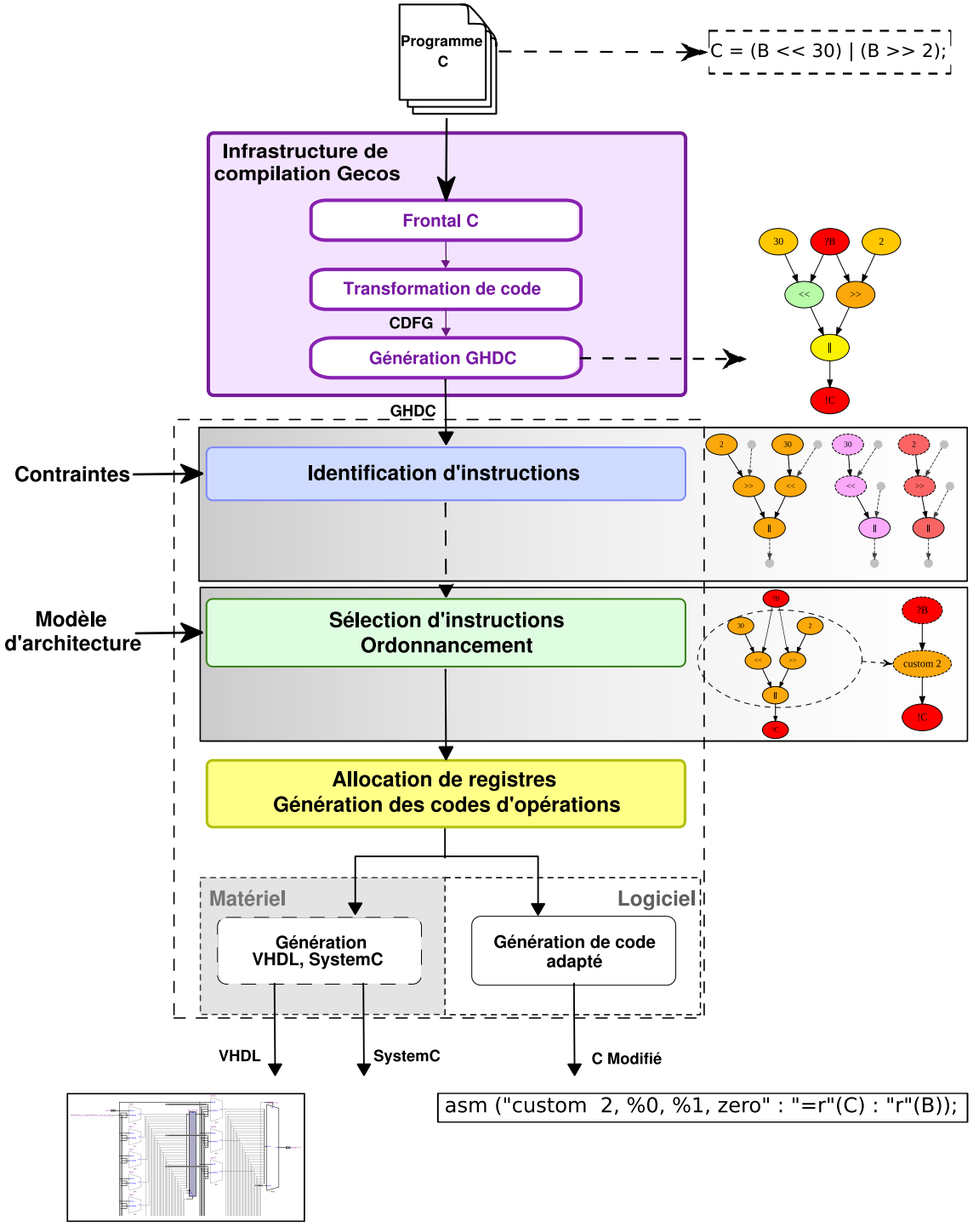


FIG. 8 – Flot de conception proposé dans cette thèse

en fonction du nombre de nœuds. L'identification d'instructions se résout donc en temps exponentiel. De même, la sélection d'instructions peut être modélisée sous la forme du problème de couverture de sommets d'un graphe qui est NP-complet [64]. L'ordonnancement sous contraintes de temps ou de ressources et l'allocation de registres sont également NP-complets [52, 87][1]. De très nombreuses techniques de résolution de problèmes NP-complets ont été appliquées, en partant des méta-heuristiques [123] jusqu'aux algorithmes dédiés [37, 58, 215], en passant par la programmation dynamique [62] ou la programmation linéaire par nombres entiers [28, 85]. Nous verrons les succès et les limites de ces approches, en remarquant qu'aucune de ces méthodes ne permet de résoudre tous les problèmes. Dans cette thèse, nous proposons des méthodes basées sur la programmation par contraintes [178] pour résoudre les problèmes de génération d'instructions, de sélection d'instructions, d'ordonnancement, et d'allocation de registres. L'idée de la programmation par contraintes est celle-ci : l'utilisateur décrit le problème à résoudre en définissant les contraintes sous la forme d'un *problème de satisfaction de contraintes*, et le problème est résolu par un *solveur de contraintes*. Autrement dit, l'homme décrit le problème, la machine le résout.

La première étape du processus de spécialisation est l'étape de génération d'instructions spécialisées. Cette première étape vise à générer l'ensemble des instructions candidates susceptibles d'apporter un gain. La génération d'instructions est un problème difficile dont le temps de résolution est exponentiel en fonction du nombre d'opérations. Cependant, une énumération exhaustive de toutes les instructions possibles est inutile, puisque des contraintes architecturales et technologiques rendent impossibles la mise en œuvre de certaines instructions. La génération doit donc être restreinte aux instructions qui peuvent être mises en œuvre matériellement. La première contribution de cette thèse consiste en **la formalisation du problème d'identification d'instructions spécialisées sous contraintes technologiques et architecturales, sous la forme d'un problème de satisfaction de contraintes**. La figure 8 montre les motifs générés à partir du graphe flot de données avec les contraintes suivantes : 2 entrées et 1 sortie. Les motifs étant générés à partir d'un code applicatif donné, cette approche a été vérifiée en utilisant au maximum les motifs générés pour exécuter ce code.

La deuxième étape du processus de spécialisation est l'étape de sélection d'instructions spécialisées. Cette étape consiste à sélectionner, parmi l'ensemble des candidats, ceux qui optimisent une certaine fonction de coût (typiquement la minimisation du temps d'exécution de l'application). La sélection d'instructions consiste à conserver un certain nombre d'instructions parmi un ensemble de candidats, étant donné le jeu d'instructions de base du processeur. La sélection d'instructions devient un problème de partitionnement logiciel/matériel, puisque certaines opérations de l'application sont exécutées par le

[1]Le problème de l'allocation de registres est NP-complet lorsqu'il est modélisé par le problème de coloriage d'un graphe

processeur alors que d'autres sont exécutées par l'extension. La deuxième contribution de cette thèse est **la formalisation du problème de sélection d'instructions sous la forme d'un problème de satisfaction de contraintes**. La figure 8 montre le résultat de la couverture du graphe, le graphe est couvert par un seul motif correspondant à une instruction spécialisée. De ce graphe couvert, il est possible de générer un nouveau graphe, appelé *graphe réduit*, dans lequel tous les groupes d'opérations sont *réduits* en un seul nœud. C'est ce graphe réduit qui est ensuite placé sur l'architecture et ordonnancé.

Nous avons choisi d'utiliser le processeur NiosII d'Altera pour valider le concept. Dans ce type d'architecture, l'extension du processeur est une unité fonctionnelle reconfigurable très fortement couplée au chemin de données du processeur. Nous avons appliqué la sélection d'instructions pour deux modèles d'architecture de l'extension : un premier modèle simple où toutes les données sont stockées dans la file de registres du processeur, et un second modèle où l'extension possède sa propre file de registres.

L'extension opère en parallèle de l'Unité Arithmétique et Logique (UAL) du processeur. Dans le cas où une instruction spécialisée nécessite plusieurs cycles pour se réaliser, l'UAL du processeur est libre pour exécuter une instruction en attendant. Nous proposons un ordonnancement d'application qui prend en compte cette possibilité. Cette technique est une amélioration par recalage d'instructions, elle constitue la troisième contribution de la thèse. Elle concerne **l'exploitation du parallélisme de l'UAL du processeur et de l'extension pour une exécution en parallèle d'une instruction spécialisée et d'une instruction du processeur**.

Une fois les instructions spécialisées sélectionnées, il faut mettre en œuvre ces instructions dans l'extension. Un aspect important du travail porte sur la génération du chemin de données de chaque instruction sélectionnée et la génération de l'architecture de l'extension, c'est-à-dire l'interconnexion entre les chemins de données et l'interface avec le processeur. Dans le cas où l'extension possède sa propre file de registres, il faut en plus résoudre le problème d'allocation de registres qui est un problème NP-complet lorsqu'il est modélisé comme un problème de coloriage de graphe [52]. La quatrième contribution de cette thèse est **l'utilisation de la programmation par contraintes pour réaliser l'allocation de registres dans le but de minimiser la surface de l'extension**. Pour l'aspect logiciel, nous utilisons les informations d'ordonnancement pour générer du code C incluant du code assembleur « inline » qui exploite les nouvelles instructions.

Enfin, les outils développés sont intégrés dans une infrastructure logicielle existante appelée Gecos [89] (*Generic Compiler Suite*), une plate-forme de compilation générique dont le flot est contrôlé par un système de script. Cet outil offre des passes de compilation classiques facilement réutilisables (propagation de constantes, transformation de code, etc.). Il met également à disposition une partie frontale (*front-end C*) qui permet de créer un graphe flot de données et de contrôle à partir d'un programme C. La dernière contribution majeure de cette thèse est **la mise en œuvre de la chaîne de compilation**

DURASE **(*generic environment for the Design and Utilisation of Application Specific Extensions*) qui permet d'étendre le jeu d'instructions d'un processeur à partir d'une application décrite en langage C**. Les outils sont intégrés dans l'infrastructure Gecos et écrits en langage Java.

Organisation du document

Le document comporte 6 chapitres. Le chapitre 1 présente un état de l'art sur les ASIP, leurs architectures, les méthodologies et outils de conception. Avant de détailler les contributions du travail de thèse, le chapitre 2 donne une vue d'ensemble sur la problématique de l'extension de jeu d'instructions.

La première étape du processus d'extension de jeu d'instructions est la génération d'instructions. Le chapitre 3 décrit la contribution sur ce sujet. La deuxième étape du processus est la sélection d'instructions. Le chapitre 4 explique comment la programmation par contraintes permet de résoudre ce problème. Ces étapes ont fait l'objet d'expérimentations sur des extraits de code issus d'applications multimédias et cryptographiques des benchmarks MediaBench [133], MiBench [96], PolarSSL [165], Mcrypt [150], et DSPstone [68]. Par la suite, ces portions de code sont appelées algorithmes.

Le chapitre 5 porte sur l'allocation de registres et les générateurs ; générateur de code, générateur d'architecture, et générateur de SystemC [189] pour la simulation. Le chapitre 6 présente le flot de conception *DURASE* et déroule ce flot à travers l'exemple de l'algorithme de chiffrement GOST. La simulation a permis de valider fonctionnellement l'extension générée et le code applicatif, et de mesurer les performances réelles.

Enfin, une conclusion résume les contributions apportées par cette thèse et propose des pistes pour des travaux futurs.

1

Conception d'ASIP : méthodologies et outils

Pour faire face à la complexité grandissante des applications et aux demandes croissantes de puissance de calcul, les systèmes embarqués modernes nécessitent des plates-formes de calcul efficaces. Les ASIC constituaient une réponse intéressante mais leurs coûts de fabrication et de développement deviennent prohibitif. De plus, ces plates-formes doivent être flexibles pour pouvoir s'adapter facilement aux nouvelles spécifications et réduire le temps de conception et ainsi satisfaire les exigences de coût et de temps de mise sur le marché. Les ASIP (*Application Specific Instruction-set Processor*) permettent de répondre à toutes ces attentes et sont en passe de supplanter les ASIC [154]. Un ASIP est un processeur dont l'architecture et le jeu d'instructions sont optimisés pour une application ou un domaine d'applications. L'ASIP est programmable et présente une grande flexibilité tout en proposant de meilleures performances qu'un processeur à usage général grâce à sa spécialisation.

Pour programmer un ASIP, l'algorithme à exécuter est décrit dans un langage de haut niveau et traduit par un compilateur en un code dans un langage machine, interprétable par l'ASIP. La génération d'un code qui exploite parfaitement l'architecture de l'ASIP est un véritable défi pour le compilateur. Le compilateur nécessite des informations précises sur l'architecture du processeur et l'algorithme doit être décrit de façon à faciliter le travail du compilateur. Une fine adéquation entre l'algorithme, l'architecture, et le compilateur est la base indispensable du succès d'un ASIP.

La conception d'un ASIP ne se résume pas à la seule conception architecturale du composant, il faut également disposer de la suite d'outils logiciels (compilateur, simulateur, débogueur, etc.) qui le rend exploitable. Développer ces outils manuellement pour chaque nouveau processeur est long, sujet à erreur, et coûteux, d'où la nécessité d'automatiser le processus.

Pour cela, des approches basées sur des langages de description d'architecture proposent de générer automatiquement les outils de développement et la description architecturale du processeur à partir d'un modèle générique de celui-ci. Cette approche conduit à générer complètement l'architecture du processeur et les outils associés pour chaque nouveau processeur. Nous la désignons par la suite comme étant de la conception complète d'ASIP.

Les dernières années ont vu apparaître les ASIP configurables [93, 180, 190, 199], qui proposent une configuration ou une extension de leur jeu d'instructions à travers une personnalisation des instructions et de leurs unités fonctionnelles, ou bien à travers des instructions spécialisées et leurs unités fonctionnelles associées. Ce type d'ASIP présente une grande facilité d'adaptation à de nouvelles applications. Leur architecture est composée de deux parties : une partie *statique*, qui correspond à l'architecture minimale, et une partie configurable, adaptée à une application donnée. Ces ASIP proposent des mécanismes d'extensions de jeu d'instructions qui permettent d'ajouter facilement une instruction spécialisée, et possèdent leur panoplie d'outils de programmation et vérification. La conception de ce type d'ASIP repose sur un processeur et des outils existants, qu'il n'est pas nécessaire de générer pour chaque nouveau processeur et correspond à de la conception partielle.

Ce chapitre est structuré en trois parties. Dans la première partie, nous présentons la conception d'ASIP basée sur les langages de description d'architecture. Dans la deuxième partie, nous détaillons la conception partielle d'ASIP. Enfin, la dernière partie résume les différents avantages et inconvénients de ces approches.

1.1 Conception complète d'ASIP

La conception complète consiste à concevoir complètement l'architecture du processeur et à développer la suite d'outils associés pour le programmer. Cette approche est basée sur des langages de description d'architecture tels que LISA [106], nML [71], ISDL [97], EXPRESSION [98], Armor [153].

L'idée est de décrire l'architecture du processeur grâce à un langage de modélisation, pour créer un modèle du processeur. À partir du modèle du processeur, il est possible de générer la description du processeur dans un langage de description architecturale (VHDL, Verilog) qui peut être traité par n'importe quel outil de synthèse standard et de générer la suite d'outils associés (compilateur, simulateur, etc.). Le langage LISA, exemple typique de cette approche, est maintenant décrit.

1.1.1 Exemple du langage LISA

1.1.1.1 Le langage LISA

Le langage LISA (*Language for Instruction Set Architectures*) a été développé pour permettre la génération automatique d'un code HDL synthétisable et des outils de développement associés. La syntaxe du langage LISA permet de décrire le jeu d'instructions d'un

processeur, son modèle d'exécution (SIMD, MIMD, VLIW) et son pipeline. Le processus de génération s'appuie sur des informations relatives au jeu d'instructions et aux propriétés architecturales du processeur. Ces informations sont décrites dans six parties distinctes.

1. Le modèle de mémoire (*memory model*) contient les informations sur les registres et les mémoires du système, leur nombre, leur largeur de bits, et leur plage d'adresse.
2. Le modèle de ressources (*ressource model*) décrit les ressources disponibles et les besoins en ressources de chaque opération.
3. Le modèle du jeu d'instructions (*Instruction Set Model*) identifie les combinaisons valides entre les opérations matérielles et les opérandes possibles.
4. Le modèle comportemental (*behavioral model*) est utile pour la simulation. Il permet d'élever le niveau d'abstraction et contient un modèle du comportement des opérations plutôt qu'une description de l'activité réelle du matériel.
5. Le modèle de temps (*timing model*) contient les informations sur les séquences d'activation des opérations et des unités de calcul, les informations sur les latences des instructions. Ces informations sont utilisées par le compilateur lors de la phase d'ordonnancement.
6. Le modèle micro-architectural (*micro-architectural model*) permet de grouper des opérations matérielles pour créer des unités fonctionnelles et contient la mise en œuvre exacte au niveau micro-architecture des composants comme des additionneurs, multiplieurs, etc.

Par ailleurs, un aspect clé du développement d'architecture est la capacité à disposer d'outils opérant à plusieurs niveaux de précision. Par exemple, la simulation comportementale permet de valider rapidement le processeur au niveau fonctionnel. Ensuite, une simulation au cycle près, plus précise mais plus longue, permet de simuler plus finement le processeur. Le langage LISA permet de générer les outils avec plusieurs niveaux de précisions.

1.1.1.2 Environnement de conception

La conception et la mise en œuvre d'un processeur embarqué nécessite généralement des phases d'exploration d'architecture, puis de mise en œuvre d'architecture, de développement de l'application, et enfin de vérification et d'intégration au niveau système. L'environnement de conception LISA permet de générer automatiquement les outils nécessaires à chaque étape. Trois environnements de conception sont distingués.

Le premier est l'environnement de conception d'architecture pour l'exploration et la génération du processeur qui permet de raffiner le processeur à travers un processus itératif comme illustré par la figure 1.1. À chaque itération, la description du processeur est enrichie, puis le processeur et la suite d'outils sont générés. Il faut ensuite vérifier si d'un côté le processeur répond aux critères (surface, fréquence d'horloge, ou consommation d'énergie par exemple), et de l'autre, la suite d'outils permet d'évaluer les performances qu'on peut

espérer du processeur. La génération de l'architecture du processeur et de la suite d'outils à partir d'une même et unique représentation permet de garantir la cohérence entre ces deux aspects.

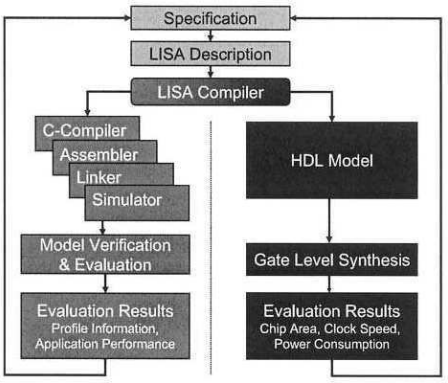


FIG. 1.1 – Exploration et implémentation avec LISA

Le deuxième est un environnement de développement logiciel pour le développement de l'application et propose une version améliorée des outils obtenus à l'étape précédente. En particulier, la vitesse des outils de simulation est améliorée lorsque cela est possible [105].

Une fois disponible, le processeur doit être intégré et vérifié dans le contexte d'un système complet, comportant des mémoires, des bus d'interconnexions, etc. Le troisième environnement fournit une API (*Application Programming Interface*) pour connecter le simulateur généré par LISA avec d'autres simulateurs. Cet outil aide à la vérification et à l'intégration au niveau système.

1.1.1.3 Mise en œuvre de l'architecture

Même si l'environnement permet de générer automatiquement une partie de l'architecture du processeur, une part non négligeable de la description doit être faite manuellement. Le générateur ne permet de générer que les parties suivantes :

- les structures gros grain du processeur, telles que la file de registres, le pipeline, les registres de propagation d'informations ;
- le décodeur d'instructions qui gère les signaux de contrôle et de données pour l'activation des unités fonctionnelles ;
- le contrôleur du pipeline qui gère l'interblocage du pipeline, le *flush* du pipeline, et les mécanismes de *data forwarding*[1].

Le générateur s'appuie sur les descriptions LISA qui sont composées de ressources et d'opérations. Les ressources décrivent les éléments matériels de l'architecture, et les opérations décrivent le jeu d'instructions, le comportement et les temps d'exécution de l'opération. La figure 1.2 présente un extrait de la description LISA de l'architecture ICORE [91]. L'architecture ICORE est présentée par la figure 1.3.

[1] le *data forwarding* est expliqué p. 44

```
RESOURCE
{
  REGISTER S32          R([0..7])6; /* GP Registers */
  REGISTER bit[11]      AR([0..3]); /* Address Registers */

  DATA_MEMORY S32 RAM([0..255]);    /* Memory Space */
  PROGRAM_MEMORY U32 ROM([0..255]);/* Instruction ROM */

  PORT bit[1]           RESET;          /* Reset pin */
  PORT bit[32]          MEM_ADDR_BUS; /* External address bus */

  PIPELINE ppu_pipe = { FI; ID; EX; WB };
  PIPELINE_REGISTER IN ppu_pipe {
    bit[6] Opcode;
    ...
  };
}
```

FIG. 1.2 – Déclaration des ressources dans un modèle LISA

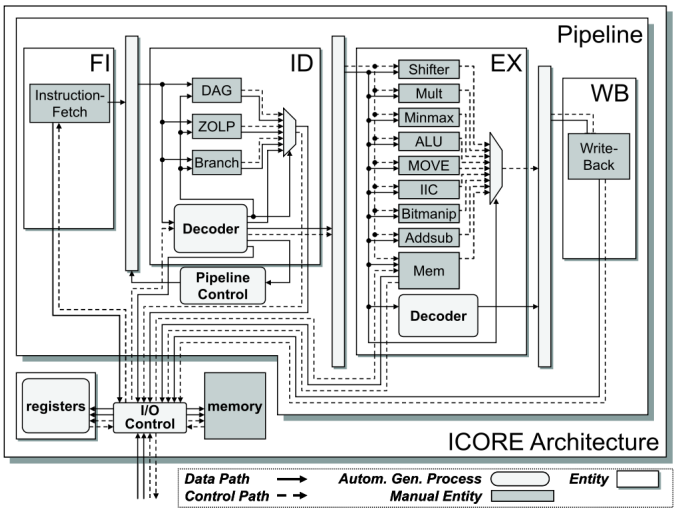


FIG. 1.3 – Exemple du processeur ICORE généré par LISA

L'architecture ICORE est composée de deux files de registres : une première à usage général appelée *R*, composée de huit registres de 32 bits et une seconde dédiée aux registres d'adresses, appelée *AR*, composée de quatre registres de 11 bits. Les ressources mémoires pour les données et pour les instructions sont décrites, ainsi que les ports du processeur. La description contient les informations sur le pipeline du processeur. L'architecture ICORE possède quatre étages de pipeline : FI (*Instruction Fetch*) l'étage qui permet de chercher l'instruction à décoder, ID (*Instruction Decode*) l'étage de décodage de l'instruction, EX (*Instruction Execution*) l'étage d'exécution de l'instruction, et WB (*Write-Back to registers*) l'étage d'écriture dans la file de registres. La figure 1.3 fait apparaître ces quatre étages de pipeline. Le langage LISA permet d'ajouter des *registres de pipeline* (`PIPELINE_REGISTER`), qui propagent des informations telles que le code d'opération et les registres opérandes à travers les étages de pipeline.

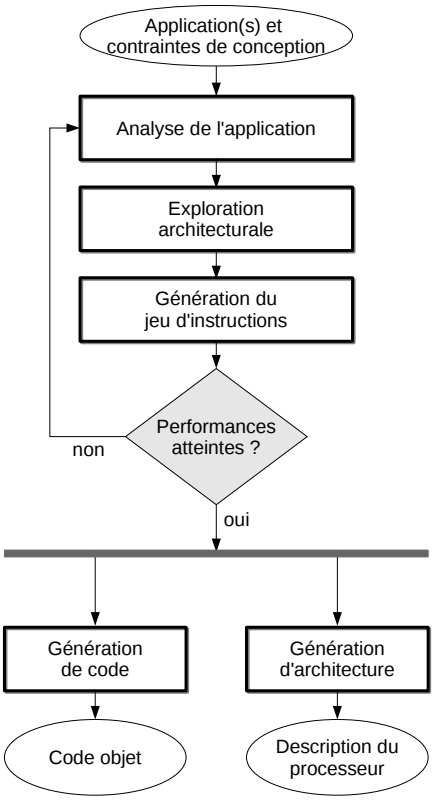


FIG. 1.4 – Méthodologie de conception complète d'ASIP

Les opérations peuvent être fusionnées au sein d'une unité fonctionnelle. Le générateur permet seulement de générer l'interface de l'unité fonctionnelle sous forme de *wrapper*. Le contenu de l'unité fonctionnelle doit être complété manuellement. Les parties grisées de la figure 1.3 mettent en évidence les parties de l'architecture qui doivent être décrites manuellement. La figure montre bien que malgré la génération automatique du squelette du processeur, une grosse partie de l'architecture doit être décrite à la main.

1.1.2 Méthodologie

Le processus de conception d'ASIP suit généralement une méthodologie assez générique comme celle présentée dans [115] où se dégagent cinq étapes majeures. Ces cinq étapes sont illustrées par la figure 1.4.

1. **Analyse de l'application :** les entrées sont une application ou un ensemble d'applications et des contraintes de conception. Cette étape transforme une application écrite dans un langage de haut niveau en une représentation intermédiaire. Des transformations peuvent avoir lieu sur cette représentation. L'analyse permet de détecter les caractéristiques de l'application pour guider l'exploration architecturale.
2. **Exploration architecturale :** l'espace des solutions architecturales est exploré et l'exploration est guidée par les contraintes de conception et les caractéristiques de

l'application. Les performances des architectures potentielles sont estimées et celles qui respectent les contraintes sont sélectionnées.

3. **Génération du jeu d'instructions :** un jeu d'instructions spécifique est généré pour l'application donnée et pour l'architecture sélectionnée.
4. **Génération de code :** le code objet de l'application est généré pour le jeu d'instructions.
5. **Génération d'architecture :** le jeu d'instructions et l'architecture du processeur sont décrits par un langage de description architecturale puis synthétisés par les outils de synthèse standards.

Entre l'étape de génération du jeu d'instructions, la génération du processeur et du code applicatif, une phase d'estimation des performances permet de tester si le processeur est conforme aux spécifications. Si oui, alors le processeur peut être généré ; sinon, le processus est relancé à partir de l'étape 1.

1.1.2.1 Estimation des performances

Pour savoir si un processeur répond aux exigences, il est nécessaire de mesurer ses performances. Deux techniques majeures sont distinguées pour l'estimation des performances. La première est basée sur la simulation, et la seconde sur l'ordonnancement.

L'estimation des performances peut être réalisée par simulation. Un modèle de simulation du processeur est généré d'après ses caractéristiques, et l'application est simulée sur ce modèle pour connaître ses performances [42, 61, 125, 218]. L'inconvénient est qu'il est nécessaire de générer un modèle du processeur et la simulation est souvent longue.

Lorsque l'estimation des performances est basée sur l'ordonnancement, le problème est formulé comme un problème d'ordonnancement sous contraintes de ressources, où les ressources sont les composants de l'architecture. L'application est ordonnancée pour avoir une estimation rapide du nombre de cycles nécessaires à son exécution [116, 85, 166]. Cette approche est plus rapide que la simulation.

Les recherches sur la spécialisation de processeurs se sont focalisées majoritairement sur l'amélioration des performances sous contrainte de surface. Des approches plus récentes proposent de prendre en compte plusieurs objectifs (consommation d'énergie, surface, performance) pour aboutir à des compromis [45, 171]. Dans [45], les auteurs présentent un algorithme d'approximation d'évaluation des compromis de conception. En particulier, ils s'intéressent au compromis performance/surface, où il n'est pas possible d'améliorer un objectif sans empirer l'autre. Dans [171], l'objectif est à la fois de réduire le temps d'exécution et la consommation d'énergie.

1.1.2.2 Méthodologie MESCAL

La méthodologie MESCAL [94, 109] propose de concevoir un ASIP autour de cinq éléments essentiels.

1. **Élément 1 : « Judiciously Using Benchmarking »**, utiliser les applications de manière judicieuse. Le développement d'un ASIP est par définition dirigé par les applications cibles. La conception est souvent guidée par un ensemble d'applications de référence, qui doivent être représentatives du domaine d'applications visé.
2. **Élément 2 : « Inclusively Identify the Architectural Space »**, identifier l'espace architectural complet. L'espace de conception d'un ASIP consiste en un large éventail de possibilités selon plusieurs dimensions. Si l'espace entier n'a pas besoin d'être considéré pour un domaine d'applications précis, il est important que la base de conception initiale puisse inclure une large gamme de déclinaisons possibles. En effet, écarter dès le début un espace de conception peut éliminer certaines architectures intéressantes.
3. **Élément 3 : « Efficiently Describe and Evaluate the ASIPs »**, décrire et évaluer efficacement l'ASIP. Pour considérer une gamme de déclinaisons de l'architecture la plus large possible, il est important de pouvoir décrire et évaluer l'ASIP pour les applications considérées. Les outils logiciels correspondants à l'ASIP doivent donc être disponibles.
4. **Élément 4 : « Comprehensively Explore the Design Space »**, explorer l'espace de conception de manière approfondie. L'exploration de l'espace de conception est un processus itératif avec asservissement où chaque point de conception est évalué selon une métrique. L'exploration est réalisée grâce à un environnement qui permet de compiler l'application pour l'architecture et de simuler l'exécution pour obtenir des résultats quantitatifs et apprécier les performances. L'aspect clé est de pouvoir recibler rapidement les outils pour chaque point de conception.
5. **Élément 5 : « Successfully Deploy the ASIP »**, déployer l'ASIP avec succès. Un ASIP peut être ultra performant en proposant une adéquation excellente entre l'application et l'architecture et à la fois complètement sous exploité si il est mal programmé. Programmer un ASIP directement en assembleur est impensable aujourd'hui étant donnée la complexité des applications et les temps de développement très limités. Il est donc important de disposer d'un compilateur suffisamment puissant pour exploiter au maximum les capacités d'une architecture.

De la même façon, un compilateur puissant est sous exploité si l'architecture est peu adaptée à l'application, et un compilateur et une architecture ne peuvent que faiblement améliorer les performances d'une application mal décrite. La conception d'un ASIP conduit idéalement à une adéquation parfaite entre l'algorithme, l'architecture, et le compilateur.

La méthode de conception complète propose de générer un processeur et sa suite d'outils associés à partir d'un même modèle du processeur pour garantir l'adéquation entre l'architecture et le compilateur. L'application est généralement décrite dans un langage de haut niveau. Nous avons vu à travers l'exemple du langage LISA que le processus n'est pas entièrement automatisé et qu'il faut toujours mettre en œuvre manuellement certaines parties critiques du processeur pour aboutir à un processeur optimisé. Chaque nouveau processeur doit être vérifié et optimisé, ce qui entraîne des coûts d'ingénierie non récurrents.

Pour réduire ces coûts, une autre approche de conception d'ASIP vise à considérer un processeur existant et à concevoir seulement la partie dédiée, c'est la conception partielle qui est maintenant décrite.

1.2 Conception partielle d'ASIP

La conception partielle consiste à partir d'un processeur existant, vérifié et optimisé, qui possède sa suite d'outils associés (compilateur, simulateur, etc), à le configurer, et à étendre son jeu d'instructions pour le spécialiser en regard des exigences d'une application ou d'un domaine d'applications. Ce type de processeur est configurable à travers des éléments comme la file de registres, la profondeur du pipeline, la présence ou non de multiplieurs, etc. Par ailleurs, ce type de processeur dispose de mécanismes pour ajouter facilement une nouvelle instruction au jeu d'instructions, et pour qu'elle soit prise en compte par le compilateur. En général, cette technique demande au concepteur de fournir une description architecturale de la nouvelle instruction et de modifier manuellement le code applicatif qui fait appel à la nouvelle instruction. Dans un premier temps, nous donnons un exemple de processeur configurable et extensible, puis nous passons en revue les autres architectures existantes. Ensuite, nous décrivons les méthodologies et techniques de compilation qui permettent d'automatiser le processus d'extension de jeu d'instructions.

1.2.1 Exemple

Pour illustrer ce qu'est un processeur configurable et extensible, nous proposons d'étudier le processeur NiosII. C'est le processeur que nous avons choisi d'utiliser pour valider notre méthodologie et nos outils développés pendant cette thèse. Présenter ce processeur en détail permet de bien comprendre les aspects décrits tout au long du document.

Le processeur NiosII est un processeur RISC (*Reduced Instruction Set Computer*) 32 bits décliné en trois versions : une première version dite « économique », une deuxième dite « standard », et une troisième dite « rapide » (*Nios2Fast*). La grande différence entre les trois versions se situe au niveau du nombre d'étages de pipeline. La déclinaison économique du NiosII est un processeur qui possède un seul étage de pipeline, et chaque instruction nécessite 6 cycles pour être exécutée. Un NiosII standard a cinq étages de pipeline. Le *Nios2Fast* en a six. Un processeur NiosII a une file de registres de 32 registres de 32 bits. Une liste non exhaustive des aspects configurables du processeur NiosII est donnée :

- présence de multiplieurs ou non,
- unité de gestion mémoire (MMU),
- taille du cache et des mémoires,
- vecteur d'interruptions,
- module de debug JTAG,
- etc.

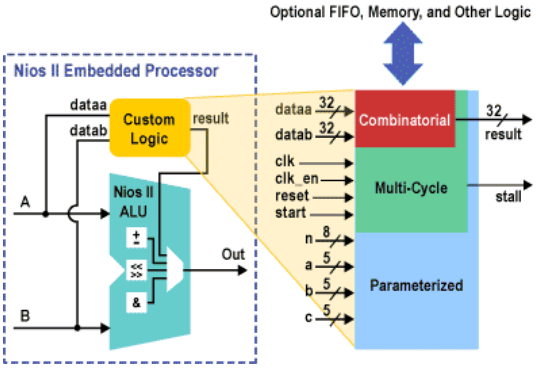


FIG. 1.5 – Ajout d'un module spécialisé à un processeur NIOSII

La figure 1.5 montre l'intégration du module spécialisé fortement couplé au chemin de données du processeur. Le module spécialisé, qui met en œuvre l'instruction spécialisée, doit respecter une interface avec le processeur. En particulier, le nombre d'opérandes d'entrée et de sortie de l'instruction spécialisée est le même que pour une instruction de base.

Un processeur NIOSII, quelle que soit sa déclinaison, met à disposition 8 bits dans l'encodage d'une instruction pour les instructions spécialisées, ce qui permet d'ajouter $2^8 = 256$ instructions. La figure 1.6 illustre le codage d'une instruction spécialisée pour le NIOSII, appelée instruction de type R, où les six premiers bits servent à indiquer que l'instruction est une instruction spécialisée, les huit bits suivants (`N`) spécifient le numéro de l'instruction spécialisée. Le NIOSII autorise plusieurs types d'instructions spécialisées, qui sont décrits en détail dans le chapitre 5, p. 107. Nous expliquons notamment l'utilisation des bits `readra`, `readrb`, et `readrc`. Les quinze bits restant servent à encoder les opérandes d'entrée et sortie. Chaque opérande est codé sur cinq bits, ce qui permet d'adresser les 32 registres de la file de registres du processeur.

Nous proposons d'illustrer ce mécanisme d'extension de jeu d'instructions par l'ajout d'une instruction spécialisée très simple qui réalise l'échange de la partie haute avec la partie basse d'un mot de 32 bits. En logiciel, cette opération consiste en une série d'instructions de décalage et masquage, ce qui nécessite plusieurs cycles, alors qu'en matériel, ce simple échange peut être fait en un seul cycle. La figure 1.7 montre le code VHDL qui réalise cet échange. On remarque que l'instruction spécialisée respecte l'interface avec le processeur, à savoir un port `dataa` qui véhicule l'opérande d'entrée, et un port `result` pour l'opérande de sortie. Ce module est intégré dans le processeur par le biais de l'outil

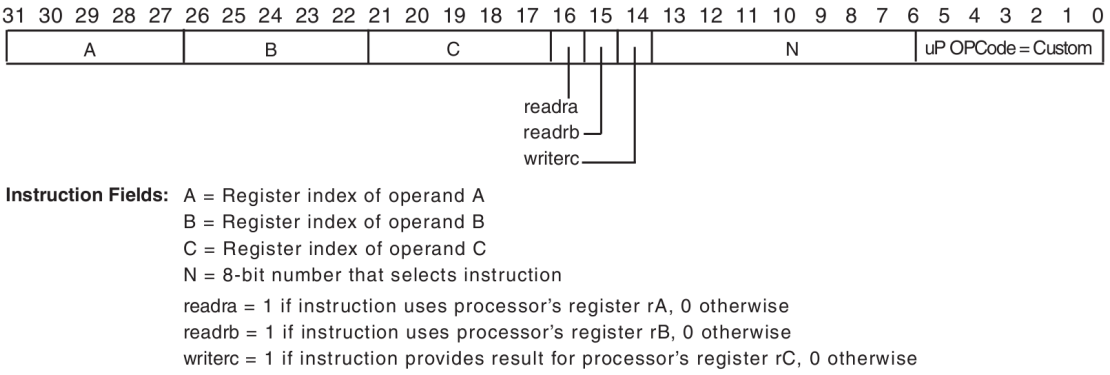


FIG. 1.6 – Codage d'une instruction de type R du NIOSII [19]

de conception Altera SOPCBuilder qui permet de générer un système complet, processeur et périphériques.

```
entity HSWAP is
  port (
    dataa : in std_logic_vector(31 downto 0);
    result : out std_logic_vector(31 downto 0)  );
end entity HSWAP;

architecture arch of HSWAP is
     signal vhdl_tmp : std_logic_vector(31 downto 0);
begin
    vhdl_tmp(15 downto 0)  <= dataa(31 downto 16);
    vhdl_tmp(31 downto 16) <= dataa(15 downto 0);
    result <= vhdl_tmp;
end architecture arch;
```

FIG. 1.7 – Code VHDL pour l'échange de demi-mots

Pour pouvoir exploiter une instruction spécialisée, le code applicatif doit respecter une interface précise. La figure 1.8 montre un extrait du fichier d'entête `system.h` dans lequel sont déclarées les macros pour l'utilisation des instructions spécialisées. Ce fichier est automatiquement généré par l'outil SOPCBuilder. Dans cet exemple, `ALT_CI_HSWAP_N` désigne le numéro de l'instruction spécialisée et vaut 0 (le numéro de l'instruction spécialisée peut être précisé par le concepteur ou géré automatiquement par l'outil SOPCBuilder). La macro `ALT_CI_HSWAP(A)` est associée à une fonction intrinsèque de GCC, qui permet de faire le lien entre la macro et l'instruction spécialisée. L'utilisation se fait comme n'importe quelle macro. La figure 1.9 illustre un petit exemple d'utilisation dans lequel la macro `ALT_CI_HSWAP(A)` est appelée à la ligne 8. La figure 1.10 montre le code assembleur généré correspondant au code de la figure 1.9. Le code assembleur réalise le chargement du registre `r2` avec la valeur `0x12345678` (un premier chargement de la partie haute 0x1234, c.-à-d. 4660 en décimal, réalisé par l'instruction `movhi` puis un ajout de la partie basse 0x5678, c.-à-d. 22136 en décimal, réalisé par l'instruction `addi`), puis l'appel de l'instruction spécialisée `custom` avec le numéro 0, le registre `r2` est à la fois opérande de sortie

et opérande d'entrée. Le registre `zero` est un registre qui vaut toujours zéro et sert ici à combler le mot d'instruction puisqu'un seul opérande d'entrée est nécessaire à l'opération.

```
#define ALT_CI_HSWAP_N 0x00
#define ALT_CI_HSWAP(A) __builtin_custom_ini(ALT_CI_HSWAP_N,(A))
```

FIG. 1.8 – Extrait du fichier d'entête `system.h` pour la définition des macros des instructions spécialisées

```
1  #include "system.h"
2
3  int main (void)
4  {
5   int a = 0x12345678;
6   int a_swap = 0;
7
8   a_swap = ALT_CI_HSWAP(a);
9   return 0;
10 }
```

FIG. 1.9 – Appel de l'instruction spécialisée dans le code source

```
1 movhi   r2,4660
2 addi    r2,r2,22136
3 custom  0,r2,r2,zero
```

FIG. 1.10 – Code assembleur généré correspondant au code de la figure 1.9

Ce court exemple permet de montrer avec quelle facilité le concepteur peut ajouter une instruction spécialisée au processeur NIOSII, où le processeur et sa suite d'outils associés existent déjà. Il reste cependant à la charge du concepteur la description de l'architecture de l'instruction spécialisée et l'écriture du code applicatif qui exploite ces instructions.

1.2.2 Architecture des processeurs extensibles

Le processeur NIOSII est un exemple de processeur extensible, mais il existe d'autres solutions architecturales. Nous proposons une classification de ces ASIP à travers trois critères : le modèle d'exécution, le couplage, et la technologie.

1.2.2.1 Modèle d'exécution

Un processeur peut exploiter deux formes de parallélisme pour accélérer les traitements : le parallélisme temporel, et le parallélisme spatial. Ces deux formes de parallélisme sont illustrées par la figure 1.11.

Pour exécuter une instruction, un processeur effectue une série de tâches : récupérer l'instruction dans la mémoire et la charger dans le registre d'instruction, modifier la valeur

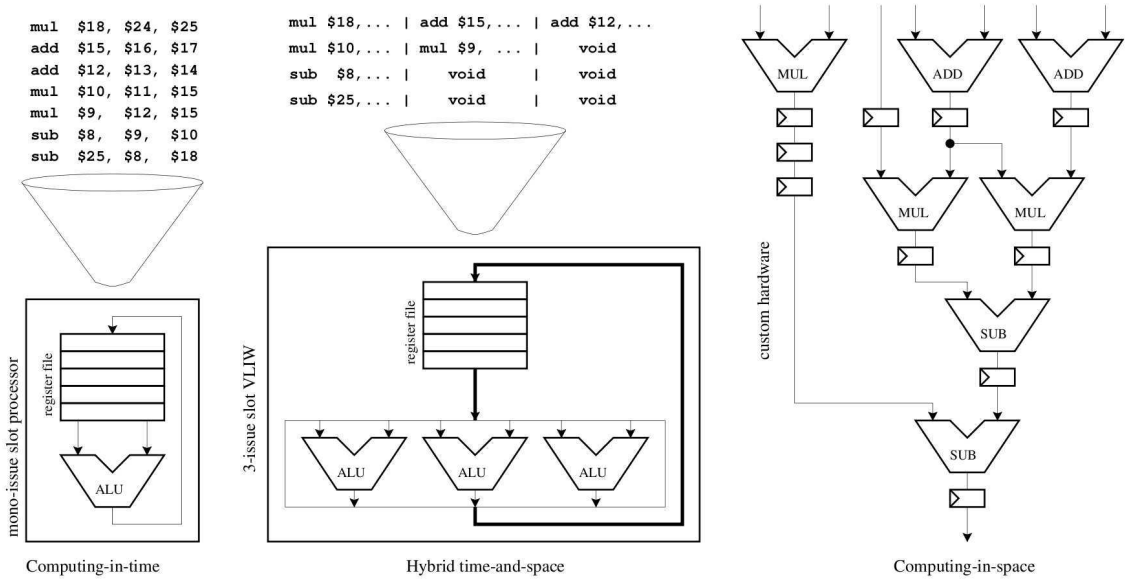


FIG. 1.11 – Parallélisme spatial et temporel, exemple issu de [122]

du compteur ordinal (compteur d'instruction), déterminer le genre d'instruction venant d'être chargée, exécuter l'instruction, retourner à la première étape pour exécuter l'instruction suivante. Implémenter un chemin de donnée qui réalise toutes ces tâches en un cycle résulte en un chemin critique très long qui contraint la vitesse d'horloge du processeur. Le parallélisme temporel, connu sous le nom de technique du pipeline (*pipelining*), consiste à *casser* le chemin critique en décomposant les instructions en de nombreuses sections élémentaires, chacune pouvant être exécutée en parallèle par un composant matériel spécifique. L'architecture ICORE illustrée par la figure 1.3 est composée de quatre étages de pipeline.

Le parallélisme spatial est l'exécution en parallèle et au même instant de plusieurs instructions non dépendantes en données. L'exemple de la figure 1.11 met en exergue l'exécution de deux additions et une multiplication au même moment, puis de deux multiplications. C'est le *parallélisme au niveau instruction*.

Un processeur peut suivre plusieurs modèles d'exécution que nous ne détaillons pas ici, et un ASIP suit en général deux modèles d'exécution très répandus : le modèle d'exécution RISC, et le modèle VLIW.

Modèle d'exécution RISC. Un processeur RISC (*Reduced Instruction Set Computer*) est un processeur possédant un jeu d'instructions réduit. Son mode d'exécution, aussi appelé *single-issue*, est le mode d'exécution le plus simple. Il correspond au mode de fonctionnement SISD (*Single Instruction Single Data*) selon la taxonomie définie par Flynn [76] et est basé sur une architecture de Von Neumann. Dans ce mode d'exécution, une seule instruction est exécutée à la fois. De nombreux ASIP suivent ce mode d'exécution : Garp [175], Chimaera [212], OneChip [204], etc. L'exemple de la figure 1.11 illustre ce

fonctionnement où les instructions sont exécutées une par une, et se partagent l'unique unité fonctionnelle du processeur.

Modèle d'exécution VLIW. Un processeur VLIW (*Very Long Instruction Word*) est un processeur comprenant plusieurs *slots* d'exécution, c'est-à-dire plusieurs unités fonctionnelles capables de s'exécuter en parallèle. Un processeur VLIW est donc capable d'exécuter en même temps plusieurs instructions sur plusieurs données. Pour pouvoir encoder toutes ses informations (instructions à exécuter et registres opérandes), le mot de l'instruction doit être très long, d'où le nom de processeur à mot d'instructions très long (128 bits par exemple au lieu de 32 bits pour un processeur RISC). Un processeur VLIW permet d'exploiter le parallélisme au niveau instruction. C'est le compilateur qui a la charge de détecter le parallélisme et de générer le code capable d'exploiter les unités fonctionnelles. De nombreux processeurs DSP s'appuient sur ce type de modèle d'exécution, comme par exemple le TCI6487 de Texas Instruments [191]. Le nombre de slots d'exécution varie selon l'architecture. Par exemple, le nombre de slots de l'architecture ADRES [151] est paramétrable.

Dans [118], les auteurs présentent une architecture basée sur un processeur VLIW à quatre PE (*Processing Element*), entièrement mise en œuvre dans un FPGA. Chaque PE contient une UAL et un module qui peut accueillir des instructions spécialisées. L'architecture PADDI [213] présente également un modèle d'exécution VLIW où chaque slot d'exécution est un nanoprocesseur.

1.2.2.2 Couplage

La spécialisation partielle consiste à ajouter un module qui met en œuvre les instructions spécialisées. Ce module spécialisé peut être intégré à différents endroits. Les approches architecturales de l'intégration de la logique spécialisée diffèrent selon le degré de couplage entre la logique spécialisée et le processeur à usage général. Nous distinguons trois degrés de couplage, illustrés par la figure 1.12 :

1. Unités fonctionnelles – couplage fort
2. Co-processeurs – couplage lâche
3. Unités de traitement attachées ou externes – couplage très lâche

Unités fonctionnelles. Dans ce cas de figure, la logique spécialisée est fortement couplée au chemin de données du processeur, en parallèle de l'UAL du processeur. Cette unité est accessible à travers des instructions spécialisées du jeu d'instructions du processeur. Ces architectures font partie de la classe DEMETER d'après la classification proposée par [172]. Un des pionniers en la matière est l'architecture PRISC [175, 174], qui combine un processeur RISC avec des PFU (*Programmable Functionnal Unit*). Chaque PFU met en œuvre une fonction combinatoire à deux entrées et produit un résultat. L'architecture OneChip [204]

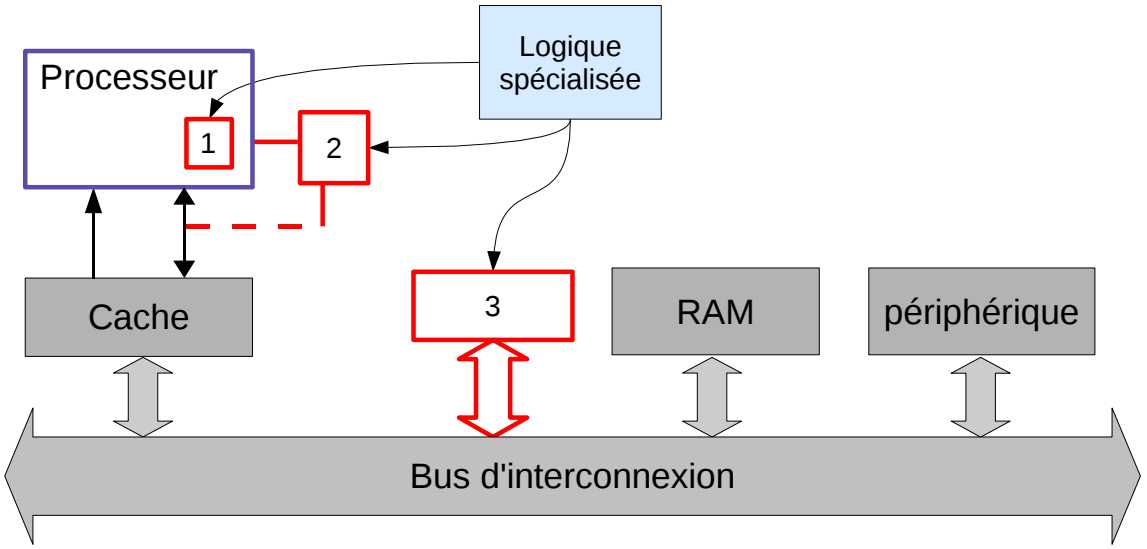


FIG. 1.12 – Différents couplages de la logique spécialisée

a anticipé les capacités d'intégration des FPGA et associe un processeur MIPS à des PFU, le tout mis en œuvre dans un FPGA.

Un autre exemple est l'architecture Chimaera [100, 212], dans laquelle l'unité fonctionnelle reconfigurable travaille en parallèle de l'ALU et possède un accès à des registres *fantômes* (registres dupliqués, dans la partie reconfigurable, à partir d'un sous-ensemble des registres du processeur). L'architecture Chimaera est placée sur un FPGA. Enfin, l'architecture ConCISE [123] se base sur un processeur MIPS et utilise un CPLD (*Complex Programmable Logic Device*) pour mettre en œuvre les unités fonctionnelles reconfigurables.

Co-processeurs. Dans le cas où l'unité de traitement spécialisée est lâchement couplée, celle-ci est vue comme un co-processor. L'unité est couplée au processeur en parallèle de l'UAL mais possède un accès direct à la mémoire à travers un bus local ou bien à travers le processeur par le biais de liens dédiés [27]. En général, les co-processeurs sont capables d'effectuer des calculs sans communiquer constamment avec le processeur. Le processeur et le co-processeur peuvent travailler simultanément. Il existe beaucoup d'exemples dans la littérature qui s'appuient sur ce type de couplage.

Garp [101] est un exemple de ce type d'architecture utilisée pour accélérer des boucles ou des sous-programmes. Cette architecture intègre sur un même substrat un processeur MIPS-II et un co-processeur reconfigurable. Le processeur donne la main au co-processeur pour réaliser la fonction appelée. Le co-processeur peut accéder à la mémoire globale et à la mémoire du processeur. Le processeur est suspendu pendant la durée d'exécution des traitements par le co-processeur.

L'architecture Molen [194, 81] est composée d'un GPP (*General Purpose Processor*) qui contrôle l'exécution et la (re)configuration d'un co-processor reconfigurable, le spécialisant

pour des applications définies par le biais d'instructions spécialisées.

Un autre exemple est Spyder [112], basé sur une architecture de processeur de type VLIW [75], où un co-processeur et plusieurs unités de traitement peuvent s'exécuter en parallèle.

D'autres exemples sont REMARC [156], un co-processeur reconfigurable constitué d'une unité qui contrôle 64 blocs logiques programmables ou MorphoSys [181] composé d'un processeur MIPS et d'une unité de traitement reconfigurable munie d'une interface mémoire très performante, spécialement efficace pour la compression vidéo ou encore NAPA [179].

Unités de traitement attachées ou externes. L'unité de traitement peut être complètement détachée du processeur. L'unité est connectée au processeur via un bus d'interconnexion, et peut même se trouver sur une autre puce que le processeur. Ce type d'architecture sort du cadre de cette thèse. On peut tout de même citer des exemples comme Xputer [99], RAW [200], Splash [23], DECPerLe-1 [157].

Dans [148], le couplage du module spécialisé est défini en fonction de la caractéristique de la boucle à accélérer. Si le corps de la boucle est relativement petit et qu'il ne contient pas d'instructions de branchement, alors le module spécialisé est très fortement couplé au processeur. À l'inverse, si le corps de la boucle est complexe (beaucoup de branchements, ou nombre d'instructions important), alors le module est couplé comme co-processeur.

1.2.2.3 Technologie

Les ASIP peuvent se présenter sur différents supports technologiques. Nous décrivons les principales approches illustrées par des exemples académiques et industriels.

La technologie ASIC. Les processeurs sont câblés « en dur ». Les processeurs ASIC suivent le processus standard de conception d'ASIC et peuvent être facilement intégrés dans des SoC. Ils présentent l'avantage d'une flexibilité au niveau processus de fabrication. Une même description de processeur peut aboutir à différents circuits selon les optimisations appliquées par l'outil de synthèse : performance, énergie, surface. Plusieurs exemples commerciaux existent dans ce domaine. Tensilica propose le processeur Xtensa [93]. La société Virage Logic commercialise les processeurs ARC [199]. La figure 1.13 montre l'exemple d'un processeur ARC 750 configurable et extensible. Le processeur de base est un cœur de processeur ARC 700 32-bit (blocs blancs de la figure 1.13), et configurable par l'ajout d'une unité de gestion mémoire (MMU,*Memory Management Unit*), d'extensions DSP, ou encore d'extensions spécialisées.

La technologie FPGA. Les processeurs dits « soft-core » sont des IP (*Intellectual Property*) et sont mis en œuvre dans un FPGA à partir de la logique programmable existante. Chaque vendeur de FPGA propose son processeur soft-core. Altera propose le

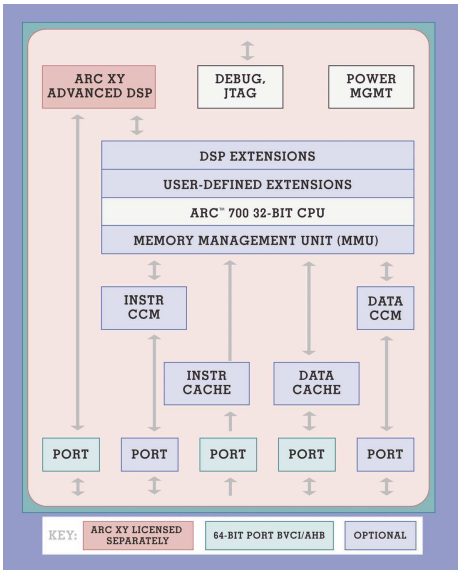


FIG. 1.13 – Exemple du processeur ARC 750D

processeur NIOSII largement présenté en exemple figure 1.5, et Xilinx met à disposition le MicroBlaze [211]. Les soft-core permettent de faire du prototypage rapide. Le nombre de soft-core n'est limité que par les capacités en ressources logiques du composant. Si le composant est reconfigurable dynamiquement, alors le soft-core l'est aussi.

D'autres processeurs soft-core sont indépendants du fabricant. Dans [67], les auteurs présentent le processeur CUSTARD, un processeur soft-core paramétrable mis en œuvre dans un FPGA. Le produit phare de la société Aeroflex Gaisler [13] est le modèle de processeur LEON, basé sur l'architecture d'un processeur SPARC V8. La figure 1.14 illustre un processeur LEON4. La configuration minimale, mise en évidence par les blocs de couleur blanche, inclut un pipeline à sept étages, une file de registres à quatre ports, un cache pour les instructions et un pour les données, une unité de gestion mémoire et un bus d'interface de type AMBA AHB. Le processeur LEON4 est hautement configurable à travers des modules spécialisés pour les multiplications, les divisions, les MAC, et les opérations arithmétiques en virgule flottante. Les instructions spécialisées sont mises en œuvre dans un co-processeur.

Les soft-core sont en général moins performants que leurs équivalents ASIC, consomment plus d'énergie et sont plus gourmands en surface. Ils peuvent être 3 à 5 fois moins rapides et jusqu'à 35 fois plus gros [50, 128].

Les processeurs « en dur » associés à de la logique reconfigurable. Ce type d'architecture propose un cœur de processeur « câblé » et de la logique reconfigurable (statiquement ou dynamiquement). Ces processeurs s'appuient sur une technologie *hybride* et associent la performance des cœurs de processeurs câblés avec la flexibilité de la logique reconfigurable. C'est le cas du processeur S6000 de Stretch [185] présenté en introduction de ce document (figure 6). Les fabricants de FPGA proposent également ce genre de

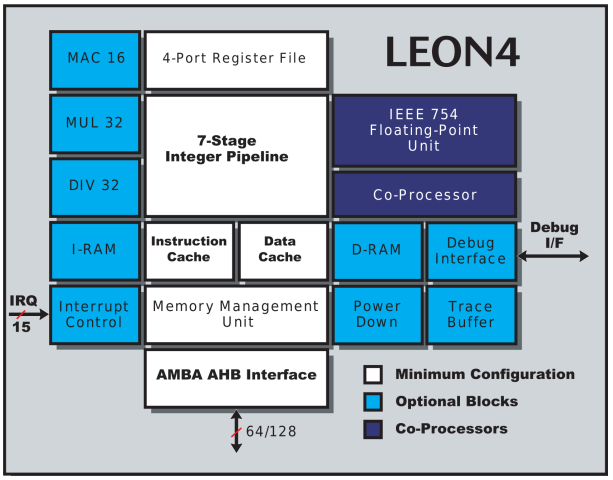


FIG. 1.14 – Processeur LEON4

ASIP	configurable	reconfigurable	extensible	mode	couplage	technologie
Xtensa LX3 [190]	✓		✓	VLIW	fort	dur
ARC750 [199]	✓		✓	RISC	fort	dur
S6000 [185]	✓	✓	✓	VLIW	lâche	hybride
NIOSII [17]	✓	✓	✓	RISC	fort	soft
MicroBlaze [211]	✓	✓	✓	RISC	lâche	soft
Leon [13]	✓	✓	✓	RISC	lâche	soft
Garp [101]	✓	✓	✓	RISC	lâche	soft

TAB. 1.1 – Tableau récapitulatif des ASIP

produit. Xilinx a intégré un PowerPC 405 dans sa famille de FPGA Virtex, et Altera produit Excalibur, qui intègre un processeur ARM922T.

1.2.2.4 Classification

Le tableau 1.1 résume quelques exemples de processeurs ASIP et leurs caractéristiques. Un ASIP est dit configurable si un des composants du cœur de processeur est configurable (file de registres, unité fonctionnelle, etc.). Un ASIP est extensible si il est possible d'étendre son jeu d'instructions par l'ajout d'instructions spécialisées. Dans notre contexte, un ASIP est reconfigurable si il s'appuie sur une architecture reconfigurable pour mettre en œuvre son cœur de processeur ou son extension. Le tableau 1.1 précise le mode d'exécution de l'ASIP : RISC ou VLIW. Le couplage du module spécialisé avec le cœur du processeur est fort lorsque le module est une unité fonctionnelle du processeur et lâche pour un co-processeur. La technologie utilisée pour mettre en œuvre le processeur peut être *dur* lorsque le processeur est câblé, *soft* pour la logique reconfigurable (FPGA), et *hybride* lorsque les deux sont mélangées.

1.2.3 Méthodologie

L'exemple de l'extension du jeu d'instructions du processeur NiosII a mis en évidence les étapes manuelles réalisées par le concepteur (description de l'architecture de l'instruction et adaptation du code applicatif). Pour des applications complexes, ce processus est fastidieux et sujet à erreur. C'est pourquoi il est important de disposer d'une méthodologie pour détecter automatiquement les portions de code à déporter en matériel, et des outils pour la génération automatique de l'architecture des instructions spécialisées et du code applicatif qui exploite ces instructions.

1.2.3.1 Granularité

La spécialisation peut avoir lieu à différents niveaux de granularité[2]. Nous distinguons deux niveaux : grain fin et gros grain. La spécialisation à grain fin opère au niveau instruction et met en œuvre matériellement un petit ensemble d'opérations [25, 28, 55]. La spécialisation à gros grain opère au niveau boucle ou procédure, et déporte une boucle ou une procédure entière sur du matériel dédié [31, 90, 101, 174, 203]. Le compilateur C2H d'Altera [17] et l'outil Cascade de CriticalBlue [65] comptent parmi les outils commerciaux qui proposent une solution automatisée de synthèse de co-processeur à partir d'une fonction écrite en langage C. La communication entre le processeur et le co-processeur ne fait pas appel au mécanisme d'extension de jeu d'instructions.

