Outils pour l'exploration d'architectures programmables embarquées dans le cadre d'applications industrielles (Tools for exploration of embedded programmable architectures in industrial applications)

F. Nacabal

THÈSE DE DOCTORAT

présentée par

François NAÇABAL

pour obtenir le grade de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(arrêté ministériel du 30 mars 1992)

Spécialité : **Microélectronique**

OUTILS POUR L'EXPLORATION D'ARCHITECTURES PROGRAMMABLES EMBARQUÉES DANS LE CADRE D'APPLICATIONS INDUSTRIELLES

Date de Soutenance : 27 Février 1998

Composition du jury :

Messieurs :	Pierre GENTIL	*Président*
	Yvon SAVARIA	*Rapporteur*
	Sanjay RAJOPADHYE	*Rapporteur*
	Joseph BOREL	*Examinateur*
	Pierre PAULIN	*Examinateur*
	Ahmed JERRAYA	*Examinateur*

Thèse préparée au sein du Laboratoire TIMA-INPG à Grenoble
et du département Central R&D de SGS-THOMSON Microelectronics à Crolles

Remerciements

Ce travail a été effectué au sein du laboratoire des Techniques de l'Informatique et de la Microélectronique pour l'Architecture d'ordinateurs (TIMA) de l'Institut National Polytechnique de Grenoble (INPG), ainsi qu'au département de Recherche et Développement Central de SGS-THOMSON Microelectronics à Crolles. Je tiens à remercier tout particulièrement Messieurs Ahmed Jerraya et Pierre Paulin pour leur aide dans l'orientation de mon travail, et leur précieux support.

Je remercie Monsieur Pierre Gentil, de l'INPG, qui m'a fait l'honneur de présider mon jury de thèse.

Que Messieurs Yvon Savaria, de l'Ecole Polytechnique de Montréal, et Sanjay Rajopadhye, de l'IRISA, trouvent ici l'expression de ma reconnaissance pour avoir accepté d'être rapporteurs de mon travail.

Je remercie de plus Monsieur Joseph Borel, de SGS-THOMSON, qui a accepté de faire partie de mon jury de thèse.

Je remercie l'équipe de conception du circuit IVT de SGS-THOMSON, en particulier Michel Harrand, Olivier Deygas et José Sanches.

Je tiens à témoigner ma reconnaissance à tous ceux qui m'ont aidé durant ces trois années, aussi bien dans l'équipe Embedded Systems Technology de SGS-THOMSON que dans le groupe System-Level Synthesis du TIMA, et plus spécialement à Marco Cornero, Clifford Liem et Miguel Santana.

Je remercie très particulièrement Chantal Brunel et Philippe Guillaume pour leur support constant et leur profonde amitié.

Je remercie enfin toute ma famille, pour m'avoir toujours soutenu au cours de mes études, et à qui je souhaite dédier cette thèse.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Grâce à l'amélioration constante des technologies sub-microniques, l'industrie de la microélectronique peut de nos jours produire des circuits intégrés de plusieurs dizaines de millions de transistors [16]. Cette haute densité permet l'intégration de systèmes complets sur une seule puce, alors que ceux-ci nécessitaient auparavant plusieurs circuits réunis sur une carte. De tels circuits sont appelés systèmes intégrés ou mono-puces (*system-on-a-chip*) [85].

Les domaines d'application de ces systèmes intégrés comme la téléphonie mobile, la télévision numérique, le son numérique, ou la visiophonie sont régis par des normes (par exemple GSM, DECT, et UMTS, MPEG2, MPEG2-audio et Dolby, H.261 et H.263) [7][40][1]. Pour suivre l'évolution constante de ces normes, et également du marché, il est indispensable d'accroître la flexibilité des systèmes intégrés [36][44]. Un système purement matériel (composés de blocs câblés spécifiques) coûte très cher à faire évoluer. C'est pourquoi il devient nécessaire d'intégrer dans ces systèmes une part de programmabilité, en remplaçant certains opérateurs par des processeurs associés à un logiciel. La haute densité d'intégration permet de réunir sur une même puce le couple processeur-logiciel, lui-même embarqué dans le système complet. Nous parlons alors de processeurs embarqués [22]. Le logiciel est généralement stocké dans une mémoire non-volatile (ROM, EEPROM). Le suivi des normes au cours de la vie du produit consiste alors à remplacer le logiciel, sans nécessairement re-concevoir le matériel. Un processeur embarqué se caractérise donc par plusieurs aspects :

- le processeur est embarqué dans un système et intégré sur la même puce,
- le logiciel qui tourne dessus est également embarqué,
- ce logiciel est destiné à être souvent modifié,
- les performances puis le coût en surface et la consommation sont critiques.

Il existe essentiellement trois types de processeurs embarqués, pouvant remplacer un circuit câblé. La figure 1.1 présente les principaux avantages et inconvénients des différentes solutions. Une solution à base de processeur standard offre un degré maximum de flexibilité, ainsi qu'un temps de développement très réduit puisqu'aucune conception matérielle n'est nécessaire et que les outils de développement logiciel sont disponibles [61][97]. Par contre le bas coût en surface, la basse consommation et surtout les performances ne sont pas assurés [50]. A l'autre extrême, un ASIC (*Application-Specific Integrated Circuit*) présente toutes les garanties de faible surface, de faible consommation et de performances alors que la flexibilité, le temps de développement et les possibilités de ré-utilisation sont ses points faibles [99]. Deux solutions intermédiaires émergent : l'utilisation d'un processeur standard existant, conçu pour un domaine d'application donné [78][32][96] et la conception d'un

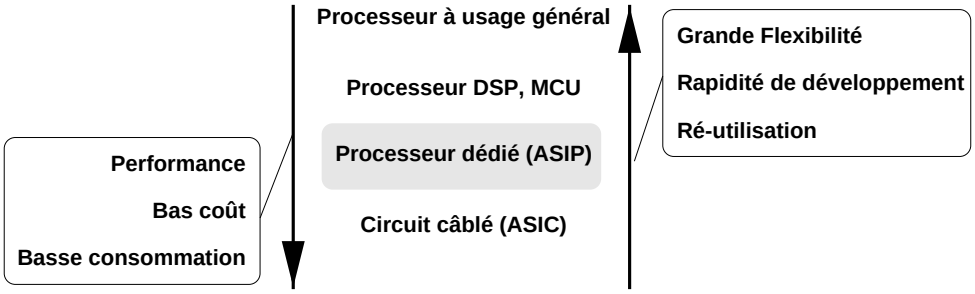

FIG. 1.1: Comparaison des différentes solutions (de l'ASIC au processeur à usage général)

processeur dédié à une application, aussi appelé ASIP (*Application-Specific Instruction-set Processor*) [85][83][79][107][86]. Dans cette thèse, nous nous concentrerons sur la conception conjointe de ce dernier et de son logiciel embarqué.

Un exemple de cette évolution des techniques de conception de circuits vers plus de programmabilité est le système IVT (*Integrated Videotelephone Terminal*) [53] conçu à SGS-THOMSON. Le premier circuit (STi1100 [95]) respectant la norme H.261 comporte seulement deux processeurs programmables. Avant même que la nouvelle norme H.263 ne soit définitivement figée, une nouvelle génération du circuit est conçue, exploitant la haute densité d'une nouvelle technologie de fabrication. Afin de pouvoir suivre la finalisation de la norme, le système IVT est composé de cinq processeurs embarqués. La mise en concordance des algorithmes avec la norme est ainsi reculée jusqu'au départ du circuit en fonderie, et ne retarde pas la conception de la plupart des parties matérielles. Les travaux de cette thèse s'appuieront sur les besoins exprimés lors de cette nouvelle expérience de conception.

Le logiciel destiné à un processeur embarqué est traditionnellement écrit en grande partie en assembleur. Le délai lié au développement de compilateurs spécifiques et le manque de performances des compilateurs C dans des domaines comme le traitement du signal et le respect du temps réel motivent un tel choix [23]. Pourtant un certains nombre d'arguments poussent maintenant en faveur d'une migration vers un langage de haut-niveau tel que le C [41][16] : la concision des programmes C par rapport aux programmes assembleur, les possibilités de ré-utilisation du code et la validation immédiate des programmes C par exécution sur la station de travail, éventuellement en liaison avec le reste du système. Pour ces raisons, le langage C est de plus en plus employé pour la modélisation et la réalisation d'applications embarquées, ou au moins certaines parties d'entre-elles [22][11].

1.1 Objectifs

L'objectif principal de cette thèse est de réduire le temps de mise au point d'un processeur spécifique et de son logiciel. La validation du logiciel en environnement réel est facilitée par la co-simulation à haut-niveau, alors que l'exploration architecturale et le raffinement de l'encodage du jeu d'instructions sont aidés par la re-configuration automatique d'un compilateur reciblable et l'analyse du code généré.

L'élévation du niveau d'abstraction du logiciel embarqué vers le langage C nécessite une modification profonde du flot de conception logiciel [21]. Celui-ci doit notamment prendre en charge la compilation C et la conception d'architectures nouvelles spécifiques, dans un environnement très contraint en temps de développement [38][49]. La conception d'un logiciel embarqué écrit en langage C peut être décrite par le flot de la figure 1.2. La plupart des outils de co-simulation existants effectuent une co-simulation assembleur-VHDL, donc à partir du code binaire du logiciel. Ce procédé retarde l'étape de validation (le compilateur

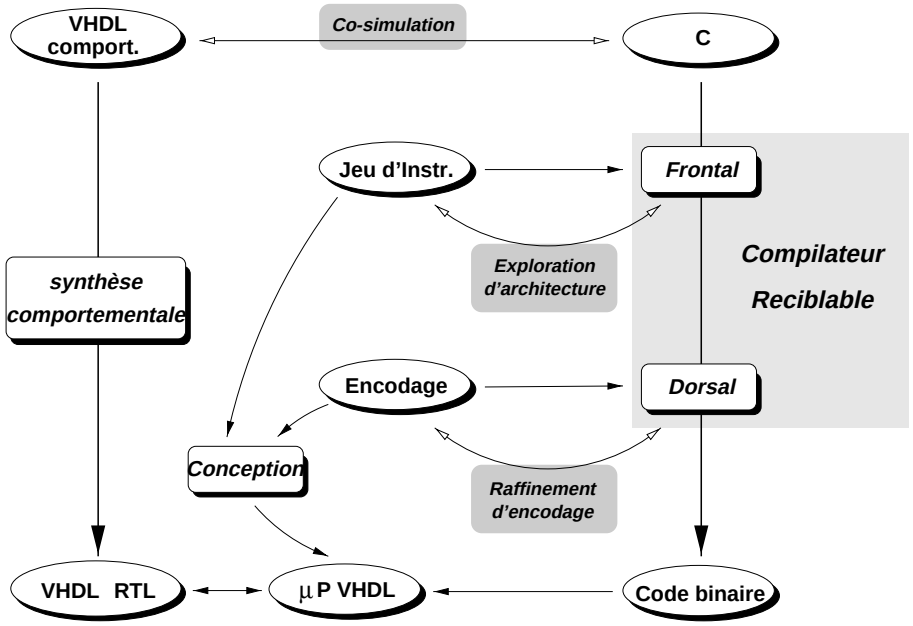

FIG. 1.2: Flot de conception d'un système contenant un processeur embarqué spécifique

doit être développé) et nécessite un modèle matériel du processeur. Au contraire, la co-simulation C-VHDL permet de s'affranchir de ces contraintes tout en conservant un niveau d'abstraction élevé du logiciel.

Au centre, le jeu d'instructions du processeur embarqué sert de spécification pour le compilateur, et pour le processeur lui-même. Le code binaire de l'application produit par le compilateur est finalement chargé dans la mémoire du processeur, lequel est connecté au reste du système. La définition du jeu d'instructions passe par une phase d'exploration (figure 1.2). Cette phase consiste à évaluer plusieurs alternatives liées à l'architecture, afin d'identifier celle qui est le mieux adaptée (en performances, taille de code et consommation) à l'application visée. La conception classique d'un jeu d'instructions s'effectue indépendamment du compilateur, généralement avant. L'interaction est ainsi très limitée, ce qui entraîne une mauvaise adaptation du jeu d'instructions aux besoins de l'application et du compilateur. C'est pourquoi l'exploration architecturale basée sur l'analyse du code issu du frontal du compilateur permet de prendre en compte le fonctionnement particulier de celui-ci.

Enfin, le raffinement qui consiste à faire évoluer un jeu d'instructions existant, pour qu'il s'adapte plus efficacement à l'application, est introduit. Le processus de raffinement se différencie de l'exploration par son espace des solutions plus réduit. Une méthodologie incluant le raffinement est indispensable au développement et au support de familles de processeurs par les fabricants de semi-conducteurs. La réduction du coût lié à la re-conception *(re-design)* est alors considérable. Nous nous intéresserons plus particulièrement au raffinement de l'encodage du jeu d'instructions, tel qu'il est situé en figure 1.2. Les étapes d'itération de l'exploration et du raffinement doivent être effectuées le plus rapidement possible de façon à étudier le maximum de solutions possibles dans un minimum de temps. Cette rapidité d'itération nécessite l'emploi de techniques de ré-utilisation (généricité des outils de base) et de génération automatique performantes [82]. L'exploration et le raffinement d'architecture sont illustrés par l'application des deux outils FlexPlore et ImmPlore.

1.2 Contributions

Les principales contributions de cette thèse sont :

- la spécification, le développement et l'application de l'outil CoSim à SGS-THOMSON, basé sur les principes d'un générateur d'interface développé au laboratoire TIMA[1] .
- Le développement de trois compilateurs reciblés pour les processeurs MSQ, BSP et VIP du système IVT.
- La conduite des explorations architecturales basées sur l'environnement FlexPlore, et l'exploitation des résultats collectés pour le processeur expérimental D960.
- La conception et le développement d'un prototype d'outil de raffinement de l'encodage des champs immédiats, ImmPlore, et son application à plusieurs architectures DSPs.

1.3 Plan de la thèse

Le déroulement de cette thèse suivra les étapes suivantes.

Le chapitre 2 présente le contexte industriel dans lequel se sont déroulés ces travaux, et plus particulièrement les architectures de processeurs étudiées. Il expose également les principes de l'outil de compilation reciblable Flexcc, ayant permi de produire des compilateurs pour toutes ces architectures. Enfin, ce chapitre décrit les modifications et extensions qui y ont été apportées pour l'adapter au flot de conception existant.

Le chapitre 3 expose la spécification, le développement et l'application d'un outil de co-simulation C-VHDL intégré au flot de conception de SGS-THOMSON, assurant une validation fonctionelle en environnement réel du logiciel écrit en C.

Le chapitre 4 décrit une expérience d'exploration architecturale menée autour du processeur expérimental D960, basée sur le re-configuration automatique du compilateur reciblable Flexcc. L'analyse des résultats est présentée dans le but de proposer une architecture dérivée optimisée en taille de code.

Le chapitre 5 se concentre sur un aspect précis de l'encodage d'un jeu d'instructions : les champs immédiats. Il présente un prototype d'outil asssurant l'estimation du coût de plusieurs solutions d'encodage proposées par le concepteur, grâce à l'analyse de code assembleur pré-compilé. Le raffinement d'un encodage est ainsi facilité et accéléré, autorisant une exploration plus exhaustive des solutions.

Enfin le chapitre 6 présente les conclusions et perspectives attachées aux travaux menés dans le cadre de cette thèse.

[1] Laboratoire TIMA : Techniques de l'Informatique et de la Microélectronique pour l'Architecture d'Ordinateurs. `http ://tima-cmp.imag.fr`

Chapitre 2

Contexte industriel

Ce chapitre présente les architectures des processeurs qui ont été utilisés dans le cadre de cette thèse, ainsi que la chaîne de compilation reciblable Flexcc. En troisième partie la contribution à la compilation reciblable, pour plusieurs de ces architectures, est présentée.

Les travaux de cette thèse sont directement reliés à des projets et outils développés à SGS-THOMSON. Les directions de recherche sont fortement influencées par les besoins exprimés dans l'entreprise, dont l'environnement de développement matériel et logiciel sert de lieu privilégié pour valider par la pratique les principes et outils développés au cours de la thèse. Ce chapitre présente les architectures de processeurs et les outils de développement avec lesquels ces travaux ont été menés. Dans un premier temps, le système mono-puce IVT est décrit, ainsi que ses trois processeurs embarqués. Ensuite deux circuits de SGS-THOMSON, le processeur de traitement du signal dédié DAP et le coeur standard de traitement du signal D950, sont présentés.

La chaîne de compilation reciblable Flexcc (principes et outils) qui a servi de base à tous les développements liés à la compilation est alors décrite. Enfin, plusieurs améliorations qui ont été apportées à la chaîne existante afin de répondre à des besoins particuliers (facilités et optimisations d'écriture, adaptation au flot de conception) sont présentées. Elles constituent le travail de base nécessaire à la mise en place d'un flot de conception-validation complet, tel qu'il a été appliqué dans le projet IVT et relaté dans le chapitre 3 sur la validation fonctionnelle par co-simulation C-VHDL.

2.1 Le système Integrated Videotelephone Terminal (IVT)

2.1.1 Le Micro-Sequencer (MSQ)

Fonctionnalité - architecture

Le MSQ (*Micro-SeQuencer*) est le contrôleur du circuit. Il coordonne l'exécution des tâches sur tous les opérateurs, y compris les processeurs. Une partie des choix de codage des données sur la ligne téléphonique doit être effectuée au niveau du MSQ [70], car lui seul a connaissance de l'état de tous les opérateurs. L'architecture du MSQ est présentée en figure 2.1. C'est un processeur 8 bits, dans lequel est intégré l'ensemble de l'application MSQ (ROM programme). Les instructions sont codées sur 19 bits. Les bus instructions et données sont séparés, de façon à exécuter systématiquement une instruction par cycle. Les instructions sont directement décodées et commandent les blocs fonctionnels du processeur. Le coeur opératoire (parties non-grisées) est constitué d'une U.A.L. (Unité Arithmétique et Logique) d'un banc de registres 8 bits (données), d'un accumulateur 8 bits et d'un compteur programme. Les instructions implémentées dans l'U.A.L. sont conçues pour effectuer les

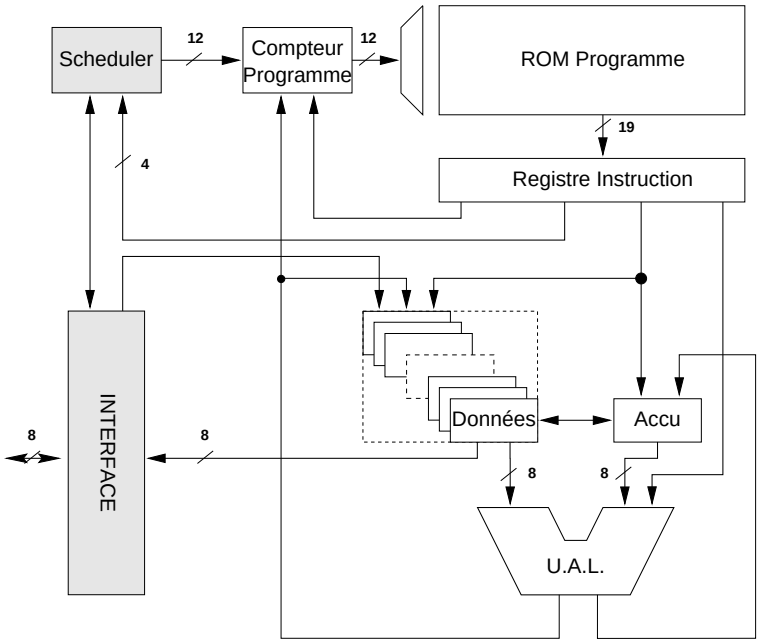

FIG. 2.1: Architecture générale du MSQ

tâches propres au contrôle que sont les manipulations d'octets bit à bit (masquage) en un cycle ou bien les branchements conditionnels complets.

Blocs spécifiques

Les parties grisées représentent les blocs spécifiques au MSQ. Le reste constitue le coeur opératoire générique, utilisé par les autres processeurs de l'IVT. L'interface gère l'échange de données avec l'extérieur du processeur, c'est-à-dire les opérateurs ou la mémoire centrale. Par l'intermédiaire de cette interface, le MSQ peut accéder en lecture et en écriture à certains registres (de statut et de contrôle) des autres opérateurs.

Le second bloc spécifique au MSQ est l'ordonnanceur (*Scheduler*). En effet, le MSQ gère plusieurs tâches en parallèle (chaque étape de codage ou décodage des images est associée à une tâche de contrôle), et exécute donc plusieurs séquences en parallèle. Afin d'éviter le développement, coûteux en temps, d'un noyau opératif (*kernel*), l'ordonnancement des tâches est contrôlé par le matériel. A chaque tâche est associé un registre contenant l'adresse en mémoire programme du début de la séquence de code correspondante. Ce registre est vidé dans le compteur programme pour activer cette tâche. L'hypothèse est faite que toute tâche rend le contrôle (se termine) au bout d'un laps de temps donné en exécutant l'instruction `SCHED`. Celle-ci ordonne l'exécution de la tâche suivante. Les priorités entre les tâches peuvent être gérées dynamiquement.

2.1.2 Le Bit Stream Processor (BSP)

Fonctionnalité - architecture

Le BSP (*Bit Stream Processor*) est un processeur 8 bits dédié au traitement du flot de bits envoyé et reçu sur la ligne téléphonique. Ce traitement consiste à décoder les trains de bits selon des formats donnés pour en extraire des images, qui seront stockées en mémoire centrale (VideoRAM). Inversement, les images à transmettre sont encodées puis envoyées sur la ligne. De plus, dans le but de limiter le nombre de bits transférés sur la ligne,

un codage à longueur variable est utilisé. Il nécessite l'implantation de deux algorithmes spécifiques (un dans chaque sens de transmission) : le VLC (*Variable Length Coding*) et le VLD (*Variable Length Decoding*).

Pour réaliser ces traitements, le même coeur opératoire que celui du MSQ est ré-utilisé (figure 2.2). A ses côtés, plusieurs opérateurs matériels sont ajoutés pour traiter efficace-

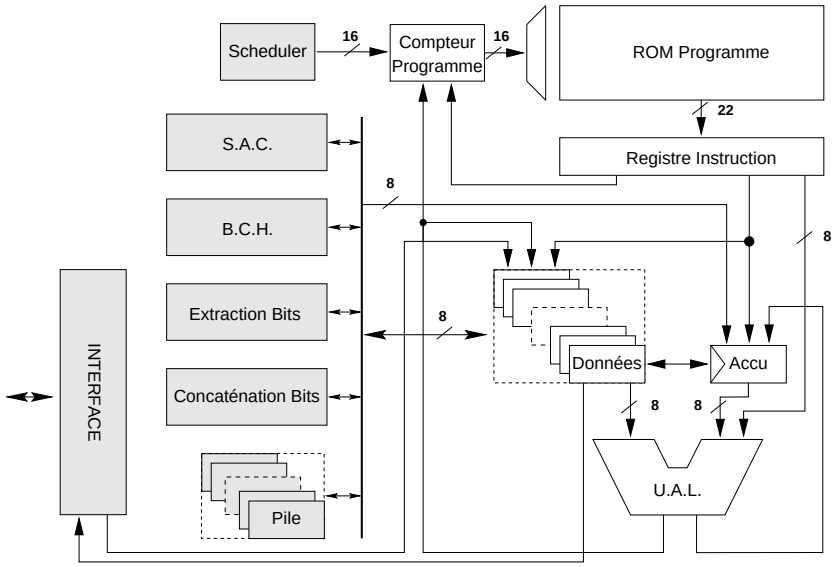

FIG. 2.2: Architecture générale du BSP

ment le flot de bits. A chaque opérateur sont associées plusieurs instructions spécifiques permettant d'y accéder, selon un format adapté aux paramètres à transmettre. Un banc de registres additionnel (Pile) est associé aux opérateurs afin d'échanger des données avec le coeur opératoire de manière performante. Par contre, aucun pointeur de pile d'exécution n'est disponible. Le stockage des adresses de retour lors d'appels de fonctions est assuré par un ensemble restreint de registres (2). Ce mécanisme permet donc d'imbriquer seulement deux appels de fonctions. Comme il n'y a pas de pointeur de pile, les variables locales ne sont pas autorisées.

Blocs spécifiques

Le SAC est un opérateur destiné à l'algorithme SAC, contenu dans la norme H.263. Il comporte essentiellement les opérations de multiplication et division sur 32 bits. Dans la version actuelle du processeur BSP, correspondant à une première version du système IVT, cet algorithme n'est pas implanté.

L'opérateur BCH est associé à la correction d'erreur à base de polynômes. Deux registres de 32 bits (Poly et Bch) issus de la pile sont utilisés pour échanger ces valeurs avec l'U.A.L. (via l'accumulateur). Le résultat du calcul, effectué en seulement un cycle, est disponible dans le registre Bch sur 32 bits pour être traité par le coeur.

L'instruction `XTA` permet d'accéder au bloc d'extraction de bits. Elle a pour but d'extraire un certain nombre (variable) de bits à partir d'une certaine position, dans une valeur 32 bits stockée dans un registre de la Pile. L'instruction `CAT` du bloc de Concaténation de bits concatène un certain nombre de bits dans un registre 32 bits, également par l'intermédiaire de la Pile. L'accumulateur est utilisé pour stocker les bits à concaténer.

Le bloc Scheduler est comparable à celui implanté dans le MSQ. Il sert ici à gérer l'exécution de plusieurs tâches en parallèle (VLC, VLD, Framer et Unframer). Le bloc

d'interface permet au BSP de communiquer avec les autres processeurs, principalement le MSQ (contrôle) et le MCC (mémoire).

Mémoires

Le banc de registres du coeur opératoire est composé de 512 registres de 8 bits. Une instruction spéciale, `LDX`, permet d'adresser ces registres via un index, contenu dans l'accumulateur. La ROM programme a une largeur de 22 bits, et une hauteur de 64 Koctets. Une ROM données de 4Koctets sur 8 bits est également implantée dans le BSP, afin de stocker des coefficients figés.

Technique d'inversion de bits

Parmi les 22 bits du mot-instruction, un bit d'inversion est réservé pour minimiser la consommation de courant lors de la lecture des instructions. En effet, la lecture dans une ROM d'un 0 consomme nettement plus de courant que la lecture d'un 1 (jusqu'à six fois plus). Or nous avons constaté que dans un code binaire classique, les bits 0 et les bits 1 ne sont pas également répartis. Ainsi il apparaît en moyenne 70% de 0 pour 30% de 1. Il est donc simple d'inverser cette tendance et même d'optimiser l'encodage en associant à chaque instruction un bit d'inversion. Ce bit indique si les autres bits du mot-instruction doivent être inversés avant d'être transmis au décodeur d'instruction. Le compilateur se charge de mettre le bit d'inversion à 0 ou à 1 selon le nombre relatifs de 0 et de 1 dans chaque instruction générée. Les résultats de l'expérimentation de cette technique sont présentés plus loin (2.4.7).

2.1.3 Le Vliw Image Processor (VIP)

Le processeur VIP *(Vliw Image Processor)* est associé à la tâche de prédiction d'images. Dans la norme H. 263, l'image est divisée en macro-blocs, eux-mêmes divisés en blocs. Au final, la partie d'image élémentaire est un bloc de 8 points sur 8. Le débit nécessaire pour assurer une fluidité suffisante des images transmises impose qu'une ligne de 8 points puisse être traitée par le VIP à chaque cycle du système. Pour respecter ces contraintes de performances, l'architecture du VIP est composée de huit unités de calcul fonctionnant en parallèle, en supplément de l'U.A.L. fournie par le coeur opératoire. Ces huit U.A.L. parallèles sont identiques et spécifiques. Elles proposent exactement les fonctionnalités requises pour le traitement numérique des points élémentaires, comme le décalage à gauche d'un des deux opérandes, le décalage à droite du résultat de l'opération, ou l'accumulation. Le fait que ces unités soient conçues sur mesure permet de conserver une taille relativement réduite du processeur malgré la multiplication des ressources. La figure 2.3 décrit l'architecture du VIP.

La commande des huit U.A.L. parallèles du VIP se fait grâce à un mot-instruction large [56]. Ce mot est composé de deux parties. La première partie concerne le coeur opératoire du processeur, et est similaire au jeu d'instructions du MSQ ou du BSP pour la plupart des instructions. Une seconde partie du mot-instruction est dédiée aux U.A.L. parallèles, lorsque le traitement d'une ligne complète est nécessaire. Cette seconde partie est elle-même subdivisée en huit champs, chacun accédant à une U.A.L. parallèle donnée. Ainsi un seul mot-instruction de 32 bits commande l'ensemble du processeur (coeur opératoire et unités parallèles). Les opérations effectuées par les unités parallèles peuvent évidemment être toutes différentes, selon les besoins. En effet, sur une ligne de huit points les points à l'extrême gauche et à l'extrême droite doivent subir un traitement particulier pour assurer la continuité du traitement entre les blocs.

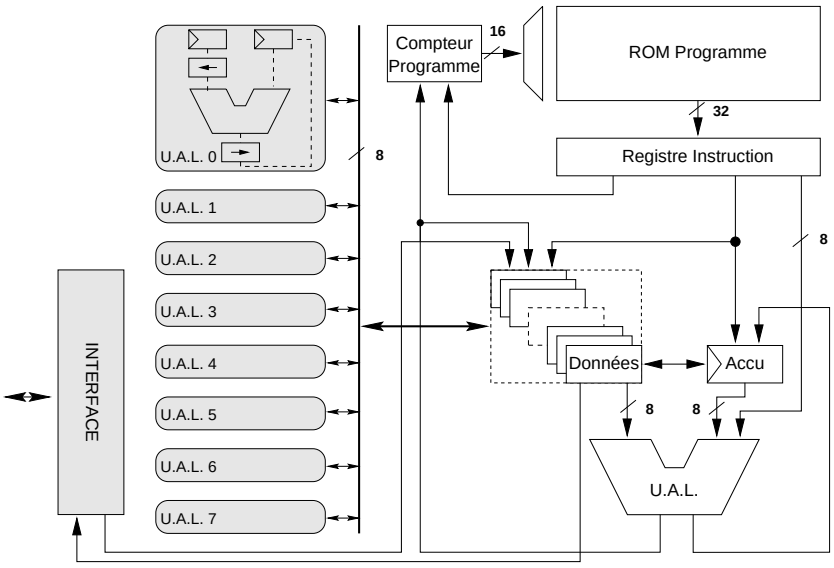

FIG. 2.3: Architecture générale du VIP

Le bloc d'interface permet au VIP de dialoguer avec le MSQ pour le contrôle et avec le MCC pour les accès à la mémoire centrale.

2.2 Les processeurs DSP

Deux processeurs de traitement du signal, développés à SGS-THOMSON, ont servi d'exemples d'architectures DSP durant cette thèse. le DAP est un processeur dédié, bas-coût, alors que le D950 est un coeur à usage général (dans le domaine du traitement du signal).

2.2.1 Le Digital Audio Processor (DAP)

Architecture matérielle

Le DAP (*Digital Audio Processor*) est un processeur de traitement du signal 16 bits dédié, conçu par la division DPG *(Dedicated Product Group)* de SGS-THOMSON. Il est spécialement adapté à un type d'application très précis : le traitement du son audio pour des appareils grand-public, à bas coût. Ainsi les données ont une largeur courante de 16 bits alors que le jeu d'instructions est codé sur 24 bits, autorisant un degré de parallélisme intéressant pour les fonctions de traitement du signal classique et une orthogonalité suffisante pour une compilation C efficace.

2.2.2 Le D950

Le D950 est un coeur de processeur de traitement du signal standard. Il est destiné à être intégré dans un circuit dédié en y associant des périphériques, issus de librairies de composants ou spécifiquement conçus. Ces périphériques peuvent être : un co-processeur, un contrôleur d'interruptions, un contrôleur de DMA (*Direct Memory Access*), un contrôleur d'entrées/sorties parallèles, un contrôleur de bus (*Bus Switch*), ou enfin une unité de test et d'émulation.

Cette architecture sert de base à une évolution, le D960, dont l'objectif principal est de permettre une bonne densité de code tout en facilitant la production de code efficace par

un compilateur.

Architecture matérielle

Le coeur (figure 2.4) est composé d'une unité de calcul sur les données (DCU : *Data Calculation Unit*), de deux unités de calcul d'adresses similaires, associées aux deux bancs mémoire X et Y (ACU : *Address Calculation Unit*), et d'une unité de contrôle de flot (PCU : *Program Control Unit*), non représentée. Trois espaces mémoire séparés sont gérés :

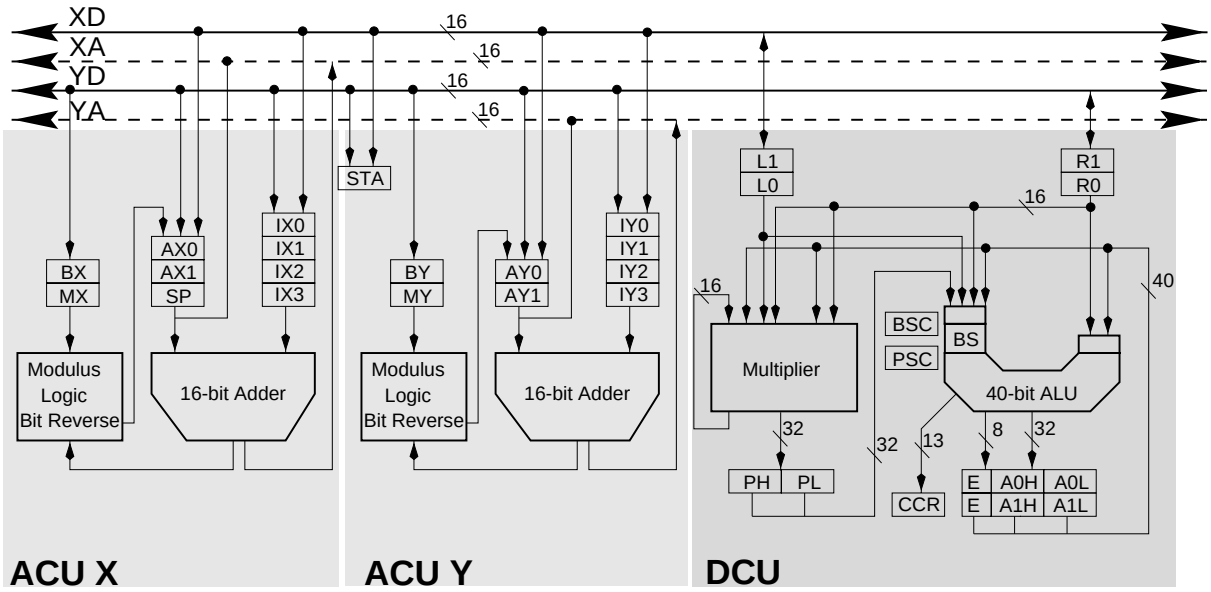

FIG. 2.4: Architecture générale du D950

une mémoire programme ROM, et deux mémoires données X et Y en RAM. Les unités communiquent ensemble et avec les mémoires grâce à cinq bus de 16 bits : le bus instruction connecté à la PCU, deux bus d'adresses XA et YA, et deux bus de données XD et YD.

Chaque unité de calcul d'adresses (ACU) est centrée autour d'une unité opérative produisant une addition sur 16 bits ainsi que la logique de modulo et d'inversion de bits (*bit-reverse*). Trois groupes de registres sont disponibles dans ACU/X : les deux registres d'adresse (AX0 et AX1), les quatre registres d'index (IX0 à IX3), et les deux registres de gestion du modulo (BX et MX). La situation est identique dans l'ACU/Y. Par contre, seule l'ACU/X dispose d'un registre de pointeur de pile, SP. Cela implique que la pile est exclusivement gérée en mémoire X.

L'unité de calcul de données est composée d'un multiplieur 16 bits vers 32 bits avec arrondi programmable, ainsi que d'une U.A.L. complète. L'U.A.L. dispose d'un additionneur 16 bits. De plus, chacune des deux entrées peut subir une extension de signe jusqu'à 40 bits. L'une des deux entrées dispose d'un décaleur (*Barrel-Shifter*) programmable. La sortie du multiplieur peut être directement introduite dans l'une des entrées de l'U.A.L. afin d'assurer une multiplication accumulation efficace. Deux registres gauche (L0 et L1) et deux registres droit (R0 et R1) assurent le stockage des opérandes en entrée, aussi bien pour le multiplieur que pour l'U.A.L. La sortie du multiplieur est stockée dans un registre 32 bits (P), alors que la sortie de l'U.A.L. est destinée à deux accumulateurs (A0 et A1) de 40 bits chacun.

L'unité de contrôle de flot (PCU) dispose des moyens classiques de branchements conditionnels et inconditionnels, d'interruption, d'appel de fonctions mais également de boucles

câblées. Trois groupes de trois registres (LC, LE et LS) chacun peuvent gérer jusqu'à trois niveaux d'imbrication de boucles câblées. Le registre LC contient le compteur de boucle, décrémenté jusqu'à 0 ; le registre LE contient l'adresse de fin du corps de boucle ; et le registre LS contient l'adresse du début de corps de boucle.

Encodage du jeu d'instructions

Le jeu d'instructions du D950 est constitué d'instructions de largeur fixe, 16 bits. Cette largeur réduite est sensée assurer une bonne densité de code. Pour atteindre ce degré de densité, l'orthogonalité de jeu d'instructions a été sacrifiée.

Définition Un jeu d'instructions est considéré comme orthogonal lorsque aucune restriction sur le choix des opérandes (et notamment des registres) n'entrave la composition des instructions [44]. Ainsi une opération d'addition pouvant prendre n'importe quel registre du circuit en entrée, aussi bien à gauche qu'à droite, est orthogonale alors que si l'entrée gauche ne peut pas utiliser un registre de sortie du multiplieur, par exemple, l'opération n'est plus orthogonale.

Il est donc clair que l'orthogonalité d'un jeu d'instructions coûte cher en largeur d'instruction, puisqu'il faut pouvoir coder tous les registres du circuit, pour chaque opérande. Les contraintes de composition des instructions, et de choix des registres en particulier, sont très difficile à gérer pour un compilateur. Chaque contrainte est une situation particulière, devant être gérée spécifiquement. La conséquence d'un encodage du jeu d'instructions aussi peu orthogonal est que le compilateur produit un code peu performant, aussi bien en temps d'exécution qu'en taille de code en mémoire programme.

C'est pourquoi une évolution du D950, appelée dans cette thèse D960, a été étudiée par les équipes de conception de SGS-THOMSON. Les expérimentations menées pendant cette thèse autour de l'architecture du D950 visent à définir les choix d'encodage et quelques modifications architecturales permettant d'aboutir à un D960 plus performant en taille de code.

2.2.3 Le D960

Le D960 est un projet d'architecture développé par le groupe PPG. C'est une évolution du D950 vers une largeur des données de 24 bits. La plupart des solutions architecturales sont conservées, permettant la ré-utilisation de certains blocs existants. De plus, la compatibilité logicielle doit être assurée (les programmes du D950 tourneront sur le D960). Le diagramme du D950 (figure 2.4) convient au D960, à quelques exceptions près :

1. les données ont une largeur de 24 bits,
2. les mots-instructions ont une largeur de 24 bits,
3. l'ALU, le décaleur et les accumulateurs sont sur 56 bits,
4. les registres d'adresse des ACUs sont au nombre de 8 (4 AX et 4 AY),
5. les registres de base des ACUs sont au nombre de 8 (4 BX et 4 BY),
6. deux pointeurs de piles sont disponibles.

La largeur du mot-instructions, étendue à 24 bits, doit permettre une meilleure orthogonalité du jeu d'instructions, et des performances accrues. Par exemple plusieurs instructions supportent l'exécution de déplacements registre-registre (move) en un seul cycle. De plus des opérations de manipulation bit-à-bit sont introduites.

2.3 Chaîne de compilation Flexcc

La chaîne de compilation reciblable Flexcc a été utilisée pour le développement des compilateurs C [84][69] de tous les processeurs étudiés dans cette thèse (DAP, D950, D960, MSQ, BSP, VIP). Les trois derniers constituent une contribution importante dans cette thèse. La chaîne Flexcc est basée sur la technique de reciblage à base de règles, qui a été présentée dans [51]. La compilation complète est divisée en plusieurs phases successives, chacune pouvant être programmée spécifiquement par le développeur. L'enchaînement des phases se rapproche de la méthode classique de compilation [2], comme présenté en figure 2.5.

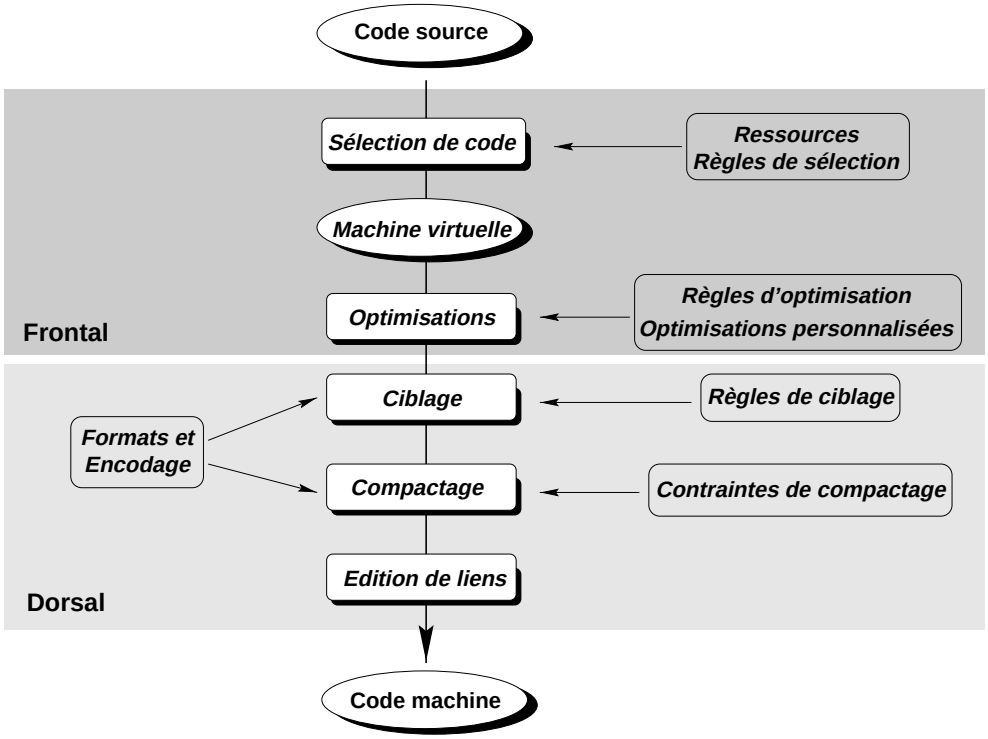

FIG. 2.5: Enchaînement des phases de compilation

Plus précisément, la compilation est divisée en quatre phases principales :

1. La sélection de code virtuel est faite pour une machine virtuelle, entièrement spécifiée par le développeur. Elle peut même se confondre avec la machine cible. Le code généré est strictement séquentiel, repoussant le compactage (exploitation du parallélisme) plus tard dans le flot.
2. L'optimisation consiste à remplacer une suite d'instructions par une autre, selon des règles de substitution définies par le développeur. Les règles acceptent une syntaxe simple à base d'expressions régulières. De plus, cette étape dans le flot est un endroit propice à l'intégration d'optimisations personnalisées, puisque la représentation du code virtuel est assez proche de code final.
3. La traduction vers l'assembleur cible consiste à remplacer les instructions virtuelles en micro-opérations réelles, selon des règles d'équivalence donnée par le développeur. Nous obtenons alors une suite séquentielles de micro-opérations qui peuvent maintenant être compactées si l'architecture le permet.
4. Le compactage de code consiste à produire un code binaire selon les instructions effectivement implémentées dans l'architecture, en respectant les contraintes d'encodage

et de parallélisation. Là encore, un ensemble de règles dictées par le développeur guide le compactage.

Chacune de ces phases est maintenant décrite dans le détail.

2.3.1 Sélection de code virtuel

La sélection de code pour la machine virtuelle se base sur deux types d'informations : la description des ressources (principalement les bancs de registres, les formats d'opérandes, les modes d'adressages) et un ensemble de règles de sélection.

La déclaration des bancs de registres sépare ceux-ci selon plusieurs catégories fonctionnelles, comme par exemple les types de données (en C) que ces registres peuvent stocker. La définition des modes d'adressage contient toutes les façons d'accéder aux variables en mémoire (valeur immédiate, adressage direct, indexé,...).

Les règles de sélection de code consistent à associer à chaque construction C une séquence d'instructions virtuelles. Pour ce faire, le développeur dispose d'un langage de programmation relativement riche, proche du C, auquel sont ajoutées des primitives de haut-niveau permettant l'accès aux arbres syntaxiques correspondant au code C à compiler. Les règles de sélection sont conditionnées par les types de noeuds rencontrés dans l'arbre, et les contraintes correspondantes sont propagées à travers l'arbre.

Grâce à la richesse des primitives disponibles, le développeur peut aussi bien produire rapidement un ensemble de règles générales convenant à la plupart des cas de figures, qu'isoler certains cas précis qui exigent un traitement complexe. Par exemple les restrictions d'allocation des registres pour certaines opérations peuvent être ainsi prise en compte, lorsqu'elles apparaissent. Ces restrictions sont très répandues dans les architectures de processeurs embarqués.

Après la sélection de code, une allocation des registres automatique est appliquée. L'approche d'allocation des registres est relativement standard. Elle est décrite plus en détail dans ([51]).

2.3.2 Traduction vers l'assembleur cible

Cette étape consiste à traduire les instructions virtuelles issues de la sélection de code en instructions réelles, non parallélisées. Cette traduction est généralement assez directe et systématique. Cependant, un langage proche du C permet de manipuler les instructions virtuelles (conditions selon certaines valeurs, formats) afin de produire un code efficace dans la syntaxe voulue. Il est par exemple possible de tester si deux registres d'une opération sont identiques, auquel cas une instruction triadique peut être réduite en instruction dyadique.

2.3.3 Compactage de code

Cette étape opère un traitement sur le code réel et séquentiel produit précédemment. Les contraintes de parallélisme sont ici prise en compte, et ce indépendamment des phases précédentes. L'algorithme de compactage se base sur deux types de contraintes : les contraintes d'encodage des micro-instructions (instructions séquentielles) dans le mot-instruction et des contraintes d'occupation de ressources fournies par le développeur. Le premier type de contraintes consiste à attribuer à chaque micro-instructions un certain nombre de bits d'encodage dans le mot-instructions, hiérarchisés par des champs pré-définis. Ainsi deux micro-instructions ne peuvent être compactées en une seule instruction que si les champs qu'elles occupent ne se recouvrent pas. Le second type de contraintes permet au développeur de déclarer un nombre quelconque de ressources (bus, registres, mémoires, ...) et d'attribuer

à chaque micro-instruction une liste d'occupation de ces ressources (accès en lecture, en écriture, ou occupation complète). Là encore, le compactage n'a lieu que si aucun conflit d'occupation de ressources n'est constaté.

Cette technique de compactage, qui peut être facilement personnalisée suivant les architectures, présente des résultats généralement très satisfaisants. Cependant, la description des contraintes sous forme de composition de champs de bits semble difficile à manipuler pour des jeux d'instructions très fortement encodés. Il peut alors être nécessaire d'introduire une phase supplémentaire personnalisée, s'appliquant après le compactage classique.

2.3.4 Résumé de l'approche

Bien que la maîtrise des différents langages et concepts introduit par cette approche nécessite un certain temps d'apprentissage, le temps de reciblage est relativement court. En effet, suivant les projets et la complexité des architectures, ce temps peut varier entre 1 homme/mois et 6 hommes/mois. L'intérêt majeur de cette approche basée sur un ensemble de règles est d'être extrêmement flexible. Ainsi l'effort peut être immédiatement porté sur les situations critiques ou compliquées correspondant à des particularités architecturales, grâce à un ensemble de primitives puissantes. Les situations classiques sont généralement bien traitées, automatiquement.

La qualité du code produit dépend essentiellement de l'effort donné sur les cas critiques. S'il est assez rapide d'obtenir un code correct sur la plupart du code (quelques semaines), il est par contre nécessaire d'exploiter les ressources de l'outil au maximum pour appliquer les dernières optimisations. L'expérience a montré que la complexité et la spécificité des optimisations exploitant une architecture très particulière ne permettent pas d'envisager une automatisation réellement efficace de l'ensemble de la génération de compilateur avec cette approche.

La flexibilité des langages, de sélection de code et de compaction notamment, permet d'envisager l'exploration de plusieurs solutions architecturales en un temps réduit, permettant ainsi de fournir au concepteur un avis (*feedback*) précis, basé sur l'analyse de code produit avec telle ou telle configuration. Cette caractéristique de l'approche, très intéressante dans le contexte des processeurs embarqués dédiés, a rendu possible les explorations menées durant cette thèse, comme cela est relaté dans les chapitres suivants.

2.4 Compilation pour processeurs embarqués dédiés

La conception de compilateurs pour des architectures dédiées présente un certain nombre de contraintes liées aux besoins particuliers de ce type de processeurs [67]. Le langage couramment utilisé en développement d'applications, notamment embarquées, est le langage C [88]. Pourtant, sa syntaxe et les fonctionnalités qu'il propose ne remplissent pas toujours les conditions nécessaires pour être utilisées telles quelles. Notamment, les spécificités architecturales rencontrées (espaces mémoires multiples, opérateurs matériels dédiés [75]) sont difficilement maîtrisable avec le le langage C standard[23]. Leur compilation efficace l'est encore moins [23]. Il est parfois apparu indispensable de s'écarter du standard C ANSI [31], ou parfois d'ajouter un pré-traitement du code C avant la compilation. L'ensemble de ces techniques spécifiques est maintenant décrit, et illustré par l'implantation effective de chacunes d'elles dans les trois compilateurs (`msqcc` [70], `bspcc` et `vipcc`) développés dans le cadre de cette thèse. De plus, une technique d'inversion de bits dans le code binaire a été expérimentée sur un des processeurs dans le but de réduire la consommation de courant lors de la lecture des instructions en ROM.

2.4.1 Utilisation du langage C et de ses différents niveaux de codage

Le langage de programmation généralement utilisé dans le milieu industriel pour développer du logiciel, notamment embarqué, est le langage C [41]. Même si le langage C n'est pas idéal dans tous les cas de figures (d'autres langages plus performants dans certains domaines co-existent [52] : Silage, C++, Fortran, Ada,...)[54], la profusion d'outils associés au langage C tels que les débogueurs, les profileurs, ou les gestionnaires de librairie, milite en sa faveur. De plus, beaucoup d'organismes de normalisation (ISO : *International Standards Organization*, ITU : *International Telecommunications Union*) produisent leurs modèles de référence sous forme de programmes C exécutables sur station de travail. Quelques exemples sont le GSM (téléphonie mobile), les H.261 et H.263 (visiophonie), le Dolby (codage du son), le MPEG (codage d'images animées et de sons) ou le JPEG (images fixes).

Le codage d'un algorithme en langage C peut se faire à plusieurs niveaux d'abstraction selon les besoins et les contraintes. Nous pouvons identifier quatre niveaux d'abstraction que l'on retrouve dans la compilation d'applications embarquées, à chacun correspondant un style de codage :

1. Haut-niveau : il s'agit d'un style d'écriture comportemental, parfaitement défini dans la norme ANSI C. Les structures de données les plus complexes peuvent être manipulées.
2. Niveau intermédiaire : la plupart des structures de données de haut-niveau sont transformées en variables simples et pointeurs. Des directives standard peuvent être employées pour stocker certaines variables dans les registres, par exemple.
3. Bas-niveau : à ce niveau, des spécificités architecturales peuvent être directement exploitées (choix de registres précis), mais alors la norme ANSI C n'est plus respectée et la portabilité compromise.
4. Niveau assembleur : le programmeur peut ici coder directement en assembleur, ou le mixer avec du C d'un niveau supérieur.

Le C à haut-niveau est idéal du point de vue du programmeur, puisque toutes les facilités du C peuvent être exploitées. La lecture d'un tel code est très aisée. La portabilité est maximale. Par contre, la compilation efficace d'un C de haut-niveau représente un véritable défi pour la plupart des compilateurs, a fortiori reciblables.

Le niveau intermédiaire est nettement moins difficile à compiler, tout en conservant une facilité de programmation suffisante. L'indépendance par rapport au matériel est encore assurée. L'allocation de registres par le compilateur est facilitée (donc plus performante) par les directives fournies par le programmeur.

Le C de bas-niveau fait largement appel à des constructions non-standard, permettant de renseigner plus précisément le compilateur sur certaines spécificités matérielles à exploiter. Il peut toutefois être compilé sur la station de travail, pourvu que des librairies adaptées soient disponibles (librairies *bit-true*, entrées/sorties).

Les figures 2.6, 2.7, 2.8, 2.9 donnent des exemples d'une même opération écrite à plusieurs niveaux. A haut-niveau (2.6), la construction de boucle for est utilisée, de même que des tableaux avec indices.Au niveau intermédiaire (2.7), la boucle for est remplacée par une fonction pré-définie (boucle `loop`) laissant l'opportunité au compilateur de la réaliser en boucle câblée[1]. Les références de tableau sont remplacées par des pointeurs, une fonction pré-définie (`MULT`) est invoquée afin d'exploiter directement le multiplieur, et enfin certaines variables sont placées dans un espace mémoire spécifique (`MEMORY2`). A part les bancs mémoires, aucune spécificité architecturale n'est explicitement désignée.A bas niveau (2.8) les

[1]Une fonction pré-définie permet d'accéder directement à certaines instructions spécifiques par l'intermédiaire du compilateur, comme expliqué plus loin en détail (2.4.6).

```
int a[10], b[10],i ;

for(i=0 ;i<9 ;i++)
{
        b[i] = a[i] * b[i+1] >> 2 ;
}
```

FIG. 2.6: Exemple au niveau 1 : C haut-niveau

```
int b[10] ; int *bp, *bp1 ;
int _MEMORY2_ a[10] ;
int _MEMORY2_ *ap ;

ap = &a[0] ;
bp = &b[0] ; bp1 = &bp[1] ;

loop(9)
{
        *bp = MULT(*ap,*bp1) >> 2 ;
        bp++ ; ap++ ; bp1++ ;
}
```

FIG. 2.7: Exemple au niveau 2 : C niveau intermédiaire

pointeurs sont explicitement assignés à des registres spécifiques. De plus, l'utilisation de registres spécifiques dans le corps de boucle permet d'effectuer manuellement du pipeline logiciel (*software pipelining*). Enfin, au niveau assembleur (2.9) l'assembleur en ligne est utilisé pour exploiter directement certaines instructions assembleur et certains registres (`L0`, `R0` et `L1` servent à passer les valeurs aux instructions). Le traitement de l'assembleur en ligne dans un compilateur n'est pas simple. Les contraintes imposées par les instructions assembleur (notamment l'utilisation des registres) doivent être propagées au reste du code C. La gestion de ces contraintes se complique encore lorsque plusieurs insertions d'assembleur sont introduites à plusieurs endroits dans le code.

2.4.2 Configuration mémoire

Le langage C ne propose aucune directive permettant de gérer plusieurs espaces mémoires, et d'affecter des variables ou des fonctions à ceux-ci. Pourtant, il est commun dans un processeur embarqué et tout particulièrement dans un DSP de disposer de plusieurs espaces. Nous pouvons citer notamment :

- un espace mémoire programme isolé, généralement en ROM,
- un ou deux espaces données en RAM, pour manipuler deux données en parallèle (fonction de multiplication-accumulation commune en traitement du signal),
- un espace de données supplémentaire, éventuellement en ROM, pour stocker des coefficients ou des tables (algorithmes de codage à longueur variable),
- un espace destiné aux registres d'entrées-sorties représentés en mémoire (*memory-mapped I/O*).

```
int b[10] ;
register int *bp At_reg(AX[0]) ;
register int *bp1 At_reg(AX[1]) ;

int _MEMORY2_ a[10] ;
register int _MEMORY2_
        *ap At_reg(AY[0]) ;

register int x ;
register int _RIGHT_ y ;

ap = &a[0] ;
bp = &b[0] ; bp1 = &bp[1] ;

x = *ap ; y = *bp1 ;
loop(9)
{
        *bp = MULT(x,y) >> 2 ;
        x = *ap ; y = *bp1 ;
        bp++ ; ap++ ; bp1++ ;
}
```

FIG. 2.8: Exemple au niveau 3 : C bas-niveau

Les différents processeurs pour lesquels des compilateurs ont été écrits durant cette thèse ont tous présenté une combinaison de plusieurs de ces configurations. Il est donc absolument nécessaire de pouvoir introduire, dans un programme C standard, des directives appropriées. Le traitement de ces directives spéciales doit être effectué avant la phase de compilation par le frontal de Flexcc, puisque celui-ci ne reconnaît que le C standard (à quelques exceptions près). Un pré-processeur traitant ces directives est donc spécialement développé (`prep`). Il est alors facile d'ajouter au programme C un ensemble de directives destinées à la configuration mémoire, ayant la syntaxe appropriée.

Cependant, une autre contrainte vient s'ajouter : le programme C qui est pré-processé puis compilé doit également être simulé sur la station de travail, et éventuellement co-simulé avec le reste du matériel. Cela implique de compiler le programme sur la station de travail, avec les compilateurs C classiques sous Unix (CC, gcc,...)[98]. Il est donc indispensable de mettre en place certaines conventions sur la syntaxe des directives. Le moyen le plus simple, qui a été choisi, est d'utiliser la primitive du pré-processeur C standard (également reconnue par le frontal Flexcc) appelée `#pragma`. Elle est systématiquement ignorée par tous les compilateurs. Le reste de la ligne sur laquelle est définie une `#pragma` est également ignoré, ce qui permet d'indiquer des commandes et paramètres selon la syntaxe voulue. Seul un outil spécialement conçu, appelé `prep`, reconnaît et interprète ces pragmas. Il est inséré dans la chaîne de compilation avant le frontal de Flexcc, comme indiqué sur la figure 2.10.

Cette technique a été appliquée au compilateur pour l'architecture BSP du système IVT (2.1.2). Ce processeur contient en effet deux espaces mémoires disjoints (une ROM et une RAM) destinés aux données. De plus, l'espace RAM est divisé en deux parties. Ainsi, certaines variables et tables de coefficients sont placées dans l'un de ces trois espaces. La directive `#pragma mem_select`, dont la syntaxe est donnée en figure 2.11, est donc

```
int b[10];
register int *bp At_reg(AX[0]);
register int *bp1 At_reg(AX[1]);

int _MEMORY2_ a[10];
register int _MEMORY2_
        *ap At_reg(AY[0]);

register int x At_reg(L[0]);
register int y At_reg(R[0]);
register int z At_reg(L[1]);

ap = &a[0];
bp = &b[0]; bp1 = &bp[1];

x = *ap; y = *bp1;
loop(9)
{
        INLINE(L[0],R[0]);
         mult L0, R0, L1
         left_shift L1, 2, L1
        END_INLINE(L[1]);
        *bp = z;
        x = *ap; y = *bp1;
        bp++; ap++; bp1++;
}
```

FIG. 2.9: Exemple au niveau 4 : niveau assembleur

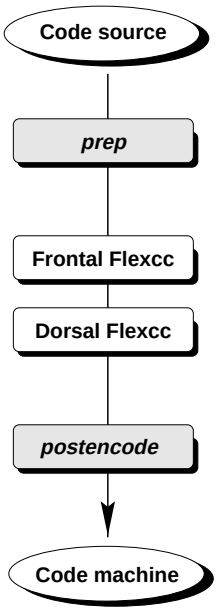

FIG. 2.10: Personnalisation du flot de compilation Flexcc

introduite.Le mot-clé `mem_select` indique que cette pragma est destinée à la configuration

```
#pragma mem_select <mem_ref> N @ <adr> ;
```

FIG. 2.11: Syntaxe de la directive #pragma mem_select

des données en mémoire. Il sera ainsi possible de définir d'autres pragmas, pour d'autres usages. Le champ `mem_ref` indique la référence mémoire concernée par cette directive, c'est-à-dire le nom de la variable (ou constante, ou tableau). Cette référence est sensée être déclarée ailleurs dans le code C. Le champ `N` représente l'espace mémoire dans lequel la donnée doit être stockée. C'est un nombre entre 0 et 2, indique ainsi le premier espace RAM (0), le second espace RAM (1) ou l'espace ROM coefficients (2). Enfin, le champ `adr` est optionnel. Il contient une adresse absolue, et permet donc de placer manuellement une donnée à une adresse précise dans son espace. Cette directive est prioritaire par rapport aux autres données du programme, qui seront automatiquement placées par le compilateur en respectant cette contrainte. Si plusieurs directives de placement manuel sont incompatibles, l'erreur est signalée. La figure 2.12 donne un extrait du code applicatif `vld.c`, destiné à l'architecture BSP de l'IVT, contenant quelques directives de configuration des variables en mémoire. La variable `mux`, qui est en fait un tableau, est ainsi stockée dans le premier

```
#pragma mem_select mux     0 @ 0x10 ;
#pragma mem_select demux   1 @ 0x80 ;
#pragma mem_select table1  2 @ 0x05 ;
```

FIG. 2.12: Exemple de directives pragma pour la configuration de la mémoire (architecture BSP)

espace RAM, à partir de l'adresse hexadécimale `0x10`. Le tableau `demux` est stocké dans le second espace RAM et commence en `0x80`. Le tableau de coefficients `table1` est lui stocké dans l'espace ROM données, à l'adresse `0x05`.

2.4.3 Appel de fonctions

Les trois architectures MSQ, BSP et VIP ne disposent pas de pointeur de pile. L'implantation d'une pile n'a pas été jugée nécessaire, malgré le fait que l'absence de pile interdit l'utilisation de variables locales, et également l'appel de fonctions (impossibilité de stocker les adresses de retour de fonctions et les paramètres). Si l'indisponibilité de variables locales n'a pas été jugée gênante par les concepteurs, l'appel de fonctions, même limité, est indispensable pour organiser le code et réduire sa taille en mémoire programme. Une pile matérielle est donc fournie dans ces trois architectures. Elle est constituée de quelques registres (entre 1 et 3 selon les processeurs), destinés uniquement à stocker les adresses de retour des fonctions. La profondeur des imbrications de fonctions est évidemment limitée au nombre de registres disponibles dans la pile matérielle. Elle est fixée selon les besoins de chaque application embarquée. Le passage de paramètres dans les fonctions n'est également pas assuré. Seul le retour d'une valeur est assuré par les fonctions. La figure 2.13 décrit l'évolution du registre d'adresse de retour lors de l'appel d'une fonction. On considère ici qu'un seul registre d'adresse de retour est disponible.Le programme principal appelle la fonction située à l'adresse `0x100` par l'instruction call. A ce moment l'adresse de retour dans le programme principal (`0x04`) est stockée dans le registre d'adresse de retour (à gauche). La

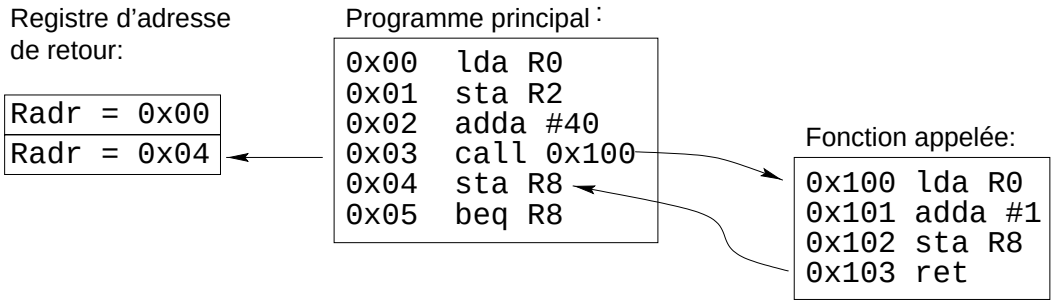

FIG. 2.13: Appel de fonctions avec pile matérielle

précédente valeur de ce registre est écrasée : ici un seul niveau d'imbrication de fonctions est autorisé. A la fin de la fonction appelée, l'instruction ret (en `0x103`) effectue un saut inconditionnel à l'adresse stockée dans le registre d'adresse de retour (`0x04`), afin de poursuivre l'exécution du programme principal.

