THESE DE DOCTORAT DE L'UNIVERSITE PARIS VI

Spécialité

INFORMATIQUE

Présentée par

Yann BAJOT

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE PARIS VI

Sujet

ETUDE ET SPECIFICATION D'UN COEUR DE PROCESSEUR DE TRAITEMENT DU SIGNAL CONFIGURABLE POUR SYSTEMES EMBARQUES SPECIALISES

Soutenue le **7 Novembre 2001**

Devant le jury composé de :

M. Eric	**MARTIN**	**rapporteur**
M. Olivier	**SENTIEYS**	**rapporteur**
M. Régis	**LEVEUGLE**	**examinateur**
M. Jean	**GOBERT**	**examinateur**
M. Habib	**MEHREZ**	**directeur de thèse**

Résumé

Le travail effectué dans cette thèse concerne l'étude et la spécification d'un modèle de cœur de processeur configurable pour systèmes embarqués, et la définition d'une méthodologie de conception associée permettant la réalisation accélérée de processeurs DSP spécialisés.

Nous débutons par une étude détaillée des différents processeurs DSP existants, généraux puis spécialisés. Cette étude nous amène à proposer un modèle de processeur reposant sur une architecture configurable de type VLIW, qui permet à la fois de faire varier le nombre de ressources matérielles générales et d'intégrer des unités matérielles spécialisées destinées à accélérer certains algorithmes critiques. Le contrôle du processeur est assuré par un jeu d'instructions générique autorisant un degré de parallélisme variable et pouvant être spécialisé par l'ajout d'instructions utilisateur.

La méthodologie de conception associée est constituée de deux phases. La première phase d'exploration de l'espace de configuration vise à déterminer la meilleure configuration pour le processeur, en fonction des caractéristiques de l'application cible. Nous présentons le simulateur de jeu d'instructions et le compilateur reciblable nécessaires à l'implémentation de cette phase. Pour la deuxième phase de réalisation matérielle, nous proposons une méthode basée sur l'emploi de générateurs de macroblocs paramétrables et portables.

Nous mettons en œuvre cette méthodologie pour converger vers un processeur DSP spécialisé implémentant les fonctions bas-niveau du protocole GSM. L'analyse des résultats en terme de performance et de complexité matérielle montre clairement l'intérêt d'un modèle de processeur configurable, qui permet d'obtenir des processeurs dont le facteur performance/coût est meilleur que ceux des processeurs DSP généraux.

Mots Clés

Processeur DSP, modèle configurable, architecture VLIW, jeu d'instructions générique, exploration d'architecture, compilateur reciblable, simulateur de jeu d'instructions ,codeur EFR, algorithme de Viterbi, générateur paramétrable

Abstract

The work presented here deals with the study and specification of a configurable DSP core model intended to be used in embedded systems. It also includes the definition of an associated design methodology which aims to speed up the design process of specialized DSP processors.

We first present a detailed study of existing general/ specialized DSP processors. This study lead us to propose a DSP processor model based on a configurable VLIW architecture. It allows the user to adjust the number of general hardware resources and to include specialized custom units to improve performance of some critical applications. The control of the processor is managed by a generic, customizable, and variable length instruction-set.

The design methodology include two phases. The first phase is called the design space exploration phase and consists to determine the best configuration for the processor model with regard to the application constraints. We present the instruction-set simulator and retargetable compiler used in this phase. The second phase deals with the hardware synthesis of the final configured processor. We propose an implementation method based on portable and parametrizable macroblocks generators.

We apply the methodology to design a specialized DSP processor able to efficiently implement the GSM baseband functions. The analysis of the results in terms of performance and hardware cost clearly demonstrates the interest of a configurable processor model, which allows to obtain processors with best performance/cost ratio than those of general DSP processors.

Keywords

DSP processor, configurable model, VLIW architecture, generic instruction-set, design space exploration, retargetable compiler, instruction-set simulator, EFR vocoder, Viterbi algorithm, parameterizable macroblocks generator.

.

Table des Matières

Glossaire

AGU (Address Generation Unit) : unité de calcul d'adresse.

ASIC (Application-Specific Integrated Circuit) : Un circuit spécifique conçu pour implanter d'une façon optimale une fonction précise dans un système.

ASIP (Application-Specific Instruction-set Processor): processeur dont les caractéristiques matérielles et logicielles sont spécialement adaptés pour optimiser l'implémentation d'une application particulière.

BCU (Bus Controller Unit) : Unité de contrôle de bus.

BDTI (Berkeley Design Technology, Inc.) : organisme spécialisé dans l'estimation de performance des processeurs DSP (www.bdti.com).

CDFG (Control Data Flow Graph) : graphe utilisé par les compilateurs et modélisant les dépendances de données et de contrôle entre les opérations élémentaires d'un programme

Conception Matérielle/Logicielle : Une méthodologie de conception visant la réduction de l'écart entre l'implantation d'une fonction sur un ASIC ou en logiciel. Elle permet la conception simultanée de systèmes hétérogènes impliquant des ASICs et des processeurs de type général.

DRAM (Dynamic Random Access Memory) : Mémoire dynamique à accès aléatoire. L'information n'est pas statique et doit être rafraîchie périodiquement.

xDSL (Digital Subscriber Line): technologie permettant le passage simultané de la voix et de données numériques sur une ligne téléphonique classique, avec un débit très supérieur à celui offert par les modems analogiques classiques.

DSP (Digital Signal Processor) : Un processeur spécialisé pour les applications du traitement du signal.

DSP (Digital Signal Processing): Traitement numérique du signal.

FIFO (First In First Out) : Elément de mémorisation restituant l'information sauvegardée dans son ordre de stockage.

FFT (Fast Fourier Transform) : Transformée de Fourier rapide

FPGA (Field Programmable Gate Array) : Circuit spécifique dont la fonctionnalité est programmée par l'utilisateur.

FPU (Floating Point Unit) : Unité de calcul sur des nombres représentés en virgule flottante.

GSM (Global System for Mobile Communication): standard de téléphonie mobile utilisé en Europe et dans une grande partie du monde.

ICU (Interrupt Controller Unit) : Unité de contrôle d'interruptions.

ILP (Instruction Level Parallelism) : Parallélisme existant au niveau instruction.

IP (Intellectual Property) : Terme désignant une fonction réutilisable, déjà préconçue, et pouvant exister sous différentes formes (modèle HDL, modèle physique, etc.).

MAC: Multiplication-Accumulation.

MIPS (Million d'instructions par seconde): unité de mesure utilisée pour caractériser la puissance d'un processeur ou la complexité d'une application.

PDA (Personal Digital Assistant): assistant numérique personnel (Psion, Palm Pilot…)

RISC : Reduced Instruction Set Computer

SRAM : Static Random Access Memory

SoC (System On Chip): système complet composé de plusieurs circuits et intégré sur le même silicium

TNS: Traitement numérique du signal

VLIW : Very Long Instruction Word

VLSI : Very Large Scale Integration

Introduction

Pour la première fois dans la courte histoire des circuits intégrés, les microcontrôleurs ne dominent plus le marché des processeurs embarqués et ont été dépassés par les processeurs DSP, désormais leader dans le domaine [1]. L'explication provient bien sûr de l 'émergence de nouveaux domaines d'application très demandeurs en capacité de traitement du signal, la téléphonie mobile en étant l'exemple le plus marquant. C'est précisément ce domaine d'applications qui « consomme » actuellement le plus de circuits ; suivent ensuite les circuits utilisés dans les modems et les contrôleurs de disque dur.

Le point commun de ces domaines d'applications est qu'ils requièrent des puissances de calcul significatives pour effectuer en temps réel les traitements nécessaires à l'exécution des algorithmes DSP. Ils sont de plus soumis à de fortes contraintes de consommation électrique et de coût matériel. Les processeurs DSP, avec leur architecture et leur jeu d'instructions spécialement adaptés à ce type de contraintes, représentent très souvent le meilleur compromis « performance / coût / consommation / temps de développement » par rapport aux autres solutions intégrées : ASIC, microcontrôleurs et microprocesseurs généraux.

On distingue principalement deux types de processeurs DSP : les processeurs généraux et les processeurs spécialisés. Les premiers sont fabriqués par quelques grands constructeurs (principalement *Texas Instruments*, *Motorola*, *Lucent/Agere* et *Analog Devices*) et visent un panel d'applications assez large de manière à assurer une large distribution auprès des fabricants de systèmes de nature diverse : téléphonie mobile, vidéo numérique, traitement audio, etc. Ces processeurs intègrent une architecture et un jeu d'instructions volontairement général qui leur permet d'assurer de bonnes performances sur la plupart des applications, excepté pour celles dont les contraintes « hors-norme » réclament des architectures spécialisées. La généralité a évidemment un prix, celui du coût matériel et de la consommation électrique : pour assurer les performances, ces processeurs intègrent de grandes quantités de matériel, et fonctionnent à fréquence élevée.

A l'inverse, les processeurs spécialisés sont conçus exclusivement pour un domaine d'application clairement défini, ce qui leur permet en adaptant leur structure (architecture et jeu d'instructions) de « coller » au plus près des contraintes du cahier des charges en terme de performance, tout en réduisant au minimum les autres facteurs de coût : matériel et consommation. Ce genre de processeurs, connus sous le nom d'ASIPs (*Application Specific Instruction Set Processors*), est conçu en interne par les fabricants de circuits et intégrés dans leurs produits en lieu et place d'un processeur général. Cela permet d'une part de disposer d'un processeur performant et parfaitement adapté, mais

aussi de réduire le coût global du système en économisant le prix d'achat de la licence nécessaire pour l'usage d'un processeur provenant d'un autre fabricant.

Cette solution a néanmoins un inconvénient majeur : le temps de conception. Développer un processeur spécialisé prend du temps : il faut concevoir et tester le processeur lui-même, mais aussi les outils de développement logiciels et les librairies applicatives, sans lesquelles le processeur n'a aucune chance de succès. L'amortissement de ce coût de développement ne peut se faire que si le nombre d'unités vendues est suffisant, c'est pourquoi on trouve les ASIPs surtout dans des domaines d'application grand public comme la téléphonie mobile.

Avec l'accroissement de la complexité des applications et des architectures des processeurs DSP (les nouvelles architectures VLIW ont introduit de nouvelles difficultés liées à la gestion du parallélisme d'instructions), et la durée de mise sur le marché des produits toujours plus courte, les contraintes de développement qui pèsent sur les équipes de conception sont de plus en plus fortes. Il devient dès lors nécessaire de développer de nouvelles méthodologies permettant de réduire la durée du cycle de développement.

Un moyen de réduire le temps de développement consiste à tirer profit des développements précédents en réutilisant à la fois les blocs matériels et les outils logiciels. Et une manière efficace d'implémenter ce concept de réutilisation consiste à utiliser un modèle de processeur configurable. En fonction des caractéristiques de l'application cible, le concepteur paramètre le modèle matériel (architecture) et logiciel (jeu d'instructions) du processeur afin d'obtenir une version finale performante et à faible coût. Les ressources matérielles intégrées peuvent être les mêmes que celles utilisées dans un projet précédant. Et surtout, les outils de développement logiciel et de synthèse matérielle prennent en compte les paramètres du modèle et n'ont donc pas à être reconçus pour chaque nouveau processeur.

Objectifs de nos recherches

Le but de ce travail est l'étude et la spécification d'un modèle de cœur de processeur configurable pour systèmes embarqués, et la définition d'une méthodologie de conception logicielle et matérielle associée permettant la réalisation rapide de processeurs *ASIP* performants et à faible coût. Pour supporter les charges de calcul élevées des applications modernes, notre modèle se base sur une architecture VLIW qui permet une meilleure exploitation du parallélisme d'instructions. Le nombre de ressources matérielles générales du processeur (unités fonctionnelles, bande passante mémoire, registres internes, etc.) doit être paramétrable afin de satisfaire les exigences de l'application cible. Le modèle doit aussi permettre l'intégration de ressources matérielles spécialisées commandées par le jeu d'instructions afin d'optimiser l'exécution de certaines parties critiques de l'application qui ne peuvent se satisfaire d'un jeu d'instruction général.

La méthodologie de conception peut être divisée en deux étapes distinctes. La première consiste en une phase d'exploration de l'espace de configuration du processeur, qui doit permettre de déterminer les valeurs optimales des différents paramètres du modèle, et de définir d'éventuelles structures spécialisées à intégrer dans le processeur. Une fois le modèle entièrement spécifié, le processeur correspondant peut être réalisé. C'est le but de l'étape de réalisation matérielle.

Notre travail porte avant tout sur la première phase d'exploration. Notre objectif est de définir une méthode qui permette, à partir du code source de l'application cible, d'extraire les caractéristiques

pertinentes de l'application, de proposer en conséquence différentes configurations du modèle de processeur et de les évaluer en regard des contraintes particulières de l'application en terme de performance, consommation ou coût matériel. Ceci passe entre autres par la définition d'une méthode d'analyse de l'application cible, et la réalisation d'un simulateur d'instructions et d'un compilateur pour permettre l'évaluation des différentes configurations.

Pour vérifier l'intérêt de la méthode sur un exemple significatif, nous allons l'appliquer à la conception d'un processeur spécialisé pour la téléphonie mobile, qui aura pour charge d'implémenter l'ensembles des fonctions de base du protocole GSM. Les résultats en performance et en coût matériel devront être comparés aux implémentations équivalentes sur des processeurs DSP généraux.

La réalisation physique proprement dite n'étant pas l'objet de ce travail, nous nous contenterons de proposer une stratégie globale de synthèse matérielle basée sur l'emploi de macro-générateurs paramétrables et d'un langage de description de jeu d'instructions.

Organisation du mémoire

Le manuscrit est organisé en deux grandes parties : une partie « état de l'art » présentant les solutions existantes en matière de processeurs DSP généraux, spécialisés et configurables (chapitres 1 et 2), et une partie exposant nos travaux.

Détaillons maintenant le contenu de chaque chapitre :

- Parce que ces dernières années ont été riches en innovations dans le domaine des processeurs DSP, il nous a semblé indispensable pour la compréhension de la suite de rappeler les caractéristiques principales de ces processeurs et de présenter les dernières techniques architecturales et logicielles qu'elles implémentent. Le lecteur déjà familiarisé avec ce type de processeur peut éventuellement se dispenser de la lecture de ce premier chapitre. Nous présentons tout d'abord les caractéristiques architecturales et logicielles des premiers processeurs DSP dits « conventionnels », qui sont représentatifs des spécificités de ces processeurs par rapport aux autres solutions d'implémentation. Nous nous intéressons ensuite aux nouvelles architectures DSP apparues très récemment, qui utilisent les différentes formes de parallélisme présentes dans les programmes pour accroître leur performance : processeurs « conventionnels étendus », processeur VLIW, processeur SIMD, et processeurs hybrides RISC/DSP. Nous discutons du rôle de plus en plus important des compilateurs dans les nouvelles méthodologies de développement de code pour ces nouveaux processeurs. Enfin, nous concluons sur l'état actuel de l'offre en terme de processeurs DSP et sur les futures tendances.
- Alors que le chapitre précédant traitait des processeurs DSP généraux, nous nous intéressons ici aux processeurs spécialisés et configurables. Après avoir illustré le concept des processeurs DSP spécialisés (les *ASIP*s) et les raisons de leur utilisation, nous énumérons les différentes techniques de spécialisation architecturales et logicielles qui permettent d'optimiser le comportement d'un processeur en fonction des contraintes d'une application particulière. Nous présentons ensuite le concept de processeur

configurable, qui permet au concepteur d'ajuster un certain nombre de paramètres de l'architecture et du jeu d'instructions pour coller au plus près des besoins d'une certaine application. Nous examinons les processeurs configurables existants, le flot de conception associé, et les différentes techniques mises en œuvre pour la configuration de l'architecture et du jeu d'instructions. Nous concluons sur l'intérêt d'un processeur VLIW configurable et spécialisable.

- Dans le troisième chapitre, nous proposons un modèle de processeur VLIW configurable destiné à la réalisation de processeurs spécialisés de type ASIPs. L'architecture générale du processeur est composée de trois unités matérielles : une unité de calcul, une unité d'adressage et une unité de contrôle. Pour chacune de ces unités, nous proposons une architecture matérielle configurable et un jeu de paramètres associé. Le grand nombre de paramètres structurels et la possibilité d'intégrer des structures matérielles et logicielles « utilisateur » offrent au processeur un haut degré de spécialisation qui doit permettre la réalisation de processeurs ASIP performants et faible coût. Pour exploiter l'architecture configurable et spécialisable du processeur, nous proposons un jeu d'instructions générique supportant un parallélisme matériel variable et permettant l'ajout d'instructions utilisateur. Nous présentons alors la syntaxe du jeu d'instructions, l'ensemble des opérations de base pour chacune des unités, et les possibles aléas liés à la structure du pipeline.
- Le quatrième chapitre présente la méthodologie de conception associée au modèle de processeur configurable présenté précédemment. Comme nous avons choisi de valider cette méthodologie sur une application réelle, nous présentons brièvement les différentes fonctions composant la chaîne de traitement GSM dont plusieurs vont servir de fonctions test pour la spécialisation et la définition des paramètres du processeur. Le flot de conception général regroupant les deux phases d'exploration et de synthèse matérielle est ensuite présenté. Dans la troisième partie, nous détaillons ce qui constitue l'essentiel de ce travail : la phase d'exploration de l'espace de configuration. Nous illustrons sur un exemple d'application (le codeur de voix EFR) la première étape d'analyse de complexité qui examine les caractéristiques dynamiques du code source de l'application, identifie les fonctions critiques et fournit une liste d'opérateurs arithmétiques à intégrer au jeu d'instructions du processeur. Puis nous présentons la structure et le fonctionnement des deux outils logiciels indispensables aux étapes suivantes de la phase d'exploration que nous avons réalisé : le simulateur d'instructions et le compilateur. Pour terminer, nous discutons de la dernière phase de réalisation matérielle, et présentons le concept de macro-générateurs paramétrables utilisés pour la réalisation du chemin de données du processeur.
- Le cinquième et dernier chapitre présente la mise en œuvre de la méthodologie pour aboutir à une version finale entièrement configuré du modèle de processeur offrant des performances optimales pour les fonctions critiques du protocole GSM (Codeur de voix EFR et algorithme de Viterbi), et ce à un coût matériel réduit. La première section présente le processeur DSP de *Texas Instruments* TMS320C62, dont l'architecture et le jeu d'instruction très généraux vont nous permettre par comparaison de tester l'impact des

différentes optimisations présentes dans le processeur final. La phase d'analyse de complexité ayant été effectuée au chapitre précédant, nous débutons par l'étape de codage en assembleur des fonctions critiques. La méthodologie de programmation employée est illustrée au travers de l'exemple de codage de la fonction la plus complexe du codeur EFR, la fonction « search_10i40 ». Nous donnons alors les résultats de performance pour l'ensemble des fonctions critiques de l'EFR et l'algorithme de Viterbi, et les comparons à ceux du TMS320C62. Une fois les fonctions critiques programmées, nous déterminons pour chaque unité matérielle du processeur les paramètres de configuration en tenant compte des contraintes et exigences de ces fonctions. Nous proposons ensuite une structure d'encodage des opérations élémentaires et des instructions VLIW, en fonction des instructions sélectionnées du processeur et du parallélisme d'instruction requis par les fonctions critiques. Au final, nous estimons et comparons la complexité matérielle de trois versions différentes de notre processeur ASIP et du TMS320C62 et énumérons les différents compromis performance/coût matériel possibles. Au regard des résultas obtenus, nous concluons sur l'intérêt de notre approche. Enfin, nous identifions les futurs travaux et perspectives de recherche à mener dans le futur.

- La conclusion résume les différents travaux effectués, les résultats obtenus, et présente les contributions apportées par cette thèse dans différents domaines de recherche. Enfin, elle resitue les recherches menées dans le contexte industriel actuel.

Chapitre 1

Les processeurs de traitement du signal

Sommaire :

1.1 Introduction

La croissance phénoménal du marché des circuits de traitement du signal est avant tout lié au succès des processeurs DSP, généraux ou spécialisés. Ce succès est en partie du aux limitations des autres solutions matérielles (ASIC, microprocesseurs, microcontrôleurs, FPGA) du point de vue du traitement du signal. A ce sujet, on pourra consulter le rapport [2] qui présente en détail l'ensemble des solutions numériques de traitement du signal. Dans ce premier chapitre, nous nous intéressons à ce qui fait la spécificité des processeurs DSP et en quoi ils sont particulièrement adaptés aux applications de traitement du signal.

L'histoire des processeurs DSP a connu un bouleversement au milieu des années 90 avec l'apparition du premier processeur VLIW, le *TMS320C6x* de *Texas Instruments*. Avant cette date, quasiment tous les processeurs se basaient sur une architecture à un voire deux multiplieurs et un jeu d'instructions de type RISC avec un encodage complexe des instructions. Cette architecture et ce type de jeu d'instruction découlaient directement de la structure des filtres numériques de type FIR (Figure 1) [3]. Aujourd'hui, l'architecture des DSP se diversifie et emprunte aux microprocesseurs généraux des techniques évoluées destinées à accroître la performance en profitant du parallélisme de données ou d'instructions des algorithmes. Cela s'est traduit par l'apparition de nombreux processeurs à architecture VLIW et dans une moindre mesure superscalaire.

La prochaine section explore les caractéristiques générales que l'on retrouve dans la plupart des DSPs, et qui sont la conséquence directe de leur spécialisation pour le traitement du signal. On présentera en particulier l'architecture dite « conventionnelle » que l'on trouve dans les DSPs de première génération et dans encore une grande majorité des processeurs d'aujourd'hui. Après l'examen des différentes formes de parallélisme inhérentes à une application, on s'intéressera aux nouvelles architectures des DSPs les plus récents qui permettent de les exploiter. On reviendra ensuite sur le rôle et la performance des compilateurs actuels pour DSP. En conclusion, on fera le point sur l'état actuel de la technologie DSP et sur les futures tendances concernant son évolution.

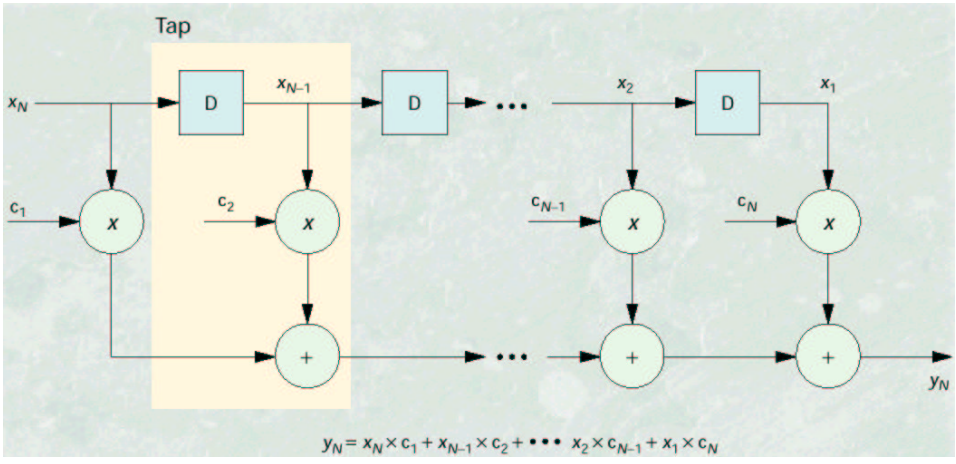


Figure 1 Architecture parallèle pour un filtre FIR

1.2 Caractéristiques et architectures des DSPs conventionnels

On présente ici quelques unes des caractéristiques communes aux différents processeurs DSP ainsi que le type d'architecture classique que l'on trouve dans les processeurs dits « conventionnels » (en

opposition aux processeurs les plus récents de type SIMD, VLIW, superscalaire qui seront présentés plus loin). Cette présentation ne prétend pas être exhaustive et peut être complétée par la lecture de [3] pour des informations complémentaires.

Comme on l'a dit précédemment, l'architecture des processeurs conventionnels est directement inspirée de la structure des filtres numériques comme le filtre FIR de la Figure 1. Son mécanisme de fonctionnement est simple. Les blocs D correspondent à des registres mémorisant les échantillons d'entrée durant un cycle et formant une « ligne à retard ». A un instant donné, les N-1 derniers échantillons reçus résident dans la ligne à retard, N étant la longueur du filtre. A l'arrivée d'un nouvel échantillon, ceux déjà présents dans la ligne sont décalés d'un rang et une nouvelle valeur en sortie est calculée en sommant les résultats des N multiplications Xk*Ck (Xk étant le $k^{ème}$ échantillon mémorisé et Ck le $k^{ème}$ coefficient du filtre). En termes mathématiques, cela revient à calculer à chaque cycle le produit scalaire de deux vecteurs réclamant *N* multiplications et *N-1* additions. La structure du filtre de la figure 1 apparaît comme la mise bout à bout d'une structure de base appelé *tap* et réalisant une opération de multiplication-accumulation (MAC), l'opération la plus massivement utilisée dans les algorithmes de traitement du signal.

1.2.1 Chemin de données

Le chemin de données et l'architecture mémoire sont les deux caractéristiques principales qui différencient les processeurs DSP des processeurs classiques. Bien évidemment, les chemins de données des processeurs DSP sont spécialement conçus pour traiter efficacement les calculs spécifiques au traitement du signal. Cela passe par l'implémentation efficace des opérations les plus utilisées (Multiplication-accumulation) et une topologie différente de celle des processeurs classiques.

1.2.1.1 Arithmétique fixe / flottante

Il existe deux catégories bien distinctes de DSPs : les processeurs entiers (ou *virgule fixe*) et les processeurs flottants. Les premiers sont, et de loin, les plus utilisés, principalement pour une question de coût (Figure 3). On ne détaillera pas ici les principes mathématiques de ces deux approches. Pour une explication détaillée, on pourra se reporter à [3] ou [4]. On rappellera simplement qu'en virgule fixe les nombres sont codés en complément à 2 et peuvent être de deux types : entiers ou fractionnaires (compris entre –1 et 1). Un nombre flottant utilise trois champs distincts : un bit de signe, une mantisse et un exposant (Figure 2).

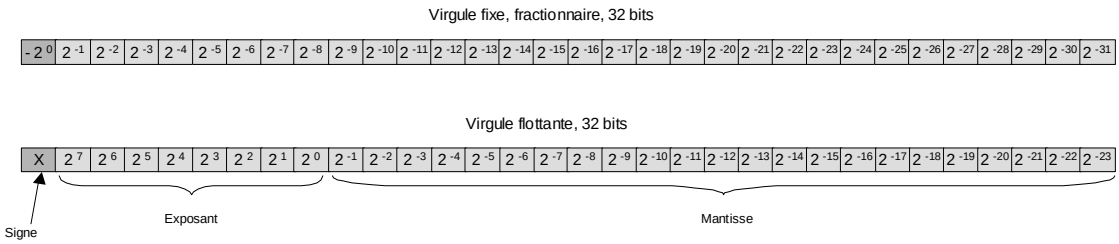


Figure 2 Représentation des nombres en virgule fixe et en flottant

Il est cependant intéressant d'expliquer les raisons qui conduisent un utilisateur à privilégier l'une ou l'autre des deux approches. Ce choix résulte d'un compromis entre trois principaux facteurs : la simplicité d'utilisation, la précision numérique requise et le coût matériel (surface et consommation).

On a souvent coutume de dire que les processeurs flottants offrent d'avantage de précision que les processeurs à arithmétique entière. En fait, cela n'est pas toujours vrai et dépend du type de signal traité. En entier, la précision (c'est à dire la plus petite différence représentable entre deux valeurs consécutives) est constante tout au long de l'intervalle de représentation des nombres, alors qu'elle est variable en flottant (inversement proportionnelle à la valeur absolue du nombre). La dynamique du signal (le rapport entre la plus petite et la plus grande valeur représentable) est beaucoup plus grande en flottant (1535dB en 32 bits) qu'en entier (187dB en 32 bits).

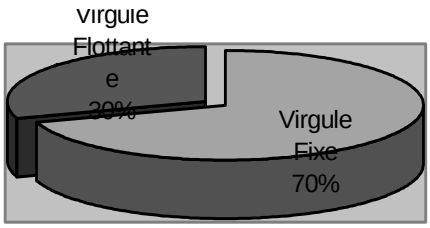


Figure 3 Usage des processeurs Virgule Fixe / Virgule Flottante

En général, les signaux issus du « monde réel » (c'est à dire provenant directement de capteurs comme dans les applications de radar, sonar ou radio) se prêtent mieux au flottant, car ils réclament une dynamique de signal élevée : les réponses perçues par de tels systèmes peuvent être très grandes ou très faibles, et sont soumises aux conditions d'environnement et aux bruits, ce qui les rend difficilement quantifiables. Traiter de tels signaux en arithmétique entière nécessite une mise à l'échelle des valeurs reçues (à l'aide d'opérations de décalage) pour qu'elles tiennent dans l'intervalle des valeurs représentables. Et dans le cas de signaux ayant un rapport signal / bruit faible, cela peut conduire à la perte des informations pertinentes.

A l'inverse, les signaux dits « reconstruits », c'est à dire ceux qui sont issus de dispositifs électroniques, ont des amplitudes réduites (souvent égales à la tension d'alimentation du dispositif) et requièrent donc une dynamique plus faible. C'est le cas des algorithmes utilisés en télécommunication (compression, décompression, filtrage…) qui opèrent sur des signaux dont la dynamique est fixée, et qui ont été conçus et étudiés pour permettre une implémentation efficace en arithmétique entière. Par exemple, les applications Telecom requièrent généralement une dynamique aux alentours de 50dB, et les applications Hi fi environ 90dB, qui sont des valeurs compatibles avec une réalisation en arithmétique entière 32 bits [5].De plus, le rapport signal / bruit de ces applications est généralement plus grand que dans le cas des applications traitant des signaux réels, ce qui les rend moins sensibles aux opérations de mise à l'échelle.

La raison pour laquelle les processeurs flottants sont moins utilisés est qu'ils sont plus coûteux au niveau matériel : les opérateurs flottants sont plus complexes, consomment plus et occupent plus de silicium que leurs équivalents entiers. Or, le marché des processeurs DSP est actuellement dominé par les applications de Telecom embarquées, qui ont de fortes contraintes de coût et dont la nature des signaux traités ne justifie pas l'usage du flottant.

Une raison qui peut justifier l'emploi du flottant est sa facilité d'utilisation qui réduit considérablement le temps de développement par rapport à une solution en virgule fixe. En flottant, il n'y a pas besoin d'opérations de décalage pour la mise à l'échelle : la programmation se fait de façon naturelle, toutes les opérations de normalisation sont réalisées automatiquement par le matériel et sont

transparentes du point de vue de l'utilisateur. En revanche, en virgule fixe, tout l'art de la mise à l'échelle consiste à insérer, au bon endroit dans le code de l'application, les opérations de décalage qui assureront une dynamique et une précision maximale tout en évitant les erreurs liées aux débordements arithmétiques. Cette opération nécessite au préalable une étude précise à l'aide d'outils d'analyse et de simulation qui modélisent le comportement de l'application au bit près. Le code final inclut, en plus des opérations de calcul proprement dites, les opérations de décalages et de gestion de la précision (arrondi, saturation) qui alourdissent considérablement le programme ; pour un compilateur, générer du code flottant est évidemment beaucoup plus simple.

Enfin, il est intéressant de noter que la plupart des processeurs virgule fixe permettent, lorsque l'application l'exige, de simuler des calculs en virgule flottante. Ces processeurs utilisent la technique dite du « block floating-point » qui consiste à associer à un groupe de valeurs un ensemble de mantisses et un exposant unique pour l'ensemble des valeurs. Grâce à des mécanismes matériels ajoutés au chemin de données, cette technique permet d'effectuer des calculs flottants et donc de bénéficier d'une meilleure dynamique. Son utilisation reste cependant plus complexe et plus limitée que la programmation sur un « vrai » processeur flottant.

Notre travail portant sur les cœurs de processeurs pour systèmes embarqués faible coût, nous avons choisi une implémentation matérielle en virgule fixe. Toutes les considérations qui suivent décrivent les caractéristiques des processeurs DSP à virgule fixe.

1.2.1.2 Largeur de données

La largeur de données native d'un processeur est la largeur maximum des données pouvant circuler sur ses bus et dans son chemin de données. La plupart des processeurs fonctionnent en 16/32bits, ces deux chiffres correspondant à des données en simple/double précision.. Certains processeurs fonctionnent aussi en 24/48 bits.

La largeur de données influe directement sur la précision arithmétique et la dynamique maximale du signal, mais aussi et surtout sur le coût matériel et sur la consommation du circuit: la largeur de données conditionne la taille des bus, le dimensionnement du chemin de données, la taille de la mémoire, le nombre de pattes du circuit, etc. Là encore, il s'agit de faire un compromis entre la performance et le coût désirés. Les applications virgule fixe sont spécifiées pour des largeurs de données particulières : les algorithmes d'encodage de voix du GSM par exemple utilisent deux types de données uniques de largeur 16 et 32 bits. La solution la plus simple consiste alors à utiliser un DSP dont la largeur de données est supérieure ou égale à celle de l'application visée. Le choix des largeurs 16/32 ou 24/48 des constructeurs n'est pas innocent et découle de l'analyse des algorithmes les plus utilisés dans les domaines d'application correspondant à leur processeurs : la téléphonie mobile et les applications de type modem utilisent du 16 bits, tandis que les applications audio demandent une dynamique plus large et utilisent du 24 bits.

Pour réduire les coûts, il est aussi possible d'utiliser un processeur de largeur inférieure à celle de l'application; dans ce cas, c'est le logiciel qui devra compenser l'insuffisance matérielle en décomposant par exemple une addition 32 bits en deux additions séquentielles 16 bits. Les principaux inconvénients de cette solution sont une complexité de programmation accrue et une diminution de la performance.

A l'instar du filtre FIR, la plupart des algorithmes DSP utilisent principalement l'opération MAC qui consiste à accumuler le résultat d'un produit avec un résultat précédent. Le produit de deux termes de N bits donnant un résultat sur 2N bits, les registres et les unités fonctionnelles inclus dans l'opérateur MAC sont surdimensionnées pour ne pas perdre de bits significatifs. L'ALU et les accumulateurs A et B du processeur de la Figure 4 ont une largeur de 56 bits pour une largeur de donnée native de 24 bits. En plus des 48 bits codant le résultat de la multiplication 24*24, on trouve 8 bits supplémentaires appelés « bits de garde ». Ils permettent d'éviter les erreurs de débordements lors des opérations d'accumulation des valeurs 48 bits, autorisant ainsi l'enchaînement séquentiel de plusieurs opérations MAC. Ce genre de mécanisme est généralisé dans la plupart des processeurs DSPs (les processeurs 16 bits utilisent des accumulateurs de 40 bits).

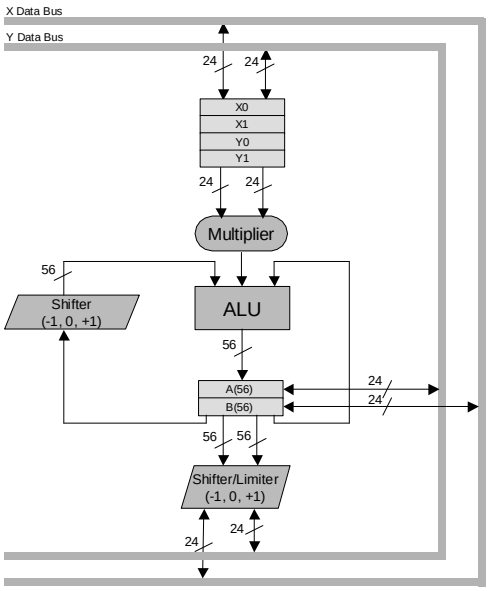


Figure 4 Une architecture classique: le Motorola DSP5600x

1.2.1.3 Unités fonctionnelles

Principalement dédiés aux calculs mathématiques, les processeurs DSP disposent d'unités fonctionnelles spécialisées effectuant ces calculs plus rapidement et de manière plus précise que les processeurs classiques. Dans ces derniers, tous les calculs arithmétiques sont effectués au moyen d'une unité arithmétique et logique (voire un multiplieur câblé pour les processeurs les plus récents) dont les performances et fonctionnalités sont limitées et la gestion de la précision difficile à contrôler.

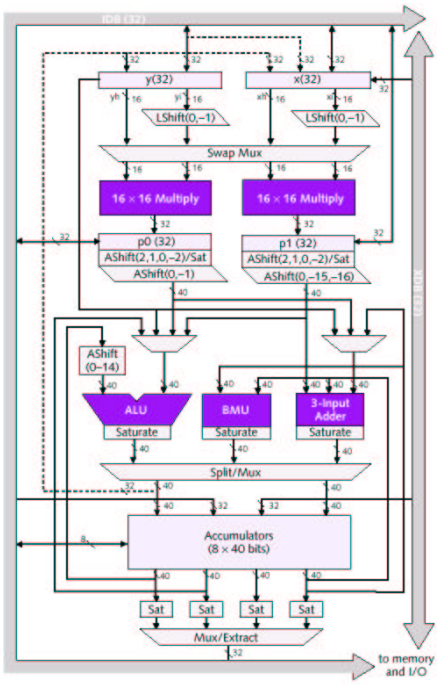
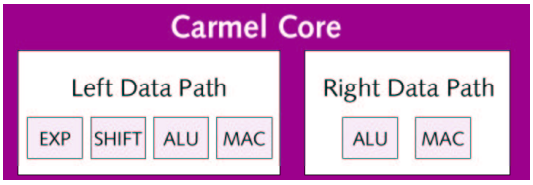


Figure 5 Chemin de données du DSP16xxx

Bien qu'il existe de nombreuses variantes, on peut classer les unités fonctionnelles des DSPs en trois grandes catégories que l'on retrouve quasi-systématiquement dans tous les processeurs :

- **Unité Arithmétique et Logique** (ALU) : cette unité dispose des fonctionnalités classiques présentes dans tous les processeurs : addition / soustraction et opérations logiques élémentaires (AND, OR, XOR, NOT) (cf. l'exemple du processeur Siemens CARMEL Figure 6). Elle intègre aussi des fonctionnalités plus spécifiques liées à la gestion de la précision comme les fonctions de saturation ou d'arrondi, et des opérations arithmétiques évoluées comme la valeur absolue, le maximum/minimum de deux valeurs, le calcul itératif de division, etc.

- **Multiplieur** ou Multiplieur-Accumulateur (MAC) : cette unité effectue en une seule instruction une multiplication, plus une addition dans le cas du MAC. La plupart des DSPs exécutent ces opérations en un cycle machine, ce que même les microprocesseurs les plus puissants sont généralement incapables de faire. Grâce à l'ALU et au multiplieur, les DSP sont donc capables de calculer une Multiplication-Accumulation à chaque cycle. La combinaison d'un multiplieur et d'un additionneur dans la même unité fonctionnelle forment un opérateur MAC. Il en existe deux types : purement combinatoire (cas du 5600x), ou pipelinés au moyen d'un registre « produit » entre le multiplieur et l'accumulateur (registres p0 et p1 du Lucent DSP16xxx de la Figure 5). Parmi les fonctions proposées, on trouve la multiplication simple et l'addition, la multiplication-accumulation, la multiplication-soustraction et parfois la fonction « carrée » (Figure 6).



Operation	Data Path	Operation
Arithmetic		**Multiply**
Add/Subtract	L & R	Multiply-Accumulate
Add Absolute Value	L & R	Mul-Accum Aligned
Add w/Carry	L & R	Multiply-Subtract
Subtract w/Borrow	L & R	Square
Add/Sub High/Low	L & R	Square-Accumulate
SIMD Add/Sub	L & R	**Bit Manipulation**
Negate	L & R	Change/Set/Clear Bit
Round	L & R	Test Bit
Limit	L & R	Test Field
Absolute Value	L & R	Compare
Exponent	L	Shift Arith/Logical
Divide Step	L & R	Insert/Extract
Min/Max	L & R	Rotate thru Carry
Increase/Decrease	L & R	**Logical**
Clear Accumulator	L or R	AND/OR
Clear & Rnd Accum	L or R	NOT/XOR

Figure 6 Unités fonctionnelles et opérations du Siemens CARMEL

- **Décaleurs** (*Shifters*) et unités de manipulation de bits (BMU). Ces unités se chargent de tous les traitements incluant des opérations de décalage et de test au niveau bit. Elles peuvent aller du simple décaleur « 1 bit vers la droite ou 1 bit vers la gauche » du 5600x (Figure 4) à une unité complexe dite de « manipulation de bit » effectuant des opérations de décalage multi-bits, de comparaison, de rotation, de détection d'exposant, d'insertion et d'extraction de champs de bits (Figure 6). Les opérations de décalage sont généralement utilisés pour la mise à l'échelle des valeurs afin d'éviter les débordements arithmétiques ; associées à la détection d'exposant, elles permettent la normalisation des valeurs dans les calculs de type « block floating-point ». Les opérations de manipulation de bits sont surtout utiles pour les applications incluant du contrôle et les algorithmes traitant les données sous forme de flux de bits de longueur variable. Elles servent aussi pour l'implémentation d'applications « bit-exact » lorsque les opérations réalisables par les unités fonctionnelles du processeur ne correspondent pas exactement aux opérateurs arithmétiques utilisés par l'algorithme.

Cette classification n'est évidemment pas universelle, la complexité, le type et le nombre d' unités variant selon les processeurs : le Siemens CARMEL utilise deux unités distinctes pour effectuer les opérations de décalages et de détection d'exposant (EXP et SHIFT, cf. Figure 6), alors que le Lucent DSP16xxx les regroupe toutes en une seule unité (BMU, cf. Figure 5). On retrouve cependant la plupart de ces fonctionnalités dans tous les processeurs.

Bien que qualifiés de « généraux », certains processeurs DSPs disposent en plus d'unités fonctionnelles spécialisées pour un domaine d'application particulier et qui leur permettent d'accélérer l'exécution de certaines fonctions critiques. C'est le cas par exemple de la plupart des DSPs destinés à la téléphonie mobile qui disposent de dispositifs matériels pour le décodage de Viterbi utilisé dans la norme GSM. Le récent TMS320C64x de Texas Instruments inclut un multiplieur spécial dit « multiplieur de Galois » afin d'accélérer le décodage « Reed Solomon » présent dans la norme ADSL. L'usage de ce genre d'unités est très répandu dans les processeurs ASIP (*Application Specific Instruction-Set Processor*, cf. Chapitre 2) qui sont spécialisés pour un domaine d'application particulier (téléphonie, audio, vidéo, etc.).

1.2.1.4 Topologie

Dans les microprocesseurs classiques, la mémorisation interne des données se fait au moyen d'un banc de registres central à l'architecture, souvent multi-ports, et qui sert à la fois aux calculs arithmétiques et aux calculs d'adresses ; dans ce type d'architecture, chaque registre du chemin de données a la même visibilité vis-à-vis des unités fonctionnelles. Du point de vue de la programmation, cela signifie que les registres sont complètement interchangeables. On parle dans ce cas d'un jeu de registres « homogène ».

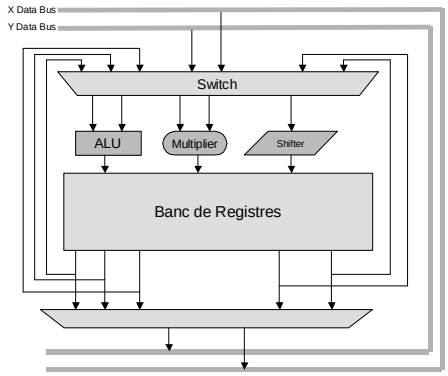


Figure 7 Chemin de données homogène

La stratégie utilisée dans les processeurs DSP classiques est radicalement différente. Certains registres sont dédiés à des opérateurs particuliers et ne peuvent être accédés que comme opérandes sources ou résultat des instructions associées à l'opérateur. Le DSP16xxx est un bon exemple d'architecture à jeu de registres « hétérogène » : les registres X et Y sont dédiés aux deux multiplieurs pour fournir les opérandes sources, et les résultats des multiplications ne peuvent être rangés que dans les registres p0 et p1. L'utilisation d'une donnée située dans le banc de registres « Accumulateurs » comme opérande source d'une multiplication ne peut se faire qu'en transférant au préalable cette donnée dans un des registres X ou Y.

Historiquement, la raison d'être de ce type d'architecture se justifie par la recherche du coût matériel minimum, les DSPs ayant des contraintes de coût et de consommation plus fortes que celles des microprocesseurs. La Figure 7 montre une architecture à jeu de registres « homogène » équivalente à celle du 5600x. Elle utilise un banc de registres multi-ports (3 ports d'entrée, 5 ports de sortie) permettant de fournir suffisamment de données pour effectuer en parallèle une multiplication, une addition et un décalage, l'équivalent des capacités du 5600x. Or, à nombre de registres égal, la solution banc de registres est beaucoup plus coûteuse à tous les niveaux (en performance, en matériel et en consommation). De plus, la largeur en bits des bus et des registres étant fixée par les données de plus grande taille (ici 56 bits en sortie du multiplieur), la taille de la plupart des éléments du chemins de données passe de 24 bits à 56 bits, rendant cette solution globalement prohibitive en terme de coût et de performance. Les concepteurs ont donc choisi d'utiliser des architectures hétérogènes minimisant le matériel et favorisant la performance. Les topologies de chemin de données choisies, bien que variables selon les processeurs, tendent toutes à optimiser l'exécution des algorithmes DSP les plus répandus, filtres FIR en tête.

L'inconvénient principal de ce type d'architecture est son manque de souplesse au niveau de la programmation. Comme on l'a précédemment souligné, la circulation des données dans ce type d'architecture est plus complexe que dans le cas des structures homogènes du fait de fortes contraintes topologiques : tel registre n'est connecté qu'à telle unité, tel résultat doit aller obligatoirement dans tel registre et pas un autre. De plus, les ressources de mémorisation (registres internes) sont réduites au minimum et rendent la tache d'allocation de registres particulièrement délicate. C'est pourquoi la programmation de ce type de DSP est très difficile et demande beaucoup d'attention et d'expérience. C'est aussi la raison pour laquelle les compilateurs ont des performances peu satisfaisantes, qui obligent la plupart du temps les utilisateurs à coder manuellement leurs applications directement en assembleur.

1.2.2 Architecture Mémoire

1.2.2.1 Architecture Harvard

Une grande différence entre les microprocesseurs et les DSP concerne l'architecture mémoire, qui définit la manière dont le cœur du processeur accède aux données et aux instructions. Traditionnellement, les microprocesseurs se basent sur une architecture de type *Von Neumann*, qui utilise un bus unique pour l'accès aux données et aux instructions (Figure 8a). Cette solution, satisfaisante pour les applications d'usage général, trouve ses limites dans le cas des applications traitement du signal. En effet, la plupart des algorithmes DSP exigent une bande passante mémoire supérieure à celle que peut fournir l'architecture *Von Neumann*. Pour le filtre FIR par exemple, dans le cas où l'on désire effectuer l'équivalent d'un *tap* par cycle d'instruction, le processeur doit calculer une multiplication-accumulation et accéder plusieurs fois à la mémoire dans la même instruction. Précisément, à chaque *tap*, le processeur doit :

- charger l'instruction de Multiplication-Accumulation
- lire l'échantillon approprié dans la ligne à retard
- lire le coefficient du *tap* courant
- calculer le produit *Coefficient*Echantillon* et l'additionner au résultat précédant.
- écrire l'échantillon lu dans l'emplacement suivant de la ligne à retard, afin de décaler les échantillons dans la ligne.

Cela représente un total de quatre accès à la mémoire par cycle. En pratique, diverses techniques permettent de réduire ce nombre à trois voire deux accès par cycle, ce qui reste de toute façon au delà de la limite de « 1 accès par cycle » de l'architecture *Von Neumann*. Un processeur DSP possédant une unité arithmétique capable d'effectuer une unité MAC et utilisant une architecture de ce type serait inefficace puisque incapable de fournir un échantillon et un coefficient par cycle, nécessaires pour nourrir l'unité MAC.

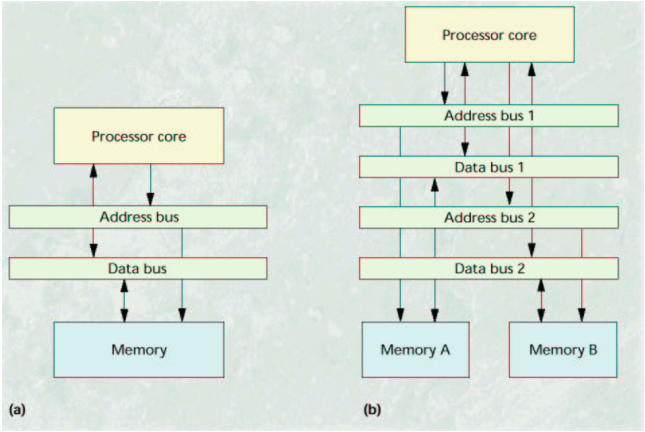


Figure 8 Architecture mémoire de type Von Neumann (a) et Harvard (b)

C'est pourquoi les processeurs DSP utilisent une architecture mémoire différente, dite de type *Harvard* (Figure 8b). Dans cette architecture, plusieurs bus d'adresses et de données sont utilisés, qui permettent d'effectuer plusieurs accès simultanés par cycle à la mémoire. L'architecture Harvard classique inclut un jeu de bus (Adresse et Données) *Programme* et un jeu de bus *Données* , le premier servant à charger l'instruction suivante et le deuxième à charger une donnée requise par l'instruction courante, offrant ainsi un débit effectif d'une instruction et une donnée par cycle. Des architectures plus avancées, dites *Enhanced Harvard*, proposent d'augmenter encore le nombre de bus afin d'offrir des bandes passantes plus importantes de 2, 4, voire 8 données par cycle. C'est le cas du DSP5600x de Motorola (Figure 4), qui dispose de trois espaces mémoire distincts et leurs bus associés : un pour les instructions, et deux pour les données (bus X et Y). Ce processeur peut donc, à chaque cycle, charger une instruction et deux données.

1.2.2.2 Mémoire On Chip

Contrairement aux microprocesseurs, la plupart des DSP n'utilise pas de mécanismes de mémoires cache, ni pour les données et encore moins pour les instructions. En effet, la taille des algorithmes de traitement du signal est généralement faible (quelques dizaines de ligne de C), le code résultant pouvant tenir dans des espaces mémoire restreints. De ce fait, la philosophie dominante pour les processeurs DSP est de placer la mémoire instruction directement sur le circuit, en général sous forme de mémoire rapide de type SRAM répondant en un cycle ; cela permet d'assurer un débit d'une instruction par cycle, l'équivalent d'un cache instruction.

Un espace est aussi réservé sur le circuit pour des blocs de mémoire Donnée, offrant là encore un débit maximal grâce à des accès en un cycle. La plupart des algorithmes DSP consomment les données sous forme de flux : un échantillon est lu, utilisé pour le calcul puis l'échantillon suivant est lu et placé à l'endroit du précédent. La taille mémoire nécessaire à la mémorisation des données d'un algorithme de traitement du signal est donc généralement faible, ce qui fait qu'une faible quantité de mémoire *Donnée* intégrée directement sur le circuit suffit à assurer une performance optimale. L'accès aux données se fait souvent via des mémoires double accès, plus souples d'utilisation mais plus coûteuses. Souvent, les solutions adoptées combinent les deux types de mémoires.

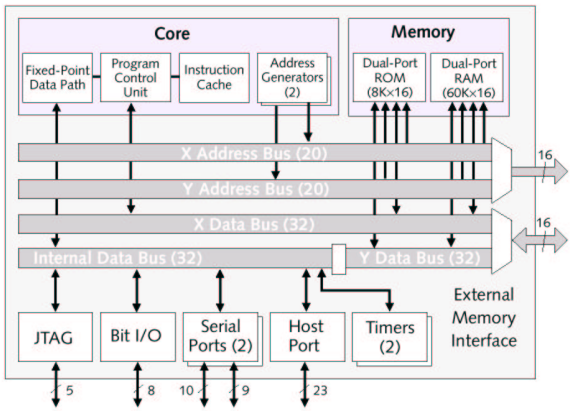
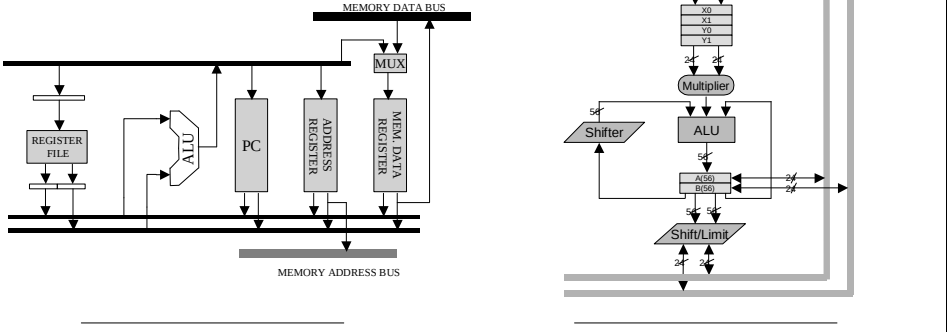


Figure 9 Architecture globale du Lucent DSP16xxx

Il est à noter que l'absence de caches permet aussi d'éviter les problèmes de non-prédictibilité du temps d'exécution dus aux mécanismes de contrôle dynamique caractéristiques des architectures superscalaires. Cependant, on verra dans la section suivante que certains des processeurs les plus récents ont adopté des mécanismes de caches afin de tenir compte de la complexité croissante des applications.

Les blocs de mémoire intégrés directement sur le circuit étant très coûteux en terme de silicium, les DSPs intègrent toujours des interfaces pour mémoires externes, qui permettent d'augmenter la capacité de mémorisation du système. L'accès aux données ou instructions en mémoire externe est évidemment beaucoup plus lent, c'est pourquoi l'implémentation efficace d'un algorithme passe par un partitionnement rigoureux des données et instructions dans les différents bancs mémoires, les parties du code et les données les plus utilisées devant autant que faire se peut se trouver sur le circuit tandis que le reste du code et les données peu utilisées peuvent être stockées en externe. Souvent, l'accès à la mémoire externe se fait via un bus externe unique, les différents bus internes étant alors multiplexés (cf. le DSP16xxx, Figure 9).




```
loop:
        mov     *r0, x0
        mov     *r1, x1
        mpy     x0, x1
        add     a, b
        mov     y0, *r2
        inc     r0
        inc     r1
        inc     r2
        dec     ctr
        tst     ctr
        jnz     loop
```

```
Do N:
   A = A + x0 * y0, x0 = *pt0++, x1 = *pt1++ ;
```

Figure 10 Filtre FIR sur une architecture de type RISC et sur un DSP.

1.2.3 Unités de calcul d'adresses

Les boucles de calcul intensives des algorithmes DSP font de très fréquents accès à la mémoire pour charger des données sources ou stocker des résultats. Le filtre FIR, par exemple, réclame à chaque calcul d'un *tap* le chargement d'un échantillon et d'un coefficient pour effectuer la multiplication-accumulation. Les pointeurs doivent ensuite être incrémentés afin de pointer sur l'échantillon et le coefficient suivant. Après le calcul du dernier *tap*, les pointeurs doivent être réinitialisés pour pointer sur le premier coefficient (c1) et le nouvel échantillon de départ avant de démarrer le calcul du « y » suivant.

Afin d'accélérer l'exécution, la plupart des processeurs DSP sont capables de calculer l'opération MAC et de modifier les pointeurs en un seul cycle machine. Cela signifie qu'ils doivent effectuer 3 opérations arithmétiques en un cycle : l'opération MAC et les deux incrémentations de pointeurs. Le matériel du chemin de données étant déjà utilisé par l'opération MAC, les calculs d'adresses sont pris en charge par des unités spécialisées : les unités de calcul d'adresse (*Address Generation Unit* ou *AGU*). Ces unités travaillent en parallèle avec le chemin de données et le délestent ainsi de toutes les opérations liées à l'adressage des données.

La Figure 11 illustre l'architecture d'une AGU utilisée dans l'*ADSP21xx* d' *Analog Devices*. Les pointeurs d'adresse sont rangés dans le banc de registres I et les opérations arithmétiques sur les pointeurs (additions et soustractions) sont calculés par l'unité *ADD.*

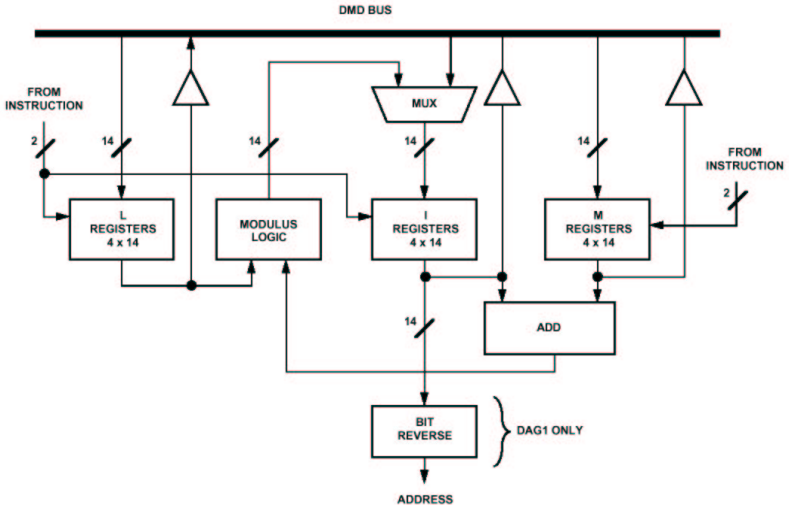


Figure 11 Unité de calcul d'adresse de l'ADSP21xx

Les AGU des DSPs gèrent les modes d'adressage classiques utilisés dans la plupart des processeurs : adressage immédiat, adressage direct, adressage indirect (par registre). La plupart proposent aussi l'adressage indirect indexé. Mais surtout, ils disposent de fonctionnalités supplémentaires spécifiques aux calculs DSP:

- l'auto incrémentation ou décrémentation des pointeurs
- l'adressage modulo
- l'adressage bit-reverse

Le code assembleur de l'exécution du filtre FIR sur DSP de la Figure 10 montre l'instruction « x0 = *pt0++ », qui représente le chargement dans le registre x0 de la valeur pointé par *ptr0*, suivi de l'incrémentation automatique de la valeur du pointeur. La plupart des algorithmes DSP travaillent sur des flux de données et accèdent aux données en série, ce qui rend cet adressage particulièrement pratique, comme dans le cas du filtre FIR où le pointeur sur les coefficients doit se déplacer d'une unité à chaque cycle (Figure 12). A la fin du calcul d'un résultat de sortie, le pointeur sur les coefficients doit être initialisé au début de la table. L'adressage modulo permet d'éviter le surcoût lié aux instructions de réinitialisation : c'est le matériel qui détecte que le pointeur est à la fin de la zone mémoire affectée aux coefficients, et l'instruction d'incrémentation suivante ramènera le pointeur l'adresse de début du tableau des coefficients.

Dans le cas de l'ADSP21xx, le fonctionnement est le suivant : un registre d'adresse I est affecté au pointeur courant, la longueur N du tableau est rangé dans un registre de longueur L, et le pas d'incrémentation est rangé dans un registre M (il est possible d'avoir des pas d'incréments différents de 1). Pour un adressage de type post-incrément, l'AGU calcule la somme de la valeur du pointeur courant avec celle de l'incrément et la compare avec l'adresse de fin du tableau (c'est le rôle du bloc modulo); dans le cas d'un dépassement, le bloc modulo renvoie l'adresse *modulo* la longueur du filtre, provoquant le renvoi du pointeur en début de tableau, et ceci sans aucun surcoût logiciel.

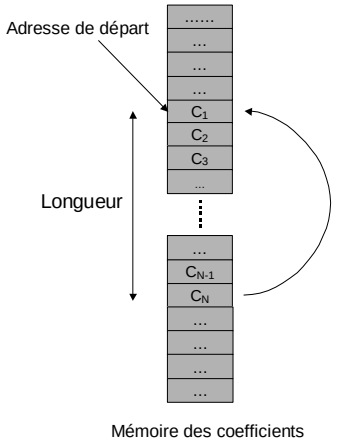


Figure 12 Principe de l'adressage modulo

L'adressage Bit Reverse est un type d'adressage particulier utilisé dans l'algorithme de la Transformée de Fourier rapide (FFT), qui manipule les échantillons dans un ordre différent de l'ordre séquentiel. Le bloc câblé *Bit Reverse* de l'ADSP21xx transforme un adressage séquentiel en un adressage de type Bit Reverse, ce qui évite le calcul logiciel des adresses. La FFT étant massivement utilisée en traitement du signal, on retrouve cette fonctionnalité dans toutes les AGUs.

1.2.4 Unité de contrôle

L'unité de contrôle des DSPs, qui contrôle le flot des instructions à exécuter, gère évidemment les modes basiques de contrôle des processeurs : exécution séquentielle, sauts et branchements conditionnels, appels de sous-programmes, interruptions. Les algorithmes DSP faisant un usage intensif de boucles logicielles (le plus souvent des boucles *for*), les processeurs DSP disposent de mécanismes matériels qui permettent d'accélérer l'exécution de ces boucles. Sur un processeur classique, une boucle logicielle est codée à l'aide d'une instruction de saut conditionnelle précédée d'une décrémentation et d'un test du compteur de boucle (cf. la boucle du filtre FIR Figure 10). Sur un processeur DSP, toutes ces opérations sont transparentes : il suffit d'indiquer le début et la fin de la boucle, d'initialiser en logiciel un compteur de boucle, et les opérations de modification de la valeur du compteur et de saut au début de la boucle seront effectuées par le matériel sans pertes de cycles machines, d'où le nom de « zero-overhead loop ». Certains processeurs n'autorisent que les boucles d'une seule instruction (*single instruction loop*), mais tous les processeurs modernes permettent l'exécution de boucles de longueur supérieure à 1 (*multiple instruction loop*). Ils autorisent aussi plusieurs niveaux de boucles imbriqués, dont le nombre est variable et limité par le nombre de registres internes mémorisant les caractéristiques des différentes boucles.

Pour réduire le nombre de cycles dus aux appels et aux retours de sous programme, les processeurs possèdent parfois une pile matérielle qui mémorise les adresses de retour des sous-programmes ou des interruptions afin de permettre des retours rapides en un cycle (pas d'accès à la pile logicielle). Certains processeurs utilisent aussi des registres cachés (« shadow registers ») utilisés pour la sauvegarde et la restitution automatiques du contexte ; cela permet d'alléger et d'accélérer les routines de traitement des interruptions qui n'ont à sauvegarder qu'une partie du contexte. Pour les routines

d'interruption critiques appelées régulièrement (comme celles générées par un périphérique d'acquisition des données de type interface série), le gain en performance peut être très important.

Enfin, les processeurs DSP intègrent souvent de la logique supplémentaire dans le circuit afin de faciliter le débogage temps réel des applications (pour certains systèmes DSP, le débogage ne peut se faire qu'en temps réel). La technique utilisée est généralement basée sur une interface JTAG permettant de lire et modifier des registres et la mémoire du circuit, fixer des points d'arrêt, le tout sans perturber le comportement dynamique de l'application exécutée sur le processeur.

1.2.5 Jeu d'instruction

Les microprocesseurs classiques utilisent des jeux d'instructions de type RISC dont les fonctionnalités sont générales et dans lequel une instruction correspond à une seule opération. A l'inverse, les processeurs DSP ont un jeu d'instruction contenant des instructions spécialisées : opérations arithmétiques complexes, modes d'adressage spécifiques, etc. De plus, ces instructions encodent plusieurs opérations en parallèle comme l'instruction du filtre FIR de la Figure 10. L'architecture des DSPs conventionnels comportant un faible nombre d'unités d'exécution (ALU, Multiplieur et les AGUs) et utilisant un jeu de registres hétérogène (la sortie d'une unité fonctionnelle n'étant parfois connectée que vers un registre unique), le nombre de bits nécessaire pour encoder une opération est inférieur à celui requis par une architecture plus générale et homogène. On parle alors d' « encodage complexe » pour traduire le fait que plusieurs opérations sont encodées dans un nombre réduit de bits. Ce type de jeu d'instruction induit une programmation très contrainte car toutes les combinaisons d'opérations élémentaires ne correspondent pas forcément à une instruction existante. On verra à la sectionque cette approche, utilisée pour l'ensemble des DSPs des premières générations, trouve ses limites dès lors qu'on cherche à exploiter d'avantage le parallélisme d'instruction (ILP) des applications.

1.2.6 Périphériques spécialisés

Exceptés les cœurs de processeurs constitués uniquement des unités de calcul et éventuellement de blocs mémoires, les processeurs DSP «sur étagère » intègrent sur le silicium des périphériques spécialisés et des interfaces Entrée/Sortie évoluées afin de diminuer le coût global du système. Parmi les périphériques les plus couramment utilisés, on trouve :

- Ports série : la plupart du temps de type synchrone, ils servent à la communication entre le processeur et des périphériques externes (par exemple des convertisseurs CAN ou CNA) ou entre processeurs.
- Ports parallèles
- Timers : utilisés pour générer des interruptions périodiques. Ils utilisent un diviseur d'horloge parfois connecté vers l'extérieur pour servir de générateur de fréquence variable.
- « Host Ports » : ports spécialisés permettant de se connecter à un microprocesseur classique ou un autre DSP. Ils sont utilisés pour l'échange de données entre processeurs ou pour contrôler le fonctionnement du DSP.

- Ports de communication : ports parallèles spéciaux destinés à la communication multiprocesseurs. Ils diffèrent des ports parallèles et des host ports de deux manières : ils ne permettent pas de contrôler les processeurs, et les processeurs connectés doivent tous être du même type.
- Ports d'entrée/sortie « bit » : de largeur 1 bit, ils sont configurables en entrée ou en sortie. Généralement utilisés pour le contrôle, ils peuvent aussi servir pour le transfert de données.
- Lignes d'interruption : servent aux périphériques extérieurs pour générer des interruptions vers le processeur (deux types : sur front ou sur niveau).
- Convertisseurs CAN et CNA : en général de largeur 16 bits, ils sont présents dans certains DSPs utilisés en traitement de parole (téléphonie mobile).
- Contrôleurs DMA : l'utilisation des canaux DMA permet de délester le processeur de la gestion des transferts mémoire de type *mémoire/mémoire* ou *mémoire/périphérique*. Le contrôleur DMA, une fois configuré en lui précisant l'adresse et la taille de la zone à transférer, prend le contrôle du bus et accède lui-même aux périphériques et à la mémoire. Selon le nombre de bancs mémoires et de bus internes, le processeur est soit gelé pendant la durée du transfert ou peut continuer à travailler normalement si la bande passante mémoire restante est suffisante. Certains contrôleurs DMA sont aussi capables de gérer plusieurs canaux DMA en parallèle.

1.3 Nouvelles architectures DSP

Jusqu'au milieu des années 90, la plupart des processeurs DSP reposaient sur l'architecture conventionnelle présentée précédemment : des unités fonctionnelles spécialisées, un jeu d'instruction complexe, une instruction par cycle. Les progrès en terme de performance de ces processeurs doivent en fait plus à l'évolution des procédés de fabrication qu'à de réels bouleversements en matière d'architecture. Aujourd'hui, l'offre tend à se diversifier avec de nouvelles architectures empruntant aux microprocesseurs généraux : VLIW, superscalaire, SIMD. La raison de ce bouleversement est que les DSPs conventionnels ont atteint leurs limites et ne peuvent satisfaire les demandes des applications modernes de plus en plus exigeantes en terme de puissance de calcul. En effet, la complexité des applications DSP continue de croître rapidement et le rôle des compilateurs, très souvent négligé des concepteurs et des utilisateurs de processeurs DSP, devient indispensable pour maintenir un temps de développement raisonnable. Dans le même temps, la principale contrainte pour les concepteurs de processeurs se situe maintenant au niveau de la consommation électrique. La simple augmentation de fréquence du processeur pour améliorer la performance (principal facteur de progression de la performance des DSPs ces dernières années) conduit à une surconsommation qui n'est plus tolérable du point de vue des systèmes embarqués.

A l'heure actuelle, la tendance favorise donc la réduction des fréquences de fonctionnement au profit de plus de parallélisme matériel permettant de diminuer la consommation tout en maintenant le niveau de performance. La surface de silicium est une préoccupation qui passe désormais en second plan par rapport à la consommation.

Texas Instruments a été le premier constructeur à introduire une architecture VLIW dans un processeur DSP avec le TMS320C6x, traduisant une évolution vers l'utilisation de techniques architecturales plus évoluées que la simple architecture basée sur la structure des filtres numériques. Certaines techniques comme le superscalaire sont largement connues et utilisées dans le domaine des microprocesseurs généraux où elles ont fait la preuve de leur efficacité. A l'inverse, les architectures VLIW et SIMD, bien qu'anciennes du point de vue théorique, ont connu moins de succès mais semblent prometteuses pour l'intégration dans des systèmes de traitement du signal. L'idée fondatrice de ces techniques consiste à augmenter la performance en tirant parti du parallélisme d'instructions et de données présent dans les applications ; en effet, les caractéristiques des architectures DSP conventionnels (une instruction par cycle, faible nombre d'unités fonctionnelles, jeu d'instruction complexe) sont inadaptées pour l'exploitation optimale du parallélisme.

La prochaine section examine les différents degrés de parallélisme potentiellement présents dans une application. Les sections suivantes présentent en détail les nouvelles architectures DSP au travers de leur implémentation dans différents processeurs. L'évolution des caractéristiques des processeurs DSP, des conventionnels aux plus récents, est décrite en détail dans [6], [7] et [8].

1.3.1 Le parallélisme dans les applications DSP

Bien que destinées à exploiter plus de parallélisme en général au sein d'une application, les différentes techniques architecturales des derniers DSPs ne visent pas le même type de parallélisme. On peut considérer le parallélisme global d'une application comme la somme de deux composantes :

- le parallélisme de taches : englobe le parallélisme au niveau instruction (ILP ou *Instruction-Level Parallelism*) et au niveau processus (*multi-threading*).
- le parallélisme de données : traduit la tendance de l'application à appliquer les mêmes traitements à plusieurs données différentes.