Chacune de ces granularités présente des avantages et des inconvénients. Si la spécialisation à gros grain permet d'obtenir d'importantes accélérations, sa flexibilité est limitée. En effet, il est peu probable que plusieurs applications possèdent exactement le même cœur de boucle ou la même procédure. À l'inverse, plus la spécialisation est fine et plus la probabilité de trouver une même suite d'opérations, dans un ensemble d'algorithmes, est grande. Par exemple, l'opération MAC (Multiply-ACcumulate) est très présente dans les applications du traitement du signal.

1.2.3.2 Automatisation

La figure 1.15 illustre une méthodologie générique appliquée lors d'un processus classique d'extension de jeux d'instructions. Cette méthodologie se distingue de la méthodologie de conception complète au niveau de l'étape de génération du jeu d'instructions. L'extension d'un jeu d'instructions consiste à générer un ensemble d'instructions spécialisées candidates, puis à sélectionner un sous-ensemble de ces instructions. Comme pour la conception complète, le processus peut être itératif, et le jeu d'instructions est progressivement enrichi de nouvelles instructions. Lorsque le processus n'est pas itératif, il s'agit de générer, en une seule passe, la meilleure architecture possible avec les contraintes de conception appliquées en entrée.

Dans [136], un flot de conception semi-automatisé est présenté, où l'identification d'instructions spécialisées est intentionnellement réalisée à travers un processus interactif avec

[2] à ne pas confondre avec la granularité d'une architecture, même si les deux sont intimement liées

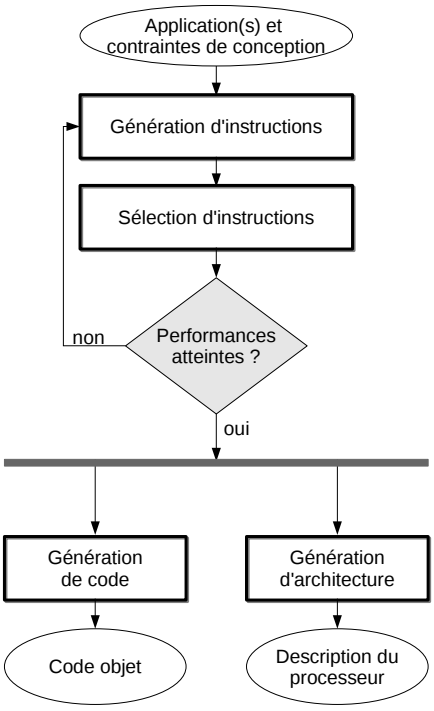


FIG. 1.15 – Méthodologie d'extension de jeu d'instructions

le concepteur.

Dans [218], le flot de conception est itératif. L'ASIP subit une première passe de génération d'extensions de jeu d'instructions. Il est ensuite généré et simulé. Si les performances sont atteintes, alors le processus est terminé ; sinon, une nouvelle passe de génération d'extensions est lancée.

1.2.4 Compilation

La compilation d'une application pour un processeur se déroule généralement en trois étapes [50] :

1. le *front-end*, il transforme le code source en une représentation intermédiaire ;
2. le *middle-end*, en charge des transformations de code et optimisations en tout genre ;
3. le *back-end*, effectue l'allocation de registres, l'ordonnancement de l'application, et la sélection de code.

Il est important de noter que les deux premières étapes sont indépendantes de l'architecture, c'est tout l'intérêt de l'utilisation de représentations intermédiaires. Seul le back-end doit disposer d'informations relatives à l'architecture.

Il existe deux manières de générer le code objet pour un processeur : soit par le biais d'un compilateur dédié, soit par un compilateur multi-cibles ou reciblable (*retargetable compiler*) [134].

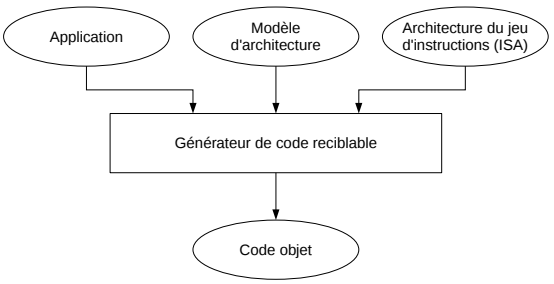


FIG. 1.16 – Compilateur reciblable

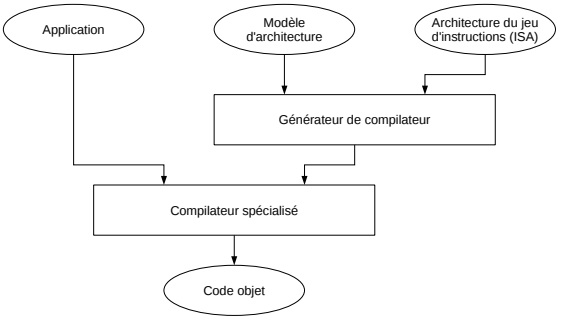


FIG. 1.17 – Compilateur spécifique

1.2.4.1 Compilateur dédié

Une première approche consiste à générer un compilateur propre à chaque processeur [137]. À partir du modèle d'architecture et de l'architecture du jeu d'instructions, un compilateur spécifique est généré (figure 1.17). Ce compilateur spécifique permet ensuite de générer le code objet correspondant à l'application. Cette approche est utilisée lors de la conception complète et illustrée dans la première partie du chapitre à travers l'exemple du langage LISA.

1.2.4.2 Compilateur reciblable

Les deux premières étapes du travail du compilateur (traduction de l'application dans une représentation intermédiaire, transformations et optimisations) sont indépendantes de l'architecture. Seule la partie back-end est dépendante de l'architecture. Un compilateur multi-cibles (*retargetable compiler*) [138] capitalise ces étapes communes et possède plusieurs back-ends pour être capable de générer du code pour plusieurs cibles. Pour cela, le compilateur s'appuie sur le modèle d'architecture et l'architecture du jeu d'instructions pour générer du code correspondant à l'application. La figure 1.16 illustre ce compilateur. L'exemple par excellence est le compilateur GCC (*GNU Compiler Collection*) [88], qui est capable de générer du code pour de nombreux processeurs, à partir de plusieurs langages de haut niveau. Un autre exemple est le compilateur CHESS [147] qui permet de générer du code pour des processeurs DSP.

1.2.4.3 Représentations intermédiaires

Les compilateurs s'appuient sur des représentations intermédiaires, indépendantes de l'architecture, sous forme de graphes pour appliquer les différentes transformations. Il est assez commun de représenter une application sous forme de graphe, où les nœuds représentent les opérations et les liens les dépendances de données et de contrôle. Il existe plusieurs types de représentations, nous proposons de les illustrer par le code de la figure 1.18.

```
int foo(int a, int b) {
    int result;
    if (a > 0){
        result = a + b;
    } else {
        result = b - a;
    }
    return result;
}
```

FIG. 1.18 – Fonction C quelconque

CDFG. Le *Control Data Flow Graph*, graphe flot de donnée et de contrôle, est la représentation traditionnelle qui représente de manière explicite les dépendances de données et les dépendances de contrôle [79]. Un CDFG est un CFG (*Control Flow graph*), un graphe flot de contrôle, où les liens représentent les dépendances de contrôle et les nœuds sont des *blocs de base* de l'application. Un bloc de base est un flot de données, un ensemble d'instructions qui traitent des données, jamais interrompu par une instruction de contrôle. Le flot de contrôle *entre* à la première instruction et *sort* à la dernière.

La figure 1.19 illustre le CDFG représentant le code de la figure 1.18. Le flot de données d'un bloc de base peut être des DFT (*Data-Flow Tree*) ou bien un DAG (*Directed Acyclic Graph*).

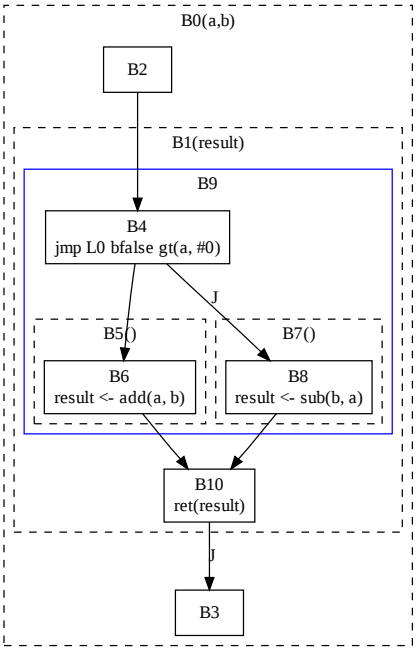


FIG. 1.19 – CDFG du code de la figure 1.18

DFT. Le *Data-Flow Tree* est un arbre flot de données. La représentation sous forme d'arbre est la plus élémentaire, où chaque nœud est une opération, et les fils sont les opérandes. La représentation sous forme d'arbre est limitée dans le cas où une même donnée est utilisée par plusieurs opérations. L'expression est alors découpée en plusieurs arbres, où un arbre produit la donnée, et les autres arbres utilisent cette donnée en entrée. La figure 1.20 illustre les deux arbres flots de données correspondant aux blocs `B6` et `B8` du CDFG de la figure 1.19.

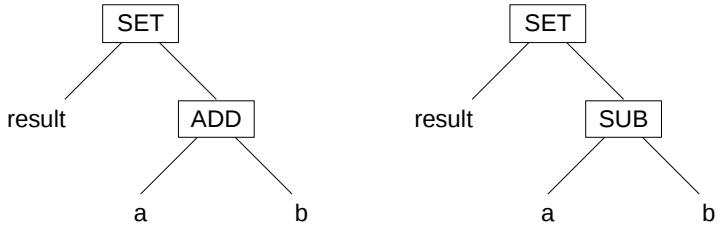


FIG. 1.20 – DFT du code de la figure 1.18

DAG. Le *Directed Acyclic Graph*, graphe acyclique dirigé, est un graphe flot de donnée uniquement [16]. Contrairement aux arbres, le DAG permet de regrouper au sein de la même représentation l'ensemble de l'expression, faisant ainsi apparaître le parallélisme au niveau instructions. La figure 1.21 montre les deux graphes flot de données de l'exemple. Les nœuds avec des points d'interrogation `?` symbolisent les entrées, et les points d'exclamation `!` symbolisent les sorties.

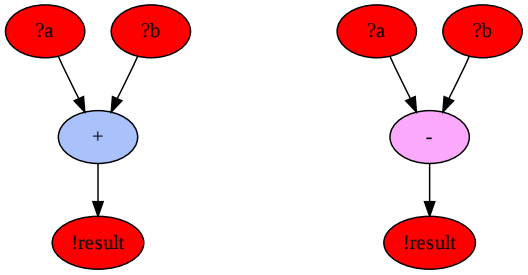


FIG. 1.21 – DAG du code de la figure 1.18

HCDG. Le *Hierarchical Conditionnal Dependency Graph*, ou GHDC (Graphe Hiérarchisé aux Dépendances Conditionnées), est un graphe hiérarchique qui permet de condenser au sein d'une même représentation le flot de données et le flot de contrôle [127]. La figure 1.22 montre le HCDG représentant le code de la figure 1.18. Dans cette représentation, chaque

nœud est contrôlé par une horloge. Les horloges sont un type spécial de nœuds et correspondent à des conditions booléennes qui *gardent* l'exécution des opérations et les affectations aux valeurs. Une garde est une horloge, qui vaut vrai si elle est activée et faux sinon. Les gardes sont représentées par les formes en rectangle dans la figure 1.22. La garde *root* est la *racine*, la garde principale dont toutes les gardes dépendent, et vaut toujours vrai. La valeur d'une garde est calculée d'après sa propre dépendance de contrôle et sa dépendance de données. Ainsi, sur l'exemple de la figure 1.22, la garde *H1* vaut vrai si sa garde vaut vrai (en l'occurrence, *root* vaut toujours vrai) *et* si $a > 0$. Les liens représentent les dépendances entre les nœuds. On peut en distinguer deux types : les dépendances de contrôle (en rouge sur la figure), et les dépendances de données (en bleu sur la figure).

Grâce à cette représentation HCDG, il est possible de connaître les gardes mutuellement exclusives, c'est-à-dire des gardes qui ont forcément des valeurs opposées. Dans l'exemple de la figure 1.22, si *H1* vaut vrai, alors *H2* vaut faux, et vice versa. Les gardes *H1* et *H2* sont mutuellement exclusives.

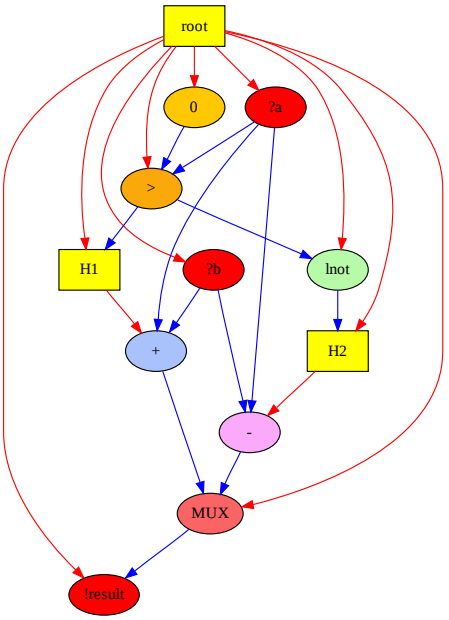


FIG. 1.22 – HCDG du code de la figure 1.18

1.3 Synthèse

La conception d'ASIP nécessite la construction de l'architecture du processeur et des outils pour le rendre exploitable. Nous avons présenté deux approches différentes pour la conception d'ASIP. La première approche, appelée conception complète, se base sur des langages de description d'architecture pour la génération simultanée de l'architecture et

des outils. La deuxième approche, appelée conception partielle, consiste à considérer un processeur existant avec sa suite d'outils, et à le spécialiser à travers l'ajout d'instructions spécialisées.

La conception complète de processeur propose de générer un ASIP et sa suite d'outils associés à partir d'un modèle du processeur. Cette technique permet de garantir la cohérence entre l'architecture et les outils. Cependant, le concepteur doit décrire une bonne partie du processeur manuellement. En particulier, il n'est pas possible de fusionner automatiquement des opérations pour créer une unité fonctionnelle.

La conception partielle de processeur permet de s'affranchir de nombreuses difficultés. L'approche permet de bénéficier des vérifications et optimisations apportées au processeur de base, et de profiter de la suite d'outils associés. La spécialisation d'un processeur existant est réalisée à travers le processus d'extension de jeu d'instructions.

Dans cette thèse, nous cherchons à tirer partie des outils et des processeurs existants et générons automatiquement des extensions de jeux d'instructions de processeurs.

2

L'extension de jeu d'instructions

L'extension d'un jeu d'instructions par le biais d'instructions spécialisées est devenue une technique courante pour réduire le temps d'exécution d'une application. Identifier les portions de code de calcul intensif et les mettre en œuvre sous forme de matériel dédié permet d'obtenir des gains significatifs en termes de performance. Outre un temps d'exécution réduit, un autre avantage est la réduction de la taille du code applicatif, car plusieurs instructions sont condensées au sein d'une seule, de manière similaire aux fonctions. Ce qui entraîne le troisième avantage : une baisse de la consommation d'énergie. Les instructions à exécuter sont moins nombreuses, ce qui conduit à une baisse d'activité (chargement, décodage, etc.) et résulte en une consommation d'énergie plus faible [20].

L'extension de jeu d'instructions suscite beaucoup d'intérêt et amène à se poser plusieurs questions. Comment peut-on étendre le jeu d'instructions d'un processeur ? Quelles nouvelles instructions ? Comment exploiter ces nouvelles instructions ? Comment générer l'architecture qui met en œuvre ces instructions ? Ce sujet a déjà fait l'objet de nombreuses études [82] et nous allons voir comment la littérature répond à ces questions.

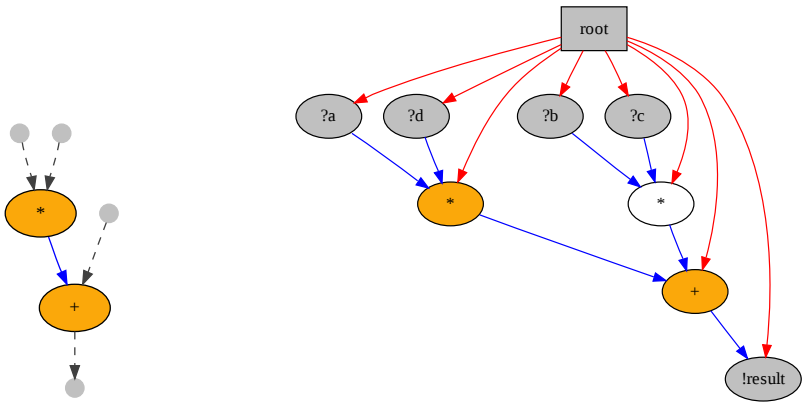
Le problème de l'extension de jeu d'instructions est un problème de partitionnement matériel/logiciel au niveau instruction puisqu'il s'agit de déterminer quelle instruction sera exécutée par le processeur et quel groupe d'instructions fera l'objet d'une instruction spécialisée. Si on considère que le partitionnement au niveau tâche est à gros grain, et au niveau fonctionnel à grain fin, alors le partitionnement au niveau instruction est à grain très fin.

Ce chapitre présente une vue d'ensemble du problème de l'extension de jeu d'instructions. Tout d'abord, la première section décrit les problèmes à résoudre. Ensuite, la section 2 présente les techniques applicables à la conception d'un jeu d'instructions étendu. Puis la section 3 décrit les techniques mises en œuvre pour résoudre ces problèmes.

2.1 Quels sont les problèmes à résoudre ?

L'extension de jeu d'instructions est un sujet bien spécifique qui s'appuie sur des concepts et techniques de domaines divers : architecture des processeurs, techniques de compilation, complexité algorithmique, théorie des graphes. En particulier l'utilisation de la théorie des graphes est l'approche dominante car elle fournit le cadre analytique idéal. Il est en effet assez classique de représenter une application sous forme de graphe, où les nœuds représentent les opérations et les liens les dépendances de données et de contrôle. Une instruction complexe, qui regroupe plusieurs opérations, correspond à un sous-graphe.

Le processus d'extension de jeu d'instructions consiste en deux étapes majeures : (1) génération d'instructions, et (2) sélection d'instructions. Ce processus s'appuie sur le concept de motif, ou *pattern*. Un motif représente un ensemble d'opérations candidates à une implémentation matérielle par le biais d'instructions spécialisées. Une occurrence, ou *match*, est une instance de motif dans un graphe d'application donné. La figure 2.1 montre un motif et une occurrence de ce motif dans un graphe d'application.



(a) motif constitué de deux opérations

(b) occurrence du motif dans un graphe

FIG. 2.1 – Exemple de motif et son occurrence

2.1.1 Motifs générés vs motifs prédéfinis

Un motif est un groupe d'opérations élémentaires. L'opération *MAC* (Multiply-Accumulate), très présente dans les applications du traitement du signal est un exemple typique de motif.

Les techniques de spécialisation de processeur s'appuient sur deux approches différentes : soit il existe une bibliothèque de motifs prédéfinis (comme par exemple le MAC), soit cette bibliothèque est générée à partir de l'application cible.

Dans le cas où une bibliothèque de motifs est utilisée, le processus se résume à l'étape de

sélection d'instructions. Il faut alors analyser l'application pour trouver quelles instructions de la bibliothèque sont nécessaires, c'est-à-dire trouver les occurrences des motifs dans le graphe d'application. C'est le problème de l'isomorphisme de sous-graphes. Plusieurs approches s'appuient sur l'existence d'une bibliothèque de motifs [57, 173, 142].

Cependant, afin de disposer d'instructions vraiment adaptées à l'application cible et gagner en performance, la plupart des approches construisent la bibliothèque en générant les instructions candidates [29, 44, 48, 86, 167, 217, 219].

Dans le cadre de nos travaux, c'est l'approche qui est retenue.

2.1.2 Génération d'instructions

La génération d'instructions, l'identification d'instructions, ou la découverte d'instructions désignent le même processus : la détermination des sous-graphes, à partir d'un graphe d'application cible, candidats potentiels à une mise en œuvre en matériel. Elle constitue la première étape du processus d'extension de jeu d'instructions. Pour des raisons liées à la prise en compte de l'architecture et de la technologie, cette étape est souvent sujette à des contraintes. Les contraintes les plus courantes concernent le nombre d'entrées et de sorties des motifs, la surface, et la consommation d'énergie. De plus, certaines instructions de l'application ne sont pas appropriées à une mise en œuvre matérielle. Par exemple, les instructions de contrôle sont en général exclues. Ces nœuds *indésirables* sont appelés *nœuds interdits*. De la même façon, de nombreux travaux excluent les instructions de chargement et de rangement (*load*,*store*) [30, 57, 186, 166].

2.1.2.1 Complexité de l'exploration

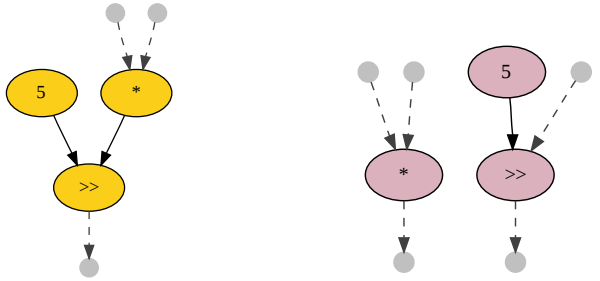
L'application étant représentée sous forme de graphe, la génération d'instructions s'apparente au problème de génération de sous-graphes ou motifs, où chaque sous-graphe est un candidat. Dans le cas général, chaque nœud du graphe peut faire partie ou non d'un candidat, ce qui entraîne un nombre de combinaisons en $O(2^n)$, où n est le nombre de nœuds du graphe.

L'énumération exhaustive de tous les sous-graphes est en temps exponentiel, et n'est donc pas applicable pour des exemples réalistes. Cette complexité rend le problème intraitable. Beaucoup de techniques ont été proposées pour résoudre ce problème, elles sont détaillées dans la section 2.3. Nous verrons qu'en imposant des contraintes sur les motifs, l'espace de recherche peut être grandement réduit. Une des premières contraintes est la contrainte de connexité : un motif peut être connexe ou non-connexe.

2.1.2.2 Motif connexe vs non-connexe

Un motif connexe est un motif tel que pour toute paire de nœuds, il existe un chemin entre les deux nœuds. Un motif non-connexe est un motif qui ne satisfait pas cette condition. Les motifs non-connexes font apparaître plus de parallélisme au niveau instructions, donc potentiellement de meilleurs gains. Malgré les possibilités de meilleures accélérations,

la plupart des auteurs se limitent à la génération de motifs connexes [25, 57, 62]. La figure 2.2 illustre deux motifs, le premier est connexe 2.2(a), et le second non-connexe 2.2(b).



(a) motif connexe (b) motif non-connexe

FIG. 2.2 – Motif connexe et non-connexe

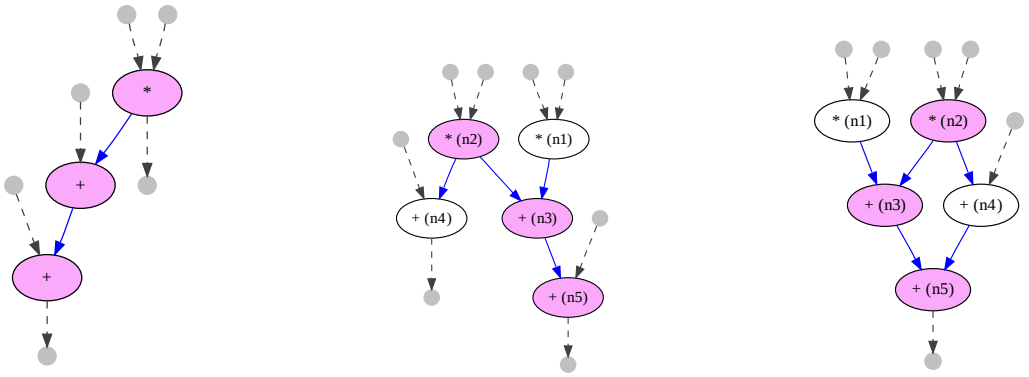
Dans [37], les auteurs ont observé que certains motifs générés par l'algorithme optimal proposé dans [30] sont des sous-graphes indépendants. Les auteurs en concluent qu'un algorithme d'identification d'instructions ne doit pas se limiter à identifier les sous-graphes connexes seulement. Plusieurs travaux traitent de la génération de motifs non-connexes [216, 217, 169, 84].

2.1.2.3 Convexité et instruction ordonnançable

Lorsqu'un ensemble d'opérations sont regroupées pour ne former qu'une instruction spécialisée exécutée en matériel, l'instruction doit être exécutable. Pour cela, il faut que toutes les données en entrée soient disponibles au lancement de l'instruction, et que toutes les sorties soient produites à la fin de l'exécution.

Formellement parlant, une occurrence (une instance d'un motif) est dite non-convexe lorsqu'il existe un chemin entre deux nœuds de l'occurrence qui passe par un nœud n'appartenant pas à cette occurrence. La figure 2.3(a) montre l'exemple d'un motif, et la figure 2.3(c) une occurrence non-convexe de ce motif : il existe un chemin entre le nœud $n2$ et le nœud $n5$ qui passe par le nœud $n4$.

La convexité permet de garantir un ordonnancement possible de l'instruction qui met en œuvre un motif. La convexité s'appuie sur la notion de contexte. En effet, un motif seul est toujours convexe, c'est l'occurrence du motif, à savoir son instance, c'est-à-dire là où il apparaît dans le graphe d'application, qui est non-convexe. Un motif est construit à partir d'une de ses occurrences, l'occurrence mère ou *template*. Il est donc possible de générer seulement les motifs dont l'occurrence mère est convexe et par raccourci, ceci est désigné comme étant la génération de motifs convexes. Dans la littérature, de nombreux travaux traitent de la génération de motifs convexes [30, 29, 43, 217].



(a) motif

(b) occurrence convexe du motif

(c) occurrence non-convexe du motif

FIG. 2.3 – Motif : occurrence convexe et non-convexe

2.1.2.4 Instructions à sortie unique vs sortie multiple

Selon l'architecture ciblée, des limitations quant au nombre maximum d'entrées et de sorties peuvent être imposées sur les instructions à générer. Les principales raisons de ces limitations proviennent de l'encodage d'instruction, ou du nombre de ports en lecture et en écriture de la file de registres. Les instructions à générer sont essentiellement de deux types, selon leur nombre de sorties : *Multiple Input Single Output* (MISO) et *Multiple Input Multiple Output* (MIMO).

MISO. Une instruction à entrée multiple sortie unique (*Multiple Input Single Output*) est adaptée à des architectures simples qui ne possèdent qu'un port d'écriture. Les conflits d'écriture sont alors évités [168]. La notion de MISO est introduite dans [16], où un MISO de taille maximum est appelé *MaxMISO*. Aucun MaxMISO ne peut être contenu dans un autre MISO.

La figure 2.4 montre un graphe d'application et ses différents sous-graphes à sortie unique (MISO). Les boites en trait plein mettent en évidence les MaxMISO.

La génération de MISO et de MaxMISO se différencie au niveau de la complexité du problème. Comme mentionné auparavant, dans un cas général, chaque nœud du graphe peut être inclus ou exclu d'une instruction candidate, ce qui conduit à un nombre exponentiel de candidats. Une des propriétés intéressantes des MaxMISO est que l'intersection des MaxMISO est vide. Si l'énumération des MISO est similaire au problème de l'énumération de sous-graphes, l'énumération des MaxMISO est de complexité linéaire [81].

MIMO. Les instructions à entrée multiple et sortie multiple (*Multiple Input Multiple Output*) permettent d'améliorer les performances de façon significative par rapport aux instructions à sortie unique, comme expliqué dans [20, Chap.7]. Il existe un nombre expo-

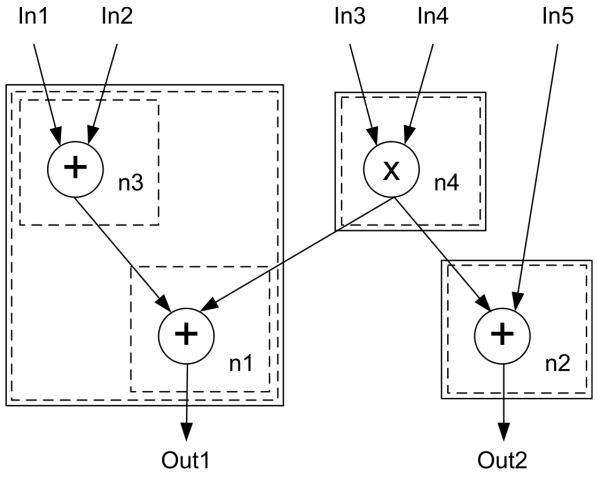


FIG. 2.4 – MISO d'un graphe d'application, exemple issu de [81]. Les boîtes en trait pointillé mettent en évidence les sous-graphes à sortie unique (MISO), les boîtes en trait plein les MaxMISO.

nentiel d'instructions MIMO. Beaucoup d'approches proposées dans la littérature tentent de résoudre le problème de façon optimale, ou bien par des heuristiques. Toutes les techniques sont décrites de façon détaillée dans la section 2.3.

La génération de motifs (c.-à-d. génération d'instructions) est une étape cruciale du processus d'extension de jeu d'instructions. La qualité des motifs générés va grandement influer sur la qualité du jeu d'instructions étendu. Un motif peut être connexe ou non-connexe, à sortie unique ou sortie multiple, et chaque instance de ce motif (occurrence dans un graphe cible) doit être convexe pour assurer un ordonnancement possible du graphe cible. Mais un motif est aussi sujet à d'autres contraintes, notamment architecturales et technologiques. Ces contraintes permettent de réduire énormément l'espace des solutions.

2.1.3 Limites architecturales et technologiques

Les limites architecturales et technologiques présentées concernent le nombre d'entrées, le nombre de sorties, la surface, le chemin critique, et la consommation d'énergie.

2.1.3.1 Nombre d'entrées

La limitation sur le nombre d'entrées est une conséquence des contraintes d'encodage de l'instruction mais aussi du nombre de ports en lecture disponibles. Dans le cas général, une instruction du processeur dispose de deux opérandes d'entrée.

Une solution qui vient immédiatement à l'esprit est d'augmenter le nombre de ports en lecture. Le problème est que les temps d'accès sont plus élevés et la consommation d'énergie augmente cubiquement avec le nombre de ports [177]. Une autre solution, mise en œuvre

dans Chimaera [212], est la réplication de la file de registres. Le principal inconvénient est qu'il en résulte un important gaspillage de ressource et d'énergie.

La solution proposée dans [117] est d'exploiter le *forwarding* pour ajouter des opérandes en entrée et disposer de deux entrées en plus. Le *data forwarding*, transfert de données, encore appelé *register bypassing* (contournement de registre), est une technique qui permet de réduire les incohérences de données dans les pipelines de processeurs. Selon la profondeur du pipeline, une donnée produite par l'étage d'exécution nécessite plusieurs cycles avant d'être disponible dans la file de registres. Le *data forwarding* consiste à ajouter un chemin de données direct en entrée de l'étage d'exécution pour pouvoir utiliser une donnée sans attendre son écriture dans la file de registres.

La solution décrite dans [61] est d'introduire des registres *fantômes* (*shadow registers*). Le nombre de registres fantômes est généralement faible (3 au maximum) ce qui permet de disposer de seulement 3 entrées en plus. Pozzi [170] exploite le *pipelining* pour améliorer la bande passante des données. Mais l'algorithme proposé a une telle complexité qu'il ne passe pas à l'échelle. Des améliorations de cette technique sont apportées dans [196, 198].

Une solution architecturale proposée par Xilinx pour son Microblaze est l'utilisation de *liens de données dédiés*. La logique spécialisée n'est pas autorisée à accéder directement à la file de registres. Une interface spéciale appelée *LocalLink* est fournie pour permettre un accès rapide entre le Microblaze et la logique spécialisée.

Dans [166], les auteurs réalisent un *timeshapping of patterns* qui consiste à ajouter le nombre de cycles de transfert de données nécessaires en fonction du nombre de registres disponibles.

2.1.3.2 Nombre de sorties

Pour des raisons similaires à celles évoquées pour le nombre d'entrées, le nombre de sorties est limité : taille de l'encodage de l'instruction, nombre de ports en écriture. Dans le cas général, une instruction du processeur dispose d'un seul opérande de sortie.

Il n'existe malheureusement pas beaucoup de solutions architecturales pour traiter ce problème. Seules les techniques de *pipelining* [170] et de *timeshapping* [166] sont applicables.

2.1.3.3 Surface

Pour chaque instruction spécialisée, il y a un surcoût en matériel associé. Ce surcoût peut être pris en compte dès l'étape de génération d'instructions. Afin de connaître ou de déduire le coût matériel total d'une instruction spécialisée, il est nécessaire de disposer d'estimations précises sur le coût matériel d'une ou d'un ensemble d'opérations. Une approche simple est de récupérer les résultats de la synthèse logique pour connaître le coût en surface [85, 218]. Des méthodes pour une estimation rapide du coût en surface des instructions spécialisées sont présentées dans [131, 69].

La prise en compte de la surface est généralement considérée lors de l'étape de sélection

d'instructions, quand il s'agit de minimiser le temps d'exécution sous contrainte matérielle [81].

2.1.3.4 Chemin critique

Dans le cas d'une extension fortement couplée au chemin de données du processeur, le chemin critique d'une instruction spécialisée devient extrêmement important. En effet, le chemin critique définit la fréquence d'horloge de toute l'architecture. Abaisser la fréquence d'horloge globale du processeur pour qu'elle corresponde au chemin critique d'une instruction spécialisée conduirait à des résultats désastreux. Pénaliser tout un ensemble irait à contresens de l'objectif visé. L'instruction spécialisée doit donc s'adapter à la fréquence d'horloge du processeur (et non l'inverse !). Le gain apporté par une instruction spécialisée qui s'exécute en un cycle étant limitée, plusieurs techniques peuvent être mises en place pour l'exécution d'instructions multi-cycles : pipeline, horloges multiples.

À notre connaissance, seule la technique basée sur la programmation linéaire proposée dans [27] prend en compte le chemin critique pendant la génération d'instructions. Nous verrons au chapitre 3 comment nous avons intégré cette contrainte dans notre génération d'instructions.

2.1.3.5 Puissance et consommation d'énergie

La prise en compte de la consommation d'énergie est similaire à celle du coût en surface. Les outils de génération et de sélection d'instructions ont besoin d'estimations précises de la consommation. Malgré les nombreux efforts apportés dans ce domaine ces dernières années, les outils de synthèse ne sont pas capables de donner des résultats aussi précis pour la consommation d'énergie que pour le coût matériel d'une instruction.

Il existe beaucoup d'études qui montrent de manière empirique que déporter en matériel une partie de l'application permet de réduire de façon globale la consommation d'énergie [36, 54, 72]. Quelques études qui visent l'extension de jeu d'instructions dans le but de réduire de façon optimale la consommation d'énergie existent [40].

2.1.4 Sélection d'instructions

La sélection d'instructions est une étape majeure de la spécialisation de processeur. Cette étape consiste à conserver parmi un ensemble d'instructions candidates (générées ou non), celles qui optimisent une fonction de coût pour une application donnée. Dans le cadre de l'extension de jeu d'instructions, certaines instructions font partie du jeu d'instructions de base du processeur, d'autres sont candidates à une mise en œuvre matérielle dans l'extension. Dans ce contexte, la sélection d'instructions devient un problème de partitionnement logiciel/matériel au niveau instruction. Les applications et les instructions étant représentées sous forme de graphe, la sélection d'instructions s'apparente au problème de la *couverture de sommets* qui est NP-complet [64].

La fonction de coût à optimiser diffère selon l'objectif : minimisation du nombre de motifs utilisés ou du nombre d'occurrences [56], minimisation du nombre de nœuds non-couverts [141], minimisation de la longueur du chemin critique [145], minimisation du chemin critique par nombre d'occurrences de certains motifs [48, 173], partage de ressources [109, 159], par occurrence de nœuds spécifiques [121, 186], réutilisation de ressources par similarité des motifs [90], minimisation du temps d'exécution [62, 85].

2.1.5 Isomorphisme de graphes et de sous-graphes

Lorsqu'une nouvelle instruction spécialisée est générée, il convient de savoir si cette instruction fait déjà partie de la bibliothèque. Pour cela, nous utilisons l'isomorphisme de graphes. Nous avons aussi besoin de savoir où cette instruction apparaît dans le graphe d'application. Pour cela, nous utilisons l'isomorphisme de sous-graphes.

Certains algorithmes sont capables de résoudre seulement le problème d'isomorphisme de graphes, d'autres seulement le problème d'isomorphisme de sous-graphes. Des outils capables de résoudre les deux problèmes existent aussi.

2.1.5.1 Isomorphisme de graphes

De manière générale, le problème de l'isomorphisme de deux graphes consiste à prouver que les deux graphes sont isomorphes, c'est-à-dire qu'ils sont identiques [183]. Dans notre cas, il s'agit de déterminer si oui ou non, deux graphes donnés possèdent le même nombre de nœuds, le même nombre de liens, la même structure, tout en vérifiant les étiquettes des nœuds et des liens. C'est un isomorphisme de graphes particulier où les étiquettes des nœuds doit également correspondre. Par la suite, nous utiliserons le terme d'isomorphisme de graphes pour désigner l'isomorphisme de graphes où les étiquettes sont équivalentes. L'exemple de la figure 2.5 illustre deux graphes isomorphes.

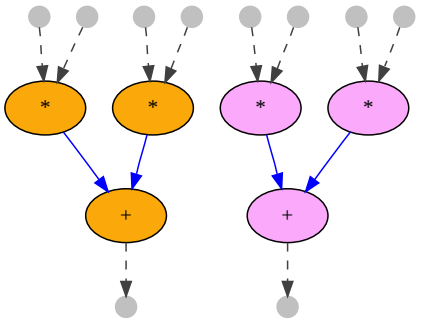


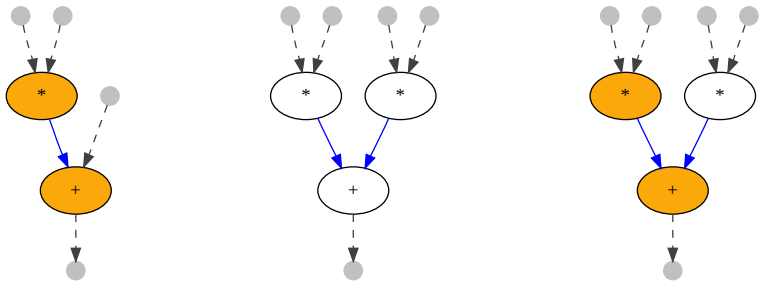
Fig. 2.5 – Exemple de deux graphes isomorphes

La complexité théorique de l'isomorphisme de graphes n'est pas parfaitement déterminée [183]. Il n'en reste pas moins un problème NP-difficile. *The Nauty package* [149]

permet de résoudre le problème d'isomorphisme de graphes. Une contrainte globale pour l'isomorphisme de graphes est introduite dans [184] pour une résolution du problème par la programmation par contraintes.

2.1.5.2 Isomorphisme de sous-graphes

L'isomorphisme de sous-graphes consiste à trouver si un graphe sujet *apparaît* dans un graphe cible, autrement dit, si le graphe sujet est un sous-graphe du graphe cible. Soit le graphe sujet de la figure 2.6(a), et le graphe cible de la figure 2.6(b), la figure 2.6(c) montre la correspondance entre le graphe sujet et un sous-graphe du graphe cible. Dans le contexte de l'extension de jeu d'instructions, le graphe sujet est un motif, le graphe cible est le graphe d'application. La figure 2.6(c) montre une *occurrence* du motif, c'est-à-dire une instance du motif, dans le graphe d'application.



(a) Graphe sujet (b) Graphe cible (c) Isomorphisme de sous-graphe

FIG. 2.6 – Illustration de l'isomorphisme de sous-graphes

Si la complexité de l'isomorphisme de graphes n'est pas connu, en revanche, l'isomorphisme de sous-graphes est un problème NP-complet [87]. Ullman a été le premier à proposer un algorithme pour résoudre le problème d'isomorphisme de sous-graphes [192]. Puis Larossa et Valiente [132] ont étudié le problème et des méthodes pour le résoudre par la programmation par contraintes.

2.1.5.3 Isomorphisme de graphes et de sous-graphes

Il existe des outils capables de résoudre les deux problèmes d'isomorphisme de graphes et de sous-graphes. L'algorithme `VF2` est présenté dans [162]. L'algorithme est décrit par le moyen de *State Space Representation (SSR)*. Dans chaque état, une solution de correspondance partielle est maintenue et seuls les états cohérents sont conservés. Ces états sont générés en utilisant des *règles de faisabilité* qui suppriment des paires de nœuds qui ne peuvent pas être isomorphes.

L'isomorphisme de graphes et de sous-graphes peut être résolu par la programmation

par contraintes. JaCoP (Java Constraint Programming solver) [114] est un solveur de contraintes écrit en Java. Ce solveur possède une contrainte `GraphMatch` spécialement conçue pour l'isomorphisme de graphes et de sous-graphes. Bien que la version 2 soit libre d'accès, une version 2 étendue, possédant les possibilités d'isomorphisme de (sous-)graphes, est protégée. Dans le cadre de nos travaux, nous avons le privilège de pouvoir utiliser ces contraintes sur l'isomorphisme de (sous-)graphes.

L'isomorphisme de graphes et de sous-graphes permet de vérifier si deux graphes sont structurellement équivalents. Il peut donc arriver que deux motifs fonctionnellement équivalents soient considérés comme différents. Dans [26], les auteurs présentent une méthode qui permet de vérifier si deux motifs sont fonctionnellement équivalents et non structurellement seulement.

2.1.6 Exploitation des nouvelles instructions

À ce stade du processus d'extension, nous avons trouvé tous les candidats à une mise en œuvre matérielle satisfaisant les contraintes imposées (nombre d'entrées/sorties, surface, chemin critique, consommation) et nous avons sélectionné, parmi ces candidats, ceux qui offrent les meilleurs résultats. Il s'agit maintenant d'exploiter au mieux le jeu d'instructions complet (base + extension) pour une application donnée. Cette phase est la sélection de code. Il est possible de modifier la phase de sélection de code du compilateur du processeur étendu pour prendre en compte les nouvelles instructions.

2.1.6.1 Changement du compilateur

La sélection de code peut être résolue de manière optimale [14] par la programmation dynamique [161] ou par des générateurs de type *BURG* (*Bottom-Up Rewrite Generator*) [77] lorsque les représentations sont sous forme d'arbre binaire. Cependant, l'utilisation de ces techniques pour les processeurs étendus peut conduire à la génération de code de mauvaise qualité [139]. La représentation la plus appropriée est la représentation plus générale sous forme de graphe flot de données, qui permet de mettre en évidence le parallélisme au niveau instruction. Ainsi, au lieu de modifier le compilateur pour la prise en compte des nouvelles instructions, une autre approche est d'agir directement sur le code source de l'application.

2.1.6.2 Changement du code source

Il est possible d'appeler directement une instruction spécialisée dans le code source de l'application. Par exemple, une instruction spécialisée pour un NiosII est appelée par le biais de *Macros*. Ces *Macros* sont remplacées par l'instruction spécialisée correspondante dans le code binaire généré par le compilateur. Il est également possible de modifier le code source pour y insérer directement les instructions assembleur qui font appel aux instructions spécialisées.

2.2 Quand les résoudre ?

Les étapes de génération d'instructions, de sélection d'instructions, et de génération de code peuvent être réalisées pendant la phase de conception du processeur ou bien lorsque celui-ci est déjà en service.

2.2.1 À la conception

L'extension du jeu d'instructions d'un processeur peut se faire au moment de sa conception (*at design time*). Mais avant de dérouler le processus de génération d'instructions, sélection d'instructions, modification du code, quelques techniques sont applicables. Nous évoquons ici deux techniques largement répandues : le profilage et la transformation de code.

2.2.1.1 Profilage

Le but du profilage est de détecter les portions du code les plus fréquemment exécutées. Ces portions sont appelées *points chauds* (*hot spots*) ou partie critique. Une règle énoncée dans [163] stipule que seulement 10% du code d'un programme est responsable de 90% du temps d'exécution. Il est donc important d'accélérer ces 10% du code. Beaucoup d'approches s'appuient sur le profilage pour détecter les « points chauds » d'une application [46, 136, 120, 195].

On ne parlera ici que de deux types de profilage, un premier où le code source est modifié, appelé profilage intrusif ; un deuxième où le code source reste inchangé, appelé le profilage non-intrusif.

Le profilage intrusif est le profilage par instrumentation. Le code source est modifié pour y ajouter des compteurs. On peut par exemple mettre un compteur d'appel pour une fonction particulière, on connaîtra alors le nombre d'appels de la fonction pendant l'exécution du programme. L'ajout d'instructions dans le code source présente l'inconvénient de parasiter le code original et de perturber les mesures, les rendant inexactes. Pour disposer de mesures exactes, il existe le profilage non-intrusif.

Le profilage non-intrusif est le profilage par émulation. Le programme est simulé par un simulateur de jeu d'instructions. Le code source n'a pas besoin d'être modifié. Le principal inconvénient est la lenteur du simulateur, un programme simulé s'exécutant moins vite qu'un programme réel.

2.2.1.2 Transformation de code

La transformation de code est une étape classique de la compilation. Il existe de nombreuses transformations qui permettent à un compilateur d'optimiser le code [15]. Nous citerons ici les transformations intéressantes dans le contexte d'extension de jeu d'instructions.

Évaluation et propagation de constantes. L'évaluation et la propagation de constantes est une passe classique de la compilation [15]. Il s'agit tout simplement d'évaluer statiquement, lorsque cela est possible, le résultat d'une opération impliquant des constantes. Si un résultat est évalué, on peut propager sa valeur jusqu'à ce qu'aucune autre évaluation statique ne soit possible.

Transformations algébriques. Les travaux présentés dans [164] s'appuient sur les propriétés des opérateurs (associativité, distributivité) pour appliquer des transformations algébriques qui permettent de faire apparaître des motifs qui correspondent à des instructions spécialisées existantes. Les transformations algébriques incluent les simplifications, comme l'élimination des éléments neutres (par exemple, l'ajout d'un 0).

Nous avons vu que les instructions de contrôle ne se prêtent pas à une mise en œuvre matérielle. Nous présentons ici deux techniques qui permettent de *casser* le flot de contrôle : le déroulage de boucle et la conversion de *if*.

Le déroulage de boucle. Le déroulage de boucle (total ou partiel) consiste à répliquer le corps de la boucle autant de fois que nécessaire. Le nombre d'itérations doit donc être connu. Cette transformation permet de faire apparaître le graphe flot de données qui correspond à la boucle déroulée. À partir de ce graphe, il est possible de générer des instructions spécialisées de taille plus grande et d'obtenir des meilleures performances [27].

If-conversion. Une passe de transformation *If-conversion* permet également d'obtenir des meilleures performances en exploitant le parallélisme inhérent d'un bloc `if()...else()`. La sélection peut se faire par un multiplexeur dans l'extension [41]. Les deux branches sont systématiquement exécutées, et la bonne sortie est sélectionnée suivant la condition. La transformation est appliquée lorsque les deux branches sont équilibrées et qu'elles ne contiennent que peu d'instructions.

2.2.2 À la volée

Le processus de génération d'instructions, sélection d'instructions, modification du code peut être fait à la volée (*at runtime*). L'extension de jeu d'instructions d'un processeur lorsque celui-ci est déjà en service sous-entend une architecture reconfigurable dynamiquement. Cette approche sort du cadre de cette thèse et nous invitons le lecteur à lire l'étude sur les processeurs embarqués extensibles adaptatifs à la volée [110].

2.3 Comment les résoudre ?

Nous avons vu que l'extension de jeu d'instructions soulève beaucoup de problèmes. La génération d'instructions se résoud en temps exponentiel et la sélection d'instructions est un problème NP-complet.

Aucun algorithme en temps polynomial n'a encore été découvert pour un problème NP-complet. La plupart des théoriciens de l'informatique pensent que les problèmes NP-complets sont intraitables. En effet, si l'un des problèmes NP-complets pouvait être résolu en temps polynomial, alors tous les problèmes NP-complets auraient un algorithme permettant de les résoudre en temps polynomial [64].

Dans [53], une méthode exhaustive pour l'énumération des instructions est présentée, mais cette solution ne passe pas à l'échelle. Pour traiter des problèmes de taille plus grande, il faut trouver autre chose. La sélection d'instructions est un problème d'optimisation. Il existe plusieurs techniques pour résoudre ces problèmes. Nous commençons cette revue par les métaheuristiques : recherche avec tabous, recuit simulé, algorithmes génétiques et algorithmes de colonies de fourmis. Puis, nous verrons la programmation dynamique, les algorithmes gloutons et les algorithmes dédiés. Enfin, nous décrirons les concepts de programmation linéaire par nombre entier et de programmation par contraintes.

2.3.1 Recherche Tabou

2.3.1.1 Définition

La recherche tabou (*tabu search*) est une métaheuristique qui guide une heuristique de recherche locale pour explorer l'espace des solutions au-delà d'un optimum local [92]. L'heuristique locale part d'une position donnée et explore son voisinage pour *bouger* vers une position qui optimise la fonction de coût. Lorsqu'aucune des positions voisines ne conduit à un meilleur résultat, l'opération de mouvement sort de l'optimum local en empirant la solution. Pour éviter de retomber dans cet optimum local à l'étape suivante, les positions explorées sont marquées comme *tabou*, grâce à un effet mémoire. Les mémoires utilisées peuvent être de différents types (mémoire à court terme, mémoire à long terme, etc), caractérisées par quatre dimensions : la récence, la fréquence, la qualité, et l'influence. Ces mémoires peuvent même évoluer pendant la résolution. Une des composantes principales de la recherche tabou est donc l'utilisation de *mémoires adaptatives* qui donnent un comportement flexible à la recherche. Les stratégies basées sur la mémoire sont une marque de fabrique de la recherche tabou.

2.3.1.2 Utilisation

La recherche tabou n'a pas été utilisée à proprement parler pour un des problèmes spécifiques liés à l'extension de jeux d'instructions (identification d'instructions, sélection d'instructions). Mais cette recherche apparaît dans plusieurs articles qui traitent du partitionnement logiciel/matériel, soit au niveau système dans [70], soit au niveau fonctionnel dans [202].

2.3.2 Recuit simulé

Le recuit simulé est une métaheuristique testée dans le contexte d'extension de jeux d'instructions.

2.3.2.1 Définition

Le recuit simulé est une métaheuristique inspirée d'un processus utilisé en métallurgie. Ce processus alterne des cycles de refroidissement lent et de réchauffage (recuit) pour minimiser l'énergie du matériau. Le recuit simulé s'appuie donc sur un paramètre qui incarne la *température*. Par analogie avec le processus physique, l'algorithme de recuit simulé modifie une solution donnée pour en obtenir une autre. Si la solution obtenue est meilleure que la solution initiale, alors elle tend vers l'optimum local. Au contraire, une solution moins bonne permet d'explorer plus largement l'espace des solutions et d'éviter de se retrouver coincé dans un optimum local. Il peut être intéressant de conserver une solution moins bonne. Ceci est fait par un calcul de probabilité dépendant de la différence du coût et de la température.

Cette méthode a été décrite par S. Kirkpatrick, C.D. Gelatt et M.P. Vecchi en 1983 [126], et indépendamment par V. Cerny en 1985 [51]. Le recuit simulé est une adaptation de l'algorithme de Metropolis-Harding, une méthode de Monte-Carlo qui permet de décrire un système thermo-dynamique.

2.3.2.2 Utilisation

Le recuit simulé est utilisé par [122] pour résoudre le problème de sélection d'instructions. À chaque étape de l'algorithme, une des trois actions est aléatoirement choisie : *ajout*, *suppression*, ou *échange*. L'action *ajout* consiste à ajouter un candidat dans l'ensemble des candidats sélectionnés. L'action *suppression* retire un des candidats sélectionnés, choisi aléatoirement. L'action *échange* permute un des candidats sélectionnés (aléatoirement) avec un des candidats non sélectionnés. Le coût de la solution obtenue est comparé avec le coût de la solution initiale. Si l'action résulte en un meilleur résultat, elle est conservée ; si l'action empire la solution, elle est acceptée selon une règle de probabilité donnée. La probabilité d'accepter une solution moins bonne diminue avec la température. À la fin, seules les meilleures solutions sont conservées, jusqu'à un optimum.

Dans [108], le recuit simulé est utilisé pour résoudre le problème de sélection d'instructions pour un jeu d'instructions complet, modélisé sous forme d'un problème d'ordonnancement modifié.

2.3.3 Algorithme génétique

2.3.3.1 Définition

Les algorithmes génétiques appartiennent à la famille des algorithmes évolutionnistes (un sous-ensemble des métaheuristiques). Les algorithmes génétiques sont des algorithmes stochastiques qui s'appuient sur des méthodes de recherche adaptatives pour résoudre des problèmes d'optimisation. Ces algorithmes s'appuient sur la théorie de l'évolution et le principe de sélection naturelle de Darwin et les appliquent à une population de solutions possibles au problème. L'algorithme commence avec une population de base, générée

aléatoirement, parmi l'ensemble des solutions. Puis cette population subit de manière cyclique les étapes d'évaluation, de sélection, de croisement et mutation. L'algorithme s'arrête après un nombre donné d'itérations.

2.3.3.2 Utilisation

Dans [171], un algorithme génétique est présenté pour le partitionnement logiciel/-matériel au niveau tâche avec une fonction à objectifs multiples d'optimisation du temps d'exécution et de la consommation d'énergie.

Dans [103], un algorithme génétique est utilisé pour le problème de partitionnement matériel/logiciel au niveau fonctionnel.

Dans [38], un algorithme génétique est utilisé pour la génération d'instructions. Ces instructions spécialisées peuvent contenir des accès mémoire en lecture seule (dont le temps d'accès est fixe). Les résultats intermédiaires des instructions spécialisées sont réutilisés dans l'extension à travers des registres.

2.3.4 Algorithme de colonies de fourmis

2.3.4.1 Définition

Les algorithmes de colonies de fourmis, ou *Ant Colony Optimisation*, sont des algorithmes inspirés du comportement des fourmis et constituent une famille de métaheuristique d'optimisation. Bien qu'une fourmi seule ait des capacités cognitives limitées, un groupe de fourmis est capable de trouver le chemin le plus court entre une source de nourriture et leur colonie. Lorsqu'une fourmi trouve de la nourriture, elle rentre à la colonie en laissant derrière elle des phéromones. Les fourmis sont attirées par les phéromones et auront tendance à emprunter le même chemin, renforçant encore plus ce chemin. Au contraire, un chemin peu emprunté perd petit à petit ses phéromones et finit par disparaître.

2.3.4.2 Utilisation

Un algorithme de colonies de fourmis est utilisé par [201] pour résoudre le problème de partitionnement matériel/logiciel au niveau tâche ou au niveau fonctionnel.

Les métaheuristiques permettent de résoudre des problèmes d'optimisation et nous avons vu celles qui ont été appliquées dans le contexte d'extension de jeu d'instructions. Une autre façon de résoudre ces problèmes est l'utilisation de la programmation dynamique.

2.3.5 Programmation dynamique

2.3.5.1 Définition

La programmation dynamique résout les problèmes en combinant les solutions de sous-problèmes. La programmation dynamique est intéressante surtout lorsque les sous-problèmes ne sont pas indépendants, c'est-à-dire lorsque des sous-problèmes ont en com-

mun des sous-sous-problèmes. Un algorithme de programmation dynamique résout chaque sous-sous-problème une seule fois, et mémorise la solution. Le recalcul de la solution n'est donc pas effectué à chaque fois que le sous-sous-problème est rencontré. La programmation dynamique est donc intéressante lorsque des sous-problèmes ont de nombreux sous-sous-problèmes en commun.

2.3.5.2 Utilisation

Dans [62], le gain d'un motif est calculé comme étant l'accélération qu'il apporte (c'est-à-dire le rapport entre son temps d'exécution en logiciel et son temps d'exécution en matériel) multiplié par son nombre d'occurrences. La sélection de motifs est ensuite réalisée sous contrainte de ressources, et formulée comme un *problème du sac à dos* (qui est aussi un problème NP-complet [119]). La programmation dynamique permet de résoudre de façon optimale le problème du sac à dos. La sélection de motifs sous contraintes de ressources est également formulée comme un problème du sac à dos et résolue par la programmation dynamique dans [60].

Dans [24], une approche basée sur la programmation dynamique est proposée pour générer des instructions avec un nombre arbitraire d'entrées et de sorties pour des processeurs VLIW. Cette approche s'applique sur des graphes (pas forcément des arbres), mais génère des instructions de petite taille.

La programmation dynamique peut être utilisée pour résoudre de manière optimale la sélection de code lorsque la représentation est sous forme d'arbres [161].

2.3.6 Algorithmes gloutons

2.3.6.1 Définition

Un algorithme glouton (*greedy algorithm*) est un algorithme basé sur une approche qui consiste à toujours faire le choix qui semble le meilleur sur le moment. Il fait un choix optimal localement dans l'espoir que ce choix mènera à la solution optimale globalement. À chaque étape du processus de résolution, le choix sélectionné est celui qui produit le meilleur gain immédiat tout en maintenant la faisabilité [160].

2.3.6.2 Utilisation

Les algorithmes gloutons ont été utilisés pour la sélection d'instructions. Dans [187] et [59], l'algorithme choisit l'instruction qui présente le meilleur taux $\frac{value}{cost}$, où $value$ est le nombre de cycles économisés, et $cost$ est le coût en surface de l'instruction. Une fois l'instruction sélectionnée, l'algorithme itère la liste des candidats restant en supprimant les nœuds du graphe d'application couverts par les instructions sélectionnées. Le processus de sélection est répété jusqu'à épuisement du budget en surface. Dans [122], le sélecteur parcourt tous les candidats non sélectionnés et, pour chaque candidat, calcule le coût total si ce candidat est sélectionné, où le coût total est le temps d'exécution total. Le candidat

qui produit la meilleure réduction de coût est conservé. Le processus de sélection est répété jusqu'à atteindre le nombre maximum d'instructions autorisées dans l'extension.

2.3.7 Heuristiques

Contrairement à une métaheuristique, qui est une méthode approximative générale pouvant s'appliquer à n'importe quel problème d'optimisation, une heuristique vise un problème bien précis.

2.3.7.1 Définition

Une heuristique est un algorithme qui fournit en temps polynomial une solution réalisable mais pas forcément optimale pour un problème d'optimisation NP-difficile. Une heuristique est un algorithme dédié pour un problème donné. La connaissance de la structure même du problème est donc primordiale et permet d'émettre des hypothèses sur les solutions à conserver.

2.3.7.2 Utilisation

Une technique pour résoudre le problème de la couverture de sommets (c.-à-d. sélection d'instructions) est d'appliquer une heuristique basée sur le *stable* de taille maximum (*maximum independent set*) dans un graphe de conflit [121, 95]. Cette heuristique est aussi mise en œuvre pour la génération d'instructions [166]. La complexité de l'énumération est en $O(2^{N_C})$, où N_C est le nombre de nœuds dans le graphe de conflit. Le nombre de nœuds dans un graphe de conflit étant inférieur au nombre de nœuds du graphe d'origine, le temps de résolution est diminué. Dans [197], l'algorithme de génération d'instructions se base sur le problème de la clique maximale, également connu pour être NP-complet [87]. Il est d'ailleurs démontré dans [87, p.54] que le problème de la clique maximale peut être transformé en problème de stable maximum, et vice versa.

Dans [215], l'idée est de donner des priorités aux instances des instructions spécialisées. Les auteurs ont testé trois fonctions de priorité différentes : une première qui favorise les instructions présentant le meilleur taux performance/coût, une deuxième qui donne priorité aux instructions les plus fréquemment exécutées, et une troisième où les instructions sont classées en fonction de l'accélération qu'elles apportent. La première fonction convient lorsque les contraintes en surface sont fortes. La deuxième vise à accélérer les portions de code coûteuses en temps d'exécution. La troisième maximise le gain en performance quand il n'y a pas de contraintes sur la surface.

Les travaux dans [58] s'appuient sur une fonction guide (*guide function*) qui écarte les chemins qui ne valent pas la peine d'être explorés. L'heuristique de Kernighan-Lin [124], à l'origine pour le partitionnement de circuit, est adaptée dans [37] pour le partitionnement au niveau instruction. Cette heuristique est aussi utilisée dans les travaux de [193] pour le partitionnement au niveau fonctionnel.

L'algorithme présenté dans [83] génère en temps linéaire des instructions à entrées multiples et sortie unique. Le graphe d'application est tout d'abord partitionné en *MaxMISO*. Chaque *MaxMISO* est ensuite décliné en *SubMaxMISO*, l'ensemble des *MaxMISO* générés à partir du *MaxMISO* d'origine auquel on a retiré un nœud. Si l'ensemble des *subMaxMISO* ne satisfait pas une propriété fournie, alors l'ensemble est rejeté et le processus est relancé en considérant un autre nœud, jusqu'à ce que la propriété soit satisfaite. Les *MISO* générés sont ensuite combinés selon d'autres heuristiques pour former des *MIMO*. Dans l'algorithme de groupement (*clustering*) *Nautilus* [81], les nœuds sont regroupés autour d'un nœud *graine* (*seed node*). Une autre heuristique appelée *recherche en spirale* (*spiral search*) [84], basée sur la vis sans fin d'Archimède, regroupe les nœuds selon leur niveau dans un graphe. Ces deux algorithmes, *Nautilus* et *spiral*, sont de complexité linéaire.