Les restrictions concernant l'appel de fonctions (pas de paramètres, pas de variables locales, niveau d'imbrication limité) représentent un risque de dysfonctionnement de l'application si elles ne sont pas respectées par le programmeur. C'est pourquoi les compilateurs pour les MSQ, BSP et VIP implémentent la vérification de ces restrictions. Des messages d'erreur sont produits lorsqu'une fonction est appelée avec des paramètres, ou si elle contient des variables locales, ou encore si le niveau d'imbrication de fonctions maximal est dépassé.

2.4.4 Expansion en ligne

L'expansion en ligne (*in-line expansion*) consiste à insérer le code d'une fonction à l'endroit où elle est appelée, au lieu d'effectuer l'appel standard. L'avantage est d'économiser (supprimer) toutes les instructions de traitement de l'appel (branchements, sauvegarde de contextes, passage de paramètres). L'inconvénient majeur est que le code d'une fonction expansée est dupliqué autant de fois que la fonction est appelée, ce qui entraîne un surcoût en taille de code qui peut être inacceptable. L'utilisation de l'expansion en ligne se justifie particulièrement dans le cas où le nombre d'appels de fonctions imbriqués est limité (voir paragraphe 2.4.3).

L'implantation standard de l'expansion en ligne dans la chaîne de compilation Flexcc utilise une directive du pré-processeur spéciale, appelée `#macro`. Elle est placée devant la fonction à expanser. Cette directive est spécifique au pré-processeur de Flexcc, et n'est donc pas reconnue par les compilateurs C standard sous Unix. Lorsque l'application est déboguée sous Unix, il faut donc faire passer le pré-processeur de Flexcc avant le compilateur C. Mais alors le code C donné au compilateur est déjà expansé, et donc débogué sous cette forme. Ainsi mis à plat, l'application est difficilement manipulable. Pour éviter ce problème, la directive `#macro` n'est pas utilisée. A la place, une pragma spécifique (appelée `inline`) indique qu'une fonction, définie plus loin, doit être expansée (figure 2.14). Il y a autant de

```
#pragma inline function_name
```

FIG. 2.14: Exemple de pragma d'expansion de fonctions en ligne

directives `#pragma` de ce type que de fonctions à expanser. Ainsi lorsque l'application est compilé sous Unix, les pragmas sont ignorées et les fonctions demeurent des fonctions. Le

déboguage n'est pas pénalisé. Par contre, lorsque l'application est compilée par Flexcc le pré-processeur additionnel `prep`, déjà utilisé pour traiter les pragmas relatives à la configuration mémoire, reconnaît les pragmas `inline` et expanse lui-même les fonctions concernées (2.10).

2.4.5 Insertion d'instructions assembleur

L'insertion de séquences d'instructions assembleur dans le code C permet d'exécuter des instructions très spécifiques à l'architecture tout en gardant une interface C. La référence à une variable C (son adresse) dans une instruction assembleur est autorisée, de même que l'utilisation de certaines classes de registres. L'allocateur de registres du compilateur peut alors prendre en compte ces contraintes et les propager au rester du code C. Un exemple d'insertion de séquences assembleur, telle qu'elle a été implémentée dans le compilateur `vipcc` pour le processeur VIP, est fourni en figure 2.15. Dans cet exemple, les références

```
/$
LDAI Iu#0xa5
ADDI Is#-0x4 RAM#_x
STA RAM#_y
$/
```

FIG. 2.15: Exemple d'insertion d'instructions assembleur en C (`vipcc`)

`RAM#_x` et `RAM#_y` concernent les variables x et y visibles en C.

L'usage d'instructions assembleur insérées dans le programme C doit cependant être limité le plus possible, ne serait-ce que pour conserver l'indépendance (portabilité) du programme C avec le jeu d'instructions.

2.4.6 Fonctions pré-définies

Une fonction pré-définie (*built-in function*) possède la même syntaxe qu'une fonction C standard. Grâce à son nom, elle est reconnue par le compilateur qui génère alors une séquence d'instructions assembleur. Il n'y a pas d'appel de fonction effectif. Cette technique se rapproche de l'insertion d'assembleur en ligne (*in-line assembly*), à ceci près que le code assembleur n'est pas fourni par l'application mais par le compilateur. Cette différence assure l'indépendance du code C avec l'architecture et le jeu d'instructions. En cas de changement de ce dernier, il n'est pas nécessaire de modifier le code C, mais seulement le compilateur.

Les fonctions pré-définies peuvent servir à implanter les opérations qui ne sont pas disponibles dans le langage C[23]. Nous pouvons citer : les instructions et fonctions d'interruption, les boucles câblées (*hardware do-loops*), les mécanismes de blocage en attente (*wait*), les opérateurs matériels spécifiques, la sélection de modes d'adressage spécifiques (*modulo*, *bit-reverse*).

Les fonctions pré-définies ont été largement utilisées dans le compilateur `bspcc`. Le processeur BSP est constitué de plusieurs opérateurs matériels, associés à des opérations particulières et critiques. Un exemple est l'opérateur de calcul du BCH (2.1.2). Ce bloc matériel prend en entrée trois valeurs : une donnée, un nombre de bits entre 0 et 7, et une longueur de polynôme (entre 1 et 31). Une instruction `BCH` assembleur est associée à ce calcul, mais elle ne prend en entrée que deux paramètres (le nombre de bits et la longueur du polynôme). Le troisième, la donnée, doit être chargé dans l'accumulateur avant l'appel de l'instruction `BCH`. Une technique possible, pour utiliser cet opérateur dans le programme C, est d'insérer une séquence d'instructions comme celle donnée en figure 2.16. Cette écriture,

```
/$
LDA RAM#_x
BCH i#0x4 n#0x1f
$/
```

FIG. 2.16: Lancement d'un calcul BCH en assembleur en ligne

bien que correcte, est assez lourde à manipuler dans un programme C. L'utilisation d'une fonction pré-définie permet de la remplacer par une écriture plus pratique, comme en figure 2.17. La syntaxe est exactement celle d'une fonction C standard, mais aucun appel de

```
BCH (x, 4, 31) ;
```

FIG. 2.17: Lancement d'un calcul BCH avec une fonction pré-définie

fonction n'est généré. Seules les deux instructions assembleur (`LDA` et `BCH`) sont produites. En pratique, le compilateur Flexcc n'accepte pas de fonctions pré-définies ayant plus de deux paramètres en entrée. Une indirection supplémentaire, totalement transparente pour le programmeur, a été utilisée pour conserver une interface C simple[2].

Les fonctions pré-définies ont également été largement utilisées pour gérer l'ensemble de registres spéciaux (appelé Pile). Tous les accès à cette Pile sont effectués par l'intermédiaire de deux fonctions pré-définies, `PUSH` et `PULL`. Des exemples sont donnés en figure 2.18. La

```
PUSH( VALID, -128) ;
PUSH ( READER, x) ;
 y = PULL( POLY) ;
```

FIG. 2.18: Exemple d'utilisation de fonctions pré-définies pour accéder à la Pile (`bspcc`)

fonction `PUSH` permet de stocker une valeur (ici -128) dans un registre de la pile même si ce registre a une taille de 32 bits (`VALID`). Dans la deuxième ligne, nous pouvons voir qu'une variable peut également être lue puis transférée dans cette Pile. Enfin, la fonction `PULL` prend le nom d'un registre en entrée et transfère sa valeur dans la variable `y`. Si le registre concerné par le transfert (quel que soit le sens) fait plus de 8 bits, autant de chargements ou déchargements que nécessaire seront automatiquement générés par le compilateur.

Pour terminer, les fonctions pré-définies ont permi d'implanter efficacement les opérations parallèles du processeur VIP. Ce processeur possédant huit unités de calcul en parallèle en plus du coeur opératoire classique, la commande de ces unités en C n'est pas chose facile. Afin d'éviter de développer un algorithme de compactage d'instructions séquentielles spécifique, et d'introduire des contraintes trop fortes sur l'écriture du programme C, quelques fonctions pré-définies accèdent exclusivement aux unités parallèles. Un exemple, issu de l'application de prédiction d'image tournant sur le VIP, est donné en figure 2.19. Chacun des quatre champs de l'instruction `load` correspond à un champ de l'instruction assembleur `load` qui commande les unités parallèles. Le premier champ indique le type d'opération effectuée (chargement simple, rotation des bits de poids faible). Le second contient les valeurs

[2]La fonction BCH fait en réalité appel à deux fonctions pré-définies, l'une chargeant l'accumulateur avec la donnée et l'autre exécutant effectivement l'opération BCH.

```
load (READ, COEFF_12222221, LDA, src[il]);
load (RLSB, COEFF_01111110, ACC, src[il]);
      store (DIV_14444441, des[7]);
```

FIG. 2.19: Exemple d'accès aux unités parallèles du VIP par fonctions pré-définies (`predictor.c`)

des huit coefficients donnés aux huit U.A.L. parallèles. Le troisième champ indique s'il y a accumulation du résultat ou simplement stockage, et enfin le dernier champ donne les pointeurs et index à utiliser. L'instruction `store` fonctionne suivant le même principe, avec seulement deux paramètres.

Le principal intérêt de cette écriture des opérations parallèles spécifiques est de fournir un style d'écriture qui convient aussi bien à la validation de l'algorithme sur la station de travail qu'à la compilation efficace en code cible. En effet, il suffit de définir deux fonctions en C `load` et `store` qui manipulent quelques variables cachées (correspondant aux registres des unités parallèles) pour compiler l'algorithme sur station de travail. Pour la compilation effective, ces mêmes fonctions sont identifiées comme fonctions pré-définies par le compilateur Flexcc et sont alors traduites en instructions assembleur parallèles. Un second avantage de cette écriture et de permettre au programmeur d'accéder très précisément aux ressources spécifiques (opérations des U.A.L.) de l'architecture, sans pour autant rendre le source C dépendant du jeu d'instructions.

2.4.7 Encodage réduisant la consommation

Selon l'étude [114], la lecture de bits 0 dans une ROM embarquée consomme nettement plus de courant que la lecture d'un bit 1[3]. Or si nous observons un code binaire classique, nous trouvons une majorité de 0 (voir les résultats d'analyse plus loin). Nous avons donc expérimenté, sur le processeur BSP un encodage à inversion de bits. Le mot-instruction a une largeur de 21 bits à l'origine. Nous ajoutons un bit supplémentaire (bit 22) qui indique si les autres bits doivent être inversés avant d'être décodés. La figure 2.20) donne un extrait d'un rapport de statistiques portant sur l'algorithme VLD qui tourne sur le processeur BSP.

```
Word Size = 22
Number of Instructions = 2882
Number of Used |Rev| Bit0  | Bit1  | % 0 | Gain
859            | 0%| 40288 | 23116 | 63% | 0%
```

FIG. 2.20: Statistiques sur le premier microcodage du VLD, sans optimisation

L'encodage utilisé pour produire de code binaire est un encodage non optimisé, donc le bit 22 est à 0. Nous pouvons voir que la majeure partie des bits est à 0 (**63%**). Nous activons alors l'option d'inversion de bits du compilateur bspcc. Cette option lance un programme, `postencode` (figure 2.10) qui traite le code binaire en sortie du dorsal de Flexcc. Pour chaque instruction binaire, le nombre de bits à 0 est compté. S'il dépasse 10, tous les bits sont inversés, y compris le bit 22 qui passe à 1. La figure 2.21 donne un extrait du résultat d'analyse. Ce rapport relate que 83% des instructions ont été inversées. Cela a permi de

[3]Cet effet dépend de la technologie employée.

```
Word Size = 22
Number of Instructions = 2882
Number of Used |Rev| Bit0  | Bit1  | % 0 | Gain
859            |83%| 22496 | 40908 | 35% | -28%
```

FIG. 2.21: Statistiques sur le premier microcodage du VLD, avec optimisation

réduire la proportion de bits 0 à **35%**, ce qui représente un gain de **28%** par rapport à l'encodage non optimisé.

Dans un deuxième temps, un nouvel encodage du jeu d'instructions est conçu de manière à augmenter volontairement le nombre de 0, de sorte qu'un maximum de bits soient inversés. Il s'agit de s'éloigner d'un encodage où il y aurait autant de 0 que de 1, ce qui ne favorise pas l'optimisation. Les statistiques pour ce deuxième encodage sont données en figure 2.22. Le

```
Word Size = 22
Number of Instructions = 2882
Number of Used |Rev| Bit0  | Bit1  | % 0 | Gain
859            | 0%| 46984 | 16420 | 74% | 0%
```

FIG. 2.22: Statistiques sur le second microcodage du VLD, sans optimisation

pourcentage de 0 atteint alors 74%, ce qui laisse présager une optimisation plus efficace en inversant les bits instructions concernées. En figure 2.23, il apparaît effectivement que 89% des instructions ont été inversées, contre 83% avec le premier encodage. Le pourcentage de 0 dans le code final descend alors à **24%**, ce qui représente un gain de 50% par rapport à l'encodage sans optimisation.

```
Word Size = 22
Number of Instructions = 2882
Number of Used |Rev| Bit0  | Bit1  | % 0 | Gain
859            |89%| 15836 | 47568 | 24% | -50%
```

FIG. 2.23: Statistiques sur le second microcodage du VLD, avec optimisation

Cette expérimentation a montré qu'une simple technique d'inversion de bits, associée à la conception d'un encodage adapté, permet de réduire considérablement le nombre de 0 dans le code binaire stocké en ROM (de 63% il a été réduit à 24%).

2.5 Conclusion

Ce chapitre a présenté le contexte industriel dans lequel ces travaux de thèse ont été effectués, ainsi que le développement de base qu'il a été nécessaire de mettre en oeuvre dans le domaine de la compilation reciblable, autour de la chaîne Flexcc.

Plusieurs processeurs embarqués de SGS-THOMSON ont servi de base aux expérimentations architecturales : les trois processeurs dédiés de l'IVT (MSQ, BSP, VIP), un processeur DSP dédié (le DAP) et un processeur DSP standard (D950). Leurs caractéristiques principales ont été exposées.

La chaîne de compilation reciblable Flexcc qui a été utilisée pour produire les compilateurs C des processeurs ci-dessus, ainsi que pour mener à bien les explorations architecturales, a été présentée. Enfin, les développements additionnels autour la chaîne Flexcc afin de prendre en compte les contraintes liées au flot de conception logiciel ont été présentés.

Le développement des compilateurs pour les trois processeurs MSQ, BSP et VIP a permi de mettre en place un flot de conception-validation logiciel complet, tel qu'il a été appliqué dans le développement du système IVT relaté dans le chapitre 3 sur la co-simulation C-VHDL. Les processeurs D950 et DAP ont fait l'objet de l'exploration de l'encodage des champs immédiats présentée dans le chapitre 5, alors que l'évolution 24 bits du D950, appelée D960, a servi de base architecturale à l'exploration automatisée décrite dans le chapitre 4.

Chapitre 3

La co-simulation C-VHDL

Ce chapitre décrit la spécification, la conception et la réalisation de l'outil de co-simulation C-VHDL CoSim, ainsi que les résultats obtenus lors des expérimentations menées à SGS-THOMSON en vue de son industrialisation. Le rôle principal de la cosimulation C-VHDL est de permettre la validation fonctionnelle d'un logiciel embarqué dans son environnement matériel réel, avant réalisation.

Les spécifications fonctionnelles ont été établies en collaboration avec les concepteurs du circuit IVT du département R&D. La réalisation du modèle de co-simulation C-VHDL couvre plusieurs aspects : la communication inter-processus (couche de communication), l'intégration de cette communication dans le simulateur VHDL, la synchronisation des deux parties communicantes, et enfin l'échange de données de types différents entre le logiciel C et le simulateur VHDL.

L'expérimentation de l'outil a été menée sur deux projets : un modèle simplifié de circuit multi-processeurs (Calc) servant de tutorial et de référence de test, et le modèle réel du circuit IVT validant l'outil par rapport à un flot existant et un environnement complexe. Ces expériences ont abouti à l'industrialisation de l'outil dans sa version finale, et à son intégration dans la chaîne de conception interne Unicad de SGS-THOMSON.

Ce chapitre est composé de sept sections. La première section expose les motivations soutenant ce travail sur la cosimulation C-VHDL. La deuxième section retrace l'état de l'art de la cosimulation, aussi bien sur les principes et méthodes que sur les outils existants. Ensuite deux sections décrivent les spécifications, principes et réalisation de l'outil CoSim. Une cinquième section décrit l'intégration de la cosimulation avec CoSim dans le flot de conception d'un système. Ensuite l'application industrielle de l'outil au circuit de test Calc et au circuit mono-puce IVT est exposée. Enfin, des orientations possibles pour une future version de l'outil sont données.

3.1 Motivations

La motivation principale qui nous pousse à nous intéresser à la validation du logiciel embarqué dans un système mono-puce concerne les exigences de développement rapide de tels systèmes. Le temps de mise sur le marché, toujours plus court, associé à la complexité croissante des algorithmes demandent un effort très coûteux en temps de mise-au-point. De plus, les conditions particulières de développement de systèmes embarqués (processeurs spécifiques, logiciel embarqué) rendent la validation logicielle particulièrement cruciale. Le flot de conception classique de circuits intégrés, même s'il dispose d'une validation logicielle efficace, semble mal adapté à la taille de ces nouveaux systèmes, notamment en ce qui concerne la validation fonctionnelle au niveau système [101][16].

3.1.1 Validation du logiciel embarqué

La conception d'un système intégré contenant un (ou plusieurs) processeur spécifique requiert l'utilisation d'un flot de conception conjointe matériel-logiciel [73][90]. En effet, la particularité de la conception d'un système mono-puce est que le logiciel embarqué doit être développé et validé avant que le matériel ne soit réalisé : le logiciel étant stocké en mémoire interne, elle-même intégrée au circuit, il n'est pas possible de modifier le logiciel une fois que le circuit est fondu[1]. La validation du logiciel embarqué intervient donc avant ou pendant le développement du processeur sur lequel il devra tourner.

Le fait que le processeur réel ne soit pas disponible pour le développement du logiciel rend celui-ci plus difficile, puisqu'on ne peut ni explorer plusieurs solutions logicielles, ni tester la solution finale en conditions réelles (conjointement avec le reste du système). Ces conditions de développement, très particulières, contribuent à rendre la validation du logiciel embarqué cruciale pour la conception d'un système complexe en un temps réduit [66]. Nous nous intéresserons donc à améliorer les conditions de validation du logiciel embarqué, notamment en anticipant la validation fonctionnelle en environnement réel [94][62].

Plus précisément, la validation d'un logiciel embarqué peut généralement être décomposée en trois phases :

1. la validation interne, où l'on s'assure de l'exécution correcte d'une grande partie de l'algorithme indépendamment du reste du système,
2. la validation de la communication avec l'extérieur, où l'on s'assure que les informations échangées avec le reste du système sont correctement transmises et interprétées,
3. la validation en environnement réel, où l'on vérifie que le logiciel réagit correctement lorsque le système complet fonctionne.

Pour les deux premières phases, il est nécessaire de disposer d'un ensemble de stimuli permettant de simuler, au moins en partie, le fonctionnement du reste du système. Il s'agit ici de mimer les réactions des autres composants dans certains cas précis, sans entrer dans le détail du comportement de chaque composant. L'écriture de ces stimuli peut demander un temps de développement non négligeable, surtout pour la phase 2.

Clairement la troisième étape de la validation, en environnement réel, nécessite impérativement de simuler le système complet, incluant le logiciel embarqué. Nous nous intéressons tout particulièrement à cette dernière étape, car la méthode classique impose ici des contraintes assez pénalisantes en termes de temps de développement.

3.1.2 Validation tardive avec le flot de conception classique

Le flot de conception classique de circuit intégrés propose une étape de validation complète du logiciel embarqué, apparaissant relativement tard dans le temps. Le processeur spécifique doit notamment être développé et validé. Dans ce contexte, l'exploration (de l'architecture ou/et du logiciel) est rendue difficile et coûteuse en temps.

Description

La figure 3.1 représente le flot de conception classique (actuel) d'un système contenant un processeur et son logiciel embarqué.

[1] La mémoire stockant le programme est le plus souvent une ROM, non-réinscriptible. L'équivalent en RAM occuperait une surface bien plus grande, et donc augmenterait considérablement le coût du circuit. Des technologies telles que l'EEPROM ou la FLASH peuvent cependant se subsituter à la ROM, ce qui autorise la modification du programme après la cuisson. Mais le coût en surface est encore assez élevé.

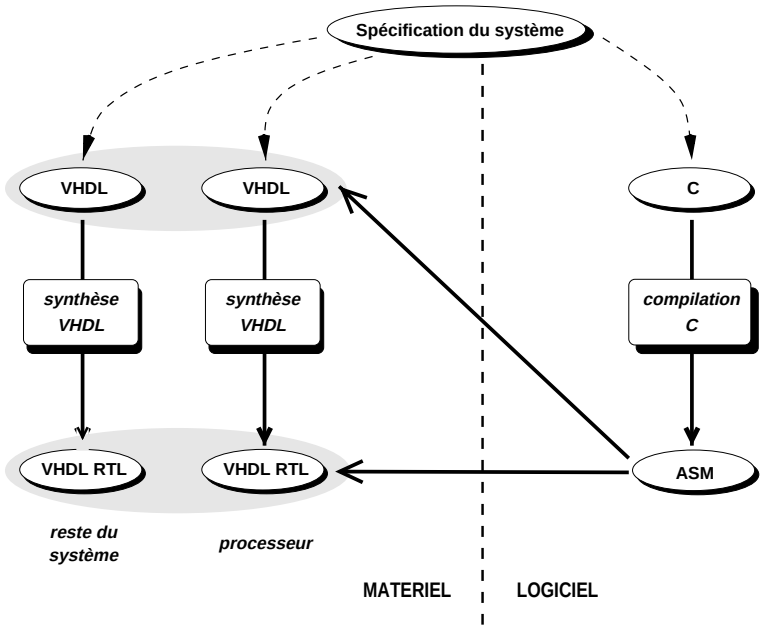

FIG. 3.1: Flot de conception classique

A partir des spécifications complètes du système, les descriptions de haut-niveau des parties matérielle et logicielle sont produites (généralement manuellement). Pour la partie matérielle, un langage tel que VHDL ou VERILOG est le plus souvent utilisé. Nous nous restreignons ici au langage VHDL, couramment utilisé en Europe. La partie logiciel utilise un flot de compilation C, relativement classique.[2]

La partie matérielle (à gauche) est constituée d'une part, du processeur embarqué - pour simplifier nous ne considérons qu'un seul processeur -, et d'autre part du reste du système, composé de blocs matériels divers. Nous représentons ici un flot de synthèse comportementale du matériel, générant des modèles VHDL-RTL à partir de modèles comportementaux VHDL. Cette étape de synthèse peut être remplacée par l'écriture manuelle des modèles VHDL-RTL.

Le logiciel écrit en C (à droite) est compilé en code assembleur (ASM), par un compilateur développé spécialement pour ce processeur. Ce code assembleur, sous sa forme binaire, est ensuite chargé dans la mémoire programme du modèle VHDL du processeur (comportemental ou RTL), lequel est simulé conjointement avec le reste du système par un unique simulateur VHDL. Cette simulation est appelée simulation au niveau assembleur. Grâce à l'exactitude temporelle des simulations VHDL, la validation du logiciel embarqué atteint un niveau de précision au cycle près [105]. Le prix à payer est une vitesse de simulation relativement faible, puisqu'elle ne dépasse pas quelques instructions par seconde pour un seul processeur[74].

Localisation de l'interface matériel-logiciel

L'interface entre le logiciel et le matériel se situe entre le logiciel applicatif et le processeur, et plus précisément entre les instructions en code machine et le décodeur d'instructions (Figure 3.2).

[2]Les exigences de temps de développement réduit (*time-to-market*) et de réutilisation du code rendent indispensable l'utilisation d'un langage de haut-niveau, au lieu de langages assembleur dépendant du jeu d'instructions. Par ailleurs, les normes et recommendations concernant les algorithmes, les formats de données et les protocoles, sont de plus en plus souvent spécifiées en C.

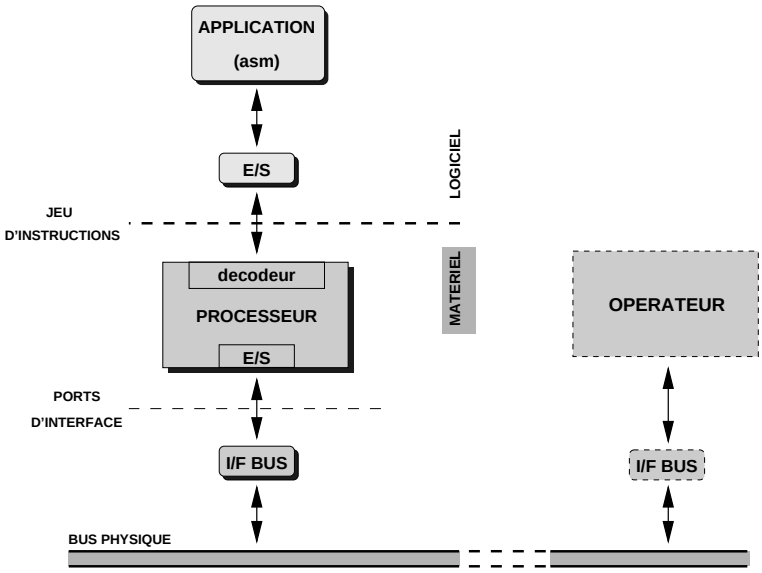

FIG. 3.2: Interface logiciel-matériel en simulation VHDL

Le code assembleur (ou code machine) de l'application, comprenant également les entrées/sorties avec le reste du système, est directement traité par le décodeur du processeur. Les opérations d'entrée/sortie destinées au reste du système sont transmises à l'interface bus, via le bloc d'entrée/sortie du processeur. L'interface bus gère la transmission des données et signaux de contrôle sur le bus physique, partagé avec les autres opérateurs du système.

L'interface matériel-logiciel est donc entièrement définie par le jeu d'instructions du processeur.

Principal inconvénient : validation tardive

Cette approche de la validation présente un inconvénient majeur : La simulation du logiciel avec le matériel intervient tard dans le flot de développement. La figure 3.3 représente les étapes nécessaires pour aboutir à la validation du logiciel embarqué. Le logiciel doit

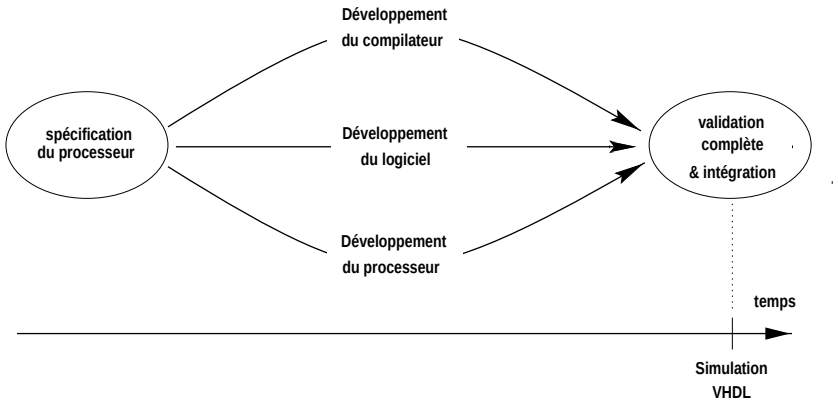

FIG. 3.3: Étapes pour la validation du logiciel dans un flot classique

évidemment être développé (en C), de même que le compilateur complet (comprenant au minimum le générateur de code, l'assembleur et l'éditeur de liens) doit être développé et

validé. De plus, le processeur lui-même doit être disponible, et donc ses spécifications figées (interface et jeu d'instructions).

Il est observé que plus la validation intervient tôt dans la conception d'un système, plus une erreur est corrigée rapidement (voir Figure 3.4) [20]. Dans ce contexte, la moin-

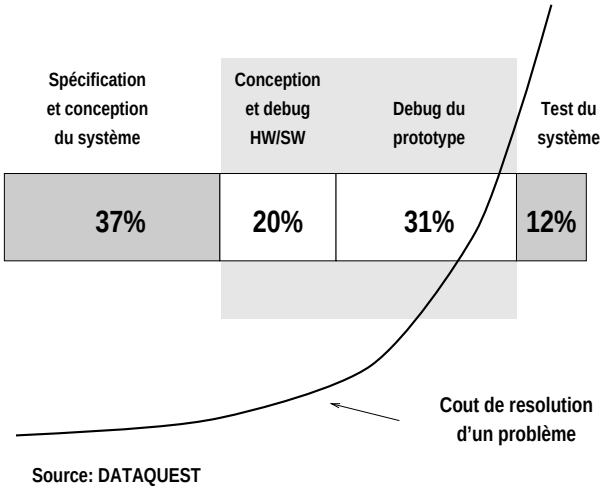

FIG. 3.4: Temps de conception et coût de déboguage

dre modification des spécifications (processeur, interface, jeu d'instructions) implique de re-générer le modèle du processeur ainsi que le compilateur. Le temps de cycle est alors trop long pour espérer explorer plusieurs solutions, et très pénalisant pour appliquer les optimisations nécessaires [115][16].

3.1.3 La co-simulation assembleur-VHDL au niveau instructions

L'inconvénient principal du flot de conception classique, précédemment décrit, est une validation tardive du logiciel embarqué, essentiellement dûe à la nécessité de disposer du modèle matériel réel du processeur embarqué. L'objectif est donc de s'affranchir de l'indisponibilité de ce composant lors de la validation logicielle. Le logiciel serait alors exécuté conjointement avec le matériel, par l'intermédiaire d'un outil de liaison ne faisant pas partie du système final. Par opposition à l'approche classique de simulation VHDL du système complet comprenant le processeur, le terme de co-simulation, sous-entendu matériel-logiciel, désigne cette simulation conjointe. Une définition de la co-simulation peut être trouvée dans [92]. Deux approches peuvent être envisagées, à différents niveaux :

- La première approche consiste à substituer au modèle réel du processeur un simulateur de jeu d'instructions. Cette co-simulation, appelée co-simulation assembleur-VHDL, se situe pratiquement au même niveau logique que la simulation classique, à savoir au niveau VHDL (pour le matériel) et jeu d'instructions (pour le logiciel).
- La seconde approche se situe à un plus haut niveau, puisque le logiciel est exclusivement décrit en langage C. Il est compilé puis exécuté sur la station de travail. La communication avec la partie matérielle est assurée par un noyau d'inter-connexion.

Ce paragraphe décrit dans le détail la première de ces deux techniques alors que la seconde (la co-simulation C-VHDL) est décrite dans le paragraphe suivant (3.1.4).

La méthode de co-simulation la plus couramment utilisée, employée par la plupart des outils de co-simulation et de conception conjointe commerciaux, consiste à substituer au modèle réel du processeur un modèle de simulation pouvant exécuter l'ensemble du

jeu d'instructions du processeur. Cela s'appelle un simulateur de jeu d'instructions, ou *Instruction-Set Simulator* (*ISS*) [82]. La précision de simulation obtenue est très proche de la simulation réelle du processeur, notamment au niveau temporel (*cycle-true simulation*). La vitesse de simulation est également considérablement augmentée. Pourtant, cette co-simulation nécessite la production du code machine du logiciel embarqué, et donc le développement antérieur de la chaîne de compilation. Ce délai doit pouvoir être éliminé si nous ne considérons que la phase de validation fonctionnelle du logiciel.

Flot de conception

La figure 3.5 décrit cette méthode. Nous retrouvons un flot de conception similaire à

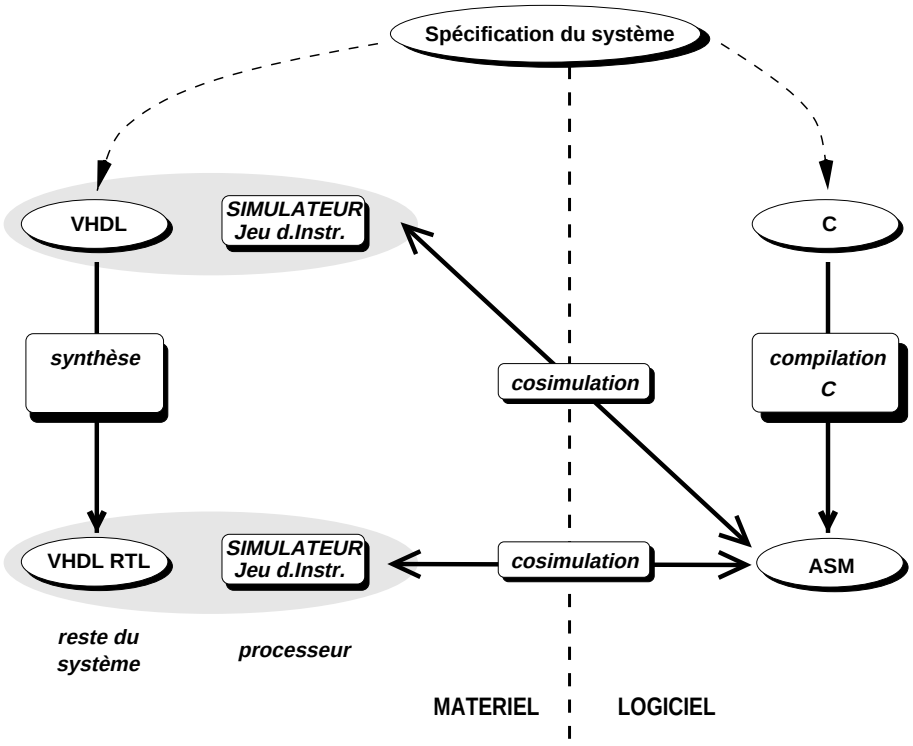

FIG. 3.5: Flot de conception avec simulation du jeu d'instructions

la méthode classique de simulation matérielle, à ceci près que le modèle matériel réel du processeur (VHDL ou VHDL-RTL) est remplacé par un modèle générique de simulation, assurant un comportement identique au niveau instructions. Nous conservons cependant une précision de simulation comparable, pour peu que le modèle de simulation soit décrit avec suffisamment de précision.

Le modèle de simulation peut être écrit en VHDL, ou bien en C. Dans le premier cas, l'intégration au reste du système, décrit lui aussi en VHDL, est immédiate. Dans le second cas, une interface C-VHDL bas-niveau doit être développée pour relier le simulateur de jeu d'instructions au simulateur VHDL contenant le reste du système.

Les étapes nécessaires à la validation du processeur embarqué selon cette méthode sont représentée en figure 3.6. Cette figure est à comparer avec la figure 3.3 du flot classique. La validation complète du logiciel peut être accomplie dès le développement du logiciel embarqué et du compilateur, sans que le modèle réel du processeur ne soit nécessaire. Il sera utilisé par la suite lors de la phase de son intégration dans le système.

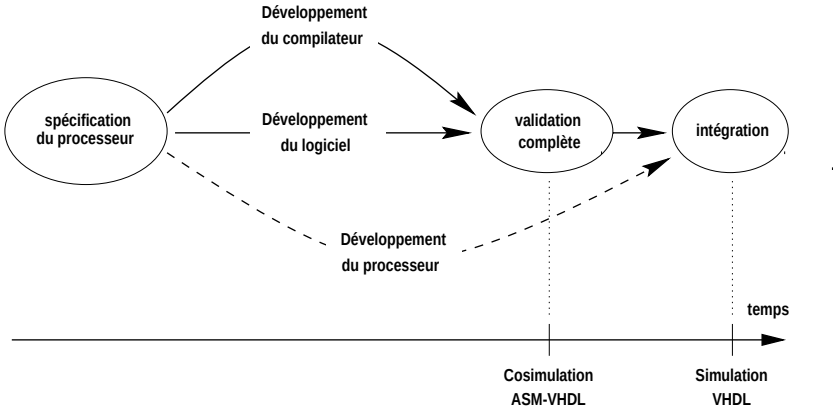

FIG. 3.6: Étapes vers la validation du logiciel avec la co-simulation ASM-VHDL

Localisation de l'interface matériel-logiciel

L'interface matériel-logiciel est à nouveau localisée entre le code applicatif en assembleur et le décodeur du processeur, implémenté dans le simulateur (Figure 3.7). Le jeu d'instruc-

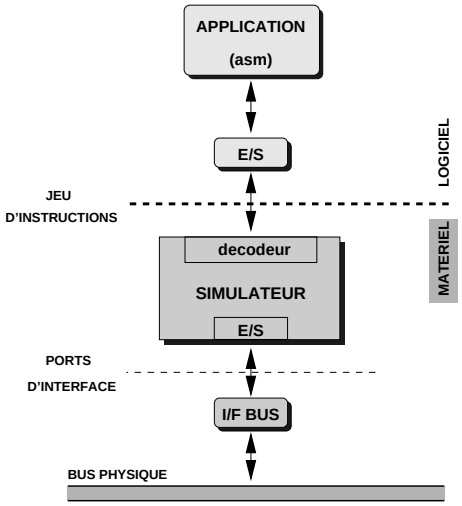

FIG. 3.7: Interface matériel-logiciel en co-simulation assembleur-VHDL

tions du processeur définissant strictement cette interface, le comportement du simulateur, vu de l'extérieur (de l'application ou du reste du système), est parfaitement similaire au modèle réel du processeur pour peu que le modèle de simulation soit suffisamment précis.

Avantages et inconvénients

L'avantage essentiel de cette approche est qu'elle ne nécessite pas de concevoir ni de réaliser le modèle matériel réel du processeur, réduisant ainsi considérablement le temps de développement du système complet, et la date de disponibilité du logiciel embarqué. Cela est vrai à condition que le temps de développement du modèle de simulation soit nettement plus court que celui du modèle réel du processeur. Pour ce faire, des techniques de réutilisation de blocs génériques et de génération automatique peuvent être employées. Les simulateurs de jeu d'instructions reciblables répondent à ces critères.

De plus, la précision temporelle obtenue par la simulation VHDL peut facilement être atteinte avec cette approche pour peu que les modèles de simulation, de communication avec

l'extérieur, et d'horlogerie interne soient suffisamment évolués. Une telle simulation peut être réaliste au cycle près (*cycle-true simulation*). Cela permet de valider complètement le logiciel, y compris au niveau temporel.

La vitesse d'exécution d'un simulateur de jeu d'instructions est nettement plus élevée que celle de la simulation VHDL du processeur réel. Des mesures ont montré qu'un simulateur de jeu d'instructions en VHDL peut atteindre plusieurs milliers d'instructions par seconde (1-10 Kips), alors qu'un simulateur de jeu d'instructions en C peut lui atteindre une centaine de milliers d'instructions par seconde (100 Kips)[85][86][118].

La co-simulation assembleur-VHDL est une étape utile et nécessaire. Elle présente cependant quelques inconvénients puisque le flot de conception reste similaire à celui de l'approche classique, notamment en ce qui concerne la partie logicielle. L'inconvénient majeur dans cette approche demeure la nécessité de disposer du code binaire (ou assembleur) de l'application. Que l'application soit écrite en assembleur ou en C, le programme d'assemblage et/ou le compilateur doivent être disponibles. Cela comprend également la partie appelée dorsal (*back-end*), composée de l'éditeur de liens et du chargeur en mémoire programme. L'expérience a montré que le temps passé à concevoir ces derniers outils est loin d'être négligeable comparé au temps de développement du générateur de code proprement dit.

Quelques autres inconvénients mineurs peuvent également être identifiés :

- Il faut que les spécifications complètes du processeur (architecture, jeu d'instructions) soient disponibles et figées pour développer la chaîne de compilation.
- Plusieurs validations sont menées de front : modèle de simulation, compilateur, encodage des instructions et réalisation du logiciel. L'identification d'un éventuel bogue en est rendue plus délicate.
- Enfin, le logiciel est exécuté et débogué sous la forme de code machine. Le suivi de l'exécution et la mise au point de celui-ci n'est pas aisée pour le concepteur (notamment pour le suivi du flot de contrôle et des appels de fonctions). D'autant plus que le code machine ainsi débogué est généré par le compilateur.

En résumé, la co-simulation assembleur-VHDL apporte un gain de temps de développement important en substituant au modèle VHDL du processeur un modèle de simulation de jeu d'instructions, plus rapide à obtenir pour la validation logicielle. De plus la vitesse de simulation est considérablement augmentée. Par contre, la simulation du système complet se fait toujours au niveau assembleur, impliquant les inconvénients cités plus haut. Afin d'identifier le moyen de s'affranchir de ces inconvénients, il est nécessaire d'étudier plus précisément la tâche de validation du logiciel embarqué.

Validation fonctionnelle et validation temporelle

Analysons l'adaptation de la co-simulation assembleur-VHDL à chacune des trois phases de la validation du logiciel décrites plus haut (paragraphe 3.1.1) :

1. La validation interne, où la plus grande partie du code logiciel est exécutée, est pénalisée par la faible vitesse de simulation. Il est matériellement impossible de couvrir la totalité du code.
2. Par contre, la validation de la communication avec l'extérieur requiert une précision, temporelle notamment, que seule la simulation VHDL peut assurer.
3. Enfin la validation en environnement réel, avec le système complet, demande des temps de simulation trop élevés. La simulation VHDL est à nouveau inadaptée.

Afin de s'affranchir de ces limitations, il faut d'abord distinguer deux types de validations : la validation fonctionnelle et la validation temporelle. La première vise simplement à valider le comportement du logiciel, communicant éventuellement avec le reste du système. La seconde s'attache à vérifier que les différents signaux de contrôle et données sont traités dans un intervalle de temps compatible avec les exigences de performances, et qu'ils sont disponibles à la date prévue.

Partant de cette distinction, il apparaît que la phase 1 (validation interne) relève en grande partie de la validation fonctionnelle. Au contraire la phase 2 exige essentiellement un réalisme temporel, même si la fonctionnalité est encore importante. Enfin, la phase 3 exige aussi bien une validation fonctionnelle que temporelle. Il semble donc qu'une partie de la validation du logiciel relève de la validation fonctionnelle, ne nécessitant pas de précision temporelle au cycle près. De plus, la simulation au niveau assembleur rend difficile, voire impossible, la validation du logiciel dans son ensemble (phase 1) et dans son environnement réel (phase 3).

Nous pouvons alors spécifier une alternative efficace à la simulation au niveau assembleur : à condition de négliger la dimension temporelle, une partie importante de la validation du logiciel embarqué peut s'effectuer à un niveau d'abstraction supérieur, sans requérir le code machine. La précision temporelle est dans ce cas inutile. Au contraire, elle induit une vitesse de simulation très faible (pas plus de quelques instructions par seconde) incompatible avec une simulation exhaustive d'un algorithme un tant soit peu complexe.

Nous en déduisons que la co-simulation assembleur-VHDL, même si elle est nécessaire pour atteindre la précision temporelle indispensable à la validation finale, n'est pas adaptée à une partie importante de la validation du logiciel embarqué, la validation fonctionnelle. Élever le niveau d'abstraction du logiciel devrait autoriser une validation fonctionnelle efficace, appliquée plus tôt dans le flot de conception du système.

3.1.4 Apport de la co-simulation C-VHDL au niveau fonctionnel

La co-simulation fonctionnelle C-VHDL [76][69] fait intervenir le code C haut-niveau du logiciel embarqué, à la place du code machine ou assembleur. Le code C est exécuté en parallèle avec la simulation VHDL du reste du système (ne comprenant pas le processeur sur lequel tournera le logiciel). Elle ne requiert aucune description du processeur, mise-à-part la définition des signaux d'interface entre le processeur et le reste du système. La communication entre le logiciel en C et le reste du système en VHDL est assurée par un module spécifique, généré manuellement ou automatiquement.

Flot de conception

La figure 50 représente le flot de conception et de validation utilisant la co-simulation C-VHDL. Pour simplifier le dessin, les étapes de validation classiques (avec simulateur de jeu d'instructions ou avec le modèle réel du processeur) ne sont pas représentées. Elles viennent en complément de la co-simulation C-VHDL. A partir des spécifications du système, les parties matérielle (en VHDL) et logicielle (en C) sont produites. Dès ce niveau, le code C du logiciel est exécuté et communique directement avec la simulation VHDL du reste du système.

Le modèle réel du processeur sur lequel tournera ce logiciel n'est pas nécessaire. Seule une description, brève, de son interface doit être fournie pour assurer la connexion entre le logiciel et le reste du système. Cette co-simulation C-VHDL peut aussi bien être effectuée au niveau VHDL comportemental qu'au niveau VHDL-RTL, après synthèse.

La figure 3.9 décrit plus précisément l'enchaînement des étapes aboutissant à la validation complète du logiciel et à l'intégration du processeur dans le système. Elle doit

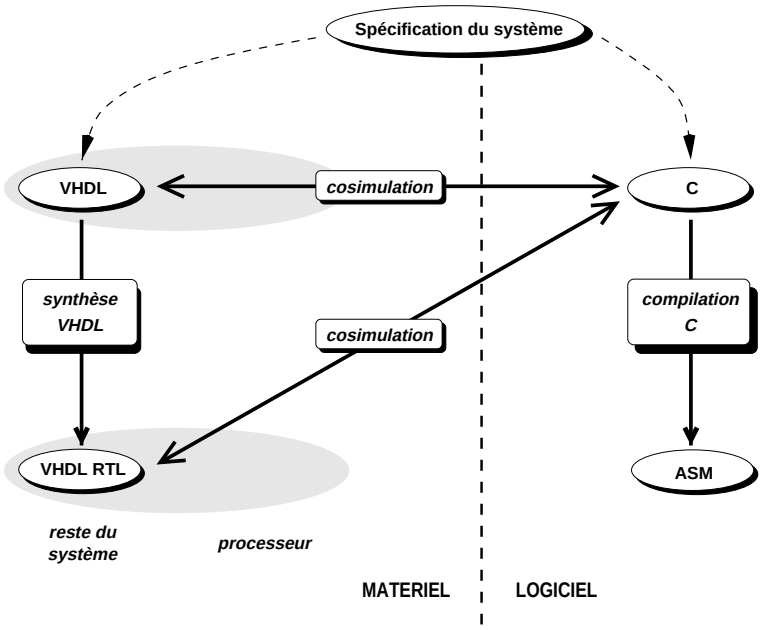

FIG. 3.8: Flot de conception avec co-simulation C-VHDL

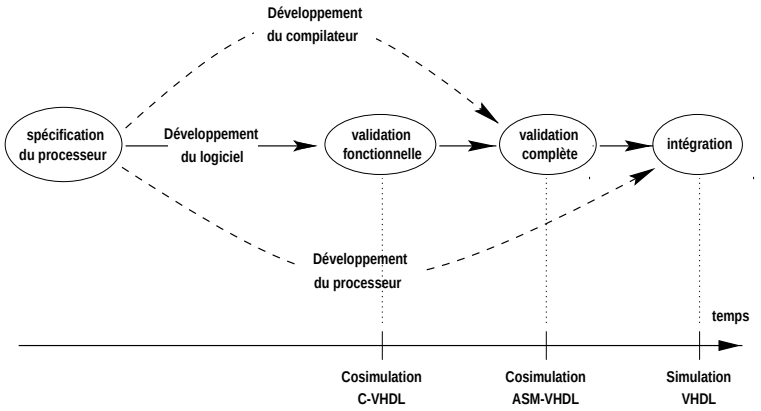

FIG. 3.9: Étapes vers la validation du logiciel avec la co-simulation C-VHDL

être comparée aux figures 3.3 et 3.6 des flots classique et assembleur-VHDL. La validation fonctionnelle, excluant toute caractérisation temporelle, est ici accomplie dès le développement du logiciel embarqué, sans requérir ni le compilateur ni un modèle du processeur. Ces derniers ne sont nécessaires que pour la validation temporelle et l'intégration finale. La date de la validation fonctionnelle est donc à nouveau considérablement avancée.

Localisation de l'interface matériel-logiciel

L'interface matériel-logiciel se situe directement entre l'application en langage C et l'interface bus, en VHDL (voir Figure 3.10). Plus exactement, seules les opérations d'en-

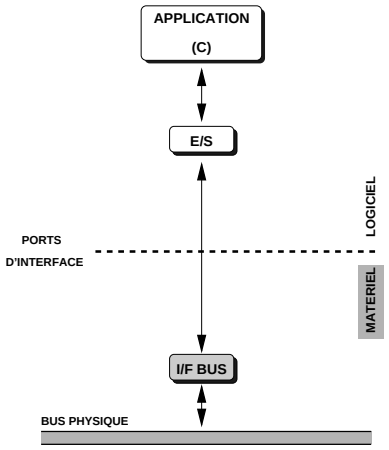

FIG. 3.10: Interface matériel-logiciel en co-simulation C-VHDL

trée/sortie de l'application dialoguent avec l'interface bus. L'interface n'est donc plus assurée par le jeu d'instructions du processeur, mais par les ports d'interface du processeur avec le reste du système.

La nature des informations échangées entre les deux parties dépend de la fonctionnalité attribuée à l'interface bus. Dans le cas le plus simple, l'interface bus se contente d'échanger des signaux de contrôle et des données avec le bus physique. Les informations échangées avec l'application C (via son bloc d'entrée/sortie) sont alors un ensemble de bits et d'octets. Si au contraire l'interface bus assure un rôle de temporisation, de stockage temporaire ou d'arbitrage d'accès au bus, les informations échangées avec l'application seront de nature plus protocolaire (plus haut-niveau) mettant en œuvre des octets de commandes, de données et d'état (*status*).

Avantages et inconvénients

Les avantages d'une telle approche sont multiples :

- Validation au niveau C : l'application est décrite en langage C, considérée comme la source. La lecture et le déboguage fonctionnel sont bien plus aisés qu'avec le code binaire. De plus, cela autorise l'utilisation d'outils de développement logiciel standards, disponibles sur le marché (débogueurs, profileurs, ...).
- Avant l'écriture du modèle VHDL : il n'est plus nécessaire de concevoir ni de réaliser l'architecture du processeur, ni le jeu d'instructions. Au contraire, cette méthode permet de prendre en compte, après analyse, les statistiques de profilage obtenues au cours de la validation fonctionnelle dans la réalisation du processeur.

- Avant l'écriture du compilateur : le compilateur et/ou l'assembleur ne sont pas requis pour la validation fonctionnelle. Leur développement peut se faire en parallèle (au fur et à mesure de la conception de l'architecture) avec la validation de l'application.
- Validation de l'algorithme uniquement : seule la fonctionnalité de l'algorithme est validée, indépendamment des autres parties (processeur, micro-codage) qui seront validées lors des étapes suivantes.

Les inconvénients sont liés au fait que le niveau d'abstraction (langage C) est plus élevé que pour les simulations antérieures.

- Les contraintes temporelles ne sont absolument pas vérifiées en co-simulation C-VHDL. Par exemple, le système matériel peut obtenir de la part du logiciel un résultat en un cycle alors qu'en réalité le calcul peut prendre plusieurs milliers de cycles avec le modèle VHDL du processeur.
- La communication de données entre deux mondes différents (VHDL et C) peut engendrer des problèmes de cohérence (types de données hétérogènes, absence de parallélisme en C). Ces problèmes doivent être résolus par l'adjonction de fonctions spécifiques.

En résumé, la co-simulation fonctionnelle C-VHDL autorise une validation fonctionnelle à un niveau d'abstraction supérieur (le langage C), et ceci très tôt dans le flot de conception du système, avant même d'avoir conçu et développé un modèle matériel du processeur embarqué. Au prix d'une précision de simulation moindre (comparée à la simulation de jeu d'instructions), le temps gagné et la facilité de développement du logiciel au niveau C font de la co-simulation C-VHDL un outil de validation fonctionnelle efficace pour la conception de systèmes complexes, et en un temps compatible avec les exigences du marché.

3.1.5 Principe de la co-simulation C-VHDL

Les concepts de base de la co-simulation C-VHDL sont ici exposés. Ils comprennent l'exécution de programmes C lors d'une simulation VHDL, l'exécution concurrente de plusieurs programmes communicants (C et VHDL), et enfin la communication inter-processus.

Simulateurs VHDL et programmes C

La plupart des simulateurs VHDL commerciaux proposent en standard un moyen d'exécuter du code C pendant une simulation, et d'échanger bi-directionnellement des données entre le C et le VHDL. Trois simulateurs, parmis les plus répandus, ont été étudiés dans le cadre de ces travaux : VSS de Synopsys [103], Leapfrog de Cadence [18] et Voyager de Ikos. Bien que le principe reste identique quel que soit le simulateur, le paquetage servant à exécuter une fonction C pendant la simulation est appelé différemment selon le vendeur : CLI (*C Language Interface*) pour VSS Synopsys, FMI (*Foreign Model Interface*) pour Leapfrog Cadence et FKI (*Foreign Kernel Interface*) pour Ikos.

Le principe de fonctionnement de l'appel d'une fonction C pendant la simulation VHDL est le suivant : le code C est écrit sous forme d'une unique fonction, qui sera exécutée chaque fois que un ou plusieurs signaux donnés du système changeront d'état (selon le même modèle de sensibilité que les process en VHDL). De plus, une librairie permet à la fonction C d'accéder aux structures internes du modèle VHDL contenant les valeurs de tous les signaux et ports du circuit, en lecture et en écriture [19].

La figure 3.11 décrit plus précisément le mécanisme d'appel d'une fonction C pendant la simulation VHDL. La première ligne indique les changements d'état du signal d'horloge. Nous choisissons ici, arbitrairement, de conditionner l'appel de la fonction C par un front

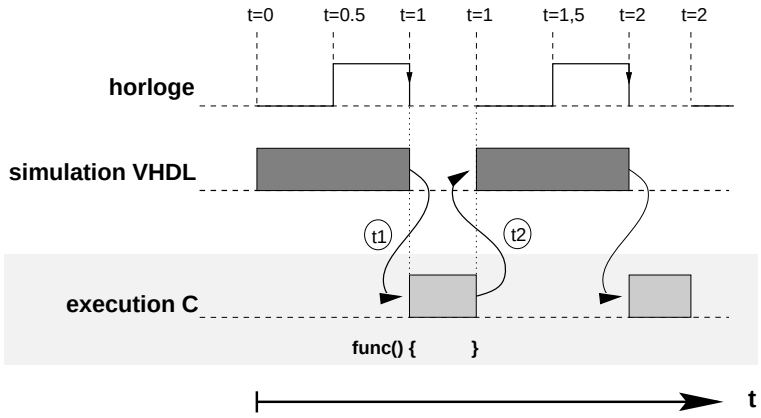

FIG. 3.11: Appel d'une fonction C pendant la simulation VHDL

descendant de l'horloge. La deuxième ligne représente l'activité de la simulation VHDL dans le temps : les parties grisées symbolisent l'avancement de la simulation alors que les intervalles blancs (pointillés) symbolisent un état de blocage de la simulation. Pendant le laps de temps où la simulation est bloquée, la fonction C est exécutée dans sa totalité. La dernière ligne représente l'activité de la fonction C (appelée ici `func()`), qui est exactement complémentaire de l'activité de la simulation VHDL. Les flèches symbolisent les transitions entre la simulation VHDL et l'exécution C.

La transition VHDL-C, t1, intervient immédiatement après le front descendant de l'horloge (selon la sensibilité choisie au départ). La transition C-VHDL, t2, intervient lorsque la fonction C se termine. La simulation VHDL reprend alors, à l'instant exact où elle s'était interrompue (juste après le front descendant de l'horloge). La sensibilité du ou des signaux déclenchant l'appel de la fonction C est définie par l'utilisateur dans le code VHDL, donc statiquement avant la simulation.

Malheureusement, cette méthode impose une restriction importante sur l'architecture du code C lui-même. En effet, il doit être écrit sous la forme d'une unique fonction, elle-même pouvant éventuellement en appeler d'autres. Le point d'entrée (le début de la fonction) doit toujours être le même durant toute la simulation.

Suivant le type de l'application, cette restriction est ou n'est pas acceptable. Il faut distinguer deux types d'applications :

- les applications orientées données,
- les applications orientées contrôle.

Nous disons qu'une application est orientée données lorsqu'elle peut être écrite sous la forme d'une unique fonction, prenant une ou plusieurs données en entrée, et produisant une donnée en sortie. Il s'agit d'un traitement, ou production, atomique de données. Au contraire, une application orientée contrôle se caractérise par la présence d'interactions multiples avec l'environnement matériel. La figure 3.12 décrit ces deux types d'applications.

Dans une application orientée données (figure 3.12a), la même fonction `func()` est appelée à chaque interruption de la simulation VHDL (in1, in2, ...), et est exécutée dans sa totalité. Les seules interactions avec l'extérieur se produisent au début et à la fin de la fonction, par l'échange de ces données. Des applications typiques sont les fonctions de traitement du signal, comme les filtres, les transformées,... Pour ce type d'applications, le modèle standard de fonction C appelée par le simulateur VHDL est tout à fait adapté.

Dans le cas d'une application orientée contrôle (3.12b), nous ne pouvons pas identifier une unique fonction qui sera appelée à chaque interruption du simulateur. A la limite, la

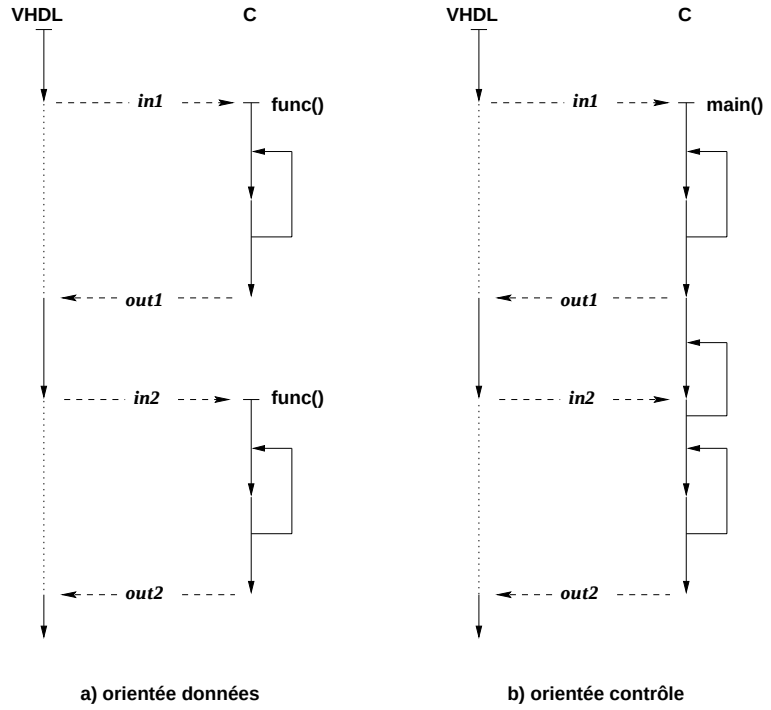

FIG. 3.12: Les deux types d'applications.

fonction principale (`main()`) peut être considérée comme le point d'entrée (in1). Mais dès la deuxième interruption (in2), le modèle ne fonctionne plus : il ne peut que rappeler la fonction `main()` depuis le début, alors que l'on souhaiterait qu'il reprenne l'exécution du programme plus loin (après le retour out1). De plus il apparaît un parallélisme d'exécution du simulateur VHDL et de l'application entre deux interactions (entre out1 et in2).

Nous pourrions alors imaginer de transformer le code C de manière à obtenir une machine d'états finis, contenant autant de branches que de chemins possibles dans le programme. La figure 3.13 décrit cette transformation. Le programme original (3.13a) est découpé en blocs indépendants, chacun n'ayant d'interaction avec le VHDL qu'au début et à la fin (3.13b). Le point d'entrée est alors unique, c'est le début de la fonction `main()`. Cette stratégie, bien que fonctionnellement correcte, présente plusieurs inconvénients :

1. la transformation en machine d'états finis est fastidieuse, d'autant plus que ce n'est pas la manière naturelle d'écrire un code applicatif, surtout lorsqu'il est complexe ;
2. si pour éviter la transformation manuelle nous concevons un traducteur automatique, nous obtenons un code C généré automatiquement qu'il est difficile de déboguer pour l'utilisateur. Cela est d'autant plus vrai que l'architecture même du programme est complètement transformée ;
3. le fait d'opérer une transformation sur le code original pour la validation ne garantit pas que l'on valide exactement le code qui sera ensuite compilé et implanté.

Pour ces raisons le mécanisme d'appel d'une fonction C, fourni par la plupart des simulateurs VHDL commerciaux, est insuffisant pour co-simuler une application qui n'est pas exclusivement orientée données.

Exécution concurrente C-VHDL

Afin de lever la limitation sur le style d'écriture de l'application C, celle-ci doit être exécutée séparément du simulateur VHDL. Cela signifie que le simulateur VHDL et l'ap-

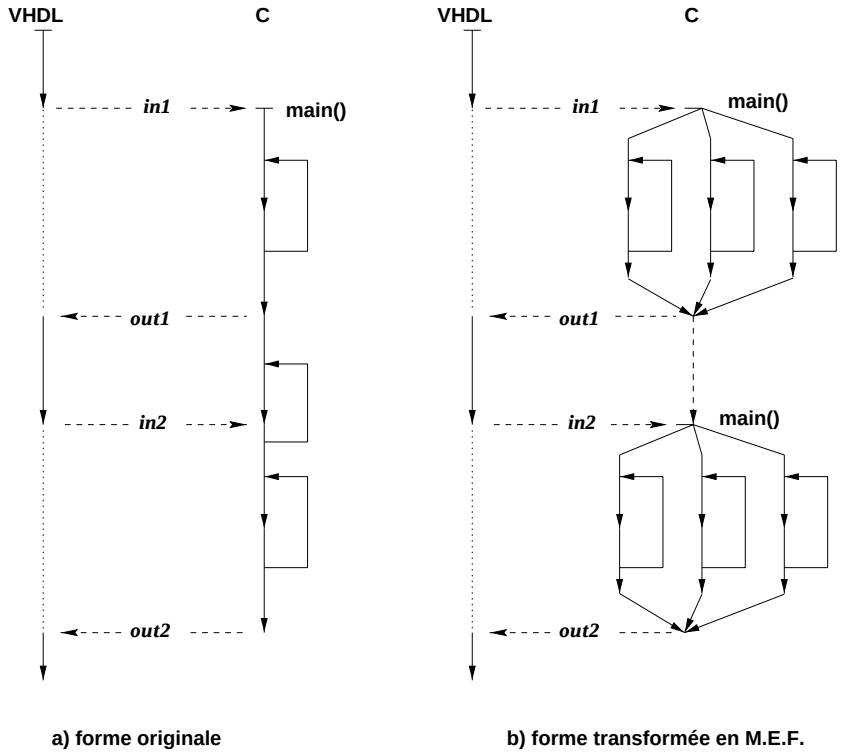

FIG. 3.13: Transformation en Machine d'États Finis

plication C s'exécutent en parallèle [6] sur la station de travail, générant deux processus différents (au sens Unix). Un modèle d'exécution similaire est décrit dans [37].