Chaque type de parallélisme est exploitable à différents niveaux : instruction, circuit et système. L'accélération d'une application sur un processeur consiste à tirer parti du parallélisme global en augmentant la capacité de traitement du processeur pour un ou plusieurs types de parallélisme.

Niveau de parallélisme	**Tâche**	**Donnée**
Instruction	VLIW / Superscalaire	« Split-ALU »
Circuit	Chemins de données parallèles avec contrôle MIMD	Chemin de données parallèles avec contrôle SIMD
Système	Système multiprocesseur avec contrôle décentralisé	Système multiprocesseur avec contrôle centralisé

Figure 13 Les différents types de parallélisme et les architectures associées

La Figure 13 classe les différentes architectures matérielles selon le type de parallélisme exploité. On voit que les nouvelles techniques employées dans les DSP (VLIW, superscalaire et SIMD) ne recouvrent qu'une partie des différents parallélisme exploitables. L'architecture idéale est évidemment celle qui les exploiterait tous. Certaines solutions sont à l'étude, mais le coût et la complexité de telles architectures restent très élevés et ne se justifient pas forcément, car une application donnée réclame rarement l'exploitation optimale de l'ensemble des types de parallélisme.

1.3.1.1 Le parallélisme de taches

L'exploitation du parallélisme de taches (multithread) n'est pour l'instant pas ou très peu prise en compte dans les DSPs. Historiquement, les algorithmes DSP classiques sont codés en langage de haut niveau comme le langage C. La nature purement séquentielle de ce type de description ne donne aucune indication concernant le parallélisme disponible dans l'application. Pour exécuter l'application sur un système multi-tâches, il est donc nécessaire de réécrire l'application sous forme d'un ensemble de taches indépendantes pouvant s'exécuter simultanément. Malheureusement, l'extraction du parallélisme de taches d'une application décrite de manière séquentielle est un processus très complexe, pour lequel il n'existe aucune méthode automatique donnant des résultats satisfaisants. L'extraction « manuelle » est actuellement la seule solution efficace, sous réserves que la nature de l'application elle-même puisse offrir de tels niveaux de parallélisme, ce qui n'est pas forcément évident pour toutes les applications DSP. En fait, le meilleur moyen pour permettre l'emploi efficace du multithread est que dès le départ les applications soient conçues et décrites sous forme parallèle.

L'accroissement continuel de la complexité des applications a poussé les programmeurs d'applications DSP à utiliser des langages à plus haut niveau d'abstraction comme le C++. Les notions d' « orienté-objet » et de « multi-tâches » commencent à faire leur apparition dans les applications complexes. Ainsi, la norme de compression multimédia MPEG-4 modélise les flux d'images comme un ensemble d'objets indépendants possédant leurs caractéristiques et leur déplacements propres. Par exemple, deux personnages dans une scène seront modélisés par deux objets différents. La composition d'une séquence d'images animées revient alors à calculer les caractéristiques et les déplacements des divers objets consistant la scène et à les superposer pour former l'image finale. Dans ce cas, chaque traitement d'un objet particulier peut être géré par un processus indépendant, et l'animation de l'ensemble devient ainsi une application multi-tâches.

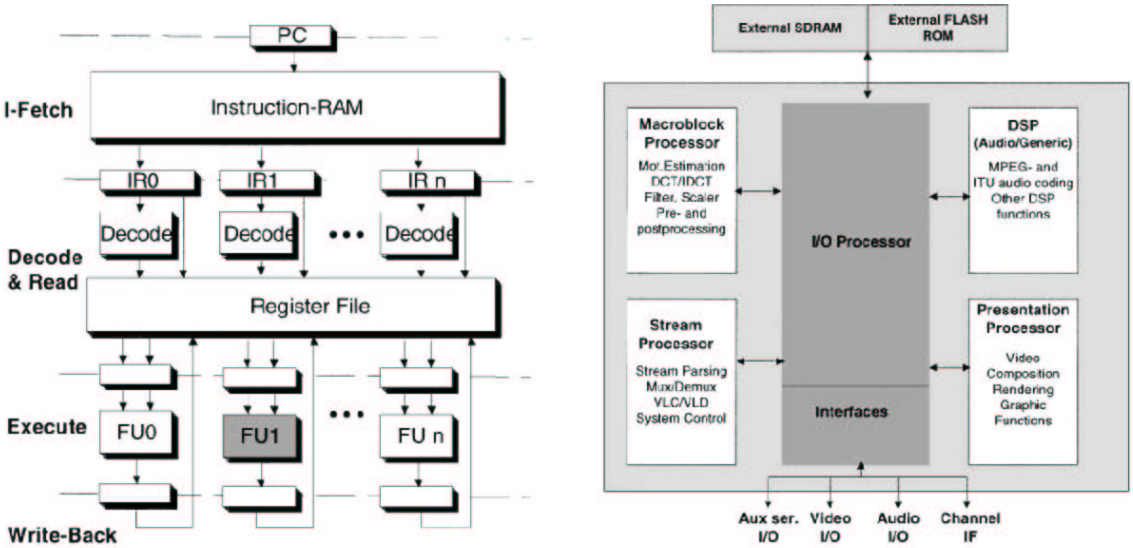


Figure 14 Architectures pour MPEG-4:processeur MIMD et système multiprocesseur Bosch M-Pire

Une architecture de type MIMD (*multiple instruction multiple data*) spécialisé pour l'implémentation d'un décodeur MPEG-4 est présenté en [9] (cf. *Figure 14*). Ce type d'architecture qui traite plusieurs flux d'instructions distincts en parallèle est encore très peu répandu dans les DSPs généraux. Pour

gérer le multi-tâches, les processeurs les plus récents se contentent d'inclure des dispositifs matériels permettant une meilleure gestion de la pile logicielle et la sauvegarde automatique de certains registres (afin de diminuer le temps nécessaire aux changements de contexte). Pour l'implémentation d'applications multi-tâches complexes, une solution plus répandue consiste à utiliser plusieurs processeurs indépendants et spécialisés sur une structure de type *SoC* (processeur M-Pire Figure 14). Dans ce cas de figure, toute la difficulté consiste à répartir les taches sur les différents processeurs et à définir leurs architectures respectives.

L'étude menée en [10] sur un décodeur vidéo MPEG-2 a montré que lorsque l'application s'y prête, le gain obtenu par l'exploitation du parallélisme de taches peut être exceptionnel : dans le cas d'une implémentation du décodeur MPEG-2 sur une machine multiprocesseur, le facteur d'accélération obtenu est quasi-linéaire par rapport au nombre de processeurs de la machine, ce qui traduit une utilisation optimale du matériel. Ce cas est cependant loin d'être représentatif de la majorité des applications DSP.

A l'heure actuelle, compte-tenu du coût des architectures MIMD ou multiprocesseurs (en terme de surface matériel et de consommation), de la faiblesse des outils d'aide à l'extraction de parallélisme et de la nature des algorithmes DSP classiques, les architectes privilégient plutôt l'usage des deux autres types de parallélisme : le parallélisme d'instruction et le parallélisme de données.

1.3.1.2 Le parallélisme d'instruction

C'est sur ce type de parallélisme que repose le principe des architectures VLIW et superscalaires. Alors que les processeurs RISC exécutent les instructions une à une de manière séquentielle, les processeurs VLIW (ou superscalaires) exécutent ces mêmes instructions élémentaires en parallèle, sous réserve qu'il n'existe pas de contraintes entre les différentes instructions. Le principe consiste à augmenter le nombre d'unités d'exécution afin de pouvoir exécuter une instruction sur chacune d'entre elles, augmentant ainsi l'IPC (nombre *d'Instructions par cycle*) du processeur. Idéalement, un processeur possédant 4 unités d'exécution devrait pouvoir consommer quatre instructions par cycle, soit un IPC de 4. Dans la réalité, deux facteurs principaux limitent la performance de ces architectures : le parallélisme intrinsèque de l'application, et les ressources matérielles du processeur.

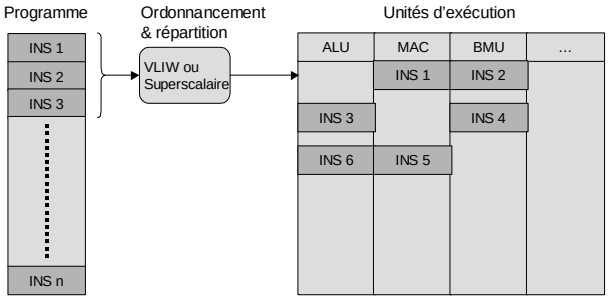


Figure 15 Exploitation du parallélisme d'instruction

On entend par parallélisme intrinsèque d'une application le degré de parallélisme maximal exploitable par une architecture parallèle idéale dont les ressources matérielles seraient infinies. A l'origine, les premières architectures exploitant l'ILP étaient destinées aux microprocesseurs grand public exécutant un large spectre d'applications de nature parfois radicalement différentes du point de vue de

l'ILP. Les nombreuses études sur l'ILP montrent que les applications les plus couramment utilisés sur stations de travail ou sur PC (comme celles formant le benchmark *SPECint*) offrent moins de parallélisme que les applications multimédia qui utilisent essentiellement des algorithmes de traitement du signal [11] (voir Figure 16, les quatre programmes Unix sont les plus à droite). Ceci est du à la structure des algorithmes DSP souvent constitués de quelques boucles *for* répétant la même suite d'instructions (généralement longue de quelques dizaines d'instructions, voir [11]). sur plusieurs données différentes.

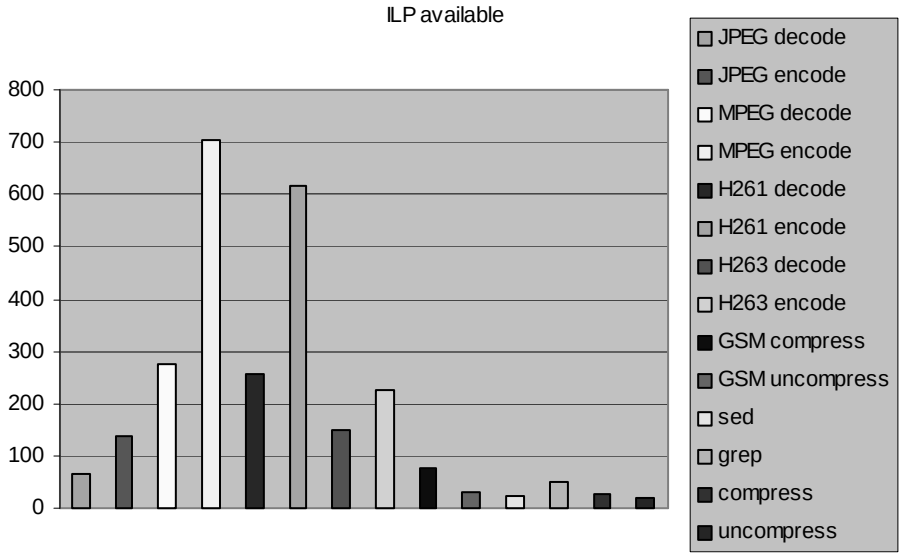


Figure 16 Parallélisme intrinsèque pour différents programmes Unix et Multimédia

Le parallélisme intrinsèque est limité par deux types de contraintes issues de la structure du CDFG (*control-data flow graph*) de l'application (Figure 17). Une instruction produisant un résultat doit obligatoirement s'exécuter *avant* l'instruction qui consommera ce résultat. On parle de *dépendance de données* entre les deux instructions : c'est le type de contrainte la plus forte et qui ne peut être supprimée qu'en modifiant profondément la structure de l'algorithme.

Les *dépendances de contrôle* interviennent entre des instructions n'appartenant pas au même *Basic Block*. Elles traduisent une rupture dans le déroulement séquentiel du programme, due à la présence dans le programme source d'une structure de contrôle évoluée (*if-then-else*, boucles *for* et *while*, appel de fonction, etc.). A priori, ce type de dépendance limite donc le parallélisme exploitable aux instructions appartenant au même *Basic Block*, puisqu'on ne sait pas à priori comment se fera l'enchaînement des instructions entre les *Basic Blocks*. On verra par la suite que certaines techniques matérielles ou logicielles permettent d'exploiter le parallélisme d'instructions au niveau global (entre Basic Blocks) et donc d'accroître les gains en performance. Les dépendances de contrôle sont généralement beaucoup moins nombreuses dans les algorithmes DSP que dans les programmes pour stations de travail où figurent beaucoup de structures de contrôles complexes. C'est ce qui explique pourquoi l'ILP intrinsèque est plus grand dans les algorithmes DSP.

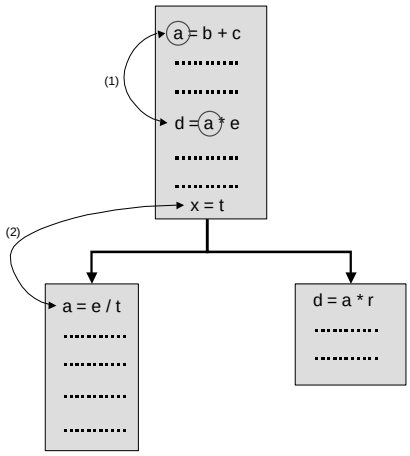


Figure 17 Dépendances de données (1) et de contrôle (2) dans un CDFG

Bien qu'offrant potentiellement moins de perspective de gains que pour les applications DSP, les architectures destinées à l'exploitation du parallélisme d'instructions ont été d'abord développées pour les microprocesseurs généraux exécutant du code « orienté-contrôle ». Les gains en performance sont pourtant impressionnants, c'est pourquoi l'usage d'architectures parallèles pour le traitement du signal semble particulièrement approprié.

Les problèmes de ressources matérielles surviennent lorsque plusieurs instructions désirant s'exécuter simultanément veulent accéder à la même ressource (unité d'exécution, registre, emplacement mémoire). Ces contraintes sont évidemment diminuées en surdimensionnement l'architecture : duplication des unités fonctionnelles, augmentation du nombre de registres et de la bande passante mémoire. Le matériel requis est cependant très dépendant de l'application : une application tirera grand parti de deux ALU et de 16 registres, tandis qu'une autre se contentera de 8 registres mais nécessitera deux multiplieurs pour une accélération maximale. Il n'existe pas de configuration universelle, c'est pourquoi les processeurs à architecture générale ne peuvent rivaliser avec les processeurs dédiés à une classe d'application.

1.3.1.3 Le parallélisme de données

Le parallélisme de données est la forme de parallélisme la moins utilisée dans les architectures actuelles, qui se focalisent principalement sur l'ILP. Les études montrent pourtant qu'il offre pour les applications multimédia des gains potentiels largement supérieurs à ceux basés sur la simple exploitation de l'ILP [12].

```
for (m=0; m<8; m++)
  for(n=0; n<8; n++)
  {
          y[m][n]=0.0;

          for(i=0; i<8; i++)
            for(j=0; j<8; j++)
                    y[m][n] = y[m][n] + x[i][j] * c[i][m] * c[j][n];
  }
```

Figure 18 Code C de l'algorithme DCT-2D

Le parallélisme de données est le parallélisme existant entre des données d'un programme ayant peu ou aucune dépendance entre elles. Il peut à priori exister entre n'importe quelles données, mais on le trouve plus régulièrement entre des données « éloignées » dans le domaine spatial (applications 2D comme le traitement d'images) ou temporel (application 1D ou « flux » comme les traitements audio »). L'indépendance de ces données permet de leur appliquer des traitements en parallèle.

Les traitements effectués dans la plupart des applications DSP sont réguliers et se codent souvent à l'aide de boucles, comme l'algorithme de la DCT-2D de la Figure 18 utilisé pour la compression JPEG et MPEG. En conséquence, le parallélisme de données dans de telles applications revient souvent à un parallélisme de niveau boucle. L'examen de la fonction montre que l'instruction « y[m][n] = y[m][n] + x[i][j] * c[i][m] * c[j][n] » crée une dépendance de boucles pour le calcul des deux boucles les plus profondes. Par contre, il n'y a aucune dépendance pour les deux boucles de niveau supérieur, on peut donc calculer simultanément les 64 itérations calculant les « y[m][n] ». On voit à travers cet exemple que la granularité du parallélisme de données, agissant au niveau boucle (donc ici sur une longueur de 64 instructions), se situe entre l'ILP (1 instruction) et le parallélisme de taches (plusieurs centaines ou milliers d'instructions).

L'étude menée par [12] sur plusieurs noyaux de calculs de traitement vidéo montre que le parallélisme de données domine largement l'ILP en terme de gain en performance pour ce type d'application. L'avantage est que ces deux types de parallélisme peuvent être exploités simultanément. Des solutions architecturales implémentant les deux types de parallélisme ont donc un grand intérêt.

1.3.2 Les DSPs conventionnels « étendus »

Une première solution pour augmenter les performances des processeurs DSPs a consisté à améliorer l'architecture conventionnelle pour faire en sorte qu'elle puisse exploiter d'avantage de parallélisme. Les densités d'intégration croissantes, offrant toujours plus de transistors, ont permis aux concepteurs de rajouter du matériel dans leur architecture :

- ajout d'unités fonctionnelles
- bande passante mémoire plus large
- capacités SIMD limitées ou étendues
- chemin de données spécialisé pour un domaine d'application particulier
- coprocesseurs matériels

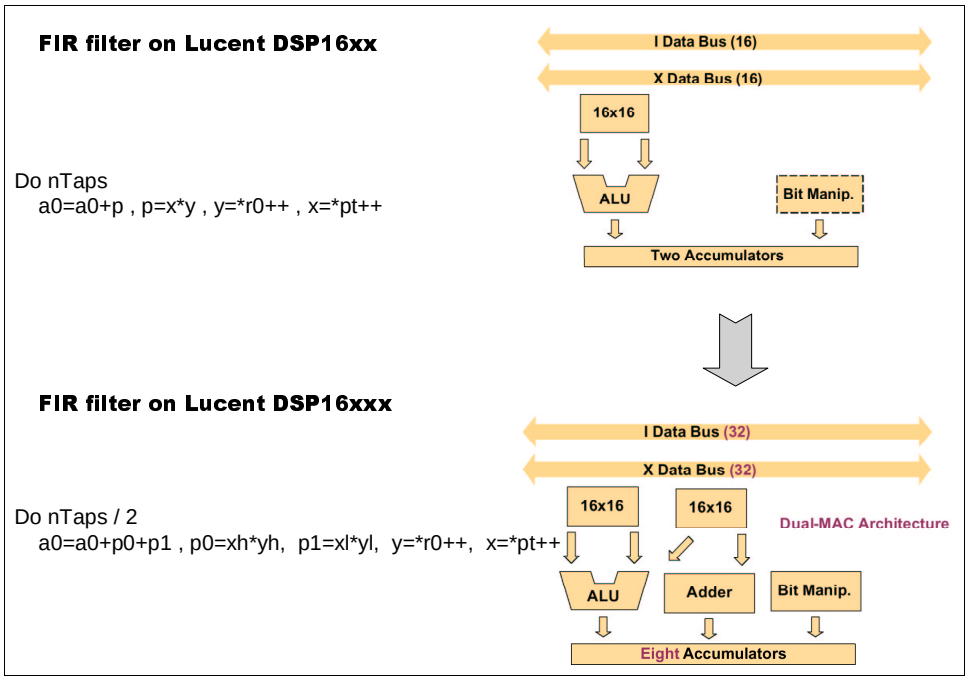


Figure 19 Evolution d'une architecture conventionnelle: du 16xx au 16xxx (source BDTI)

Le processeur Lucent DSP16xxx [13], successeur du 16xx, est un bon exemple d'extension d'une architecture classique. Le matériel intégré au chemin de données a été augmenté : ajout d'une ALU et d'un multiplieur, doublement de la largeur des bus mémoire, augmentation du nombre de registres (Figure 19). Le processeur visant les applications Telecom type GSM, le chemin de données a été spécialisé de différentes manières :

- intégration de décaleurs à capacité limitée et de saturateurs à la sortie des unités fonctionnelles pour gérer automatiquement les opérations de mise à l'échelle (Figure 5 p.23).
- ALU spécialisée capable d'opérations SIMD 16/32 bits et de l'opération ACS (Add / Compare / Select) utilisée pour le décodage de Viterbi.
- coprocesseur « Trace back encoder » pour l'accélération du décodage de Viterbi.

Ce choix de l'extension d'une architecture et d'un jeu d'instruction déjà existant permet d'augmenter la performance de manière significative (cf. Figure 20(c)) tout en conservant la plupart des atouts des DSP conventionnels : faible consommation, faible coût matériel et bonne densité de code. En effet, la spécialisation du chemin de données (jeu de registres hétérogène, topologie non régulière, unités spécialisées) tend à réduire le matériel et la consommation au minimum, au détriment de la généricité de l'architecture. Le jeu d'instruction encode encore plus d'opérations élémentaires par instruction sur un nombre de bits restreint (en général 16, voir Figure 21), donnant ainsi d'excellents résultats en terme de densité de code. Et surtout, ce code reste compatible avec celui des processeurs d'anciennes générations, évitant ainsi aux utilisateurs d'avoir à reprogrammer une application ayant déjà fait l'objet d'un portage sur DSP. La coutume voulant que la plupart des applications DSP soient programmées manuellement en assembleur pour des raisons de performance, la possibilité de réutiliser une partie du code (et éventuellement de l'optimiser sur la nouvelle architecture) et le fait que l'environnement de programmation et le jeu d'instructions soient très ressemblants permet de réduire considérablement le temps de développement.

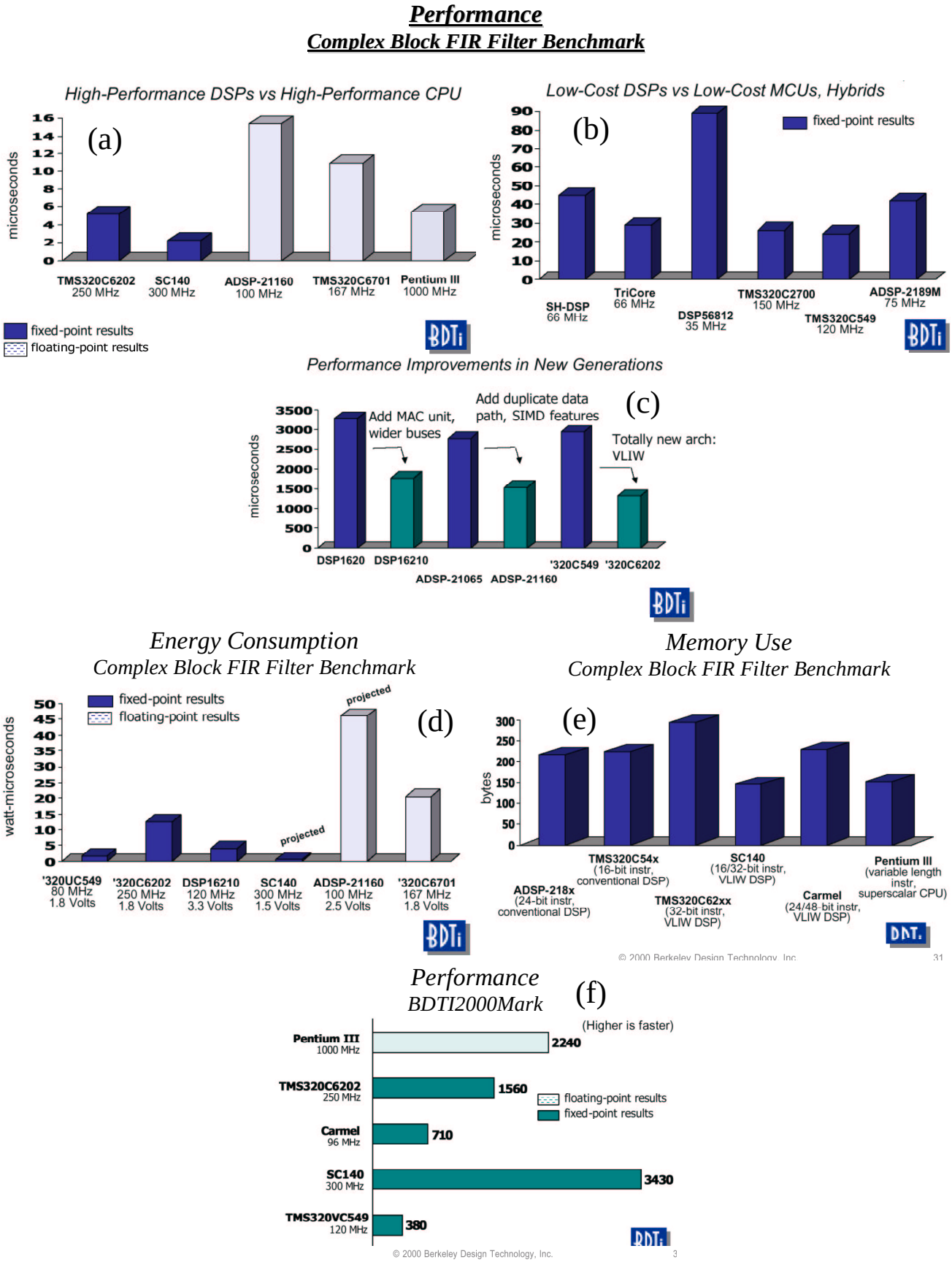


Figure 20 Benchmark BDTI [14]

Operation (execution unit)	Combinations supported in a single instruction				
Add/Subtract (ALU)	●				
Round (ALU)		●			
Absolute value (ALU)			●		
Minimum (ALU)					●
Compare (ALU)				●	
Add/Subtract (Adder)	●			●	
2 Multiplications (Mul)	●				
Shift (BMU)		●	●		
Exponent (BMU)					●
2 Data transfers	●	●	●	●	●

Figure 21 Combinaisons d'opérations supportées en une instruction pour le DSP16xxx (source [13])

Les inconvénients de l'architecture étendue sont les mêmes que ceux de l'architecture classique, voire accentués : la complexité accrue du jeu d'instruction rend encore plus difficile le travail des compilateurs, tandis que la programmation manuelle en assembleur, déjà complexe, devient encore moins évidente à cause de la gestion du parallélisme et des nombreux bits de contrôle configurant le chemin de données (cf. section 1.3.3.2).

Enfin, les perspectives d'évolution de ce type d'architecture pour exploiter encore plus de parallélisme sont très limitées ; l'ajout de deux multiplieurs supplémentaires dans une architecture de type DSP16xxx aboutirait à une architecture et un jeu d'instructions abominablement complexes, donc inexploitables.

Le processeur ADSP-2116x d'Analog Devices, et le PalmDSPCore de DSP Group constituent d'autres exemples de DSPs étendus [15].

1.3.3 Les DSPs VLIW et superscalaires

L'arrivée en 1997 du TMS320C62 de Texas Instruments basée sur une architecture VLIW a constitué une petite révolution dans le monde des processeurs DSPs [16]; c'était en effet la première fois qu'un DSP s'affranchissait de l'architecture et du jeu d'instruction classiques des processeurs conventionnels pour s'aventurer sur une nouvelle voie. Les raisons qui ont poussé à l'abandon de l'architecture conventionnelle s'expliquent par la volonté des concepteurs d'aller plus loin dans l'exploitation du parallélisme d'instructions, mais aussi d'offrir une architecture plus souple, moins complexe et qui soit plus facilement exploitable par un compilateur.

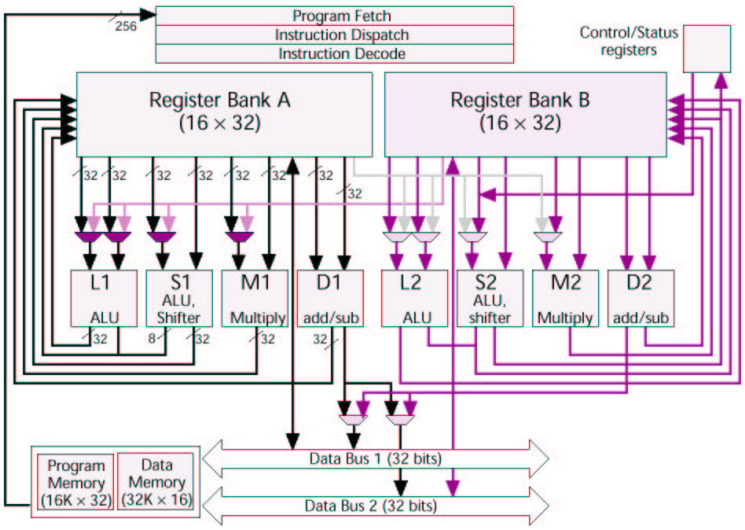


Figure 22 Un exemple d'architecture VLIW: le TMS320C62x

1.3.3.1 Architecture

Le concept du VLIW repose sur une architecture régulière et homogène, constituée d'unités fonctionnelles disposées en parallèle et de bancs de registres multi-ports pour le stockage des calculs temporaires. Le TMS320C62, présenté Figure 22, est composé de huit unités fonctionnelles (4 ALUs, 2 Multiplieurs, 2 additionneurs-soustracteurs chargés principalement des calculs d'adresses) distribuées en deux « clusters » de 4 unités disposant chacun d'un banc de registres multi-ports. La bande passante vers la mémoire données est importante (2 * 32 bits par cycle) de manière à nourrir efficacement les nombreuses unités fonctionnelles.

De nombreuses variantes d'architecture existent, en particulier dans le choix et le nombre des unités fonctionnelles, la manière de structurer le chemin de données en un ou plusieurs clusters, le nombre de bancs de registres, et la bande passante mémoire (Figure 23). Dans tous les cas, la philosophie reste la même :

- chemin de données régulier composé d'unités fonctionnelles travaillant en parallèle
- jeu de registres homogène composé de bancs de registres multi-ports
- bande passante mémoire importante

Contrairement aux architectures conventionnelles, on ne retrouve pas dans les processeurs VLIW de registres spéciaux associés à un seul opérateur ou d'enchaînement « vertical » de certaines unités fonctionnelles. En effet, ces caractéristiques étaient issues de la spécialisation des architectures pour des domaines d'application bien définis, alors que les architectures VLIW sont par nature beaucoup plus générales.

Le chemin de données des processeurs superscalaires repose sur les mêmes principes que ceux des VLIWs. C'est surtout au niveau du jeu d'instruction que ces processeurs diffèrent.

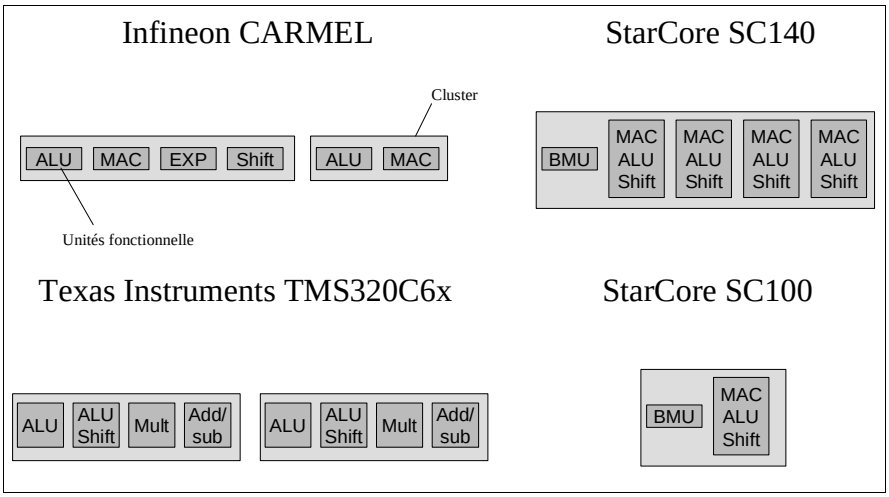


Figure 23 Architectures de chemins de données VLIW

1.3.3.2 Jeu d'instruction

On a vu précédemment que les jeux d'instruction conventionnels encodent de nombreuses opérations élémentaires (équivalent d'une instruction d'un processeur RISC) en une instruction de largeur réduite afin d'offrir des instructions à la fois performantes et prenant peu de place en mémoire. Le faible nombre de bits nécessaires pour coder une instruction est due à une stratégie d'encodage complexe n'autorisant que certaines combinaisons d'opérations élémentaires, les combinaisons élues étant choisies d'après l'étude des algorithmes DSP les plus classiques (souvent les mêmes algorithmes formant les benchmarks pour DSPs comme celui du BDTI). Dès lors qu'une combinaison d'opérations « sort de l'ordinaire », elle risque de ne pas trouver d'équivalent dans le jeu d'instructions et devra donc être réalisée séquentiellement à l'aide d'instructions plus simples mais moins efficaces. Le DSP16xx par exemple n'autorise pas l'exécution simultanée d'un calcul d'exposant et de deux multiplications (cf. Figure 21) car cette instruction n'est pas ou peu utilisée dans la plupart des algorithmes DSP classiques.

C'est le décodeur d'instruction (Figure 24) qui extrait du mot d'instruction les différentes commandes pour les unités fonctionnelles. A cause des contraintes d'encodage, il est rare que l'ensemble des unités puissent être sollicitées par une instruction unique.

La stratégie des processeurs VLIW comme superscalaire repose au contraire sur un ensemble d'instructions élémentaires beaucoup plus simples équivalent à un jeu d'instruction de type RISC [17]. Pour augmenter la performance, ces processeurs autorisent l'exécution de *plusieurs* de ces instructions en parallèle dans le même cycle machine. L'ensemble des combinaisons d'instructions élémentaires autorisées est beaucoup plus vaste que dans le cas des jeux d'instructions fortement encodé des processeurs conventionnels.

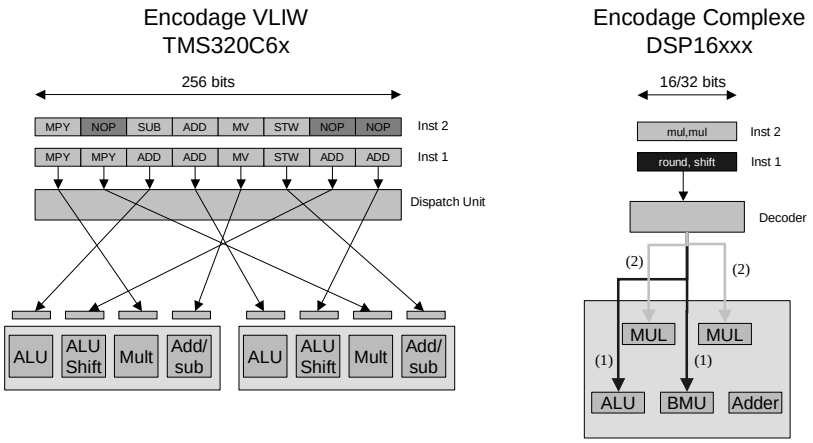


Figure 24 Encodage des instructions

L'encodage des instructions dans un processeur VLIW est simple : les instructions élémentaires, qui correspondent chacune à l'activation d'une des unités fonctionnelles du processeur, sont concaténées pour former une super-instruction qui commandera l'ensemble du chemin de données (Figure 24). Cette super-instruction est alors traitée par l'unité de répartition (*Dispatch unit*) qui isole les champs associés aux unités et propage les commandes vers ces unités. La largeur importante du mot d'instruction permet d'activer l'ensemble des unités fonctionnelles simultanément afin de profiter du maximum de parallélisme.

1.3.3.3 Différences entre VLIW et superscalaire

La principale différence entre processeurs superscalaire et VLIW se situe au niveau de la représentation et de la gestion du parallélisme d'instructions. Dans une architecture VLIW, le parallélisme est analysé statiquement avant exécution par le programmeur ou le compilateur puis traduit sous forme d'un enchaînement de super-instructions rangées dans l'ordre en mémoire programme. On notera au passage qu'à cause de la taille des super-instructions la largeur de la mémoire programme et des bus associés est beaucoup plus grande que pour les processeurs conventionnels (256 bits pour le TMS320C6x). Lors de l'exécution d'un programme, le processeur VLIW charge les super-instructions une à une et les exécutent dans l'ordre de chargement à l'instar d'un processeur RISC.

Dans les processeurs superscalaires, le parallélisme est analysé dynamiquement par le processeur lui-même. Le programme est stocké en mémoire sous forme d'une séquence d'instructions élémentaires de type RISC (séquence générée par le programmeur ou le compilateur). Dans cette représentation, il n'existe encore aucune expression réelle de l'ILP.

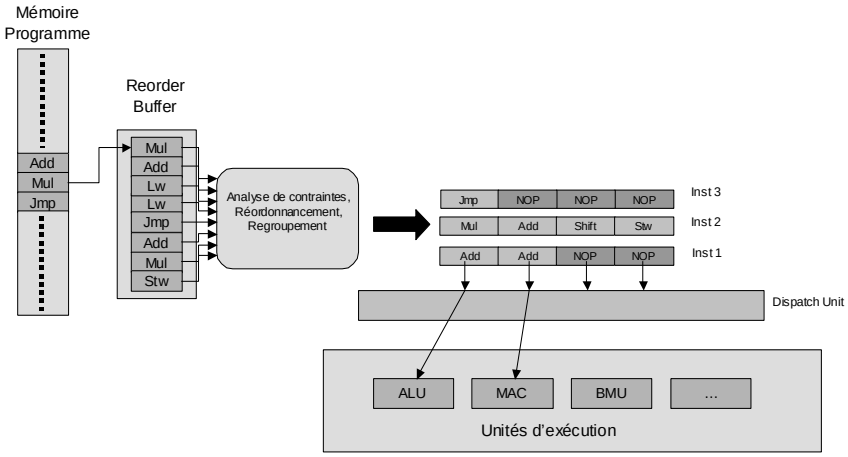


Figure 25 Exécution sur un processeur superscalaire

Lors de l'exécution, les instructions sont chargées par le processeur dans un buffer d'instructions. Des mécanismes matériels complexes procèdent à une analyse des contraintes entre les différentes instructions de manière à déterminer lesquelles peuvent être exécutées en parallèle. Ces contraintes tiennent compte à la fois des dépendances de données et de contrôle entre les différentes instructions mais aussi de l'occupation des ressources du processeur. Dans cette phase, les instructions peuvent être réordonnées pour minimiser les contraintes et offrir ainsi plus de parallélisme exploitable. Un certain nombre d'instructions est alors choisi, regroupés sous forme d'un paquet d'exécution et exécuté à la manière d'une instruction VLIW.

1.3.3.4 Intérêts et limitations

Le gain en performance des architectures exploitant le parallélisme d'instructions dépend bien évidemment du degré de parallélisme de l'application visée. La Figure 26 compare les performances d'un processeur VLIW et d'un DSP conventionnel pour trois algorithmes différents. A fréquence égale, le facteur d'accélération n'est significatif que pour la FFT et le décodage Viterbi. Le filtre IIR, basé sur une structure de calcul récursive (le résultat y(n) dépend de y(n-1)), présente trop de dépendances de données entre les calculs, interdisant une implémentation efficace sur une architecture parallèle. A l'inverse, la FFT est un algorithme facilement parallélisable qui tire un profit maximum de l'architecture VLIW.

Benchmark	Speed ratio 'C6202 (250 MHz) : 'C549 (120 MHz)
IIR Filter	2.2 : 1
256-point FFT	8.7 : 1
Viterbi decoder	3.7 : 1

Figure 26 Comparatif de performance VLIW / conventionnel (source BDTI)

Cependant, l'étude de performance sur un benchmark représentatif d'algorithmes montre qu'en moyenne l'architecture VLIW accélère de façon significative la plupart des noyaux de calcul DSP (Figure 20 (f)). Autre avantage non négligeable, l'efficacité des compilateurs et de manière générale la facilité de programmation sur ce type d'architecture est nettement meilleure, ce qui laisse entrevoir

dans un avenir proche des stratégies de développement logiciel basées presque exclusivement sur du code compilé, accélérant ainsi considérablement le temps de développement.

Il est plus difficile de se faire une idée sur l'efficacité des processeurs superscalaires. A l'heure actuelle, il n'existe qu'un DSP de ce type : le LSI401Z de ZSP Corporation [18]. Sa performance sur le BDTIMark est à fréquence égale légèrement inférieure à celle du TMS320C60, mais largement supérieure à celle des processeurs conventionnels, ce qui laisserait présager que le superscalaire peut être aussi une solution viable pour les calculs DSP. Cependant, la plupart des constructeurs semblent pour l'instant privilégier le VLIW, sans doute à cause des problèmes liés aux principes mêmes du superscalaire : l'exécution dans le désordre rend le comportement dynamique difficilement prédictible, ce qui est problématique pour l'implémentation d'applications soumises à de fortes contraintes temps réel. L'optimisation de code est encore plus délicate que pour les VLIW puisqu'on n'est pas sûr de l'ordre dans lequel les instructions vont s'exécuter. Cependant, les problèmes d'analyse de dépendance et d'ordonnancement étant réalisées par le matériel, les outils de génération de code requièrent moins de complexité que pour le VLIW. Enfin, le superscalaire permet la compatibilité binaire du code entre deux générations de processeurs, avantage capital dans le monde des microprocesseurs généraux et de plus en plus recherché dans le monde de l'embarqué (à cause de l'accroissement continu de la complexité des applications).

Les deux solutions souffrent des mêmes désavantages par rapport aux processeurs conventionnels : un coût matériel et une consommation électrique supérieures (Figure 20 (d)). La régularité et la largeur de leur chemin de données, responsables de leur puissance et de leur souplesse d'utilisation, se payent en terme de complexité matérielle et de puissance dissipée: unités fonctionnelles plus nombreuses, bancs de registres multi-ports (très coûteux en terme de surface, de performance et de consommation), bus plus larges ou plus nombreux, etc.

Avec l'accroissement des densités d'intégration, la part de la mémoire embarquée dans le coût d'un System-on-chip continue d'augmenter, rendant le coût du cœur de processeur en surface de silicium presque négligeable par rapport à la mémoire. La densité de code devient ainsi un paramètre de plus en plus critique puisqu'il permet de réduire la quantité de mémoire « programme » à intégrer dans le circuit.

La structure des jeux d'instructions classiques VLIW consistant à concaténer les champs des différentes unités en une super-instruction très large (Figure 24) ne permet pas d'obtenir une densité de code aussi bonne que celles des jeux d'instruction conventionnels. Par exemple, le TMS320C6x qui utilise cette technique fait un moins bon usage de la mémoire qu'un DSP conventionnel comme le C54x (cf. Figure 20). On peut l'expliquer de plusieurs manières.

Tout d'abord, les caractéristiques du chemin de données (petit nombre de registres et d'unités fonctionnelles, topologie irrégulière du chemin de données) et du jeu d'instruction (opérandes implicites, comportement dépendant de nombreux bits de contrôle, nombre d'opérations par instruction limité) des DSP conventionnels offrent une densité de codage potentielle plus grande (moins de bits par instruction pour coder le même ensemble d'opérations). La structure classique du jeu d'instruction VLIW est trop générale (car permettant d'encoder plus de combinaisons d'opérations) pour offrir de bons résultats en terme de densité.

A cela s'ajoute le problème de la sous-utilisation des unités fonctionnelles : l'ILP d 'une application étant limité, il arrive très fréquemment que l'application n'offre pas suffisamment de parallélisme pour faire fonctionner toutes les unités fonctionnelles. Dans ce cas, les champs commandant les unités « passives » contiendront la valeur « NOP » (pour *No Operation*), ce qui aura pour effet de diminuer encore l'efficacité de codage.

Enfin, l'utilisation efficace du parallélisme oblige parfois à recourir à des méthodes d'optimisations logicielles (déroulement de boucles, pipeline logiciel) qui consistent à dupliquer certaines instructions dans le code afin d'offrir plus de parallélisme exploitable [19]. Ces techniques, utilisées pour l'amélioration des performances, ont un impact négatif sur la taille du code.

Le code VLIW dans sa forme basique est donc a priori peu compact et gourmand en espace mémoire. Mais parce que les architectures VLIW offrent des possibilités dont il serait dommage de se priver, les derniers processeurs en date proposent des techniques d'encodage plus évoluées qui permettent d'améliorer de façon spectaculaire la densité du code. Des techniques comme le VLES du Motorola SC140 ou le CLIW du Infineon Carmel donnent des résultats équivalents voire meilleurs que ceux obtenus avec l'encodage complexe des jeux d'instructions conventionnels (Figure 20). La Figure 27 résume les caractéristiques des principaux processeurs VLIW disponibles sur le marché.

Processor	Issue width	Data memory bandwidth (16-bit words)	Instruction Size	Clock (MHz)	Pipeline Depth	Notable characteristics
TMS320C62xx	8	4 words/cycle	32 bits	300	11	First VLIW-based DSP processor
SC140	6	8 words/cycle	16 bits w/ 16-bit prefixes	300	5	Targeting compact code, low energy
SC110	3	4 words/cycle	16 bits w/ 16-bit prefixes	300*	5	Scaled-down SC140
Carmel	2, 6	4 words/cycle	24/48 bits	180	8	CLIW instructions, 4 AGUs
TMS320C64xx	8	8 words/cycle	32 bits	600*	11	Enhanced 'C62xx
TMS320C55xx	2	3 words/cycle	8-48 bits	160*	7	Based on 'C54xx

Figure 27 Caractéristiques des principaux DSP VLIW (source BDTI)

1.3.4 Fonctionnalités SIMD

Contrairement aux termes *conventionnels*, *VLIW* ou *superscalaire* qui caractérisent un type de processeur à travers son architecture et son jeu d'instructions, le terme *SIMD* (pour *Single Instruction Multiple Data*) traduit simplement la capacité d'un processeur à exploiter le parallélisme de données. Cette fonctionnalité était déjà disponible sur quelques DSPs conventionnels sous une forme assez basique, celle des unités fonctionnelles dites « split-alu », qui permettaient d'effectuer, avec une ALU 32 bits, au choix 2 opérations 16 bits ou une opération 32 bits. On retrouve ce genre de capacités, peu coûteuses en terme de complexité matérielle, dans la plupart des DSPs spécialisés pour le domaine de la téléphonie dans lequel les données sont codés sur 16 et 32 bits (Figure 28).

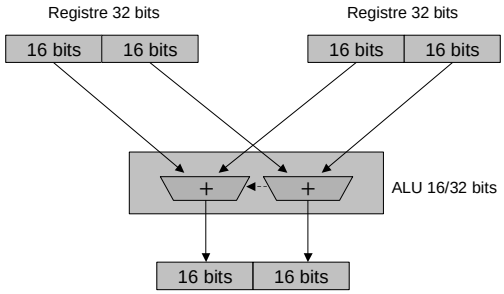


Figure 28 Split-ALU 16/32 bits

Ce type de traitement SIMD consistant à diviser la largeur de la donnée opérande et à appliquer le même traitement à chaque « morceau » obtenu est connu sous le nom de « parallélisme de sous mots » (*subword parallelism* ou *SWP*). C'est la forme la plus élémentaire du parallélisme de données ; sa granularité est de quelques instructions (2 dans le cas d'une instruction SIMD 16/32 bits).

L'autre forme de parallélisme de données est basée sur le parallélisme de boucles de plus grandes taille (voir 1.3.1.3) et n'a fait pour l'instant l'objet que d'une seule implémentation dans un processeur DSP. Il s'agit du *TigerSHARC* d'Analog Device [20]. Son jeu d'instructions intègrent des instructions SIMD dites « hiérarchiques » qui traitent les deux formes de parallélisme de données : le SWP au niveau des unités fonctionnelles et un parallélisme de plus haut niveau de granularité « chemin de données ».

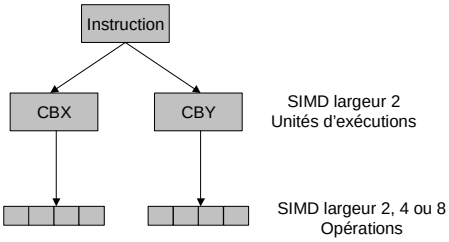


Figure 29 Instructions SIMD hiérarchiques du TigerSHARC

Le TigerSHARC permet de manipuler des données de taille 64, 32, 16 ou 8 bits. La capacité à manipuler des données de taille différente est intéressante dans la mesure où on demande de plus en plus aux processeurs d'être capable de gérer différents types de médias, comme des images fixes, des vidéos ou de l'audio. Dans le cas où la taille requise est inférieure à 64 bits, le processeur peut tirer parti des capacités SWP de ces unités fonctionnelles pour paralléliser les calculs.

Le chemin de données est composé de 2 unités de calcul identiques (CBX et CBY, Figure 30) disposant chacune de trois unités fonctionnelles et d'un banc de registres associé. Ces deux unités peuvent être commandés indépendamment à la manière d'un processeur VLIW classique (le SHARC autorise deux instructions indépendantes pour chaque unité, soit 4 instructions par cycle au total), ou être commandé de manière synchrone par une instruction unique. Dans ce cas, les opérations spécifiées dans l'instruction sont exécutées simultanément dans les deux chemins de données. Ce type d'instruction SIMD de haut niveau permet l'exploitation efficace du parallélisme de boucle, chaque chemin de données effectuant en parallèle les mêmes traitements sur deux données différentes correspondant à deux occurrences indépendantes d'une boucle.

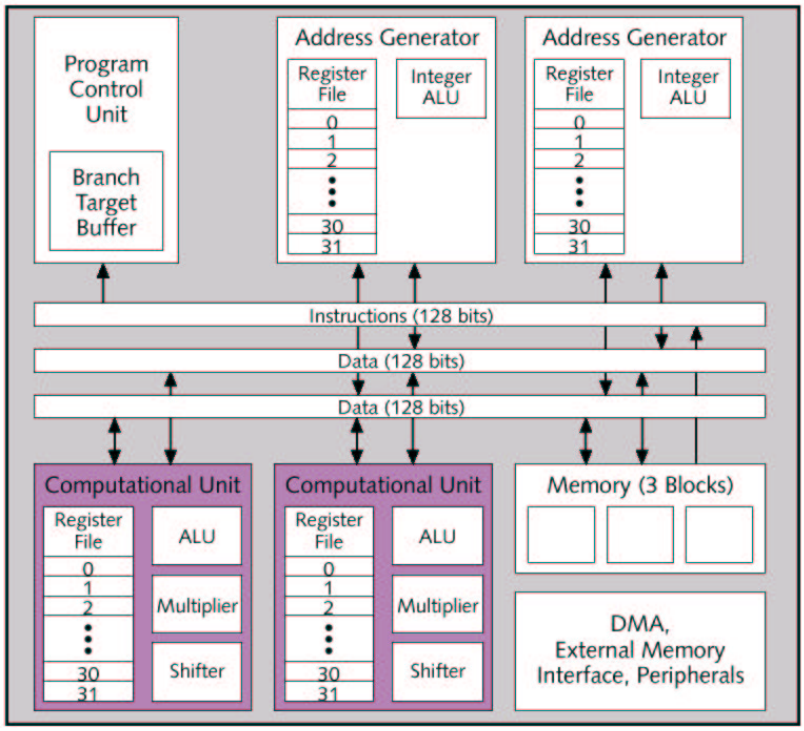


Figure 30 Architecture du TigerSHARC

Les performances de ce processeur sur des algorithmes réguliers exposant beaucoup de parallélisme de données comme le filtre FIR ou la FFT sont excellentes [21], la capacité de calcul maximale du TigerSHARC étant largement supérieure à celle des autres DSPs (8 multiplications 16*16 bits par cycle, contre 2 voire 4 maximum pour les autres processeurs). C'est la nature de ces algorithmes composés de boucles indépendantes traitant les données par blocs qui permettent l'utilisation optimale du jeu d'instruction SIMD et des ressources de calcul du processeur. Dans ce genre de cas, le SIMD permet des gains supérieure à ceux des architectures basées sur le parallélisme d'instructions.

Malheureusement, tous les algorithmes n'ont pas la structure d'un filtre FIR. Les avantages liés au SIMD disparaissent dès lors que les algorithmes traitent les données non plus par bloc mais une à une sous forme de flux, ou font usage de boucles récursives (le résultat d'une itération dépendant de celui de l'itération précédente). Dans certains cas, il est possible de modifier la structure de l'algorithme en faisant par exemple disparaître les dépendances entre itérations de boucle afin de l'adapter aux exigences du SIMD. Mais cela demande un travail important de la part du programmeur, et représente une perte de généralité du code qui devient spécialisé pour l'architecture cible et donc moins lisible et exploitable.

De plus, la syntaxe assembleur SIMD et la nécessité d'exposer et d'utiliser le maximum de parallélisme font que la programmation d'une application en assembleur devient un exercice particulièrement ardu. Pour s'en convaincre, il suffit de voir le code source du filtre FIR pour le *TigerSharc* présenté en [21] : une trentaine de lignes, des boucles déroulées, des registres ou clusters implicites…alors qu'un filtre FIR se code en quelques lignes sur un DSP classique ou VLIW. Ce problème pourrait être contourné par des outils de production de code performants détectant et exploitant le parallélisme de données, mais la performance des compilateurs actuels rend la programmation assembleur indispensable dans la plupart des cas (cf. 1.4).

Les capacités SIMD complexes comme celles du TigerSHARC posent aussi des problèmes d'alignement des données en mémoire et requièrent une bande passante importante. De manière générale, la quantité mémoire requise pour une implémentation SIMD efficace est supérieure à celle requise par les architectures classiques (déroulement de boucles, réarrangement des données en mémoire).

Pour résumer, on peut dire que contrairement aux concepts VLIW ou superscalaire dont l'efficacité est à priori assez peu dépendante du domaine d'application visé, l'intérêt d'une architecture SIMD évoluée (c'est à dire au delà d'une simple unité « split-ALU ») dépend de la nature des algorithmes mis en jeu dans l'application et de l'importance qu'on attache à la difficulté de programmation et à la taille du code généré.

1.3.5 Architectures Hybrides RISC / DSP

La plupart des circuits spécialisés en téléphonie mobile intègrent souvent à la fois un microcontrôleur (ou un microprocesseur de type embarqué) et un processeur DSP. Le premier est chargé de toutes les taches de type contrôle (interface utilisateur, synchronisation du système) tandis que le second est utilisée pour exécuter les applications DSP réclamant une forte puissance de calcul et donc une architecture spécialisée. Cette approche bi-processeurs a plusieurs inconvénients : nécessité de développer du code pour deux jeux d'instructions différents, coût lié à l'utilisation de deux processeurs (encombrement, consommation, prix), difficulté d'interfaçage, etc.

Certains constructeurs se sont donc intéressés à des solutions permettant de réaliser avec un circuit unique les taches assignées au microprocesseur et au processeur DSP. Au niveau architectural, le problème revient à faire cohabiter de façon efficace et peu coûteuse des capacités de contrôle et de traitement DSP. Plusieurs solutions ont été proposées :

- Microcontrôleur RISC + coprocesseur DSP. Exemple, le *Massana FILU-200* [22]. Le microcontrôleur assure la plupart des traitements et délègue les calculs orientés DSP au coprocesseur spécialisé. Les deux cœurs peuvent travailler en parallèle pour gagner en performance. Le principal inconvénient provient de la complexité potentielle de la programmation : manipulation de deux jeux d'instructions et gestion efficace des communications entre les cœurs. La complexité de l'interface entre les deux cœurs peut rendre les communications RISC-coprocesseur très coûteuse en temps d'exécution.
- Microcontrôleur avec capacités DSP. Exemple, le *SH-DSP* de Hitachi. Ce circuit est une extension du microcontrôleur SH-2 auquel on a ajouté un chemin de données et des instructions spécialisées DSP. Un jeu d'instruction unique contrôle les deux chemins de données, ce qui simplifie la programmation et permet la réutilisation du code du microcontrôleur, mais ne permet pas de faire fonctionner simultanément les deux chemins de données, ce qui limite la performance.
- Processeur DSP avec capacités de contrôle. Exemple, le *TMS320C27x*. Cette solution présente les mêmes défauts et avantages que l'option microcontrôleur + capacités DSP, mis à part qu'elle privilégie le coté « DSP » aux capacités de contrôle.

- Processeur à architecture mixte DSP/Contrôle. Exemple, le *TriCore* de Infineon [23]. Ce processeur est basé sur une architecture superscalaire intégrant des unités fonctionnelles spécialisées pour les calculs DSP et des capacités de contrôle développées. L'architecture proposée n'étant pas une modification d'une architecture existante, elle est plus homogène et donc potentiellement plus simple d'emploi. Elle ne permet par contre plus la ré-utilisation de code existant.

Quelque soit la solution adoptée, les hybrides RISC/DSP sont en fait des compromis entre des architectures résolument orientées DSP et d'autres définitivement spécialisées contrôle. De ce fait, leurs performances brutes ne peuvent rivaliser avec celles des processeurs DSP « haute-performance » et se situent plutôt aux environs de celles des DSPs d'entrée de gamme (cf. Figure 23 (d)). Elles ont cependant un intérêt certain pour des systèmes réclamant une double compétence contrôle/DSP, sous réserves que la performance requise ne soit pas trop élevée.

Benchmark	*Nombre de Cycles (%)*	*Mémoire Programme (%)*	*Mémoire Donnée (%)*
N Real Update	373	1306	53
N Complex Update	137	882	26
Dot Product	620	1309	680
Matrix	24	414	5
Convolution	1076	1545	100
FIR	1045	769	97
FIR 2D	427	743	158
IIR N Biquad	140	764	55
LMS	406	-35	78

Figure 31 Comparaison Assembleur Compilé / Assembleur Optimisé pour le compilateur du C54x

1.4 Rôle et performance des compilateurs

Pendant longtemps le rôle des compilateurs dans la méthodologie de développement logiciel des applications DSP a été négligeable. Parce que les architectures et les jeux d'instructions des processeurs cibles étaient complexes et inadaptés à la génération automatique de code, les programmeurs avaient pris pour habitude de coder les applications directement en assembleur. Les fabricants de DSP eux-mêmes favorisaient cet état de fait tant la qualité des compilateurs qu'ils proposaient était médiocre (voir l'exemple du compilateur du TMS320C54x pour le benchmark DSPstone Figure 31 [24]). La programmation assembleur avait ainsi l'avantage de garantir à la fois une performance optimale et une bonne compacité de code. L'étude de [25] sur les habitudes de travail des équipes de conception dans une grande entreprise de Telecom en 1995 (*Northern Telecom*) illustre bien l'importance de la programmation en assembleur par rapport à l'emploi d'un langage de haut niveau (Figure 31).

C DSP	8%
Assembleur DSP	55%
C microcontroleur	9%
Assembleur microcontroleur	28%

Figure 32 Usage de l'assembleur et du C chez Northern Telecom

1.4.1.1 Les limites du « tout-assembleur »

L'arrivée des nouvelles architectures DSP et l'accroissement de la complexité des applications ont contribué à faire évoluer les mentalités. La tendance actuelle dans le monde de l'embarqué consiste à proposer des services plus nombreux et performants tout en réduisant le plus possible le temps de développement. Dans ce contexte, la méthode du « tout-assembleur » a deux principaux inconvénients. D'une part, il est très difficile de diminuer le temps de développement. La programmation assembleur est un exercice délicat et laborieux qui prend nécessairement du temps, d'autant plus que les programmeurs doivent maintenant tenir compte des nouveaux problèmes ayant trait à la gestion du parallélisme dans les processeurs modernes. Le deuxième problème concerne la ré-utilisation du code. Le code assembleur étant intrinsèquement non portable, un changement de processeur oblige à reprogrammer l'ensemble de l'application, occasionnant une perte de temps considérable.

1.4.1.2 Adéquation Compilateur / Architecture

La bonne performance des processeurs DSP provient avant tout de la présence de mécanismes matériels spécialisés intégrés à l'architecture et qui permettent d'exploiter efficacement les caractéristiques algorithmiques communes des applications DSP : opérateurs arithmétiques performants (MAC, BMU), mécanisme de boucles zéro-cycle, modes d'adressages spécifiques, partitionnement de la mémoire en plusieurs bancs. Dans un soucis d'économie de matériel, les processeurs conventionnels possèdent une architecture très contrainte : peu de registres, topologie irrégulière du chemin de données, nombreux bits de mode, encodage complexe des instructions, etc. C'est cette complexité, très contraignante pour les outils de génération de code, qui explique la faiblesse des compilateurs pour ce type de processeur.

Les nouveaux processeurs exploitant le parallélisme d'instructions ont à l'inverse des architectures plus régulières et des jeux d'instruction moins complexes. De nombreuses études sur la conception de compilateurs optimisés pour architectures parallèles ont été lancées et commencent à donner des résultats intéressants. Ces meilleurs résultats sont dus pour une part à la simplification des architectures cibles et d'autre part aux progrès des algorithmes de génération de code. La Figure 33 illustre la supériorité d'un compilateur pour DSP VLIW par rapport à un compilateur pour DSP conventionnel [26].

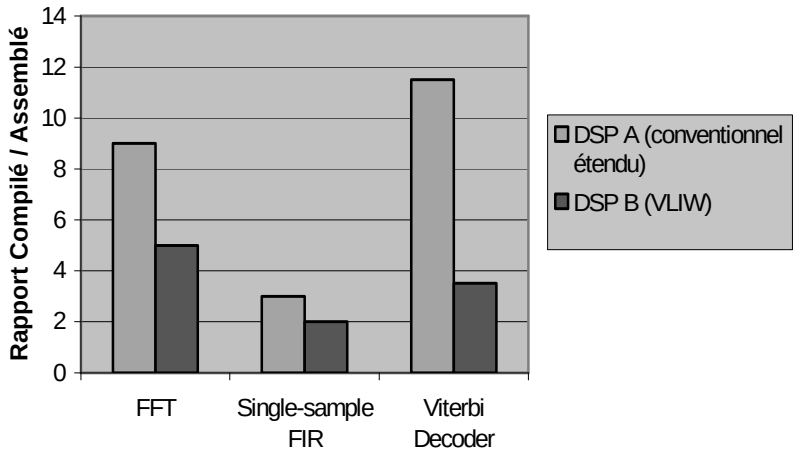


Figure 33 Comparaison de performance (vitesse) Code Compilé / Assemblé

Pour produire un code DSP performant, un compilateur doit tout d'abord être capable d'utiliser efficacement les mécanismes matériels « spécialisés DSP » inclus dans le processeur : unités fonctionnelles spécialisées, boucles *For* câblées, adressage circulaire, bancs mémoire multiples, etc. Il doit aussi être capable d'extraire de l'application le maximum de parallélisme possible (instruction, données) afin d'exploiter au mieux les capacités du processeur. Les gains en performances obtenus par rapport à un compilateur «classique» sont alors très intéressants (Figure 34) [27].

	Low-Overhead Looping	***Memory Partitioning***	***Operation Compaction***	***Overall***
Kernels	1,78	1,49	3,48	4,86
Applications	1,23	1,07	2,11	2,83

Figure 34 Impact des différentes optimisations sur la performance du compilateur

1.4.1.3 Nouvelles méthodologies de conception

L'amélioration de la performance des compilateurs ouvre désormais la voie à de nouvelles méthodologies de production de code DSP. Le « 100% assembleur » laisse petit à petit la place à des développement mixtes Assembleur/Compilateur. L'implémentation du codeur de voix GSM EFR sur le DSP *StarCore* présentée en [28] repose sur le principe du « 90 / 10 » : les fonctions comptant pour 90% du temps d'exécution après compilation de l'application entière sont recodées manuellement en assembleur tandis que le reste du code (10% du temps d'exécution) est conservé. Cette méthode se base sur le constat que pour la plupart des applications, 90% du temps d'exécution s'effectue dans seulement 10% du code [29]. Ce postulat se vérifie particulièrement bien pour les applications DSP dont les noyaux de calculs intensif sont des boucles composées de seulement quelques instructions. C'est dans ces noyaux critiques que l'on retrouve le plus de parallélisme susceptible d'être exploité par des architectures parallèles [27].

L'étude présentée en [30] étudiant la performance de compilateurs DSP récents montre que la programmation assembleur reste indispensable pour coder les fonctions critiques des applications. En revanche, l'écart de performance plus faible constaté pour les portions non critiques du code montre que le compilateur peut être une alternative très intéressante : les faibles pertes en performance et en taille de code générés sont largement compensées par le gain obtenu en temps de développement. Une solution de plus en plus répandue consiste à optimiser à la main les portions critiques directement en

assembleur pour maximiser la performance, et de compiler le reste du code avec les options d'optimisation portant sur la taille du code généré.

Les performances obtenues par le compilateur du processeur VLIW TMS320C6x montre que de grands progrès ont été effectués dans le domaine de l'extraction et de l'exploitation du parallélisme d'instructions. Les autres formes de parallélisme sont pour l'instant beaucoup moins bien exploitées. Particulièrement, l'usage efficace des propriétés SIMD des processeurs pose pour l'instant de gros problèmes. Quant au parallélisme de taches, l'absence d'architecture réellement spécialisées (de type MIMD) et la description des algorithmes en langages de haut niveau essentiellement séquentiels le rendent pour l'instant presque inexploitable.

1.5 Conclusion

La dernière décennie a suscité de nombreux bouleversements dans le monde des processeurs DSP : nouveaux marchés et nouvelles applications, nouvelles architectures inspirées des microprocesseurs haut de gamme, émergence de compilateurs plus efficaces. L'amélioration des performances ne s'explique plus uniquement par l'évolution des procédés de fabrication mais aussi grâce aux nombreuses innovations architecturales, méthodologiques et logicielles.

Avec le succès des applications embarquées, principalement du à l'explosion de la téléphonie mobile, les critères d'excellence ont changés et la course à la performance se double maintenant d'un intérêt pour les circuits très basse consommation. Alors que les processeurs d'il y'a dix ans se basaient sur un modèle unique sensé fonctionner pour tous les types d'application, on assiste aujourd'hui à une diversification des solutions proposées, afin de s'adapter aux contraintes parfois diamétralement opposées des différents domaines d'application. Selon le positionnement de l'application dans l'espace des « trois P » (*Price*, *Power*, *Performance*), un processeur donné sera ou non adapté à l'application visée.

Architecture	Issue Width	Instruction Scheduling	Instruction Type	When	SIMD	Typical Clock (MHz)
Conventional	1	Compile-time	Complex	1980-now	None to minimal	75-150
Enhanced Conventional	1	Compile-time	Complex	1996-now	Minimal to extensive	100-150
VLIW	2-8	Compile-time	Simple	1996-now	Minimal to extensive	100-300
Superscalar	2-4	Run-time	Simple	1997-now	Minimal	200

Figure 35 Les différentes familles de processeur DSP (source BDTI)

La famille de processeurs de Texas Instruments illustre très clairement cette tendance, qui propose deux gammes de circuits bien différenciés : des processeurs destinés aux applications embarquées basse consommation de type téléphone mobile dont l'architecture est de type « conventionnelle étendue » (les C54xx et C55xx), et des processeurs VLIW offrant de meilleures performances mais au détriment du coût et de la puissance dissipée (C62xx et C64xx, cf. Figure 36).

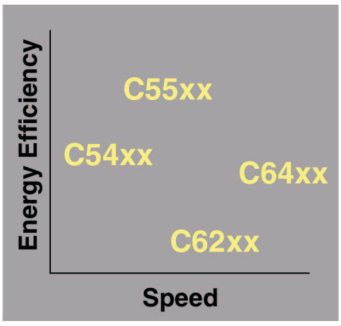


Figure 36 Les deux grandes gammes de processeurs Texas Instruments

Une autre approche radicalement différente consiste à utiliser un même modèle de processeur et à le spécialiser en fonction des exigences liées au domaine d'application visé. Les futurs processeurs SC110 et SC140 de Motorola sont en fait deux versions différentes de la même architecture baptisée StarCore [31]. Le premier intègre un seul opérateur MAC et vise l'intégration dans des systèmes « clients » faible consommation / performance moyenne. Le second possède 4 opérateurs MAC et doit être utilisé dans des systèmes « serveurs » réclamant de très grandes puissances de calcul, comme les « *base stations* » des réseaux téléphoniques mobiles. Les avantages d'une architecture configurable sont nombreux : spécialisation de l'architecture en fonction des besoins de l'application, utilisation des mêmes outils logiciels, portabilité possible du code assembleur, méthodologie unique, etc.

Une autre tendance très nette dans la conception des systèmes DSP consiste à effectuer de plus en plus de taches en logiciel, et de réserver les blocs matériels spécialisés pour les fonctions critiques qui ne peuvent pas encore être réalisées en logiciel. Une solution intermédiaire très prisée dans l'industrie consiste à utiliser des processeurs « ASIPs » (*Application Specific Instruction-Set Processor*) dont l'architecture est spécialisée pour un domaine d'application particulier, et qui utilisent des instructions et des opérateurs matériels spécifiques. Ces processeurs ont pour vocation d'offrir des performances égales ou mêmes supérieures à celles des processeurs DSP généraux tout en minimisant le coût d'architecture grâce à la spécialisation.

C'est précisément sur la conception de ce type de processeur que porte cette thèse. Notre objectif est de proposer un modèle de processeur configurable et une méthodologie de conception associée qui permette d'accélérer le temps de développement de ce type de processeur. Avant de présenter le modèle de processeur, nous allons dans le prochain chapitre présenter deux notions fondamentales de notre approche : la notion de spécialisation d'architecture et de jeu d'instructions en fonction d'une application cible, et la notion de modèle de processeur configurable.

Chapitre 2

Processeurs spécialisés et configurables

Sommaire :

2.1 Introduction : les processeurs ASIP

Si on demande à un utilisateur « lambda » quel type de processeur est intégré dans son PC ou son Macintosh, les chances de bonne réponse sont réelles. Si par contre on lui pose la même question concernant sa console de jeux ou son téléphone portable, la probabilité devient quasi nulle. La sur-médiatisation des processeurs généraux fait penser à tort qu'ils sont les leaders de l'industrie en terme de volume de ventes de processeurs. Pourtant, ce sont les microcontrôleurs qui prédominent encore aujourd'hui du fait de leur utilisation massive dans les produits électroniques grand public. Les processeurs DSP connaissent eux aussi de plus en plus de succès grâce à l'explosion des domaines du multimédia et de la téléphonie mobile.

L'étude menée dans une grande entreprise de télécommunications (*Northern Telecom*) en 1995 illustre une autre tendance, celle de l'utilisation de processeurs « maison », c'est-à-dire développés en interne par les équipes de conception pour un projet particulier [25] (cf. Figure 37).