La complexité linéaire de ces algorithmes montre bien la force des heuristiques, qui sont capables de fournir une solution en temps polynomial. Malheureusement, ces algorithmes n'aboutissent pas à des solutions optimales. Pour obtenir une solution optimale (et le prouver), il existe les algorithmes par séparation et évaluation.

2.3.8 Séparation et évaluation

2.3.8.1 Définition

Un algorithme par séparation et évaluation, ou *branch and bound*, est un algorithme d'énumération systématique de toutes les solutions, où de larges ensembles de sous-solutions sont écartées grâce à une analyse de la quantité à optimiser. Dans un algorithme par séparation et évaluation, l'espace des solutions est représenté sous la forme d'un arbre de recherche, ou arbre de décision. La figure 2.7 montre un exemple d'arbre de recherche, où les valeurs de quatre variables valent soit 0, soit 1. L'algorithme parcourt en profondeur l'arbre de décision et garde en mémoire le coût de la meilleure solution trouvée jusque là. La méthode de séparation découpe le problème en sous-problèmes, résolus de manière optimale par une énumération exhaustive. La méthode d'évaluation permet d'évaluer rapidement le coût d'une solution en un certain point. Si le coût de la solution en ce point n'est pas intéressant, alors il n'est pas nécessaire de continuer la recherche puisque toutes les solutions dans cet ensemble seront mauvaises. Ainsi, un bon algorithme par séparation et évaluation écarte de grands ensembles de mauvaises solutions, grâce à des *techniques d'élagage*, et s'attarde à énumérer toutes les solutions potentiellement intéressantes. Pour cela, l'algorithme s'appuie sur le choix d'une heuristique qui guide le parcours de l'espace des solutions.

2.3.8.2 Utilisation

Dans [43], un algorithme par séparation et évaluation permet d'énumérer les sous-graphes sous contraintes d'entrées/sorties et de convexité en un temps polynomial. Les sous-graphes obtenus sont connexes. Des améliorations sont apportées dans [44], où les sous-graphes générés peuvent être non connexes. Les techniques d'élagage exploitent les

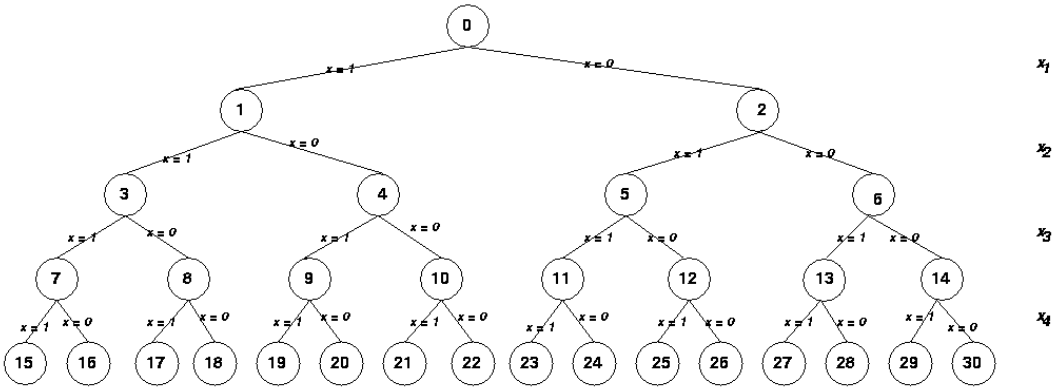


FIG. 2.7 – Exemple d'un arbre de recherche pour quatre variables pouvant prendre les valeurs 0 ou 1

contraintes d'entrées/sorties, de convexité, et de nœuds *interdits*. La preuve de la complexité en $O(n^{N_{in}+N_{out}})$ est apportée, où N_{in} est le nombre d'entrées, et N_{out} le nombre de sorties.

Dans [35], un algorithme par séparation et évaluation est utilisé pour résoudre le problème de partitionnement logiciel/matériel où des blocs de base entiers du CDFG sont sélectionnés pour être exécutés en matériel. L'algorithme cherche à minimiser la surface (en nombre de portes logiques) sous contrainte de temps d'exécution (nombre de cycles) et de consommation d'énergie, tout en prenant en compte le partage de ressources.

2.3.9 Programmation linéaire par nombres entiers

2.3.9.1 Définition

La programmation linéaire offre un cadre mathématique pour la résolution de problèmes d'optimisation, sous contraintes linéaires. Le problème à résoudre est exprimé à l'aide de variables, sujettes à des contraintes linéaires (d'égalité ou d'inégalité) et d'une fonction de coût à optimiser. Lorsque les valeurs des variables sont des nombres entiers, le problème est appelé problème de *programmation linéaire par nombres entiers*, ou *Integer Linear Programming* (ILP), et devient NP-complet [119]. Il existe de nombreux algorithmes capables de résoudre des problèmes de programmation linéaire, mais l'utilisation de *solveur* est très répandue.

2.3.9.2 Utilisation

Une formalisation du problème de sélection d'instructions est proposée dans [215], où le problème peut être résolu de manière optimale. Mais cette solution ne passe pas à l'échelle. Une autre proposition est faite dans [85] qui utilise un cas spécial de la programmation linéaire par entier, la programmation linéaire binaire, où les variables ne peuvent avoir que deux valeurs : 0 ou 1. À chaque nœud du graphe est associée une variable, qui vaut 0 si le

nœud est exécuté par le processeur, 1 si il est exécuté en matériel. L'objectif de la fonction de coût est de minimiser le temps d'exécution sous contrainte matérielle.

La programmation linéaire par nombres entiers peut être utilisée pour résoudre le problème d'identification d'instructions. Une formulation est proposée dans [28] où des contraintes sont imposées sur le nombre d'entrées, le nombre de sorties et la convexité. Chaque itération de l'algorithme tente de résoudre le problème et fournit un motif candidat. Les nœuds du graphe d'application couverts par ce candidat sont exclus de la recherche suivante. L'algorithme s'arrête lorsqu'il n'y a plus de solution.

Un autre exemple d'utilisation est présenté dans [135, 139], où la programmation linéaire résout le problème de sélection de code pour des processeurs possédant des instructions SIMD.

2.3.10 Programmation par contraintes

> « Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming : the user states the problem, the computer solves it. »
>
> Inaugural issue of the *Constraints Journal*, 1997 [78]
>
> Eugene C. Freuder

L'idée de la programmation par contraintes est bien celle-ci : l'utilisateur décrit le problème et l'ordinateur le résout.

2.3.10.1 Définition

De manière formelle, un *problème de satisfaction de contraintes* (ou *CSP* pour *Constraint Satisfaction Problem*) est défini par un triplet $\mathcal{S} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ où $\mathcal{V} = \{x_1, x_2, \ldots, x_n\}$ est un *ensemble de variables*, $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$ un ensemble de *domaines finis* (*FD* pour *Finite Domains*) de ces variables (c.-à-d. l'ensemble des valeurs possibles), et $\mathcal{C}$ un ensemble de *contraintes*. Un gros avantage par rapport à la programmation linéaire par entier est la possibilité d'utiliser des contraintes non-linéaires. La figure 2.8 montre l'exemple d'une contrainte non-linéaire, la contrainte cumulative, où le nombre de ressources disponibles est égal à 9 et le nombre de tâches égal 5, chaque tâche ayant un temps d'exécution et un nombre de ressources nécessaires. La contrainte cumulative permet de s'assurer qu'à chaque instant, le nombre de ressources utilisées ne dépasse pas le nombre de ressources disponibles.

Des techniques de consistance sont appliquées, elles permettent de réduire le domaine des variables et de garantir la validité de la solution.

2.3.10.2 Utilisation

Dans [205], la programmation par contraintes est utilisée pour résoudre les problèmes d'isomorphisme de graphes et de sous-graphes, où la contrainte `GraphMatch`, une contrainte

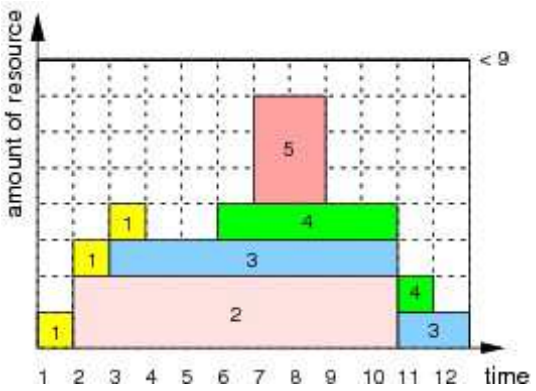


FIG. 2.8 – Exemple de la contrainte cumulative

sur la similarité de graphes qui n'est pas une contrainte « classique » de la programmation par contraintes [178], a été spécialement conçue pour le système $UPaK$. Cette contrainte est particulièrement utile pour trouver toutes les occurrences d'un motif dans un graphe (isomorphisme de sous-graphes), et pour savoir si un motif généré fait déjà partie de la liste de candidats (isomorphisme de graphes).

Les problèmes de sélection d'instructions, de *binding* et d'ordonnancement sont résolus simultanément, et pour la première fois, par la programmation par contraintes dans [206]. L'ordonnancement est effectué sous contrainte de temps ou contrainte de ressources. La méthode s'appuie sur la contrainte non-linéaire `Diff2`[1].

2.3.11 Comparaison des approches

L'extension de jeu d'instructions regorge de nombreux problèmes, pour la plupart intraitables, et qui nécessitent des temps de calcul exponentiels pour une résolution optimale. Beaucoup de techniques sont possibles pour résoudre un problème particulier mais jusqu'ici, il n'existe aucune méthode qui permette de résoudre tous les problèmes.

Parmi les nombreuses techniques présentées, nous proposons ici de faire un point quant à l'optimalité des approches. Nous faisons ensuite un bref comparatif entre les différentes métaheuristiques. Puis nous comparons la programmation linéaire et la programmation par contraintes. Enfin, nous justifions notre choix de la technique choisie dans le cadre de nos travaux.

2.3.11.1 Optimalité

Les métaheuristiques sont par définition des méthodes d'approximation. Non seulement elles sont incapables de trouver la solution optimale, mais en plus, c'est impossible de le prouver. Ainsi, concernant les algorithmes génétiques, il est impossible de s'assurer que la solution obtenue soit la meilleure. Il en est de même pour la recherche tabou et les

[1] La contrainte `Diff2` est décrite plus loin dans ce document p. 113

algorithmes de colonies de fourmis. En revanche, le recuit simulé peut converger vers un optimum global sous certaines conditions [12].

Même si les algorithmes gloutons prennent la décision optimale localement, ils peuvent échouer à trouver l'optimum global. Ils parviennent tout de même à trouver la solution optimale de nombreux problèmes [160].

La programmation dynamique permet de trouver la solution optimale à un problème. Les algorithmes par séparation et évaluation trouvent la solution optimale et le prouvent par énumération exhaustive des solutions. La preuve de l'optimalité est également apportée par la programmation par contraintes et la programmation linéaire.

Les travaux présentés dans [21] mettent en relation l'optimalité d'un algorithme et son passage à l'échelle. Si l'algorithme ILP (*Integer Linear Programming*) permet de trouver des résultats optimaux sur des graphes de quelques centaines de nœuds, il est incapable de fournir une solution (même sous-optimale) pour des graphes plus gros alors que l'algorithme génétique trouve des résultats sous-optimaux mais pour des instances beaucoup plus grandes (plusieurs milliers de nœuds). Il ne faut donc pas perdre de vue qu'un algorithme de complexité exponentielle (en $O(x^n)$) va toujours fournir une solution (et optimale) pour n très petit. Il est donc important de disposer d'une méthode de résolution adaptée à la taille du problème et de trouver le compromis optimalité/taille du problème. Pour des instances de problème de taille très grande, il s'agit parfois tout simplement d'obtenir une solution, même sous-optimale, en utilisant les métaheuristiques.

2.3.11.2 Les métaheuristiques

Les métaheuristiques ne conduisent pas à la solution optimale (ou rarement), mais fournissent une solution. Cela amène à se poser la question : cette solution est-elle vraiment loin de la solution optimale ?

De nombreuses études ont cherché à comparer différentes métaheuristiques. Dans [202], les auteurs comparent la recherche tabou, un algorithme génétique et le recuit simulé. Les auteurs montrent que la recherche tabou donne de meilleurs résultats et plus rapidement. L'algorithme génétique a besoin de beaucoup de mémoire pour stocker les informations alors que la recherche tabou et le recuit simulé gèrent la mémoire plus efficacement. Dans [70], la recherche tabou donne également de meilleurs résultats que le recuit simulé.

Il existe d'autres métaheuristiques qui ne sont pas détaillées dans ce document, comme par exemple l'optimisation par essaims particulaires [33], la méthode *GRASP* [73], etc.

2.3.11.3 Programmation linéaire vs programmation par contraintes

Bien que la programmation linéaire et la programmation par contraintes soient considérées comme très proches, il existe de nombreuses différences entre ces deux approches, comme illustré par le tableau 2.1. Un apport significatif de la programmation par contraintes est la possibilité d'utiliser des contraintes non-linéaires (comme la contrainte cumulative présentée figure 2.8, ou la multiplication). L'autre avantage majeur est l'application d'heu-

ristiques qui permettent de guider la recherche et d'atteindre l'optimalité de la solution (et de le prouver !).

	Programmation par contraintes	Programmation linéaire
Variables	discrètes	continues et discrètes
Contraintes	linéaires & non-linéaires contraintes globales	linéaires
Modélisation	flexible structure préservée	modèle linéarisé
Résolution	consistance locale Parcours en profondeur	relaxation linéaire (simplex) séparation et évaluation
Recherche	heuristiques possibles	méthodes standards

TAB. 2.1 – Comparaison de la programmation par contraintes et la programmation linéaire

2.4 Synthèse

Dans ce chapitre, nous avons présenté les problèmes soulevés par l'extension de jeu d'instructions, et les méthodes existantes pour les résoudre. Une manière classique est de représenter les applications sous forme de graphes, et les instructions spécialisées sous forme de motifs de calcul.

La programmation par contraintes permet de résoudre les problèmes de manière optimale. Il est possible de diriger la recherche par le biais d'heuristiques. La modélisation d'un problème est flexible et facilitée par l'existence de contraintes non-linéaires. Cette méthode a été appliquée avec succès aux problèmes d'extension de jeux d'instructions grâce à l'apport d'une contrainte spécifique sur l'isomorphisme de graphes et de sous-graphes. Dans nos travaux, nous pouvons bénéficier de cet outil.

Le premier problème est celui de la génération d'instructions. Le nombre d'instructions potentielles est exponentiel en fonction du nombre de nœuds du graphe. Les instructions spécialisées doivent satisfaire des contraintes comme la connexité, le nombre d'entrées/sorties, le chemin critique, la surface, ou l'énergie. À notre connaissance, aucun outil de génération de motifs ne permet de prendre en compte toutes ces contraintes. Nous proposons une méthode flexible, basée sur la programmation par contraintes, qui permet de résoudre de manière exacte le problème de génération d'instructions sous contraintes architecturales et technologiques.

Le second problème est celui de la sélection d'instructions. La sélection d'instructions est un problème NP-complet, et la résolution du problème peut s'effectuer selon plusieurs objectifs. Grâce à la flexibilité de la programmation par contraintes, nous proposons une méthode unique de résolution de la sélection d'instructions où l'objectif est fixé par l'utilisateur.

Nous utilisons également la programmation par contraintes pour résoudre le problème d'ordonnancement sous contraintes de ressources ou sous contraintes de temps, et le problème d'allocation de registres.

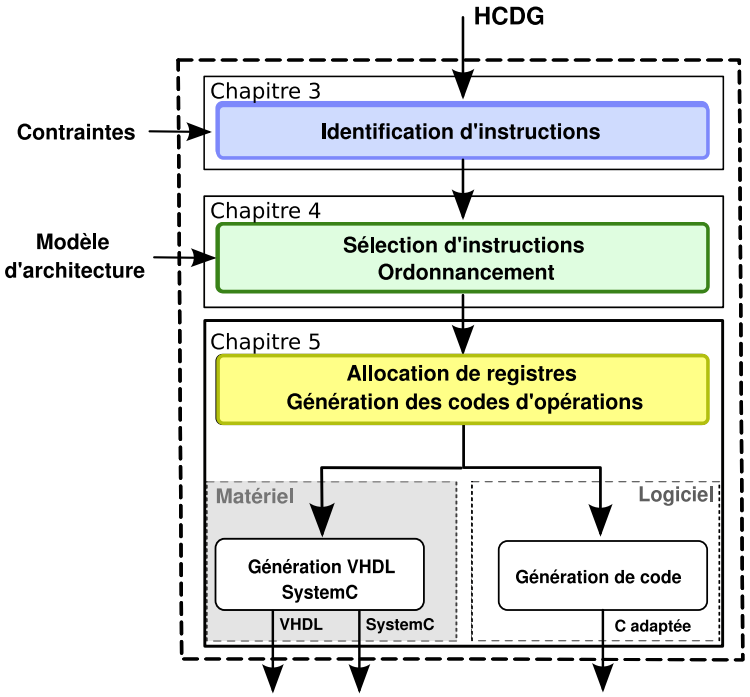


FIG. 2.9 – Principales étapes du flot de conception. Le chapitre 3 détaille l'étape d'identification d'instructions, le chapitre 4 la sélection d'instructions, et le chapitre 5 décrit les étapes d'allocation de registres, génération d'architecture et adaptation du code.

La figure 2.9 illustre les différentes étapes du flot de conception et situe leur description dans les chapitres de ce document. Ces étapes sont décrites dans trois chapitres. Le chapitre 3 décrit notre contribution concernant l'identification d'instructions. Le chapitre 4 présente nos techniques de sélection d'instructions et d'ordonnancement. Le chapitre 5 détaille les étapes d'allocation de registres, de génération des codes d'opérations, de génération d'architecture et de modèles, et d'adaptation du code.

3

L'identification d'instructions

L'identification d'instructions constitue la première étape du processus d'extension de jeu d'instructions, comme illustrée par la figure 3.1. Cette étape permet d'identifier, générer ou découvrir, pour une application donnée l'ensemble des instructions spécialisées candidates. L'application cible étant représentée sous forme de graphe, la génération d'instructions s'apparente alors à l'énumération de sous-graphes. Nous avons vu que l'énumération de sous-graphes est un problème dont la complexité est exponentielle en fonction du nombre de nœuds, donc pas applicable à des problèmes réalistes. Par ailleurs, nous avons vu également que l'énumération exhaustive de tous les sous-graphes n'est pas nécessaire puisque les instructions doivent satisfaire des contraintes architecturales et technologiques.

Dans ce chapitre, nous présentons notre méthodologie de résolution du problème d'identification d'instructions par la programmation par contraintes. Après avoir convenu de quelques définitions, nous présentons notre algorithme de génération de motifs, puis nous formalisons le problème pour la programmation par contraintes.

3.1 Définitions

La génération de motifs est appliquée à un graphe d'application acyclique $G = (N, E)$ où N est un ensemble de nœuds et E un ensemble de liens. Un motif est un graphe $P = (N_p, E_p)$. Une occurrence du motif P est un sous-graphe $M_p = (N_{M_p}, E_{M_p})$ du graphe G où $N_{M_p} \subseteq N$ et $E_{M_p} \subseteq E$. L'occurrence M_p est également sous-graphe isomorphe au graphe G.

L'ensemble des nœuds *successeurs directs* du nœud n est défini par $succ(n) = \{n' : (n, n') \in E\}$. De la même manière, nous définissons les *prédécesseurs directs* d'un nœud n comme $pred(n) = \{n' : (n', n) \in E\}$.

L'ensemble de *tous les successeurs* d'un nœud n est défini récursivement de la façon

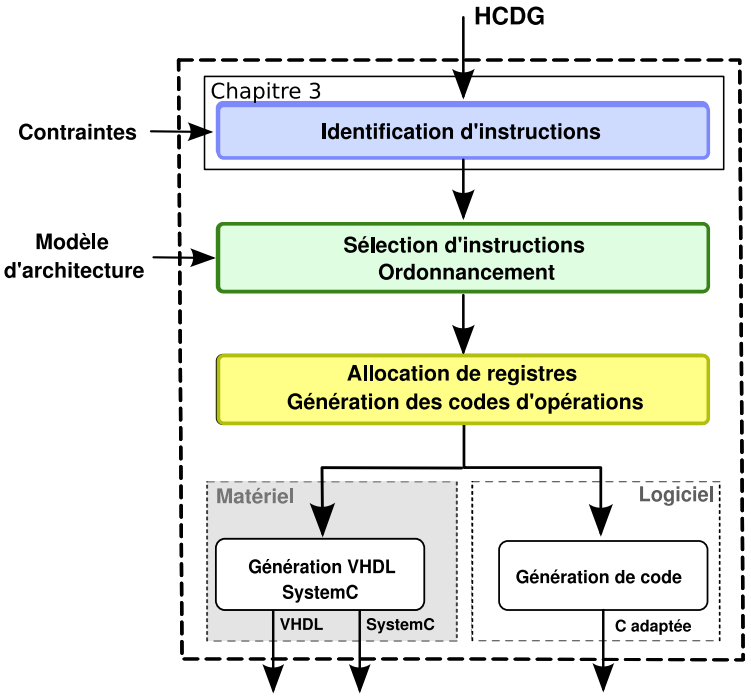


FIG. 3.1 – Le but du chapitre 3 est de présenter l'étape d'identification d'instructions

suivante : $allsucc(n) = \{n' \cup allsucc(n') : (n, n') \in E\}$.

Un chemin entre deux nœuds n_i et n_j dans un graphe G est défini tel que $path(G, n_i, n_j) = (n_i, n_{i+1}, \ldots, n_j)$ où deux nœuds consécutifs n_k, n_{k+1} de ce chemin respectent soit $(n_k, n_{k+1}) \in E$ soit $(n_{k+1}, n_k) \in E$.

3.2 Algorithme de génération de motifs

L'objectif de l'algorithme est de générer, à partir d'un graphe G, tous les motifs qui respectent les contraintes fixées. Le résultat est un ensemble de motifs définitivement identifiés (EMDI). Dans notre approche, un motif est construit autour d'un *nœud graine* (*seed node*).

3.2.1 Expansion d'un motif à partir d'un nœud graine

La figure 3.2 précise dans un pseudo langage de programmation notre algorithme de génération de motifs. Au démarrage, l'ensemble des motifs définitivement identifiés (EMDI) est vide (ligne 8). Chaque nœud du graphe est tour à tour considéré comme un nœud graine (ligne 9). Pour chaque nœud graine, la méthode $TrouverTousLesMotifs(G, n_s)$ ligne 11, implémentée en utilisant la programmation par contraintes et présentée plus loin dans ce document, identifie tous les motifs qui respectent les contraintes architecturales et technologiques fournies par l'utilisateur. Ces motifs sont stockés dans l'ensemble des motifs courants (EMC). Ensuite, pour chaque motif p de l'ensemble des motifs cou-

rants ($p \in EMC$), si p n'est pas déjà présent dans l'ensemble des motifs définitivement identifiés (c.-à-d. si il est isomorphe avec aucun des motifs, ligne 13), alors il est ajouté à l'ensemble des motifs temporaires (EMT). Finalement, les motifs ajoutés à l'ensemble des motifs définitivement identifiés (EMDI) sont les motifs de l'ensemble des motifs temporaires dont le nombre d'occurrences dans le graphe G est supérieur ou égal au nombre d'occurrences du nœud graine seul pondéré par un coefficient de filtrage $coef$ (où $0 \leq coef$) (ligne 18). Le nombre d'occurrences d'un motif dans le graphe d'application est aussi obtenu en utilisant les méthodes de programmation par contraintes implémentées par la fonction $TrouverToutesLesOccurrences(G, p)$.

```
1  // Entrées: G(N,E) -- graphe d'application,
2  //          N -- ensemble des noeuds,
3  //          E -- ensemble des liens,
4  //          EMDI -- Ensemble des Motifs Définitivement Identifiés,
5  //          EMC -- Ensemble des Motifs Courants,
6  //          EMT -- Ensemble des Motifs Temporaires
7  //          n_s -- noeud graine du motif
8  EMDI ← ∅
9  pour chaque n_s ∈ N
10   EMT ← ∅
11   EMC ← TrouverTousLesMotifs(G, n_s)
12   pour chaque p ∈ EMC
13       si ∀_{pattern∈EMDI} p ≢ pattern
14           EMT ← EMT ∪ {p},
15           NMP_p ← TrouverToutesLesOccurrences(G, p)
16   NMP_{n_s} ← TrouverToutesLesOccurrences(G, n_s)
17   pour chaque p ∈ EMT
18       si coef · NMP_{n_s} ≤ NMP_p
19           EMDI ← EMDI ∪ {p}
20 retourner EMDI
```

FIG. 3.2 – Algorithme d'identification des motifs

Il est également possible d'identifier les motifs pour seulement quelques nœuds graines bien choisis selon des heuristiques. Par exemple, lorsqu'un nœud n'a pas de prédécesseur et un seul successeur direct, l'ensemble des motifs générés par ce nœud est un sous-ensemble des motifs générés par son nœud successeur. Il suffit donc de générer les motifs pour le nœud successeur.

3.2.2 Limitation du nombre de motifs : *Smart filtering*

L'utilisation du *smart filtering* est illustrée à la ligne 18 de l'algorithme donné à la figure 3.2. Ce filtre réalise une sorte de pré-sélection, afin de limiter le nombre de motifs générés. Un nombre limité de motifs candidats facilite le travail de sélection de motifs. En effet, plus le nombre de candidats est élevé, plus le nombre de possibilités est grand et plus le temps de recherche de la solution optimale est long.

L'idée du *smart filtering* est de comparer le nombre d'occurrences d'un motif avec le nombre d'occurrences de son nœud graine seul pondéré par un coefficient de filtrage $coef$. Pour conserver tous les motifs, il faut fixer $coef$ à 0. Plus $coef$ est grand, et plus le nombre de motifs retenus sera faible.

3.2.2.1 Influence du filtrage

Le *smart filtering* permet de filtrer les motifs les moins fréquents dans un graphe. Le coefficient de filtrage doit dépendre de la régularité du graphe. En effet, un graphe très régulier fait apparaître le même motif de nombreuses fois, alors que dans un graphe irrégulier, il est difficile de trouver plusieurs fois le même motif. La figure 3.3 montre l'influence du filtrage sur le nombre de motifs générés et sur la couverture finale du graphe de l'algorithme *idct* issu du benchmark MediaBench [133]. Le diagramme de la figure 3.3 montre bien qu'il est possible de générer moins de motifs tout en conservant une qualité de couverture de graphe.

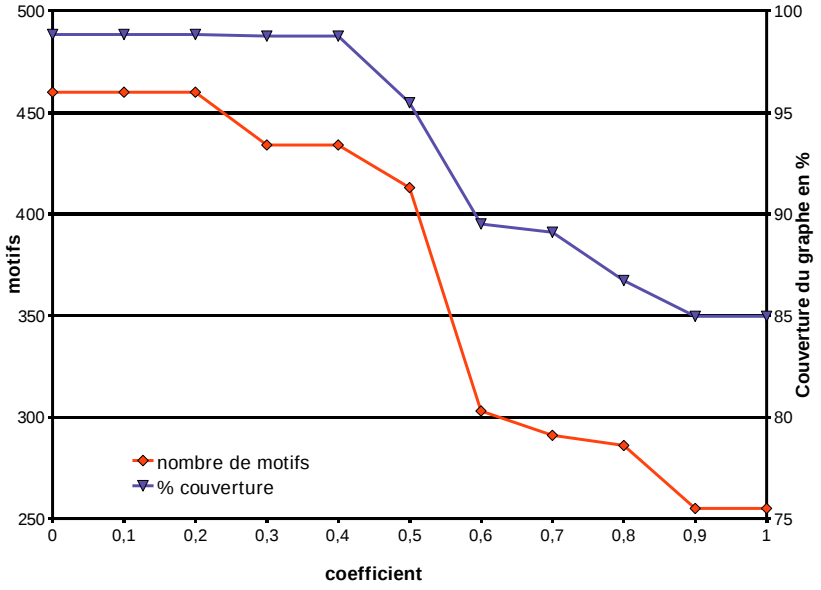


FIG. 3.3 – Influence du coefficient de filtrage $coef$ sur le nombre de motifs générés et la couverture du graphe pour l'algorithme *idct*. Plus $coef$ est grand, et moins le nombre de motifs générés est élevé, pour autant la couverture de graphe reste de bonne qualité.

3.2.3 La contrainte `GraphMatch`

La contrainte `GraphMatch` est une contrainte au sens de la programmation par contraintes mais n'est pas une contrainte « classique » interprétable par n'importe quel solveur de contraintes [178]. Cette contrainte a été spécialement conçue pour résoudre les problèmes d'isomorphisme de graphes et de sous-graphes et n'est fournie que par le solveur de contraintes JaCoP [114].

Cette contrainte est utilisée dans la méthode $TrouverToutesLesOccurrences(G, p)$ de l'algorithme donné à la figure 3.2, et s'applique à un graphe étiqueté spécialement construit que l'on note G_{csp}, qui contient un ensemble de nœuds. Chaque nœud contient des ports d'entrée ou de sortie qui sont explicitement connectés aux autres nœuds du graphes. Les étiquettes sont utilisées pour définir les opérations des nœuds (« `+` », ou « `*` »), et le type des ports. La contrainte `GraphMatch` définit un ensemble de règles qui spécifient les conditions d'isomorphisme de graphes. Elle associe un nœud du motif à un nœud du graphe, c'est-à-dire $f : V_t \rightarrow V_p$, où V_t est un nœud du graphe cible (t comme *target*) et V_p un nœud du motif (p comme *pattern*). L'association est faite si quatre conditions sont satisfaites.

1. Correspondance des étiquettes. Les étiquettes des nœuds doivent être les mêmes.
2. Correspondance au niveau port. Chaque port du motif doit être associé à un port du graphe.
3. Correspondance de lien entre les nœuds associés.
4. Non-chevauchement des motifs dans le graphe cible. Cette condition, propre à notre contexte, est nécessaire pour empêcher qu'un même nœud puisse être associé à plusieurs motifs.

La fonction d'association f peut être partielle, alors $f : V_t \rightarrow V_p \cup \{\perp\}$. Cette caractéristique est utilisée pour réaliser un isomorphisme de sous-graphes. L'isomorphisme est alors restreint à une partie du graphe cible où les nœuds non associés sont marqués de la valeur $\perp$.

3.3 Contraintes technologiques et architecturales

La génération de motifs sous contraintes technologiques et architecturales est nécessaire pour, d'une part, garantir une mise en œuvre matérielle des instructions spécialisées et d'autre part, réduire l'espace de recherche et éviter le temps exponentiel de l'énumération exhaustive. Nous avons vu que la contrainte architecturale majeure se situe au niveau des entrées et des sorties. Le nombre de ports en lecture et en écriture de la file de registres étant limité, le processus de génération de motifs doit garantir le respect des contraintes sur le nombre d'entrées et de sorties. Les autres contraintes à prendre en compte sont des contraintes technologiques. Il est important de pouvoir limiter le chemin critique d'une instruction en l'adaptant à la fréquence d'horloge de l'architecture. De même, il est utile de pouvoir contrôler le coût en surface d'une instruction spécialisée. Les considérations énergétiques peuvent par ailleurs être intéressantes à prendre en compte.

Toutes ces contraintes sont posées dans un problème de satisfaction de contraintes qui constitue le cœur de la méthode $TrouverTousLesMotifs(G, n)$ (figure 3.2, ligne 11). Celles-ci sont formalisées dans la section suivante.

3.4 Formalisation pour la programmation par contraintes

Le motif de calcul créé autour du nœud graine n_s dans le graphe d'application G est un graphe acyclique $P_{n_s} = (N_p, E_p)$. Un exemple de motif est illustré figure 3.4. Ce motif est constitué de 3 nœuds, il a 3 entrées et 2 sorties.

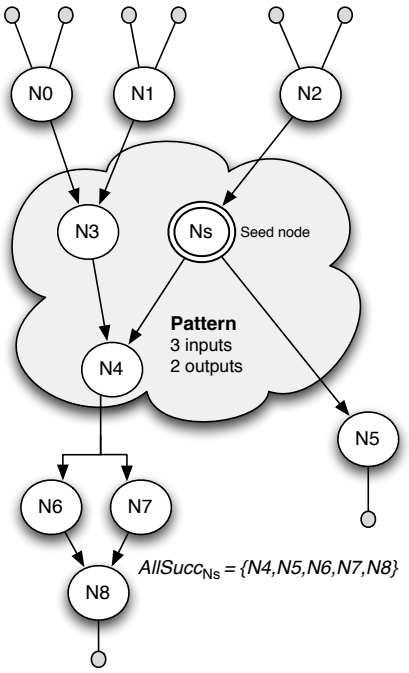


FIG. 3.4 – Motif à 3 entrées et 2 sorties

Les motifs valides sont les motifs qui respectent toutes les contraintes imposées. Du point de vue de la programmation par contraintes, seules les solutions qui respectent toutes les contraintes sont acceptées. Nous décrivons tout d'abord la contrainte de connexité. Puis nous verrons les contraintes relatives au nombre d'entrées et de sorties. Enfin, nous détaillons les contraintes technologiques.

3.4.1 Contrainte de connexité

La contrainte de connexité permet d'assurer qu'un motif généré est connexe, c'est-à-dire qu'il existe une connexion entre chaque paire de nœuds du motif. Nous présentons ici deux contraintes de connexité. La première contrainte de connexité présentée est basée sur les contraintes dites « classiques » de la programmation par contraintes [113]. Nous l'appellerons contrainte de connexité *classique* et la distinguerons de la contrainte spécialement définie pour nos besoins et appelée contrainte de connexité *spéciale*.

3.4.1.1 La contrainte de connexité classique

La contrainte définie par l'équation (3.1) spécifie que pour tout nœud $n \in N_p$, différent du nœud graine n_s, il existe un chemin dans le motif P_{n_s} entre ce nœud et le nœud graine, et ce quel que soit le sens.

$$\forall n \in N_p \wedge n \neq n_s \;\; \exists path(P_{n_s}, n, n_s) \tag{3.1}$$

À chaque nœud est associée une variable n_{sel}. Cette variable définit si le nœud appartient au motif ou non. Elle vaut 1 si le nœud appartient au motif $n \in N_p$ et 0 sinon. La variable du nœud graine vaut toujours 1 ($n_{s_{sel}} = 1$), car le nœud graine n_s appartient forcément au motif. Les contraintes de connexité imposées à un nœud diffèrent selon que ce nœud appartient à l'ensemble de tous les successeurs du nœud graine ou non. Ainsi, deux ensembles de nœuds sont constitués, comme schématisé par la figure 3.5. Le premier ensemble est l'ensemble de tous les nœuds successeurs du nœud graine ($n \in allsucc(n_s)$), les nœuds en *gris foncé* sur la figure 3.5. Le deuxième ensemble (ensemble des nœuds en *gris clair*) correspond à l'ensemble des autres nœuds ($n \in N - (allsucc(n_s) \cup n_s)$). On voit bien sur la figure que l'ensemble des nœuds en gris clair n'est pas simplement l'ensemble des prédécesseurs du nœud graine. Cet ensemble inclut également les nœuds qui ne sont ni prédécesseur, ni successeur du nœud graine. Le nœud graine n'appartient à aucun des deux ensembles.

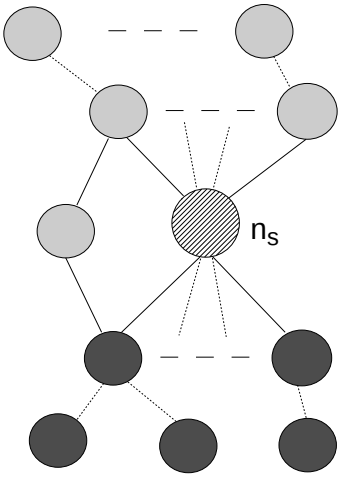


FIG. 3.5 – Partage des nœuds autour du nœud graine en deux ensembles

$$\forall n \in (N - (allsucc(n_s) \cup n_s)) : n_{sel} = 1 \;\Rightarrow \sum_{m \in succ(n)} m_{sel} \geq 1 \tag{3.2}$$

$$\forall n \in (N - (allsucc(n_s) \cup n_s)) : \sum_{m \in succ(n)} m_{sel} = 0 \Rightarrow \; n_{sel} = 0 \tag{3.3}$$

Les équations (3.2) et (3.3) expriment les contraintes imposées à l'ensemble des nœuds en gris clair ($n \in N - (allsucc(n_s) \cup n_s)$). Pour le motif de la figure 3.4, ce sont par exemple les nœuds $n \in \{N0, N1, N2, N3\}$.

La contrainte (3.2) impose la sélection d'au moins un nœud parmi les successeurs du nœud n si ce nœud appartient au motif. À l'inverse, la contrainte (3.3) empêche le nœud n de faire partie du motif si aucun de ses successeurs n'appartient au motif. En d'autres termes, le nœud n ne peut faire partie du motif que si au moins un de ses successeurs directs appartient au motif.

$$\forall n \in allsucc(n_s) : n_{sel} = 1 \;\Rightarrow \sum_{m \in (pred(n) \cap (allsucc(n_s) \cup n_s))} m_{sel} \geq 1 \tag{3.4}$$

$$\forall n \in allsucc(n_s) : \sum_{m \in (pred(n) \cap (allsucc(n_s) \cup n_s))} m_{sel} = 0 \;\Rightarrow\; n_{sel} = 0 \tag{3.5}$$

Les contraintes (3.4) et (3.5) concernent l'ensemble des nœuds en gris foncé ($n \in allsucc(n_s)$). Un *nœud en gris foncé* appartient au motif si et seulement si au moins un de ses prédécesseurs directs appartient au motif (équation 3.4). Si aucun de ses prédécesseurs directs n'appartient au motif, alors ce nœud n'appartient pas au motif (3.5).

Cette contrainte de connexité est exprimée à l'aide de contraintes dites « classiques », ce qui génère beaucoup de variables et de nombreuses contraintes. Afin de diminuer le nombre de variables et de contraintes, et ainsi réduire le temps de résolution du problème, une contrainte de connexité spéciale a été étudiée et proposée.

3.4.1.2 La contrainte de connexité spéciale `ConnectivityConstraint`

Dans le cadre d'une collaboration avec le Professeur Kuchcinski de l'Université de Lund en Suède, une contrainte a été spécialement conçue pour imposer la connexité avec un nœud graine n_s dans un graphe G_{csp} (équation 3.6).

$$ConnectivityConstraint(G_{csp}, n_s[, depth]) \tag{3.6}$$

où *depth* est optionnel et définit le rayon d'action de la contrainte. La contrainte est imposée aux nœuds à une distance *depth* de n_s. Le graphe G_{csp} est le même graphe que celui utilisé pour l'isomorphisme de (sous-)graphes (cf. section 3.2.3).

3.4.1.3 Comparaison des deux approches

Afin de comparer les deux approches, deux mesures différentes ont été réalisées pour chaque contrainte. La première mesure concerne le temps de génération total des motifs. La deuxième mesure est chargée de calculer le temps cumulé de résolution des problèmes. Pour chaque nœud graine, un problème de satisfaction de contraintes est posé. Nous avons

mesuré le temps nécessaire à la résolution de ce problème. Le temps cumulé de résolution des problèmes est la somme du temps de résolution de chaque problème (c'est-à-dire pour chaque nœud).

La figure 3.6 montre les mesures de temps effectuées pour les deux contraintes de connexité pour différents algorithmes. Pour chaque algorithme, quatre barres sont présentées, deux pour la contrainte classique et deux pour la contrainte spéciale. La première barre montre le temps de résolution pour la contrainte classique (*classic solving time*), et la deuxième barre pour la contrainte spéciale (*special solving time*). La troisième barre indique le temps de génération total des motifs pour la contrainte classique (*classic total time*), et la quatrième barre pour la contrainte spéciale (*special total time*). On s'aperçoit que le temps de résolution avec la contrainte spéciale est plus long que le temps de résolution avec la contrainte classique. Ce qui se répercute sur le temps global de génération des motifs : le temps de génération des motifs est plus long avec la contrainte spéciale qu'avec la contrainte classique.

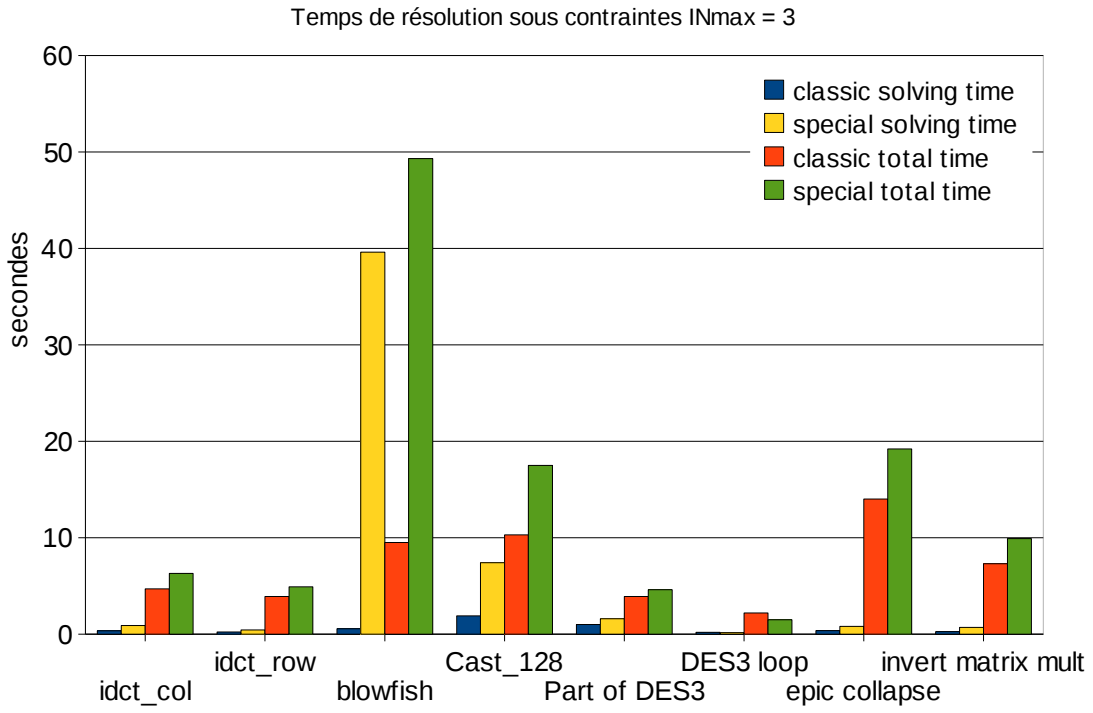


FIG. 3.6 – Mesure de temps pour les deux contraintes de connexité. Le temps de résolution des problèmes (*solving time*) est plus long avec la contrainte spéciale, ce qui se répercute sur le temps total de génération des motifs (*total time*).

Le tableau 3.1 montre le nombre de motifs générés pour les deux contraintes, où $NNmax$ désigne le nombre maximal de nœuds autorisés et $INmax$ le nombre maximal d'entrées. On constate que le nombre de motifs générés n'est pas le même. En particulier, pour l'algorithme *epic collapse*, le nombre de motifs générés sous contrainte de 3 nœuds

maximum est de 158 avec la contrainte spéciale, alors que ce nombre est de seulement 140 avec la contrainte classique. Un nombre de motifs générés différent n'est évidemment pas normal et nous avons donc cherché à identifier les motifs générés supplémentaires lorsque la contrainte spéciale est appliquée. La figure 3.7(a) montre le seul motif supplémentaire pour l'algorithme $DES3$ lorsque le nombre de nœuds est limité à 4. Ce motif a trois occurrences dans le graphe d'application, et toutes ses occurrences sont similaires à celle présentée figure 3.7(b). L'occurrence est non-convexe et toutes les occurrences de ce motif sont non-convexes. Quelle que soit la position du nœud graine parmi les quatre possibilités, la contrainte de connexité classique ne permet pas de trouver ce motif. On peut en déduire que l'espace de recherche n'est pas le même pour les deux contraintes, ce qui explique la différence du temps de résolution.

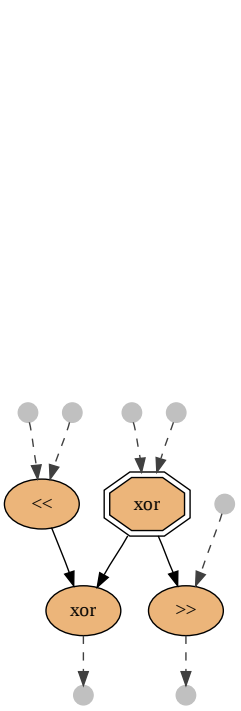
Benchmarks	NNmax = 3		INmax = 3		NNmax = 5	
	classique	spéciale	classique	spéciale	classique	spéciale
idct col	170	170	118	118	1380	1380
idct row	154	154	80	80	1406	1406
blowfish	61	61	57	57	354	354
cast 128	128	128	60	60	1073	1073
DES3	115	115	96	96	**911**	**938**
DES3 loop kernel	67	67	37	37	364	364
epic collapse	**140**	**158**	56	56	-	-
invert matrix multiplication	43	43	18	18	213	-

TAB. 3.1 – Nombre de motifs générés selon la technique de connexité utilisée sous différentes contraintes

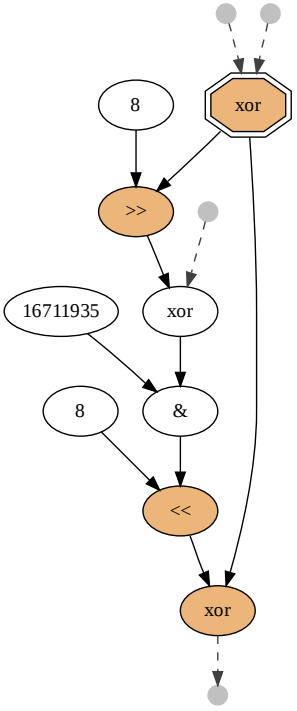
Les figure 3.8(a) et 3.8(b) schématisent l'espace de recherche exploré par les deux contraintes (classique et spéciale). Il est important de noter qu'un motif est construit à partir d'une de ses occurrences, que l'on peut appeler *occurrence mère*. Si cette occurrence mère satisfait les contraintes, alors elle sert de base à la génération du motif. La forme en ellipse représente l'ensemble des solutions possibles, découpé selon la connexité et la convexité de l'occurrence mère. La contrainte spéciale explore l'ensemble des solutions parmi les occurrences mères connexes, alors que la contrainte classique n'explore qu'une partie de cet espace. La contrainte de connexité classique permet donc de générer l'ensemble des motifs dont l'occurrence mère est connexe et convexe, plus un sous-ensemble des motifs dont l'occurrence mère est connexe et non-convexe. La contrainte de connexité classique offre un avantage puisque les occurrences non-convexes ne permettent pas d'obtenir un ordonnancement possible et sont filtrées par la suite lors de l'étape de sélection des motifs.

3.4.2 Contrainte sur le nombre d'entrées et de sorties

Les motifs identifiés avec la contrainte de connexité (paragraphe 3.4.1) peuvent avoir n'importe quel nombre d'entrées et de sorties. Pour satisfaire les spécifications architecturales sur les interfaces, nous avons besoin de contrôler le nombre d'entrées et le nombre de sorties du motif. Nous ajoutons pour cela des contraintes. Pour définir ces contraintes,

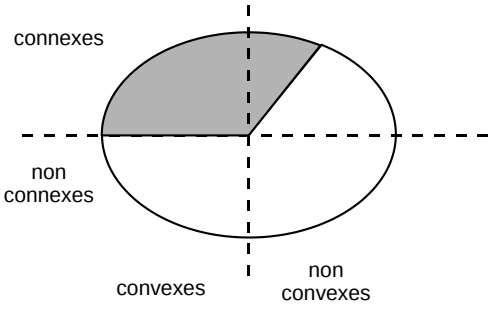


(a) Motif de 4 nœuds

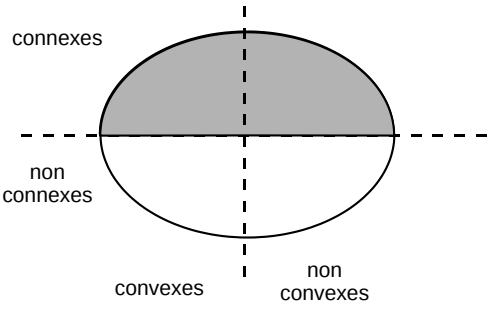


(b) Occurrence non-convexe du motif dans le graphe

FIG. 3.7 – Exemple d'un motif trouvé uniquement dans le cas de la contrainte spéciale



(a) Espace de recherche pour la contrainte de connexité classique



(b) Espace de recherche pour la contrainte de connexité spéciale

FIG. 3.8 – Espace de recherche des solutions pour les deux contraintes de connexité

nous avons besoin d'introduire, pour chaque nœud $n \in N$, les constantes $indegree_n$ et $outdegree_n$ qui définissent respectivement leur nombre de liens entrants et de liens sortants. Certains de ces liens peuvent devenir des entrées ou des sorties du motif. Le nombre d'entrées des motifs identifiés ne doit pas dépasser $INmax$, et le nombre de sorties ne doit pas dépasser $OUTmax$.

$$\forall n \in N : n_{in} = indegree_n - \sum_{m \in pred(n)} m_{sel} \tag{3.7}$$

$$\sum_{n \in N} (n_{sel} \cdot n_{in}) \leq INmax \tag{3.8}$$

$$\forall n \in N : n_{out} = outdegree_n - \sum_{m \in Succ_n} m_{sel} \tag{3.9}$$

$$\sum_{n \in N} (n_{sel} \cdot n_{out}) \leq OUTmax \tag{3.10}$$

Un lien entrant d'un nœud $n \in N_p$ devient une entrée du motif si ce lien est un lien externe du nœud n ou si le nœud source de ce lien n'appartient pas au motif. De la même façon, un lien sortant devient une sortie si c'est un lien externe ou si le nœud destination n'appartient pas au motif. Les contraintes (3.8) et (3.10) limitent le nombre d'entrées et de sorties du motif créé, où n_{in} désigne le nombre d'entrées pour le nœud n. Cette variable est pondérée par la variable de sélection du nœud n_{sel}. Ainsi, si le nœud n'est pas sélectionné, alors $(n_{sel} \cdot n_{in}) = 0$ et son nombre d'entrées n'est pas pris en compte dans le calcul. Il en est de même pour les sorties où n_{out} désigne le nombre de sorties du nœud n.

Dans notre contexte de motifs de calcul et de graphe qui représente un flot de données, une donnée produite par un nœud peut être utilisée par plusieurs nœuds. Pour les nœuds $n \in N_p$ possédant plusieurs liens sortants, il faut dans ces cas là ne compter qu'une seule sortie. La contrainte (3.9) est alors remplacée par la contrainte (3.11).

$$\forall n \in N : \sum_{m \in succ(n)} m_{sel} < outdegree_n \Leftrightarrow n_{out} = 1 \tag{3.11}$$

3.4.3 Contrainte sur le chemin critique

Comme nous l'avons vu dans la section 2.1.3, le chemin critique définit la fréquence d'horloge de l'architecture. Il est donc important de pouvoir le contrôler. Nous introduisons pour cela trois nouvelles variables, $start_n$, $delay_n$, et $endTime_n$, pour chaque nœud $n \in N$. Ces variables définissent la date de début, le délai et la date de fin de chaque nœud. Pour chaque nœud qui n'appartient pas au motif, nous imposons leur délai à 0 (contrainte 3.12).

$$\forall n \notin N_p : delay_n = 0 \tag{3.12}$$

Sur l'exemple de la figure 3.4, le délai de chaque nœud $n \in \{N0, N1, N2, N5, N6, N7, N8\}$

vaut 0. Les dépendances de données pour chaque lien $(n, m) \in E$ entre le nœud n et le nœud m sont définies par la contrainte (3.13). Cette contrainte permet d'assurer que la date de début du nœud m soit supérieure ou égale à la date de fin du nœud n.

$$\forall (n, m) \in E : start_n + (delay_n \cdot n_{sel}) \leq start_m \tag{3.13}$$

La contrainte (3.14) définit la date de fin $endTime_n$ pour chaque nœud $n \in N$.

$$\forall n \in N : (start_n + delay_n) \cdot n_{sel} = endTime_n \tag{3.14}$$

La date de fin $EndTime_n$ est différente de zéro seulement pour les nœuds qui appartiennent au motif $n \in N_p$. La contrainte (3.15) impose le délai maximal pour le motif entier, où le délai maximal autorisé pour le chemin critique est défini par $PatternCriticalPath$.

$$\forall n \in N : endTime_n \leq PatternCriticalPath \tag{3.15}$$

3.4.4 Contrainte de ressources

Dans certains cas, il peut être intéressant de contrôler non seulement le délai d'un motif, mais aussi sa taille. La contrainte (3.16) impose le nombre de nœuds maximum du motif créé, où $PatternNumberOfNodes$ est le nombre maximal de nœuds autorisés. Il est facile de contrôler plus finement la taille d'un motif en utilisant une somme pondérée en fonction du type du nœud (3.17). Il est ainsi possible de contrôler le *poids* d'un motif, où $PatternWeight$ désigne le poids maximum autorisé, et $weight_n$ désigne le poids du nœud n.

$$\sum_{n \in N} n_{sel} \leq PatternNumberOfNodes \tag{3.16}$$

$$\sum_{n \in N} weigth_n \cdot n_{sel} \leq PatternWeight \tag{3.17}$$

3.4.5 Contrainte d'énergie

La contrainte sur la consommation d'énergie d'un motif ressemble fortement à la contrainte sur la taille. Il suffit de faire la somme de la consommation de chaque nœud seul (contrainte 3.18). Pour avoir une estimation précise de la consommation en énergie d'un motif, il est important de disposer de chiffres précis.

$$\sum_{n \in N} \varepsilon_n \cdot n_{sel} \leq PatternEnergy \tag{3.18}$$

où ε_n est la consommation en énergie du nœud n.

opérateur	latence (ps)
additionneur	2500
multiplieur	7500
and	2200
xor	2200
shift	2500

TAB. 3.2 – Latence matérielle des opérateurs pour un StratixII

Nous avons présenté les contraintes supportées par notre système de génération de motifs, et avons montré comment les exprimer par la programmation par contraintes. Il est important de noter que ces contraintes peuvent être imposées les unes indépendemment des autres. Ainsi, l'utilisateur définit une combinaison de contraintes qu'il souhaite appliquer.

3.5 Résultats d'expérimentations

3.5.1 Le processeur cible

Dans le cadre de nos expérimentations, notre processeur cible est un NiosII avec une extension sous forme de bloc logique spécialisé étendu. Le composant cible est un FPGA StratixII EP2S60 FC672 d'Altera. Pour connaître les latences matérielles de chaque opérateur, nous avons synthétisé chaque opérateur seul, opérant sur des données de 32 bits, en utilisant l'outil Synplify pro 8.9 [188]. Le tableau 3.2 montre la latence matérielle de quelques opérateurs. La latence est exprimée en picosecondes. Pour la latence logicielle, nous avons pris les chiffres d'un NiosII version rapide (*Nios2Fast*) où les multiplications et les décalages se font en 3 cycles (1 cycle d'exécution + 2 cycles de latence), grâce aux multiplieurs embarqués de la famille des Stratix [18]. Toutes les autres instructions s'exécutent en un seul cycle.

3.5.2 Environnement d'exécution

Les expérimentations sont réalisées sur un PC muni d'un Intel Centrino Duo (2 processeurs T7600 cadencés à 2,33 GHz) et de 2 Go de mémoire RAM. Le système d'exploitation est un Linux (Fedora Core 9). Tous nos outils sont programmés en langage Java et interprétés par la machine virtuelle Java de Sun (version 1.6).

3.5.3 Benchmarks

Nos techniques ont été appliquées sur des extraits de code issus d'applications multimédias et cryptographiques provenant de nombreux benchmarks. MediaBench [133] est un benchmark qui contient des applications en relation avec le multimédia et les télécommunications : MPEG2, JPEG, ADPCM, GSM, etc. MediaBenchII a poursuivi cette idée en apportant les applications H.264 et JPG2000. MiBench [96] est un benchmark qui propose

cinq grandes familles d'applications : *automotive*, *network*, *office*, *security*, et *telecomm*. PolarSSL [165] est une bibliothèque open-source d'applications cryptographiques et de protocoles de sécurisation des échanges sur internet SSL/TLS *(Secure Sockets Layer/ Transport Layer Security)*. Cette bibliothèque contient les applications typiques que l'on trouve dans les systèmes embarqués : AES, DES/3DES, MD5, SHA. Mcrypt [150] et sa bibliothèque associée libmcrypt a pour objectif de remplacer l'ancien Unix crypt, sous licence GPL tout en proposant un choix plus élargi d'algorithmes. DSPstone [68] propose de nombreuses applications du traitement du signal propices à une exécution sur des processeurs de type DSP. En se basant sur l'observation que le code assembleur écrit à la main est bien plus performant que celui généré par un compilateur, DSPstone a voulu fournir des outils permettant d'évaluer les compilateurs et les processeurs de type DSP. Toutes les applications de tous les benchmarks cités sont écrites en langage C.

Le tableau 3.3 résume les différentes applications utilisées, le benchmark dont elles sont issues, et leurs caractéristiques. Les caractéristiques retenues ont trait au parallélisme présent dans les graphes, à leur régularité, leur chemin critique, leur connexité et leurs accès mémoire. Les différentes caractéristiques sont notées selon leur degré : fort, moyen, ou faible. Il est intéressant de noter que selon les caractéristiques des applications, certaines techniques présentées dans ce document s'appliquent plus ou moins bien. Par exemple, un filtre FIR présente beaucoup de régularité. Ainsi, le *smart filtering* s'applique particulièrement bien.

Algorithmes	Benchmark	Régularité	Parallélisme	Chemin critique	Connexité	Accès mémoire
JPEG BMP header	MediaBench	moyenne	moyen	court	moyenne	fort
JPEG downsample	MediaBench	faible	faible	long	faible	fort
JPEG IDCT	MediaBench	faible	fort	court	forte	moyen
EPIC collapse	MediaBench	forte	fort	court	moyen	moyen
Blowfish	MiBench	forte	moyen	long	moyenne	moyen
Cast128	Mcrypt	moyen	moyen	long	moyenne	fort
MESA invert matrix	MediaBench	forte	fort	moyen	moyenne	fort
DES3	PolarSSL	faible	moyen	moyen	moyen	faible
SHA	MiBench	faible	faible	court	moyen	moyen
FIR		fort	faible	long	faible	fort
FFT	MiBench	fort	moyen	court	fort	fort
IIR	DSPStone	fort	faible	long	faible	fort

Tab. 3.3 – Caractéristiques des applications utilisées

3.5.4 Couverture maximale

Notre méthode de génération de motifs a été évaluée sur des algorithmes issus des benchmarks MiBench [96] et MediaBench [133]. Afin d'évaluer la qualité des motifs générés, nous avons cherché à couvrir au maximum les graphes d'application avec les motifs générés. La technique de couverture est présentée dans la section 4.1.2.3 de ce document. Le tableau 3.4 présente les résultats obtenus pour les contraintes suivantes :

– 2 entrées/1 sortie et 4 entrées/2 sorties,

– 10 nœuds maximum,
– chemin critique de 15 ns, correspondant à 3 cycles d'un processeur NiosII dans sa déclinaison « rapide » cadencé à 200 MHz.

Le signe « - » dans le tableau indique qu'aucun motif respectant les contraintes n'a été trouvé. Les résultats montrent que pour trois algorithmes, il est possible de couvrir entièrement le graphe avec des motifs ayant au maximum 4 entrées et 2 sorties, c'est-à-dire, dans ces cas aucune instruction de base du processeur n'est utilisée. Dans le cas 2 entrées/1 sortie, il est possible de couvrir le graphe à 78% pour une *idct* ou pour l'encryptage *Blowfish*. La couverture atteint même les 88% pour l'algorithme *JPEG Write BMP Header*. En moyenne, la couverture est de 53% pour 2 entrées/1 sortie, et grimpe jusqu'à 87% pour des motifs à 4 entrées et 2 sorties.

Algorithmes		2 entrées / 1 sortie				4 entrées / 2 sorties			
	$\|N\|$	identifiés	sélectionnés	temps (s)	couverture [%]	identifiés	sélectionnés	temps (s)	couverture [%]
JPEG BMP Header	34	6	2	0.7	88	82	3	2.1	**100**
JPEG Downsample	66	11	1	1.2	19	74	5	1.1	98
JPEG IDCT	250	28	10	9.0	78	254	21	42.3	93
EPIC Collapse	274	7	4	9.5	68	157	9	1533.0	73
BLOWFISH encrypt	201	11	3	13.9	78	145	6	96.2	86
SHA transform	53	17	9	0.6	64	134	13	1.67	**100**
MESA invert matrix	152	2	2	3.2	11	53	12	8.4	72
FIR unrolled	67	3	2	1.8	9	24	3	3.6	**100**
FFT	10	-	-	-	-	6	2	0.2	60
Moyenne			4		**53**		8		**87**

Tab. 3.4 – Résultats obtenus par notre approche pour des algorithmes issus de MediaBench et MiBench.