L'interface entre le logiciel et le modèle matériel du reste du système est constituée des ports d'entrée-sortie du processeur embarqué. Une communication synchronisée doit donc être établie entre les deux parties, afin d'échanger les valeurs des ports d'interface au bon moment. La figure 3.14 représente schématiquement le modèle d'exécution concurrente requis : la simulation VHDL et l'application C s'exécutent en parallèle, et communiquent régulièrement via une couche de communication.

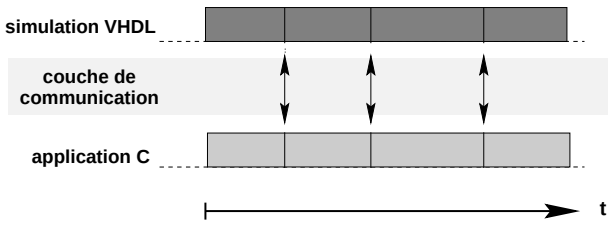

FIG. 3.14: Exécution concurrente du simulateur VHDL et de l'application C

Les simulateurs étudiés lors de ces travaux ne proposent pas de moyen de communication avec d'autres processus. Par contre, une interface permet d'exécuter une fonction C régulièrement pendant la simulation, comme cela a été décrit plus haut. Il est donc possible d'établir une communication inter-processus entre le simulateur VHDL et l'application C en utilisant une couche de communication écrite en C. La figure 3.15 décrit les couches nécessaires à la co-simulation C-VHDL. L'interface C du simulateur VHDL n'est alors utilisée que pour engager régulièrement une communication bi-directionnelle avec l'application C, via la couche de communication inter-processus.

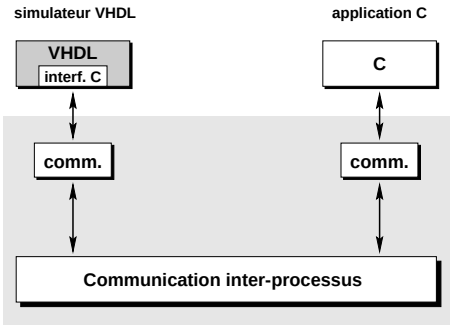

FIG. 3.15: Structure en couches de la communication VHDL-C

Communication inter-processus

La couche de communication servant à échanger les données entre le simulateur VHDL et l'application C doit être soigneusement choisie, en fonction des caractéristiques de la communication que l'on souhaite établir. Même si la plupart des mécanismes disponibles peuvent être adaptés aux exigences de la co-simulation C-VHDL, certains s'en rapprochent plus que d'autres. Cette section résume les caractéristiques des principaux mécanismes et explique le choix qui a été fait.

Le système d'exploitation le plus communément utilisé en conception de circuits est le système UNIX. Plusieurs mécanismes de communication inter-processus co-existent. Les plus courants sont Berkeley IPC, RPC et System V IPC [29].

Berkeley IPC (*sockets*) Les *sockets* de Berkeley fonctionnent sur un modèle client-serveur, au travers d'un réseau. Dans un premier temps le serveur ouvre un *socket*, associé à un nom. Un client peut alors faire une requête de connexion qui sera acceptée ou refusée par le serveur. Si la connexion est acceptée, un tube (*pipe*) bi-directionnel est ouvert entre le client et le serveur. Chacun peut alors écrire ou lire des données dans ce tube. Deux modes de connexion sont possibles : le mode flux (*stream*) et datagramme (*datagram*). Dans le premier cas, le transfert sans erreur et l'ordre des données sont garantis par le service. Au contraire, dans le mode datagramme des paquets indépendants sont émis. Ils peuvent éventuellement être perdus ou inversés.

Remote Procedure Call (RPC) RPC est basé sur un appel de procédures à distance, éventuellement au travers d'un réseau. Un client envoie une requête au serveur (*call*) qui répond en exécutant la routine de service correspondante (*accept*) [29]. La primitive *call* est composée de deux étapes de communication. La première étape consiste à envoyer au serveur le contenu de la requête. La seconde étape consiste à attendre la réponse du serveur. Le client est donc synchronisé avec le serveur par l'intermédiaire d'échanges bloquants. Le client et le serveur peuvent aussi bien tourner sur la même machine, que sur deux machines connectées à un réseau.

Ce protocole fonctionne donc sous le mode client-serveur, nécessitant la désignation d'un client (l'appelant) et d'un serveur (l'appelé) à priori.

Inter-Process Communication (IPC) Le service IPC (*Inter-Process Communication*) fournit deux types de communications pour le transfert de données : par files de messages (IPC/f) ou par mémoire partagée (IPC/m). Un troisième service, gérant les sémaphores, permet la synchronisation de processus indépendants sans échange de données, et ne concerne donc pas notre application. Les services IPC sont nécessairement restreints à une

seule machine.

Schématiquement, la connexion de deux processus communicant par le biais de files de messages est décrite par la figure 3.16. Les deux processus A et B (au sens processus

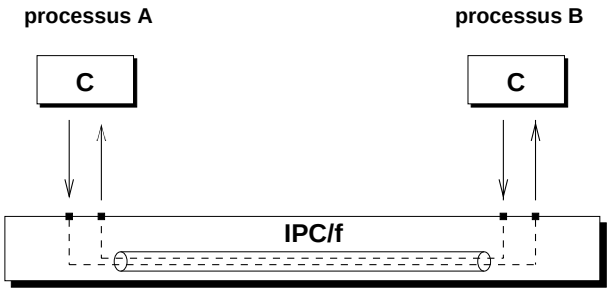

FIG. 3.16: Communication par files de messages IPC/f

Unix) exécutent deux programmes C différents, et communiquent via deux files (une pour chaque sens) à travers quatre ports d'accès. Les processus dialoguent en fait avec les ports d'accès à la file grâce à des primitives de lecture et d'écriture. Ces primitives peuvent être bloquantes pu non. Une primitive de lecture est bloquante lorsqu'elle atteint indéfiniment qu'un message arrive lorsqu'il n'y en a pas, alors qu'une primitive non bloquante retourne immédiatement un code d'erreur s'il n'y a pas de message. Symétriquement, une primitive d'écriture bloquante attend que la file se vide lorsqu'elle est pleine.

Le fonctionnement de la mémoire partagée est représenté figure 3.17. Le segment de

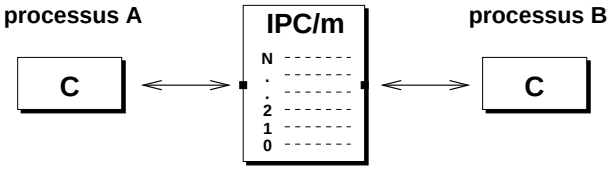

FIG. 3.17: Communication par mémoire partagée IPC/m

mémoire partagée est un ensemble de variables, de types quelconques, qui peuvent être accédées en lecture ou en écriture par l'un ou l'autre des processus connectés. Les droits en lecture ou écriture sur le segment de mémoire peuvent éventuellement être modifiés pour chacun des processus.

Motivation du choix d'IPC/f La co-simulation d'un logiciel avec un modèle matériel du reste du système ne présente pas à priori un comportement comparable au modèle client-serveur. En effet le logiciel peut aussi bien être le contrôleur du circuit (rôle de maître ou serveur) qu'un co-processeur dédié (rôle d'esclave ou client). Au contraire la co-simulation requiert une communication équilibrée : les échanges sont bi-directionnels et initiés par l'une ou l'autre des parties. Il n'est donc pas souhaitable d'introduire une hiérarchie à priori. Même si le modèle client-serveur, comme RPC, semble convenir pour certaines configurations (l'application à l'IVT relatée plus loin en est un exemple), il n'est pas suffisamment flexible pour s'adapter à tous les mécanismes de communication envisagés pour notre co-simulation. En effet, le modèle client-serveur implémente, par construction, une restriction du modèle par file de messages : les échanges sont obligatoirement bloquants (synchronisés). Or certaines applications de co-simulation présentent des caractéristiques non synchronisées, comme par exemple la communication par bus partagés.

Nous cherchons donc un modèle de communication pouvant fournir le plus large éventail

de mécanismes, suivant les besoins. C'est pourquoi le mécanisme IPC a été choisi pour réaliser la communication inter-processus dans CoSim.

Parmi les protocoles fournis par IPC, celui basé sur les files de messages a été retenu. La raison principale qui a motivé ce choix est que la mémoire partagée ne garantit pas la séquentialité des évènements, par exemple lorsqu'une nouvelle donnée efface la précédente. Cette dernière est d'ailleurs perdue. Au contraire une file de messages garantit l'ordre des messages par construction, sans nécessiter l'ajout d'un protocole supplémentaire. Par contre, l'application a des protocoles de communication évolués, se rapprochant par exemple du modèle client-serveur, peut nécessiter le développement de primitives supplémentaires de plus haut niveau, se substituant ainsi au mécanisme RPC.

3.2 État de l'art

De nombreux outils de co-simulation matériel-logiciel existent, aussi bien sur le marché que dans le monde universitaire [17][26][57][58][106][14][35][48]. Les environnements complets de conception conjointe disposent également de moyens de co-simulation. Dans cette étude, nous nous intéressons au domaine d'application de ces outils. Si la co-simulation assembleur-VHDL est proposée dans presque tous les environnements considérés, la co-simulation C-VHDL est plus rare.

Seamless de Mentor Graphics et EagleI de Eagle Design Automation sont deux outils de co-simulation matériel-logiciel au niveau assembleur.

Cossap de Synopsys et SPW de Alta Group of Cadence sont deux environnements de conception de systèmes à base de processeur de traitement du signal, et incluent la co-simulation au niveau assembleur.

Enfin, Coware et Synthesia proposent deux outils de co-simulation matériel-logiciel, aussi bien au niveau assembleur qu'au niveau C.

3.2.1 Seamless

Seamless de Mentor Graphics [45] fournit une interface de co-simulation pour connecter un simulateur de jeu d'instructions à un simulateur matériel. Les avantages et inconvénients de cette approche ont été discutés précédemment. Les principales caractéristiques de cet outil sont une optimisation des échanges entre les deux parties selon la densité de la communication, ainsi qu'une interface de programmation standard.

Optimisation des échanges

L'outil de Mentor Graphics autorise cependant une certaine flexibilité dans la localisation de l'interface matériel-logiciel utilisée pendant la co-simulation (indépendamment de l'interface matériel-logiciel du système réel). En effet, il a été observé qu'une très grande partie de la simulation (jusqu'à 90 %) d'un processeur embarqué concerne la partie opérative (CPU) et les échanges avec la mémoire (*read-write*). Le reste du temps est consacré aux échanges communiquant avec le reste du système. Il paraît donc souhaitable d'optimiser la simulation du CPU (par l'utilisation d'un simulateur de jeu d'instructions à la place d'un modèle VHDL), mais aussi la simulation des échanges-mémoire, et de la mémoire elle-même. L'outil Seamless permet de simuler la mémoire en logiciel, indépendamment de sa réalisation finale dans le système (nécessairement matérielle). Plusieurs niveaux d'optimisation sont en fait proposés, permettant au choix d'optimiser tous les échanges, les échanges-données et les échanges-instruction (*fetch*). La contrepartie de ces optimisations est une perte de précision dans la validation. Par exemple, l'optimisation des échanges-instruction ne permettra pas

de valider précisément les mécanismes de *fetch*. Cela devra être assuré séparément, avec une configuration adaptée.

Dans le meilleur des cas, Mentor Graphics assure pouvoir atteindre une vitesse de simulation de l'ordre de 100 000 instructions par seconde.

Interface de programmation standard

Une autre caractéristique intéressante de cet outil concerne la réalisation de la communication entre le simulateur matériel et le simulateur logiciel. Dans la configuration standard de l'outil, le simulateur VHDL et le simulateur de jeu d'instructions sont fournis par Mentor Graphics, ou par Cadence pour le simulateur Verilog. La communication entre les deux simulateurs est immédiatement disponible. Mais en cas de besoin, un simulateur de jeu d'instructions client peut être intégré à l'outil. Pour cela, une interface de programmation standard (API) est disponible. Le travail de reciblage de l'interface et d'intégration est assuré par Mentor Graphics, car il requiert une expertise dans la programmation interne de l'outil.

Remarques

L'outil Seamless autorise la co-simulation de plusieurs logiciels, chacun étant associé à une API propre. Les mémoires de chacun des processeurs sont simulées indépendamment. En conséquence, la co-simulation de plusieurs processeurs et d'une mémoire partagée ne peut se faire qu'en implémentant cette mémoire dans la partie matérielle. L'optimisation des accès à une mémoire partagée est impossible.

Pour des raisons qui ne semblent pas être d'ordre technique, la co-simulation de logiciel avec les simulateurs VHDL de Cadence (Leapfrog) et Synopsys (VSS) n'est pas envisagée.

3.2.2 EagleI

L'outil EagleI de Eagle Design Automation [12][13] propose un environnement de co-simulation matériel-logiciel multi-niveau, basé sur la technologie VSP (*Virtual Software Processor*, ou Processeur Logiciel Virtuel). Le Processeur Virtuel est une boîte noire ayant la même interface matérielle que le processeur réel, connectée d'une part au logiciel et d'autre part au reste du système. La connexion avec le reste du système est assurée par un module d'interface matériel, décrivant le comportement des signaux sur le bus avec un réalisme au niveau cycle. La connexion avec le logiciel est implémentée différemment selon la configuration choisie (C, simulateur logiciel ou émulateur), tout en gardant la même interface. La possibilité de garder le même environnement de co-simulation du système tout au long du cycle de conception, en changeant simplement de configuration suivant les éléments disponibles, permet d'assurer une continuité dans le flot de validation (les mêmes stimuli de test peuvent être réutilisés).

La configuration de co-simulation C-VHDL, appelée VSP/Link, consiste à compiler le code C du logiciel sur la station de travail, puis à l'exécuter en parallèle avec la simulation VHDL du reste du système. La communication entre les deux parties est assurée par un ensemble de fonctions C de base (typiquement `read_port` et `write_port`), qui interagissent avec les ports matériels modélisés dans le module d'interface. La gestion des signaux d'interruption est assurée. Dans cette configuration, la mémoire du processeur est modélisée par logiciel (mémoire d'exécution sur la station de travail) assurant une vitesse de co-simulation bien plus grande qu'avec un modèle matériel. Cette vitesse peut varier entre 3 Kips (milliers d'instructions par seconde) et 3 Mips (millions d'instructions par seconde).

Le développement du modèle VSP/Link (fonctions de communication et modèle matériel de l'interface) est estimé entre 4 et 12 semaines.

La configuration de co-simulation matériel-VHDL, appelée VSP/Sim fait appel à un simulateur de jeu d'instructions sur lequel tourne le code assembleur du logiciel, par l'intermédiaire du système d'exploitation final. La communication est à nouveau assurée au niveau cycle avec le reste du système. La précision de simulation logicielle (réalisme temporel) est bien meilleure qu'avec le modèle VSP/Link, au prix d'une vitesse de simulation nettement plus faible.

La dernière configuration possible (VSP/Tap) fait appel à un émulateur matériel, implémentant la fonctionnalité du processeur cible (*In-Circuit Emulation*). Cette étape finale consiste à valider encore plus finement le logiciel embarqué dans des conditions proches du temps réel. Elle intervient après la modélisation du processeur en VHDL, donc assez tard dans le flot. Elle autorise cependant une validation du système en temps réel, avant le prototypage matériel qui est bien plus coûteux.

3.2.3 Cossap et SPW

Synopsys propose un environnement de conception de systèmes de traitement du signal, Cossap[93]. Partant d'un modèle de description du système à haut-niveau (saisi graphiquement par composition d'éléments de librairie), le partitionnement produit un modèle VHDL pour la partie matérielle (comportemental ou RTL synthétisable), et un modèle logiciel en C ou assembleur (figure 3.18). Le modèle C peut être standard, ou bien optimisé pour un

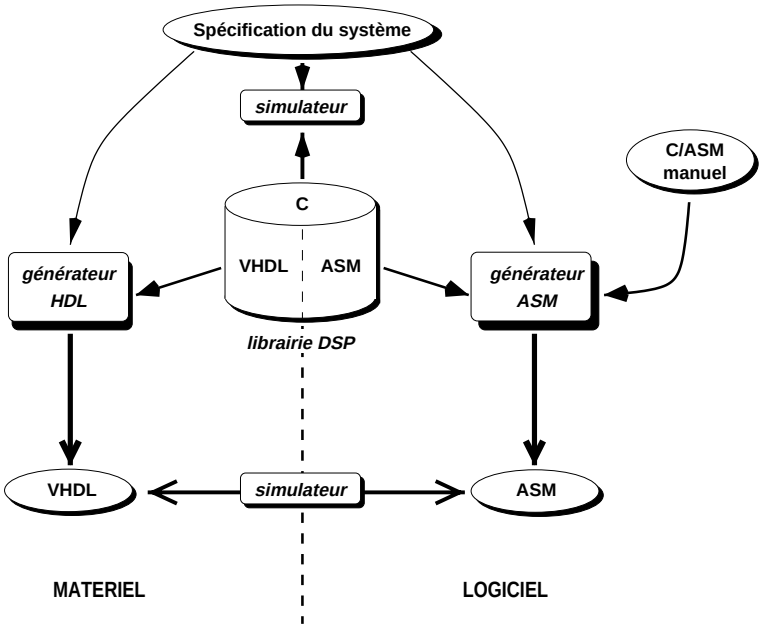

FIG. 3.18: Flot de conception dans l'environnement Cossap

processeur donné s'il est disponible en librairie. Le code assembleur, spécifique à un processeur, doit lui aussi être disponible en librairie. L'utilisateur peut par ailleurs introduire son propre code C ou assembleur.

On remarque que le modèle C du logiciel est rarement utilisé pour produire le code assembleur final. Cela est essentiellement dû à l'inexistence de compilateurs C performants dans le domaine du traitement du signal. En pratique, le modèle C est uniquement utilisé pour la validation de la fonctionnalité à haut-niveau (enchaînement des traitements, modèles

de données, etc...) et du partitionnement matériel-logiciel. Ce modèle sert de référence pour choisir et ordonner les composants de base de la bibliothèque. Alors, les modèles C bas-niveau ou assembleur de ces composants sont effectivement utilisés pour la réalisation finale. Les modèles de la bibliothèques sont écrits à la main, et fournis par Synopsys. Après compilation éventuelle des blocs écrits en C, nous obtenons un unique code assembleur pour l'application. C'est alors que ce code assembleur est introduit dans un simulateur de jeu d'instructions, pour être validé dans l'environnement matériel.

Une telle approche n'est pas adaptée à un flot de conception continue du logiciel, depuis le modèle en C haut-niveau jusqu'au code assembleur. La validation effective du logiciel réellement implémenté intervient au niveau assembleur, nécessitant un simulateur de jeu d'instructions dédié. Mais l'obstacle majeur de cette méthodologie pour la conception et le développement de processeur dédiés, est que le code (C bas-niveau ou assembleur) utilisé pour la réalisation doit être disponible sous forme de fonctions de base, dans une librairie. Cela interdit évidemment toute exploration d'architecture (il faudrait regénérer la librairie complète), et pénalise le développement et la validation rapide de systèmes dédiés.

Alta Group, de Cadence Design Systems, offre un environnement de conception et de validation d'un système à base de processeur de traitement du signal, SPW [46][47]. La méthodologie est très similaire à Cossap de Synopsys. La validation du logiciel final est effectuée grâce à la simulation du code assembleur par un simulateur de jeu d'instructions, lequel est relié à l'environnement matériel.

3.2.4 Coware

Coware, développé initialement à l'IMEC (Leuven, Belgique) puis industrialisé par CoWare Inc., est un environnement de co-conception matériel-logiciel à haut-niveau [28] [91] [27]. Utilisant les langages de programmation et de description classiques (C, C++, VHDL), il est ouvert aux outils de développement standards.

Flot de conception

Le système est décrit sous forme de *threads*, c'est-à-dire un ensemble de blocs de code écrits en C, C++ ou VHDL, pouvant être éventuellement exécutés en parallèle. Les interactions entre les blocs (passage de valeurs) ne peuvent avoir lieu qu'au début ou à la fin d'un *thread* (comme une fonction). Lorsque ce partitionnement est validé, la partie logicielle est optimisée afin de minimiser le nombre de *threads*. Pour des applications mono-flot, nous obtenons en fait un unique programme séquentiel en C ou C++. Celui-ci peut alors être compilé (si un compilateur est disponible) ou servir de référence pour l'écriture d'un code assembleur. Dans la cas d'applications à contrôle complexe, un noyau temps réel peut être invoqué (s'il est disponible en librairie). Pour le choix du processeur embarqué sur lequel doit tourner le logiciel, deux approches sont possibles : soit un coeur de processeur standard est utilisé, soit un processeur dédié est conçu. Dans le premier cas, l'interface du coeur est disponible en librairie. L'utilisateur fait alors appel à des primitives de communication existantes, spécifiques à l'interface.

Validation par la co-simulation

Dans les toutes premières étapes de la conception du système, nous disposons d'un modèle mixte VHDL-C organisé sous forme de *threads* parallèles. Ce modèle peut alors être co-simulé. Le mécanisme utilisé est l'appel de procédures à distance (RPC), bien adapté au modèle de *threads*. Le partitionnement et la fonctionnalité du système sont ici validés. Le modèle C utilisé pour la partie logicielle n'est cependant pas exactement celui qui sera

implémenté, essentiellement à cause de l'existence de *threads* multiples. La phase de minimisation du nombre de *threads* introduit une modification sensible du code C, lequel sera ensuite compilé.

Après compilation du logiciel, il est possible d'effectuer une co-simulation au niveau assembleur. En effet l'environnement Coware rend possible la connexion d'un simulateur de jeu d'instructions au simulateur VHDL. Quelques procédures de base du simulateur de jeu d'instructions (initialisation, simulation d'un seul cycle) doivent être accessibles par l'environnement.

Modélisation de l'interface

L'interface entre le matériel et le logiciel (i.e. l'interface du processeur embarqué) est directement disponible si le processeur est en librairie. Sinon, l'interface peut être générée. En fait, les interfaces (standards ou nouvelles) sont décrites suivant plusieurs niveaux d'abstraction :

- à haut-niveau, nous distinguons simplement un ensemble de données,
- ensuite, une taille est associée à chaque donnée,
- enfin, un ensemble de ports est alloué pour transmettre la donnée (il peut y avoir un vecteur de bits pour la donnée elle-même et un ou plusieurs bit d'acquittement, associés à des délais de temporisation).

Avantages et inconvénients

L'environnement Coware est particulièrement adapté au développement rapide de systèmes embarqués. La disponibilité de coeurs de processeurs et des interfaces correspondantes, aussi bien en VHDL qu'en C, permet d'obtenir facilement un prototype réaliste. Les possibilités de co-simulation, aussi bien au niveau C qu'au niveau assembleur, autorisent une validation continue tout au long de la conception, et ce dans un environnement unifié. Notamment, la co-simulation C-VHDL intervenant avant le choix du processeur et de l'interface (grâce au modèle de *threads*), la validation fonctionnelle de l'application complète peut être effectuée très tôt dans le flot de conception. L'utilisation des outils de développement standards aussi bien en VHDL qu'en C avantage la conception rapide et le déboguage du système.

Le niveau d'abstraction élevé de la description de l'interface peut être un atout pour l'écriture de protocoles complexes. De plus, il facilite l'interaction entre le modèle VHDL (dont le niveau d'abstraction est bas) et le logiciel en C ou C++ dont le niveau est plus élevé.

L'utilisation d'un modèle évolué de représentation du système à haut-niveau (*multi-thread*) peut s'avérer être un choix pertinent pour le développement de systèmes à contrôle relativement complexe, nécessitant un noyau multi-tâches en temps réel.

Cependant, la traduction du modèle logiciel original (contenant beaucoup de *threads*) en un programme C compilable introduit une modification sensible de celui-ci, nécessitant une validation ultérieure. Par ailleurs il est apparu que le code ainsi généré comportait des parties inutilisées, impliquant une consommation de mémoire programme inutile si le compilateur n'est pas capable de les éliminer.

3.2.5 Synthesia

Synthesia [5][3][4] propose un environnement de co-vérification matériel-logiciel destiné à la conception conjointe d'un système, décrit en VHDL pour la partie matérielle, et en C

(ou C++ et Ada) pour la partie logicielle. Cette co-vérification intervient après le partitionnement du système, dans le but de valider celui-ci avant la réalisation finale. L'ensemble du matériel tourne sur un unique simulateur VHDL, alors que le ou les logiciels sont directement exécutés sur la station de travail. La communication entre le simulateur et les programmes est assurée par un mécanisme standard de communication inter-processus appelé RPC (*Remote Procedure Call* - appel de procédure à distance) [29]. L'interface entre le matériel et le logiciel est constituée d'un ensemble de données, de type simple standard (bit, octet, mots) ou complexes (structures, types énumérés, unions). Les simulateurs actuellement supportés sont Leapfrog de Cadence et VHDL Testbench.

Technique de communication - synchronisation

Du fait des limitations fonctionnelles de l'interface C de la plupart des simulateurs VHDL, un mécanisme de communication inter-processus est nécessaire pour assurer une validation fidèle de l'application logicielle qui doit être implémentée. Il s'agit ici du mécanisme d'appel de procédure à distance, dont le principe est exposé au paragraphe 3.1.5.

Implémentations logicielle et matérielle

La communication entre le simulateur et l'application est assurée, de chaque côté, par une bibliothèque de procédures spécifiques. En C, un ensemble de procédures d'initialisation, d'envoi de requête et d'attente sont directement appelées par l'application utilisateur. En VHDL, nous disposons à la fois d'un *package* VHDL utilisé par le modèle matériel, et d'un groupe de fonctions C (non visibles par l'utilisateur) gérant l'appel des primitives système RPC. En effet, la gestion du mécanisme RPC n'est pas possible en VHDL. Il faut faire appel à des routines C, qui sont liées au simulateur VHDL lui-même. La gestion des appels de procédure RPC fait usage de primitives-système UNIX dont la mise en oeuvre nécessite un temps de développement et des connaissances spécifiques importants. C'est pourquoi l'essentiel des opérations de communication sont génériques. L'utilisateur ne voit donc de chaque côté (VHDL et C) qu'un ensemble de fonctions (*package* VHDL et bibliothèque C) gérant l'initialisation, l'envoi et la réception de requêtes.

Implémentation par l'utilisateur

La réalisation d'un modèle communicant nécessite l'utilisation de quelques primitives d'échange, basées sur un modèle client-serveur. Pour la partie VHDL, quelques dizaines de lignes de code comportemental décrivent l'initialisation et le protocole d'échange de données. Ce code doit être ajouté au modèle VHDL du circuit, et agir par effet de bord sur l'état du système. Pour la partie C, un code de taille équivalente remplit exactement la même fonctionnalité.

Performances du mécanisme RPC

Une étude menée par Synthesia sur les performances de RPC révèle que le protocole de communication nécessite moins d'un quart de la puissance de calcul de la station de travail, en moyenne et pour une configuration à un seul client. Par ailleurs, le coût de la communication (intra ou inter-machine) implique que la répartition des processus sur plusieurs machines n'est intéressante qu'à partir de trois processus (un simulateur VHDL et deux applications C). En effet, deux processus communicants, sur deux machines différentes, tournent sensiblement à la même vitesse que sur une machine unique.

Avantages et inconvénients

L'approche de Synthesia est parfaitement adaptée à la conception d'applications logicielles en C embarquées, autorisant l'utilisation d'outils de développement en C standards. La flexibilité de l'interface entre le logiciel et le matériel (un ensemble de données de types variés) autorise une exploration de l'interface finale, ainsi qu'un temps de mise en place réduit. La généricité du code (aussi bien VHDL que C) disponible sous forme de *packages* ou bibliothèques facilite la mise en place de la co-simulation, et autorise la réalisation de configurations variées. Le choix du mécanisme de communication RPC semble bien adapté au type d'informations échangées (quelques dizaines d'octets à la fois), et autorise la répartition des tâches sur plusieurs machines à travers le réseau.

On peut toutefois regretter le choix du niveau d'interaction entre l'application, le modèle VHDL, et les bibliothèques de communication. En effet, deux aspects de cette interaction peuvent être pénalisants à l'usage : le premier concerne le moyen de transmettre les données à la partie distante. Il faut utiliser une seule structure (car la procédure n'admet qu'un seul argument) contenant l'ensemble des informations. Celles-ci doivent donc être regroupées avant d'être envoyées, et décompactées après réception. Cela introduit une modification de la mémorisation des données dans le code et de leur traitement, entraînant une différence de fonctionnalité par rapport à l'application qui sera implémentée.

La seconde restriction concerne les procédures de communication (aussi bien C que VHDL) qui doivent être appelées par l'application et le modèle matériel. Reste à la charge de l'utilisateur la gestion de l'initialisation de la communication (choix d'un numéro de canal), ainsi que le déroulement du protocole d'échange de données (envoi de requête, attente de réponse, test de disponibilité du canal). Nous pouvons identifier deux inconvénients à cette approche. En premier lieu, une mauvaise gestion de la communication peut entraîner des erreurs liées à la co-simulation et non pas à l'application de l'utilisateur. En second lieu, le code applicatif est notablement différent de celui qui sera implémenté.

Concernant le réalisme temporel de la co-vérification avec Synthesia, nous notons l'absence d'un modèle d'annotation du programme C, permettant de simuler les délais de traitement de chaque instruction assembleur à prévoir en fonctionnement réel. Un tel modèle permettrait de retarder la simulation VHDL pour certaines parties de l'application, et donc d'approcher un réalisme en terme de cycles d'instruction.

3.3 Spécification de l'outil CoSim

Le besoin d'un outil de co-simulation C-VHDL a été exprimé lors de la conception d'un circuit à très haute densité d'intégration développé au département Central R&D de SGS-THOMSON. Le circuit IVT (*Integrated Videotelephone Terminal*) est dédié au traitement d'images en temps réel, pour une application de visiophonie. La puissance de calcul nécessaire, ainsi que la nécessité de conserver une grande flexibilité des algorithmes, ont conduit les concepteurs à réaliser un système multi-processeur, où chacun des processeurs est dédié à une tâche précise. L'utilisation exclusive de compilateurs C dans le développement des logiciels embarqués introduit un niveau supplémentaire de modélisation du système complet, nécessitant logiquement une nouvelle étape de validation. Le fait que le logiciel soit écrit dans un langage de haut-niveau (ici le C) donc compilable et exécutable sur n'importe quelle station de travail, permet d'espérer une vitesse de simulation au niveau système compatible avec la complexité du système.

Afin de pouvoir disposer d'un outil utilisable par l'équipe de Central R&D dans le délai qui était imparti à la conception et à la validation des processeurs embarqués, le domaine d'application de la première version de l'outil est strictement défini à l'avance. En parallèle,

l'extension des fonctionnalités de l'outil est étudiée, afin de spécifier les caractéristiques de la première version de production devant intégrée à la chaîne de conception Unicad de SGS-THOMSON [87].

Les spécifications de l'outil de co-simulation C-VHDL CoSim ont été définies en collaboration avec l'équipe de conception du circuit IVT. Ce circuit est représentatif des futurs systèmes mono-puces intégrant à la fois des fonctions multimedia demandant beaucoup de calcul, et des fonctions de contrôle complexes[3]. L'objectif était d'appliquer et d'étendre le modèle de co-simulation C-VHDL développé dans l'outil VCI [112][113] du laboratoire TIMA, en tenant compte des contraintes industrielles (flots de développement, outils commerciaux utilisés, temps de conception) ainsi que des exigences liées à un système aussi complexe que l'IVT.

3.3.1 Fonctionnalités

La spécification des fonctionnalités de l'outil CoSim est largement influencée par les besoins exprimés lors de la conception du circuit multi-processeur IVT. Toutefois, une attention particulière est apportée à généraliser les méthodes employées, de sorte que l'outil ne soit pas spécifiquement dédié à un flot particulier. Les caractéristiques fonctionnelles de l'outil sont ici détaillées.

Application à la validation fonctionnelle

L'application principale de la co-simulation à haut-niveau est la validation fonctionnelle d'un logiciel embarqué communicant avec le reste du système, matériel ou logiciel. La validation fonctionnelle consiste à vérifier que la réalisation des spécifications d'un algorithme se comporte comme prévu, aussi bien en fonctionnement interne que lorsqu'il communique avec l'extérieur (reste du système). Cette phase de validation n'est pas concernée par l'aspect temporel, c'est-à-dire le temps mis pour exécuter une tâche, ou les délais de communication entre deux tâches [105].

Utilisation de descriptions haut-niveau

Le logiciel est initialement écrit en C, et doit être co-simulé sous cette forme. En effet, seule la description haut-niveau du logiciel est indépendante de l'architecture du processeur sur lequel il devra tourner. Cela est indispensable pour pouvoir exécuter la co-simulation assez tôt dans le flot de conception, avant la réalisation du processeur lui-même. De plus, l'exécution d'un programme C (compilé sur la station de travail) est considérablement plus rapide que la simulation du jeu d'instructions à partir du code assembleur. Cette vitesse d'exécution permet d'envisager une simulation globale du système, sur une longue période de temps simulé. Le taux de couverture de la validation en est considérablement amélioré (cf. Résultats).

Unicité du code compilé et simulé

Afin de procéder à une validation exacte du logiciel qui sera implanté dans le système, le code C utilisé pour la co-simulation doit être le plus proche possible de celui effectivement compilé. Les seules différences porteront sur les fonctions de communication bas-niveau (accès aux ports d'entrée-sortie) qui seront implémentées différemment pour la co-simulation, selon le mécanisme de communication employé.

[3] L'architecture et le flot de conception de ce circuit sont présentés en détail dans le chapitre 2.

Compatibilité avec les environnements standards

Le développement de logiciel en C est considérablement aidé par de nombreux d'outils pour écrire, maintenir, archiver, profiler et analyser les programmes les plus complexes. Il est indispensable de pouvoir utiliser ces outils lors de la co-simulation du logiciel avec le reste du système. La co-simulation C-VHDL devra donc être transparente par rapport à l'environnement de développement logiciel utilisé, quel qu'il soit. De même les environnements de conception matérielle doivent pouvoir être utilisés de la même façon que pour une simulation VHDL.

Applications connexes

Grâce à la rapidité de reconfiguration d'un logiciel écrit en C, il est envisageable d'explorer la répartition des fonctionnalités sur les différents processeurs embarqués en un temps réduit. Cette exploration d'architecture-système, faite très tôt dans le flot de conception, doit permettre d'optimiser notablement les communications entre les processeurs, ainsi que le taux d'occupation de chaque processeur.

3.3.2 Configuration du modèle VHDL du système

La configuration d'un système pour la co-simulation d'un de ses processeurs embarqués consiste à substituer au modèle VHDL du processeur, encore inexistant, une boite noire connectée au logiciel du processeur[4]. La partie gauche de la figure 3.19 représente la configuration du système complet où le processeur est modélisé en VHDL (en grisé). Ce dernier

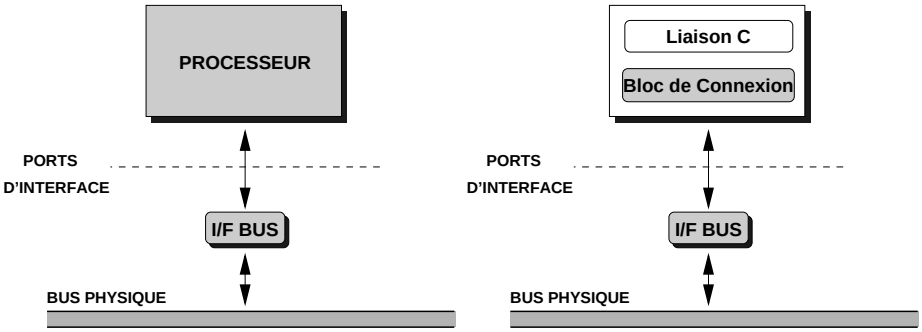

FIG. 3.19: Substitution de l'architecture du processeur embarqué par son modèle C

est connecté au bus physique du système via le bloc interface bus (I/F bus). Dans la partie droite de la figure 3.19, le modèle VHDL du processeur est remplacé par un bloc ayant la même interface (même connexion à l'interface bus) mais ne contenant que deux parties : le bloc de connexion en VHDL, et le bloc de liaison C, permettant la co-simulation avec le modèle C du logiciel embarqué.

Il est important de remarquer que dans cette dernière configuration, seul le contenu du bloc instanciant le processeur est modifié. L'interface avec le reste du système, via l'interface-bus, reste inchangée. Le rôle du bloc de connexion sera détaillé plus loin. Schématiquement, il assure la transmission et la conversion des données entre le monde VHDL et le monde C. Notamment il gère la validité dans le temps des données venant du programme C, par rapport aux diverses horloges et signaux internes. Le bloc de liaison C est

[4]Afin de simplifier l'exposé, on considère ici la cosimulation d'un seul processeur embarqué. L'outil CoSim est cependant conçu pour cosimuler un nombre quelconque de processeurs.

directement connecté au logiciel applicatif embarqué. Sa réalisation est également décrite plus loin.

3.4 Mise en œuvre de la co-simulation C-VHDL sous Unix

Cette section détaille la configuration des différentes fonctionnalités nécessaires à la mise en place de la co-simulation C-VHDL. La communication inter-processus par file de messages, l'interfaçage avec un simulateur VHDL, et la synchronisation de deux processus communicants sont successivement abordés.

3.4.1 Communication inter-processus

Application du mécanisme IPC/f à la co-simulation

La co-simulation requiert une communication bi-directionnelle entre les deux processus (simulateur VHDL et application C). Le mécanisme IPC autorise aux deux processus communicants d'écrire et de lire dans une même file, ce qui en fait une file bi-directionnelle. La distinction entre les messages tramsmis est assurée par un identificateur de type, associé à chaque message. Par exemple, deux types IN et OUT suffisent à exploiter une unique file pour une communication bi-directionnelle fiable. Cette technique a été employée ici. Toutefois, pour la clarté de cet exposé, nous considérerons par la suite deux files unidirectionnelles appelées IN et OUT.

Le message transmis est une suite d'octets de taille prédéfinie, dont le format est connu à l'avance par chacune des parties communicantes.

La figure 3.20 représente plus précisément l'évolution dans le temps des deux processus A et B communicant, ainsi que l'échange de messages entre eux, dans les deux sens. Le

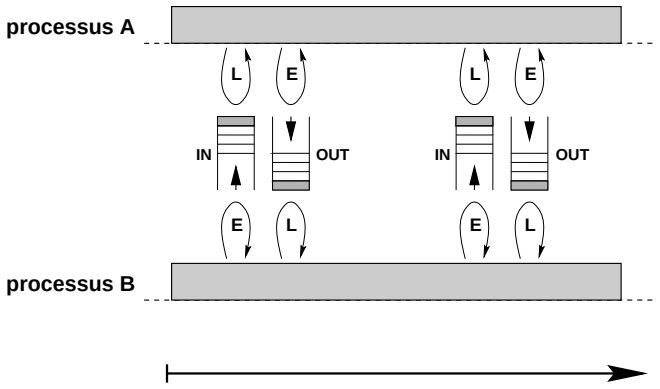

FIG. 3.20: Communication inter-processus par file de messages

processus A communique avec le processus B via les deux files de messages appelées IN et OUT, par convention. Le processus A ne peut que lire la file IN et écrire dans la file OUT. Inversement, le processus B ne peut qu'écrire dans la file IN et lire dans la file OUT.

On note ici que les deux files fonctionnent en alternance, assurant ainsi une progression équilibrée des deux processus. La synchronisation des messages (gestion de l'inter-blocage, initialisation) sera expliquée plus loin.

Réalisation en C

La librairie IPC/f standard fournit un ensemble de fonctions d'accès aux files de messages, gérant leur création, leur destruction, la lecture et l'écriture. Afin de simplifier l'u-

tilisation du mécanisme IPC/f et d'assurer la robustesse de la couche de communication, un ensemble restreint de fonctions de plus haut-niveau a été développé. Ces primitives font exclusivement appel aux fonctions de base d'IPC/f. L'accès aux files de messages se fait alors très rapidement à l'aide des seuls types de données et fonctions définis en figure 3.21.

```
typedef int queue_t;
struct ports_t {
       int din;
       int cmd;
       int reset;
}

/* WRITE update and go */
void IPC_Write_Nowait (queue_t, struct ports_t *);

/* READ update and go */
void IPC_Read_Nowait (queue_t, struct ports_t *);

/* wait for update and READ it */
void IPC_Read_Wait (queue_t, struct ports_t *);
```

FIG. 3.21: Fonctions haut-niveau d'accès aux files de messages

Le type `queue_t` est un identificateur (nombre entier) de la file de messages, et le type `ports_t` est une structure contenant les valeurs des ports d'interface pour une direction donnée.

La fonction `IPC_Write_Nowait()` prend en paramètres un identificateur de file d'attente et un pointeur sur une structure de ports en sortie. Elle écrit les valeurs des ports dans la file, après avoir vérifié que la file existe. Si elle n'existe pas, elle la crée.

Les fonctions `IPC_Read_Wait()` et `IPC_Read_Nowait()` permettent de lire une structure de ports en entrée. La différence entre les deux est leur comportement lorsqu'aucun message n'est présent dans la file au moment de l'appel de la fonction. Dans le premier cas (Wait), la fonction se met en attente d'un message[5], alors que dans le second cas (Nowait) la fonction n'attend pas et retourne immédiatement un code d'erreur. Ce code d'erreur devra être interprété au niveau supérieur, selon le protocole[6].

Nous disposons donc d'un moyen de communication bi-directionnel entre deux programmes C. Reste à connecter les deux parties (simulateur VHDL et application C) entre elles par l'intermédiaire de ce mécanisme.

3.4.2 Liaison avec le simulateur VHDL

Le mécanisme de communication retenu, IPC/f, est écrit en C sous Unix. Aucun simulateur VHDL commercial ne fournit d'accès à la communication inter-processus en standard (ni en VHDL, ni autrement). Il font donc écrire une librairie d'accès à IPC en C, puis appeler les fonctions de cette librairie pendant la simulation VHDL. Heureusement, la quasi-totalité des simulateurs VHDL propose l'appel de fonctions C pendant la simulation, comme cela a été décrit plus haut. Cet appel de fonction C va alors servir uniquement à établir une communication avec l'application C, via IPC.

[5] Un *timeout* est toutefois implémenté afin d'éviter d'attendre indéfiniment. Le programme est alors stoppé.

[6] Dans le but de clarifier cet exposé, le code fourni ici est volontairement simplifié. Le code réellement implémenté dans l'outil CoSim, disponible en annexe, est plus complexe donc plus difficile à lire. Les noms de fonctions employés et leur syntaxe peuvent parfois différer, mais la fonctionnalité est identique.

La communication entre le simulateur VHDL et l'application C, via IPC/f, est assurée par la structure représentée en figure 3.22.

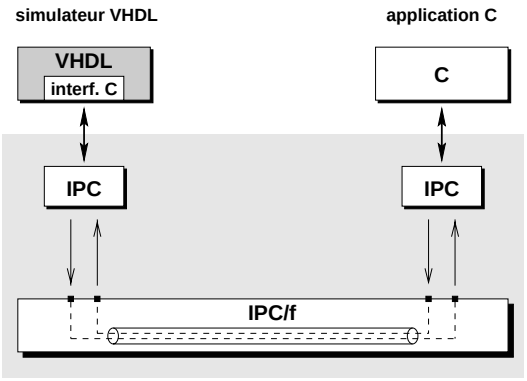

FIG. 3.22: Connexion des couches IPC et C-VHDL

Nous utilisons ici deux files de messages IPC, auxquelles sont connectés deux blocs de communication appelés IPC, nécessairement écrits en C. Il faut distinguer le moyen de communication lui-même, à savoir les files de messages IPC/f, des fonctions d'accès à celui-ci, contenues dans les blocs IPC. Chaque bloc IPC contient les fonctions standard de la librairie IPC/f, ainsi que les fonctions de plus haut-niveau développées spécialement et décrites plus haut. L'un des blocs est connecté au simulateur VHDL via l'interface C de ce dernier (à gauche), et l'autre est connecté à l'application C elle-même (à droite). Cette structure en couches (applicatifs C-VHDL et IPC) permet d'isoler les fonctionnalités propres à la co-simulation, et de conserver les modèles VHDL et C du système intacts. De plus, l'implantation de la communication inter-processus est elle-même interchangeable, autorisant l'évolution de l'outil vers d'autres mécanismes, si nécessaire.

L'évolution dans le temps du mécanisme complet de co-simulation (interface C-VHDL et communication IPC) est schématisée figure 3.23. Il nécessite l'assemblage du mécanisme de communication (IPC) et de l'appel d'une fonction C pendant la simulation VHDL (interface C-VHDL). La surface grisée représente la communication IPC entre les deux fonctions `vhdl_transaction()` et `c_transaction()`, via les deux files de messages IN et OUT au centre. La fonction `vhdl_transaction()` est appelée en haut par le simulateur VHDL et la fonction `c_transaction()` est appelée en bas par l'application C elle-même. Le code de ces deux fonctions est différent, mais la fonctionnalité est exactement la même. Ces fonctions gèrent la synchronisation et l'échange de messages, comme cela est détaillé dans la section suivante.

Nous retrouvons sur ce schéma la structure en couche, avec d'une part la couche de communication (blocs IPC et files de messages) et d'autre part la partie applicative (application logicielle et le reste du système matériel).

3.4.3 Synchronisation

La synchronisation des deux parties communicantes (simulateur VHDL et application C) est le point-clé de la co-simulation C-VHDL. Elle pose plusieurs problèmes qui peuvent être séparés comme suit :

- avancement concurrent de la co-simulation,
- initialisation et terminaison de la communication,
- gestion des interruptions et du RESET.

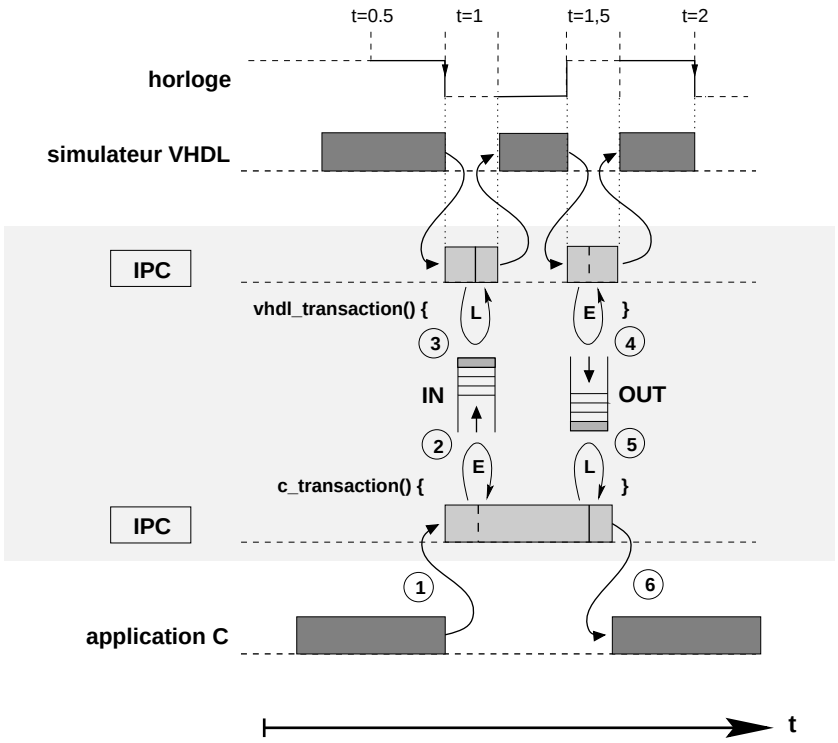

FIG. 3.23: Cosimulation C-VHDL par file de messages

Le modèle de synchronisation doit respecter plusieurs critères :

1. Il doit permettre de gérer plusieurs applications C embarquées dans un même système, pouvant éventuellement échanger des données entre-elles.
2. Il doit supporter n'importe quel protocole de communication utilisateur, et garantir un comportement identique à celui de la simulation matérielle.
3. Enfin, il doit être suffisamment robuste pour supporter un fonctionnement imprévisible de l'application, puisque la co-simulation est essentiellement utilisée pour la validation et le déboguage.

Avancement concurrent de la co-simulation

La progression de la co-simulation ne doit en aucun cas être dépendante du protocole de communication utilisateur, c'est-à-dire qu'aucune hypothèse ne doit être faite sur celui-ci. Notamment nous ne pouvons pas supposer que le protocole utilisateur fonctionne correctement. C'est pourquoi il a été choisi d'établir une communication permanente entre les deux parties, même si le protocole utilisateur ne progresse pas lui-même. Une transaction est engagée régulièrement entre le simulateur VHDL et l'application C. Cette régularité élimine immédiatement tout risque de blocage de la communication.

Les deux parties communicantes (simulation VHDL et application C) progressant en parallèle, la synchronisation doit être assurée des deux côtés. Du côté VHDL il est facile d'utiliser le signal d'horloge lui-même, ou un signal qui en dérive, pour réguler les transactions. Définir un signal de synchronisation qui dérive de l'horloge permet de réduire la fréquence des échanges, et donc d'accélérer la co-simulation. Il peut être également adapté suivant les cas (co-simulation très précise ou co-simulation globale, plus rapide).

Du côté C la transaction n'est engagée que lorsqu'une instruction d'entrée-sortie est invoquée, quelle que soit la direction de cette entrée-sortie. Ainsi, nous nous assurons que

l'échange n'est effectué que si cela est nécessaire. Les phases de calcul interne (entre chaque entrée-sortie) peuvent se dérouler à la vitesse d'exécution maximale de la station de travail, sans interruption inutile.

La transaction elle-même s'effectue en deux étapes, une pour chaque direction. Illustrons cela avec la figure 3.23 précédente. A un moment donné, l'application C engage une transaction en appelant la fonction `c_transaction()` (1). Celle-ci envoie une mise-à-jour dans la file d'attente IN (2), et se bloque en attente de lecture sur la file OUT (5).[7] Le simulateur VHDL, qui était en attente de lecture sur la file IN (3), lit alors le message de mise-à-jour. Après avoir fait la mise-à-jour des signaux internes, il reprend la simulation pour une demi-alternance (ici un demi-cycle). A l'alternance suivante, le simulateur écrit un message dans la file OUT (4) et reprend la simulation. Cela débloque l'application C qui était en attente de lecture sur OUT (5), qui peut alors mettre à jour ses propres variables et reprendre son exécution interne (6) jusqu'à la prochaine entrée-sortie.

Nous notons qu'avec ce mécanisme, un message ne peut être écrit que si le précédent a été lu, donc ôté de la file. Par conséquent, il n'y aucun risque de débordement des files de messages.

Afin d'optimiser la fréquence des transactions, un second mode de synchronisation a été implémenté. Il permet de modifier la fréquence des transactions pendant la co-simulation, selon l'avancement de celle-ci. Ce mode est particulièrement adapté aux applications où la communication n'est pas très fréquente, ou n'est pas constante durant l'ensemble de la simulation. Il sera décrit en détail dans la section se rapportant à l'application industrielle de CoSim.

Initialisation et terminaison de la communication

La communication par file de messages IPC ne nécessite pas que les deux parties communicantes soient disposées à émettre ou recevoir en même temps pour que l'échange ait lieu. Par contre, l'avancement de la co-simulation est stoppé dès que l'une des parties ne répond plus, comme exposé plus haut. Ce comportement est mis à profit pour assurer une co-simulation précise dès le lancement du simulateur.

Il est arbitrairement convenu que l'application C initie l'échange par la création des files de messages et l'émission immédiate d'un message de synchronisation, vide de sens[8]. En parallèle, le simulateur VHDL commence la simulation, et atteint très rapidement un front déclenchant une transaction (front descendant de l'horloge par exemple). Il se met alors en attente du message de synchronisation venant de l'application, ou des applications s'il y a plusieurs processeurs. La simulation ne commence réellement que si tous les messages de synchronisation de toutes les applications C sont parvenus. Cela garantit une initialisation correcte de toutes les communications. De plus, si un signal RESET est émis par le modèle VHDL dès le début de la simulation, il sera assurément pris en compte par les applications puisque les communications seront déjà initialisées.

La terminaison de la co-simulation s'effectue en deux étapes : dans un premier temps la co-simulation du système est stoppée, c'est-à-dire que le temps simulé est arrêté. Dans un deuxième temps l'utilisateur quitte le simulateur, donc le processus de ce dernier ainsi que celui de l'application C sont détruits. C'est alors que les files de messages sont elles-mêmes détruites.

La fin de la co-simulation du système intervient dans deux cas : soit la simulation VHDL

[7] Un message de mise-à-jour contient la valeur de tous les ports d'interface du processeur cosimulé pour une direction donnée, qu'ils aient changé ou non.

[8] Ce comportement est assuré par la fonction de transaction fournie en bibliothèque. Le code utilisateur lui-même n'est pas tenu de s'en charger.

est stoppée (ou interrompue), soit l'application C est terminée (ou interrompue). Dans ces deux cas il n'est pas souhaitable que les files de messages soient détruites, afin qu'il soit toujours possible de reprendre la co-simulation (par exemple par l'envoi d'un signal RESET manuel). Pour ce faire l'application C ne doit jamais se terminer, et continuer d'échanger des messages avec le simulateur. En effet nous avons vu que la co-simulation ne progresse que s'il y a en permanence un échange de messages.

Au contraire, la terminaison des processus intervient lorsque l'on quitte réellement le simulateur VHDL. Alors, les applications C sont définitivement stoppées (les processus sont tués) et les files de messages sont détruites. Il n'est pas nécessaire que la co-simulation du système soit terminée avant de procéder à la destruction des processus. Ainsi, même en cas de dysfonctionnement de la station de travail (destruction accidentelle de processus, ou bogue), la session de co-simulation se termine toujours correctement.

Réalisation en C

La réalisation en C du mécanisme de synchronisation est ici décrite. Il faut distinguer la réalisation de la synchronisation dans le simulateur VHDL de sa réalisation dans l'application C. Chacun appelle une fonction de transaction dédiée.

Le simulateur VHDL appelle régulièrement la fonction `vhdl_transaction()`. Le code simplifié de celle-ci est présenté en figure 3.24. Cette fonction a accès aux identificateurs des

```
vhdl_transaction {
      queue_t IN, OUT;
      struct ports_t in_ports, out_ports;

      /* write on rising edge */
      if (clock == 1) {
         Get_VHDL_values (&out_ports);
         IPC_Write_Nowait (OUT, out_ports);
      }

      /* wait for update and read it */
      else {
         IPC_Read_Wait (IN, &in_ports);
         Put_VHDL_values (in_ports);
      }
}
```

FIG. 3.24: Fonction de transaction appelée par le simulateur VHDL

deux files IN et OUT (variables globales), alors que deux structures temporaires internes, `in_ports` et `out_ports`, contiennent les valeurs des ports en entrée et en sortie respectivement.[9] Selon la nature du front de l'horloge de synchronisation, une lecture ou une écriture est exécutée. Dans le cas d'un front montant de l'horloge, la fonction `vhdl_transaction()` est appelée juste après celui-ci. Le signal `clock` a donc déjà la valeur 1. Un message de mise-à-jour doit alors être écrit dans la file OUT. Mais auparavant, la structure `out_ports` doit être remplie avec les valeurs des ports d'interface issus du modèle VHDL : la fonction `Get_VHDL_Values()` accède aux structures de données internes au simulateur VHDL grâce à un ensemble de pointeurs et fonctions fournis par le vendeur du simulateur. Les structures de données étant différentes selon le vendeur, le code de la fonction `Get_VHDL_Values()` en est extrêmement dépendant. Cependant la fonctionnalité est tout à fait comparable d'un vendeur à l'autre. Une fois que la structure `out_ports` est renseignée, elle est écrite dans

[9] Les ports bidirectionnels `INOUT` apparaissent dans les deux structures.

la file OUT avec la fonction `IPC_Write_Nowait()`. La fonction `vhdl_transaction()` se termine et la simulation reprend.

Si au contraire l'horloge n'est pas à 1 (donc à 0, front descendant) la fonction `IPC_Read_Wait` est appelée pour recevoir une mise-à-jour venant du C. A son retour, la structure temporaire `in_ports` contient ces nouvelles valeurs. Alors, la structure de données interne au simulateur est elle-même mise à jour par la fonction `Put_VHDL_Values()` avant de revenir à la simulation VHDL. La fonction `Put_VHDL_Values()` est également très dépendante du vendeur du simulateur.

Le code C de la transaction engendrée par l'application C est donné en figure 3.25.

```
queue_t IN, OUT;

void c_transaction () {
struct ports_t in_ports, out_ports;

      /* write */
         Get_C_values (&out_ports);
         IPC_Write_Nowait (IN, out_ports);

      /* wait for update and read it */
         IPC_Read_Wait (OUT, &in_ports);
         Put_C_values (in_ports);
}
```

FIG. 3.25: Fonction de transaction appelée par l'application C

Le fonctionnement de la transaction C est comparable à celui de la transaction VHDL, mis-à-part que les deux phases de la transaction, écriture et lecture, sont ici exécutées dans le même appel de la fonction `c_transaction()`. Les deux fonctions `Get_C_Values()` et `Put_C_Values()` jouent le même rôle que leurs contreparties en VHDL. Les structures de données du côté C sont toutefois plus simples, puisqu'elles se résument le plus souvent à un ensemble de variables globales portant exactement le même nom, écrit en majuscules, que le port d'interface dont elles conservent la valeur. Cette convention, même si elle impose une restriction sur les noms des variables de l'application, s'est avérée suffisante car efficace et simple à mettre en œuvre. Toutefois, il est tout à fait possible de concevoir une structure plus compliquée, selon les besoins de l'application.

3.5 Intégration au système applicatif

La mise en œuvre des techniques de communication et d'interfaçage décrite dans la section précédente fournit tous les éléments nécessaires à la mise en place effective de la co-simulation. Toutefois, son intégration dans un milieu industriel contraint requiert le développement d'une sur-couche applicative importante, basée sur la génération automatique de code et la fourniture de modèles prédéfinis réutilisables.

3.5.1 Génération de code d'interface

La mise en place de la co-simulation C-VHDL pour un système donné requiert le développement et la mise-au-point de code C spécifique, aussi bien pour établir la communication IPC que pour traiter les messages transmis et utiliser l'interface C d'un simulateur VHDL particulier. Ce développement spécifique est pénalisant à plusieurs égards :

1. le temps de développement varie de une à dix semaines selon la complexité de l'interface et du protocole,
2. il requiert la connaissance de la programmation d'IPC et de l'interface C du simulateur (CLI, FMI, FKI,...),
3. il est source de bogues et d'erreurs d'interprétation pendant la validation du circuit,
4. sa fonctionnalité est susceptible de varier suivant les applications.

Afin de limiter l'impact de ces inconvénients il est indispensable, dans le contexte d'un développement industriel très contraint en temps, de développer la généricité de l'ensemble du code nécessaire à la co-simulation. Cela nous a conduit à concevoir un outil de génération de code d'interface, s'appuyant sur la description des ports d'interface d'un processeur pour produire, compiler et assembler tout le code nécessaire. Le coeur de cet outil de génération est inspiré de l'outil VCI, cité plus haut.

Modularité du générateur de code d'interface

La structure modulaire de cet outil de génération assure une maintenance réduite et une fiabilité accrue, comme cela a été vérifié dans son utilisation industrielle. L'outil est subdivisé en quatre modules.

1. Le premier module est une bibliothèque qui regroupe l'ensemble des fonctions de bas-niveau d'accès à IPC et à l'interface C du simulateur VHDL (actuellement les simulateurs de Cadence et de Synopsys sont reconnus). Ce module est entièrement générique.
2. Le deuxième module réalise les fonctionnalités de base d'accès aux ports d'interface d'un processeur donné, ainsi que la construction des structures de données à échanger. Cela signifie que ce module est très dépendant de l'interface du processeur. Afin de réduire le temps de développement et de fiabiliser la réalisation de la co-simulation, ce module est entièrement généré. L'outil de génération prend en entrée une description de l'interface (liste des ports, directions, types) issue de l'entité VHDL du processeur, et produit le code C dédié à cette interface.
3. Le troisième module est à nouveau générique. Il implémente essentiellement le protocole de synchronisation, et le lien avec l'interface C du simulateur. Il fait appel aux fonctions générées du module précédent.
4. Enfin le quatrième module a pour charge de compiler et assembler le code des précédents modules. Il produit deux librairies exécutables. L'une est destinée à être intégrée au simulateur VHDL, alors que la seconde librairie est automatiquement liée à l'application utilisateur pour produire un programme exécutable.

La modularité de cet outil a permi de mettre au point chaque partie indépendamment les unes des autres, limitant ainsi les effets de bord. De plus, la maintenance en est grandement facilitée.

3.5.2 Bloc matériel de connexion C/interface-bus

Le bloc de connexion a pour fonction d'assurer le transfert des signaux entre le monde VHDL et le monde C. Il est connecté d'une part au logiciel C via le bloc de liaison, et d'autre part au bus physique du système via l'interface-bus (figure 3.26).

Le transfert des signaux du monde VHDL vers le monde C et vice-versa pose deux sortes de problèmes. Le premier problème concerne le type des données échangées, qui ne sont pas de même nature. En C les données sont des nombres entiers ou des structures. En VHDL

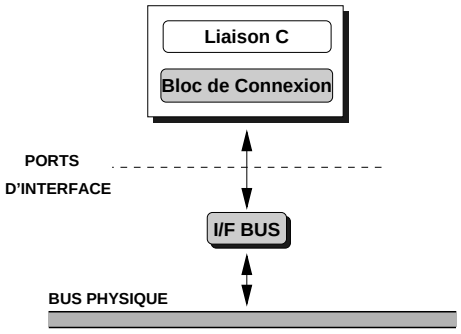

FIG. 3.26: Configuration du modèle mixte C-VHDL

les signaux peuvent avoir des valeurs particulières, comme la haute impédance, qui ne sont pas directement représentables en C. Le second problème concerne la validité des signaux par rapport à l'horloge. Les données venant du C doivent être présentées au modèle VHDL (plus exactement à l'interface-bus) à un instant précis par rapport à l'horloge, suivant les fronts de celle-ci. Inversement, les données venant du VHDL et transmises au C doivent être lues au bon moment. Enfin, le bloc de connexion a en charge la génération d'une horloge de synchronisation additionnelle à partir de l'horloge interne du circuit. Cette synchronisation dédiée à la co-simulation permet de garder une indépendance par rapport au protocole de communication utilisateur, afin de garantir l'avancement de la co-simulation même si la communication du système est bloquée suite à un bogue, par exemple.