	Usage dans les équipes de conception (%)	*Volume de ventes généré (%)*
DSP Maison	16	32
DSP Commerce	28	23
microcontrôleur maison	14	31
microcontrôleur commerce	42	14

Figure 37 Les types de processeurs utilisés à Northern Telecom

La différence de pourcentage entre l'usage des différents types de processeurs au sein des équipes de conception et le volume de vente généré par ces mêmes processeurs s'explique par le fait que les processeurs généraux sont avant tout utilisés dans des projets nécessitant un temps de développement très court, tandis que les processeurs maisons sont surtout prédominants dans des applications grand public à fort volume de production où le coût joue un rôle crucial.

Contrairement aux processeurs du commerce dont la vocation est de pouvoir exécuter un panel très large d'applications, les processeurs maison sont spécialement conçus pour un domaine d'application particulier, voire parfois même pour une seule application. L'intérêt recherché est d'obtenir des processeurs possédant les performances minimales requises par l'application en terme de puissance de calcul (et éventuellement de consommation) et dont le coût est inférieur à celui des processeurs du commerce. De tels processeurs spécialisés sont souvent appelés *ASIPs* (pour *Application-Specific Instruction-Set Processors*).

Les ASIPs sont fréquemment utilisés dans les domaines d'application DSP les plus populaires comme le multimédia ou la téléphonie mobile (Figure 38). Souvent, la première génération d'un produits utilise des processeurs standards du commerce. La disponibilité immédiate du processeur et de ses outils logiciels permet de raccourcir le temps de développement afin de sortir au plus vite le produit. Si le lancement est un succès, une seconde génération basée sur des ASIPs maison permet de réduire le coût par unité du circuit et d'augmenter les marges bénéficiaires, le coût de développement de l'ASIP étant amorti par le grand nombre d'unités vendues. La flexibilité de la solution ASIP permet aussi la réutilisation du circuit dans d'autres produits visant le même type d'application, avec pour

seule modification la réécriture du logiciel embarqué, alors qu'un circuit basé sur un ASIC serait plus difficilement portable.

	Processeurs	***Applications***
Philips	Trimedia VLIW	MPEG1-2 Audio/Video
	EPICS	GSM
Italtel	DSP ASIP	Station de base GSM
France Telecom	ASIP DSP VLIW controlled multiprocessor system	Terminal main libre
Chromatic	Mpact	Carte PC 3D
Atari	64 bit ASIP	Console de jeu Jaguar

Figure 38 Exemples d'utilisation des processeurs ASIP

Dans les cas où les processeurs du commerce ne peuvent assurer les contraintes du cahier des charges du fait de leur trop grande généralité, les ASIPs tendent à remplacer de plus en plus les ASICs, illustrant en cela la tendance générale en vogue dans la conception de circuits qui privilégie de plus en plus le logiciel en remplacement du matériel. Ceci est principalement du à l'utilisation généralisée de standards dans tous les produits grand public, dont la complexité croissante et l'évolution continue plaide en faveur de solutions programmables. L'ASIP VLIW *France Telecom* de la Figure 38, par exemple, a permis une réduction de 50% de la consommation électrique par rapport à une solution basée sur un processeur du commerce, rendant la solution programmable envisageable pour l'application embarquée visée, en l'occurrence un terminal main-libre.

La plupart des ASIPs sont intégrés dans des systèmes complexes souvent réalisés sous forme de SoC. C'est pourquoi on entend surtout par ASIPs des cœurs de processeurs spécialisés destinés à l'intégration dans des systèmes complets.

Les ASIPs n'ayant pas la même souplesse d'utilisation que les processeurs généraux, leur plage d'utilisation en terme de diversité d'application est limitée. Le développement d'un processeur spécialisé coûtant très cher, ils ne sont donc rentables que s'ils sont produits en grande quantité, et ce alors qu'ils ne sont utilisables que pour quelques gammes de produits. Etant principalement utilisés dans des marchés très concurrentiels, leur durée de conception doit aussi être minimisée. En résumé, le coût de conception et le temps de développement sont les deux faiblesses majeures de la solution ASIP.

Pour éviter d'avoir à reprendre entièrement le flot de conception pour chaque nouvel ASIP, certains fabricants utilisent des cœurs de processeurs dits « configurables ». Ces cœurs ont des architectures et des jeux d'instructions possédant certains « degrés de liberté » qui leur permettent d'être paramétrés en fonction de l'application cible. Les outils de développement associés prennent en compte les divers paramètres, ce qui permet de tester différentes configurations, de sélectionner la meilleure et d'obtenir rapidement le circuit correspondant.

Ce chapitre s'intéresse à deux notions fondamentales pour notre travail qui sont la spécialisation d'un processeur pour une application, et la notion de modèle de processeur configurable.

2.2 Adéquation Algorithme / Architecture et spécialisation

La notion d'Adéquation Algorithme / Architecture (*AAA*) traduit la capacité d'un système matériel à satisfaire les contraintes requises par une application donnée, ces contraintes étant généralement formulées en terme de performance, de complexité matérielle et de consommation. Sur l'échelle du critère « AAA », les ASIPs sont placés juste en dessous des circuits ASICs, qui représentent le stade ultime de la spécialisation d'une architecture pour un algorithme donné (Figure 39).

Figure 39 Adéquation Algorithme/Architecture vis à vis des applications DSP

Les problèmes liés à la spécialisation peuvent se résumer ainsi : les améliorations apportées par la spécialisation sont elles suffisantes pour justifier l'accroissement de la complexité et la perte de généralité de la solution finale ? Les architectures spécialisées sont en effet sensées avoir une densité de calcul par unité de surface supérieure à celle des architectures générales, prouvant ainsi l'intérêt de la spécialisation [32].

Prenons l'exemple d'un processeur général auquel on ajoute une unité fonctionnelle et des instructions spécialisées pour faciliter un certain traitement complexe. La performance brute du nouveau processeur est évidemment supérieure, mais au prix d'une augmentation de la surface du circuit, liée à la fois au coût matériel de l'unité ajoutée et à l'augmentation de la taille du code (la largeur du mot d'instruction doit augmenter pour pouvoir encoder les instructions spécialisées). Ces ajouts n'ont donc d'intérêt que si le taux d'utilisation de la nouvelle unité est suffisant.

C'est pourquoi le choix de l'utilisation d'un ASIP et sa spécification sont généralement issus d'une étude précise de l'application cible, qui détermine quelles sont les contraintes critiques de l'application et si ces contraintes sont susceptibles d'être résolues de manière « rentable » par l'emploi de structures spécialisées. La recherche de l'architecture optimale passe par une phase d'exploration de « l'espace de conception » du processeur, qui doit déterminer le niveau (domaine d'application, application, algorithme, cf. Figure 40) et les types de spécialisation à mettre en œuvre.

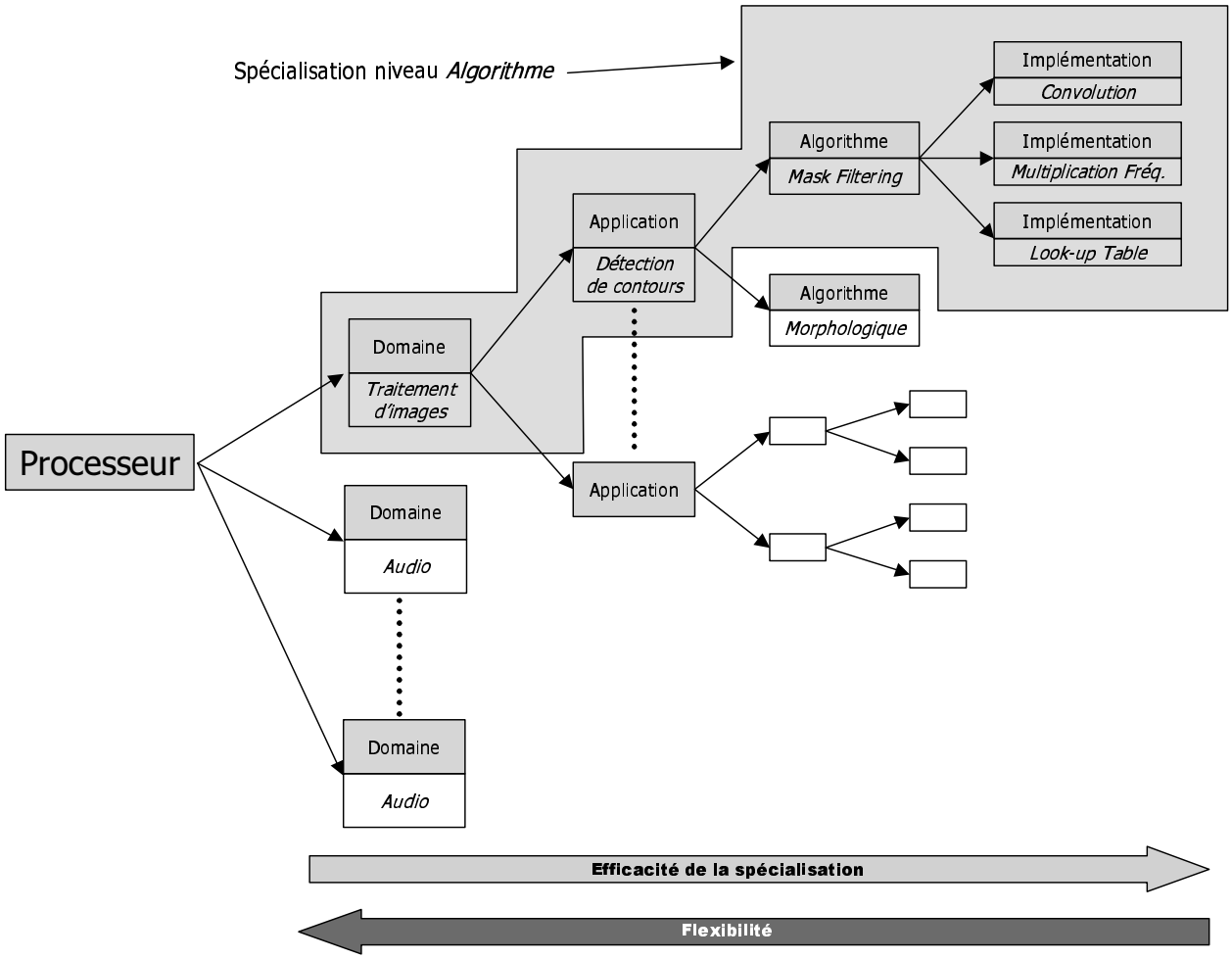


Figure 40 Les différents niveaux de spécialisation d'un processeur

2.2.1 Niveau de spécialisation

Il n'existe évidemment pas de modèle de processeur « universel » capable de traiter efficacement n'importe quel type d'application, tant leurs contraintes respectives peuvent être différentes et réclament des solutions parfois totalement divergentes. Même les microprocesseurs dits généraux sont en fait conçus pour un domaine d'application particulier, celui des applications informatiques « domestiques » (système d'exploitation, suites bureautiques, jeux…). De même, la grande variété des processeurs DSP permet de les classer par domaines d'applications (multimédia, audio/vidéo, communication). En fait, la spécialisation au niveau « domaine d'application » est une règle de base dans la conception de tous les processeurs.

Comme leur nom l'indique, les processeurs ASIP résultent d'une spécialisation poussée plus loin, au minimum au niveau *Application* (Figure 40). Le principe même des ASIPs repose sur l'idée que toutes les applications (même celles appartenant au même domaine) diffèrent l'une de l'autre vis-à-vis du parallélisme disponible et des besoins en ressources matérielles. C'est particulièrement vrai en traitement du signal où les contraintes et caractéristiques sont très variables selon les applications. La spécialisation peut descendre jusqu'au niveau le plus bas, celui de l'algorithme. La structure des

applications de traitement du signal dans lesquelles la majeur partie du temps du calcul est passée dans des boucles logicielles implémentant des algorithmes de bas niveau fait que la spécialisation sera d'autant plus efficace qu'elle sera poussée le plus bas possible. Cela se fait au détriment de la flexibilité, puisque la puissance du processeur est conditionnée à l'usage d'un algorithme unique, ce qui limite son champ d'applications à celles qui utilisent cet algorithme.

Une autre alternative consiste au contraire à limiter la spécialisation à un niveau supérieur, ce qui limite le gain potentiel mais laisse plus de place à l'utilisation pour des applications de nature différente.

[33] et [34] se sont intéressés aux différences existant entre la spécialisation au niveau *application* et celle au niveau plus large du *domaine d'application*. A partir d'un ensemble d'applications (*I, II, III, IV, V, VI*) appartenant au même domaine et d'un modèle de processeur paramétrable (Nombre et types d'unités fonctionnelles, nombre de registres, nombre de ports du banc de registres, taille des caches, etc.), ces deux études explorent l'ensemble des solutions et sélectionnent les configurations donnant les meilleurs résultats pour chacune des applications. Elles comparent ensuite leurs performances avec celles des configurations dites « moyennes » qui optimisent l'ensemble du domaine, afin de déterminer si il existe réellement une grande différence entre les deux niveaux de spécialisation. A titre d'exemple, la Figure 41 présente les performances de trois architectures : *A*, *B* et *Moyenne*, optimisées respectivement pour les applications *I* et *II*, et l'ensemble complet d'applications (*I,II, … VI)* [33]. Les conclusions qui en découlent sont les suivantes :

- Une optimisation en fonction d'une seule application permet dans certains cas d'obtenir un gain très supérieur à la solution moyenne (cas de l'application *I* Figure 41(a)), alors que dans d'autres cas , la différence est beaucoup plus mince (application *II*). Dès lors que les caractéristiques de l'application s'éloignent de celles de l'algorithme « moyen » du domaine d'application concerné (qui pourrait être par exemple la convolution dans le cas du traitement d'images), les différences avec la solution moyenne sont élevées. Dans le cas contraire (par exemple un filtre de type FIR basé comme la convolution sur une boucle de multiplication-accumulation), l'intérêt de la spécialisation diminue.
- Plus la surface de silicium autorisée est élevée, moins la différence sera grande entre la solution optimisée par une application et la solution « moyenne » (Figure 41(b)). Parfois, un changement mineur dans une architecture optimisée pour l'application A permet d'augmenter de manière spectaculaire son comportement pour les autres applications, moyennant une légère perte de performance pour *I*. La différence entre les deux niveaux de spécialisation sera donc d'autant plus grande que le coût matériel autorisé sera faible.

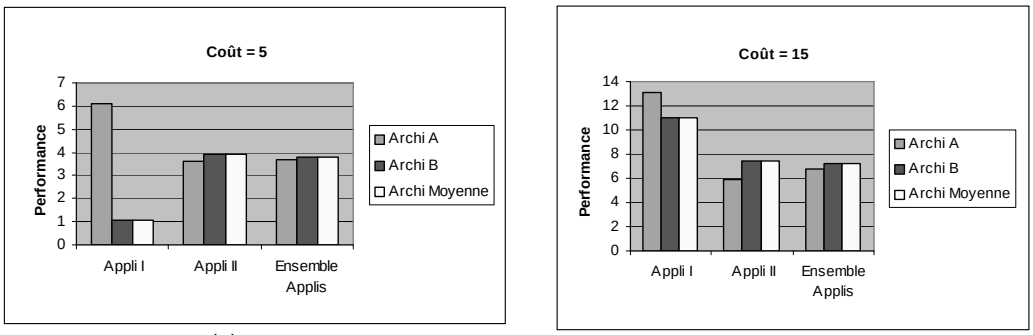


Figure 41 Influence de la spécialisation selon le coût matériel faible(a) ou fort(b)

Ce dernier principe se vérifie aisément dans le domaine des processeurs DSP. Les premiers processeurs dits « conventionnels » qui ne bénéficiaient pas encore des densités d'intégration actuelles avaient des contraintes très fortes en terme de nombre de transistors, les obligeant à employer des architectures très spécialisées et réduites en matériel. Pour offrir de bonnes performances dans quelques applications clairement définies, ces processeurs intègrent donc des unités fonctionnelles câblées très spécialisées et s'articulent sur une topologie de chemin de données très contraignante qui excluent l'exécution satisfaisante d 'applications autres que celles ciblées par le processeur.

A l'inverse, les processeurs les plus récents pour lesquels les contraintes en matériel sont moins fortes optent souvent pour des architectures plus générales et moins spécialisées. Grâce à une plus grande exploitation du parallélisme d'instructions et à des fréquences de fonctionnement plus élevées, ces processeurs peuvent obtenir les mêmes performances sur les applications critiques tout en améliorant considérablement le comportement des autres applications, ce qui leur confère une plage d'utilisation beaucoup plus grande.

A l'heure actuelle, la tendance est à une plus grande généralisation des architectures, qui passent du niveau de spécialisation *Application* à celui du *Domaine d'Application* pour permettre une plus grande flexibilité. C'est particulièrement vrai pour les processeurs destinés aux applications hautes performances pour lesquels le coût matériel et la consommation électrique ne sont pas déterminants. En revanche, dans le domaine des applications embarquées faible consommation, les processeurs spécialisés de type ASIP ont énormément de succès, grâce notamment à leur très faible consommation. On doit enfin noter que de nombreux DSPs dits généraux intègrent en fait des unités spécialisées : presque tous les processeurs DSP possèdent par exemple des mécanismes matériels permettant l'accélération du décodage de *Viterbi*, un algorithme utilisé dans de nombreux protocoles de communication (entre autres le *GSM* et *l'ADSL*). L'utilité de tels mécanismes se justifie souvent pour des raisons de performance, les solutions purement logicielles basée sur des unités fonctionnelles générales ne permettant pas d'atteindre les niveaux de calculs requis. On peut parler dans ce cas d'intégration d'unités spécialisées *Application* (ou même *Algorithme*) dans des architectures spécialisées *Domaine d'Application*. Si le taux d'utilisation de l'unité est suffisant, ce type de spécialisation peut s'avérer très payant.

2.2.2 Amélioration de la performance

Il existe deux manières principales d'accélérer le traitement d'une tache sur un processeur programmable : augmenter la fréquence de fonctionnement du processeur, ou faire en sorte que le processeur effectue plus de travail à chaque cycle machine. La première solution consiste à utiliser une technologie plus performante et ne nécessite pas de changement radical d'architecture. Pour des applications basse-consommation, cette solution n'est pas toujours satisfaisante car l'accroissement de la fréquence s'accompagne obligatoirement d'une augmentation de la puissance dissipée.

La deuxième solution, adoptée par la plupart des ASIPs, revient à exploiter le parallélisme intrinsèque des applications en effectuant plusieurs opérations élémentaires en un cycle. Les DSP généraux eux-mêmes intègrent de nombreuses techniques permettant l'exploitation du parallélisme (voir le chapitre précédant). Ces techniques peuvent être qualifiées de « générales » car elles sont applicables à n'importe quel type d'application (architecture VLIW) ou tout du moins au « domaine » traitement du signal (architecture Harvard, adressage circulaire, mécanismes de boucles matérielles, etc.).

Les processeurs ASIP poussent plus loin l'exploitation du parallélisme en adaptant leur structure aux formes de parallélisme dépendantes de l'application, qui ne peuvent être prises en compte par des architectures générales. Pour ce faire, deux méthodes sont souvent utilisées : la duplication de matériel à usage général, et l'utilisation de structures matérielles spécialisées.

2.2.2.1 Duplication du matériel

Etendre une architecture consiste à dupliquer une partie du matériel afin d'exploiter au mieux le parallélisme d'instructions disponible, toutes les applications n'ayant pas les mêmes besoins en terme d'unités fonctionnelles, registres, bande passante mémoire, etc. C'est le principe même des architectures VLIW et superscalaires, dont le chemin de données est formé de quelques unités fonctionnelles à usage général réparties horizontalement. Dans le domaine DSP, ces unités peuvent être des ALUs, des MACs, ou des unités de manipulation de bits.

La *Figure 42* illustre l'influence de la variation de la largeur d'exécution sur la performance d'une architecture VLIW constituée respectivement de 1, 2 ou 4 clusters de 6 unités fonctionnelles chacun (4 ALUs et 2 Multiplieurs) [35]. Les résultats montrent que l'efficacité du parallélisme *horizontal* varie grandement selon l'application, le gain en performance passant de moins de 10% pour *tjpeg* à plus de 40% par *csc* pour le passage de 1 à 4 clusters. Dans tous les cas, le gain apparaît relativement faible par rapport au surcoût matériel.

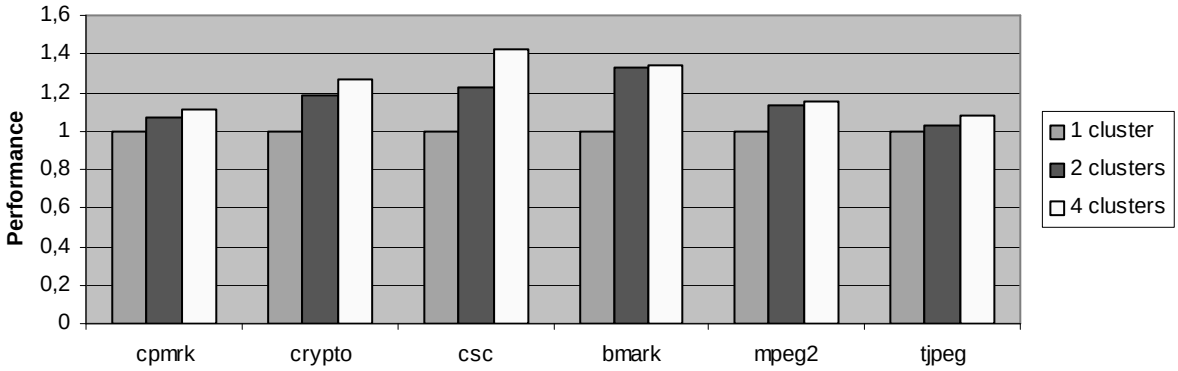


Figure 42 Influence de l'extension d'architecture sur la performance.

A titre de comparaison, l'augmentation de la fréquence de travail permet un gain quasi linéaire en performance quelque soit l'application, et a le mérite d'être plus simple à mettre en œuvre. La méthode de duplication du matériel apparaît donc « ingrate » vis-à-vis des résultats atteignables.

On peut cependant facilement diminuer le surcoût matériel en ne dupliquant que les unités réellement utiles à l'exploitation du parallélisme. La répartition des types d'opérations extraite de l'analyse d'un ensemble d'applications DSP (Figure 43) suggère la répartition en unités matérielles suivante [12]:

(Integer ALU, Load/Store Unit, Branch Unit, Shifter, Floating Point Unit, Multiplier) = (4,2,1,1,1,1)

Ces chiffres représentent évidemment une moyenne sur l'ensemble des applications et peuvent connaître de grands écarts d'une application à l'autre : les opérations de transfert mémoire par exemple représentent à peine 10% du total des opérations pour les applications de type audio contre plus de 30 % pour les applications graphiques, ce qui plaiderait en faveur de l'utilisation d'une unité *Load/Store* supplémentaire pour ces dernières. Il faut aussi noter que dans ce mode de calcul, on néglige le problème du parallélisme d'instructions qui est supposé infini. Or, bien que les opérations utilisant l'ALU soient majoritaires, elles sont parmi celles dont les contraintes de parallélisme (en particulier les dépendances de données) sont les plus grandes et qui risquent de réduire le parallélisme exploitable à 3 voire 2 ALUs au lieu de 4.

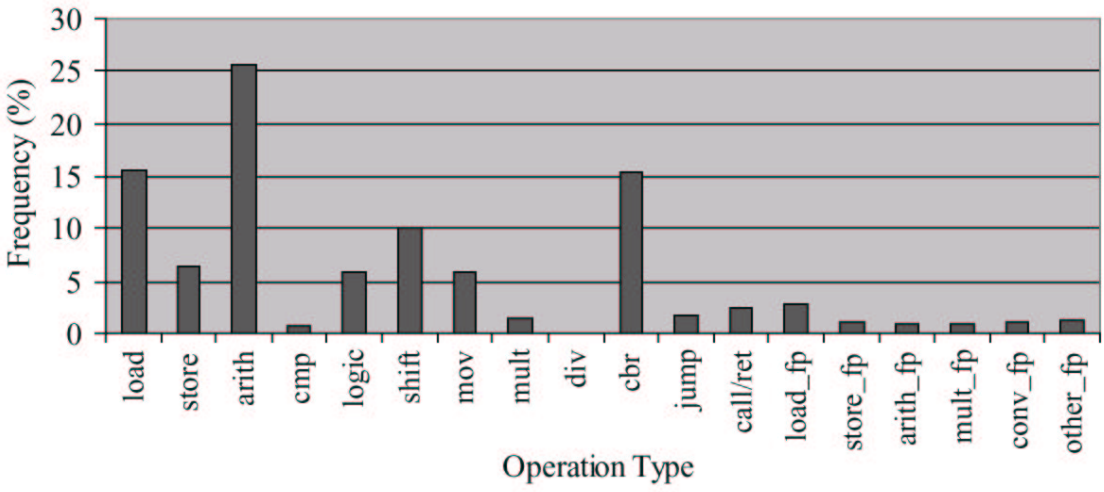


Figure 43 Fréquence moyenne des opérations pour les applications MediaBench

Dans l'optique de la conception d'un ASIP, il est donc utile d'effectuer au préalable une analyse dynamique de l'algorithme afin de déterminer les besoins quantitatifs pour les différents types de ressources, avec si possible une prise en compte du parallélisme intrinsèque de l'application. Une méthodologie de conception basée sur cette approche est présentée au chapitre 4.

2.2.2.2 Structures matérielles spécialisées

L'intégration d'unités fonctionnelles et d'instructions spécialisées dans les processeurs DSP est monnaie courante même dans les processeurs qualifiés de généraux. En général, ces unités sont utilisées pour accélérer le comportement d'un algorithme dont l'exécution à l'aide de ressources générales n'a pas été jugée satisfaisante. Ces algorithmes constituent souvent les parties critiques d'applications de plus haut niveau soumises à de fortes contraintes temps réel. En intégrant des unités matérielles capables d'effectuer des multiplications dites « de Galois », le récent *C64x* de Texas Instruments accélère considérablement l'implémentation de l'algorithme de décodage *Reed Solomon* utilisé dans la norme ADSL [36]. Les processeurs spécialisés dans le domaine des télécommunications comme le *DSP16xxx* de Lucent possèdent presque tous l'instructions ACS (*Add/Compare/Select*) nécessaire à l'implémentation efficace du décodage de Viterbi [37].

Lorsque l'analyse de l'application visée montre qu'une certaine fonction ne peut être codée de manière satisfaisante sur une structure de DSP classique, l'architecte peut choisir de « spécialiser » certaines ressources du processeur afin de tenir compte des spécificités de l'algorithme. Plusieurs solutions d'implémentation existent :

- Ajout de fonctionnalités supplémentaires dans des ressources matérielles existantes, et ajout des instructions correspondantes dans le jeu d'instruction. C'est le cas par exemple de la fonctionnalité « multiplication de Galois » du *C64x* qui a été ajoutée aux unités de multiplication à usage général déjà présentes dans la version antérieure du processeur, le *C62x*.
- Ajout de ressources matérielles distinctes contrôlées implicitement ou par des instructions supplémentaires. Le DSP16xxx intègre par exemple un bloc matériel baptisé « Traceback encoder » accélérant le décodage de Viterbi. Cette unité composée d'un bloc de calcul combinatoire et d'un registre de contrôle est activée de manière implicite par certaines instructions de jeu d'instruction [37]. Ce type de ressources supplémentaires peuvent être considérées comme des coprocesseurs spécialisés intégrés au chemin de données.

On va maintenant illustrer ces deux méthodes par deux exemples d'implémentation.

<u>L'instruction Add/Compare/Select du DSP16xxx</u>

L'algorithme de Viterbi permet la recherche de la prochaine transition la plus probable dans un graphe d'état. Dans les applications de communication, il est utilisé pour la détection et la correction d'erreurs dans la transmission d'un flux de données [38].

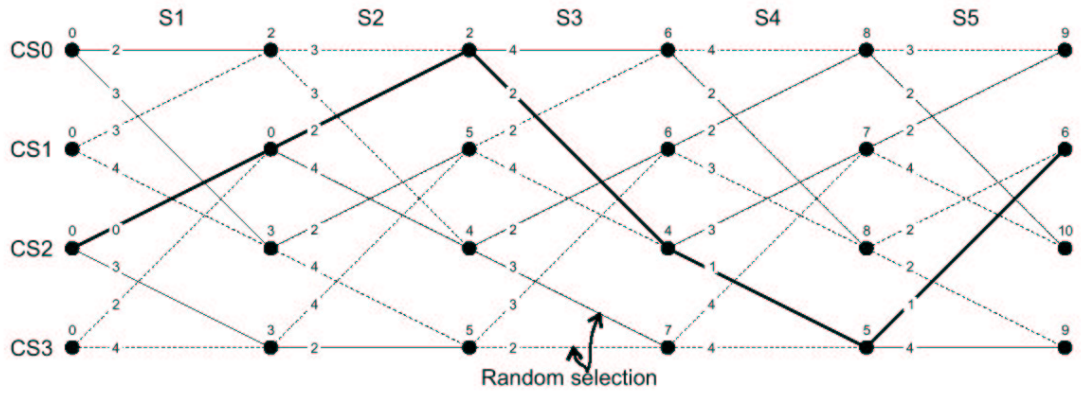


Figure 44 Treillage 4 états (CSx) 5 transitions (Sy)

Chaque transition sur le treillage correspond à la réception d'une donnée. L'examen par l'algorithme de Viterbi des transitions correspondant à la réception des données permet de détecter certaines erreurs et de les corriger. L'algorithme parcourt le treillage, détecte les transitions erronées, et fournit en résultat le chemin le plus probable correspondant à la suite de données corrigée.

La Figure 45 présente le pseudo-code de la fonction « *Survivor path update* » exécutée à chaque état *Sy* du treillage. Elle calcule pour chaque état CSx la somme des contributions de l'état et de la branche menant à cet état, et sélectionne le minimum. L'ajout dans le jeu d'instructions de l'instruction Add/Compare/Select permet de réduire à 5 cycles le temps d'exécution de la fonction. Avec un jeu d'instruction classique, il faudrait quatre instructions distinctes (somme, comparaison, saut conditionnel puis affectation) pour effectuer le même calcul, dont une (le saut conditionnel) très coûteuse en nombre de cycles à cause des effets de pipeline. Si on estime la pénalité liée au branchement à 5 cycles, ce qui correspond à la longueur moyenne des pipelines de contrôle dans les DSPs, l'instruction ACS permet de réduire d'un facteur 8 le temps d'exécution de la fonction, et ce à un coût matériel faible, puisque la fonctionnalité ACS peut être facilement intégré dans une ALU existante (cas du *DSP16xxx*).

```
min = infinite                // Init. new survivor path metric

if (branch[1]+path[1]) < min // Add-compare-select the best
   min = branch[1]+path[1]   // survivor path metric

if (branch[2]+path[2]) < min
   min = branch[2]+path[2]

if (branch[3]+path[3]) < min
   min = branch[3]+path[3]

if (branch[4]+path[4]) < min
   min = branch[4]+path[4]

// min = New survivor path metric found
```

Figure 45 Fonction "Survivor path update " de l'algorithme de Viterbi

Unité matérielle spécialisée « Flux de bits » pour l'implémentation du MPEG4

Extraire un nombre aléatoire de bits d'un flux de données séquentiel est la tâche principale de la fonction VLD (*Variable Length Decoding*), une des fonctions principales des algorithmes de compression vidéo MPEG. Parce que les manipulations s'effectuent au niveau bit et nécessitent de

nombreuses opérations de décalages et de branchements, les implémentations logicielles sur des processeurs conçus pour manipuler surtout des « mots » ont des performances généralement insuffisantes.

L'algorithme VLD fait un usage intensif de trois fonctions de manipulation de bits appelées *ShowBits*(), *GetBits*() et *FlushBits*(). La seule fonction *ShowBits()*, dont le rôle est la lecture d'un certain nombre de bits à partir d'un pointeur courant vers un registre destination, est à elle seule appelée environ tous les trois bits du flux entrant. Afin d'accélérer l'exécution du VLD, [39] propose d'utiliser un module matériel séparé implémentant efficacement ces trois fonctions.

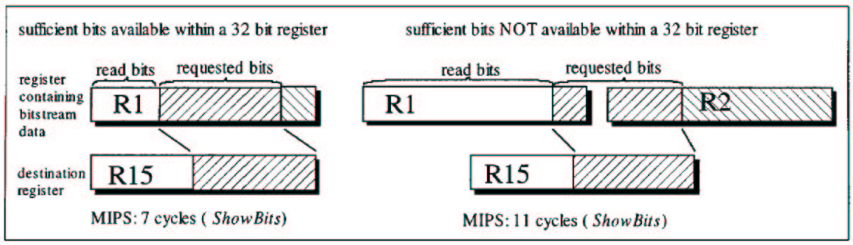


Figure 46 Fonction ShowBits() avec entrée dans deux registres

La Figure 46 illustre les problèmes posés par la fonction *ShowBits()* dans les cas où les bits requis sont dans un ou deux registres. Selon le cas, le codage avec un jeu d'instruction RISC classique 32 bits prendra de 7 à 11 cycles. Le module matériel spécialisé présenté Figure 47 permet d'exécuter les trois fonctions de manipulation de bits en un seul cycle. L'utilisation d'un décaleur 64 bits et d'un compteur de bits permet de rendre transparentes les opérations d'extraction de bits entre deux registres consécutifs et l'écriture dans les registres d'un nouveau mot de 32 bits. Au total, cinq nouvelles instructions sont ajoutées qui commandent le module et permettent une accélération globale de la fonction VLD d'un facteur 3.

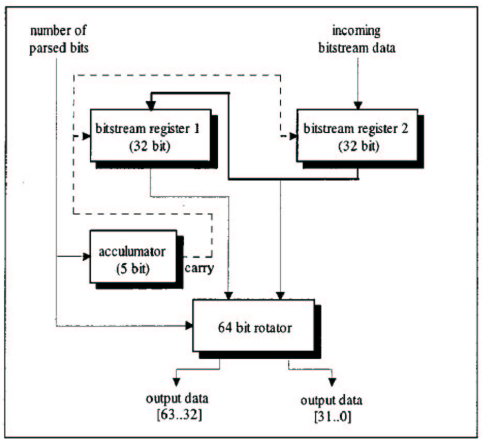


Figure 47 Architecture de l'unité spécialisée "Bit-processing"

La norme MPEG-4, qui standardise les méthodes de compression audio et vidéo pour des domaines d'application beaucoup plus nombreux que ceux ciblés par les normes précédentes (*MPEG-1/2*, *H.263*), utilise pour cela un très grand nombre d'algorithmes et de schémas de codage différents, ce qui tend à favoriser l'approche logicielle, plus flexible que l'approche matérielle sous formes d'ASICs. Néanmoins, pour des fonctions critiques comme le VLD, les niveaux de performance requis ne peuvent être obtenus que par accélération matérielle. De nombreux autres parties de l'algorithme

MPEG4 peuvent être accélérées grâce à des instructions et des modules matériels spécifiques [39]. Cette application est représentative du futur des applications Multimédia et illustre bien le passage progressif de solutions purement matérielles à des approches mixtes matérielles/logicielles. Associé à l'augmentation du nombre des ressources de calcul, l'usage d'instructions et de modules matériels spécialisés apparaît dès lors comme un bon moyen d'optimiser la puissance de calcul des processeurs en tenant compte des spécificités des applications.

2.2.3 Réduction du coût matériel et de la consommation

La section précédente s'est attachée à décrire les méthodes pour améliorer le comportement dynamique d'une application particulière sur un processeur. On s'intéresse maintenant aux moyens de limiter le coût matériel et la consommation afin d'obtenir des processeurs à la fois performants pour un ensemble défini d'applications et pouvant répondre aux contraintes des systèmes embarqués faible coût/faible consommation.

2.2.3.1 Réduction de la surface de silicium

Avant qu'apparaissent les problèmes de consommation liées à l'accroissement des fréquences de fonctionnement des processeurs et aux contraintes toujours plus exigeantes imposées par les applications embarquées, la limitation de la surface de silicium était la priorité majeure des concepteurs de circuits, principalement pour deux raisons. D'une part, le nombre de transistors relativement faible qu'offraient les technologies de l'époque imposaient de limiter le matériel pour « tenir » sur le circuit. D'autre part, le coût de fabrication étant proportionnel à la surface de silicium, il était économiquement indispensable de fabriquer le circuit le plus petit possible.

	1982	1992	2002
Die size (mm)	50	50	50
Technology size (microns)	3	0.8	0.18
MIPS	5	40	5,000
MHz	20	80	500
RAM (words)	144	1,000	16,000
ROM (words)	1,500	4,000	64,000
Price (dollars)	150	15	1.50
Power dissipation (mW/MIPS)	150	12.5	0.1
Transistors	50,000	500,000	5 million
Wafer size (inches/mm)	3 / 75	6 / 150	12 / 300

Figure 48 Deux décennies d'évolution technologique pour les DSPs

Si cette dernière préoccupation est toujours d'actualité, la première l'est de moins en moins, grâce aux dizaines (et bientôt centaines) de millions de transistors rendus disponibles par les technologies submicroniques (Figure 48). Pour le cas des cœurs de processeurs destinés à l'intégration dans des systèmes On Chip, le problème n'est pas tant de réduire la surface du cœur mais plutôt celle de la mémoire associée au processeur. Dans le domaine des processeurs DSP, la part de la mémoire embarquée par rapport à la surface totale du circuit se monte actuellement à près de 80%, et la tendance continue de s'accentuer à mesure que la technologie évolue. L'évolution du nombre de registres disponibles dans les cœurs de DSP traduit bien cet état de fait, qui est passé d'un ou deux

registres isolés pour les processeurs conventionnels de première génération à un banc de registres multi-ports de 64 registres pour le dernier C64x de Texas Instruments.

Les techniques de limitation de la surface de silicium sont nombreuses et bien connues, on peut citer entre autres :

- la limitation en nombre et en complexité des ressources de calcul.
- la limitation des capacités de mémorisation (registres et mémoire embarquée).
- la réduction de la connectivité du chemin de données et des bancs de registres (Figure 49).

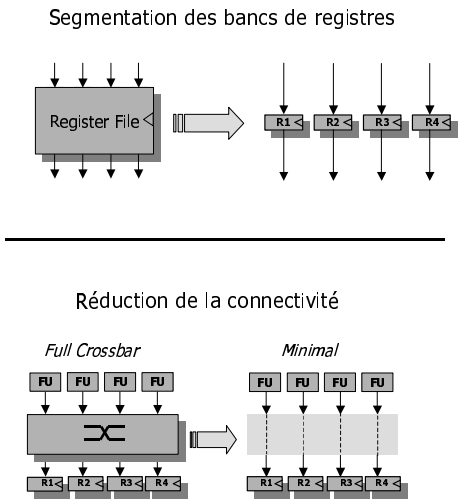


Figure 49 Réduction du matériel dans un chemin de données

Un bon exemple de réduction matérielle est illustré Figure 50 avec l'architecture d'un ASIP destiné aux applications Telecom. Par rapport à un DSP classique, ce processeur ne dispose que d'un bloc mémoire unique, les connexions entre les ressources sont réduites au strict minimum (en écriture la mémoire n'est connectée qu'à un seul registre : R1) et la largeur du chemin de données est optimisée en fonction de l'application (bus de données de 8,12 et 16 bits). Pour un coût matériel moindre, ses performances sont pourtant équivalentes à celles d'un DSP du commerce, le chemin de données ayant été conçu pour optimiser le comportement du processeur vis à vis des fonctions critiques de l'application.

Outre l'influence sur la surface du cœur, le fait de réduire les ressources matérielles a une influence direct sur le nombre de bits nécessaires pour encoder le jeu d'instruction, le nombre d'opcodes différents et le nombre des opérandes possibles étant réduits. Le bénéfice de la réduction se reporte donc aussi sur la taille de la mémoire programme, ce qui peut occasionner des gains en surface plus grands que ceux obtenus par la diminution du cœur lui-même. Profitant de la plus faible largeur des instructions élémentaires, certains ASIPs comme celui de Northern Telecom utilisent du coup des jeux d'instruction VLIW, plus gourmands en mémoire mais qui permettent de tirer parti du maximum de parallélisme disponible dans l'application. De manière générale, les jeux d'instructions des ASIPs comportent beaucoup moins d'instructions que ceux des processeurs du commerce de manière à optimiser l'encodage. Une pratique courante consiste à partir d'un jeu d'instruction complet, calculer les taux d'utilisation de chacune des instructions et ne sélectionner au final que les instructions les plus utilisées.

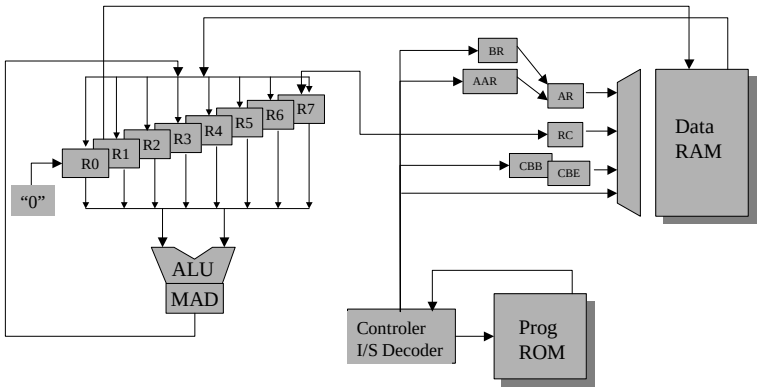


Figure 50 BNR/Northern Telecom DSP ASIP

La stratégie d'encodage a donc une importance capitale sur le coût final d'un processeur au travers de la taille de la mémoire programme, surtout dans le cas des architectures VLIW. La taille de la mémoire Donnée peut elle aussi réduite en privilégiant au maximum l'emploi de bancs mémoire simple port plutôt que double port. Ces derniers sont en effet beaucoup plus coûteux en terme de surface, mais aussi de temps de cycle et de consommation (Figure 51).

	Simple Port	***Double Port***
Surface	1	1.7
Temps de cycle	1	1.1
Puissance dynamique	1	3

Figure 51 Coût des mémoires simple/double port (0.35µ)

2.2.3.2 Réduction de la consommation

La réduction de la surface de silicium et la diminution de la consommation d'un processeur sont des problèmes intimement liés, puisque la consommation d'un circuit dépend directement du nombre de transistors utilisés. De ce fait, toutes les techniques de réduction du matériel embarqué dans un processeur ont à priori une influence bénéfique sur la consommation, plus ou moins grande selon le type de matériel. Dans le cas des circuits destinés aux applications embarquées, c'est même souvent la volonté de réduire la consommation qui motive la réduction de la surface plus que le désir de réduire le coût.

La relation qui lie la surface et la consommation n'est cependant pas triviale, et il existe de nombreuses autres techniques pour minimiser la consommation d'un circuit. Le problème de la réduction de consommation en général étant trop complexe pour être abordé de manière exhaustive, on ne considère ici que les méthodes de réduction au niveau architecture de processeur. Des méthodes de réduction de consommation à d'autres niveaux (système, algorithme, logique, technologie) sont décrites en détail dans [40] et [41].

La puissance dynamique d'un circuit est modélisée par la formule suivante :

$$Pdyn = \alpha \bullet C_l \bullet Vdd^2 \bullet f$$

où α est proportionnel à l'activité de transition des signaux et dépend des données, *Cl* est la capacité équivalente du circuit, *Vdd* la tension d'alimentation et *f* la fréquence de travail. Les méthodes de réduction de consommation tendent toutes à réduire un ou plusieurs de ces paramètres, de manière directe où indirecte.

Diminution de la fréquence

Si on fait abstraction des améliorations liées à l'emploi de technologies performantes basse-consommation, le moyen le plus employé par les concepteurs pour réduire la consommation consiste à diminuer la fréquence de travail du processeur. Les applications de traitement du signal réclamant de grandes puissance de calcul, cette diminution n'est possible que si le processeur effectue plus de travail par cycle machine, c'est-à-dire exploite plus de parallélisme. C'est pourquoi la plupart des ASIPs basse consommation utilisent des architectures fortement pipelinées et des jeux d'instructions parallèles de type VLIW, qui leur permettent d'effectuer le même travail que des processeurs RISC mono-scalaires à une fréquence plus faible. L'exploitation du parallélisme utilise les techniques décrites dans la section précédente, à savoir la duplication des ressources et la spécialisation d'architecture.

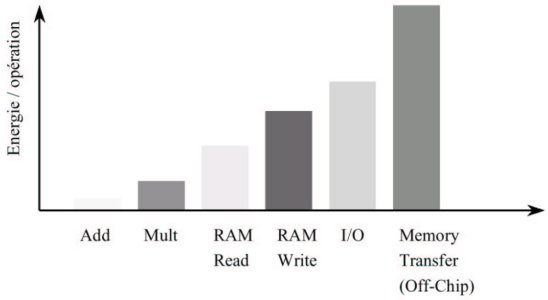
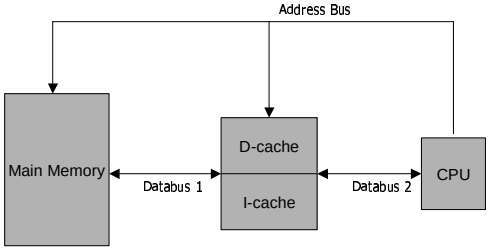


Figure 52 Répartition de la consommation par opérations pour l'application H.263 (source [40])

Réduction de la consommation mémoire

Une deuxième possibilité consiste à diminuer la consommation des différentes ressources matérielles. Les chiffres de la Figure 52 montrent clairement que dans un système « cœur de DSP + mémoire On Chip », ce sont les opérations d'accès à la mémoire qui sont responsables de la majeure partie de la consommation. Pour réduire l'importance de ce type de consommation, le concepteur peut jouer sur deux paramètres :la consommation par accès, et le nombre d'accès.



	CPU	Caches	Memory	All buses	Total
cpr	465	252	24	1018	1760
diesel	3.084	0.257	0.056	1.574	4.971
key	446	46	4.9	606	1104
mpeg	727	438	1.7	1547	2715

*Energy (Joule * 10-3)*

Figure 53 Répartition de la consommation dans un SoC

La consommation par accès provient de la consommation du bloc mémoire et de celle des bus reliant le processeur à la mémoire. L'étude menée en [42] sur la consommation dans les systèmes On Chip montre que la consommation des unités mémoires (mémoire principale et caches) est relativement faible par rapport à la consommation des bus (Figure 53). Cette dernière dépend avant tout de propriétés physiques du circuit (résistance et capacités linéaires des fils de routage), de la longueur des bus et de leurs charges respectives.. L'utilisation de blocs de caches, placés plus près du processeur que la mémoire centrale, a une influence bénéfique du fait de la diminution de la longueur des bus. Le concepteur a aussi tout intérêt à réduire au minimum la largeur des bus d'accès.

La consommation de la mémoire elle même n'est pas négligeable et peut être réduite en diminuant la taille des blocs, en hiérarchisant l'espace mémoire et en privilégiant au maximum l'usage de mémoires simple-port plutôt que double-port. Là encore, une stratégie d'encodage efficace des instructions permet de réduire à la fois la taille de la mémoire programme mais aussi la largeur des bus d'accès, occasionnant une réduction significative de la consommation globale du système.

Mais la méthode la plus efficace consiste à diminuer le nombre global de transactions mémoires. Les accès à la mémoire externe, qui sont de loin les plus coûteux, peuvent être restreints en faisant résider les données les plus fréquemment accédées dans les bancs mémoire On Chip, qui doivent pour cela être d'une taille suffisante. Pour les accès à la mémoire interne, la conservation des données fréquemment utilisées dans les registres du processeur permet d'éviter les accès redondants. Au niveau architectural, cela signifie que le processeur doit disposer de suffisamment de registres. La duplication de registre (registres *shadow*) destinée à accélérer les changements de contextes permet d'économiser les accès mémoires nécessaires à la sauvegarde et à la restitution du contexte.

Une dernière approche intéressante consiste à stocker les mots de données en mémoire sous forme compressée et à utiliser un décodeur matériel en sortie des mémoires caches. La diminution des accès à la mémoire centrale et l'accroissement de la capacité du cache liée à la compression permettent des réductions importantes de la consommation du système, jusqu'à 80% par rapport à un système sans compression [42]

Réduction de la consommation du cœur

La consommation, la surface et le temps de cycle des opérateurs arithmétiques utilisés dans les DSPs (ALUS, MACs) varient beaucoup selon le type d'architecture choisi (Figure 54). De nombreux compromis sont possibles : privilégier la consommation au détriment de la performance, favoriser la performance plutôt que la surface, etc.

	16 bit	32 bit	64 bit
Ripple Carry	3.09	0.81	0.27
Carry Lookahead	10.0	3.54	1.76
Carry Bypass	5.45	2.39	0.99
Carry Select	4.44	2.08	1.00
Conditional Sum	3.82	1.23	0.42

*Figure 54 Rapport «1 /(consommation*temps de propagation) » des architectures d'additionneurs (source [40])*

La consommation globale d'un cœur de processeur (hors accès mémoires) est principalement due à la consommation dans les blocs combinatoires (unités fonctionnelles, multiplexeurs, logique de

contrôle) et à l'écriture de données dans les registres. Un moyen de limiter la consommation des blocs combinatoires est de faire en sorte que les entrées d'un bloc ne varient que lorsque celui-ci est réellement utilisé, ce qui évite les transitions combinatoires inutiles et responsables des pertes d'énergie (les fameux *Glitchs*). Une méthode simple consiste à disposer des registres à l'entrée et à la sortie de chaque unité fonctionnelle du chemin de données, et de ne valider l'écriture dans le registre d'entrée que lorsque l'instruction en cours réclame l'utilisation de cette unité.

De manière plus générale, il est souhaitable de n'activer à chaque cycle machine que les parties du processeur réellement concernées par l'instruction en cours, en inhibant l'horloge dans toutes les autre parties suite au décodage de l'instruction. Ce même décodage, très complexe dans les processeurs VLIW, a une part non négligeable dans la consommation du cœur. Un découpage hiérarchique du décodage des instructions VLIW permet de n'activer que les décodeurs des unités fonctionnelles concernées par l'instruction, ces unités étant connues après le pré-décodage de l'opcode.

2.3 Les processeurs configurables

Les cœurs de processeurs configurables, encore inconnus il y'a quelques années, sont le reflet de trois grandes tendances actuelles en microélectronique : le développement des *SoC*, le raccourcissement nécessaire du temps de développement lié aux contraintes des marchés, et la complexité croissante des nouvelles applications multimédia. Une solution basée sur un processeur spécialisé peut apporter de nombreux bénéfices par rapport à une solution générale, mais le temps nécessaire à sa conception doit être impérativement court. D'où l'idée d'un « modèle configurable » de cœur de processeur, disposant de plusieurs degrés de liberté, et qui sont figés par l'utilisateur en fonction des besoins de son application.

L'offre en matière de cœurs configurables progresse rapidement, et avec elle le degré de configuration des processeurs qui permet des gains en performance et des réductions de coût plus importants. L'*Xtensa* [43] de *Tensilica* et les cœurs d'*ARC* [44] sont les principaux cœurs de RISC configurables. Dans le domaine des processeurs DSP commerciaux, on trouve actuellement le *REAL* de *Philips* [45] et le *CARMEL10xx* [46] d'*Infineon*. Les DSP configurables ont aussi fait l'objet de nombreuses recherches dans le domaine universitaire ; on peut citer en exemple les projets *Metacore* [47], *Miliwatt* [48] et *Flexible DSP Core* [49].

Le degré de spécialisation lié aux paramètres de configuration varie selon les processeurs. Pour ceux cités précédemment , il reste cependant assez faible, tant au niveau matériel que logiciel. Les futurs projets *CARMEL20xx* d'*Infineon* [50], *StarCore* de *Motorola/Lucent* [31], *ST200-Lx* de *ST-Microelectronics/ Helwett Packard* [51] et *JAZZ* d'*Improv* [52]ont pour but d'offrir des degrés de spécialisation plus poussés grâce à des architectures matérielles extensibles et paramétrables, et des outils logiciels évolués et reciblables capables de générer le logiciel et le matériel pour chacune des architectures potentielles.

2.3.1 Le flot de conception

L'usage de processeurs configurables doit permettre d'atteindre des niveaux de performance supérieurs à ceux des processeurs à architecture générale tout en raccourcissant le temps de développement par rapport à celui d'un ASIC ou d'un ASIP. Pour ce faire, tous les processeurs

configurables s'appuient sur un flot de conception comportant deux étapes : l'exploration et la génération (Figure 55).

La phase d'exploration a pour but de déterminer la valeur optimale des paramètres de configuration du processeur. Elle s'appuie sur un ensemble d'outils logiciels (assembleur, simulateur d'instructions et éventuellement compilateur) dont le comportement tient compte des paramètres, et fournit au final une estimation précise de la performance des applications sur le modèle proposé. Si les chiffres obtenus satisfont les contraintes du cahier des charges, on peut alors passer à la réalisation matérielle du processeur correspondant aux valeurs courantes des paramètres. Dans le cas contraire, le modèle courant doit être modifié par le changement de la valeur d'un ou plusieurs paramètres.

La phase de génération permet ensuite d'obtenir l'image matérielle du cœur de processeur qui sera utilisée pour la réalisation physique du circuit. Cette description matérielle est produite à partir du modèle paramétré. Le niveau de description obtenu est variable selon les cœurs et dépend du type de marché visé, celui des cœurs logiciels (« soft ») ou des cœurs matériels (« hard »). Les cœurs logiciels sont fournis sous forme de netlist RTL synthétisable, en général au format *VHDL* ou *Verilog*. Leur grand intérêt provient de leur indépendance vis-à-vis de la technologie, contrairement aux cœurs matériels qui sont fournis sous forme de macro-cellules optimisées pour une technologie donnée. La représentation bas-niveau d'un cœur logiciel est obtenu par synthèse logique à l'aide d'outils de CAO. Les performances de ce type de cœur sont évidemment inférieures à celles des cœurs matériels plus proches de l'approche « full-custom ».

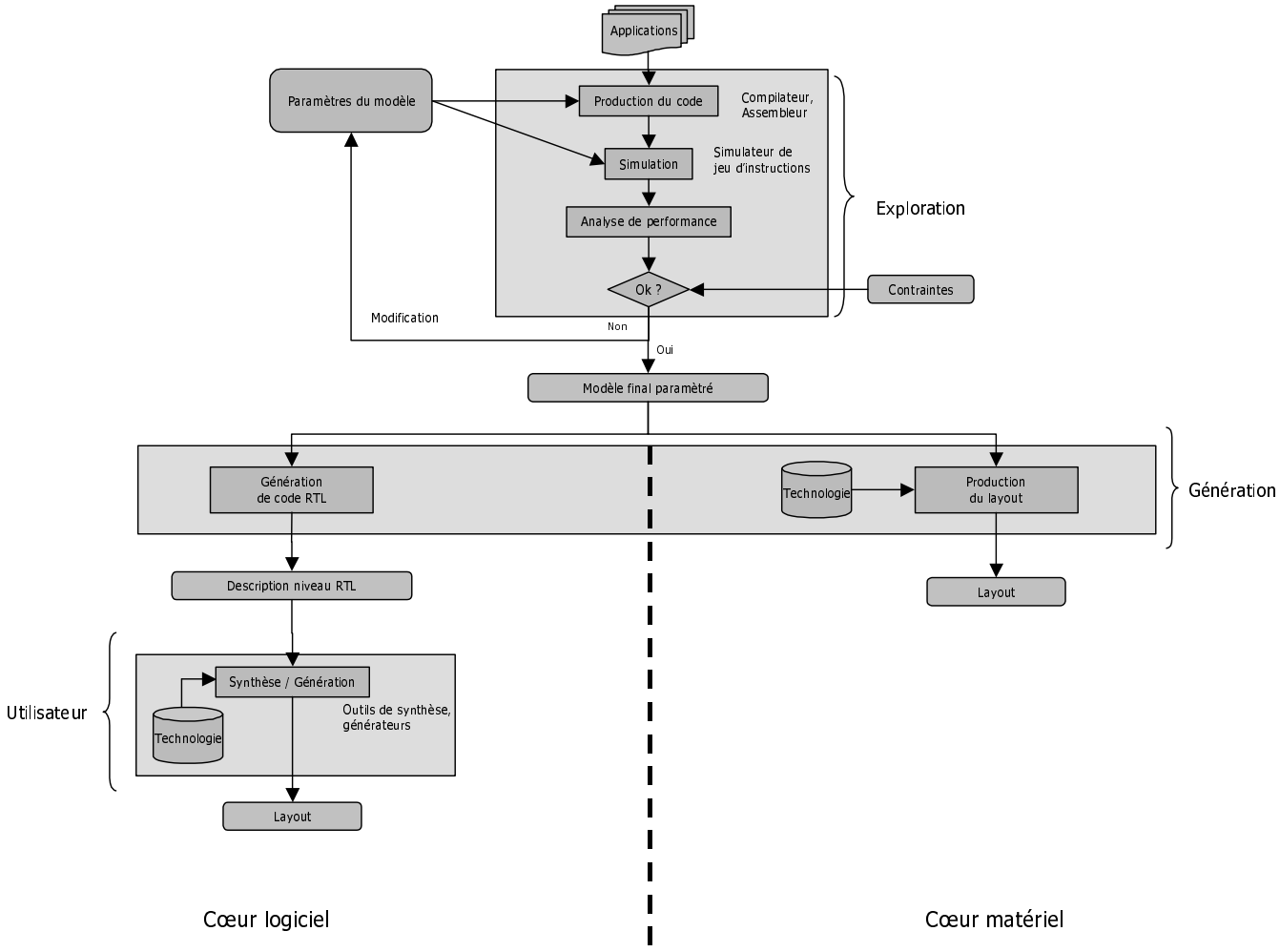


Figure 55 Flot de conception d'un processeur basé sur un modèle configurable

Le raccourcissement du temps de développement impose que le flot de conception soit le plus direct et le plus automatisé possible. Si l'approche « presse-bouton » est encore impensable au vu de la complexité du problème, il est cependant indispensable de disposer d'outils logiciels performants qui limitent l'action de l'utilisateur à des tâches non-évidentes : prises de décision, optimisations, etc. Le flot de conception basé sur la variation des paramètres de l'architecture impose aux outils d'être génériques vis-à-vis de ces paramètres pour pouvoir tester rapidement l'influence de leurs variations sur les performances. D'où l'utilisation d'outils de développement « reciblables » à chaque étape du flot de conception. C'est la principale difficulté inhérente au développement de processeurs configurables : si certains outils peuvent s'accommoder facilement de la gestion de paramètres, d'autres deviennent très complexes à réaliser. C'est le cas par exemple des outils de production de code (compilateurs et producteurs de code RTL), qui doivent générer un code fiable et efficace pour toutes les configurations possibles. Et plus le degré de spécialisation du processeur est grand, plus les outils doivent être complexes.

2.3.2 Les jeux d'instructions extensibles ou configurables

Outre le type d'implémentation matérielle proposé, l'autre différence entre les cœurs configurables concerne le degré de spécialisation du processeur, qui peut aller du simple ajout d'instructions dans un jeu d'instructions déjà existant jusqu'aux possibilités d'intégration d'unités fonctionnelles (UFs) utilisateurs complexes ou de variation de certains paramètres architecturaux.

Processeur	Jeu d'instruction	Architecture matérielle	Type de coeur
ARC core	• RISC 24 bits • instructions utilisateurs • code conditions utilisateurs	• 32 bits • nbre de registres calculs, registres adresses et bancs mémoires paramétrables • unité MAC à choisir parmi 16*16, 24*24, dual 16*16	????
Xtensa	• RISC 16/24 bits • instructions utilisateurs pour UF utilisateurs	• 32 bits • nbre de registres et d'interruptions, taille des caches paramétrables • unité MAC en option • UFs + registres utilisateurs	Soft
Miliwat	• RISC 21 bits • Instructions utilisateurs 21 bits	• 16 bits	Soft
Metacore	• RISC largeur paramétrable • Sélection parmi un JI prédéfini • Instructions utilisateurs	• Nbre de registres, largeur des registres et des bus paramétrables. • UFs utilisateurs	Soft
Flexible DSP Core	• RISC largeur paramétrable • Instructions utilisateurs	• Largeur des bus, des UFs et des registres paramétrables • Substitution des UFs de base par des UFs étendues (fonctions utilisateurs supplémentaires) • Nbre de registres de calculs, d'index et de boucles paramétrable	Soft
R.E.A.L.	• RISC 16/32 bits • Instructions utilisateurs VLIW (ASI)	• 16 ou 24 bits	Soft
JAZZ	• VLIW, nombre de slots et d'UFs par slot paramétrables	• Nbre de regstres et de unités d'accès mémoire paramétrables • Type et nombre d'UFs paramétrables • UFs utilisateurs	????
CARMEL 20xx	• RISC 24/48 bits • Instructions utilisateurs VLIW (CLIW)	• 16 bits • UFs utilisateurs ou provenant d'une librairie (« Power Plug Modules »	Soft
ST200	• VLIW longueur variable	• 32 bits • Nombre de clusters paramétrables (de 1 à 4)	Hard
StarCore	• VLIW longueur variable	• 32 bits • Nombre d'unités MAC paramétrables (de 1 à 4)	

Figure 56 Caractéristiques des principaux DSP configurables

Les processeurs configurables les plus simples reposent sur une architecture matérielle quasi-figée mais autorisent l'utilisation d'instructions supplémentaires définies par l'utilisateur, qui encodent des ensembles d'opérations non définies dans le jeu d'instructions de base. Il est en effet possible

d'accélérer certains algorithmes en combinant en une seule instruction quelques opérations élémentaires disponibles séparément dans le jeu d'instructions de base. La prise en compte des nouvelles instructions ne requiert aucune modification de l'architecture matérielle et est donc assez simple à mettre en œuvre. Seul le décodeur d'instructions devra changer pour prendre en compte le décodage des nouvelles instructions. Les processeurs *Miliwat* et *ARC* reposent sur ce principe. Dans *Miliwat*, la modification du décodeur est effectuée à l'aide d'un simple script *Perl* qui modifie le code VHDL du bloc de décodage.

Les instructions VLIW configurables des processeurs *CARMEL* et *R.E.A.L.* se fondent sur la même idée : accélérer les parties critiques des algorithmes, qui en traitement du signal sont souvent constituées de boucle de faible longueur dans lesquelles il est indispensable d'exploiter le maximum de parallélisme possible. Pour minimiser la taille du code généré, ces processeurs utilisent un jeu d'instructions « hybride » à la fois RISC et VLIW. La partie RISC du jeu d'instructions est destinée à coder les parties non critiques des algorithmes exposant peu de parallélisme d'instruction et représentant la majeure partie du code généré. Le fort taux d'encodage inhérent aux jeux d'instruction de type RISC permet ainsi d'assurer une bonne compacité du code. L'usage des instructions VLIW est réservée au codage des boucles critiques. Les nombreux degrés de liberté offerts par le codage VLIW permettent d'utiliser en parallèle un grand nombre de ressources du processeur, alors que le même travail nécessiterait plusieurs instructions RISC. Généralement, l'utilisation de quelques dizaines d'instructions VLIW seulement suffit pour atteindre des gains de performance importants.

Compte tenu du faible nombre d'instructions VLIW nécessaires, et pour éviter d'avoir à mettre en œuvre des mécanismes complexes de gestion d'instructions à longueur variable, ces processeurs utilisent un mécanisme astucieux de re-direction basé sur des tables d'instructions VLIW (Figure 57).

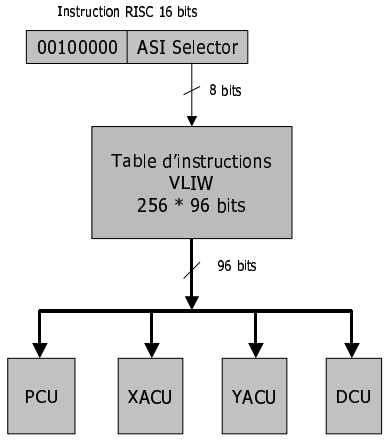


Figure 57 Principe des instructions ASI du processeur R.E.A.L.

Les instructions VLIW sont accédées par l'intermédiaire d'une instruction RISC spéciale dont la seconde partie constitue un pointeur sur une ligne de la table contenant les instructions VLIW proprement dites. La structure d'une instruction VLIW est constitué d'un nombre défini de champs correspondant à l'activation de plusieurs ressources de calcul du processeur. La table utilisée dans le R.E.A.L ne comporte que 256 entrées, illustrant le fait que très peu d'instructions larges sont nécessaires pour accélérer significativement les algorithmes. Pour fonctionner, ce type de codage

hybride nécessite un pipeline plus profond, qui effectuent les opérations de chargement et de décodage de l'instruction RISC, puis celles de l'instructions VLIW.

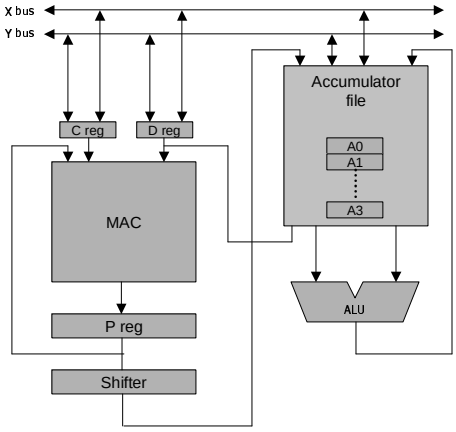
La prise en compte des instructions utilisateurs par les outils utilisés dans la phase d'exploration (compilateur, assembleur, simulateur) ne pose pas non plus de réelles difficultés. Le seul problème peut provenir de la capacité du compilateur à exploiter efficacement ce type d'instructions. Si les instructions utilisateurs de type RISC sont relativement bien prises en compte, il semble que les mécanismes d'instructions VLIW redirigés posent beaucoup plus de problèmes aux compilateurs et nécessitent une programmation manuelle pour être réellement efficaces.

2.3.3 Les architectures matérielles configurables

La prise en compte d'une architecture matérielle à plusieurs degrés de liberté constitue un problème beaucoup plus complexe que celui posé par de simples instructions utilisateurs. Parmi les processeurs existants, le degré de spécialisation est très variable, certains se limitant à des paramètres matériels relativement simple à mettre en œuvre comme la largeur des bus et des UFs, tandis que d'autres proposent d'intégrer des UFs utilisateurs dans le chemin de données du processeur.

2.3.3.1 Paramètres structurels simples

La solution la plus simple consiste à rendre paramétrables les caractéristiques de l'architecture ayant une influence faible sur la complexité des outils logiciels d'exploration et de génération. La variation de caractéristiques telles que la largeur des bus, des connexions du chemin de données et des UFs ou le nombre de registres dans les bancs de registres a peu d'influence sur la structure et le format des instructions. Seul le nombre de bits nécessaires pour coder les immédiats ou les opérandes de type registres changera. La topologie du chemin de données restant la même, les algorithmes de génération de code en seront peu affectés, à l'exception de la partie allocation de registres qui devra prendre en compte la variation des registres disponibles. L'automatisation de la phase de génération peut se faire relativement simplement à l'aide de scripts paramétrables modifiant les descriptions RTL en fonction de la valeur des paramètres et de générateurs de macro-blocks configurables générant automatiquement les unités arithmétiques régulières. Le « Flexible DSP Core » fait partie de ces processeurs qui permettent la variation de nombreux paramètres structurels (Figure 58). Bien que reposant sur un principe simple, les gains en matériel et en performance peuvent être très intéressant. Pour ce processeur, la réduction de la largeur de données de 16 bits à 12 a permis une réduction de 25% de la surface.



Parameter	Range	Default	Description
data word	8-64 bits	16 bits	
data addresss	8-23 bits	16 bits	
program word	32-...	32 bits	
program address	8-19 bits	16 bits	<=dataword
multiplier width	8-64 bits	dataword	Word length of multiplier operands
multiplier guard bits	0-16 bits	8 bits	<= dataword – 2
mac type	0-...	0(basic unit)	
alu type	0-...	0(basic unit)	
shifter type	0-...	0(basic unit)	
index regs	8 or 16	8	Number of index registers
accumulators	2,3,4	4	
enable cd	0 or 1	0(not enabled)	Use of C and D reg as ALU operands
modifier only	0 or 1	0(not enabled)	Only odd-numbered index registers can be modifiers
loopregs	0-8	0(no loop hardware)	Number of hardware looping units

Figure 58 Chemin de données paramétrable du « Flexible DSP Core »

2.3.3.2 Substitution d'unités fonctionnelles

La spécialisation des unités fonctionnelles étant un moyen très efficace d'augmenter la performance d'un processeur, le cœur « Flexible DSP Core » offre la possibilité de substituer les UFs de base par des UFs utilisateur, la seule contrainte étant que les nouvelles unités doivent posséder les mêmes fonctionnalités que celles d'origine. Le jeu d'instructions se trouve donc étendu aux instructions de contrôle des nouvelles unités. La génération matérielle reste simple puisque la nouvelle unité se substitue à une ancienne à la même place dans le chemin de données. Il suffit à l'utilisateur de décrire en VHDL le comportement de l'unité ou de fournir le macro-block correspondant, la phase de génération assemblant l'unité utilisateur et le cœur dans lequel le décodeur et une partie de la logique de contrôle auront été modifiés.

2.3.3.3 Variation du nombre et du type de ressources

Les processeurs configurables les plus évolués permettent de faire varier le nombre et la nature des unités fonctionnelles intégrées au chemin de données. La plupart d'entre eux imposent une structure de base composée d'UFs fixées à laquelle peuvent s'ajouter une ou plusieurs UFs utilisateur, les UFs de base assurant l'exécution d'un jeu d'instruction minimal nécessaire au bon fonctionnement du processeur. On trouve à la fois des processeurs basés sur une architecture « conventionnelle » à jeu d'instruction RISC (*Xtensa*, *Metacore*) et des processeurs VLIW (*CARMEL20xx*, *ST200*, *JAZZ*). La grande différence réside dans l'exploitation potentielle du parallélisme : tandis que les premiers ne peuvent exécuter qu'une instruction par cycle, ce qui revient à n'activer qu'une seule FU par cycle, les processeurs VLIW peuvent en activer plusieurs et permettent donc de meilleurs rendements des UFs.