3.5.5 Discussion

Afin de situer notre approche par rapport aux solutions existantes, nous avons comparé notre technique avec le système $UPaK$ [205, 207]. Le système $UPaK$ génère les motifs incrémentalement, en ajoutant, à chaque étape, un nœud voisin aux motifs courants. Le processus s'arrête lorsque le nombre de nœuds est égal à un paramètre donné. $UPaK$ permet donc de contrôler uniquement le nombre de nœuds d'un motif. Dans nos expérimentations, nous avons fixé le nombre de nœuds à 8. Le tableau 3.5 montre les résultats obtenus par le système $UPaK$ seul, puis par une combinaison des approches, où notre technique de génération de motifs est appliquée aux motifs identifiés par $UPaK$ et non aux graphes flot de données directement.

Le système $UPaK$ permet localement d'obtenir de meilleurs résultats. Par exemple, pour la transformée de Fourier (FFT), la couverture est de 100%. En imposant les contraintes de 4 entrées et 2 sorties sur les motifs découverts par $UPaK$, la couverture est de 60%, comme pour notre approche. Globalement, notre approche permet d'obtenir de

Algorithmes	$\|N\|$	**UPaK** identifiés	**UPaK** sélectionnés	**UPaK** temps (s)	**UPaK** couverture [%]	**UPaK + notre approche** 2 in / 1 out identifiés	2 in / 1 out sélectionnés	2 in / 1 out temps (s)	2 in / 1 out couverture [%]	4 in / 2 out identifiés	4 in / 2 out sélectionnés	4 in / 2 out temps (s)	4 in / 2 out couverture [%]
JPEG BMP Header	34	3	1	14.0	76	3	1	0.2	76	4	1	0.2	76
JPEG Downsample	66	21	10	1.3	72	-	-	-	-	17	8	0.2	72
JPEG IDCT	250	13	7	6.8	57	20	3	0.2	44	55	11	3.5	60
EPIC Collapse	274	10	2	22.2	53	1	1	0.1	53	10	2	0.1	54
BLOWFISH encrypt	201	26	3	113.9	67	2	1	0.2	51	11	2	2.5	59
SHA transform	53	6	6	27.3	27	-	-	-	-	7	6	0.1	27
MESA invert matrix	152	8	2	510.0	58	-	-	-	-	5	3	0.1	57
FIR unrolled	67	28	3	15.2	98	4	2	0.2	6	8	3	0.5	98
FFT	10	22	2	0.1	100	-	-	-	-	12	2	0.3	60
Moyenne			5		67		2		45		4		62

TAB. 3.5 – Résultats obtenus par $UPaK$ pour des algorithmes issus de MediaBench et MiBench.

meilleurs résultats que $UPaK$. De plus, contrairement à $UPaK$ qui est basé sur une heuristique, notre méthode est une méthode exacte.

Le tableau 3.6 positionne notre approche par rapport aux techniques existantes. À notre connaissance, notre approche est la seule qui permette de prendre en compte autant de contraintes architecturales et technologiques avec autant de flexibilité.

	Yu 2007 [217]	Bonzini 2007 [43]	Atasu 2003 [30, 169]	Atasu 2007 [27]	$UPaK$ 2007 [205]	Notre approche
support pour non connexité	✓		✓	✓		✓
contrôle du nombre d'entrées	✓	✓	✓	✓		✓
contrôle du nombre de sorties	✓	✓	✓	✓		✓
contrôle du nombre de nœuds					✓	✓
contrôle du chemin critique				✓		✓

TAB. 3.6 – Positionnement de notre approche par rapport aux approches existantes pour la génération d'instructions

Un autre sujet de discussion concerne les estimations du chemin critique d'un motif. Notre méthode évalue grossièrement le chemin critique d'un motif comme étant la somme des délais des nœuds. Nous disposons d'une bibliothèque des valeurs du chemin critique pour chaque type de nœud. On sait par exemple que le délai d'un multiplieur est de 6,6 ns sur un FPGA Altera CycloneII, et que le délai d'un additionneur est de 4,2 ns sur ce même composant. Notre calcul nous donne donc un délai de 10,8 ns pour une multiplication directement suivie d'une addition. Or, le résultat de la synthèse nous fournit le chiffre de 7,8 ns. Le délai d'un MAC n'est donc pas simplement la somme des délais d'un multiplieur et d'un additionneur.

Afin d'obtenir des mesures plus fines, une idée serait de disposer d'une bibliothèque de motifs à plusieurs nœuds (par exemple un MAC) avec des chiffres précis en termes de délai. Le délai d'un nœud serait alors variable en fonction de son voisinage. La prise en

compte de cette possibilité risque de complexifier grandement les contraintes à imposer au problème. Le même raisonnement peut être appliqué pour le coût en ressources matérielles d'un motif ou bien sa consommation d'énergie.

3.6 Conclusion

Nous avons vu dans ce chapitre comment générer, à partir d'un graphe d'application, des instructions spécialisées qui respectent des contraintes architecturales et technologiques. Ces instructions sont générées à l'aide de la programmation par contraintes. Notre technique permet de prendre en compte les contraintes sur les entrées et les sorties, sur la taille, le chemin critique, la consommation d'énergie et la connexité d'un motif, et à notre connaissance, est la seule à prendre en compte autant de contraintes architecturales et technologiques. L'utilisateur est libre de choisir les contraintes à imposer. Notre méthode résout le problème de l'identification d'instructions de manière exacte et les résultats obtenus sont meilleurs que ceux fournis par le système $UPaK$. Notre méthode a fait l'objet d'une publication à la conférence *SAMOS* (*International Symposium on Systems, Architectures, MOdeling and Simulation*) [1].

4

Sélection d'instructions et ordonnancement

La sélection d'instructions est la deuxième étape fondamentale du processus d'extension de jeu d'instructions, comme illustrée par la figure 4.1. Cette étape a pour objectif de conserver, parmi un ensemble d'instructions candidates, celles qui optimisent une fonction de coût. Cette fonction de coût diffère selon les objectifs : minimisation du temps d'exécution, du nombre d'instructions différentes, etc. Dans notre contexte d'extension de jeu d'instructions, la sélection d'instructions revient à déterminer les opérations de l'application exécutées à l'aide des instructions de base du processeur (c'est-à-dire en logiciel) et les opérations exécutées à l'aide d'instructions spécialisées (c'est-à-dire en matériel). C'est le problème du partitionnement logiciel/matériel au niveau instruction. Ce problème est modélisé de différentes manières : couverture de sommets d'un graphe, graphe de conflit puis clique maximale ou bien stable de taille maximum, problème du sac à dos, etc. Quelle que soit la modélisation, le problème à résoudre est un problème NP-complet.

Ce chapitre présente tout d'abord notre méthode de sélection d'instructions basée sur la programmation par contraintes. Dans un deuxième temps, ce chapitre détaille notre technique d'ordonnancement par recalage d'instructions qui permet d'exécuter en parallèle une instruction sur l'extension et sur l'unité arithmétique du processeur. Cette technique s'appuie également sur la programmation par contraintes.

4.1 Sélection d'instructions

4.1.1 Définition d'une occurrence

Comme défini dans la section 2.1, une occurrence m est une instance d'un motif p dans un graphe G. Une occurrence est un isomorphisme de sous-graphes du motif p dans le graphe G. Un exemple d'occurrence est illustré à la figure 2.1(b) page 40.

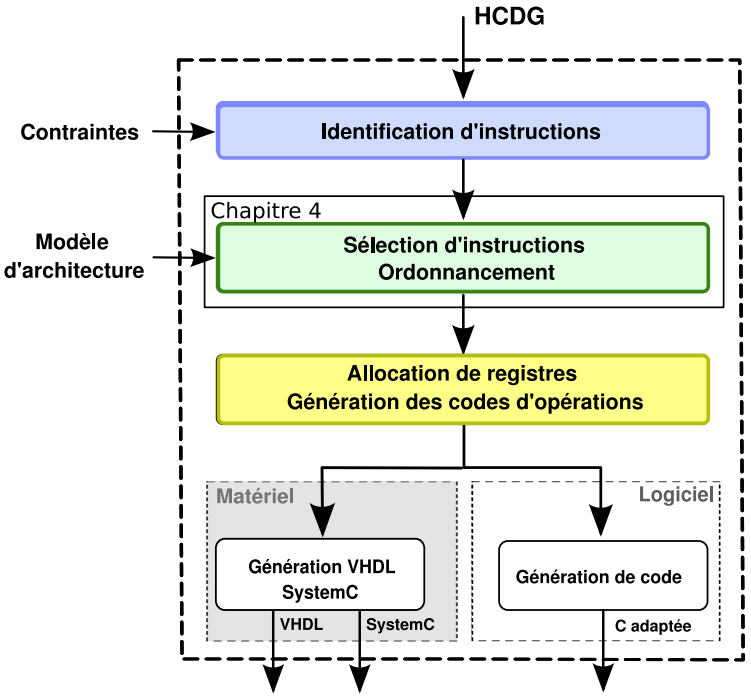


FIG. 4.1 – Le but du chapitre 4 est de présenter les étapes de sélection d'instructions et d'ordonnancement

4.1.2 Définition du problème de sélection d'instructions

L'objectif de la sélection d'instructions est de conserver parmi un ensemble d'instructions candidates (générées par l'étape d'identification d'instructions ou bien issues d'une bibliothèque), celles qui optimisent une fonction de coût. Nous avons vu que les stratégies sont diverses et variées. Dans notre cas, nous avons choisi de sélectionner les instructions qui optimisent le temps d'exécution de l'application. L'application est représentée sous forme de graphe, et chaque instruction candidate est un sous-graphe isomorphe au graphe d'application. Cela veut dire que chaque nœud du motif peut *couvrir* un nœud du graphe d'application. Le problème de la sélection d'instructions peut donc être modélisé par une technique de couverture de sommets d'un graphe.

4.1.2.1 Couverture de sommets

La couverture de sommets est un problème NP-complet [64]. Nous utilisons la programmation par contraintes pour résoudre ce problème. Pour le modéliser sous la forme d'un problème de satisfaction de contraintes, nous avons tout d'abord besoin d'identifier l'ensemble des instructions qui couvrent chaque nœud du graphe. L'algorithme qui permet de trouver toutes les occurrences qui couvrent un nœud est donné à la figure 4.2. Après exécution de celui-ci, à chaque nœud $n \in N$ est associé un ensemble d'occurrences ($matches_n$) qui peuvent le couvrir. La méthode `TrouverToutesLesOccurrences(G,p)` à la ligne 11 est identique à celle utilisée lors de la génération d'instructions, elle est chargée

```
1  // Entrées:
2  //  G=(N,E)-- graphe d'application,
3  //  EMDI-- Ensemble des Motifs Définitivement Identifiés
4  //  M_p-- ensemble des occurrences pour un motif p,
5  //  M-- ensemble de toutes les occurrences,
6  //  matches_n-- ensemble des occurrences qui peuvent
7  //              couvrir un noeud n,
8
9  M ← ∅
10 pour chaque p ∈ EMDI
11   M_p ← TrouverToutesLesOccurrences(G, p)
12   M ← M ∪ M_p
13 pour chaque m ∈ M
14   pour chaque n ∈ m
15     matches_n ← matches_n ∪ {m}
```

FIG. 4.2 – Algorithme trouvant toutes les occurrences qui couvrent un nœud dans un graphe

de trouver tous les isomorphismes de sous-graphes du motif p dans le graphe G.

Évidemment, dans la couverture finale du graphe G, un nœud $n \in N$ ne peut être couvert que par une et une seule occurrence. À titre d'illustration, la figure 4.3 montre les occurrences identifiées pour l'exemple de la figure 4.4. Les étoiles grises représentent les occurrences possibles alors que les étoiles noires indiquent les occurrences sélectionnées. Nous modélisons la sélection d'une occurrence donnée dans la couverture finale en utilisant une variable à domaine fini m_{sel} associée à chaque occurrence $m \in M$, où M est l'ensemble des occurrences. La valeur de la variable m_{sel} vaut 1 si l'occurrence m est sélectionnée, 0 sinon. La contrainte (4.1) impose que chaque nœud ne peut être couvert que par une et une seule occurrence.

$$\forall n \in N : \sum_{m \in matches_n} m_{sel} = 1 \tag{4.1}$$

4.1.2.2 Minimisation du temps d'exécution

Nous nous intéressons à la sélection de motifs en vue de la minimisation du temps d'exécution séquentielle d'une application sur un processeur possédant une extension, capable d'exécuter une instruction spécialisée à la fois. Cet objectif peut être obtenu en minimisant la fonction de coût définie par l'équation (4.2) où le délai introduit par l'exécution d'une occurrence (m_{delay}) dépend du modèle de l'architecture et est présenté en détail dans la section 4.1.3. Dans ce cas précis, le problème de la sélection de motifs corres-

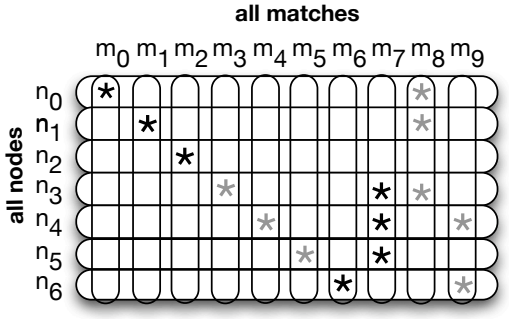


FIG. 4.3 – Occurrences identifiées et sélectionnées pour l'exemple de la figure 4.4.

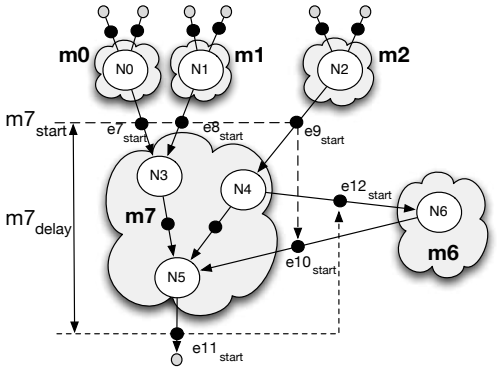


FIG. 4.4 – Exemple d'une occurrence non-convexe

pond au problème de couverture de graphe où tous les nœuds doivent être couverts par une occurrence et la somme des délais des occurrences qui couvrent ce graphe doit être minimale.

$$ExecutionTime = \sum_{m \in M} m_{sel} \cdot m_{delay} \tag{4.2}$$

Nous pourrions également sélectionner les motifs de calcul en vue d'un ordonnancement sous contraintes de ressources. Dans ce cas, $ExecutionTime$ est limité par une constante $ExecutionTime_{max}$ et le nombre de motifs différents sélectionnés est minimisé. Ce nombre est calculé par la contrainte (4.4) où la variable p_s vaut 1 si au moins une des occurrences m du motif p a été sélectionnée pour couvrir le graphe (4.3), où $EMDI$ est l'ensemble des motifs définitivement identifiés.

$$\forall p \in EMDI : \sum_{m \in M_p} m_{sel} \ > 0 \Leftrightarrow p_s = 1 \tag{4.3}$$

$$NumberOfPatterns = \sum_{p \in EMDI} p_s \tag{4.4}$$

4.1.2.3 Minimisation du nombre de nœuds couverts par un motif à un nœud

Les motifs à un nœud représentent les instructions de base du processeur. Une autre stratégie de couverture est de chercher à minimiser le nombre d'instructions exécutées par le processeur, ce qui revient à minimiser le nombre de nœuds couverts par un motif à un nœud. Pour cela, la fonction de coût à minimiser est donnée par l'équation 4.5.

$$OneNodeMatches = \sum_{m \in ONM} m_{sel} \tag{4.5}$$

où $OneNodeMatches$ représente l'ensemble des occurrences à un nœud sélectionnées et ONM est l'ensemble des occurrences à un nœud. Cette stratégie a été utilisée dans le chapitre précédent (section 3.5.4) pour couvrir au maximum le graphe avec des instructions

spécialisées.

4.1.2.4 Filtrage d'occurrences

Il peut arriver que certaines occurrences de certains motifs ne nous intéressent pas. C'est le cas par exemple des occurrences non-convexes. Nous avons donc mis en place, avant de résoudre le problème de couverture du graphe, un système de filtrage qui permet d'éliminer certaines occurrences. Un filtre des occurrences non-convexes permet alors de supprimer de l'ensemble des solutions les occurrences non-convexes.

La détection de la convexité d'une occurrence est coûteuse en temps de calcul puisqu'il faut parcourir chaque nœud de l'occurrence, puis pour chaque nœud il faut regarder ses prédécesseurs directs, et pour chaque prédécesseur direct qui n'appartient pas à l'occurrence, il faut assurer qu'aucun nœud parmi tous ses prédécesseurs n'appartient à l'occurrence. Par exemple, prenons l'occurrence $m7$ de la figure 4.4. Les nœuds de l'occurrence sont $N3, N4, N5$. Lorsque l'on testera le nœud $N5$, on cherche ses prédécesseurs directs qui sont $N3, N4, N6$. Comme le nœud $N6$ n'appartient pas au motif, on cherche tous les prédécesseurs de $N6$ qui sont $N4, N2$. Or, $N4$ appartient déjà à l'occurrence. Cette occurrence n'est donc pas convexe et est supprimée de l'espace des solutions.

Outre les occurrences non-convexes, nous filtrons également les occurrences avec des valeurs immédiates qui sont particulièrement pénalisantes dans le cas du NiosII. En effet, l'envoi d'une valeur immédiate vers l'extension coûte un cycle de plus puisqu'il faut explicitement réaliser une instruction de chargement d'un registre avec une valeur immédiate (`movi`), pour ensuite pouvoir exécuter une instruction spécialisée avec en opérande ce même registre. Le graphe de la figure 4.5(a) montre une partie du graphe d'une transformée en cosinus discrète inverse (*idct*) tirée du benchmark MediaBench [133] et la figure 4.5(b) correspond à un motif candidat. La figure 4.5(c) met en évidence le résultat de la couverture du graphe par ce motif candidat. Une telle solution serait traduite en le code assembleur suivant :

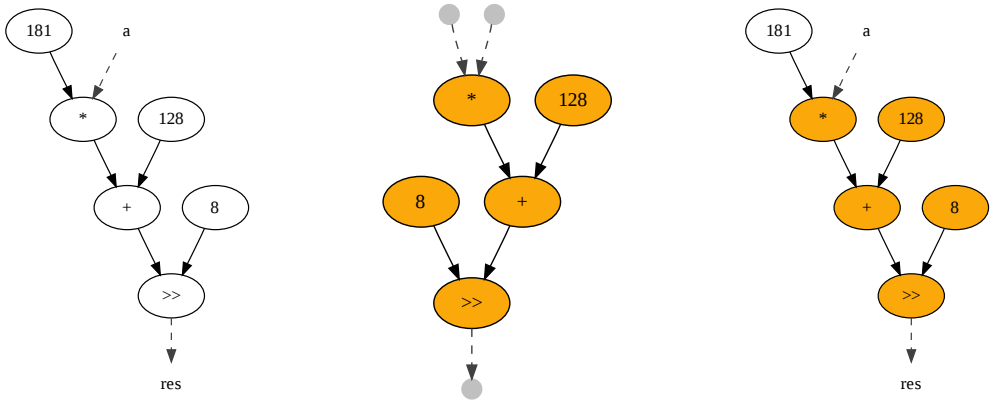
```
...
movi r3,#181
custom 10, r2, r3, r4
...
```

où `movi r3,#181` réalise le chargement de la valeur 181 dans le registre `r3` et en supposant que la valeur de a est stockée dans le registre `r4`, que le numéro de l'instruction spécialisée est le numéro `10`, et que le résultat res est rangé dans le registre `r2`.

Nous avons choisi de filtrer ces occurrences, c'est-à-dire de les enlever des solutions possibles, mais nous pourrions également choisir de prendre en compte cette pénalité dans la modélisation du temps d'exécution de l'occurrence.

4.1.2.5 Contraintes temporelles

Le filtrage des occurrences non-convexes n'est pas suffisant pour garantir un ordonnancement possible à partir des instructions sélectionnées. Il peut arriver que deux occurrences,



(a) Une partie du graphe idct de MediaBench

(b) Un motif candidat

(c) Couverture avec une valeur immédiate en entrée du motif

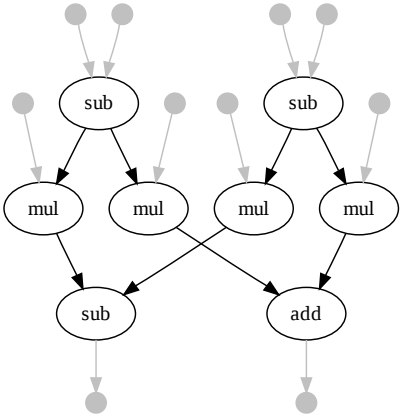
FIG. 4.5 – Filtrage des occurrences avec valeur immédiate

distinctes et convexes, puissent provoquer une situation d'interblocage lorsqu'elles sont sélectionnées simultanément. Par exemple, la figure 4.6(a) montre le graphe d'une transformée de Fourier (FFT) tirée du benchmark MiBench [96]. La figure 4.6(b) montre deux motifs candidats dont les occurrences sont convexes lorsqu'elles sont sélectionnées seules, et la figure 4.6(c) montre que la sélection simultanée de ces deux occurrences provoque une interdépendance entre les deux motifs et qu'un tel graphe est impossible à ordonnancer. La figure 4.6(d) montre bien l'apparition du cycle lorsque le graphe est réduit.

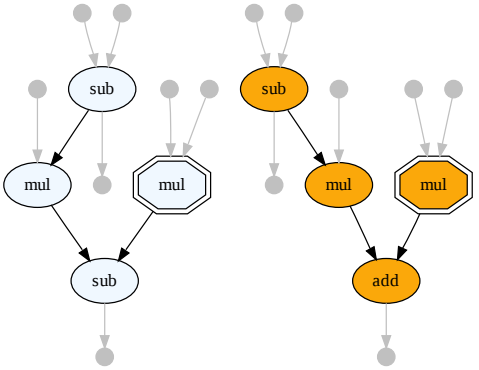
Afin d'éviter ces cycles, nous imposons des contraintes sur les entrées et les sorties. Ces contraintes résultent des dépendances de données de l'occurrence. Pour cela, nous définissons tout d'abord une variable e_{start} pour chaque lien $e \in E$ du graphe d'application. Cette variable définit l'instant de début de l'exécution d'une occurrence. Puis, nous imposons la contrainte (4.6) pour chaque occurrence, qui force chaque lien sortant de l'occurrence à ne débuter qu'après l'instant de début de l'occurrence plus son délai.

$$m_{sel} = 1 \Rightarrow \forall_{e \in m_{in}} : e_{start} = m_{start} \wedge \forall_{e \in m_{out}} : m_{start} + m_{delay} \leq e_{start} \tag{4.6}$$

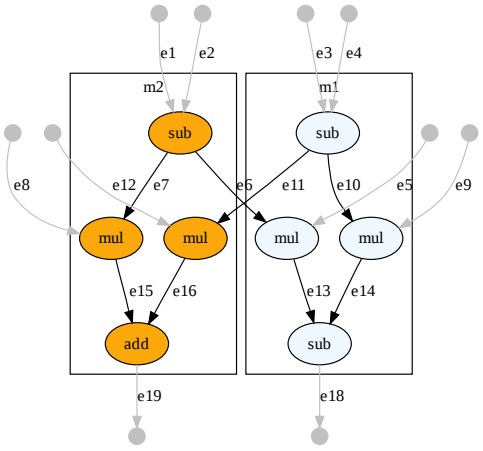
où la variable m_{start} définit l'instant de début de l'exécution de l'occurrence m et la variable m_{delay} spécifie sa durée d'exécution. L'ensemble m_{in} est l'ensemble des liens entrants de l'occurrence, c'est-à-dire les liens qui sont soit des entrées de l'application, soit des liens dont le nœud source n'appartient pas au motif. L'ensemble m_{out} est l'ensemble des liens sortants de l'occurrence, c'est-à-dire les liens qui sont soit des sorties de l'application, soit des liens dont le nœud destination n'appartient pas au motif.



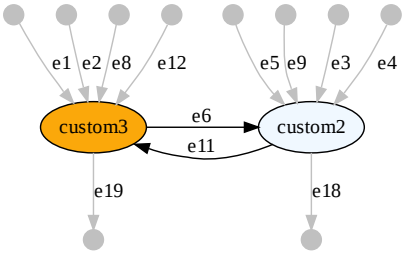
(a) Graphe d'application FFT



(b) Motifs candidats de la FFT



(c) Interdépendance entre les deux occurrences



(d) Introduction d'un cycle dans le graphe réduit

FIG. 4.6 – Exemple d'interdépendance entre deux occurrences sur une FFT

Par exemple, soit l'occurrence m_2 dans la figure 4.6(c) sélectionnée, alors :

$$\begin{aligned} m2_{sel} = 1 \quad &\Rightarrow \quad m2_{start} = e1_{start} = e2_{start} = e8_{start} = e12_{start} = e11_{start} \\ &\wedge \quad m2_{start} + m2_{delay} \leq e6_{start} \\ &\wedge \quad m2_{start} + m2_{delay} \leq e19_{start} \end{aligned}$$

Si l'occurrence m_1 est sélectionnée en même temps, on a donc :

$$\begin{aligned} m1_{sel} = 1 \quad &\Rightarrow \quad m1_{start} = e3_{start} = e4_{start} = e5_{start} = e6_{start} = e9_{start} \\ &\wedge \quad m1_{start} + m1_{delay} \leq e11_{start} \\ &\wedge \quad m1_{start} + m1_{delay} \leq e18_{start} \end{aligned}$$

Cette solution n'est pas valide car elle conduit à la contradiction suivante :

$$e11_{start} + m2_{delay} \leq e6_{start} \wedge e6_{start} + m1_{delay} \leq e11_{start}$$

indiquant que $e6$ doit être disponible avant $e11$ et en même temps que $e11$ doit être disponible avant $e6$. Par conséquent, si l'occurrence $m2$ est sélectionnée, alors l'occurrence $m1$ ne peut pas être sélectionnée, et vice versa.

Cette contrainte permet à la fois d'éviter ces situations d'interdépendence et empêche la sélection des occurrences non-convexes. Par exemple, si l'occurrence m_7 dans la figure 4.4 est sélectionnée, alors :

$$\begin{aligned} m7_{sel} = 1 \quad &\Rightarrow \quad m7_{start} = e7_{start} = e8_{start} = e9_{start} = e10_{start} \\ &\wedge \quad m7_{start} + m7_{delay} \leq e11_{start} \\ &\wedge \quad m7_{start} + m7_{delay} \leq e12_{start} \end{aligned}$$

Cette solution n'est pas valide car au même moment, l'occurrence $m6$ doit être sélectionnée et cela conduit à la contradiction $e10_{start} \leq e12_{start}$ indiquant que la sortie doit être disponible avant l'entrée. Par conséquent, l'occurrence $m7$ ne peut pas être sélectionnée pour le graphe d'application de la figure 4.4.

4.1.3 Modélisation du temps d'exécution d'une occurrence

L'étape de sélection d'instructions vise deux modèles d'architecture différents, appelés modèle A, et modèle B. Ces deux modèles sont brièvement présentés ici et sont détaillés dans le chapitre 5, page 110.

Le modèle A est un modèle d'architecture simple. Toutes les données sont sauvegardées dans la file de registres du processeur. Tous les opérandes nécessaires à l'exécution d'une instruction spécialisée sont envoyés vers l'extension au lancement de l'instruction et tous les résultats sont récupérés à la fin de l'exécution de l'instruction. Lorsque le nombre d'opérandes d'entrée ou de sortie est supérieur aux capacités architecturales (par exemple,

2 entrées et 1 sortie pour un NiosII), alors des cycles additionnels de transfert de données sont nécessaires.

Dans le modèle B, l'extension du processeur possède sa propre file de registres pour sauvegarder les données. Les instructions de base du processeur, représentées par des motifs composés d'un seul nœud, utilisent la file de registres du processeur comme dans le modèle A mais l'extension peut sauvegarder ses données dans ses propres registres. Les résultats produits par l'exécution d'une occurrence et utilisés par le processeur sont sauvegardés dans les registres du processeur. Dans ce modèle B, le nombre de cycles additionnels nécessaires au transfert des données est réduit mais sa complexité est plus élevée puisque le nombre de registres externes et de connexions externes est plus grand.

Le délai d'une occurrence, m_{delay}, exprimé en nombre de cycles du processeur, est la somme de trois composantes, comme défini par l'équation (4.7).

$$m_{delay} = \delta_{in_m} + \delta_m + \delta_{out_m} \tag{4.7}$$

où δ_m est le temps d'exécution de l'occurrence m, δ_{in_m} représente le temps de lecture des opérandes d'entrée pour l'occurrence m et δ_{out_m} le temps d'écriture des résultats.

Le temps d'exécution d'une occurrence (δ_m) est le même pour les deux architectures mais le temps de transfert est différent.

Pour connaître le temps de transfert, il faut calculer le nombre de données à transférer, et le diviser par le nombre de données transférables par cycle, puis conserver l'arrondi supérieur. Par ailleurs, au lancement d'une instruction spécialisée, il est parfois possible d'envoyer et de recevoir des données, ce qui permet de transférer des données sans pénalité. On définit donc un nombre de cycles de transfert sans pénalité qui va être retranché pour trouver le nombre de cycles de transfert réel. Autrement dit, pour connaître le nombre de cycles de pénalité, il faut :

1. compter le nombre de données à transférer et le diviser par le nombre de données transférables par cycle,
2. conserver l'arrondi supérieur,
3. retrancher le nombre de cycles de transfert sans pénalité.

Pour déterminer le nombre de données à transférer en entrée, il s'agit simplement de compter le nombre de nœuds prédécesseurs directs de l'occurrence. Pour déterminer le nombre de données à transférer en sortie, il faut compter le nombre de nœuds terminaux de l'occurrence. Un nœud terminal est un nœud dont au moins une des sorties est connectée à un nœud qui n'appartient pas à l'occurrence.

Pour le modèle A, le temps de transfert des données en lecture est défini par l'équation 4.8, et en écriture par l'équation 4.9.

$$\delta_{in_m} = \left\lceil \frac{|\,pred(m)\,|}{in_PerCycle} \right\rceil - \Delta_{in} \tag{4.8}$$

où :

- $|\,pred(m)\,|$: le nombre de nœuds prédécesseurs directs de l'occurrence m dans le graphe G,
- $in_PerCycle$: le nombre de registres lus par cycle du processeur,
- Δ_{in} : le nombre de cycles de transfert de données en entrée sans pénalité.

$$\delta_{out_m} = \left\lceil \frac{|\,last(m)\,|}{out_PerCycle} \right\rceil - \Delta_{out} \tag{4.9}$$

où :

- $|\,last(m)\,|$: le nombre de nœuds terminaux de l'occurrence m,
- $out_PerCycle$: le nombre de registres écrits par cycle du processeur,
- Δ_{out} : le nombre de cycles de transfert des données en sortie sans pénalité.

Pour l'exemple d'un processeur NIOSII, $in_PerCycle = 2$, $out_PerCycle = 1$, $\Delta_{in} = 1$, et $\Delta_{out} = 1$. Si le temps d'exécution d'une occurrence est $\delta_m = 1$, $|\,pred(m)\,| = 2$ et $|\,last(m)\,| = 1$, alors le délai de l'occurrence est $m_{delay} = 1$.

Pour le modèle B, le temps de transfert des données est variable. Le temps de transfert en lecture est défini par les équations (4.10)-(4.11).

$$IN = \sum_{n \in pred1(m)} n_{sel} \tag{4.10}$$

$$\delta_{in_m} = \left\lceil \frac{IN}{in_PerCycle} \right\rceil - \Delta_{in} \tag{4.11}$$

où :

- $pred1(m)$: l'ensemble des occurrences exécutées par le processeur qui sont des prédécesseurs de l'occurrence m dans le graphe G,
- n_{sel} : une variable qui vaut 1 si l'occurrence n est sélectionnée et 0 sinon.

La valeur de la variable IN est égale au nombre d'opérations de lecture dans la file de registres du processeur.

Le temps de transfert en écriture pour le modèle B est défini par les équations (4.12) à (4.14).

$$\forall n \in last(m) : \sum_{m_1 \in succ1(n)} m_{1_{sel}} > 0 \Leftrightarrow B_n = 1 \tag{4.12}$$

$$OUT = \sum_{n \in last(m)} B_n \tag{4.13}$$

$$\delta_{out_m} = \left\lceil \frac{OUT}{out_PerCycle} \right\rceil - \Delta_{out} \tag{4.14}$$

où :

- $last(m)$: l'ensemble des nœuds terminaux de l'occurrence m,
- $succ1(n)$: l'ensemble des occurrences à un nœud successeurs du nœud n dans le

graphe G,

- $m_{1_{sel}}$: une variable qui vaut 1 si l'occurrence m_1 est sélectionnée et 0 sinon,
- B_n : une variable qui vaut 1 si au moins une des occurrences de l'ensemble $succ1(m)$ est sélectionnée.

La valeur de la variable OUT est égale au nombre d'opérations d'écriture dans la file de registres du processeur. En d'autres termes, il s'agit de compter le nombre de données à transférer en provenance du processeur et vers le processeur, puis calculer le nombre de cycles nécessaires pour transférer ces données.

4.1.4 Résultats d'expérimentation

Le tableau 4.1 montre les résultats obtenus pour des motifs sous contraintes (2 entrées et 1 sortie), pour les deux modèles d'architecture. La première colonne indique les algorithmes considérés. La deuxième colonne montre le nombre de nœuds des graphes flots de données considérés et la troisième colonne donne le nombre de cycles nécessaires pour exécuter ces flots de données par un processeur NiosII dans sa version dite « rapide ». Le signe « - » dans le tableau indique qu'aucun motif respectant les contraintes n'a été trouvé. Les résultats montrent que même pour des contraintes aussi sévères, le facteur d'accélération peut atteindre 2,25 pour une *idct* pour le modèle d'architecture B.

Le tableau 4.2 montre les résultats obtenus pour des motifs sous contraintes à 4 entrées et 2 sorties, pour les deux modèles d'architecture. Le facteur d'accélération atteint 3,35 pour l'algorithme SHA pour le modèle d'architecture B. En moyenne, le facteur d'accélération est de 2,3 pour des motifs à 4 entrées et 2 sorties pour le modèle d'architecture B.

Pour les deux tableaux, les autres contraintes sont un nombre de 10 nœuds maximum par motif et un chemin critique de 15 ns, ce qui correspond à 3 cycles d'un processeur NiosII dans sa déclinaison « rapide » cadencé à 200 MHz.

			2 entrées / 1 sortie											
					modèle A					**modèle B**				
Algorithmes	nœuds	cycles	coef	identifiés	sélectionnés	couverture	cycles	accélération	optimalité	sélectionnés	couverture	cycles	accélération	optimalité
JPEG BMP Header	34	34	0	6	2	82%	14	2,42	✓	2	82%	14	2,42	✓
JPEG Downsample	66	78	0	5	2	19%	68	1,14	✓	2	19%	68	1,14	✓
JPEG IDCT	250	302	0,5	28	10	76%	214	1,41		10	76%	134	2,25	
EPIC Collapse	278	287	0	11	8	68%	165	1,74		8	68%	165	1,74	
BLOWFISH encrypt	201	169	0,5	11	3	74%	90	1,87		3	74%	90	1,87	
SHA transform	53	57	0	5	3	64%	28	2,03	✓	3	64%	28	2,03	✓
MESA invert matrix	152	334	0,5	2	2	10%	320	1,04	✓	2	10%	320	1,04	✓
FIR unrolled	67	131	0	3	2	9%	126	1,04	✓	2	9%	126	1,04	✓
FFT	10	18	0	0	-	-	-	-		-	-	-	-	
Moyenne						50%		**1,5**			50%		**1,7**	

Tab. 4.1 – Résultats obtenus pour les benchmarks MediaBench et MiBench pour un processeur NiosII avec l'environnement *DURASE* sous contraintes de 2 entrées et 1 sortie.

Les tableaux 4.1 et 4.2 indiquent les algorithmes pour lesquels l'optimalité de la sélection d'instructions a été prouvée. Pour les autres algorithmes, les résultats corres-

Algorithmes	nœuds	cycles	4 entrées / 2 sorties modèle A coef	identifiés	sélectionnés	couverture	cycles	accélération	optimalité	modèle B sélectionnés	couverture	cycles	accélération	optimalité
JPEG BMP Header	34	34	0	66	2	88%	12	2,83	✓	3	88%	12	2,83	✓
JPEG Downsample	66	78	0	49	4	95%	44	1,77	✓	4	100%	35	2,22	✓
JPEG IDCT	250	302	0,5	254	13	83%	141	2,36		15	89%	112	2,69	
EPIC Collapse	278	287	0	111	11	71%	156	1,83		14	71%	159	1,83	
BLOWFISH encrypt	201	169	0	153	8	90%	81	2,08		7	92%	73	2,31	
SHA transform	53	57	0	48	8	98%	22	2,59	✓	6	95%	17	3,35	✓
MESA invert matrix	152	334	0,5	53	9	65%	262	1,27		9	65%	243	1,37	
FIR unrolled	67	131	1	10	2	94%	98	1,30	✓	2	97%	67	1,95	✓
FFT	10	18	0	12	2	60%	10	1,80	✓	2	60%	10	1,80	✓
Moyenne						83%		**2**			84%		**2,3**	

TAB. 4.2 – Résultats obtenus pour les benchmarks MediaBench et MiBench pour un processeur NIOSII avec l'environnement *DURASE* sous contraintes de 4 entrées et 2 sorties.

pondent aux meilleures solutions trouvées au terme d'une temporisation (*time-out*) de 10 minutes. On remarque que l'optimalité de la sélection est prouvée pour des graphes allant jusqu'à une centaine de nœuds. À partir d'une centaine de nœuds, la solution peut être optimale mais elle n'est pas prouvée. Nous avons augmenté la temporisation jusqu'à plusieurs heures pour essayer d'obtenir des meilleurs résultats ou bien la preuve de l'optimalité des solutions, mais sans succès. On en déduit que le solveur de contraintes aboutit rapidement à une solution, peut être sous-optimale, mais la preuve de l'optimalité, qui nécessite un parcours exhaustif de l'arbre de recherche, est longue à obtenir.

La figure 4.7 présente les accélérations obtenues selon le modèle d'architecture et le nombre d'entrées et de sorties. La figure montre bien l'impact de la relaxation du nombre d'entrées/sorties et du modèle d'architecture. Les meilleurs résultats sont sans surprise obtenus pour le modèle d'architecture B pour des motifs à 4 entrées et 2 sorties. On observe également que pour l'algorithme FFT ou $EPIC\,collapse$ par exemple, le modèle d'architecture B n'améliore pas les performances. Cela signifie qu'il n'y a pas de connexion directe entre deux instructions spécialisées et que les données transitent forcément par le processeur.

Les tableaux 4.1 et 4.2 indiquent également le pourcentage de couverture du graphe par des instructions spécialisées. La figure 4.8 met en évidence la différence de couverture des graphes selon l'objectif visé, soit la couverture maximale, soit le temps d'exécution minimal. On remarque que la couverture est plus faible lorsque l'objectif est de minimiser le temps d'exécution que lorsque l'objectif est de minimiser le nombre d'instructions exécutées par le processeur. Ces résultats montrent qu'une instruction spécialisée n'apporte pas forcément de gain lorsque les pénalités dues au transfert de données entre le processeur et son extension sont prises en compte.

Optimisation sur les pénalités de transfert des données. Une même donnée en entrée peut être utilisée plusieurs fois par la même instruction spécialisée. Afin de limiter

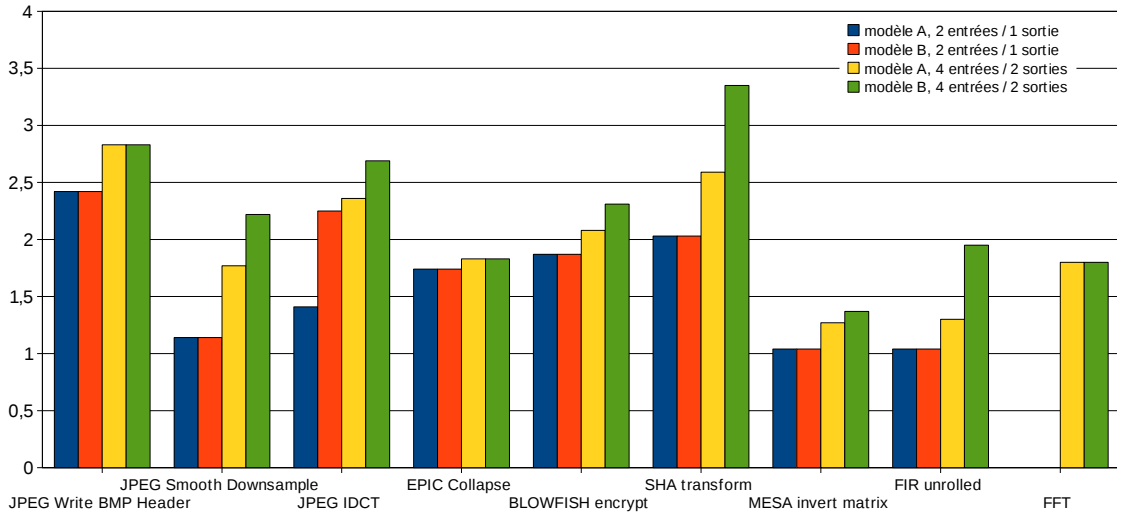


FIG. 4.7 – Impact des modèles d'architecture et de la relaxation du nombre d'entrées/sorties

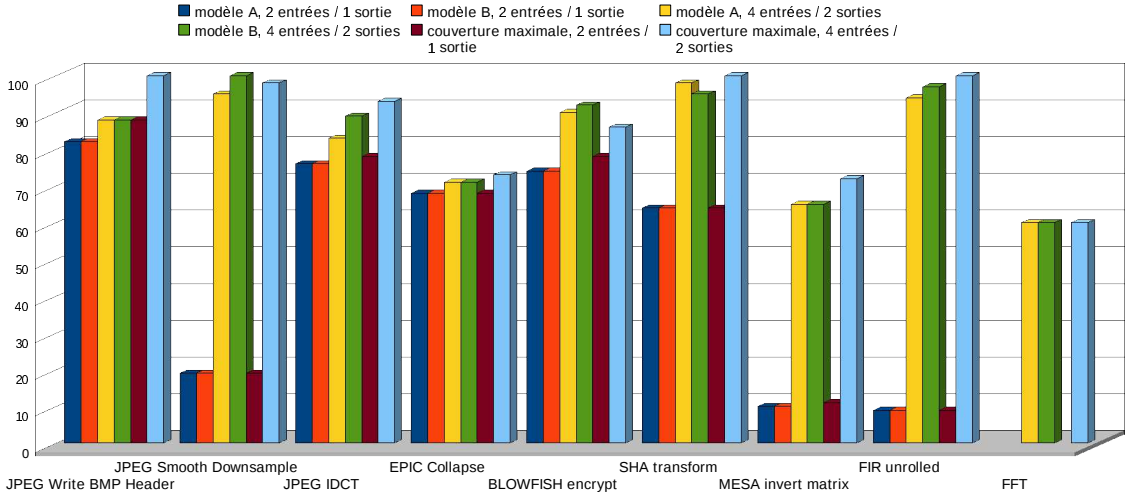


FIG. 4.8 – Différence de couverture de graphe selon l'objectif

le nombre de transferts de données en entrée, il faut transférer une seule fois cette donnée du processeur vers l'extension. Cet aspect est bien pris en compte dans notre modélisation du temps d'exécution d'une occurrence. Par ailleurs, une même donnée peut également être utilisée par plusieurs instructions spécialisées. L'idée est de pouvoir transférer une seule fois la donnée pour toutes les instructions qui l'utilisent.

Pour cela, il faut modifier la modélisation du temps d'exécution d'une occurrence et calculer le coût des transferts de manière globale. Ainsi, d'après l'équation 4.2 :

$$\begin{aligned}
\text{ExecutionTime} &= \sum_{m \in M} m_{sel} \cdot m_{delay} && (4.15)\\
&= \sum_{m \in M} m_{sel} \cdot (\delta_{in_m} + \delta_m + \delta_{out_m}) && (4.16)\\
&= \underbrace{\sum_{m \in M} m_{sel} \cdot \delta_{in_m}}_{\text{input overhead}} + \sum_{m \in M} m_{sel} \cdot \delta_m + \underbrace{\sum_{m \in M} m_{sel} \cdot \delta_{out_m}}_{\text{output overhead}} && (4.17)
\end{aligned}$$

Le temps d'exécution total est donc le temps d'exécution de chaque occurrence sélectionnée *plus* le surcoût lié au transfert des données en entrée (*input overhead*) et en sortie (*output overhead*).

Cependant, le calcul du surcoût des transferts des données est dépendant de Δ_{in} et Δ_{out} qui permettent de transférer des données sans pénalité. Le coût du transfert devient dépendant du *moment* où la donnée est transférée. Prenons l'exemple d'une même donnée en entrée de deux instructions spécialisées, comme illustré par la figure 4.9 où le résultat de l'instruction `add` est envoyé vers les instructions spécialisées `custom 1` et `custom 2`. L'instruction spécialisée `custom 2` prend en entrée deux données, qui peuvent être transférées au lancement de l'instruction. Par contre l'instruction spécialisée `custom 1` prend en entrée trois données et nécessite un cycle additionnel de transfert de données. Si l'instruction `custom 2` est exécutée avant `custom 1`, alors il n'y a pas de pénalité de transfert car les deux données en entrée de cette instruction peuvent être transférées au lancement de l'instruction. Par contre si `custom 1` est exécutée avant `custom 2`, alors il faut un cycle de plus pour transférer la donnée puisque l'instruction `custom 1` possède trois entrées. Pour déterminer les cycles additionnels de transfert de données, il faut savoir quelle instruction est exécutée à quel moment.

La prise en compte de manière globale des pénalités de transfert des données nécessite des informations d'ordonnancement. Cette optimisation n'est donc possible que si l'ordonnancement et la sélection d'instructions sont résolus conjointement.

4.2 Ordonnancement

A ce stade du processus, les instructions qui minimisent le temps d'exécution du graphe ont été sélectionnées. On a pu trouver qu'il est possible d'exécuter le graphe en un temps donné mais on n'a pas encore déterminé quelle instruction exécuter à quel moment. Ceci

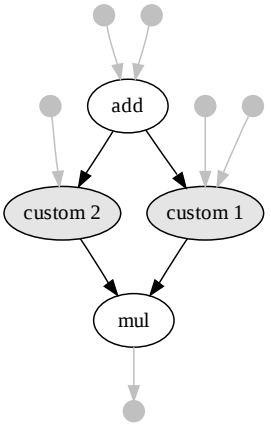


FIG. 4.9 – Envoi de la même donnée vers plusieurs instructions spécialisées

est fait par une phase d'ordonnancement. Cette section décrit les étapes de placement et d'ordonnancement.

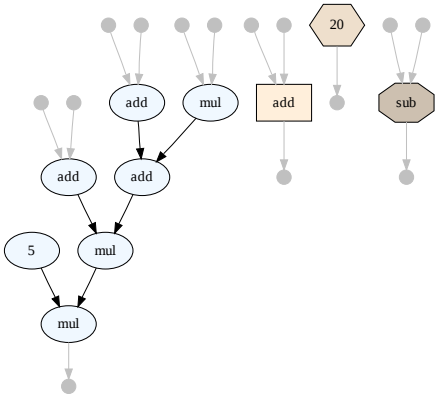
4.2.1 Placement

Le placement (*binding*) a pour objectif d'attribuer les instructions aux unités fonctionnelles. Dans le cas de nos deux modèles d'architecture, nous possédons deux unités fonctionnelles : l'UAL du processeur, capable d'exécuter les instructions de base du processeur, et l'extension, responsable de l'exécution des instructions spécialisées. Le placement est immédiat puisqu'on sait qu'une occurrence à un nœud est forcément exécutée par l'UAL du processeur et qu'une occurrence à plusieurs nœuds est forcément sur l'extension.

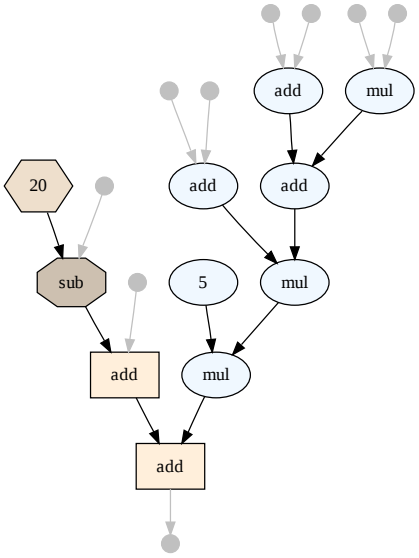
4.2.2 Ordonnanceur

Le rôle de l'ordonnanceur est de répondre à la question : quelle instruction exécuter et quand ? Dans le contexte où les instructions sont représentées par des nœuds dans un graphe d'application, l'ordonnanceur est chargé de déterminer la date d'exécution de chaque nœud. L'ordonnancement s'effectue sur un *graphe réduit*. Un graphe réduit est un graphe dans lequel toutes les occurrences sont modélisées par un seul nœud. La figure 4.10 montre un exemple d'un graphe couvert, des motifs sélectionnés, et du graphe réduit créé à partir du graphe couvert. Les occurrences à un nœud restent inchangées. Les occurrences qui couvrent plusieurs nœuds sont condensées en un seul nœud. Dans le cas du NIOSII, nous réduisons également les nœuds avec valeur immédiate, comme par exemple la soustraction avec une valeur immédiate de la figure 4.10(b), qui est réduite en `subi`. Soit ce nouveau graphe $G' = (N', E')$.

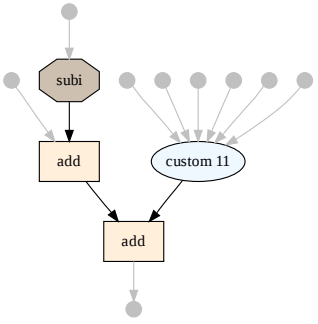
Nous pouvons appliquer sur ce graphe réduit un ordonnancement par liste où les nœuds



(a) Motifs sélectionnés



(b) Graphe couvert



(c) Graphe réduit

FIG. 4.10 – Motifs sélectionnés, graphe couvert et graphe réduit

représentent les tâches et dans lequel on dispose d'une seule ressource. Cela revient à placer tous les nœuds du graphe dans une colonne de largeur 1.

4.2.3 Ordonnancement avec recalage d'instructions

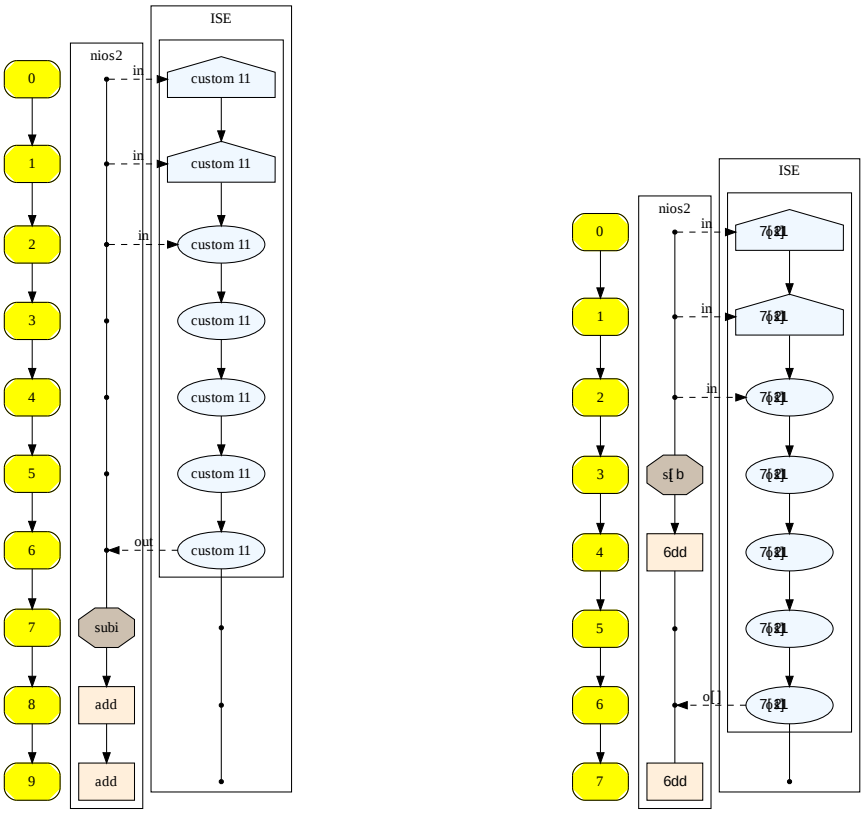
Lorsque le temps d'exécution d'une occurrence (δ_m) sur l'extension est suffisamment long[1], il est possible d'exécuter en parallèle une instruction sur le processeur en attendant la fin de l'exécution de l'occurrence. Il faut alors « tromper » le processeur en lui faisant croire que l'exécution de l'instruction spécialisée ne nécessite qu'un cycle d'exécution. Ainsi, le processeur garde la main.

Le lancement des instructions est exclusif mais leur exécution peut être parallèle. Nous proposons un nouvel ordonnancement qui prend en compte cette opportunité, toujours grâce à la programmation par contraintes. Le graphe réduit de la figure 4.10(c) montre bien le parallélisme au niveau instruction. L'idée est de pouvoir exécuter les instructions `subi` et `add` sur le processeur pendant que l'extension exécute l'instruction `custom 11`.

La figure 4.11 montre un exemple d'ordonnancement sans recalage 4.11(a) et avec recalage d'instructions 4.11(b) pour le graphe réduit de la figure 4.10(c). La première colonne symbolise les cycles. La deuxième colonne montre l'occupation du processeur (*nios2*), et la troisième colonne représente l'extension (*ISE*). Les nœuds en forme de *maison* représentent les cycles de transfert de données en entrée. On voit sur la figure 4.11(a) que pendant les cycles 3, 4 et 5, le processeur est libre. La figure 4.11(b) montre le recalage des instructions `subi` et `add` aux cycles 3 et 4 car les dépendances de données le permettent. Il faut néanmoins attendre la fin de l'instruction `custom 11` pour pouvoir exécuter la dernière instruction `add`.

Tout d'abord, nous modélisons chaque occurrence sélectionnée sous forme de rectangle (figure 4.12(c)), de largeur égale au nombre de ressources utilisées et de longueur égale au temps d'occupation des ressources. Une occurrence exécutée sur le processeur est modélisée par un rectangle de largeur 1 (car l'instruction occupe une ressource, le processeur) et de longueur égale au nombre de cycles nécessaires pour exécuter cette instruction (typiquement 1). La figure 4.12(a) montre l'exemple pour l'instruction *mul* d'un NiosII (version rapide) dont le temps d'exécution est de 1 cycle et le temps de latence est de 2 cycles. Le rectangle est donc de largeur 1 et de longueur 1. Une occurrence exécutée sur l'extension est modélisée par un double rectangle de largeur 1. Le premier rectangle correspond au nombre de cycles nécessaires à la communication pour les entrées plus le premier cycle toujours nécessaire pour le lancement de l'instruction ($L_{in_m} = \delta_{in_m} + 1$). Le second rectangle correspond au nombre de cycles nécessaires à la communication pour les sorties ($L_{out_m} = \delta_{out_m} + \delta_I$), où δ_I est défini par l'équation (4.18). Ces deux rectangles sont séparés d'une distance constante ($d_{in_m/out_m} = \delta_m - 1 - \delta_I$). L'équation (4.19) montre bien

[1] strictement supérieur à 2 cycles pour le modèle d'architecture A, supérieur à 1 cycle pour le modèle B dans le cas où le processeur ne lit pas le résultat de l'instruction



(a) Sans le recalage d'instructions (b) Avec le recalage d'instructions

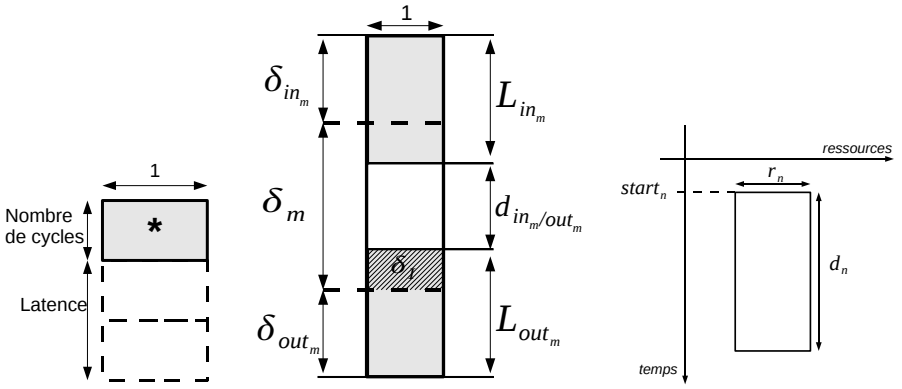
FIG. 4.11 – Exemple d'ordonnancement du graphe de la figure 4.10(c) et de placement sur le modèle d'architecture B

l'équivalence avec l'équation (4.7), illustrée par la figure 4.12(b).

$$\delta_I = \begin{cases} 1 & \text{si } \delta_m > 1 \\ 0 & \text{sinon} \end{cases} \tag{4.18}$$

$$m_{delay} = \delta_{in_m} + \delta_m + \delta_{out_m} = L_{in_m} + d_{in_m/out_m} + L_{out_m} \tag{4.19}$$

Dans notre architecture cible, nous disposons d'une seule ressource, le processeur. Le processeur ne peut exécuter qu'une seule instruction à la fois (soit un lancement, soit une communication), chaque instruction étant modélisée par un rectangle. Pour garantir l'exclusivité d'utilisation du processeur, il convient d'assurer que les rectangles ne se chevauchent pas dans une colonne de largeur 1. Nous voulons donc obtenir un placement optimal de tous les rectangles, sans chevauchement, sur une colonne de largeur égale à 1 tout en respectant les dépendances de données. Pour cela, nous utilisons une contrainte cumulative. La contrainte cumulative permet d'assurer qu'à chaque instant, le total des



(a) Instruction *mul* sur un NiosII (b) Instruction spécialisée (c) Un rectangle pour la contrainte cumulative

FIG. 4.12 – Modélisation des instructions sous forme de rectangles

ressources utilisées ne dépasse pas une certaine limite. La contrainte cumulative prend en paramètre un ensemble de rectangles et une limite sur le nombre de ressources disponibles. Tous les rectangles sont ajoutés à cette contrainte cumulative et nous fixons le nombre de ressources à 1, comme défini dans l'équation 4.20. Enfin, nous définissons la fonction de coût à minimiser pour cet ordonnancement (4.23) et nous imposons les contraintes sur les dépendances de données (4.21) et (4.22).

$$\forall n \in N' : Cumulative(\{\ldots\{start_n^i\}, \{r_n^i\}, \{d_n^i\}\ldots\}, 1)\ i \in R_n \tag{4.20}$$

où R_n est l'ensemble des rectangles modélisant un nœud n.

$$\forall n \in N' : start_n + delay_n = end_n \tag{4.21}$$

$$\forall (n,m) \in E' : end_n \leq start_m \tag{4.22}$$

$$\text{ExecutionTime } = max(end_n \forall n \in N') \tag{4.23}$$

4.2.4 Optimisation sur les transferts de données

Nous avons vu à la section 4.1.4 que lorsqu'une même donnée est utilisée par plusieurs instructions spécialisées, il faut transférer une seule fois cette donnée pour réduire le nombre de cycles de pénalité dus au transfert de données. Cette optimisation qui nécessite des informations d'ordonnancement est maintenant possible. Pour prendre en compte cette optimisation, il suffit de modifier la modélisation du nombre de cycles nécessaires à la communication pour les entrées. Jusqu'ici, ce nombre de cycles était fixe, il devient variable en fonction du contexte. Le temps d'exécution de l'instruction et le nombre de cycles pour la communication en sortie restent fixes.