La figure 3.27 décrit le contenu du bloc de connexion. Les conversions de type sont

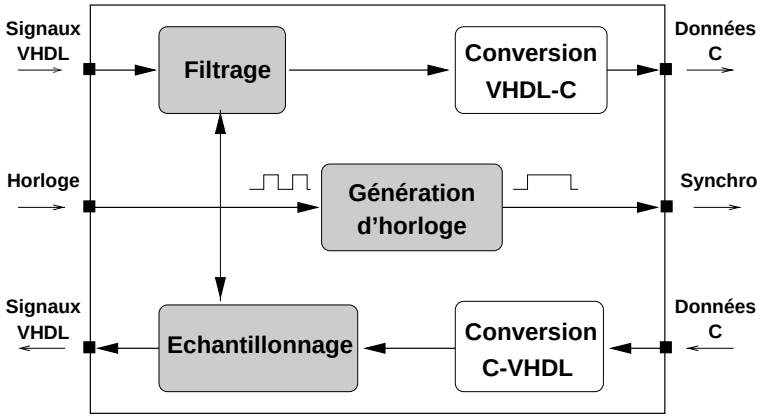

FIG. 3.27: Bloc de connexion C-VHDL

accomplies dans les deux sens par les deux blocs correspondants. Les blocs de filtrage et d'échantillonnage gèrent la lecture et l'écriture des signaux VHDL en fonction de l'horloge interne. Enfin le bloc de génération d'horloge produit un signal périodique utilisé pour la synchronisation des transactions avec l'application C.

Conversion de types entre le C et le VHDL

La conversion de types de VHDL vers C et vice-versa assure la cohérence des représentations de données échangées entre le C et le VHDL.

Lorsqu'un type de données existe aussi bien en C qu'en VHDL, avec la même sémantique, la conversion est quasi-immédiate (par exemple les nombres entiers de moins de 32 bits). Nous faisons alors appel aux fonctions de conversions standard, fournies par le langage

VHDL.

Lorsqu'un type n'est défini que dans un seul des deux langages, il faut le représenter dans l'autre langage par un nouveau type. Par exemple le type `STD_ULOGIC` en VHDL peut prendre non seulement les valeurs 0 et 1, mais aussi H, L, X, U, Z, W ou -. Afin de pouvoir utiliser ces autres valeurs en C, un type énuméré peut alors être utilisé. L'ordre des différentes valeurs dans le type énuméré est fixé dans les paquetages VHDL du simulateur, et doit impérativement être respecté. De plus, l'application C doit utiliser le type énuméré pour accéder à la donnée car les valeurs 0 et 1 ne sont pas forcément représentées par les nombres 0 et 1. Le type `STD_LOGIC` est implémenté de la même façon.

La représentation en C des vecteurs de bits en VHDL (`BIT_VECTOR`) se fait tout naturellement par un entier contenant la valeur entière du vecteur. Le cas des vecteurs `STD_LOGIC_VECTOR` ou `STD_ULOGIC_VECTOR` est particulier. Si toutes les valeurs du vecteur sont parfaitement définies (0 ou 1), nous pouvons convertir ce vecteur en une valeur entière. Cette valeur entière est alors représentée en C par un entier 32 bits classique (`int` ou `unsigned int`). Si au contraire certains bits ont des valeurs différentes de 0 et 1, des règles de conversion sont appliquées bit à bit. Au final la valeur en C peut être ou ne pas être définie, et ne reflète donc pas forcément l'état exact en VHDL du vecteur transmis. C'est pourquoi il est préférable de tester les valeurs de ces données avant de les convertir, afin de générer un message d'alerte pendant la co-simulation lorsque la conversion est susceptible d'engendrer des approximations.

Le tableau 3.1 résume les conversions de types implémentées dans l'outil CoSim.

VHDL		**C**	
Type	**Bits**	**Type**	**Taille**
BIT	1	unsigned char	8
STD_LOGIC	1 à 32	enum (int)	32
STD_ULOGIC	1 à 32	enum (int)	32
BIT_VECTOR	1 à 8	char, unsigned char	8
BIT_VECTOR	9 à 32	int, unsigned int	32
STD_LOGIC_VECTOR	1 à 8	char, unsigned char	8
STD_ULOGIC_VECTOR	9 à 32	int, unsigned int	32

TAB. 3.1: Règles de conversion de types VHDL-C

Nous avons observé que la plupart des applications embarquées développées en interne utilisent uniquement les types simples du langage C (`char`, `int`, `unsigned char`, ...) pour représenter les données transmises sur les ports d'interface, puisque ces derniers sont le plus souvent des registres de 8 ou 16 bits. Les types complexes comme les structures ne sont pas utilisés. C'est pourquoi la version actuelle de CoSim ne reconnaît que les types de 32 bits et moins. Toutefois si l'on désire transmettre des données de types plus complexes, il est tout à fait envisageable de regrouper plusieurs ports logiques afin d'obtenir une structure de plus de 32 bits.

Le code VHDL correspondant à la conversion de types peut prendre la forme présentée en figure 3.28. La donnée `DOUT` est en VHDL un vecteur de 8 signaux `STD_ULOGIC`. En C, la

```
C_DOUT <= SBVtoI (To_bitvector (DOUT));

DIN <= to_StdUlogicVector (ItoBV (C_DIN, 8));
```

FIG. 3.28: Exemple de code VHDL de conversion de types

variable correspondante est, selon le tableau de conversion donné plus haut, un caractère de 8 bits. La première ligne de VHDL convertit la donnée VHDL `DOUT` en un vecteur de bits, qui est lui-même converti en un entier 32 bits. Cet entier sera tronqué pour ne garder que les 8 bits de poids faible, lorsque sa valeur sera passée à l'application C. Le passage par le type intermédiaire `BIT_VECTOR` est nécessaire pour filtrer les valeurs indésirables, comme expliqué dans la section suivante.

La seconde ligne de code VHDL de la figure 3.28 correspond à la conversion de types de C vers VHDL. La variable entière `C_DIN`, venant du logiciel, est tout d'abord convertie en un `BIT_VECTOR` de 8 bits (là encore l'entier de 32 bits est tronqué), qui est lui-même converti en un `STD_ULOGIC_VECTOR` pour obtenir la donnée `DIN`.

Les types C correspondant aux types VHDL les plus courants (simples et énumérés) sont pré-définis dans un paquetage spécifique, `vhdl_types.h`, qu'il suffit d'inclure dans le code C applicatif pour assurer une communication fiable avec le modèle VHDL.

Filtrage

Le filtrage agit sur chacun des signaux venant du modèle VHDL et transmis au C [10]. Il consiste à lire un signal VHDL lorsque celui-ci est stable, par exemple sur un front donné de l'horloge [11].

La valeur ainsi obtenue peut être de deux types : soit c'est une valeur bien définie en C (0 ou 1), soit c'est une valeur de type particulier (haute impédance, non-définie,...). Dans le second cas, deux traitements sont possibles. La valeur particulière peut être directement transmise à l'application C en utilisant le type énuméré correspondant (pas de filtrage). Si au contraire les valeurs de types particuliers ne doivent pas être transmises au C, le filtrage permet de les supprimer. La figure 3.29 représente le filtrage d'un signal VHDL prenant alternativement une valeur bien définie en C (valeurs hexadecimales A et B) et une valeur non-définie (ZZ : haute impédance). Le filtrage consiste à ne pas transmettre la valeur ZZ,

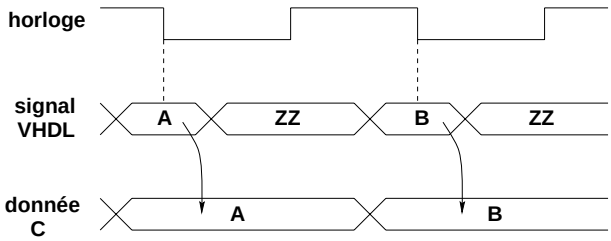

FIG. 3.29: Exemple de filtrage des signaux VHDL

pour obtenir une donnée C toujours correctement définie (valeurs A et B).

Ce filtrage est entièrement paramétrable par l'utilisateur, afin de s'adapter à tous les cas de figure. La solution par défaut consiste à filtrer systématiquement les signaux de données.

Un exemple de code de filtrage est donné en figure 3.30. La donnée VHDL `DOUT` est à la fois convertie et filtrée, grâce au passage par le type `BIT_VECTOR`. En effet la fonction `to_BITVECTOR` assure le filtrage de valeurs indésirables. De plus, la variable `C_DOUT` n'est affectée par la nouvelle valeur de `DOUT` que sur le front montant de l'horloge `CLK`, assurant alors que la donnée lue est dans un état stable (selon le comportement - connu - de l'interface-bus). De même, le signal `RESET` est transmis en synchronisation avec l'horloge.

[10] Le terme d'échantillonnage pourrait être utilisé pour définir cette action, mais il est réservé à l'opération inverse (C vers VHDL) décrite plus loin.

[11] Eventuellement d'autres signaux internes au VHDL, comme d'autres horloges déphasées, peuvent également être utilisés.

```
synchro_out : process
begin
     wait until CLK'event and CLK = '1';
     C_DOUT <= SBVtoI (to_BITVECTOR (DOUT));
     C_RESET <= RESET;
end process synchro_out;
```

FIG. 3.30: Exemple de code VHDL de filtrage

Échantillonnage

Au contraire du filtrage, l'échantillonnage traite les variables venant du C et converties en signaux VHDL. Ces variables, après avoir subi la conversion de types, doivent être positionnées sur le port d'interface à un moment précis, afin d'être stables lorsqu'elles seront lues par le modèle VHDL.

Là encore, l'horloge ou d'autres signaux internes sont utilisés pour définir le moment opportun. Sur la figure 3.31, la donnée venant de l'application C avec la valeur A n'est transmise au modèle VHDL que sur le front montant du signal d'horloge. Cette valeur

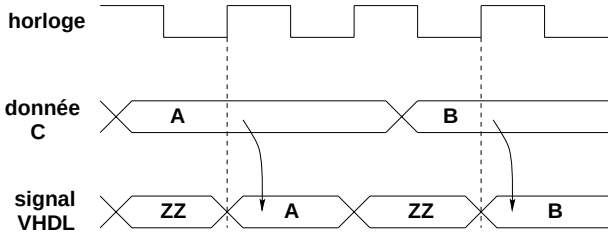

FIG. 3.31: Exemple d'échantillonnage des variables C

est maintenue pendant un temps bien défini (ici un cycle), puis relâchée. Elle est alors remplacée par une haute impédance, par exemple.

En d'autres termes, le bloc d'échantillonnage fournit un comportement temporel à l'application C, qui n'en a pas par nature. Il doit en fait simuler le comportement temporel de l'interface du futur processeur embarqué.

Un exemple de code VHDL d'échantillonnage est donné en figure 3.32. Les valeurs des

```
synchro_in : process
begin
     wait until CLK'event and CLK = '0';
     DIN <= C_DIN;
     CMD <= C_CMD;
end process synchro_in;
```

FIG. 3.32: Exemple de code VHDL d'échantillonnage

variables `C_DIN` et `C_CMD`, venant de l'application C, sont transmises aux données `DIN` et `CMD` en synchronisation avec l'horloge interne, ici sur le front descendant de `CLK`. Ce code assure que les données VHDL `DIN` et `CMD` sont stables sur le front montant de `CLK`, là où elles seront lues par l'interface-bus.

Génération de l'horloge de synchronisation

L'horloge de synchronisation est un signal périodique utilisé par le bloc de liaison C pour engager régulièrement des transactions avec l'application.

Dans le cas le plus simple, cette horloge peut tout simplement être l'horloge interne du circuit. Nous obtenons alors une précision de simulation optimale en termes de cycles, puisqu'une transaction est engagée à chaque cycle. L'inconvénient est le grand nombre d'échanges engendrés.

Pour réduire le nombre des transactions, nous pouvons exploiter les caractéristiques de l'application simulée. Lorsque les échanges entre le logiciel et le matériel ne sont pas fréquents, il n'est pas nécessaire d'effectuer des transactions à chaque cycle. Nous pouvons alors réduire la fréquence des échanges en divisant la fréquence de l'horloge de synchronisation. La contrepartie est que chaque opération d'entrée-sortie semble durer plusieurs cycles : lorsqu'il y a des échanges en rafales, les performances peuvent être dégradées. De plus, ce retard engendre un fonctionnement différent de celui obtenu en simulation. Ce mode accéléré ne doit donc être utilisé qu'après avoir validé l'ensemble avec une fréquence de synchronisation élevée (l'horloge interne), et dans le but de poursuivre la simulation plus loin dans le temps simulé.

Une dernière possibilité est de modifier, pendant la simulation, la fréquence de l'horloge de synchronisation. Lorsqu'il y a beaucoup de communication (initialisation, transfert de données en rafale, envoi de commandes,...), l'horloge de synchronisation peut être réglée sur l'horloge interne, donc à la fréquence maximale. Puis lorsque l'application entre dans une partie de traitement interne, sans interaction avec l'extérieur, l'horloge de synchronisation peut être considérablement réduite, voire stoppée, jusqu'à l'envoi d'une nouvelle commande. Ce mécanisme évolué a été appliqué avec succès pour effectuer des simulations globales du système IVT, décrit plus loin.

3.5.3 Intégration aux flots de conception existants

La rapidité de configuration de l'outil CoSim pour un flot de conception conjointe existant est déterminante. En raison de la complexité des environnements de conception, matériels notamment, il est indispensable de limiter le nombre de manipulations nécessaires à la configuration du modèle VHDL. Du côté logiciel, le code applicatif ne doit pas requérir de modifications profondes, afin de respecter les exigences du flot de compilation (optimisations mais aussi limitations).

Flot de conception du processeur

Le flot de conception matériel du système applicatif peut se décomposer en plusieurs tâches, éventuellement concurrentes, où chaque bloc du système est conçu indépendamment. Les spécifications d'un bloc comprennent la description physique et logique de l'interface, et le comportement du bloc par rapport aux stimuli extérieurs à celui-ci (commandes, signaux d'entrée, interruptions...).

Le principe de base de la co-simulation C-VHDL est de co-simuler le logiciel embarqué dans un processeur sans requérir aucune description de ce processeur, pas même comportementale. Seule l'interface doit être connue, afin de connecter correctement les deux parties (le logiciel et le reste du système matériel). L'intégration de la co-simulation consiste donc à configurer le modèle VHDL du système complet de telle sorte que l'entité du processeur embarqué demeure intacte, alors que son architecture est modélisée par le logiciel en C. La figure 3.33a représente le couple entité-architecture de la configuration finale, comprenant le modèle RTL du processeur. La figure 3.33b représente le couple entité-architecture de la configuration co-simulation. L'entité demeure inchangée, alors que l'architecture RTL est remplacée par une connexion vers le logiciel C.

Le code de cette connexion dépend du vendeur du simulateur VHDL, car il utilise des attributs d'extension au langage VHDL standard. Un exemple est donné en figure 3.34,

```
entity PROC is
port(
    CLK    : in  BIT;
    CMD    : in  BIT;
    DIN    : in  INTEGER;
    DOUT   : out INTEGER;
    RESET  : in  STD_ULOGIC
);
end PROC;

architecture RTL of PROC is
begin
.
.
-- RTL model of the processor
.
.
end RTL;
```

```
entity PROC is
port(
    CLK    : in  BIT;
    CMD    : in  BIT;
    DIN    : in  INTEGER;
    DOUT   : out INTEGER;
    RESET  : in  STD_ULOGIC
);
end PROC;

architecture COSIM of PROC is
.
.
-- Connection with C code
.
.
begin
end COSIM;
```

FIG. 3.33: Configurations VHDL d'un processeur embarqué

correspondant au simulateur VSS de Synopsys. L'interface C de ce simulateur est appelée CLI (*C Language Interface*).

```
architecture COSIM of PROC is
  attribute FOREIGN of COSIM:
            architecture is "SYNOPSYS:CLI";

  attribute CLI_PIN of CLK:
         signal is CLI_EVENT;

  attribute CLI_PIN of DIN:
         signal is CLI_PASSIVE;
  attribute CLI_PIN of CMD:
         signal is CLI_PASSIVE;
  attribute CLI_PIN of RESET:
         signal is CLI_PASSIVE;
begin
end COSIM;
```

FIG. 3.34: Code VHDL de connexion C (Synopsys CLI)

La principale fonction de ce code est d'indiquer au simulateur VHDL le nom des signaux qui déclencheront l'appel de la fonction de transaction, et donc la communication avec l'application C. Pour la co-simulation C-VHDL seul le signal d'horloge `CLK` doit être pris en compte. C'est pourquoi l'attribut `CLI_EVENT` est associé à `CLK`, alors que tous les autres signaux en entrée de l'interface (`IN` ou `INOUT`) reçoivent l'attribut `CLI_PASSIVE`. Le signal `CLK` représente l'horloge de synchronisation utilisée pour engager les transactions avec l'application C. Dans le cas où l'horloge de synchronisation ne correspond pas à l'horloge interne du circuit (afin de réduire le nombre de transactions), un nouveau signal appelé par exemple `CLK_SYNC` peut être utilisé afin de conserver le signal `CLK` original dans la déclaration de l'interface.

La figure 3.35 contient le code VHDL correspondant au simulateur Leapfrog de Cadence, dont l'interface C est appelée FMI (*Foreign Model Interface*).

La fonctionnalité est exactement la même que pour l'interface de Synopsys décrite précédemment, seule la syntaxe change quelque peu. Les attributs `COSIM_SENSITIVITY`, `COSIM_ACTIVE` et `COSIM_PASSIVE` sont pré-définis par l'outil CoSim afin de garantir un fonctionnement identique et une syntaxe similaire quel que soit le simulateur utilisé.

```
architecture COSIM of PROC is
  attribute FOREIGN of COSIM:
           architecture is "Clib:PROC";

  attribute COSIM_SENSITIVITY of CLK:
         signal is COSIM_ACTIVE;

  attribute COSIM_SENSITIVITY of DIN:
         signal is COSIM_PASSIVE;
  attribute COSIM_SENSITIVITY of CMD:
         signal is COSIM_PASSIVE;
  attribute COSIM_SENSITIVITY of RESET:
         signal is COSIM_PASSIVE;

begin
end COSIM;
```

FIG. 3.35: Code VHDL de connexion C (Cadence FMI)

La déclaration de l'architecture COSIM est la seule manipulation à effectuer sur les sources VHDL du système. Ainsi il est aisé de passer d'une configuration à l'autre, ou de modifier une configuration pour co-simuler plusieurs processeurs embarqués successivement.

Flot de conception logiciel

Le flot de conception du logiciel embarqué est exposé en détail dans le chapitre sur la compilation. Pour résumer, le code C est compilé par un compilateur multi-cible pour obtenir un code machine utilisant les instructions du processeur embarqué. Ce même code C est utilisé pour la co-simulation, afin de valider sa fonctionnalité dans l'environnement matériel réel. Il est donc impératif de minimiser les changement apportés à ce code pour effectuer la co-simulation.

C'est pourquoi un nombre limité de conventions sur l'accès aux ports d'entrée-sortie ont été établies, formant une API *(Application Programming Interface* ou Interface de programmation applicative) réduite (figure 3.36). Tous les ports d'entrée-sortie sont représentés par

```
/* PROC.h */

CHAR C_DIN;
CHAR C_DOUT;
STD_ULOGIC_VECTOR C_CMD;
CHAR C_RESET;

void PROC_Transaction (void);
```

FIG. 3.36: Interface de programmation applicative (API)

des variables globales, dont le nom est le même que celui trouvé dans l'interface de l'entité VHDL, préfixés par `C_` et en majuscules. Le type de ces variables dépend du type des signaux VHDL correspondant (caractères, entiers ou types énumérés). Ces variables sont automatiquement déclarées dans un fichier d'en-tête généré par l'outil. Ce fichier porte le nom de l'entité VHDL du processeur co-simulé, en majuscules (ici `PROC.h`). De plus, ce fichier d'en-tête contient le prototype de la fonction de transaction qui servira à l'application à communiquer avec le système VHDL.

Cette interface, très simple, suffit à mettre en pratique la co-simulation. L'application peut mettre à jour ou lire les variables globales (pour respectivement écrire ou lire un port

de l'interface), et engager une transaction par l'appel de la fonction `PROC_Transaction()`. Toutefois, l'application de cette technique à un flot de conception logiciel nouveau, à savoir la compilation systématique du source C, met en évidence une contrainte supplémentaire. Le code source de l'application qui est utilisé pour la co-simulation doit être le plus proche possible du code utilisé pour la compilation. Or dans ce dernier code, les opérations d'entrée-sortie sont représentées par des opérations assembleur du style "`WRITE Port, Valeur`" et " `READ Accu, Port`". Traduites en C, ces opérations prennent la forme "`WRITE(Port, Valeur) ;`" et "`a = Read(Port) ;`", typiquement. Il est donc souhaitable que le code destiné à la compilation respecte exactement ce format. Ainsi, le seul appel des fonctions `Read()` et `Write()` peut alternativement être associé à la compilation et à la co-simulation. Un simple changement de fichier d'en-tête (*include*) permet de choisir l'implantation. Le code applicatif ne subit alors aucune modification textuelle.

Cette technique est implémentée par le code proposé en figure 3.37.

```
#include "vhdl_types.h"
#include "PROC.h"

int io_read (cmd)
int cmd;
{
    switch(cmd) {
      case STATUS:
         C_CMD = STATUS_FLAG;
         io_transaction();
      break;
      .
      .
    }
return (C_DIN);
}

void io_write (cmd, value)
int cmd, value;
{
    switch(cmd) {
      case DMA:
         C_CMD = DMA_FLAG;
         C_DOUT = value;
         io_transaction();
      break;
      .
      .
    }
}

void io_transaction() {
   PROC_Transaction();
   if(C_RESET == 1) exit (22);
}
```

FIG. 3.37: Interface d'appel des fonctions d'entrée-sortie

La fonction `io_read()` permet de lire une donnée venant du VHDL, équivalente à l'opération assembleur "`READ A, Port`". Selon la commande venant de l'application (`cmd`), une transaction est engagée à travers l'interface, avec le signal de commande `C_CMD` propre au protocole de l'interface-bus. La fonction `io_transaction()` se contente d'appeler la

fonction `PROC_Transaction()` de l'API, en y ajoutant l'interprétation du signal RESET - et éventuellement d'autres interruptions. Ici un état haut du signal RESET engendre la terminaison immédiate du programme, et sa relance par le shell du niveau supérieur. Au retour de l'appel de `PROC_Transaction()`, la variable globale `C_DIN` est mise à jour. Après avoir appelé `io_transaction()`, la fonction `io_read()` retourne la nouvelle valeur obtenue par `C_DIN`.

La fonction `io_write()` agit de manière comparable, mais positionne les valeurs à écrire (ici `C_CMD` et `C_DOUT`) avant d'appeler `io_transaction()`, de façon à faire une mise-à-jour. Le signal RESET et les interruptions sont à nouveau testés dans `io_transaction()`.

Ce code d'interface, relativement simple, peut être réutilisé d'un processeur à l'autre. Seuls les noms des signaux et les commandes sont susceptibles de changer. Son utilisation garantit de conserver un code applicatif indépendant de la co-simulation, respectant totalement le flot de conception logiciel existant.

3.6 Expérimentations industrielles

Les expérimentations industrielles de l'outil CoSim ont été menées sur plusieurs projets, à chaque fois dans un but bien précis. Le premier projet, Calc, est un modèle simplifié de circuit multi-processeur. Il a servi a valider les fonctionnalités de base, ainsi qu'à exposer les principes de la co-simulation au travers d'un tutorial. Le deuxième projet est un circuit multimedia, l'IVT, conçu en R&D et destiné à être industrialisé prochainement. Son ampleur (plusieurs millions de transistors) et sa complexité (jusqu'à cinq processeurs embarqués) permettent de s'assurer que l'outil CoSim s'intègre facilement dans un flot existant et complexe.

3.6.1 Circuit de test Calc

Description

Le circuit de test Calc a été spécialement conçu pour valider les fonctionnalités de l'outil CoSim, et pour servir de tutorial à de nouveaux utilisateurs. Sa conception est volontairement simplifiée, mais présente néanmoins les caractéristiques essentielles des circuits susceptibles de requérir l'utilisation de CoSim. La figure 3.38 décrit l'architecture de Calc. Il est composé de trois blocs principaux : le contrôleur, l'additionneur et le multiplieur. L'additionneur et le multiplieur ne sont pas des blocs matériels câblés, mais représentent deux processeurs. Ainsi la fonctionnalité exacte de chacun d'eux n'est pas fixée matériellement mais dépend de l'application logicielle qui tourne dessus.

Chaque processeur communique avec le contrôleur via des bus de contrôle et de données, en utilisant un protocole de communication de type poignée-de-main (*handshake*). Le contrôleur ordonne l'exécution d'une opération en fournissant les données et une commande. Cette dernière fait référence à une opération particulière implémentée dans le logiciel du processeur correspondant. Les types de données utilisés aussi bien dans les modèles matériels (VHDL) que dans les modèles C sont suffisamment variés pour valider l'implantation de la plupart des types reconnus par CoSim.

Les expérimentations menées sur le circuit Calc portent essentiellement sur la validation des fonctionnalités de CoSim. Trois fonctionnalités ont été particulièrement étudiées :

1. la co-simulation de plusieurs processeurs embarqués en même temps,
2. la gestion des interruptions, modélisées par le RESET, pendant une session de déboguage interactif,

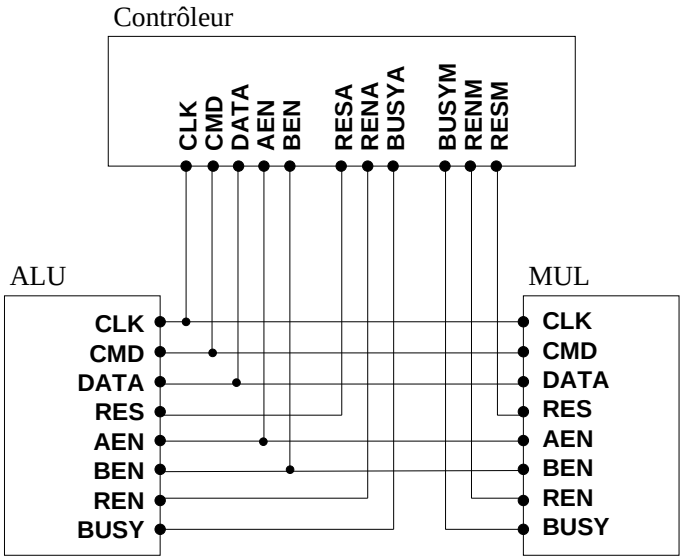

FIG. 3.38: Architecture du circuit de test Calc

3. l'intégration d'un même circuit dans deux flots de conception matérielle différents (Cadence et Synopsys).

Cosimulation multi-processeurs

La co-simulation de plusieurs processeurs en même temps demande une isolation parfaite des contextes d'exécution des logiciels embarqués.

La communication entre le logiciel et le matériel est d'ores-et-déjà isolée par construction. En effet les identificateurs de files de messages sont uniques, et associés à un logiciel donné. Chaque voie de communication est synchronisée indépendamment des autres, grâce à la définition d'une horloge de synchronisation dans chaque bloc de connexion. Dans le cas le plus simple, ces horloges sont identiques à l'horloge interne, mais ce n'est pas imposé. De même, les programmes applicatifs tournent indépendamment les uns des autres, puisqu'ils s'exécutent sur des processus différents.

Le risque de chevauchement des contextes apparaît au niveau de l'interface C du simulateur VHDL. En effet, l'implémentation des interfaces C dans les simulateurs VHDL utilisés (VSS et Cadence) est assurée par une librairie, statique ou dynamique, commune à tous les processeurs. La table des symboles, qui contient les noms des fonctions et des variables globales de la librairie, est donc partagée entre tous les processeurs. La conséquence immédiate est que deux processeurs ne peuvent posséder le même nom de modèle, puisque ce nom est utilisé pour identifier l'interface et les ports associés. Cette limitation est gênante lorsqu'un système instancie plusieurs processeurs d'un même modèle. La solution consisterait à ajouter un niveau d'indirection supplémentaire dans la génération du code de co-simulation. Bien que cela soit tout à fait envisageable, la version actuelle de CoSim ne la met pas en œuvre, et possède donc cette limitation sur l'unicité des noms de processeurs embarqués.

Interruption matérielle du logiciel

L'interruption de l'exécution du logiciel par un signal matériel (appelé lui-même interruption) est implémentée dans CoSim d'une manière entièrement générique. L'utilisateur indique les signaux devant être traités comme une interruption, ainsi que le front ou l'é-

tat déclenchant, et enfin la fonction à appeler dans ce cas. Une application extrême du mécanisme d'interruption est le signal RESET (ré-initialisation), qui doit entraîner l'arrêt immédiat de l'exécution du logiciel et sa relance.

En pratique, l'arrêt du programme est très simple à réaliser : il suffit d'appeler la fonction C `exit()` dès que le signal RESET est actif. La relance du programme ne peut être faite qu'à un niveau système supérieur. Pour ce faire, un programme shell[12] exécute une boucle qui exécute le programme C indéfiniment. Lorsque celui-ci est stoppé, la boucle refait une itération et le relance. L'arrêt complet de la co-simulation est assuré par la terminaison du shell lui-même, exclusivement.

Cette technique est efficace lorsque le programme C lui-même est appelé par le shell. Par contre, une session de déboguage fait intervenir un programme débogueur, qui lui-même exécute le programme C. La relance automatique du programme C par un shell n'est plus possible. Lorsque le signal RESET devient actif, le programme C est stoppé et le contrôle est rendu au débogueur. L'utilisateur doit alors manuellement relancer le programme, en prenant garde à respecter le délai d'attente (*timeout*) de la couche de communication, puisque le simulateur VHDL est toujours actif. Si cela est envisageable lorsqu'il y a un processeur dans le système, cela devient très pénible si l'on co-simule plusieurs processeurs. Afin de s'affranchir de cette limitation, dans certains cas inacceptable, le débogueur doit être modifié pour jouer le rôle du shell précédemment décrit, à savoir relancer systématiquement le programme C lorsqu'il se termine. Cette modification a été apportée au débogueur utilisé dans le cadre du développement du circuit IVT (xxgdb de GNU), lequel est disponible dans la distribution de CoSim. Le portage de cette modification n'est pas planifié pour d'autres débogueurs.

Intégration aux flots de conception Cadence et Synopsys

La coexistence de deux flots de conception VHDL différents, observée à SGS-THOMSON, conduit à rendre l'outil CoSim compatible avec les deux principaux fournisseurs de simulateur VHDL, Synopsys et Cadence. De plus, il est souhaitable que les descriptions utilisateur (modèles VHDL et C) contiennent le minimum de dépendances avec le vendeur du simulateur, et que celles-ci soient localisées clairement.

Dans le modèle C (du logiciel embarqué), aucune référence n'est faite au modèle de simulateur utilisé. Par contre les types des données en C, correspondant aux ports d'interface, dépendent précisément des types VHDL. C'est pourquoi il est indispensable que le sous-ensemble de types VHDL utilisés pour les ports d'interface soient communs aux deux modèles de simulateurs, et compatibles. Un sous-ensemble défini aussi bien en VHDL pour Synopsys, en VHDL pour Cadence, et en C, est proposé par défaut dans CoSim. Des extensions à ce sous-ensemble sont possibles, mais la compatibilité entre les simulateurs n'est alors plus assurée.

Dans le modèle VHDL (du système), la seule dépendance envers le type de simulateur apparaît dans la déclaration de l'architecture du processeur embarqué. Cette architecture fait appel à l'interface C du simulateur, dont la syntaxe est spécifique. Les figures 3.34 et 3.35 de la section précédente représentent deux syntaxes différentes selon le simulateur VHDL utilisé (Synopsys et Cadence respectivement). Grâce aux attributs `COSIM_SENSITIVITY`, `COSIM_ACTIVE` et `COSIM_PASSIVE` prédéfinis par CoSim, la re-configuration d'un simulateur à l'autre est immédiate. Lorsque le modèle VHDL du système est stocké dans une base de données de type CVS ou SCCS, seuls deux fichiers contenant les deux architectures

[12] Un programme shell permet de programmer l'execution de commandes habituellement tapées interactivement par l'utilisateur.

possibles sont maintenus. Un simple changement de nom suffit à choisir l'une ou l'autre des configurations.

3.6.2 Circuit IVT

Description

L'architecture du circuit IVT est détaillée dans le chapitre 2. Les expérimentations conduites avec ce circuit visent essentiellement à :

1. valider l'outil CoSim sur un circuit très complexe, faisant appel à des fonctionnalités avancées telles que le mode de synchronisation 'asynchrone',
2. mesurer le temps de co-simulation et le comparer au temps de simulation RTL,
3. mesurer le surcoût de temps d'exécution dû au mécanisme de communication IPC.

Mode de synchronisation évolué

Le choix de la fréquence de synchronisation dans le bloc de connexion détermine la vitesse de co-simulation du système global. La validation globale d'un circuit de la complexité de l'IVT exige d'atteindre des temps simulés de plusieurs secondes, ce qui se traduit par des temps de simulation de plusieurs heures. Il est donc primordial d'optimiser la fréquence des échanges issus de la co-simulation. Une variante du mode de synchronisation par l'horloge a été développée et appliquée a l'IVT. Ce nouveau mode, appelé abusivement asynchrone, régule le rythme des transactions selon n'importe-quel signal VHDL, et non pas exclusivement l'horloge de synchronisation. Ainsi un ou plusieurs signaux du système, soigneusement choisis, engendrent une transaction lorsque cela est nécessaire. Ces signaux sont choisis en analysant le comportement de l'interface-bus du processeur, et le comportement de l'algorithme tournant sur ce processeur.

Parmi les processeurs de l'IVT, le VIP est celui sur lequel tournent les algorithmes nécessitant des temps simulés les plus éloignés. En effet la prédiction de mouvements requiert le traitement de plusieurs images successives. Les échanges entre ce processeur et le reste du circuit respectent un certain protocole. En simplifiant, le protocole fonctionne par l'envoi de commandes du MSQ vers le VIP, qui exécute la tâche correspondante et signifie la terminaison de celle-ci au MSQ. Le système matériel peut donc anticiper les échanges réellement nécessaires, en excluant les échanges inutiles. A partir des valeurs de plusieurs signaux de contrôle appartenant au MSQ, au bus du système, voire même à d'autres blocs du circuit, il est possible de générer un signal passant actif lorsqu'un échange avec le VIP est nécessaire. Ce nouveau signal de synchronisation, qui n'a pas d'existence fonctionnelle en dehors de la co-simulation, se substitue à l'horloge de synchronisation utilisée par défaut.

L'utilisateur indique simplement le nom du ou des signaux de sensibilité dans le code VHDL de connexion C (figures 3.34 et 3.35) en leur donnant un attribut `CLI_ACTIVE` ou `COSIM_ACTIVE`, selon le simulateur. La génération du signal de synchronisation doit être faite soigneusement, afin d'éviter de bloquer la co-simulation si certains messages importants sont transmis au mauvais moment. L'application de ce mode synchronisation a la co-simulation du VIP a permi d'atteindre des temps simulés nettement plus éloignés qu'avec le mode de synchronisation "synchrone", basé sur un signal périodique. Le seul inconvénient de ce mode évolué est qu'il requiert une connaissance accrue du protocole du système, et impose de faire des hypothèses sur le fonctionnement global. De ces hypothèses dépend la validité de la co-simulation. Le mode synchrone, au contraire, assure un fonctionnement similaire à la simulation, et garantit donc une validation fonctionnelle complète.

Comparaison des temps de co-simulation

La co-simulation à haut-niveau d'un logiciel embarqué permet de simuler fonctionnellement un système sans disposer de modèles (comportementaux ou autres) du processeur en question. A la place, nous exécutons seulement le programme C, directement sur la station de travail. Nous nous attendons donc à ce que la vitesse de co-simulation soit nettement plus élevée. Cependant le code C nécessaire à la co-simulation (communication inter-processus, interface C du simulateur VHDL, conversions) est exécuté à chaque transaction, et engendre donc un surcoût qui n'est peut-être pas négligeable. Afin d'évaluer précisément l'impact de ce code supplémentaire sur le temps de (co)simulation globale du système, plusieurs mesures sont effectuées.

Le circuit IVT est constitué de trois processeurs embarqués (le MSQ, le BSP et le VIP). Afin de comparer le temps nécessaire à une co-simulation C-VHDL avec le temps nécessaire à une simulation VHDL pure nous devons disposer à la fois d'un modèle VHDL et de son équivalent en C, pour une fonctionnalité donnée. A l'époque de cette étude, seul l'algorithme VLD (*Variable Length Decoding* ou Décodage à Longueur Variable) est disponible aussi bien sous la forme d'opérateur VHDL (RTL) que sous la forme d'un logiciel en C. Cet algorithme est destiné à être exécuté sur le processeur BSP, dédié au traitement du flux de bits issu de la ligne téléphonique. Sa taille est suffisamment grande pour qu'il soit représentatif d'une application de traitement intensif, supervisée par un contrôleur externe (ici le processeur MSQ).

L'expérience consiste donc à mesurer le temps vu par l'utilisateur (avec un chronomètre) pour effectuer le traitement d'un train de bits donné, avec l'une et l'autre des configurations précédemment décrites. La station de travail est évidemment isolée du réseau, de sorte qu'aucune autre application n'occupe les ressources internes.

L'expérience a été menée à plusieurs reprises, afin de s'affranchir des fluctuations dûes au fonctionnement du système Unix. La figure 3.39 représente la moyenne des temps de simulation obtenus. La co-simulation de l'algorithme en C avec le reste du système est donc

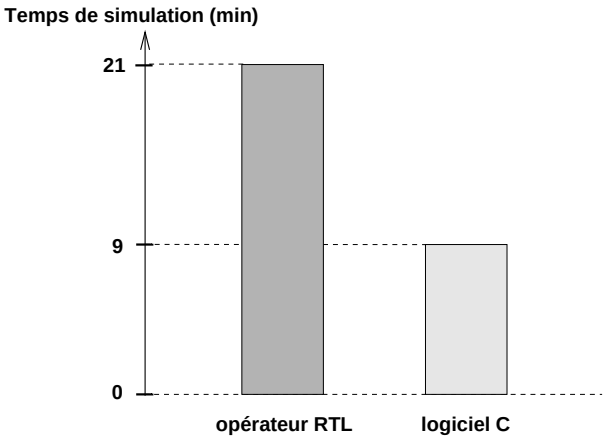

FIG. 3.39: Comparaison des temps de simulation du VLD

plus de deux fois plus rapide que la simulation VHDL de l'opérateur RTL correspondant. Ce résultat confirme donc que la substitution du modèle RTL d'un opérateur câblé par son modèle logiciel en C permet de gagner un temps considérable sur la simulation globale du système, malgré le surcoût engendré par la communication entre le C et le VHDL. Toutefois cette simulation globale par co-simulation C-VHDL souffre d'une perte de précision, notamment temporelle, par rapport à la simulation VHDL pure (uniquement pour les opérateurs co-simulés). Elle doit être réservée à la validation strictement fonctionnelle

de l'ensemble, dans le but d'atteindre des temps simulés nettement plus avancés qu'avec la simulation purement VHDL.

Le modèle VHDL utilisé dans cette expérience est un modèle câblé de l'opérateur VLD, et non pas un modèle VHDL du processeur BSP sur lequel tournerait le logiciel en langage machine. Nous pouvons supposer que la simulation du processeur est encore plus longue que la simulation de l'opérateur câblé, ne serait-ce qu'en considérant la partie contrôle. Cette dernière est bien plus complexe dans le processeur où tout le jeu d'instructions doit être décodé, alors que le contrôleur de l'opérateur contient exactement le nombre d'états nécessaires à l'algorithme VLD. Il est donc probable que la différence de temps d'exécution entre la co-simulation C-VHDL et la simulation du processeur BSP soit encore plus flagrante. La série de mesures comparatives effectuée sur un autre processeur embarqué dans l'IVT (le VIP) pour lequel nous disposons du modèle RTL, confirme cette hypothèse comme cela est exposé dans la section suivante.

Influence de la complexité algorithmique sur le temps de simulation

Parmi tous les processeurs embarqués dans le circuit IVT, seul le VIP est destiné à exécuter plusieurs algorithmes différents, selon les choix de codage disponibles. De plus, ces algorithmes contiennent aussi bien du code de contrôle que du code de traitement d'image intensif. Le VIP est donc un bon candidat pour effectuer des mesures sur un ensemble d'algorithmes suffisamment large, tout en gardant un environnement invariant. De plus, un modèle VHDL RTL du processeur lui-même est disponible, permettant de comparer cette configuration avec la co-simulation du même logiciel avec le reste du système.

Les différents algorithmes correspondent aux divers modes de fonctionnement du protocole de prédiction de mouvements. Le premier mode, appelé H. 261, correspond à une ancienne norme du même nom, dont l'algorithme est relativement simple. Les modes suivants (*Unrestricted Motion Vector*, *Advanced Prediction*, *PB Frame*, *PB Frame with Unrestricted Motion Vector*, et *PB Frame with Advanced Prediction*) correspondent à la nouvelle norme H263, et sont classés par ordre de complexité algorithmique croissante.

Le diagramme présenté en figure 3.40 résume les temps d'exécution correspondant aux deux configurations (co-simulation C-VHDL et simulation VHDL pure - avec le modèle du processeur). Deux observations peuvent être faites au vu de ces mesures. En premier lieu

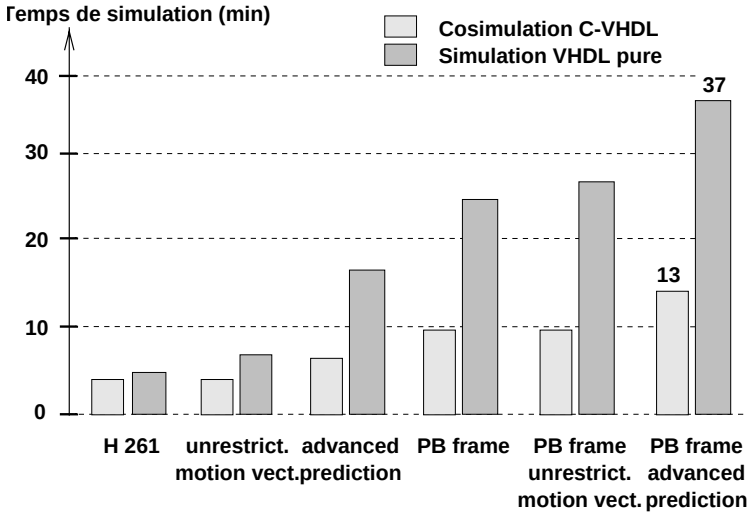

FIG. 3.40: Comparaison des temps de simulation du VIP

la co-simulation C-VHDL, en gris clair, est toujours plus rapide que la simulation RTL, en

gris foncé. Cela confirme les résultats obtenus par l'expérience précédente du BSP.

De plus, cette différence s'accroît avec la complexité de l'algorithme. En effet, l'algorithme le plus simple en termes de calcul interne (H261) nécessite sensiblement le même temps de simulation dans les deux configurations. Au contraire, la co-simulation C-VHDL du mode le plus complexe (*PB Frame with Advanced Prediction*) est jusqu'à **2,8** fois plus rapide que la simulation VHDL pure. Ceci s'explique par le fait que dans ce mode, la plus grande partie du temps d'exécution est consommée par du calcul interne, rendant négligeable le temps passé à échanger les commandes et données nécessaires à son exécution. Il n'est pas surprenant de vérifier que les phases de calcul interne au processeur sont exécutées beaucoup rapidement en C qu'en simulation VHDL RTL.

3.7 Travaux futurs - Conclusion

3.7.1 Travaux futurs

Annotation temporelle

La co-simulation C-VHDL telle qu'elle a été implémentée dans la version actuelle de CoSim (1.0) est purement fonctionnelle. Aucun aspect temporel n'est géré, notamment en ce qui concerne l'application C. Afin d'aboutir à une co-simulation précise au niveau de l'instruction, le code C pourrait par exemple être annoté après compilation du code C (*back-annotation*) [104]. En effet, le code binaire produit par le compilateur peut être lu et relié au code C original, pour associer à chaque instruction C le nombre d'instructions réelles correspondant, sur le processeur cible [81][55]. Il est également envisageable d'utiliser le résultat d'outils d'estimation, comme par exemple dans [77]. Le réalisme temporel est alors grandement amélioré, ne serait-ce que du point de vue des opérateurs matériels du système, modélisés en VHDL [117]. De plus, l'exploration au niveau des systèmes des temps de traitement relatifs (entre les blocs) serait ainsi aidée sans requérir le développement d'un modèle de simulation spécifique.

Cependant, la nécessité de disposer du compilateur pour effectuer une telle annotation peut remettre en cause l'utilité de la co-simulation C-VHDL, qui est, à l'origine, de valider un système mixte C et VHDL très tôt dans le flot de conception. Si un compilateur est disponible, le code binaire obtenu peut être simulé directement par un simulateur de jeu d'instructions pour peu que ce dernier soit rapidement produit. Des outils génériques, rapidement reconfigurables, existent et fournissent une précision temporelle bien supérieure à l'annotation du code C par le nombre d'instructions assembleur. En effet, un tel simulateur peut atteindre une précision au cycle près, simulant correctement les effets de pipeline et de cache, notamment.

Il demeure que le temps de simulation avec un simulateur de jeu d'instructions est probablement nettement plus élevé qu'avec une co-simulation C-VHDL, même annotée. Une vitesse de 1 million d'instructions simulées par seconde semble être un ordre de grandeur maximum pour un simulateur de jeu d'instructions écrit en C. Une station de travail atteint généralement une à plusieurs centaines de millions d'instructions par seconde, ce qui représente deux ordres de grandeur de plus. Même en considérant l'ajout d'instructions dûes à la communication avec le simulateur VHDL, il demeure une différence notable. La co-simulation C-VHDL reste donc une alternative performante pour la simulation mixte au niveau système. L'annotation temporelle, appliquée à l'exploration (grossière) des performances à ce niveau, demeure un bénéfice intéressant.

Cosimulation multi-langage

La co-simulation C-VHDL permet de simuler un système mixte (logiciel/matériel) modélisé dans les deux langages les plus généralement utilisés dans le domaine (C et VHDL). Cependant, d'autres langages équivalents existent, comme le Verilog pour le matériel, ou le Fortran, Pascal, C++, Java pour le logiciel. L'implémentation d'une co-simulation intégrant ces langages alternatifs pourrait aider à élargir le domaine d'application.

Par ailleurs, de nouveaux langages existent (ou émergent), modélisant plus efficacement certains composants d'un système ou même le système lui-même. Par exemple, le langage Matlab est conçu pour modéliser finement un composant en continu (par opposition aux modélisations discrètes généralement employées en conception numérique). L'intégration d'un composant qui requiert ce type de simulation dans un système nécessite alors une co-simulation Matlab-C-VHDL, par exemple. De même, un langage tel que Silage est couramment utilisé pour modéliser un algorithme de traitement du signal (orienté flot de données). SDL est également reconnu dans la modélisation de la communication entre les composants d'un système.

3.7.2 Conclusion

Ce chapitre a présenté la conception, le développement et l'application d'un outil de co-simulation C-VHDL. Son utilisation pour la validation d'un système intégré complexe a mis en évidence l'apport non négligeable, en termes de temps de développement tout particulièrement, de la simulation conjointe à un niveau intermédiaire entre le niveau système et le niveau de réalisation (VHDL-RTL et assembleur).

La possibilité nouvelle de simuler l'ensemble des logiciels embarqués conjointement au système matériel complet sur un intervalle de temps très étendu a permi de mettre au point très tôt certains aspects de communication système (contrôle et séquencement des opérateurs), avant même la réalisation de la plupart des processeurs.

L'intégration de l'outil CoSim au flot de conception actuel, apportant un gain de productivité sans remettre en cause l'environnement existant, a garanti une évolution douce vers la conception au niveau système au rythme des besoins exprimés par les concepteur de systèmes mono-puces complexes. Notamment, l'utilisation des descriptions matérielles et logicielles existantes (VHDL et C) a évité le développement et la validation de modèles spécifiques à la co-simulation haut-niveau, réduisant ainsi le temps d'implémentation d'un tel environnement.

Chapitre 4

Exploration d'architectures

Ce chapitre présente une étude d'exploration architecturale de processeurs basée sur un ensemble restreint de critères ayant un impact sur la taille du code programme. L'application à l'architecture expérimentale D960 est détaillée.

Pour des applications multimédias ou de téléphonie mobile (GSM, DECT), le logiciel embarqué est suffisamment complexe pour que la mémoire programme intégrée (ROM) occupe une grande partie de la surface du circuit, rendant la surface du coeur minoritaire [109][110]. Ce phénomène est amplifié par l'ajout de caches internes devenus indispensables pour que le flot de données puisse suivre la cadence interne du processeur [108]. Pour réduire le coût global d'un système mono-puce, il est donc primordial de minimiser la taille du code programme embarqué. Ce chapitre expose un certain nombre de techniques généralement employées pour réduire la taille du code programme embarqué. Certaines d'entre-elles font l'objet d'une exploration exhaustive afin de quantifier précisément leur impact sur la taille de code. Elles concernent notamment la configuration des bancs de registres et l'encodage des champs immédiats. L'utilisation de l'environnement FlexPlore permet d'automatiser ces explorations. En outre, la reconfiguration automatique de la chaîne de compilation reciblable Flexcc permet d'étudier l'interaction entre les choix architecturaux et les algorithmes de compilation. Les résultats de ces explorations sont analysés, puis une ébauche d'architecture alternative, plus performante en taille de code, est proposée.

4.1 Motivations et objectifs

4.1.1 Motivations

Trois motivations nous poussent à nous intéresser à l'exploration d'architectures dédiées basée sur l'analyse du code compilé. En premier lieu, le développement de systèmes mono-puces à base de processeurs embarqués fait apparaître que la surface de tels circuits est majoritairement occupée par de la mémoire, et plus spécialement par la mémoire programme. La taille du code embarqué est donc particulièrement critique. Ensuite, le raffinement de l'architecture d'un processeur embarqué nécessite d'étudier précisément les performances obtenues selon le logiciel, c'est-à-dire son adaptation à la classe d'application visée [89]. Enfin l'utilisation de compilateurs pour la majeure partie du code applicatif rend délicate l'analyse manuelle du code machine produit. L'optimisation de celui-ci nécessite de bien comprendre le comportement du compilateur, en lui-même et selon l'architecture.

Prépondérance de la taille mémoire du logiciel embarqué

La conception d'une architecture dédiée passe par l'élaboration d'un jeu d'instructions qui doit être adapté aux ressources matérielles (pour être en mesure d'en exploiter les capacités), mais également adapté aux applications qui tourneront sur l'architecture. Plusieurs critères d'optimisation peuvent être pris en compte dans la définition d'un jeu d'instructions. Généralement, les critères identifiés comme prépondérants comprennent les suivants :

- taille du code programme (ou densité de code),
- performance (temps d'exécution des parties critiques),
- consommation en énergie.

Bien que ces trois critères soient tous importants [100][63], un choix a dû être fait afin de restreindre le champ de notre étude. Aidés en cela par plusieurs équipes de conception de processeurs de SGS-THOMSON en charge du développement de processeurs embarqués dédiés ou standards, la taille de code à été identifiée comme étant le critère primordial. Plusieurs arguments ont motivé ce choix.

En premier lieu la taille croissante des applications embarquées actuelles et futures, notamment dans les domaines du multimédia et de la téléphonie mobile, contribue à ce que la majeure partie de la surface d'un système embarqué soit réservée aux mémoires (programmes, données, caches [80]) [99]. Le coût de production d'un circuit intégré étant directement lié à la surface de silicium, les systèmes destinés à un marché de gros volume (ce qui est le cas dans ces domaines), se doivent d'être particulièrement optimisés en surface.

De plus, en l'absence de compilateurs C réellement efficaces en traitement du signal sur des architectures dédiées [65][64], la méthodologie actuelle de développement du logiciel embarqué fait encore largement intervenir l'expertise du programmeur dans la production de code assembleur purement DSP[1]. La taille de ce code critique étant relativement réduite[2], sa conception est généralement assurée par les programmeurs et les choix architecturaux liés au traitement DSP bien maîtrisés. Ce qui n'est pas le cas de la majeure partie du code (donc orienté contrôle) qui est générée par le compilateur, ne serait-ce qu'en raison de sa taille. Il semble donc raisonnable d'orienter nos efforts sur l'exploration d'architectures optimisées en taille de code.

Raffinement d'architectures

Le raffinement d'une architecture consiste à modifier quelques caractéristiques d'un processeur sans remettre en cause l'ensemble de la conception. On obtient ainsi relativement rapidement (grâce à la réutilisation d'une grande partie de l'existant) un circuit optimisé et mieux adapté à de nouvelles contraintes [71]. On peut citer trois cas de figures entraînant le raffinement d'une architecture existante : la réduction de coût d'un circuit à fonctionnalités égales, la conception d'une architecture dérivée de l'architecture initiale, la conception d'une architecture spécifique à une application donnée.

La réduction de coût à fonctionnalités égales constitue généralement la deuxième génération d'un circuit, dont les caractéristiques et performances sont par ailleurs satisfaisantes. Le but est alors d'éliminer les configurations inutilisées ou rarement utilisées[3].

[1] Le compilateur intervient par contre efficacement dans la génération de la majorité du code restant, orienté contrôle.

[2] Il est courant d'observer que 80% du code stocké en mémoire concerne les fonctions contrôle, alors que le traitement DSP ne compte que pour 20%.

[3] Il peut s'agir de certaines connexions entre les blocs fonctionnels, ou bien d'instructions particulières.

Parfois une réorganisation du jeu d'instructions, favorisant les instructions les plus fréquemment utilisées, permet d'améliorer les performances et donc de re-dimensionner certains blocs (cache-instructions, décodeur d'instructions,...).

La conception d'une architecture dérivée consiste à améliorer les performances, ou au contraire à limiter la puissance de l'architecture initiale de manière à mieux s'adapter à une classe d'applications particulière. On obtient alors une famille de processeurs, ayant en commun la plupart des caractéristiques matérielles (afin de réutiliser au maximum les blocs existants).

La conception d'une architecture spécifique à une application vise à produire un circuit dédié, dont les ressources matérielles sont calculées au plus juste. Le faible coût du circuit obtenu est généralement l'objectif principal. Cela concerne de préférence les circuits destinés à un marché de gros volume ou très compétitif. Ce cas de figure diffère du précédent (architecture dérivée) dans la mesure où l'application devant tourner dessus est unique et connue. Cette connaissance permet d'atteindre un niveau d'adapation très élevé.

Dans ces trois cas de figures nécessitant le raffinement d'une architecture, intervient inévitablement une application (ou classe d'applications) donnée. C'est-à-dire que les choix architecturaux sont dictés par les performances obtenues pour un logiciel ou un ensemble de logiciels bien précis.

L'inter-dépendance compilateur/architecture

Dans les trois cas de figures observés ci-dessus, le raffinement se base sur une analyse très précise des besoins d'une ou plusieurs applications, ainsi que sur le taux d'utilisation des ressources matérielles (blocs fonctionnels, registres, instructions, champs dans les instructions). Le raffinement d'une architecture est donc extrêmement lié à une application. Concevoir une architecture sans prendre en compte le logiciel devant tourner dessus présente un risque certain de ne pas identifier précisément les insuffisances matérielles (le chemin critique) et ainsi de ne pas adresser les vrais problèmes.

Le logiciel est exécuté sur l'architecture sous sa forme code machine, produite par le compilateur C. Il est donc particulièrement important de prendre en compte l'inter-dépendance entre le code généré et les ressources disponibles, c'est-à-dire entre le compilateur et l'architecture [79][24]. Plus que l'architecture isolée, il s'agit de raffiner le couple compilateur-architecture. Ce raffinement passe donc par la maîtrise du comportement du compilateur, motivant ainsi une étude poussée de celui-ci.

4.1.2 Objectifs

Nous nous sommes fixés deux objectifs pour cette étude. Le premier est d'aider la définition et le raffinement d'un jeu d'instructions par la mesure quantitative précise d'un certain nombre de critères jugés importants, du point de vue de l'architecte. Le second objectif vise à tirer des enseignements sur l'interaction entre les algorithmes de compilation et le jeu d'instructions, afin d'adapter l'un à l'autre. Ces leçons seront utiles au concepteur de compilateur reciblables pour développer ceux-ci, mais aussi pour conseiller l'architecte sur les moyens d'adapter l'architecture aux contraintes de compilation.

Fournir des mesures précises pour la définition ou le raffinement d'un jeu d'instructions

Les choix architecturaux faits lors de la définition d'une architecture dédiée originale, de même que lors du raffinement d'une architecture existante, sont dictés par les performances (au sens large : taille de code, temps d'exécution, occupation des ressources, etc...) obtenues

pour un ensemble de logiciels bien précis. La qualité et la précision de l'analyse de ces performances conditionnent l'efficacité de l'architecture finale. C'est pourquoi la mesure précise d'un certain nombre de critères, à définir, apportera les informations nécessaires à l'obtention d'une solution optimisée [42].

Le jeu d'instructions est souvent considéré comme une spécification de référence pour la conception parallèle d'une architecture et de l'environnement de compilation. Il contient en effet l'ensemble des opérations exécutables sur l'architecture, les ressources accessibles (registres, blocs fonctionnels), ainsi que l'encodage des instructions en code machine (binaire). La totalité des choix architecturaux liés aux besoins logiciels est donc contenue dans les spécifications du jeu d'instructions[4]. L'exploration de ces choix passe alors par l'exploration du jeu d'instructions.

Cette exploration est aidée par la mesure de certains critères, jugés importants dans le contexte choisi. Ces critères sont liés à l'occupation des ressources disponibles, aux besoins particuliers de l'application utilisées, mais aussi aux performances des algorithmes de compilation.

Fournir des directives pour adapter le jeu d'instructions à la compilation

La qualité du code généré par un compilateur dépend à la fois de la performance du compilateur et de la richesse du jeu d'instructions. Mais elle dépend aussi d'un troisième paramètre, qui est l'adéquation entre le compilateur et le jeu d'instructions. Plus précisément, les algorithmes de compilation doivent pouvoir tirer parti des instructions disponibles, et celles-ci doivent être adaptées aux qualités mais aussi aux faiblesses du compilateur. La méconnaissance de ces interactions conduit à définir des instructions qui ne seront pas utilisées, ou au contraire à ne pas fournir les instructions absolument nécessaires pour obtenir un code performant. Il est important de remarquer qu'un jeu d'instructions efficace pour l'écriture de code manuellement ne l'est pas forcément pour la génération automatique. L'objectif est donc d'appréhender précisément (en les chiffrant) les interactions entre le compilateur et le jeu d'instructions, de manière à identifier les erreurs à ne pas commettre et les bonnes décisions à prendre dans la définition d'un jeu destiné à la compilation.

4.2 Techniques de minimisation de code

Cette section présente les principes de base d'un certain nombre de techniques permettant d'augmenter la densité de code d'une application complète, soit en réduisant la largeur moyenne du mot-instruction soit en réduisant le nombre d'instructions. Chaque technique est illustrée par des résultats d'expérience, permettant d'évaluer l'impact de chacune.

Afin de pouvoir comparer ces techniques entre elles, il est nécessaire d'utiliser systématiquement la même architecture de base, ainsi que les mêmes outils de génération et d'analyse. C'est pourquoi il a été choisi de prendre comme référence une étude menée à l'université de Manchester de 1993 à 1996 [33][34], basée sur un jeu d'instructions Sparc. Il est difficile d'extrapoler les résultats à d'autres architectures, mais la comparaison relative des diverses techniques demeure pertinente.

[4] Les choix architecturaux non liés au logiciel, tels que par exemple la hiérarchie des blocs fonctionnels, les détails d'implémentation du pipeline, ou la taille et le fonctionnement des caches, ne se retrouvent pas dans le jeu d'instructions. Bien qu'ayant un impact certain sur les performances finales, ils ne sont pas concernés par notre étude qui se concentre sur les spécificités liées à des besoins logiciels particuliers.

4.2.1 Définitions

Cette sous-section rappelle la définition de grandeurs et termes couramment employés pour caractériser l'encodage d'un jeu d'instructions et son impact sur la place mémoire occupée par un programme.

La densité d'encodage et la densité de code mesurent la compacité d'un jeu d'instructions et d'un programme, respectivement. Les techniques employées pour réduire la taille mémoire visent soit à réduire le jeu d'instructions en largeur, soit à réduire le nombre d'instructions dans un programme. Elles peuvent s'appliquer à plusieurs niveaux, plus ou moins liés à l'architecture matérielle du processeur embarqué.

Code-opération, opérandes et formats

Une instruction asssembleur (code machine) est composées de un ou plusieurs champs de bits. Nous distinguons dans une instruction le code-opération du (ou des) opérande(s). Le code-opération indique la nature de l'opération effectuée en designant l'opérateur matériel concerné ainsi que des commandes qui lui sont destinées. Un code-opération peut être composé de plusieurs champs. Le (ou les) opérande(s) de l'opération est (sont) également composé(s) d'au moins un champ. Un opérande peut être une référence à un registre, une valeur constante, une adresse en mémoire, ou une combinaison arithmétique de chacuns. Dans le cas des valeurs constantes, le champ (groupe de bits) code directement cette valeur en binaire. C'est un champ immédiat.

Le nombre de bits requis dépend de la valeur elle-même (il faut $n+1$ bits pour coder la valeur entière non-signée 2^n). Dans le cas le plus simple, le champ immédiat est composé du même nombre de bits quelles que soient la valeur à coder et le code-opérations. Nous disons alors qu'un seul format immédiat est disponible. Sa taille est généralement grande, et correspond à la taille des données traitées dans les calculs (8, 16 ou 32 bits).

Lorsque de petites valeurs sont codées par ce format, il apparait un gaspillage de bits, préjudiciable à une densité de code optimale. C'est pourquoi il peut être intéressant de réduire la taille de ce format, ou même de rajouter un ou plusieurs autres formats de tailles réduites. Dans le cas où une valeur constante ne peut pas être codées par le plus grand des formats disponibles, elle est stockée en mémoire à l'initialisation et chargée par un accès direct avant son utilisation. Généralement la mémoire programme (ROM) est utilisée pour stocker toutes les valeurs initialisées (constantes). L'instruction de chargement par accès direct est par elle-même coûteuse en largeur de mot-instruction puisqu'elle contient une adresse absolue. La conséquence est donc un surcoût en mémoire programme.

Densité d'encodage

La densité d'encodage est une grandeur associée à un jeu d'instructions donné. Elle concerne l'utilisation des bits dans les mot-instructions, indépendamment de toute application.

On peut mesurer la densité d'encodage d'un jeu d'instructions en rapportant le nombre d'instructions effectivement encodées au nombre maximum théorique d'instructions pouvant être codées avec le même nombre de bits. On considère dans un premier temps un jeu d'instructions dont la largeur du mot-instruction est fixe. Si l est la largeur en bits du mot-instruction, le nombre maximum d'instructions pouvant être codées est $N_{max} = 2^l$. Si l'on note N_{reel} le nombre d'instructions réellement encodées dans le jeu d'instructions, la densité d'encodage est définie par :

$$D_e = \frac{N_{reel}}{N_{max}}$$

La mesure de la densité d'encodage permet d'évaluer l'efficacité de l'encodage d'un jeu d'instructions donné, et donc d'optimiser cet encodage a priori, sans utiliser une application

particulière. Dans le cas où la largeur du jeu d'instructions n'est pas fixe, la densité d'encodage s'exprime comme la moyenne des densités d'encodage de chaque mot-instruction. Nous parlerons alors de largeur moyenne du mot-instruction pour un jeu d'instructions complet.