L'ajout d'unités fonctionnelles utilisateurs accroît la complexité du processeur de plusieurs manières. Les outils de génération de code doivent gérer les nouvelles instructions de commande des unités. La topologie du chemin de données étant modifiée, c'est une grande partie de la description matérielle qui doit être régénérée. La phase de génération matérielle devient alors particulièrement critique. Un seul processeur (*Metacore*) a poussé la flexibilité jusqu'à rendre configurable la connectivité du

chemin de données. Le processus de génération matérielle pour ce processeur est illustré sur la Figure 59.

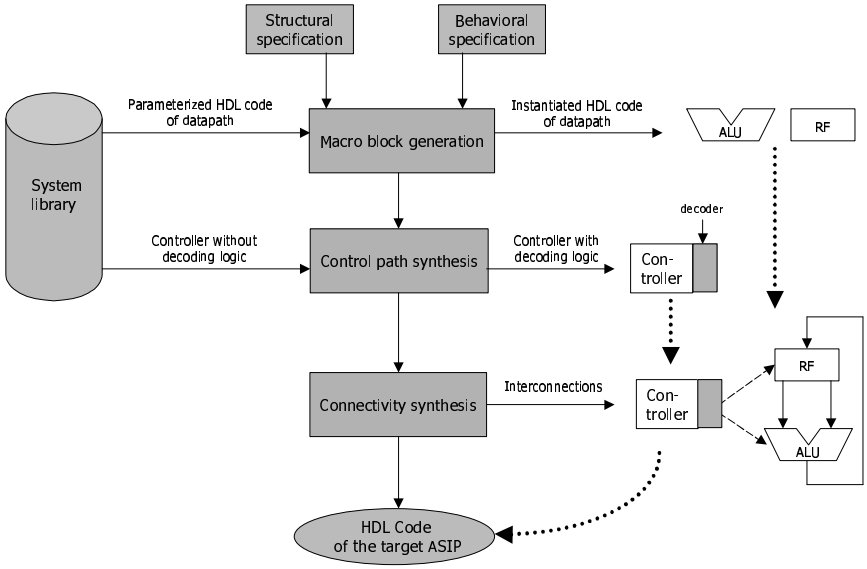


Figure 59 Flot de conception matérielle du processeur Metacore

Par rapport aux processeurs conventionnels intégrant des UFs utilisateurs, la difficulté de la phase de génération matérielle pour les processeurs VLIW est semblable, et principalement liée à la destruction de la topologie et de la régularité de l'architecture. C'est au niveau des outils de génération de code que la complexité s'accroît. Les problèmes de gestion du parallélisme dans les compilateurs pour processeurs VLIW sont bien connus (ordonnancement, allocation de ressources) mais ne sont que partiellement résolus, l'efficacité de compilateurs VLIW étant encore souvent jugée insuffisante. La compilation reciblable pour processeurs VLIW intégrant des ressources de nature diverses et en nombre variable reste encore un problème complexe et très ouvert, qui continue de susciter quantité de recherches et d'approches différentes [53].

2.4 Conclusion

De manière générale, le concept des processeurs configurable en est à ses débuts, comme en témoigne le faible nombre de processeurs commerciaux réellement disponibles à l'heure actuelle. Les premiers processeurs, de type conventionnels, se sont attachés à résoudre le problème de l'intégration et de l'exploitation d'architectures matérielles spécialisées (souvent des unités fonctionnelles utilisateurs) et de la génération matérielle associée, souvent sous forme de netlist RTL pour cœur soft. La voie suivie par les quelques processeurs VLIW configurables prévus prochainement (*ST200*, *StarCore*) semble différente puisqu'elle privilégie avant tout l'usage d'architectures extensibles en terme de ressources matérielles (nombre de clusters pour le *ST200*, nombre d'unités MAC pour le *StarCore*) et disponibles uniquement sous forme de cœurs hard.

L'aspect spécialisation permettant l'intégration de ressources utilisateurs (codées en VHDL ou Verilog) est cependant planifiée dans la majorité des cas. Les structures de jeux d'instructions adoptées pour gérer le parallélisme variable des architectures diffèrent selon les processeurs : encodage à longueur variable pour le *StarCore* et le *ST200* ou longueur fixe pour *CARMEL* et *JAZZ*.

Le concept du jeu d'instruction hybride RISC/VLIW semble ne pas être suivi à cause de la difficulté qu'ont les compilateurs à l'exploiter.

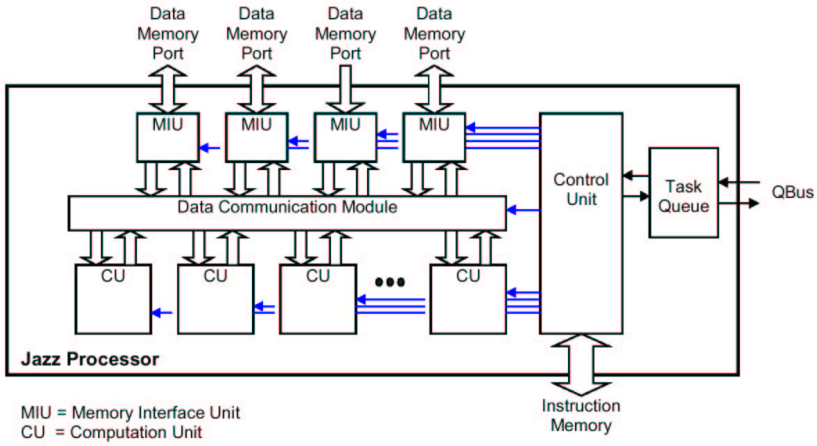


Figure 60 Modèle d'architecture du processeur JAZZ

Le modèle d'architecture du processeur JAZZ (Figure 60) est assez représentatif de ce que peut apporter une architecture VLIW configurable : un nombre d'unités de calcul variable et de natures différentes dont certaines très spécialisées permettant de s'adapter aux spécificités du parallélisme intrinsèque de l'application, et une bande passante mémoire adaptable aux besoins de l'application qui offrent des performances proches de celles des structures très spécialisées (ASIC ou ASIP) sans le surcoût de matériel lié à l'usage d'une architecture « universelle » et figée. Pour connaître le succès, ce type de processeur doit être obligatoirement associé à une méthodologie de conception basée sur des outils logiciels reciblables et indépendants de la technologie, qui permet de rendre raisonnable le temps de conception. C'est dans cette optique que nous proposons dans les deux prochains chapitres un modèle de cœur de processeur VLIW configurable destiné à la conception d'ASIP ainsi que la méthodologie de conception associée.

Chapitre 3

Modèle proposé de processeur VLIW configurable pour systèmes embarqués

Sommaire :

3.1 Introduction

Nous présentons dans ce chapitre l'architecture interne et le jeu d'instructions d'un modèle de cœur de processeur VLIW configurable destiné à la conception d'ASIP pour systèmes embarqués. Concevoir un cœur de processeur DSP destiné à l'intégration dans un SoC passe en premier lieu par la recherche d'une architecture et d'un jeu d'instructions adaptés aux besoins en puissance de calcul du domaine d'application visé. Comme nous l'avons étudié au chapitre précédent, il existe plusieurs manières d'optimiser une architecture en fonction d'une ou plusieurs applications, les deux principales étant l'extension et la spécialisation des ressources matérielles et du jeu d'instructions. Les contraintes de plus en plus fortes liées au temps de développement des circuits plaident en faveur de l'utilisation de modèles d'architecture configurables qui permettent d'accélérer la conception de cœurs de processeurs spécialisés. Afin de « coller » au plus près des contraintes du système cible, nous proposons un modèle de processeur VLIW hautement configurable associé à un jeu d'instructions générique lui-aussi configurable, et qui une fois configurés doivent permettre d'obtenir un processeur ASIP offrant le niveau de performance requis par les applications cibles tout en minimisant les contraintes matérielles de surface et/ou de consommation du système global. L'architecture générale du processeur ainsi que les trois principales unités matérielles la composant (unité de calcul, unité d'adressage et unité de contrôle) et leurs paramètres de configuration sont décrits en détail dans les quatre prochaines sections. La dernière section présente le jeu d'instruction VLIW générique associé au modèle du processeur.

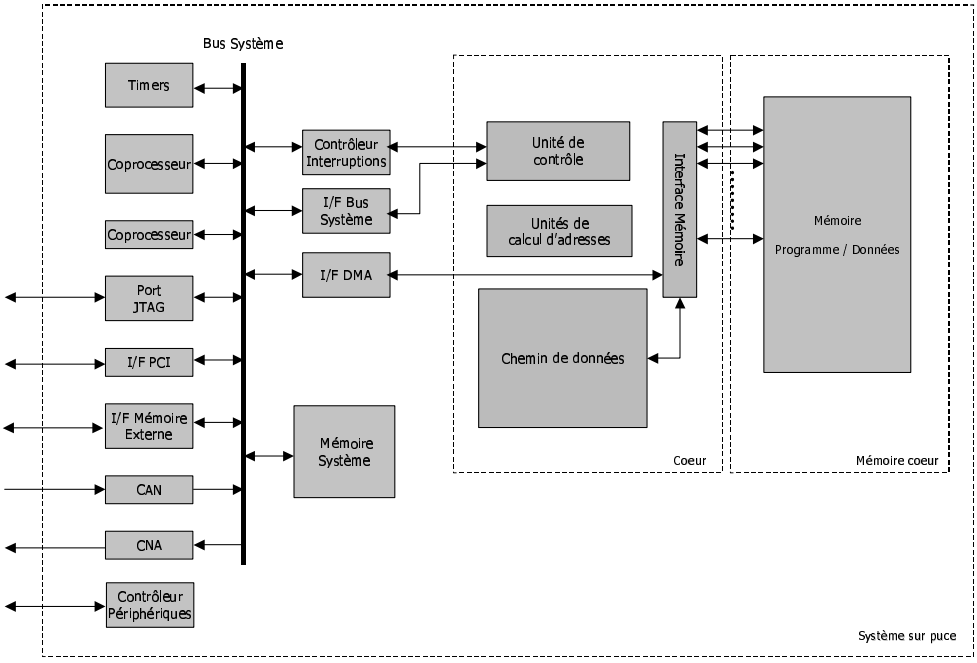


Figure 61 Intégration du cœur dans une architecture de type SoC

3.2 Architecture générale

Il est important de préciser que l'objet de la présente étude porte sur l'aspect « unité de calcul intensif » du processeur, et son aptitude à accélérer les traitements DSP des applications. Les autres aspects qui concernent le traitement de taches de contrôle et l'interfaçage avec le reste du système, tout aussi complexes, dépassent le cadre de cette étude et ne seront donc pas abordés ici. Un exemple

d'architecture de *SoC* typique des circuits DSP complexes est illustré Figure 61. Le transfert de données et d'informations de contrôle entre le processeur et le bus système, la communication et la configuration du processeur par le port hôte du circuit, le contrôle des interruptions et les mécanismes de DMA sont autant de problèmes supplémentaires à prendre en compte par les concepteurs du système embarqué. Le cœur de processeur étant destiné à l'intégration dans une grande variété d'environnement différents, son interface ne peut dépendre d'une implémentation particulière et devra donc être générique. Les mécanismes de contrôle dépendant de l'environnement extérieur (interface DMA, contrôleur d'interruptions) devront être intégrés sous forme de blocs matériels extérieurs au cœur et connectés via l'interface générique. C'est cette interface générique qui manque actuellement à la description de notre modèle de processeur et qui devra être spécifiée en vue de l'intégration dans des systèmes embarqués. Elle n'a cependant aucune influence sur le comportement en mode calcul intensif du processeur, si ce n'est en ce qui concerne l'accès aux données qui exige une bande passante suffisante pour nourrir le cœur. Les chiffres de performance présentées par la suite supposent donc un accès aux données externes sans goulot d'étranglement..

Le cœur de l'ASIP est constitué de quatre unités matérielles principales capables de fonctionner en parallèle et communiquant entre elles par un réseau de bus d'échanges. Ces unités sont l'unité de calcul (CU), l'unité de calcul d'adresses (AGU), l'unité de contrôle (PCU) et l'interface mémoire (DMU). L'unité d'exécution prend en charge toutes les opérations arithmétiques utilisées pour le traitement des données (multiplication, addition, opérations logiques, décalages). Elle est composée de plusieurs unités fonctionnelles (UF) et d'un banc de registres central pour la mémorisation des données. L'unité de calcul d'adresses est chargée de calculer les adresses des données devant être lues ou écrites en mémoire. L'unité de contrôle gère le flot d'instructions du programme, calcule l'adresse de la prochaine instruction, décode l'instruction courante et prend en charge les interruptions. Enfin, l'unité d'interface mémoire est utilisée par toutes les autres unités pour accéder au système mémoire du processeur, que ce soit pour des données ou des instructions.

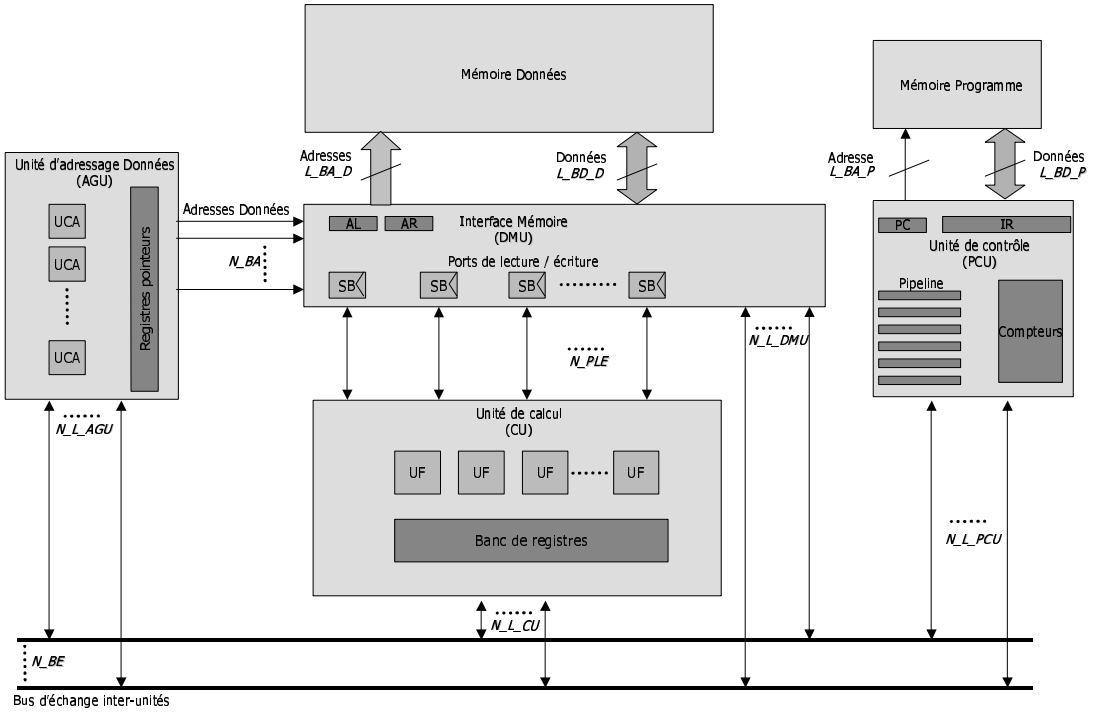


Figure 62 Vue générale de l'architecture du processeur

Afin de s'adapter au mieux aux besoins spécifiques en bande passante des applications, le nombre de connexions entre les différents blocs est configurable (Figure 63). Ces connexions sont de deux types. Des liaisons point à point constitués de simple « fils » de routage relient les blocs entre lesquels le trafic de données est supposé important : Interface mémoire / Unité de calcul, Unité AGU / Interface Mémoire, etc. En supplément, des bus d'échanges inter-unités autorisent des transferts de données entre blocs distants non reliés par des connexions directes. Ces bus, dont le taux d'utilisation est supposé plus faible que celui des liaisons directes, sont avant tout destinés à être utilisés par des instructions d'initialisations, de transferts de registres (par exemple entre les registres d'adresses de l'AGU et le banc de registres de l'unité CU) ou de sauvegarde de contexte.

Paramètres	***Nom***
Ports de lecture/écriture	N_PLE
Bus d' échanges	N_BE
Bus d'adresses Données	N_BA (=N_PLE)
Liaisons AGU / bus d'échanges	N_L_AGU
Liaisons CU / bus d'échanges	N_L_CU
Liaisons DMU / bus d'échanges	N_L_DMU
Liaisons PCU / bus d'échanges	N_L_PCU

Figure 63 Paramètres du modèle : connexions inter-unités

3.3 Chemin de données

Les principales exigences liées au chemin de données concernent à la fois la possibilité d'étendre ou de réduire le matériel embarqué pour s'adapter aux caractéristiques de l'application (unités de calcul,

registres, connectivité), et aussi la possibilité d'intégrer des unités fonctionnelles utilisateurs permettant de spécialiser l'architecture pour un type de traitement particulier. De plus, l'architecture doit être suffisamment homogène et régulière pour permettre une bonne adéquation avec un compilateur afin de produire du code de qualité. La solution la plus employée actuellement dans le domaine des processeurs DSP pour répondre à l'ensemble de ces contraintes réside dans le choix d'une architecture de chemin de données de type VLIW. Bien que le terme VLIW désigne avant tout la structure d'un jeu d'instructions, il caractérise aussi un type particulier d'architecture basée sur l'emploi d'unités fonctionnelles multiples réparties « horizontalement » et fonctionnant simultanément, et dans laquelle la mémorisation des données se fait au moyen de larges bancs de registres multi-ports. Il implique aussi une bande passante mémoire très large, permettant à la fois le chargement des instructions VLIW et l'alimentation en données des nombreuses unités fonctionnelles. Du fait du grand nombre de données circulant à chaque cycle entre les différents éléments matériels, la connectivité est importante et utilise des routeurs/multiplexeurs de données possédant de nombreux ports d'entrée et de sorties. L'architecture que nous proposons pour le chemin de données repose sur ce concept d'architecture VLIW extensible et spécialisable, dans laquelle de nombreux paramètres peuvent être ajustés pour s'adapter aux exigences de l'application (Figure 64).

La largeur de l'ensemble des éléments matériels est déterminée par un paramètre particulier de l'architecture qui définit le nombre de bits de base d'un mot de données (paramètre N_DATA_BITS). Notre architecture supporte les calculs en double précision massivement utilisés en traitement du signal. Les connexions entre éléments, registres et unités fonctionnelles peuvent donc être de largeur *N_DATA_BITS* ou *2*N_DATA_BITS*. Cette caractéristique est commune à l'ensemble des processeurs DSP qui supportent tous la double précision, les valeurs les plus courantes étant 16/32 bits et 24/48 bits. Nous ne nous sommes pas intéressés ici au problème des bits de garde utilisés dans certains processeurs pour gérer la précision. En effet, l'application choisie comme test (l'algorithme d'encodage de voix du GSM) pour évaluer les performances de notre modèle ne nécessite pas de gestion des bits de garde, la précision étant assurée par l'intégration directe dans les unités fonctionnelles de fonction de saturation et de mise à l'échelle. On peut penser que cette solution peut être facilement utilisée pour tous les algorithmes et a l'avantage de simplifier à la fois la structure matérielle du chemin de données mais aussi la définition du jeu d'instructions.

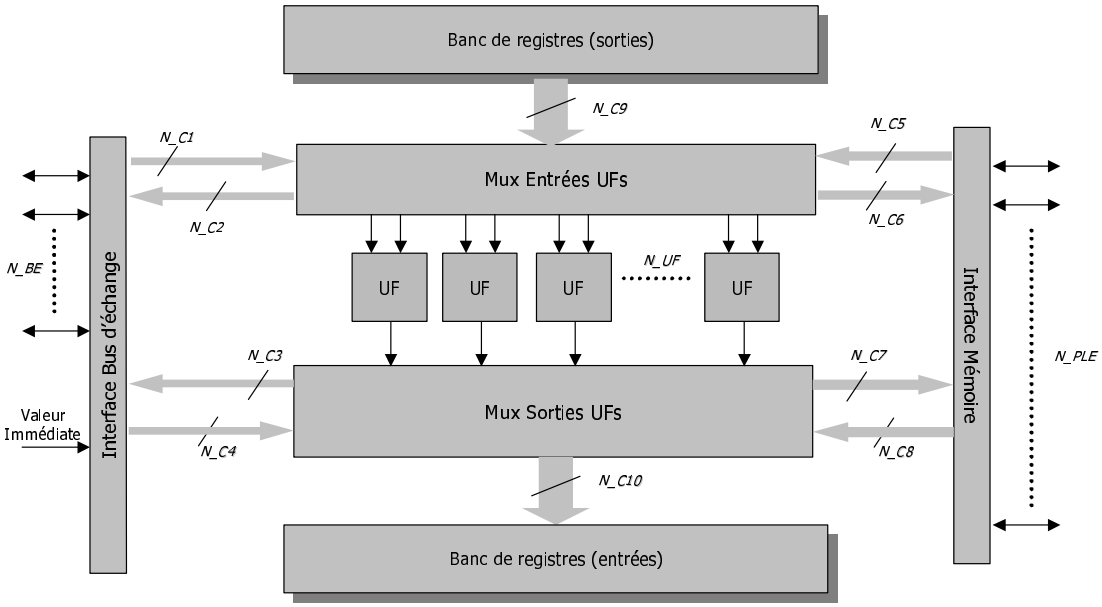


Figure 64 Architecture du chemin de données

3.3.1 Unités fonctionnelles

La performance d'une architecture particulière pour une application donnée dépend pour beaucoup du nombre et du type des UFs intégrées au chemin de données. Le modèle d'ASIP que nous proposons prévoit la possibilité d'intégration d'unités fonctionnelles générales (ALU, MAC, BMU) ou spécialisées, chacune des unités pouvant être dupliquées plusieurs fois pour augmenter la capacité de traitement du processeur.

Les UFs sont modélisées sous forme de boites noires auxquelles sont attachées une interface définissant les signaux d'entrée/sortie et de contrôle, et un ensemble de mnémoniques décrivant les opérations effectuées par l'UF. La plupart du temps, les unités utilisées sont des blocs purement combinatoires répondant en un cycle et dont le comportement à un instant t est entièrement spécifié par la mnémonique de commande. On peut cependant envisager que certaines applications requièrent l'usage d'unités spécialisées complexes mêlant blocs combinatoires et registres d'état, et répondant en plus d'un cycle. Dans ce cas, la règle est que les registres internes à l'UF doivent être invisibles du point de vue jeu d'instructions : l'écriture, la lecture et le test d'éventuels registres ou drapeaux devra se faire au moyen de mnémoniques dédiées, les registres ne pouvant être accédés par les mnémoniques générales de transfert de données (de type *MOV*). Cette restriction a pour but de spécifier de manière claire la façon dont le jeu d'instructions accède aux UFs, ce qui permet de définir une structure de jeu d'instructions générique et indépendante des UFs et des éventuels registres spécialisés qui leur sont attachés (cf. 3.6). Elle permet aussi de simplifier et de rendre plus performante la génération de code, la prise en compte de bits de contrôle statiques influant sur le comportement des opérateurs est en effet source de grandes difficultés pour un compilateur.

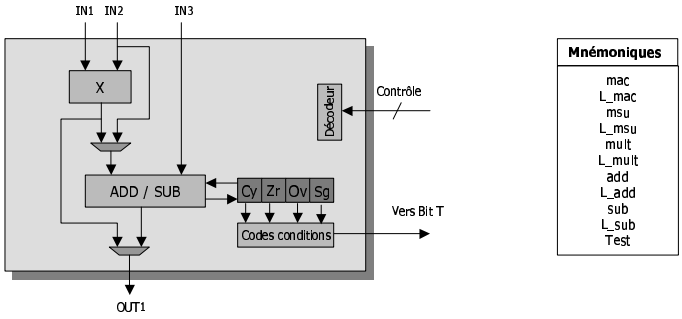


Figure 65 Structure et mnémoniques d'une unité MAC

L'interface d'une unité peut aussi comporter, en plus des entrées/sorties de données et du mot de commande, une sortie spéciale dite « Bit T » (cf. 3.6.5.2) qui permet d'exporter le calcul d'une certaine condition logique vers le bit de condition de l'unité de contrôle. Le calcul de la condition se fait en interne dans l'unité et peut porter sur la valeur des éventuels registres internes, ce qui permet de tester leur contenu et de conditionner les instructions suivantes en fonction de leur valeur.

La Figure 65 présente l'exemple d'une unité MAC capable d'effectuer des opérations de multiplication-accumulation et accumulation-soustraction, ainsi que de simples soustractions et additions, ces opérations travaillant sur des données en simple ou double précision (mnémoniques commençant par le préfixe « *L_* »). L'interface comporte trois entrées et une sortie pour les données sources et résultat, et une entrée pour le mot de contrôle codant l'opération à effectuer. Les valeurs des drapeaux sont utilisées par un bloc combinatoire calculant la valeur des codes conditions de l'unité : l'instruction « Test(reg1, reg2, >=) » effectue la soustraction des deux opérandes sources de type registres, et utilise la valeur des deux drapeaux « Signe » (*Sg*) et « Zéro » (*Zr*) pour calculer la valeur de la condition « >= », qui est alors exporté vers le bit T.

Dans les jeux d'instruction de nombreux DSP, certaines combinaisons d'opérations sont interdites, traduisant le fait que les UFs associées à ces opérations ne peuvent fonctionner en parallèle. Dans le processeur CARMEL par exemple, une opération de calcul d'exposant ne peut se faire en même temps 'une opération de décalage, les UFs « Shifter » et « Exponent » étant disposées dans le chemin de données de manière « verticale » : leurs entrées et sorties empruntent les mêmes signaux d'interface. Dès lors, ces deux unités fonctionnelles peuvent être vues comme une seule super-unité dont l'interface et les mnémoniques combinent les signaux et mnémoniques des deux unités de base. L'arrangement « vertical » ou « horizontal » des unités fonctionnelles en fonction des besoins de l'application peut permettre de grandes économies matérielles, à travers la réduction de la connectivité lié à la réduction des interfaces, et grâce à la diminution de la largeur du mot d'instruction. Le choix du CARMEL, que l'on retrouve dans de nombreux autres processeurs, provient de l'analyse du flot de données des principaux noyaux de calculs DSP, de laquelle il découle que les opérations de décalage et de normalisation nécessitent rarement d'être effectuées en même temps.

Dans notre modèle, l'arrangement vertical de plusieurs unités fonctionnelles en une seule macro-unité est très simple, et consiste juste à définir une interface commune, et à déclarer dans la table des mnémoniques globale l'ensemble des mnémoniques des sous-unités.

3.3.2 Banc de registres

L'architecture du modèle repose sur l'utilisation d'un banc de registres central utilisé pour le stockage temporaire des variables de calcul. L'utilisation d'un banc de registres homogène multi-ports facilite la programmation et le travail du compilateur en éliminant les contraintes de connectivité existant dans les chemins de données à jeu de registres hétérogène. Cette solution est cependant très lourde en terme de coût matériel. Il est donc indispensable de réduire au minimum le nombre de registres du banc ainsi que le nombre de ports d'entrée/sortie.

Le banc de registres est configurable en largeur de mot, en nombre de registres et en nombre de ports d'entrée/sortie. Il permet l'accès à des données en simple ou double précision, les mots en double précision étant rangés dans deux registres successifs alignés sur une adresse paire. Le nombre de ports d'entrée et de sortie sera déterminé selon les besoins de l'application en terme de nombre de lecture et d'écriture maximum par instruction. Les multiplexeurs en entrée et en sortie des UFs redirigent les données lues ou écrites en fonction de leur émetteur/destinataire. Là encore, le nombre de combinaisons différentes de routage autorisées par les multiplexeurs peut être réduit en examinant les besoins en routage pour les parties critiques de l'application.

3.3.3 Connectivité

La connectivité du modèle du chemin de données « complet » prévoit tous les cas de figure possibles en terme de circulation des données entre les différents composants. Chaque flèche du schéma de la Figure 64 comme celle allant de la sortie du banc de registres au multiplexeur d'entrée des UFs constitue un bus de largeur *M * N_DATA_BITS*, *M* étant la largeur physique du bus en nombre de données. La largeur des nappes de fils et surtout la complexité des réseaux de multiplexeurs routant les données entre les éléments peut rapidement devenir problématique en terme de surface de silicium occupée et surtout en terme de performance, du fait du grand nombre de couches combinatoires à traverser. Il est donc indispensable d'essayer de réduire au maximum la largeur des liaisons entre éléments et même de supprimer certaines liaisons non indispensables. Par exemple, la liaison reliant la sortie des UFs au multiplexeur du bus d'échange n'est pas physiquement indispensable, puisque les données peuvent transiter temporairement par le banc de registres avant de ressortir vers le bus d'échange. La suppression de ce lien aura pour conséquence principale que les transfert de données UFs / Bus d'échange devront être décomposés en deux transferts séquentiels. Cependant, si ce type de transfert n'est pas utilisé dans les boucles critiques, sa suppression permettra une économie matérielle substantielle et une amélioration de la performance liée au raccourcissement de la chaîne combinatoire.

L'ensemble des paramètres du modèle pour le chemin de données est décrit sur la Figure 66, les valeurs *N_Cx* représentant la largeur en nombre de mots simple précision (*N_DATA_BITS* bits de large) de la liaison.

Paramètres	*Nom*
Largeur de donnée simple précision	N_DATA_BITS
Nombre et type d'UFs	N_UF
Nombre de registres	N_Reg
Ports de lecture Registres	N_Reg_PL
Ports d'écriture Registres	N_Reg_PE
Connectivité des liaisons inter-éléments	N_C1, N_C2, ...N_C10

Figure 66 Paramètres du chemin de données

3.4 Unité d'adressage et interface mémoire

En traitement du signal, la plupart des traitements manipulent les données sous forme de flux souvent représentés sous forme de tableaux. Avant même d'effectuer des calculs sur les données proprement dites, le processeur doit d'abord déterminer leur adresse en mémoire. Le calcul des adresses dans une unité matérielle dédiée permet de libérer l'unité CU qui peut ainsi travailler en parallèle sur d'autres données déjà lues. C'est pourquoi l'ensemble des processeurs DSP intègrent des unités spécialisées souvent appelées unités AGU (pour *Address Generation Unit*). Cette unité propage les adresses calculées vers l'interface mémoire Donnée (DMU ou *Data Memory Unit*), chargée de communiquer avec le système mémoire physique et de renvoyer les données correspondantes.

3.4.1 Hypothèses et contraintes sur le système mémoire

La spécification du cœur d'ASIP ne présume pas des caractéristiques du système mémoire réel qui lui est attaché dans une réalisation de type SoC. Il existe en effet de nombreuses solutions d'implémentation, souvent issues de compromis coût matériel / performance / consommation : bancs mémoire simple ou double ports, RAM ou ROM, SRAM ou DRAM, partitionnement du système en blocs de tailles différentes, etc. Si le système est très sensible au coût, il est par exemple préférable de privilégier une implémentation à base de bancs mémoire simple ports, moins coûteuse en surface (40% d'écart entre les blocs simple et double ports du CARMEL [54]).

Pour fonctionner, le modèle d'ASIP impose néanmoins certaines contraintes au système mémoire formé par le bloc d'interface (unité DMU) et la mémoire physique :

- Le système mémoire est séparé en deux sous-systèmes « Programme » et « Données ». L'accès aux instructions par l'unité de contrôle se fait de façon directe par l'utilisation d'un bus d'adresses et d'un bus de données dédiés, tandis que l'accès aux données passe par l'intermédiaire de l'unité DMU. Le choix de séparer l'espace en deux plutôt que d'utiliser une structure mémoire unifiée se justifie par le fait que la taille des bus données et adresses pour les espaces « Programme » et « Données » peuvent être très différentes. En effet, dans un processeur ASIP, la volonté de spécialiser le matériel et le jeu d'instructions a pour but de réduire au minimum la largeur d'encodage des instructions mais aussi la taille de la mémoire programme. Un ASIP travaillant sur des données de 16/32 bits mais dont la largeur d'instructions optimale est de 19 bits aura donc beaucoup de mal à s'accommoder d'un espace mémoire unifié dont la largeur du bus de données serait de 16 ou 19 bits. Le problème est identique pour la largeur des bus d'adresses puisque les espaces mémoire et programme n'ont

pas les mêmes besoins en nombre de mots adressables. Dès lors, la séparation des espaces mémoires permet d'optimiser chaque partie indépendamment l'une de l'autre.

- Les accès données peuvent se faire en simple ou double précision. La taille des bus de données correspondant à la largeur en bits d'un mot simple précision, un accès double précision nécessitera l'utilisation simultanée de deux bus de données. Si le système fonctionne sur le principe des adresses alignées (dans notre cas des multiples de 2 pour des accès double précision), les outils de génération de code devront prendre en compte ces contraintes pour générer l'image mémoire adéquate.

- L'interface mémoire gèle le cœur en cas de conflits d'accès mémoire, lorsque plusieurs requêtes ne peuvent être servies dans le même cycle. C'est par exemple le cas lorsqu'on tente d'accéder à deux lignes différentes d'un bloc de mémoire de type SRAM double port. L'interface mémoire doit être capable de détecter ce type de conflit. Pour cela, elle doit détenir des informations sur les caractéristiques physiques de la mémoire embarquée, et est donc forcément dépendante de l'implémentation. Lorsqu'un conflit est détecté, elle génère un signal activant le gèle du processeur jusqu'à ce que toutes les requêtes aient été servies.

La bande passante requise est d'une instruction par cycle pour le sous-système « Programme » et de *N_PLE* données par cycle pour le sous-système « Données ». Le calcul des adresses est effectué dans l'unité PCU pour les instructions et dans l'unité AGU pour les données.

Paramètres	***Nom***
Largeur bus de données « Données »	L_BD_D (=N_DATA_BITS)
Largeur bus d'adresses « Données"	L_BA_D
Largeur bus de données « Programme »	L_BD_P
Largeur bus d'adresses "Programme"	L_BA_P

Figure 67 Paramètres du système mémoire

3.4.2 Unité d'adressage Données

L'architecture présentée Figure 68 est semblable à celle que l'on retrouve dans la plupart des unités de génération d'adresses pour DSP. Elle repose sur un ensemble de registres pointeurs mémorisant les adresses des tableaux de données, et sur des blocs combinatoires chargés des calculs d'adresse : les *UCAs* (pour *Unité de Calcul d'Adresses*). Les fonctions remplies par ces unités dépendent des modes d'adressage définis dans le jeu d'instructions, qui seront présentés en détail en section 3.6. Dans la majorité des processeurs DSP, les *UCAs* sont capables des calculs suivants :

- Addition / soustraction de deux opérandes (registres, immédiat) utilisé dans les modes d'adressages indexés ou post-incrémentés

- Gestion de l'adressage modulo pour la gestion des tampons circulaires. Ce type d'adressage nécessite l'usage de registres spéciaux supplémentaires (index, base, modulo) ainsi que l'ajout dans l'*UCA* de la logique combinatoire calculant le modulo (cf. section 1.2.3). Suivant le même principe que pour les unités fonctionnelles du chemin de données, les registres

utilisés pour l'adressage modulo sont considérés comme des registres spécialisés internes aux *UCAs* et accessibles au moyen d'instructions dédiées.

- Gestion de l'adressage Bit-Reverse utilisé pour les calculs FFT. La fonctionnalité est en générale réalisée sous forme de bloc combinatoire intégré dans une *UCA*.

En fonction des besoins de l'application, les différentes *UCAs* peuvent inclure des fonctionnalités différentes. Il n'est en général pas utile d'inclure les fonctionnalités modulo ou Bit-Reverse dans toutes les *UCAs*.

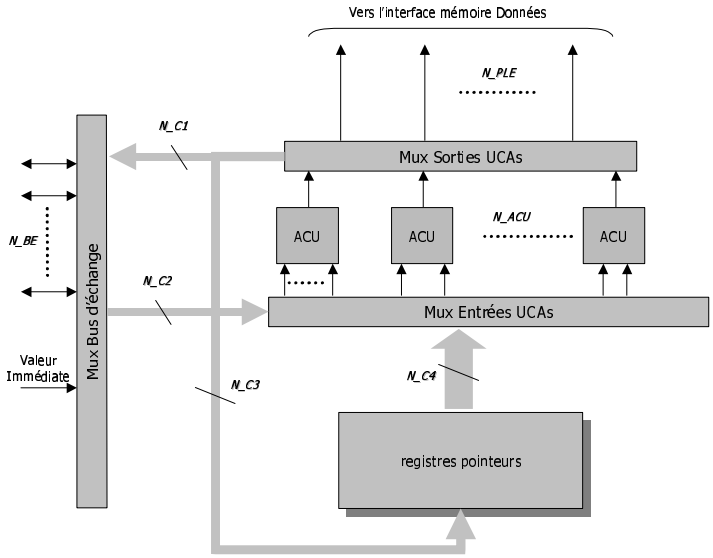


Figure 68 Architecture de l'unité d'adressage

Le bus d'entrée « N_C2 » permet d'acheminer des données de type « immédiat » ou provenant d'autres unités via les bus d'échanges, et de les utiliser comme opérandes source des calculs d'adresses. Cette capacité est surtout intéressante en ce qui concerne les accès aux registres compteurs de l'unité PCU, qui peuvent très souvent servir d'index pour les calculs d'adresses. En effet, en traitement du signal, on constate qu'une grande majorité des accès mémoires sont effectués dans des boucles logicielles et que l'adresse de ces accès est souvent de la forme « adresse de base du tableau +/- indice de boucle ». C'est le cas par exemple des variables *h* et *x* du filtre FIR Figure 69. L'unité PCU utilisant des compteurs de boucle pour implémenter les mécanismes de boucle « zéro-cycles », on a tout intérêt à réutiliser la valeur de ces compteurs comme index pour les calculs d'adresses. Dans le cas du FIR, l'adresse de la variable *x(i)* peut être calculée comme égale à « *p0+i* », i étant le compteur de boucle et p0 contant la valeur de base du tableau.

$$y = \sum_{i=0}^{N-1} x(i) * h(N-i)$$

Figure 69 Equation du filtre FIR

Par rapport à une implémentation à l'aide de l'adressage modulo, cette méthode dispense de l'utilisation des registres spéciaux (*index, modulo et base)* : les valeurs de l'index et de la longueur sont déjà mémorisées dans les registres de boucle de l'unité PCU, et l'adresse de base peut être

mémorisé dans un simple registre pointeur. La logique combinatoire destinée au calcul du modulo devient elle aussi inutile. Ce type d'adressage peut donc permettre de faire l'économie de plusieurs registres et de matériel. Il a aussi le mérite de rendre le code assembleur plus lisible et de faciliter la production de code par le compilateur, qui a juste à substituer à la variable de boucle le registre compteur correspondant, sans avoir à s'occuper d'initialiser les différents registres spéciaux de l'adressage modulo.

Il faut cependant noter que ce type d'adressage n'offre pas autant de possibilités que l'adressage modulo, notamment lorsqu'il s'agit d'implémenter des structures de données de type file d'attente ou FIFO. Selon l'application, il peut donc être nécessaire de conserver la capacité d'adressage modulo, par exemple dans une seule des *UCAs*. Dans ce cas, le jeu d'instructions devra inclure des instructions supplémentaires permettant d'accéder aux registres locaux de l'*UCA* concernée (registres *Longueur* , *Adresse de Base* et « Flag » d'activation du mode modulo). Un exemple de ce cas de figure est présenté au chapitre 6 (section 5.5.1).

Contrairement à certains processeurs, on ne prévoit pas de registre spécial « SP » dédié à la pile logicielle. C'est au concepteur de choisir un registre pointeur particulier du banc de registres AGU pour matérialiser la pile. Ce paramètre devra être pris en compte au moment de la synthèse de l'unité de contrôle (PCU), car les instructions *CALL* et *RTS* du processeur (cf. 3.6.4) utilisent la pile pour sauvegarder/restaurer l'adresse de retour d'un appel de sous programme. Il devra aussi être indiqué au compilateur afin qu'il sache quel registre utiliser pour manipuler la pile.

Comme dans le chemin de données, le nombre d'*UCAs* ainsi que la connectivité des données entre les éléments sont configurables pour permettre de s'adapter au plus près aux contraintes de l'application, le concepteur ayant tout intérêt à réduire ces paramètres au minimum pour diminuer l'impact sur le coût matériel et la longueur de la chaîne critique. La Figure 70 résume les différents paramètres de l'AGU.

Paramètres	***Nom***
Nombre et type d'unités UCA	N_UCA
Nombre Registres Pointeurs	N_Preg
Nombre Ports lecture Registres	N_Preg_PL
Nombre Ports écriture Registres	N_Preg_PE
Connectivité	N_C1, N_C2, ..., N_C4
Registre de Pile	SP

Figure 70 Paramètres de l'unité d'adressage

3.5 Unité de contrôle

L'unité de contrôle est chargée de deux tâches principales : la gestion du flot d'instructions au travers du calcul de l'adresse de l'instruction suivante, et le décodage du mot d'instruction et sa répartition vers les différentes unités matérielles du processeur. Pour permettre de bonnes performances matérielles, l'exécution d'une instruction suit un pipeline à six étages dans lequel chaque étage correspond à l'exécution d'une partie de l'instruction. L'unité comprend aussi des mécanismes

matériels dédiés à l'implémentation des boucles « zéro-cycle » pour améliorer le comportement du processeur dans les boucles de calcul critiques.

La présentation qui suit s'attache avant tout à présenter les caractéristiques nécessaires à de bonnes performances pour des applications de type traitement du signal. Des aspects plus généraux concernant la gestion des interruptions et des exceptions, ou l'intégration de fonctionnalités évoluées de type « contrôle » comme la gestion du multi-thread ou l'adressage relatif au CP, dépassent le cadre de cette étude et ne seront pas abordés ici. En effet ces fonctionnalités, dont certaines sont indispensables pour le bon fonctionnement d'un processeur (gestion des interruptions) et d'autres plus optionnelles, ajoutent de la complexité au processeur mais n'ont pas d'influence radicale sur les choix architecturaux et les performances finales liés à la spécialisation pour une application DSP, qui sont l'objet de cette étude.

3.5.1 Gestion du flot d'instruction

L'unité autorise les principaux modes de séquencement que l'on retrouve dans l'ensemble des processeurs: exécution séquentielle, sauts inconditionnels et conditionnels, appels et retours de fonctions, retours d'interruption, etc. La structure de l'unité présenté Figure 71 peut être séparée en trois blocs distincts. Le bloc 1 contenant le compteur de programme (CP) est chargé du calcul de l'adresse de l'instruction suivante, après analyse du type de séquencement spécifié dans l'opcode de l'instruction courante. En fonction de la valeur de ses entrées, le bloc de calcul du CP suivant sélectionne la bonne valeur à envoyer sur le bus d'adresses de la mémoire programme. L'adresse peut prendre six valeurs différentes : le CP incrémenté de 1, une valeur immédiate, une valeur provenant des bus d'échanges (par exemple un registre d'une autre unité d'exécution), une adresse fournie par le contrôleur d'interruptions, une adresse provenant de la pile LSA (*Loop Start Address*) et une de la pile CP. Cette dernière est utilisée pour stocker les valeurs des adresses de retour lors de branchements à des sous-programmes ou à des requêtes d'interruption, tandis que la pile LSA sert à mémoriser les adresses de début de boucle pour les boucles « zéro-cycle ».

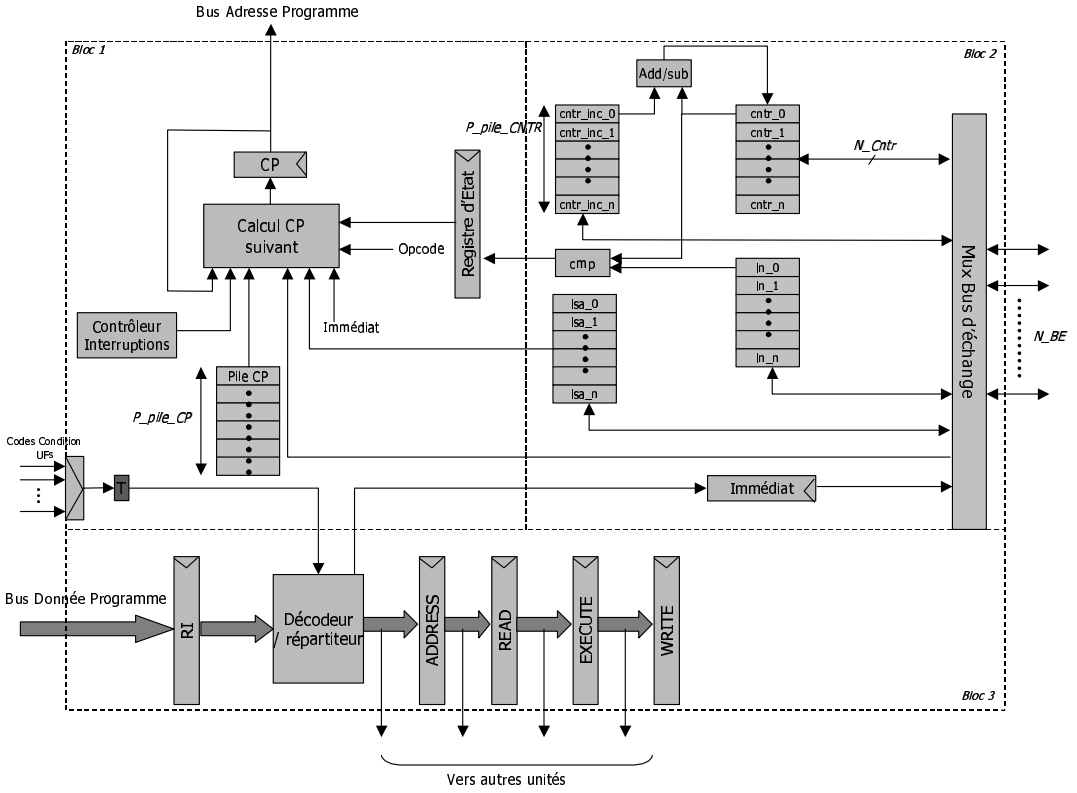


Figure 71 Architecture de l'unité de contrôle (PCU)

Le bloc 2 contient l'ensemble des éléments matériels implémentant les boucles « zéro-cycle ». On rappelle que ce mécanisme est utilisé pour exécuter des boucles logicielles de type *For* ou *While* en évitant les pertes en cycles dues aux opérations de fin de boucle : incrémentation/décrémentation du compteur, comparaison par rapport à la valeur finale et branchement conditionnel en début de boucle. Ces opérations sont exécutées matériellement en parallèle avec les instructions « utiles » de la boucle. Pour ce faire, l'unité PCU dispose de quatre piles mémorisant respectivement la valeurs courante du compteur de boucle (*cntr_x*), la valeur de fin correspondant à la condition d'arrêt (*cntr_inc_x*), la valeur d'incrément du compteur (*ln_x*) et l'adresse de début de boucle (*lsa_x*). Des piles sont utilisées pour permettre de mémoriser ces valeurs pour plusieurs niveaux de boucles imbriquées, la profondeur de la pile équivalant au nombre maximum de boucles imbriquées autorisé. Il et cependant toujours possible d'obtenir des niveaux de boucle supplémentaires en les programmant logiciellement comme on le ferait sur un processeur RISC classique. Un additionneur/soustracteur est utilisé pour incrémenter le compteur à chaque occurrence de la boucle. Un comparateur permet de détecter la fin de boucle et de repasser en mode séquentiel. La pile mémorisant la valeur des compteurs courants dispose de plusieurs pointeurs de lecture (*N_Cntr*) pour pouvoir exporter ces valeurs vers l'unité AGU qui les utilisera comme index pour les adressages indirects (cf. 3.4.2).

On remarquera que contrairement à la plupart des autres processeurs DSP, il n'est pas nécessaire de disposer d'une cinquième pile mémorisant les adresses de fin de boucle et d'un comparateur de détection de fin de boucle. Un autre système basé sur des marqueurs de fin de boucle encodés directement dans le mot d'instruction permettent au processeur de localiser la fin de boucle et d'anticiper les opérations de modification du compteur et de branchement en début de boucle (cf. 3.6).

Le registre d'état mémorise l'état courant du processeur sous forme d'un ensemble de champs de bits de largeurs et de significations différentes. Dans les processeurs DSP, les registres d'états sont souvent relativement larges à cause des nombreux bits de contrôle statiques influant sur le comportement des unités fonctionnelles (bits définissant le type d'arithmétique utilisée, le nombre de décalage à appliquer à la sortie d'une UF pour la gestion automatique de la précision, etc.). Comme nous avons fait le choix d'interdire l'utilisation de ce type de bits pour rendre générique le jeu d'instructions et faciliter la génération de code, les informations contenues dans le registre d'état sont beaucoup moins nombreuses : un ensemble de bits *Enable_Loop_Flag_*x définissant l'état des boucles matérielles (actives ou inactives), et un bit spécial (bit *T*) utilisé comme bit de condition pour les opérations conditionnelles.

3.5.2 Pipeline de contrôle

Le modèle de processeur proposé repose sur une structure de contrôle à six étages de pipeline, dénommés respectivement F (*Fetch*), D (*Decode*), A (*Address*), R (*Read*), E (*Execute*), et W (*Write*). Ce choix de 6 étages, que l'on pourrait qualifier d'étages « logiques », ne correspond évidemment en rien à une implémentation physique définitive puisque l'on parle d'un modèle de processeur et non d'un circuit figé. Il est le résultat d'un certain découpage fonctionnel des instructions en micro-opérations exécutées séquentiellement dans chaque étage du pipeline. Ce découpage a cependant l'ambition d'être réaliste et relativement équilibré, puisqu'il se base sur l'analyse des structures de pipeline de nombreux processeurs DSP existants. La réalisation physique correspondant à un ASIP particulier et à une certaine technologie pourra éventuellement nécessiter un rééquilibrage de la structure pipeline en subdivisant les étages les plus longs en étages plus courts. De tels changements nécessiteront certaines modifications au niveau de l'implémentation matérielle de l'unité PCU mais ne devraient pas altérer la nature du jeu d'instructions et des autres unités d'exécution. Les chiffres de performances présentés dans la suite de cette étude, en particulier ceux issus de la simulation cycle précis des algorithmes grâce à un simulateur d'instruction, se basent sur un modèle à six étages. Le rallongement de la structure de pipeline a une influence négative sur le comportement dynamique des instructions de rupture du flot d'instructions (saut, appels de sous-programmes, etc.) en augmentant le nombre de cycle nécessaires à leur exécution, et peuvent donc occasionner des pertes en performance pour des applications où ce type d'instructions est massivement utilisé. Notre étude portant sur l'analyse de fonctions DSP ayant peu de structures de contrôle, et tenant compte du fait que le jeu d'instructions permet par des mécanismes d'instructions retardées et d'exécution conditionnelle de masquer les effets les plus indésirables du pipeline, on peut cependant estimer que les chiffres de performance obtenus sont réalistes et très peu dépendants de la longueur du pipeline.

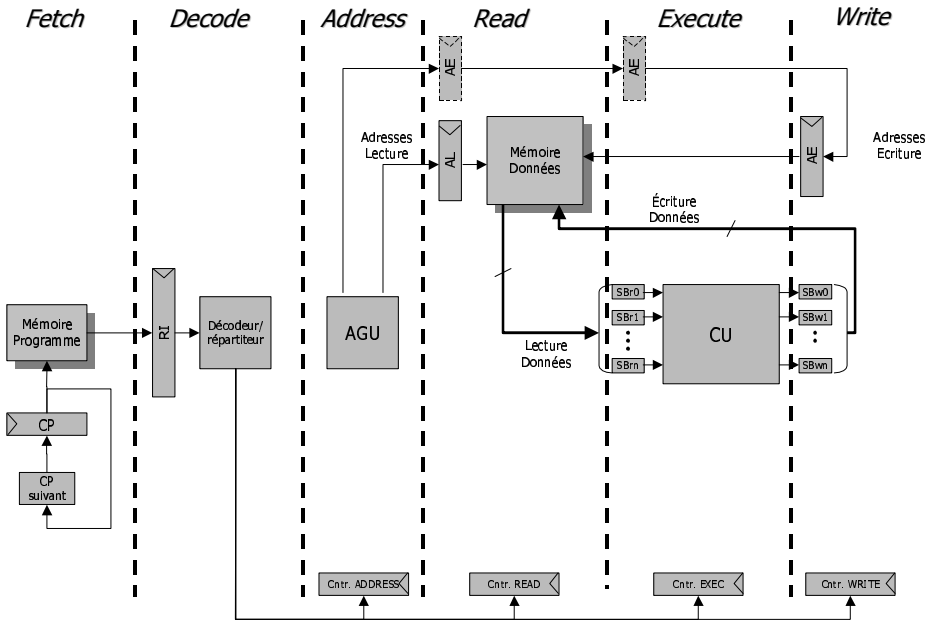


Figure 72 Pipeline de contrôle

3.5.2.1 Architecture à accès mémoire direct

La Figure 73 résume les principales opérations effectuées dans chaque étage du pipeline. La principale particularité de cette structure provient des étages dédiés à la lecture et à l'écriture des opérandes mémoires (étages *READ* et *WRITE*) qui traduisent la capacité de l'architecture à effectuer dans la même instruction la lecture des données en mémoire, les calculs sur les données et l'écriture directe des résultats en mémoire. Ce type d'architecture est à l'opposé des architectures « load/store » utilisées dans la plupart des processeurs RISC et DSP, dans lesquelles ces trois opérations requiert trois instructions différentes et sont exécutées dans le même étage de pipeline. Présente dans certains DSPs d'ancienne génération qui compensaient leur faible nombre de registres internes par la capacité d'adresser directement la mémoire, ce type d'architecture est à l'heure actuelle assez peu utilisée, les processeurs disposant de beaucoup plus de registres internes. Parmi les DSP VLIW, seul le CARMEL d'Infineon a opté pour cette solution.

Ses inconvénients proviennent avant tout de l'augmentation de la longueur du pipeline, qui a pour conséquences l'augmentation de la pénalité en cycles pour les instructions de branchements non retardées, et l'apparition de nouveaux conflits entre les étages de pipeline nécessitant soit du matériel supplémentaire de détection soit une plus grande complexité des outils logiciels qui devront prendre en compte la structure du pipeline. Au niveau matériel, cette solution réclame plusieurs registres de contrôle supplémentaires dans l'unité DMU pour mémoriser les adresses de lecture et d'écriture (registres *AL* et *AE* de la Figure 72), les données lues ou écrites dans l'unité CU (registres *SBr* et *SBw*), et les informations de contrôle se propageant d'étages en étages. On notera à ce titre qu'il peut être intéressant pour réduire le matériel de limiter certains ports de lecture/écriture de l'unité DMU (Figure 63) à la seule fonction de lecture. Le rapport entre le nombre d'accès en lecture et en écriture constaté en moyenne pour les applications multimédia étant d'environ 3 pour 1 [12], il est donc possible de limiter les ports d'écriture sans pour autant dégrader la performance globale du système. Dans notre architecture, cette limitation permettra de réduire le nombre de registres nécessaires pour

conserver les adresses d'écriture jusqu'à l'étage *WRITE* , et surtout de diminuer le coût du système mémoire *Données* grâce à la diminution du nombre de ports d'écriture.

Etage de pipeline	***Description***
FETCH	• Lecture de l'instruction courante • Calcul du prochain CP
DECODE	• Décodage de l'instruction • Répartition du contrôle vers les autres unités d'exécution
ADDRESS	• Calcul des adresses de lecture et d'écriture Données • Actualisation des registres pointeurs
READ	• Lecture des données sources et écriture dans les registres SBrx
EXECUTE	• Lecture des operandes registres • Lecture des opérandes mémoires (SBrx) • Calcul des UFs • Ecriture des opérandes registre • Ecritures des opérandes mémoire (SBwx)
WRITE	• Ecriture des données résultats (SBwx) en mémoire

Figure 73 Description des étages de pipeline

Le premier avantage de la solution à accès mémoire direct est qu'elle rend la programmation des algorithmes en assembleur plus intuitive, le programmeur n'ayant pas à décomposer chaque opération utilisant une variable mémoire en deux opérations séquentielles, l'une de chargement de l'opérande et l'autre d'exécution de l'opération. Il en résulte un gain très net en terme de lisibilité du code assembleur. Pour s'en convaincre, il suffit de comparer à fonction égale le code assembleur du CARMEL (accès direct) et celui du C62x (Load/Store) : le code de ce dernier est beaucoup moins lisible, essentiellement à cause du pipeline logiciel employé pour masquer les effets de l'architecture « Load/Store ».

Le surcroît de complexité résultant de l'emploi de la technique du pipeline logiciel est illustré Figure 74 au travers de l'exemple d'une simple boucle *For* décalant tous les éléments d'un tableau de longueur *n*. Avec une architecture à accès direct, le code assembleur est basé sur l'équation « normale » de l'algorithme, tandis qu'une architecture « Load/Store » disposant du même matériel utilise la forme « pipeline » pour faire usage du parallélisme. Si, pour les grandes valeurs de *n*, la différence de performance est négligeable, on constate que le coût matériel de la deuxième solution est supérieure à cause des deux registres supplémentaires mais surtout à cause de l'accroissement considérable de la taille du code généré. La forme pipeline réclame en effet 3 fois plus d'instructions, et ces mêmes instructions encodent un plus grand nombre d'opérations, nécessitant un mot d'instruction plus large. Enfin, il faut noter que l'ajout de variables registres supplémentaires dans les boucles critiques risque fort d'accroître le nombre de ports d'entrée/sortie nécessaires pour le banc de registres de l'unité CU, ce qui peut entraîner une diminution des performances matérielles.

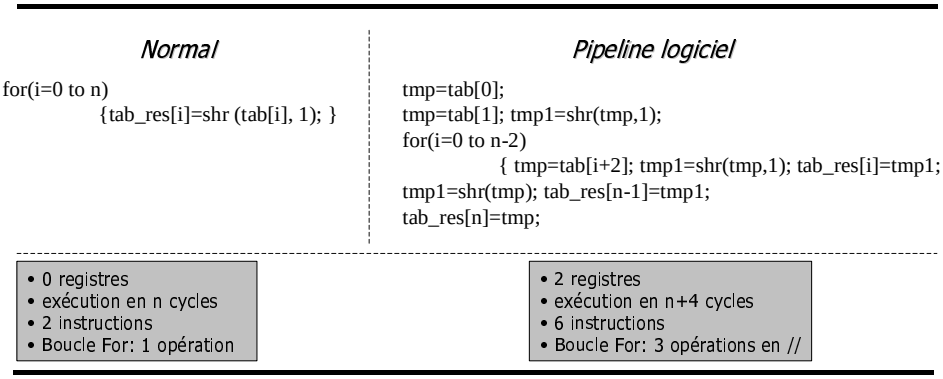


Figure 74 Application de la technique du pipeline logiciel

La complexité inhérente à l'exploitation du parallélisme dans les architectures « Load/Store » a aussi une influence notable sur la difficulté des compilateurs à générer du code optimisé pour les boucles critiques. Le plus grand nombre de registres complique la tache d'allocation de registres, de même que le plus grand nombre d'opérations à exécuter en parallèle rend plus difficile les tâches d'allocation de ressources et d'ordonnancement.

Bien entendu, l'exemple spectaculaire de la Figure 74 est un cas extrême et n'est pas représentatif de l'ensemble des algorithmes utilisés en traitement du signal. La différence entre les deux types d'architecture est moins marqué pour des portions de code plus éloignées de cet exemple, comme par exemple du code de contrôle ou des boucles dont la longueur et le parallélisme sont suffisants pour se dispenser de pipeline logiciel. Mais au vu de la nature des algorithmes DSP qui accèdent fréquemment à la mémoire et dont la localité temporelle de données est faible (les données sont souvent traitées sous forme de flux), la solution à accès directe semble une alternative intéressante du point de vue des performances et de la qualité d'interaction avec les compilateurs.

3.5.2.2 Calcul d'adresses

Le choix d'allouer un étage complet au calcul d'adresse n'est pas une conséquence directe du choix de l'architecture à accès direct, mais un choix lié à la volonté de disposer dans l'unité AGU de modes d'adressages évolués. Pour les processeurs qui n'autorisent pour l'adressage indirect que la forme post-incrémentée, il est en effet parfaitement possible d'effectuer les calculs d'adresses et l'actualisation des registres pointeurs dans l'étage *READ*, tandis que la valeur courante des pointeurs est utilisée pour adresser la mémoire *Données*. C'est parce que nous autorisons des modes d'adressages évolués permettant de propager directement des adresses issues de la somme du contenu de deux registres (pointeur, compteur, immédiat, cf. 3.6) que cet étage a été ajouté afin de couper une hypothétique mais probable chaîne longue partant des registres sources du calcul d'adresse, traversant les UCAs, la mémoire *Données* et arrivant à l'entrée des registres *SBRx*. Il sera donc possible le cas échéant d'intégrer aux UCAs des fonctionnalités supplémentaires destinés à de nouveaux modes d'adressage sans pour autant déséquilibrer le pipeline.

3.5.2.3 Décodage, répartition et exécution conditionnelle

La structure de l'étage de décodage présentée précédemment (Figure 71 et Figure 72) peut faire penser à tort que le décodage de l'instruction VLIW se fait entièrement dans l'étage *DECODE* et que

l'ensemble des signaux de contrôle sont ensuite propagés vers les registres de contrôle du pipeline puis routés vers les unités d'exécution correspondantes. Une telle structure a le désavantage de réclamer un nombre de fils de routage et d'éléments de mémorisation très importants pour conserver les bits de commande de toutes les unités sous forme décodés dans chaque étage du pipeline, et d'occasionner ainsi un coût matériel et une consommation très élevée. La solution que nous préconisons pour l'implémentation matérielle consiste à effectuer un décodage hiérarchique de l'instruction VLIW. Les seules informations extraites du mot d'instructions dans l'étage DECODE seront celles utilisées au cycle suivant ; dans notre cas, il s'agit avant tout des opérations de l'unité AGU ainsi que certaines opérations effectuées dans l'étage ADDRESS comme l'incrémentation des compteurs de boucle. C'est aussi dans cet étage que se fera l'analyse et la répartition sous forme encodée des différentes opérations de l'instruction vers les registres de contrôle correspondant aux différentes unités d'exécution. Le décodage proprement dits de ces opérations peut être différé plus loin dans le pipeline à l'aide de décodeurs locaux, juste avant l'étage d'exécution de l'opération. Ce retard permet à la fois de répartir le coût combinatoire lié au décodage de l'instruction complète, mais aussi de n'activer un décodeur local que si celui-ci est réellement concerné par l'instruction en cours. Cette technique permet de réduire la consommation du circuit en inhibant l'écriture dans les registres d'entrée des décodeurs locaux et en supprimant les transitions combinatoires inutiles de ces mêmes décodeurs. Dans le même esprit, la possibilité de conditionner l'exécution de certaines opérations en fonction de la valeur d'un bit de prédicat (le bit T) explique la connexion de ce bit avec le décodeur (Figure 71).

L'unité de contrôle offre un degré de paramétrage inférieur à celui des autres unités, la spécialisation du modèle portant avant tout sur les unités de calcul et d'adresses. Il est néanmoins possible de configurer certains paramètres comme la profondeur des piles matérielles servant à mémoriser les informations des boucles « zéro-cycle », la profondeur de la pile CP utilisée pour la mémorisation des adresses de retour, et le nombre de registres compteurs exportés vers les autres unités.

Paramètres	*Nom*	*Valeur*
Nombre de sorties registres compteur	N_Cntr	param
Profondeur de la pile CP	P_pile_CP	param
Profondeur de registres compteurs	P_pile_CNTR	param

Figure 75 Paramètres de l'unité de contrôle

3.6 Jeu d'instructions

Après avoir vu l'architecture matérielle du processeur, nous allons maintenant nous intéresser au jeu d'instructions associé au modèle d'ASIP configurable. La prise en compte des nombreux paramètres du modèle ainsi que l'intégration d'unités matérielles utilisateurs imposent au jeu d'instruction d'être générique à la fois du point de vue de la syntaxe, de la structure et de la stratégie d'encodage. Ces deux premiers points vont être présentés dans les prochaines sections. La question de l'encodage sera étudiée au chapitre 5.

3.6.1 Syntaxe

Il existe principalement deux types de syntaxe assembleur dans le domaine des processeurs DSP, l'une basée sur l'emploi de mnémoniques et l'autre dite « algébrique » se rapprochant de la syntaxe d'un langage de programmation séquentiel et utilisant des opérateurs généraux : **+, -, *, /, >,** etc. Jusqu'à une époque récente, l'ensemble des jeux d'instructions DSP faisait usage des mnémoniques pour commander les chemins de données complexes et peu réguliers des DSPs mono-scalaires. Chaque mnémonique représente un ensemble de commandes à exécuter, ces commandes pouvant être de natures diverses (par exemple une multiplication en parallèle avec une addition en parallèle avec un décalage …). L'avantage de cette technique est qu'il existe une correspondance quasi directe entre le nom du mnémonique et la valeur de l'opcode à encoder dans le mot d'instructions, cet opcode définissant à lui seul l'ensemble des opérations à exécuter dans le processeur, et permettant du coup un fort taux d'encodage du jeu d'instructions.

Avec l'apparition des processeurs conventionnels étendus et des processeurs VLIW est apparue la nécessité d'encoder dans la même instruction un nombre supérieur d'opérations, lié au plus grand nombre d'unités matérielles susceptibles de fonctionner en parallèle. La principale conséquence a été un accroissement significatif du nombre de combinaisons d'opérations pouvant être utilisées comme instructions, et par conséquent du nombre de mnémoniques nécessaires. Or, s'il est possible de trouver un nom « parlant » pour un mnémonique encodant une voire deux opération élémentaires (ex : MAC pour multiplication-accumulation), cela devient plus difficile pour un nombre d'opérations supérieures et s 'appliquant à des unités fonctionnelles distinctes. L'usage unilatéral des mnémoniques a été abandonné pour introduire la notion de parallélisme d'exécution, souvent matérialisé sous forme de symboles (« , », « || ») séparant des « opérations élémentaires » s'exécutant en parallèle. C'est au niveau de la description de ces opérations qu'est apparue dans certains processeurs la syntaxe algébrique pour remplacer les mnémoniques. Dans le *Lucent DSP16xxx*, une opération de multiplication-accumulation est ainsi décrite : « a0=a0+p1*p2 ». Le principal avantage de ce type de description est d'améliorer la lisibilité du code par rapport à l'usage des mnémoniques.

```
        [AGU]     op0,  op1,  …,   opN
//      [CU]      op0,  op1,  …    opM
//      [PCU]     op0,  op1,  …    opL    ;
```

Figure 76 Structure d'une instruction générique

La syntaxe et la structure d'une instruction pour le modèle d'ASIP est représentée Figure 76. Chaque instruction est séparée en trois champs séparés par le symbole « || » et décrivant les opérations prenant place dans les unités AGU, CU et PCU (les opérations de l'unité DMU étant implicitement décrites dans celles d'AGU et CU). Au sein de chaque champ, les opérations effectuées en parallèle sont séparées par des virgules. Un champ peut être omis si aucune opération particulière n'est à exécuter dans l'unité correspondante. Le nombre d'opérations parallèles maximum dans chaque champ est paramétrable et est évidemment lié au parallélisme matériel des unités associées. On verra plus loin que la description des opérations dans les différents champs utilise à la fois les mnémoniques et la syntaxe algébrique.

3.6.2 Opérations de l'unité CU

Puisque l'unité CU permet d'intégrer des unités fonctionnelles utilisateurs aux fonctionnalités très diverses, la syntaxe algébrique à base d'opérateurs généraux (+,-,*,/) n'est pas le meilleur choix car rien ne garantie que ces fonctionnalités puissent être décrites simplement à l'aide de ces opérateurs. Il est dans ce cas beaucoup plus simple d'associer à chaque opération d'une UF un mnémonique particulier. L'ensemble des opérations exécutables par l'unité CU est décrite dans une table d'allocation reliant les mnémoniques à leur unité fonctionnelle correspondante. Cette table décrit les fonctionnalités du chemin de données et permet d'ajouter une UF de manière simple en créant de nouvelles entrées correspondant aux mnémoniques de commande de la nouvelle unité. Il est possible que certains mnémoniques correspondent à plusieurs UFs lorsque l'opération associée peut être prise en charge par les deux unités. Une autre table spécifiant le nombre d'UFs par type complète la description du chemin de données et fixe le parallélisme matériel. Ces deux tables sont utilisées par le simulateur d'instructions et le compilateur pour prendre en compte les caractéristiques du processeur cible.

Mnémonique	*UF*
mac	MAC
msu	MAC
add	ALU / MAC
sub	ALU / MAC
round	ALU
abs	ALU
shr	BMU
shl	BMU
norm_s	BMU

Figure 77 Table d'allocation Mnémonique/UF

Une opération de l'unité CU est donc décrite par un mnémonique décrivant la nature de l'opération et un ensemble d'opérandes sources et destination. Les UFs peuvent accéder à trois types d'opérandes : des registres du banc de registre de CU (nommés ***Ax***), des registres de l'unité DMU correspondant aux ports d'accès mémoire (nommés ***SBx***), et des registres provenant des autres unités (AGU, PCU) par le biais des bus d'échanges inter-unités. La taille des registres ***Ax*** et ***SBx*** est par défaut de respectivement 2*N_DATA_BITS (double précision) et N_DATA_BITS (simple précision), mais ils peuvent être adressés par demi-registres pour les registres ***Ax*** ou par deux registres à la fois pour les registres ***SBx*** à l'aide de suffixes spéciaux. Trois exemples d'opérations CU sont présentés sur la Figure 78 pour N_DATA_BITS=16, ce qui correspond à un chemin de données 16/32 bits. Les registres ***SBx*** contiennent la valeur écrite ou lue à l'adresse formée par le $\boldsymbol{x}^{ème}$ champ de la ligne [AGU]. Un registre SB donné ne pas être à la fois lu et écrit dans le même cycle puisqu'il correspond à un seul port d'accès mémoire. Deux registres SB peuvent par contre être concaténés pour permettre des accès mémoire double précision.