Nous proposons d'illustrer cette optimisation à travers l'exemple du graphe réduit de la figure 4.13 pour un processeur NiosII. Dans ce graphe, les occurrences sont réduites

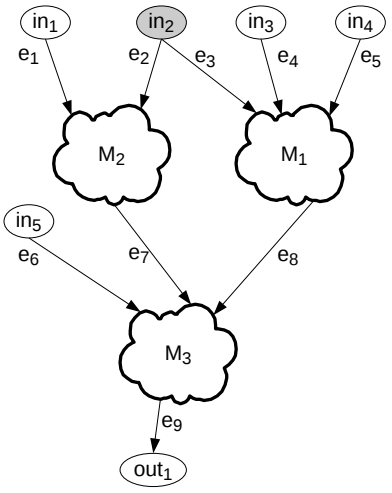


FIG. 4.13 – Exemple d'optimisation sur le transfert de données

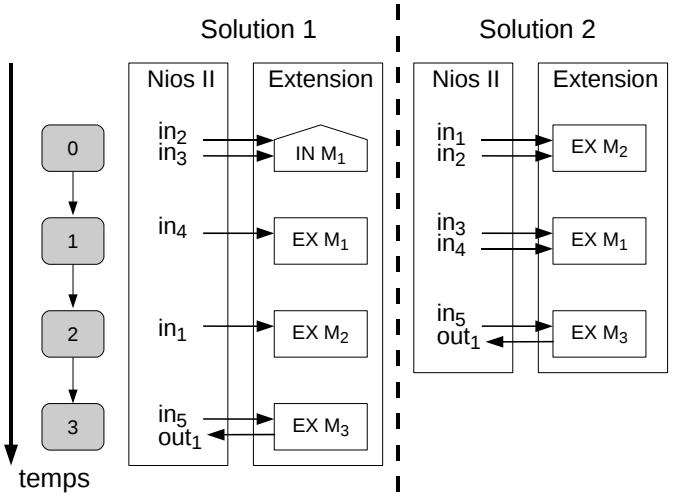


FIG. 4.14 – Ordonnancements possibles du graphe de la figure 4.13

en un seul nœud. Le graphe est composé de trois nœuds, exécutés par l'extension, les occurrences M_1, M_2, et M_3. Les nœuds *in* représentent les données en provenance de la file de registres du processeur. Dans cet exemple, le nœud in_2, mis en évidence par une zone grisée, est utilisé à la fois par le nœud M_1 et le nœud M_2. Le nœud out_1 représente une donnée à envoyer vers la file de registres du processeur.

La figure 4.14 montre deux ordonnancements possibles de ce graphe. La solution 1 illustre le cas où l'occurrence M_1 est exécutée avant l'occurrence M_2. Dans ce cas, un cycle additionnel de transfert de données est nécessaire puisque l'occurrence M_1 possède trois entrées. La solution 2 illustre le cas où l'occurrence M_2 est exécutée avant l'occurrence M_1. Dans ce cas, aucun cycle additionnel de transfert de données n'est nécessaire puisque l'occurrence M_2 possède deux entrées et la donnée in_2 est déjà présente lors de l'exécution de M_1. Pour des raisons de clarté, l'exemple présente des instructions qui nécessitent un seul cycle pour être exécutées.

Pour pouvoir prendre en compte ces optimisations lors de l'ordonnancement, nous modifions légèrement le modèle présenté dans la section précédente. En effet, dans la section précédente, nous avons considéré un nombre de cycles additionnels de transfert de données fixe pour chaque occurrence, défini par δ_{in_m}. Ce nombre devient variable.

Dans un premier temps, nous identifions l'ensemble des nœuds *partagés*, c'est-à-dire les nœuds qui représentent une donnée utilisée par plusieurs occurrences. Nous définissons cet ensemble S. Dans notre exemple $S = \{in_2\}$. À chaque nœud appartenant à cet ensemble ($\forall n \in S$) est associée une variable n_τ indiquant si la donnée a été transférée ($n_\tau = 0$) ou non ($n_\tau = 1$). Puis nous identifions l'ensemble des occurrences qui partagent ce nœud, et notons cet ensemble M_n où n est le nœud partagé. Dans notre exemple $M_{in_2} = \{M_1, M_2\}$.

Dans un deuxième temps, pour chaque nœud, nous partageons l'ensemble des nœuds prédécesseurs en deux ensembles. Le premier ensemble, appelé $predst(n)$ définit l'ensemble

des prédécesseurs *statiques* du nœud n, c'est-à-dire l'ensemble des nœuds qui ne sont pas partagés et qui nécessitent un transfert. Le deuxième ensemble, appelé $preddyn(n)$ définit l'ensemble des prédécesseurs *dynamiques* du nœud n, c'est-à-dire l'ensemble des nœuds dont le transfert est conditionné par l'ordre d'exécution des nœuds. De manière formelle, nous avons (équation 4.24) :

$$pred(n) = predst(n) \cup preddyn(n) \wedge preddyn(n) = pred(n) \cap S \tag{4.24}$$

Enfin, nous modifions la définition de δ_{in_m}, qui permet de modéliser le nombre de cycles additionnels de transfert de données pour une occurrence m. L'idée est de savoir si une occurrence qui utilise un nœud partagé est la première à être exécutée parmi celles qui utilisent ce nœud. Dans ce cas, il faut prendre en compte le transfert de données. Nous nous inspirons du calcul des équations 4.10 et 4.11 et définissons δ_{in_m}, le temps de transfert de données en entrée pour chaque nœud m, par les équations (4.25 à 4.27).

$$\forall m_s \in M_n : start_m \leq start_{m_s} \Leftrightarrow n_\tau = 1 \tag{4.25}$$

$$IN_m = |\, predst(m) \,| + \sum_{n \in preddyn(m)} n_\tau \tag{4.26}$$

$$\delta_{in_m} = \left\lceil \frac{IN_m}{in_PerCycle} \right\rceil - \Delta_{in} \tag{4.27}$$

où :

- n_τ : une variable qui vaut 0 si la donnée représentée par le nœud n est déjà transférée, 1 sinon ;
- $in_PerCycle$: le nombre de registres lus par cycle du processeur ;
- Δ_{in} : le nombre de cycles de transfert de données en entrée sans pénalité ;
- $start_m$: l'instant de début du nœud m, défini à l'équation (4.21).

	Solution 1	Solution 2
$\delta_{in_{M_1}}$	1	0
$\delta_{in_{M_2}}$	0	0

TAB. 4.3 – Nombre de transferts de données en entrée pour les deux ordonnancements de la figure 4.14

Reprenons l'exemple de la figure 4.13, et les solutions de la figure 4.14 pour un processeur NIOSII, où $in_PerCycle = 2$ et $\Delta_{in} = 1$. Le tableau 4.3 résume les différentes valeurs du nombre de transferts de données en entrée pour les deux solutions d'ordonnancement. Il apparaît clairement que la solution 1 nécessite un cycle de pénalité alors que la solution 2 n'en nécessite aucun.

4.2.5 Résultats d'expérimentation

4.2.5.1 Résultats avec et sans recalage

Pour qu'il y ait opportunité de recalage d'instructions, il faut que les instructions spécialisées nécessitent plusieurs cycles d'exécution, et en même temps il faut une instruction à exécuter sur le processeur. De plus, le recalage est possible si il n'y a pas de dépendance de données entre les instructions exécutées en parallèle. Le recalage ne se prête pas à tout type de graphe, il convient aux graphes qui présentent beaucoup de parallélisme. Afin d'obtenir un potentiel de recalage, nous avons intentionnellement généré et conservé des *gros* motifs, c'est à dire des motifs qui nécessitent au moins deux cycles d'exécution. Nous avons retenu les algorithmes qui présentent des flots de données importants (graphes de plus de 50 nœuds) et un minimum de parallélisme.

La figure 4.15 montre les résultats obtenus pour cinq algorithmes. La première colonne indique le nombre de cycles nécessaires à un NiosII sans extension pour exécuter les graphes flot de données considérés. La deuxième colonne correspond au nombre de cycles nécessaires à un NiosII avec extension lors d'un ordonnancement par liste. La troisième colonne représente le nombre de cycles pour un NiosII avec une extension sans recalage d'instructions et la quatrième, avec recalage d'instructions. On remarque que notre ordonnancement, même sans recalage, est meilleur que l'ordonnancement par liste. L'ordonnancement par recalage d'instructions permet d'économiser des cycles sur l'ensemble des algorithmes, sauf l'algorithme $DES3$. Pour tous les algorithmes testés, le problème de l'ordonnancement avec recalage d'instructions est résolu de manière optimale.

Algorithmes	Nombre de cycles		Gain
	ordonnancement par liste	ordonnancement avec recalage	
IDCT	152	97	1,57
Cast 128	299	197	1,52
DES3	81	69	1,17
Blowfish	280	173	1,62
Invert Matrix	287	137	2,09

Tab. 4.4 – Gain de l'ordonnancement par recalage d'instructions par rapport à l'ordonnancement par liste

Le tableau 4.4 montre le nombre de cycles nécessaires pour exécuter les différents algorithmes selon l'ordonnancement par liste ou l'ordonnancement par recalage d'instructions. La dernière colonne du tableau 4.4 montre le gain qu'apporte l'ordonnancement par recalage d'instructions par rapport à l'ordonnancement par liste. Les résultats montrent que pour l'algorithme $Invert\,Matrix$, le gain grimpe à 2,09.

4.2.5.2 Discussion

Notre approche d'ordonnancement par la programmation par contraintes est radicalement différente de celle présentée dans le système $UPaK$ [207, 206]. En effet, les deux approches diffèrent dans la modélisation du temps d'exécution d'une occurrence. Le système

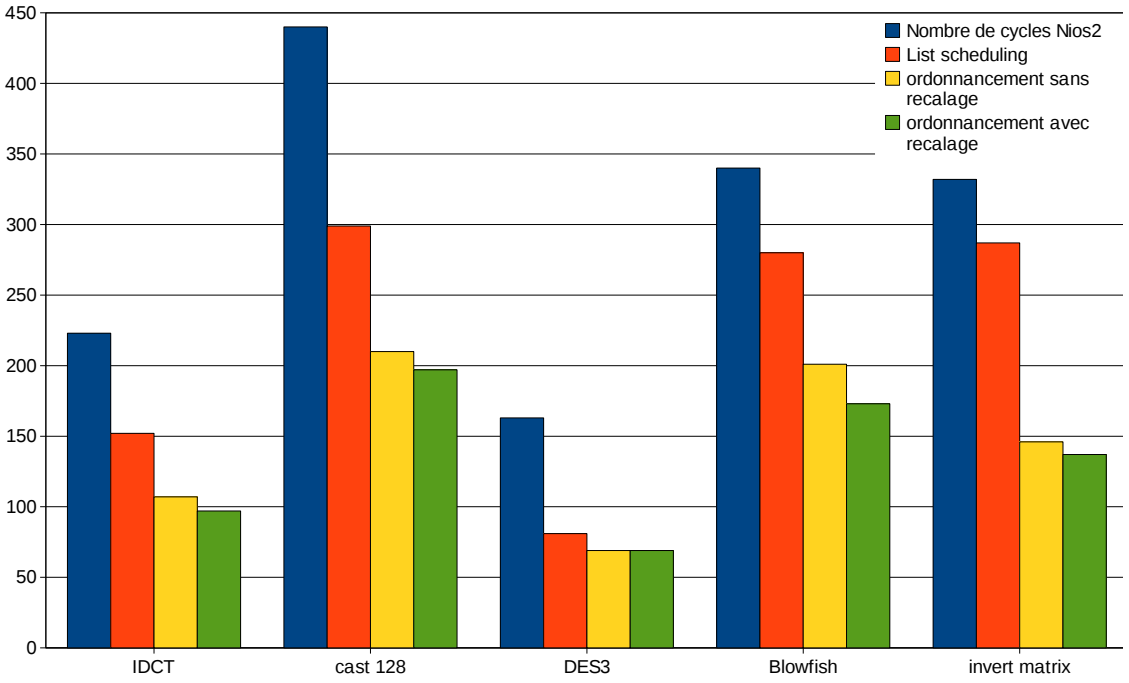


FIG. 4.15 – Résultats avec et sans recalage d'instructions

UPaK permet de résoudre simultanément la sélection d'instructions, le *binding*, et l'ordonnancement. Pour cela, la modélisation du temps d'exécution d'une occurrence s'appuie sur la topologie du graphe. Dans le cas où un nœud d'une occurrence possède à la fois des dépendances sur des nœuds appartenant à la même occurrence et sur des nœuds n'appartenant pas à l'occurrence, l'introduction de *dummy nodes* (nœuds factices) est nécessaire pour assurer le bon *timing* des signaux d'entrée. La figure 4.16 montre l'introduction d'un *dummy node* pour assurer que tous les liens d'un même niveau topologique possèdent le même *timing*. Cette modélisation permet également d'écarter les occurrences non convexes. Le gros inconvénient de cette technique est l'introduction d'un nombre considérable de nœuds, ce qui complexifie le problème et gêne la recherche de la solution optimale.

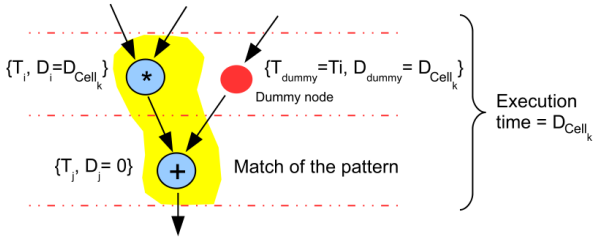


FIG. 4.16 – Introduction des *dummy nodes* dans le système *UPaK*

4.3 Conclusion

Dans ce chapitre, nous avons présenté nos méthodes de sélection d'instructions et d'ordonnancement. Ces méthodes s'appuient sur la programmation par contraintes. Le problème de sélection d'instructions est modélisé par le problème de couverture de sommets d'un graphe où tous les nœuds doivent être couverts par une occurrence et la somme des délais des occurrences est minimisée. La sélection d'instructions est résolue de manière optimale pour certains algorithmes et le facteur d'accélération estimé est de 2,3 pour des motifs à 4 entrées et 2 sorties pour le modèle d'architecture B et atteint 3,35 pour l'algorithme SHA.

Notre méthode d'ordonnancement permet d'exploiter le parallélisme de l'UAL du processeur et de l'extension pour exécuter deux instructions de manière concurrente. La preuve de l'optimalité de la solution trouvée est apportée et l'ordonnancement avec recalage d'instructions apporte un gain significatif par rapport à l'ordonnancement par liste. Pour l'algorithme $Invert\,Matrix$, le nombre de cycles estimés pour exécuter l'algorithme est divisé par deux grâce à notre technique d'ordonnancement avec recalage par rapport à l'ordonnancement par liste.

Ces techniques ont fait l'objet d'une publication à la conférence *ASAP* (*Application-specific Systems, Architectures and Processors*) [2], et à la conférence *SympA'13* (*Symposium en Architecture de machines*) [3].

Pour compléter le processus d'extension de jeu d'instructions, il reste à générer l'architecture de l'extension et générer le code applicatif qui exploite les instructions spécialisées.

5

Génération d'architecture et adaptation du code

La génération d'architecture est l'étape en charge du passage de la représentation des instructions spécialisées sous forme de motifs de calcul à leur mise en œuvre matérielle. L'adaptation du code est l'étape à l'issue de laquelle le code applicatif exploite les instructions spécialisées. Ces deux étapes constituent la *partie finale* de notre outil, ces étapes sont communément appelées *back-end* dans le monde de la compilation. Contrairement à notre méthode de génération d'instructions, qui est complètement générique, et notre méthode de sélection d'instructions, facilement *reciblable*, la partie finale vise une architecture concrète et un processeur bien précis. Pour valider à la fois l'architecture et le code adapté, il est possible de s'appuyer sur la simulation. Dans ce contexte, nous cherchons à nous raccrocher à la simulation avec SoCLib, et nous générons automatiquement le modèle SystemC des instructions spécialisées pour un processeur NiosII.

La figure 5.1 replace ce chapitre dans le flot de conception. La première partie du chapitre présente l'architecture de l'extension, les différents types d'instructions spécialisées existantes pour un NiosII et les deux modèles d'architecture considérés pour l'extension. La deuxième partie détaille notre technique d'allocation de registres par la programmation par contraintes. La troisième partie décrit la phase de génération des codes d'opérations spécialisées et de génération de la description architecturale de l'extension. Ensuite, nous expliquons comment générer le code source qui exploite les instructions spécialisées. Enfin, nous décrivons la génération des modèles des instructions spécialisées pour la simulation SystemC avec SoCLib.

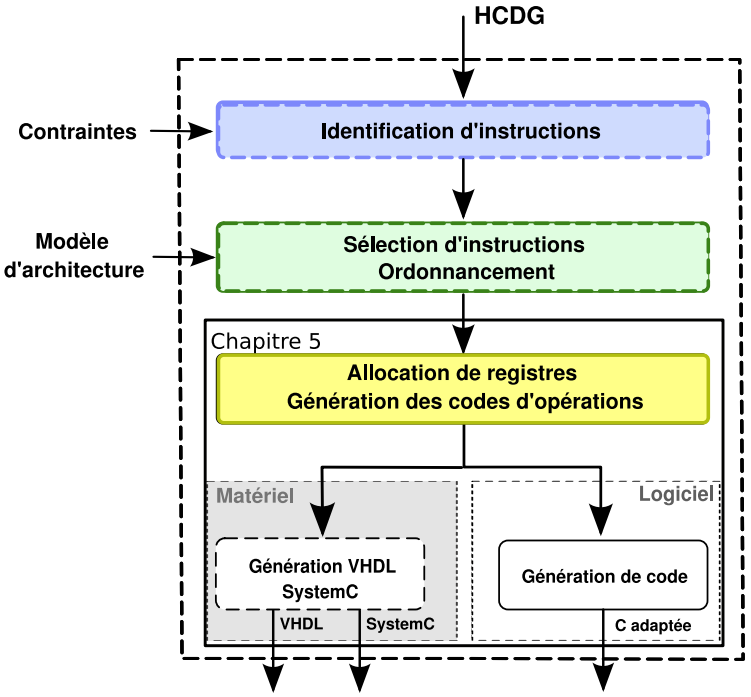


FIG. 5.1 – Le chapitre 5 présente les étapes d'allocation de registres, génération des codes d'opération, génération d'architecture et de modèles SystemC, et adaptation du code.

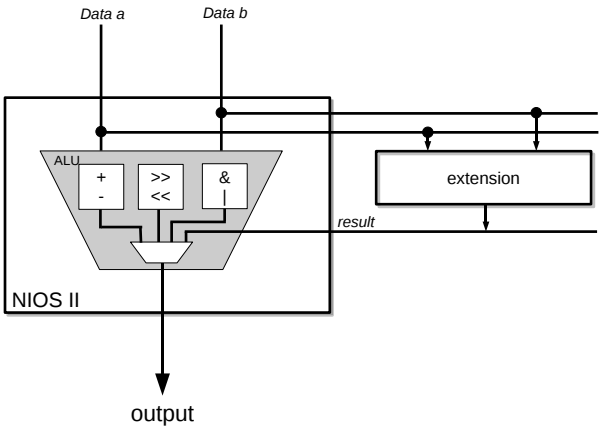


FIG. 5.2 – Un NIOSII avec son extension

5.1 Architecture de l'extension

Nous nous plaçons dans le contexte d'un processeur synthétisable de type NiosII avec une extension très fortement couplée à son chemin de données. La figure 5.2 illustre l'unité de traitement d'un processeur NiosII avec son extension. De façon identique à l'UAL du processeur, l'extension peut lire deux données en entrée symbolisées par *dataa* et *datab* et écrire une donnée en sortie dénotée *result*.

5.1.1 Différents types d'instructions spécialisées

Les instructions spécialisées du processeur NiosII sont de différents types : combinatoires, multi-cycles, étendues, ou paramétrées. Le type d'instruction détermine l'interface entre l'extension et le processeur. La figure 5.3 illustre les différents types d'instructions spécialisées du NiosII, et montre les ports nécessaires à chaque type d'instructions. La figure 5.3 montre également une interface vers de la logique externe qui offre un mécanisme d'interface spécialisée vers des ressources en dehors du chemin de données du processeur [19].

Dans le cas d'une instruction combinatoire *(Combinational)*, l'instruction spécialisée s'exécute en un cycle d'horloge et le bloc logique ne nécessite que les signaux `dataa` et `datab` en entrée et `result` en sortie.

Lorsqu'une instruction spécialisée nécessite plusieurs cycles pour s'exécuter, elle est dite *Multi-cycle* et des ports de contrôle additionnels sont requis. En particulier, les ports `clk`, `clk_en` et `reset` doivent obligatoirement être gérés. Par ailleurs, le pipeline du processeur est gelé le temps de l'exécution de l'instruction[1]. Le nombre de cycles pour exécuter une instruction multi-cycle peut être fixe ou variable. Lorsque ce nombre est fixe, il est fourni lors de la génération du processeur et le processeur attend le nombre de cycles spécifié avant de lire `result`. Lorsque le nombre de cycles est variable, la synchronisation suit un protocole de *handshaking* réalisé par le signal `start` qui détermine l'instant de début de l'instruction et le signal `done` qui signale la fin de l'instruction.

Une instruction spécialisée dite étendue *(Extended)* permet à un unique bloc logique spécialisé de mettre en œuvre plusieurs opérations différentes. Le numéro d'opération est contrôlé par l'indice `n`, codé sur 8 bits, autorisant le bloc logique à mettre en œuvre jusqu'à 256 opérations différentes. Une instruction étendue peut être combinatoire ou multi-cycle. Les instructions spécialisées étendues occupent plusieurs indices d'instructions spécialisées. Par exemple, si une instruction étendue occupe 4 indices, alors il reste $256-4 = 252$ indices disponibles pour les autres instructions spécialisées.

Une instruction paramétrée *(Internal Register File)* correspond au cas où l'extension possède sa propre file de registres. Ce type d'instruction offre la possibilité de spécifier si l'instruction spécialisée lit ses opérandes à partir de la file de registres du processeur ou bien de l'extension. En outre, l'instruction spécialisée peut écrire le résultat dans la

[1] Il faut donc « tromper » le processeur en lui faisant croire qu'une instruction est mono-cycle pour pouvoir appliquer notre ordonnancement avec recalage d'instructions

file de registres du processeur ou de l'extension. Les signaux `readra`, `readrb`, et `readrc` permettent de déterminer si l'instruction spécialisée doit utiliser la file de registres de l'extension ou bien les signaux `dataa`, `datab`, et `result`. Les ports `a`, `b`, et `c` précisent le registre de l'extension à lire ou à écrire. Ces ports ont une largeur de cinq bits, permettant d'adresser jusqu'à 32 registres. Par exemple, si le signal `readra` est désactivé, cela indique une lecture dans la file de registres de l'extension et le port `a` fournit l'indice du registre.

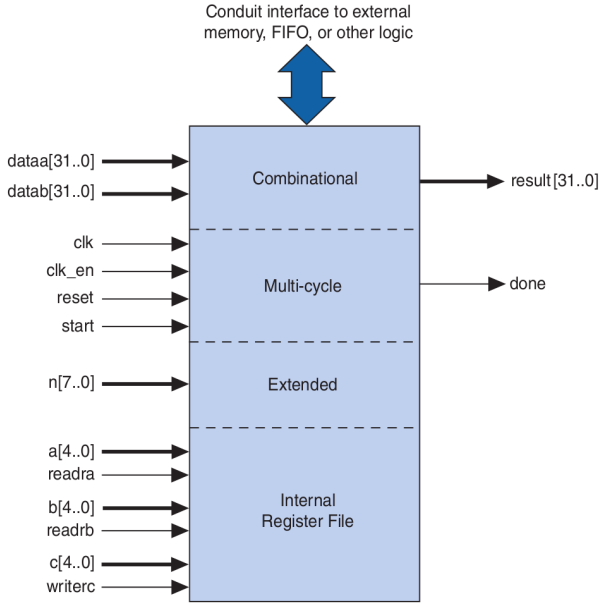


FIG. 5.3 – Les différents types d'instructions spécialisées du NIOSII et interface à respecter avec le processeur

5.1.2 Modèles d'architecture

Dans le cadre de nos travaux, nous considérons deux modèles d'architecture pour l'extension. Nous les appelons *modèle A* et *modèle B*. Chaque instruction spécialisée est mise en œuvre à travers un *Composant Instruction Spécialisée* (CIS). Pour les deux modèles d'architecture, on se place dans le contexte d'un bloc logique spécialisé unique qui contient tous les CIS. Ce bloc correspond à une instruction spécialisée dite « étendue » du NIOSII. Si toutes les instructions du bloc sont combinatoires, alors le bloc entier est un bloc logique combinatoire. Mais si une seule instruction du bloc est multi-cycle, alors cela force le bloc logique à présenter les ports de contrôle obligatoires des instructions multi-cycles (les ports `clk`, `clk_en`, et `reset`) et le bloc logique entier est alors multi-cycle.

5.1.2.1 Modèle A

La figure 5.4 illustre le modèle d'architecture A où CIS_1 représente la mise en œuvre du motif 1, CIS_n la mise en œuvre du motif n, r les registres d'entrée et ro les registres

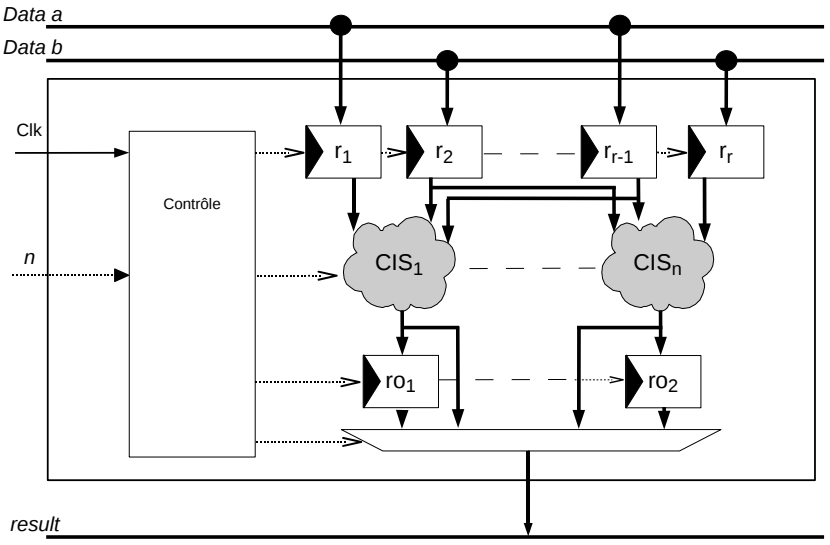


FIG. 5.4 – Modèle d'architecture A : l'extension possède des registres en entrée et en sortie

de sortie. Dans ce modèle, l'extension ne possède des registres qu'en entrée et en sortie. Chaque CIS peut accéder indépendemment à tous les registres d'entrée et de sortie. Lorsque l'instruction spécialisée requiert plus de deux entrées et une sortie, des transferts de données avec la file de registres sont nécessaires. Ces registres en entrée et en sortie assurent le stockage temporaire de données. Les données en entrée *dataa* et *datab* peuvent aller directement vers un CIS ou bien être stockées dans les registres d'entrée. Le résultat *result* renvoyé vers le processeur peut provenir d'un registre de sortie ou d'un CIS directement. Par souci de clarté, toutes les connexions ne sont pas représentées sur la figure. Notons qu'en s'appuyant sur les scenarii d'exécution des applications, il est possible d'appliquer des optimisations réduisant les connexions, le nombre de registres, et contribuant ainsi à diminuer la complexité de l'architecture.

Il est important de remarquer qu'un résultat produit par un CIS ne peut pas être directement utilisé par un autre CIS. La donnée doit être renvoyée vers le processeur par la sortie *result* avant de revenir par l'entrée *dataa* ou *datab*. Ce modèle d'architecture présente l'avantage de posséder une interconnexion relativement simple. L'inconvénient majeur est le surcoût en nombre de cycles de transfert de données nécessaires pour récupérer une donnée produite par une instruction spécialisée. Pour économiser ces cycles, il est intéressant de pouvoir récupérer directement dans l'extension une donnée produite par un CIS. Ceci est possible dans le modèle d'architecture B.

5.1.2.2 Modèle B

Le modèle d'architecture B est illustré figure 5.5 où les CIS_i ($\forall i \in \{1, n\}$) représentent la mise en œuvre des motifs. Dans ce modèle, l'extension possède une file de registres accessible en lecture et en écriture par tous les CIS. Une donnée produite par un CIS dans l'extension peut être directement utilisée par un autre CIS en passant par la file

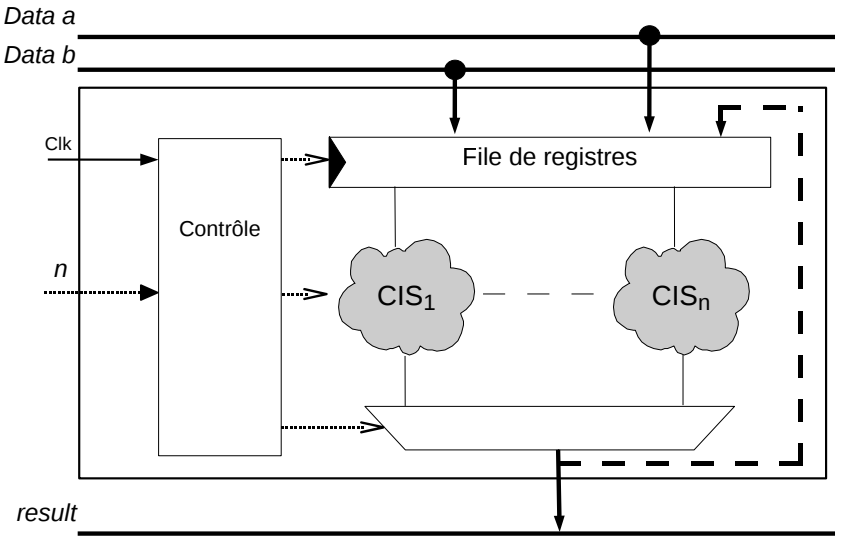


FIG. 5.5 – Modèle d'architecture B : l'extension a une file de registres

de registres de l'extension. Ce lien est symbolisé par le trait pointillé dans la figure 5.5. Sauvegarder des données dans la file de registres de l'extension permet de réduire le nombre de *register spills* dans le code généré pour le processeur [143].

Comme pour le modèle A, à partir des scenarii d'exécution, il est possible d'appliquer des optimisations sur les connexions entre les CIS et les registres pour diminuer le nombre de registres et la complexité de l'interconnexion entre les composants.

Bien que le NIOSII offre la possibilité d'avoir des instructions spécialisées « paramétrées » qui permettent de coder l'accès aux registres de l'extension, nous avons choisi de ne pas l'exploiter puisque cela oblige à choisir entre un opérande de la file de registres du processeur ou de l'extension. Cette solution augmente le nombre de transferts de données. Or, nous avons vu que le transfert de données est un goulot d'étranglement. Nous avons choisi d'encoder la sélection de l'instruction spécialisée et les accès aux registres de l'extension à travers l'indice `n`. Une instruction spécialisée peut avoir plusieurs contextes d'exécution différents (lecture et écriture dans des registres différents). Une instruction spécialisée nécessite autant d'indices que de contextes d'exécution.

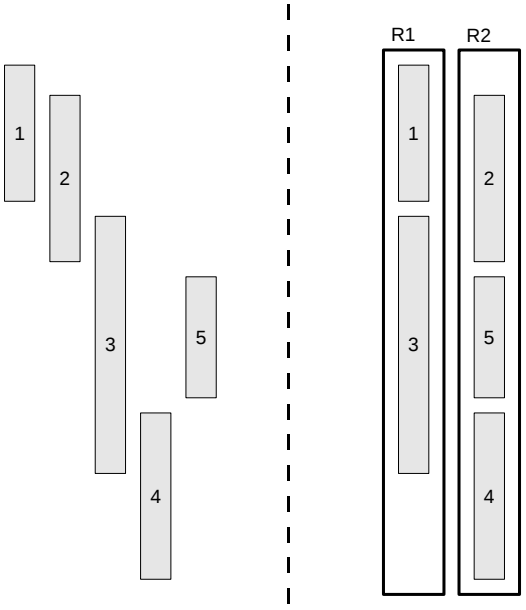
Les deux modèles d'architecture impliquant des registres, il est nécessaire de procéder à une étape d'allocation de registres afin de déterminer leur nombre et leur utilisation.

5.2 Allocation de registres

Lors de la sélection d'instructions, nous avons fait l'hypothèse que l'extension possède des registres, sans en connaître le nombre. Le but de notre technique d'allocation de registres est de déterminer le nombre de registres nécessaires à l'exécution de l'application selon l'ordonnancement trouvé à l'étape précédente. Ainsi, notre étape d'allocation de registres n'est pas concernée par les problèmes de *spilling*, *coalescing*, etc [74].

L'allocation de registres est un problème complexe [52, 47] et les techniques de résolu-

tion sont nombreuses [52, 129]. Dans le cas présent, nous cherchons à allouer les registres dans un contexte bien précis. En particulier, nous voulons allouer les registres, pas uniquement dans le but de minimiser leur nombre, mais de manière plus générale, dans le but de minimiser la surface de l'extension. Il convient donc de considérer les registres et les ressources qui permettent de réaliser l'interconnexion entre les registres et les CIS. Nous proposons une technique basée sur la programmation par contraintes, qui prend en compte les contraintes sur les registres, mais aussi les contraintes sur les ressources d'interconnexion.



(a) Variables et leur durée de vie (b) allocation

FIG. 5.6 – Illustration de l'allocation de registres

La figure 5.6 présente un exemple d'allocation de registres très simple où les rectangles à gauche représentent des variables et leur durée de vie, et à droite ces mêmes rectangles sont placés dans des registres. Dans ce cas, deux registres (`R1` et `R2`) suffisent pour stocker les données.

L'allocation de registres s'effectue sur un graphe réduit et s'appuie sur les informations d'ordonnancement. Chaque nœud représente une occurrence exécutée par le processeur ou par l'extension. Chaque lien représente une dépendance de donnée et correspond à une variable. Pour conserver la cohérence des notations avec les chapitres précédents, les motifs sont notés p comme *Pattern* (motif) et les occurrences sont notées m pour *Match* (occurrence). Pour résoudre le problème d'allocation de registres, nous utilisons la programmation par contraintes et en particulier la contrainte appelée `Diff2`.

5.2.1 Contrainte `Diff2`

La contrainte `Diff2` prend en paramètre une liste de rectangles à deux dimensions (c.-à-d. bidimensionnel). Chaque rectangle bidimensionnel est représenté par un *tuple* $[O_1, O_2, L_1, L_2]$ où O_i et L_i sont respectivement l'origine et la longueur du rectangle dans la dimension i. La contrainte `Diff2` impose que les rectangles ne se recouvrent pas. Formellement parlant, la contrainte assure que, pour chaque paire i, j (avec $i \neq j$) de rectangles bidimensionnels, il existe au moins une dimension k dans laquelle i est après j, ou j est après i. Dans le cas d'une `Diff2`, k vaut 1 ou 2 puisque la contrainte est en deux dimensions. On parle de `Diffn` pour cette contrainte dans n dimensions.

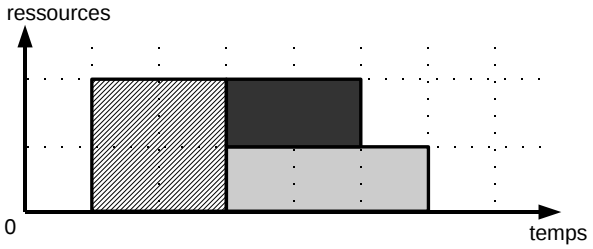


FIG. 5.7 – Exemple de la contrainte `Diff2`

La figure 5.7 montre un exemple de la contrainte `Diff2` impliquant trois rectangles, où la dimension y représente les ressources et la dimension x représente le temps. Le rectangle *hachuré* utilise deux ressources sur une durée égale à 2. Les rectangles *gris foncé* et *gris clair* utilisent seulement une ressource pendant une durée respective de 2 et 3. Il est ensuite possible de chercher à minimiser l'une ou l'autre des dimensions, c'est-à-dire minimiser le nombre de ressources ou minimiser le temps. La contrainte `Diff2` est notamment très pratique pour résoudre les problèmes d'ordonnancement sous contraintes de ressources ou sous contraintes de temps.

Il est assez intuitif de représenter les données à stocker et leur durée de vie sous forme de rectangles, c'est pourquoi nous avons décidé d'utiliser la contrainte `Diff2` pour modéliser les variables dans le problème d'allocation de registres.

5.2.2 Allocation de registres appliquée au modèle A

Pour le modèle d'architecture A, l'allocation de registres est très simple à effectuer. Il suffit de prêter attention au cas où une même donnée est transférée plusieurs fois, il faut alors réserver un registre d'entrée pour cette donnée et ce jusqu'à sa dernière utilisation. Pour les autres données, nous définissons une convention d'écriture et de lecture des registres. Par exemple, nous considérons que les deux premières entrées d'une occurrence exécutée par l'extension sont toujours *dataa* et *datab*, et que la première sortie est toujours *result*. Les autres entrées ont besoin d'un registre d'entrée pour stocker temporairement la donnée et les autres sorties ont besoin d'un registre de sortie.

Pour commencer, nous distinguons les nœuds qui représentent une donnée utilisée une seule fois (c.-à-d. les nœuds à un seul lien sortant) de ceux qui représentent une donnée

utilisée plusieurs fois (c.-à-d. les nœuds à plusieurs liens sortants). Nous définissons N_u, les nœuds à sortie unique, et N_{mult} les nœuds à sortie multiple. À chaque lien est associée une variable, notée R, dont la valeur représente le numéro de registre. Pour les entrées, le numéro 0 symbolise l'entrée *dataa*, et le numéro 1 l'entrée *datab*. Pour les sorties, le numéro 0 symbolise la sortie *result* (le numéro 1 est interdit pour éviter toute confusion). Les registres 0 et 1 sont donc des registres virtuels qui modélisent les entrées *dataa* et *datab* en entrée, et *result* en sortie. Pour le lien issu d'un nœud à sortie unique, la valeur de R varie de 0 à R_{max} où R_{max} désigne une borne maximale du nombre de registres. Pour les liens issus de nœuds à sortie multiple, la valeur de R vaut au minimum 2 (puisque la donnée doit forcément être stockée dans un registre réel) et nous imposons une contrainte d'égalité qui force les variables à posséder la même valeur. Enfin, pour chaque nœud, un rectangle qui représente la donnée est créé et ajouté à la contrainte `Diff2` (équation 5.1) :

$$\texttt{Diff2}([T_{R_i}, R_i, D_{R_i}, 1] \forall i \in N_u, [T_{R_j}, R_j, \texttt{Max}(D_{R_l}), 1] \forall j \in N_{mult}, \forall l \in out_j) \tag{5.1}$$

où :

- T_R : la date de début du rectangle (c'est-à-dire la date de début de vie de la donnée) ;
- R : le numéro du registre ;
- D_R : la durée du rectangle (c'est-à-dire, la durée de vie de la donnée) ;
- le chiffre 1 car un seul registre est nécessaire pour stocker la valeur ;
- out_j : les liens de sortie du nœud j ;
- $\texttt{Max}(D_{R_l})$: permet de déterminer la dernière utilisation de la donnée représentée par les liens $l \in out_j$.

La contrainte `Diff2` seule n'est pas suffisante, il faut en plus assurer qu'un registre différent est alloué pour chaque donnée en entrée d'une occurrence. Pour cela, la contrainte `AllDiff` qui impose que chaque donnée possède un registre différent (équation 5.2) est utilisée.

$$\forall m \in M : \texttt{AllDiff}(R_k) \forall k \in in_m \tag{5.2}$$

où :

- M : l'ensemble des occurrences,
- in_m : l'ensemble des entrées de l'occurrence m.

Nous proposons d'illustrer cette allocation de registres à travers un exemple simple. Soit le graphe de la figure 5.8, composé de trois nœuds, tous exécutés par l'extension. On distingue trois occurrences (M_1, M_2, M_3), et deux motifs (P_1, P_2). Le motif P_1 est un motif à quatre entrées et deux sorties et possède deux occurrences (M_1 et M_3). Le motif P_2 est un motif à trois entrées et une sortie et possède une occurrence (M_2). À chaque sortie de motif est attribué un numéro unique ; dans l'exemple, la sortie 1 du motif P_1 a le numéro dix (10), la sortie 2 du motif P_1 le numéro onze (11), et la sortie 1 du motif P_2 le numéro douze (12). Les nœuds `in` représentent des nœuds d'entrée qui proviennent de la file de registres du processeur. Les nœuds `out` sont des nœuds de sortie, qui représentent

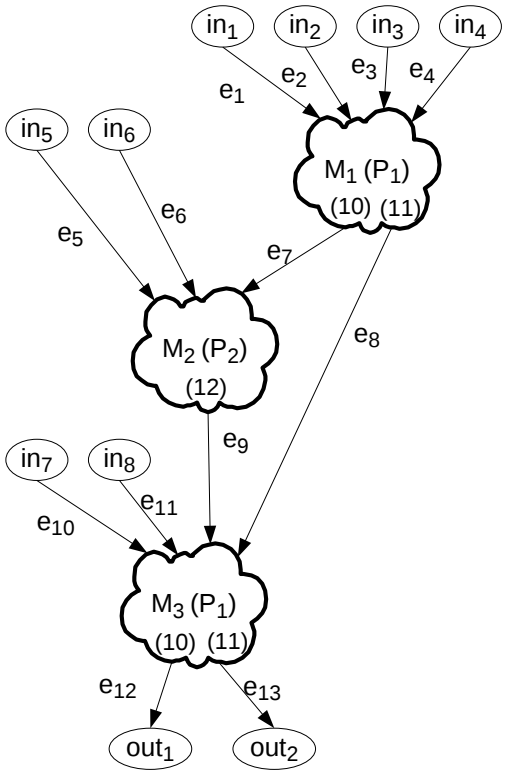


FIG. 5.8 – Exemple d'un graphe composé de trois nœuds exécutés par l'extension

des données à renvoyer vers la file de registres. Les liens représentent les dépendances de données entre les nœuds. Dans l'exemple, le graphe est constitué de treize liens, numérotés de e_1 à e_{13}.

La figure 5.9 illustre l'ordonnancement et le placement du graphe de la figure 5.8 sur le modèle d'architecture A. Ce graphe nécessite huit cycles pour être exécuté. Les cycles sont représentés par les rectangles grisés à gauche, numérotés de 0 à 7. La première colonne montre l'activité du processeur NIOSII et la deuxième l'activité de l'extension. Dans cet exemple, tous les nœuds sont exécutés par l'extension. Le NIOSII exécute une instruction spécialisée à chaque cycle (un transfert, ou un lancement).

Le motif P_1 est un motif à quatre entrées et deux sorties. Il nécessite donc $\lceil \frac{4}{2} \rceil - 1 = 1$ cycle additionnel de transfert de données en entrée, et $\lceil \frac{2}{1} \rceil - 1 = 1$ cycle additionnel de transfert de données en sortie. Ces cycles de transfert sont symbolisés par les formes de « maison » pour les transferts en entrée et les formes de « maison inversée » pour les transferts en sortie. L'exécution de l'occurrence M_1 se déroule de la façon suivante.

- Un cycle additionnel de transfert de données en entrée, représenté par la forme de « maison » INM_1 ; ce cycle permet d'envoyer les données représentées par les liens e_3 et e_4.
- Un cycle pour l'exécution de l'instruction spécialisée, représenté par le rectangle EXM_1 ; au lancement de l'instruction, il est possible d'envoyer deux données (e_1 et

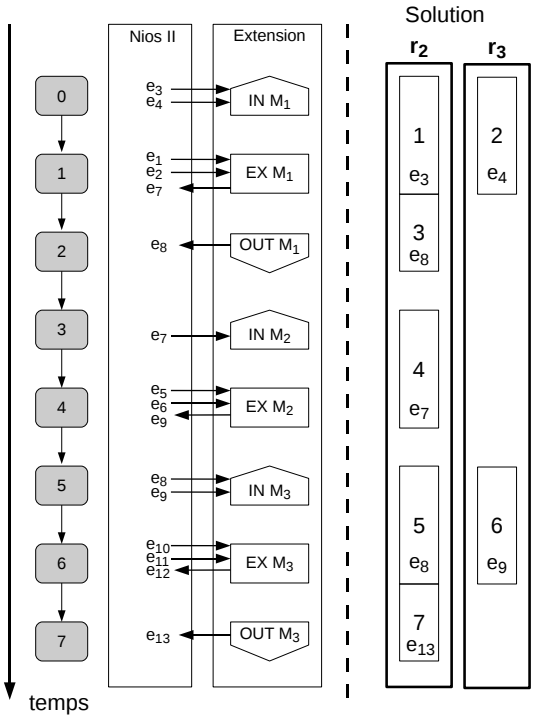


FIG. 5.9 – Ordonnancement et allocation de registres pour le modèle A

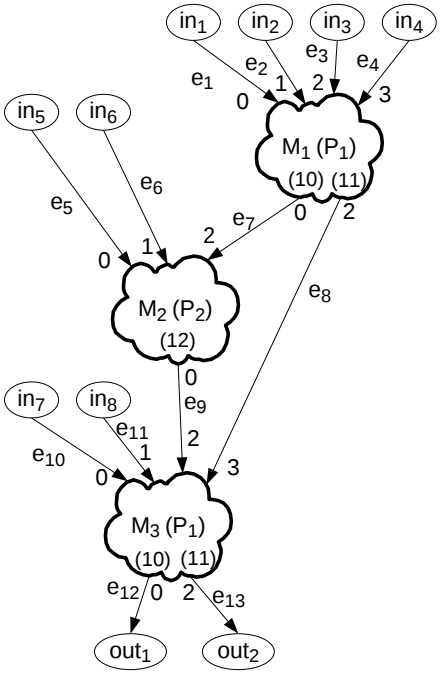


FIG. 5.10 – Solution de l'allocation de registres du graphe de la figure 5.8 pour le modèle d'architecture A

e_2) et de récupérer un résultat (e_7).

– Un cycle additionnel de transfert de données en sortie, représenté par la forme de « maison inversée » $OUTM_1$; ce cycle permet de récupérer le second résultat produit par l'instruction (e_8).

Ce schéma est répété pour l'exécution de l'occurrence M_3.

Le motif P_2 est un motif à trois entrées et une sortie. Il nécessite donc $\lceil \frac{3}{2} \rceil - 1 = 1$ cycle additionnel de transfert de données en entrée et $\lceil \frac{1}{1} \rceil - 1 = 0$ cycle additionnel de transfert de données en sortie. Le cycle de transfert de données en entrée est symbolisé par la forme de « maison » $IN\,M_2$. Le rectangle $EX\,M_2$ symbolise l'exécution de l'occurrence M_2. Pour des raisons de simplicité, l'exemple présente des instructions spécialisées qui nécessitent un seul cycle pour être exécutées.

La figure 5.10 montre le résultat de l'allocation de registres pour le graphe de la figure 5.8. Chaque entrée et chaque sortie d'une occurrence se voit attribuer un numéro. Pour les entrées, le numéro 0 symbolise l'entrée *dataa*, et le numéro 1 l'entrée *datab*. Pour les sorties, le numéro 0 symbolise la sortie *result* et le numéro 1 est interdit pour éviter toute confusion. Tous les autres numéros correspondent au numéro du registre qui stocke la donnée. La numérotation des registres commence donc à deux. Par exemple, pour la solution de la figure 5.10, l'entrée n° 1 de l'occurrence M_1 provient de *dataa*, l'entrée 2 de *datab*, l'entrée 3 provient du registre `r_2`, et l'entrée 4 du registre `r_3`. La sortie n° 1 de l'occurrence M_1 est directement renvoyée vers le processeur à travers le signal *result*, et la sortie n° 2 est stockée dans le registre `r_2`. Pour cet exemple, deux registres sont

nécessaires, les registres `r_2` et `r_3`.

La figure 5.9 montre également cette solution d'allocation de registres. Les registres `r_2` et `r_3` sont représentés par des colonnes et les rectangles qui les composent représentent l'occupation des registres par les données. Si on prend l'exemple du registre `r_2`, le rectangle 1 sert à stocker la donnée représentée par le lien e_3 jusqu'au début de l'exécution de l'occurrence M_1 ; le rectangle 3 sert à stocker la donnée représentée par le lien e_8 de la fin de l'exécution de l'occurrence M_1 jusqu'à la lecture du registre symbolisée par la forme de « maison inversée » $OUT M_1$. Le même schéma est répété pour les autres occurrences : le rectangle 4 pour le lien e_7, le rectangle 5 pour le lien e_8 et le rectangle 7 pour le lien e_{13}.

5.2.3 Allocation de registres appliquée au modèle B

Minimiser le coût en surface de silicium du module spécialisé ne se résume pas à minimiser le nombre de registres, il faut également prendre en compte le coût de l'interconnexion entre les registres et les CIS. Cette interconnexion est réalisée à l'aide de multiplexeurs, mis en œuvre à l'aide de LUT (*Look-Up Table*). La surface peut être exprimée en nombre de LE (*Logic Element*). Le LE est l'élément de base d'un FPGA d'Altera, contenant une bascule D et une LUT. La figure 5.11 illustre la structure d'un LE composé d'une LUT à quatre entrées, capable de mettre en œuvre un multiplexeur `2:1` et d'une bascule D capable de mettre en œuvre un registre.

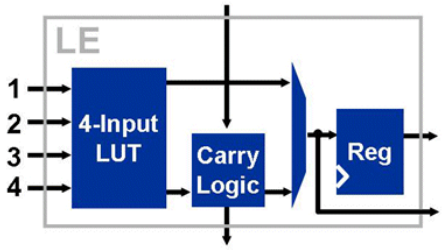


FIG. 5.11 – Élement logique (LE) du FPGA d'Altera

Le problème de la minimisation de la surface totale, en tenant compte des registres, de l'interconnexion et des multiplexeurs, n'est pas trivial. Dans notre approche, nous utilisons la programmation par contraintes pour pouvoir représenter, au sein d'un même problème de satisfaction de contraintes, l'ensemble des ressources impliquées dans le calcul de la surface totale. Nous avons développé une technique qui permet d'effectuer l'allocation de registres selon trois objectifs. Le premier est tout simplement de minimiser le nombre de registres. Le deuxième objectif est l'allocation de registres dans le but de minimiser le nombre de LUT. Le troisième objectif est de minimiser le nombre de LE.

5.2.3.1 Minimisation du nombre de registres

Pour minimiser le nombre de registres, nous utilisons la contrainte `Diff2`, où la première dimension représente le temps et la seconde les ressources. Chaque donnée, représentée par

un rectangle, utilise une ressource pendant un temps donné (défini par l'ordonnancement). On cherche à minimiser la dimension des ressources.

Nous partageons les liens qui composent le graphe en trois ensembles :

1. l'ensemble des liens qui représentent une communication processeur/processeur ; ces liens ne sont pas concernés par l'allocation de registres de l'extension,
2. les liens qui représentent une communication entre le processeur et l'extension ; c'est-à-dire les liens qui nécessitent un transfert de données,
3. les liens qui représentent une communication extension/extension ; c'est-à-dire utilisation de la file de registres pour stocker une donnée.

À chaque lien symbolisant une communication processeur/extension correspond une variable à domaine fini au sens de la programmation par contraintes, dont la valeur représente le numéro du registre. Comme pour le modèle A, en entrée, la valeur 0 symbolise *dataa*, et la valeur 1 symbolise *datab* ; en sortie, la valeur 0 représente *result* et la valeur 1 est prohibée pour éviter toute confusion.

À chaque lien symbolisant une communication extension/extension est également associée une variable dont la valeur vaut au minimum 2 (puisque la communication passe forcément par la file de registres de l'extension). Cette variable est notée R. Pour prendre en compte l'optimisation sur le transfert de données, pour chaque lien issu d'un même nœud, nous imposons une contrainte d'égalité qui force les variables à posséder la même valeur.

Pour illustrer notre technique d'allocation de registres, nous proposons de reprendre l'exemple du graphe de la figure 5.8. E_m définit l'ensemble des liens qui représentent une communication extension/extension, c'est-à-dire les liens qui relient deux nœuds exécutés par l'extension. Dans notre exemple, $E_m = \{e_7, e_8, e_9\}$. La variable correspondant au lien e_7 est donc $R_{e_7} = \{2..R\}$, dont le domaine peut varier de 2 à R, où R est la borne maximale du nombre de registres. E_t définit l'ensemble des liens qui représentent une communication processeur/extension, c'est-à-dire les liens de transfert de données du processeur vers l'extension ou inversement, de l'extension vers le processeur. Dans notre exemple, $E_t = \{e_1, e_2, e_3, e_4, e_5, e_6, e_{10}, e_{11}, e_{12}, e_{13}\}$.

Comme pour le modèle A, nous imposons aux deux premiers liens entrants d'un nœud exécuté par l'extension et provenants d'un nœud exécuté par le processeur de prendre respectivement les valeurs 0 (*dataa*) et 1 (*datab*). C'est le cas dans notre exemple des liens $e_1, e_2, e_5, e_6, e_{10}, e_{11}$. Tous les autres liens entrants du même type nécessitent un registre pour stocker temporairement la donnée. Ce sont les liens e_3, e_4 dans notre exemple. On définit un nouvel ensemble, l'ensemble E_τ, pour désigner ces liens. Nous avons donc $E_\tau = e_3, e_4$. On impose la contrainte 5.3 qui définit que pour chaque lien appartenant à E_m et chaque lien appartenant à E_τ, un rectangle est créé et ajouté à la contrainte.

$$\texttt{Diff2}([T_{R_i}, R_i, D_{R_i}, 1] \forall i \in E_m, [T_{R_j}, R_j, D_{R_j}, 1] \forall j \in E_\tau) \tag{5.3}$$

où :

- T_R : la date de début du rectangle (c'est-à-dire la date de début de vie de la donnée) ;
- R : le numéro du registre ;
- D_R : la durée du rectangle (c'est-à-dire, la durée de vie de la donnée) ;
- 1 car un seul registre est nécessaire pour stocker la valeur.

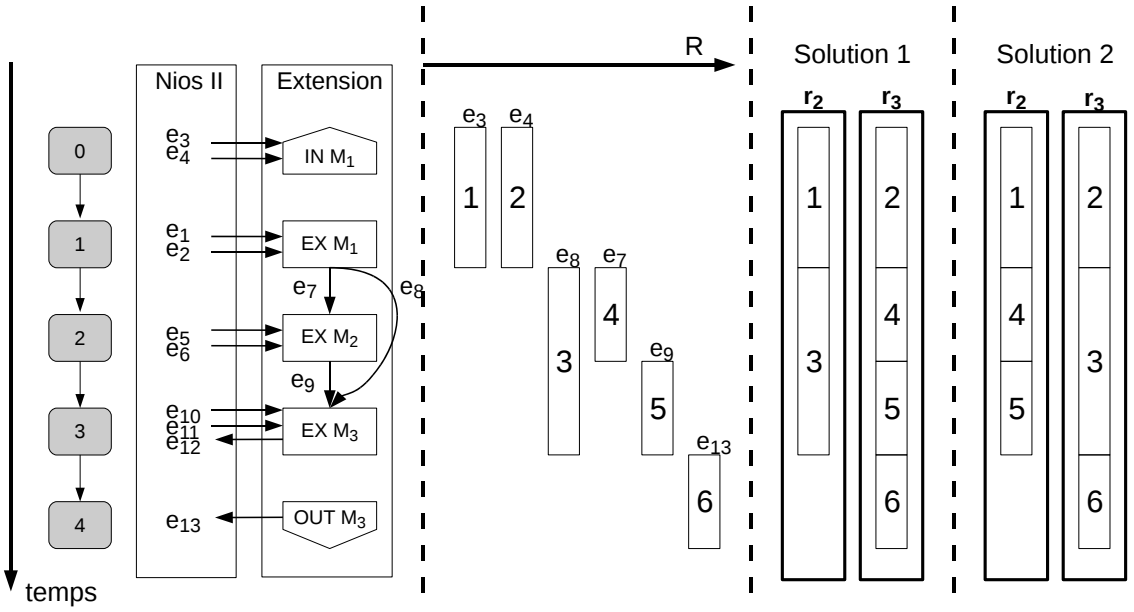


FIG. 5.12 – Exemple d'ordonnancement et d'allocation de registres pour le graphe de la figure 5.8

La figure 5.12 montre l'ordonnancement et le placement du graphe de la figure 5.8 sur le modèle d'architecture B, ainsi que les rectangles créés. Dans cet exemple, le graphe nécessite cinq cycles pour être exécuté. Les cycles sont représentés par les rectangles grisés à gauche, numérotés de 0 à 4. Une première colonne montre l'activité du processeur NIOSII, et une deuxième l'activité de l'extension. Dans cet exemple, tous les nœuds sont exécutés par l'extension ; le NIOSII exécute une instruction spécialisée à chaque cycle (soit un transfert, soit un traitement). Le nombre de cycles nécessaires pour exécuter ce graphe sur le modèle B est inférieur au nombre de cycles nécessaires pour l'exécuter sur le modèle A car le nombre de cycles additionnels de transfert de données est moins élevé. Ainsi, pour exécuter l'occurrence M_1, il faut $\lceil \frac{4}{2} \rceil - 1 = 1$ cycle additionnel de transfert de données en entrée comme pour le modèle A mais il ne faut aucun cycle additionnel de transfert de données en sortie puisqu'aucune des données produites n'est envoyée vers le processeur. Pour exécuter l'occurrence M_2, deux données proviennent du processeur, il faut donc $\lceil \frac{2}{2} \rceil - 1 = 0$ cycle additionnel de transfert de données en entrée. De même, pour exécuter l'occurrence M_3, seulement deux données proviennent du processeur, aucun cycle additionnel de transfert en entrée n'est donc nécessaire. Par contre, le cycle additionnel de transfert de données en sortie est toujours requis.

Pour cet exemple, six rectangles sont créés. Ils sont représentés sur la figure 5.12

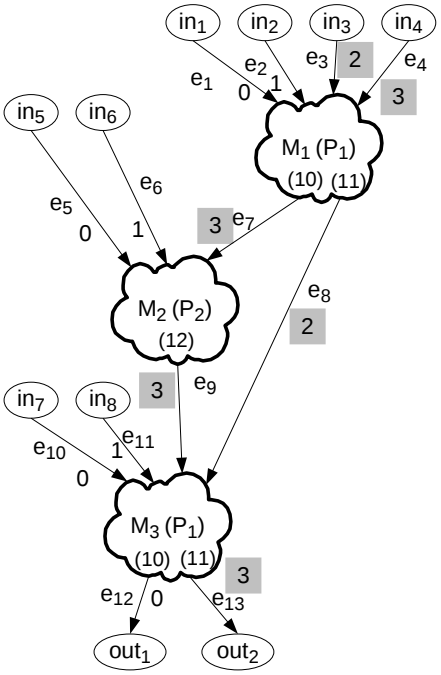


FIG. 5.13 – Solution 1

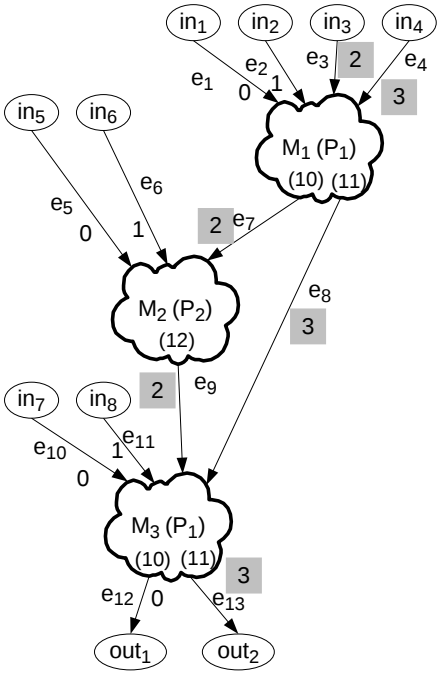


FIG. 5.14 – Solution 2

numérotés de 1 à 6. Les rectangles 1 et 2 servent à stocker respectivement les données représentées par les liens e_3 et e_4 jusqu'à la fin de l'exécution de l'occurrence M_1. Le rectangle 3 sert à stocker la donnée représentée par le lien e_8. La figure 5.12 montre par ailleurs ce lien direct entre l'exécution de l'occurrence M_1 et l'occurrence M_3. De même, les rectangles 4 et 5 stockent les données représentées par les liens e_7 et e_9. Enfin, le rectangle 6 stocke la donnée représentée par le lien e_{13} jusqu'à la lecture par le processeur.

La contrainte `Diff2` seule n'est pas suffisante, il faut en plus assurer qu'un registre différent est alloué pour chaque donnée en entrée d'une occurrence. Pour les sorties, il faut prêter attention au cas où une même donnée est utilisée par plusieurs nœuds. Il faut assurer qu'un registre différent est alloué pour chaque sortie différente d'une occurrence. La contrainte `AllDiff` impose que chaque donnée possède un registre différent (équation 5.4 et 5.5).

$$\forall m \in M : \texttt{AllDiff}(R_k) \forall k \in in_m \tag{5.4}$$

$$\forall m \in M : \texttt{AllDiff}(R_o) \forall o \in out_m \tag{5.5}$$

où :

- M : l'ensemble des occurrences,
- in_m : l'ensemble des entrées de l'occurrence m,
- out_m : l'ensemble des sorties différentes de l'occurrence m.