Densité de code

La taille-mémoire nécessaire pour stocker un code programme est fonction de deux variables : la largeur moyenne l du mot-instruction (en bits) et le nombre n d'instructions (en mots-instructions). Le nombre de bits total nécessaires pour stocker ce code est appelé taille-mémoire et s'exprime par :

$$T_{mem} = n * l$$

La densité de code est généralement utilisée pour comparer deux implémentations différentes d'un même code programme. Les deux implémentations peuvent correspondre à des optimisations d'encodage du jeu d'instructions différentes pour une même architecture, ou alors à deux architectures complètement différentes, dont on veut comparer l'efficacité de l'encodage. C'est pourquoi la densité de code est souvent exprimée d'une manière relative, par un pourcentage ou un coefficient multiplicateur.

On peut toutefois définir la densité de code absolue comme l'inverse de la taille-mémoire occupée par un code programme. La densité de code s'exprime alors par :

$$D_c = \frac{1}{T_{mem}} = \frac{1}{n*l}$$

Par la suite, nous nous attacherons à la variation de la densité de code entre deux jeux d'instructions pour un même code, traduisant aussi bien une variation du nombre d'instructions, de la largeur du mot-instruction, que des deux simultanément.

Deux axes de minimisation de la taille de code

Minimiser la taille-mémoire $T_{mem} = n * l$ revient à minimiser le nombre d'instructions n du programme et/ou à minimiser la largeur moyenne l du mot-instruction. Ces deux grandeurs définissent deux axes d'optimisation de la densité d'un code programme : la largeur moyenne du mot-instruction et le nombre d'instructions.

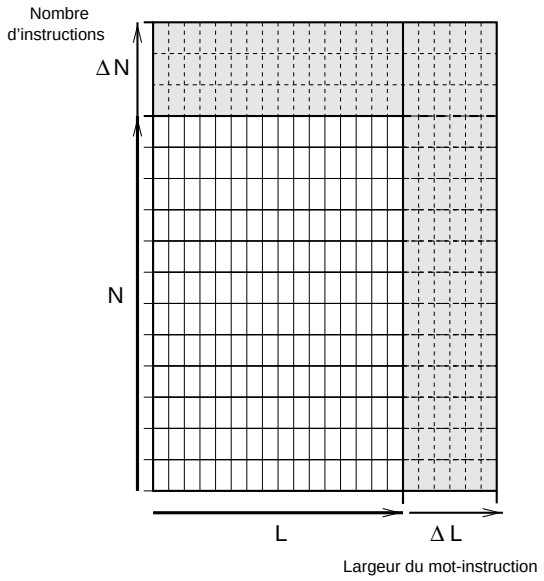

FIG. 4.1: Surface-mémoire d'un programme

Afin de répertorier les différentes techniques de compaction, il est nécessaire de distinguer au moins quatre façons d'augmenter la densité de code :

- réduire la largeur du mot-instruction en conservant le nombre d'instructions constant,
- réduire la largeur du mot-instruction en augmentant le nombre d'instructions,
- réduire le nombre d'instructions en conservant la largeur du mot-instruction constante,
- réduire le nombre d'instructions en augmentant la largeur du mot-instruction.

Chacune de ces approches est maintenant décrite, illustrée par quelques techniques de minimisation de la taille de code couramment utilisées, ainsi que quelques résultats d'analyses destinés à estimer le gain que l'on peut espérer pour chacune des techniques.

Réduction de la largeur du mot-instruction La réduction de la largeur du mot-instruction, même de quelques bits, permet d'économiser une quantité non négligeable de mémoire, dès lors que le nombre d'instructions dans le programme est suffisamment grand.

Les techniques couramment employées pour compacter le code en largeur peuvent se subdiviser en deux groupes : celles pour lesquelles le nombre d'instructions reste rigoureusement constant et celles qui entraînent une augmentation du nombre d'instructions total. Dans le premier cas, la technique compresse (ou simplifie) l'encodage du mot-instruction, mais le comportement même de l'instruction n'est pas affecté. Ainsi le nombre d'instructions reste constant. Tous les bits économisés en largeur se retrouvent économisés sur la surface totale du code en mémoire, améliorant directement la densité de code. Sur la figure 4.1, cette approche se traduit par un ΔL négatif (diminution de la largeur moyenne d'instructions) et un ΔN nul (nombre d'instructions constant). Dans le second cas, la réduction en largeur entraîne une augmentation du nombre d'instructions ($\Delta L < 0$ et $\Delta N > 0$). Par exemple la suppression de certaines instructions pour réduire la largeur moyenne peut nécessiter d'ajouter une ou plusieurs instructions dans le code programme pour retrouver le même comportement. Il est alors nécessaire de s'assurer que l'augmentation du nombre d'instructions est plus que compensée par l'économie obtenue en largeur moyenne du mot-instruction.

Réduction du nombre d'instructions La réduction du nombre d'instructions peut également se faire à largeur constante ou à largeur croissante, selon les techniques employées. La réduction du nombre d'instructions n'entraînant pas d'augmentation de la largeur ($\Delta N < 0$ et $\Delta L = 0$ sur la figure 4.1) s'obtient par exemple par des techniques de suppression d'instructions inutiles, ou de remplacement de séquences d'instructions par une unique. La largeur moyenne du mot-instruction subit une augmentation ($\Delta N < 0$ et $\Delta L > 0$), lorsque des techniques d'ajout d'instructions (triadiques ou dyadiques) ou d'ajout de registres (nécessitant des références plus larges) sont employées.

Trois niveaux d'optimisations

Nous avons identifié au moins trois niveaux auxquels des techniques d'optimisation de la densité de code peuvent être appliquées : le niveau architectural, le niveau du jeu d'instructions, et les optimisations du compilateur.

Le niveau architectural correspond aux choix faits dans la conception de l'architecture du coeur du processeur. On peut y classer les opérateurs, les registres ou bancs de registres, ainsi que les interconnexions possibles entre ces composants. Ainsi le nombre total de registres est un critère relevant du niveau architectural. Les choix faits à ce niveau revêtent une importance particulière dans la mesure où ils interviennent très tôt dans le flot

de conception d'un circuit, et conditionnent le développement de toutes les composantes d'un processeur embarqué : l'architecture du coeur, l'encodage du jeu d'instructions, et la chaîne de compilation associée. En conséquence, il est très difficile (ou coûteux) de revenir sur ces choix pendant le développement de l'ensemble. Ils doivent donc être soigneusement justifiés.

Au niveau du jeu d'instructions, les choix portent sur les opérations effectivement accessibles par le compilateur (ou le programmeur) et peuvent être un sous-ensemble de toutes les opérations théoriquement possibles avec l'architecture du coeur. La suppression de certaines combinaisons permet de réduire la largeur moyenne du jeu d'instructions, au risque d'obliger le compilateur à choisir une séquence sous-optimale. La complexité de l'encodage d'un jeu d'instructions, ainsi que les interactions étroites avec les algorithmes du compilateur rendent l'encodage d'un jeu d'instructions performant particulièrement délicat.

La chaîne de compilation dont nous disposons autorise la définition d'un certain nombre de paramètres, influençant directement les algorithmes internes. Ceux-ci concernent notamment l'allocation de registres, la génération de code et la compaction des instructions. La modification de ces paramètres a un impact réel sur la qualité du code produit. Il est donc indispensable d'acquérir la maîtrise de ces paramètres pour atteindre une densité de code optimale.

4.2.2 Réduction de la largeur à nombre d'instructions constant

Cette sous-section répertorie plusieurs techniques visant à réduire la largeur moyenne du mot-instruction d'un jeu d'instructions donné, en conservant le nombre d'instructions du code programme strictement constant.

Suppression de champs/bits inutilisés

Dans le but de réduire la complexité et d'augmenter la vitesse du décodage des instructions, l'encodage de celles-ci est souvent simplifié. On peut citer plusieurs types de simplifications.

- La définition d'une largeur unique des instructions facilite le chargement depuis la mémoire.
- Si la mémoire est accessible par octets, ou si les instructions ne sont pas de largeur fixe, il est préférable d'avoir des instructions dont la largeur est un multiple de 8.
- L'alignement des champs dans le mot-instruction suivant les blocs fonctionnels activés, commun à toutes les instructions, facilite grandement le décodage des celles-ci.

Ces techniques de simplification du décodage des instructions introduisent un gaspillage de bits, ou même de champs entiers, dans les instructions les moins complexes. Il en résulte une faible densité d'encodage.

Suppression d'instructions peu utilisées

Une approche plus agressive pour réduire la largeur des mot-instruction consiste à réduire le nombre d'instructions, et ainsi la taille des codes-opérations. Le choix des instructions à supprimer est particulièrement crucial. En effet, il faut distinguer trois types d'instructions :

1. les instructions irremplaçables : elles correspondent à des modes opératoires particuliers (exécution privilégiée), ou mettant en œuvre des ressources matérielles spécifiques. Elles ne peuvent être supprimées, puisqu'aucune séquence d'instructions "classiques" ne peut produire la même fonctionnalité.

2. les instructions facilement remplaçables : une séquence dont la fonctionnalité est équivalente, et dont la longueur n'est pas excessive (de 1 à 5 instructions), peut être trouvée. Un exemple peut être une opération faisant appel à des modes d'adressage en mémoire complexes pour ses trois opérandes (deux sources et une destination). Dans ce cas, l'utilisation d'un registre ou plusieurs temporaires permettrait d'employer des instructions plus simples.
3. les instructions difficilement remplaçables : une séquence équivalente peut être trouvée, mais sa longueur est très pénalisante en terme de taille de code (plusieurs dizaines d'instructions). Un bon exemple est l'opérateur de multiplication.

Seules les instructions de la catégorie numéro 2 sont susceptibles d'être efficacement remplacées. De plus, il est préférable de remplacer des instructions apparaissant peu fréquemment dans le code du programme.

Codage à largeur variable (Huffman)

Plusieurs analyses montrent qu'un sous-ensemble restreint d'instructions apparaît très souvent, alors qu'une majorité d'instructions ne sont utilisées que très rarement [32][39]. A condition de disposer d'un jeu d'instructions à largeur variable, il est donc intéressant, en termes de taille de code (et également de performance), d'attribuer aux instructions les plus fréquentes les encodages les plus courts.

Le codage de Huffman, considéré comme optimum, subdivise un jeu d'instructions en n blocs en fonction des fréquences d'apparition des instructions dans le code programme, jusqu'à ce que chaque bloc ne contienne qu'une instruction (algorithme de dichotomie). L'encodage du jeu d'instructions obtenu devient relativement complexe, puisque la largeur du mot-instruction varie d'une instruction à l'autre. Cependant, des techniques matérielles permettent d'assurer un délai de décodage suffisamment rapide pour les instructions les plus fréquentes, ce qui permet d'atteindre des performances proche des décodeurs à largeur fixe.

Réduction du nombre d'opérandes

Certains jeux d'instructions, comme celui du processeur Sparc [116], implémentent systématiquement les opérations arithmétiques avec trois opérandes (instructions triadiques), dont au moins deux sont des références aux registres : par exemple $R3 = R1 + R2$ ou $R2 = R1 + 2$.

Si deux des registres sont les mêmes, l'opération $R1 = R1 + 2$ sera codée en référençant deux fois le registres $R1$, introduisant ainsi une redondance inutile. L'ajout d'une instruction dyadique permettrait de ne référencer qu'une seule fois le registre $R1$, en écrivant $R1 += 2$ (écrit en langage C). Cet ajout d'instruction économise une référence à un registre, ce qui représente souvent 3 à 6 bits (suivant le nombre de registres). Ces bits peuvent aussi bien être supprimés, réduisant la largeur de l'instruction, ou être réutilisés par d'autres formats. Le prix à payer est un plus grand nombre de codes-opérations dans le jeu d'instructions.

Réutilisation du dernier résultat

La sous-section précédente a évoqué la redondance de références à un même registre dans une instruction. La même observation peut être faite entre deux instructions adjacentes. En effet, il apparaît souvent que le résultat d'une instruction est utilisé comme source de l'instruction suivante (29% des instructions utilisent le résultat de l'instruction précédente [33]).

Là encore, la redondance de référence peut être éliminée par la définition de nouvelles instructions. Dans celles-ci, certains opérandes ne référencent pas un registre donné mais un registre implicite, celui où le résultat de l'instruction précédente a été stocké.

4.2.3 Réduction de la largeur à nombre d'instructions croissant

Cette sous-section présente un certain nombre de techniques de réduction de la largeur moyenne du mot-instruction mais qui entraînent une augmentation sensible du nombre total d'instructions. L'estimation vise alors à garantir qu'il y a plus de bits économisés en largeur que de bits consommés en instructions supplémentaires.

Réduction de la taille du format immédiat

Les champs immédiats dans les instructions permettent de coder des valeurs constantes, utilisées aussi bien dans les calculs arithmétique ou logique que dans les branchements relatifs ou les accès à la mémoire par le mode d'adressage base+déplacement.

La distribution des valeurs codées par ces champs immédiats n'est pas homogène. Certaines valeurs sont plus souvent utilisées que d'autres pour l'une des raisons suivantes :

- les valeurs 0, 1, -1, 2, -2 sont souvent utilisées pour calculer des indices de compteur (initialisation, incrémentation, décrémentation),
- les valeurs proches des puissances de 2 sont souvent utilisées dans les opérations logiques de masquage,
- les valeurs faibles (inférieures à 16 ou 32) sont souvent utilisées pour les branchements relatifs (généralement courts), ou les accès-mémoire par base+déplacement (dans le cas de structures ou tableaux).

Par conséquent, on constate souvent un gaspillage de bits dans l'utilisation des formats immédiats, généralement réduits à un unique format de grande taille (8, 16 ou 32 bits). La réduction de ce format unique d'immédiat à une largeur l inférieure à la taille des données généralement manipulées permet d'économiser des bits dans la plupart des cas (lorsque les valeurs à coder sont relativement faibles), mais empêche le codage des grandes valeurs (supérieures à 2^l).

Ajout de formats immédiats intermédiaires

La technique précédente, consistant à réduire la largeur du format immédiat unique, devient inefficace lorsque cette largeur atteint un certain plancher, en-deçà duquel le surcoût en mémoire programme l'emporte sur le gain en largeur des instructions. Pourtant, les valeurs constantes faibles demeurent codées sur plus de bits que nécessaire, engendrant toujours un gaspillage.

L'ajout d'un ou plusieurs formats intermédiaires de tailles réduites permet alors d'optimiser le codage des valeurs faibles tout en conservant un codage possible de valeurs plus élevées. A nouveau, l'ajout de formats supplémentaires augmente la largeur des codes-opérations, et doit donc être mené soigneusement. Le choix d'une configuration proche de l'optimum se fait en ajustant deux grandeurs :

1. le nombre de formats intermédiaires
2. la largeur en bits de chaque format

Il n'est pas possible de déterminer a priori la configuration optimale, étant donné que le nombre de bits économisés par cette optimisation dépendent étroitement des valeurs constantes encodées et donc du code programme concerné. On peut cependant anticiper

qu'un format très court (1, 2 ou 3 bits) apporte un gain assez important puisque le nombre de bits économisés est assez grand (de plus de 8 bits à 3 bits).

Transformation en plusieurs instructions dyadiques

Si l'ajout de nouvelles instructions dyadiques dans le jeu d'instructions n'est pas possible, la transformation en séquences dyadiques est alors envisageable. Cette transformation consiste à remplacer une instruction triadique par une séquence d'instructions (souvent deux ou trois) dyadiques. Par exemple, l'instruction triadique 4.1 peut être remplacée par la séquence de deux instructions dyadiques 4.2 et 4.3.

$$R3 = R1 + R2 \tag{4.1}$$

$$R3 = R1 \tag{4.2}$$

$$R3+ = R2 \tag{4.3}$$

Cette alternative permet de limiter la largeur moyenne des instructions (en réduisant le nombre de codes-opérations), au prix d'un nombre d'instructions dans le code programme plus grand. L'application de cette technique doit donc faire l'objet d'un compromis et d'une étude au cas par cas.

BULU

La technique Break-Up/Look-Up (BULU) exploite la localité les codes-opérations et des références aux registres [39]. Partant du constat que sur une séquence d'instructions donnée, seul un sous-ensemble de codes-opérations et de références aux registres est utilisé, un programme est divisé en plusieurs sous-blocs de longueur donnée. Dans chaque sous-bloc, seuls les codes-opérations et références aux registres effectivement utilisés sont codés[5]. La largeur des instructions est ainsi considérablement réduite. Afin de décoder de telles instructions, une table de translation est associée à chaque sous-bloc. La taille de cette table doit être ajoutée à la taille totale du programme, c'est pourquoi cette technique est classée parmi les techniques impliquant une augmentation du nombre d'instructions (bien qu'il ne s'agisse pas à proprement parlé d'instructions). Les champs immédiats ne sont pas concernés par cette technique.

Préfixe

La technique préfixe est une des implémentations possibles d'un jeu d'instructions à largeur variable. En fait, chaque instruction ne peut avoir qu'une largeur multiple de 8 bits (octets). Le premier octet d'une instruction contient la largeur en octets de celle-ci, permettant ainsi d'anticiper le chargement et le décodage des octets suivants, s'il y en a. L'avantage principal de cette approche est de limiter le nombre de bits inutilisés dans l'encodage d'instructions simples. Cependant, la subdivision des instructions en plusieurs octets exige plusieurs cycles pour charger la totalité d'une instruction longue.

La technique préfixe est par exemple utilisée pour l'encodage du jeu d'instructions du Transputer [33].

[5]Les valeurs immédiates ne sont pas prises en compte.

4.2.4 Réduction du nombre d'instructions à largeur croissante

Cette sous-section répertorie quelques techniques permettant de réduire le nombre d'instructions utilisées pour un programme donné, au prix d'un élargissement des mots-instructions. La encore, l'élargissement doit être plus que compensé par la réduction du nombre d'instructions.

Ajout de registres

Le temps d'accès à une mémoire RAM est généralement nettement supérieur au temps d'accès aux registres internes du processeur. De plus, la référence à une donnée en mémoire nécessite un nombre de bits relativement important (entre 8 et 32, typiquement), alors qu'une référence à un registre n'utilise qu'entre 3 et 8 bits. Pour ces raisons, il est plus efficace d'exécuter les calculs arithmétiques et logiques à partir du banc de registre interne du processeur, et à n'accéder à la mémoire que pour charger les données en entrée du calcul et sauvegarder les résultats de celui-ci.

Favoriser l'utilisation de registres à la place de cases mémoire permet ainsi de réduire la taille du programme et d'améliorer les performances. La conséquence de cette optimisation est de nécessiter un plus grand nombre de registres internes. Le nombre de bits codant les registres dans les instructions doit alors croître, contribuant à augmenter la largeur moyenne du mot-instruction.

Homogénéisation des bancs de registres

La spécialisation des registres apparaît souvent dans les architectures de traitement du signal. Le principe est de réserver un nombre réduit (1 à 4, typiquement) de registres à des tâches particulières, et ce pour chaque bloc fonctionnel du circuit. Par exemple 2 registres sont exclusivement utilisés pour stocker les opérandes en entrée du multiplieur. Aucun autre registre ne peut servir à cet usage. Les avantages d'une telle architecture sont les suivants :

- Les connexions entre les registres et les blocs fonctionnels sont limitées, réduisant ainsi la surface du coeur.
- L'encodage des références aux registres dans les instructions spécifiques (par ex. la multiplication) est réduit en largeur puisque le nombre de registres possible est très faible. La densité de code est améliorée.

Pourtant, une telle approche est très exigeante en ce qui concerne l'allocation des registres. Cela peut même devenir pénalisant si l'on utilise un compilateur C. En effet, un mauvais choix de registre peut entraîner, plus loin dans le code, une situation de pénurie qui est résolue par l'insertion de *mov* entre plusieurs registres, voire par l'utilisation de la mémoire. Il en découle un surcoût en nombre d'instructions, ainsi qu'une réduction des performances.

C'est pourquoi l'approche inverse, consistant à homogénéiser tous les bancs de registres, semble être plus adaptée à certains compilateurs. Tous les blocs fonctionnels peuvent alors référencer n'importe quels registres. Le prix à payer est une augmentation de la largeur du mot-instruction, dûe aux références de registres plus larges.

Chargement/stockage multiples

Le chargement ou le stockage de plusieurs données en mémoire nécessite généralement autant d'instructions qu'il y a de données. Lorsque les circonstances le permettent (adresses mémoire consécutives, pas de dépendances entre les données) une seule instruction peut suffire à coder une telle opération. Elle doit contenir la référence mémoire de départ, ainsi

que les registres concernés. Par exemple, un champ de n bits, dans lequel un bit est associé à chaque registre, peut permettre de coder un chargement multiple pour un banc de n registres.

L'apport d'une telle instruction peut réduire le nombre d'instructions total d'un programme, au prix d'une augmentation de la largeur des instructions (code-opération supplémentaire, largeur de cette nouvelle instruction). De plus, l'accès à la mémoire peut profiter du mode en rafale (*burst*) de certaines mémoires, permettant d'accroître la vitesse moyenne de transfert d'adresses consécutives.

Le processeur ARM implémente cette technique. Le faible nombre de registres internes (16) autorise une telle approche. Pour une machine qui aurait 32 registres ou plus, l'application de cette technique demanderait un jeu d'instructions très large, par exemple à largeur variable ou à préfixe.

Généralisation des instructions conditionnelles

Dans les architectures classiques, les seules instructions conditionnelles (c'est-à-dire dont l'exécution dépend du résultat d'une opération effectuée précédemment) sont les branchements. Pour réaliser l'exécution conditionnelle d'une opération quelconque, même élémentaire, un branchement conditionnel est effectué après l'opération de test [9].

Dans la séquence de code présentée en figure 4.2a), le résultat de la comparaison de $R0$ avec la valeur 0 conditionne le branchement en label1, où le registre $R1$ est chargé avec la valeur 5. Dans le cas où la comparaison échoue, le branchement conditionnel n'est pas pris

```
        test R0, #0              load R1, #3
        cbranch label1           test R0, #0
        load R1, #3              cload R1, #5
        branch label2            store R1, $addr
label1: load R1, #5
label2: store R1,$addr

            a)                        b)
```

FIG. 4.2: Codes sans chargement conditionnel (a), avec chargement conditionnel (b)

en compte, et le registre $R1$ est chargé avec 3. Le branchement inconditionnel en label2 permet de retrouver le flot du programme, à savoir le stockage de $R1$ à l'adresse-mémoire `$addr`. S'il existe une opération de chargement conditionnel d'un registre, appelée `cload`, la même séquence s'écrit comme représenté en figure 4.2b).

Le gain en nombre d'instructions est alors assez important puisque 5 instructions sont remplacées par seulement 3 (60%). De plus, les performances sont également améliorées puisque les branchements conditionnels sont souvent très coûteux (machines pipelinées). Le prix à payer est à nouveau l'ajout d'instructions supplémentaires, donc l'élargissement des codes-opérations et par conséquent des mot-instructions.

4.2.5 Tableau comparatif

Le tableau ci-dessous récapitule les techniques de compaction de code exposées plus haut, en associant à chacune d'elles son impact sur la largeur moyenne du mot-instruction, sur le nombre d'instructions du programme, ainsi que le gain en densité de code observé [33][34].

Technique	Largeur inst.	Nb instruc.	Gain
Suppression de champs inutilisés	↘	⟷	**3%**
Suppression d'instructions inutilisées	↘	⟷	**8-21%**
Codage à largeur variable de Huffman	↘	⟷	**25%**
Ajout d'instructions dyadiques	↘	⟷	**5%**
Réutilisation du dernier résultat	↘	⟷	**8%**
Réduction du format immédiat	↘	↗	**7%**
Ajout d'un format intermédiaire	↘	↗	**4%**
Réduction du nombre d'opérandes	↘	↗	**12%**
Break-Up / Look-Up	↘	↗	**30%**
Préfixe	↘	↗	n. disp.
Ajout de registres	↗	↘	**10%**
Homogénéisation des registres	↗	↘	**10%**

La plupart des techniques apportent un gain entre 3 et 10%. Les deux techniques produisant un gain plus important, le codage à longueur variable et BULU, mettent en jeu des opérateurs matériels de décodage spécifiques, ayant ainsi un impact sur le flot de conception matériel non négligeable. La combinaison de plusieurs techniques est envisageable, selon les relations d'inter-dépendances entre elles. Par exemple l'optimisation des codes-opérations peut être couplée avec la réduction du format immédiat sans restriction, alors qu'elle n'est pas forcément compatible avec la ré-utilisation du dernier résultat (qui rajoute des codes-opérations). Il n'est donc pas toujours possible d'estimer le gain apporté par la combinaison de plusieurs techniques en ajoutant le gain de chacune d'elles. Les exemples d'implémentations réelles d'une ou plusieurs de ces techniques décrits dans le paragraphe suivant donnent quelques indications sur l'efficacité de ces combinaisons.

4.3 Cas concrets de minimisation de code

Le gain théorique en densité de code évalué pour une technique de compaction donnée peut s'avérer différent du gain effectivement mesuré dans un cas réel, où des contraintes diverses peuvent parfois limiter son impact. C'est pourquoi il est intéressant de confronter les estimations précédentes aux mesures obtenues pour des processeurs existants, d'origines industrielle et universitaire. Nous nous concentrerons sur trois processeurs : le Thumb de ARM, le Hobbit de AT&T et le CRISCO de l'université de Berkeley. Le choix de ces processeurs a été fait avec le souci d'illustrer certaines approches originales, sans prétendre représenter l'ensemble des solutions disponibles.

4.3.1 Le Thumb de ARM

Le processeur Thumb [111][10], fourni par ARM (Advanced Risc Machines), est dérivé de l'ARM7 dont il conserve le coeur opératoire. Il est d'ailleurs entièrement compatible avec le jeu d'instructions 32 bits de l'ARM7. Cependant, le Thumb peut également exécuter un jeu d'instructions de seulement 16 bits de large, ce qui doit permettre d'améliorer notablement de la densité de code.

Le jeu d'instructions 16 bits, appelé jeu d'instructions Thumb, est en fait un sous-ensemble des instructions du jeu 32 bits. Seules les instructions fréquemment utilisées dans le code sont retenues[6]. De plus, celles-ci doivent remplir certains critères, explicités plus loin.

[6] 36 instructions pour le Thumb.

La compression des instructions 32 bits en instructions 16 bits est obtenue en appliquant simultanément plusieurs des techniques précédemment exposées. Il s'agit de :

- la sélection de codes-opérations courts pour les instructions fréquentes,
- la transformation d'opérations triadiques en opérations dyadiques,
- la réduction de la largeur du format immédiat.

La figure 4.3 montre un exemple de compression d'une instruction 32 bits courante.Pour

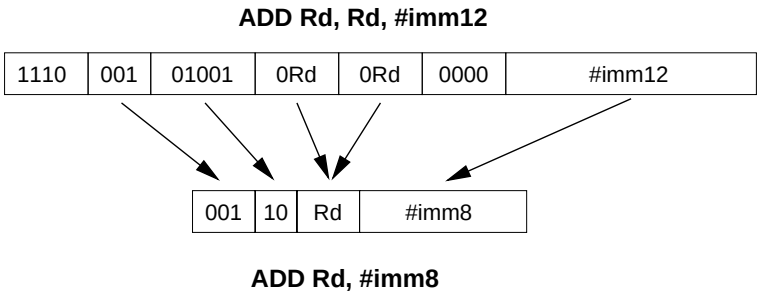

FIG. 4.3: Compression d'une instruction 32 bits courante en 16 bits

être compressée, une instruction 32 bits doit remplir plusieurs conditions.

1. Tout d'abord le code-opération (ici `ADD`) doit faire partie du sous-ensemble de codes-opérations pouvant être compressés. Le code-opération entier (`1110 001 01001`) est alors traduit en format court (`001 10`).
2. Ensuite, une opération triadique doit pouvoir être transformée en opération dyadique. Ici, le registre destination et le premier registre source sont les mêmes (`Rd`), donc une seule référence suffit.[7]
3. Enfin, l'opérande immédiat doit pouvoir être codé avec le format immédiat court, ici 8 bits au lieu de 12. Il doit donc être compris entre 0 et +255 en non-signé et -128 et +127 en signé.

Ces instructions 16 bits sont décompressées par un bloc additionnel, avant d'être décodées comme les autres instructions 32 bits. Le processeur peut ainsi exécuter aussi bien du code 16 bits que du code 32 bits, toujours nécessaire pour les instructions inexistantes dans le code 16 bits ou à des fins de compatibilité[8]. La surface de silicium supplémentaire, dûe au bloc de décompression, reste raisonnable (5% pour le Thumb).

Le jeu d'instructions Thumb, de part sa taille restreinte, présente un certain nombre de limitations : seuls 8 registres peuvent être adressés ; aucune opération de contrôle système, de support de coprocesseur ou de traitement d'exception n'est disponible ; aucune opération ne peut être conditionnelle. Cette dernière restriction est particulièrement importante puisque l'on a vu précédemment que les instructions conditionnelles étaient particulièrement efficaces en termes de performances. L'indisponibilité d'opérations de traitement d'exceptions ou de contrôle système dans le jeu 16 bits représente une réelle faiblesse dans le cas d'applications embarquées ayant des contraintes de temps réel. Pour de telles applications, une procédure entière doit être codée en 32 bits si une seule de ses instructions l'exige. La quantité de code 32 bits peut ainsi demeurer importante.

[7] Les seules opérations purement tryadiques pouvant être codées sur 16 bits sont ADD et SUB. Les autres, ne pouvant pas être transformées en opérations dyadiques, utilisent le jeu 32 bits.

[8] Dans le cas du Thumb, le passage du jeu d'instructions 16 bits au jeu 32 bits et vice-versa ne peut pas être effectué à n'importe quel moment de l'exécution, mais seulement d'une procédure à l'autre. Cette contrainte peut entraîner une réécriture du flot de contrôle afin d'isoler les instructions purement 32 bits, lorsqu'elles sont rares.

La densité de code est grandement améliorée (jusqu'à être doublée localement) pour les portions de code faisant exclusivement appel au jeu 16 bits, et ce sans que les performances ne soient affectées par rapport au jeu 32 bit. Par contre le code programme ne pouvant pas être codé en 16 bits demeure inchangé. Ainsi en moyenne, pour une même application, le code Thumb est entre 25% et 35% plus compact que le code ARM7 32 bits. Les résultats complets sont publiés dans [33].

4.3.2 Le Hobbit de AT&T

Le processeur Hobbit [8] est issu du projet d'architecture CRISP [15] développé par AT&T dans le but de supporter efficacement les programmes issus de la compilation C. C'est pourquoi la plupart des choix architecturaux ont été fait à partir d'analyses de performances de programmes C. Un des résultats les plus intéressants est que sur certains programmes, près de 50% du temps d'exécution est passé en appel de procédure, alors que seulement 5% du nombre total d'instructions exécutées est issu de ces procédures. Cela signifie que le changement de contexte (sauvegarde et restauration des registres) est très coûteux en temps et en taille de code.

La solution retenue dans le projet CRISP consiste à supprimer la visibilité (et donc les références) des registres par le jeu d'instructions. Au contraire, toutes les opérations accèdent directement aux données par référence mémoire. Deux opérandes peuvent ainsi être référencés, alors qu'un troisième opérande implicite (accumulateur) stocke le résultat des opérations. Afin de limiter les accès effectifs à la mémoire externe, une pile interne est constituée par un sous-ensemble de registres. Elle mémorise les opérandes les plus récemment utilisés, rendant ainsi inutile de sauvegarder et de restaurer explicitement les registres lors des appels et retours de procédures.

CRISP combine une architecture RISC et quelques spécificités des jeux d'instructions CISC. En effet les opérations mémoire-mémoire, la présence d'un accumulateur et l'encodage des instructions à largeur variable sont caractéristiques des processeurs CISC. Encore une fois, la constatation faite qu'un sous-ensemble restreint d'instructions est exécuté la plupart du temps suggère d'attribuer des codes-opérations courts à quelques instructions fréquentes. Dans ce but, trois longueurs d'instruction sont possibles : 10 octets, 6 octets et 2 octets (voir figure 4.4). Les deux premiers bits de chaque instruction indiquent la largeur de celle-ci. Par exemple les instructions requérant le codage d'adresses absolues 32 bits ou de grandes constantes utilisent une largeur de 10 octets. Pour des adresses relatives ou des constantes de faible valeur, le format de 6 octets de large peut convenir. Enfin, le format de 2 octets est réservé aux 32 instructions les plus fréquentes, contenant un champ code-opération restreint à 14 bits. Les analyses montrent qu'avec un tel encodage, 80% des instructions générées utilisent le format court de 2 octets.

Les instructions de 2, 6 ou 10 octets sont lues en mémoire programme puis stockées dans un cache-instruction appelé *prefetch buffer*, avant d'être décodées sous la forme d'une instruction RISC large de 192 bits destinée à l'unité d'exécution. Cette instruction large est également stockée dans un second cache-instruction, appelé *decoded instruction cache*.

L'architecture CRISP a servi de base au développement du processeur 92010 Hobbit de AT&T. Ce dernier implémente toutes les caractéristiques du CRISP, ainsi que certaines dispositions à la faible consommation. Il dispose de 64 registres de 32 bits et d'un espace adressable linéaire de 32 bits. Les mesures de taille de code montrent que le code Hobbit est presque deux fois plus dense que le code d'autres processeurs RISC équivalents, comme le Sparc ou le R3000. Le gain en densité de code est attribué au codage des instructions en largeur variable, améliorant nettement les performances du cache *(cache hits)* et réduisant les accès à la mémoire externe. Le taux d'instructions de 2 octets de large,

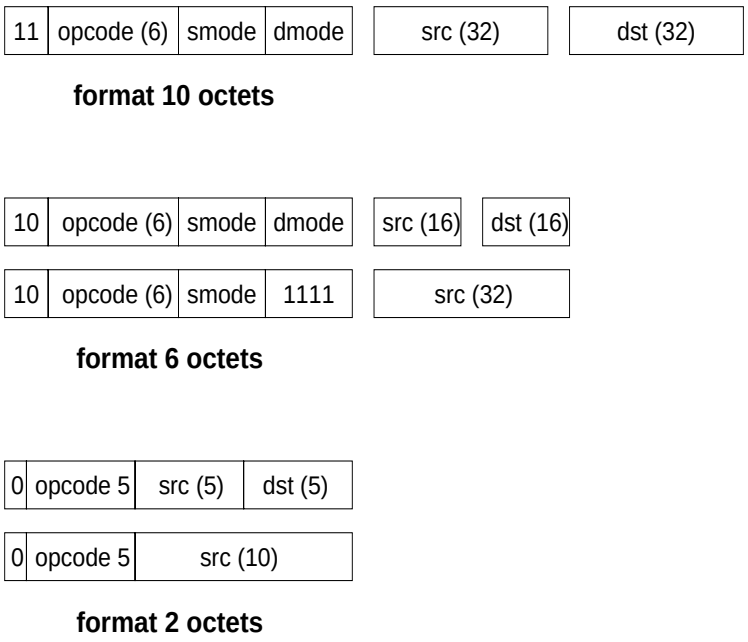

FIG. 4.4: Formats des instructions à largeur variable du CRISP

effectivement mesuré sur le Hobbit, dépasse 65% des instructions exécutées, mais n'atteint pas les 80% espérés lors de l'évaluation de l'architecture CRISP. Une des raisons avancées est que les techniques de compilation ne s'adaptent peut-être pas idéalement au concept de registres invisibles. Cependant, cette approche rend le code indépendant de la configuration de registres du coeur, autorisant ainsi un raffinement et une spécialisation de l'architecture sans aucune modification du code existant ni de la chaîne de compilation. Cet avantage peut s'avérer intéressant dans un contexte industriel très contraint en temps de mise sur le marché (*time-to-market*).

4.3.3 CRISCO

CRISCO (Compressed Reduced Instruction Set Computer) est un processeur expérimental développé à l'université de Californie à Berkeley, dans le département *Electrical Engineering and Computer Sciences* [39]. L'objectif est, à partir d'un coeur et d'un jeu d'instructions RISC 32 bits, d'obtenir une densité de code nettement meilleure par la compression de ce jeu d'instructions.

Deux approches sont envisagées pour parvenir à ce résultat :

1. modifier le jeu d'instructions dans le but de réduire le nombre de bits utilisés, ou
2. compresser le jeu d'instructions original sans le modifier.

L'implémentation successive des deux méthodes, à partir du même jeu d'instructions, permet de comparer les performances relatives. Pour réduire le nombre de bits utilisés dans le jeu d'instructions, la technique Break-up/Look-up (BULU) a été spécialement développée (voir plus haut). L'approche alternative, basée sur la compression du jeu existant, fait appel au codage à largeur variable de Huffman. Seule une version simplifiée du décodeur de Huffman, ne traitant pas les champs immédiats, a été implémentée. La compression ne concerne donc que les codes-opérations et les références aux registres, similairement à l'approche BULU.

Afin d'étudier l'efficacité de ces techniques, et de s'assurer de leur validité sur un large spectre d'applications, plusieurs programmes ont été utilisés pour les mesures. Quelques

programmes de jeu, particulièrement représentatifs d'applications embarquées exigeantes en termes de performances, ont été retenus. A ceux-ci s'ajoute un benchmark issu de SPECint95 (`099.go`), exigeant en puissance de calcul.

Pour résumer, l'encodage de Huffman procure un taux de compression de l'ordre de 30% (entre 27 et 34), champs immédiats non compris. L'encodage BULU atteint, voire dépasse ce taux sur les mêmes programmes. La table de translation est bien sûr prise en compte dans le calcul des tailles de codes. Les meilleurs résultats sont atteint par la technique BULU en choisissant soigneusement deux paramètres : le nombre de bits alloués à l'encodage des codes-opérations dans une fenêtre donnée et le nombre de bits alloués à l'encodage des registres. En effet les différentes expériences montre que le taux de compression peut varier de 11% à 37%.

4.3.4 Conclusion

Les trois exemples concrets d'implémentation de techniques d'optimisation de taille de code présentés ci-dessus produisent un gain en taille de code variant de 25 à 50%. Ces résultats confirment l'efficacité des quelques techniques exposées, tout en respectant les contraintes liées à la réalisation effective de ces solutions. La combinaison de plusieurs techniques a été expérimentée (jusqu'à trois dans le Thumb de ARM), avec de bons résultats. Le choix des techniques combinées est particulièrement crucial : pour le Thumb, chacune des techniques employées est indépendante des deux autres. En effet la sélection des codes-opérations courts pour les instructions fréquentes est indépendante de la largeur du format immédiat, de même que la réduction du nombre d'opérandes. Elles contribuent toutes au même objectif de réduire le nombre de bits utilisés par une instruction. Ainsi la combinaison des trois permet d'atteindre une largeur d'instruction très réduite (16 bits). Il serait intéressant d'explorer la combinaison de techniques non-indépendantes, ayant une influence l'une sur l'autre. L'efficacité de cette combinaison n'est obtenue que si un compromis est trouvé entre chacune. Ce compromis est l'un des objectifs de l'expérience du D960 relatée ci-dessous.

4.4 Description de l'exploration du D960

Ce paragraphe présente l'environnement dans lequel l'expérience d'exploration architecturale basée sur le D960 a été menée. Les résultats sont présentés dans la section suivante (page 111). L'objectif est d'expérimenter une exploration à grande échelle, mettant en jeu plusieurs centaines de configurations différentes [108]. Celles-ci comprennent la définition de quelques paramètres architecturaux (nombre de registres,...), la conception d'un encodage du jeu d'instructions à la fois compact et performant, et enfin la paramétrisation du compilateur. L'exploration d'un grand nombre d'architectures dérivées de l'architecture initiale, ainsi que la mesure systématique de la taille de code pour chaque alternative permet de dégager des tendances, selon les valeurs des paramètres expérimentées.

Tout d'abord l'ensemble des programmes (benchmarks et applications) compilés pour obtenir le code machine nécessaire à l'analyse est présenté, rappelant les critères qui ont motivé leur choix. Ensuite, la chaîne de compilation reciblable est brièvement décrite. Une présentation plus détaillée est fournie dans le chapitre 3. La technique de paramétrage de cette chaîne, spécialement développée, est ensuite exposée. Puis l'environnement d'exploration automatique, centré sur l'outil Flexplore, est décrit. Enfin, les outils spécifiques destinés à analyser et présenter les statistiques produites par l'exploration sont présentés.

4.4.1 Benchmarks et applications

Choix des programmes

Les programmes utilisés dans cette expérience ont été choisis selon trois critères essentiels :

- le code C doit être similaire au code de contrôle que l'on trouve dans les applications DSP complexes,
- un grand ensemble de fonctionnalités doit être couvert, au-delà des fonctions DSP de base,
- la quantité de code machine produit doit être suffisamment grande.

Afin de respecter ces critères, l'ensemble des programmes est composé de deux types de code :

1. des benchmarks typiques, de petite taille, et de fonctionnalités diverses assurent la couverture d'un maximum d'opérations,
2. des programmes d'applications existantes, orientées contrôle, assurent la représentativité des applications réelles, et garantissent un grand nombre de lignes de code.

Caractéristiques

Les caractéristiques principales des programmes de l'expérience sont résumées dans le tableau 4.2.

Nom	Type	Nb. lignes C
V42 modem	application	5690
Ackermann	benchmark	130
Driver ATM	application	315
automobile	application	3992
Convolution	benchmark	204
Dhrystone modifié	benchmark	503
Fibonacci itératif	benchmark	109
Fibonacci récursif	benchmark	109
Machine d'États Finis	benchmark	103
Driver disque dur	application	3831
Iti	benchmark	735
Quicksort	benchmark	250
Total		**15971**

TAB. 4.2: Caractéristiques des applications et *benchmarks* utilisés pour l'exploration

4.4.2 Chaîne de compilation

La chaîne de compilation Flexcc utilisée pour l'exploration d'architectures issues du D960 est décrite en détails dans le chapitre 2. Ce paragraphe rappelle les principes de base du reciblage de la chaîne pour une architecture donnée, puis décrit la technique de paramétrage qui a été mise en place pour générer automatiquement les compilateurs correspondant à un grand nombre d'architecture alternatives.

Description

Pour résumer, plusieurs fichiers de description et de configuration écrits manuellement permettent de générer un compilateur reciblé sur une architecture donnée. Les informations contenues de ces fichiers sont de plusieurs natures :

1. caractéristiques de l'architecture (registres, bancs mémoires),
2. règles de génération de code (instructions assembleur à produire pour chaque opération C, traitement des différents types de données, ...),
3. règles d'optimisation au niveau instructions (substitutions),
4. encodage du jeu d'instructions (formats, champs, codes binaires, ...),.

Ces informations sont réparties dans trois fichiers principaux :

- le CIF (*Compiler Information File*) contient les caractéristiques architecturales et les règles de génération de code (1 et 2),
- le RLS (*Rules*) contient les règles d'optimisation (3),
- le MDF (*Micro-instructions Definition File*) contient l'encodage du jeu d'instructions (4).

Chaque fichier doit respecter un certain langage, propre au type d'informations représentées : les règles du CIF ont une syntaxe riche et des constructions proches du C (fonctions, tests, variables) ; les règles du RLS sont de simples substitutions de chaînes de caractères, associées à des expressions régulières ; le MDF utilise des déclarations de données et de formats, ainsi que quelques tests.

Notre expérience d'exploration se base sur une proposition d'architecture nouvelle, le D960. Une première version du jeu d'instructions complet est disponible, permettant donc de produire un prototype de compilateur C. Dans un premier temps ce compilateur présente quelques restrictions par rapport à la norme C ANSI, mais qui n'ont pas impact sur la validité de notre expérience. Par exemple, le type d'entiers longs (`long int`) n'est pas implémenté. Les rares benchmarks utilisant ce type de données ont été modifiés afin de les remplacer par des entiers moyens (`int`).

Paramétrage des fichiers de configuration

Les trois fichiers de configuration (CIF, RLS et MDF) ont été écrits pour l'architecture D960. L'objectif étant d'explorer plusieurs architectures alternatives, mais proches, les informations contenues dans les fichiers de configurations ont été classées en deux catégories :

1. les informations génériques, issues de l'architecture originale D960 et communes à toutes les alternatives,
2. les informations paramétrées, propres à chaque alternative.

A partir des trois fichiers de configuration du D960 sont produits trois fichiers de configuration "gabarits". Chacun contient les informations génériques et un certain nombre de variables correspondant aux informations paramétrées. Sur la figure 4.5, le fichier `D960.cif` correspondant à l'architecture originale du D960 sert de base à la construction du fichier `gabarit.cif`. Toutes les informations génériques sont conservées alors que les informations paramétrées, ici la taille des registres L et R, sont remplacées par des variables. Un préprocesseur de texte tel que M4 (disponible en standard sous Unix) remplace chaque variable par sa valeur, selon l'alternative voulue. Il génère un nouveau fichier CIF, ici `D961.cif`. Ce dernier peut alors être utilisé pour générer un compilateur pour l'architecture D961, via Flexcc.

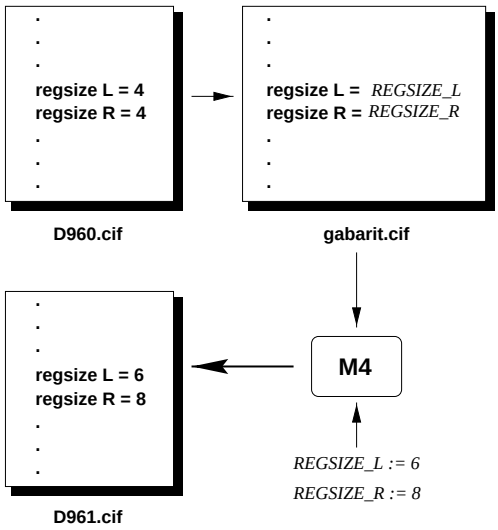

FIG. 4.5: Paramétrage d'un fichier gabarit par M4

Cette technique est employée pour deux des trois fichiers de configuration (CIF et MDF), car les règles d'optimisation ne sont pas concernées par les critères d'exploration retenus. L'ensemble des variables propres à une architecture alternative est appelée une configuration. La génération automatique du compilateur correspondant à une configuration permet d'explorer rapidement un très grand espace de solutions, sans intervention manuelle.

4.4.3 Environnement d'exploration FlexPlore

Un environnement générique d'exploration, indépendant aussi bien de l'architecture du processeur que des *benchmarks* utilisés, a été développé. Il permet d'automatiser toutes les phases de l'exploration, de la génération des compilateurs jusqu'à l'extraction de statistiques après compilation de tous les benchmarks. Ce paragraphe décrit succinctement les différentes phases de l'exploration, puis l'implémentation logicielle de ce prototype.

Grâce à la génération automatique du compilateur pour une configuration donnée, l'exploration de plusieurs solutions peut être complètement automatisée. L'environnement d'exploration - Flexplore - prend en charge pour chaque configuration la génération du compilateur correspondant (par Flexcc) et la compilation des *benchmarks* retenus. Pour notre expérience, 330 configurations ont été générées pour couvrir l'essentiel de l'espace des solutions.

4.5 Exploration autour du D960

Ce paragraphe décrit une expérience d'exploration architecturale autour du D960 (sa description est donnée dans le chapitre 2). L'environnement FlexPlore est utilisé pour produire plusieurs centaines de variations proches du D960, selon un ensemble restreint de critères choisis pour leur importance dans le domaine des architectures DSP 24 bits fortement encodées. Il se différencie ainsi d'environnements d'exploration portant sur des critères plus généraux (nombre d'additionneurs, de multiplieurs,...) comme par exemple Castle [60].

La comparaison des tailles de code obtenues pour chaque variation avec l'architecture originale du D960 permet d'évaluer le gain obtenu. Puis une architecture proche du D960 mais optimisée en taille de code est proposée.

4.5.1 Configuration des bancs registres

Les choix de conception architecturaux sont particulièrement importants car faits très tôt dans le flot de conception d'un processeur embarqué. Il est par la suite difficile de revenir dessus parce que l'ensemble du coeur, du jeu d'instructions, et de la chaîne de développement logicielle en dépendent.

Le choix du nombre de registres revêt une grande importance dans la conception d'une architecture [102][59][68]. En effet, autant le manque de registres que l'excès ont des conséquences très néfastes sur la densité de code d'une application donnée, ainsi que sur ses performances [25]. En simplifiant, un manque de registres entraîne l'utilisation d'emplacements mémoire (pour stocker les valeurs intermédiaires d'un calcul par exemple), introduisant de nouvelles instructions de chargement et stockage contenant les adresses mémoires absolues. Parfois, l'utilisation de certains registres pour des opérations auxquelles ils ne sont initialement pas destinés entraîne la génération de transferts entre registres (*moves*) supplémentaires. Il est donc important de mesurer l'impact de la variation du nombre de registres afin de déterminer notamment le seuil en-deçà duquel le manque de registres est trop coûteux en densité de code.

L'impact de l'homogénéisation des registres doit également faire l'objet d'une étude précise, car il n'est pas évident a priori. Une trop grande disparité des bancs de registres introduit des limitations lors du choix des opérandes d'une instruction, entraînant à nouveau l'insertion d'instructions supplémentaires.

Par ailleurs, la bonne utilisation de l'ensemble des registres d'un coeur est étroitement liée au comportement du compilateur, et plus précisément de l'allocateur de registres. On ne peut donc pas espérer fixer la configuration optimale des bancs de registres d'une architecture sans prendre en compte les performances du compilateur qui sera utilisé pour générer le code. C'est pourquoi il est intéressant d'explorer, pour un compilateur donné, l'impact de la variation du nombre de registres et de leur homogénéité sur la densité de code, et ce sur un large ensemble d'applications.

Variation du nombre de registres

Dans l'architecture originale du D960, les registres de données sont divisés en trois bancs : le banc de registres gauche, le banc de registres droit, et le banc de registres accumulateur. L'expérience consiste à faire varier le nombre total de registres données, en maintenant l'équilibre entre chaque banc. Le nombre de registres dans chacun des trois bancs varie de 2 à 10, donc le nombre total de registres varie de 6 à 30. La figure 4.6 représente en ordonnée le gain en taille de code obtenu pour chaque valeur du nombre total de registres (en abscisse). La référence, de gain 0, correspond à la configuration originale du D960 où chaque banc contient 2 registres.

L'ajout de registres améliore la taille de code. Le gain est assez net lorsque le nombre de registres données passe de 6 à 9 et de 9 à 12, avec une valeur de **-4%** pour chaque pas de 3 registres supplémentaires. Par la suite, et jusqu'à 30 registres, le gain avoisine -1%. Il finit par stagner pour 30 registres à une valeur de **-9%** par rapport à l'architecture originale du D960.

En résumé, l'ajout de registres données apporte un gain notable jusqu'à 12 registres (-8%). Au-delà, le gain n'excède pas -9%.

Homogénéisation des bancs de registres

L'architecture D960 est composée à l'origine de plusieurs bancs de registres : gauche, droit, accumulateur, sortie du multiplieur, pointeurs, index. Chaque banc est spécialisé

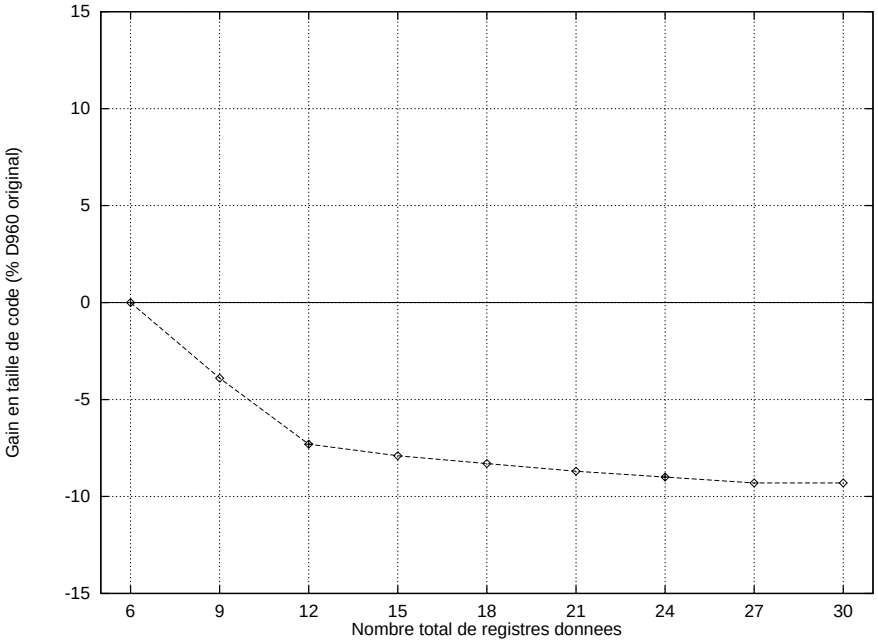

FIG. 4.6: Gain en taille de code selon le nombre de registres données

dans une fonction de stockage. Cette spécialisation permet d'exploiter au maximum le parallélisme des ressources (chemins de données), et également de réduire la largeur des références aux registres dans les instructions. En effet, le choix des registres en entrée d'une ressource (comme le multiplieur) est alors assez réduit.

La contrepartie de cette spécialisation des bancs de registres est de rendre plus difficile l'écriture de programmes, et a fortiori la compilation C. En premier lieu l'allocation de registres devient très complexe : les contraintes très fortes sur les registres utilisés par chaque ressources doivent être propagées dans tout le code afin d'éviter des transferts de registres inutiles. En second lieu, certains bancs de registres peuvent être plus souvent utilisés (et donc plus facilement saturés) alors que d'autres restent disponibles, rendant difficile l'exploitation optimale des ressources de stockage. Enfin, la vision par le programmeur ou le développeur du compilateur d'un espace de stockage homogène et régulier facilite le codage et l'exploitation optimale de ces ressources.

L'objectif de cette expérience est d'évaluer l'impact sur la taille de code de plusieurs configurations différentes, pour lesquelles les bancs de registres sont plus ou moins homogènes. Il s'agit plus particulièrement d'évaluer le comportement de l'allocateur de registres. Pour cela trois configurations différentes sont explorées :

1. Les registres sont divisés en trois bancs comme le D960 original, mais avec des contraintes d'utilisation plus fortes : les registres gauche ne peuvent être utilisés qu'en opérande gauche et les registres droit qu'en opérande droit. De plus le registre produit disparaît.
2. Les bancs gauche et droit sont réunis pour ne former qu'un banc de registres source alors que le banc accumulateur demeure inchangé.
3. Enfin, tous les registres sont réunis en un seul banc général à accès multiples.

Pour chaque configuration plusieurs tailles des bancs de registres sont explorées. Le nombre minimum de registres est fixé à 6 pour les configurations 1 et 2, et à 4 pour la configuration 3. Ces nombres sont choisis de façon à pouvoir compiler tous les benchmarks retenus, y compris ceux qui nécessitent le plus de registres. Le nombre maximum de registres est fixé

à 30 pour les configurations 1, 2 et 4 et à 32 pour la configuration 3.

La figure 4.7 donne les gains obtenus pour chaque configuration, et selon le nombre de registres total. Les trois configurations permettent indifféremment d'atteindre un gain

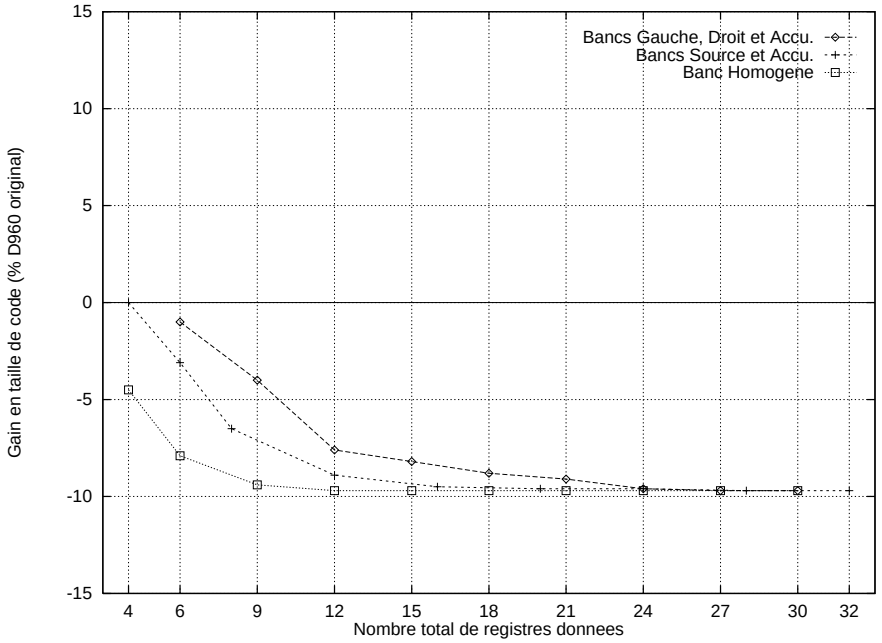

FIG. 4.7: Gain en taille de code selon l'homogénéité des registres

maximum de **-9%**, à partir de 24 registres. Par contre, seule la configuration homogène permet d'approcher ce plancher avec 12 registres seulement. Les deux autres configurations sont nettement moins performantes entre 6 et 12 registres. Si nous considérons un nombre total de 6 registres, la configuration homogène apporte un gain de près de 8% alors que les deux autres ne dépassent pas -3%.

L'homogénéisation des bancs de registres (suppression des spécialisations) apporte un gain indiscutable en taille de code lorsque le nombre de registres est assez réduit (moins de 12). Pour un nombre supérieur, le degré d'homogénéité n'influe pas nettement sur la taille de code.

4.5.2 Encodage des champs immédiats dans le jeu d'instructions

Parmi tous les aspects d'un jeu d'instructions pouvant être explorés pour améliorer la densité de code, les champs immédiats ont été identifiés comme primordiaux. En effet, le nombre d'instructions comportant des champs immédiats peut atteindre 70% du nombre d'instructions total d'un programme [39]. De plus, le grand nombre de bits utilisés par les champs immédiats laisse espérer un gain important. Enfin, les techniques d'amélioration de la densité de code n'adressent généralement pas l'encodage des constantes (cf. 4.2). Il est donc intéressant de comparer le gain apporté par l'optimisation des champs immédiats avec le gain obtenu avec les autres techniques.

L'exploration au niveau du jeu d'instructions consiste à étudier plusieurs possibilités d'encodage de certaines opérations dans le jeu d'instructions. Quatre analyses ont été effectuées sur l'encodage des opérandes immédiats (constantes). La première consiste à étudier l'évolution de la taille de code lorsqu'un unique format immédiat court est défini, alors que la deuxième ajoute un format intermédiaire optimisant l'encodage pour les valeurs moyennes. La troisième s'intéresse à l'encodage des valeurs immédiates de déplacement

dans l'adressage 'base plus déplacement'. La dernière analyse concerne le format immédiat de branchements relatifs, permettant d'opérer un branchement à une adresse proche sans nécessiter le codage de cette adresse en valeur absolue (large de 32 bits).

Largeur du format immédiat court

Les valeurs constantes d'une opération arithmétique ou logique peuvent être codées de deux façons :

1. soit la valeur est stockée en mémoire, dans la section «données initialisées»,
2. soit elle est directement codée dans l'instruction, utilisant alors un champ immédiat.

Dans le premier cas, la constante utilise systématiquement un mot-mémoire entier (ici 24 bits) quelle que soit sa valeur, et nécessite un accès mémoire à une adresse directe (32 bits). Le coût en temps d'exécution, ainsi qu'en espace mémoire occupé est alors assez pénalisant. Au contraire, la seconde solution ne nécessite aucun espace mémoire supplémentaire, et l'instruction elle-même peut être plus compacte. Cependant la largeur du champ immédiat dans cette instruction ne doit pas être trop grande pour ne pas affecter inutilement les performances lorsque les valeurs constantes sont faibles. Le nombre de bits alloués à ce champ immédiat définit un intervalle, au-delà duquel une valeur constante doit être codée selon la première technique.

Le choix de la largeur du (ou des) champ(s) immédiat(s) n'est pas évident a priori. De plus, la répartition des valeurs constantes dans un programme n'est absolument pas uniforme. Par exemple certaines valeurs (0, 1, -1, 8, 16, 255,...) sont souvent utilisées, pour des initialisations ou des masquages bit-à-bit. Une exploration systématique de toutes les valeurs possibles de la largeur de ce format immédiat, sur un grand nombre de programmes, doit permettre de déterminer une valeur optimale en termes de taille de code.

Deux formats immédiats sont définis. Le premier a une largeur de 24 bits, afin de pouvoir coder n'importe quelle valeur constante. Un second format, appelé immédiat court, est disponible pour le codage des valeurs constantes intermédiaires. L'intervalle à l'intérieur duquel une constante C peut être codée avec le format d'immédiat court est obtenu à partir de la largeur en bits l de ce format :

$C \in \left[-2^{l-1}; +2^{l-1} - 1\right]$

L'expérience consiste à produire quinze versions du compilateur différentes, où seule la largeur du format immédiat court change. Elle varie de 0 à 14 bits, par pas de 1. La valeur minimale 0 impose au compilateur de placer toutes les constantes en mémoire, selon la technique No. 1, nécessitant à chaque fois l'accès direct à un mot-mémoire. La dernière valeur, 14, constitue un maximum pour un encodage du jeu d'instructions sur seulement 24 bits.

Sur la figure 4.8 l'axe des abscisses représente les quinze configurations où la largeur du format immédiat court varie de 0 à 14. La taille de code décroît assez régulièrement de plus de 1% pour chaque bit ajouté au format immédiat, et ce jusqu'à 8 bits. Une plus nette diminution est à noter entre 7 et 8 bits. Cela s'explique pour un nombre plus élevé de valeurs constantes autour des puissances de 2, généralement pour opérer des masquages et des initialisations. Une seconde forte diminution de la taille de code apparaît pour une largeur de 12 bits, signifiant qu'un plus grand nombre de constantes ont une valeur proche de 2^{12}. Pour des largeurs plus élevées, de 13 et 14 bits, la taille de code ne diminue que faiblement. Le gain maximum en taille de code est logiquement obtenu pour une largeur de 14 bits, mais au prix d'un encodage très coûteux en bits.