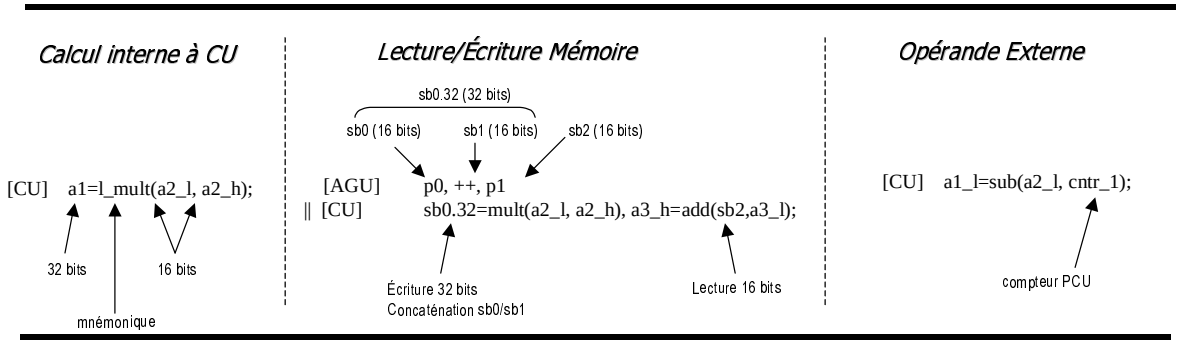


Figure 78 Exemples d'opérations CU

3.6.3 Opérations de l'unité AGU

Une instruction AGU est formée de plusieurs champs séparés par des virgules, dans lesquels sont effectués les calculs des adresses des différentes opérandes correspondant aux registres ***SBx***. Dans l'exemple de la Figure 78, l'instruction AGU contient trois opérations qui correspondent aux trois variables mémoire accédées par le chemin de données, 2 en écriture et une en lecture. Les opérations les plus courantes entre registres sont définies de façon algébrique au moyen des opérateurs +, -, ++ et --. Les principaux modes d'adressages autorisés avec des UCAs de type additionneur/soustracteur sont décrits en détail dans la Figure 79. Les opérateurs « ++ » (exemple de la Figure 78) et « -- » utilisés seuls sont des cas spéciaux signifiant que l'adresse exportée sur le port correspondant est égale à la valeur de l'adresse du port précédent incrémenté (ou décrémenté) de 1. Ces modes sont avant tout utilisés pour l'adressage de données double précision. Les opérandes sources des calculs d'adresses peuvent aussi être des immédiats ou des valeurs issues des bus d'échanges (par exemple des compteurs de boucles). Le nombre de ce type d'opérandes est limité par le nombre de bus d'échanges.

Syntaxe	*Description*
…,px,…	• exportation du contenu du registre px
…,px++,…	• exportation du contenu du registre px • post-incrémentation de px
…,px--,…	• exportation du contenu du registre px • post-décrémentation de px
…,px +/- py,…	• exportation de la somme/différence du contenu de px et py
…,py=px,…	• exportation du contenu de px • résultat dans py
…,pz=px +/- py,…	• exportation de la somme/différence du contenu de px et py • résultat dans pz
…,py+/-=px ,…	• exportation de la somme/différence du contenu de px et py • résultat dans py
…,++,…	• exportation de la valeur du port d'adresse précédent +1
…,--,…	• exportation de la valeur du port d'adresse précédent -1

Figure 79 Principaux modes d'adressage de l'AGU

Le nombre d'opérations maximum par instruction dépend à la fois du nombre de ports mémoire physiques, du nombre d'UCAs de l'unité d'adressage qui fixe le nombre de calculs d'adresses autorisés, et de contraintes d'encodage liée à la largeur maximale du mot d'instruction. Il est possible de définir dans une instruction AGU plus d'opérations que de ports mémoires physiques. Dans ce cas, seules les adresses correspondant aux premières opérations (de gauche à droite) seront exportées, dans la limite des ports physiques disponibles. Les autres opérations seront malgré tout exécutées et peuvent par exemple servir à actualiser la valeur d'un pointeur qui n'a pas besoin d'être exporté.

3.6.4 Opérations de l'unité PCU

Ces opérations sont utilisées pour tous les modes de séquencement autres que le déroulement séquentiel des instructions. Elles peuvent être classées en trois catégories : les instructions de saut (***JUMP***), les instructions liées aux sous-programmes (***CALL***, ***RTS, CALLF, RTSF***) et les instructions de boucle.

Nom	*Description*	*Durée*
JUMP	• Saut à une adresse quelconque	3 cycles
CALL	• Saut à un sous-programme	6 cycles
RTS	• Retour de sous-programme	5 cycles
ENABLE_LOOP(x,short/long)	• Activation d'une boucle zéro-cycle courte/longue de niveau x	1 cycle
BRK	• Sortie de boucle anticipée	3 cycles
CONT	• Saut au début de la prochaine occurrence de la boucle	3 cycles
Affectation	• Initialisation des registres PCU	1 cycle
CALLF	• Saut à un sous-programme et sauvegarde automatique de l'adresse de retour	3 cycles
RTSF	• Retour de sous-programme et restitution automatique de l'adresse de retour	3 cycles

Figure 80 Instructions PCU

L'adresses destination des sauts et appels de sous-programme peut provenir soit d'un immédiat soit d'un registre. A cause de la structure du pipeline, ces instructions ont un délai d'exécution supérieur à un cycle. Il faut par exemple 3 cycles pour exécuter un *JUMP* lorsque l'adresse du sous-programme est codée dans un immédiat (un cycle de chargement de l'instruction, un cycle de décodage et un cycle pour le calcul du CP) et 5 lorsqu'elle est dans un registre. Les instructions *CALLF* et *RTSF* ont exactement le même comportement que les instructions *CALL* et *RTS*, mais l'adresse de retour de la fonction est automatiquement sauvegardée/restaurée sans utiliser la pile logicielle (grâce à la pile matérielle CP), ce qui permet de réduire de deux et trois la latence des instructions.

Les opérations d'affectation des registres PCU servent à initialiser les registres de boucle : valeur initiale du compteur (*cntr_x*), valeur d'incrément (*cntr_inc_x*) ,valeur de fin (*ln_x*) et adresse de début de boucle (*lsa_x*). La fonction *Enable_Loop* active la boucle de niveau désiré et configure la boucle comme boucle courte (short) ou longue (long). Cette distinction est nécessaire à cause des mécanismes de contrôle matériels gérant les boucles « zéro-cycles » dont le comportement doit être différent selon la longueur de la boucle. Une boucle est considérée courte si elle fait au plus deux instructions, et longue au delà. Dans le cas des boucles courtes, il est inutile de mémoriser l'adresse de début de boucle car le matériel est capable de la mémoriser. Les instructions *BRK* (*break*) et *CONT* (*continue*) permettent comme leurs équivalents en langage C de sortir immédiatement de la boucle de plus haut niveau sans terminer l'occurrence courante et sans exécuter les occurrences suivantes, et de passer directement au début de l'occurrence suivante sans terminer l'occurrence courante.

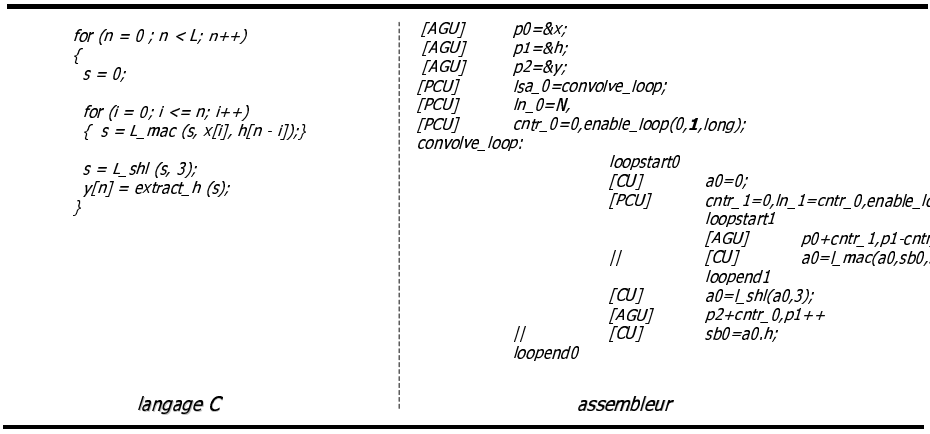


Figure 81 Fonction de convolution en C / Assembleur

La Figure 81 illustre un exemple d'utilisation des boucles zéro-cycle au travers de l'implémentation d'une fonction de calcul de convolution utilisant deux boucles de calcul imbriquées. La boucle la plus profonde (de niveau de 1) composée d'un seule instruction est définie comme courte et la boucle externe comme longue. Le début et la fin des boucles sont localisés par deux directives spéciales (*loopstart_x* et *loopend_x*). Ces directives ne sont pas des instructions proprement dites mais sont utilisées par l'outil assembleur qui insère dans le code de certaines instructions des bits supplémentaires signalant le début et la fin des boucles. Ces bits sont détectés lors du décodage de l'instruction par la logique de contrôle du processeur et lui permettent d'anticiper les opérations nécessaires au traitement des boucles, comme l'incrémentation des compteurs, le test par rapport à la valeur de fin et le branchement conditionnel vers le début de la boucle, etc.

Une fois les initialisations des registres pointeurs et des registres de boucle terminées, les deux boucles n'exécuteront que les instructions « utiles » correspondant aux calculs décrits dans l'algorithme en C, sans aucune perte de cycles dus aux effets de pipeline. Seule l'instruction de réinitialisation de la boucle 1 ne fait pas partie de l'algorithme et constitue donc une légère perte par rapport à la performance maximale atteignable, perte qui sera d'autant plus négligeable que la boucle 0 sera longue. On peut aussi remarquer l'usage des compteurs de boucle comme index pour le calcul d'adresses des opérandes, qui permettent une actualisation automatique des pointeurs entre deux occurrences de la boucle et qui évitent d'avoir à réinitialiser les pointeurs à la valeur de base du tableau à chaque fin de boucle.

3.6.5 Instructions spéciales

Nous présentons ici certaines instructions « spéciales » qui ne peuvent être rangées parmi les trois catégories précédentes.

3.6.5.1 Gestion de la pile

Il est parfaitement possible de simuler une pile logicielle sans instructions particulières, par exemple en réservant un des registres pointeurs à l'usage exclusif de la pile et en se servant de ce pointeur pour empiler ou dépiler des données comme n'importe quel instruction d'accès à la mémoire. Cependant, pour permettre une meilleure compacité du code, il peut être plus efficace de réserver des instructions dédiées uniquement à la manipulation de la pile. Ces instructions, qui sont utilisées dans tous les

programmes et plus particulièrement dans ceux résultant d'une compilation, ont en effet tout intérêt à occuper le moins de place en mémoire, et doivent donc faire l'objet d'un fort encodage. L'association d'un mnémonique et d'un opcode unique aux opérations d'empilement et de dépilement est un moyen efficace d'obtenir un bon taux d'encodage. On dispose donc dans le jeu d'instruction de base de deux instructions nommées « traditionnellement » ***PUSH*** et ***POP***, et qui permettent respectivement l'empilement et le dépilement d'un mot simple précision.

Si nécessaire, on peut imaginer l'usage de mnémoniques supplémentaires PUSH2, POP2, PUSH4, POP4, permettant des accès de plus grande largeur à la pile pour permettre des sauvegardes et des restitutions de contexte plus rapide. L'intérêt de ces instructions qui nécessitent des opcodes supplémentaires dépend bien sur du parallélisme matériel autorisé par le modèle d'ASIP et de la fréquence de ces opérations dans le code de l'application.

3.6.5.2 Test et codes condition

La plupart des processeurs disposent dans leur registre d'état de bits spéciaux (Carry, Overflow, Zero, etc.) dont la valeur peut être testée par certaines instructions pour permettre une exécution conditionnelle (par exemple pour des instructions de sauts conditionnels comme JNE (*Jump If Not Equal*), JZ (*Jump If Zero*), etc.). Ces bits sont en réalité des bits d'état générés par une unité fonctionnelle particulière (l'ALU du processeur).

Pour assurer la généricité du jeu d'instructions vis-à-vis des unités fonctionnelles, le processeur n'intègre qu'un seul bit de condition : le bit T de l'unité PCU. L'écriture de ce bit est commandé par des instructions de test spécifiques à chaque unité fonctionnelle et propageant le code de condition calculé par l'UF vers le bit T. L'instruction « test(eq, a4, 5) » de la Figure 82 est associé à l'unité fonctionnelle ALU (le mnémonique devra être présent dans la table d'allocation Mnémonique/UF), calcule la valeur du code condition ***EQ*** (*Equal*) en fonction des opérandes sources et de la valeur de bits d'état internes de l'UF (les bits ***Cy***, ***Ov***, ***Zr*** et ***Sg*** de la Figure 65), et exporte le résultat vers le bit T.

Cette structure permet ainsi de définir d'autres codes de condition relatifs à l'état de registres internes d'unités fonctionnelles utilisateurs et de les manipuler de manière simple grâce à des mnémoniques spécifiques. L'utilisation de la valeur de ces codes pour conditionner les instructions est expliquée ci-dessous.

3.6.5.3 Exécution conditionnelle

Le jeu d'instructions offre la possibilité de conditionner l'exécution de chaque opération présente dans une instruction. Cette caractéristique est présente dans de nombreux DSP et a pour but de masquer les effets indésirables du pipeline de contrôle lors de l'implémentation des structures de type *if-then-else*. En effet, une telle structure implémentée classiquement à l'aide d'une opération de test puis de branchements conditionnels est 5 fois plus coûteuse en cycles par rapport à une solution basée sur des exécutions conditionnelles (voir l'exemple de la Figure 82). La première instruction teste la condition et affecte le résultat au bit T (Vrai ou Faux). Les opérations correspondant aux corps des portions « if » et « else » sont alors concaténées dans des instructions s'exécutant en parallèle, mais dans lesquelles une seule des deux opérations est réellement exécutée. Le conditionnement s'effectue

au moyen de deux directives ***IFF*** (*If False*) et ***IFT*** (*If True*) placées avant l'opération à conditionner. L'absence d'instructions de sauts dans le code permet d'éviter les cycles inutiles liés au pipeline.

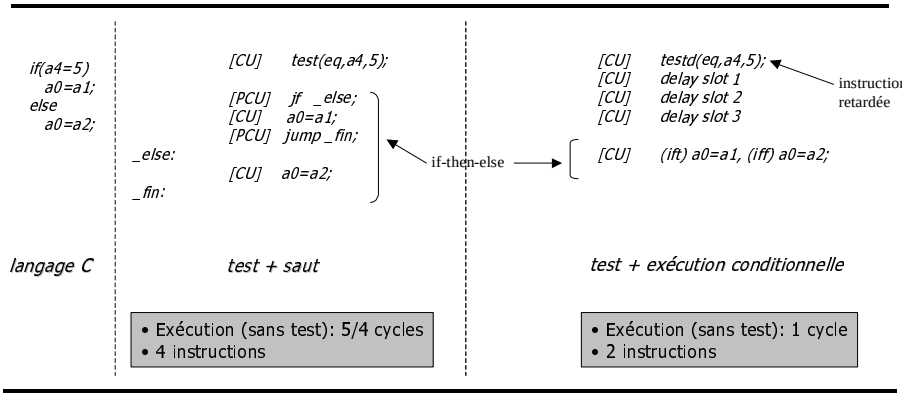


Figure 82 Coût d'une structure if-then-else

L'intérêt de cette méthode de programmation est d'autant plus grand que le nombre d'opérations dans les deux parties de la structure est faible. Elle permet surtout d'accroître considérablement les performances des boucles critiques des algorithmes qui contiennent une structure de test en parallélisant le calcul des deux parties et en supprimant les cycles d'attente des instructions de saut. Une autre conséquence est qu'il n'y a plus besoin d'instructions de sauts conditionnels de type ***jt*** (Jump if True) ou ***jf*** (Jump if False), d'où leur absence des opérations PCU présentées plus haut.

3.6.5.4 Instructions retardées

L'instruction de test utilisée dans l'exemple de la Figure 82 pose un problème lié à la structure de pipeline du processeur. Comme le calcul de la condition de test se fait dans une unité fonctionnelle de l'unité CU, le bit T n'est valide qu'au cycle suivant l'exécution du test dans l'étage EXECUTE. Dans le même temps, la sélection des opérations réellement exécutées lors d'une exécution conditionnelle se fait dans l'étage DECODE. Pour assurer un fonctionnement correct, il est donc nécessaire de geler le pipeline du processeur pendant trois cycles jusqu'à l'écriture du bit T. Ainsi, l'exécution de l'instruction « test » prendra non pas un mais quatre cycles, ce qui peut être très pénalisant si l'instruction est employée dans une boucle critique.

Pour diminuer le nombre de cycles apparents des instructions multi-cycles (essentiellement les sauts et les tests), on utilise le concept des instructions retardées, c'est à dire dont le résultat n'est pas disponible au cycle suivant mais qui ne gèlent pas le processeur et qui permettent ainsi d'intercaler dans les cycles inutiles (*Delay Slots*) des instructions supplémentaires. Evidemment, ces instructions ne doivent pas dépendre du résultat de l'instruction retardée (voir l'exemple de la Figure 82).

On dispose ainsi dans le jeu d'instructions de base de versions « retardées » des instructions multi-cycles : ***TESTD*** (pour les tests arithmétiques de l'ALU), ***CALLD***, ***RTSD***, ***BRKD*** et ***CONTD***.

3.6.6 Les aléas de pipeline

Le découpage des instructions le long du pipeline d'exécution est source de deux types de problèmes bien connus : les aléas de données et les aléas de contrôle [55].

3.6.6.1 Les aléas de données

Les aléas de données surviennent lorsque deux instructions qui s'exécutent simultanément dans deux étages de pipeline différents accèdent à la même donnée, entraînant le non respect d'une contrainte RAW (*Read after Write*) ou WAR (*Write after Read*) entre les deux accès. Les conflits les plus courants pour notre modèle d'ASIP sont présentés Figure 83 et concernent essentiellement la lecture et l'écriture des registres les plus couramment utilisés du processeur : les registres ***Ax*** de l'unité CU (écriture dans l'étage EXECUTE), les registres ***Px*** de l'unité AGU (écriture dans l'étage ADDRESS, sauf lors de l'initialisation à partir d'une valeur mémoire : l'écriture se fait dans ce cas dans l'étage EXECUTE), et les registres compteurs de l'unité PCU (incrémentation dans l'étage ADDRESS de la dernière instruction de la boucle). Le délai de la 3ème colonne correspond au nombre de cycles minimum devant séparer l'exécution des deux instructions conflictuelles.

Description	Type	Délai
Ecriture mémoire adresse X (WRITE) puis lecture mémoire même adresse (READ)	RAW	2 cycles
Ecriture registre dans l'étage EXECUTE puis lecture registre dans ADDRESS exemples : • initialisation registre pointeur [AGU] p4=a0_l ; • initialisation registre de boucle : [PCU] ln=a1_l ; • initialisation pointeur : [AGU] adr || [CU] p9=sb0 ; (EXECUTE) puis accès mémoire utilisant p9 (ADDRESS)	RAW	2 cycles
Lecture registre compteur (EXECUTE) avant la fin d'une boucle LONG et de l'incrémentation du compteur (ADDRESS): [CU] a1_l=cntr_2 ;	WAR	2 cycles

Figure 83 Aléas de données

L'utilisation de l'instruction retardée TESTD peut aussi provoquer une erreur si les opérations conditionnées par le bit T sont situées dans les trois instructions qui suivent l'instruction de test, le bit T n'ayant pas encore été actualisé.

3.6.6.2 Les aléas de contrôle

Les aléas de contrôle apparaissent lorsque deux instructions s'exécutant dans deux étages de pipeline différents veulent accéder à la même ressource matérielle. Certains étages de pipeline sont par exemple autorisés à accéder aux mêmes registres ; c'est le cas des registres pointeurs de l'AGU qui peuvent être écrits à la fois dans ADDRESS et dans EXECUTE. Certaines instructions nécessitent aussi l'échange de données entre unités d'exécution par l'intermédiaire des bus d'échanges, qui doivent donc être suffisamment nombreux pour éviter la saturation liée à une trop grande circulation de données.

Pour éliminer ce type d'aléas, le concepteur a alors le choix entre modifier le code pour faire disparaître les conflits (avec comme risque principal une diminution plus ou moins importante de la performance) ou sur-dimmensionner le matériel pour permettre les accès concurrents entre instructions. La décision se prendra bien sûr en fonction des contraintes inhérentes au système et à l'application.

Enfin, pour certains mécanismes de contrôle complexe comme la gestion des boucles « zéro-cycle », il peut exister certaines règles de codage inhérentes à l'implémentation matérielle de ces mécanismes et

qui doivent être respectées sous peine de fonctionnement erroné. Dans le cas du modèle d'ASIP, il en existe une seule : l'instruction qui suit immédiatement une boucle SHORT ne doit pas contenir une opération *Enable_loop*.

Description	*Résolution*
Ecriture mémoire (WRITE) et lecture mémoire (READ) sur le même port SBx	décaler l'instruction de lecture d'un cycle
Lectures/Ecriture multiples du même banc de registres dans des étages différents	décaler les instructions ou augmenter le nombre de ports d'accès aux bancs de registres
Echanges multiples de données entre étages au travers des bus d'échanges	ajuster le nombre de bus d'échanges

Figure 84 Aléas de contrôle

3.6.6.3 Détection et correction

Pour détecter et résoudre ces conflits, deux approches sont possibles. La première, dite « dynamique », consiste à inclure dans le processeur du matériel supplémentaire détectant les aléas et les corrigeant en gelant certains étages du pipeline pour retarder et décaler les instructions en conflit. Le principal avantage est que le programmeur de l'application ou le compilateur n'a pas à se soucier des problèmes d'aléas sachant que le matériel corrigera les éventuelles erreurs, ce qui facilite considérablement le développement du code. En contrepartie, la logique de contrôle du processeur devient beaucoup plus complexe ce qui a des conséquences sur le coût matériel comme sur les performances. L'autre inconvénient non négligeable et déjà évoqué dans le chapitre sur les microprocesseurs généraux provient de la non-prédictibilité du temps d'exécution du code, qui peut varier de plusieurs cycles selon les aléas rencontrés.

A l'inverse, l'approche statique laisse le soin de la détection et de la correction des aléas au programmeur et/ou aux outils de génération de code (assembleur et compilateur). Une fois placé dans la mémoire programme, le code est supposé sans erreurs et dans le cas contraire, le comportement de l'application ne sera pas conforme. La complexité de la gestion des aléas passe alors du matériel au logiciel et suppose des outils de génération de code intelligents capables de tenir compte des contraintes supplémentaires. Aucun ajout de matériel dans le processeur n'est requis et le nombre de cycles pour chaque fonction est déterministe.

Comme l'ensemble des processeurs DSP VLIW, notre modèle d'ASIP est basé sur l'analyse statique des aléas. Le surcoût matériel et les problèmes de déterminisme temporel liés à l'analyse dynamique sont en effet rédhibitoires pour un processeur visant des applications temps-réel et sensibles aux contraintes de coût. L'accent devra donc être mis sur la qualité des outils de génération de code pour éviter les erreurs d'aléas.

3.7 Conclusion

Dans ce chapitre, nous avons présenté un modèle de processeur configurable et son jeu d'instructions générique associé, destinés à être utilisés dans une méthodologie de conception rapide de processeurs spécialisés ASIP. Le modèle d'architecture du processeur est configurable à plusieurs niveaux : architecture globale, chemin de données, unité de calcul d'adresses et unité de contrôle. Il permet en particulier de faire varier le parallélisme matériel disponible et d'intégrer des unités fonctionnelles spécialisées pour s'adapter au mieux aux caractéristiques de l'application cible. Le jeu d'instructions

générique permet de s'adapter au parallélisme matériel variable de l'architecture et de prendre en compte de nouvelles instructions utilisateurs.

La conception accélérée d'ASIPs requiert l'usage d'une méthodologie bien déterminée, qui permette de définir les paramètres de configuration optimaux du modèle de processeur en fonction des caractéristiques de l'application cible. Cette méthodologie, ainsi que les outils logiciels sur laquelle elle repose, sont détaillés dans le prochain chapitre

.

Chapitre 4

Méthodologie de conception d'ASIPs

Sommaire :

4.1 Introduction

Le but de la méthodologie présentée est d'accélérer le temps de conception des processeurs ASIP par rapport à une méthodologie de conception classique. Traditionnellement, le flot de concept d'un ASIP démarre par l'analyse des applications cibles et des différentes contraintes matérielles et de performance liées au système de traitement global. A l'issue de cette analyse, une ou plusieurs architectures et jeux d'instructions associés sont proposés et évalués en regard des contraintes du cahier des charges. La meilleure solution doit ensuite faire l'objet d'une réalisation physique. Les concepteurs doivent tout d'abord définir la technologie cible et le type d'implémentation pour chaque bloc du processeur (synthèse de haut niveau, blocs « full-custom », générateurs de netlist de cellules standards optimisées, etc.). Le circuit est alors réalisé et testé au moyen d'outils CAO (synthétiseurs et simulateurs VHDL/Verilog, placeurs/routeurs, etc.). L'obtention du code objet décrivant l'application nécessite aussi la réalisation d'outils de génération de code : assembleur, linker, debugger, et même compilateur dans le cas où l'application est complexe. Le cas échéant, le flot de conception peut être re-parcouru plusieurs fois pour évaluer certains changements destinés à améliorer les caractéristiques du processeur. Ce processus est au final très coûteux en temps, chaque étape nécessitant un travail important de réécriture des outils logiciels et des descriptions matérielles du processeur.

Les défauts de ce type de méthodologie proviennent de leur trop grande généralité, qui ne fixent aucune contrainte en terme d'architecture, de procédé de fabrication ou d'implémentation matérielle, ce qui rend presque impossible le développement d'outils de CAO efficaces et requièrent donc un important travail « manuel » de la part des concepteurs.

La méthodologie que nous proposons tend à accélérer le temps de développement en restreignant « l'espace de conception » à un modèle d'architecture et de jeu d'instructions clairement définis et à une méthodologie d'implémentation matérielle reposant sur le concept de générateurs de cellules standards indépendants de la technologie cible. Ces restrictions permettent de définir un ensemble d'outils logiciels de production de code et de réalisation matérielle capables de prendre en compte l'ensemble des différentes possibilités et donc de réduire le temps de conception en minimisant les interventions des concepteurs. Puisqu'il n'est pas nécessaire de réécrire les outils logiciels à chaque changement du processeur, il est ainsi très simple de tester différentes architectures et de converger rapidement vers une solution optimale.

Le principe de la méthodologie consiste à partir du modèle générique non configuré du cœur d'ASIP présenté au chapitre précédant, et d'un ensemble d'applications cibles et de contraintes applicatives (performance, coût matériel, puissance dissipée, etc.). On doit alors rechercher la meilleure configuration du modèle d'ASIP en fonction des caractéristiques des applications cibles et qui satisfont les contraintes du cahier des charges.

Nous avons choisi d'illustrer cette méthodologie à travers l'exemple du développement d'un ASIP destiné à l'implémentation des principales fonctions de traitement audio du protocole GSM. Deux algorithmes particuliers seront examinés : le décodeur de Viterbi et l'algorithme de codage de voix EFR, tous deux très coûteux en temps de calcul et représentatifs des exigences requises par les applications de traitement du signal embarquées. Afin de permettre une comparaison des résultats avec les autres implémentations du protocole GSM sur les processeurs DSP du commerce, nous

limitons volontairement le parallélisme matériel du modèle d'ASIP à **deux unités MAC maximum** (c'est le cas de la plupart des processeurs VLIW actuels). Les choix de conception effectués tout au long de ce chapitre portent sur **l'optimisation de la performance** de ces fonctions sur le processeur ASIP, sans contraintes particulières sur la consommation dissipée ou la surface occupée. Les performances et le coût matériel de l'ASIP final ainsi que la comparaison avec les autres processeurs seront présentés en détail au chapitre suivant.

La prochaine section présente les principaux algorithmes formant la chaîne de traitement GSM. Le flot de conception général ainsi que la phase d'exploration de configuration du processeur sont détaillées dans les deux sections suivantes. La dernière section propose une méthode de réalisation matérielle du processeur basée sur l'emploi de générateurs de macro-blocs portables et paramétrables.

4.2 La chaîne de traitement GSM

L'interface radio du GSM, qui transporte à la fois des informations de type « voix » et de type « données », peut être divisée en six blocs fonctionnels principaux dans le récepteur comme dans l'émetteur (Figure 85) :

- Encodage / décodage de voix : l'encodeur compresse le signal de voix provenant d'un *CAN* d'entrée à un débit de 104 kbits/s (où 64kbits/s pour un *CAN* logarithmique). Trois différents algorithmes sont utilisés (*Full Rate (FR), Half Rate (HR)* et *Enhanced Full Rate (EFR)*) qui diffèrent en terme de taux de compression et de perte en qualité sonore. Les algorithmes FR et EFR utilisent la bande passante maximale autorisée par un canal GSM : 13 kbits/s, ce qui représente une réduction d'un facteur 8 par rapport au débit initial. L'algorithme HR, compressé à 6.5 kbits/s, n'utilise qu'une moitié de canal et permet de doubler la capacité au détriment de la qualité sonore.
- Encodage / décodage canal : l'encodage canal est utilisé pour la détection et la correction d'erreurs lors de la transmission. Pour ce faire, un certain nombre d'informations redondantes sont ajoutées aux informations à transmettre qui permettent la détection d'erreurs à la réception.
- Entrelacement/ désentrelacement : l'entrelaceur re-ordonne et mélange les informations codées de façon à ce que les erreurs de transmission successives apparaissent de manière non successive lors du décodage (après désentrelacement).
- Multiplexeur / démultiplexeur : les bits d'information sont assemblés selon un format de transfert spécifique (*data burst format*) contenant des bits supplémentaires de contrôle. Le multiplexeur se charge alors de la synchronisation de l'envoi des trames constituées sur les *Time Slots* correspondants.
- Modulateur/démodulateur : le modulateur transforme le signal numérique en un signal analogique modulé transmis sur le canal hertzien. Le type de modulation choisi dans la norme GSM est la modulation GMSK (pour *Gaussian-shaped Minimum Shift Keying*).
- Unité Radio-Fréquence : la partie RF amplifie et transmet le sinal modulé sur le canal de communication.

A l'exception de la partie RF et d'un partie du modulateur/démodulateur, la plupart des blocs de l'interface radio manipulent les signaux sous forme numérique, c'est pourquoi les fonctions correspondantes (dites fonctions « *Baseband* ») sont en général prises en charge par des processeurs DSP. La chaîne de traitement présentée ci- dessous ne représente cependant que le minimum vital nécessaire au respect de la norme GSM. La plupart des téléphones mobiles incluent des fonctionnalités supplémentaires comme le cryptage des données, l'annulation d'écho, la reconnaissance de parole, etc. Ces fonctions sont elles aussi prises en charge par le DSP, ce qui pousse à l'utilisation de processeurs performants ayant des « réserves » de puissance pour s'acquitter d'éventuelles fonctionnalités supplémentaires.

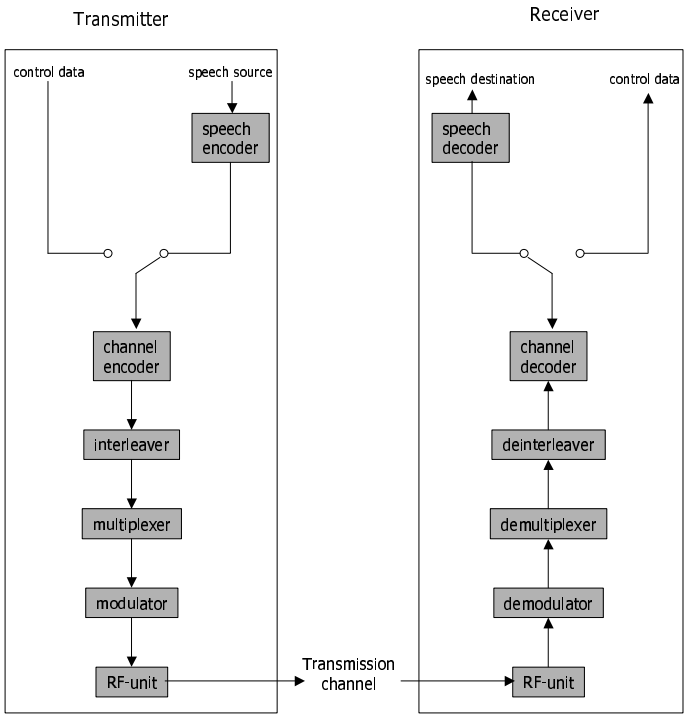


Figure 85 Transmission et réception de la voix dans le protocole GSM

4.3 Le flot de conception général

La méthodologie de conception se décompose en deux phases successives. La première phase consiste à rechercher le meilleur paramétrage du modèle de processeur en fonction des applications cibles et de certaines contraintes du système de traitement. Ces contraintes peuvent être d'ordre matériel (surface de silicium occupée, puissance dissipée) et/ou liées à la performance. La formulation même de ces contraintes varie selon les systèmes et les applications: dans les SoC de traitement du signal destiné au marché de l'embarqué, il est par exemple très courant d'essayer de limiter la puissance dissipée en imposant une fréquence de fonctionnement relativement basse. Connaissant les contraintes temps-réel du système (tant de millisecondes pour effectuer telle tâche), la performance requise pour un cœur de processeur embarqué peut être formulée en nombre de MIPS (Million d'instructions par cycle) maximum par tâches, sachant qu'une instruction correspond à un cycle machine. Au contraire, dans les systèmes où la puissance est de moindre importance, le concepteur a le choix de la fréquence de fonctionnement et doit trouver le meilleur compromis fréquence de travail/ parallélisme matériel pour assurer la performance requise tout en minimisant les autres facteurs de coût. Cette phase,

appelée « exploration de l'espace de configuration », requiert des outils logiciels de production de code et de simulation du comportement du modèle permettant d'évaluer les différentes configurations et de sélectionner la meilleure.

Une fois l'architecture du processeur définie, c'est à dire quand une valeur particulière a été attribuée à chaque paramètre du modèle, la deuxième phase consiste à obtenir l'image matérielle du processeur qui sera utilisée pour la réalisation physique du système. Cette phase s'appuie sur un environnement de développement paramétrable indépendant de la technologie qui permet de générer rapidement et de façon semi-automatique le processeur en fonction de paramètres utilisateurs. Le processeur final est disponible sous forme d'une netlist structurelle résultant d'une projection du cœur de processeur sur la librairie de cellules associée à la technologie définie par l'utilisateur. Cela permet ensuite une réalisation physique immédiate à l'aide de n'importe quelle chaîne d'outils de CAO.

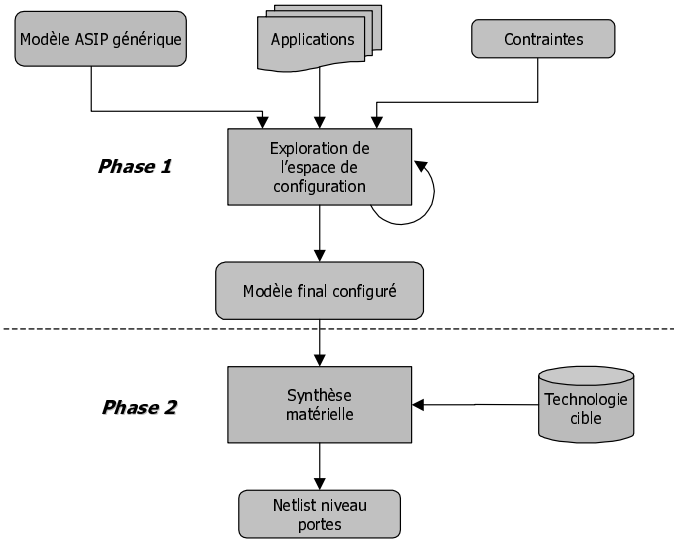


Figure 86 Flot de conception

Le succès de cette méthodologie repose principalement sur la qualité des outils logiciels utilisés dans les deux phases, qui doit garantir un temps de développement réduit par rapport à une méthodologie de conception classique. La grande difficulté réside dans le fait que ces outils doivent être « reciblables », c'est à dire qu'ils doivent prendre en compte les très nombreux paramètres du modèle de processeur et donner des résultats de qualité quelque soit le jeu de paramètres choisis, le but ultime étant de réduire au minimum les interventions de l'utilisateur dans le flot de conception.

4.4 Exploration de l'espace de configuration

La phase d'exploration vise à rechercher le meilleur jeu de paramètres pour le modèle d'architecture et le jeu d'instructions, de manière à répondre au mieux aux contraintes du cahier des charges, formulées habituellement selon trois critères : performance dynamique, consommation et surface de silicium. Nous avons choisi ici de nous intéresser au **critère de performance**. Le but de la phase d'exploration présentée par la suite est d'aider le concepteur à converger vers un processeur assurant une performance minimale pour l'application cible, tout en offrant la possibilité de minimiser les autres contraintes de conception que sont la surface et la consommation.

L'exploration de l'espace de configuration est constituée de trois étapes qui sont l'analyse de complexité de l'application cible, le codage de l'application menant à la définition d'un modèle particulier du processeur, et l'évaluation des performances de ce modèle. Si à l'issue de la dernière étape les contraintes ne sont pas satisfaites ou que la solution choisie est perfectible, le processus est réitéré jusqu'à l'obtention d'un modèle final du processeur satisfaisant l'ensemble des contraintes.

La première étape d'**analyse de complexité** a pour but :

- de définir l'ensemble des opérateurs arithmétiques élémentaires à inclure dans les unités fonctionnelles du processeur
- de séparer le code de l'application en un ensemble de fonctions critiques et le reste du code, appelé « code de contrôle ».

Elle s'appuie sur l'analyse dynamique de l'exécution du code de référence de l'application qui fournit des informations concernant le taux d'utilisation des **opérateurs** arithmétiques utilisés dans le code source (cf. section suivante), et le nombre d'appels à ces opérateurs pour chaque fonction de l'application. La phase de sélection des opérateurs repose sur un certain type de description des applications utilisant une bibliothèque d'opérateurs de base pour coder tous les calculs effectués dans l'application (cf. 4.4.1.1), et permet de ne sélectionner que les opérateurs réellement utiles pour assurer les performances requises par l'application. Cette phase constitue une première étape vers la spécification des unités fonctionnelles à inclure dans le processeur. Elle est décrite en détail dans la prochaine section.

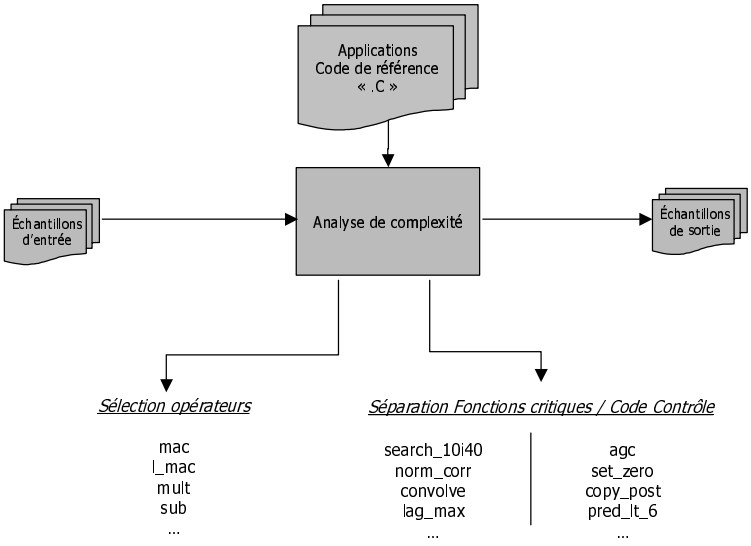


Figure 87 Analyse de complexité

L'identification des fonctions critiques de l'application permettra lors de la deuxième étape de la méthodologie de focaliser l'effort d'analyse et d'optimisation sur les parties critiques de l'application, c'est à dire celles qui requièrent le plus de puissance de calcul. Sachant que pour les applications DSP 80% du temps de calcul résulte de l'exécution d'à peine 20% du code, un bon moyen de réduire le temps de conception d'un ASIP consiste à n'optimiser que les fonctions critiques de l'application. La désignation des fonctions critiques est à la charge de l'utilisateur, qui en fonction de la charge de calcul calculée pour chaque fonction désignera comme « critiques » les *N* fonctions les plus

exigeantes, de telle sorte que la somme des *N* charges associées à ces fonctions soit supérieure à un certain seuil.

Dans la première phase de l'étape de **codage**, les fonctions critiques sélectionnées sont analysées et programmées manuellement en assembleur de manière à optimiser leurs performances. Ce choix d'introduire une étape de codage assembleur se veut réaliste, compte tenu du fait qu'il n'existe pas encore à l'heure actuelle de structures de compilation capables de produire du code de qualité égale à celle obtenue manuellement. De surcroît, cette étape constitue aussi une phase d'analyse par le concepteur de la structure des fonctions critiques à optimiser, qui peut le conduire à ajouter au processeur des instructions supplémentaires et/ou des unités fonctionnelles spécialisées qui permettent de pousser plus loin l'accélération de ces fonctions. En faisant appel à l'intelligence et à l'expérience du concepteur, l'étape de codage manuel offre des degrés d'optimisation potentiels beaucoup plus vastes que ceux obtenus uniquement avec un compilateur.

Dans un premier temps, on suppose que le modèle du processeur dispose de ressources générales infinies : nombre de registres, bande passante mémoire, nombre d'unités fonctionnelles générales (ALU/MAC/Décaleurs), etc. En fonction des caractéristiques du code assembleur (parallélisme des instructions, nombre de registres utilisés, bande passante mémoire utilisée, etc.), on détermine les valeurs à fixer aux différents paramètres du modèle, ainsi que les éventuelles instructions utilisateurs et unités fonctionnelles spécialisées à ajouter au processeur.

Une fois le modèle de processeur entièrement défini (valeurs des paramètres + ajouts utilisateur), le reste du code de l'application est obtenu par compilation du code de contrôle. Cela suppose l'existence d'un compilateur « reciblable », qui soit capable de générer du code pour différentes architectures, définies par les différentes valeurs des paramètres du modèle.

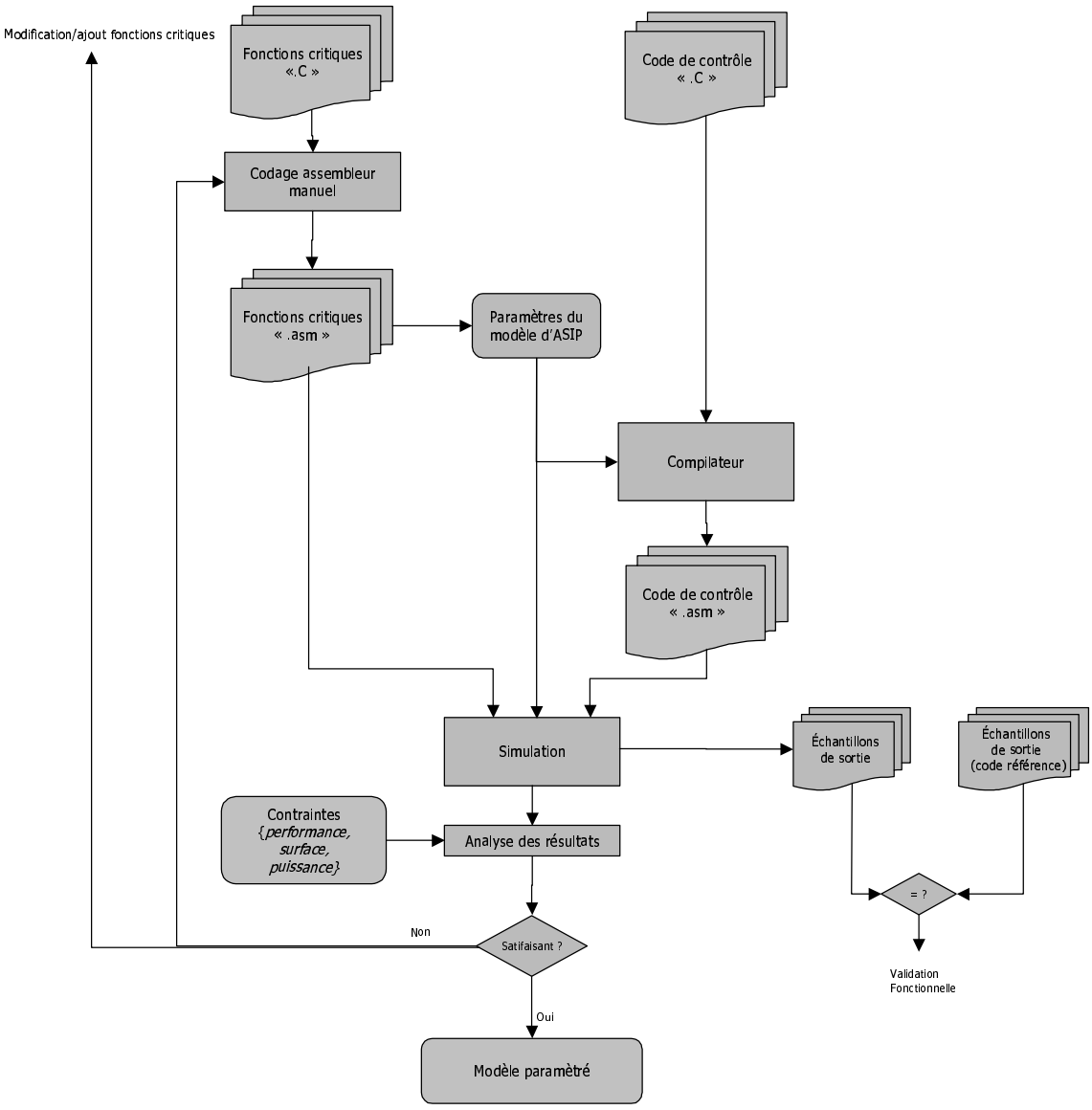


Figure 88 Codage des fonctions critiques et analyse des résultats

Dans la dernière étape de **simulation** et d'**analyse des résultats**, les deux portions de code résultant de la compilation et du codage manuel sont rassemblées. Le code complet est alors simulé sur le modèle de processeur au moyen d'un simulateur de jeu d'instructions reproduisant le comportement exact du processeur. Cette simulation permet d'une part de valider fonctionnellement le code en comparant les échantillons de sortie obtenus par rapport à ceux issus de l'exécution du code source de l'algorithme d'origine. D'autre part, elle renseigne sur les performances réelles de l'application en fournissant le nombre de cycles nécessaire à l'exécution de l'ensemble de l'application.

La phase d'analyse des résultats symbolise la prise de décision finale de l'utilisateur, qui en fonction des contraintes initiales, du nombres de cycles obtenus, de la taille du code généré et éventuellement d'autres métriques fournis par des outils d'analyse spécialisés (estimation du coût en surface et de la puissance dissipée), va décider de conserver le modèle courant du processeur et de passer à l'étape suivant de synthèse matérielle, ou de chercher une meilleure solution. Dans ce cas, cela consiste soit à reprendre l'étape de codage en modifiant les paramètres du modèle (et donc le code assembleur des

fonctions critiques), soit à remonter jusqu'à la phase d'analyse de complexité et modifier la sélection des fonctions critiques et des opérateurs.

Les prochaines sections présentent l'étape d'analyse de complexité pour les fonctions GSM, puis les outils logiciels (simulateur et compilateur) utilisés dans la phase de codage et d'analyse des résultats. L'application de la phase de codage et d'analyse pour les fonctions GSM sera présentée dans le prochain chapitre.

4.4.1 Analyse de complexité

Nous présentons ici en détail l'étape d'analyse de complexité utilisée au début de la phase d'exploration de l'espace de configuration, et qui a pour but la sélection des opérateurs arithmétiques à implémenter dans le processeur et la détermination des fonctions critiques de l'application. L'analyse présentée ici porte sur l'implémentation sur processeur DSP de la chaîne de traitement audio de la norme GSM.

La répartition de la charge de calcul au sein de l'interface radio GSM est très variable selon les blocs fonctionnels, certains réclamant beaucoup plus de puissance que d'autres. Les résultats de la Figure 89 correspondant à la programmation des fonctions GSM sur le processeur *R.E.A.L* de *Philips* montrent que les deux tâches les plus exigeantes en puissance de calcul sont le codage de voix et l'égalisation de Viterbi (utilisée dans le démodulateur). L'effort d'optimisation devra donc surtout porter sur l'accélération de ces deux fonctions, que nous allons maintenant examiner plus en détail.

Fonctions	***Charge de calcul (MIPS)***
Egalisation	2.5
Codage canal	0.5
Codage de voix (EFR)	11.5
Divers	2
Programme principal	1.5
Total	18

Figure 89 Fonctions audio GSM pour le DSP R.E.A.L.de Philips [56]

4.4.1.1 L'algorithme de codage de voix EFR

L'algorithme de codage de voix *EFR* [57] est parmi les trois algorithmes possibles celui qui offre les meilleurs résultats en terme de qualité sonore. Le nombre d'informations à transmettre en sortie du codeur reste le même (13 kbits/s) que celui de l'algorithme *FR*, la différence résidant uniquement dans la complexité de l'algorithme : 2.5 MIPS pour le *FR* contre 11.5 pour l'*EFR* [56]. Comme dans la plupart des algorithmes de codage, c'est la partie encodage qui est la plus complexe, réclamant environ 10 fois plus de calculs que la partie décodage.

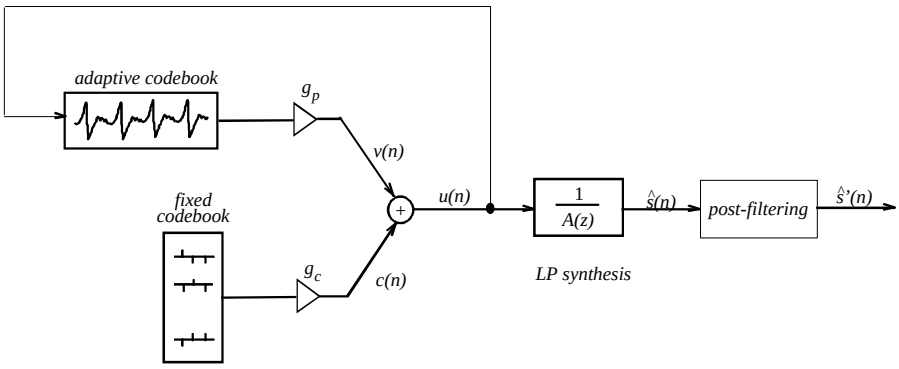


Figure 90 Structure du décodeur EFR

Le codage EFR est une version modifiée du codage CELP (*Code Excited Linear Prediction*). La philosophie du codeur CELP est d'analyser la parole, et ensuite de transmettre des paramètres au décodeur afin qu'il puisse reconstituer le signal d'origine aussi fidèlement que possible. Ces paramètres incluent le modèle mathématique d'un filtre linéaire qui simule le conduit vocal de la personne qui parle (toutes les caractéristiques qui rendent reconnaissable immédiatement une personne), et des informations relatives à un signal d'excitation du filtre. La principe du décodeur consiste alors à exciter le filtre linéaire par le signal d'excitation pour restituer la voix. Le signal d'excitation est modélisé par la somme de deux composantes, l'une stochastique (c'est à dire de bruit) et l'autre périodique. Les deux composantes sont transmises sous formes de deux index de code pointant vers des séquences d'échantillons de voix préétablis, appelés vecteurs, qui sont communs à l'émetteur et au récepteur. Ces vecteurs sont rangés dans des tables de code (*Adaptative* et *Fixed codebook*). La sélection des vecteurs pertinents se fait par une recherche basée sur la minimisation de l'erreur entre le signal original et le signal reconstruit.

L'encodeur opère sur des trames de parole de durée 20 ms correspondant à 160 échantillons (fréquence d'échantillonnage du CAN : 8kz), analyse la séquence d'échantillons et transmet les paramètres du modèle CELP : les coefficients du filtre linéaire, et les index et gains correspondant aux deux tables de codes, représentant un total de 244 bits, soit un débit en sortie de 12.2 kbits/s.

4.4.1.2 L'algorithme de Viterbi

Cet algorithme a été proposé pour la première fois en 1967 par A.J. Viterbi pour résoudre le problème du décodage des codes convolutionnels dans les systèmes de communication numériques [81]. Il est utilisé dans la plupart des systèmes de communication actuels (GSM, xDSL, UMTS et réseaux locaux de type « Wireless ») ainsi que dans d'autres domaines n'ayant rien à voir avec les communications telles que la reconnaissance de forme ou l'acquisition de cibles. Dans la norme GSM, il est utilisé à la fois dans le démodulateur transformant le signal analogique issu de la partie réception en signal numérique, ainsi que dans la partie « décodage canal » du récepteur. Pour le démodulateur, on parle de *l'égaliseur de Viterbi* (« Viterbi equalizer ») et pour le décodage canal du *décodeur de Viterbi* (« Viterbi decoder »).

Comme on peut le voir sur la Figure 89, la partie « égalisation » de la norme GSM représente une part non négligeable de la charge de calcul sur le processeur REAL, qui possède pourtant une unité câblée spécialisée pour accélérer le traitement de l'algorithme. Estimée à 2.5 MIPS, on peut estimer que la

charge requise par un processeur ne disposant pas d'unité spécialisée est au moins le double, soit 5 MIPS, ce qui représente environ un quart de la charge de calcul totale requise par les fonctions GSM. Pour un processeur DSP destiné aux application de communication, il est donc indispensable d'offrir de bonnes performances sur cet algorithme.

4.4.1.3 La bibliothèque d'opérateurs arithmétiques de l'ETSI

L'organisme de standardisation *ETSI* (*European Telecommunication Standard Institute*) fournit les sources C de l'EFR ainsi que les séquences de test qui permettent de valider le comportement d'une implémentation au bit près [58] [59]. Toutes les opérations arithmétiques décrites dans le code C sont effectuées par des appels à une bibliothèque d'opérateurs arithmétiques prédéfinis (cf. Annexe A et Figure 91). L'utilisation de ces fonctions par rapport à l'emploi des opérateurs par défaut du langage C a plusieurs avantages. D'une part, les types de données et l'arithmétique associés à chaque opérateur sont entièrement spécifiés au bit près, ce qui lève toute ambiguité concernant le comportement bit-précis des opérations lorsqu'on compile sur des architectures différentes. Dans le cas de l'EFR, les opérateurs travaillent sur des données de largeur 16 (*mac*) ou 32 bits (*L_mac*), et un certain nombre d'entre eux gèrent automatiquement la précision des résultats. L'opération *L_mac* par exemple effectue une multiplication-accumulation suivie de la saturation automatique du résultat sur 32 bits pour éviter les problèmes de débordement. D'autre part, l'emploi de ces fonctions pour la spécification des algorithmes permet de simplifier grandement le portage sur les processeurs DSP, les 41 opérateurs de la bibliothèque ayant un équivalent direct dans presque tous les jeux d'instructions des DSPs. Dans le cas où ces fonctions sont définies comme intrinsèques, elles permettent en plus d'améliorer grandement l'efficacité des compilateurs (cf. 4.4.3.3).

```
for (i = 0; i < lg; i++)
  {
     s = L_mult (x[i], a[0]);
     for (j = 1; j <= m; j++)
     {
        s = L_msu (s, a[j], yy[-j]);
     }
     s = L_shl (s, 3);
     *yy++ = round (s);            move16 ();
  }
```

Figure 91 Usage des opérateurs ETSI dans la boucle principale de la fonction « syn_filt.c »

4.4.1.4 Analyse de complexité et processeur de référence

Lorsque l'application cible est définie à l'aide d'une bibliothèques d'opérateurs de base, l'analyse de complexité consiste à compter pour chaque fonction de l'arborescence (une cinquantaine pour *l'EFR*) le nombre d'appels à ces opérateurs, qui représentent pour la plupart l'équivalent d'un cycle instruction sur les processeurs DSP. Les opérateurs les plus utilisés seront sélectionnés pour être implémentés dans le jeu d'instruction du processeur, et les fonctions totalisant le plus grand nombre d'opérations par appel formeront l'ensemble des fonctions « critiques » soumises à optimisation dans l'étape de codage manuel en assembleur.

L'analyse de complexité se base sur un modèle de processeur DSP « mono-scalaire » formé d'une seule unité fonctionnelle « universelle » capable d'exécuter n'importe lequel des opérateurs de base. Le nombre de cycles associé à l'exécution d'un opérateur dépend du type de l'opérateur. Dans un premier temps, on se base sur un modèle « crédible » de DSP capable d'exécuter en un cycle tous les opérateurs arithmétiques classiques (*add*, *sub*, *mul*…), et émulant logiciellement les opérateurs plus complexes pour lesquels on suppose qu'il n'existe pas de matériel associé (les opéraeurs de normalisatio *norm_s* et *norm_l* , et de division *div_s*). Chaque opérateur de la librairie est donc pondéré par une valeur correspondant au nombre de cycles nécessaire à son exécution sur le DSP référence (cf. Figure 92). La valeur « test » correspond au coût moyen en cycles d'une structure de type *if-then-else* conformément au modèle de pipeline du processeur présenté au chapitre précédant. Les boucles *for* ne sont pas prises en compte dans le décompte car on suppose que la plupart peuvent faire usage du mécanisme de boucles « zero-cycle » du processeur.

Opérateurs	*Nombre de cycles*
Arithmétiques	1
Logiques	1
Transferts	1
norm_s	100
norm_l	165
div_s	78
test	7

Figure 92 Pondération des opérateurs ETSI pour le processeur de référence

L'analyse de complexité se base sur le comportement dynamique de l'application et nécessite donc d'exécuter réellement l'application, avec en entrée du système encodeur/décodeur des échantillons sonores représentatifs. Pour ce faire, le code source de référence C de l'*EFR* est compilé puis exécuté sur station de travail avec comme données d'entrée un fichier de test fourni par l'*ETSI* correspondant à une séquence sonore « réelle » (Figure 93). Parmi les 20 fichiers de test disponibles, on a choisi le fichier numéro 2 qui correspond à la plus grande charge de calcul : 14.55 MCPS (Millions de cycles par seconde, qui correspondent aux cycles machine estimés pour chaque opérateur) pour l'encodeur, et 1.68 MCPS pour le décodeur. Dans le code C de l'application EFR ont été rajoutées des fonctions de comptage destinées à fournir des statistiques d'utilisation des opérateurs pour chaque sous-fonction de l'arborescence : l'instruction *move16()* du code de la Figure 91 n'a aucune action sur l'algorithme proprement dit mais sert à incrémenter une valeur de comptage correspondant à l'opérateur de transfert *DataMove16*.

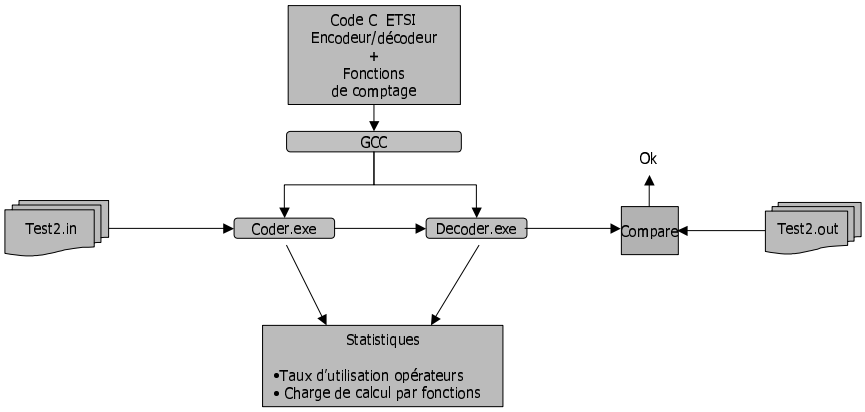


Figure 93 Extraction des caractéristiques dynamiques de l'application

L'inconvénient de cette méthode est qu'elle requiert de l'application cible l'utilisation d'une bibliothèque d'opérateurs prédéfinie et l'annotation de l'ensemble du code avec les fonctions de comptage des opérateurs. Dans le cas de l'EFR, le problème ne se pose pas puisque l'application est codée avec la bibliothèque ETSI et les fonctions de comptage y sont intégrées directement. Il n'en est malheureusement pas de même pour la plupart des programmes source d'application DSP, qui sont souvent codées en langage « naturel » utilisant les seuls opérateurs du langage C (« + », « - » ,« * »,« / »,etc.) et n'intégrant aucune fonction de comptage. Cependant, si l'application doit faire l'objet d'un portage sur processeur DSP, l'étape de réécriture du code source à l'aide d'une bibliothèque d'opérateurs arithmétiques « bit-précis » est quasi-obligatoire pour tester et valider l'implémentation en « entier largeur fixe » au regard de caractéristiques telles que la dynamique du signal obtenu, les effets des débordements, le rapport signal/bruit, etc.

Une autre méthode d'analyse existe qui consiste à compiler l'application complète sur le processeur de référence à l'aide du compilateur reciblable (et non plus sur une station de travail à l'aide de *gcc*). L'avantage est que l'application tourne réellement sur le modèle du processeur de référence et permet d'obtenir des statistiques d'exécution plus précises tenant compte des effets du pipeline de contrôle et du coût réel lié à l'absence de certains opérateurs émulés logiciellement (les opérateurs implémentés matériellement étant définis dans la table des mnémoniques associée à l'unité fonctionnelle « universelle » du processeur de référence, cf. 3.3.1). Cette méthode est indiscutablement meilleure, mais l'état d'avancement du compilateur reciblable ne nous a pas permis de l'utiliser. Les chiffres présentés ci-après concernent donc l'analyse par annotation du code avec des fonctions de comptage.

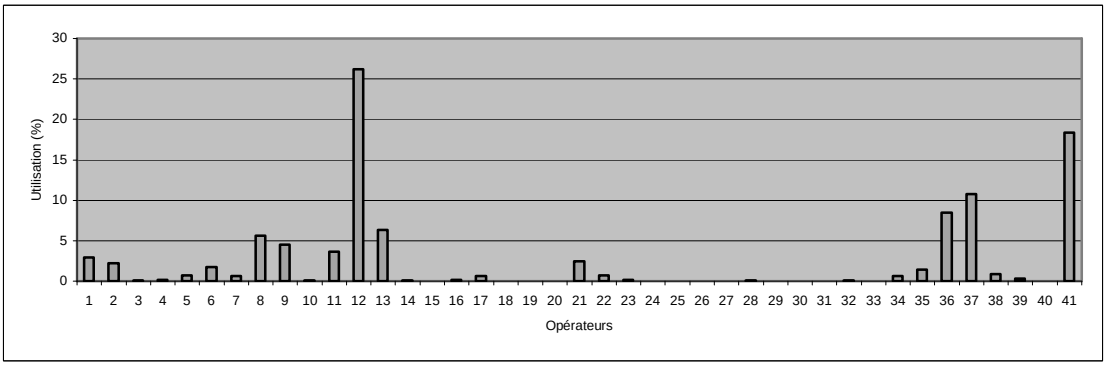


Figure 94 Histogramme d'utilisation des opérateurs

4.4.1.5 Sélection des opérateurs

La bibliothèque ETSI utilisée pour coder l'EFR comporte 41 opérateurs correspondant à des opérations de diverses natures (arithmétiques et logiques, transfert de données, tests) et de complexité variée (de la simple opération logique à la division entière). Ils ont été choisis avant tout pour leur représentativité des opérations effectuées dans les processeurs DSP dans l'implémentation d'applications typiques appartenant au domaine DSP. Parmi eux, seule une partie servent pour la description fonctionnelle de l'algorithme proprement dit (essentiellement les opérateurs arithmétiques), les autres ne sont utilisés qu'à des fins statistiques pour comptabiliser le nombre de transferts de données (exemple de l'opérateur *move16()* de la Figure 91) ou estimer le coût des structures de contrôle (opérateur *test()*).

La Figure 94 montre l'histogramme d'utilisation de l'ensemble des opérateurs de la bibliothèque ETSI pour le système encodeur/décodeur de l'EFR. Ces chiffres représentent le taux d'utilisation des opérateurs par rapport à la charge totale de calcul nécessaire pour encoder puis décoder le fichier de test numéro 2 (402 séquences de 20ms, soit environ 8 secondes de parole), chaque opérateur étant pondéré par la valeur estimée du nombre de cycles nécessaire à son exécution sur le processeur de référence. La principale information que l'on peut en tirer est la distribution très inégale du taux d'utilisation selon les opérateurs, une dizaine d'entre eux seulement comptant pour presque 90% de la charge de calcul totale (cf. Figure 95), tandis qu'une dizaine d'autres ne sont jamais utilisés. La contribution des quatre opérations de multiplication/accumulation les plus utilisés (*l_mac*, *l_msu*, *l_mult*, *mult*) totalise 43% de la charge de calcul, prouvant si il en était besoin la prépondérance de ce type d'opérations dans les algorithmes DSP.

l_mac	*Test*	*DataMove16*	*norm_l*	*l_msu*	*mult*	*l_mult*	*round*	*add*	*l_shl*	*sub*	*extract_h*	*div_s*
26.23	18.37	10.80	8.47	6.34	5.62	4.48	3.67	2.90	2.48	2.21	1.71	1.41
DataMove32	***shr***	***l_shr***	***norm_s***	***extract_l***	***l_sub***	***logic16***	***l_add***	***shl***	***mult_r***	***abs***	***l_abs***	***negate***
0.89	0.74	0.68	0.62	0.62	0.62	0.34	0.18	0.16	0.15	0.07	0.06	0.06
l_deposit_h	***l_macNS***	***l_shr_r***	***l_deposit_l***	***l_negate***	***l_msuNS***	***l_add_c***	***l_sub_c***	***shr_r***	***shl_r***	***mac_r***	***msu_r***	***l_shift_r***
0.05	0.05	0.03	0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
l_sat	***logic32***											
0.00	0.00											

Figure 95 Taux d'utilisation des opérateurs pour l'EFR (encodeur + décodeur)

Alors qu'un processeur DSP général se verrait contraint à l'implémentation de l'ensemble de ces opérateurs (mis à part quelques opérateurs très coûteux et rarement utilisés comme la division) pour assurer une performance satisfaisante sur l'ensemble du spectre des applications DSP, un processeur ASIP peut se permettre de ne sélectionner que les opérateurs les plus utiles pour l'application cible. La diminution du nombre global d'opérateurs implémentés contribue à la réduction du nombre d'opcodes utilisés pour coder les opérations des unités fonctionnelles, et favorise donc indirectement une meilleure compacité du code. La suppression de certains opérateurs contribue aussi à simplifier la structure matérielle des unités fonctionnelles du chemin de données. Sachant que ces dernières ont souvent une part importante dans la chaîne critique des processeurs DSP, la simplification matérielle peut permettre d'atteindre des fréquences de fonctionnement plus hautes et donc d'accroître les performances.

Le processus de sélection des opérateurs va consister à la fois à choisir les opérateurs et à les répartir dans les différents types d'unités fonctionnelles, cette répartition pouvant toutefois être remise en cause lors de la phase de codage manuel en assembleur si elle est incompatible avec l'exploitation optimale du parallélisme de l'application. Nous supposons un modèle de processeur de départ incluant trois types d'unités fonctionnelles :

- des unités MAC chargés de tous les calculs de type Multiplication/accumulation.
- des ALUs chargées des opérations logiques et des autres opérations arithmétiques non prises en compte par les MACs.
- des décaleurs chargés de toutes les opérations de décalage

La Figure 96 montre la répartition des opérateurs de la bibliothèque dans ces différentes unités. Cette répartition n'est pas définitive et pourra être modifiée plus tard dans la méthodologie. Parmi les 41 opérateurs de départ, seuls subsistent 22 opérateurs utiles répartis sur les trois types d'UFs précédemment cités plus un quatrième type regroupant les opérateurs réclamant un autre type d'UFs. Parmi les 19 opérateurs supprimés figurent les 10 dont le taux d'utilisation est nul et qui « encombrent » donc le jeu d'instruction. S'ajoutent à cela des opérateurs dont les taux d'utilisation sont très faibles (<< 1%) et qui peuvent facilement être émulés logiciellement à l'aide d'autres instructions plus simples. C'est le cas par exemple de tous les opérateurs complexes combinant une

opération arithmétique suivie d'un arrondi (opérateurs avec suffixe en « *r* ») dont les taux d'utilisation sont pour la plupart nuls ou très faibles et qui peuvent être remplacés par deux opérateurs séquentiels, le premier effectuant l'opération arithmétique et le deuxième l'arrondi (opérateur *round()* de l'ALU). Dans le cas de *mult_r()*, la suppression de cet opérateur permet d'éviter l'intégration d'un opérateur d'arrondi dans une unité MAC, qui aurait tendance à accroître la complexité de l'unité la plus critique du chemin de données.

Les opérateurs *negate()* et *l_negate()* peuvent être remplacés par une opération de soustraction avec la valeur zéro. De par leur très faible utilisation, l'influence de la suppression de ces opérateurs sur la performance du processeur pour l'application donnée devrait être faible, d'autant plus que le parallélisme d'instruction offert par la structure VLIW permet dans beaucoup de cas de masquer le découpage des opérateurs complexes en sous-opérateurs en effectuant les sous-opérations en parallèle, n'occasionnant ainsi aucune perte en cycles machine. D'autres opérateurs comme *l_abs()*, bien que peu utilisés, ne gagnent pas à être supprimés car leur suppression n'occasionnerait qu'une réduction très minime de la complexité matériel *(l_abs()* utilisant le soustracteur de l'ALU, la seule économie porterait sur le multiplexeur de sélection des deux valeurs positive et négative) mais serait très coûteuse à émuler logiciellement (soustraction, test, affectation du résultat avec structure *if-then-else*).