Enfin, on déclare une variable $TotalRegisters$ qui maintient le plus grand numéro de registre (équation 5.6) existant. L'équation 5.6 est la fonction de coût appliquée dans le

cas de la minimisation du nombre de registres.

$$TotalRegisters = \texttt{Max}(R_i, R_j) \forall i \in E_m, \forall j \in E_\tau \tag{5.6}$$

La figure 5.13 montre une solution d'allocation de registres pour le graphe de la figure 5.8. À chaque lien correspond un numéro, qui est le numéro de registre, sachant encore une fois que les numéros 0 et 1 en entrée symbolisent *dataa* et *datab*, et que le numéro 0 en sortie symbolise *result*. Cette solution, appelée solution 1, est également reportée sur la figure 5.12. Pour cette solution, deux registres sont nécessaires, les registres `r_2` et `r_3`. Par exemple, le registre `r_2` sert à stocker la variable représentée par le lien e_3 (rectangle 1), et la variable représentée par le lien e_8 (rectangle 3), cette variable est conservée pendant toute la durée de l'exécution de l'occurrence M_2.

5.2.3.2 Minimisation du nombre de LUT

Une même instruction spécialisée peut être exécutée plusieurs fois dans des contextes différents. Les données peuvent provenir de sources différentes : des registres ou du processeur. Il convient donc de placer des multiplexeurs en entrée des CIS pour sélectionner la bonne source. Il en est d'ailleurs de même pour les registres. Plusieurs CIS peuvent écrire dans le même registre. Des multiplexeurs sont donc utilisés pour remplir cette fonction. Or, un multiplexeur est coûteux en surface et en temps de traversée. Plus le multiplexeur est gros, plus il coûte en surface et plus le chemin critique en est affecté. Par ailleurs, il n'est pas forcément nécessaire que tous les registres soient connectés à toutes les entrées de tous les CIS. De la même façon, il n'est pas nécessaire que toutes les sorties de tous les CIS soient connectées à toutes les entrées de tous les registres. En d'autres termes, il n'est pas indispensable que tous les CIS puissent lire et écrire dans tous les registres. Notre idée est de pousser ce raisonnement encore plus loin en essayant de minimiser les interconnexions pour réduire la complexité et le coût de l'architecture. La figure 5.15 montre un exemple d'architecture simplifiée du modèle d'architecture B générique présenté figure 5.5. L'architecture possède deux CIS et trois registres. Les lignes pointillées représentent les possibilités d'écriture d'un motif dans un registre. Ainsi, CIS_1 peut écrire le résultat dans le registre r_2 et CIS_2 peut écrire dans le registre r_3. L'entrée 1 du CIS_1 peut provenir du registre r_1 ou bien de *datab*. L'entrée 2 du CIS_1 est forcément le registre r_2. Les multiplexeurs que l'on cherche à optimiser sont représentés par les formes trapézoïdales hachurées. La donnée *result* envoyée vers le processeur peut provenir de CIS_1 ou de CIS_2. Remarquons ici que le multiplexeur de sortie (pour le signal *result*) n'est pas concerné. En effet, il n'est pas possible de réduire la taille de ce multiplexeur.

C'est dans ce contexte que nous cherchons à allouer les registres de façon à minimiser le nombre et la taille des multiplexeurs en entrée des motifs et des registres. La métrique choisie pour calculer la complexité de l'architecture est le nombre de LUT. Le coût de la surface est calculé d'après le coût d'un multiplexeur `2:1` et nous supposons que ce coût croît linéairement avec le nombre d'entrées du multiplexeur, comme supposé dans [210].

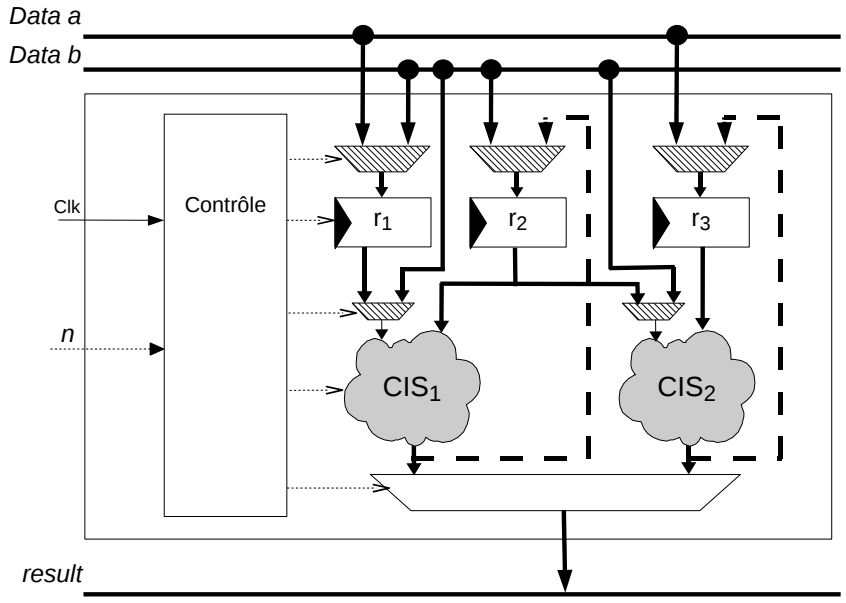


FIG. 5.15 – Modèle d'architecture B simplifié

En d'autres termes, un multiplexeur à deux entrées nécessite une LUT à quatre entrées, et pour chaque entrée supplémentaire, une autre LUT à quatre entrées est nécessaire. L'équation (5.7) indique comment calculer le nombre de LUT nécessaires pour la mise en œuvre d'un multiplexeur. C'est tout simplement le nombre d'entrées à multiplexer moins 1.

$$NbLUTs = N_{in} - 1 \tag{5.7}$$

où N_{in} désigne le nombre de sources différentes à multiplexer.

Chaque donnée étant codée sur 32 bits, il faut multiplier le nombre de LUT par 32 pour obtenir le nombre réel de LUT. Mais comme ce facteur multiplicatif n'a qu'une influence proportionnelle sur le nombre réel de LUT, nous avons choisi de ne pas compliquer les contraintes en ne pondérant pas le nombre de LUT.

Nous définissons le nombre total de LUT comme étant la somme du nombre de LUT nécessaires aux multiplexeurs en entrée des motifs plus le nombre de LUT nécessaires aux multiplexeurs en entrée des registres (équation 5.8). L'équation 5.8 correspond à la fonction de coût appliquée pour minimiser le nombre de LUT.

$$TotalLUTs = NbLUTs_{Pat} + NbLUTs_{Reg} \tag{5.8}$$

où :

- $TotalLUTs$: le nombre total de LUT,
- $NbLUTs_{Pat}$: le nombre de LUT nécessaires en entrée des motifs,
- $NbLUTs_{Reg}$: le nombre de LUT nécessaires en entrée des registres.

	Coût du nombre de registres	Coût du nombre de LUT							Coût du nombre de LE
		pour les motifs			pour les registres			total	
		P_1	P_2	total	r_2	r_3	total		
Solution 1	2	2	0	2	1	3	4	6	6
Solution 2	2	0	0	0	2	1	3	3	3

TAB. 5.1 – Coût des solutions en nombre de registres, de LUT, et de LE. La solution 2 propose une allocation de registres qui permet d'économiser 3 LUT par rapport à la solution 1, pour un même nombre de registres.

Calcul du nombre de LUT pour les multiplexeurs en entrée des motifs. Pour expliquer ce calcul, reprenons l'exemple du graphe de la figure 5.8 et en particulier la solution 1 de l'allocation de registres présentée auparavant (figure 5.13). Dans cet exemple, le motif P_1 possède deux occurrences (M_1 et M_3). Pour exécuter l'occurrence M_1, l'entrée n° 1 du motif P_1 est 0 (*dataa*), l'entrée n° 2 est 1 (*datab*), l'entrée n° 3 provient du registre 2, et l'entrée n° 4 provient du registre 3. Or, pour exécuter l'occurrence M_3, les entrées n° 1 et n° 2 sont également 0 et 1, comme pour l'occurrence M_1, mais les entrées n° 3 et n° 4 proviennent respectivement des registres 3 et 2. Pour l'entrée n° 3 du motif P_1, la source provient soit du registre 2 dans le cas où c'est l'occurrence M_1 qui est exécutée, soit du registre 3 dans le cas de l'occurrence M_3. Il faut donc placer un multiplexeur à l'entrée n° 3 du motif P_1 pour sélectionner la bonne source en fonction de l'occurrence à exécuter. Ce multiplexeur doit sélectionner parmi deux entrées, et un multiplexeur `2:1` est mis en œuvre par une LUT à quatre entrées. Il en est de même pour l'entrée n° 4 du motif P_1. Le motif P_2 est quant à lui exécuté une seule fois, à travers l'occurrence M_2, il ne nécessite donc aucun multiplexeur en entrée. Le coût de cette solution est donc de 2 registres et 2 LUT à quatre entrées pour les motifs, comme résumé dans le tableau 5.1.

Considérons maintenant la solution 2 de la figure 5.14, reportée dans la figure 5.12. Cette solution propose de stocker la variable représentée par le lien e_8 dans le registre 3, et de stocker e_7 et e_9 dans le registre 2. Les différences entre la solution 1 (figure 5.13) et la solution 2 (figure 5.14) sont mises en valeur par les cadres grisés. Dans le cas de la solution 2, pour les deux occurrences M_1 et M_3, l'entrée n° 3 est le registre 2, et l'entrée n° 4 est le registre 3. Il n'y a donc pas besoin de multiplexeur puisque la source est toujours la même. Le coût de cette solution est de 2 registres, comme pour la solution 1, mais aucune LUT pour les motifs (cf. tableau 5.1). On remarque donc qu'il est possible, pour un même nombre de registres, de les allouer de manière à réduire les multiplexeurs en entrée des motifs.

De manière plus formelle, le nombre de LUT nécessaires en entrée des motifs est égal à la somme du nombre de LUT nécessaires à chaque motif (équation 5.9).

$$NbLUTs_{Pat} = \sum_{p \in P} NbLUTs_p \tag{5.9}$$

où P est l'ensemble des motifs. Dans notre exemple, $P = P_1, P_2$.

Le nombre de LUT nécessaires pour un motif p est la somme des LUT nécessaires à

chaque entrée du motif. Il est formulé par l'équation 5.10.

$$NbLUTs_p = \sum_{i \in in} NbLUTs_p^i \tag{5.10}$$

où in est l'ensemble des entrées du motif p. $NbLUTs_p^i$ correspond au nombre de LUT nécessaires à l'entrée n° i du motif p. Par exemple, pour le motif P_1, l'équation 5.10 devient l'équation 5.11, qui calcule la somme des LUT nécessaires pour les quatre entrées du motif.

$$NbLUTs_{P_1} = \sum_{i=1}^{4} NbLUTs_{P_1}^i = NbLUTs_{P_1}^1 + NbLUTs_{P_1}^2 + NbLUTs_{P_1}^3 + NbLUTs_{P_1}^4 \tag{5.11}$$

Le nombre de LUT est directement lié au nombre de sources différentes N_{in} (équation 5.7). Pour déterminer N_{in}, le nombre de sources différentes pour chaque entrée d'un motif, nous utilisons la contrainte `Values` qui permet de déterminer, parmi une liste de variables, le nombre de valeurs différentes que possède ces variables. Nous imposons la contrainte (5.12) pour chaque entrée de chaque motif.

$$\forall p \in P, \forall i \in in_p : N_{in_p^i} = \texttt{Values}(\{List(v_m^i), \forall m \in M_p\}) \tag{5.12}$$

où :

- P : l'ensemble des motifs,
- in_p : l'ensemble des entrées du motif p,
- $N_{in_p^i}$: le nombre de sources différentes pour l'entrée n° i du motif p,
- M_p : l'ensemble des occurrences du motif p,
- v_m^i : le numéro de la source pour l'entrée n° i de l'occurrence m.

Pour chaque motif p de l'ensemble des motifs P, nous parcourons chaque entrée i parmi l'ensemble des entrées du motif in_p et nous imposons la contrainte `Values`, qui calcule le nombre de sources différentes $N_{in_p^i}$ pour l'entrée i du motif p en fonction de chaque occurrence m appartenant à l'ensemble des occurrences M_p du motif p.

Continuons avec le motif P_1 et la solution 1 de la figure 5.13, et détaillons le calcul du nombre de LUT pour l'entrée n° 1 du motif P_1 dans l'équation 5.13. Le numéro de la source pour l'entrée n° 1 de l'occurrence M_1 ($v_{M_1}^1$) est 0 (de même pour le numéro de la source pour l'entrée n° 1 de l'occurrence M_3 ($v_{M_3}^1$)). Le nombre de valeurs différentes vaut 1 puisque 0 est la seule valeur possible. On en déduit le nombre de LUT pour l'entrée n° 1 du motif P_1 ($NbLUTs_{P_1}^1$) qui vaut 0.

$$N_{in_{P_1}^1} = \texttt{Values}(\{\underbrace{v_{M_1}^1}_{0}, \underbrace{v_{M_3}^1}_{0}\}) = 1 \Rightarrow NbLUTs_{P_1}^1 = N_{in_{P_1}^1} - 1 = 1 - 1 = 0 \tag{5.13}$$

Les équations (5.14 à 5.16) montrent, avec moins de détails, le calcul du nombre de LUT pour les autres entrées du motif $P1$ pour la solution 1, et l'équation 5.17 montre le

calcul du nombre de LUT nécessaires en entrée du motif P_1. Le motif P_1 nécessite deux LUT pour mettre en œuvre les multiplexeurs sur ses entrées, une pour l'entrée n° 3, et une pour l'entrée n° 4.

$$N_{in^2_{P_1}} = \texttt{Values}(\{1,1\}) = 1 \Rightarrow NbLUTs^2_{P_1} = 0 \tag{5.14}$$

$$N_{in^3_{P_1}} = \texttt{Values}(\{2,3\}) = 2 \Rightarrow NbLUTs^3_{P_1} = 1 \tag{5.15}$$

$$N_{in^4_{P_1}} = \texttt{Values}(\{3,2\}) = 2 \Rightarrow NbLUTs^4_{P_1} = 1 \tag{5.16}$$

$$NbLUTs_{P_1} = \underbrace{NbLUTs^1_{P_1}}_{0} + \underbrace{NbLUTs^2_{P_1}}_{0} + \underbrace{NbLUTs^3_{P_1}}_{1} + \underbrace{NbLUTs^4_{P_1}}_{1} = 2 \tag{5.17}$$

Calcul du nombre de LUT pour les multiplexeurs en entrée des registres. Le calcul du nombre de sources différentes en entrée d'un registre est plus compliqué car, contrairement aux motifs dont on connaît exactement le nombre, on ne connaît pas le nombre de registres. Nous pouvons néanmoins faire une estimation grossière *dans le pire des cas* où chaque donnée nécessite un registre et aucun partage de registre n'est effectué. On borne ainsi le nombre maximal de registres, noté R. Dans notre exemple, le nombre de registres dans le pire des cas est six, qui correspond au nombre initial de rectangles. Ces registres peuvent être vus comme des *pseudo-registres* ou des registres *virtuels*. Dans l'exemple, parmi les six registres *virtuels* initiaux, seuls deux registres sont convertis en registres réels.

Le nombre de LUT nécessaires à la mise en œuvre de tous les multiplexeurs en entrée des registres est la somme des multiplexeurs en entrée de chaque registre (équation 5.18).

$$NbLUTs_{Reg} = \sum_{r \in R} NbLUTs_r \tag{5.18}$$

où R est l'ensemble des registres.

L'équation 5.19 permet de déterminer le nombre de LUT nécessaires pour mettre en œuvre le multiplexeur en entrée du registre r.

$$NbLUTs_r = NbSources_r - 1 \tag{5.19}$$

Pour trouver le nombre de sources différentes pour le registre r ($NbSources_r$), on créé un *tableau des connexions*, comme illustré par le tableau 5.2. Ce tableau représente toutes les connexions possibles entre les sources (chaque sortie s de chaque occurrence M) et les registres ($r_0 \dots r_R$). Une variable Sel, qui vaut 0 ou 1, permet de déterminer si la connexion est établie entre la sortie s d'une occurrence m, et un registre r ($_rSel^s_m = 1$) ou non ($_rSel^s_m = 0$), équation 5.20. Rappelons qu'à chaque lien correspond une variable, notée R, dont la valeur est le numéro de registre dans lequel la donnée est stockée et déterminée par l'équation 5.3. Si la valeur de la variable R est égale au numéro du registre r, alors la connexion est établie. Dans la solution 1 de notre exemple, $R_{e_7} = 3$, et le lien e_7 est

	M_1				M_m	
	s_1	...	s_n	...	s_n	
r_0	${}_{r_0}Sel^{s_1}_{M_1}\{0,1\}$ $\cdot N^{s_1}_{M_1}$		${}_{r_0}Sel^{s_n}_{M_1}\{0,1\}$ $\cdot N^{s_n}_{M_1}$		${}_{r_0}Sel^{s_n}_{M_m}\{0,1\}$ $\cdot N^{s_n}_{M_m}$	$\texttt{Values} \Rightarrow NbSources_{r_0}$
r_1	${}_{r_1}Sel^{s_1}_{M_1}\{0,1\}$ $\cdot N^{s_1}_{M_1}$		${}_{r_1}Sel^{s_n}_{M_1}\{0,1\}$ $\cdot N^{s_n}_{M_1}$		${}_{r_1}Sel^{s_n}_{M_m}\{0,1\}$ $\cdot N^{s_n}_{M_m}$	$\texttt{Values} \Rightarrow NbSources_{r_1}$
...						
r_R	${}_{r_R}Sel^{s_1}_{M_1}\{0,1\}$ $\cdot N^{s_1}_{M_1}$		${}_{r_R}Sel^{s_n}_{M_1}\{0,1\}$ $\cdot N^{s_n}_{M_1}$		${}_{r_R}Sel^{s_n}_{M_m}\{0,1\}$ $\cdot N^{s_n}_{M_m}$	$\texttt{Values} \Rightarrow NbSources_{r_R}$

TAB. 5.2 – Tableau des connexions générique

connecté à la sortie s_1 de l'occurrence M_1. Il y a connexion entre le registre r_3 et la sortie s_1 de l'occurrence M_1, donc ${}_{r_3}Sel^{s_1}_{M_1} = 1$. La contrainte 5.20 permet de déterminer si la connexion est établie ou non.

$$\forall l \in s_m \ : R_l = r \Leftrightarrow {}_rSel^s_m = 1 \tag{5.20}$$

où :

- l : le lien connecté à la sortie s_m,
- R_l : le numéro de registre pour stocker la donnée représentée par le lien l,
- s_m : la sortie s de l'occurrence m,
- r : le numéro du registre,
- ${}_rSel^s_m$: la variable Sel qui modélise la connexion entre la sortie s de l'occurrence m et le registre r.

Chaque sortie de motif se voit attribuer un numéro unique (par exemple, $N^{s_1}_{M_1} = 10$ pour la sortie 1 s_1 de l'occurrence M_1). On définit ainsi le numéro de la source qui est le numéro de la sortie du motif dont elle provient. La variable Sel est pondérée par le numéro de la source. La contrainte `Values` est encore une fois utilisée pour calculer le nombre de valeurs différentes (c'est-à-dire le nombre de sources différentes) en entrée du registre r (équation 5.21).

$$\forall r \in R, NbSources_r = \texttt{Values}(\{List({}_rSel^s_m \cdot N^s_m), \forall m \in M, \forall s \in out_m\}) \tag{5.21}$$

où :

- ${}_rSel^s_m = 1$ si la connexion est établie entre la sortie s de l'occurrence m et le registre r, ${}_rSel^s_m = 0$ sinon,
- M : l'ensemble des occurrences,
- out_m : l'ensemble des sorties de l'occurrence m,
- N^s_m : le numéro de la sortie s de l'occurrence m défini d'après le numéro de son motif.

Le tableau 5.3 présente le tableau des connexions pour l'exemple de la figure 5.8 et détaille les différentes connexions entre les registres, r_2 à r_7, et les sorties des occurrences M_1, M_2, et M_3, pour les deux solutions déjà présentées (solutions 1 et 2). Rappelons que chaque sortie d'un motif possède un numéro unique, et que dans l'exemple la sortie 1 du

	M_1		M_2	M_3		Values
	s_1 $N^{s_1}_{M_1} = 10$	s_2 $N^{s_2}_{M_1} = 11$	s_1 $N^{s_1}_{M_2} = 12$	s_1 $N^{s_1}_{M_3} = 10$	s_2 $N^{s_2}_{M_3} = 11$	
r_2	${}_{r_2}Sel^{s_1}_{M_1}\{0,1\} \cdot N^{s_1}_{M_1}$	${}_{r_2}Sel^{s_2}_{M_1}\{0,1\} \cdot N^{s_2}_{M_1}$	${}_{r_2}Sel^{s_1}_{M_2}\{0,1\} \cdot N^{s_1}_{M_2}$	${}_{r_2}Sel^{s_1}_{M_3}\{0,1\} \cdot N^{s_1}_{M_3}$	${}_{r_2}Sel^{s_2}_{M_3}\{0,1\} \cdot N^{s_2}_{M_3}$	
Solution 1	$0 \cdot 10 = 0$	$1 \cdot 11 = 11$	$0 \cdot 12 = 0$	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$2 \Rightarrow$ 1 LUT
Solution 2	$1 \cdot 10 = 10$	$0 \cdot 11 = 0$	$1 \cdot 12 = 12$	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$3 \Rightarrow$ 2 LUT
r_3	${}_{r_3}Sel^{s_1}_{M_1}\{0,1\} \cdot N^{s_1}_{M_1}$	${}_{r_3}Sel^{s_2}_{M_1}\{0,1\} \cdot N^{s_2}_{M_1}$	${}_{r_3}Sel^{s_1}_{M_2}\{0,1\} \cdot N^{s_1}_{M_2}$	${}_{r_3}Sel^{s_1}_{M_3}\{0,1\} \cdot N^{s_1}_{M_3}$	${}_{r_3}Sel^{s_2}_{M_3}\{0,1\} \cdot N^{s_2}_{M_3}$	
Solution 1	$1 \cdot 10 = 10$	$0 \cdot 11 = 0$	$1 \cdot 12 = 12$	$0 \cdot 10 = 0$	$1 \cdot 11 = 11$	$4 \Rightarrow$ 3 LUT
Solution 2	$0 \cdot 10 = 0$	$1 \cdot 11 = 11$	$0 \cdot 12 = 0$	$0 \cdot 10 = 0$	$1 \cdot 11 = 11$	$2 \Rightarrow$ 1 LUT
r_4	${}_{r_4}Sel^{s_1}_{M_1}\{0,1\} \cdot N^{s_1}_{M_1}$	${}_{r_4}Sel^{s_2}_{M_1}\{0,1\} \cdot N^{s_2}_{M_1}$	${}_{r_4}Sel^{s_1}_{M_2}\{0,1\} \cdot N^{s_1}_{M_2}$	${}_{r_4}Sel^{s_1}_{M_3}\{0,1\} \cdot N^{s_1}_{M_3}$	${}_{r_4}Sel^{s_2}_{M_3}\{0,1\} \cdot N^{s_2}_{M_3}$	
Solution 1	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$0 \cdot 12 = 0$	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$1 \Rightarrow$ 0 LUT
Solution 2	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$0 \cdot 12 = 0$	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$1 \Rightarrow$ 0 LUT
r_5	${}_{r_5}Sel^{s_1}_{M_1}\{0,1\} \cdot N^{s_1}_{M_1}$	${}_{r_5}Sel^{s_2}_{M_1}\{0,1\} \cdot N^{s_2}_{M_1}$	${}_{r_5}Sel^{s_1}_{M_2}\{0,1\} \cdot N^{s_1}_{M_2}$	${}_{r_5}Sel^{s_1}_{M_3}\{0,1\} \cdot N^{s_1}_{M_3}$	${}_{r_5}Sel^{s_2}_{M_3}\{0,1\} \cdot N^{s_2}_{M_3}$	
Solution 1	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$0 \cdot 12 = 0$	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$1 \Rightarrow$ 0 LUT
Solution 2	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$0 \cdot 12 = 0$	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$1 \Rightarrow$ 0 LUT
r_6	${}_{r_6}Sel^{s_1}_{M_1}\{0,1\} \cdot N^{s_1}_{M_1}$	${}_{r_6}Sel^{s_2}_{M_1}\{0,1\} \cdot N^{s_2}_{M_1}$	${}_{r_6}Sel^{s_1}_{M_2}\{0,1\} \cdot N^{s_1}_{M_2}$	${}_{r_6}Sel^{s_1}_{M_3}\{0,1\} \cdot N^{s_1}_{M_3}$	${}_{r_6}Sel^{s_2}_{M_3}\{0,1\} \cdot N^{s_2}_{M_3}$	
Solution 1	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$0 \cdot 12 = 0$	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$1 \Rightarrow$ 0 LUT
Solution 2	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$0 \cdot 12 = 0$	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$1 \Rightarrow$ 0 LUT
r_7	${}_{r_7}Sel^{s_1}_{M_1}\{0,1\} \cdot N^{s_1}_{M_1}$	${}_{r_7}Sel^{s_2}_{M_1}\{0,1\} \cdot N^{s_2}_{M_1}$	${}_{r_7}Sel^{s_1}_{M_2}\{0,1\} \cdot N^{s_1}_{M_2}$	${}_{r_7}Sel^{s_1}_{M_3}\{0,1\} \cdot N^{s_1}_{M_3}$	${}_{r_7}Sel^{s_2}_{M_3}\{0,1\} \cdot N^{s_2}_{M_3}$	
Solution 1	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$0 \cdot 12 = 0$	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$1 \Rightarrow$ 0 LUT
Solution 2	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$0 \cdot 12 = 0$	$0 \cdot 10 = 0$	$0 \cdot 11 = 0$	$1 \Rightarrow$ 0 LUT

TAB. 5.3 – Tableau des connexions pour l'exemple de la figure 5.8

motif P_1 possède le numéro 10 ($N^1_{P_1} = 10$), la sortie 2 du motif P_1 possède le numéro 11 ($N^2_{P_1} = 11$) et la sortie 1 du motif P_2 le numéro 12 ($N^1_{P_2} = 12$). On déduit le numéro de la sortie de l'occurrence d'après le numéro de la sortie de son motif. Par exemple, le numéro de la sortie 1 de l'occurrence M_1 est le numéro de la sortie 1 du motif P_1 ($N^1_{M_1} = N^1_{P_1} = 10$).

Pour remplir le tableau des connexions, examinons la solution 1 (figure 5.13) et regardons les connexions occurrence par occurrence et registre par registre.

- Est-ce que la sortie 1 de l'occurrence M_1 écrit dans le registre r_2 ? La réponse est non, donc ${}_{r_2}Sel^{s_1}_{M_1} = 0$ et ${}_{r_2}Sel^{s_1}_{M_1} \cdot N^{s_1}_{M_1} = 0 \cdot 10 = 0$.
- Est-ce que la sortie 2 de l'occurrence M_1 écrit dans le registre r_2 ? La réponse est oui, donc ${}_{r_2}Sel^{s_2}_{M_1} = 1$ et ${}_{r_2}Sel^{s_2}_{M_1} \cdot N^{s_2}_{M_1} = 1 \cdot 11 = 11$.
- Est-ce que la sortie 1 de l'occurrence M_1 écrit dans le registre r_3 ? La réponse est oui, donc ${}_{r_3}Sel^{s_1}_{M_1} = 1$ et ${}_{r_3}Sel^{s_1}_{M_1} \cdot N^{s_1}_{M_1} = 1 \cdot 10 = 10$.
- Est-ce que la sortie 2 de l'occurrence M_1 écrit dans le registre r_3 ? La réponse est non, donc ${}_{r_3}Sel^{s_2}_{M_1} = 0$ et ${}_{r_3}Sel^{s_2}_{M_1} \cdot N^{s_2}_{M_1} = 0 \cdot 11 = 0$.
- Est-ce que la sortie 1 de l'occurrence M_1 écrit dans le registre r_4 ? La réponse est non, donc ${}_{r_4}Sel^{s_1}_{M_1} = 0$ et ${}_{r_4}Sel^{s_1}_{M_1} \cdot N^{s_1}_{M_1} = 0 \cdot 10 = 0$.

Le tableau est rempli de cette façon pour toutes les connexions possibles.

Analysons maintenant la solution 1 pour le registre r_2, le nombre de valeurs différentes parmi l'ensemble (0,11,0,0,0) est 2, il faut donc un multiplexeur `2:1`, mis en œuvre dans une LUT. La valeur 0 ne correspond pas à une source *réelle* et le registre r_2 ne devrait pas avoir de multiplexeur en entrée puisque la sortie 1 du motif P_1 est la seule à écrire dedans. Mais nous conservons cette valeur car elle permet de représenter des registres à chargement

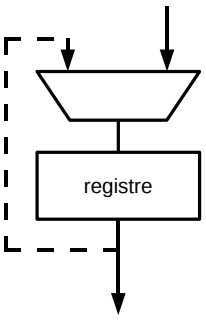


FIG. 5.16 – Registre à chargement systématique

systématique comme illustrée par la figure 5.16. Un registre à chargement systématique est un registre qui est chargé à chaque front d'horloge. Si la valeur du registre ne change pas, le registre doit être chargé par la valeur courante, d'où le bouclage, représenté par la ligne en pointillé figure 5.16.

Si le registre n'a aucune connexion, alors toutes les variables valent 0, donc le nombre de sources vaut 1 ($NbSources_r = 1$), et le nombre de LUT pour ce registre vaut 0 ($NbLUTs_r = 0$). Par exemple, les registres r_4 à r_7 n'ont aucune connexion et ne nécessitent alors aucune LUT. Ces registres restent des registres virtuels et n'ont aucune influence sur le calcul du nombre de LUT.

Pour la solution 1 le nombre de LUT nécessaires en entrée des registres est 4 et pour la solution 2 il est de seulement 3. Ces résultats sont reportés dans le tableau 5.1. En additionnant le nombre de LUT pour les motifs et pour les registres, on s'aperçoit que le nombre de LUT peut varier du simple ou double. En effet, seulement trois LUT sont nécessaires pour la solution 2 alors qu'il en faut six pour la solution 1. Il apparaît clairement qu'il est possible d'allouer les registres de façon à réduire le nombre de multiplexeurs en entrée des motifs *et* en entrée des registres. Ceci a pour effet de réduire la complexité de l'interconnexion, et, de manière globale, de réduire les ressources nécessaires à la mise en œuvre de l'architecture.

Pour minimiser la complexité de l'architecture, le solveur de contraintes cherche à affecter toujours la même valeur à une entrée d'un motif. Par exemple, que l'entrée i d'un motif p lise toujours le registre r_n. Dans notre exemple, le solveur affecte la valeur 2 à l'entrée n° 3 du motif P_1, et la valeur 3 à l'entrée n° 4.

Le problème de l'allocation de registres dans le but de minimiser le nombre de LUT est un problème complexe dont le solveur de contraintes peine à trouver une solution optimale pour des instances de problèmes de grande taille (graphe d'une centaine de nœuds). Si la solution optimale n'est pas obtenue dans le temps imparti, nous divisons le problème en deux sous-problèmes. Les deux problèmes de minimisation du nombre de LUT et d'allocation de registres sont intimement liés mais peuvent être résolus séparément. Les contraintes sont donc posées au sein d'un seul problème de satisfaction de contraintes et nous utili-

sons une méthode de recherche spécifique à la programmation par contraintes appelée *recherche combinée* (*combining search*) qui permet de résoudre des ensembles de variables séparément. Nous pouvons donc soit minimiser dans un premier temps le nombre de registres, puis minimiser le nombre de LUT, soit l'inverse, c'est-à-dire minimiser d'abord le nombre de LUT, puis le nombre de registres.

5.2.3.3 Minimisation du nombre de *Logic Element*

Un *LE* (*Logic Element*) est l'élément logique de base dans les FPGA d'Altera. Dans le cadre de nos travaux, nous ciblons des composants constitués de LE composés d'une LUT à quatre entrées et d'une bascule D, comme illustrée par la figure 5.11 (il existe des éléments logiques bien plus complexes, avec des LUT à six entrées). Une LUT à quatre entrées peut mettre en œuvre un multiplexeur à 2 entrées, et une bascule D peut mettre en œuvre un registre. Un LE étant capable de mettre en œuvre un multiplexeur à 2 entrées et un registre, le nombre de LE nécessaires à la mise en œuvre des multiplexeurs et des registres est donc le maximum entre le nombre de LUT et le nombre de registres (équation 5.22). L'équation 5.22 correspond à la fonction de coût appliquée pour minimiser le nombre de LE.

$$TotalLE = \texttt{Max}(TotalRegisters, TotalLUTs) \tag{5.22}$$

Pour l'exemple du graphe de la figure 5.8, les nombres de LE nécessaires pour les solutions 1 et 2 sont reportés dans le tableau 5.1. Le nombre de LE est de six pour la solution 1, et trois pour la solution 2 ; ce qui correspond au nombre de LUT, puisque pour chaque solution, le nombre de LUT est supérieur au nombre de registres.

5.2.3.4 Résultats d'expérimentation

Nous avons appliqué notre technique d'allocation de registres avec les différentes stratégies de minimisation à de nombreux algorithmes. Pour tous les algorithmes testés, le nombre de registres minimal a été prouvé lorsque l'on minimise le nombre de registres. Par contre, lorsque l'on applique la minimisation du nombre de LUT, la solution trouvée n'est jamais prouvée optimale. C'est également le cas lorsqu'on minimise le nombre de LE.

Le tableau 5.4 montre les résultats obtenus en termes de nombre de registres, nombre de LUT et nombre de LE pour les différentes approches de minimisation présentées. La stratégie 1 désigne la minimisation du nombre de registres, la stratégie 2 est la minimisation du nombre de LUT et la stratégie 3 est la minimisation du nombre de LE. Les résultats montrent que la minimisation du nombre de LUT permet de réduire le nombre de LUT pour tous les algorithmes sauf l'*idct*. Pour d'autres algorithmes également, testés mais non mentionnés dans le tableau, le nombre de registres, de LUT et de LE est le même quelle que soit la stratégie de minimisation. C'est le cas des algorithmes SHA, $DES3$, des filtres FIR et IIR, ou encore de la FFT. Par ailleurs, la stratégie 2 permet de réduire le nombre de LUT pour un nombre de registres identique à la stratégie 3. On note également que la

minimisation du nombre de LE fournit les mêmes résultats que la minimisation du nombre de LUT. Ces résultats sont logiques puisque le nombre de LUT est à chaque fois supérieur au nombre de registres.

Algorithmes	Stratégie 1			Stratégie 2			Stratégie 3			économie LUT (%) Stratégie 2/ Stratégie 1
	Reg	LUT	LE	Reg	LUT	LE	Reg	LUT	LE	
IDCT col	3	9	9	3	9	9	3	9	9	0
IDCT row	3	10	10	3	10	10	3	10	10	0
Blowfish encrypt 1	8	42	42	8	40	40	8	40	40	5
Blowfish encrypt 2	8	41	41	8	38	38	8	38	38	8
Cast 128 1	9	41	41	9	33	33	9	33	33	20
Cast 128 2	9	38	38	9	32	32	9	32	32	16
Invert Matrix	13	50	50	13	45	45	13	45	45	10

TAB. 5.4 – Nombre de registres, de LUT, et de LE nécessaires selon les stratégies. La stratégie 1 est la minimisation du nombre de registres, la stratégie 2 est la minimisation du nombre de LUT, et la stratégie 3 est la minimisation du nombre de LE. La stratégie 2 permet d'économiser des LUT par rapport à la stratégie 1.

La figure 5.17 illustre de manière graphique les résultats du tableau 5.4. La figure 5.17 montre bien qu'il est possible, pour un même nombre de registres, de réduire le nombre de LUT nécessaires à la mise en œuvre des multiplexeurs. Le tableau 5.4 montre également le gain en LUT (en pourcentage) de la stratégie 2 par rapport à la stratégie 1. Comme la stratégie 3 donne les mêmes résultats que la stratégie 2, le pourcentage de LUT économisées par la stratégie 3 par rapport à la stratégie 1 est le même. On estime à 20% l'économie pour le premier graphe de l'algorithme Cast 128. La stratégie 1 nécessite 41 LUT pour mettre en œuvre les multiplexeurs alors que la stratégie 2 n'en nécessite que 33.

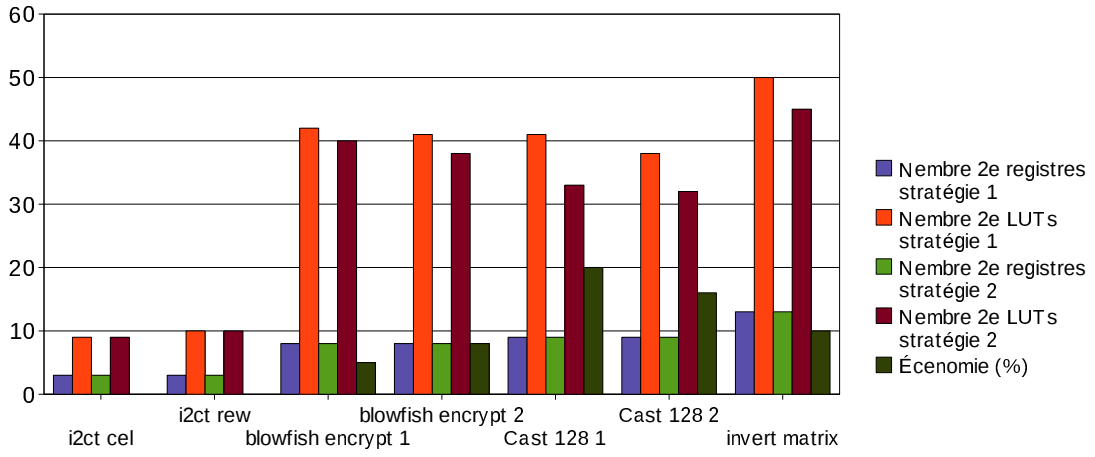


FIG. 5.17 – Comparaison du nombre de LUT selon les stratégies.

Afin de situer nos estimations par rapport aux chiffres réels, nous avons réalisé la synthèse logique de quelques extensions pour les différentes stratégies de minimisation. Le tableau 5.5 montre les résultats de la synthèse logique des extensions générées pour les

algorithmes *Invert Matrix*, *Cast* 128 et *Blowfish*. La synthèse et le placement routage ont été effectués par l'outil Quartus II, avec pour cible un FPGA Cyclone II d'Altera. Le tableau résume le nombre de LE utilisés pour mettre en œuvre le bloc logique spécialisé seul (sans les composants instructions spécialisées), et détaille leur utilisation : soit en LUT ou en registre seul, soit en LUT et registre. Les résultats montrent que notre technique de minimisation du nombre de LUT permet d'économiser 19% de LE pour l'algorithme *Invert Matrix* et jusqu'à 35% pour l'algorithme *Cast* 128 et l'algorithme *Blowfish*.

Algorithmes	Stratégie 1				Stratégie 2				économie
	Logic Elements	LUT only	Register only	LUT/ Register	Logic Elements	LUT only	Register only	LUT/ Register	(%)
Invert Matrix	2711	2315	12	384	2212	1814	34	364	**19**
Cast 128	1454	1230	0	224	942	782	0	160	**35**
Blowfish	1436	1148	5	265	925	669	29	207	**35**

TAB. 5.5 – Résultat de la synthèse après placement routage

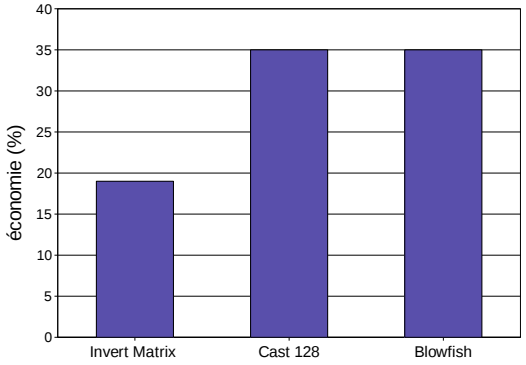


FIG. 5.18 – Économie observée en surface

Il est difficile de comparer nos estimations du nombre de LE par rapport au nombre de LE fourni par l'outil de synthèse pour plusieurs raisons. Premièrement, dans notre modélisation, nous considérons qu'une donnée peut être stockée dans un registre, c'est-à-dire une LUT. Mais en réalité, une donnée est codée sur 32 bits dans le cas du NiosII, ce qui nécessite 32 LUT. Un facteur multiplicatif explique en partie l'écart entre le nombre de LUT estimé et le nombre de LUT réel. La deuxième raison réside dans la difficulté de comparer des choses comparables. En effet, dans nos calculs, nous ne prenons pas en compte, par exemple, le nombre de LE nécessaires à la mise en œuvre du multiplexeur de sortie (sur le signal *result*) puisque ce multiplexeur n'est pas affecté par nos techniques de minimisation. L'outil de synthèse nous fournit le nombre global de LE utilisés pour le bloc logique entier alors que nos calculs se focalisent sur une partie seulement du bloc logique. Une troisième raison a trait aux optimisations propres à l'outil de synthèse utilisé. Un outil de synthèse est capable de détecter des optimisations au moment du placement routage. Par ailleurs, le placement routage est différent à chaque synthèse, ce qui donne des résultats différents à chaque fois.

Concernant les registres, plusieurs solutions s'offrent à l'outil de synthèse. Un registre

peut être mis en œuvre dans un LE comme mentionné auparavant, mais il peut également être mis en œuvre dans un bloc DSP par exemple, ou d'autres blocs mémoires, selon le composant cible. Pour pouvoir proposer une méthode précise, cette méthode doit alors être dépendante du composant cible.

Il est donc difficile de proposer une technique d'allocation de registres avec minimisation de la surface à la fois précise et générique. Cependant, on observe que, malgré une solution non-optimale, notre technique permet d'économiser jusqu'à 35% de ressources utilisées à la mise en œuvre de l'extension. On peut en conclure qu'il est possible d'allouer les registres de manière *intelligente* de façon à réduire la complexité et la surface de l'extension. Cette technique prometteuse peut faire l'objet d'améliorations.

5.3 Génération des codes d'opération et de la description de l'architecture

5.3.1 Génération des codes d'opérations

Une fois l'allocation de registres effectuée, il est possible de définir les codes d'opérations (*opcode*) qui vont contrôler l'instruction spécialisée à exécuter et préciser où les données doivent être lues et écrites. Une instruction qui lit et écrit les données toujours au même endroit nécessite un seul code d'opération. On définit un contexte d'exécution comme étant l'instruction spécialisée à exécuter plus les informations de provenance et de destination des données. De manière générale, le nombre de codes d'opération nécessaires à chaque motif est égal au nombre de contextes d'exécution différents du motif.

Pour chaque motif, nous regardons tous ces contextes d'exécution. Un code d'opération est attribué au premier contexte d'exécution. Ensuite, si un des autres contextes d'exécution est différent, c'est-à-dire, si les données ne proviennent pas des mêmes sources, ou bien si les résultats ne sont pas renvoyés vers les mêmes destinations, un nouveau code d'opération est affecté à ce contexte d'exécution.

5.3.2 Génération du bloc logique spécialisé

Nous disposons à présent de toutes les informations nécessaires pour générer l'architecture de l'extension. Nous connaissons les instructions spécialisées à mettre en œuvre, le nombre de registres et les interconnexions entre ces différents éléments. Nous savons quelle instruction exécuter d'après le code d'opération.

Nous avons développé un générateur d'architecture qui décrit, à partir de ces informations, l'architecture de l'extension dans un fichier VHDL. Ce module instancie chaque instruction spécialisée, vue comme un composant. Les registres sont mis en œuvre sous forme de `signaux` et mis à jour dans un `process`. L'indice n permet de contrôler les différentes connexions entres les composants instructions spécialisées et les registres.

5.3.3 Génération du chemin de donnée combinatoire des instructions spécialisées

Chaque instruction spécialisée, représentée par un motif, est mise en œuvre par un CIS (*Composant Instruction Spécialisée*). Pour générer automatiquement chaque composant, nous avons développé un générateur d'architecture qui, à partir de l'ensemble des motifs sélectionnés, génère un fichier VHDL contenant tous les composants. Chaque composant est décrit sous la forme d'un chemin de données combinatoire pur. Nous n'adressons pas le problème de gestion automatique du pipeline pour l'exécution d'une instruction multi-cycle.

5.3.4 Fusion du chemin de données

Nous avons supposé jusqu'ici des modèles d'architecture où tous les motifs sont mis en œuvre indépendemment, chaque motif possédant ses propres ressources. Par ailleurs, dans notre contexte, deux instructions spécialisées ne peuvent pas être exécutées en même temps. Afin d'économiser en surface, il est possible de fusionner les chemins de données et d'appliquer un partage de ressources. La figure 5.19 illustre le modèle d'architecture B où les chemins de données des motifs sont fusionnés.

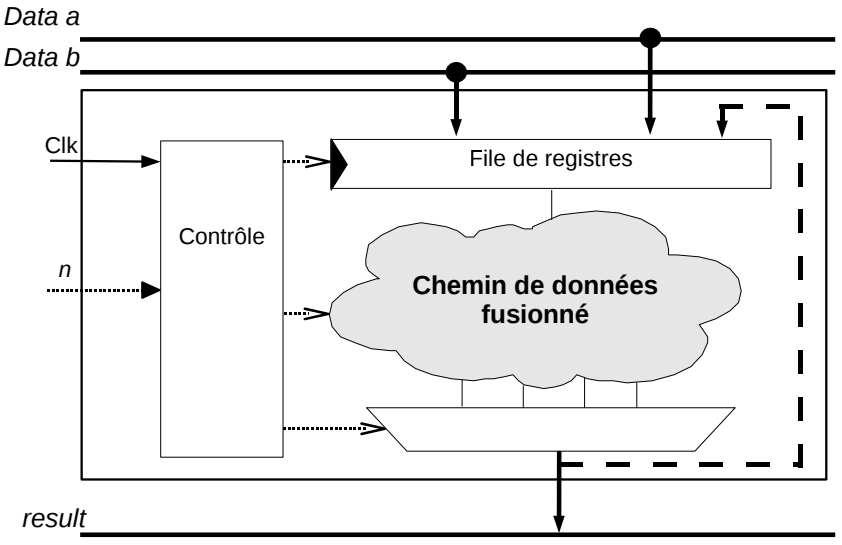


FIG. 5.19 – Fusion des chemins de données dans l'extension

Il existe de nombreuses publications à ce sujet [49, 63, 144, 159]. Au sein même de l'équipe, nous avons utilisé la programmation par contraintes pour apporter une solution à ce problème [208, 209]. Cet aspect, qui n'est pas détaillé dans ce document, fait l'objet de la thèse d'Erwan Raffin.

5.4 Génération du code source modifié

L'étape de modification de code est l'étape qui permet à une application d'exploiter les instructions spécialisées. Nous avons vu que modifier le compilateur pour ajouter les instructions dans la phase de sélection de code est compliqué à mettre en œuvre. Dans le cadre de nos expérimentations nous utilisons un NiosII et il est possible d'appeler une instruction spécialisée du NiosII par l'intermédiaire de *Macros* [19]. La correspondance entre les *Macros* et les instructions spécialisées est réalisée par le biais de fonctions intrinsèques (*Built-in Functions*). Il est également possible d'accéder aux instructions spécialisées directement par du code assembleur.

Dans notre contexte d'ordonnancement avec recalage d'instructions, il est important que le code assembleur généré corresponde exactement à nos résultats d'ordonnancement. C'est pourquoi nous avons choisi d'insérer directement dans le programme C le code assembleur correspondant aux portions de flot de données traitées. La génération du code adapté est réalisée à partir du graphe réduit et des informations d'ordonnancement.

Pour générer du code C avec assembleur inséré, nous nous appuyons sur le générateur de code C fournit par Gecos [89]. Ce générateur génère du code C à partir d'un graphe sous forme de CDFG. La première étape est donc de transformer le HCDG réduit en un CDFG dans lequel les blocs de base contiennent les instructions assembleur. Pour cela, on utilise la passe de transformation de HCDG vers CDFG. Cette passe a fait l'objet d'un travail d'ingénierie, au sein de l'équipe, auquel j'ai participé pour l'insertion des instructions assembleur spécifiques au NiosII. Puis, le générateur de code C a été enrichi pour prendre en compte ces instructions.

5.5 Génération pour la simulation SystemC

Dans le but de valider à la fois l'aspect fonctionnel du code généré et mesurer l'impact des instructions spécialisées retenues, il est possible de s'appuyer sur des techniques de simulation. Dans ce contexte, nous cherchons à nous raccrocher à la simulation SystemC avec SoCLib.

5.5.1 SystemC

SystemC est un langage de conception de système introduit pour améliorer la productivité de la conception des systèmes électroniques [39]. SystemC est un langage dérivé du langage C++, donc un langage comportemental, qui a été enrichi par des notions de signaux, synchronisation, front d'horloge, et processus concurrents.

SystemC permet aux concepteurs de disposer de modèles de composants matériels et logiciels à un haut niveau d'abstraction. Ce haut niveau d'abstraction permet de comprendre dès le début de la conception les interactions entre les différents éléments, de trouver les meilleurs compromis et de procéder rapidement à des tests et des vérifications à travers la simulation. Certes, le développement d'un modèle de composant a un certain

coût mais la productivité s'en trouve rapidement améliorée à travers la réutilisation des modèles de composants.

5.5.2 SoCLib

Le projet SoCLib (*System On Chip Library*) [182] s'inscrit dans cette lignée de disposer d'un ensemble de modèles de composants afin d'élaborer rapidement des plates-formes de prototypage virtuel. SoCLib propose une bibliothèque de modèles de composants et des outils de création de plates-formes complètes. Pour chaque composant, deux modèles sont disponibles : un modèle *CABA* (*Cycle Accurate Bit Accurate*), précis au bit près et au cycle près, et un modèle au niveau transfert de données *TLM* (*Transfer Level Model*) avec temps. La simulation au niveau *CABA* permet de simuler finement le comportement d'un composant, mais les temps de simulation sont parfois très longs. La modélisation au niveau transfert permet d'accélérer les temps de simulation et de procéder rapidement à une validation fonctionnelle de la plate-forme.

Dans le cadre du projet SoCLib, l'équipe a développé un modèle du processeur NiosII. Dans notre contexte d'extension de jeu d'instructions pour un processeur NiosII, l'idée est de pouvoir générer un modèle des instructions spécialisées pour créer un modèle du NiosII étendu et utiliser ce modèle dans une plate-forme SoCLib. La simulation du processeur avec son extension permet de valider fonctionnellement à la fois les aspects logiciels (le code applicatif) et matériels (les instructions spécialisées).

5.5.3 Génération des instructions spécialisées pour le modèle du NiosII

L'étape de génération des instructions spécialisées en SystemC ressemble fortement à la génération des chemins de données combinatoires des motifs, au détail près que le langage à générer est du langage C. Nous nous appuyons sur le générateur de code C existant, en représentant chaque motif sous la forme d'un CDFG.

5.6 Conclusion

Nous avons présenté dans ce chapitre les étapes de génération d'architecture et d'adaptation du code source. Nous avons vu comment générer l'architecture de l'extension. Nous avons créé un bloc logique spécialisé unique qui met en œuvre toutes les instructions spécialisées et qui respecte l'interface avec le processeur. Ce bloc logique contient des registres si besoin et interconnecte tous les éléments entre eux. Dans le cas où des registres sont nécessaires, nous avons alloué les registres de façon à minimiser soit le nombre de registres, soit le nombre de LUT qui mettent en œuvre les multiplexeurs, soit le nombre de LE qui mettent en œuvre à la fois les registres et les multiplexeurs. Nous avons montré qu'il est possible d'allouer les registres de façon à réduire la complexité et la surface de l'extension. Notre technique permet une économie de ressources allant jusqu'à 35%. Chaque instruction spécialisée est mise en œuvre indépendemment sous la forme d'un chemin de

données combinatoire pur. Les techniques de fusion de chemin de données pourront par la suite être appliquées pour encore économiser en surface de silicium.

L'étape d'adaptation du code génère le code pour un NiosII qui exploite les instructions spécialisées à travers des instructions assembleur « inline ».

Pour valider fonctionnellement l'architecture et le code adapté, des modèles des instructions spécialisées sont générés pour le modèle de processeur NiosII et la simulation est réalisée en utilisant SoCLib.

6

Déroulement du flot appliqué à l'algorithme GOST

Les étapes de génération automatique d'extensions de jeux d'instructions ont été présentées dans les chapitres 3, 4 et 5. Ce chapitre revient sur le flot global de conception mis en œuvre et présente l'application de ce flot à un algorithme de chiffrement (algorithme GOST). L'extension et le code applicatif générés pour cet exemple sont validés par la simulation.

Dans un premier temps, nous présentons notre flot de conception *DURASE* et l'environnement Gecos [89]. Puis, nous déroulons ce flot de conception étape par étape, appliqué à l'algorithme de chiffrement GOST. Ensuite, l'ensemble du processus est validé et les performances sont mesurées par la simulation. Enfin, nous proposons une architecture adaptée à ce type d'algorithme.

6.1 Le flot de conception *DURASE*

La figure 6.1 présente le flot de compilation *DURASE (generic environment for the Design and Utilisation of Reconfigurable Application Specific Extensions)*. Ce flot est un flot de compilation générique pour la conception et l'utilisation d'extensions reconfigurables dédiées. Le flot prend en entrée le code source de l'application écrit en langage C, le jeu d'instructions du processeur cible et le modèle d'architecture. Il génère en sortie une extension pour le processeur cible et le code source transformé qui exploite les instructions spécialisées mises en œuvre dans cette extension.

Le flot de compilation *DURASE* est un flot générique qui intègre l'aspect de fusion de chemin de données des motifs qui sort du cadre de cette thèse.

La figure 6.1 illustre par ailleurs les liens avec les outils de simulation comme SoCLib et les outils Altera qui assurent la synthèse de la nouvelle architecture et la programmation

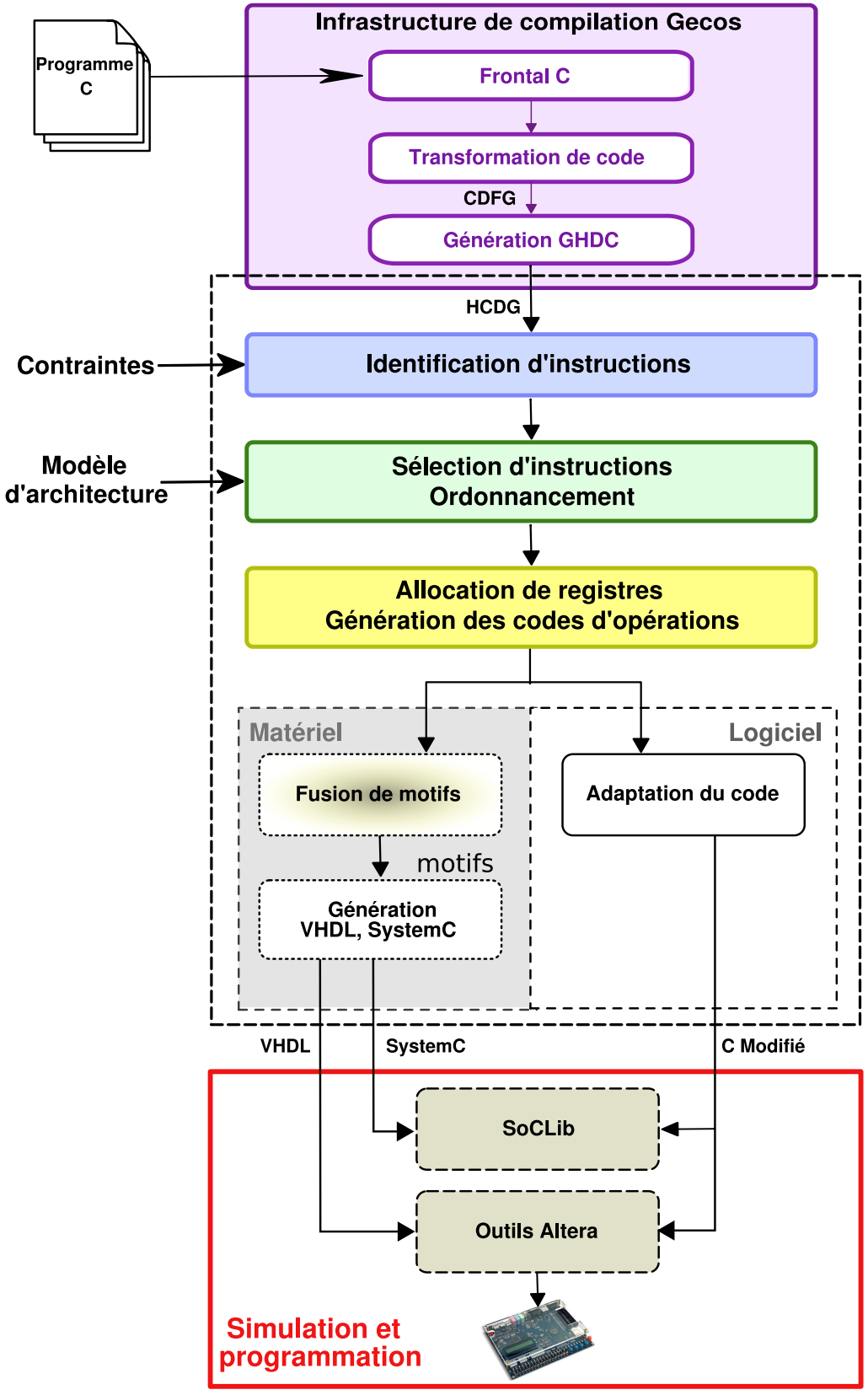


FIG. 6.1 – Flot de compilation *DURASE*

du composant. Le cadre pointillé entoure les contributions de cette thèse et les outils développés. Il est important de noter que le flot est entièrement automatisé, du fichier `c` en entrée jusqu'à la génération du fichier VHDL qui décrit l'architecture, du fichier `c` qui contient les modèles des instructions spécialisées en SystemC, et du fichier `c` contenant le code applicatif qui exploite les instructions spécialisées. Une intervention manuelle est nécessaire pour l'utilisation des outils Altera ou de SoCLib.

6.1.1 Gecos

Le flot de compilation *DURASE* s'appuie sur la plate-forme générique de compilation Gecos [89] *(Generic Compiler Suite)*. Le flot de compilation est contrôlé par un système de script qui appelle successivement des *passes*, des étapes de transformation appliquées à la représentation interne. Voici une liste non exhaustive des principales passes utilisées :

- `CFrontEnd` : frontal C qui construit un CDFG à partir d'un programme C.
- `RemoveCSpecific` : transforme les écritures spécifiques au langage C en écriture standard : pré et post incrémentation, décrémentation (par exemple, `i++` en `i = i + 1`).
- `ConstantEvaluator` : évaluation des constantes.
- `ConstantPropagator` : propagation des constantes.
- `AlgebraicSimplifier` : simplificateur algébrique qui retire les opérations neutres (addition de 0, multiplication par 1, etc).
- `ForUnroller` : déroulage total de boucle.
- `hcdg_function_set_builder` : transformation d'un ensemble de procédures CDFG en un ensemble de fonctions HCDG.
- `ProcedureSetBuilder` : transformation d'un ensemble de fonctions HCDG en un ensemble de procédures CDFG.
- `CGenerator` : générateur de code C à partir d'un CDFG.

6.1.1.1 Exemple de script

La figure 6.2 montre le script Gecos qui met en œuvre la chaîne de compilation *DURASE*. Le langage de script Gecos est un langage non typé avec déclaration implicite des variables.

La passe `CFrontEnd` (ligne 1) construit un ensemble de procédures à partir du code C, donné en premier argument par l'intermédiaire de `$1`. L'ensemble des procédures est stocké dans la variable `ps`, pour *ProcedureSet*. Une procédure représente une fonction C du programme sous la forme d'un CDFG. Ensuite, des transformations de code sont appliquées à chaque procédure.

Pour chaque procédure (ligne 3), la passe `RemoveCSpecific` transforme les expressions spécifiques au langage C (comme les pré et post incrémentations/décrémentations) en expressions classiques. Pour pouvoir appliquer les passes d'évaluation et de propagation de constantes, il faut une représentation en mode *SSA (Single Static Assignement)*, dans

```
1  ps = CFrontEnd($1);
2
3  for proc in ps do
4      RemoveCSpecific(proc);
5
6      SSAAnalyser(proc);
7      do
8          ConstantPropagator(proc);
9          ConstantEvaluator(proc);
10     while changing;
11     RemovePhyNode(proc);
12
13     AlgebraicSimplifier(proc);
14 done;
15
16 functionSet = hcdg_function_set_builder(ps);
17 filename = FindFileName($1);
18
19 nios2FunctionSet = Nios2ExtensionHcdgFunctionSetCreator(functionSet,
20                       {"INmax"->3,"OUTmax"->1,"CC"->1}, 3, filename);
21
22 Nios2ExtensionVHDLGenerator(nios2FunctionSet,filename);
23 Nios2SystemCGenerator(nios2FunctionSet,filename);
24
25 nios2ps = Nios2ProcedureSetBuilder(nios2FunctionSet);
26 Nios2AsmInlineCGenerator(nios2ps,filename);
```

FIG. 6.2 – Chaîne de compilation sous forme de script Gecos

laquelle toutes les variables ont une affectation unique [66]. La passe `SSAAnalyser` (ligne 6) permet de passer dans le mode *SSA*, et la passe `RemovePhyNode` (ligne 11) permet d'en sortir.