La taille de 9 bits retenue dans le D960 semble assez bien choisie. En effet le gain par rapport à des taille faibles est notable (plus de 10%), alors que la perte par rapport à une

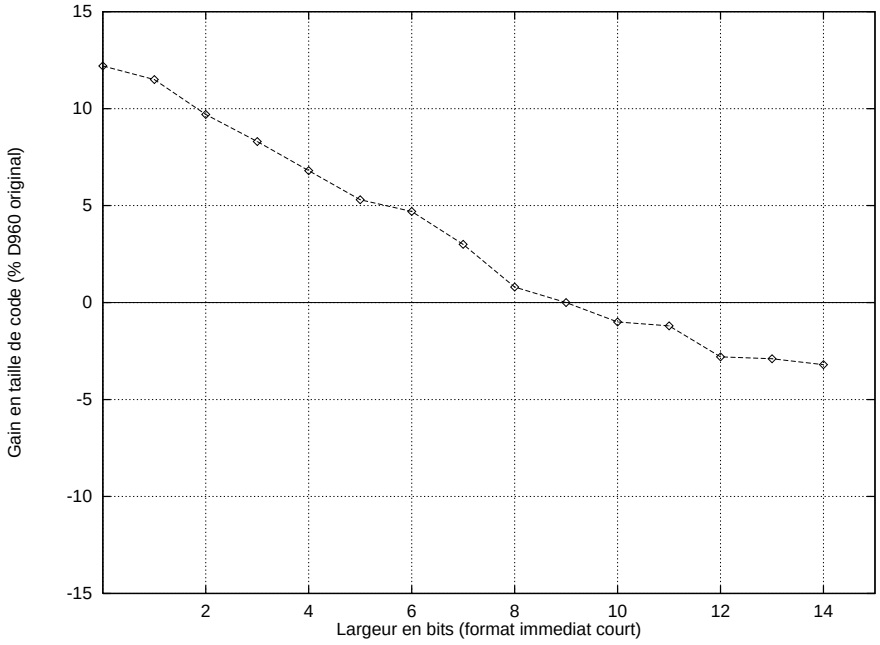

FIG. 4.8: Gain en taille de code selon la taille du format immédiat court

taille de 14 bits demeure inférieure à 4%. Si les contraintes d'encodage le permettaient, une taille de 12 bits serait également un bon choix.

Ajout d'un format immédiat intermédiaire

Un troisième format d'encodage des immédiats, appelé medium, est ajouté au jeu d'instructions. Sa taille est supérieure à celle du format court, et toujours inférieure ou égale à 14. Les deux formats court et medium varient alternativement afin de parcourir un maximum de possibilités. Le format court varie de 0 à 12, alors que le format medium varie de 2 à 14, de 2 en 2.

La figure 4.9 montre l'évolution du gain en taille de code pour plusieurs tailles du format immédiat medium (2 à 14), et cela pour plusieurs tailles du format court (0 à 12) représentées par 5 courbes différentes. Les courbes 10 et 12 se confondent, donc seule la 10 est représentée. Il apparaît logiquement que plus les tailles de formats immédiats (medium et court) sont grandes, meilleure est la densité de code. La valeur 14 pour le format medium correspond au minimum, quelle que soit la courbe, et la courbe la plus basse correspond au format court fixé à 12 bits. La plus forte décroissance de la taille de code est obtenue pour des tailles du format medium comprises entre 6 et 10 bits, alors que les valeurs extrêmes (entre 1 et 4 et entre 10 et 14) sont plus stables. Concernant le format court, les valeurs entre 5 et 12 ne présentent pas de grandes différences en taille de code (les courbes du bas sont très proches). En revanche les valeurs 3 ou 4 bits apportent un gain sensible.

Nous pouvons donc tirer comme leçon qu'un format court de **4** bits associé à un format medium de **12** bits (entre 10 et 14) apportent une taille de code réduite pour un coût d'encodage en bits pas trop élevé. Cependant, le gain obtenu par rapport au D960 original est de l'ordre de 1%, ce qui laisse penser que la solution à deux formats du D960 est assez satisfaisante.

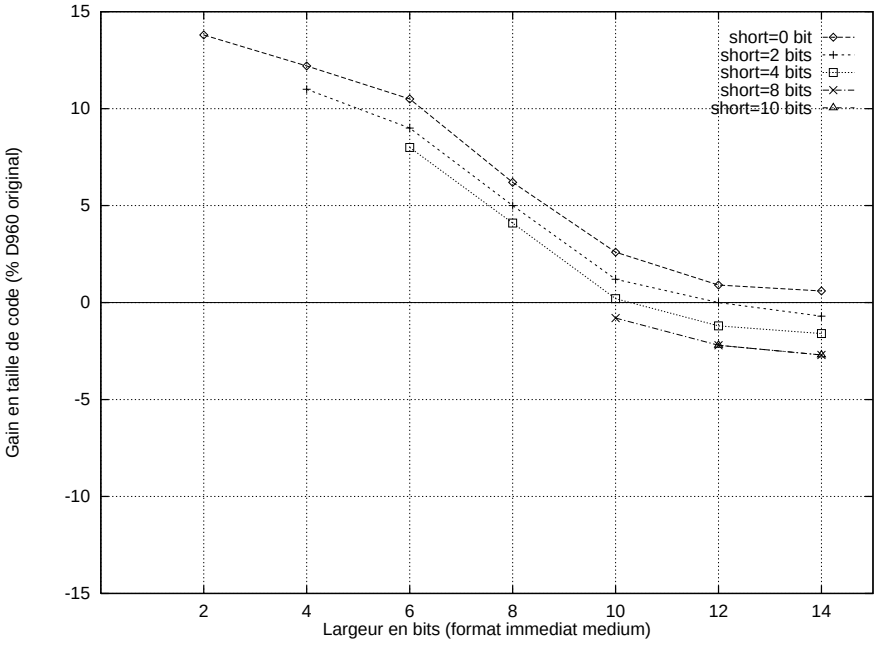

FIG. 4.9: Gain en taille de code selon les tailles de formats immédiats court et medium

Largeur des déplacements en adressage basé

L'adressage des données en mémoire peut se faire par l'ajout d'un déplacement immédiat de taille réduite à une adresse de base de 32 bits. Ce mode d'adressage est appelé 'base plus déplacement'. Il est généralement utilisé pour adresser rapidement les éléments d'une structure, par l'intermédiaire d'une unité de calcul d'adresse dédiée (ACU dans le D960). La largeur en bits de ce déplacement est volontairement réduite de manière à économiser des bits et des octets d'encodage dans le jeu d'instructions. Le choix de la valeur optimale doit être guidé par une analyse des valeurs immédiates les plus utilisées. En effet si la valeur d'un déplacement dépasse la taille maximale du champ immédiat, le calcul d'adresse ne peut être fait que par l'unité de calcul centrale (DCU) au détriment des autres opérations à effectuer, et doit être suivi d'un accès indirect par registre. Cela coûte évidemment plus cher qu'un accès 'base plus déplacement'. L'étude consiste ici à faire varier la largeur du champ de déplacement immédiat de 0 à 14 bits. La largeur 0 correspond à l'indisponibilité de l'adressage 'base plus déplacement'.

La figure 4.10 montre l'évolution du gain en taille de code selon la taille du champ de déplacement. Le gain maximum est obtenu pour une largeur de 14 bits et approche **14%** par rapport à une largeur de 0 bits, ce qui confirme l'intérêt apporté à cet aspect de l'encodage. Comme prévu, la taille de code diminue nettement lorsque la largeur du champ de déplacement augmente, jusqu'à une valeur de **7** bits. Par contre, entre 7 et 14 le gain ne dépasse pas 3%.

La largeur de 5 bits retenue dans le D960 assure une taille de code satisfaisante, même si une valeur de 9 bits apporterait un gain supplémentaire (-3%).

Longueur des branchements relatifs

Le codage des instructions de branchement relatif nécessite l'utilisation d'un champ immédiat contenant la valeur du déplacement par rapport au PC (Program Counter) courant. Pour des valeurs relativement faibles de ce déplacement, un format d'immédiat court peut être utilisé afin de réduire le nombre d'octets nécessaires au codage de l'instruction. La

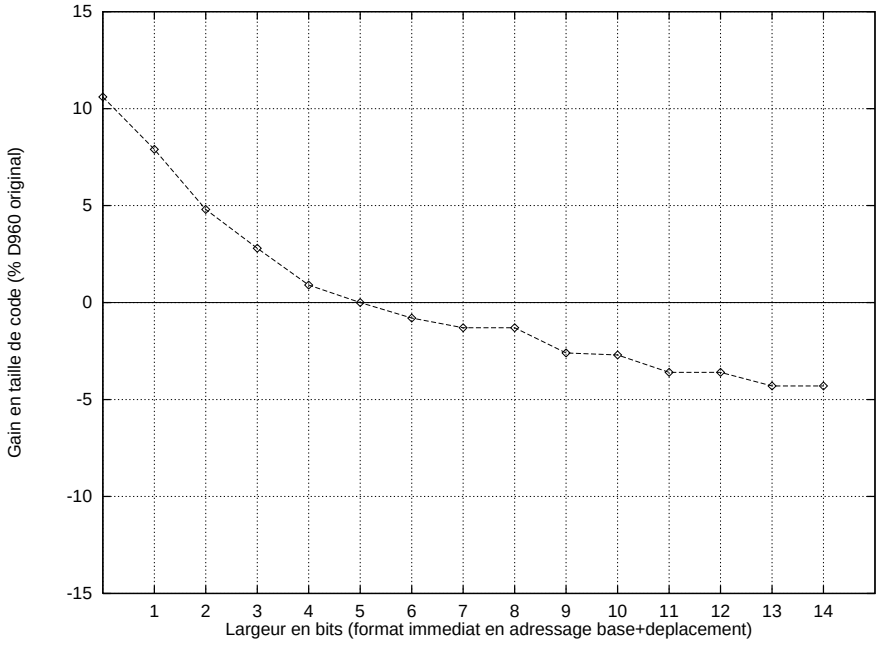

FIG. 4.10: Gain en taille de code selon la taille du champ de déplacement

détermination de la valeur de seuil distinguant les sauts courts des sauts longs est le résultat d'un compromis entre le nombre total d'instructions bénéficiant de format court et le gain en octets obtenus pour chaque saut court. La mesure du gain en taille de code totale pour un large ensemble de valeurs de la taille du format court doit permettre d'identifier pour quelles valeurs le gain est maximum. La taille du champ d'encodage du déplacement immédiat est successivement fixée de 0 à 14 bits (figure 4.11).

Pour des valeurs faibles (entre 0 et 2) de la largeur du champ de déplacement, la taille de code n'est pas réellement réduite (0.06%). Cela signifie qu'il y a très peu de sauts relatifs à moins de 2 octets (PC-2, PC-1 ou PC+1). Au contraire, la taille de code est significativement réduite à partir d'un déplacement de largeur 3 (**-2,4%**), et jusqu'à 8 bits (**-9%**). De 9 bits jusqu'à 14 bits, le gain devient minime (autour de -0,5%).

Il apparaît donc que la taille du champ immédiat dans les instructions de branchement relatif devrait se situer entre 3 et 9 bits, afin de garantir un bon compromis entre la taille de code et la consommation de bits d'encodage. Une taille de 8 ou 9 bits semble assurer une bonne densité de code (**-9%**), mais au prix d'un encodage coûteux. La solution retenue dans le D960, de 9 bits, semble particulièrement bien choisie.

4.5.3 Nombre de registres scratchs

La chaîne de compilation utilisée dans le cadre de cette exploration est basée sur un ensemble de fichiers de configuration et de spécification, contenant toutes les informations nécessaires à la compilation d'un programme C en code binaire pour une architecture donnée (voir le chapitre correspondant pour plus de détails). La modification des fichiers de configuration permet d'obtenir immédiatement un nouveau compilateur, ce qui permet d'explorer un grand ensemble de solutions en un temps réduit. Les fichiers de spécification regroupent les informations relatives à l'architecture et au jeu d'instructions, ainsi que les règles à suivre pour générer du code.

En outre, plusieurs directives peuvent être données de façon à paramétrer ou à modifier le comportement de certaines phases de la compilation (allocation de registres, génération de code, compaction des instructions). Nous nous sommes particulièrement intéressés à la

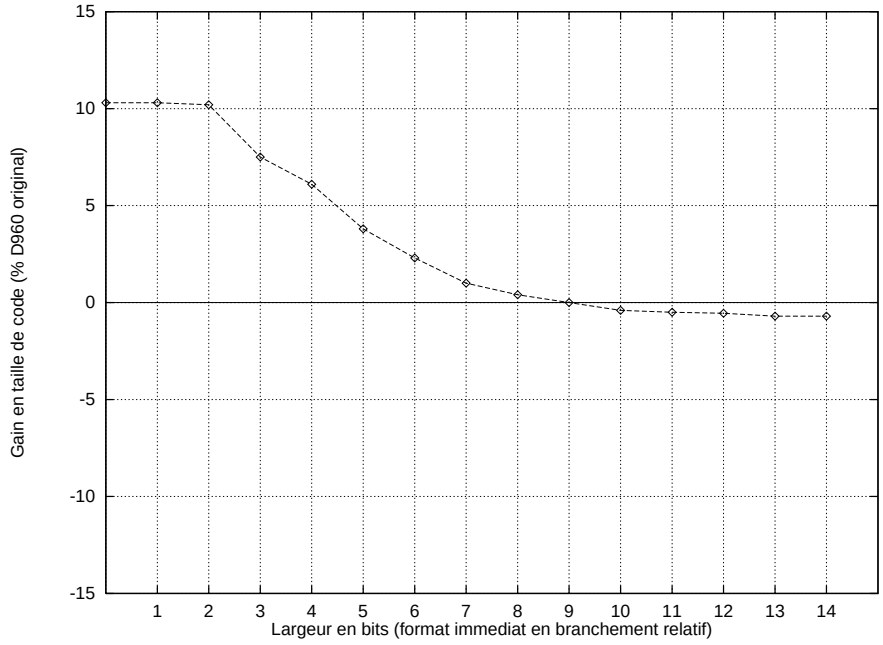

FIG. 4.11: Gain en taille de code selon la taille du champ immédiat de saut relatif

phase d'allocation de registres, de manière à analyser finement l'impact du changement de quelques paramètres sur la bonne utilisation de l'ensemble des registres disponibles. On rejoint ainsi deux des critères d'exploration primordiaux cités plus haut, le nombre et l'homogénéité des registres. Parmi l'ensemble des registres déclarés, certains peuvent être étiquetés scratchs. Cela signifie que le compilateur peut les utiliser librement, et à tout moment. De plus, ils seront systématiquement sauvegardés dans la pile lors de l'appel d'une fonction, et restaurés à son retour. Plus il y a de registres scratchs laissés à l'usage du compilateur, meilleure est la densité de code pour un bloc de code donné. Cependant, la sauvegarde systématique de ces registres est coûteuse à chaque appel de fonction. Un compromis doit donc être trouvé pour optimiser le nombre de registres scratchs.

L'expérience est conduite pour trois tailles différentes du banc de registres gauche : 8, 12 et 16. Pour chaque configuration le nombre de registres scratchs est fixé à 2, puis augmenté jusqu'au nombre maximum de registres du banc moins 2. Ainsi un minimum de 2 registres scratchs et 2 registres non-scratchs est constamment assuré. La figure 4.12 montre l'évolution du gain en taille de code par rapport au D960 original.

La faible amplitude ces courbes indique que le gain à espérer en ajustant le nombre de registres scratchs n'est pas significatif (moins de 1%). Par ailleurs, les trois courbes présentent sensiblement la même forme : pour un nombre de registres scratchs faible la taille de code est grande, puis elle diminue jusqu'à atteindre un plancher plus ou moins large selon la configuration, et enfin remonte jusqu'à un maximum. Le plancher est centré sur un nombre de registres scratchs deux fois moindre que le nombre de registres total.

Cette expérience montre que le nombre idéal de registres scratchs correspond à la moitié du nombre de registres données. La différence est toutefois très faible, et donc l'impact sur la taille de code est minime.

4.5.4 Exemple de compromis à l'encodage

L'application de chacunes des techniques de minimisation de la taille de code explorées ci-dessus implique l'utilisation de bits supplémentaires à l'encodage du jeu d'instructions. Par exemple si nous doublons le nombre de registres dans chaque banc (4 au lieu de 2), un

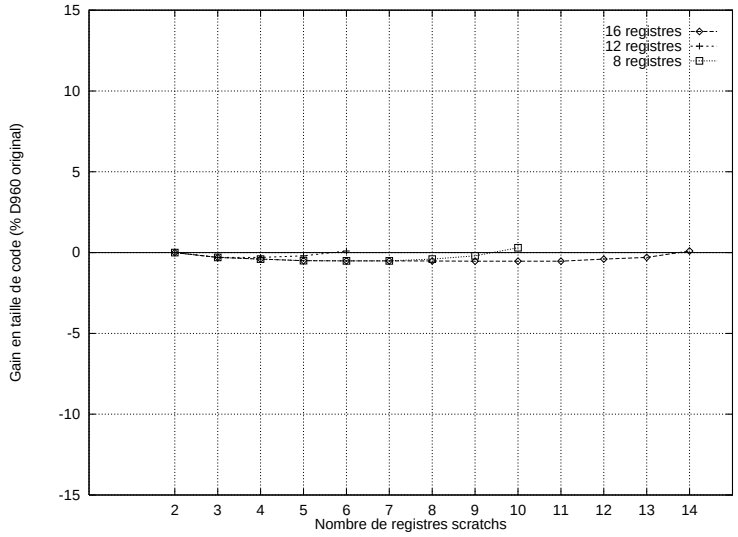

FIG. 4.12: Taille de code en fonction du nombre de registres scratchs dans le banc gauche

bit supplémentaire d'encodage est nécessaire. De même l'élargissement du format immédiat court nécessite autant de bits d'encodage que de bits ajoutés au format. Or l'encodage d'un jeu d'instructions est généralement critique, pour des raisons évidentes de densité de code (largeur moyenne des instructions). Ainsi il peut être impossible d'appliquer simultanément deux optimisations si la somme des bits supplémentaires requis par chacunes d'elles dépasse le nombre de bits disponibles dans le jeu d'instructions. Il est alors nécessaire de faire un compromis entre les deux optimisations, afin d'obtenir un gain maximum.

Nous pouvons illustrer ce compromis par un exemple. Considérons l'opération d'initialisation d'un registre par une constante, comme `LDI R1, #9`. Cette instruction contient un code-opération (LDI) et deux opérandes, le registre R1 et le champ immédiat 9. L'application des deux techniques indépendantes d'augmentation du nombre de registres et d'élargissement du format immédiat court implique l'ajout de bits d'encodage pour le champ registre et pour le champ immédiat. Le champ registre peut varier entre 1 et 4 bits : avec un bit seuls deux registres sont disponibles alors qu'avec 4 bits 16 sont disponibles (pour un banc donné). Le champ peut par exemple varier entre 7 et 10 bits (9 étant la valeur originale dans le D960). Les deux champs d'opérandes peuvent donc consommer entre 8 et 14 bits. Supposons que seuls 11 bits sont disponibles. Quelle configuration apporte le meilleur gain en taille de code ? Le diagramme représenté en figure 4.13 fournit une réponse. L'axe des abscisses donne la largeur du champ dédié au codage du registre. Il varie entre 1 et 4 bits. La somme des 2 champs étant fixées à 11, le nombre de bits alloués au champ immédiat varie donc dans le sens inverse, de 10 à 7 (cet axe n'est pas représenté). Les deux premières courbes de la légende correspondent aux gains en taille de code obtenus respectivement par la variation de la largeur du champ registre et la variation de la largeur du champ immédiat. La troisième courbe représente la somme des deux premières en chaque point, et donc le gain final obtenu pour chacune des quatre configurations.

Il apparaît que le gain maximum est obtenu pour un champ registre de 2 bits (4 registres) et donc un champ immédiat de 9 bits. La configuration suivante (3 bits registres et 8 bits immédiats) est également intéressante. Par contre les deux configurations extrêmes ne sont pas performantes en densité de code.

Cet exemple de compromis est très simplifié : nous ne considérons que deux techniques d'optimisation, et un seul format d'opération. La généralisation de ce procédé à toutes les techniques envisageables et la modélisation de toutes les contraintes d'encodage des

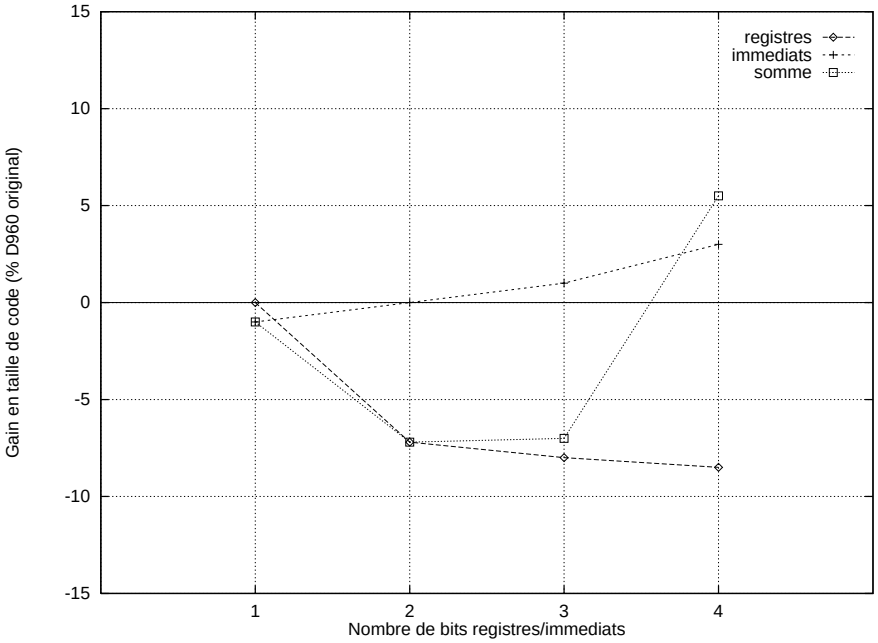

FIG. 4.13: Diagramme résolvant le compromis d'encodage

opérations est une tâche complexe, mais nécessaire pour obtenir une bonne adéquation entre les besoins d'une application et les ressources disponibles.

4.5.5 Leçons tirées de l'expérience

L'expérience menée autour de l'architecture du D960 a été enrichissante à plusieurs points de vue. En prenant comme référence de calcul du gain l'architecture originale du D960, nous pouvons identifier les quelques optimisations qui permettraient d'améliorer sensiblement la densité de code d'une architecture dérivée. Par exemple un nombre de registres données de 9 ou 12 bits serait bénéfique, associé à une homogénéité accrue. L'augmentation du nombre de bits alloués aux immédiats (constantes ou déplacement en adressage basé) serait également intéressante (3% chacun). La combinaison de ces techniques ne devrait pas poser de problèmes puisque les champs d'opérandes registres et immédiats sont indépendants. Il faut toutefois conserver à l'esprit que ces modifications ne sont envisageables que si l'encodage correspondant du jeu d'instructions reste possible. C'est pourquoi il est difficile d'estimer le gain en taille de code apporté par cette nouvelle architecture par rapport au D960. La somme des gains obtenus par chacunes techniques n'est très certainement pas réaliste. L'exemple de compromis entre le nombre de registres et la largeur du format immédiat illustre le type d'études qui devraient être menées, tenant compte de toutes les techniques d'optimisation et de toutes les contraintes d'encodage des opérations. Cela représente un travail conséquent, qui n'a pas pu être mené dans le cadre de cette thèse par manque de temps.

En deuxième lieu, l'interaction étroite entre l'architecture du processeur et les algorithmes du compilateurs a été abordée lors de l'expérimentation de plusieurs configurations de bancs de registres plus ou moins homogènes, et lors de l'étude sur le nombre de registres scratchs. Il est apparu que l'homogénéité des bancs de registres facilite nettement la compilation, et ce d'autant plus que les registres sont en quantité limitée. Lorsque leur nombre augmente les avantages de l'homogénéisation s'estompent. Le nombre de registres scratchs n'a pas d'influence sensible sur la taille de code finale, même une valeur égale à la moitié du nombre de registres total est préférable.

En dernier lieu, la mise en place d'un environnement d'exploration semi-automatisée de solutions architecturales par la compilation, a permi de mettre en évidence la puissance de paramétrisation de la chaîne Flexcc, mais aussi certaines limitations dans ce domaine. Il est par exemple apparu difficile d'envisager une exploration systématique de toutes les possibilités d'encodage des champs immédiats, en raison de la syntaxe du langage de description et de la redondance des déclarations de règles. Cette limitation a ainsi motivé l'étude d'une approche alternative, basée sur l'estimation, qui est relatée en détail dans le chapitre 5 de cette thèse.

4.6 Conclusion

Dans ce chapitre, une expérience d'exploration architecturale basée sur la re-configuration automatique de compilateurs reciblables a été décrite.

La génération automatique de compilateurs correspondant à des variations autour d'une architecture originale permet d'explorer plusieurs centaines d'alternatives, selon un ensemble restreint de critères architecturaux ou liés au jeu d'instructions.

Cet environnement d'exploration a été appliqué à une architecture DSP 24 bits, le D960, et à un ensemble d'applications logicielles significatives. Les résultats ont permi d'identifier quelques critères ayant un impact notable sur la taille du code programme, puis d'ébaucher une architecture alternative proche du D960, probablement plus performante en taille de code. La confirmation de ces performances accrues devra être apportée par la réalisation du jeu d'instructions 24 bits correspondant.

L'analyse des résultats liés à l'allocation des registres par le compilateur a permi de mieux comprendre le fonctionnement et l'impact des algorithmes de compilation sur le code produit.

Par ailleurs, l'environnement d'exploration actuel pourrait être une base de travail pour le développement d'un flot d'exploration plus global, plus générique et centré sur un coeur de compilation plus largement paramétrable. En effet la limitation qui est apparue la plus contraignante est le faible nombre de règles de compilation pouvant être efficacement paramétrées. La plupart des règles mettent en jeu beaucoup de code dispersé dans le fichier de configuration, rendant son paramétrage difficile voire impossible. Pour ne citer qu'un exemple, la paramétrisation de l'encodage des champs immédiats nécessiterait de multiplier les règles de sélection de code pour chaque format immédiat différent, et ce pour toutes les instructions où ils apparaissent. Cela a pu être fait pour trois formats différents (court, medium et long) mais la généralisation à n formats différents serait extrêmement lourde. Nous sommes par ailleurs conscient qu'une paramétrisation totale est impossible, ou alors au prix d'une standardisation des règles de compilation incompatible avec les exigences de performances (liées aux spécificités des architectures), ce qui n'est pas envisageable dans le contexte de processeurs embarqués dédiés.

Chapitre 5

Raffinement de l'encodage des immédiats

L'encodage d'un jeu d'instructions est une étape décisive dans la conception d'un processeur dédié. Il vise à coder sur un minimum de bits les instructions assembleur définies par l'architecte. La qualité de l'encodage détermine aussi bien les performances du processeur à l'exécution du code que la compacité de ce code en mémoire. Le raffinement d'un encodage existant peut être nécessaire pour réduire le coût d'un circuit, ou suivre une évolution des spécifications, ou encore être ré-utilisé pour un autre circuit. Ce raffinement nécessite plusieurs itérations, qui doivent être les plus rapides possible afin d'explorer un maximum de solutions. Le reciblage d'une chaîne de compilation complète pour chaque solution, n'est pas suffisamment rapide pour explorer chaque aspect de l'encodage. Pour ce faire, une autre approche doit être envisagée.

Ce chapitre présente un outil de raffinement de l'encodage des champs immédiats dans un jeu d'instructions. En se concentrant sur un seul aspect de l'encodage, la rapidité de reconfiguration nécessaire peut être obtenue tout en améliorant nettement le degré de précision de l'analyse produite. La prise en compte de contraintes architecturales et de directives particulières, données par le concepteur, permet d'atteindre le niveau de précision requis. Ces spécifications ont abouti à la mise au point d'un outil expérimental, ImmPlore[1]. Associé à l'interprétation d'informations de profilage, l'outil produit une estimation fine aussi bien en statique (taille de code) qu'en dynamique (performances). Au-delà de l'estimation, un algorithme basé sur un ensemble de contraintes permet de produire automatiquement un encodage des champs immédiats performant, minimisant une fonction de coût dont les paramètres sont réglés par le concepteur. Cet outil, appliqué à plusieurs processeurs de conceptions différentes a permi d'explorer des solutions jusqu'ici négligées.

Ce chapitre développe dans un premier temps les motivations et les objectifs visés par cette étude du raffinement d'encodage, puis détaille les principes et méthodes d'estimation et d'exploration automatique. Deux paragraphes exposent ensuite les résultats de l'expérimentation d'ImmPlore dans le raffinement de deux architectures DSP (le DAP et le D950). Enfin, des perspectives d'évolution sont proposées.

5.1 Motivation et objectifs

Cette section présente les motivations de ce travail sur l'aide à l'exploration d'encodage, ainsi que les objectifs que nous nous sommes fixés. La motivation essentielle concerne les limitations de l'approche courante de raffinement d'un encodage par itération du processus

[1] ImmPlore : Immediate fields encoding exploration.

de génération de code binaire. L'objectif majeur est alors de réduire ce temps de cycle itératif par le développement d'un outil basé sur l'estimation de quelques critères importants, qui ne nécessiterait pas le développement du dorsal de compilateur à chaque itération.

5.1.1 Motivations

Deux motivations principales nous poussent à nous intéresser au raffinement d'encodage des champs immédiats dans un jeu d'instructions. Lorsque nous nous concentrons sur le processus de raffinement d'un encodage existant, le temps de cycle imposé par la méthode de compilation reciblable s'avère trop long pour supporter une exploration large des solutions envisageables[72]. Il parait donc souhaitable de réduire le temps de cycle de raffinement. D'autre part, les champs immédiats occupent une grande part des bits dans un code programme (40% des instructions en contiennent), alors que leur traitement par le compilateur n'est pas facilement prédictible par le concepteur de l'application. Un retour d'informations après compilation est alors très utile [44].

Temps de cycle du raffinement

Nous appelons raffinement de l'encodage le processus consistant à modifier un encodage existant en fonction des résultats de l'analyse du code produit avec celui-ci, dans le but d'optimiser ses performances (au sens large : temps d'exécution et taille de code). Le raffinement de l'encodage d'un jeu d'instructions existant s'effectue en plusieurs itérations, chacune comportant trois étapes :

1. identification de la modification à apporter,
2. reconfiguration du fichier de spécification de l'encodage,
3. vérification du gain obtenu par analyse du code produit.

La figure 5.1 décrit ce cycle de raffinement.

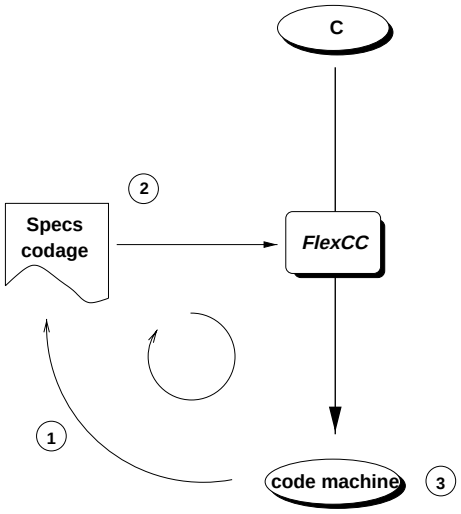

FIG. 5.1: Cycle de raffinement de l'encodage d'un jeu d'instructions

A partir d'une première version des spécifications de l'encodage, un compilateur est généré pour produire le code machine d'une application C. Ce code est analysé afin de déterminer les modifications à y apporter (phase 1). Ces modifications sont enregistrées dans une nouvelle version des spécifications de l'encodage (2), un nouveau compilateur est

généré et le code produit est à nouveau analysé pour vérifier l'utilité des modifications (3). Ce cycle est reproduit pour chaque itération du raffinement.

Il est clair que l'étape 2 est souvent la plus coûteuse en temps. Les modifications à apporter peuvent avoir un impact sur une grande partie du fichier de spécifications. De plus la validation de ces changements ne doit pas être négligée. Plusieurs expériences de développement de compilateurs pour des architectures dont les spécifications évoluent permettent d'estimer le temps de cycle de raffinement réellement observé en milieu industriel. Le tableau 5.1 récapitule quelques ordres de grandeur du temps de cycle de raffinement avec la chaîne Flexcc actuelle. La validation approfondie des modifications est exclue de ces estimations.

Processeur	Complexité	Délai mini.	Délai maxi.	Typique
MSQ	faible	1 heure	1 jour	1/2 jour
VIP	faible	1 heure	3 jours	1/2 jour
MMDSP	grande	1 heure	1 mois	1 semaine
DAP	grande	1 heure	2 semaines	3 jours

TAB. 5.1: Fourchettes du temps de raffinement d'un encodage avec Flexcc

Il ressort de ces expériences qu'un temps de cycle, en moyenne de l'ordre de quelques jours, est suffisamment faible pour suivre l'évolution de spécifications non figées sans gêner notablement le développement de la chaîne de compilation. Ce temps de cycle semble donc satisfaisant pour la conception d'un processeur dont les spécifications sont suffisamment finalisées.

Par contre, ce temps de cycle de quelques jours limite nettement l'espace des solutions envisageables dans le cadre d'une exploration pointue, lorsque l'architecture n'est pas suffisamment avancée. Une telle exploration peut nécessiter la génération de plusieurs dizaines de variantes, ce qui représente alors plusieurs semaines. Il faut remarquer que l'exploration d'architectures présentée dans le chapitre 4 a pu être menée à bien car elle se concentrait sur des critères architecturaux facilement paramétrables dans les fichiers de configuration de Flexcc (relativement à haut-niveau). Au contraire, un encodage de jeu d'instructions est trop complexe pour être paramétré, dans Flexcc. C'est pourquoi la ré-écriture manuelle est indispensable, engendrant alors un temps de cycle de plusieurs jours.

Importance des champs immédiats

L'analyse de la distribution des instructions dans une application donnée rapporte qu'en moyenne 40% des instructions comportent un champ immédiat (table 5.2). Les applications

Processeur/ application	Nb. instructions	% Instr. immédiates
DAP / audio	1070	**40 %**
D950 / gsm	14539	**40 %**

TAB. 5.2: Taux d'instructions comportant des champs immédiats

étudiées sont destinées à être embarquées dans des processeurs DSP dédiés, pour le traitement numérique du son. Ce fort taux d'instructions immédiates incite à s'intéresser particulièrement à l'encodage des champs immédiats lorsque la taille de code d'une application est une caractéristique critique.

De plus, le potentiel d'optimisation de cet encodage paraît important [32] : l'occupation efficace des champs immédiats est étroitement dépendante du code applicatif concerné. Cela

interdit donc tout espoir de définir un encodage optimal (en termes de champs immédiats) d'un jeu d'instructions a priori. Cet encodage initial ne peut être que sous-optimal, afin de s'adapter le moins mal possible à un code applicatif inconnu. Une autre raison pour laquelle les champs immédiats sont généralement mal encodés est que le compilateur est souvent développé indépendamment de l'application (en général avant). Dans ce cas, les choix d'encodage des immédiats, effectués par le compilateur, ne peuvent donc pas être implémentés en tenant compte des particularités de l'application.

5.1.2 Objectifs

L'outil d'exploration que nous nous proposons de concevoir doit répondre aux motivations exposées ci-dessus, et donc remplir un certains nombre d'objectifs. La réduction du temps de cycle de raffinement est l'objectif principal de cette approche. Un estimateur est donc chargé d'évaluer l'impact, sur quelques critères choisis, d'un encodage proposé par le concepteur. Afin d'obtenir une précision satisfaisante de cette estimation, la prise en compte de contraintes architecturales (ou liées au jeu d'instructions) doit être assurée. Des informations dynamiques, obtenues par profilage, permettent de vérifier que l'encodage exploré n'a pas d'impact négatif sur les performances de l'application. Enfin une exploration automatique, basée sur un algorithme de minimisation des bits codant les champs immédiats, doit aboutir à une solution d'encodage optimale, tout en respectant les contraintes architecturales.

Réduire le temps de cycle du raffinement

L'un des obstacles à une exploration à grande échelle des solutions d'encodage est le délai, variant de plusieurs heures à plusieurs jours, entre la définition d'une solution alternative et son implémentation effective. L'objectif de notre travail est donc de proposer une méthode d'estimation ne nécessitant pas la reconfiguration manuelle du compilateur. A la place, un estimateur évalue l'impact d'un encodage donné sur le code produit, selon quelques critères liés au champs immédiats. Ces critères sont :

- le nombre de bits utilisés par les instructions immédiates,
- le nombre de mots utilisés par les instructions immédiates,
- le nombre de cycles utilisés par les instructions immédiates.

L'estimation automatique de ces critères pour chaque encodage guide le concepteur dans la définition d'un encodage satisfaisant les contraintes.

Tenir compte des contraintes architecturales

L'estimation, pour être précise, ne doit pas se passer des contraintes d'encodage imposée par l'architecture ou le jeu d'instructions [43]. Une estimation sans contraintes ne produirait qu'une estimation des besoins de l'application, indépendamment du processeur envisagé. Cette information, bien qu'utile dans un premier temps, devient insuffisante lorsque l'on désire raffiner (donc avec précision) un encodage existant.

Nous nous basons donc sur une estimation exploitant un ensemble concis d'informations décrivant les contraintes d'encodages liées aux champs immédiats. La concision d'une telle description est indispensable pour assurer une exploration rapide, induisant un temps de cycle de raffinement réduit comme expliqué plus haut.

Prendre en compte les informations dynamiques (profilage)

Une analyse statique du code programme permet d'optimiser la taille mémoire de ce code. Cela est une priorité dans la conception d'architectures embarquées et dédiées [30]. Toutefois, l'apport de considérations dynamiques (liées à l'exécution effective de l'algorithme) permet d'anticiper l'impact de l'encodage choisi sur les performances en temps d'exécution de l'application. Si l'optimisation de l'encodage des champs immédiats n'a généralement pas pour conséquence (voulue) d'améliorer les performances, il demeure intéressant de s'assurer qu'elle ne les dégrade pas. C'est pourquoi nous souhaitons fournir, en addition de l'analyse statique du code en mémoire, une analyse dynamique qui peut être consultée par le concepteur.

Proposer un algorithme d'encodage automatique

L'exploration manuelle de l'encodage des immédiats, basée sur le raffinement successif d'une description de celui-ci par le concepteur, permet de couvrir un large spectre de solutions. Cependant, la convergence vers une solution d'encodage optimale n'est absolument pas garantie dans la mesure où les solutions sont choisies de manière totalement intuitive. Nous nous proposons donc de concevoir un algorithme produisant une solution optimale d'encodage des champs immédiats pour un ensemble de contraintes donné.

5.2 Principes et méthodes

5.2.1 Flot d'exploration

Ce paragraphe décrit l'intégration de l'outil ImmPlore dans le flot de compilation reciblable (Flexcc), puis le fonctionnement de la phase d'exploration dans ce flot.

Intégration dans le flot de compilation

L'outil ImmPlore s'intègre dans le flot de compilation existant comme indiqué sur la figure 5.2a.La chaîne de compilation, basée sur Flexcc, est décomposée en deux étapes :

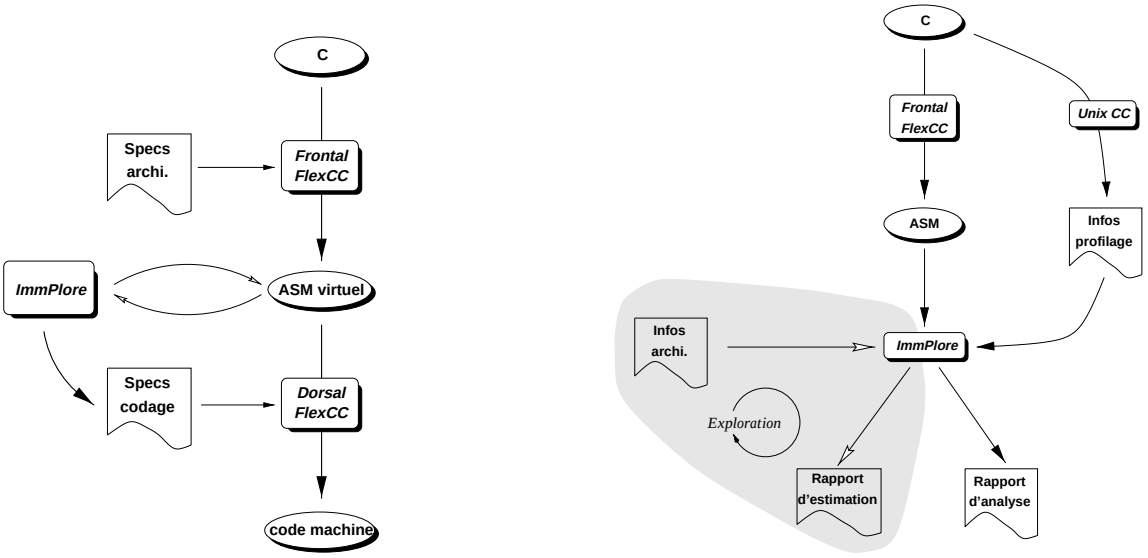

FIG. 5.2: Intégration de ImmPlore dans le flot de compilation

le frontal et le dorsal. La représentation intermédiaire entre ces deux composants est un

assembleur virtuel (ASM virtuel), proche de l'assembleur réel du processeur mais pour lequel les contraintes de parallélisme ne sont pas prises en compte. Chaque composant, frontal et dorsal, est configuré par un fichier de spécification propre. Dans un flot classique les itérations de raffinement font intervenir le fichier de spécification du codage (pour le dorsal) dont le raffinement n'est pas très aisé (cf. paragraphe 5.1.1). Au contraire, l'insertion de ImmPlore déplace ces itérations de raffinement au niveau assembleur (ASM virtuel). Lorsque l'encodage est correctement raffiné, il est réellement implémenté dans le fichier de spécification du dorsal, une fois pour toutes.

La figure 5.2b décrit le flot lié à ImmPlore, s'ajoutant au flot de compilation classique.

L'outil ImmPlore base son estimation sur le code assembleur virtuel (ASM) issu du frontal de Flexcc. Un encodage proposé par le concepteur est fourni dans un fichier, de taille réduite (`Infos archi.`). Optionnellement, l'outil tient compte d'informations dynamiques issues du profilage de l'application C sur la station de travail.

En sortie d'ImmPlore, deux types de rapports sont produits. Le rapport d'analyse est systématiquement généré, même si aucun encodage n'est proposé. Il est destiné au concepteur désireux d'analyser les besoins de l'application en codage d'immédiats, indépendamment du processeur. Cette étape peut être utile dans les toutes premières phases de la conception car instantanée. Les informations produites dans ce rapport d'analyse ont trait à la distribution des instructions immédiates dans le code, ainsi qu'à la distribution des valeurs immédiates utilisées. Si nécessaire, les informations de profilage peuvent être prises en compte dès ce niveau.

Le second rapport est un rapport d'estimation. Il ne peut être produit que si suffisamment d'informations sont fournies dans le fichier d'encodage. Il s'attache à évaluer l'impact, en termes de nombre de bits, de mots et de cycles, de l'encodage des immédiats proposé. Les résultats répertoriés dans ce rapport d'estimation servent alors de base pour la définition d'un nouvel encodage. Le cycle de raffinement continue tant que le rapport d'estimation ne satisfait pas les contraintes imposées par le concepteur.

5.2.2 Méthode d'estimation

La méthode d'estimation de l'impact d'un encodage du jeu d'instructions sur le code final, en termes de taille de code et de performances, fait intervenir trois composants (figure 5.2) :

- le code assembleur du programme à analyser,
- le fichier d'information sur l'architecture du jeu d'instructions proposé par le concepteur,
- le fichier contenant les informations issues du profilage de l'application sur la station de travail.

Code assembleur

Le code assembleur virtuel est issu de la compilation, par le frontal Flexcc seulement, de l'application écrite en C. L'analyse porte sur un seul fichier assembleur, et doit donc être reproduite pour tous les fichiers de l'application s'il y en a plusieurs.

Le code assembleur étant issu du frontal, aucun traitement lié à l'encodage des instructions n'a été effectué. En particulier, le nombre d'instructions virtuelles n'est pas exactement le même que le nombre d'instructions réelles (issues du dorsal), qui elles ont été compactées ou expansées. L'estimation du coût en nombre de mots (instructions réelles) et de cycles est assurée par l'outil, pour les instructions contenant des champs immédiats uniquement.

Fichier d'information sur l'architecture

Le fichier d'information sur l'architecture, dont un extrait est donné en figure 5.3, est composée de trois parties. La première partie définit la liste des fonctions de coût qui

```
# D950
# Definition of immediate formats

# cost functions
cost                :   Word    Cycle
weight              :   1       1

# immediate formats
immediate imm5   5  :   1       1
immediate short  6  :   1       1
immediate long   16 :   2       2

# codop classes
class ACU
subclass AX
loadsi  AX[0-1] :       short
loadi   AX[0-1] :       long

subclass AY
loadsi  AY[0-1] :       short
loadi   AY[0-1] :       long

#
class ALU
subclass L
loadsi  L[0-1]  :       short
loadi   L[0-1]  :       long

subclass R
loadsi  R[0-1]  :       short
loadi   R[0-1]  :       long
subclass A
loadsi  A[0-1]L :       short
loadi   A[0-1]L :       long
```

FIG. 5.3: Extrait d'un fichier d'information sur l'architecture (D950)

seront utilisées par l'algorithme d'exploration automatique. Ici, deux fonctions de coût Word et Cycle sont utilisées. A chacune est associée un poids (Weight) qui permet d'orienter l'algorithme vers une optimisation en taille de code, ou encore en performance.

La deuxième partie contient la déclaration des formats immédiats utilisés dans le jeu d'instructions. Pour chacun, la taille en bits du champ immédiat ainsi que les valeurs des coûts déclarés plus haut sont données. La taille en bits est utilisée pour estimer si une valeur immédiate donnée peut être codée par ce format, ou bien nécessite un format de taille supérieure. Les nombres de mots et cycles permettent d'estimer la taille de code et les performances du code final, alors que celui-ci n'est pas encore disponible (il sera généré par le dorsal Flexcc). En effet le code assembleur virtuel diffère du code final notamment sur la largeur des instructions. La prise en charge par l'estimateur de la sélection de formats

spécifiques permet de n'utiliser que le frontal Flexcc et d'estimer l'encodage que pourrait produire le dorsal.

La dernière partie du fichier d'information fournit la liste des codes-opérations des instructions contenant un champ immédiat. Pour chaque codop, une liste de formats immédiats est donnée. Ces formats correspondent à ceux déclarés dans la partie supérieure du fichier d'information. Lorsqu'un code-opération peut disposer de plusieurs formats immédiats, ceux-ci sont listés par ordre de préférence. Ainsi les formats de tailles réduites sont donnés en premier. De plus, les formats sont agencés par classes et sous-classes, définies librement par le concepteur. Cette hiérarchisation permet, lors de la lecture du rapport d'estimation, de visualiser rapidement les valeurs immédiates et les codes-opérations les plus utilisés par chaque classe. Il est ainsi possible de distinguer des autres les classes d'instructions qui exigent une optimisation.

Fichier de profilage

La technique de profilage utilisée ici fait usage de commandes et outils standards dans le monde Unix. L'utilisation d'un simulateur de jeu d'instructions spécifique n'est pas requise, ce qui accélère considérablement le temps de cycle du processus de raffinement. En effet le temps de reciblage d'un simulateur n'est pas compatible avec une étude exhaustive des solutions.

Le compilateur gcc [98] dispose d'une option d'instrumentation du code C compilé. Ainsi, lors de l'exécution de l'application sur la station de travail, un fichier additionnel (`bb.out`) est automatiquement généré. Il contient les numéros de lignes dans le code source C de tous les blocs de base qui ont été exécutés, avec pour chacun sa fréquence d'exécution. La figure 5.4 donne un exemple de contenu du fichier `bb.out`.

```
Block #42 : executed  22030 time(s)
address= 0xea60 function= class BasicFraction
BasicFraction : :round(int) const
line= 128 file=/tmp/C/lib/Fractions.C
```

FIG. 5.4: Extrait du fichier de profilage `bb.out` produit à l'exécution

Par ailleurs, le code assembleur virtuel généré par le frontal Flexcc contient, en commentaires, les lignes C correspondant à chaque groupe d'instructions assembleur. Il est alors aisé de retrouver à quel bloc de base appartient chaque instruction assembleur, et donc d'y attacher sa fréquence d'exécution.

Procédé d'estimation

L'estimation se décompose en trois étapes (figure 5.5) :

1. Le code assembleur (virtuel) produit par le frontal Flexcc est lu afin d'en extraire les instructions comportant un champ immédiat. La reconnaissance de ces instructions est assurée grâce à la liste de codes-opérations contenue dans le fichier d'information sur l'architecture. L'estimateur connaît ainsi la liste des formats immédiats disponibles pour coder la valeur immédiate de l'instruction.
2. Le premier format immédiat disponible dont la largeur est compatible avec la valeur immédiate de l'instruction est sélectionné[2] . Si aucun format ne convient, une erreur est signalée afin d'inciter le concepteur à ajouter au moins un format de taille suffisante

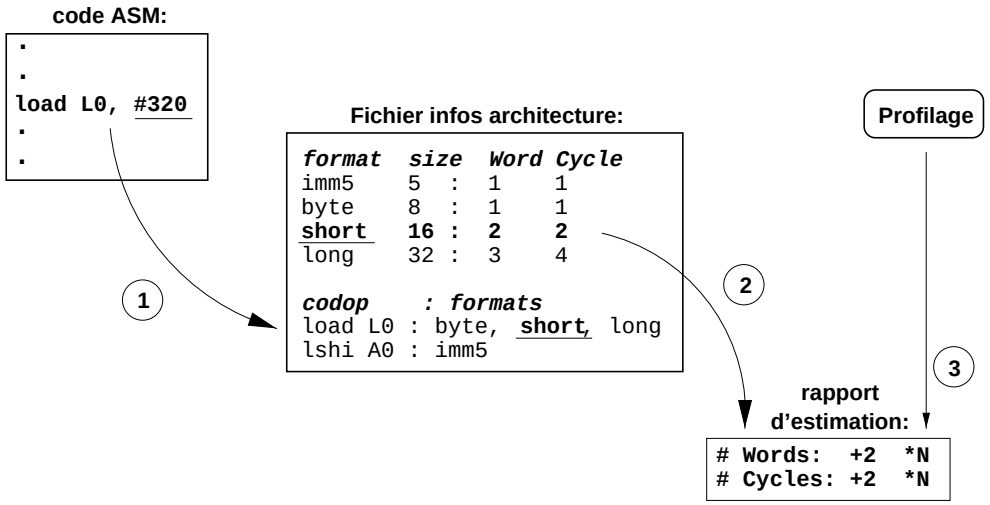

FIG. 5.5: Méthode d'estimation du coût d'un encodage des immédiats

dans l'instruction concernée. Les informations concernant le format choisi (nombres de mots et de cycles) sont dorénavant attachées à l'instruction analysée. La somme des nombres de mots et cycles obtenus pour toutes les instructions analysées (contenant un champ immédiat) permet d'évaluer, sur l'application complète, l'impact de l'encodage choisi. Le choix d'un encodage parmi plusieurs autres sera basé sur la comparaison de ces sommes entre elles.

3. Optionnellement, un fichier de profilage généré pendant l'exécution de l'application sur la station de travail en conditions réelles (grâce à des stimuli représentatifs) est pris en compte par l'estimateur. Il permet d'annoter chaque instruction par le nombre de fois ou celle-ci a été exécutée, et donc de fournir une estimation réaliste des performances en termes de nombre de cycles.

5.2.3 Présentation des résultats

Le rapport d'estimation produit par ImmPlore récapitule les principales informations concernant l'encodage des champs immédiats dans une application donnée. Dans l'exemple donné en figure 5.6, nous distinguons trois parties.

La première partie du rapport récapitule les informations générales (nom de l'architecture explorée, nom du programme analysé, nombre d'instructions au total et nombre d'instructions contenant un champ immédiat). Le pourcentage d'instructions immédiates permet d'évaluer rapidement l'impact que peuvent avoir les optimisations évaluées.

La deuxième partie du rapport classe les valeurs immédiates selon les codes-opérations qui les utilisent. Ainsi nous distinguons immédiatement les codes-opérations devant être optimisés en priorité (les plus fréquents). Pour chaque codop, le nombre d'instructions le contenant est donné, ainsi que son pourcentage par rapport à la totalité des codes-opérations. Enfin, la colonne Range indique l'excursion maximale des valeurs immédiates observée pour le code-opération correspondant. Cette information peut servir à repérer les codes-opérations dont les champs immédiats varient très peu, voire pas du tout. Il est dans ce cas intéressant d'envisager un codage de la valeur dans le code-opération de l'instruction, et non en tant que valeur absolue. En utilisant l'option adéquate, une liste

[2]La sélection du plus petit format immédiat convenable se base sur la supposition que le compilateur fera le même choix.

```
------------------------------------------------------------------
                IMMEDIATE FIELDS USAGE REPORT
------------------------------------------------------------------
Architecture definition file... : d950.def
Application assembly file...... : gsm_implode.vop

Total number of Assembly Instructions :   2816
Total number of Immediate Instructions :  1230  (43%)

Immediate values usage (sorted by codop) :
-----------------------------------------
Codop                   # Instruc.        %              Range
------------------------------------------------------------------
loadi IX0,#               540           43%               321
loadsi IX0,#              482           39%                63
loadsi R0,#                81            6%                30
asl A0,A0,#                54            4%                 5
loadsi IX1,#               41            3%                62
loadsi R1,#                14            1%                62
asr A0,R0,#                10            0%                 0
loadi R0,#                  4            1%                81
loadi IX3,#                 2            1%               772
loadi R1,#                  1            0%                 0
asr A0,A0,#                 1            0%                 0
------------------------------------------------------------------

Costs (sorted by immediate type) :
---------------------------------
Type            %       Nb. I   Bit     Word    Cycle
------------------------------------------------------------------
short           50%     618     3708    618     618
long            44%     547     8752    1094    1094
imm5            5%      65      325     65      65
------------------------------------------------------------------
Total           100%    1230    12785   1777    1777
------------------------------------------------------------------
```

FIG. 5.6: Exemple de rapport d'estimation (algorithme GSM sur le D950)

détaillée des valeurs immédiates utilisées pour chaque code-opération peut être affichée. Son étude permet d'identifier les valeurs redondantes. De même, un classement par fréquences décroissantes peut être produit pour toutes les valeurs immédiates utilisées. La version complète d'un rapport d'estimation de ce type peut être trouvée en annexe.

Ces deux premières parties du rapport d'estimation sont produites même si aucun fichier d'information sur l'architecture n'est disponible. Il s'agit alors d'un rapport d'analyse pure. En effet, la lecture du fichier assembleur se base sur la reconnaissance des valeurs immédiates, dont la syntaxe est unique dans Flexcc. Une fois ces valeurs reconnues, les instructions les contenant sont isolées et les codes-opérations correspondant répertoriés. Il est ainsi possible de produire la liste des codes-opérations utilisant des champs immédiats et les quelques statistiques associées, sans requérir le fichier d'information.

En revanche, la troisième partie du rapport concerne l'utilisation des formats immédiats dans les instructions, tels qu'ils sont définis dans le fichier d'information. Nous obtenons alors la liste classée des formats utilisés, et pour chacun :

- le pourcentage d'utilisation par rapport au nombre total d'instructions immédiates,
- le nombre d'instructions utilisant ce format,
- le nombre total (sur tout le programme) de bits utilisés par ce format,
- la valeur totale, sur l'application complète, des coûts tels qu'ils ont été définis dans le fichier d'information (ici le nombre de mots et le nombre de cycles).

Le nombre total de bits utilisés pour le codage des immédiats dans l'application est une bonne mesure (précise) de l'efficacité de l'encodage. Minimiser ce nombre permet de libérer des bits de codage dans le jeu d'instructions, permettant soit d'ajouter de nouvelles instructions, soit d'étendre certains formats, soit de diminuer la largeur de certaines instructions. Ce nombre total de bits peut être utilisé pour sélectionner un encodage parmi plusieurs possibles.

Les valeurs des coûts sont particulièrement utiles pour mesurer une autre grandeur que le nombre de bits. En effet, il n'est pas toujours possible de réutiliser des bits libérés ça et là. Mesurer le gain apporté à ce niveau n'est alors pas suffisant. Par contre, le coût associé au nombre de mots utilisés par un format permet d'étudier l'impact au niveau supérieur. En analyse statique (sans fichier de profilage), le nombre de cycles n'est pas significatif. L'apport de l'information de profilage permet d'évaluer l'impact sur les performances de tel ou tel encodage des immédiats. Généralement, cette mesure est utile pour confirmer qu'un encodage différent ne dégrade pas les performances. Cette analyse de performances ne concerne que les immédiats et ne saurait être extrapolée à l'ensemble du code.

5.2.4 Algorithme d'exploration

L'exploration automatique des formats constants a pour but d'aider à la définition d'un codage optimal des constantes, tout en tenant compte de contraintes imposées par le concepteur.

Méthode

Un encodage de départ est fourni par le concepteur. Il peut être extrêmement simple (un seul format de grande taille), ou au contraire refléter les contraintes d'encodage imposées par le jeu d'instructions existant. Ces contraintes peuvent porter sur le nombre de formats immédiats existants et leurs tailles, mais aussi sur l'importance relative des coûts en performances et en taille de code. Le concepteur fournit également le nombre maximum de formats immédiats autorisés.

A partir de cet encodage initial, l'algorithme rajoute un à un de nouveaux formats immédiats jusqu'à atteindre le nombre maximum fixé. A chaque étape, la fonction de coût calculée en fonction des paramètres fournis est minimisée.

Algorithme

L'objectif est de trouver, pour une application donnée et avec un nombre de formats immédiats n fixé à l'avance, les tailles en bits de chacun des n formats de sorte que le nombre de bits effectivement utilisés pour coder les valeurs soit minimisé. Par exemple si la valeur 6 (codée sur au minimum 3 bits) est en réalité codée dans un format de 5 bits, nous considérons que 2 bits sont gaspillés.

Graphe des formats virtuels et formats réels Toute valeur immédiate non-signée[3] peut être codée sur un nombre minimum de bits n pour lequel la valeur est inférieure à 2^n. Si cette valeur est codée par un format de taille supérieure $m > n$, nous considérons que $(m-n)$ bits sont gaspillés. Nous appelons format virtuel f_n un format qui regroupe toutes les valeurs immédiates qui peuvent être codées sur n bits sans gaspillage. Pour une application donnée, et donc un ensemble de valeurs immédiates données, autant de formats virtuels que nécessaire sont définis. L'ensemble des formats virtuels nécessaires à une application donnée est représenté par un graphe. Chaque noeud correspond à un format virtuel, et son poids est valué par la somme des fréquences de toutes les valeurs immédiates codées par ce format virtuel (figure 5.7a).

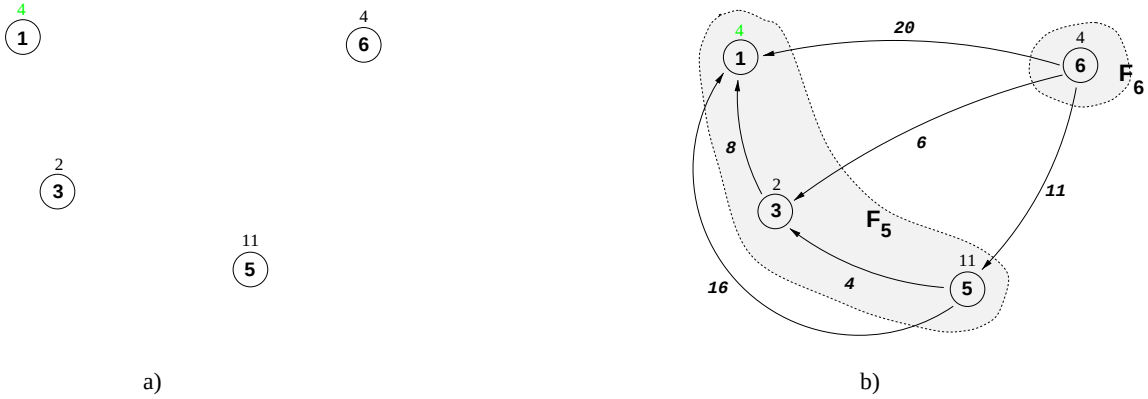

FIG. 5.7: (a) Graphe des formats virtuels valué par les fréquences d'immédiats ; (b) graphe colorié en deux formats réels, 5 bits et 6 bits

Si tous les formats virtuels étaient effectivement implémentés dans l'encodage du jeu d'instructions, celui-ci serait optimum (pour les champs immédiats). Une solution réelle utilise généralement moins de formats, exigeant alors de coder certaines valeurs par un format non minimum. Un format réel (qui sera effectivement implémenté dans le jeu d'instructions) est donc un regroupement de un ou plusieurs formats virtuels. C'est un sous-graphe du graphe principal. Si par exemple nous définissons deux formats réels F_5 et F_6 de tailles 5 et 6 bits, le graphe principal est divisé en deux sous-graphes comme sur la figure 5.7b.

Coût d'absorption et coût de regroupement L'évaluation du coût d'absorption d'un format par un autre permet de choisir entre plusieurs configurations de formats réels, celle pour laquelle un minimum de bits est gaspillé. Considérons un format de 5 bits f_5. Si aucun

[3]Pour simplifier nous considérons dans cet exposé que toutes les valeurs sont positives.

autre format, de taille inférieure, n'est disponible toutes les valeurs inférieures à 2^5 doivent être codées par ce format. Cela signifie que le format f_5 doit absorber tous les formats inférieurs (f_1, f_2, f_3, f_4), de façon à coder toutes les valeurs correspondant à ces formats. Or les valeurs immédiates du format f_3, lorsqu'elles sont codées sur 5 bits, introduisent un gaspillage de 2 bits (5 bits consommés moins 3 bits effectivement utilisés). En outre, plus la fréquence des valeurs du format f_3 est élevée, plus le coût est grand. C'est pourquoi le coût d'absorption d'un format par un autre est exprimé par le produit de la distance (différence en bits) entre les deux formats avec la fréquence du format absorbé.

Dans notre exemple l'absorption du format 3 par le format 5 coûte $(5-3)*2=4$. Il est alors possible d'estimer le poids de tous les arcs du graphe principal par le coût d'absorption d'un format par un autre. Sur la figure 5.7b, le format f_5 peut absorber les formats inférieurs (f_1, f_3). Deux arcs sont donc tracés vers les noeuds 1 et 3. Ces arcs sont valués par les coûts d'absorption des formats f_1 et f_3 par f_5, à savoir $4*4=16$ pour f_1 et $2*2=4$ pour f_3.

Algorithme d'affectation des formats réels L'affectation des formats réels, dont le nombre est fixé à l'avance (au minimum un), consiste à choisir la taille en bits de chacun et donc d'y affecter un format virtuel. L'algorithme affecte chaque format un par un, jusqu'à ce qu'il n'y en ait plus. Les différentes étapes s'exécutent dans l'ordre suivant :

1. Affecter un format réel au format virtuel le plus grand (pour être sûr de coder les valeurs immédiates les plus élevées).
2. S'il n'y a plus de format réel à affecter, terminer.
3. Affecter un nouveau format au format virtuel pointé par le format précédent et dont la somme des arcs entrants est la plus grande.
4. Supprimer les arcs venant des formats supérieurs.
5. Choisir le format virtuel libre de plus grand coût et aller en (2).

L'objectif de l'algorithme est de faire disparaître les arcs de plus fort coût, correspondant au plus grand nombre de bits gaspillés. Appliquons cet algorithme à l'exemple de la figure 5.7, en considérant que trois formats réels sont disponibles. La figure 5.8 représente les étapes d'affectation des formats.