MAC	***ALU***	***Shifter***	***Autres***	***Supprimés***
mult	add	shl	norm_s	extract_h
l_mult	sub	shr	norm_l	extract_l
l_mac	abs	l_shl	DataMove16	negate
l_msu	round	l_shr	DataMove32	l_msuNS
l_macNS	l_add		Test	l_add_c
l_add	l_sub			l_sub_c
l_sub	l_abs			l_negate
add	logic16 (OR,XOR,AND,NOT)			mult_r
sub				shr_r
				shl_r
				mac_r
				msu_r
				l_deposit_h
				l_deposit_l
				l_shr_r
				l_shl_r
				l_sat
				div_s
				logic32

Figure 96 Répartition des opérateurs dans les différentes unités fonctionnelles

L'opérateur complexe de division euclidienne *div_s()* est supprimé car il est trop rarement utilisé pour faire l'objet d'une implémentation matérielle, l'accélération qui en résulterait étant négligeable par

rapport au coût matériel engagé. Ce n'est par contre pas le cas des opérateurs de normalisation *norm_l()* et *norm_s()* , comptant pour presque un dixième de la charge de calcul et qui gagneraient à être calculés matériellement, leur coût d'émulation étant particulièrement élevé (voir Figure 92) à cause des structures de test présentes dans le calcul. Ces opérateurs ne correspondant pas vraiment au type de calculs effectués dans les trois types d'unités prédéfinies, on les range pour le moment dans la catégorie « Autres unités fonctionnelles».

Certains opérateurs comme l'addition ou la soustraction peuvent à priori être effectuées indifféremment dans une unité ALU ou MAC. La suppression de ces opérateurs d'une des deux unités permettrait de faire l'économie de deux opcodes, néanmoins, il peut être pratique de conserver ces fonctionnalités dans les deux unités pour exploiter plus de parallélisme d'instructions. La décision de supprimer ou non ces opérateurs sera donc prise plus tard, au moment de l'optimisation des fonctions critiques.

Certains résultats tendent aussi à justifier des choix architecturaux faits au chapitre précédant. L'utilisation non négligeable des opérateurs de transferts entre variables simple et double précision (opérateurs *deposit* et *extract* représentant 2.5% de la charge de calcul) montre l'intérêt d'un banc de registres central à accès simple et double précision, qui permet de se passer totalement de ces opérateurs et de réduire ainsi la charge de calcul globale. De même, le temps passé dans les structures de test « if-then-else » (18%) démontre l'intérêt des mécanismes d'instructions retardées et d'exécution conditionnelle qui permettent de réduire radicalement le nombre de cycles nécessaires à l'exécution de ces structures (jusqu'à un facteur 5, voir Figure 82 page 109).

La méthode choisie par l'ETSI pour l'annotation du code avec les fonctions de comptage des transferts de données (variables *DataMove16* et *DataMove32*) consiste dans la grande majorité des cas à ne comptabiliser que les transferts de données formant à eux seuls une ligne de code C (par exemple une affectation du type variable1=variable2). Par contre, les lignes de code combinant plusieurs opérations (exemple : « ps1 = add (ps0, dn[i4]); » : une lecture mémoire plus une opération arithmétique) ne comptent que comme une instruction (une opération « add » dans le cas de l'exemple précédant). En somme, l'annotation du code repose sur un modèle implicite de processeur à accès mémoire direct, comme celui que nous proposons pour le modèle d'ASIP. Pour mesurer l'impact de ce type d'architecture par rapport à une architecture « Load/Store », nous avons modifié l'annotation d'une des fonctions critiques de l'encodeur pour comptabiliser tous les transferts mémoire. Il en résulte une augmentation de près de 30% de la charge de calcul de la fonction, les accès mémoire représentant près de 22% du nouveau total, ce qui est conforme avec les résultats moyens observés pour les applications DSP [12]. Du coup, l'architecture à accès direct du modèle d'ASIP devrait pouvoir être pleinement exploitée, et permettre une augmentation de la performance justifiant l'accroissement de la complexité du processeur.

4.4.1.6 Détermination des fonctions critiques

Les fonctions de comptage de l'ETSI ne fournissant des statistiques que sur l'ensemble de l'application et non fonction par fonction, nous avons modifié le code source de manière à obtenir ces statistiques pour chaque fonction de l'arborescence afin de déterminer lesquelles doivent être optimisées en priorité. L'encodeur et le décodeur utilisent au total 66 fonctions de diverses natures : fonctions de traitement du signal générales (filtres, convolution, calcul de résidu, autocorrélation,

etc.), fonctions mathématiques (puissance, racine carrée, logarithme), fonctions générales (copie mémoire, initialisation de tableaux), et fonctions spécifiques à l'EFR. Les résultats de l'analyse de complexité par fonction sont illustrés sur la Figure 97 pour les 14 fonctions les plus critiques. A elles seules, ces fonctions représentent 84% de la charge totale de calcul. Leur complexité est variable, variant de quelques lignes pour des fonctions DSP simples de type filtre (« *syn_filt* ») à plusieurs dizaines de lignes pour des fonctions spécifiques complexes (« *search_10i40* »), certaines font aussi appels à des sous-fonctions. Principalement de type DSP, elles consomment la majeure partie des calculs dans des boucles *for* de faible longueur répétées de nombreuses fois. Conformément à la méthodologie, c'est sur ces fonctions que va se concentrer l'effort d'optimisation, conduisant à la spécification précise de l'architecture de l'ASIP et de son jeu d'instructions.

Fonctions	*%*
search_10i40	21.7
norm_corr / convolve	12.5
lag_max	7.7
cor_h	6.6
syn_filt	6.6
az_lsp	6.6
Vq_subvec	4.4
Vq_subvec_s	3.3
Residu	2.8
Pred_lt_6	2.6
Levinson	2.6
Autocorr	2.5
Set_sign	2
cor_h_x	2
Total	83.9

Figure 97 Fonctions critiques de l'algorithme EFR

4.4.2 Simulateur de jeu d'instructions

Utilisé dans la dernière étape de la phase d'exploration, le simulateur d'instructions est l'outil qui permet de tester et de valider le comportement du code assembleur de l'application sur le modèle d'ASIP configuré. La simulation du code assembleur prend en entrée les mêmes séquences de test que le code C de référence, et les échantillons produits en sortie sont comparés à ceux issus de l'analyse de complexité pour valider la fonctionnalité au bit près. Le simulateur fournit les performances de l'application en nombre de cycles machine, permettant de comparer plusieurs architectures et de guider les choix du concepteur lors de la spécification des caractéristiques du processeur final.

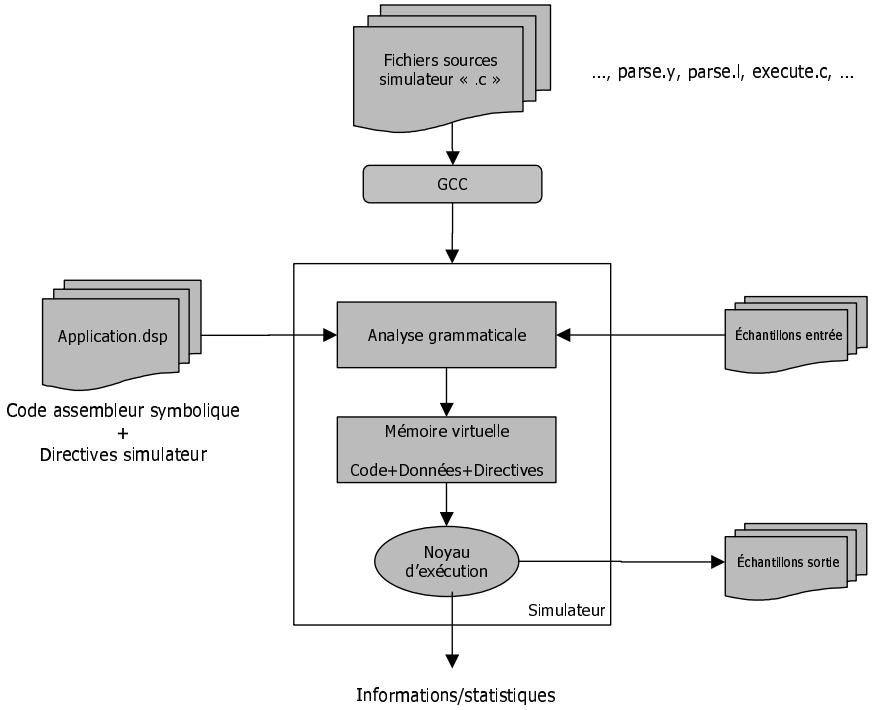


Figure 98 Fonctionnement du simulateur

4.4.2.1 Réalisation logicielle

L'ensemble des outils logiciels utilisés pour la mise en œuvre de la phase d'exploration fonctionne sous environnement UNIX et est essentiellement composé de programmes écrits en langage C++ (simulateur et compilateur) et de quelques scripts en shell UNIX. Le fonctionnement du simulateur comme du compilateur dépend de l'architecture du processeur cible et doit donc prendre en compte les paramètres de configuration du modèle. La prise en compte de ces paramètres peut être statique (nécessité de recompiler les outils à chaque changement des paramètres) ou dynamique. La deuxième solution repose sur l'utilisation d'un fichier unique décrivant complètement les différents paramètres du modèle de processeur (architecture et jeu d'instructions) et qui est pris en compte par les différents outils à chaque étape de la méthodologie. Les recherches sur la compilation reciblable ont donné lieu à plusieurs propositions de langages de description de processeur, les plus connus étant le langage nML[60] associé au compilateur CHESS[61], le langage ISDL[62] utilisé par le compilateur AVIV[63], et le langage LISA[64] utilisé pour la production de modèles de processeurs « cycle-précis ».

L'utilisation d'un tel langage offre beaucoup plus de souplesse et de confort d'utilisation mais nécessite un effort de développement important et n'apporte rien de plus à la méthodologie proprement dite. Nous avons donc fait le choix de la configuration statique, chaque changement dans la description du processeur cible nécessitant une modification des sources puis une recompilation du simulateur et du compilateur. Nous verrons par la suite que ces changements sont en général simple à mettre en œuvre et ne nécessitent pas de gros efforts de programmation.

4.4.2.2 Modèle d'exécution par défaut

L'exécution d'un programme assembleur sur le simulateur commence par l'analyse grammaticale du fichier source (« *.dsp »). Les règles définissant la syntaxe et la grammaire du langage assembleur sont contenues dans deux fichiers *Lex* et *Yacc* (« parse.l » et « parse.y »). Par défaut, ces fichiers contiennent la description du langage assembleur associé au processeur de référence du simulateur. Ce processeur possède un jeu d'instructions de base incluant toutes les instructions de contrôle (unité PCU) et les modes d'adressages (unité AGU) définies au chapitre précédant, ainsi qu'un certain nombre d'opérations arithmétiques, logiques et de décalages qui correspondent aux opérateurs exécutables « en un cycle » du processeur de référence de l'analyse de complexité (cf. 4.4.1.4). La différence par rapport à ce dernier réside dans le parallélisme matériel (unités fonctionnelles, unités de calcul d'adresse, bande passante mémoire) qui est dans un premier temps supposé infini afin de pouvoir tester différentes configurations de parallélisme. De même, les unités fonctionnelles de l'unité CU ne sont pas encore différenciées mais sont toutes « universelles », c'est à dire qu'elles sont capables d'exécuter n'importe laquelle des opérations CU définies dans le jeu d'instructions. Enfin, le simulateur suppose par défaut qu'il n'existe aucune contrainte de connectivité entre les différents unités de calcul et les éléments mémorisant du processeur, chaque opérande source d'une instruction pouvant provenir de n'importe quel registre.

Ce modèle « infini » et « universel » correspond à la structure de jeu d'instructions générique du chapitre précédant qui ne fixe aucune limite pour le nombre d'opérations exécutables par instruction, et aucune contraintes inter-opérations. Il trouve son utilité dans les premières phases de programmation en assembleur, pendant lesquelles le programmeur va tester différentes options de parallélisme matériel et de répartition des opérations dans les unités fonctionnelles.

4.4.2.3 Mémoire virtuelle et directives de simulation

Les fichiers d'entrée du simulateur peuvent contenir des instructions assembleur et des directives spécifiques au simulateur. L'analyse grammaticale du fichier d'application et du fichier contenant les échantillons d'entrée débouche sur l'initialisation d'une structure de données particulière modélisant la « mémoire virtuelle » du processeur, dans laquelle chaque entrée correspond à une instruction du jeu d'instruction du processeur, une donnée simple précision ou une directive spéciale de simulation qui sera interprétée par le simulateur lors de l'exécution.

A ce niveau, on ne fait encore aucune distinction entre mémoire « Programme » et mémoire « Données », chaque zone de l'espace adressable (infini dans le modèle par défaut) pouvant être de l'un ou l'autre des deux types. Un emplacement mémoire est une structure de données possédant six champs principaux :

- trois champs « AGU_instr », « CU_instr » et « PCU_instr » mémorisant les instructions associées à chaque unité du processeur (on rappelle qu'une « instruction » d'une unité contient une ou plusieurs « opérations » élémentaires). Ces champs ne sont actifs que dans une zone mémoire de type « Programme ».
- un champ « Data » mémorisant la valeur d'une donnée simple précision, le stockage d'une donnée double précision nécessitant deux emplacements consécutifs. Ce champ n'est actif que dans une zone mémoire de type « Donnée ».

- un champ « Label » mémorisant le label associé à l'adresse courante. Cela permet de définir des adresses symboliques facilitant le développement et la lisibilité du code assembleur.
- un champ « Debug_instr » mémorisant une instruction contenant une ou plusieurs directives de simulation

Contrairement aux simulateurs classiques, notre simulateur ne prend pas en entrée un fichier binaire formé du code « objet » de l'application (après l'assemblage du fichier source et l'édition de liens) mais directement le fichier source au format texte et décrit en assembleur symbolique. Comme le principal but du simulateur est de pouvoir tester différentes configurations d'architecture et de jeu d'instructions, il est évidemment impossible de définir un schéma d'encodage fixe pour les instructions du processeur, d'où le stockage et l'interprétation « symbolique » des instructions assembleur. L'encodage des instructions requiert la définition d'un format précis d'encodage ; cette définition sera effectuée par l'utilisateur au fur et à mesure de la phase d 'exploration en fonction des besoins de l'application. C'est dans la phase finale d'analyse des résultats que le format définitif sera pris en compte pour évaluer la taille de la mémoire « programme ».

Les directives de simulation de la Figure 99 peuvent être classée en trois catégories. La première concerne la définition et l'initialisation des zones de mémoires « Données » (directives *word16*, *word32*, *space* et *align*). Ce sont ces directives qui sont utilisés dans le fichier de test d'entrée, les échantillons étant rangés en mémoire virtuelle sous forme d'un tableau de valeurs. La deuxième permet l'envoi d'informations vers un fichier de sortie (*fopen*, *fclose*, etc.). La directive *dump_stats_register*() donne par exemple des informations sur l'utilisation des registres lors du déroulement du programme : durée de vie maximale (en cycles) de chaque registre, nombre maximum de registres en vie simultanément (et numéro du cycle d'exécution associé), date de la dernière utilisation, etc. Toutes ces informations sont très utiles dans la phase de programmation assembleur pour réduire le nombre de registres utiles. Les autres directives (*label* et *break*) servent respectivement à étiqueter les adresses mémoire et à fixer des points d'arrêt pour le débogage du programme.

Syntaxe	*Fonction*
.word16 val	Initialisation de la donnée simple précision contenue à l'adresse courante
.word32 val	Initialisation de la donnée double précision contenue à l'adresse courante
.space N	Réservation de N mots simple précision
.align N	Alignement sur une adresse multiple de N
label:	Définition d'un label (adresse symbolique)
fopen(fic)	ouverture d'un fichier de sortie fic
fclose()	fermeture du fichier courant
fprintf(string)	sortie texte vers fichier courant
fdumpmemory(adresse, N)	contenu du buffer mémoire d'adresse « adresse » et de longueur N vers fichier courant
fdumpreg(registre)	contenu de registre vers fichier courant
fdumpcycle()	affichage du numéro du cycle courant
dump_stats_register()	affichage statistiques sur l'utilisation des registres CU et AGU
.break	insère un point d'arrêt devant l'instruction suivante

Figure 99 Directives de simulation

4.4.2.4 Déroulement de la simulation et fichiers de résultats

La simulation proprement dite est prise en charge par le noyau d'exécution, qui lit et exécute les instructions et les directives présentes en mémoire virtuelle et génère en sortie les échantillons résultats et les informations et statistiques correspondant à l'exécution des directives de simulation. Le noyau utilise pour cela un ensemble de structures de données modélisant le comportement et l'état du processeur à chaque cycle d'exécution.

L'utilisateur dispose de deux modes d'exécution : normal et pas-à-pas. En mode normal, le programme correspondant au code assembleur est exécuté en intégralité jusqu'à la production du fichier de résultats, sans que l'utilisateur puisse l'arrêter (sauf en cas de point d'arrêt). Le deuxième mode sert principalement au débogage du programme et affiche à chaque cycle d'exécution le contenu des registres et des drapeaux du processeur, ainsi que les instructions se trouvant dans chaque étage du pipeline. L'utilisateur dispose aussi de commandes permettant de modifier le contenu des registres et d'afficher ou modifier le contenu de la mémoire « Données ».

Une directive *dumpmemory*() est en général insérée à la fin du fichier source pour copier la valeur des échantillons résultats vers le fichier de sortie, qui sera ensuite comparé avec le fichier référence pour valider le comportement du code.

4.4.2.5 Intégration du modèle de pipeline

Le modèle du processeur intègre toutes les informations concernant le fonctionnement du pipeline de contrôle et est donc capable de détecter les aléas de contrôle ou de données (cf. 3.6.6). Selon le type d'erreur, le simulateur affichera un message d'avertissement indiquant le type d'erreur (par exemple sur quel registre porte le conflit *RAW*) ou interrompra le déroulement en cas de conflit matériel (lecture et écriture simultanée sur le même port mémoire, écritures simultanées dans le même registre,

etc.). Comme on suppose que le compilateur intègre lui aussi un modèle du pipeline et qu'il génère donc du code « propre » vis à vis du pipeline, cette fonctionnalité est avant tout utile dans la phase de programmation manuelle des fonctions critiques de l'application. La recherche de performance pousse en général à paralléliser au maximum les traitements, multipliant les risques d'aléas et conduisant à un code non valide. La détection automatique par le simulateur évite au programmeur d'avoir à connaître tous les détails d'implémentation du pipeline et lui épargne le fastidieux travail consistant à localiser et reconnaître les aléas.

4.4.2.6 Ajout de nouvelles opérations et unités fonctionnelles utilisateurs

Tout l'intérêt de la méthodologie réside dans la possibilité d'évaluer différentes configurations d'architecture et de jeu d'instructions, dans lesquelles on pourra trouver des unités fonctionnelles et des instructions « utilisateur » destinées à accélérer telle ou telle partie de l'application cible. Pour cela, le simulateur doit pouvoir s'adapter à chaque changement matériel ou logiciel du processeur cible. En l'absence d'un langage de description de processeur et des outils de production de code associés, il est nécessaire de modifier les sources C++ du simulateur pour prendre en compte les changements.

La description fonctionnelle de toutes les opérations du jeu d'instructions est décrite dans un fichier particulier, « execute.c », dans lequel on trouve trois classes correspondant aux trois unités principales du processeur : CU, AGU et PCU. Chacune de ces classes possède une méthode *Compute*(*CU_operation op*) chargée d'exécuter l'opération passée en argument, et ce pour toutes les opérations associées à l'unité. Ajouter une nouvelle opération se fait alors en deux étapes :

- Ajout dans le corps de la méthode *Compute*() de l'unité concernée du code C++ décrivant le comportement de l'opération. Pour un opération de calcul de l'unité CU « *[CU] a1=div(a0,a1) ;* », on pourra créer une fonction *div*() appelée par *Compute*() et renvoyant le résulta du calcul en fonction de la valeur des opérandes sources.
- Ajout dans les fichiers « parse.l » et « parse.y » des règles de syntaxe de la nouvelle opération.

Une fois les modifications effectuées, il suffit de recompiler le simulateur pour prendre en compte les nouvelles opérations. Il est ainsi possible d'ajouter de nouvelles opérations de calcul, mais aussi de nouveaux modes d'adressages, les calculs correspondant étant cette fois décrits dans la méthode *Compute*() de l'unité AGU.

L'ajout d'une nouvelle unité matérielle dans l'unité CU est à peine plus compliqué. On a vu au chapitre précédant (cf. 3.3.1) qu'une unité matérielle était définie du point de vue du jeu d'instruction par un ensemble d'opérations ou *mnémoniques* de commande, les liens entre les opérations et l'unité matérielle correspondante étant déclarés au moyen d'une table spéciale, la **table d'allocation** des mnémoniques (cf. 3.6.2). Pour ajouter une nouvelle unité, il suffit de déclarer chacune des nouvelles opérations comme décrit précédemment puis d'ajouter dans la table d'allocation les entrées établissant les liens entre les opérations et l'unité concernée.

La déclaration des unités fonctionnelles du processeur et de la table d'allocation n'est pas indispensable : lorsque le noyau d'exécution du simulateur rencontre une opération non définie dans

la table d'allocation, il exécute l'opération correspondante (décrite sous forme d'une fonction C++) sans se soucier de la disponibilité d'un quelconque unité fonctionnelle associée.

Cependant, afin d'introduire la notion de parallélisme matériel, il est possible d'indiquer au simulateur le nombre d'unité fonctionnelle présentes réellement dans le chemin de données, et ce pour chaque type d'unité : ALU, MAC, BMU, ou UF utilisateur. Grâce à la table d'allocation des opérations, le simulateur peut associer une opération donnée à son unité d'exécution et ainsi vérifier que les instructions exécutées sont correctes du point de vue du parallélisme matériel défini dans le modèle du processeur. De la même manière, il est possible de spécifier d'autres paramètres structurels comme le nombre maximal d'opérations exécutables en parallèle ou le nombre de registres disponibles dans chaque unité. Dans ce cas, le simulateur vérifie la cohérence entre le code exécuté et le modèle matériel du processeur et génère le cas échéant des messages d'erreur.

4.4.3 Compilateur

La méthodologie prévoit l'utilisation du compilateur pour les fonctions de « contrôle » non critiques du point de vue de la performance. La compilation permet d'obtenir rapidement un code fonctionnellement valide pour le code périphérique pour ensuite concentrer tous les efforts sur l'optimisation des fonctions critiques, cette étape nécessitant encore la programmation manuelle des fonctions en assembleur du fait de la faible efficacité des compilateurs pour l'optimisation de performance. C'est pourquoi dans notre méthodologie la principale exigence à laquelle est soumis le compilateur concerne la compacité du code, qui doit permettre de réduire le coût matériel du système processeur/mémoire embarquée grâce à la diminution de la taille du code généré.

Parmi tous les outils logiciels nécessaires au développement de code pour processeur embarqué (compilateur, simulateur de jeu d'instructions, débogueur, assembleur, éditeur de liens, émulateur), le compilateur est de loin le plus complexe et qui requiert le plus d'efforts. Dans le cas des processeurs ASIP dont la durée de vie est relativement courte, l'utilisation d'un compilateur n'est envisageable que si son développement est rapide et peu onéreux. Dans le même temps, la complexité croissante des applications DSP embarquées rend de plus en plus irréalistes les approches « tout-assembleur ».

Ceci explique le grand intérêt depuis quelques années pour les compilateurs reciblables, capables d'être rapidement configurés pour un processeur particulier et ce de la manière la plus simple et directe possible. La principale difficulté est liée à la très grande diversité (architecturale et logicielle) des processeurs cibles potentiels, au sein d'une même catégorie (processeur DSP ou processeurs généraux). De fait, il n'existe pas à l'heure actuelle de modèle de compilation général capable de produire de bonnes performances quelque soit le processeur cible.

Le développement d'un compilateur reciblable de qualité pour notre modèle d'ASIP n'est pas le premier objectif de cette thèse, et n'est d'ailleurs pas réaliste si l'on considère l'état de nos connaissances du domaine, l'ampleur du travail de développement et le nombre de problèmes complexes à résoudre. Il n'en demeure pas moins que le compilateur est indispensable à la mise en œuvre de la méthodologie, dont l'efficacité dépend pour une grande part da la qualité (en performance et en taille) du code compilé. Nous nous sommes donc intéressé à la manière de concevoir un compilateur pour notre modèle d'ASIP, capable de générer du code fonctionnel et de prendre en compte les différents degrés liberté du modèle. Cette étude a débouché sur la réalisation d'une

première version rudimentaire d'un compilateur reciblable, dont les performances ne prétendent pas rivaliser avec celles des compilateurs pour processeurs DSP actuels, mais qui a le mérite d'aborder et dans certains cas de résoudre les problèmes liés à la prise en compte de l'architecture et du jeu d'instructions paramétrables du modèle d'ASIP.

Les problèmes spécifiques liés à la production de code optimisé pour architecture parallèle de type VLIW (en particulier la phase d'ordonnancement et de compactage des instructions) ont été volontairement mis de coté, d'une part parce qu'ils sont actuellement le nœud du problème dans la compilation pour processeurs DSP, mais surtout parce qu'ils sont complètement indépendants des problèmes liés à l'aspect configurable du modèle. Sans même parler de l'aspect reciblable, la génération de code optimisé pour processeurs VLIW reste à l'heure actuelle un problème ardu mobilisant dans l'industrie des équipes entières de recherche, et qui au vu des résultats des compilateurs actuels n'est pas encore entièrement résolu. Le compilateur présenté ici ne gère donc pas le parallélisme d'instructions et se contente de générer du code séquentiel de type « mono-scalaire ».

La compilation reciblable étant un domaine extrêmement vaste, nous ne pouvons en donner une image précise dans le cadre de ce seul chapitre. Pour connaître les enjeux et les axes de recherche actuels liés à la compilation générale et reciblable pour processeurs DSP, on pourra se reporter à [53]. Un exemple de production de compilateurs reciblables pour processeurs ASIP est présenté dans [65].

4.4.3.1 Le flot de compilation

Le processus de compilation d'un programme comprend trois étapes principales. Dans un premier temps, le programme source contenant l'application passe à travers un analyseur grammatical qui le transforme en une structure de données de type *CDFG* (*Control/Data Flow Graph*) adaptée aux algorithmes de compilation. Dans le cas d'un compilateur reciblable, le même travail est effectué pour le fichier de description du processeur qui est lui aussi analysé et rangé dans une structure de données d'un autre type. Les deux structures ainsi initialisées forment une base de données sur laquelle vont travailler l'ensemble des étapes suivantes. La compilation proprement dite est traditionnellement divisée en une phase d'optimisation et de transformation de haut-niveau indépendante du processeur cible, suivie d'une phase de génération de code prenant en compte les caractéristiques du processeur (Figure 100).

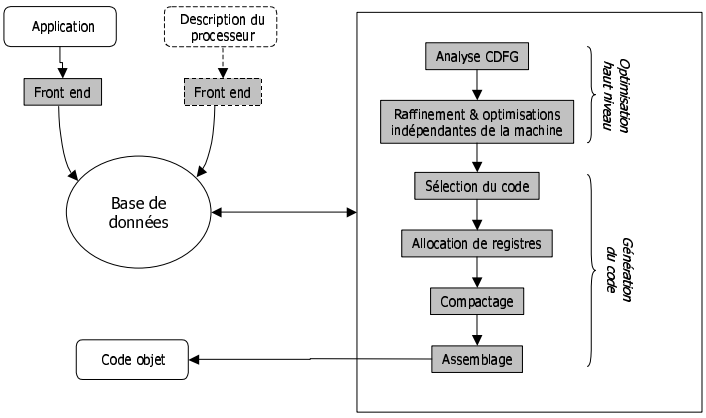


Figure 100 Les principales étapes de compilation

La phase d'optimisation haut-niveau analyse le CDFG de l'application, calcule les dépendances de données puis tente de réduire le nombre d'opérations du graphe et la séquentialité de la description qui permettra plus tard une meilleure utilisation du parallélisme d'instruction. Les types d'optimisations implémentées à ce niveau sont classiques : élimination des portions de code inutiles et des sous-expressions communes, propagation des constantes, etc.

La phase de génération de code prend en compte le *CDFG* modifié par les optimisations et la structure de données décrivant le processeur et son jeu d'instruction, et cherche à sélectionner le meilleur enchaînement d'instructions par rapport aux contraintes spécifiées dans les options de compilation, concernant principalement la performance et la taille du code. Elle comprend trois grandes étapes :

- Sélection du code : on tente de faire correspondre aux opérations élémentaires formant le CDFG de l'application, des « instructions partielles » appartenant au jeu d'instruction du processeur. Selon les cas, plusieurs opérations du CDFG peuvent être combinées pour correspondre à une instruction partielle, la décision étant prise en parcourant le CDFG et en couvrant le graphe d'opérations avec les instructions partielles correspondant au jeu d'instructions. La Figure 101 donne deux exemples de sélection de code : à partir de la description CDFG du code C, le compilateur cherche à couvrir l'arbre d'expression avec les instructions du processeur. La première solution fournit un code non optimisé n'utilisant que les instructions les plus élémentaires du jeu d'instructions, qui ont une correspondance directe avec un seul nœud du CDFG. La deuxième solution détecte l'enchaînement d'opérations correspondant à l'instruction complexe de Multiplication/Accumulation et couvre ainsi deux opérations du CDFG avec une seule instruction, le code résultat étant à la fois plus rapide et plus compact. Comme il peut exister de très nombreuses solutions de couverture, le rôle du sélecteur de code est d'identifier ces solutions et de les évaluer à l'aide d'une fonction de coût, pour au final choisir la meilleure. Il est à noter que dans le cas d'un processeur VLIW, les « instructions partielles » correspondent aux opérations élémentaires utilisées pour former les instructions larges.

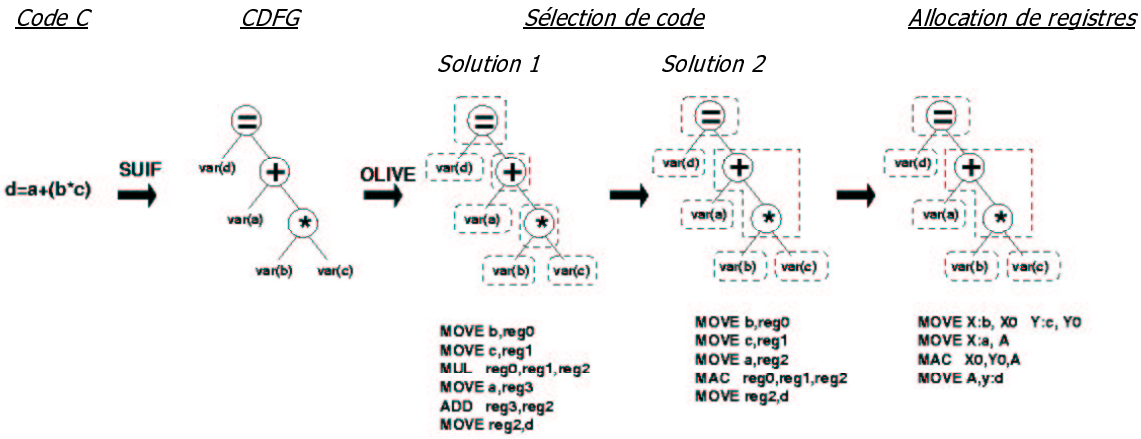


Figure 101 Transformation du code C vers du code assembleur

- Allocation de registres : le sélecteur de code manipule les variables du programme comme des registres virtuels directement accessibles par les instructions du processeur. La phase d'allocation de registres consiste à choisir pour chaque registre virtuel le meilleur

emplacement physique, soit dans un des registres internes du processeur, soit à une certaine adresse de la mémoire. Sachant que le nombre de registres physiques est limité, que les variables stockées en mémoire doivent au préalable être chargées avant d'être manipulées par les instructions du processeur (du moins pour les architectures load/store), et que tous les registres du processeur ne sont pas égaux du point de vue de l'accessibilité par les différentes unités fonctionnelles, la phase d'allocation de registre est un processus complexe qui a une grande influence sur la qualité du code généré.

- Compactage / ordonnancement : cette phase intervient avant tout dans le cas des processeurs à architecture parallèle qui peuvent exécuter simultanément plusieurs « instructions partielles ». Le sélecteur de code produit une suite séquentielle d'instruction partielles qui ne tire aucun profit du parallélisme de l'architecture. La phase d'ordonnancement consiste à réordonner et rassembler plusieurs instructions partielles en « macro-instructions » VLIW afin d'exécuter plusieurs opérations en parallèle et d'augmenter les performances. Pour cela, l'algorithme d'ordonnancement doit tenir compte de nombreuses contraintes, telles que les dépendances de données entre instructions, la disponibilité des ressources du processeur, les aléas de pipeline, et les contraintes d'encodage des instructions. C'est de l'efficacité de cet algorithme que dépend pour une grande part la qualité du code VLIW produit.
- Assemblage : la dernière phase consiste à générer le code assembleur de l'application. Le simulateur interprétant du code assembleur symbolique (fichier texte) et non du code objet, la sortie du compilateur se fera sous le même format, chaque instruction issue de la phase de compaction étant envoyée au format texte vers le fichier assembleur de sortie.

4.4.3.2 L'environnement de compilation

Le compilateur est constitué de deux parties, une partie frontale (ou « front-end ») analysant le code source et effectuant des transformations indépendantes du matériel, et une partie finale (ou « back-end ») produisant le code assembleur en tenant compte des caractéristiques du processeur cible. Entre les deux passes, le programme est représenté dans un format intermédiaire, le format « SUIF ». La réalisation du compilateur a principalement consisté à concevoir le générateur de code de la partie back-end afin de prendre en charge l'attribution des opérations du programme aux instructions du processeur, l'attribution des ressources de mémorisation aux données, ainsi que diverses optimisations liées aux caractéristiques du modèle d'ASIP.

L'environnement SUIF

L'environnement *SUIF*[66] développé à l'université de STANDFORD constitue la partie frontale du compilateur. SUIF est un outil d'aide à la construction et l'expérimentation de compilateurs. Il met à la disposition de l'utilisateur une partie frontale compatible avec le langage C (et d'autres langages comme le *C++* ou le *Fortran*), une représentation intermédiaire du code et diverses bibliothèques : algorithmes d'optimisation classiques, outils de parallèlisation, classes C++ d'accès à la représentation intermédiaire, etc. Un des intérêts de *SUIF* est le niveau d'abstraction de la représentation intermédiaire, qui conserve des informations sur les structures de contrôle.

La bibliothèque SPAM

SPAM est une bibliothèque d'algorithmes de génération et d'optimisation de code pour processeurs spécialisés (en particulier pour le traitement du signal) issue du travail de thèse de G. Araujo [67] et A. Sudarsanam [68] à l'Université de Princeton. Elle intègre des algorithmes d'analyse de code, d'ordonnancement, d'allocation de ressources et des optimisations très spécifiques (gestion de registres de mode, placement de variables en mémoire). L'ensemble des algorithmes est paramétrable par une description du jeu d'instructions du processeur cible.

Les algorithmes de la bibliothèque travaillent sur diverses représentations du programme, principalement des *DAGs* (*Direct Acyclic Graphs*) et des arbres d'expressions. Ces représentations, regroupées sous le nom de *TWIF*, s'interfacent avec *SUIF* et sont l'équivalent de la représentation de *SUIF* pour les traitements de bas niveau. *SPAM* est distribué avec le générateur de code *Olive*[69], qui effectue la sélection d'instructions et joue par conséquent le rôle d'interface entre *SUIF* et *TWIF* (Figure 102) .

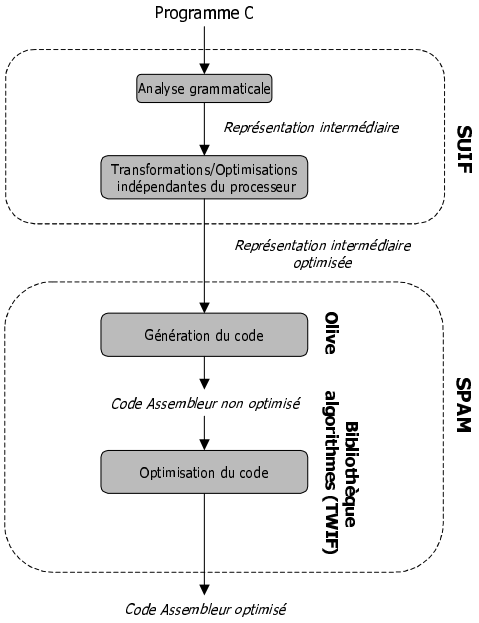


Figure 102 Vue d'ensemble de la chaîne de compilation

Production de code assembleur non optimisé puis optimisé

La génération du code assembleur de l'application se fait en deux étapes. Dans la première, le programme source (dans notre cas en C) est traité par *SUIF* qui génère une représentation intermédiaire dans un format qui lui est propre. Après les transformations et optimisations effectuées par *SUIF* (suppression des parties de code redondantes, propagation des constantes, etc.), cette représentation est traduite par un convertisseur sous forme d'un arbre d'expression (au format *OliveTree*), point d'entrée du générateur de code *Olive.* A partir de la description du jeu d'instruction du processeur cible, exprimée dans un format particulier (la grammaire *Olive*), *Olive* génère une première version du code assembleur de l'application (Figure 103).

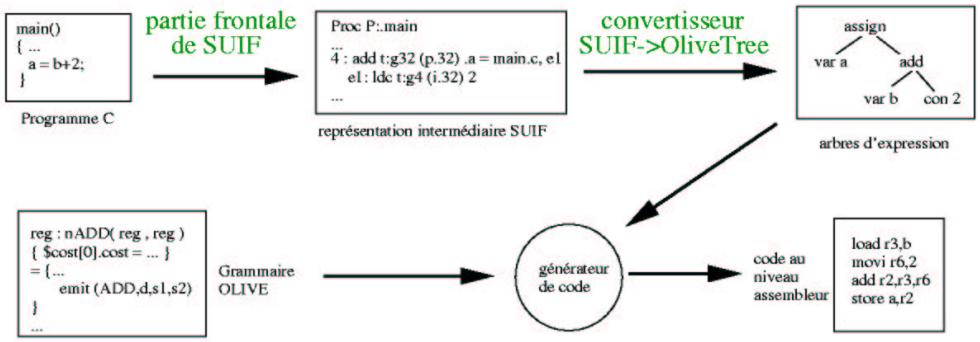


Figure 103 Exemple d'utilisation des outils SUIF et Olive pour la production de code non optimisé

La deuxième étape fait intervenir un programme conçu par l'utilisateur, qui fait appel à différents algorithmes d'optimisation de la bibliothèque *TWIF*. Cette bibliothèque regroupe une ensemble de classes et de fonctions en C++ permettant de manipuler et d'appliquer des traitements aux différents niveaux de représentations du programme : *CallGraph*, *TwifProc*, *CFGGgraph*, basic-blocs et arbres d'expression (Figure 104). La bibliothèque *SPAM* repose sur le concept des fonctions abstraites du C++, qui permet de ne définir que l'interface (ou le prototype) d'une fonction et de laisser au développeur le soin d'écrire le corps de la fonction en fonction des caractéristiques du processeur cible. *SPAM* propose par défaut un « squelette » de compilateur appelant les principales fonctions correspondant aux algorithmes classiques du flot de compilation. Ces fonctions sont soit indépendantes du processeur cible et peuvent être utilisées telles quelles, soit dépendantes et dans ce cas elles doivent être partiellement ou totalement réécrites pour tenir compte des caractéristiques du processeur . Elles sont regroupées en quatre groupes principaux : fonctions d'allocation (registres, pointeurs, variables sur pile, etc.), fonction de compactage (utilisation en parallèle d'unités fonctionnelles multiples), fonctions d'analyse *data-flow* (informations sur l'utilisation des données à l'intérieur d'une certaine région du programme) et fonctions d'optimisation, indépendantes ou dépendantes du processeur[70].

A l'issu de cette deuxième étape, on obtient la version finale du code assembleur, fonctionnel et optimisé.

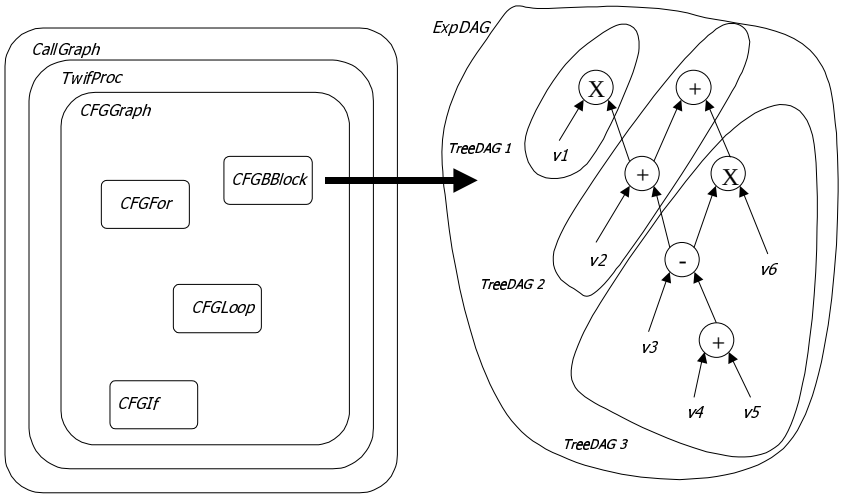


Figure 104 Hiérarchie des classes de représentation SPAM, et structure d'un Basic-Block

Représentation d'un programme C

L'environnement SPAM utilise plusieurs classes pour la représentation du code (Figure 104). En haut de la hiérarchie se trouve la classe *CallGraph* associée à un graphe orienté représentant le programme entier. Chaque nœud de ce graphe correspond à une procédure, stockée dans la classe *TwifProc*. Dans la classe *TwifProc*, la procédure est divisée en plusieurs graphes de flot de contrôle, appelés *CFGraph* (pour *Control Flow Graph*). Les nœuds du *CFGraph* peuvent être de type *CFGBBlock* (représentant un basic bloc, c'est à dire un arbre d'expression composé d'opérateurs purement « flot de données »), *CFGFor* (représentant les structures de boucles « For »), *CFGIF* (représentant les structures de type *if-then-else*), ou *CFGLoop* (représentant les structures de type *Do While*). Les nœuds représentant les structures de contrôle sont eux-mêmes hiérarchiques, et encapsulent des basic-blocs représentant les parties flots de données de la structure (un nœud de type *CFGIF* contient par exemple deux nœuds *CFGBBlock* représentant les corps du *if* et du *else*).

Le générateur de code *Olive* ne manipule pas directement les basic-blocs mais des arbres d'expression représentés par la classe *TreeDAG*. Pour obtenir ces arbres, on construit pour chaque basic bloc un graphe acyclique d'expression, appelé *ExpDAG*, qui représente le basic-bloc sous forme de nœuds correspondant aux opérations élémentaires de la représentation SUIF (addition, soustraction, lecture/écriture mémoire, constante, assignation de variable, voir exemple Figure 103) et d'arcs symbolisant les dépendances de données entre opérations. La sélection d'instruction à partir d'un *ExpDAG* étant un problème NP-complet, il est nécessaire de « démanteler » l'*ExpDAG* en un ensemble d'arbres d'expression (les *TreeDAG*), le processus de démantèlement imposant que chaque arc de sortie d'un nœud à sorties multiples n'appartiennent pas au même *TreeDAG*. Cela revient donc pour chaque nœud possédant N sorties à « couper » *N-1* arcs sortants (Figure 104). Pour un même *ExpDAG* de départ, il peut exister plusieurs solutions de démantèlement, c'est pourquoi il est conseillé dans le cas des processeurs à architecture hétérogène d'adapter l'algorithme de démantèlement à l'architecture du processeur [67] afin de sélectionner les solutions les plus adéquates. Dans le cas de processeurs à architecture homogène et régulière comme notre modèle d'ASIP, et grâce à l'utilisation des fonctions intrinsèques (voir section suivante), le démantèlement n'a aucune influence sur la qualité du code généré, c'est pourquoi on utilise un algorithme de démantèlement général.

4.4.3.3 Sélection d'instructions et génération de code par Olive

Nous allons maintenant examiner en détail le fonctionnement du producteur de code *Olive*, et la manière dont est décrit le jeu d'instructions générique du modèle d'ASIP grâce au fichier de spécifications *OLIVE*. *Olive* travaille sur des portions de programme représentées par des arbres d'expression (*TreeDAG*) issus du démantèlement des basics-blocs, et génère pour chaque *TreeDAG* une suite d'instructions assembleur équivalente. La sélection des instructions se fait après analyse du fichier de spécification « ASIP.brg » décrivant les règles de sélection de code.

Olive met en oeuvre un algorithme de production de code en deux passes. La première parcourt l'arbre d'expression des feuilles vers la racine. Elle étiquette chaque nœud avec les règles applicables à ce nœud et lui attribue un coût. La seconde passe commence son exécution à la racine de l'expression. Elle exécute le code associé à la règle dont le coût (somme des coûts des nœuds formant la règle) est minimal. Ce code prend en charge l'émission des instructions assembleur qui résultent de la

couverture optimale (à moindre coût) de l'arbre. Le fichier ASIP.brg contient les règles servant à reconnaître les enchaînements d'opérateurs correspondant à des « opérations » assembleur, à calculer les coûts et à générer le code assembleur.

Structure des règles

Les règles de production de code sont structurées en trois sections: l'expression elle-même, un calcul de coût et la partie *action* qui contient le code à exécuter lorsque la règle est choisie. La Figure 105 détaille le contenu d'une règle représentant une instruction d'addition. L'expression est composée de symboles terminaux et non-terminaux. Les terminaux correspondent aux nœuds de l'arbre à analyser. Ils doivent être déclarés au préalable à l'aide de la directive *term*. La liste des terminaux est fixée à l'avance et commune à toutes les descriptions basées sur la représentation *OliveTree*. Les non-terminaux sont spécifiques au processeur cible. Leur utilisation première est la représentation des opérandes, en général des registres du processeur. Dans la Figure 105, le non-terminal *reg* symbolise un banc de registre utilisé comme opérande source et destination de l'instruction.

Le calcul de coût travaille sur une structure à champs multiples définie par l'utilisateur. Dans notre cas, un seul champ *cost* est utilisé, et le coût d'une règle est la somme des coûts des opérandes à laquelle s'ajoute le coût de la règle (ici la constante 1). L'accès aux opérandes est réalisé à l'aide de la construction *$cost[i]* où i est la place du non-terminal dans la règle.

La partie *action* de la règle décrit les tâches à effectuer lorsque la règle est sélectionnée et appelée par une règle de plus haut niveau lors de la deuxième passe de l'algorithme. L'action associée à la règle *aADD* de la figure génère une instruction assembleur effectuant l'addition. Tout d'abord, les registres sources et destination sont alloués dynamiquement par appels des procédures *action* correspondant aux non-terminaux *reg* de la règle. L'instruction proprement dite est ensuite crée, et ajoutée à la séquence d'instructions résultat de l'exécution des règles de plus bas niveau. La récursivité se poursuit jusqu'à l'exécution de l'action de la règle de plus haut niveau, pour obtenir au final la liste d'instructions correspondant à l'arbre d'expression d'origine.

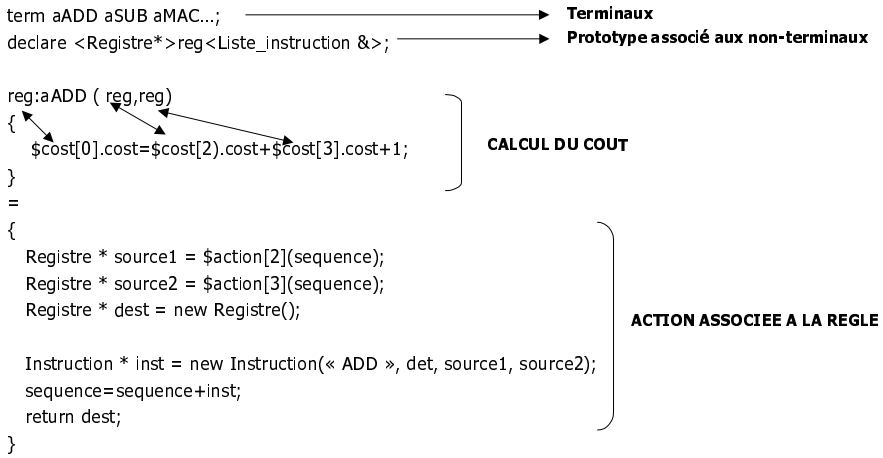


Figure 105 Structures de règles pour Olive

Exemple de couverture

La Figure 106 illustre la couverture d'une expression simple par sept règles, dont quatre représentent des instructions de calcul, et trois gèrent l'accès à la mémoire de données dans laquelle se trouve les variables.

La phase d'étiquetage débute par les nœuds *VAR*, pour lesquels seule la règle *load* convient. On leur attribue le coût correspondant à la règle *load*, par exemple 2. Aux étapes suivantes, chaque nœud est étiqueté avec la valeur correspondant à la somme du coût du nœud lui-même plus la somme du coût de ses opérandes. Pour les nœuds *SUB* et *MUL*, cela donne un coût de 5 : 1(coût de l'opération)+2(coût du *load*)+2.

Pour la couverture du nœud *ADD*, deux règles conviennent : soit la règle *ADD*, soit la règle plus complexe *ADD(MUL…)* recouvrant à la fois les nœuds *ADD* et *MUL*. C'est cette dernière, dont le coût est le plus faible, qui permet d'obtenir le coût de couverture le plus faible et qui est donc retenue. On voit donc l'importance que revêt l'attribution des facteurs de coût aux différentes règles afin d'obtenir le meilleur code. Les valeurs attribuées doivent refléter les caractéristiques architecturales du processeur et du jeu d'instruction. Dans l'exemple présenté au dessus, la présence d'un Multiplieur/Accumulateur dans le processeur cible est prise en compte grâce à la règle *ADD(MUL…)* dont le coût est inférieur à la somme des coûts correspondant à deux instructions séquentiels *ADD* puis *MUL*, favorisant ainsi son utilisation.

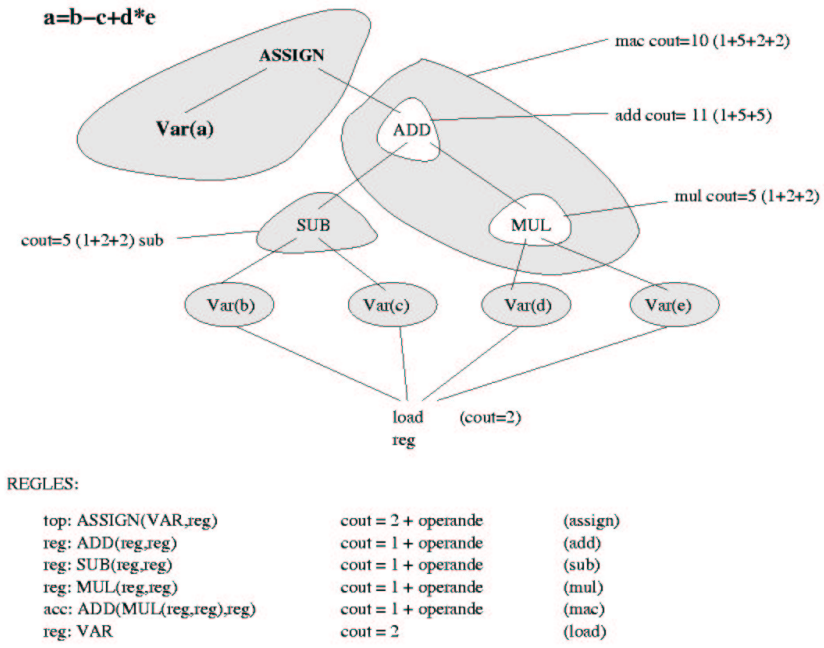


Figure 106 Couverture d'un arbre d'expression par Olive

Adaptation au modèle d'ASIP et au jeu d'instruction générique

Le fichier de règles ASIP.brg doit au minimum contenir toutes les règles permettant de couvrir l'ensemble des opérateurs utilisés dans une description de type TreeDAG. Ces opérateurs (environ une quarantaine) suffisent pour décrire n'importe quelle fonction. On y trouve des opérateurs arithmétiques et logiques (*ADD*, *SUB*, *MUL*, *DIV*, *AND*, *OR*, etc.), des opérateurs d'accès aux variables (*VAR* pour la lecture, *ASSIGN* pour l'écriture), d'indirection (*INDIR* pour la lecture, *STR* pour l'écriture), de branchements (*bTRUE*, *bFLASE*), d'appels de fonctions (*nCAL*, *nRET*), etc. Il faut

donc dans un premier temps décrire un premier ensemble de règles « minimales », qui ne tireront pas profit de toute la puissance du jeu d'instruction et de l'architecture du modèle d'ASIP, mais qui permettront de générer du code pour n'importe quelle fonction. Il est ensuite possible d'ajouter des règles plus « évoluées » décrivant des instructions de plus grande complexité recouvrant plusieurs nœuds à la fois et permettant de générer un code plus compact et plus performant.

Le fichier de description que nous avons écrit intègre les principales fonctionnalités suivantes :

- Allocation des variables locales aux registres symboliques ***Ax*** de l'unité CU. On utilise une table de hash associant les variables locales à leur registre symbolique associé afin de réduire le nombre de registres utiles en évitant de réallouer un nouveau registre à chaque accès à la même variable.
- Gestion des deux largeurs de données *simple* et *double* précision. Un registre ***Ax*** alloué dépend du type de la variable : un demi registre pour une variable simple précision (ex : *A0_l* ou *A0_h*), et un registre entier pour une variable double précision (ex : *A0*).
- Allocation des pointeurs locaux aux registres ***Px*** de l'unité AGU.
- Allocation sur la pile de tous les paramètres (variables et pointeurs) passés lors des appels de fonction.
- Accès en lecture/écriture de variables et de pointeurs stockés en mémoire : la variable (pointeur) peut être globale, dans la pile, ou accédée par adressage indirect.
- Transferts de variables (pointeurs) entre registres.
- Gestion des constantes simple et double précision.
- Utilisation des modes d'adressage évolués de l'unité AGU, en particulier les adressages indirect complexes de type « *adresse de base + offset* ».
- Gestion des appels de fonction : empilement de tous les paramètres avant l'appel, génération de l'instruction *CALL*. Pour les retours de fonction, restauration de la pile et copie de la valeur de retour dans le registre *A0* (par convention).
- Opérations de comparaison et conditions de test : génération de l'instruction « *test(reg1,reg2,cond)* » permettant d'affecter au bit T le résultat de l'un des six tests prévus par défaut dans le modèle de référence du processeur: *EQ (Equal*), *NE (Non-Equal), GT (Greater), GE (Greater or Equal), LT (Lesser), LE (Lesser or Equal).*
- Branchements inconditionnels et conditionnels : génération d'une instruction de saut éventuellement conditionnée par l'un des deux marqueurs *IFF* ou *IFT*. Un nœud de type *bTRUE* produira par exemple une instructions « ift jump ».
- Utilisation et ajout d'instructions associées aux unités fonctionnelles de l'unité CU par l'emploi d'une table des fonctions intrinsèques.

Ce dernier point est particulièrement important, car il permet d'ajouter ou d'enlever des instructions utilisateurs liées aux unités fonctionnelles de CU sans modifier en aucune manière le fichier de description des règles.

Utilisation des intrinsèques « compilateur »

Dans le code source de l'EFR, les opérateurs arithmétiques de la bibliothèque ETSI sont déclarés comme fonctions, et leur utilisation nécessitent donc un appel de fonction, au même titre que n'importe quel autre fonction de plus haut niveau. Dans la description *OliveTree*, ces appels sont modélisés par un nœud *nCAL* ayant pour opérande une constante définissant le nom de la fonction appelée. Dans le cas par exemple d'un appel à l'opérateur *l_add()* (addition 32 bits signée avec saturation automatique du résultat), la description *OliveTree* de la fonction faisant cet appel contiendra un nœud *nCAL(« l_add »)*, et le générateur de code *Olive* se contentera de générer l'instruction assembleur *CALL* avec pour adresse de saut l'adresse de la procédure *l_add()* décrite dans la classe *CallGraph* de l'application (voir Figure 104). Cette procédure, décrite fonctionnellement dans le fichier « basic_op.c » du code source de l'EFR, sera elle aussi transformée en un arbre d'expression puis par *Olive* en une suite d'instructions assembleur résultant de la couverture des *noeuds* de la description *OliveTree*. Le coût d'un appel à un opérateur de base comme *l_mac*, qui ne prendrait en assembleur manuel qu'un cycle (puisque cet opérateur correspond à une opération de l'unité fonctionnelle MAC) devient dès lors prohibitif puisqu'il faut compter le coût de l'appel de fonction et du retour, plus les opérations du corps de la fonction, ce qui représente l'équivalent d'au moins une dizaine de cycles. Pour les opérateurs les plus fréquemment utilisés (voir Figure 95), de tels résultats ont des effets désastreux sur la performance du code compilé.

```
Word32 L_add (Word32 L_var1, Word32 L_var2) {
  Word32 L_var_out;
   L_var_out = L_var1 + L_var2;
   if (((L_var1 ^ L_var2) & MIN_32) == 0)
      if ((L_var_out ^ L_var1) & MIN_32)
         L_var_out = (L_var1 < 0) ? MIN_32 : MAX_32;
   return (L_var_out);
}
```

Figure 107 Code source de l'opérateur L_add()

Pour obtenir un code performant, il est donc indispensable d'indiquer au compilateur qu'il existe une instruction particulière du processeur permettant d'exécuter l'opérateur en un seul cycle. Ceci peut se faire de trois manières différentes :

- Reconnaissance par le compilateur de l'arbre d'expression correspondant à l'opérateur, et génération de l'instruction correspondante. Le code source décrivant les opérateurs arithmétiques de la bibliothèque pouvant être complexe (une dizaine de ligne de C incluant des structures de type *For* ou *If-Then-Else*), la reconnaissance des arbres d'expression par le sélecteur de code (ici *Olive*) est au mieux très difficile, voire même impossible si les opérateurs intègrent des structures de contrôles : dans ce cas, leur description ne s'étend plus sur un mais sur plusieurs basic-blocs, alors qu'*Olive* a une visibilité limitée à un seul basic-bloc.

- Utilisation de fonctions intrinsèques « assembleur » décrivant le comportement des opérateurs directement en assembleur, et qui sont substituées dans le code source aux appels de fonctions des opérateurs , par exemple à l'aide d'une directive #define. Cette méthode offre pour l'opérateur à la fois une performance optimale et une taille de code réduite au minimum. Par contre, le code devient dépendant de l'architecture et tous les opérateurs critiques devront être reprogrammés en assembleur à chaque changement de processeur cible. Et surtout, l'emploi de lignes assembleur dans le code C empêche le compilateur d'appliquer bon nombre des algorithmes d'optimisation de code (réordonnancement et compactage des instructions, élimination des sous-expressions communes, etc.). En effet, le compilateur ne « comprend pas » les instructions assembleur (il ne connaît pas par exemple la latence des instructions) et les traitent donc comme une « boite noire » à insérer telle quelle et qui sera exclue de toute optimisation. Dans le cas où une fonction intrinsèque est utilisée dans une boucle critique, elle ne pourra donc participer à des optimisations telles que le déroulage de boucle, le pipeline logiciel, etc. Pour des processeurs disposant de la capacité d'exécuter plusieurs instructions par cycle, ces « barrières d'optimisation » sont très handicapantes et réduisent considérablement la performance du code compilé.
- Utilisation de fonctions intrinsèques « compilateur ». C'est cette méthode que nous utilisons dans le compilateur.

Cette dernière solution consiste à prévoir dans la représentation intermédiaire sur laquelle travaillent les algorithmes de compilation des structures dédiées à la représentation des fonctions intrinsèques. Par défaut, les seuls opérateurs présents dans la représentation intermédiaire d'un programme (dans notre cas les représentations *SUIF* et *OliveTree*) sont les opérateurs de base du langage C : « + » (nœud ADD), « * » (nœud MUL), « >> » (nœud SHL), etc. En ajoutant de nouveaux nœuds pour les fonctions intrinsèques disposant d'une correspondance directe avec une instructions assembleur du jeu d'instruction (par exemple un nouveau nœud L_ADD pour l'opérateur *l_add()*), il est possible de générer directement l'instruction assembleur correspondante et surtout d'éviter les barrières d'optimisation car les nouveaux nœuds sont traités de la même manière que les autres et participent à l'ensemble des phases d'optimisation. Les résultats obtenus dans le cas de processeurs à haut degré de parallélisme d'instructions peuvent alors être très supérieures à ceux obtenus avec les fonctions intrinsèques « assembleur », et ce pour une taille de code quasi-identique [71].

Pour rendre le compilateur performant, il est nécessaire de déclarer comme intrinsèques tous les opérateurs ayant une correspondance directe avec une instruction assembleur du jeu d'instructions. Notre modèle de processeur prévoyant l'ajout d'unités fonctionnelles et d'instructions utilisateurs, le compilateur doit donc être flexible du point de vue de la gestion des intrinsèques.

Pour ce faire, nous utilisons un fichier de configuration contenant l'ensemble des noms d'opérateurs considérés comme intrinsèques. Lors de la sélection de code, chaque nœud symbolisant un appel de fonction (nœud *nCAL*) est examiné, et l'opérande constante symbolisant le nom de la fonction appelée est comparé aux noms présents dans le fichier des intrinsèques. Si ce nom est trouvé, l'appel de fonction est « court-circuité » et *Olive* génère l'instruction assembleur correspondante. Sinon, l'appel de fonction est conservé et provoquera l'émission d'une instruction *CALL*, et la génération du code assembleur correspondant au corps de la déclaration de la fonction.

Cette structure configurable permet d'ajouter ou d'enlever facilement des fonctions intrinsèques sans avoir à modifier le code source du compilateur, le rendant « reciblable » du point de vue des unités fonctionnelles et de ses instructions. Dans le cas ou le processeur cible ne dispose pas d'une instruction implémentant un opérateur particulier, il suffit d'enlever du fichier de configuration le nom de l'opérateur et le compilateur générera le code assembleur à partir du code source de l'opérateur défini dans la bibliothèque. La seule contrainte de cette méthode est liée à l'emploi d'une bibliothèque d'opérateurs C pour la description des applications, qui est de toute façon un postulat de départ pour notre méthodologie.

4.4.3.4 Optimisation par la bibliothèque d'algorithmes TWIF

A l'issu du traitement par *Olive*, nous obtenons pour chaque arbre d'expression de chaque basic-bloc une portion de code assembleur résultant de la couverture de la représentation intermédiaire du programme source par le jeu d'instructions décrit dans le fichier « ASIP.brg ». Ce code a les caractéristiques suivantes :

- Enchaînement séquentiel d'instructions élémentaires. Les instructions générées par *Olive* sont constituées d 'une seule « opération » par unité matérielle (PCU, AGU ou PCU, voir la notion d'« opération » en 3.6.1) et n'exploitent pas le parallélisme architectural et la structure du jeu d'instruction VLIW.
- Manipulation de registres et de paramètres virtuels. Les variables locales sont allouées dans des registres virtuels qui n'ont pas encore de correspondance avec les registres réels de l'unité CU (registres ***Ax***). On ne tient pas compte des limites matérielles : nombre de registres et nombre de ports du banc de registres. De même, les paramètres passés par la pile sont adressés par leur nom symbolique et n'ont pas encore d'index défini dans la pile.
- Aucune gestion des boucles câblées de type *For*. Le opérations ayant trait à ces (initialisation et incrémentation du compteur, test de fin de boucles, saut au début de la boucle) boucles sont pour l'instant ignorées.
- Pas de sauvegarde de contexte lors des appels de fonction. En effet, la visibilité d'*Olive* se limite au basic-bloc, alors que la sauvegarde de contexte tient compte de l'ensemble des opérations d'une procédure.
- Pour les mêmes raisons, l'enchaînement des instructions ne tient pas compte des contraintes d'ordonnancement liées à la structure du pipeline de contrôle. Il se peut donc que des aléas de pipeline ne soient pas respectés.

Pour obtenir un code fonctionnel, il est donc nécessaire de modifier et/ou de compléter le code généré. Les traitements réalisant ces modifications sont implémentés grâce aux algorithmes de la bibliothèque *TWIF* de *SPAM*.

Gestion des sauvegardes de contexte

Lors d'un appel de fonction, *Olive* génère les instructions d'appel et de retour, mais ne s'occupe pas de la sauvegarde du contexte. Dans notre processeur, il est nécessaire de sauvegarder tous les registres

utilisés dans le corps de la fonction, c'est à dire les registres ***Ax*** de l'unité CU, les registres ***Px*** de l'unité AGU, les registres compteurs et le registre d'état.

Pour sauvegarder le contexte lors des appels, nous avons plusieurs choix possibles :

- La fonction appelante sauvegarde le contexte et la restaure à la fin de l'appel
- La fonction appelée sauvegarde le contexte et le restaure.
- Aucune des deux ne sauvegarde le contexte et essaient de ne pas utiliser les mêmes ressources.

La dernière solution est évidemment la plus performante mais n'est pas réaliste du fait du nombre de ressources limité du processeur, et requiert une phase d'allocation de registre globale difficile à implémenter. Elle interdit en outre l'usage des fonctions récursives (cependant très peu utilisées en traitement du signal).

La première solution donne de meilleurs résultats en performance car la visibilité de la fonction appelante permet de ne sauvegarder que les registres communs à la fonction appelée et appelante, et non tous les registres de la fonction appelée. La deuxième est plus avantageuse du point de vue de la taille du code, car la sauvegarde de contexte est décrite une seule fois dans le code de la fonction appelée et non à chaque appel de la fonction. Le rôle du compilateur dans la méthodologie étant principalement de générer du code pour les parties contrôle, il est logique de privilégier la taille du code par rapport à la performance. La sauvegarde du contexte se fait donc dans la fonction appelée.

Pour trouver tous les registres à sauvegarder, on parcourt l'objet de type *TwifProc* qui correspond à une procédure (ou fonction) donnée. On analyse ensuite chaque nœud du *CFGraph* (principalement des basic-blocs) de la procédure afin de détecter les registres utilisés. Une portion de code assembleur est alors ajoutée au début et à la fin de la procédure pour sauvegarder puis restaurer les registres détectés, la sauvegarde et la restauration se faisant par empilement/dépilement des valeurs des registres.

Gestion des boucles câblées

Olive génère du code pour les basic-blocs (nœuds *CFGBBlock* du *CFGGraph*) , mais ne prend pas en compte les nœuds associés aux structures de contrôle comme les boucles For (*CFGFor*). Une procédure spéciale, appelée *GenerateForNode* est prévue par SPAM pour émettre le code assembleur destiné à l'implémentation des boucles. Cette fonction a été écrite spécialement pour le modèle d'ASIP et effectue les tâches suivantes :

- Analyse des paramètres de la boucle *For* (initialisation de la variable de comptage, valeur d'incrément et test de fin de boucle). L'implémentation des boucles *For* à l'aide des structures de contrôle spécialisées du processeur n'est possible que si certaines contraintes sont respectées : la valeur de l'incrément, de la valeur initiale et de la valeur finale doivent rester fixes tout au long de la boucle. Dans le cas contraire, la boucle devra être implémentée de manière classique avec allocation d'un registre pour le comptage, et instructions explicites d'incrémentation, de test et de branchement de fin de boucle.
- Réorganisation du *CFGGraph* pour enlever certains nœuds inutiles.

- Insertion de l'instruction *LoopStart_x* au début du corps de la boucle.
- Insertion de l'instruction *LoopEnd_x* en fin de boucle.
- Substitution des opérandes registres ***Ax*** correspondant à la variable de boucle par le registre compteur correspondant. En effet, dans la représentation intermédiaire du programme, la variable de boucle est une variable comme les autres, à laquelle *Olive* a attribué un registre ***Ax*** lors de la phase de sélection de code. Pour les adressages de tableaux de type « Adresse de base + Offset » qui sont très souvent utilisés en traitement du signal, la substitution remplace les instructions d'adressages AGU de type « *Py+Az* » par leur équivalent « *Py + Cntr_x* », Py contenant l'adresse de base du tableau et *Cntr_x* la valeur du compteur/Offset, avec *x* indiquant le niveau de boucle courant.
- Substitutions des instructions *JMP* vers la fin de boucle par des instructions *CONT*. Idem pour les *JMP* de sortie de boucle remplacés par des instructions *BRK*.
- Comptage du nombre d'instructions dans la boucle.
- Insertion d'une instruction *Enable_Loop*, de type *Long* ou *Short* suivant la longueur de la boucle, avec la bonne valeur d'incrément et le niveau de boucle courant.
- Insertion des instructions d'initialisation du compteur *Cntr_x*, du registre longueur Ln_x et pour les boucles *Long* de l'adresse de début de boucle *lsa_x*.

Détection des aléas de pipeline

Lors de la génération du code avec *Olive*, la visibilité se limite au niveau arbre d'expression et il n'est donc pas possible de connaître dans tous les cas les instructions précédant ou suivant l'instruction en cours. Il faut donc attendre la fin de la génération du code pour pouvoir traiter le problème des aléas.

Le cas du traitement des aléas n'est pas explicitement prévu par SPAM, il a donc fallu créer une nouvelle fonction nommée *FixPipelineConstraints()*, appelée à la toute fin du processus de génération. Cette fonction parcourt pour chaque procédure l'ensemble des instructions générées et insère le nombre de *NOPs* nécessaires pour éviter les aléas. Le parcours se fait récursivement à travers les nœuds du *CFGGraph*, afin de détecter les aléas survenant entre des instructions appartenant à des basic-blocs différents. Actuellement, le compilateur prend en compte les deux types d'aléas de données les plus courants :

- Ecriture puis lecture dans les deux cycles qui suivent de la même donnée en mémoire.
- Ecriture dans un registre ***Ax*** (étage *EXECUTE*) puis utilisation de ce registre comme valeur d'index pour un calcul d'adresse (étage *ADDRESS*).

Le passage des registres virtuels aux registres physiques ainsi que l'allocation des paramètres sur la pile sont effectués par des appels à deux fonctions de la bibliothèque d'algorithmes d'allocation de SPAM. Une fois que l'ensemble des traitements de transformation et d'optimisation du code a été effectué, le code assembleur est généré pour chaque procédure de l'application vers un fichier de sortie associé.