Les passes de propagation et d'évaluation de constantes sont appliquées *tant qu'elles provoquent un changement*, boucle `do ... while changing;`. En effet, une première application d'une évaluation de constantes peut amener à d'autres simplifications possibles. Les passes sont appliquées tant qu'elles provoquent un changement de la représentation.

Enfin, la passe de simplification algébrique, `AlgebraicSimplifier` (ligne 13), élimine les opérations neutres, comme par exemple, l'addition d'un 0.

Une fois les transformations de code appliquées, nous transformons l'ensemble de procédures CDFG en un ensemble de fonctions HCDG par l'intermédiaire de la passe `hcdg_function_set_builder` (ligne 16).

Cet ensemble de fonctions HCDG est transmis à la passe de création d'une extension pour un NIOSII (ligne 19), appelée `Nios2ExtensionHcdgFunctionSetCreator`. Cette passe réalise les étapes de génération de motifs, sélection de motifs, ordonnancement et allocation de registres. Les autres paramètres de cette passe de création d'extension sont les contraintes sur les motifs et un seuil qui indique le nombre de nœuds minimum des composantes flot de données à traiter. La passe de création d'une extension pour un NIOSII fournit en sortie un ensemble de fonctions spécifiques au NIOSII, `nios2functionSet`.

Cet ensemble de fonctions peut alors être transmis aux passes de génération d'architecture, `Nios2ExtensionVHDLGenerator` (ligne 22), de génération de modèles SystemC

`Nios2SystemCGenerator` (ligne 23), et d'adaptation du code, `Nios2ProcedureSetBuilder` (ligne 25). La passe `Nios2AsmInlineCGenerator` (ligne 26) écrit dans un fichier le code C adapté avec assembleur « inline » pour un NiosII avec son extension.

6.2 L'algorithme de chiffrement GOST

GOST est un algorithme de chiffrement par bloc inventé en Union Soviétique et standardisé en 1990 sous l'appellation GOST 28147-89. Son nom provient de *GOsudarstvennyi STandard* qui signifie « standard gouvernemental ».

L'algorithme s'appuie sur un réseau de Feistel [152], et travaille sur des blocs de 64 bits et avec une clé de 256 bits. GOST utilise des boîtes de substitution (*S-Boxes*). Comme son nom l'indique, une boîte de substitution substitue une variable (codée sur n bits) par une autre variable (codée sur m bits), ajoutant ainsi de la confusion dans le chiffrement, et permet de casser la linéarité de la structure de chiffrement. Le réseau de Feistel comporte 32 rondes, et chaque ronde consiste à :

1. additionner une sous-clé de 32 bits (modulo 2^{32}),
2. appliquer une série de substitutions par les S-boxes,
3. décaler le résultat sur la gauche de 11 bits.

Le benchmark mcrypt [150] propose une implémentation de l'algorithme en langage C. La figure 6.3 montre la fonction `f` qui applique les étapes 2 et 3 de chaque ronde, la substitution par les S-boxes et la rotation à gauche de 11 bits. Les S-boxes sont implémentées sous forme de tableaux, nommés `gost_87`, `gost_65`, `gost_43`, et `gost_21`, et déclarés en variable globale.

```
1  unsigned int f(unsigned int x)
2  {
3      unsigned int temp;
4      /* Do substitutions */
5      x = gost_k87[x >> 24 & 255] << 24 | gost_k65[x >> 16 & 255] << 16 |
6          gost_k43[x >> 8 & 255] << 8 | gost_k21[x & 255];
7
8      /* Rotate left 11 bits */
9      temp = x << 11 | x >> (32 - 11);
10     return temp;
11
12 }
```

Fig. 6.3 – Code source de GOST tel que issu de mcrypt [150]

6.3 Déroulement du flot étape par étape appliqué à l'algorithme de chiffrement GOST

Nous proposons d'appliquer notre flot en vue de la conception d'une extension pour un processeur NiosII pour l'algorithme de chiffrement GOST. Nous illustrons chaque étape pour la fonction illustrée par la figure 6.3.

6.3.1 Paramètres d'entrée

Le processeur cible est un processeur NiosII, mis en œuvre dans un FPGA d'Altera de la famille des StratixII, cadencé à 200 MHz. Les chiffres pour les latences matérielles et logicielles sont ceux fournis à la section 3.5.1, p. 78. Le modèle d'architecture de l'extension est un modèle B, c'est-à-dire que l'extension dispose d'une file de registres interne.

Le script Gecos utilisé est celui présenté figure 6.2. Les contraintes de génération de motifs portent seulement sur le nombre d'entrées et de sorties (3 entrées, 1 sortie), et nous utilisons la contrainte de connexité dite « classique ». Les composantes conservées sont celles qui possèdent plus de trois nœuds.

6.3.2 Partie frontale

La partie frontale consiste en toutes les étapes qui précèdent la génération d'extensions. Dans un premier temps, chaque fonction C du programme en entrée est transformée en une représentation intermédiaire sous forme de CDFG. Puis des transformations sont appliquées à ce CDFG. Enfin, le CDFG est transformé en HCDG.

6.3.2.1 Transformation C vers CDFG

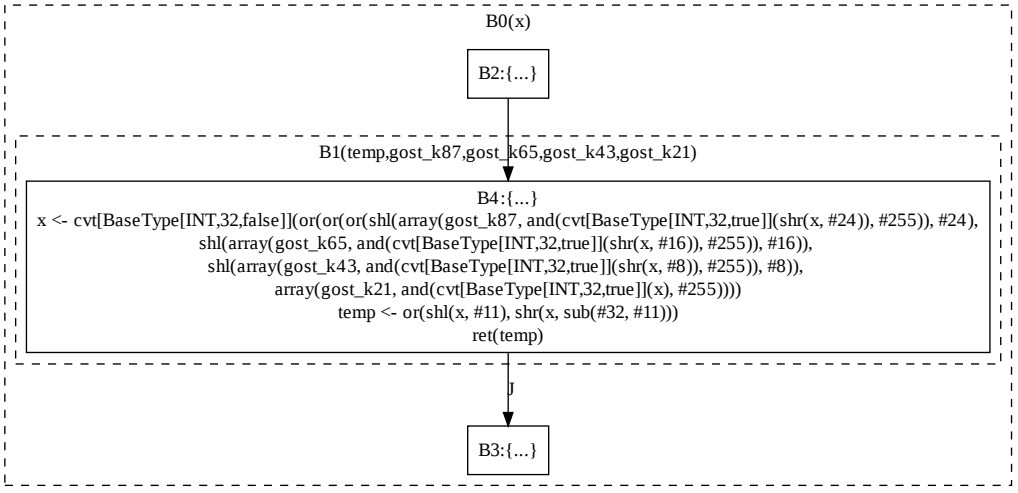


Fig. 6.4 – CDFG du code de la figure 6.3

La transformation du programme C en CDFG est réalisée par la passe Gecos `CFrontEnd` (ligne 1 de la figure 6.2). Pour chaque fonction C, un CDFG est créé. La figure 6.4 montre le CDFG créé à partir de la fonction C de la figure 6.3. La fonction est contenue dans un unique bloc, le bloc `B0` dans la figure. Ce bloc contient :

– un bloc initial qui ne contient aucune instruction mais détermine le point de départ, c'est le bloc `B2` ;
– un bloc final qui ne contient aucune instruction mais détermine le point d'arrivée, c'est le bloc `B3` ;
– un bloc intermédiaire, qui contient le corps de la fonction, c'est le bloc `B1`.

Ce bloc intermédiaire, le bloc `B1`, est lui-même décomposé en sous-blocs, selon la structure du programme. L'exemple de la fonction de la figure 6.3 est un exemple très simple où il n'y a ni instruction conditionnelle, ni boucle. Le bloc `B1` contient donc un seul sous-bloc, le bloc `B4`, qui contient toutes les instructions. Les instructions sont représentées sous forme d'arbres flot de données, mises en forme textuellement par des fonctions binaires. Par exemple, l'opération `shl(x,#11)` représente l'opération de décalage à gauche de la variable `x` de 11 bits (`x << 11` en langage C).

La passe `CFrontEnd` renvoie un objet de type `ProcedureSet`, un ensemble de procédures, où chaque procédure est un CDFG qui correspond à une fonction du programme C. Dans l'exemple, l'ensemble de procédures est stocké par la variable `ps`, et ne contient qu'une seule procédure.

6.3.2.2 Transformations de code

À partir du CDFG, il est possible d'appliquer des transformations de code. La boucle `for proc in ps do` (ligne 3) permet de parcourir l'ensemble de procédures. Pour chaque procédure, des transformations sont appliquées.

La première passe de transformation s'appelle `RemoveCSpecific` (ligne 4), et permet d'uniformiser la représentation en transformant les écritures spécifiques au langage C (comme le `i++`) en écriture classique (`i = i + 1`). L'exemple de l'algorithme GOST ne contient pas d'écriture spécifique. Il est important de noter que les passes de transformations agissent directement sur l'objet envoyé en paramètre. La procédure `proc` est envoyée en paramètre de la passe de transformation et les transformations sont appliquées sur cette procédure. La représentation intermédiaire est raffinée à chaque passe.



(a) Avant évaluation de constantes (b) Après évaluation de constantes

Fig. 6.5 – Action de l'évaluation de constantes

Les passes de transformations relatives aux constantes sont appliquées à une représentation sous forme SSA. La représentation sous forme SSA est obtenue par la passe `SSAAnalyser` (ligne 6). La passe `ConstantPropagator` (ligne 8) propage la valeur des variables constantes, et la passe `ConstantEvaluator` (ligne 9) évalue les opérations bi-

naires sur deux valeurs constantes. Puisque la représentation est raffinée par chaque passe, ces deux transformations sont successivement appliquées tant qu'elles provoquent une modification de la représentation. Lorsque la représentation est stabilisée, la passe `RemovePhyNode` (ligne 11) permet de sortir de la forme SSA. L'action de l'évaluation de constantes est illustrée sur cet exemple de l'algorithme GOST par l'évaluation de l'instruction (`(32 - 11)`), à la ligne 9 du code de la figure 6.3, représentée sous la forme `sub(#32,#11)` dans le CDFG. La figure 6.5 montre la transformation appliquée, la figure 6.5(a) avant évaluation, et la figure 6.5(b) après évaluation.

La dernière passe de transformation de code appliquée est la passe de simplification algébrique, appelée `AlgebraicSimplifier`, à la ligne 13 de la figure 6.2. Cette passe de transformation applique des simplifications algébriques et supprime les opérations neutres, comme par exemple l'addition d'un zéro. L'algorithme GOST n'est pas concerné par cette simplification.

6.3.2.3 CDFG vers HCDG

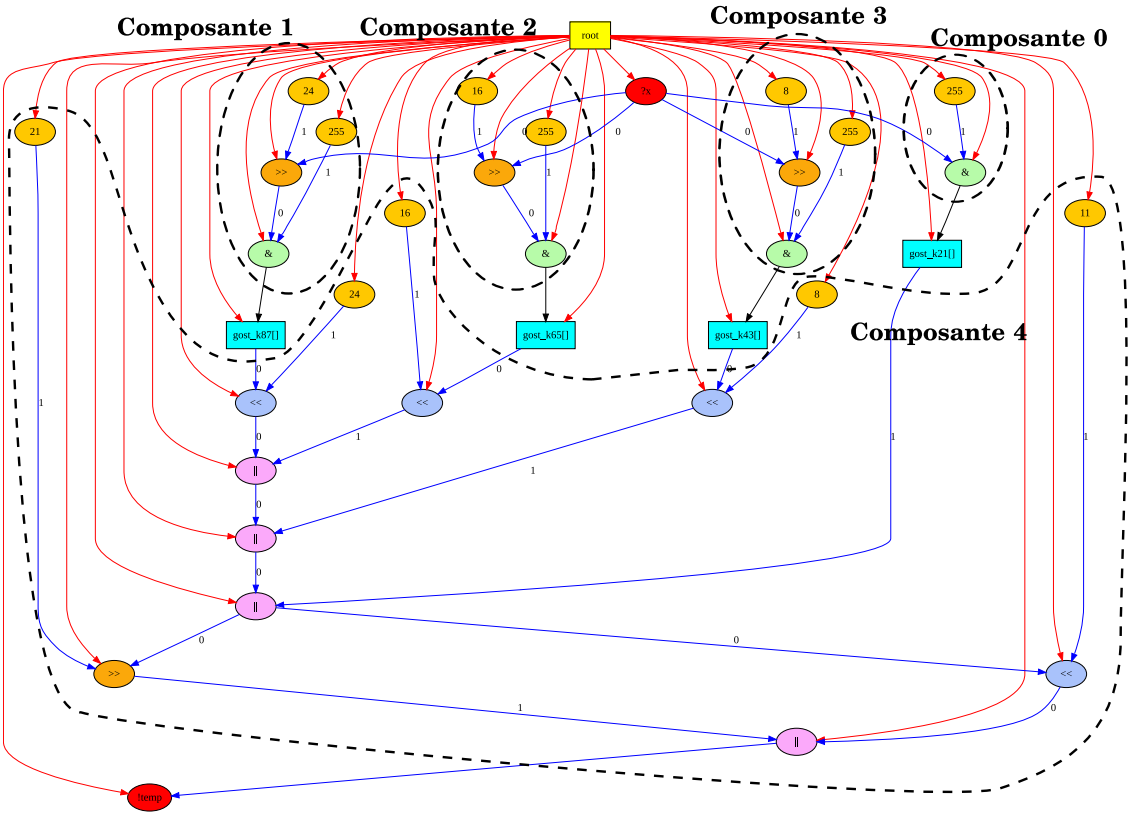


FIG. 6.6 – HCDG de GOST. Cinq composantes flot-de-données apparaissent. La composante 0 est filtrée puisqu'elle n'est composée que de deux nœuds, le seuil étant fixé à 3 nœuds. Les autres composantes sont conservées.

Après avoir appliqué les passes de transformation de code sur chaque procédure, il est possible de transformer un ensemble de procédures, sous forme de CDFG, en un

ensemble de fonctions, sous forme de HCDG. Cette transformation est réalisée par la passe `hcdg_function_set_builder` à la ligne 16, qui produit un ensemble de fonctions (`functionSet`), à partir de l'ensemble de procédures `ps`. La figure 6.6 montre le HCDG généré à partir du CDFG obtenu après transformations de code.

L'algorithme GOST ne contient aucune instruction de contrôle, le HCDG contient donc une seule garde, la garde *racine*, représentée par le rectangle jaune `root`. Les entrées et les sorties sont représentées par les nœuds de couleur rouge, où les nœuds avec les points d'interrogation `?` symbolisent les entrées, comme l'entrée `?x` par exemple, et les points d'exclamation `!` les sorties, comme la sortie `!temp`. Les tableaux sont représentés par des rectangles de couleur turquoise sur la figure 6.6. L'accès à un tableau est symbolisé par un lien d'index, de couleur noire, qui représente l'adresse. Les liens de couleur rouge symbolisent les dépendances de contrôle, et les liens de couleur bleue les dépendances de données. Les numéros sur les liens de données correspondent au numéro de l'opérande du nœud destination. L'ordre des opérandes est important dans le cas d'opérations non commutatives.

À ce stade du processus, nous disposons d'un ensemble de fonctions, chacune sous forme de HCDG et représentant une fonction du programme C fourni en entrée, sur lesquelles des transformations de code classiques de la compilation ont été appliquées. Cet ensemble de fonctions est transmis à la passe de création d'extension pour un NiosII qui constitue le cœur de la contribution de cette thèse.

6.3.3 Création d'une extension

L'objectif de notre étape est de créer une extension, pour un processeur NiosII, qui contient toutes les instructions spécialisées. Une même instruction spécialisée peut se trouver dans plusieurs fonctions du programme initial, l'étape de création d'extension s'applique donc à un ensemble de fonctions `functionSet`, et produit en sortie un ensemble de fonctions spécialisées pour un NiosII, `nios2FunctionSet`.

La passe de création d'une extension pour un NiosII est appelée à la ligne 19 de la figure 6.2 et se nomme `Nios2ExtensionHcdgFunctionSetCreator`. Elle prend quatre paramètres en entrée.

1. Le premier paramètre est l'ensemble de fonctions `functionSet`.
2. Le deuxième paramètre contient les contraintes à imposer lors de la génération de motifs. L'exemple du script de la figure 6.2 présente la contrainte `INmax` qui impose le nombre maximal d'entrées, la contrainte `OUTmax` le nombre maximal de sorties et la contrainte `CC` pour la contrainte de connexité.
3. Le troisième paramètre est un seuil qui permet à l'utilisateur de spécifier le nombre minimum de nœuds que doivent contenir les graphes à traiter.
4. Le quatrième paramètre est simplement le nom à utiliser pour la génération automatique des différents fichiers de sortie.

La création d'une extension consiste en six étapes majeures :

1. Extraction des composantes
2. Génération de motifs
3. Sélection de motifs
4. Ordonnancement
5. Allocation de registres
6. Génération des codes d'opérations

6.3.3.1 Extraction des composantes

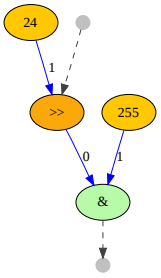


FIG. 6.7 – Composante 1

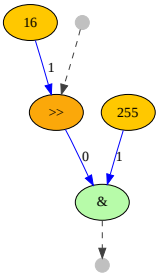


FIG. 6.8 – Composante 2

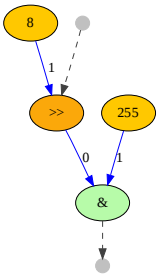


FIG. 6.9 – Composante 3

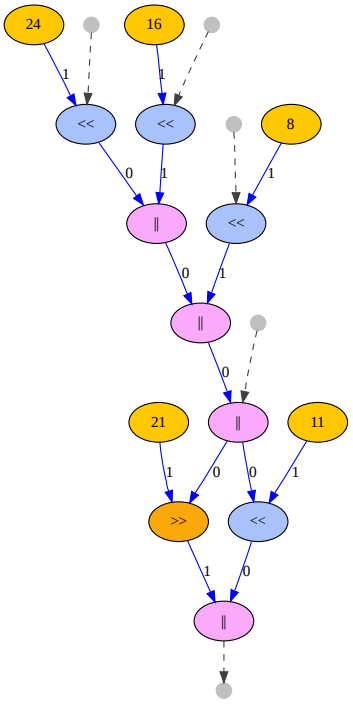


FIG. 6.10 – Composante 4

L'extraction des composantes constitue la première étape. Elle extrait, à partir d'un HCDG, les composantes (c'est-à-dire les sous-graphes) sur lesquelles seront appliquées les étapes suivantes. L'extraction des composantes est paramétrable par un système de filtre, qui spécifie l'inclusion ou l'exclusion d'un nœud du graphe. En fonction de l'architecture cible, tous les types de nœuds ne sont pas forcément adéquats pour une mise en œuvre matérielle. C'est le cas dans notre exemple des nœuds tableau, qui représentent un accès mémoire. Dans notre modèle d'architecture, l'extension ne peut pas accéder à la mémoire globale. Le filtre mis en place permet d'exclure les nœuds de type tableau, et

les gardes. Seules les composantes dont le nombre de nœuds est supérieur au seuil spécifié sont conservées.

Dans l'exemple de l'algorithme GOST, quatre composantes sont extraites. Elles sont illustrées par les figures 6.7, 6.8, 6.9, et 6.10. Les composantes 1, 2, et 3, correspondent à un calcul d'adresse pour l'accès aux données dans les tableaux, et la composante 4 correspond aux opérations de rotation et de combinaison. Les composantes sont mises en évidence dans le HCDG de la figure 6.6. La figure fait apparaître la composante 0, une composante constituée de 2 nœuds seulement et donc filtrée, puisque le seuil est fixé à 3 nœuds.

6.3.3.2 Génération de motifs

`INmax`	Nombre maximum d'entrées
`INmin`	Nombre minimum d'entrées
`OUTmax`	Nombre maximum de sorties
`OUTmin`	Nombre minimum de sorties
`CPmax`	Longueur maximum du chemin critique
`CPmin`	Longueur minimum du chemin critique
`NNmax`	Nombre maximum de nœuds
`NNmin`	Nombre minimum de nœuds
`CC`	Contrainte de connexité 0 : aucune connexité 1 : contrainte classique 2 : contrainte spéciale

Tab. 6.1 – Contraintes pour la génération de motifs

L'algorithme de génération de motifs, présenté figure 3.2 p. 67, est appliqué à chaque composante. Les contraintes imposées apparaissent à la ligne 20 du script de la figure 6.2 et portent sur le nombre d'entrées, le nombre de sorties et la connexité :

- `"INmax"->3`, maximum trois entrées ;
- `"OUTmax"->1`, maximum une sortie ;
- `"CC"->1`, contrainte de connexité « classique », c'est-à-dire la contrainte de connexité exprimée à l'aide de contraintes classiques de la programmation par contraintes.

Lorsque `CC` vaut 1, c'est la contrainte de connexité classique qui est imposée, lorsque `CC` vaut 2, c'est la contrainte de connexité spéciale qui est imposée, et lorsque `CC` vaut 0, aucune contrainte de connexité n'est imposée, comme résumé dans le tableau 6.1. Les différentes contraintes de connexité sont présentées dans la section 3.4.1, p. 70. Le tableau 6.1 résume les différentes contraintes applicables lors de la génération de motifs. Nous donnons la possibilité à l'utilisateur de spécifier non seulement un nombre maximal, mais aussi un nombre minimal sur les différentes contraintes.

La figure 6.11 montre les motifs générés sous contraintes de 4 entrées et 1 sortie, à partir de la composante 1. Les nœuds de forme octogonale représentent les nœuds graines, c'est-à-dire les nœuds à partir desquels les motifs ont été générés. Parmi les six motifs générés, on remarque que le premier (celui tout à gauche sur la figure) correspond à la

composante entière. Ces six motifs ont été générés en environ 60 ms. Pour les composantes 2 et 3, les motifs générés sont sensiblement les mêmes, à la différence près de la valeur de la constante en entrée de l'opération de décalage à droite.

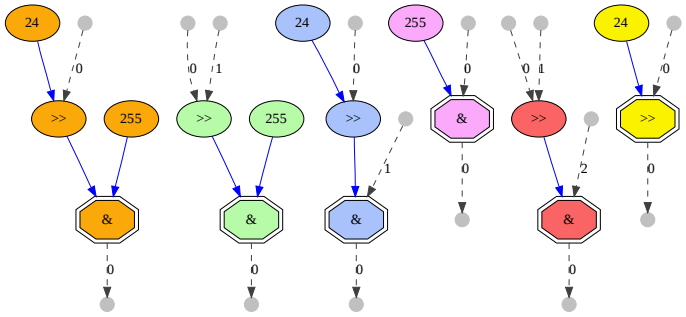


FIG. 6.11 – Motifs générés à partir de la composante 1

Pour la composante 4, 42 motifs ont été générés en 0,8 seconde. La figure 6.12 montre quelques motifs générés pour la composante 4. Pour des raisons de clarté, nous ne montrons pas tous les motifs générés, mais seulement les deux plus gros et quelques motifs de petite taille. Pour les quatre composantes, 48 motifs non isomorphes ont été générés en seulement 1,6 seconde.

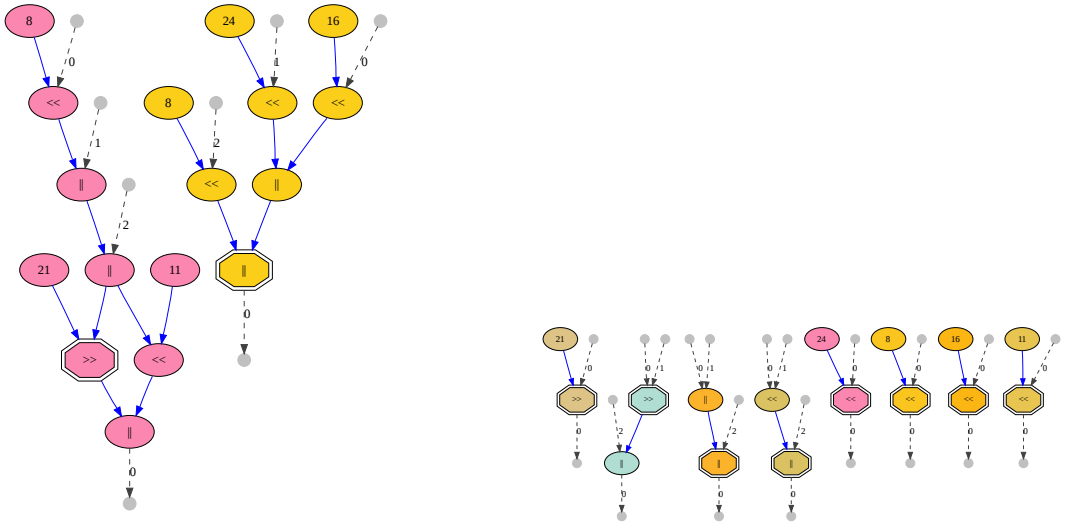


FIG. 6.12 – Quelques motifs générés à partir de la composante 4

6.3.3.3 Sélection de motifs

Avant l'étape de sélection de motifs, nous avons vu à la section 4.1.2.4 p. 87 que nous filtrons certaines occurrences, et en particulier les occurrences avec valeur immédiate. La figure 6.13 présente les motifs candidats après filtrage des occurrences. Nous remarquons que certains motifs ont disparus. En effet, lorsque toutes les occurrences d'un motif dans un graphe cible sont filtrées, ce motif est retiré de la liste des candidats. Les seuls candidats restants sont le motif à quatre nœuds qui correspond à la composante entière, et les motifs de un nœud qui représentent les opérations de base pouvant être exécutées par le processeur. Pour des raisons de clarté, nous ne montrons pas la trentaine de motifs restant après filtrage pour la composante 4.

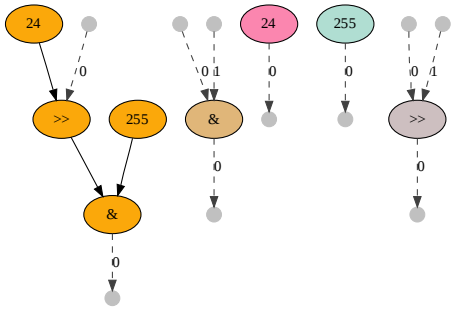


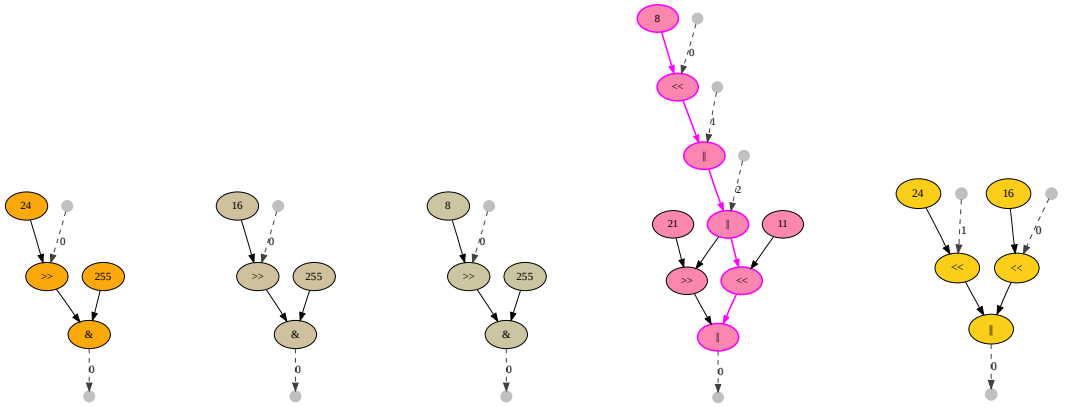
FIG. 6.13 – Motifs filtrés pour la composante 1

Comme expliqué dans le chapitre 4, p. 85, la sélection de motifs est une couverture de graphe avec minimisation du temps d'exécution séquentielle. Par exemple, l'exécution de la composante 1 sur un processeur NIOSII nécessite 4 cycles, trois cycles pour le décalage (`>>`) par une valeur constante, grâce aux multiplieurs embarqués dans les FPGA StratixII (sinon c'est autant de cycles que de décalages) et un cycle pour l'opération binaire `and` (`&`) avec une valeur constante. En matériel, le décalage avec une valeur constante est immédiat puisque les connexions entre signaux sont simplement décalées, et l'opération binaire nécessite 2,2 ns, comme précisé par le tableau 3.2 p. 78. L'exécution de la composante 1 est donc réalisable en seulement un cycle du processeur, grâce à la sélection du motif 1 (figure 6.14(a)), et le facteur d'accélération est de $\frac{4}{1} = 4$.

Le temps d'exécution d'un graphe est la somme des délais des occurrences qui couvrent le graphe, ce qui correspond à une exécution séquentielle de chaque occurrence. Comme expliqué dans la section 4.1.2.2, p. 85, le temps d'exécution est la somme pondérée des occurrences. Nous cherchons à minimiser cette somme.

Le délai d'une occurrence exécutée par le processeur correspond au temps d'exécution logicielle de l'instruction de base. Le délai d'une occurrence exécutée par l'extension est la somme de trois composantes, comme expliqué section 4.1.3, p. 90 :

1. le temps d'exécution de l'occurrence,



(a) Motif 1 (b) Motif 2 (c) Motif 3 (d) Motif 4 (e) Motif 5

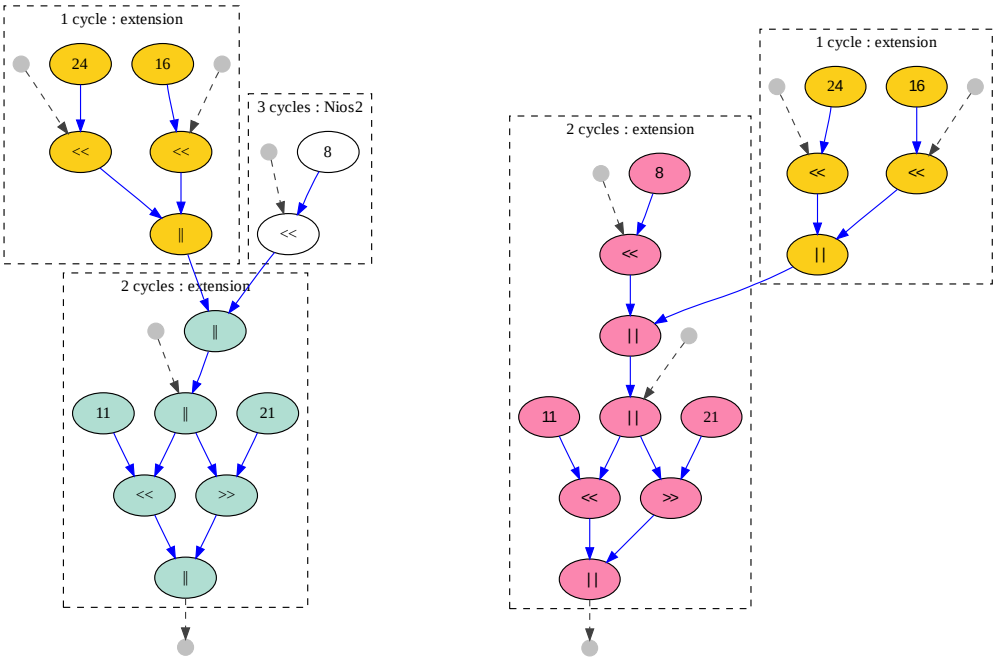
FIG. 6.14 – Les cinq motifs sélectionnés

2. le nombre de cycles pour le transfert de données en entrée,
3. le nombre de cycles pour le transfert de données en sortie.

Pour calculer le temps d'exécution d'une occurrence, il faut calculer le chemin critique de son motif puis le diviser par la période d'horloge du processeur. Dans la figure 6.14(d), le chemin critique du motif est surligné par la couleur violette et est composé de deux nœuds de décalage par une valeur constante (dont le temps de traversée est nul), et de trois nœuds `or`, dont le temps de traversée est de 2,2 ns pour chaque nœud. Le chemin critique du motif est donc de 6,6 ns. Nous supposons que le processeur est cadencé à 200 MHz, la période d'horloge est de 5 ns. L'exécution du motif nécessite $\left\lceil \frac{6,6}{5} \right\rceil = 2$ cycles du processeur.

Le nombre de cycles pour le transfert de données en entrée et en sortie dépend du modèle d'architecture et pour le modèle d'architecture B, ce nombre de cycles varie en fonction du contexte. Ainsi, même si le motif 4 (figure 6.14(d)) possède trois entrées, son occurrence dans la composante 4 (figure 6.15(b)) ne nécessite que deux entrées provenant du processeur (la troisième entrée est déjà présente puisqu'elle est produite par une occurrence exécutée par l'extension et sauvegardée dans la file de registres), ce qui n'ajoute aucune pénalité sur le transfert de données en entrée. Le délai de l'occurrence du motif 4 dans le cas de la figure 6.15(b) est donc de 2 cycles.

La figure 6.15 montre un exemple de deux solutions de couverture de la composante 4. La solution 1 (figure 6.15(a)) propose d'exécuter deux instructions spécialisées sur l'extension et d'exécuter le décalage à gauche de 8 par le processeur, illustré sur la figure 6.15(a) par les nœuds de couleur blanche. Cette solution a un coût de 6 cycles. La solution 2 (figure 6.15(b)) propose également deux instructions spécialisées mais en incluant le décalage à gauche de 8 dans une instruction spécialisée et le coût de cette solution est de trois cycles seulement. La solution 2 est la solution finalement retenue par le solveur de contraintes. Cette solution est la solution optimale : étant données les contraintes, il faut au minimum trois cycles pour exécuter la composante 4. Il existe de nombreuses autres solutions pour



(a) Solution 1 : Coût = 1 + 3 + 2 = 6 cycles

(b) Solution 2 : Coût = 1 + 2 = 3 cycles

FIG. 6.15 – Exemple de couverture et coût du temps d'exécution en nombre de cycles. Le coût est la somme pondérée des occurrences sélectionnées.

exécuter la composante 4 en trois cycles, mais aucune solution ne propose mieux.

Pour exécuter la composante 4, un processeur NiosII nécessite 19 cycles, alors qu'il est possible de l'exécuter en 3 cycles seulement en utilisant des instructions spécialisées telles que proposées par la solution 2 de la figure 6.15(b). Pour la composante 4, le facteur d'accélération atteint $\frac{19}{3} = 6,33$.

Les figures 6.16, 6.17, 6.18, et 6.19 montrent les composantes couvertes. Les composantes 1, 2, et 3 ne sont couvertes que par un seul motif. La sélection est dans notre exemple très simple et effectuée en quelques millisecondes seulement. La preuve de l'optimalité est apportée pour chaque solution retenue.

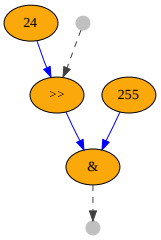


Fig. 6.16 – Composante 1 couverte

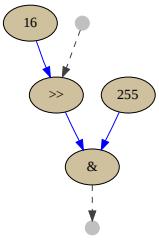


Fig. 6.17 – Composante 2 couverte

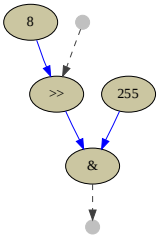


Fig. 6.18 – Composante 3 couverte

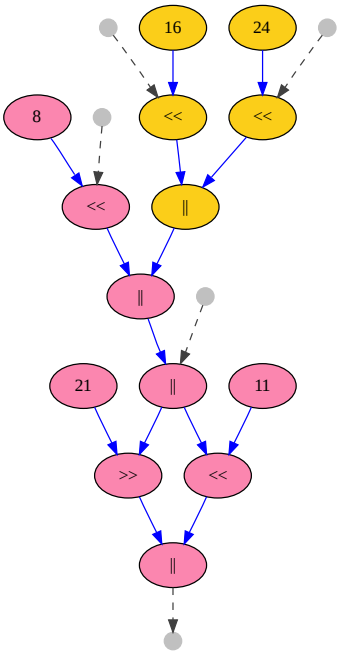


Fig. 6.19 – Composante 4 couverte

La figure 6.20 montre le HCDG couvert par les motifs. Les nœuds grisés sont les nœuds qui ne sont pas concernés par la couverture (c'est-à-dire les nœuds filtrés lors de l'extraction des composantes). La figure 6.14 montre les cinq motifs finalement sélectionnés pour l'exemple de l'algorithme GOST.

6.3.3.4 Ordonnancement

Nous savons que l'exécution séquentielle du graphe couvert nécessite un certain nombre de cycles. La phase d'ordonnancement détermine quelle occurrence exécuter à quel moment, tout en prenant en compte les optimisations sur le transfert de données (cf. section 4.2.4, p. 101) et le recalage d'instructions (cf. section 4.2.3, p. 99). L'ordonnancement

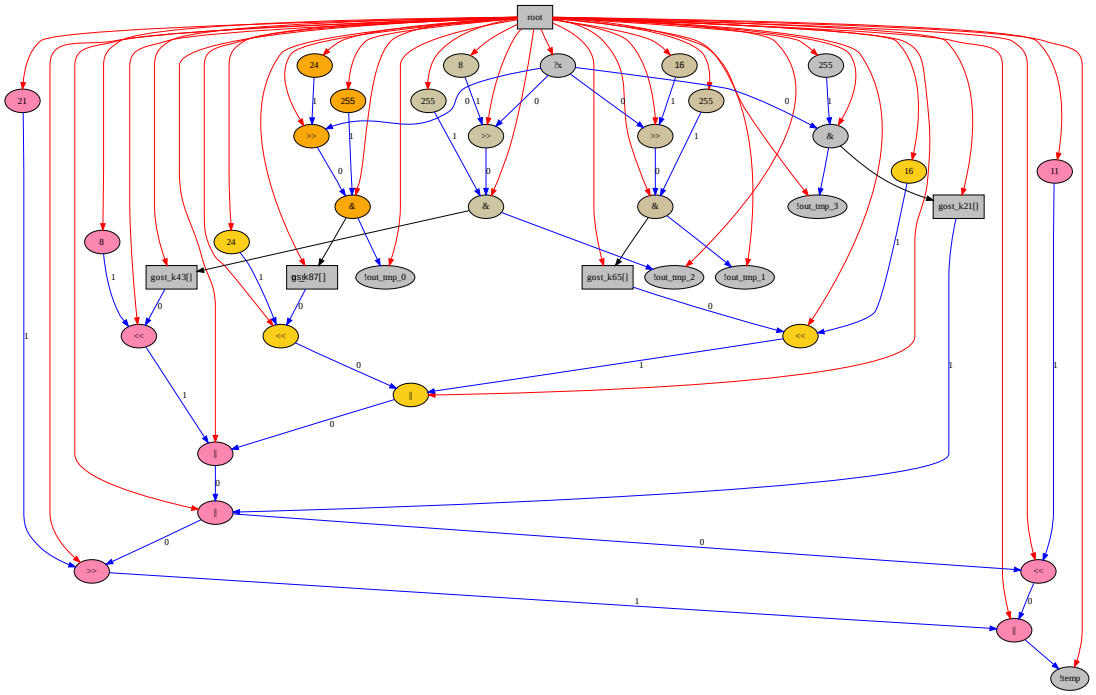


FIG. 6.20 – HCDG couvert de GOST

est appliqué sur un graphe réduit. Un graphe réduit est un graphe dans lequel toutes les occurrences sont réduites à un seul nœud, appelé le nœud « réduit ». Lorsqu'une composante est couverte par une seule occurrence, la composante réduite se résume à un seul nœud. C'est le cas de la figure 6.21 qui montre la composante 1 réduite en un seul nœud appelé `motif_1` et son graphe interne. Le graphe interne du nœud `motif_1` est encadré par la boîte en trait pointillé. La figure 6.22 montre la composante 4 réduite. La figure 6.23 montre le HCDG réduit de GOST, obtenu à partir du graphe couvert de la figure 6.20 et la figure 6.24 est le HCDG réduit de GOST qui fait apparaître les graphes internes.

Chaque composante réduite est ordonnancée séparément. Pour les composantes réduites 1, 2, et 3, l'ordonnancement est immédiat puisqu'il n'y a qu'un seul nœud à ordonnancer. La figure 6.25 montre l'ordonnancement de la composante 4 réduite. Au cycle 0, l'instruction spécialisée représentée par `motif_5` est lancée, avec en opérande `gost_87[out_tmp_1]` (simplifié sur la figure en `gost_87`) et `gost_65`. Ensuite au cycle 1, l'instruction spécialisée `motif_4` est lancée, avec en opérande `gost_43`, et `gost_21`. Le troisième opérande est disponible dans la file de registres de l'extension. L'instruction nécessite 2 cycles. Le résultat `temp` est renvoyé au cycle 2.

6.3.3.5 Allocation de registres

L'allocation de registres ne concerne que la composante 4 dans notre exemple. En effet, les autres composantes ne contiennent qu'une seule occurrence et celle-ci ne nécessite

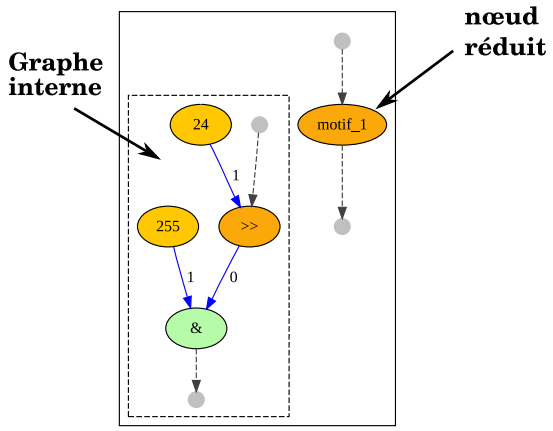


FIG. 6.21 – Composante 1 réduite avec son graphe interne

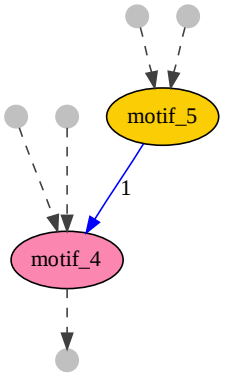


FIG. 6.22 – Composante 4 réduite

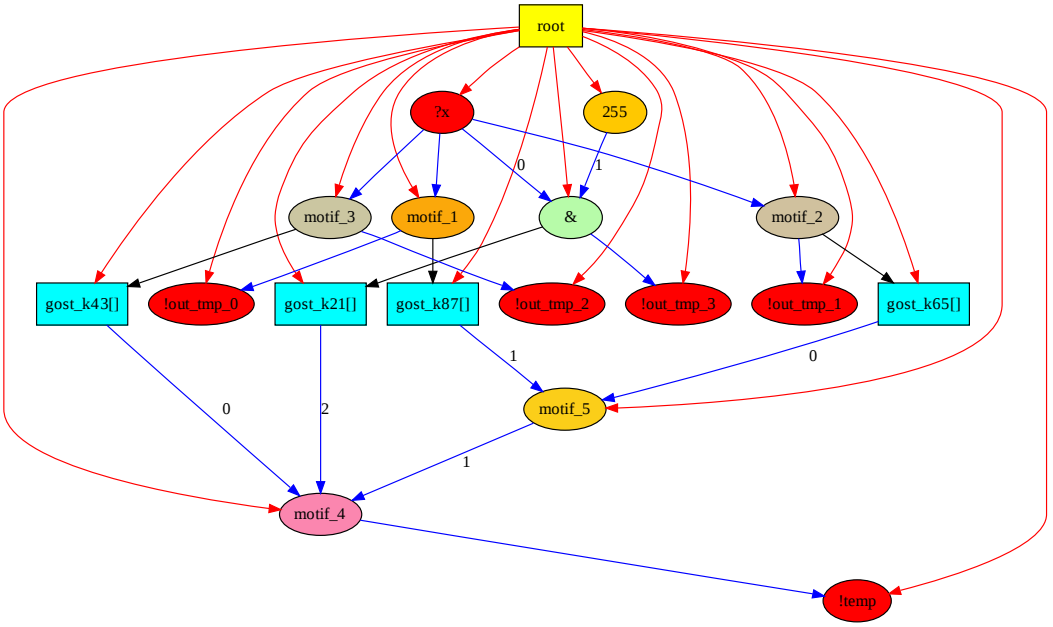


FIG. 6.23 – HCDG réduit de GOST

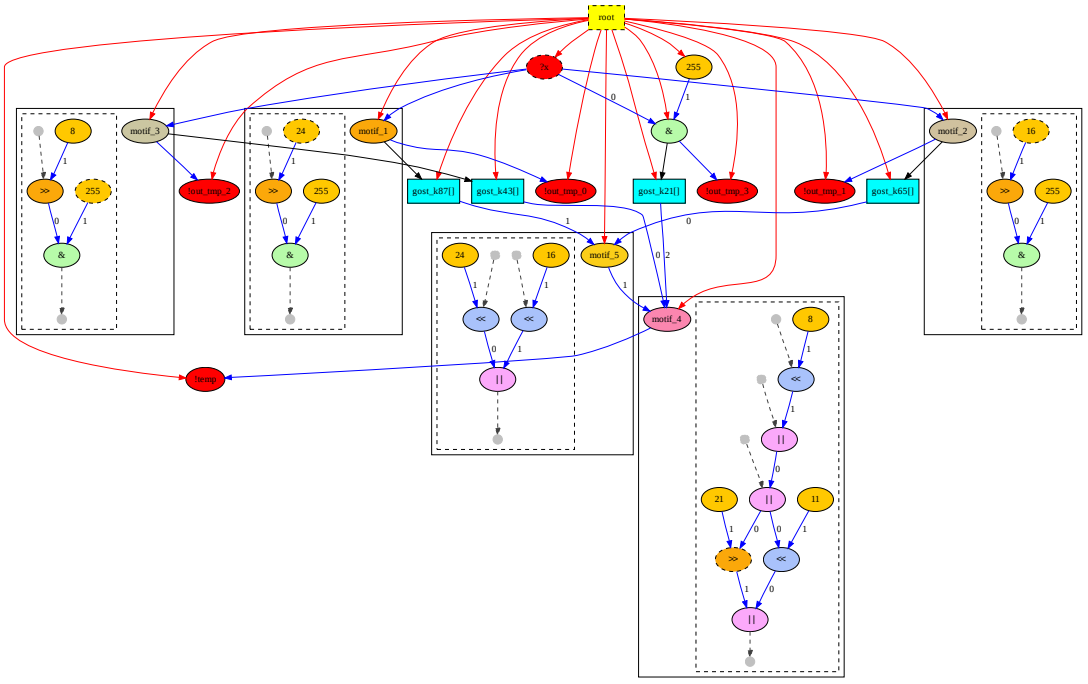


FIG. 6.24 – HCDG réduit de GOST avec graphes internes

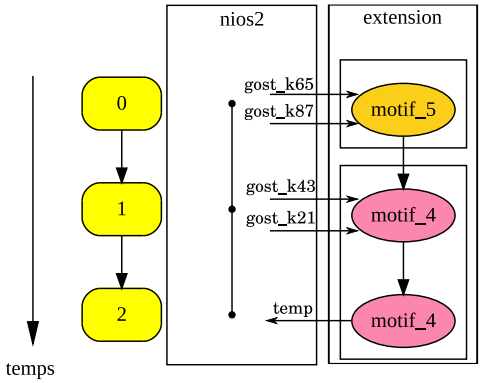


FIG. 6.25 – Ordonnancement de la composante 4

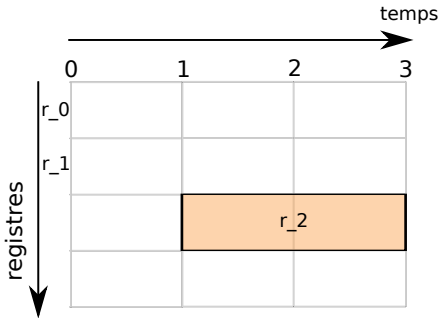


FIG. 6.26 – Allocation de registres pour la composante 4

aucun cycle additionnel de transfert de données. Pour la composante 4, un seul registre est nécessaire, il permet de véhiculer la donnée entre `motif_5` et `motif_4`. La figure 6.26 montre l'utilisation du registre `r_2` au cycle 1. La donnée produite par `motif_5` est disponible dans le registre au début du cycle 1 jusqu'au début du cycle 2. La figure montre également les registres virtuels `r_0` et `r_1`, qui symbolisent les données *dataa* et *datab* en entrée, et *result* en sortie.

6.3.3.6 Génération des codes d'opération

Pour notre exemple, la génération des codes d'opération pour les instructions spécialisées est simple à effectuer puisque chaque motif ne possède qu'une seule occurrence, donc un seul contexte d'exécution. Dans ce cas, chaque motif se voit attribuer un numéro unique. Le nombre de codes d'opération est de 5. Ces 5 codes d'opérations et leur action sont résumés dans le tableau 6.2.

Code d'opération	action
2	exécuter `motif_1` en utilisant *dataa*
3	exécuter `motif_5` en utilisant *dataa*, *datab*, et stocker le résultat dans le registre `r_2`
4	exécuter `motif_4` en utilisant *dataa*, *datab*, et le registre `r_2`
5	exécuter `motif_2` en utilisant *dataa*
6	exécuter `motif_3` en utilisant *dataa*

TAB. 6.2 – Codes d'opérations pour les instructions spécialisées

La fin de l'étape de génération des codes d'opération signe la fin de la passe de création d'une extension pour un processeur NIOSII. L'objet renvoyé en sortie de cette passe est un ensemble de fonctions spécifiques à un NIOSII (`nios2FunctionSet`) et contient toutes les informations d'ordonnancement et d'allocation de registres, les codes d'opérations, et les motifs sélectionnés. Toutes ces informations sont indispensables pour la génération d'architecture et l'adaptation du code.

6.3.4 Génération d'architecture

La passe de génération d'architecture `Nios2ExtensionVHDLGenerator` (ligne 22 de la figure 6.2) consiste à générer deux fichiers VHDL.

6.3.4.1 Fichier des *CIS*

Le premier fichier contient la description architecturale correspondant à chaque motif. Chaque motif est décrit comme un composant, défini comme étant un *Composant Instruction Spécialisée* (*CIS*). Chaque motif est tout d'abord transformé en HCDG, c'est-à-dire que des nœuds d'entrée et sortie, et un nœud de contrôle sont ajoutés et connectés correctement. La figure 6.27 montre le HCDG obtenu à partir du motif 1 (figure 6.14(a)). Ensuite, chaque HCDG est transformé en CDFG. La figure 6.28 montre le CDFG obtenu

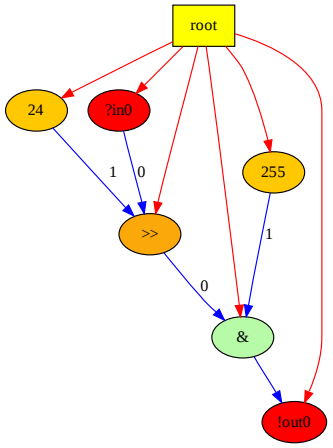


FIG. 6.27 – HCDG du motif 1

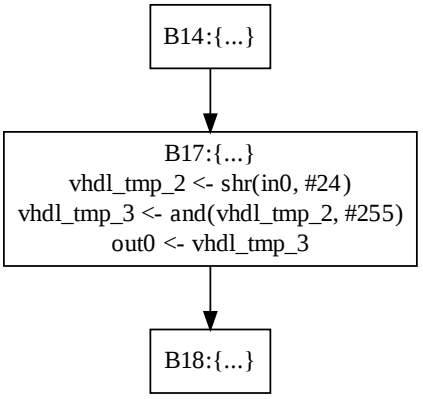


FIG. 6.28 – CDFG du motif 1

à partir du HCDG de la figure 6.27. Enfin, nous avons développé un programme qui permet de générer du VHDL combinatoire pur à partir d'un CDFG. La figure 6.29 montre le code VHDL généré à partir du CDFG de la figure 6.28. Lors d'un décalage par une valeur constante, nous incluons un paquetage appelé `package_shift` qui contient la description des différents décalages avec une valeur constante (ligne 5 de la figure 6.29).

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use ieee.std_logic_signed.all;
5  use work.package_shift.all;
6
7  entity SHRAND_18_FF is
8      port (
9          in0 : in std_logic_vector(31 downto 0);
10         out0 : out std_logic_vector(31 downto 0)  );
11 end entity SHRAND_18_FF;
12
13 architecture arch of SHRAND_18_FF is
14     signal vhdl_tmp_2 : std_logic_vector(31 downto 0);
15     signal vhdl_tmp_3 : std_logic_vector(31 downto 0);
16     constant cst_ff : std_logic_vector(7 downto 0) := "11111111";
17     constant cst_18 : integer := 24;
18 begin
19     vhdl_tmp_2 <= in0 srl cst_18;
20     vhdl_tmp_3 <= vhdl_tmp_2 and cst_ff;
21     out0 <= vhdl_tmp_3(31 downto 0);
22 end architecture arch;
```

FIG. 6.29 – Code VHDL généré pour le motif 1

6.3.4.2 Fichier extension

Le deuxième fichier contient la description architecturale de l'extension. Cette description contient la déclaration et l'instanciation de tous les *CIS*, et la déclaration des

registres, ainsi que l'interconnexion entre les registres et les *CIS*. L'entité de ce composant doit respecter l'interface avec le processeur. La figure 6.31 montre l'entité générée pour l'extension avec les signaux requis pour l'interface avec le processeur NiosII. On remarque que l'indice `n` qui contrôle les instructions spécialisées est codé sur 3 bits, puisqu'il y a cinq codes d'opération différents. La figure 6.30 montre une vue schématique de l'extension générée pour l'exemple de l'algorithme GOST.

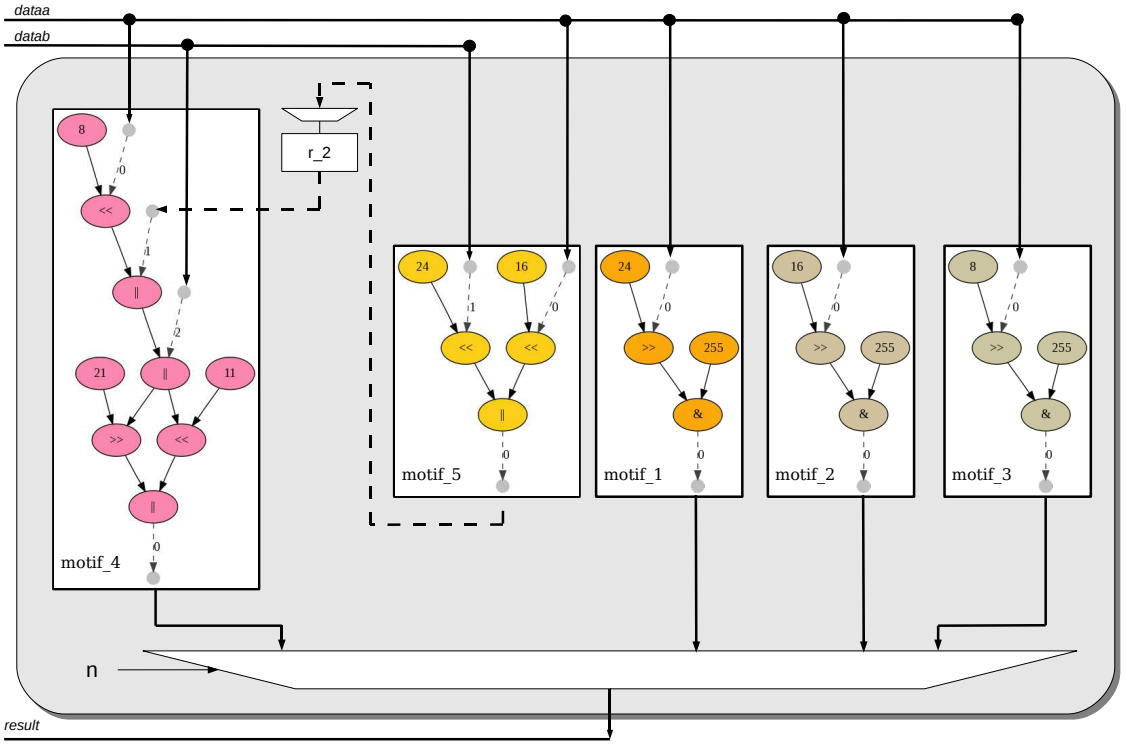


Fig. 6.30 – Extension générée pour l'algorithme GOST

```
entity GOST is
  port (
    signal clk: in std_logic;
    signal reset: in std_logic;
    signal clk_en: in std_logic;
    signal start: in std_logic;
    signal done: out std_logic;
    signal n: in std_logic_vector(2 downto 0);
    signal dataa: in std_logic_vector(31 downto 0);
    signal datab: in std_logic_vector(31 downto 0);
    signal result: out std_logic_vector(31 downto 0)
  );
end entity GOST;
```

Fig. 6.31 – Extrait du code VHDL de l'extension généré pour l'algorithme GOST

6.3.5 Adaptation du code

L'adaptation automatique du code s'effectue en deux étapes. Un générateur de code C existe, il permet de générer dans un fichier le code C correspondant à un CDFG. Ce générateur existant est utilisé. C'est pourquoi, dans un premier temps nous transformons notre HCDG en CDFG. Puis nous générons le code C à partir du CDFG.

6.3.5.1 Transformation HCDG vers CDFG

La première étape correspond à la passe `Nios2ProcedureSetBuilder` (ligne 25 de la figure 6.2). Cette passe permet de transformer un HCDG en CDFG. La passe de transformation générique existe et nous avons personnalisé cette passe pour générer un CDFG qui contient des instructions assembleur spécifiques au NiosII. Le CDFG généré est stocké dans la variable `nios2ps`.

La figure 6.32 montre le CDFG généré à partir du HCDG de la figure 6.23. Le CDFG montre bien l'appel aux instructions spécialisées par le mnémonique `custom`, suivi du numéro de l'instruction spécialisée puis de l'opérande destination et des opérandes sources.

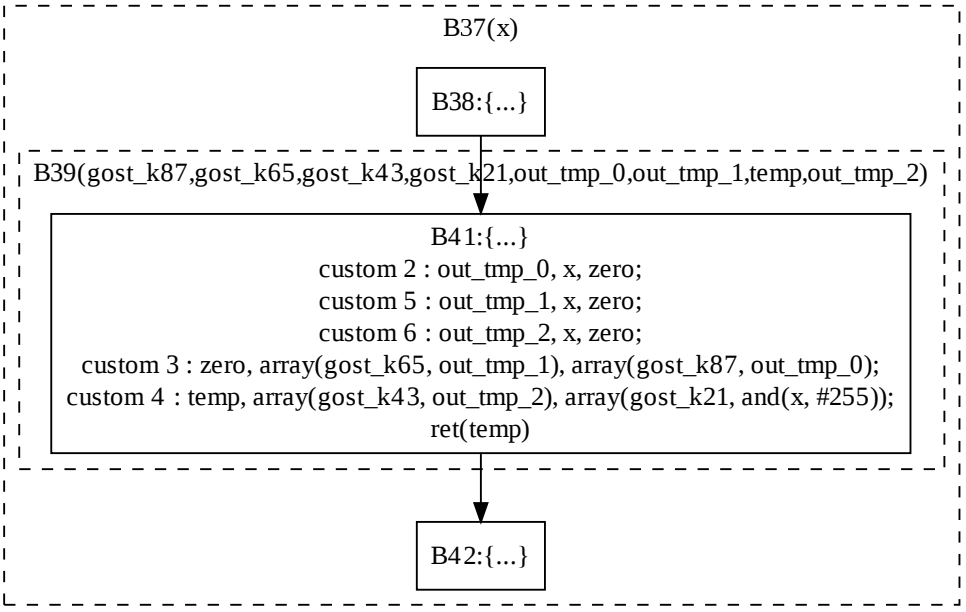


Fig. 6.32 – CDFG adapté pour un NiosII

6.3.5.2 Génération du code C avec assembleur inline pour un NiosII

La passe `Nios2AsmInlineCGenerator` (ligne 26 de la figure 6.2) permet de générer un fichier contenant le code C correspondant au CDFG fourni en entrée. Nous avons adapté la passe de génération de code pour qu'elle puisse interpréter les instructions spécifiques au NiosII. La figure 6.33 montre le code adapté, automatiquement généré, qui permet d'exploiter les instructions spécialisées.