1. Le format virtuel le plus élevé est le format f_6. Un format réel F_6 y est donc associé (figure 5.8a).
2. Il reste encore deux formats à affecter.
3. Le format virtuel f_1 est pointé par un arc venant du format réel précédent F_6, et de plus la somme de ses arcs entrants est la plus forte (20+16+8=44 est supérieur à 6+4=10 pour f_3 et à 11 pour f_5) donc un nouveau format réel F_1 est défini (figure 5.8b).
4. Les arcs venant des formats f_3, f_5 et f_6 sont supprimés (figure 5.8c).
5. Parmi les deux formats 3 et 5 restants, le format f_5 possède le coût le plus grand (11 contre 6+4=10). Un dernier format réel F_5 est donc défini (figure 5.8d).

Ce résultat traduit bien le fait que si l'on définissait un format réel F_3, le format f_5 devrait être absorbé par le format existant F_6, avec un coût correspondant au gaspillage de 1 bit pour chaque valeur du format f_3 codée (fréquence 11). Au contraire la définition d'un format réel F_5 implique l'absorption du format f_3 par celui-ci, avec un coût restant de seulement 4.

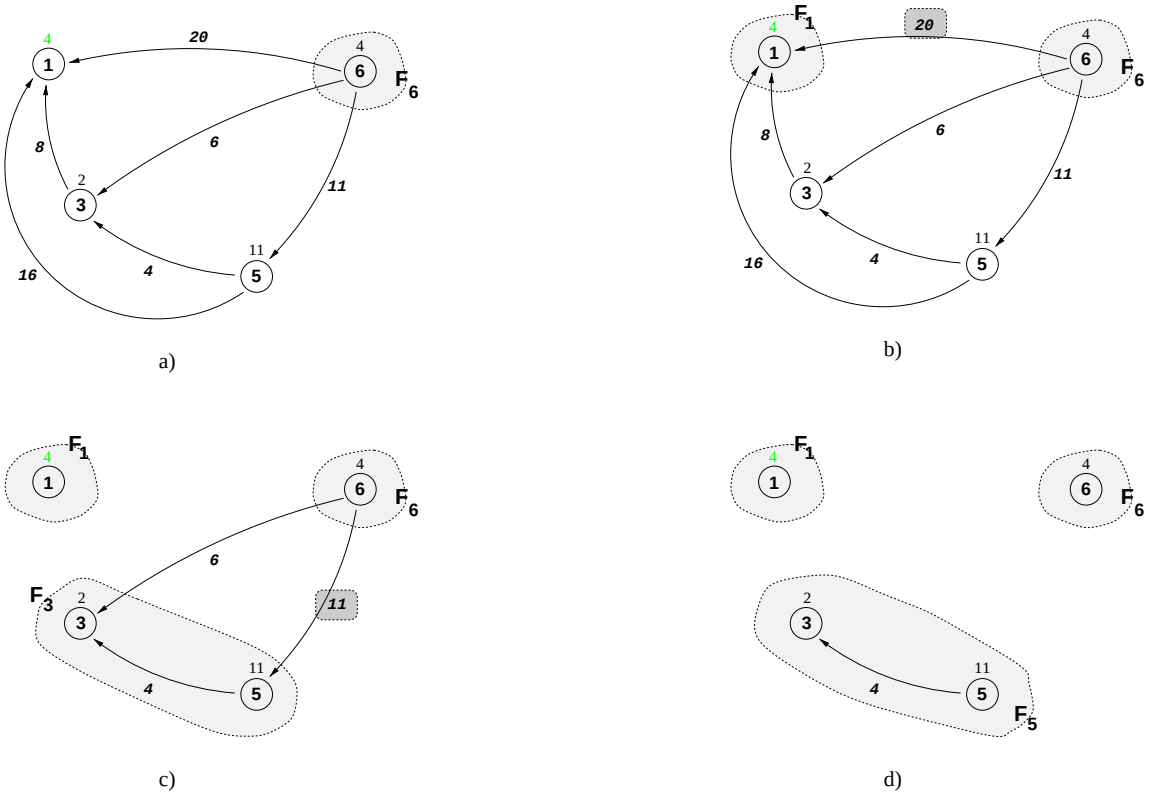

FIG. 5.8: Exemple d'affectation de trois formats réels par l'algorithme

Validation de l'algorithme

Convergence Dans un premier temps, nous voulons nous s'assurer que l'algorithme converge effectivement vers la solution de plus bas coût. Nous utilisons pour cela une architecture dérivée du D950. Elle utilise autant de formats immédiats que nécessaire (de 1 à 16 bits). Il est clair que cette dernière architecture représente l'encodage de plus bas coût, puisqu'aucun bit n'est gaspillé dans le codage d'une valeur par un format trop grand (tous les formats sont disponibles).

L'expérience consiste à proposer une architecture de départ conforme au D950 original, utilisant 3 formats immédiats. En autorisant successivement l'utilisation de 4, 5 puis jusqu'à 16 formats, nous nous attendons à ce que l'algorithme d'exploration d'ImmPlore aboutisse à un encodage dont le coût atteint celui de l'architecture de plus bas coût.

Le tableau 5.3 récapitule les étapes successives de l'exploration, et les coûts obtenus au fur et à mesure.

La différence entre les coûts de départ de l'architecture explorée et l'architecture originale provient du fait que le format immédiat (sur 5 bits) n'est pas utilisé par toutes les instructions mais réservé aux opérations de décalage. L'algorithme d'exploration ne tient pas compte de cette restriction. Il en résulte une évaluation optimiste au départ, qui devient réaliste dès que le format de 5 bits est rendu disponible pour toutes les opérations par l'algorithme. Nous vérifions que la solution de moindre coût (**6579** bits) implémentée dans l'architecture idéale est effectivement trouvée par l'algorithme, et ce dès que le nombre de formats autorisés atteint **11**. Les formats supplémentaires entre 12 et 16 sont donc superflus.

Cette expérience à elle seule ne prouve pas l'efficacité de l'algorithme. Elle révèle d'autre part que celui-ci devrait être affiné pour prendre en compte certaines restrictions architec-

Nb. formats	Coûts estimés	Référence
3	**12276**	**12785** (D950 original)
4	9292	
5	8081	
6	7073	
7	6853	
8	6721	
9	6632	
10	6585	
11	**6579**	
12	6579	
...	6579	
16	**6579**	**6579** (D950 idéal)

TAB. 5.3: Étapes d'exploration automatique pour le D950

turales qui ne sont pas indiquées dans le fichier d'information. Il est par contre intéressant de remarquer, chiffres à l'appui, que la conception manuelle de l'encodage des immédiats peut avantageusement être aidée par un outil capable d'en évaluer précisément le coût, et éventuellement d'indiquer une direction particulière parmi toutes les possibilités de combinaison de formats.

5.3 Application au DAP

5.3.1 Description - application Audio

Le DAP (*Digital Audio Processor*) est un processeur de traitement du signal 16 bits dédié, conçu par la division DPG de SGS-THOMSON. Il est spécialement adapté à un type d'applications très précis : le traitement du son audio pour des appareils grand-public, à bas coût. Ainsi les données ont une largeur courante de 16 bits alors que le jeu d'instructions est codé sur 24 bits, autorisant un degré de parallélisme intéressant pour les fonctions de traitement du signal classiques et une orthogonalité suffisante pour une compilation C efficace. Un compilateur `dapcc` complet (frontal et dorsal) a été développé avec la chaîne Flexcc. Nous n'utilisons pour cette expérience que le frontal, générant un code assembleur virtuel proche de l'assembleur final.

Nous nous intéressons particulièrement à l'encodage des champs immédiats dans le jeu d'instructions, c'est pourquoi l'architecture du processeur n'est pas décrite ici plus en détail (se réferrer au chapitre 2). L'encodage des champs constants fait un appel à un ensemble relativement restreint de formats :

- mask8 : un format de 8 bits pour le masquage bit-à-bit (codes-opérations Set et Reset),
- data14 : données sur 14 bits pour l'initialisation des registres,
- addr12 : adresses sur 12 bits pour l'adressage direct (code-opération Load),
- offset4 : 4 bits pour l'adressage avec déplacement (codes-opérations Load et Mult).

L'ensemble du code applicatif auquel est dédié le processeur DAP est rassemblé en un seul programme, `audio.c`. Il représente environ 1000 lignes de C, comme indiqué dans la table 5.4 récapitulant les caractéristiques de l'application. Le nombre de lignes (instructions) assembleur est proche du nombre de ligne C, ce qui traduit le fait que l'application est

écrite dans un C de niveau intermédiaire, et parfois bas-niveau. Les instructions assem-

Fichier C	Lignes C	Lignes ASM	Lignes CONST	%
`audio.c`	1060	1070	433	**40%**

TAB. 5.4: Caractéristiques de l'application Audio

bleur contenant un ou plusieurs champs constants (Lignes CONST) représentent 40% des instructions totales.

L'objectif de cette expérience est d'identifier les forces et faiblesses de l'encodage des champs immédiats dans la version actuelle du DAP, pour l'application Audio, afin de proposer une alternative plus efficace (plus spécifique aux besoins de l'application) pouvant servir de base à une seconde version du circuit.

5.3.2 Analyse

La phase d'analyse a pour but de fournir au concepteur de l'application embarquée un certain nombre de mesures obtenues à la lecture du code assembleur produit par le compilateur. Ces mesures permettent de raffiner le jeu d'instructions en tenant compte très précisément de l'application visée.

Limites de l'analyse

Le prototype d'outil d'analyse utilisé pour ces mesures présente quelques limitations qui peuvent avoir un impact sur les résultats obtenus.

Le lecteur (*parser*) de code assembleur virtuel ne prend en compte qu'un seul champ constant par instruction. La plupart des instructions ne contiennent effectivement qu'un champ constant. Certaines instructions parallèles, comme Mult, peuvent contenir deux champs constants. Un seul champ sur les deux sera alors reconnu pour l'analyse.

La deuxième limitation concerne les adresses directes (codées sur 12 bits) qui ne sont pas interprétées comme des champs constants. Elles sont négligées par l'analyseur. Cela n'est pas gênant pour l'exploration du jeu d'instructions dans la mesure où le codage des adresses directes n'est pas susceptible d'être optimisé dans le cadre de notre étude. En effet, l'optimisation de cet encodage nécessiterait un traitement des constantes après décodage de l'instruction par le matériel, ce qui dépasse le cadre de l'exploration de l'encodage du jeu d'instructions.

Répartition des valeurs constantes

Les valeurs constantes codées dans un programme ne sont généralement pas réparties uniformément sur l'ensemble des valeurs possibles. En effet certaines valeurs sont plus utilisées que d'autres. L'étude de cette distribution peut aider à attribuer les codes-opérations et les champs d'encodage en respectant les besoins réels de l'application, évitant ainsi de faire des hypothèses a priori.

La répartition des valeurs constantes est analysée statiquement et dynamiquement. Statiquement, cette répartition se réfère au code du programme stocké en mémoire. Dynamiquement, la répartition révèle l'utilisation des constantes pendant l'exécution du programme, tenant ainsi compte des ruptures de séquence (fonctions, sauts, boucles).

L'outil ImmPlore est invoqué en lui fournissant le code assembleur de l'application Audio (compilé par le frontal `dapcc`), ainsi que la trace d'exécution sur la station de travail. L'outil produit alors un ensemble de résultats similaires à ceux présentés en figure 5.6,

exception faite de la partie concernant les formats immédiats puisqu'aucun fichier d'information sur l'architecture n'est encore fourni. La figure 5.9 présente les répartitions statiques et dynamiques des valeurs les plus utilisées dans l'application Audio[4].

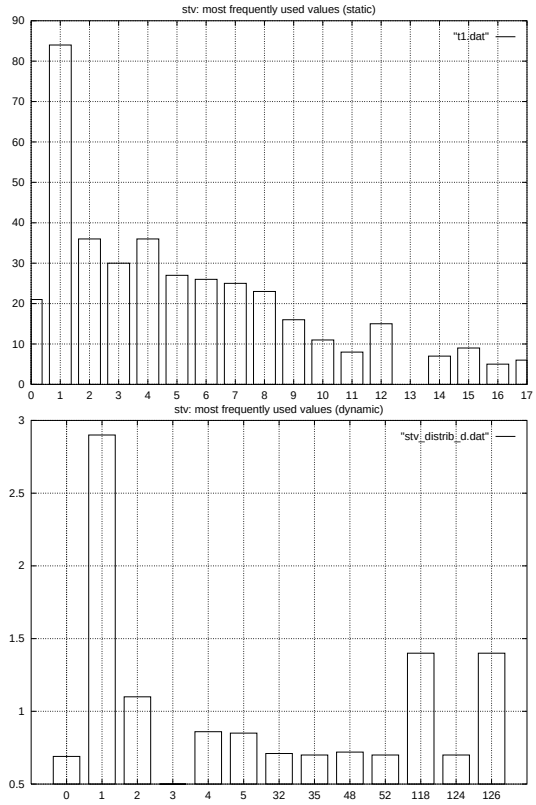

FIG. 5.9: Audio : répartition des constantes les plus utilisées (en statique et en dynamique)

En statique, les valeurs supérieures à 17 ont toutes un nombre d'occurrences nettement inférieur à 10, et ne sont donc pas représentées ici. Nous pouvons distinguer sur cette figure trois ensembles de valeurs. Le premier contient les valeurs 0 à 8. Le nombre d'occurrences de ces valeurs est nettement plus élevé que pour toutes les autres valeurs (autour de 25-30 occurrences). Dans cet ensemble, la valeur 1 a une occurrence plus de deux fois supérieure aux autres (85). Le deuxième ensemble contient les valeurs 9 à 17, qui possèdent une occurrence de l'ordre de 10. Enfin, un dernier ensemble regroupe toutes les autres valeurs (supérieures à 17), dont le nombre d'occurrences est toujours inférieur à 10, voire nul.

La même analyse en dynamique produit des résultats nettement différents. En effet apparaissent des valeurs élevées (118, 126, 32) dont le nombre d'occurrences n'est pas négligeable. Les valeurs 118 et 126 possèdent d'ailleurs un nombre d'occurrences de près de 1,5 Mi (Mega-instructions), bien supérieur à celui de toutes les autres valeurs, sauf pour 1. La valeur 1 apparaît dans près de 3 Mi, nettement plus que toutes les autres valeurs.

Si nous comparons les analyses statique et dynamique, l'interprétation diffère. Seule la valeur 1 est, dans les deux cas, très nettement plus employées que toutes les autres valeurs. Il ressort de cette analyse qu'il parait difficile de rapprocher la distribution statique de la distribution dynamique (en ce qui concerne les champs constants), et donc d'extrapoler l'une en l'absence de l'autre. En particulier la distribution statique ne suffit pas pour anticiper l'encodage des constantes en vue d'une optimisation en performances, pour laquelle l'analyse

[4] Afin de faciliter l'interprétation de ces résultats, ceux-ci sont présentés sous forme graphique et non textuelle.

dynamique reste indispensable.

La très forte occurrence de la valeur 1, observée aussi bien en statique qu'en dynamique, laisse penser que le codage de cette valeur par des champs immédiats plus grands (au moins 4 bits, qui est le plus petit format disponible) n'est pas optimal. A chaque codage de 1, un minimum de 3 bits est gaspillé. Une façon d'optimiser ce jeu d'instructions pourrait être d'encoder la valeur 1 dans le code-opération de l'instructions et non dans un champ immédiat. L'économie serait alors de plusieurs bits à chaque occurrence. Cette transformation n'est bien sûr possible que si les formats des instructions utilisant la constante 1 le permettent.

Des résultats comparables, notamment en ce qui concerne la prépondérance de valeurs basses, sont publiés dans [33] pour des applications comparables.

Répartition des codes-opérations

Un fichier d'information sur l'architecture DAP, comparable à celui présenté en figure 5.3, a été écrit. Il est disponible en annexe. Ce fichier, fourni à ImmPlore, lui permet d'interpréter le code assembleur de l'application en termes de codes-opérations et de formats immédiats réels, affinant ainsi l'analyse. La répartition des codes-opérations contenant les champs constants, aussi bien en statique qu'en dynamique, est présentée en figure 5.10.

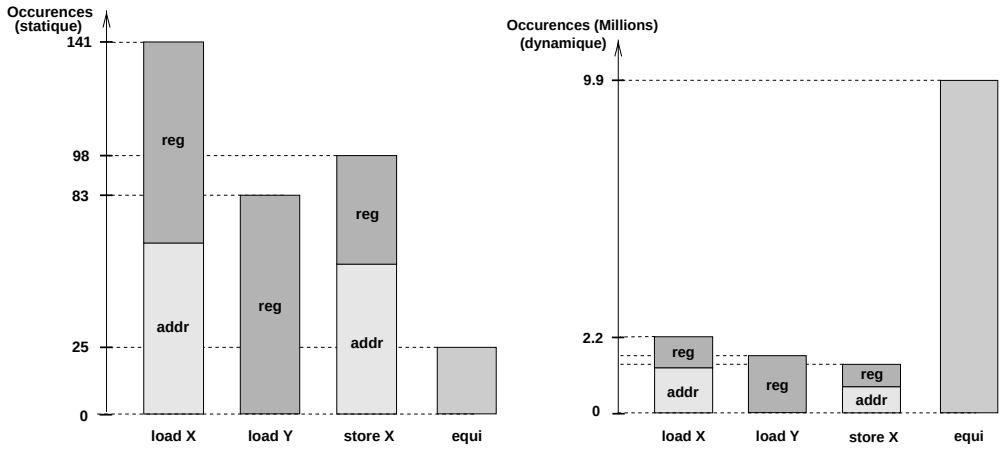

FIG. 5.10: Audio : répartition des codes-opérations les plus utilisés (en statique et en dynamique)

Ne sont représentés que les codes-opérations les plus significatifs. Tous les autres ont un nombre d'occurrences négligeable (aussi bien en statique qu'en dynamique).

L'analyse statique porte donc sur les codes-opérations `loadX`, `loadY`, `storeX` et `equi`. Les deux `load X` et `Y` correspondent au chargement de registres internes avec des valeurs situées en mémoires X et Y, respectivement. Le `storeX` correspond au stockage de la valeur d'un registre en mémoire X. Les `storeY` ne sont pas comptabilisés car ils appartiennent à la catégorie des instructions à adressage direct, ces instructions n'étant pas prises en compte par l'outil d'analyse pour les raisons exposées plus haut. L'instruction `equi` effectue une opération de comparaison entre un accumulateur et un opérande immédiat.

Nous avons distingué les deux modes d'adressage : le mode d'adressage indirect par registre avec déplacement immédiat (`reg`) et le mode d'adressage basé avec déplacement immédiat (`addr`).

Le `loadX` utilise sensiblement autant d'adressage `reg` que d'adressage `addr`. Au contraire du `loadY` qui n'utilise quasiment aucun adressage basé avec déplacement. Dans les deux cas, l'utilisation du mode `reg` est comparable. Il y a nettement plus d'accès à la mémoire

X que d'accès à la mémoire Y, concernant uniquement l'adressage avec champs constants, bien sûr.

Le `storeX` utilise un peu plus d'accès basés avec déplacement que d'accès indirects avec registre. Il a une occurrence environ deux fois moindre que l'occurrence des deux `loadX` et `loadY` réunis (44%), ce qui traduit que la plupart des opérations mémoires sont triadiques (deux opérandes en lecture et un en écriture).

Le `code-opération equi` a une occurrence assez faible en analyse statique.

L'analyse dynamique présente ici des résultats assez comparables, concernant la répartition des codes-opérations et des modes d'adressage entre `loadX`, `loadY` et `storeX`, mais avec une différence de taille : l'occurrence très forte du `code-opération equi`. Il apparaît au moins cinq fois plus souvent que n'importe quel autre codop, et encore trois plus que les deux `load` réunis. En effet le `code-opération equi` correspond à une instruction de test, utilisée dans tous les branchements conditionnels.

Répartition des formats

La répartition des formats de champs constants utilisés par l'application Audio, aussi bien en statique qu'en dynamique, est présentée en figure 5.11. L'analyse statique rapporte

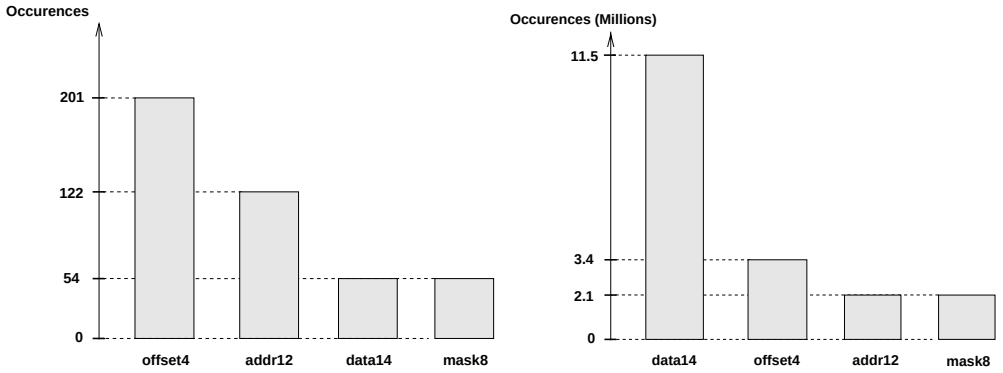

FIG. 5.11: Audio : répartition des formats immédiats (en statique et en dynamique)

que le format le plus utilisé est l'`offset4` (201 occurrences). Vient en second l'`addr12` avec 122 occurrences. Il faut se rappeler que l'application Audio contient 1000 instructions assembleur dont 400 utilisent des champs constants. Sur ces 400, 201 utilisent donc l'`offset4`, ce qui est relativement important. Les formats de données (`data14` et `mask8`) sont nettement moins utilisés.

En dynamique, l'ordre des formats est complètement bouleversé : le format de données `data14` est très nettement plus employé que tous les autres. Viennent ensuite l'`offset4`, puis l'`addr12` et le `mask8`. Nous pressentons alors que l'encodage des données (actuellement en `data14`) peut avoir un impact important (11.5 Mi sur un total de 19 Mi) sur les performances de l'application complète, si un encodage moins coûteux peut être trouvé.

5.3.3 Exploration manuelle

L'objet est ici d'illustrer la méthode d'exploration manuelle dans un cas réel, mettant en jeu l'architecture du processeur DAP. L'optimisation de l'encodage d'un jeu d'instructions, aussi bien dans le but d'améliorer les performances que de réduire la taille de code, passe par la minimisation du nombre de bits consommés pour encoder les instructions. Dans le cas des champs immédiats, le choix de la largeur maximale pour une instruction donnée détermine

cette consommation. L'optimisation de l'encodage des champs immédiats se décompose en deux étapes :

1. identification d'une ou plusieurs instructions dont la fréquence d'occurrence élevée justifie un effort d'encodage,
2. analyse des valeurs immédiates à encoder dans cette instruction pour l'application concernée, afin de déterminer la largeur maximale du champ immédiat de cette instruction,
3. Évaluation du gain en bits obtenu avec ce nouvel encodage.

Pour le DAP et l'application Audio, l'instruction la plus souvent exécutée qui comporte un champ immédiat est l'instruction `equi`. Celle-ci compte pour **52%** des instructions immédiates exécutées, alors que les autres ne dépassent pas 8%. Il parait donc judicieux d'optimiser l'encodage de cette instruction en premier lieu. L'instruction `equi` comporte un seul champ immédiat, d'une largeur unique de 14 bits. Une optimisation possible consiste à introduire un format supplémentaire pour cette instruction, où le champ immédiat est d'une largeur réduite. Bien entendu, le format de 14 bits est conservé pour coder les valeurs élevées. Pour simplifier les manipulations, un format plus court que 14 bits existant déjà dans le jeu d'instructions peut être utilisé. Le format `mask8`, d'une largeur de 8 bits et destiné originellement aux opérations de masquage, est un bon candidat.

Grâce au fichier de configuration de l'outil d'exploration ImmPlore, la déclaration d'un nouveau format pour `equi` est instantanée. La figure 5.12 représente la ligne de configuration correspondant à l'encodage de l'instruction `equi`. A gauche, seul le format `data14` est autorisé pour `equi`, comme dans le jeu d'instructions original. A droite, un format supplémentaire est autorisé, et sera utilisé de préférence car placé en premier (`mask8`). Le format `data14` demeure disponible pour les valeurs supérieures à $2^8 = 256$. L'instruction `equi` peut

```
equi ACC[0-1] : data14          ⇒          equi ACC[0-1] : mask8, data14
```

FIG. 5.12: Exemple de reconfiguration de l'encodage d'une instruction

maintenant respecter deux formats différents selon la valeur immédiate à coder (8 bits ou 14 bits). L'évaluation de ce nouvel encodage est alors lancée. La figure 5.13 représente un extrait des estimations rapportées par ImmPlore pour les deux architectures (DAP original et DAP modifié). Le rapport concernant le DAP modifié indique une consommation de 157

DAP original

Type	%	Bit
data14	60%	160535732
offset4	17%	13470888
addr12	11%	25596720
mask8	10%	16559520
Total	100	**216162860**

DAP modifié

Type	%	Bit
mask8	62%	95822120
offset4	17%	13470888
addr12	11%	25596720
data14	8%	21826182
Total	100	**156715910**

FIG. 5.13: Rapports d'estimation d'ImmPlore pour deux architectures du DAP

Mbits alors que la même application consommait 216 Mbits avec l'encodage du DAP original. Le gain de cette optimisation est donc théoriquement de **27,5%** en nombre de bits. Ce gain concerne les instructions contenant un champ immédiat, qui représente 40% de toutes les instructions. Ramené au code tout entier, ce gain est alors de **11,1%** en nombre de bits.

Cet exemple d'exploration manuelle de l'encodage des immédiats montre qu'un gain relativement important peut être facilement obtenu par un choix judicieux des formats

disponibles. Il faut cependant noter que cette optimisation est extrêmement dépendante de l'application observée, et ne s'applique donc qu'au raffinement d'une architecture dédiée. De plus, le gain final est conditionné par les possibilités de réutilisation des bits économisés. En effet si la largeur du jeu d'instructions n'est pas suffisamment flexible, il est indispensable de pouvoir attribuer les bits économisés à d'autres formats existants, sous peine de ne pas pouvoir profiter de ces bits supplémentaires.

5.4 Application au D950

5.4.1 Description - application GSM

Le D950 est un coeur de processeur de traitement du signal 16 bits, conçu par la division PPG de SGS-THOMSON. D'un usage général (non dédié à une application précise), il doit être capable d'exécuter toute une palette d'applications DSP complètes, avec des performances élevées. Son jeu d'instructions est codé sur seulement 16 bits, afin de favoriser la compacité du code programme. Ce jeu est donc très fortement encodé. La compacité s'obtient au détriment de l'orthogonalité du jeu d'instructions, très faible, rendant la compilation difficile en traitement du signal, mais aussi en code de contrôle. L'architecture de ce processeur est détaillée dans le chapitre 2. Toutefois l'analyse et l'exploration des champs immédiats ne concernent qu'une petite partie du jeu d'instructions, et ne requiert donc pas la connaissance de l'architecture. L'objectif de cette expérience est d'évaluer la qualité de l'encodage des immédiats dans le D950, afin d'en tirer un certain nombre d'enseignements qui pourrait être utiles à la définition d'une variante du D950, appelée D960.

Le choix du code applicatif utilisé pour l'analyse et l'exploration doit répondre à deux critères pour obtenir des résultats significatifs :

1. il doit être d'une taille suffisamment grande,
2. il doit être représentatif d'une application complète, comportant aussi bien du code de traitement du signal que du code de contrôle.

Le code C implémentant la norme de téléphonie mobile GSM 6.10, et en particulier les fonctions d'encodage et de décodage de la voix, remplissent ces critères. Une des futures applications développées pour la famille de coeurs issue du D950 concerne d'ailleurs précisément cette norme.

Le code utilisé pour nos expériences est disponible publiquement et a notamment été exécuté en temps réel sur une station de travail, à partir d'échantillons de voix numérisée. Il contient aussi bien du code DSP que du code de contrôle. Nous avons retenu un sous-ensemble de ce code (éliminant le code spécifique à l'intégration, propre au test sur station de travail), totalisant finalement 2500 lignes de C. La table 5.5 récapitule les caractéristiques des algorithmes retenus, indiquant notamment le pourcentage d'instructions assembleur contenant un champ constant. Ils sont classés par nombre d'instructions assembleur décroissant.

L'ensemble du code GSM a été compilé sur l'architecture D950 par le prototype D950cc issu de la chaîne Flexcc. Le code assembleur produit n'est pas directement exécutable sur D950 puisque seul le frontal du compilateur a été développé. La phase de compaction, permettant de générer les instructions réelles du D950, n'est pas inclue dans notre flot. Cependant, les instructions générées par notre compilateur sont fonctionnellement similaires aux instructions réelles, et peuvent donc servir de base réaliste pour notre expérience.

Mise-à-part le programme `gsm_option.c`, de taille négligeable et donc peu représentative, tous les programmes contiennent entre **38** et **43%** de champs constants. La moyenne pondérée est exactement de **40%**.

Fichier C	Lignes C	Lignes ASM	Lignes CONST	%
lpc.c	291	2872	1175	**40%**
gsm_implode.c	280	2816	1230	**43%**
short_term.c	430	2114	807	**38%**
gsm_encode.c	208	2084	864	**41%**
rpe.c	488	1988	828	**41%**
gsm_decode.c	126	1186	363	**30%**
long_term.c	604	1170	492	**42%**
decode.c	63	284	111	**39%**
gsm_option.c	38	25	6	**24%**
TOTAL	**2528**	**14539**	**5876**	**40%**

TAB. 5.5: Caractéristiques de l'application GSM

5.4.2 Analyse

Limites de l'analyse

L'analyse se base sur un code assembleur virtuel issu du frontal `d950cc`. Les résultats obtenus se rapportent précisément à ce code. L'extrapolation des analyses au code objet final (après compactage) doit se faire avec prudence. En ce qui concerne les champs immédiats, le compactage a un impact assez limité. Le plus souvent elle concerne les portions de code de traitement du signal, utilisant quelques instructions parallèles. Les champs immédiats interviennent alors dans l'incrémentation des registres d'adresse. Ces quelques instructions sont facilement identifiables, et peuvent aisément être isolées lors de l'analyse.

Pour des raisons de manque de temps de développement du prototype ImmPlore, les labels utilisés en adressage direct ne sont pas considérés comme des champs immédiats. Cela influe sur la validité des résultats relatifs (pourcentages), puisque toutes les adresses directes sont ignorées. Cependant, les champs d'adressage direct ne font généralement pas l'objet d'exploration, puisque leur taille est déterminée de façon directe par la taille de la mémoire adressable. Il n'est pas envisageable de diminuer la largeur de ces champs selon les valeurs utilisées. Le format de champ constant correspondant à ce mode d'adressage est le plus souvent imposé par avance.

Toutefois, pour remédier à l'imprécision des mesures relatives, deux possibilités existent : soit reconnaître tous les labels d'adresse déclarés (dans la table des symboles) et les affecter à une valeur 'non-définie' de constante, de largeur 16 bits (dans le cas du D950) ; soit procéder à l'analyse après traitement de ces labels par le compilateur, c'est-à-dire après l'édition de liens. Dans ce dernier cas, l'analyse serait très précise puisque travaillant sur les valeurs exactes, mais deviendrait alors dépendante de la localisation du programme en mémoire. Cela peut être un inconvénient majeur pour une analyse objective.

Une troisième restriction concerne l'interprétation des valeurs immédiates utilisées. La notion d'entier signé ou non-signé n'est actuellement pas gérée par l'analyseur. Il faudrait pour cela que les codes-opérations soient différents selon que la valeur est signée ou non, ce qui n'est pas le cas dans le code généré par le frontal Flexcc.

Répartition des valeurs constantes

Le premier type d'analyse possible consiste à étudier la répartition des valeurs constantes, et notamment le classement ordonné par le nombre d'occurrences. La figure 5.14 représente le nombre total d'occurrences de quelques valeurs dans toute l'application GSM retenue. L'analyse est effectuée en statique, sans information de profilage. Les valeurs

choisies sont celles qui apparaissent le plus souvent. Il apparaît clairement que les con-

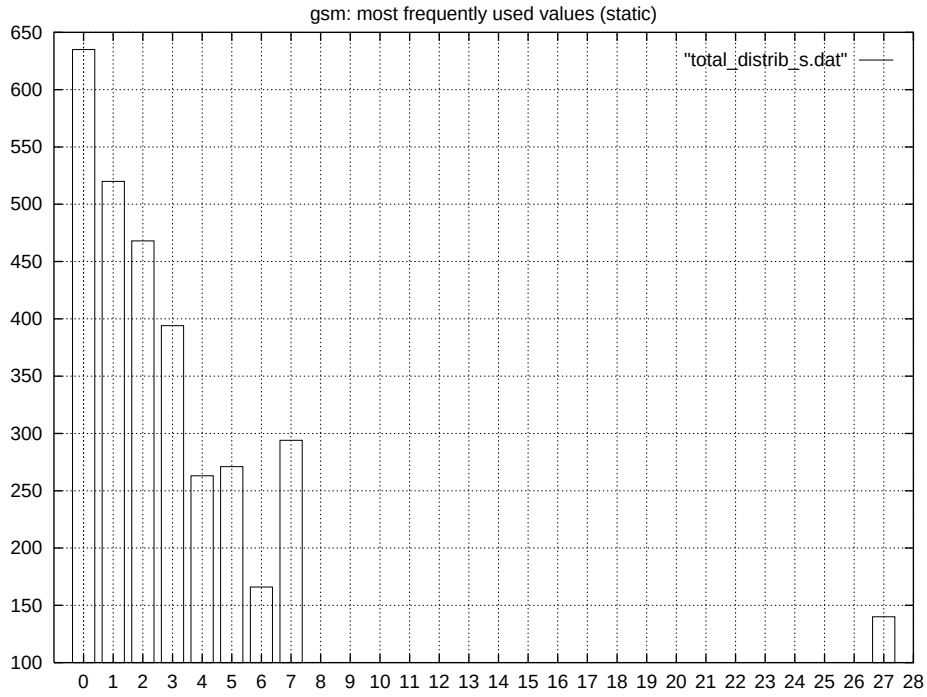

FIG. 5.14: Distribution des constantes les plus utilisées (en statique)

stantes entre 0 et 7 sont très largement utilisées, et plus précisément les valeurs entre 0 et 3.

Afin de confirmer cette tendance, il est nécessaire de refaire cette analyse en tenant compte des informations dynamiques. Pour cela l'application complète est exécutée sur des échantillons sonores suffisamment nombreux. Seul le programme `lpc.c`, très représentatif de l'application, a été profilé. L'analyse de la distribution des constantes, aussi bien en statique qu'en dynamique, produit les diagrammes de la figure 5.15.

Nous retrouvons sensiblement la même distribution, notamment en ce qui concerne les petites valeurs (entre 0 et 8). Leur fréquence est largement plus élevée que celle de toutes les autres valeurs.

Nous pouvons toutefois noter quelques différences par rapport à la distribution en statique. La fréquence de la valeur 1 atteint et même dépasse la fréquence de 0. Il y a alors une nette différence (de l'ordre d'un tiers) avec toutes les autres valeurs. La valeur 8, très peu fréquente dans l'analyse statique, atteint une fréquence élevée (proche de celle de 7) en dynamique.

Répartition des codes-opérations

Nous nous intéressons ici à la répartition des codes-opérations, c'est-à-dire aux types d'instructions qui utilisent le plus de champs constants, et avec quelles valeurs. La figure 5.16 représente les codes-opérations les plus utilisés sur toute l'application GSM, en statique. Nous remarquons immédiatement que l'instruction de chargement d'un registre par une constante (`loadi` ou `loadsi`) est de loin la plus utilisée, parmi toutes les instructions qui utilisent des constantes. Le registre le plus souvent chargé par une constante est `IX0`. Il faut ici considérer la somme des occurrences de `loadi` et `loadsi`, puisque la seule différence entre les deux instructions est la taille du champ constant.

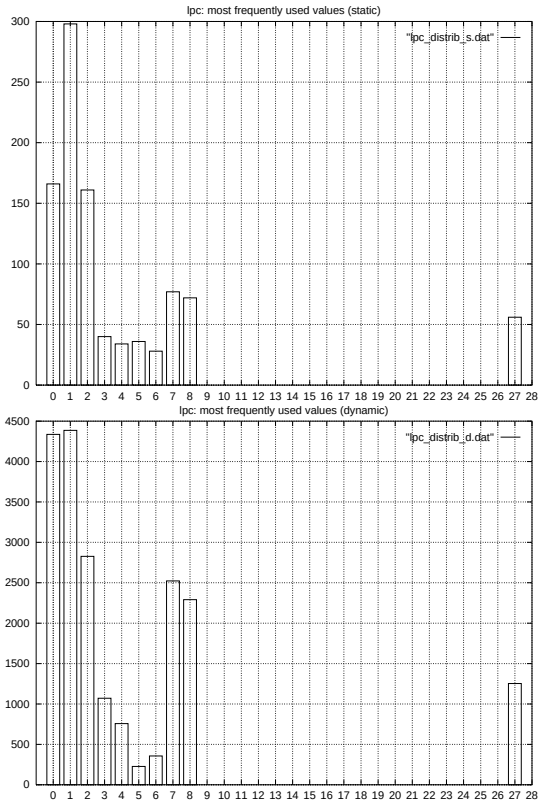

FIG. 5.15: `lpc.c` : Distribution des constantes les plus utilisées (en statique et en dynamique)

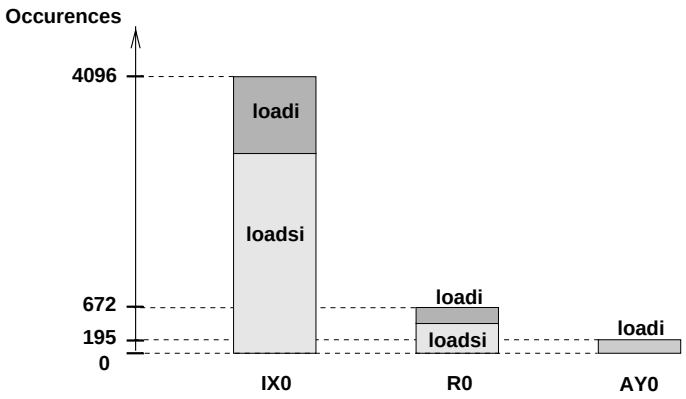

FIG. 5.16: GSM : répartition des codes-opérations les plus utilisés (en statique)

Il convient maintenant de comparer les résultats obtenus par l'analyse statique avec ceux obtenus par l'analyse dynamique. Nous utilisons pour cela à nouveau le programme `lpc.c`. La figure 5.17 présente les résultats de l'estimation. Il est intéressant de remarquer

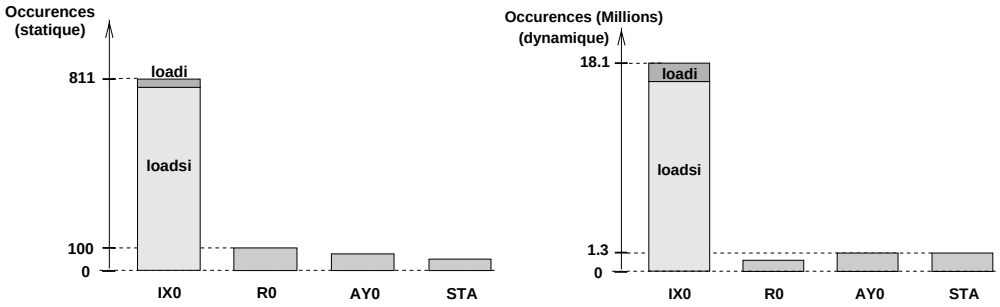

FIG. 5.17: `lpc.c` : répartition des codes-opérations les plus utilisés (en statique et en dynamique)

que la répartition des codes-opérations est quasiment la même en analyse statique qu'en analyse dynamique. Une explication possible est que l'algorithme utilisé ne met pas en œuvre beaucoup de boucles, mais qu'il suit assez linéairement le code en mémoire. La répartition des codes-opérations est comparable à celle obtenue pour l'application GSM complète, même si le `code-opération loadi` semble avoir une importance relative plus grande, par rapport à `loadsi`.

Répartition des formats

L'analyse de la répartition des formats de champs constants (`long` 16 bits, `short` 6 bits et `imm5` 5 bits) permet d'optimiser l'utilisation des formats intermédiaires, dans le but de réduire l'utilisation du format de 16 bits. Cette analyse a été faite sur les programmes de l'application GSM, en statique. Les résultats sont présentés en figure 5.18. Dans le but de

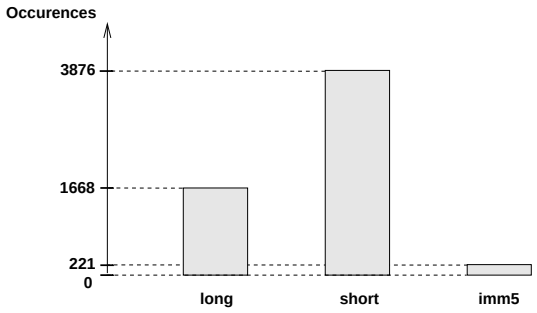

FIG. 5.18: GSM : répartition des formats de champs constants (en statique)

comparer les résultats issus de l'analyse statique avec ceux issus de l'analyse dynamique, le programme `lpc.c` est analysé dans les deux cas. Les résultats sont présentés en figure 5.19. Nous en déduisons que les répartitions des formats en statique et en dynamique sont très proches.

5.4.3 Exploration manuelle

L'exploration manuelle des formats de champs constants consiste à proposer successivement plusieurs encodages, et d'estimer pour chacun un ensemble de caractéristiques (coûts en bit, mots, et cycles, répartition des formats, ...) permettant de choisir l'encodage le plus

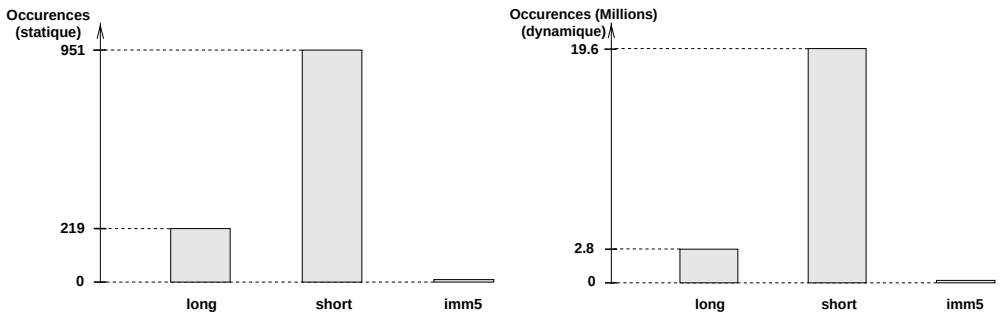

FIG. 5.19: `lpc.c` : répartition des formats de champs constants (en statique et en dynamique)

adapté. Chaque encodage se distingue par le nombre de formats autorisés, leurs tailles et leurs coûts respectifs, ainsi que par la répartition des formats selon les codes-opérations. Le fichier d'information sur l'architecture n'excédant pas quelques dizaines de lignes, l'implémentation d'un nouvel encodage se fait très rapidement (quelques minutes). L'analyse, pour une application donnée, en prenant en compte les informations de profilage, dure quelques dizaines de secondes. Il est donc très facile d'expérimenter un grand nombre de possibilités de codage.

L'exploration manuelle doit permettre de trouver le meilleur compromis entre deux grandeurs contradictoires : le nombre de formats et le nombre de bits utilisés pour l'encodage des constantes. Nous pouvons en déduire deux situations extrêmes :

- autant de formats que de tailles différentes de constantes, induisant un nombre de bits utilisés minimum (à l'exception des bits utilisés pour l'encodage des formats),
- un seul format de la taille de la plus grande constante à coder, induisant un gaspillage de bits.

Grâce au fichier d'information sur l'architecture, il est facile d'évaluer ces deux types de codage extrêmes. Le premier constitue le meilleur cas (au prix d'un encodage plus dense), alors que le second constitue le pire cas. Ces bornes délimitent la marge de manœuvre dont nous disposons en optimisant le codage des champs constants.

Mesure de la précision de l'analyse

Afin de valider la précision de l'estimation, nous dérivons de l'architecture originale D950 deux architectures, l'une avec tous les formats d'immédiats possibles (de 1 à 16 bits), l'autre avec un seul format (de 16 bits). Un compilateur C est également dérivé pour chacune des architectures. Nous analysons alors le code généré pour une même application par ces trois compilateurs. Les résultats sont alors comparés avec ceux produits par l'outil d'estimation : ce flot consiste à utiliser uniquement le compilateur C de l'architecture originale, conjointement à un fichier d'information contenant la description de l'encodage des champs constants. Il suffit alors de dériver deux fichiers d'information (un pour chaque architecture) pour obtenir les résultats d'estimation. La comparaison des coûts calculés et estimés devra permettre de mesurer la précision des résultats produits par l'estimation.

Architecture D950A Le compilateur `D950cc` implémente la version comprenant seize formats différents. Ainsi chaque constante est codée avec le nombre strictement nécessaire de bits. Les seize formats sont préfixés par `'imm_'`. Les huit programmes C sont compilés à la fois sur l'architecture D950a par le compilateur `D950acc`, et sur l'architecture originale

D950 par le compilateur `D950cc`. Dans ce dernier cas l'outil d'estimation utilise le fichier d'information `D950a.def` contenant l'encodage de l'architecture dérivée.

Le fichier d'information `D950a.def` contient la définition des 16 formats (`imm_1` à `imm_16`), et autorise tous les codes-opérations à les utiliser de manière optimale. Ceci afin de respecter l'implémentation qui a été faite dans le compilateur `D950acc`.

Nous comparons deux valeurs : le coût en bits et le coût en mots. Le coût en bits correspond aux bits utilisés pour coder les valeurs dans l'instruction. Le coût en mots correspond aux mots-instructions utilisés.

Les résultats de la comparaison sont donnés dans la table 5.6.

D950A	**Coût en bits**			**Coût en mots**		
Fichier C	**compil.**	**estim.**	**Erreur**	**compil.**	**estim.**	**Erreur**
`gsm_implode.c`	6702	6578	1,8%	1777	1777	0%
`lpc.c`	4410	4339	1,6%	1318	1318	0%
`gsm_encode.c`	5619	5495	2,2%	1407	1407	0%
`rpe.c`	2612	2541	2,7%	855	855	0%
`short_term.c`	4089	4045	1,1%	981	981	0%
`long_term.c`	2314	2301	0,6%	535	534	0,2%
`gsm_decode.c`	1141	1107	2,9%	368	367	0,3%
`gsm_option.c`	23	23	0%	8	8	0%
TOTAL	**26910**	**26429**	**1,8%**	**7249**	**7247**	**0%**

TAB. 5.6: Comparaison entre le compilateur `D950acc` et l'estimation

En ce qui concerne le coût en bits, l'erreur fluctue entre 0 et 2.9%, avec une moyenne de **1.8%**. Le coût en mots est pratiquement le même, quel que soit le flot d'analyse utilisé.

Architecture D950L Le compilateur `D950Lcc` implémente la version avec un seul format de 16 bits (`long`). La même procédure a été suivie pour comparer les résultats issus de l'estimation à partir de l'architecture originale associée au fichier d'information et les résultats issus de l'analyse de l'architecture dérivée D950L. Sur l'ensemble des programme GSM, l'erreur est systématiquement nulle : les coûts en bits et en mots sont strictement identiques.

Nous pouvons d'ores-et-déjà considérer que l'évaluation des coûts à partir du code compilé sur l'architecture originale, associé à un fichier de spécification d'encodage, fournit des résultats tout à fait réalistes. Nous pouvons donc nous en servir à la fois pour évaluer les performances finales (plus exactement l'impact de l'encodage des champs constants sur les performances de l'application complète), et pour comparer diverses solutions d'encodage.

Interprétation des résultats de l'analyse

L'analyse, en dynamique, du code généré par ce compilateur pour le programme `lpc.c` fait apparaître l'utilisation des différents formats par nombre d'occurrences décroissant. Ainsi, les constantes de 1 bit apparaissent dans **38%** des instructions, et les constantes entre 1 et 6 bits dans **88%**. Le reste des valeurs apparaît en minorité (entre 1 et 5%).

Il apparaît ainsi que le choix d'encodage des champs constants peut avoir un impact non négligeable, à la fois sur le nombre de bits utilisés dans le jeu d'instructions et sur la taille et les performances de l'application complète. Ainsi, le surcoût de la solution la plus défavorable (un seul format) par rapport à la solution optimale (16 formats) est de **85%**

en nombre de mots. Rapporté au nombre total d'instructions du programme `lpc.c`, cela représente un surcoût de **34%** en taille de code.

5.4.4 Exploration automatique

L'algorithme d'exploration automatique des formats d'encodage est expérimenté sur le jeu d'instructions du D950 afin d'évaluer l'ordre de grandeur du gain que nous pouvons espérer, en termes de bits consommés par l'encodage. Ces résultats pourront servir à concevoir un encodage plus efficace pour le successeur du D950, le D960. La diminution du coût en bits est une condition nécessaire à l'optimisation d'un encodage des formats immédiats, mais ne suffit pas. Encore faut-il pouvoir exploiter cette économie de bits en termes de mots instructions, et de cycles. Cela est conditionné par les contraintes d'encodage, dont la modélisation n'est pas aisée. C'est pourquoi nous nous proposons de procéder par étapes. Une fois que le coût en bits est suffisamment réduit par cet algorithme d'exploration, l'utilisateur peut évaluer l'encodage proposé par la méthode manuelle (via le fichier de configuration), cette fois-ci en termes de mots et cycles.

L'algorithme choisi, `lpc.c`, fait partie de l'application GSM. Il atteint près de 3000 lignes d'assembleur, une fois compilé par le frontal `D950cc`. Les informations de profilage obtenues lors de son exécution sur la station en travail avec des échantillons sonores représentatifs sont fournis à ImmPlore. L'exploration porte sur un maximum de 16 formats, permettant ainsi de voir le coût converger vers son minimum. L'encodage de départ est composé de deux formats : un format 16 bits (`long`) et un format 5 bits (`imm5`, utilisé pour les opérations de décalage). La figure 5.20 présente le rapport textuel tel qu'il est généré par ImmPlore au cours de l'exécution de l'algorithme d'exploration automatique. Le coût de l'encodage à deux formats est alors de 133 Mbits.

Lorsqu'un troisième format est exploré, le coût total chute à 98,4 Mbits, soit **26%** de moins. La taille optimale pour ce troisième format est estimée par l'algorithme à 1 bit, de façon à coder les constantes 0 et 1 seulement. Cela est confirmé par la distribution dynamique des valeurs (paragraphe 5.4.2), qui rapporte que les valeurs 0 et 1 sont les plus souvent utilisées, à hauteur de **38%**.

Remarque : Comme il a été précisé plus haut, l'algorithme d'exploration utilisé pour ces mesures ne tient pas compte des contraintes relatives aux codes-opérations. Il fait l'hypothèse optimiste que tous les codes-opérations peuvent utiliser tous les formats. Il convient donc de confirmer l'exploration automatique par l'analyse réelle de l'encodage suggéré. Il suffit de modifier le fichier de configuration pour obtenir la définition de l'encodage appelé D950u (`ultra short`), qui comporte trois formats : 16 bits, 5 bits et 1 bit. L'analyse en dynamique rapporte que le coût en bits chute de 359 Mbits à 236 Mbits, soit de **34%**, et que ce nouveau format de 1 bit a un taux d'utilisation de **36%**. Ces mesures confirment donc son utilité.

En rajoutant un quatrième format, l'exploration automatique suggère une taille de 2 bits. Nous pouvons espérer un gain supplémentaire de **11.9%**. L'ajout de formats supplémentaires se poursuit jusqu'à ce que le gain en bits s'annule. A partir de 13 formats immédiats, le coût se stabilise à 74.7 Mbits. La figure 5.21 récapitule l'évolution du coût en bits de l'encodage pendant l'exploration des formats. Cette courbe confirme que le gain est important jusqu'à cinq formats immédiats (plus de 40%), puis se stabilise pour finir à 44% avec 13 formats.

La seconde partie du rapport d'estimation (figure 5.20) récapitule, après exploration de tous les formats autorisés initialement (16), les coûts statiques et dynamiques calculés pour chacun des formats. Ces coûts représentent l'utilisation relative de ces formats par l'application.

```
* Exploring  2 :  Cost = 133329794  ( 2 used : 5 16)
* Exploring  3 :  Cost = 98442202  ( 3 used : 1 5 16)
* Exploring  4 :  Cost = 86749042  ( 4 used : 1 2 5 16)
* Exploring  5 :  Cost = 79021064  ( 5 used : 1 2 3 5 16)
* Exploring  6 :  Cost = 76384173  ( 6 used : 1 2 3 4 5 16)
* Exploring  7 :  Cost = 75304933  ( 7 used : 1 2 3 4 5 8 16)
* Exploring  8 :  Cost = 74755604  ( 8 used : 1 2 3 4 5 8 15 16)
* Exploring  9 :  Cost = 74731388  ( 9 used : 1 2 3 4 5 8 9 15 16)
* Exploring 10 :  Cost = 74725331  (10 used : 1 2 3 4 5 8 9 12 15 16)
* Exploring 11 :  Cost = 74721126  (11 used : 1 2 3 4 5 8 9 12 14 15 16)
* Exploring 12 :  Cost = 74719109  (12 used : 1 2 3 4 5 7 8 9 12 14 15 16)
* Exploring 13 :  Cost = 74719107  (13 used : 1 2 3 4 5 7 8 9 11 12 14 15 16)
* Exploring 14 :  Cost = 74719107  (13 used : 1 2 3 4 5 7 8 9 11 12 14 15 16)
* Exploring 15 :  Cost = 74719107  (13 used : 1 2 3 4 5 7 8 9 11 12 14 15 16)
* Exploring 16 :  Cost = 74719107  (13 used : 1 2 3 4 5 7 8 9 11 12 14 15 16)

SIZE : 1 bits.    COST : 464 (static)  8721898 (dynamic)
SIZE : 2 bits.    COST : 402 (static)  7795440 (dynamic)
SIZE : 3 bits.    COST : 525 (static)  11591967 (dynamic)
SIZE : 4 bits.    COST : 448 (static)  10547564 (dynamic)
SIZE : 5 bits.    COST : 365 (static)  7241320 (dynamic)
SIZE : 7 bits.    COST : 56 (static)  14119 (dynamic)
SIZE : 8 bits.    COST : 56 (static)  1063104 (dynamic)
SIZE : 9 bits.    COST : 81 (static)  36324 (dynamic)
SIZE : 11 bits.   COST : 22 (static)  22 (dynamic)
SIZE : 12 bits.   COST : 24 (static)  24204 (dynamic)
SIZE : 14 bits.   COST : 84 (static)  58870 (dynamic)
SIZE : 15 bits.   COST : 555 (static)  8086035 (dynamic)
SIZE : 16 bits.   COST : 1264 (static)  19538240 (dynamic)
```

FIG. 5.20: Rapport d'exploration automatique produit par ImmPlore (D950)

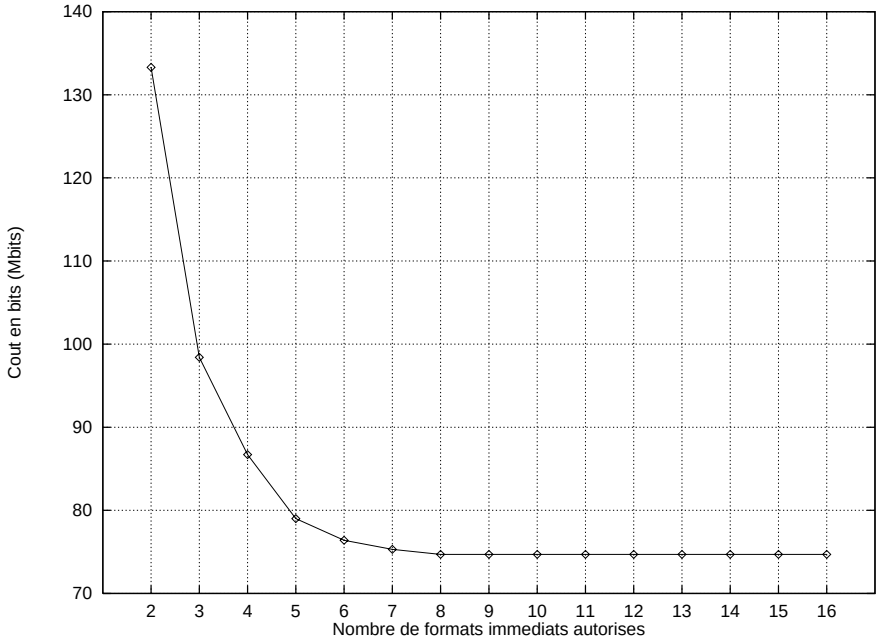

FIG. 5.21: Évolution du coût en bits au cours de l'exploration automatique (D950 en dynamique)

5.5 Conclusion

La conception d'architectures programmables embarquées et spécifiques en un temps réduit requiert le développement de nouveaux outils, associés à la compilation reciblable.

Ce chapitre a présenté un outil de raffinement de l'encodage des champs constants. Le raffinement est une étape importante dans la vie d'un processeur spécifique, afin de suivre l'évolution des spécifications ou faire l'objet d'une ré-utilisation.

L'estimation du coût d'un encodage est assurée par l'analyse d'un code assembleur pré-compilé, et ainsi ne nécessite pas la production d'un compilateur complet pour chaque solution explorée. L'utilisation d'informations de profilage a permi une analyse dynamique sans requérir un simulateur du jeu d'instructions, contribuant à accélérer le temps de cycle du raffinement. Un algorithme générant un encodage de plus bas coût sous contraintes architecturales est proposé et implémenté.

L'application à plusieurs architectures de processeurs a confirmé l'intérêt porté à l'encodage des champs constants, puisqu'un grand potentiel d'optimisations a été observé.

Les expériences menées ont permi d'identifier un certain nombre d'améliorations et extensions pouvant être appliquées à cet outil. Tout d'abord, l'algorithme d'exploration automatique actuel minimise le nombre de bits consommés par l'encodage des champs immédiats. Afin de tenir compte des contraintes liées à l'encodage du jeu d'instructions complets, et notamment à la définition d'une largeur d'instruction fixe, il serait utile d'étendre l'estimation du coût aux nombres de mots et de cycles utilisés par les formats immédiats. De même l'algorithme d'exploration ne prend pas en compte les codes-opérations dans l'affectation de formats immédiats aux valeurs. Ainsi, quel que soit le code-opération d'une instruction immédiate, le choix du format optimum ne se base que sur la valeur de l'immédiat. Dans certains cas d'encodage (comme cela a été constaté sur le D950), cette limitation introduit une erreur d'estimation. La précision globale serait donc améliorée si l'estimateur tenait compte des déclarations de codes-opérations du fichier d'informations sur l'architecture.

Actuellement, l'analyse des champs immédiats et des codes-opérations est assurée au niveau des instructions assembleur. Or certaines instructions C mettent en œuvre des modes d'adressage complexes (tableaux, indirections) qui sont compilés en plusieurs instructions assembleur. L'analyse de ces instructions une par une ne reflète alors pas exactement l'opération réellement programmée dans le source C, mais seulement une partie. L'interprétation des résultats d'analyse pourrait être facilitée si l'instruction C concernée était indiquée en même temps que les instructions assembleur correspondantes. Pour ce faire, les informations de déboguage présentes dans le code assembleur et issues du frontal Flexcc peuvent être interprétées afin d'extraire, pour chaque instruction C, les instructions assembleur correspondantes.

L'application de l'outil ImmPlore pourraient être étendue à d'autres architectures. Dans un premier temps, l'application à l'architecture D960 permettrait de comparer les résultats avec ceux obtenus lors de l'exploration d'envergure relatée dans le chapitre 4. En effet lors de cette exploration seules quelques configurations d'encodage des immédiats ont été étudiées (2 à 3 trois formats, quelques largeurs). L'outil ImmPlore permet d'explorer plus exhaustivement l'encodage, et également d'identifier une solution de plus bas coût. L'encodage produit pourrait être comparé avec celui du D960 original afin d'évaluer le gain éventuel.

Une seconde application de l'outil pourrait être l'étude d'un nouvel encodage basé sur celui du DAP. En effet, une seconde génération de ce processeur, appelée DAP2, est prévue afin d'élargir l'éventail d'applications tournant dessus. La conception d'un encodage adapté à cette évolution serait aidée par cet outil.

Enfin, la généralisation du raffinement d'encodage à des aspects autres que les champs immédiats permettrait d'adresser le problème de l'encodage de jeu d'instructions dans son ensemble. Ainsi il serait possible d'évoluer vers un outil de conception et de raffinement de jeu d'instructions complet, lié au dorsal du compilateur reciblable assurant le génération de code binaire.

Chapitre 6

Conclusions et perspectives

La conception de processeurs embarqués spécifiques en un temps réduit requiert l'utilisation de compilateurs rapidement reciblables, mais aussi de nouveaux outils de validation et d'aide à la conception. Un modèle de processeur embarqué suit un cycle de vie qui comprend souvent une étape de réduction de coût, ou sa ré-utilisation dans une application proche. Le raffinement de l'architecture ou du jeu d'instructions est un processus itératif, où le temps de cycle est un critère d'efficacité primordial.

Cette thèse a présenté plusieurs techniques visant à réduire le temps de développement du couple logiciel-processeur embarqué dans un système mono-puce. La validation fonctionnelle du logiciel a été adressée par la co-simulation haut-niveau alors que les aspects liés au raffinement de l'architecture générale d'un processeur et de son jeu d'instructions ont été étudiés.

La validation de la description haut-niveau du logiciel est assurée dans son environnement matériel réel grâce à la co-simulation C-VHDL. Sa mise au point est facilitée par l'utilisation d'outils de développement de haut-niveau courants, et accélère son déboguage par la faculté à simuler le système complet sur un intervalle de temps étendu. La prise en compte des contraintes spécifiques liées à l'intégration dans un flot existant a permis d'enrichir la chaîne de conception Unicad à SGS-THOMSON de l'outil CoSim. Ces travaux pourraient être suivis par l'extension de la co-simulation à d'autres langages de description, afin d'élargir le domaine d'application à des systèmes analogiques par exemple.

L'exploration basée sur la re-configuration automatique d'un compilateur reciblable a permis d'évaluer un grand nombre d'alternatives à une architecture existante sans requérir de développement logiciel. Elle a aidé à identifier précisément des voies pour améliorer la taille de code, en particulier pour l'architecture D960. Cette étude a également permis de mieux appréhender le fonctionnement de l'allocateur de registres du compilateur reciblable. Ces travaux représentent une première étape vers la synthèse de compilateurs et d'architectures. Pour y parvenir la totalité des paramètres du compilateur reciblable doivent être manipulés, indépendamment les uns des autres. Cela suppose un langage de description des règles et une structure de données extrêmement flexibles. Par ailleurs, des critères autres que la taille de code pourraient être pris en compte. Les performances en nombre de cycles, la consommation en énergie mais aussi la taille mémoire occupée par les données ou les accès à la mémoire-cache pourraient être pris en compte.