4.4.3.5 Performances

L'état d'avancement actuel de ce compilateur ne permet pas de générer automatiquement du code pour une application complexe comme l'EFR composé de plusieurs dizaines de fonctions . Il est cependant capable de générer du code séquentiel fonctionnellement correct pour les quelques fonctions provenant de l'EFR qui ont servi de test pour son développement. Les performances sur trois de ces fonctions sont comparées par rapport à leur équivalents programmés manuellement et optimisés pour un processeur de référence possèdent 2 unités MAC et une ALU, et avec une bande passante et un parallélisme d'instructions suffisants pour utiliser toutes les unités en parallèle (Figure 108). Les résultats sont très inégaux selon le type de fonction compilée.

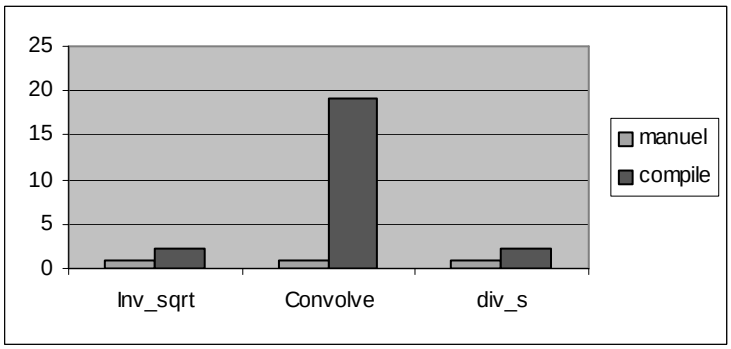


Figure 108 Rapport du nombre de cycles Code Compilé / Code optimisé

Inv_srqt() calcule l'inverse de la racine carrée d'une valeur entière. Le code utilise les fonctions intrinsèques et ne comporte pas de boucle. Le résultat obtenu est assez bon pour un compilateur car le nombre de cycles du code compilé est seulement le double du code écrit manuellement.

div_s(var1,var2) calcule la partie entière du résultat de la division de *var1* par *var2*. Le code source comprend plusieurs structures *If-Then-Else* imbriquées et utilise une boucle *For*. Là encore, le résultat obtenu (un rapport de 2.5) peut être considéré comme correct.

Convolve(x[],h[]) calcule la valeur de la convolution entre deux vecteurs *x[]* et *h[]* passés en paramètre et écrit le résultat dans un vecteur *y[]*. L'algorithme est composé de deux boucles *For* imbriquées:

```
for (n = 0; n <= 39; n++) {
  s = 0;
  for (i = 0; i <= n; i++)
     s = L_mac (s, x[i], h[n - i]);
  s = L_shl (s, 3);
  y[n] = extract_h (s);
}
```

Le résultat pour cette fonction (un rapport de 19) est assez catastrophique et montre la marge de progression du compilateur. Contrairement aux deux fonctions précédentes qui exposaient assez peu de parallélisme, cette fonction est typique des algorithmes DSP de type « filtre » basé sur l'utilisation de plusieurs boucles imbriquées dans lesquelles le calcul se résume à quelques instructions. Alors que la version en assembleur optimisé utilise le maximum de parallélisme disponible dans le processeur

en déroulant la boucle de niveau 2 pour effectuer 2 opérations MAC dans le même cycle, le compilateur insère plusieurs instructions supplémentaires dans les deux boucles :

- initialisation de pointeurs qui sont pourtant invariants pendant toute la durée du calcul.
- insertion de NOPs pour éviter les aléas de pipeline, eux-mêmes provoqués par les instructions d'initialisation de pointeurs inutiles.
- décomposition en deux instructions distinctes (une instruction de chargement mémoire + une instruction de calcul dans l'unité CU) des calculs sur des données en mémoire, alors que l'architecture permet de les effectuer en une seule instruction.

Ces instructions inutiles dégradent considérablement les performances, d'autant plus qu'elles sont insérées à l'intérieur des boucles critiques. Le déplacement des initialisations de pointeurs à l'extérieur des boucles, optimisation simple à mettre en œuvre, permettrait à elle seule de doubler la performance de la fonction. Bien que l'objectif du compilateur ne soit pas d'être performant sur les fonctions critiques comme *Convolve* mais plutôt sur les fonctions de contrôles, de telles optimisations ont des effets bénéfiques sur les deux types de code et sont donc indispensables à mettre en œuvre.

4.4.3.6 Améliorations à apporter

L'objectif poursuivi dans le cadre du développement de ce compilateur était de définir et de tester la structure d'un modèle de compilation adapté à notre processeur configurable. La première version du compilateur est capable de générer du code utilisant l'ensemble des structures matérielles « générales » du processeur : modes d'adressages et unités fonctionnelles du processeur de référence, mécanismes de boucles matérielles « zéro-cycles », et prise en compte du pipeline de contrôle. L'utilisation du fichier de fonctions intrinsèques permet en outre l'insertion simple d'unités fonctionnelles utilisateurs et de leur instructions associées.

La raison pour laquelle le compilateur ne peut encore générer le code fonctionnel pour l'ensemble de l'algorithme EFR s'explique par la diversité des structures de calcul présentes dans les 66 fonctions composant l'algorithme. Le fichier de description du jeu d'instructions « ASIP.brg » utilisé par Olive n'est en effet pas encore suffisamment complet pour prendre en compte tous les scénarii possibles. L'obtention d'un version stable du compilateur passe donc par quelques ajouts dans ce fichier, auxquels il faut ajouter la correction de quelques inévitables « bugs », la bibliothèque SPAM n'étant pas exempte d'erreurs.

La première version du compilateur n'utilise cependant qu'une petite partie des optimisations proposées dans *SPAM*, et les améliorations à apporter sont nombreuses.

Prise en compte du parallélisme et des paramètres matériels configurables

L'optimisation la plus importante concerne la gestion du parallélisme d'instructions et des ressources matérielles. Le code que nous générons est de type séquentiel et ne sollicite qu'une seule unité fonctionnelle à la fois. L'obtention d'un code performant passe obligatoirement par l'exécution en parallèle d'instructions indépendantes du point de vue des données et ne sollicitant pas les mêmes ressources au même moment.

La première version de SPAM dispose d'algorithmes spécialement dédiés pour le processus de « compactage » des instructions élémentaires en macro-instructions VLIW. Elle propose deux classes spéciales implémentant deux types d'algorithmes de compactage, dont le plus connu est l'ordonnancement par liste. Ces algorithmes analysent les dépendances de données entre instructions, réordonnent et assemblent les instructions indépendantes. La prise en compte du matériel s'effectue grâce à une troisième classe (*CompactionTable*), dans laquelle le concepteur décrit les combinaisons parallèles d'instructions autorisées par le processeur et le jeu d'instructions, et qui prend donc en compte à la fois les contraintes matérielles de l'architecture (nombre et types des unités fonctionnelles, aléas de pipeline) et les contraintes d'encodage du jeu d'instructions. Le code résultant est capable d'utiliser plusieurs unités fonctionnelles en parallèle, et aussi de profiter de l'architecture à accès mémoire directe en parallélisant les accès et les calculs sur les données. La deuxième version de SPAM, apparue plus récemment, propose des algorithmes de compaction et d'ordonnancement plus évolués spécialisés pour les architectures VLIW[72].

L'implémentation de ces algorithmes permettrait de prendre en compte les degrés de liberté « structurels » du modèle de processeur. En l'état actuel, le compilateur gère uniquement l'aspect « nouvelles instructions utilisateur » au moyen des fonctions intrinsèques. Les contraintes liées au nombre d'unités fonctionnelles disponibles (déclarées par la même table d'allocation de ressources que pour le simulateur), à la connectivité et à la bande passante peuvent être pris en compte efficacement en utilisant les algorithmes généraux de compaction de SPAM.

Amélioration de la qualité du code

Bien que le code généré soit fonctionnellement correct, il est perfectible à de nombreux égards, en particulier sur les points suivants :

- Allocation de registre. L'algorithme de passage des registres virtuels d'*Olive* aux registres physiques du processeur est très rudimentaire. Pour améliorer les performances, SPAM dispose de plusieurs algorithmes d'optimisation facilement intégrables dans le code du compilateur. Ces algorithmes prennent en compte le nombre de registres disponibles du processeur, et sont capables de générer des instructions destinées à sauvegarder temporairement en mémoire puis restaurer certains registres pour « tenir » dans le nombre de registres impartis. Le choix des registres à sauvegarder est d'ailleurs primordial pour la performance et requiert une analyse de la durée de vie des registres, réalisée au moyen des algorithmes d'analyse « data-flow » de TWIF. On pourra de même utiliser les algorithmes destinés à améliorer l'allocation des registres pointeurs et le calcul des adresses pour les accès aux variables mémoire (« *Offset Assignment* »).
- Utilisation de la capacité d'exécution conditionnelle pour la génération de structures de type *if-then-else* n'utilisant pas d'instructions de saut. Ces dernières sont en effet très coûteuses en cycles à cause de la structure du pipeline. Lorsque le corps des basic-blocs associés aux structures *if*, *then* et *else* comporte peu d'instructions, l'utilisation concurrentielle des préfixes *IFF* et *IFT* permet d'améliorer considérablement les performances (cf. 3.6.5.3).
- et bien d'autres : intégration des techniques évoluées d'extension du parallélisme (déroulage de boucle, pipeline logiciel, modification de la structure du CDFG guidé par simulation),

« inlining » de certaines fonctions, prise en compte des structures de mémoires données hétérogènes (bancs multiples, simple et double ports), etc.

4.5 Méthode d'implémentation matérielle

Traditionnellement , les cœurs de processeur pour systèmes embarqués proposés par les grands fabricants sont disponibles sous deux formes : les cœurs « matériels » décrits au niveau transistors pour une technologie donnée , et les cœurs « logiciels » décrits en langage de description matériel de plus haut niveau (VHDL, RTL) et indépendants de la technologie cible. Les premiers sont intrinsèquement plus performants mais sont totalement liés à une technologie particulière, ce qui est problématique si l'on vise une intégration dans un système sur puce qui contient d'autres blocs matériels n'ayant pas forcément d'équivalents dans cette technologie. De plus, la description bas niveau de ce type de cœur offre des degrés de paramètrabilité beaucoup plus faibles que pour les cœurs logiciels qui sont décrits dans des langages de haut niveau. Pour la réalisation physique de notre processeur configurable, nous préconisons donc la solution logicielle, car elle offre une plus grande souplesse du point de vue de la technologie cible et permet d'atteindre un haut degré de configurabilité.

On doit ensuite définir le niveau et le type de description utilisé pour modéliser le processeur. Deux alternatives principales s'offrent à nous. On peut utiliser une description très haut niveau modélisant le comportement du processeur (en VHDL ou en Verilog par exemple) et obtenir la description physique en utilisant des outils de synthèse de haut niveau. La qualité de la représentation obtenu dépend de la performance des outils de CAO, et l'intervention du concepteur est très limité. Le principal avantage de cette méthode est la rapidité avec laquelle on peut obtenir la description matérielle. La deuxième alternative, dite « semi-automatique », consiste à utiliser un niveau de description inférieur (comme par exemple le niveau RTL) dans lequel le concepteur a une influence plus importante sur les choix d'implémentation matériels. Cette approche est mois immédiate mais permet d'obtenir pour le circuit final des caractéristiques (performance, consommation, coût) bien meilleures que celles obtenues par synthèse entièrement automatique. Nous proposons ici une méthode de réalisation semi-automatique basée sur l'usage de générateurs paramétrables et portables, et d'un langage de description de jeu d'instructions pour la synthèse de la partie contrôle.

4.5.1 Synthèse semi-automatique du chemin de données par l'emploi de générateurs paramétrables

Depuis plusieurs années des efforts considérables sont menés, dans notre laboratoire, pour le développement d'une bibliothèque d'opérateurs arithmétiques paramétrables, portables et très performants. Nous disposons maintenant d'un environnement de conception, GENOPTIM [73], qui a déjà permis de réaliser de nombreux circuits (ex : convolueur et réseau de neurones cellulaires [74], architectures pour la reconnaissance de formes [75], calcul de DCT [76]). La bibliothèque d'opérateurs contient à la fois des opérateurs arithmétiques (addition, multiplication, division, racine carrée) et des opérateurs à usage général (décaleur, compteur, banc de registres). Tous ces opérateurs sont disponibles sous forme de générateurs paramétrables reposant sur le concept de cellules virtuelles, qui permet une portabilité de l'opérateur sur n'importe quelle technologie.

4.5.1.1 Fonctionnement d'un générateur

Un générateur est un programme exécutable prenant en compte un certain nombre de paramètres, qui sont : le nom du bloc à générer, la taille du circuit en nombre de bits (correspondant à la précision pour les générateurs d'opérateurs arithmétiques) et une liste d'options (Figure 109). Ces options indiquent le type d'optimisation électrique et de placement à réaliser, la bibliothèque de portes logiques et le nom de la chaîne de CAO utilisée.

A partir de paramètres simples, le générateur crée automatiquement plusieurs vues: la liste d'interconnexions (Structurel), le placement de chaque porte logique, le comportement du circuit, le fichier de caractéristiques du circuit (Compte-rendu) et un jeu de vecteurs de test (Vecteurs).

GENOPTIM utilise une librairie de cellules virtuelles. L'architecture est projetée sur une librairie réelle au moment de l'exécution du générateur selon le choix spécifié par le concepteur. La génération d'un circuit se déroule en plusieurs étapes.

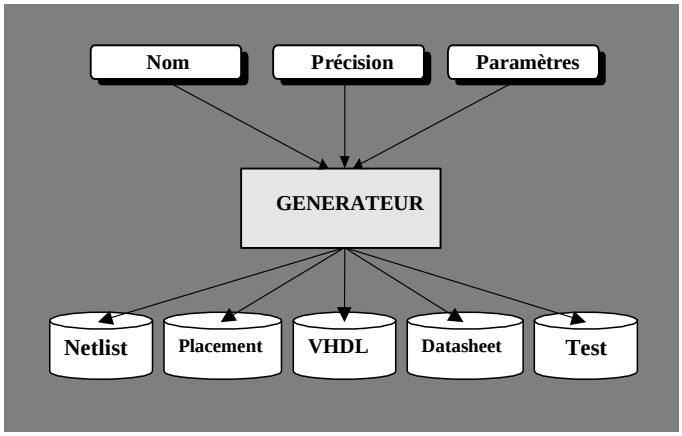


Figure 109 Interface d'un générateur GENOPTIM

Lors de l'exécution d'un générateur, GenOptim commence par construire les listes d'interconnexions et de placements de l'architecture en fonction de certains paramètres. Les paramètres minimum à fournir sont le nom du circuit, la taille en nombre de bits et la bibliothèque de cellules logiques cible. Le liste d'interconnexions et le placement sont ensuite mises à plat et toutes les hiérarchies sont éliminées. Les modifications et les optimisations de placement sont alors réalisées en prenant en compte la taille réelle des cellules. L'exécution, en amont, des modifications de placement garantit, par la suite, une estimation correcte des capacités de routage. Puis, une phase de projection structurelle transforme les cellules virtuelles en portes réelles de la bibliothèque cible. A partir des caractéristiques technologiques de la bibliothèque, GenOptim estime les capacités de routage et calcule les délais, afin de déterminer les zones critiques du circuit à optimiser. Ensuite, différentes techniques seront déployées pour limiter les effets capacitifs sur la chaîne longue, et, ainsi, diminuer le temps de propagation des blocs générés. Enfin, des traducteurs permettent de choisir le format des différentes vues, en fonction de la chaîne de conception de circuits utilisée. Le générateur est donc exécuté en prenant en compte des paramètres supplémentaires concernant, en outre, le choix du traducteur, le type de vues à générer et des directives d'optimisations électriques et de placements. Ces options sont données sur la ligne de commande ou par l'intermédiaire d'un fichier de «Paramètres».

A mi-chemin entre «blocs optimisés» (Full-Custom) et synthèse logique, cette méthode de conception permet la réalisation rapide de blocs portables, optimisés en surface et en performance. L'ensemble des blocs constitue une bibliothèque, comprenant plus de cinquante générateurs de base (arithmétique et usage général).

4.5.1.2 Intérêt pour la réalisation du chemin de données

Pour la réalisation d'un processeur DSP paramétrable, l'utilisation de macroblocs paramétrables et portables peut permettre d'accélérer considérablement la réalisation d'une grande partie du processeur, en particulier pour les blocs matériels réguliers du chemin de données. En effet, même si notre modèle de processeur permet l'intégration de ressources spécialisées « utilisateur », la plus grande partie du matériel est occupé par des ressources générales que l'on retrouve dans tous les processeurs DSP : unités de calcul (ALU, MAC, Décaleur), unités de calcul d'adresses (Additionneurs/Soustracteurs, bloc modulo ou bit–reverse), unités de mémorisation (banc de registres), unités de routage des données (multiplexeurs). Ces ressources possèdent en commun une grande régularité architecturale et se prêtent très bien à la modélisation sous forme de générateurs de bloc paramétrables.

Dès lors, l'unité de calcul CU et l'unité d'adressage AGU de notre processeur peuvent être modélisées sous forme de macro-générateurs faisant appel aux générateurs de la bibliothèque pour générer les différents blocs les constituant : unités de calcul, multiplexeurs, registres, etc. L'intégration dans ces unités de blocs spécialisés (coprocesseur Viterbi, unité de manipulation de bit) ne faisant pas partie de la bibliothèque requiert simplement de décrire ces blocs en langage GENOPTIM pour les interfacer avec le macrogénérateur. Quelques générateurs supplémentaires spécialisés « processeur DSP » ont déjà été développés: unités d'arrondi et de saturation , unité de calcul de modulo, fonction bit-reverse, multiplexeur paramétrable pour le routage de données.

Avec ces blocs particuliers, l'unité CU (Figure 64 de la page 88) peut par exemple être décrite presque uniquement à l'aide de générateurs de la bibliothèque : multiplexeurs de données paramétrables pour les routeurs de données (« *Mux Entrées UFs* » et « *Mux Sorties UF s* ») et les interfaces externes, générateur de banc de registres, et générateurs d'opérateurs arithmétiques de haut niveau pour les unités de calcul générales : ALU, Décaleur, BMU, etc. L'appel à des générateurs déjà existants et validés permet de réduire considérablement le temps de développement ; les seuls générateurs qui devront être écrits sont ceux implémentant des fonctionnalités non présentes dans la bibliothèque, comme par exemple des unités fonctionnelles spécialisées de type coprocesseur de Viterbi. Ces générateurs pourront ensuite être réutilisés dans d'autres réalisations.

L'estimation de complexité matérielle présentée au chapitre 5 repose sur la réalisation à l'aide de générateurs GENOPTIM de l'ensemble des éléments matériels composant les unités AGU, CU et une partie de l'unité PCU. L'encapsulation de ces générateurs en macrogénérateurs haut niveau modélisant ces unités ne présente à priori pas de difficulté particulière. Pour l'unité CU du processeur, on décrira dans le macrogénérateur associé l'architecture de l'unité (Figure 64 de la page 88) et une liste de paramètres structurels reflétant les différents degrés de spécialisation autorisé par l'unité : nombre et types d'unités fonctionnelles, largeur de données, nombre de registres et de ports d'accès, et connectivité des différents multiplexeurs. Le fait que les générateurs GENOPTIM soient décrits en langage C++ permet d'utiliser les structures de contrôle évolués de ce langage afin d'y insuffler

l'intelligence nécessaire pour la prise en compte des nombreux paramètres structurels. L'architecture VLIW étant particulièrement régulière, la seule réelle difficulté réside dans l'interfaçage avec des unités fonctionnelles spécialisées absentes de la bibliothèque de générateurs. Il suffit de passer en paramètre une table d'allocation (semblable à celle utilisée par les outils de génération de code) reliant les unités fonctionnelles spécialisées à leur générateur associé.

Au final, il est tout à fait possible de concevoir trois macrogénérateurs paramétrables : unité CU, unité AGU et partie « chemin de données » de l'unité PCU. Dans ce cas, la réalisation de toute la partie matérielle « chemin de données » du processeur revient à décrire un fichier de paramètres pour chaque macrogénérateur, à concevoir les générateurs pour la réalisation des blocs matériels utilisateurs, puis à « lancer » les trois générateurs. Le gain en temps de développement est alors significatif par rapport à une réalisation classique.

4.5.2 Synthèse de la partie contrôle

Comme on le verra plus loin (cf. 5.5), la structure d'encodage utilisée pour coder notre jeu d'instructions générique permet de faire varier la largeur du mot d'instruction (nombre d'opérations élémentaires codé dans un mot) et les formats d'encodage pour chaque type d'opération (largeur en bits, opcodes, opérandes, etc.). Pour la réalisation matérielle, cela signifie que le répartiteur d'opérations et les décodeurs locaux associés à chaque unité matérielle « commandable » du processeur (unités fonctionnelles de CU, UCAs, interface mémoire, etc.) doivent éventuellement être réécrits lorsque certaines caractéristiques du jeu d'instructions sont modifiées.

Pour la réalisation du répartiteur, qui n'est finalement qu'un routeur de données à architecture régulière, on peut envisager de définir un générateur prenant en paramètre le nombre d'opérations par instruction, la largeur de chaque champ d'opération et le nombre de décodeurs locaux cibles. Ce générateur produit alors les multiplexeurs nécessaires au routage de différents opérations vers les décodeurs locaux adéquats.

Les décodeurs locaux correspondent à des structures de contrôle peu régulières et ne se prêtent pas du tout à une modélisation sous forme de générateur GENOPTIM. Un langage de description comportemental haut niveau comme le VHDL constitue la méthode la plus simple pour décrire les équations logiques gérant l'activation des signaux de contrôle des unités en fonction de la valeur des opcodes des différentes opérations, l'image matérielle étant ensuite obtenu par synthèse logique. Il faut aussi noter qu'un autre fichier VHDL est nécessaire pour décrire les mécanismes de contrôle fixes (c.-à-d. indépendants des paramètres du modèle) du processeur : commandes du pipeline de contrôle, logique de contrôle des boucles matérielles, gel du processeur lors de certaines d'instructions, etc. Ce fichier, toujours très complexe à mettre au point quelque soit le processeur, décrit cependant des fonctionnalités beaucoup plus simples que celles requises par les processeurs généraux de type superscalaire ; c'est un des gros avantages de l'approche VLIW, qui délègue la plus grande partie des mécanismes de contrôle dynamique non pas au processeur mais au compilateur.

Avec cette méthode, la réalisation matérielle du processeur final requiert donc de paramétrer les différents générateurs du chemin de données, de décrire dans un fichier VHDL la structure d'encodage des opérations, puis d'assembler manuellement les blocs avant placement routage. Afin de simplifier le processus, il peut être utile d'utiliser un fichier unique décrivant l'ensemble des

informations nécessaires à la réalisation matérielle, mais aussi celles utilisés pour la configuration des outils logiciels de simulation et de compilation (cf. 4.4.2.1). Ces informations peuvent être centralisées à l'aide d'un langage de description de processeur, à partir duquel on peut alors paramétrer l'ensemble des outils de génération de code et de synthèse matérielle. On reviendra sur l'intérêt d'un tel langage dans la conclusion générale.

4.6 Conclusion

Nous avons présenté dans ce chapitre la méthodologie de conception associée au modèle de processeur configurable. Cette méthodologie se décompose en deux phases. La première consiste en une exploration de l'espace de configuration du processeur, afin de déterminer les meilleurs paramètres matériels et logiciels en fonction du cahier des charges de l'application cible. Elle repose sur une première étape d'analyse de complexité déterminant les fonctions critiques de l'application et les opérateurs de calcul à inclure dans le processeur. Elle requiert ensuite l'usage d'outils logiciels performants et paramétrables pour la génération de code et l'analyse de performance : simulateur de jeu d'instructions et compilateur. Pour la deuxième étape de réalisation matérielle, nous avons proposé une méthode basée sur l'emploi de macrogénérateurs paramétrables et portables.

Le chapitre suivant présente un exemple d'application de cette méthodologie pour la spécification d'un processeur ASIP implémentant les fonctions clés de la norme GSM, qui va nous permettre d'estimer la validité et l'intérêt du concept de modèle de processeur configurable.

Chapitre 5

Performances et estimation de complexité matérielle

Sommaire :

5.1 Introduction

Partant du modèle d'ASIP par défaut sans contraintes de ressources, nous allons maintenant mettre en œuvre la méthodologie de conception présentée précédemment afin d'obtenir une version finale de l'ASIP qui soit à la fois performante pour les applications visées et la plus économe possible en terme de coût matériel et de consommation. Les applications cibles sont ici constituées des fonctions « clé » de la norme de communication GSM : l'algorithme de Viterbi et le codeur de voix EFR.

La première étape de la méthodologie, l'analyse de complexité, nous a déjà permis d'identifier les fonctions critiques de l'application ainsi que les opérateurs arithmétiques indispensables qui devront être intégrés au jeu d'instructions du processeur. Ces opérateurs constituent pour l'instant le seul paramètre déjà fixé du modèle de processeur, et les étapes suivantes de la méthodologie vont devoir définir l'ensemble des autres paramètres de l'architecture et du jeu d'instructions.

Pour se faire une idée précise de la valeur des performances de l'ASIP final, nous allons les comparer avec celles d'autres processeurs DSP du commerce utilisés dans les télécommunications. Cette comparaison n'a de sens que si le parallélisme de l'architecture de ces processeurs est du même ordre que celui de notre ASIP. La plupart des processeurs DSP les plus récents intégrant des architecture VLIW Bi-MACs, nous avons volontairement décidé de limiter à 2 le nombre d'unités MAC dans le modèle d'ASIP, qui correspond de toute façon aux besoins moyens des applications comme le GSM. Le processeur choisi pour la plupart des comparaisons est le *Texas Instruments* TMS320C62, et ce pour deux raisons : d'une part, c'est le seul processeur pour lesquels on dispose de chiffres précis concernant l'implémentation des fonctions critiques de l'algorithme EFR. D'autre part, il intègre une architecture VLIW très générale qui constitue un bon point de comparaison par rapport à l'architecture résolument très spécialisée de l'ASIP.

Après un bref rappel des caractéristiques principales du TMS320C62, la section 3 présente l'étape de codage des fonctions critiques et compare les performances obtenues pour ces fonctions par le modèle d'ASIP Bi-MAC non paramètre et le TMS320C62. La configuration précise de l'ASIP et le problème de l'encodage des instructions VLIW sont présentés en section 4 et 5. Afin de choisir parmi plusieurs configurations de processeurs possibles, nous procédons à une estimation de la complexité matérielle de chaque solution (section 6). Nous pouvons alors conclure sur l'intérêt de notre modèle de processeur configurable pour la conception de processeurs spécialisés. En dernier lieu, nous présentons les travaux futurs et les perspectives de recherche dans le même domaine.

5.2 Le processeur TMS320C62x

Ce processeur a été le premier à délaisser l'architecture conventionnelle des processeurs DSP pour adopter une architecture plus régulière de type VLIW. Il a en quelque sorte « essuyé les plâtres » et introduit des techniques architecturales et logicielles dont certaines ont été reprises dans des processeurs plus récents, et d'autres abandonnées pour des raisons de performance ou de complexité matérielle. Son architecture rappelée Figure 110 est divisée en deux « clusters » contenant chacun un banc de registres et quatre unités fonctionnelles (L, S, M, et D). Nous ne rentrerons pas ici dans les détails de l'architecture, qui sont présentés et analysés de manière détaillée en [16]. Nous nous contentons de donner un aperçu de ses principales caractéristiques:

Le TMS320C62 est plus « général » du point de vue architecture et jeu d'instructions que les processeurs DSP antérieurs puisqu'il n'utilise pas d'unités dédiés pour les calculs d'adresse et ne prévoit pas de mécanismes d'auto-incrémentation pour les pointeurs. Ainsi, les calculs sur les adresses et sur les données sont effectués en parallèle dans le chemin de données, les unités fonctionnelles et les registres travaillant à la fois sur des variables et sur des pointeurs. De même, il ne possède pas de mécanismes matériels permettant l'exécution implicite et rapide des boucles « zéro-cycles », et intègre des unités fonctionnelles « Multiplieur » plutôt que des unités MAC comme dans la plupart des DSPs. Même si l'on retrouve dans le processeur de nombreuses fonctionnalités clairement orientées DSP (adressage circulaire, unité de normalisation), le TMS320C62 présente un degré de spécialisation moins élevé que les processeurs DSP classiques.

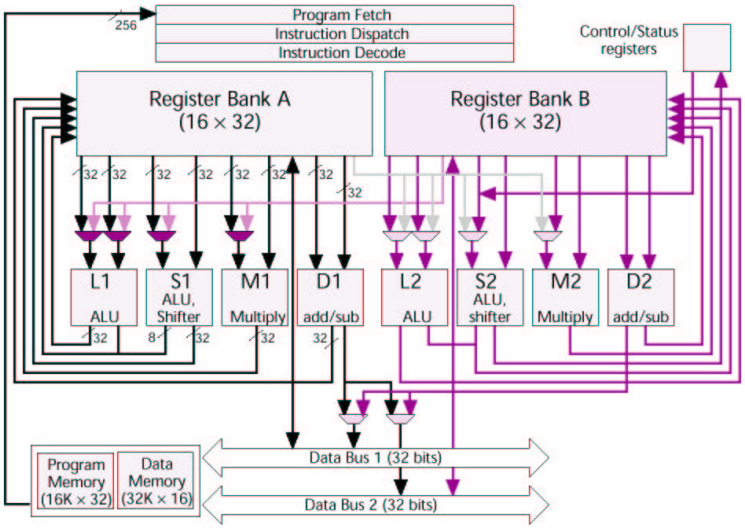


Figure 110 Architecture du TMS320C62

Parce qu'il est principalement destiné aux applications serveur réclamant de grandes puissance de calcul, le TMS320C62 a été conçu pour fonctionner à haute fréquence, et implémente pour cela un pipeline d'instructions très complexe de 11 étages, longueur très inhabituelle dans le monde des DSPs. Les opérations de calcul simples s'exécutent en un cycle, les multiplications en 2 cycles, les accès mémoire en 4 cycles et les branchements en 5 cycles. L'accès aux données en mémoire est de type « Load/Store », chaque calcul sur une donnée non présente dans le banc de registres devant être décomposée en une instruction d'accès et une instruction de calcul.

La largeur du mot d'instruction VLIW est de 256 bits et code jusqu'à 8 opérations s'exécutant en parallèle sur les 8 unités fonctionnelles du chemin de données. La répartition des différentes opérations vers les unités fonctionnelles concernées est prise en charge par l'unité de répartition (« Instruction Dispatch »), puis décodée à l'aide de l'unité de décodage (« Instruction Decode »).

La prise en compte du parallélisme d'instructions et des contraintes liées à la structure du pipeline rendent ce processeur extrêmement complexe à programmer, et requiert de nouvelles méthodologies de développement reposant de plus en plus sur les compilateurs plutôt que sur la programmation manuelle en assembleur. Pour illustrer cette complexité, il suffit d'examiner l'instruction principal du noyau de calcul d'un filtre FIR composée de 8 opérations parallèles (cf. Figure 111). A la nième exécution de cette boucle, les multiplications effectuées concernent le calcul du n-2ème élément, les additions le n-1ème, les chargements de données (LDW) le n-5ème, et le branchement le n-6ème. Ce genre

d'instructions est lié à l'usage de la technique du pipeline logiciel, utilisé pour masquer les effets de latence du pipeline. On comprend mieux à quel point la compréhension du code et à plus forte raison la programmation du processeur sont complexes. Néanmoins, pour son architecture VLIW générale de type Bi-MAC et ses excellentes performances pour les fonctions critiques des applications DSP, ce processeur est un bon élément de comparaison afin d'évaluer les performances de notre processeur.

```
loop:
        ADD.L1     A0,A3,A0   ;A0=A0+A3
||      ADD.L2     B1,B7,B1   ;B1=B1+B7
||      MPYHL.M1X  A2,B2,A3   ;A3=A2(hi)×B2(lo)
||      MPYLH.M2X  A2,B2,B7   ;B7=A2(lo)×B2(hi)
||      LDW.D2     *B4++,B2   ;load into B2
||      LDW.D1     *A7--,A2   ;load into A2
||      ADD.S2     -1,B0,B0   ;decrement counter
|| [B0] B.S1 loop             ;branch if B0 nonzero
```

Figure 111 Noyau de calcul du filtre FIR sur le TMS320C62

5.3 Analyse, codage et performances des fonctions critiques

A l'issue de la première étape de la méthodologie de conception (« Analyse de complexité », voir Chapitre 4), on dispose principalement de deux informations :

- La sélection des opérateurs arithmétiques indispensables au processeur, qui devront être implémentés dans les unités fonctionnelles du chemin de données. Ces opérateurs ont fait l'objet d'une première répartition dans différents types d'unités fonctionnelles : MAC, ALU, Shifter, etc (Figure 96 Chapitre 4).
- L'identification des fonctions critiques de l'application. C'est sur elles que doit porter l'effort d'analyse et d'optimisation. C'est à partir de leurs caractéristiques intrinsèques que seront définis la plupart des paramètres du modèle.

Cette étape est la plus déterminante de la méthodologie puisque c'est à ce niveau que l'on définit de manière précise l'architecture finale et le jeu d'instruction du processeur. Les fonctions critiques vont tout d'abord être programmées puis simulées sur un premier modèle du processeur très général dont les caractéristiques sont les suivantes :

- Les unités fonctionnelles sont celles décrites au chapitre 4 Figure 96 à l'issue de la sélection des opérateurs. Le nombre pour chaque type d'unité est encore non défini et par défaut supposé infini, sauf pour les unités MAC dont le nombre est fixé à 2 maximum pour les raisons de comparaison avec les autres processeurs DSP. Les opérateurs n'ayant pas d'*UFs* associées sont considérés comme faisant partie d'une *UF* fictive appelée « Autres » dont le nombre est lui aussi supposé infini. Il en est de même pour les *UCAs* de l'unité de calcul d'adresses, chacune d'entre elles étant supposée implémenter l'ensemble des modes d'adressages définies en 3.6.3. Il est aussi possible de simuler le comportement d'instructions supplémentaires complexes « utilisateur », à la condition que le comportement de ces instructions ait été décrit et rajouté au code du simulateur d'instructions (cf. 4.4.2.6).
- Les nombres de registres de calcul de l'unité CU, de registres pointeurs de l'unité AGU, de registres SB (qui détermine la bande passante vers la mémoire *Données*) et de registres de

boucles zéro-cycles de l'unité PCU sont supposés infinis. Il n'existe enfin aucune contrainte de connectivité entre les éléments composant les différents unités (registres et unités de calcul). Ce modèle général correspond au modèle par défaut du simulateur d'instruction (cf. 4.4.2.2).

Les prochaines sections présentent les résultats en performance obtenus pour les deux fonctionnalités les plus importantes de la norme de base GSM : le codeur EFR et l'algorithme de Viterbi. Ces chiffres sont issus de la simulation du code des fonctions critiques sur le modèle général du processeur grâce au simulateur de jeu d'instruction. La méthode employée pour coder et optimiser une fonction critique est tout d'abord illustrée au travers de l'exemple de la fonction « search_10i40 », la fonction la plus complexe du codeur GSM. Les résultats obtenus pour 12 fonctions critiques de l'EFR sont alors comparés à ceux du processeur de référence ayant servi lors de l'analyse de complexité, ainsi qu'à ceux provenant de l'implémentation sur le processeur TMS320C62x. On s'intéresse ensuite à l'implémentation de l'algorithme de Viterbi, de l'analyse de l'algorithme jusqu'à la définition des structures matérielles spécialisées et leurs instructions associées qui seront incluses dans le processeur. En dernier lieu, on comparera les performances du modèle d'ASIP et du TMS320C62x sur le système complet « EFR+Viterbi ».

5.3.1 Exemple de codage d'une fonction critique : la fonction search_10i40 de l'encodeur

La fonction de recherche dans le livre de code algébrique de l'encodeur appelée « search_10i40 » représente à elle seule un cinquième de la charge de calcul requise par le système encodeur/décodeur de l'EFR. Il est donc indispensable d'optimiser son exécution sur notre processeur cible. Conformément aux principes de la méthodologie énoncés précédemment, la fonction sera codée manuellement en assembleur, en privilégiant la performance plutôt que la taille du code. L'analyse des parties les plus critiques de l'algorithme va nous amener à proposer plusieurs solutions d'optimisation, basées sur l'emploi de techniques logicielles d'exploitation du parallélisme et sur l'utilisation d'instructions et de structures matérielles spécialisées. Le code correspondant à chaque solution est à chaque fois validé avec le simulateur d'instructions en utilisant les patterns de test fournis par l'ETSI. Selon le coût matériel et les gains en performance atteignables, seules un certain nombre de ces solutions seront retenues et implémentées dans la version finale du processeur.

5.3.1.1 Structure et complexité de la fonction

La longueur et la structure complexe des calculs mis en œuvre dans la fonction *search_10i40* la rendent plus difficile à optimiser que les algorithmes classiques de type filtre, dans lesquels il suffit généralement de dérouler quelques boucles pour exposer suffisamment de parallélisme et permettre une implémentation efficace sur la plupart des processeurs DSP. La fonction travaille sur un sous-ensemble de 40 échantillons d'entrée, et doit déterminer la position optimale de dix impulsions *{i0, i1….i9}*, chaque impulsion ayant une amplitude de +1 ou –1 et pouvant prendre un nombre fini de positions (10) parmi les 40 possibles (d'où son nom). La recherche des impulsions se fait par groupe de deux, et est répétée 4 fois après rotation circulaire des impulsions (Figure 112).

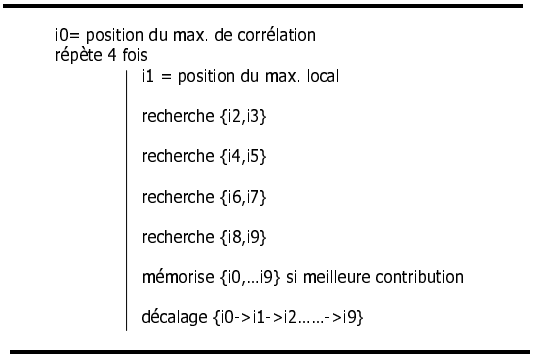


Figure 112 Structure globale de la fonction

Les 4 fonctions de recherche sont quasi-identiques du point de vue de la structure, chacune étant formée de trois boucles *for* dont deux sont imbriquées (boucles 2 et 3 de la Figure 113). Seuls le nombre d'instructions dans la boucle 1 et la première partie de la boucle 2 varie selon le couple d'impulsions cherché (pour le calcul de alp1, de 3 instructions pour {i2,i3} à 9 instructions pour {i8,i9}). La performance de cette fonction de recherche sur un processeur classique est limité par quatre facteurs principaux :

- Grande dépendance entre les données : les instructions responsables du calcul de *rrv[ix]* dans la boucle 1,de *alp1* dans la boucle 2 et des variables de la boucle 3 sont très dépendantes les unes des autres et exposent du coup très peu de parallélisme d'instructions. L'emploi des techniques de déroulage de boucles et de pipeline logiciel sont indispensables pour profiter du parallélisme matérielle du processeur.

```
/*----------------------------------------------------------------*
 * i8 and i9 loop:                                                *
 *----------------------------------------------------------------*/

ps0 = ps;
alp0 = L_mult (alp, _1_2);

for (i9 = ipos[9]; i9 < L_CODE; i9 += STEP)
{
   s = L_mult (rr[i9][i9], _1_16);
   s = L_mac (s, rr[i0][i9], _1_8);
   s = L_mac (s, rr[i1][i9], _1_8);
   s = L_mac (s, rr[i2][i9], _1_8);
   s = L_mac (s, rr[i3][i9], _1_8);
   s = L_mac (s, rr[i4][i9], _1_8);
   s = L_mac (s, rr[i5][i9], _1_8);
   s = L_mac (s, rr[i6][i9], _1_8);
   s = L_mac (s, rr[i7][i9], _1_8);
   rrv[i9] = round (s);
}

/* Default value */
sq = -1;
alp = 1;
ps = 0;
ia = ipos[8];
ib = ipos[9];

for (i8 = ipos[8]; i8 < L_CODE; i8 += STEP)
{
   ps1 = add (ps0, dn[i8]);

   alp1 = L_mac (alp0, rr[i8][i8], _1_128);
   alp1 = L_mac (alp1, rr[i0][i8], _1_64);
   alp1 = L_mac (alp1, rr[i1][i8], _1_64);
   alp1 = L_mac (alp1, rr[i2][i8], _1_64);
   alp1 = L_mac (alp1, rr[i3][i8], _1_64);
   alp1 = L_mac (alp1, rr[i4][i8], _1_64);
   alp1 = L_mac (alp1, rr[i5][i8], _1_64);
   alp1 = L_mac (alp1, rr[i6][i8], _1_64);
   alp1 = L_mac (alp1, rr[i7][i8], _1_64);

   for (i9 = ipos[9]; i9 < L_CODE; i9 += STEP)
   {
      ps2 = add (ps1, dn[i9]);

      alp2 = L_mac (alp1, rrv[i9], _1_8);
      alp2 = L_mac (alp2, rr[i8][i9], _1_64);

      sq2 = mult (ps2, ps2);

      alp_16 = round (alp2);

      s = L_msu (L_mult (alp, sq2), sq, alp_16);

      if (s > 0)
      {
         sq = sq2;
         ps = ps2;
         alp = alp_16;
         ia = i8;
         ib = i9;
      }
   }
}
i8=ia;
i9=ib;
```

boucle 1

boucle 2

boucle 3

Figure 113 Recherche des impulsions {i8,i9}

- Coût des tests : la présence d'une structure en « *if* » dans la boucle de niveau le plus profond est très problématique du point de vue de la performance, car elle implique l'utilisation d'instructions de test et de sauts conditionnels, très coûteux en nombre de cycles du fait de la structure de contrôle pipelinée du processeur. Elle constitue de plus une rupture dans le flot d'instructions exécutables en parallèle car toutes les instructions suivantes dépendent du résultat de la structure « if », et interdit donc le déroulage de boucle et le pipeline logiciel.
- Adressage de tableaux bidimensionnels : l'architecture des unités de calcul d'adresses n'autorise que l'adressage de type « base + offset » et est insuffisante pour calculer en un cycle l'adresse d'un accès à un tableau à deux dimensions. Comme ces calculs nécessite des multiplications, ils doivent être effectués dans l'unité de calcul générale, puis le résultat est transféré dans un registre d'adresse pour accéder à la donnée. De nombreux cycles supplémentaires sont donc nécessaires par rapport à un adressage unidimensionnel.

5.3.1.2 Codage et première simulation sur le modèle par défaut

La première étape consiste à coder en assembleur la fonction « telle quelle », c'est à dire sans restructurer le code et sans chercher à exploiter le parallélisme éventuellement disponible, ce qui revient à transformer l'une après l'autre chaque ligne du source C de la fonction en son équivalent en assembleur. Le code assembleur est ensuite simulé en utilisant le modèle d'exécution par défaut du simulateur, qui modélise un processeur sans aucune contrainte de parallélisme ou de jeu d'instruction (cf. 4.4.2.2). De par la nature séquentielle du code simulé, le nombre de cycles obtenu est proche de celui qu'on obtiendrait sur un processeur DSP mono-scalaire. Pour ce modèle, une occurrence d'exécution de la fonction prend sur le simulateur **13500 cycles**.

5.3.1.3 Optimisation de la performance

A partir de ce code de base, on va appliquer plusieurs types de modifications afin d'accroître la performance. Au vu de la structure de la fonction, on a tout intérêt à privilégier les modifications qui accélèrent les boucles de plus grande profondeur, en l'occurrence les trois boucles des 4 fonctions de recherche.

Décomposition de somme

La première méthode consiste à exploiter le parallélisme d'instructions présent dans les boucles en parallélisant tous les calculs non contraints par des dépendances de contrôle ou de données. Le nombre d'opérations par instruction autorisé est illimité, la seule limite concerne les deux opérations de multiplication par cycle (architecture 2 MACs). Les opérations successives de multiplication-accumulation utilisées dans le calcul des termes *rrv[i9]* et *alp1* utilisent la même variable d'accumulation ***s***, et sont de ce fait toutes interdépendantes, n'offrant aucune possibilité d'exécution simultanée des calculs. On peut cependant contourné le problème en décomposant la somme de produits ***s*** en deux sous-sommes ***s1*** et ***s2*** pouvant s'exécuter en parallèle. Une opération supplémentaire devra être ajoutée à la fin pour sommer ***s1*** et ***s2***.

Cette technique dite de « décomposition de somme » permet d'effectuer 2 multiplications en parallèle et donc de tirer un meilleur profit de l'architecture. Elle est intéressante dès lors que le nombre de

termes à sommer dépasse 4 et est très utilisée pour l'implémentation d'algorithmes réguliers basés sur des sommes de produits comme les fonctions de filtrage ou de convolution. Elle présente cependant un léger inconvénient lié à la précision des calculs. La décomposition en sous-sommes a en effet pour conséquence que les produits sont sommés dans un ordre différents de celui spécifié dans la description d'origine. Si les opérateurs de sommation ont un comportement linéaire, cela ne pose pas de problème. L'opérateur *L_mac* utilisé dans le code C de référence ETSI prévoit la saturation automatique du résultat si l'opération provoque un débordement de capacité, introduisant une composante non-linéaire dans le comportement de l'opérateur. Dans ce cas, la précision des résultats au bit-près ne peut être garantie car le résultat final dépend de l'ordre dans lequel les sommes ont été effectuées. Si dans le cas de la fonction search_10i40, ce cas de figure ne s'est pas présenté, on le retrouve par exemple dans le cas de la fonction décrivant le filtre de synthèse de voix *syn_filt*. Les résultats renvoyés par la fonction, bien que très proches en valeur absolue de ceux du code de référence, ne sont cependant pas rigoureusement exacts et ne peuvent convenir pour une implémentation au bit-près.

La technique de décomposition a été utilisée pour accélérer la boucle 1 et la première partie de la boucle 2, à travers le calcul de *rrv[]* et de *alp1*. La structure de la boucle 3 offre à l'origine un peu plus de parallélisme d'instruction, permettant par exemple le calcul de *ps2* et *alp2* en parallèle, mais ce parallélisme est limité par la structure de contrôle en *if*. Un appel de la fonction prend alors **9370 cycles**, correspondant à un gain en performance de 30%.

Pipeline logiciel

Les instructions de la boucle 3 des fonctions de recherche sont celles qui sont exécutées le plus grand nombre de fois : 256 par appel de la fonction. Cette boucle étant identique pour les quatre fonctions de recherche, la réduction de la longueur de la boucle d'une seule instruction permet d'économiser 1024 cycles, soit plus de 10% du coût global de la fonction. A cause des nombreuses dépendances de données présentes dans la boucle, on va devoir avoir recours à la technique du pipeline logiciel pour augmenter le parallélisme exploitable. Cette technique consiste à « replier » les instructions de fin de boucle en début de boucle, l'occurrence *N* d'une boucle exécutant simultanément le début du traitement de l'échantillon *N* et la fin du traitement de l'échantillon *N-1*. Dans le cas de la boucle 3, la technique a consisté à replier les instructions d'affectation qui suivent le test au début de la boucle, réduisant ainsi la longueur de deux instructions. Le coût en cycles par appel passe alors à **7222 cycles**, soit un gain en performance supplémentaire de 16%.

Ajout d'une instruction supplémentaire

L'instruction de test de la boucle 3 compare le résultat d'une multiplication-soustraction par rapport à zéro et affecte le drapeau de condition qui sera utilisé pour conditionner les instructions d'affectation qui suivent. L'instruction de comparaison qui serait normalement réalisée par une instruction retardée « testd » peut être supprimée en remplaçant l'instruction « *l_msu* » par une instruction « *l_msu_with_test* » effectuant le même calcul mais affectant directement au bit T la valeur de la comparaison du résultat du calcul par rapport à zéro. Cette nouvelle instruction nécessite seulement le rajout d'un bloc combinatoire de détection de la valeur zéro dans l'unité MAC, dont le coût matériel est insignifiant par rapport au reste de l'unité. Au niveau encodage, son coût est aussi nul car il reste

des opcodes de commandes non utilisés pour l'unité MAC (cf. Figure 131 page 185: seulement 6 opcodes sur 8 - encodage 3 bits- sont utilisés). Le gain obtenu est pourtant significatif : la compaction de deux instructions en une seule permet de diminuer encore d'une instruction la boucle 3 en poussant plus loin le pipeline logiciel. Le coût d'un appel n'est alors plus que de **6166 cycles**, soit un gain supplémentaire de 8%.

Traitement « multi-échantillons »

La technique dite du « déroulement de boucle » permet d'augmenter le taux de parallélisme exploitable pour des algorithmes travaillant sur plusieurs échantillons de données et appliquant séquentiellement le même traitement à chaque échantillon. Si les traitements entre échantillons ne sont pas dépendants du point de vue des données, il est alors possible de modifier la description de l'algorithme de manière à traiter plusieurs échantillons à la fois. Dans ce cas, chaque opération constituant le traitement est appliquée en parallèle autant de fois qu'il y'a d'échantillons à traiter, à la manière des opérations SIMD. Si les ressources matérielles du processeur sont suffisantes, le traitement de *N* échantillons en parallèle permet en théorie une accélération de l'algorithme d'un facteur *N*.

L'examen des dépendances de données existantes dans les fonctions de recherche montre que les opérations nécessaires aux calculs de *rrv[ix]* (boucle 1) et à la sélection des couples d'impulsions pertinents (boucle 2 et 3) peuvent être effectuées en parallèle puisque elles ne travaillent pas sur les mêmes données. Le calcul des termes *rrv[]* peut donc être accéléré en déroulant la boucle 1 d'un facteur correspondant au nombre d'échantillons à traiter. Notre degré de parallélisme d'instructions maximum étant de deux MACs par cycle, la boucle a donc été déroulée d'un facteur 2. Le gain en performance est légèrement inférieur à 2, à cause de la contrainte d'une seule opération *round* par instruction. Cette contrainte provient du fait que le modèle final du processeur ne comporte qu'une seule ALU (l'unité implémentant *round*). Nous expliquerons ce choix plus en détail en 5.3.2.1.

Suivant le même principe, nous avons déroulé la boucle 2, ce qui nous a permis de paralléliser les opérations de la boucle 2 mais aussi de la boucle3. La sélection des couples d'impulsions se fait maintenant deux par deux, chaque occurrence de la boucle 2 analysant deux couples distincts {i2_1, i3} et {i2_2, i3}. Grâce à l'emploi de l'instruction « *l_msu_with_test* » et en appliquant la technique du pipeline logiciel pour accélérer la boucle 3, la fonction ne prend plus que **5283 cycles** (cf. Figure 149 et Figure 150 de l'annexe B). Quand elle est applicable, cette technique du traitement multi-échantillons est très efficace et permet d'étendre le parallélisme jusqu'aux limites du parallélisme matériel du processeur (dans notre cas 2 opérations MAC par instruction). Dans ce cas précis, elle permet d'obtenir des performances meilleures que pour la décomposition de somme.

5.3.1.4 Conclusion

Le facteur d'accélération final par rapport à l'implémentation sur le modèle de départ avec une seule unité fonctionnelle est de ***2,55***. Il est du en premier lieu à l'exploitation efficace du parallélisme architectural du processeur: unités fonctionnelles multiples et unités d'adressage travaillant en parallèle avec l'unité de calcul. Les deux unités MACS sont le plus souvent employées au maximum, ce qui permet d'obtenir pour la boucle 1 un facteur d'accélération proche de 2.

Au delà du facteur 2, le surplus d'accélération est plutôt à mettre à l'actif du jeu d'instructions. La largeur théoriquement infinie du mot d'instruction permet d'encoder un grand nombre d'opérations s'exécutant en parallèle, dont plusieurs sont de simples opérations de transfert de données qui ne nécessitent pas d'unités fonctionnelles supplémentaires mais qui sont requises lorsqu'on applique les techniques de pipeline logiciel et de déroulement de boucles. Effectuer les transfert de données en parallèle avec les calculs des UFs proprement dits permet ainsi d'atteindre des facteurs d'accélération supérieurs au nombre d'unités fonctionnelles. La première instruction de la boucle 3 Figure 149 effectue par exemple dans la même instruction un calcul d'adresses, une addition et 5 opérations de transfert, soit un total de 7 opérations dans la même instruction. Un tel nombre de transferts impose évidemment de prévoir des mécanismes matériels appropriés dans l'unité CU pour faire transiter les données, ces mécanismes étant cependant moins coûteux que l'ajout d'unités fonctionnelles.

Dans le cas de la fonction « search_10i40 », les transferts effectués en parallèle sont des opérations conditionnelles résultant du test « s>0 ». L'emploi de la fonction spéciale « l_msu_test » permet de compacter en une seule opération les opérations de multiplication-accumulation et de test par rapport à la valeur zéro. Cette opération s'effectue en un cycle, et fonctionne comme l'instruction retardée « testd » qui affecte au bit T la valeur de la comparaison sans geler le pipeline du processeur. Sans le mécanisme d'instruction retardée, l'instruction de test prendrait non pas un mais 4 cycles, ce qui diviserait la performance de la fonction par deux. De même, sans le mécanisme d'exécution conditionnelle, la structure en « if » devrait être codée au moyen d'instructions de saut conditionnels, ce qui aurait un effet désastreux sur la performance. On peut d'ailleurs estimer le gain lié à ces mécanismes en comparant le résultat de la fonction sur le modèle de départ avec une seule UF (13500 cycles), et le résultat du même modèle utilisant la structure en « if » classique avec un test non retardé et un saut conditionnel (16950 cycles). Pour la présente fonction, la seule utilisation de ces mécanismes de contrôle permettent déjà de réduire la charge de calcul de 20% (Figure 114).

L'examen des trois boucles critiques de la fonction prouve aussi tout l'intérêt de l'architecture à accès mémoire direct, la quasi totalité des instructions de ces boucles effectuant au moins un voire deux accès à la mémoire. Effectuer le même travail avec une architecture « load/store » aurait nécessité plus de registres pour mémoriser les données accédées, mais aussi un mot d'instruction plus large car le nombre d'opérations à encoder par instruction augmente du fait des opérations supplémentaires de lecture/écriture mémoire. La possibilité d'écrire un résultat de fin de boucle directement en mémoire (dernière instruction de la boucle 1) évite d'avoir à employer la méthode complexe du pipeline logiciel pour faire disparaître la pénalité d'un cycle liée à l'instruction supplémentaire d'écriture mémoire. Ce type d'architecture contribue donc aussi à diminuer la complexité des compilateurs pour l'obtention de code de qualité.

Pour finir, on notera que la seule instruction « l_msu_test » offre un gain en performance de 16% par rapport à la version optimisée multi-échantillons et 6% sur l'ensemble de la fonction. Elle n'est employée que deux fois, mais permet de réduire de 2 cycles chaque occurrence de la boucle 3, soit un total de 1024 cycles sauvés par appel. Sur l 'ensemble du codeur EFR (encodeur + décodeur), le gain supplémentaire est d'environ 4% par rapport à la version optimisée multi-échantillons, ce qui est peut être intéressant compte-tenu de la très faible complexité matérielle requise (une détection de zéro).

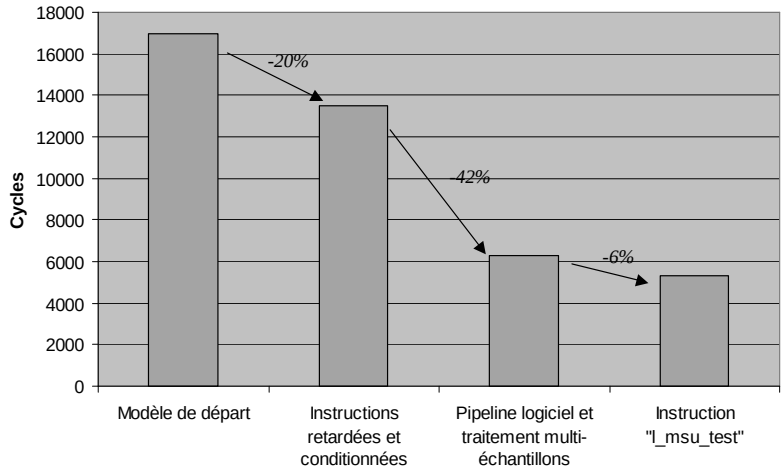


Figure 114 Contributions successives des différentes techniques matérielles/logicielles à l'optimisation de la performance de « search_10i40 »

5.3.2 Résultats pour l'ensemble des fonctions critiques

Les mêmes techniques de codage présentées pour la fonction « search_10i40 » ont été utilisées pour optimiser l'exécution de 11 autres fonctions critiques de l'EFR. On présente ici les résultats pour l'ensemble des fonctions critiques représentant environ 80% de la charge de calcul du codeur EFR. Les chiffres sont donnés en terme de nombre de cycles moyen par appels, certaines des fonctions incluant des structures de contrôle qui rendent le nombre de cycles par appels dépendant des données d'entrée. La prochaine section compare ces résultas à ceux issus de l'analyse de complexité, pour évaluer le gain de performance par rapport à un processeur DSP classique de type mono-scalaire, et identifier les caractéristiques les plus déterminantes de l'architecture et du jeu d'instructions qui doivent être intégrées à la version finale de l'ASIP. La section suivante s'intéresse aux résultats de ces mêmes fonctions sur un processeur DSP du commerce, le TMS320C62. Les différences de performance sont examinées et permettent de mettre à jour les forces et faiblesses respectives des approches « processeur spécialisé » et « processeur généraliste » pour des algorithmes de traitement du signal complexes de type EFR.

5.3.2.1 Gain par rapport au processeur de référence

La Figure 115 présente les charges de calcul requises par chaque fonction critique exprimées en millions de cycles par seconde (*MCPS*), ces chiffres étant calculés de manière à satisfaire la contrainte temps réel de l'EFR : traitement d'une séquence de 160 échantillons sonores d'entrée toutes les 20ms. Les chiffres donnés pour le processeur de référence sont ceux issus de l'analyse de complexité (cf. 4.4.1.4). Les chiffres du modèle ASIP se basent sur la même architecture Bi-MAC que celle utilisée pour le codage de « search_10i40 ». Le nombre d'unités MAC mis à part, le modèle du processeur est toujours celui par défaut (cf. 4.4.2.2), c'est à dire qu'il n'existe aucune contrainte matérielle concernant les transferts de données, le nombre d'opérations maximum exécutables en parallèle, etc.

Fonctions	*Charge de calcul (MCPS)*
search_10i40	3.52
norm_corr	1.83
lag_max	1.25
cor_h	1.07
syn_filt	1.07
az_lsp	1.07
Vq_subvec	0.71
Vq_subvec_s	0.53
Residu	0.45
Pred_lt_6	0.42
Autocorr	0.41
convolve	0.20
Total	12.54

Figure 115 Charge de calcul requise par fonction critique

Ces fonctions ont été codées selon le même principe que la fonction « search_10i40 » présentée précédemment, avec comme objectif unique l'optimisation de la performance, et en utilisant les mêmes techniques:

- Déroulement des boucles internes ou externes pour exposer plus de parallélisme.
- Pipeline logiciel
- Réduction des pénalités d'implémentation des structures en « if » par le conditionnement d'instructions et l'usage d'instruction de test « retardées ».

Dans un seul cas, celui de la fonction « autocorr », une méthode plus complexe destinée à accroître le parallélisme exploitable a été employée, qui consiste à supprimer une boucle 1 en déplaçant son contenu vers deux autres boucles (2 et 3), ce qui permet d'accroître le parallélisme exploitable dans 2 et 3 et de pouvoir profiter de l'architecture parallèle du processeur (Figure 116). Dans le cas de « autocorr », cette opération est particulièrement payante car lors de la plupart des appels de la fonction, la boucle 3 n'est pas exécutée (*sum* est presque toujours inférieur à MAX32). Du coup, le fait de rassembler les boucles 1 et 2 permet de les exécuter simultanément et diviser le temps d'exécution par2. Ce type de restructuration du code est cependant rarement applicable et la plupart du temps, les seules restructurations possibles consistent à simplement dérouler les boucles.

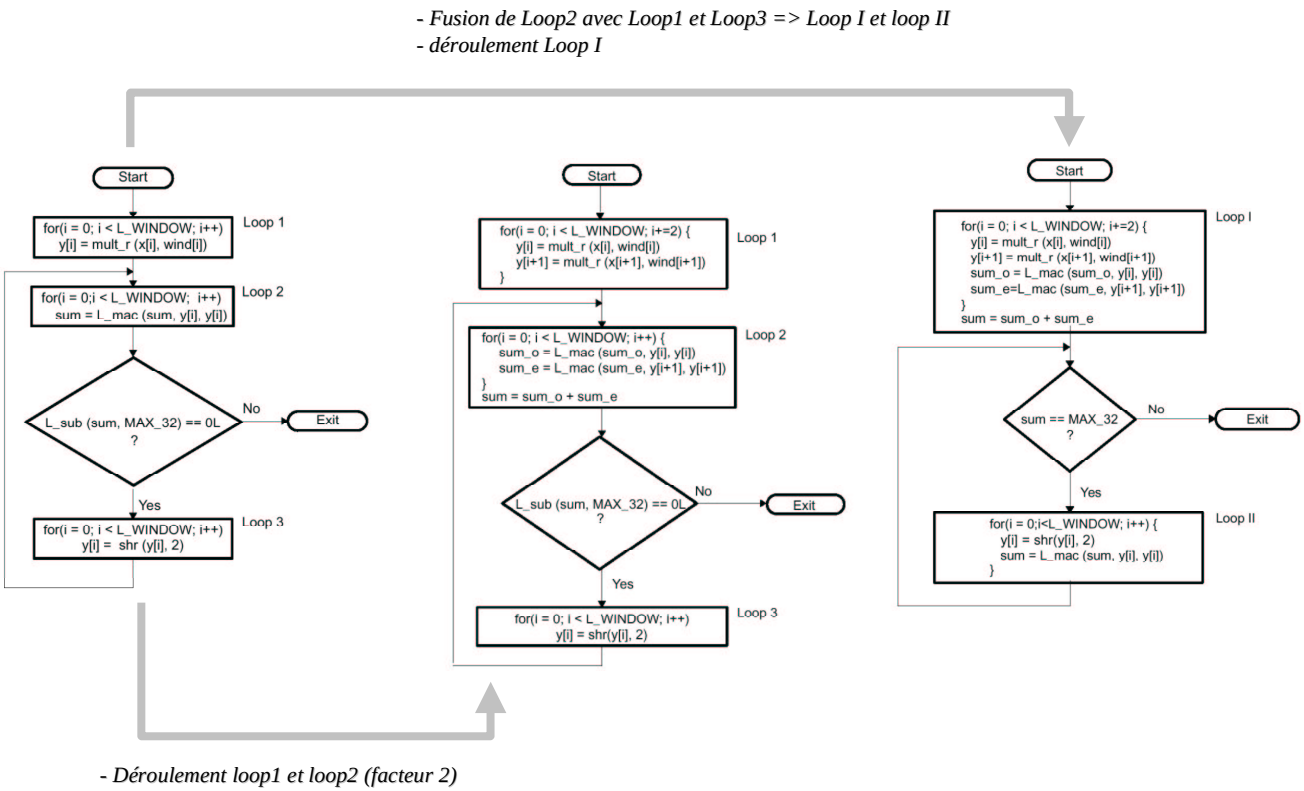


Figure 116 Exemples de restructuration de la fonction « Autocorr »

Les méthodes d'optimisation employées varient selon les fonctions (Figure 117), la seule technique utilisée presque systématiquement étant le déroulement de boucle. Les gains en performance par rapport au processeur de référence sont eux aussi très variables (Figure 118). Les deux fonctions qui tirent le plus de bénéfices de l'optimisation sont *search_10i40* et *Vq_subvec_s*. Le fait que le gain soit supérieur à 2 peut dans un premier temps paraître surprenant si l'on considère que le processeur ne dispose que de deux unités MAC, et que ce nombre détermine la limite d'accélération atteignable pour les algorithmes DSP classiques basés sur des successions de multiplication-accumulation. Comme pour *search_10i40*, le surcroît en performance de *vq_subvec_s* est du à la meilleure implémentation des structures en « if », le coût d'un « if » sur le processeur de référence étant de 7 cycles. Cette explication s'applique aussi aux fonctions *vq_subvec* (facteur 2,4) et *norm_corr* (facteur 2.25), le facteur 2.3 de *autocorr* provenant quant à lui de la restructuration du code.

Fonctions	*Techniques d'accélération*
search_10i40	Déroulement de boucle, pipeline logiciel, instruction spécialisée, structure en « if » accélérée
norm_corr / inv_sqrt	déroulement de boucle, exploitation // intrinsèque, structure en « if » accélérée
lag_max	déroulement de boucle
cor_h	déroulement boucle interne
syn_filt	déroulement total de la boucle interne, copie locale tableau de coeffs vers registres, pipeline logiciel
az_lsp / chebps	exploitation // intrinsèque
Vq_subvec	exploitation du // de la boucle interne, pipeline logiciel, 1 structure en « if » accélérée
Vq_subvec_s	exploitation du // de la boucle interne, pipeline logiciel, 2 structures en « if » accélérées
Residu	déroulement total de la boucle interne, copie locale tableau de coeffs vers registres, pipeline logiciel
Pred_lt_6	déroulement de boucle
Convolve	déroulement de boucle
Autocorr	Restructuration du code C, pipeline logiciel

Figure 117 Méthodes employées pour l'optimisation des fonctions critiques.

Les autres fonctions, « az_lsp » mis à part, ont toutes un facteur d'accélération proche de 2, conséquence directe d'un déroulement de boucle d'un facteur 2 permettant d'utiliser les deux unités MAC. Les valeurs légèrement supérieures proviennent de l'exécution en parallèle de certaines opérations présentes à l'extérieur des boucles critiques, l'architecture à accès mémoire directe permettant par exemple d'accéder à une donnée et de lui appliquer une opération dans la même instruction.

La seule fonction présentant un faible facteur d'accélération est « az_lsp ». Cette fonction fait de nombreux appels à une sous-fonction appelée « Chebps » évaluant la valeur d'un polynôme. Les calculs effectués dans cette fonction se font principalement en double précision et présentent de nombreuses dépendances de données qui limitent très fortement le parallélisme d'instructions, et par là même la performance de la fonction.

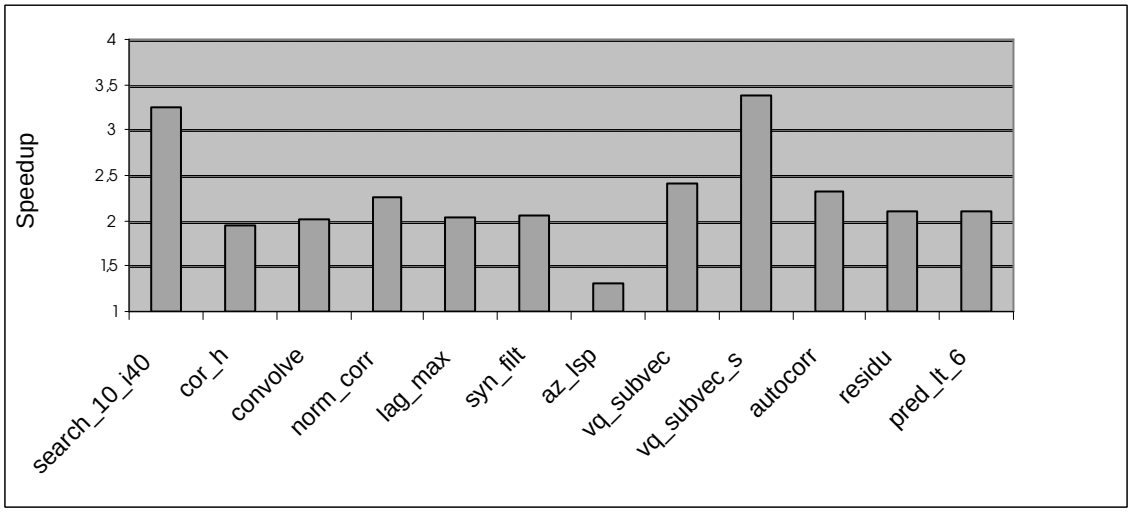


Figure 118 Facteur d'accélération ASIP/Processeur de référence

La programmation de ces fonctions critiques permet de tirer des premiers enseignements sur le nombre d'unités fonctionnelles nécessaires à l'implémentation efficace de l'algorithme. La majorité du temps, l'exploitation du parallélisme des boucles critiques est uniquement limité par le nombre de MACs. Si on se base sur une répartition « logique » des opérateurs arithmétiques dans les différents types d'unités fonctionnelles (opérateurs de multiplication et d'addition/soustraction dans les MACs, opérateurs de décalage dans le décaleur, opérateurs de manipulation de bits dans le BMU, addition/soustraction et le reste dans l'ALU) on s'aperçoit très rapidement qu'une seule ALU et une seule unité faisant à la fois office de décaleur et de BMU suffisent dans la grande majorité des cas pour assurer une performance maximale. Les seules exceptions concernent les fonctions « cor_h », « convolve » et « norm_corr », la première réclamant une deuxième ALU et les deux autres un deuxième décaleur. Dans le cas de « cor_h », le problème peut être contourné en décalant les instructions de l'ALU à l'aide du pipeline logiciel. Pour les deux autres fonctions, les pertes en performance sont respectivement de 4 et 8%, ce qui est loin d'être suffisant pour justifier l'ajout d'un décaleur. La décision est donc prise de limiter le parallélisme matériel de l'unité CU à **2 MACs, 1 ALU et 1 décaleur/BMU**. Les chiffres présentés Figure 118 tiennent compte de cette limitation.

5.3.2.2 Comparaison avec le TMS320C62x

Les résultats de l'implémentation de 7 noyaux de calcul critiques de l'EFR sur le TMS320C62x sont illustrés Figure 119 [77]. La méthodologie de programmation utilisée est la même que celle que nous utilisons pour coder les fonctions critiques : réécriture éventuelle du source C pour exposer plus de parallélisme, et utilisation des techniques de pipeline logiciel et de déroulement de boucles. Les chiffres obtenus pour le TMS320C62x correspondent à un degré d'optimisation de performance maximal : utilisation de toutes les ressources du processeur et de toute la largeur du jeu d'instructions VLIW. Pour ne pas fausser la comparaison, le code source C restructuré à partir duquel sont programmés les noyaux critiques est identique pour les deux processeurs. C'est par exemple le cas pour la fonction « autocorr » qui est implémentée dans sa forme restructurée de la Figure 116. Par contre, des facteurs de déroulements de boucles différents peuvent être appliqués pour s'adapter aux caractéristiques de chaque processeur. Il en est de même pour l'utilisation du pipeline logiciel.