```
unsigned int f(unsigned int x) {
    unsigned int out_tmp_0;
    unsigned int out_tmp_1;
    unsigned int out_tmp_2;
    unsigned int temp;
    asm volatile ("custom  2, %0, %1, zero" : "=r"(out_tmp_0) : "r"(x));
    asm volatile ("custom  5, %0, %1, zero" : "=r"(out_tmp_1) : "r"(x));
    asm volatile ("custom  6, %0, %1, zero" : "=r"(out_tmp_2) : "r"(x));
    asm volatile ("custom  3, zero, %0, %1" : : "r"(gost_k65[out_tmp_1]), "r"(↩
        gost_k87[out_tmp_0]));
    asm volatile ("custom  4, %0, %1, %2" : "=r"(temp) : "r"(gost_k43[out_tmp_2])↩
        , "r"(gost_k21[(x&255)]));
    return temp;
}
```

FIG. 6.33 – Code source de GOST modifié pour le NiosII

6.3.6 Validation par simulation avec SoCLib

Dans le but de valider aussi bien au niveau fonctionnel qu'au niveau cycle les instructions spécialisées et le code C adapté, nous nous appuyons sur la simulation en utilisant SoCLib.

6.3.6.1 Génération des modèles SystemC pour la simulation avec SoCLib

La passe de génération de modèles SystemC des instructions spécialisées, nommée `Nios2SystemCGenerator`, est appelée à la ligne 23 de la figure 6.2. Elle prend en paramètre un ensemble de fonctions spécialisées pour un NiosII, `nios2functionSet` et un nom pour la génération du fichier. Cette passe est l'équivalent de la passe de génération d'architecture, à la différence près que chaque instruction spécialisée est décrite comme une fonction C.

Comme pour la génération d'architecture, à chaque motif est associé un HCDG qui est ensuite transformé en CDFG. Nous utilisons la passe existante de génération de code C pour créer un fichier contenant toutes les instructions spécialisées sous forme de fonction C.

La figure 6.34 montre le code de l'instruction spécialisée correspondant au motif 1 de la figure 6.27 et généré pour la simulation avec SoCLib.

```
int SHRAND_FF_18(int in0) {
    int out0;
    out0 = in0 >> 24 & 255;
    return out0;
}
```

FIG. 6.34 – Modèle de l'instruction spécialisée générée à partir du motif 1 de la figure 6.27

6.3.6.2 Résultats de la simulation

Pour valider à la fois les instructions spécialisées et le code applicatif générés, nous avons créé un nouveau modèle de processeur NIOSII, basé sur le modèle du processeur NIOSII fourni par SoCLib et incluant les instructions spécialisées pour l'algorithme GOST.

Nous avons créé une plate-forme élémentaire, composée d'un processeur NIOSII étendu, d'une mémoire, et de quelques périphériques (*timer*, *tty*), connectés par un bus d'interconnexion générique. La figure 6.35 illustre cette plate-forme. Le code applicatif embarqué sur la plate-forme est compilé avec l'option d'optimisation `O2`.

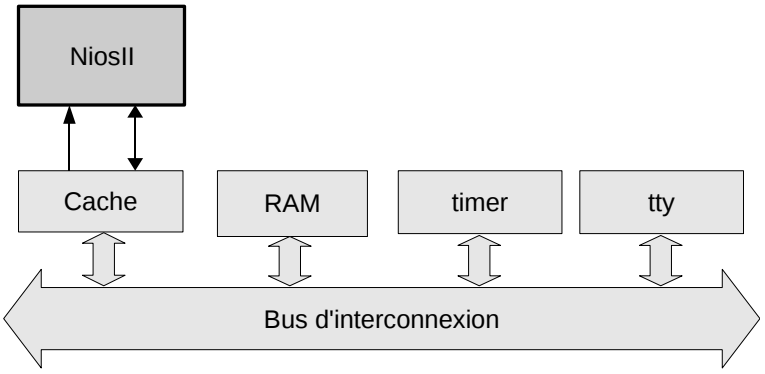


FIG. 6.35 – Plate-forme élémentaire composée d'un processeur NIOSII, d'une mémoire, et de quelques périphériques pour la simulation en utilisant SoCLib

La simulation a tout d'abord permis de valider au niveau fonctionnel les instructions spécialisées et le code applicatif qui exploite ces instructions spécialisées. La validation fonctionnelle de l'ensemble du processus d'extension constitue un réel motif de satisfaction.

Pour mesurer le gain apporté par nos instructions spécialisées, nous avons mesuré le nombre de cycles nécessaires à l'exécution de la fonction `f` pour le cryptage et le décryptage d'un bloc, dans un premier cas pour un processeur NIOSII sans extension, et dans un second cas pour un processeur NIOSII avec extension. La fonction `f` est appelée 32 fois par ronde, chaque bloc nécessitant une ronde pour le cryptage et une ronde pour le décryptage. Pour notre test, la fonction `f` est appelée 32 fois pour le cryptage et 32 fois pour le décryptage.

Le tableau 6.3 présente les résultats obtenus par la simulation. Le tableau 6.3 montre que le nombre de cycles nécessaires à l'exécution de la fonction `f` pour le cryptage et le décryptage d'un bloc est de 3 829 pour un NIOSII sans extension, et de 1 984 pour un

NiosII avec extension. Le facteur d'accélération apporté par les instructions spécialisées est de 1,93.

	NiosII sans extension	NiosII avec extension	gain
Cryptage et décryptage (nb de cycles)	3 829	1 984	1,93
Nombre d'instructions	31	22	1,41
Cœur de la fonction `f` (nb de cycles)	54	31	1,74

Tab. 6.3 – Résultats de la simulation obtenus pour deux modèles du processeur NiosII, un premier sans extension, et un second avec extension. La première ligne indique le nombre de cycles pour les 64 exécutions de la fonction `f` nécessaires au cryptage et au décryptage d'un bloc. Les instructions spécialisées générées par nos outils, pour des contraintes de 3 entrées et 1 sortie, permettent une accélération de 1,93 par rapport à un processeur NiosII sans extension.

Le tableau 6.3 présente également le nombre d'instructions assembleur qui composent la fonction `f`. L'utilisation d'instructions spécialisées permet d'économiser environ 30% de code. La dernière ligne du tableau indique le nombre de cycles pour l'exécution du cœur de la fonction `f`. Une analyse du code assembleur montre que si l'on tient compte des pénalités dues au retard sur les écritures des registres (instruction de chargement *load*), ce code pourrait au mieux s'exécuter en 26 cycles.

Étant données les contraintes, aussi bien sur l'architecture que sur les instructions spécialisées, il est difficile de faire mieux. Les substitutions par les S-boxes, décrites dans le code applicatif par un accès tableau et concrétisées par un accès mémoire, sont incontournables et constituent la principale raison des limitations des performances.

6.3.7 Résumé

Nous avons montré à travers un exemple très simple notre flot de conception entièrement automatisé qui permet de générer une extension pour un processeur NiosII et d'adapter le code applicatif qui exploite les instructions spécialisées. Cet exemple montre avec quelle facilité l'utilisateur peut spécifier les contraintes qu'il souhaite imposer aux instructions.

6.4 Proposition d'architecture

Nous avons vu les facteurs d'accélération que nous pouvons espérer sur ce genre d'algorithme et l'accès aux données en mémoire devient le point critique de notre modèle d'architecture. Par ailleurs, nous remarquons que pour notre exemple de l'algorithme GOST, la fonction étudiée ne fait que lire les données en mémoire. Nous avons vu qu'il est possible d'accélérer les calculs d'adresse. Pour poursuivre notre objectif de spécialiser une architecture pour un algorithme donné, nous proposons un nouveau modèle d'architecture, basé sur le modèle A ou B, et doté d'une mémoire fortement couplée à l'extension et d'un générateur d'adresse pour le calcul d'adresse. De cette façon, les données peuvent aller

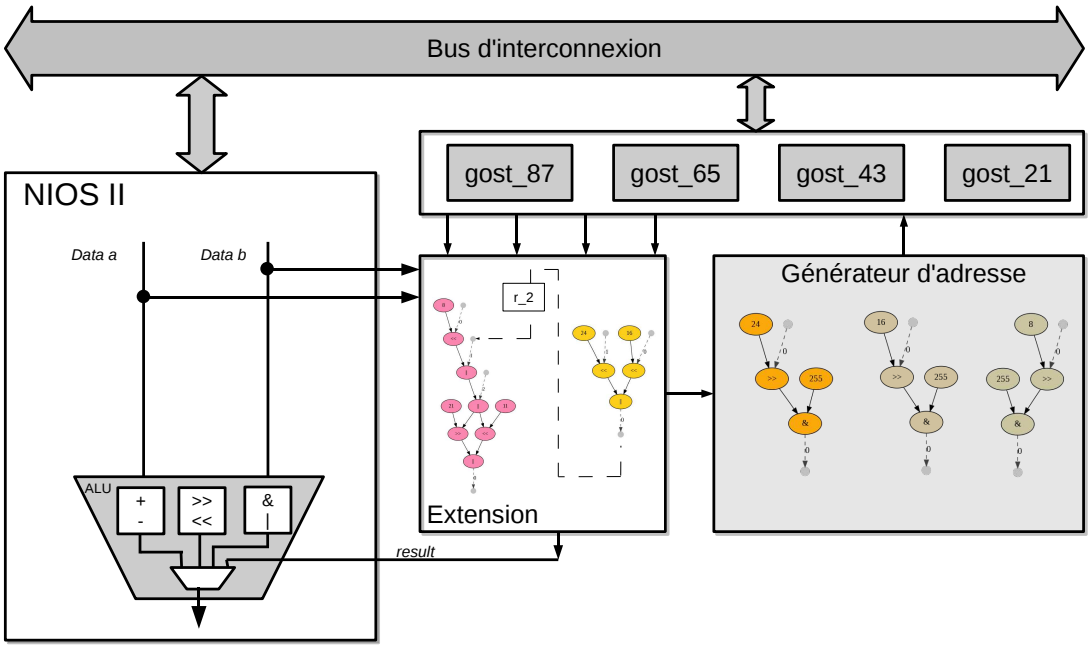


FIG. 6.36 – Architecture adaptée à l'algorithme GOST

directement de la mémoire vers l'extension, sans passer par le processeur.

La figure 6.36 illustre ce nouveau modèle d'architecture appliqué à l'algorithme GOST. La mémoire est représentée par les quatre éléments gost_87, gost_65, gost_43, et gost_21. Le générateur d'adresse contient les trois motifs qui représentent les calculs d'adresse et permet de calculer l'adresse des données en mémoire. L'extension contient les instructions spécialisées qui constituent le cœur de l'algorithme. Les mémoires peuvent être initialisées à travers le bus d'interconnexion par le processeur ou encore par un autre module matériel.

6.5 Conclusion

Ce chapitre a présenté le flot de compilation *DURASE* mis en place pour la génération automatique d'extensions de jeux d'instructions de processeurs. Ce flot de compilation s'appuie sur l'environnement Gecos [89]. Le flot de conception est déroulé et appliqué à l'exemple de l'algorithme de chiffrement GOST pour un processeur NIOSII mis en œuvre dans un FPGA StratixII d'Altera. La simulation avec SoCLib a permis de valider fonctionnellement l'extension et le code généré. Les résultats de la simulation montrent que les accès mémoire constituent le point faible de notre architecture. Une architecture est proposée en vue d'améliorer l'accès aux données en mémoire par l'extension.

Conclusion

Un ASIP (*Application Specific Instruction-set Processor*) est un processeur dédié à une application ou à un domaine d'applications. L'ASIP est un composant de base intéressant dans l'élaboration des systèmes embarqués modernes. Que ce soit dans le cadre de plates-formes hétérogènes ou homogènes, l'ASIP offre un compromis séduisant qui allie flexibilité et performance. L'ASIP est flexible car il est programmable, comme tout processeur, et propose de meilleures performances qu'un processeur à usage général car il est spécialisé pour une application donnée.

Pour éviter de repartir de zéro dans la conception de chaque nouvel ASIP, l'approche des dernières années consiste à considérer un processeur existant, vérifié et optimisé, et à le spécialiser par les options de configuration qu'il propose, ou par l'ajout de nouvelles instructions. Ainsi, de nombreux ASIP configurables et extensibles ont vu le jour, dans le monde industriel ou académique. L'ajout d'une nouvelle instruction consiste à greffer un module matériel spécialisé qui met en œuvre une instruction spécialisée pour étendre le jeu d'instructions du processeur. C'est l'extension de jeux d'instructions.

De nombreux efforts ont été investis ces dernières années pour fournir des méthodes qui permettent de spécialiser de manière automatique un processeur par le biais de l'extension de son jeu d'instructions. Les problèmes impliqués dans ce processus sont complexes : génération d'instructions, sélection d'instructions, ordonnancement, allocation de registres. Soit ils sont NP-complets, soit ils nécessitent des temps de résolution largement en dehors des capacités de calcul actuelles. Par conséquent, la conception d'ASIP nécessite des méthodologies et des outils de conception adaptés.

Contributions

Cette thèse se situe dans cette volonté de disposer de méthodologies et d'outils de conception permettant l'extension de jeux d'instructions. Nous avons mis en place un flot de conception générique complètement automatisé, qui génère une extension et le code qui exploite cette extension, à partir d'un programme C, des contraintes utilisateur et du modèle d'architecture. Nous avons choisi un processeur NiosII d'Altera pour valider notre approche. Nous avons utilisé avec succès une seule et même technique, la programmation par contraintes, pour résoudre tous les problèmes sous-jacents à l'extension de jeu d'instructions.

Les outils développés sont intégrés à l'infrastructure de compilation Gecos. La métho-

dologie proposée applique successivement les étapes de génération d'instructions, sélection d'instructions, ordonnancement, allocation de registres, puis génération d'architecture et adaptation du code. Quatre contributions majeures au problème de l'extension de jeu d'instructions sont apportées.

Génération d'instructions

Toutes les instructions ne sont pas appropriées à une mise en œuvre matérielle. En particulier, les instructions doivent respecter des contraintes liées à l'architecture, comme le nombre d'entrées et le nombre de sorties. En outre, des contraintes technologiques, comme la surface en silicium par exemple, sont également à prendre en compte.

Notre technique résout de manière exacte le problème de génération d'instructions sous contraintes architecturales et technologiques. À notre connaissance, elle est la seule à pouvoir prendre en compte les contraintes sur le nombre d'entrées, le nombre de sorties, la longueur du chemin critique, la surface, et la consommation d'énergie. Par ailleurs, elle est très flexible, l'utilisateur choisissant la combinaison de contraintes qu'il souhaite imposer.

Sélection d'instructions

Notre technique de sélection d'instructions permet de conserver, parmi un ensemble d'instructions candidates, celles qui minimisent le temps d'exécution séquentielle de l'application ou celles qui minimisent le nombre d'instructions exécutées par le processeur.

Les instructions spécialisées qui respectent le nombre d'entrées et de sorties contraints par l'architecture n'apportent que peu de performance. Lorsque le nombre d'entrées et de sorties d'une instruction spécialisée dépasse les capacités architecturales du processeur, des cycles additionnels de transfert de données sont nécessaires. L'extension doit alors posséder des registres pour sauvegarder temporairement les données et notre technique de sélection d'instructions prend en compte ces coûts de transfert en fonction de deux modèles d'architecture. Dans le premier modèle, l'extension possède des registres en entrée et en sortie. Dans le deuxième modèle, l'extension possède une file de registres. Les cycles de transfert de données sont pénalisants et la file de registres réduit la communication entre la file de registres du processeur et l'extension, et diminue ainsi le nombre de cycles de transfert.

Lorsque l'objectif est de minimiser le temps d'exécution séquentielle de l'application, le facteur d'accélération en temps d'exécution est de 2,3 en moyenne, et grimpe jusqu'à 3,35 pour l'algorithme de chiffrement SHA, pour des instructions à 4 entrées et 2 sorties et une extension contenant une file de registres.

Ordonnancement

Le module spécialisé étant en parallèle de l'unité arithmétique du processeur, il est possible d'exécuter de manière concurrente une instruction par l'extension et une par l'unité arithmétique du processeur.

Notre technique d'ordonnancement, appelée ordonnancement avec recalage d'instructions, permet de prendre en compte cette possibilité. Les résultats obtenus sur l'algorithme

Invert Matrix montrent que l'ordonnancement avec recalage d'instructions permet de diviser par 2 le temps d'exécution par rapport à un ordonnancement par liste, pour un même processeur et avec la même extension.

Allocation de registres

L'extension possédant des registres, il est nécessaire de procéder à une étape d'allocation de registres pour en déterminer le nombre et leur utilisation. Nous ne cherchons pas uniquement à minimiser le nombre de registres, nous voulons réduire de manière globale la complexité de l'interconnexion dans l'extension pour en minimiser sa surface en silicium.

Nous avons montré qu'il est possible d'allouer ces registres de manière à diminuer l'interconnexion entre les registres et les composants qui mettent en œuvre les instructions spécialisées, ce qui conduit à réduire la surface en silicium de l'extension. En particulier, concernant des extensions générées pour les algorithmes $Blowfish$ et $Cast128$, notre technique réalise une économie de 35% de silicium par rapport à une technique d'allocation de registres qui minimise seulement le nombre de registres.

Perspectives

Cette thèse n'a pas la prétention d'avoir résolu de manière définitive le problème de l'extension de jeux d'instructions mais peut servir de base pour des travaux futurs. À court terme, les outils peuvent faire l'objet de quelques améliorations. À moyen terme, les perspectives d'avenir de l'ASIP sont grandes.

À court terme

Fusion des chemins de données des instructions spécialisées. Pour compléter notre chaîne de conception, il faut intégrer la fusion des chemins de données des instructions spécialisées dans le flot pour réduire la surface de l'extension.

Résolution simultanée de la sélection d'instructions et de l'ordonnancement. Le chapitre 4 met en évidence que l'optimisation sur les transferts de données entre le processeur et l'extension n'est possible que si la sélection d'instructions et l'ordonnancement de l'application sont réalisées simultanément. L'optimisation sur les transferts de données permettrait de gagner en performance.

Création du pipeline des instructions spécialisées. Un autre aspect à prendre en considération est la gestion automatique des instructions multi-cycles. En effet, lorsqu'une instruction spécialisée nécessite plusieurs cycles pour être exécutée, il faut *casser* son chemin de données en insérant des registres afin de créer un pipeline, tout en se gardant une latence la plus courte possible. Il est certainement possible de résoudre ce problème de manière efficace par la programmation par contraintes.

Étude du compromis spécialisation/nombre de processeurs dans des plate-formes multi-processeurs. Nous générons automatiquement le modèle des instructions spécialisées pour une simulation avec SoCLib mais actuellement, les études se limitent au contexte de plates-formes mono-processeur élémentaires. Il serait intéressant de pouvoir profiter pleinement des services offerts par SoCLib en examinant, dans le contexte des plates-formes multi-processeurs, si il est préférable de disposer d'une plate-forme hétérogène ou homogène, munie de nombreux cœurs peu spécialisés ou bien de quelques cœurs hautement spécialisés.

À moyen terme

Ajout d'un accès mémoire à l'extension. Le faible nombre d'entrées et de sorties d'une instruction spécialisée est la principale limitation, et les cycles additionnels de transfert de données pour contourner cette limitation dégradent les performances. Pour augmenter la bande passante des données, des architectures existantes, comme le processeur LX de Xtensa ou le NiosII d'Altera, donnent la possibilité à l'extension d'accéder à de la logique externe par le biais d'interfaces. Il devient possible de coupler l'extension à la mémoire globale ou une mémoire dédiée, dont l'accès est aiguillé par un générateur d'adresses. Le défi est aujourd'hui d'aboutir à ce type d'architecture, qui ajoute de la logique complexe dans le chemin de données du processeur, sans pénaliser sa fréquence d'horloge, et de vérifier le bon fonctionnement du système global.

Degré d'automatisation. Un sujet largement ouvert encore aujourd'hui concerne le degré d'automatisation du processus d'extension de jeu d'instructions ou plutôt, le degré d'intervention manuelle dans ce processus. Passer en revue les quelques milliers de lignes de code d'un programme est un travail extrêmement fastidieux pour un concepteur, et par dessus tout, sujet à erreur. Il est donc nécessaire d'un côté d'automatiser le processus. D'un autre côté, une méthode, aussi intelligente soit-elle, ne peut remplacer les années d'expérience d'un concepteur capable de définir de manière précise les portions de code problématiques. L'efficacité du génie humain aboutit à des solutions de qualité. L'automatisation est une réponse à la complexité des problèmes et aux exigences de temps de mise sur le marché. Le processus idéal est certainement une combinaison judicieuse du génie humain et du génie logiciel.

Consommation d'énergie. L'évolution des batteries des systèmes embarqués n'arrive pas à suivre celle de leur consommation. Même pour des dispositifs branchés en continu, comme des *set-top box* par exemple, la consommation d'énergie est une véritable préoccupation. La chaleur dégagée doit être dissipée et devient une contrainte majeure dans la construction de systèmes. Dans ce contexte, la conception de processeurs « faible consommation » n'est pas un sujet nouveau mais les techniques basées sur la programmation par contraintes présentées dans ce document sont facilement adaptables pour intégrer cette dimension de consommation.

Publications

Conférences

[1] K. Martin, C. Wolinski, K. Kuchcinski, F. Charot et A. Floc'h : Constraint-Driven Identification of Application Specific Instructions in the DURASE system. *In SAMOS '09: Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, p. 194–203, Samos, Greece, 2009. Springer-Verlag.

[2] K. Martin, C. Wolinski, K. Kuchcinski, F. Charot et A. Floc'h : Constraint-Driven Instructions Selection and Application Scheduling in the DURASE system. *In ASAP '09: Proceedings of the 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, p. 145–152, Boston, USA, 2009. IEEE Computer Society.

[3] K. Martin, C. Wolinski, K. Kuchcinski, A. Floc'h et F. Charot : Sélection automatique d'instructions et ordonnancement d'applications basés sur la programmation par contraintes. *In 13ème Symposium en Architecture de machines (SympA'13)*, Toulouse, France, 2009.

[4] C. Wolinski, K. Kuchcinski, K. Martin, E. Raffin et F. Charot : How Constraints Programming Can Help You in the Generation of Optimized Application Specific Reconfigurable Processor Extensions. *In International Conference on Engineering of Reconfigurable Systems & Algorithms (ERSA 2009)*, Las Vegas États-Unis d'Amérique, 2009.

Workshops, Démonstrations, Posters

[5] K. Kuchcinski, C. Wolinski, K. Martin, F. Charot et A. Floc'h : Identification and Selection of Embedded Processor Extensions. *In 16th International Conference on Principles and Practices of Constraint Programming*, 2010.

[6] K. Martin et F. Charot : Utilisation combinée d'approches statique et dynamique pour la génération d'instructions spécialisées. GDR SoCSiP, 2008.

[7] K. Martin, C. Wolinski, K. Kuchcinski, F. Charot et A. Floc'h : Design of Processor Accelerators with Constraints. *In SweConsNet Workshop*, Linköping, Sweden, 2009.

[8] K. Martin, C. Wolinski, K. Kuchcinski, F. Charot et A. Floc'h : DURASE : Generic environment for Design and Utilization of Reconfigurable Application-Specific Processors Extensions. *DATE U-Booth*, 2009.

[9] K. Martin, C. Wolinski, K. Kuchcinski, F. Charot et A. Floc'h : Extraction automatique d'instructions spécialisées en utilisant la programmation par contraintes. GDR SoCSiP, 2009.

Bibliographie

[10] Berkeley Design Technology, Inc, http://www.bdti.com/.

[11] ITRS design report 2007, http://www.itrs.net/Links/2007ITRS/Home2007.htm.

[12] E. H. L. Aarts et v. Laarhoven : Statistical cooling : A general approach to combinatorial optimization problems. *Philips J. Res.*, 40(4):193–226, 1985.

[13] Aeroflex Gaisler, http://www.gaisler.com.

[14] A. V. Aho, M. Ganapathi et S. W. K. Tjiang : Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst.*, 11(4):491–516, 1989.

[15] A. V. Aho, R. Sethi et J. D. Ullman : *Compilers : principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[16] C. Alippi, W. Fornaciari, L. Pozzi et M. Sami : A DAG-based design approach for reconfigurable VLIW processors. *In DATE '99: Proceedings of the conference on Design, automation and test in Europe*, p. 57, New York, NY, USA, 1999. ACM Press.

[17] Altera, http://www.altera.com.

[18] Altera : NiosII Core Implementation Details, http://www.altera.com.

[19] Altera : NiosII Custom Instruction User Guide, http://www.altera.com.

[20] K. K. Andrew Mihal, Scott Weber : *Customizable Embedded Processors : Design Technologies and Applications*. Elsevier, 2006.

[21] P. Arató, S. Juhász, Z. Ádám Mann, Z. D. Mann, D. Papp et A. Orbán : Hardware-Software Partitioning in Embedded System Design. 2003.

[22] ARM, http://www.arm.com.

[23] J. M. Arnold, D. A. Buell et E. G. Davis : Splash 2. *In SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, p. 316–322, New York, NY, USA, 1992. ACM.

[24] M. Arnold : *Instruction Set Extension for Embedded Processors*. Thèse de doctorat, Unniversity of Delft, 2001.

[25] M. Arnold et H. Corporaal : Designing domain-specific processors. *Hardware/Software Codesign, 2001. CODES 2001. Proceedings of the Ninth International Symposium on*, p. 61–66, 2001.

[26] N. Arora, K. Chandramohan, N. Pothineni et A. Kumar : Instruction Selection in ASIP Synthesis Using Functional Matching. *In VLSI Design, 2010. VLSID '10. 23rd International Conference on*, p. 146 –151, jan. 2010.

[27] K. Atasu : *Hardware/Software partitioning for custom instruction processors*. Thèse de doctorat, Bogaziçi University, 2007.

[28] K. Atasu, G. Dündar et C. Özturan : An integer linear programming approach for identifying instruction-set extensions. *In CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, p. 172–177, New York, NY, USA, 2005. ACM.

[29] K. Atasu, O. Mencer, W. Luk, C. Ozturan et G. Dundar : Fast custom instruction identification by convex subgraph enumeration. *In Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, p. 1–6, 2008.

[30] K. Atasu, L. Pozzi et P. Ienne : Automatic application-specific instruction-set extensions under microarchitectural constraints. *In DAC '03: Proceedings of the 40th conference on Design automation*. ACM Press, 2003.

[31] P. M. Athanas et H. F. Silverman : Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11–18, 1993.

[32] Berkeley Design Technology Inc : Evaluating DSP processor performance. http://www.bdti.com/articles/benchmk_2000.pdf, 2002.

[33] A. Bhattacharya, A. Konar, S. Das, C. Grosan et A. Abraham : Hardware Software Partitioning Problem in Embedded System Design Using Particle Swarm Optimization Algorithm. *In CISIS '08: Proceedings of the 2008 International Conference on Complex, Intelligent and Software Intensive Systems*, p. 171–176, Washington, DC, USA, 2008. IEEE Computer Society.

[34] J. Bier : Use a Microprocessor, a DSP, or Both? Embedded Systems Conference, april 2008.

[35] N. N. Bình, M. Imai, A. Shiomi et N. Hikichi : A hardware/software partitioning algorithm for designing pipelined ASIPs with least gate counts. *In DAC '96: Proceedings of the 33rd annual Design Automation Conference*, p. 527–532, New York, NY, USA, 1996. ACM.

[36] P. Biswas, S. Banerjee, N. Dutt, P. Ienne et L. Pozzi : Performance and energy benefits of instruction set extensions in an FPGA soft core. *VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design., 19th International Conference on*, p. 6 pp.–, Jan. 2006.

[37] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi et P. Ienne : ISEGEN : Generation of High-Quality Instruction Set Extensions by Iterative Improvement. *42nd Design Automation Conference (DAC)*, p. 1246–1251, 2005.

[38] P. Biswas, V. Choudhary, K. Atasu, L. Pozzi, P. Ienne et N. Dutt : Introduction of local memory elements in instruction set extensions. *DAC*, p. 729–734, 2004.

[39] D. C. Black et J. Donovan : *SystemC : From the Ground Up*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[40] P. Bonzini, D. Harmanci et L. Pozzi : A Study of Energy Saving in Customizable Processors. *In Embedded Computer Systems: Architectures, Modeling, and Simula-*

tion, vol. 4599 de *Lecture Notes in Computer Science*, p. 304–312. Springer Berlin / Heidelberg, 2007.

[41] P. Bonzini et L. Pozzi : Code Transformation Strategies for Extensible Embedded Processors. *In Proceedings of CASES 2006*, p. 242–252, Seoul, South Korea, oct. 2006.

[42] P. Bonzini et L. Pozzi : A Retargetable Framework for Automated Discovery of Custom Instructions. *In ASAP*, p. 334–341, 2007.

[43] P. Bonzini et L. Pozzi : Polynomial-time subgraph enumeration for automated instruction set extension. *In DATE '07: Proceedings of the conference on Design, automation and test in Europe*, p. 1331–1336, San Jose, CA, USA, 2007. EDA Consortium.

[44] P. Bonzini et L. Pozzi : On the Complexity of Enumeration and Scheduling for Extensible Embedded Processors. Rap. tech. 2008/07, University of Lugano, déc. 2008.

[45] U. Bordoloi, H. P. Huynh, S. Chakraborty et T. Mitra : Evaluating design trade-offs in customizable processors. *In Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, p. 244 –249, july 2009.

[46] E. Borin, F. Klein, N. Moreano, R. Azevedo et G. Araujo : Fast instruction set customization. *Embedded Systems for Real-Time Multimedia, 2004. ESTImedia 2004. 2nd Workshop on*, p. 53–58, 6-7 Sept. 2004.

[47] F. Bouchez, A. Darte, C. Guillon et F. Rastello : Register Allocation: What Does the NP-Completeness Proof of Chaitin et al. Really Prove? Or Revisiting Register Allocation: Why and How. *In LCPC'06: Proceedings of the 19th international conference on Languages and compilers for parallel computing*, p. 283–298. Springer-Verlag, Berlin, Heidelberg, 2007.

[48] P. Brisk, A. Kaplan, R. Kastner et M. Sarrafzadeh : Instruction generation and regularity extraction for reconfigurable processors. *In CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, p. 262–269, New York, NY, USA, 2002. ACM.

[49] P. Brisk, A. Kaplan et M. Sarrafzadeh : Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. *In DAC '04: Proceedings of the 41st annual Design Automation Conference*, p. 395–400, New York, NY, USA, 2004. ACM.

[50] J. M. Cardoso et P. C. Diniz : *Compilation Techniques for Reconfigurable Architectures*. Springer Publishing Company, Incorporated, 2008.

[51] V. Cerný : Thermodynamical approach to the traveling salesman problem : An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, January 1985.

[52] G. Chaitin : Register allocation and spilling via graph coloring. *SIGPLAN Not.*, 39(4):66–74, 2004.

[53] N. Cheung, S. Parameswaran et J. Henkel : INSIDE : INstruction Selection/Identification & Design Exploration for Extensible Processors. *In ICCAD '03: Pro-*

ceedings of the 2003 IEEE/ACM international conference on Computer-aided design, p. 291, Washington, DC, USA, 2003. IEEE Computer Society.

[54] N. Cheung, S. Parameswarani et J. Henkel : A quantitative study and estimation models for extensible instructions in embedded processors. *In ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, p. 183–189, Washington, DC, USA, 2004. IEEE Computer Society.

[55] H. Choi, J.-S. Kim, C.-W. Yoon, I.-C. Park, S. H. Hwang et C.-M. Kyung : Synthesis of Application Specific Instructions for Embedded DSP Software. *IEEE Trans. Comput.*, 48(6):603–614, 1999.

[56] H. Choi, J. H. Yi, J.-Y. Lee, I.-C. Park et C.-M. Kyung : Exploiting intellectual properties in ASIP designs for embedded DSP software. *In DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, p. 939–944, New York, NY, USA, 1999. ACM.

[57] N. Clark, H. Zhong et S. Mahlke : Processor acceleration through automated instruction set customization. *In MICRO*, p. 129–140, 2003.

[58] N. Clark, H. Zhong, W. Tang et S. Mahlke : Automatic design of application specific instruction set extensions through dataflow graph exploration. *Int. J. Parallel Program.*, 31(6):429–449, December 2003.

[59] N. T. Clark : *Customizing the computation capabilities of microprocessors*. Thèse de doctorat, Ann Arbor, MI, USA, 2007. Adviser-Mahlke, Scott.

[60] N. T. Clark, H. Zhong et S. A. Mahlke : Automated Custom Instruction Generation for Domain-Specific Processor Acceleration. *IEEE Trans. Comput.*, 54(10):1258–1270, 2005.

[61] J. Cong, Y. Fan, G. Han, A. Jagannathan, G. Reinman et Z. Zhang : Instruction set extension with shadow registers for configurable processors. *In FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, p. 99–106, New York, NY, USA, 2005. ACM Press.

[62] J. Cong, Y. Fan, G. Han et Z. Zhang : Application-specific instruction generation for configurable processor architectures. *In FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, p. 183–189, New York, NY, USA, 2004. ACM Press.

[63] M. Corazao, M. Khalaf, L. Guerra, M. Potkonjak et J. Rabaey : Performance optimization using template mapping for datapath-intensive high-level synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 15(8):877–888, Aug 1996.

[64] T. H. Cormen, C. E. Leiserson, R. L. Rivest et C. Stein : *Introduction to Algorithms*. MIT Press, Cambridge, MA, second édn, 2001.

[65] CriticalBlue, http://www.criticalblue.com.

[66] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman et F. K. Zadeck : An efficient method of computing static single assignment form. *In POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 25–35, New York, NY, USA, 1989. ACM.

[67] R. Dimond, O. Mencer et W. Luk : CUSTARD - a customisable threaded FPGA soft processor and tools. *In Field Programmable Logic and Applications, 2005. International Conference on*, p. 1–6, Aug. 2005.

[68] DSPstone, http://www.iss.rwth-aachen.de/Projekte/Tools/DSPSTONE/dspstone.html.

[69] B. K. Dwivedi, A. Kejariwal, M. Balakrishnan et A. Kumar : Rapid Resource-Constrained Hardware Performance Estimation. *In RSP '06: Proceedings of the Seventeenth IEEE International Workshop on Rapid System Prototyping*, p. 40–46, Washington, DC, USA, 2006. IEEE Computer Society.

[70] P. Eles, Z. Peng1, K. Kuchcinski et A. Doboli : Hardware/Software Partitioning with Iterative Improvement Heuristics. *System Synthesis, International Symposium on*, 0:71, 1996.

[71] A. Fauth, J. Van Praet et M. Freericks : Describing instruction set processors using nML. *In EDTC '95: Proceedings of the 1995 European conference on Design and Test*, p. 503, Washington, DC, USA, 1995. IEEE Computer Society.

[72] Y. Fei, S. Ravi, A. Raghunathan et N. K. Jha : Energy Estimation for Extensible Processors. *In DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, p. 10682, Washington, DC, USA, 2003. IEEE Computer Society.

[73] T. A. Feo et M. G. C. Resende : Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[74] Fernando Magno Quintão Pereira : *Register Allocation by Puzzle Solving*. Thèse de doctorat, University of California, Los Angeles, 2008.

[75] J. A. Fisher, P. Faraboschi et C. Young : *Embedded Computing : A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, December 2004.

[76] M. Flynn : Some Computer Organizations and Their Effectiveness. *IEEE Transaction on Computers*, 21(C):948–960, 1972.

[77] C. W. Fraser, R. R. Henry et T. A. Proebsting : BURG : fast optimal instruction selection and tree parsing. *SIGPLAN Not.*, 27(4):68–76, 1992.

[78] T. Fruewirth et S. Abdennadher : *Essentials of Constraint Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[79] D. D. Gajski, N. D. Dutt, A. C.-H. Wu et S. Y.-L. Lin : *High-level synthesis : introduction to chip and system design*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[80] D. D. Gajski, A. C.-H. Wu, V. Chaiyakul, S. Mori, T. Nukiyama et P. Bricaud : Essential Issues for IP Reuse. *Asia and South Pacific Design Automation Conference*, 0:37, 2000.

[81] C. Galuzzi : *Automatically Fused Instructions*. Thèse de doctorat, TU Delft, 2009.

[82] C. Galuzzi et K. Bertels : The Instruction-Set Extension Problem: A Survey. *In ARC '08: Proceedings of the 4th international workshop on Reconfigurable Computing*, p. 209–220. Springer-Verlag, 2008.

[83] C. Galuzzi, K. Bertels et S. Vassiliadis : A Linear Complexity Algorithm for the Generation of multiple Input Single Output Instructions of Variable Size. *In SAMOS*, p. 283–293, 2007.

[84] C. Galuzzi, K. Bertels et S. Vassiliadis : The Spiral Search : A Linear Complexity Algorithm for the Generation of Convex MIMO Instruction-Set Extensions. *In Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, p. 337–340, Dec. 2007.

[85] C. Galuzzi, E. M. Panainte, Y. Yankova, K. Bertels et S. Vassiliadis : Automatic selection of application-specific instruction-set extensions. *In CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, p. 160–165, New York, NY, USA, 2006. ACM.

[86] C. Galuzzi, D. Theodoropoulos, R. Meeuws et K. Bertels : Algorithms for the automatic extension of an instruction-set. *In DATE*, p. 548–553. IEEE, 2009.

[87] M. R. Garey et D. S. Johnson : *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[88] GCC : the GNU Compiler Collection, http://gcc.gnu.org/.

[89] Gecos : Generic compiler suite, http://gecos.gforge.inria.fr/.

[90] W. Geurts : *Accelerator Data-Path Synthesis for High-Throughput Signal Processing Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[91] T. Glokler, S. Bitterlich et H. Meyr : ICORE : a low-power application specific instruction set processor for DVB-T acquisition and tracking. *In ASIC/SOC Conference, 2000. Proceedings. 13th Annual IEEE International*, p. 102 –106, 2000.

[92] F. Glover et M. Laguna : *Tabu Search*. Kluwer Academic, Dordrecht, 1997.

[93] R. Gonzalez : Xtensa : a configurable and extensible processor. *Micro, IEEE*, 20(2):60–70, Mar/Apr 2000.

[94] M. Gries et K. Keutzer : *Building ASIPs: The Mescal Methodology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[95] Y. Guo, G. J. Smit, H. Broersma et P. M. Heysters : A graph covering algorithm for a coarse grain reconfigurable system. *In LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, 2003.

[96] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge et R. B. Brown : MiBench : A free, commercially representative embedded benchmark suite. *In WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, p. 3–14, Washington, DC, USA, 2001.

[97] G. Hadjiyiannis, S. Hanono et S. Devadas : ISDL : an instruction set description language for retargetability. *In DAC '97: Proceedings of the 34th annual Design Automation Conference*, p. 299–302, New York, NY, USA, 1997. ACM.

[98] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt et A. Nicolau : EXPRESSION : A Language for Architecture Exploration through Compiler/Simulator Retargetability. *Design, Automation and Test in Europe Conference and Exhibition*, 0:485, 1999.

[99] R. W. Hartenstein, J. Becker, R. Kress et H. Reinig : High-performance computing using a reconfigurable accelerator. *Concurrency - Practice and Experience*, 8(6):429–443, 1996.

[100] S. Hauck, T. Fry, M. Hosler et J. Kao : The Chimaera reconfigurable functional unit. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(2): 206–217, Feb. 2004.

[101] J. Hauser et J. Wawrzynek : Garp : a MIPS processor with a reconfigurable coprocessor. *In FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, p. 12–21, Apr 1997.

[102] J. Henkel : Closing the SoC Design Gap. *Computer*, 36(9):119–121, 2003.

[103] J. Hidalgo et J. Lanchares : Functional Partitioning for Hardware-Software Codesign using Genetic Algorithms. *EUROMICRO Conference*, 0:631, 1997.

[104] A. Hoffmann, H. Meyr et R. Leupers : *Architecture Exploration for Embedded Processors with Lisa*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[105] A. Hoffmann, A. Nohl, S. Pees, G. Braun et H. Meyr : Generating production quality software development tools using a machine description language. *In DATE '01: Proceedings of the conference on Design, automation and test in Europe*, p. 674–678, Piscataway, NJ, USA, 2001. IEEE Press.

[106] A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, O. Wahlen et H. Meyr : A methodology for the design of application specific instruction set processors (ASIP) using the machine description language LISA. *In ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, p. 625–630, Piscataway, NJ, USA, 2001. IEEE Press.

[107] B. K. Holmer : *Automatic Design of Computer Instruction Sets*. Thèse de doctorat, University of California, Berkeley, 1993.

[108] I.-J. Huang et A. M. Despain : Synthesis of instruction sets for pipelined microprocessors. *In DAC '94: Proceedings of the 31st annual Design Automation Conference*, p. 5–11, New York, NY, USA, 1994. ACM.

[109] Z. Huang, S. Malik, N. Moreano et G. Araujo : The design of dynamically reconfigurable datapath coprocessors. *ACM Trans. Embed. Comput. Syst.*, 3(2):361–384, 2004.

[110] H. P. Huynh et T. Mitra : Runtime Adaptive Extensible Embedded Processors – A Survey. *In SAMOS '09: Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, p. 215–225, Berlin, Heidelberg, 2009. Springer-Verlag.

[111] Y.-T. Hwang, J.-Y. Chen et J.-J. Chiu : HW/SW Auto-Coupling for Fast IP Integration in SoC Designs. *Embedded Software and Systems, Second International Conference on*, 0:556–563, 2008.

[112] C. Iseli et E. Sanchez : Spyder : a SURE (SUperscalar and REconfigurable) processor. *J. Supercomput.*, 9(3):231–252, 1995.

[113] JaCoP : JaCoP Library User's Guide, http://jacopguide.osolpro.com/guideJaCoP.html.

[114] JaCoP : Java Constraint Programming solver, http://www.jacop.eu.

[115] M. K. Jain, M. Balakrishnan et A. Kumar : ASIP Design Methodologies : Survey and Issues. *In VLSID '01: Proceedings of the The 14th International Conference on*

VLSI Design (VLSID '01), p. 76, Washington, DC, USA, 2001. IEEE Computer Society.

[116] M. K. Jain, M. Balakrishnan et A. Kumar : An efficient technique for exploring register file size in ASIP synthesis. *In CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, p. 252–261, New York, NY, USA, 2002. ACM.

[117] R. Jayaseelan, H. Liu et T. Mitra : Exploiting forwarding to improve data bandwidth of instruction-set extensions. *In DAC '06: Proceedings of the 43rd annual conference on Design automation*, p. 43–48, New York, NY, USA, 2006. ACM Press.

[118] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas et J. Foster : An FPGA-based VLIW processor with custom hardware execution. *In FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, p. 107–117, New York, NY, USA, 2005. ACM.

[119] R. M. Karp : Reducibility Among Combinatorial Problems. *In* R. E. Miller et J. W. Thatcher, éds : *Complexity of Computer Computations*, p. 85–103. Plenum Press, 1972.

[120] K. Karuri, M. A. Al Faruque, S. Kraemer, R. Leupers, G. Ascheid et H. Meyr : Fine-grained application source code profiling for ASIP design. *In DAC '05: Proceedings of the 42nd annual Design Automation Conference*, p. 329–334, New York, NY, USA, 2005. ACM.

[121] R. Kastner, S. Ogrenci-Memik, E. Bozorgzadeh et M. Sarrafzadeh : Instruction Generation for Hybrid Reconfigurable Systems. *In ICCAD*, p. 127, 2001.

[122] B. Kastrup : *Automatic Synthesis of Reconfigurable Instruction Set Accelerators*. Thèse de doctorat, Eindhoven University of Technology, 2001.

[123] B. Kastrup, A. Bink et J. Hoogerbrugge : ConCISe : A Compiler-Driven CPLD-Based Instruction Set Accelerator. *In FCCM '99: Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, p. 92, Washington, DC, USA, 1999. IEEE Computer Society.

[124] B. W. Kernighan et S. Lin : An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal*, 49(1):291–307, 1970.

[125] J. Kin, C. Lee, W. H. Mangione-Smith et M. Potkonjak : Power Efficient Mediaprocessors : Design Space Exploration. *Design Automation Conference*, 0:321–326, 1999.

[126] S. Kirkpatrick, C. D. Gelatt et M. P. Vecchi : Optimization by Simulated Annealing. *Science*, 220(4598):671–680, may 1983.

[127] A. A. Kountouris : *Outils pour la Validation Temporelle et l'Optimisation de Programmes Synchrones*. Thèse de doctorat, Université de Rennes 1, 1998.

[128] I. Kuon et J. Rose : Measuring the gap between FPGAs and ASICs. *In FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, p. 21–30, New York, NY, USA, 2006. ACM.

[129] F. J. Kurdahi et A. C. Parker : REAL: a program for REgister ALlocation. *In DAC '87: Proceedings of the 24th ACM/IEEE Design Automation Conference*, p. 210–215, New York, NY, USA, 1987. ACM.

[130] M. Labrecque, P. Yiannacouras et J. G. Steffan : Custom code generation for soft processors. *SIGARCH Comput. Archit. News*, 35(3):9–19, 2007.

[131] S.-K. Lam et T. Srikanthan : Rapid design of area-efficient custom instructions for reconfigurable embedded processing. *Journal of Systems Architecture*, 55(1):1 – 14, 2009.

[132] J. Larrosa et G. Valiente : Constraint satisfaction algorithms for graph pattern matching. *Mathematical Structures in Computer Science*, 12:403–422, 2002.

[133] C. Lee, M. Potkonjak et W. H. Mangione-Smith : MediaBench : A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. *In International Symposium on Microarchitecture*, p. 330–335, 1997.

[134] R. Leupers : *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[135] R. Leupers : Code selection for media processors with SIMD instructions. *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, p. 4–8, 2000.

[136] R. Leupers, K. Karuri, S. Kraemer et M. Pandey : A design flow for configurable embedded processors based on optimized instruction set extension synthesis. *DATE*, p. 581–586, 2006.

[137] R. Leupers et P. Marwedel : Retargetable Generation of Code Selectors from HDL Processor Models. *In EDTC '97: Proceedings of the 1997 European conference on Design and Test*, p. 140, Washington, DC, USA, 1997. IEEE Computer Society.

[138] R. Leupers et P. Marwedel : *Retargetable compiler technology for embedded systems : tools and applications*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[139] R. L. Leupers et S. Bashford : Graph-based code selection techniques for embedded processors. *ACM Trans. Des. Autom. Electron. Syst.*, 5(4):794–814, 2000.

[140] L. L'Hours : Generating Efficient Custom FPGA Soft-Cores for Control-Dominated Applications. *In Application-Specific Systems, Architectures and Processors, IEEE International Conference on*, vol. 0, p. 127–133, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[141] S. Liao, S. Devadas, K. Keutzer et S. Tjiang : Instruction selection using binate covering for code size optimization. *In ICCAD '95: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, p. 393–399, Washington, DC, USA, 1995. IEEE Computer Society.

[142] C. Liem, T. May et P. Paulin : Instruction-set matching and selection for DSP and ASIP code generation. *In European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings.*, p. 31–37, Feb-3 Mar 1994.

[143] H. Lin et Y. Fei : Utilizing custom registers in application-specific instruction set processors for register spills elimination. *In GLSVLSI '07: Proceedings of the 17th ACM Great Lakes symposium on VLSI*, p. 323–328, New York, NY, USA, 2007. ACM.

[144] H. Lin et Y. Fei : Resource sharing of pipelined custom hardware extension for energy-efficient application-specific instruction set processor design. *In Computer Design, 2009. ICCD 2009. IEEE International Conference on*, p. 158 –165, oct. 2009.

[145] Y.-s. Lü, L. Shen, L.-b. Huang, Z.-y. Wang et N. Xiao : Customizing computation accelerators for extensible multi-issue processors with effective optimization techniques. *In DAC '08: Proceedings of the 45th annual Design Automation Conference*, p. 197–200, New York, NY, USA, 2008. ACM.

[146] G. Martin : Recent Developments in Configurable and Extensible Processors. *Application-Specific Systems, Architectures and Processors, IEEE International Conference on*, 0:39–44, 2006.

[147] P. Marwedel, D. Lanneer, J. V. Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, G. Goossens et G. Goossens : *CHESS : Retargetable Code Generation For Embedded DSP Processors*, chap. 5, p. 85–102. Kluwer Academic Publishers, 1995.

[148] M. M. Mbaye, N. Bélanger, Y. Savaria et S. Pierre : A Novel Application-specific Instruction-set Processor Design Approach for Video Processing Acceleration. *J. VLSI Signal Process. Syst.*, 47(3):297–315, 2007.

[149] B. D. McKay : The nauty page. http://cs.anu.edu.au/ bdm/nauty/, 2004.

[150] mcrypt, http://sourceforge.net/projects/mcrypt/.

[151] B. Mei, S. Vernalde, D. Verkest, H. D. Man et R. Lauwereins : ADRES : An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. *In Field-Programmable Logic and Applications*, vol. 2778/2003, p. 61–70. Springer Berlin / Heidelberg, 2003.

[152] A. J. Menezes, S. A. Vanstone et P. C. V. Oorschot : *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[153] V. Messé : *Production de compilateurs flexibles pour la conception de processeurs programmables spécialisés*. Thèse de doctorat, Université de Rennes I, 1999.

[154] H. Meyr : System-on-chip for communications: the dawn of ASIPs and the dusk of ASICs. *In Signal Processing Systems, 2003. SIPS 2003. IEEE Workshop on*, p. 4 – 5, aug. 2003.

[155] MIPS : MIPS technologies inc., http://www.mips.com.

[156] T. Miyamori et K. Olukotun : A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:2, 1998.

[157] L. Moll, J. Vuillemin et P. Boucard : High-energy physics on DECPeRLe-1 programmable active memory. *In FPGA '95: Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, p. 47–52, New York, NY, USA, 1995. ACM.

[158] G. E. Moore : Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[159] N. Moreano, E. Borin, C. de Souza et G. Araujo : Efficient datapath merging for partially reconfigurable architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(7):969–980, July 2005.

[160] B. Moret et H. D. Shapiro : *Algorithms from P to NP*. Addison Wesley, 1991.

[161] S. S. Muchnick : *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[162] L. P. Cordella, P. Foggia, C. Sansone et M. Vento : A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[163] D. A. Patterson et J. L. Hennessy : *Computer architecture : a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[164] A. Peymandoust, L. Pozzi, P. Ienne et G. D. Micheli : Automatic Instruction Set Extension and Utilization for Embedded Processors. *Application-Specific Systems, Architectures and Processors, IEEE International Conference on*, 0:108, 2003.

[165] PolarSSL : Small cryptographic library, http://polarssl.org/.

[166] N. Pothineni, A. Kumar et K. Paul : Application Specific Datapath Extension with Distributed I/O Functional Units. *In VLSID '07: Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference*, p. 551–558, Washington, DC, USA, 2007. IEEE Computer Society.

[167] N. Pothineni, A. Kumar et K. Paul : Exhaustive Enumeration of Legal Custom Instructions for Extensible Processors. *In VLSI Design, 2008. VLSID 2008. 21st International Conference on*, p. 261–266, Jan. 2008.

[168] L. Pozzi : *Methodologies for the Design of Application Specific Reconfigurable VLIW Processors*. Thèse de doctorat, Politecnico di Milano, 2000.

[169] L. Pozzi, K. Atasu et P. Ienne : Exact and approximate algorithms for the extension of embedded processor instruction sets. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(7):1209–1229, July 2006.

[170] L. Pozzi et P. Ienne : Exploiting pipelining to relax register-file port constraints of instruction-set extensions. *In CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, p. 2–10, New York, NY, USA, 2005. ACM.

[171] M. Purnaprajna, M. Reformat et W. Pedrycz : Genetic algorithms for hardware-software partitioning and optimal resource allocation. *Journal of Systems Architecture*, 53(7):339 – 354, 2007.

[172] B. Radunovic et V. M. Milutinovic : A Survey of Reconfigurable Computing Architectures. *In FPL '98: Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm*, vol. 1482/1998, p. 376–385, London, UK, 1998. Springer-Verlag.

[173] D. S. Rao et F. J. Kurdahi : Partitioning by regularity extraction. *In DAC '92: Proceedings of the 29th ACM/IEEE Design Automation Conference*, p. 235–238, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[174] R. Razdan, K. Brace et M. Smith : PRISC software acceleration techniques. *In Computer Design: VLSI in Computers and Processors, 1994. ICCD '94. Proceedings., IEEE International Conference on*, p. 145–149, Oct 1994.

[175] R. Razdan et M. D. Smith : A high-performance microarchitecture with hardware-programmable functional units. *In MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, p. 172–180, New York, NY, USA, 1994. ACM.

[176] Retarget : TARGET The ASIP company, http://www.retarget.com.

[177] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi et J. Owens : Register organization for media processing. *In High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, p. 375–386, 2000.

[178] F. Rossi, P. v. Beek et T. Walsh : *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[179] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. M. Arnold et M. Gokhale : The NAPA Adaptive Processing Architecture. *In FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, p. 28, Washington, DC, USA, 1998. IEEE Computer Society.

[180] Silicon Hive, http://www.siliconhive.com.

[181] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh et E. Chaves Filho : MorphoSys : an integrated reconfigurable system for data-parallel and computation-intensive applications. *Computers, IEEE Transactions on*, 49(5):465–481, May 2000.

[182] SoCLib : Open platform for virtual prototyping of multi-processors system on chip (MP-SoC), http://www.soclib.fr.

[183] S. Sorlin : *Mesurer la similarité de graphes*. Thèse de doctorat en informatique, Université Claude Bernard Lyon I, nov. 2006.

[184] S. Sorlin et C. Solnon : A global constraint for graph isomorphism problems. *Lecture notes in computer science ISSN 0302-9743*, p. 287–301, avr. 2004.

[185] Stretch, http://www.stretchinc.com/.

[186] F. Sun, S. Ravi, A. Raghunathan et N. K. Jha : Synthesis of custom processors based on extensible platforms. *In ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, p. 641–648, New York, NY, USA, 2002. ACM.

[187] F. Sun, S. Ravi, A. Raghunathan et N. K. Jha : A Scalable Application-Specific Processor Synthesis Methodology. *In ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, p. 283, Washington, DC, USA, 2003. IEEE Computer Society.

[188] Synopsys : Synplify pro, http://www.synopsys.com.

[189] SystemC, http://www.systemc.org.

[190] Tensilica, http://www.tensilica.com/.

[191] Texas Instrument, http://www.ti.com.

[192] J. R. Ullmann : An Algorithm for Subgraph Isomorphism. *J. ACM*, 23(1):31–42, 1976.

[193] F. Vahid et T. D. Le : Extending the Kernighan/Lin Heuristic for Hardware and Software Functional Partitioning. *Design Automation for Embedded Systems*, 2:237–261, 1997.

[194] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov et E. M. Panainte : The MOLEN Polymorphic Processor. *IEEE Trans. Comput.*, 53(11): 1363–1375, 2004.

[195] B. Veale, J. Antonio, M. Tull et S. Jones : Selection of instruction set extensions for an FPGA embedded processor core. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, p. 8 pp.–, April 2006.

[196] A. Verma, P. Brisk et P. Lenne : Fast, quasi-optimal, and pipelined instruction-set extensions. *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, p. 334–339, March 2008.

[197] A. K. Verma, P. Brisk et P. Ienne : Rethinking custom ISE identification : a new processor-agnostic method. *In CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, p. 125–134, New York, NY, USA, 2007. ACM.

[198] A. K. Verma, P. Brisk et P. Ienne : Fast, Nearly Optimal ISE Identification With I/O Serialization Through Maximal Clique Enumeration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(3):341 –354, march 2010.

[199] Viragelogic, http://www.viragelogic.com.

[200] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe et A. Agarwal : Baring It All to Software : Raw Machines. *Computer*, 30(9):86–93, 1997.

[201] G. Wang, W. Gong et R. Kastner : System Level Partitioning for Programmable Platforms Using the Ant Colony Optimization. *In International Workshop on Logic & Synthesis (IWLS'04)*, Temecula, California, 2004.

[202] T. Wiangtong, P. Y. K. Cheung et W. Luk : Comparing Three Heuristic Search Methods for Functional Partitioning in Hardware/Software Codesign. *Design Automation for Embedded Systems*, volume 6:425–449, july 2002.

[203] M. Wirthlin et B. Hutchings : A Dynamic Instruction Set Computer. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:0099, 1995.

[204] R. Wittig et P. Chow : OneChip : an FPGA processor with reconfigurable logic. *In FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, p. 126–135, Apr 1996.

[205] C. Wolinski et K. Kuchcinski : Identification of Application Specific Instructions Based on Sub-Graph Isomorphism Constraints. *In Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, p. 328–333, July 2007.

[206] C. Wolinski et K. Kuchcinski : Automatic selection of application-specific reconfigurable processor extensions. *In DATE '08: Proceedings of the conference on Design, automation and test in Europe*, p. 1214–1219, New York, NY, USA, 2008. ACM.

[207] C. Wolinski, K. Kuchcinski et A. Postula : UPaK : Abstract Unified Pattern Based Synthesis Kernel for Hardware and Software Systems. *In DATE U-Booth*, 2007.

[208] C. Wolinski, K. Kuchcinski et E. Raffin : Automatic design of application-specific reconfigurable processor extensions with UPaK synthesis kernel. *ACM Trans. Des. Autom. Electron. Syst.*, 15(1):1–36, 2009.

[209] C. Wolinski, K. Kuchcinski, E. Raffin et F. Charot : Architecture-Driven Synthesis of Reconfigurable Cells. *In 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD 2009)*, p. 531–538, Patras Greece, 2009.

[210] C. Wolinski, K. Kuchcinski, J. Teich et F. Hannig : Optimization of Routing and Reconfiguration Overhead in Programmable Processor Array Architectures. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:306–309, 2008.

[211] Xilinx, http://www.xilinx.com.

[212] Z. A. Ye, A. Moshovos, S. Hauck et P. Banerjee : CHIMAERA : a high-performance architecture with a tightly-coupled reconfigurable functional unit. *SIGARCH Comput. Archit. News*, 28(2):225–235, 2000.

[213] A. Yeung et J. Rabaey : A reconfigurable data-driven multiprocessor architecture for rapid prototyping of high throughput dsp algorithms. *In System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, vol. i, p. 169–178 vol.1, Jan 1993.

[214] P. Yiannacouras, J. G. Steffan et J. Rose : Application-specific customization of soft processor microarchitecture. *In FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, p. 201–210, New York, NY, USA, 2006. ACM Press.

[215] P. Yu et T. Mitra : Characterizing embedded applications for instruction-set extensible processors. *In DAC '04: Proceedings of the 41st annual Design Automation Conference*, p. 723–728, New York, NY, USA, 2004. ACM.

[216] P. Yu et T. Mitra : Scalable custom instructions identification for instruction-set extensible processors. *In CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, p. 69–78, New York, NY, USA, 2004. ACM.

[217] P. Yu et T. Mitra : Disjoint pattern enumeration for custom instructions identification. *In Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, p. 273–278, Aug. 2007.

[218] K. Zhao, J. Bian, S. Dong, Y. Song et S. Goto : Automated Specific Instruction Customization Methodology for Multimedia Processor Acceleration. *Quality*

Electronic Design, 2008. ISQED 2008. 9th International Symposium on, p. 321–324, 17-19 March 2008.

[219] K. Zhao, J. Bian, S. Dong, Y. Song et S. Goto : Fast Custom Instruction Identification Algorithm Based on Basic Convex Pattern Model for Supporting ASIP Automated Design. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, E91-A(6):1478–1487, 2008.

Résumé

Les processeurs à jeux d'instructions spécifiques (ASIP) sont des processeurs spécialisés qui combinent la flexibilité d'un processeur programmable avec la performance d'un processeur dédié. L'une des approches de conception de tels processeurs consiste à spécialiser un cœur de processeur existant en y ajoutant des instructions spécialisées, mises en œuvre dans un module matériel fortement couplé au chemin de données du processeur. C'est l'extension de jeu d'instructions.

La conception d'un ASIP nécessite des méthodologies et des outils logiciels appropriés garantissant une maîtrise des contraintes de conception et de la complexité grandissante des applications. Dans ce contexte, cette thèse vise à proposer une méthodologie de génération automatique d'extensions de jeux d'instructions. Celle-ci consiste à tout d'abord identifier l'ensemble des instructions candidates qui satisfont les contraintes architecturales et technologiques, afin de garantir leurs mises en œuvre. Ensuite, les instructions candidates qui minimisent le temps d'exécution séquentielle de l'application sont sélectionnées. Les ressources matérielles de l'extension, telles que les registres et les multiplexeurs, sont optimisées. Enfin, la dernière étape génère la description matérielle et le modèle de simulation de l'extension. Le code applicatif est adapté pour tenir compte des nouvelles instructions.

Cette thèse propose des techniques basées sur la programmation par contraintes pour résoudre les problèmes difficiles (voir intraitables) que sont l'identification d'instructions, la sélection d'instructions et l'allocation de registres.

Mots-clés : ASIP, extension de jeux d'instructions, processeur spécialisé, méthodologie de conception, compilation, système sur puce, programmation par contraintes

Abstract

ASIPs (*Application Specific Instruction set Processors*) are custom processors that offer a good trade-off between performance and flexibility. A common processor customization approach is to augment its standard instruction set with application-specific instructions that are implemented on specifically designed hardware extensions (reconfigurable cells). These extensions are often directly connected to the processor's data-path.

The design of the ASIP processor must rely on dedicated methodologies and software tools that manage both the design constraints and the growing complexity of applications. In this context, the aims of this thesis were to propose a new methodology for the automatic generation of instruction-set extensions.

In the first step of our proposed design flow, we generate the candidates instruction that satisfy some architectural and technological constraints. In the second step, we identify the set of standard and customized instructions that minimizes the sequential application's execution time. In the next step, optimized hardware extensions and the corresponding application program including new instructions are generated. During the hardware generation, the optimizations of the hardware resources such as registers and multiplexers are simultaneously carried out.

In our proposed design flow we used the constraint-based approach to solve the computationally complex problems of instruction identification, instruction selection and register allocation.

Keywords : ASIP, instruction set extension, custom processors, design methodology, compilation, system on chip, constraint programming