Enfin, l'étude détaillée d'un aspect particulier de l'encodage d'un jeu d'instructions a été menée à partir d'estimations de coût fournies par l'outil ImmPlore, dans le but de raffiner la répartition des formats immédiats en un temps réduit. L'application à plusieurs processeurs existants a révélé un potentiel certain d'optimisation, en taille de code et performances, dans ce domaine. Une suite de ces travaux pourrait être la généralisation du processus de raffinement à d'autres aspects de l'encodage (autres formats d'instructions), tout en

considérant les contraintes liées à celui-ci.

Le développement actuel de véritables familles de processeurs, chacune centrée sur une architecture particulière, au lieu d'une collection de processeurs différents réclame une méthodologie rigoureuse de généricité et de ré-utilisation, aussi bien pour le matériel que pour les outils de développement. Dans ce contexte, la disponibilité d'outils facilement reconfigurables, et même de familles d'outils est un gage de gain en productivité important. Conserver la performance des outils sur-mesure tout en réduisant le temps de développement grâce la généricité est un défi majeur pour les outils de développement d'applications embarquées de demain.

Bibliographie

[1] International Standard ISO/IEC 13818-2. *"Generic Coding of Moving Pictures and Associated Audio"*. ISO/MPEG, 1995.

[2] A. Aho, R. Sethi, and J. Ullman. *"Compilers : Principles, Techniques and Tools"*. Addison-Wesley, 1988.

[3] M. Altmae. "Hardware/Software Coverification". *Proceedings of EDA Traff*, March 1995.

[4] M. Altmae. *"Synthesia tools documentation"*. Synthesia, August 1995.

[5] M. Altmae and P. Gibson. "Hardware/Software Coverification". *Proceedings of VHDL-Forum for CAD*, April 1994.

[6] G. R. Andrews and F. B. Schneider. "Concepts and Notations for Concurrent Programming". *Computing Surveys*, 15(1), March 1983.

[7] R. Aravind et al. *"Image and Video Coding Standards"*. ATT, January 1993.

[8] P. V. Argade et al. "Hobbit : A High-Performance, Low-Power Microprocessor". *IEEE*, 1993.

[9] Argonaut. "ARC - The Argonaut RISC Core". Technical report, Argonaut, 1996.

[10] ARM. *"An Introduction to Thumb"*. Advanced RISC Machines Ltd, 1995.

[11] M. Auguin, F. Boeri, and C. Carriere. "Automatic Exploration of VLIW Processor Architectures From a Designer's Experience Based Specification". *IEEE Proceedings of CODES*, pages 108 – 115, 1994.

[12] Eagle Design Automation. *"Providing Virtual Solutions for Embedded Systems"*. Eagle Design Automation, December 1995.

[13] Eagle Design Automation. *"Co-Verification of Embedded Systems Design Using Virtual System Integration"*. Eagle Design Automation, 1996.

[14] D. Becker, R. Sigh, and S. Tell. "An Engineering Environment for Hardware-Software Cosimulation". *Proceedings of DAC*, pages 129 – 134, 1992.

[15] A. D. Berenbaum, D. R. Ditzel, and H. R. McLellen. "Introduction to the CRISP Instruction Set Architecture". *Proceedings of Spring COMPCON*, 1987.

[16] I. Bolsens. "Specification, co-simulation and Hardware/Software Interfacing for Telecom Systems". *Leuwen Codesign Course*, February 1997.

[17] J. T. Buck et al. "Ptolemy : A Framework for Simulating and Prototyping Heterogeneous Systems". *International Journal on Computer Simulation*, January 1994.

[18] Cadence. *"Overview of the Leapfrog C Interface"*. Cadence, June 1994.

[19] Cadence. *"The VHDL Design Access Library"*. Cadence, June 1994.

[20] J. P. Calvez, D. Heller, and O. Pasquier. "System Performance Modeling and Analysis with VHDL : Benefits and Limitations". *Proceedings of VHDL-Forum Europe Conference*, April 1995.

[21] R. Camposano and J. Wilberg. *"Design Automation for Embedded Systems"*. Kluwer Academic Publishers, January 1996.

[22] G. Castelli. "The Seemingly Unlimited Market for Microcontroller-based Embedded Systems". *IEEE Micro*, pages 6 – 8, October 1995.

[23] B. C. Cole. "Digital Signal Processing on Embedded Systems". *Electronic Engineering Times*, pages 46 – 86, April 1995.

[24] B. C. Cole. "Processor Architectures". *EE-times*, February 1995.

[25] M. Coram. "The Emergence of CRISC", April 1996.

[26] S. L. Coumeri and D. E. Thomas. "A Simulation Environment for Hardware-Software Codesign". *Proceedings of ICCD*, October 1995.

[27] Coware. "ABS Design Example". Technical report, Coware, 1996.

[28] CoWare. *"CoWare : Product Description"*. CoWare, 1996.

[29] D. A. Curry. *"Using C on the UNIX System"*. O'Reilly & Associates, February 1991.

[30] J. W. Davidson, J. R. Rabung, and D. B. Whalley. "Relating Static and Dynamic Machine Code Measurements". *IEEE Transactions on Computers*, 41(4) :444 – 454, April 1992.

[31] Analog Devices. *"Anatomy of an ADSP-2100 Family C Program"*. Analog Devices, 1996.

[32] M. Dolle and M. Schlett. "A Cost-Effective RISC/DSP Microprocessor for Embedded Systems". *IEEE Micro*, pages 32 – 40, 1995.

[33] P. B. Endecott. "Processor Architectures for Power Efficiency and Asynchronous Implementation". Master's thesis, University of Manchester, 1993.

[34] P. B. Endecott. *"SCALP : A Superscalar Asynchronous Low-Power Processor"*. PhD thesis, University of Manchester, 1996.

[35] R. Ernst, J. Henkel, and T. Benner. "Hardware-Software Cosynthesis for Microcontrollers". *IEEE Design & Test of Computers*, 10(4) :64 – 75, December 1993.

[36] A. Fauth, J. Van Praet, and M. Freericks. "Describing Instruction Set Processors Using nML". *Proceedings of EDTC*, pages 503 – 507, 1995.

[37] J. W. G. Fleurkens. *"Interactive Modelling and Simulation of Heterogeneous Systems"*. PhD thesis, University of Eindhoven, March 1996.

[38] D. Gajski and F. Vahid. "Specification and Design of Embedded Hardware-Software Systems". *IEEE Design & Test of Computers*, pages 53 – 67, 1995.

[39] G. Galicia and P. Warner. "Compressed Reduced Instruction Set Computing". Technical report, University of Berkeley, 1996.

[40] D. J. Le Gall. *"The MPEG Video Compression Algorithm"*. -, 1992.

[41] N. Gehani. *"C : An Advanced Introduction, ANSI C Edition"*. Computer Science Press, 1988.

[42] J. Gong and D. D. Gajski andS. Bakshi. "Model Refinement for Hardware-Software Codesign". *Proceedings of EDTC*, 1996.

[43] J. Gong, D. Gajski, and S. Narayan. "Software Estimation Using a Generic-Processor Model". *Proceedings of EDTC*, pages 498 – 502, 1995.

[44] G. Goossens, J. Van Praet, D. Lanneer, and W. Geurts. "Programmable Chips in Consumer Electronics and Telecommunications". *Proceedings of NATO ASI on Codesign*, June 1995.

[45] Mentor Graphics. "A Cosimulation Tool from Mentor Graphics for ST20", 1996.

[46] Alta Group. *"Application-Specific Design Automation Tools"*. Cadence Design Systems, July 1996.

[47] Alta Group. *"Signal Processing WorkSystem : DSP Processor Models User's Guide"*. Cadence Design Systems, June 1996.

[48] R. K. Gupta, C. N. Coelho, and G. De Micheli. "Synthesis and Simulation of Digital Systems Containing Interactive Hardware and Software Components". *Proceedings of DAC*, pages 225 – 234, 1992.

[49] R. K. Gupta and G. De Micheli. "System-Level Synthesis Using Reprogrammable Components". *Third European Conf. Design Automation, IEEE CS Press*, pages 2 – 7, 1992.

[50] R. K. Gupta and G. De Micheli. "Constrained Software Generation for Hardware-Software Systems". *Proceedings of CODES*, pages 56 – 63, 1994.

[51] R. P. Gurd. "Experience Developing Microcode Using a High-Level Language". *Proceedings of the 16th Annual Microprogramming Workshop*, pages 179 – 184, October 1983.

[52] M. Haden. "Embedded C++ Slashes Code Size and Boosts Execution". *Electronic Design*, pages 85 – 92, October 1997.

[53] M. Harrand et al. "A Single Chip Videophone Video Encoder/Decoder". *Proceedings of Intl. Solid-State Circuits Conf.*, pages 292 – 293, February 1995.

[54] P. Hilfinger and J. Rabaey. "DSP Specification using the SILAGE Language". *in Anatomy of a Silicon Compiler, Kluwer Academic Publishers*, 1992.

[55] A. Hondroudakis. "Performance Analysis Tools for Parallel Programs". Technical report, University of Edinburgh, July 1995.

[56] N. P. Jouppi. "The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance". *IEEE Transactions on Computers, Special Issue*, December 1989.

[57] A. Kalavade and E. A. Lee. "A Hardware-Software Codesign Methodology for DSP Applications". *IEEE Design and Test of Computers*, September 1993.

[58] G. Koch, U. Kebshull, and W. Rosenstiel. "A Prototyping Environment for Hardware-Software Codesign in the Cobra Project". *Proceedings of 3th Intl. Workshop on Hw-Sw Codesign Grenoble*, pages 10 – 16, September 1994.

[59] D. J. Kolson, A. Nicolau, and N. Dutt K. Kennedy. "Optimal Register Assignment to Loops for Embedded Code Generation". *Proceedings of ISSS*, pages 42 – 47, 1995.

[60] A. Kuth, J. Wilberg, H. T. Vierhaus, and R. Camposano. "Castle : A Design Exploration Environment", 1997.

[61] P. Lapsley. "NSP Shows Promise on Pentium, PowerPC". *Microprocessor Report*, pages 11 – 15, May 1995.

[62] R. Lauwereins, M. Engels, M. Ade, and J. A. Peperstraete. "Grape-II : A System-Level Prototyping Environment for DSP Applications". *Proceedings of 5th Intl. Workshop on Rapid System Prototyping*, 1994.

[63] M. T. Lee et al. "Power Analysis and Low-Power Scheduling Techniques for Embedded DSP Software". *Proceedings of ISSS*, pages 110 – 115, 1995.

[64] R. Leupers and P. Marwedel. "A BDD-based Frontend for Retargetable Compilers". *Proceedings of EDTC*, pages 239 – 243, 1995.

[65] R. Leupers and P. Marwedel. "Time-constrained Code Compaction for DSPs". *Proceedings of ISSS*, pages 54 – 59, 1995.

[66] Y. S. Li, S. Malik, and A. Wolfe. "Software Performance Estimation with Instruction Cache Modeling Using Integer Linear Programming". *Proceedings of ICCAD*, April 1995.

[67] C. Liem, M. Cornero, M. Santana, P. Paulin, A. Jerraya, J. M. Gentit, J. Lopez, X. Figari, and L. Bergher. "An Embedded System Case Study : the FirmWare Development Environment for a Multimedia Audio Processor". *Proceedings of DAC*, 1997.

[68] C. Liem, T. May, and P. Paulin. "Register Assignment Through Resource Specification for ASIP Microcode Generation". *Proceedings of the Intl. Conference on Computer Aided Des*, pages 397 – 402, November 1994.

[69] C. Liem, F. Naçabal, C. Valderama, P. Paulin, and A. Jerraya. "System-on-a-Chip Cosimulation and Compilation". *IEEE Design and Test of Computers*, pages 16 – 25, April 1996.

[70] C. Liem, P. Paulin, M. Cornero, and A. Jerraya. "Industrial Experience using Rule-driven Retargetable Code Generation for Multimedia Applications". *Proceedings of the Intl. Symposium on System Synthesis*, pages 60 – 65, September 1995.

[71] C. Liem, P. Paulin, and A. Jerraya. "Address Calculation for Retargetable Compilation and Exploration of Instruction-Set Architectures". *Proceedings of DAC*, pages 597 – 600, 1996.

[72] C. Liem, P. Paulin, and A. Jerraya. "ReCode : the Design and Re-design of the Instruction Codes for Embedded Instruction-Set Processors". *Proceedings of EDTC*, 1997.

[73] V. K. Madisetti and T. W. Egolf. "Virtual Prototyping of Embedded Microcontroller-Based DSP Systems". *IEEE Micro*, pages 9 – 21, October 1995.

[74] W. Meier et al. "Design of Multimedia Systems : Anatomy of an MPEG2 Decoder". *Leuwen Codesign Course*, February 1997.

[75] Motorola. *"DSP96002 User's Manual"*. Motorola, 1996.

[76] F. Naçabal, O. Deygas, P. Paulin, and M. Harrand. "C-VHDL Co-simulation : Industrial Requirements for Embedded Control Processors". *Proceedings of EuroDac (Designer Session)*, pages 55 – 60, September 1996.

[77] S. Narayan and D. Gajski. "Rapid Performance Estimation for System Design". *Proceedings of EuroDac*, pages 206 – 211, 1996.

[78] M. Nogaki. "Exploit Software Flexibility to Add PC Multimedia Functions". *Electronic Design*, pages 42 – 54, October 1997.

[79] F. Onion, A. Nicolau, and N. Dutt. "Incorporating Compiler Feedback Into the Design of ASIPs". *Proceedings of EDTC*, pages 508 – 513, 1995.

[80] P. Panda, N. Dutt, and A. Nicolau. "Memory Organization for Improved Data Cache Performance in Embedded Processors". *Proceedings of the Intl. Symposium on System Synthesis*, pages 90 – 95, 1996.

[81] M. F. Parkinson and S. Parameswaran. "Profiling in the ASP Codesign Environment". *Proceedings of ISSS*, pages 128 – 133, 1995.

[82] P. Paulin. *"FlexWare Environment"*. SGS-THOMSON, January 1997.

[83] P. Paulin, M. Cornero, C. Liem, F. Naçabal, C. Donawa, S. Sutarwala, T. May, and C. Valderrama. "Trends in Embedded Systems Technology : An Industrial Perspective". *in Hardware/Software Co-Design, Kluwer Academic Publishers*, 1996.

[84] P. Paulin, J. Fréhel, M. Harrand, E. Berrebi, C. Liem, F. Naçabal, and J. C. Herluison. "High-Level Synthesis and Codesign Methods : An Application to a Videophone Codec". *Proceedings of EuroDac*, September 1995.

[85] P. Paulin, C. Liem, M. Cornero, F. Naçabal, and G. Goossens. "Embedded Software in Real-time Signal Processing Systems : Application and Architecture Trends". *Proceedings of the IEEE, special issue on Hardware-Software Co-design*, pages 444 – 451, 1997.

[86] P. Paulin, C. Liem, T. May, and S. Sutarwala. "FlexWare : A Flexible FirmWare Development Environment". *in Code Generation for Embedded Processors, Kluwer Academic Publishers*, 1995.

[87] P. Paulin, F. Naçabal, and E. Lantreibecq. *"CoSim v1.0 : User Manual"*. SGS-THOMSON, January 1997.

[88] Philips. *"TM-1 Preliminary Data Book"*. Philips Semiconductors, February 1996.

[89] P. G. Ploger, J. Wilberg, M. Langevin, and R. Camposano. "WWW Based Structuring of Codesigns". *Proceedings of ISSS*, pages 138 – 143, 1995.

[90] M. A. Richards. "The Rapid Prototyping of Application Specific Signal Processors (RASSP)". *Proceedings of 5th Intl. Workshop on RSP*, June 1994.

[91] K. Van Rompaey et al. "CoWare - A Design Environment for Heterogeneous Hardware-Software Systems". *Proceedings of EuroDac*, page 252, September 1996.

[92] J. A. Rowson. "Hardware-Software Co-Simulation". *Proceedings of DAC*, pages 439 – 440, June 1994.

[93] P. Runstadler and R. Crevier. "Virtual Prototyping for System Design and Verification". Synopsys documentation, March 1995.

[94] L. Schwoerer, M. Luck, and H. Schroder. "Integration of VHDL into a System Design Environment". *Proceedings of EuroDac*, pages 268 – 273, 1995.

[95] SGS-THOMSON. *"STi1100 VideoPhone CODEC : Preliminary Data Specification"*. SGS-THOMSON, August 1993.

[96] M. Slater. "MicroUnity Lifts Veil on MediaProcessor". *Microprocessor Report*, pages 11 – 18, October 1995.

[97] M. Slater. "System Architects Look to the Future". *Microprocessor Report*, pages 22 – 24, December 1995.

[98] R. Stallman. *"Using and Porting GNU CC"*. Free Software Foundation, June 1994.

[99] M. Strik and J. Van Meerbergen. "Efficient Code Generation for In-House DSP-Cores". *Proceedings of EDTC*, pages 244 – 249, 1995.

[100] C. Su, C. Tsui, and A. Despain. "Saving Power in the Control Path of Embedded Processors". *IEEE Design and Test of Computers*, pages 24 – 30, 1994.

[101] P. A. Subrahmanyam and S. Parameswaran. "Hardware-Software Codesign of Embedded Systems". *Proceedings of Eurodac (Tutorial)*, September 1996.

[102] A. Sudarsanam and S. Malik. "Memory Bank and Register Allocation in Software Synthesis for ASIPs". *Proceedings of the Intl. Conference on CAD*, pages 388 – 392, 1995.

[103] Synopsys. *"Using the C-Language Interface"*. Synopsys, December 1994.

[104] J. Teich, L. Thiele, and E. A. Lee. "Modeling and Simulation of Heterogeneous Real-Time Systems Based on a Deterministic Discrete Event Model". *Proceedings of ISSS*, pages 156 – 161, 1995.

[105] K. ten Hagen and H. Meyr. "Timed and Untimed Hardware/Software Co-simulation : Application and Efficient Implementation". *Proceedings of CODES*, 1993.

[106] D. E. Thomas, J. K. Adams, and H. Schmit. "A Model and Methodology for Hardware-Software Codesign". *IEEE Design & Test of Computers*, pages 16 – 28, September 1993.

[107] V. Tiwari, S. Malik, and A. Wolfe. "Power Analysis of Embedded Software : A first Step Towards Software Power Minimization". *IEEE Transactions on VLSI Systems*, 2(4) :437 – 445, December 1994.

[108] M. Tremblay, G. Maturana, A. Inoue, and L. Kohn. "A Fast and Flexible Performance Simulator for Micro-Architecture Trade-off Analysis on UltraSparc-I". *Proceedings of DAC*, 1995.

[109] J. Turley and P. Lapsley. "New 56301 DSP Doubles 24-Bit Performance". *Microprocessor Report*, pages 14 – 15, December 1995.

[110] J. L. Turley. "NEC unveils new MIPS chip for Nintendo". *Microprocessor Report*, 9(6) :1–8, May 1995.

[111] J. L. Turley. "Thumb Squeezes ARM Code Size". *Microprocessor Report*, 9(4), March 1995.

[112] C. A. Valderrama et al. "A Unified Model for Co-Simulation and Co-Synthesis of Mixed Hardware/Software Systems". *Proceedings of ED&TC*, March 1995.

[113] C. A. Valderrama, F. Naçabal, P. Paulin, and A. Jerraya. "Automatic Generation of Interfaces for Distributed C-VHDL Cosimulation of Embedded Systems : an Industrial Experience". *Proceedings of the Intl. Workshop on Rapid Systems Prototyping*, pages 72 – 77, June 1996.

[114] P. Vanoostende, G. Van Wauwe, and L. Rottiers. "Issues in Low-Power Design for Telecom". *Proceedings of ESSCIRC*, September 1995.

[115] D. Verma. "Very Large Scale Integrated Circuit Architecture Performance Evaluation Using SES Modelling Tools". Technical report, VLSI Technology, 1994.

[116] D. L. Weaver and T. Germond. *"The SPARC Architecture Manual"*. Prentice Hall, 1994.

[117] W. Ye, R. Ernst, T. Benner, and J. Henkel. "Fast Timing Analysis for Hardware-Software Co-Synthesis". *Proceedings of ICCD*, pages 452 – 457, 1993.

[118] V. Zivojnovic. "DSP Processor/Compiler Co-Design : A Quantitative Approach". *Proceedings of Aachen Dagstuhl*, 1996.

Annexe A

Publications

La liste des publications relatives aux travaux exposés est donnée ci-dessous.

A.1 Publications dans des ouvrages

- P. Paulin, M. Cornero, C. Liem, F. Naçabal, C. Donawa, S. Sutarwala, T. May, C. Valderrama, "Trends in Embedded Systems Technology : An Industrial Perspective", Hardware/Software Co-Design, Kluwer Academic Publishers, 1996.

A.2 Publications dans des revues

- C. Liem, F. Naçabal, C. Valderrama, P. Paulin, A. Jerraya, "System-on-a-chip Cosimulation and Compilation", IEEE Design&Test of Computers, Special Issue on Methods and Tools in Europe, pages 16 - 25, Avril-Juin 1997.
- P. Paulin, G. Goossens, C. Liem, M. Cornero, F. Naçabal, "Embedded Software in Real-Time Signal Processing Systems : Application and Architecture Trends", Proc. IEEE, Special Issue on HW/SW Co-Design, 1997.

A.3 Publications dans des conférences internationales

- F. Naçabal, O. Deygas, P. Paulin, M. Harrand, "C-VHDL Co-simulation : Industrial Requirements for Embedded Control Processors", Proc. of EuroDAC/EuroVHDL Designer Session, pages 55 - 60, Septembre 1996.
- C. Valderrama, F. Naçabal, P. Paulin, A. Jerraya, "Automatic Generation of Interfaces for Distributed C-VHDL Cosimulation of Embedded Systems : an Industrial Experience", Proc. of International Workshop on Rapid Systems Prototyping, pages 72 - 77, Juin 1996.
- P. Paulin, J. Fréhel, M. Harrand, E. Berrebi, C. Liem, F. Naçabal, J-C. Herluison, "High-Level Synthesis and COdesign Methods : An Application to a Videophone Codec", Proc. of EuroDAC/EuroVHDL, Septembre 1995.

A.4 Documentations internes

- P. Paulin, F. Naçabal, E. Lantreibecq, "CoSim 1.0 : User Manual', SGS-THOMSON Microelectronics, Janvier 1997.

- F. Naçabal, "CoSim 1.0 : Programming Manual", SGS-THOMSON Microelectronics, Juin 1996.
- F. Naçabal, "BSPCC : a C Compiler for the Bit-Stream Processor", SGS-THOMSON Microelectronics, Octobre 1996.

Annexe B

Sigles et Acronymes

ACU	Address Calculation Unit - Unité de calcul d'adresses
ALU	Arithmetic and Logic Unit - Unité Arithmétique et Logique
ASIC	Application-Specific Integrated Circuit - Circuit intégré dédié
ASIP	Application-Specific Instruction-set Processor - Processeur à jeu d'instructions dédié
BSP	Bit-Stream Processor - Processeur dédié embarqué dans le circuit IVT
BULU	Break-Up/Look-Up
CIF	Compiler Information File
CLI	C Language Interface - Interface en langage C (VSS de Synopsys)
CODEC	Coder/Decoder - Codeur/Décodeur
DAP	Digital Audio Processor - Processeur dédié de SGS-THOMSON DPG
DCT	Discrete Cosinus Transform - Transformée en cosinus discrète
DCU	Data Calculation Unit - Unité de calcul de données
DECT	Digital European Cordless Telephone - Téléphone mobile européen numérique
DSP	Digital Signal processing Processor - Processeur de traitement numérique du signal
EEPROM	Electrically Erasable Programable Read Only Memory - mémoire morte re-programmable
FKI	Foreign Kernel Interface - Interface en langage C (Voyager de Ikos)
FMI	Foreign Model Interface - Interface en langage C (Leapfrog de Cadence)
GCC	GNU C compiler - compilateur C de GNU
GSM	Groupe Special Mobile - standard européen de téléphonie mobile
HDL	Hardware Description Language - Langage de description matérielle
IDCT	Inverse Discrete Cosinus Transform - Transformée en cosinus discrète inverse
IMEC	Interuniversitair Micro-Elecktronica Centrum (Belgique)
INPG	Institut National Polytechnique de Grenoble
IPC	Inter-Process Communication - Méchanisme de communication inter-processus
ISO	International Standards Organization - Organisation de standardisation internationale
ITU	International Telecommunications Union
IVT	Integrated Videotelephone Terminal - circuit de visiophonie de SGS-THOMSON R&D
MAC	Multiply Accumulator
MCU	Microcontroller Unit - Unité de micro-contrôle
MI	Micro-Instruction
MDF	Micro-instructions Definition File
MIPS	Million Instructions Per Second - Million d'instructions par seconde
MMDSP	MultiMedia Digital processing Processor - Processeur dédié de TCEC
MMIO	Memory-Mapped Input/Output - Entrée/Sortie adressée par la mémoire
MOP	Micro-Operation

MPEG	Motion Picture Experts Group - Standard européen de codage de vidéo numérique
MSQ	Micro-SeQuencer - Processeur dédié embarqué dans le circuit IVT
PC	Program Counter - Compteur Ordinal
RAM	Random Access Memory - Mémoire vive
RISC	Reduced Instruction-Set Computer - Ordinateur à jeu d'instructions réduit
ROM	Read Only Memory - Mémoire morte
RPC	Remote Procedure Call - Appel de procédure à distance
RTL	Register Transfer Level - Description matérielle au niveau transfert de registres
SHM	SHared Memory - Méchanisme de communication par partage de mémoire
ST	SGS-THOMSON Microelectronics
TCEC	Thomson Consumer Electronics Components
TIMA	Techniques de l'Informatique et de la Microélectronique pour l'Architecture d'ordinateurs
UAL	Unité Arithmétique et Logique
UMTS	Universal Mobile Telecommunication System
VHDL	VHSIC Hardware Description Language - Langage de description de circuits intégrés
VHSIC	Very High Speed Integrated Circuit - circuit intégré à très haute vitesse
VIP	VLIW Image Processor - Processeur dédié embarqué dans le circuit IVT
VLC	Variable Length Coding - Codage à longueur variable
VLD	Variable Length Decoding - Décodage à longueur variable
VLIW	Very Large Instruction Word - Mot-instruction très large
VLSI	Very Large Scale Integration - Intégration à très grande échelle
VOP	Vertical Operation
VSS	VHDL Synopsys Simulator - Simulateur VHDL de Synopsys

Annexe C

Fichiers d'exemples

Cette annexe fournit à titre d'illustration quelques programmes et fichiers développés durant cette thèse, concernant les outils spécifiques ajoutés dans la chaîne Flexcc, la configuration de la co-simulation, ainsi que le raffinement d'encodage.

Le code complet des outils CoSim, FlexPlore et ImmPlore, ainsi que les fichiers de configuration des compilateurs Flexcc sont propriété de SGS-THOMSON Microelectronics et sont donc soumis aux règles de confidentialité industrielle.

C.1 Flexcc : outils personnalisés

Les deux programmes `prep` et `profile`, écrits en langage Perl 5, sont insérés dans la chaîne de compilation Flexcc. `prep` traite les directives pragmas afin d'expanser les fonctions indiquées, avant toutes les autres étapes de la compilation. `profile` gère l'inversion des bits dans le code machine final afin de réduire la consommation lors de la lecture des mots-instructions en ROM. Il est lancé à la fin de la chaîne.

prep

```
#! /user201/contrib/S1/bin/perl
#  $Id : prep,v 1.4 1996/07/08 12 :28 :20 nacabalf Exp nacabalf $
#
#  'prep'  1.10
#  usage : prep <infile> <outfile>
#  - substitute the main function name with Main
#  - expand functions into __inline__ in accordance with the pragma directive
#
#  Francois Nacabal - April, 1996
#    Embedded Systems, SGS-Thomson Microelectronics, Crolles
#
$infile = @ARGV[0] ;
$outfile = @ARGV[1] ;
if (!$infile) { exit(0) ; }
```

```
$errors = 0 ;
    unless (open(IN_FILE, "<$infile")) {
        print STDERR "Error : cannot open input file : $infile" ;
        exit(1) ;
    }
    # collect pragmas
    $lineno = 0 ;
    while(<IN_FILE>) {
        $lineno++ ;
        if (/^\#[ |]+pragma\s+inline\s+(\w+)/) {
            # inline pragma
            if ($inline{$1}) {
                print STDERR "Warning ($infile :$lineno) : multiple
inline pragma for function $1 :\n" ;
                print STDERR "$_\n" ;
                $errors++ ;
            } else {
                $inline{$1} = "INLINE" ;
            }
        }
    }
    close(IN_FILE) ;
if ($errors) { exit(1) ; }
&inline_expansion() ;
sub inline_expansion {
            unless (open(IN_FILE, "<$infile")) {
                print STDERR "Error : cannot open input file : $infile" ;
                exit(1) ;
            }
            unless (open(OUT_FILE, ">$outfile")) {
                print STDERR "Error : cannot open output file : $outfile" ;
                exit(1) ;
            }
            while(<IN_FILE>) {
                if (/^\s*(\w+)\s*(\w*)\s*(\w*)\s*\(\s*\w*\s*\)\s*[{]*$/) {
                   if ($3) {
                        $func = $3 ;
                        $type = "$1" . " " . "$2" ;
                   }
                   elsif ($2) {
                        $func = $2 ;
                        $type = $1 ;
                   }
                   else {
                        $func = $1 ;
                        $type = "int" ;
                   }
                   if ($inline{$func} eq "INLINE") {
                        if ($type ne "void") {
                           print STDERR "Warning ($infile :$lineno) : Function
$func must return a 'void' ; '$type' ignored.\n" ;
                        }
                        $new_line = $_ ;

$new_line =~ s/\w*\s*\w*\s*$func\s*\(((\s*\w*\s*)\)\s*([{]*)$/extern
```

```
__inline__ void $func ($1) $2/;
                        print OUT_FILE "$new_line\n";
                    }
                    else {
                        $new_line = $_;
                        $new_line =~ s/^(\s*)\w*(\s*)main\s*\(/$1$2void Main \(/;
                        print OUT_FILE $new_line;
                    }
                }
                else {
                    print OUT_FILE $_;
                }
            }
            close(IN_FILE);
            close(OUT_FILE);
}
```

profile

```
#! /user201/contrib/S1/bin/perl
#  $Id : profile,v 1.5 1996/04/16 16 :28 :12 nacabalf Exp nacabalf $
#
#  'profile'
#  usage : prof <prog_name>
#
#  Francois Nacabal - April, 1996
#    Embedded Systems, SGS-Thomson Microelectronics, Crolles
#
$progname = @ARGV[0];
$nb_bits = @ARGV[1];
$lstfile = $progname . ".lst";
$romfile = $progname . ".rom";
$profile = $progname . ".prof";
if (!$progname ||!$nb_bits) { exit(0); }
$errors = 0;
$nb_code = 0;
$max_adr = 0;
&read_lst();
&read_rom();
if (($last_adr - $first_adr + 1) != $nb_ins) {
   print STDERR "Mismatch of code size between $lst_file
($first_adr :$last_adr) and $rom_file ($nb_ins)\n";
   exit 1;
}
&print_stats();
```

```
sub read_lst {
    unless (open(LST_FILE, "<$lstfile")) {
        print STDERR "Error : cannot open input file : $lstfile";
        exit(1);
    }
    $first = 1;
    while (<LST_FILE>) {
        if (/^([0-9a-fx]+)\s+([0-9a-f]+)\s+\d+\s+(.*)\s+$/) {
        $rom_hex = $1;
        $rom_adr = hex($1);
        $rom_code = $2;
        $asm_code = $3;
        if (substr($asm_code,0,1) NE ".") {
                $hex_tab[$rom_adr] = $rom_hex;
                $rom_tab[$rom_adr] = $rom_code;
                $asm_tab[$rom_adr] = $asm_code;
                if ($first) {
                   $first = 0;
                   $first_adr = $rom_adr;
                }
                $last_adr = $rom_adr;
                }
        }
    }
    close(LST_FILE);
}
sub read_rom {
    local($linenum) = 0;
    unless (open(ROM_FILE, "<$romfile")) {
        print STDERR "Error : cannot open input file : $romfile";
        exit(1);
    }
    while (<ROM_FILE>) {
        if (/^([0-9a-f]+)/) {
        $rom_code = $1;
        $rom_tab[$linenum] = $rom_code;
        $a = hex($rom_code);

        &stock_code($a);

        $nb_bit1 = &count_bit1($a);
        $bit1_tab[$linenum] = $nb_bit1;

        $linenum++;
        }
    }
    $nb_ins = $linenum;
    close(ROM_FILE);
}
sub get_asm_code {
        local($adr) = $_[0];
        print "ASM ($adr) : $asm_tab{$adr}\n";
}
sub get_rom_code {
        local($adr) = $_[0];
        print "ROM ($adr) : $rom_tab{$adr}\n";
}
sub dump_rom {
        for ($i=0; $i<$nb_ins; $i++) {
            print "$asm_tab[$i]\n";
        }
```

```
}
sub print_stats {
        local($i);
        local($fmax) = 0;
        local($j);
        local($f);

        unless (open(PROF_FILE, ">$profile")) {
        print STDERR "Error : cannot open output file : $profile";
        exit(1);
        }
        $nb_reversed = 0;
        $prev_bit0 = 0;
        $total_bit0 = 0;
         for ($j = 0; $j < $nb_ins; $j++) {
            $a = hex($rom_tab[$j]);
            $binary = &get_bin($a);
            $nb_bit0 = $nb_bits - $bit1_tab[$j];
            $total_bit0 += $nb_bit0;
            $pc_bit0 = int($nb_bit0 * 100 / $nb_bits);
            if (($a >> ($nb_bits-1)) & 1) {
                $tag = "*";
                $nb_reversed ++;
                $prev_bit0 += $bit1_tab[$j];
            }
            else {
                $tag = " ";
                $prev_bit0 += $nb_bit0;
            }
            printf PROF_FILE "%08x %3d %2d%% %s
%s %s %s\n", $hex_tab[$j], $f_tab[$a], $pc_bit0,
$tag, $rom_tab[$j], $binary, $asm_tab[$j];
         }
        $pc_bit0 = int($total_bit0 * 100 / ($nb_bits*$nb_ins));
        $pc_reversed = int($nb_reversed * 100 / $nb_ins);
        $pc_gain = int($total_bit0 * 100 / $prev_bit0);
        print "\n\n";
        print "Number of Instructions : $nb_ins\n";
        print "Number of Instr. used : $nb_code\n";
        print "Number of Reversed    : $nb_reversed\t( $pc_reversed% )\n";
        print "Number of Zeros       : $total_bit0\t( $pc_bit0% )\n";
        print "Gain                  : $pc_gain%\n";
        print PROF_FILE "\n\n";
        print PROF_FILE "Number of Instructions : $nb_ins\n";
        print PROF_FILE "Number of Instr. used : $nb_code\n";
        print PROF_FILE "Number of Reversed    : $nb_reversed\t( $pc_reversed% )\n";
        print PROF_FILE "Number of Zeros       : $total_bit0\t( $pc_bit0% )\n";
        print PROF_FILE "Gain                  : $pc_gain%\n";
        close(PROF_FILE);
}
sub count_bit1 {
        local($value) = $_[0];
        local($nb) = 0;
        local($i);
        for ($i = $nb_bits-1; $i >= 0; $i--) {
            $nb += ($value >> $i) & 1;
        }
        $nb;
}
```

```
sub get_reverse {
        local($value) = $_[0];
        local($rev);
        $rev = ($value ^ 0xffffffff) & ~(0xffffffff << $nb_bits);
}
sub get_bin {
        local($value) = $_[0];
        local($i);
        local($tmp);
        local($str) = "";
        for ($i = $nb_bits-1; $i >= 0; $i--) {
            $temp = ($value >> $i) & 1;
            $str = $str . "$temp";
        }
        $str;
}
sub stock_code {
        local($code) = $_[0];

        if ($f_tab[$code] == 0) {
           $nb_code++;
           $f_tab[$code] = 1;
        }
        else {
           $f_tab[$code] ++;
        }
        if ($code > $max_code) {
           $max_code = $code;
        }
}
sub stock_code2 {
        local($adr) = $_[0];
        local($code) = hex($_[1]);
        local($i);
        $found = 0;
        for ($i = 0; $i < $nb_code; $i++) {
            if ($uniq_tab[$i] == $code) {
                $f_tab[$i] ++;
                $found ++;
                break;
            }
        }
        if ($found == 0) {
           $uniq_tab[$nb_code] = $code;
           $f_tab[$nb_code] = 1;
           $adr_tab[$nb_code] = $adr;
           $nb_code++;
        }
}
```

C.2 CoSim

C.2.1 Application au circuit Calc

vhdl_types.h

Ce fichier contient la déclarations des types de données partagés entre le C et le VHDL.

```
/* vhdl_types.h */

#ifndef VHDL_TYPES
#define VHDL_TYPES

/*------------------------------------------------------------------------*/
/* Types                */
/*------------------------------------------------------------------------*/
typedef enum {
        BIT_0 = 0,
        BIT_1
} BIT;

typedef int INTEGER;
typedef enum {
        STD_LOGIC_U = 0,
        STD_LOGIC_X,
        STD_LOGIC_0,
        STD_LOGIC_1,
        STD_LOGIC_Z,
        STD_LOGIC_W,
        STD_LOGIC_L,
        STD_LOGIC_H,
        STD_LOGIC_D
} STD_LOGIC;
typedef enum {
        STD_ULOGIC_U = 0,
        STD_ULOGIC_X,
        STD_ULOGIC_0,
        STD_ULOGIC_1,
        STD_ULOGIC_Z,
        STD_ULOGIC_W,
        STD_ULOGIC_L,
        STD_ULOGIC_H,
        STD_ULOGIC_D
} STD_ULOGIC;
typedef long BIT_VECTOR;
#endif
```

alu.ccf

Ce fichier configure la session de co-simulation, en fixant les chemins d'accès aux modèles C et VHDL, et les différentes options.

```
# CoSim V 0.3 - C Configuration file

# unix path to a working directory (for generated intermediate files)
work_dir = ./work

# unix path to C source files :
sce_dir = .

# list of C source files to be compiled
c_files = io_alu.c plus.c

# C defines to be set at compilation time
#define = DEBUG_C0 VHDL_PRINT_INFOS VHDL_PRINT_DEBUG
define = DEBUG_C0

# command line to run the executable program (with arguments)
```

```
cmd_line = plus

# C output log file
log = alu.log

# debugger command
# xxgdb

# name of the VHDL file containing the entity declaration
entity_file = ../vhdl_vss/c_alu.all.vhd

# name of the VHDL entity to be cosimulated (substitued with the C program)
entity_name = c_alu

# synchronisation mode (sync or async)
mode = sync
```

plus.c

Ce programme représente le code applicatif C tournant sur le processeur ALU du circuit Calc. Toutes les opérations d'entrée/sortie sont accomplies par les fonctions `io_read` et `io_write`, dont le code est donné plus loin dans le fichier `io_alu.c`.

```
#include <stdio.h>
#include "vhdl_types.h"
#include "io_alu.h"

main() {
int Cmd;
int Data;
int a, b, r;
setbuf(stdout, (char *)NULL);
while (1) {
 printf ("plus : Start ...\n");
 while (io_read(READ_CMD) != 1); /* Wait for a command (must be 1) */
        printf ("plus : io_read ok : CMD = 1\n");
        printf ("plus : io_write BUSY = 1 ...\n");
 io_write(WRITE_BUSY, BIT_1);   /* When received command, set BUSY flag */
 a = io_read(READ_A);           /* Read A value */
        if (a!=5) printf ("Error a = %d\n", a);
        else printf ("plus : io_read ok : A = %d\n",a);
 b = io_read(READ_B);           /* Read B value */
        if (b!=2) printf ("Error b = %d\n", b);
        else printf ("plus : io_read ok : B = %d\n",b);
        printf ("\nplus : Compute a + b ...\n");
 r = a + b;                     /* compute the A + B */
        if (r!=7) printf ("Error r = %d\n", r);
        printf ("plus : io_write!! RES = 9!! ...\n");
 io_write(WRITE_RES, 9);        /* Write a bad result, without ENABLE flag */
        printf ("plus : io_write RES = %d ...\n" , r);
 io_write(WRITE_RES, r);        /* Write the right result, sustain it */
        printf ("plus : io_write REN = 1\n");
 io_write(WRITE_REN, STD_ULOGIC_1);     /* Write ENABLE - Result is still on the bus */
        printf ("plus : io_write REN = 0 ...\n");
 io_write(WRITE_REN, STD_ULOGIC_0);     /* Release ENABLE */
        printf ("plus : io_write RES = 0 ...\n");
        printf ("plus : io_write BUSY = 0 ...\n");
 io_write(WRITE_RES, 0);
 io_write(WRITE_BUSY, BIT_0);           /* Reset BUSY flag */
        printf ("plus : End.\n");
        fflush(stdout);
 }
}
```

io_alu.h

Ce fichier contient les déclarations nécessaires à la communication entre l'application C et le reste du système en VHDL. Les signaux d'interface sont identifiés par les `#define`.

```
/* io_alu.h */

#define READ_CMD 1
#define READ_A 2
#define READ_B 3
#define WRITE_REN 4
#define WRITE_RES 5
#define WRITE_BUSY 6

io_transaction() ;
io_read() ;
io_write() ;
```

io_alu.c

Ce fichier contient les fontions de communication avec le reste du système, c'est-à-dire `io_read` et `io_write`.

```
#include <stdio.h>
#include <sys_types.h>
#include "vhdl_types.h"
#include "C_ALU.h"
#include "io_alu.h"

int io_read(cmd)
int cmd ;
{
 int value ;
 switch (cmd) {
        case READ_CMD :
               C_ALU_Transaction() ;
               value = C_CMD ;
               break ;
        case READ_A :
               while (C_AEN != STD_LOGIC_1) C_ALU_Transaction() ;
               value = C_DATA ;
               break ;
        case READ_B :
                while (C_BEN != STD_LOGIC_1) C_ALU_Transaction() ;
               value = C_DATA ;
               break ;
        }
 return (value) ;
}

int io_write(cmd, value)
int cmd ;
int value ;
{
 switch (cmd) {
        case WRITE_RES :
               C_RES = value ;
               break ;
        case WRITE_REN :
               C_REN = value ;
               break ;
        case WRITE_BUSY :
               C_BUSY = value ;
               break ;
        }
```

```
C_ALU_Transaction() ;
}
```

Les trois fichiers `alu.ent.vhd`, `alu.cos.vhd` et `c_alu.all.vhd` correspondent au modèle VHDL de co-simulation du processeur ALU.

alu.ent.vhd

```
-- alu.ent.vhd
library IEEE ;
    use IEEE.std_logic_1164.all ;
entity ALU is
        port
        (
          CLK           : in BIT := '0' ;
          CMD           : in INTEGER := 0 ;
          AEN           : in STD_ULOGIC := '0' ;
          BEN           : in STD_ULOGIC := '0' ;
          REN           : out STD_ULOGIC ;
          BUSY          : out BIT ;
          DATA          : in STD_ULOGIC_VECTOR(7 downto 0) := "00000000" ;
          RES           : out STD_ULOGIC_VECTOR (7 downto 0)
         ) ;
end ALU ;
```

alu.cos.vhd

```
-- alu.cos.vhd
library IEEE ;
    use IEEE.std_logic_arith.all ;
architecture COSIM_ALU of ALU is
--
-- CLI/FMI component (linked to the C part)
-- same port map as the original entity, with C_ prefix
-- data types are BIT or INTEGER (for boolean and 32 bits max. integers)
--
    component C_ALU
        port
        (
        C_CLK           : in BIT ;
        C_CMD           : in INTEGER ;
        C_AEN           : in STD_ULOGIC ;
        C_BEN           : in STD_ULOGIC ;
        C_REN           : out STD_ULOGIC ;
        C_BUSY          : out BIT ;
        C_DATA          : in INTEGER ;
        C_RES           : out INTEGER
        ) ;
    end component ;

signal C_CLK            : BIT ;
signal C_CMD            : INTEGER ;
signal C_AEN            : STD_LOGIC ;
signal C_BEN            : STD_LOGIC ;
signal C_REN            : STD_ULOGIC ;
signal C_BUSY           : BIT ;
signal C_DATA           : INTEGER ;
signal C_RES            : INTEGER ;

begin
```

```
--
-- Instance of the CLI cpomponent
--
    C_ALUINST : C_ALU
        port map
        (
        C_CLK           => C_CLK,
        C_CMD           => C_CMD,
        C_AEN           => C_AEN,
        C_BEN           => C_BEN,
        C_REN           => C_REN,
        C_DATA          => C_DATA,
        C_BUSY          => C_BUSY,
        C_RES           => C_RES
        );
--
-- Transmission of the Clock signal (could be divided)
--
        C_CLK <= CLK;
--
-- Type convertion for C->VHDL signals (clock-true)
--
synchro_out : process
begin
        wait until CLK'event and CLK = '1';
        C_DATA <= to_integer( DATA );
        C_CMD  <= CMD;
        C_AEN  <= AEN;
        C_BEN  <= BEN;
end process synchro_out;
--
-- Type convertion for VHDL->C signals (clock-true sampling)
--
synchro_in : process
begin
        wait until CLK'event and CLK = '0';
        BUSY <= C_BUSY;
        RES <= To_StdULogicVector( C_RES, 8 );
        REN <= C_REN;
end process synchro_in;
end COSIM_ALU;
```

c_alu.all.vhd

```
-- c_alu.all.vhd
library IEEE;
    use IEEE.std_logic_1164.all;

entity C_ALU is
        port
        (
        C_CLK           : in BIT := '0';
        C_CMD           : in INTEGER := 0;
        C_AEN           : in STD_ULOGIC := '0';
        C_BEN           : in STD_ULOGIC := '0';
        C_REN           : out STD_ULOGIC;
        C_DATA          : in INTEGER := 0;
        C_BUSY          : out BIT;
        C_RES           : out INTEGER
        );
```

```
end C_ALU ;

architecture FMI of C_ALU is
attribute FOREIGN of FMI :architecture is "Clib :C_ALU" ;

begin end FMI ;

configuration C_ALU_CFG of C_ALU is
for FMI end for ;
end C_ALU_CFG ;
```

C.2.2 Application au système IVT

Le fichier `msq.ccf` est un fichier destiné à configurer l'outil CoSim pour la co-simulation du logiciel chargé dans le processeur MSQ (contrôleur du circuit).

msq.ccf

```
# Configuration file for CoSim
#============================
work_dir = ../msq_work
sce_dir = .
h_files = msq_debug.h msq_io.h msq_constants.h msq.h
c_files = msq_io.c msq.c
cmd_line = msq
define = DEBUG_C0 DEBUG
entity_name = msq_cli
entity_file = ../msq_vhdl/msq_cli.ent.vhd
mode = sync
log = msq.log
```

Les fichiers `msq.ent.vhd`, `msq.cos.vhd` et `msq_cli.ent.vhd` et `msq_cli.cli.vhd` décrivent la configuration matérielle, en VHDL, correspondant à la co-simulation du MSQ. En particulier, son interface est inchangée alors que son architecture est substituée par une liaison CLI, vers le C.

msq.ent.vhd

```
------------------------------------------------------------------------------
--                                Copyright 1996                            --
--                         SGS Thomson - Microelectronics                   --
--                              All Rights Reserved                         --
------------------------------------------------------------------------------
-- MSQ entity
------------------------------------------------------------------------------
-- $Id : msq.ent.vhd,v 1.2 1996/05/03 15 :07 :31 visioha Exp $
------------------------------------------------------------------------------

library ieee ;
use ieee.std_logic_1164.all ;

entity msq is
```

```
port (
        msq_a    : out   std_logic;                     -- MSQ address phase
        msq_n    : out   std_logic;                     -- MSQ next/new data
        msq_e    : inout std_logic;                     -- MSQ event
        msq_rw   : out   std_logic;                     -- MSQ RW
        msq_ad   : inout std_logic_vector(7 downto 0);  -- MSQ address/data bus
        mcc_rw   : in    std_logic;                     -- MCC RW
        mcc_addr : in    std_logic_vector(5 downto 0);  -- MCC address bus
        mcc_data : inout std_logic_vector(31 downto 0); -- MCC data bus
        msq_inrq : out   std_logic;                     -- MCC input request
        msq_ourq : out   std_logic;                     -- MCC output request
        clock    : in    std_ulogic;
        reset    : in    std_ulogic
);
end msq;
```

msq.cos.vhd

```
-------------------------------------------------------------------------------
--                              Copyright 1996                               --
--                        SGS Thomson - Microelectronics                     --
--                             All Rights Reserved                           --
-------------------------------------------------------------------------------
-- Cosimulation architecture of MSQ
-------------------------------------------------------------------------------
-- $Id : msq.cos.vhd,v 1.2 1996/05/03 15 :07 :28 visioha Exp $
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

architecture cos of msq is
component msq_cif
port (
        msq_a    : out   std_logic;                     -- Address phase
        msq_n    : out   std_logic;                     -- Next address/New data
        msq_rw   : out   std_logic;                     -- RW
        msq_e    : inout std_logic;                     -- MSQ Event
        msq_ad   : inout std_logic_vector(7 downto 0);  -- Address/Data bus
        mcc_rw   : in    std_logic;                     -- MCC RW
        mcc_addr : in    std_logic_vector(5 downto 0);  -- MCC address
        mcc_data : inout std_logic_vector(31 downto 0); -- MCC data
        msq_inrq : out   std_logic;                     -- MSQ input request
        msq_ourq : out   std_logic;                     -- MSQ output request
        clock    : in    std_ulogic;
        reset    : in    std_ulogic;
        clk_eval : out   bit;                           -- C exchange clock
        c_cif    : in    integer;                       -- C command
        c_op     : in    integer;                       -- Operator
        c_add    : in    integer;                       -- Address
        c_dout   : in    integer;                       -- Write data
        c_din    : out   integer;                       -- Read data
        c_reset  : out   integer                       -- C reset
);
end component;
```

```
component msq_cli
        port (
                clk_eval    : in  bit;     -- Evaluation clock
                c_cif       : out integer; -- C command to VHDL
                c_op        : out integer; -- Operator (base)
                c_add       : out integer; -- Address (offset)
                c_dout      : out integer; -- Write data
                c_din       : in  integer; -- Read data
                c_reset     : in  integer  -- Reset
        );
end component;
signal c_add            : integer;
signal c_cif            : integer;
signal c_din            : integer;
signal c_dout           : integer;
signal c_op             : integer;
signal c_reset          : integer;
signal clk_eval         : bit;
begin
        cif : msq_cif
                port map (
                        msq_a           => msq_a          ,
                        msq_n           => msq_n          ,
                        msq_rw          => msq_rw         ,
                        msq_e           => msq_e          ,
                        msq_ad          => msq_ad         ,
                        mcc_rw          => mcc_rw         ,
                        mcc_addr        => mcc_addr       ,
                        mcc_data        => mcc_data       ,
                        msq_inrq        => msq_inrq       ,
                        msq_ourq        => msq_ourq       ,
                        clock           => clock          ,
                        reset           => reset          ,
                        clk_eval        => clk_eval       ,
                        c_cif           => c_cif          ,
                        c_op            => c_op           ,
                        c_add           => c_add          ,
                        c_dout          => c_dout         ,
                        c_din           => c_din          ,
                        c_reset         => c_reset
                );
        cli : msq_cli
                port map (
                        clk_eval        => clk_eval       ,
                        c_cif           => c_cif          ,
                        c_op            => c_op           ,
                        c_add           => c_add          ,
                        c_dout          => c_dout         ,
                        c_din           => c_din          ,
                        c_reset         => c_reset
                );
end cos;
```

msq_cli.ent.vhd

```
--------------------------------------------------------------------------------
--                              Copyright 1996                                --
--                        SGS Thomson - Microelectronics                      --
--                             All Rights Reserved                            --
```

```
-------------------------------------------------------------------------------
-- entity description for CLI interface
-------------------------------------------------------------------------------
-- $Id : msq_cli.ent.vhd,v 1.2 1996/05/03 15 :07 :42 visioha Exp $
-------------------------------------------------------------------------------

entity msq_cli is
        port (
                clk_eval    : in  bit;     -- Evaluation clock
                c_cif       : out integer; -- C command to VHDL
                c_op        : out integer; -- Operator (base)
                c_add       : out integer; -- Address (offset)
                c_dout      : out integer; -- Write data
                c_din       : in  integer; -- Read data
                c_reset     : in  integer  -- Reset
        );
end msq_cli;
```

msq_cli.cli.vhd

```
-------------------------------------------------------------------------------
--                                Copyright 1996                             --
--                         SGS Thomson - Microelectronics                    --
--                              All Rights Reserved                          --
-------------------------------------------------------------------------------
-- CLI architecture for interfacing MSQ VHDL to C
-------------------------------------------------------------------------------
-- $Id : msq_cli.cli.vhd,v 1.2 1996/05/03 15 :07 :39 visioha Exp $
-------------------------------------------------------------------------------

library synopsys;
use synopsys.attributes.all;

architecture cli of msq_cli is

        attribute foreign       of cli : architecture is "SYNOPSYS :CLI";
        attribute cli_elaborate of cli : architecture is "MSQ_CLIOpen";
        attribute cli_evaluate  of cli : architecture is "MSQ_CLIEval";
        attribute cli_error     of cli : architecture is "MSQ_CLIError";
        attribute cli_close     of cli : architecture is "MSQ_CLIClose";

        attribute cli_pin of clk_eval    : signal is cli_event;
        attribute cli_pin of c_reset     : signal is cli_passive;
        attribute cli_pin of c_din       : signal is cli_passive;

begin
end cli;
```

Les fichiers `msq_io.h` et `msq_io.c` permettent de relier l'application C au modèle VHDL du reste du système, en échangeant les valeurs des signaux d'interface. Les variables préfixées par `c_` représentent ces signaux, alors que les deux fonctions `io_read()` et `io_write()` permettent à l'application d'y accéder.

msq_io.h

```
#ifndef MSQ_IO_H
#define MSQ_IO_H

#ifdef COMPILE
/* For ARCHELON */
#else
/* For UNIX */
#define CIF_RADD   0x10
#define CIF_DATA   0x11
#define CIF_WADD   0x20

extern int c_cif;
extern int c_op;
extern int c_add;
extern int c_din;
extern int c_dout;
extern int c_reset;

int io_transaction();
int io_read();
int io_write();
#endif


#endif
```

msq_io.c

```
#include <stdlib.h>
#include <stdio.h>

#include "msq_io.h"

int io_transaction()
{
#ifdef DEBUG_C1
        fprintf(stderr,"io_transaction : Send ...\n");
        fprintf(stderr,"                 c_cif  = %d\n", c_cif);
        fprintf(stderr,"                 c_op   = %d\n", c_op);
        fprintf(stderr,"                 c_add  = %d\n", c_add);
        fprintf(stderr,"                 c_dout = %d\n", c_dout);
#endif
        MSQ_CLI_Transaction();
#ifdef DEBUG_C1
        fprintf(stderr,"io_transaction : Wait for Receive ...\n");
#endif
        if (c_reset == 1) exit (22);
#ifdef DEBUG_C1
        fprintf(stderr,"                 c_din  = %d\n", c_din);
#endif
        return 0;
}

int io_read(operator, address)
int operator;
int address;
```

```
{
        c_cif = CIF_RADD ;
        c_op  = operator ;
        c_add = address ;
        io_transaction() ;
        c_cif = CIF_DATA ;
        io_transaction() ;

#ifdef DEBUG_C1
        fprintf(stderr,"io_read : %d, %d, %d\n", c_op, c_add, c_din) ;
#endif
        return (c_din) ;
}

int io_write(operator, address, data)
int operator ;
int address ;
int data ;
{
        c_cif  = CIF_WADD ;
        c_op   = operator ;
        c_add  = address ;
        c_dout = data ;
        io_transaction() ;
#ifdef DEBUG_C1
        fprintf(stderr,"io_write : %d, %d, %d\n", c_op, c_add, c_dout) ;
#endif
        return 0 ;
}
```

C.3 ImmPlore

ImmPlore est un outil d'aide au raffinement de l'encodage des champs immédiats dans un jeu d'instructions (voir chapitre 5). Il se base le code assembleur pré-compilé d'une application donnée, et sur un fichier de spécification d'encodage. Quelques exemples de fichiers de spécification sont donnés, pour le DAP et le D950. Ces architectures sont décrites dans le chapitre 2.

C.3.1 dap.def

```
# DAP
# Definition of immediate formats for the DAP (.vop files)

#!!!!!!!!!!
# Parallel mults are not correctly considered (one imm instead of 2)

# immediate operands

cost                    :       Word    Cycle
weight                  :       1       1
immediate data14  14    :       1       1
immediate addr12  12    :       1       1
immediate offset4  4    :       1       1
immediate mask8    8    :       1       1
#immediate loop8    8   :       1       1

# CODOP classes
```

```
class ALU
subclass ALU
andi    ACC0    :       data14
cmpri           :       data14
equi    ACC0    :       data14

class ACU
subclass X
load_X Dx[0-1],x\[Ax[0|1]\+ :   offset4
loadb_X Dx[0-1],x\[Ax[0|1]\+ :  offset4
load_X Dx[0-1],x\[.*\+ :        addr12
load_X Dy[0-1],x\[.*\+ :        addr12

storeb_X x\[Ax[0|1]\+ : offset4
store_X x\[.*\+ :       addr12

subclass Y
load_Y Dy[0-1],y\[Ay[0|1]\+ :   offset4
loadb_Y Dy[0-1],y\[Ay[0|1]\+ :  offset4
load_Y Dy[0-1],y\[.*\+ :        addr12

class DCU
subclass MULT
mult_5 P,Dx[0-1],x\[Ax[0-1]\+ : offset4
mult_5 P,Dx[0-1],none,Dy[0-1],y\[Ay[0-1]\+ : offset4

mult_5 P,Dy[0-1],y\[Ay[0-1]\+ : offset4

mult_5 P,Dx[0-1],x\[Ax[0-1].*\],Dx[0-1],x\[Ax[0-1]\+ : offset4
mult_5 P,Dx[0-1],x\[Ax[0-1].*\],Dy[0-1],u\[Ay[0-1]\+ : offset4

mult.*_5 P,Dx[0-1],x\[Ax[0-1].*\],Dy[0-1],y\[Ay[0-1]\+ : offset4
mult.*_5 P,Dy[0-1],y\[Ay[0-1].*\],Dy[0-1],y\[Ay[0-1]\+ : offset4

subclass SET
stshfb_X x\[Ax[0-1]\+ : offset4
stshfb_Y y\[Ay[0-1]\+ : offset4
set     :       mask8
reset   :       mask8
```

C.3.2 d950.def

```
# D950
# Definition of immediate formats for .vop files

# immediate operands
cost                :   Word    Cycle
weight              :   1       1

immediate imm5   5  :   1       1
immediate short  6  :   1       1
immediate long   16 :   2       2

# codop classes
class ACU
subclass AX
loadsi  AX[0-1] :       short
loadi   AX[0-1] :       long

subclass AY
```

```
loadsi  AY[0-1] :       short
loadi   AY[0-1] :       long

subclass IX
loadsi  IX[0-3] :       short
loadi   IX[0-3] :       long

subclass IY
loadsi  IY[0-3] :       short
loadi   IY[0-3] :       long

class ALU
subclass L
loadsi  L[0-1]  :       short
loadi   L[0-1]  :       long

subclass R
loadsi  R[0-1]  :       short
loadi   R[0-1]  :       long

subclass A
loadsi  A[0-1]L :       short
loadi   A[0-1]L :       long

loadasi A[0-1]H :       short
loadai  A[0-1]H :       long

loadsi  A[0-1]H :       short
loadi   A[0-1]H :       long

asl     A[0-1],A[0-1]   :       imm5
asl     A[0-1],L[0-1]   :       imm5
asl     A[0-1],R[0-1]   :       imm5

asr     A[0-1],A[0-1]   :       imm5
asr     A[0-1],L[0-1]   :       imm5
asr     A[0-1],R[0-1]   :       imm5

lsr     A[0-1],A[0-1]   :       imm5
lsr     A[0-1],L[0-1]   :       imm5
lsr     A[0-1],R[0-1]   :       imm5

subclass STA
loadsi  STA     :       short
loadi   STA     :       long

class LOOP
subclass LOOP
LC =    :       short,  long
LS =    :       short,  long
PL =    :       short,  long
```

Résumé

Les applications complexes comme la téléphonie mobile, la télévision numérique ou la visiophonie exigent une grande puissance de calcul, mais aussi une flexibilité accrue afin de suivre l'évolution des standards. L'intégration de tels systèmes sur une seule puce requiert l'embarcation de processeurs devant respecter des contraintes de performances, de coût en surface et de faible consommation. Leur conception en un temps réduit met en oeuvre des compilateurs rapidement reciblables, ainsi que de nouveaux outils d'aide à la conception. Ceux-ci sont nécesssaire pour suivre le cycle de vie de tels processeurs, composé d'étapes de réduction de coût et de ré-utilisation. Cette thèse présente plusieurs techniques visant à réduire le temps de développement du couple logiciel-processeur embarqué, à savoir la validation fonctionnelle à haut-niveau et l'aide au raffinement de l'architecture et du jeu d'instructions.

La validation de la description haut-niveau du logiciel embarqué est assurée dans son environnement matériel réel grâce à la co-simulation C-VHDL, développée durant cette thèse. La mise au point du logiciel est alors facilitée par l'utilisation d'outils de développement standard, et par la faculté à simuler le système complet sur un large intervalle de temps.

L'aide au raffinement d'architecture est assurée par la re-configuration automatique d'un compilateur reciblable, afin d'explorer un grand nombre de solutions en un temps réduit. L'analyse de codes applicatifs typiques ainsi compilés permet d'isoler les configurations architecturales performantes. De plus, un outil d'estimation se concentrant sur l'encodage des champs constants dans le le jeu d'instructions est proposé.

Abstract

Emerging applications like mobile phones, digital video or videophones require powerful computing as well as good flexibility in order to track the evolving standards. The integration of such systems in one single chip often involves dedicated processors, with design constraints on performance, area cost and power consumption.This thesis deals with the co-design of a dedicated processor and its embedded software. The main goal is to reduce the development time of the processor-software pair, concentrating on two complementary issues : the high-level functional validation of software in its real environment and the processor architecture exploration by means of its instruction-set.

Functional high-level validation in real environment involves the co-simulation of the application software written in C with the rest of the hardware, described in VHDL. This co-simulation is performed without the need of a simulation model of the processor, unlike the classical approach using instruction-set level co-simulation. Starting from an existing communication model, a C-VHDL co-simulation environment is developed in order to fulfil requirements from the industrial design of a complex system, a videophone terminal.

Processor architecture exploration is obtained by the automatic reconfiguration of a retargetable compiler. Based on statistics from a number of variations around an original DSP architecture, an optimised solution in terms of code size is identified. A restricted set of critical criteria including register sets size and configuration is selected. An alternative approach based on the estimation of pre-compiled assembly code is experienced, concentrating on a peculiar issue in instruction-set design, the constant fields encoding. An interactive tool for instruction-set refinement is proposed.