	ASIP	***TMS320C62x***
Search_10i40 Boucle 1	45	45
Search_10i40 Boucles 2 et 3	288	400
Cor_h	2130	2533
Lag_max	N*45	N*43
Produit scalaire	N/2+1	N/2+1
Residu	246	231
Autocorr	517	519

Figure 119 Performance des boucles critiques pour la recherche du couple {i8,i9}

Sur ces 7 fonctions, cinq d'entre elles présentent des résultats quasi-identiques sur les deux processeurs. Le degré d'accélération est ici limité par l'architecture Bi-MAC, les algorithmes étant principalement formés de boucles de multiplication-accumulation de type produit scalaire. Pour ce type d'algorithme et pour une contrainte de deux opérations MAC par cycle, les performances

obtenus par les deux processeurs sont quasi-optimales. Les techniques de programmation employées ne sont pourtant pas identiques, le TMS320C62 utilisant presque systématiquement du pipeline logiciel pour contourner les pénalités liées à son architecture et à la longueur de son pipeline d'exécution (cf. 5.2). L'ASIP n'utilise quant à lui le pipeline logiciel que pour palier son manque en unités fonctionnelles, ce qui lui permet d'atteindre des performances très proches de celles du TMS320C62 avec 1 ALU et un décaleur de moins. L'écart le plus grand (fonction « résidu ») est du à l'incapacité de l'ASIP d'effectuer deux opérations de décalage dans la même instruction, ce qui est crucial dans la boucle critique de cette fonction. La performance sur l'ASIP de la fonction « autocorr » est par contre identique à celle du TMS320C62, ceci grâce à l'emploi du pipeline logiciel qui permet de décaler l'exécution de deux occurrences successives de la boucle critique, et de pouvoir ainsi effectuer les deux opérations d'arrondis (opérateur « round ») requises avec une seule ALU.

Pour les fonctions « search_10i40 loop 2 et 3 » et « cor_h », l'ASIP donne même de meilleurs résultats que le TMS320C62. Pour les deux fonctions, l'explication provient de la longueur des boucles « interne » et « externe », qui sont plus courtes dans le code de l'ASIP, respectivement de 1 et 3 instructions pour les boucles de « cor_h », et de 2 et 1 instructions pour « search_10i40 ». Pour la boucle interne de « search_10i40 », le gain de 2 cycles est uniquement du à l'ajout dans le jeu d'instruction de l'instruction spéciale « l_msu_with_test ». Dans le cas de la boucle interne de « cor_h », c'est l'architecture « load/store » du TMS320C62 qui est responsable de la longueur supplémentaire en fractionnant l'accès et le traitement des données en deux instructions séquentielles, ce qui limite l'efficacité du pipeline logiciel. La plus grande longueur des boucles externes du code TMS320C62 est quant à elle due aux instructions d'initialisation et de complétion nécessaires à l'usage du pipeline logiciel dans les boucles internes.

Comme on pouvait le prévoir, l'usage intensif du pipeline logiciel, l'architecture « Load/Store » et l'absence de certaines instructions complexes dans le jeu d'instructions (auto-incrémentation des index, boucles « zéro-cycles) ont un effet désastreux sur la taille du code du TMS320C62 (Figure 120), qui encode jusqu'à trois fois plus d'opérations élémentaires que pour l'ASIP. Cet excédant est du pour une part aux opérations nécessaires pour émuler les fonctionnalités «boucles zéro-cycle », « adressage auto-incrémenté », etc. Mais l'essentiel des opérations supplémentaires est du au pipeline logiciel : la boucle 1 de la fonction « search_10i40 » nécessite à elle seule 26 opérations d'initialisation avant de démarrer le pipeline. Le TMS320C62 paye le prix de la complexité de son architecture et en particulier la longueur de son pipeline d'exécution, qui lui permet de fonctionner à des fréquences très élevées et d'offrir des performances très supérieures à la moyenne de celles des processeurs DSP (cf. Figure 20 p41) ; il en résulte en contrepartie une compacité de code très moyenne qui en fait un processeur réservé à des applications de type serveur où la performance prime sur les contraintes de coût et de consommation.

	ASIP	*TMS320C62*
Search_10i40 Boucle 1	55	92
Search_10i40 Boucles 2 et 3	98	133
Cor_h	33	74
Lag_max	21	64
Residu	49	73
Autocorr	74	86

Figure 120 Nombre d'opérations encodées

5.3.2.3 Conclusion

La comparaison avec le TMS320C62 montre clairement qu'il n'existe pas de différence de performance flagrante entre le processeur ASIP et un processeur DSP général pour l'implémentation de l'EFR, excepté pour deux fonctions particulières, dont l'une tire parti d'une instruction spécialisée et l'autre de l'architecture « Load/Store » de l'ASIP. Cette constatation se trouve confirmée au vu des estimations de performance de l'EFR sur d'autres processeurs DSP. Si on applique un facteur de réduction moyen de 2.1 (inférieur à la moyenne des fonctions critiques qui est de 2.26) au reste du code, la performance de l'ASIP pour l'EFR est de 7.3 MIPS, ce qui est très proche des résultats du StarCore et du CARMEL, des processeurs DSP généraux dont l'architecture est similaire à celle de notre modèle d'ASIP (Figure 121).

Ces chiffres sont cependant à prendre avec circonspection, tout d'abord parce que les méthodologies de programmation diffèrent selon les processeurs : l'EFR sur *Starcore* est composé à 90% de code assemblé manuellement et de 10% de code compilé, tandis que le code du *CARMEL* est entièrement codé à la main. Les résultats obtenus dépendant aussi des séquences d'échantillons de test utilisés, il existe une différence significative entre le pire cas pour une séquence particulière et une moyenne sur un grand nombre de séquences. Enfin, il faut tenir compte du fait que ces chiffres sont avant tout utilisés comme arguments commerciaux pour démontrer la puissance d'un processeur pour un domaine d'application stratégique (la téléphonie mobile), et ne sont pas d'une objectivité et d'une rigueur scientifique sans failles. On peut par exemple s'étonner du fait que le *StarCore* avec ses 4 unités MACs et ses 4 décaleurs soit plus lent que le *CARMEL* qui intègre deux fois moins d'unités.

Au final, les performances des différents processeurs (y compris notre ASIP) sont très proches et semblent indiquer une limite de performance aux alentours de 6.5/7MIPS du aux limites du parallélisme intrinsèque de l'application, confirmée par la « faible » différence de performance entre le StarCore (4 MACs) et les processeurs DSP Bi-MACs.

Processeur	*EFR encodeur + décodeur (MIPS)*
ASIP	~7.3
Infineon CARMEL [78]	7
Motorola STARCORE [28]	7.5
Philips REAL [56]	11.5

Figure 121 Performance de l'EFR sur différents processeurs

L'instruction « l_msu_with_test » mise à part, il n'y a en fait pour l'instant aucune réelle spécialisation de l'architecture ou du jeu d'instruction de notre ASIP, ce qui explique que ses performances soient les mêmes que celles des DSP généraux. Le codeur EFR est en effet avant tout composé d'algorithmes DSP très classiques dont la structure de type produit scalaire s'adapte parfaitement aux architectures de ces processeurs. Les spécialisations « classiques » communes aux processeurs DSP conventionnels qu'intègrent notre ASIP (boucles « zéro-cycles », unité d'adressage séparé de l'unité de calcul, modes d'adressages post-incrémenté, etc.) contribuent d'ailleurs moins au gain de performance qu'à la réduction de la taille du code en diminuant le nombre d'informations à encoder et en limitant le pipeline logiciel.

L'intérêt principal de la spécialisation pour ce type d'algorithme réside plus dans la réduction du coût matériel du processeur et de la taille du code embarqué. On a montré qu'il était possible d'obtenir une performance équivalente avec deux unités fonctionnelles de moins (1 ALU et un décaleur). De même, on verra plus loin qu'il est possible de réduire de manière significative le coût matériel du chemin de données et la taille du code en limitant le nombre d'opérations à encoder. Ceci s'explique par le fait que les processeurs DSP généraux sont souvent surdimensionnés (par exemple en terme de nombre de registres et d'unités fonctionnelles) afin d'assurer de bonnes performances pour une très grande variété d'applications, tandis qu'un ASIP peut se focaliser uniquement sur ses applications cibles.

5.3.3 Fonctions « Viterbi »

Contrairement à ce que l'on a vu pour l'EFR, la spécialisation peut avoir un impact important sur la performance lorsque l'algorithme visé sort du cadre de l'algorithme DSP classique à base de boucles MAC. C'est par exemple le cas de tous les algorithmes manipulant et testant les données au niveau bit, pour lesquels les processeurs DSP sont par nature inadaptés. Dans ce cas, l'intégration de structures matérielles et logicielles spécialisées peut devenir indispensable pour maintenir le niveau de performance. C'est ce que nous allons illustrer maintenant au travers de l'implémentation de deux fonctions GSM particulières, le décodeur et l'égaliseur de Viterbi (cf. 4.4.1.2122).

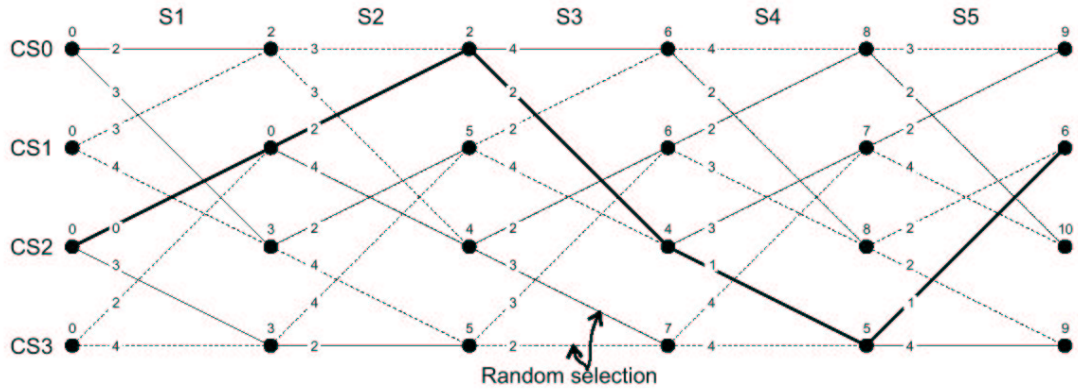


Figure 122 Treillage 4 états (CSx) 5 transitions (Sy)

5.3.3.1 Principe

De manière formelle, l'algorithme de Viterbi permet la recherche de la prochaine transition la plus probable dans un graphe d'état. Dans le cas de l'égaliseur utilisé dans le démodulateur de la partie réception, une transition *Sy* correspond à un symbole reçu par le récepteur, et *CSx* correspond à l'état

numéro x d'un graphe d'état modélisant le canal de transmission (Figure 122). De chaque nœud « état » partent deux arcs correspondant à la valeur « 1 » ou « 0 » du symbole reçu. Le nombre d'états du graphe ***S*** est égal à 2^L, ***L*** étant la longueur du vecteur représentant l'état du canal. A chaque transition *Sx* correspondent plusieurs transitions possibles entre états. Lorsque le récepteur a reçu *N* symboles, l'algorithme de Viterbi calcule les coûts des différents nœuds et arcs du graphe et en déduit le chemin le moins coûteux du graphe, qui correspond à la séquence de *N* symboles la plus probable, permettant ainsi de corriger d'éventuels erreurs de transmission dues à des perturbations sur le canal.

L'algorithme de Viterbi est de nature itérative. A chaque itération correspondant à la réception d'un symbole *Sx*, l'algorithme effectue les tâches suivantes:

- Calcul des coûts associés à chaque branche de l'étape courante, et sélection (1 parmi 2) pour chaque nœud « état » destination de la branche qui le relie et dont le coût est le moins élevé. Dans le cas de la transition S2, le nœud « état » destination CS1 choisit la branche étiquetée 2 au lieu de celle étiquetée 4.
- Mise à jour du coût des nœuds état destination, dont la valeur est la somme du coût de la branche sélectionnée et du coût du nœud source. Pour le nœud CS1 de la transition S2, sa valeur vaut 2+3=5, 2 étant la valeur de la branche sélectionnée et 3 la valeur du nœud source.

Une fois tous les symboles reçus (5 dans l'exemple de la figure), on parcourt le graphe en sens inverse en partant du nœud de coût minimal de la dernière itération (ici 6) et en suivant à chaque étape la branche présélectionnée. On en déduit alors la valeur (« 0 » ou « 1 » suivant la branche) de chaque symbole. Cette dernière étape porte le nom de « Trace Back ».

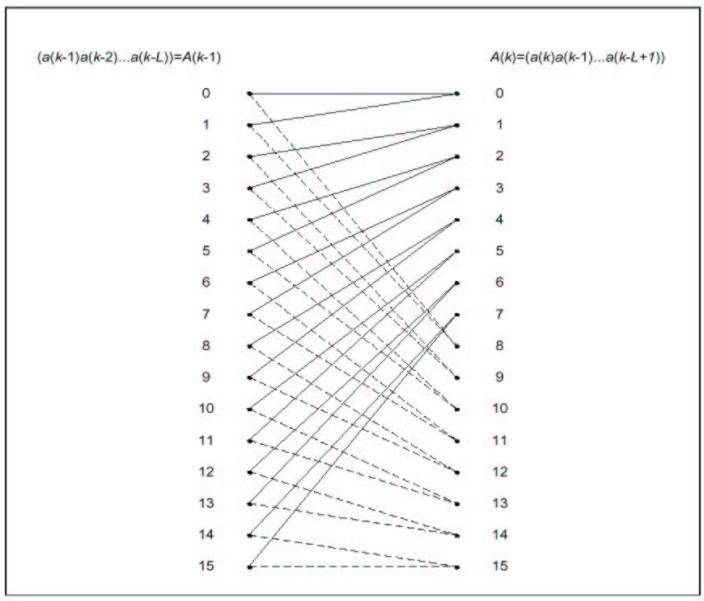


Figure 123 Treillage d'une transition pour un canal S=16 états (L=4)

La méthode la plus communément employée pour mettre en œuvre l'algorithme consiste à traiter le graphe sous forme de « papillons ». Chaque treillage de S états correspondant à une itération est composé d'un ensemble de S/2 papillons de Viterbi (Figure 123). Dans chaque papillon, on calcule les coûts $M_i(k)$ et $M_{i+S/2}(k)$ des nœuds *i* et *i+S/2*. de l'itération k, ce qui nécessite au préalable d'avoir calculé le coût des branches qui dépendent de 2 paramètres supplémentaires, $\alpha_{2i}(k)$ et $\alpha_{2i+1}(k)$ (Figure

124). L'algorithme global de calcul de coût pour le décodage de *K* symboles est décrit Figure 125. La signification des paramètres de calcul $y^R(k)-d_{2i}$ est expliqué en détail en [79]. Les valeurs *â(k-L)* correspondent à des marqueurs qui mémorisent quelle est la branche sélectionnée pour chaque nœud du treillage, et qui seront utilisés lors de l'étape « Trace Back » pour restituer la valeur des symboles.

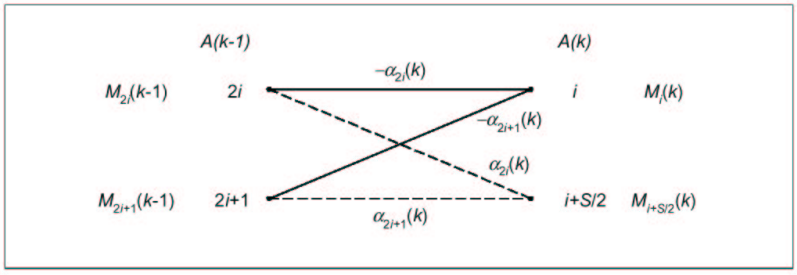


Figure 124 Le $i^{ème}$ papillon de l'égaliseur de Viterbi

Le papillon utilisé dans le décodeur de Viterbi repose sur le même principe, seul le calcul du coût des branches diffère légèrement. L'étape de « Trace Back » est quant à elle commune aux deux algorithmes.

```
for(k=1 ;l<=K ;k++)                    // Un bloc de K symboles
{
        for(i=0 ;i<S/2 ;i++)           // S/2 papillons
        {
                // Paramètres de côut des branches
                α2i(k)=yR(k)-d2i;
                α2i+1(k)=yR(k)-d2i+1;
                // Coûts des noeuds i et i+S/2
                Mi(k)=max{ M2i(k-1) - α2i(k), M2i+1(k-1) - α2i+1(k) };
                Mi+S/2(k)=max{ M2i(k-1) + α2i(k), M2i+1(k-1) + α2i+1(k) };
                // Sélection de la branche valide pour noeuds i et i+S/2
                â(k-L) | A(k)=i =   0 if M2i(k-1) - α2i(k) > M2i+1(k-1) - α2i+1(k);
                                    1 autrement;
                â(k-L) | A(k)=i+S/2 =   0 if M2i(k-1) + α2i(k) > M2i+1(k-1) + α2i+1(k);
                                    1 autrement;
        }
}
```

Figure 125 Egaliseur de Viterbi pour un bloc de K symboles

5.3.3.2 Implémentation

Il y'a encore quelques années, le traitement de l'algorithme de Viterbi dans les circuits destinés aux applications de communication numérique se faisait à l'aide de coprocesseurs matériels spécialisés totalement séparés du processeur DSP central. Les performances des processeurs de cette époque n'étaient en effet pas suffisantes, principalement à cause de l'architecture DSP conventionnelle qui

s'accorde très mal de la nature « non-MAC » de l'algorithme. Avec l'explosion des applications de communication numérique, un certain nombres de processeurs ont pris le parti de la spécialisation et intègrent des modules spécialisés pour le traitement de Viterbi. Parmi eux, on trouve le CARMEL d'*Infineon* [80], le *Motorola* StarCore [81], le DSP16xxx de *Lucent Technologies* [37] ou le *Philips* R.E.A.L [56]. La technique d'implémentation est très similaire pour tous les processeurs, elle consiste en l'emploi de quelques instructions spécialisées ajoutées au jeu d'instructions général, ces instructions manipulant de manière implicite des structures matérielles spécialisées ajoutées au chemin de données (souvent des registres et un peu de logique combinatoire). L'intérêt de cette solution est avant tout de supprimer le besoin d'un coprocesseur spécialisé, ce qui permet au fabricant de processeur de proposer une solution « clé en main » pour les applications de communication numérique.

Nous allons présenter ici les spécialisations matérielles et logicielles apportées à notre modèle d'ASIP pour optimiser le traitement du décodeur et de l'égaliseur de Viterbi. Les instructions et structures matérielles ajoutées ne sont en elles-mêmes pas réellement nouvelles et s'inspirent fortement de celles présentes dans les processeurs cités précédemment. Notre intérêt ici est plutôt de mesurer l'impact de l'intégration de structures spécialisées dans une architecture générale, tant au niveau de la performance qu'au niveau de la complexité matérielle.

Dans l'égaliseur comme dans le décodeur, c'est l'étape de calcul du coût des nœuds du graphe qui occupe la majorité de la charge de calcul, la dernière étape de « Trace Back » étant beaucoup moins complexe. Pour un graphe à S états, on calcule les coûts deux par deux, selon la méthode du papillon illustrée Figure 124, ce qui revient à calculer $S/2$ papillons à chaque itération k.

Pour l'égaliseur, le calcul d'un papillon requiert les opérations suivantes :

- 6 additions/soustractions
- deux opérations « MAX »
- mémorisation de deux bits de transition $\hat{a}(k\text{-}L)$
- 4 lectures: d_{2i}; d_{2i+1}; $M_{2i}(k\text{-}1)$, $M_{2i+1}(k\text{-}1)$ ($y^R(k)$ n'est lu qu'une seule fois pour tous les papillons d'une itération k).
- 2 écritures : $M_i(k)$, $M_{i+S/2}(k)$

L'implémentation du papillon « décodeur » sur le processeur ASIP est présenté Figure 126. Chaque papillon requiert de calculer quatre termes (C(« 0 »), C(« 1 »), D(« 0 »), D(« 1 »)) correspondant aux 2 valeurs possibles du coût pour chacun des deux nœuds C et D. Les coûts de l'itération précédente A et B sont mémorisés dans un tampon (« Old_metrics ») et référencés par un pointeur associé « Old-ptr ». Deux autres tampons sont utilisés pour accéder aux variables d_i (« D_mem ») et écrire les nouvelles valeurs de coûts des nœuds destination (« New_metrics »).

La première instruction calcule les quatre termes du papillon et les mémorise dans deux registres généraux 32 bits *Creg* et *Dreg*. Les deux termes α_{2i} et α_{2i+1} utilisés pour le calcul sont accédés via un registre 32 bits αreg. Comme l'ASIP ne dispose que de 3 unités capables d'effectuer une addition/soustraction en parallèle (2 MACS et 1 ALU), on ajoute au jeu d'instructions deux instructions de type SIMD « l_add_2 » et « l_sub_2 ». Ces instructions sont prises en charge par

l'ALU et effectuent en parallèle deux opérations 16 bits sur des valeurs sources concaténées dans deux registres 32 bits. Grâce à ces instructions, il devient possible de calculer les quatre termes en une seule instruction, l'ALU effectuant deux soustractions tandis que les 2 MACs réalisent les 2 additions restantes. L'instruction « l_add_2 », non utilisée ici, est utilisée de la même façon dans le papillon du décodeur.

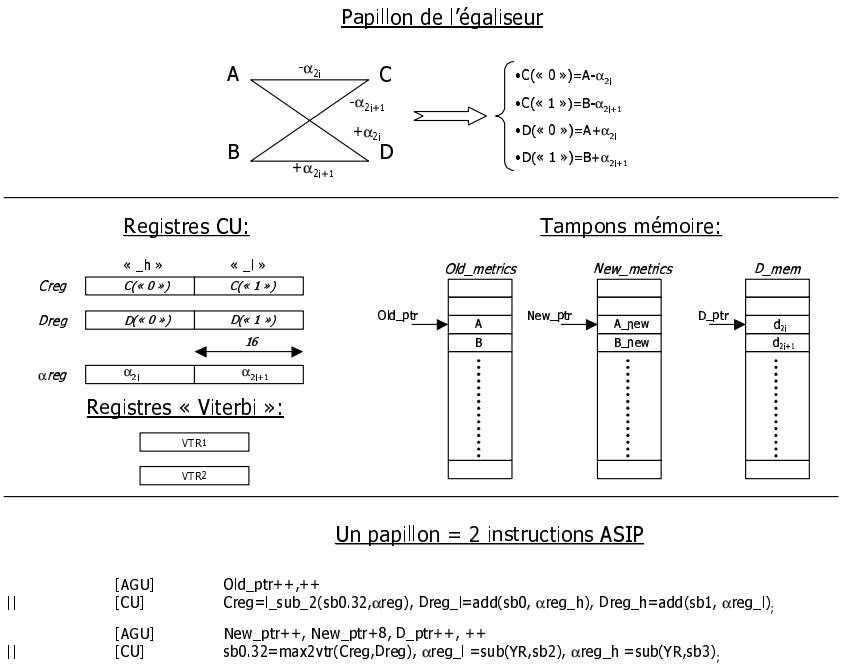


Figure 126 Implémentation du papillon du décodeur Viterbi

Une partie de la deuxième instruction actualise la valeur des paramètres α_{2i} et α_{2i+1} pour le calcul du papillon suivant. La deuxième partie consiste à effectuer une opération de maximum sur les deux valeurs possibles de chaque nœud et à ranger les résultats dans la mémoire « New_metrics ». Sans instruction particulière, le calcul d'un maximum nécessite une soustraction suivie d'un test puis d'une affectation conditionnée par le résultat du test, ce qui correspond à au moins 3 instructions séquentiels. L'instruction spéciale « max2vtr » de type SIMD permet d'effectuer toutes ces opérations en un seul cycle. Les quatre termes de 16 bits sont passés via deux registres 32 bits à l'ALU qui calcule en interne et en parallèle deux opérations de soustraction 16 bits et propage les résultats des deux comparaisons vers la sortie (Figure 127). Cette fonction génère aussi en sortie les deux bits de transition $\hat{a}(k\text{-}L)|_{A(k)=i}$ et $\hat{a}(k\text{-}L)|_{A(k)=i+S/2}$ qui mémorisent les branches sélectionnées pour chaque nœud C et D. Ces bits sont stockés dans deux registres spécialisés VTR1 et VTR2, qui sont automatiquement décalés d'un bit vers la gauche à chaque appel de la fonction.

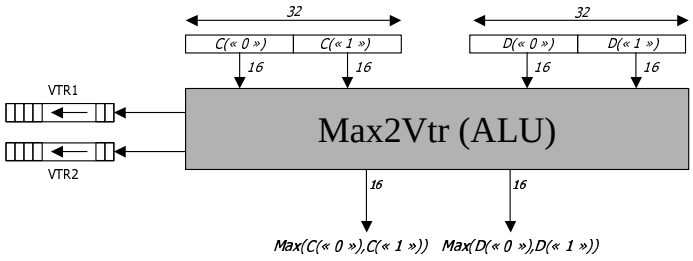


Figure 127 Instruction "Max2Vtr"

A la fin de l'exécution des S/2 papillons, le tampon « New_metrics » contient les S valeurs des nœuds destination, le registre VTR1 contient les bits de transition pour les nœuds *0…S/2-1* et VTR2 ceux des nœuds *S/2…S-1*. Pour démarrer une nouvelle itération, il suffit d'inverser les pointeurs « Old_ptr » et « New_ptr » pour que les nœuds destination de l'itération k deviennent les nœuds source de l'itération k+1. La nouvelle valeur de $y^R(k)$ est lue en mémoire et stockée dans le registre correspondant. Enfin, les deux registres VTR1 et VTR2 sont accédés via une instruction spéciale de l'ALU (« get_vtr() ») qui concatène le contenu des 2 registres et génère une sortie sur 32 bits. La largeur des registres VTR étant de 16 bits, et sachant que chaque bit mémorise une transition pour un nœud du graphe, le nombre d'états S ne peut dépasser 32. Ce nombre S varie selon les applications. Pour le décodeur de Viterbi du GSM, la longueur du canal L est fixé à 4, ce qui correspond à S=16. Dans ce cas, chaque registre VTR ne contient que 8 bits significatifs. Afin de faciliter la dernière étape « Trace Back », les bits significatifs des deux registres doivent être concaténés dans un seul registre de 16 bits. Cette opération peut être faite de manière directe par une instruction spéciale « insert » à inclure dans le jeu d'instruction. Cette instruction permet d'insérer une portion de largeur N bits provenant d'un registre 16 bits op1 à une position particulière P dans le registre destination 16 bits op2 (Figure 128). Dans le cas du décodeur GSM, elle permet en un seul cycle de rassembler les 8 bits significatifs de VTR1 et ceux de VTR2 dans un seul registre destination de 16 bits. Sans cette instruction spéciale, quatre opérations seraient nécessaires (deux masquages, un décalage et un XOR) pour obtenir le résultat, ce qui dégraderait la performance de l'ensemble de l'algorithme d'environ 15%.

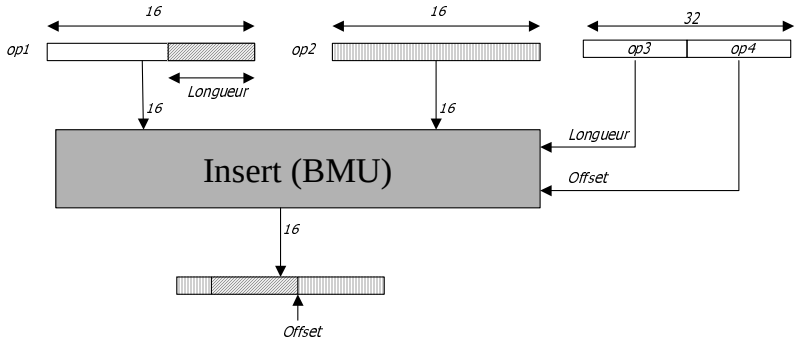


Figure 128 Instruction "insert"

5.3.3.3 Résultats

La Figure 129 présente les résultats d'implémentation du décodeur et de l'égaliseur de Viterbi sur l'ASIP et sur le processeur TMS320C62. Contrairement à l'ASIP, le C62 n'intègre pas de fonction Viterbi spécialisée et n'utilise que ses unités fonctionnelles générales pour effectuer les calculs. Cette

fois-ci, et contrairement au cas de l'algorithme EFR, la différence de performance entre le processeur spécialisé et le processeur général est significative : un rapport de performance de 2 (décodeur) à 3 (égaliseur) en faveur de l'ASIP. La différence plus grande pour l'égaliseur est due à la plus grande complexité de calcul pour le papillon : l'égaliseur réclame 6 additions/soustractions par papillon contre 4 pour le décodeur. Grâce aux instructions spécialisées, l'ASIP ne requiert que 2 cycles par papillon, contre 3 (décodeur) ou 4 (égaliseur) pour le TMS320C62.

Processeur	*Viterbi Decoder*	*Viterbi Equalizer*
ASIP	N*19	N*19
C62	N*38	N*43

Figure 129 Performance des algorithmes de Viterbi pour un bloc de longueur N

Sachant que l'égaliseur de Viterbi occupe environ 5 MIPS sur le TMS320C62, la spécialisation du processeur permet de réduire la charge de calcul d'environ 2.8 MIPS, ce qui est loin d'être négligeable puisque cela représente plus d'un tiers de la charge de calcul de l'EFR complet. Du coup, il devient possible soit de diminuer la fréquence de fonctionnement afin de réduire la consommation du processeur, soit d'intégrer au code embarqué des fonctionnalités supplémentaires synonymes de valeur ajoutée pour le produit final.

Sous réserves que le coût matériel supplémentaire lié aux instructions spécialisées soit raisonnable, ce que nous vérifierons plus loin (cf. 5.6.2), la spécialisation pour l'algorithme de Viterbi semble avoir énormément d'intérêt pour n'importe quel processeur visant des applications de communication numérique. De fait, la totalité des processeurs DSP récents (y compris le successeur du TMS320C62, le TMS320C64) visant de près ou de loin ce type d'applications intègrent maintenant tous des instructions spécialisées Viterbi.

5.3.4 Conclusion

Sur l'ensemble des fonctions critiques comparées (les 6 noyaux critiques EFR + égaliseur et décodeur de Viterbi), le gain en performance moyen est de **37%** en faveur de l'ASIP par rapport au TMS320C62. Comme le reste du code de l'EFR est principalement de type contrôle, on suppose que la charge de calcul requise par cette portion de code est la même sur les deux processeur. Si on tient compte d'une charge supplémentaire incompressible de 3 MIPS due au programme principal et à diverses fonctions d'initialisation et de contrôle (cf. Figure 89), le gain en performance sur l'ensemble de la chaîne de traitement GSM est de 24%. Comme on le voit sur la Figure 130, la réduction de la charge de calcul provient essentiellement de l'accélération des algorithmes de Viterbi : 55% d'écart entre les deux processeurs, contre seulement 21% pour l'EFR.

Alors que le code source C de l'EFR est composé de 52 fonctions représentant plusieurs centaines de lignes de code, les deux fonctions de Viterbi totalise à peine une vingtaine de lignes, et c'est pourtant sur cette portion de code que se joue l'écart de performance et sur lequel s'appuie la spécialisation. Ces résultats vérifient une fois de plus la règle « 80/20 » (80% du temps de calcul dans 20% du code) caractérisant les applications DSP.

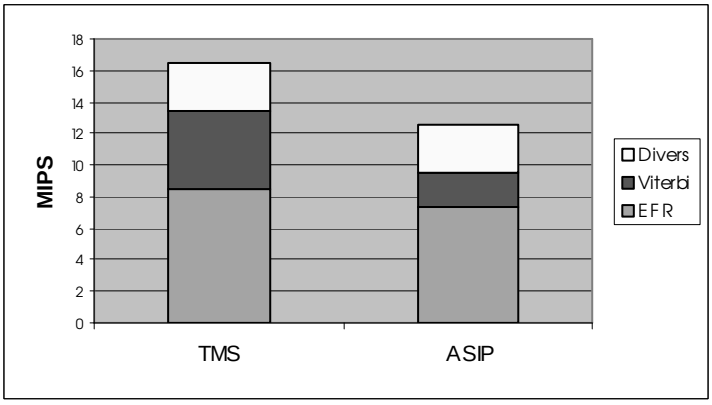


Figure 130 Comparaison de performance sur l'ensemble de la chaîne de traitement GSM

Cette première phase d'analyse et de codage des fonctions critiques nous a permis d'identifier les spécialisations matérielles et logicielles nécessaires pour optimiser la performance du processeur sur l'application visée. A ce stade, nous avons une estimation très précise de la performance finale de l'ASIP dans sa version non paramétrée, c'est à dire sans aucune contrainte en terme de ressources matérielles et de format d'encodage. La prochaine étape consiste maintenant à spécifier les différents paramètres de l'architecture et du jeu d'instructions afin d'obtenir une version finale « réaliste » de l'ASIP en accord avec les autres contraintes du cahier des charges : surface de silicium et/ou la consommation électrique.

5.4 Spécification des paramètres du modèle d'ASIP

L'amélioration de la performance, liée avant tout à l'exploitation d'un plus grand parallélisme, n'est possible qu'au prix d'une augmentation du nombre de certaines ressources du processeur. Traiter en parallèle deux échantillons réclame par exemple deux fois plus de registres pour mémoriser les variables intermédiaires, et une bande passante mémoire supérieure. De même, le nombre de transferts par cycle entre les différents éléments du chemin de données augmente, nécessitant une connectivité plus grande et un plus grand nombre de ports d'entrée/sortie pour les bancs de registres.

Même si les premiers résultats sur le codage des fonctions critiques indiquent que l'ASIP peut fonctionner plus vite tout en ayant un moins grand nombre d'unités fonctionnelles, le coût matériel économisé à un endroit risque de s'être déplacé ailleurs dans le processeur. Comme les fonctions critiques de l'étape précédente ont été simulées sur un modèle non contraint (sauf pour le nombre d'unités fonctionnelles), il est maintenant nécessaire d'analyser précisément leur besoins pour les autres ressources paramétrables du processeur : nombre de registres, connectivité, bande passante, parallélisme d'instruction, etc. Les chiffres présentés par la suite ont été fournis par le simulateur, et dans certains cas par l'analyse « manuelle » du code assembleur. Les prochaines sections présentent le paramétrage des différentes unités (CU, AGU, PCU) du processeur ainsi que de l'architecture globale du modèle comprenant les bus d'échanges et le système mémoire.

5.4.1 Unité CU

La seule information issue du codage des fonctions critiques qui concerne l'architecture du chemin de données est le besoin en nombre d'unités fonctionnelles : 2 MACs, 1 ALU, et une unité Décaleur/BMU (cf. 5.3.2.1). Cette première donnée est issue de l'analyse des fonctions critiques du

point de vue du nombre et du type d'opérateurs utilisés à chaque instruction. Cette configuration suggérée du chemin de données repose sur l'hypothèse d'une répartition logique des opérateurs en fonction de leur nature dans quatre types d'unités fonctionnelles : MAC, ALU, Décaleur et BMU. C'est cette répartition que nous allons maintenant finaliser.

5.4.1.1 Allocation des opérateurs et spécification des unités fonctionnelles

L'étape d'analyse de complexité du code source de l'application a déjà permis d'identifier les opérateurs arithmétiques de l'EFR indispensables qui devront être intégrés au jeu d'instruction (cf. 4.4.1.5). Ces opérateurs, auxquels on doit aussi ajouter les instructions spécialisées nécessaires à l'implémentation des algorithmes de Viterbi, doivent maintenant être répartis dans les différentes unités fonctionnelles du chemin de données. Nous supposons un modèle comportant quatre types d'UFs : MAC, ALU, décaleur et BMU, cette dernière implémentant les opérations de manipulation de bits comme l'instruction « insert » de l'algorithme de Viterbi. Deux critères principaux guident cette répartition. On doit tout d'abord tenir compte de la structure matérielle interne des différentes unités fonctionnelles: on choisira pour intégrer un opérateur particulier l'unité fonctionnelle dont l'architecture matérielle s'accorde le plus avec le type d'opération à réaliser. Selon ce principe, tous les opérateurs basés sur une opération de multiplication sont évidemment alloués à une unité de type MAC. De même, tous les opérateurs basés sur des opérations de décalage sont intégrés à l'unité Décaleur. Les opérateurs de normalisation « norm_l » et « norm_s » consistant à détecter la position du bit le plus significatif dans un nombre fractionnaire sont de type manipulation de bit et sont donc intégrés à l'unité BMU, de même que l'instruction « insert ». L'ensemble des autres opérations (arithmétiques, logiques et Viterbi) est pour l'instant supposé être pris en charge par l'ALU.

MAC	***ALU***	***Shifter/BMU***
mult	test / test2	shl
l_mult	abs	shr
l_mac	round	l_shl
l_msu	l_add	l_shr
l_msu_with_test	l_sub	norm_s
add	l_abs	norm_l
sub	logique (XOR,NOT,AND,OR)	insert
	add	
	sub	
	clear / clear2	
	l_add_2	
	l_sub_2	
	max2vtr	
	get_vtr	

Figure 131 Répartition des opérateurs sur les différents types d'UF.

Le deuxième critère est lié au parallélisme requis par les opérateurs entre eux : si 2 opérateurs particuliers sont utilisés régulièrement en parallèle dans la même instruction, ils doivent pouvoir être

exécutés parallèlement sur deux unités fonctionnelles, soit de natures différentes, soit de même nature et donc dédoublées.

L'analyse du parallélisme dans les fonctions critiques nous fournit les renseignements suivants :

- Les opérateurs de décalage et de manipulation de bits ne sont jamais sollicités dans la même instruction, ce qui est compréhensible : en général, l'opérateur de normalisation (BMU) est appelé en premier pour obtenir la valeur de l'exposant, puis le nombre est décalé de cette même valeur. Comme suggéré en 5.3.2.1, on a donc intérêt à fusionner les deux types d'unités en un seul type « Décaleur/BMU ». Cela permet de réduire le nombre d'unités fonctionnelles et par voie de conséquence de diminuer le nombre de ports de lecture et d'écriture du banc de registres de l'unité CU et la connectivité globale du chemin de données.
- De nombreuses instructions nécessitent l'exécution simultanée de deux opérateurs d'addition/soustraction sur 16 bits « add » ou « sub ». Matériellement, ces opérations peuvent être à priori exécutées indifféremment sur une structure de type ALU ou de type MAC. L'analyse rapporte aussi que les autres opérateurs de l'ALU (logiques, arrondi, test, Viterbi) ne sont jamais sollicités plus d'une fois par instruction et ne nécessitent donc pas le doublement de l'ALU. En conséquence, ce sont les unités MAC qui doivent prendre en charge ces opérateurs, qui sont donc ajoutés à la table des mnémoniques associée aux MACs.
- Le nombre maximum d'appels aux opérateurs par instruction est égal à 3. En général, ce sont de instructions de type « MAC MAC ALU » ou « MAC MAC BMU ». Une seule instruction parmi les noyaux critiques sollicite les unités ALU et BMU dans la même instruction. Cette instruction située dans le noyau critique de « *convolve* » peut être facilement supprimée en réécrivant une partie du noyau, et ce sans aucune perte de performance. Du coup, il devient possible de fusionner les unités ALU et BMU pour réduire encore le nombre d'unités fonctionnelles. L'unité CU ne comporte plus maintenant que trois unités : 2 unités MACs et une unité « Décaleur/BMU/ALU ».
- Les opérations spécialisées « Viterbi » (« max2vtr », « get_vtr ») et les instructions de type SIMD (« l_add_2 » et « l_sub_2 ») ne requièrent aucun parallélisme d'exécution. De par la nature de ces opérations, on choisit de les intégrer aux opérateurs associés à l'ALU.

La répartition des opérateurs sur les différents types d'UF est résumée Figure 131. Par souci de clarté, les opérateurs de l'ALU et du Décaleur/BMU sont présentées séparément. La fusion des unités BMU et Décaleur permet de réduire le coût matériel global des UFs en économisant une structure de décalage. En effet, l'instruction « insert » du BMU ne peut se faire sans une opération de décalage, et la fusion des unités permet de n'en utiliser qu'une seule. La fusion avec l'ALU n'apporte par contre aucune économie supplémentaire, les structures matérielles mises en œuvre pour réaliser les opérations de l'ALU étant très différentes de celles du Décaleur/BMU. L'intérêt est ailleurs, dans la réduction du nombre de ports du banc de registre, de la connectivité du chemin de données et de la largeur du mot d'instruction.

La Figure 132 présente l'architecture globale de l'unité ALU/BMU/Décaleur (sans les signaux de commande). La partie ALU s'articule autour d'un bloc combinatoire de type additionneur/soustracteur réalisant à la fois les opérations arithmétiques (A+/-B) et logiques

(XOR,NOT,AND,OR) élémentaires. L'architecture choisie est celle de l'additionneur à retenue anticipée, légèrement modifiée afin de prendre en compte les opérations logiques, la gestion de l'arrondi et des opérations SIMD. Ce choix se justifie pour des raisons de performance, l'architecture à retenue anticipée étant reconnu comme étant la plus rapide. Cette unité contient également les registres spéciaux de Viterbi VTR1 et VTR2, ainsi que deux multiplexeurs spéciaux commandés par les signaux « Max_LSB » et « Max_MSB », qui sélectionnent le maximum pour les deux couples d'opérandes 16 bits de « Max2vtr ». Enfin, une sortie vers le bit T permet de propager le résultat d'une instruction de test, implémentée par une soustraction des deux opérandes suivi du calcul du code condition qui précise le type de test (égal, supérieur, inférieur, etc.).

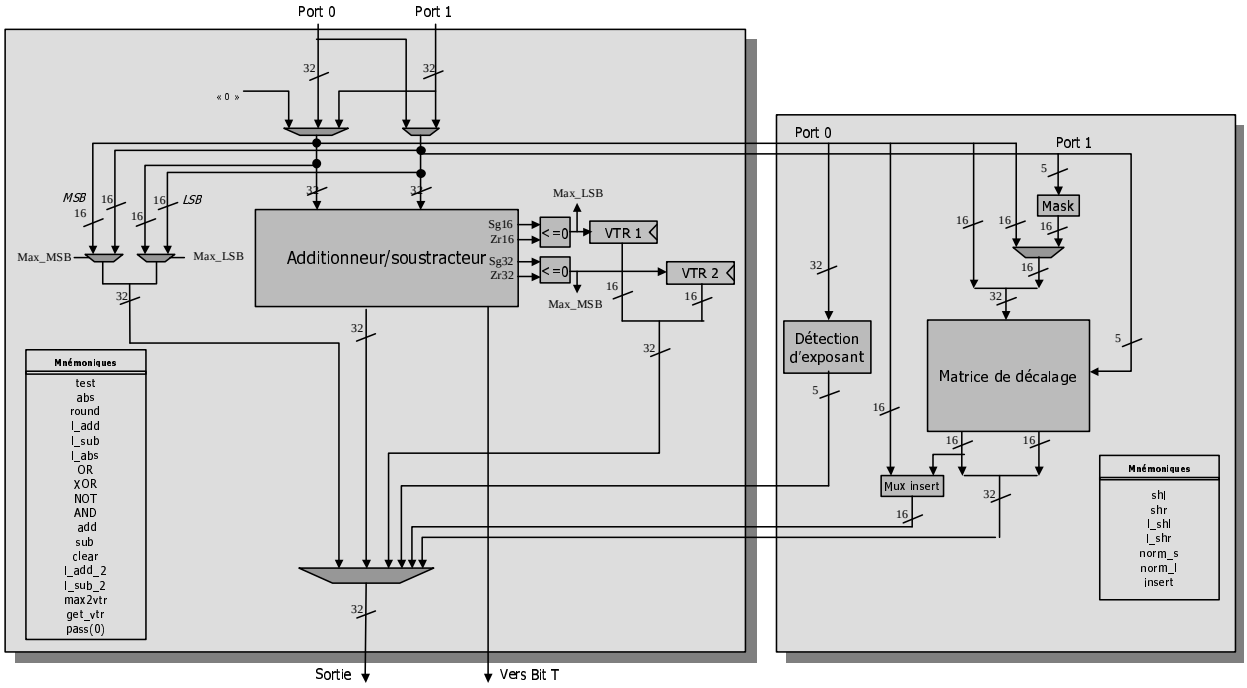


Figure 132 Unité ALU/BMU/Décaleur

La partie BMU/Décaleur est formée d'une matrice de décalage 32 bits vers 32 bits, commandé par un signal de commande de décalage sur 5 bits, qui implémente l'ensemble des opérateurs de décalage. Les opérations de détection d'exposant (opérateurs « norm_s » et « norm_l ») sont effectuées par un bloc spécialisé indépendant fournissant en sortie la position (sur 5 bits) du bit le plus significatif de l'opérande d'entrée. L'opération « insert » réclame un multiplexeur et un bloc combinatoire supplémentaire « mask » qui permet de générer le masque de bits nécessaire à l'opération d'insertion.

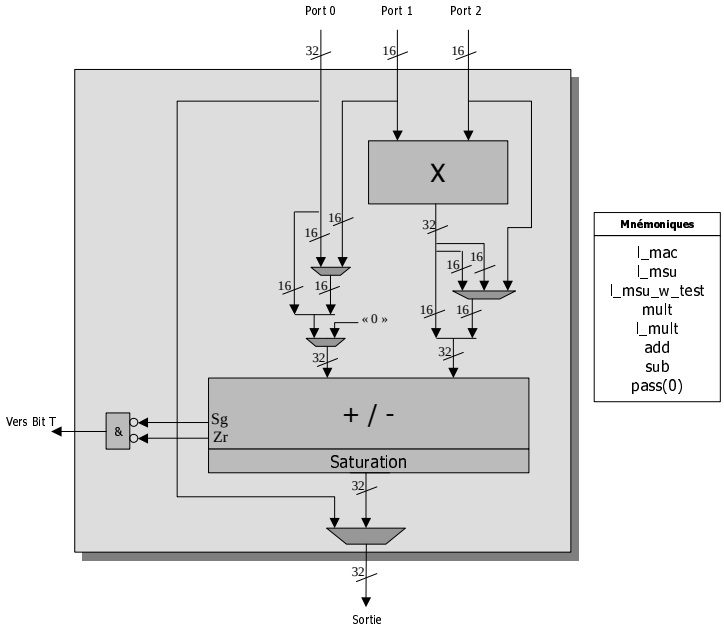


Figure 133 Unité MAC

L'architecture de l'unité MAC est présentée Figure 133. Elle se compose d'un multiplieur 16*16 bits et d'un additionneur 32 bits, avec saturation automatique du résultat en cas de débordement arithmétique. Le calcul de la condition « >0 » est propagé vers le bit T pour être utilisé dans le cas d'une instruction « l_msu_with_test ».

On notera que les deux unités autorisent la mnémonique « pass(0) », qui permet de rendre la cellule « transparente » pour propager la valeur présente sur le port 0 vers la sortie. Cette fonctionnalité permet de faire passer les données au travers des UFs pour exécuter des instructions de transfert de type « registre-registre » ou « mémoire-registre ».

5.4.1.2 Dimensionnement du banc de registres

La Figure 134 donne les besoins en nombre de registres des fonctions critiques de l'EFR. On constate que ces besoins diffèrent grandement selon les fonctions, de 6 registres pour « convolve » jusqu'à 32 pour « search_10i40 ». La méthodologie de programmation des fonctions critiques employée dans l'étape précédente visait à maximiser la performance et se basait sur un modèle sans contrainte en terme de nombre de registres. Les variables locales des fonctions ont donc été affectées presque systématiquement à des registres physiques du processeurs, ce qui explique que les fonctions complexes comme « search_10i40 » qui manipulent beaucoup de variables locales aient des besoins en registres supérieurs à ceux des petites noyaux de calcul comme « convolve ». De plus, l'utilisation poussée des techniques de déroulement de boucles et de pipeline logiciel contribue elle aussi à l'augmentation du nombre de registres nécessaires : la parallèlisation des calculs requiert plus de registres pour mémoriser les résultats. Ceci explique pourquoi la fonction « search_10i40 », qui possède le plus de variables locales et qui utilise massivement les techniques de parallèlisation, a un besoin en registres très supérieur à celui des autres fonctions.

	Nbre Registres CU 16bits	*Nbre registres AGU*
az_lsp	8	7
vq_subvec	9	5
search_10i40	32	17
vq_subvec_s	10	5
lag_max	9	2
cor_h	5	9
norm_corr	9	5
syn_filt	19	5
residu	18	3
autocorr	12	4
convolve	6	3

Figure 134 Besoins en registres des fonctions critiques de l'EFR.

Le nombre de registres de l'unité CU influe évidemment sur la complexité matérielle du processeur, le banc de registre représentant généralement une part non négligeable du coût du chemin de données des processeurs DSP. Mais il a aussi une influence cruciale sur la taille du code généré au travers du nombre de bits nécessaires pour coder les opérandes des instructions. A ce niveau, le concepteur se trouve face à un compromis. Il peut tout d'abord privilégier la performance maximale au dépend du coût matériel et conserver le nombre de registres utilisé par la fonction la plus exigeante. Dans le cas de l'EFR, cela signifie utiliser 13 registres supplémentaires pour une seule fonction, puisque la deuxième fonction la plus gourmande en registres (« syn_filt ») n'en requiert que 19 contre 32 pour « search_10i40 ».

Cette solution n'est évidemment pas satisfaisante, et il vaut mieux tenter de céder un peu de performance afin de diminuer le coût global du processeur. En analysant la structure le la première version assembleur de « search_10i40 », on constate qu'il est parfaitement possible de passer de 32 à 18 registres en désallouant certaines variables « registres » et en les réaffectant en mémoire sur la pile logicielle. La perte en performance qui en résulte est négligeable : 1.5%. On fixe donc le nombre de registres à 20, afin de s'accommoder des exigences de la fonction « syn_filt » (19) qui devient du coup la plus exigeante. Ce compromis peut être évidemment poussé plus loin, tout dépend alors de la borne inférieure que l'on impose pour la performance de l'application sur le processeur.

Plus important que le nombre de registres en ce qui concerne le coût matériel du chemin de données, le nombre de ports d'accès au banc de registres peut être analysé de la même manière et faire l'objet des mêmes compromis. L'analyse des fonctions critiques indique un besoin initial de 6 ports de lecture et de 6 ports d'écriture par cycle. Comme pour le nombre de registres, les exigences les plus fortes sont liées à l'emploi des techniques de parallèlisation de code qui augmente le nombre d'accès aux registres par instruction.

Sachant que l'unité CU comporte deux unités MACs réclamant chacune trois ports d'entrée et un port de sortie, ainsi qu'une unité ALU/BMU/Décaleur à deux entrées et une sortie, les nombre théoriques d'accès devraient être de 8 ports de lecture et de 3 ports d'écriture. Les chiffres constatés lors de

l'analyse des fonctions révèlent donc un nombre relativement faible de ports de lecture pour un nombre relativement élevé de ports d'écriture. Les 6 ports de lecture sont liés au fait que les unités fonctionnelles utilisent aussi des valeurs immédiates ou provenant de la mémoire (registres SBx) comme opérandes source, et n'accèdent jamais toutes à la fois au maximum d'opérandes registres. Le nombre de ports d'écriture élevé est lié à des instructions de transfert « registre-registre » exécutés en parallèle avec des opérations des UFs, et qui exigent donc des ports d'écritures supplémentaires.

Il n'est pas possible de réduire encore plus le nombre de ports de lecture sans toucher aux instructions « clé » des fonctions critiques, ce qui entraînerait d'une part des pertes en performance non négligeables mais qui rendrait aussi l'architecture plus contrainte et donc plus difficile à gérer par un compilateur. On peut par contre réduire le nombre de ports d'écriture de 6 à 5 sans aucune perte de performance, simplement en concaténant deux instructions de transfert 16 bits en un seul transfert 32 bits.

Le banc de registres de l'unité CU comportera donc au final **10 registres 32 bits** (l'équivalent de 20 registres 16 bits, accessibles en mode 16/32 bits), **6 ports de lecture** (N_C5=6) et **5 d'écriture** (**N_C10=5**). Les sorties du banc de registre pouvant être routées à la fois vers des entrées d'UFs 16 et 32 bits, les ports de lecture seront tous de largeur 32 bits. Il en sera de même pour les ports d'écriture, ce qui est de toutes façons égal du point de vue du coût matériel du banc de registres (cf. 5.6.1.2).

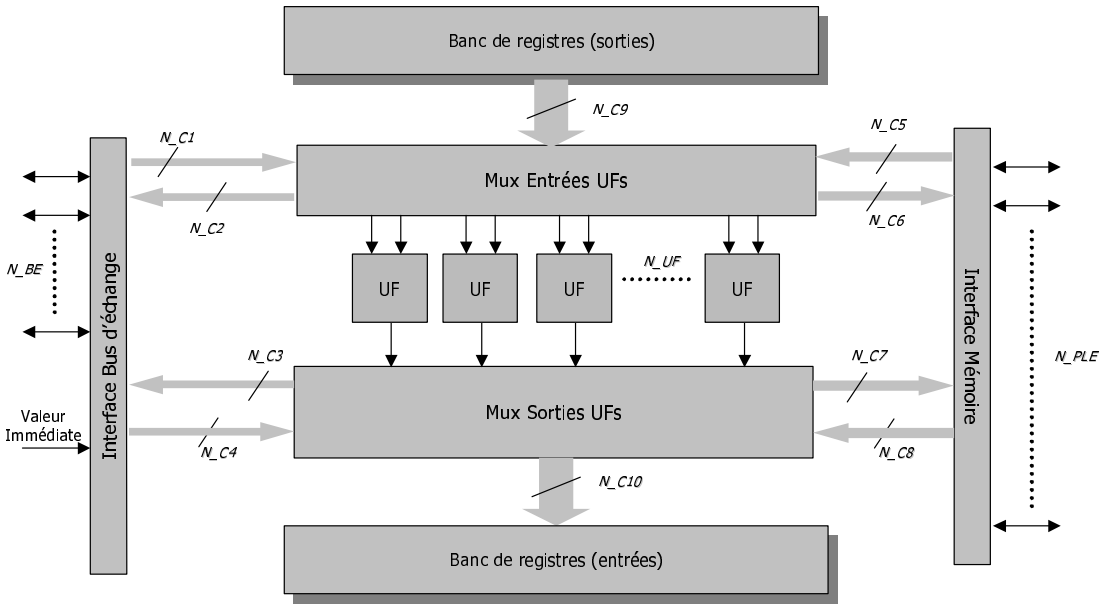


Figure 135 Modèle de l'architecture de l'unité CU

5.4.1.3 Accès aux données externes

Les données manipulées par les UFs de l'unité CU peuvent soit provenir des registres internes du banc de registre de l'unité, soit provenir de l'extérieur : de la mémoire via les registres *SBx*, ou d'une autre unité du processeur (AGU, PCU) via les bus d'échanges ou la valeur immédiate.

Le modèle général du processeur est configurable du point de vue du nombre d'accès parallèles à ces différentes ressources externes. Nous allons maintenant déterminer précisément la bande passante

associée à chaque type d'accès en fonction des besoins des fonctions critiques, ce qui revient à déterminer la valeur des paramètres *N_C1, N_C2, N_C3, N_C4, N_C6, N_C7, N_C8,* et *N_C9* de la Figure 135.

L'analyse des fonctions critiques fournit les données suivantes :

- La bande passante mémoire requise est de 4 données simple précision par cycle en lecture et de 2 données simple précision par cycle en écriture, les accès en double précision se composant de deux accès simple précision. Le chiffre en lecture n'a rien d'étonnant et correspond à la bande passante minimale nécessaire pour alimenter les deux unités MACs. Le besoin en écriture de 2 données 16 bits par cycle est plus dicté par la volonté d'accélérer les sauvegardes et restitutions de contexte lors d'appels de fonction, que par les besoins en calcul des fonctions critiques qui pourraient facilement être ramenés à 1 donnée par cycle sans beaucoup de difficultés. Le doublement de la bande passante en écriture permet de diviser d'un facteur 2 le temps passé dans la gestion du contexte, et aussi de sauvegarder en un seul cycle les données double précision stockée dans les registres 32 bits. Le nombre d'accès (lecture et écriture) par instruction ne dépassant jamais 4, le système mémoire associé au cœur devra donc être capable de délivrer une bande passante de 4 données par cycle, de 4 lectures jusqu'à 2 lectures/2 écritures simultanées. On peut donc fixer les valeurs des paramètres suivants : **N_C5=4** et **N_C6=2**. Les fonctionnalités « transparence » des unités fonctionnelles permettant de véhiculer les données de la mémoire directement vers les ports d'écriture des registres, les connexions « N_C7 » et N_C8 » deviennent inutiles, on aura donc **N_C7=0** et **N_C8=0**.
- Les seules données utilisées en entrée des UFs et provenant de l'extérieur sont des valeurs immédiates, dont le nombre ne dépasse pas une donnée par instruction. Une seule connexion suffit donc entre l'interface « Bus d'échange » et le multiplexeur d'entrée des UFs : **N_C1=1**.
- La connexion « N_C3 » permet d'exporter le résultat d'un calcul vers les bus d'échanges pour être stocké dans un registre d'une autre unité, comme par exemple les registres pointeurs de l'AGU. Ce type d'instruction, bien qu'autorisé dans le jeu d'instructions du modèle par défaut, n'est jamais utilisé dans les fonctions critiques, d'où **N_C3=0**. La connexion « N_C4 » est sollicitée par les instructions de transfert « registre d'une unité externe vers registre de l'unité CU ». Les opérations de ce type les plus utilisées sont l'initialisation d'un registre *Ax* par une valeur immédiate ou le transfert du contenu d'un compteur de boucle *cntr_x* ou d'un registre pointeur *Px*. Aucune instruction ne contenant plus d'une opération de ce type, une seule connexion suffit, d'où **N_C4=1**.
- La connexion « N_C2 » exporte le contenu d'un registre *Ax* vers un registre externe. Une seule opération de ce type par instruction suffit pour assurer les performances des fonctions critiques : **N_C2=1**.

On constate que les connexions concernant les transferts de données entre l'unité CU et les autres unités extérieurs (interface mémoire mise à part) ont soit une largeur nulle ou égale à 1, traduisant le fait que ce type d'instructions sert principalement lors des phases d'initialisation. Les noyaux

critiques des fonctions ne manipulent presque exclusivement que des données stockées en interne dans les registres *Ax* ou dans la mémoire Donnée.

5.4.1.4 Réduction de la connectivité

La dernière phase de configuration du chemin de données du processeur va consister à définir la connectivité, c'est à dire la structure de routage des données dans les multiplexeurs d'entrée et de sortie des UFs.

Certaines instructions clé des fonctions critiques imposent de pouvoir effectuer jusqu'à 5 écritures « registre » dans la même instruction. Sachant que chaque unité fonctionnelle ne produit qu'un résultat par cycle, il faut prévoir deux voies supplémentaires pour acheminer les données. Pour les instructions produisant 5 résultats, une des données résultat provient toujours du contenu d'un compteur de boucle de l'unité PCU véhiculé par les bus d'échanges, d'où la connexion du port d'écriture 0 du banc de registres avec l'interface « Bus d'échange ». Le dernier résultat (port d'écriture 1 du banc de registres) provient d'opérations de transfert « registre-registre » et doit donc être connecté à un port de sortie du banc de registre. 3 ports de sortie servent déjà pour les fonctions « transparences » des UFs. L'examen des opérations effectuées dans les instructions produisant 5 résultats montre que le port de lecture 1 de la première unité MAC n'est jamais utilisé, c'est donc lui qui sera connecté au port d'écriture 1 au travers d'une UF « fantôme » représentée Figure 136.

5.4.1.4 Réduction du multiplexeur d'entrée

Chaque opérande d'entrée d'une UF peut provenir à priori de trois sources : une valeur immédiate, une donnée lue en mémoire, et une donnée provenant du banc de registre. L'analyse des instructions utilisées permet de réduire le nombre de connexions physiques en identifiant celles qui ne sont jamais utilisées. Par exemple, les ports 0 des deux unités MAC n'ont pas besoin d'être connectés à la valeur immédiate et aux registres *SBx* puisque ceux-ci ne sont jamais employés comme première opérande dans une opération de multiplication-accumulation. De même, la valeur immédiate n'est jamais utilisée comme deuxième opérande d'une unité MAC et ne sera donc connecté qu'aux ports d'entrée 2 des MACs. Cette valeur est par contre disponible pour l'ensemble des unités fonctionnelles (et en particulier les 2 MACs) à chaque cycle, ce qui permet d'exécuter dans la même instruction deux opérations accédant à la même opérande immédiate. Ce genre d'instructions est très courant lorsqu'on utilise la technique du calcul multi-échantillons puisqu'on effectue sur deux données différentes les mêmes opérations, qui peuvent nécessiter l'usage de deux immédiats identiques. C'est par exemple le cas de toutes les instructions composant la boucle 1 de la fonction « search_10i40 » (cf. Figure 149 page 223)

La réduction du nombre de ports de lecture du banc de registres pose un problème au niveau du routage des données. En effet, les 3 UFs du chemin de données cumulent 8 ports d'entrée pour seulement 6 ports de sortie du banc de registres, elles devront donc partager certains ports. L'analyse des instructions exécutées permet d'identifier les ports partagés. Ainsi, le port d'entrée 1 de l'ALU et le port d'entrée 0 de MAC2 partageront le même port de sortie (3), sachant qu'ils n'accèdent jamais à une donnée « registre » dans la même instruction. Le même genre de considération conduit à connecter le port 3 de MAC2 et le port 0 de l'ALU au port de sortie 5. Au final, le multiplexeur

d'entrée se résume aux 5 « petits » multiplexeurs connectés aux ports d'entrée des UFs (cf. Figure 136).

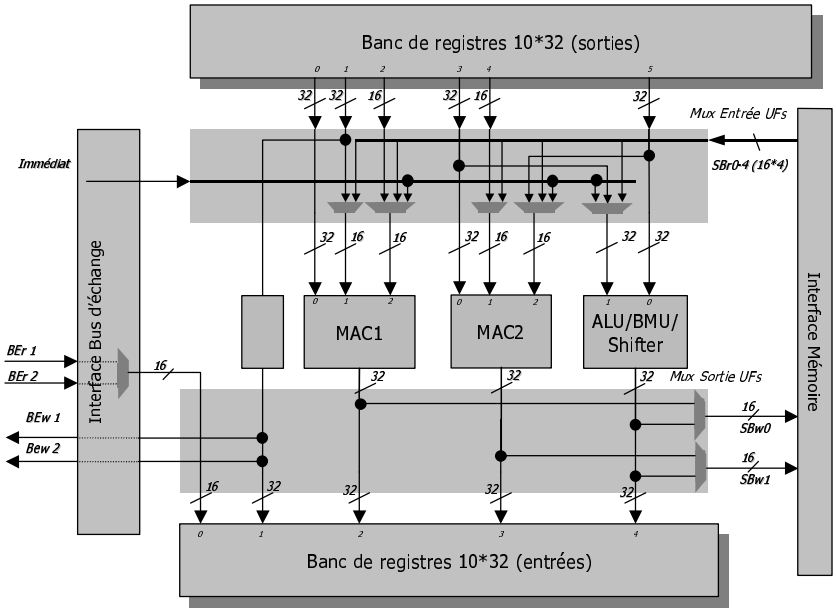


Figure 136 Version finale de l'unité CU (ASIP niveau 3)

5.4.1.4 Réduction du multiplexeur de sortie

Le multiplexeur de sortie est beaucoup moins complexe, en grande partie à cause du fait que le nombre de ports d'écriture dans le banc de registres est égal au nombre de données entrantes, ce qui élimine la nécessité d'un partage des ports. Les seuls multiplexeurs se trouvent au niveau des ports de sortie vers la mémoire. L'analyse des fonctions critiques indique qu'il n'y aucun besoin en écriture mémoire 32 bits de résultats provenant des unités MAC, la seule écriture 32 bits possible passe donc par l'ALU.

5.4.1.4 Impact de la réduction sur le jeu d'instruction

La réduction du nombre de connexions entre les éléments mémorisant (registres) et les unités fonctionnelles a pour effet d'ajouter de nombreuses contraintes au jeu d'instructions. La Figure 137 présente une architecture équivalente à celle de l'ASIP en nombres de registres et d'UFs mais générale au niveau des connexions de données : le nombre de ports de lecture du banc de registres est égal au nombre de ports d'entrée des UFs (9), et les multiplexeurs d'entrée des UFs ainsi que les multiplexeurs de sortie vers l'interface mémoire autorisent toutes les possibilités de routage. Avec ce type d'architecture, il n'existe aucune contrainte de ressources (en terme d'accès aux registres) entre les différentes opérations CU de la même instruction. Cela veut dire qu'une opération de type MAC peut être exécutée indifféremment sur MAC1 ou MAC2, et que chaque opération a la garantie de pouvoir accéder à toutes ses opérandes sources quelque soit leur provenance : registre, immédiat, mémoire.

En réduisant les connexions et le nombre de ports de lecture, on introduit des contraintes inter-opérations. Comme le port 3 du banc de registres est partagé entre l'unité MAC2 et l'ALU, les

instructions de type «[CU] MAC(…,…,...) MAC(reg,…,…) ALU(…,reg) » sont interdites puisque le même port de lecture est sollicité pour deux accès registres dans le même cycle. Une solution consiste à inverser les 2 opérations MAC, mais elle ne fonctionne que si l'autre opération MAC n'accède pas à un registre comme première opérande. On voit donc que la phase d'allocation de ressources pour les différentes opérations, très simple dans le cas de l'architecture générale, devient nettement plus complexe avec la réduction des connexions. L'algorithme d'allocation pourrait être le suivant :

- Allouer les ressources en fonction des mnémoniques codant les opérations CU. Un mnémonique de type MAC sera affecté à l'unité MAC1 et un mnémonique de type ALU/BMU/Décaleur à l'unité ALU/…/….. Une opération de transfert registre-registre sera affectée à l'unité « fantôme ». Une éventuelle deuxième opération MAC utilisera MAC2, ce qui correspond à une allocation « dans l'ordre » des opérations sollicitant la même UF.
- Parcourir les opérandes sources de chaque opération CU et vérifier l'absence de conflit de ressources entre ALU et MAC2 (ports 3 et 5 du banc de registre), et MAC1 et l'unité « fantôme » (port 1).
- Si un conflit est détecté, tenter d'inverser les deux unités MAC, ou d'effectuer le transfert registre-registre par une unité non utilisée plutôt que par l'unité fantôme.
- Si le conflit persiste….casser l'instruction en deux !

Avec cette connectivité réduite, de nombreuses autres instructions sont interdites : utilisation d'un immédiat comme première opérande d'une opération MAC, exportation d'un résultat 16 bits de MAC2 vers le deuxième port mémoire, etc. Bien sûr, si ces connections sont absentes, c'et parce que l'étude des fonctions critiques a montré qu'elles n'étaient pas indispensables dans les noyaux de calcul intensifs des applications visées. Par contre, rien ne permet d'affirmer à ce stade qu'elles ne seraient pas utiles pour l'implémentation du code de contrôle.

Surtout, l'ajout de contraintes structurelles au jeu d'instruction rend beaucoup plus complexes les algorithmes de génération de code utilisés par le compilateur, en particulier la phase de compaction (comprenant l'ordonnancement des instructions et l'allocation de ressources), puisque le nombre de contraintes inter-opérations augmente. Le risque encouru est que le compilateur (ou le processeur si la phase d'allocation est prise en compte au niveau matérielle) ne puisse résoudre de manière satisfaisante les conflits entre opérations élémentaires et qu'il génère un code plus séquentiel et donc moins performant que pour une architecture moins contrainte. Là encore, tout est affaire de compromis entre la performance requise et le coût matériel (surface et consommation) autorisé. Cependant, pour le cas particulier de notre architecture, on montrera plus loin que l'économie matérielle réalisée par la réduction de la connectivité ne semble pas suffisante pour justifier l'ajout de complexité dans le compilateur et la perte de performance du code de contrôle qui en résulte.

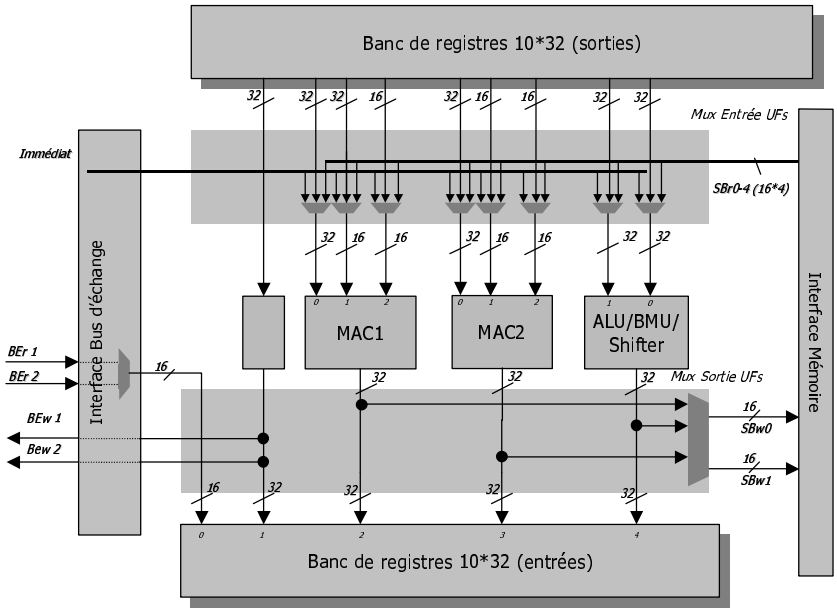


Figure 137 Version « générale » de l'unité CU (ASIP niveau 2)

5.4.2 Unité AGU

Selon les mêmes principes que pour l'unité CU, nous allons maintenant examiner les caractéristiques du code des fonctions critiques du point de vue des calculs d'adresses afin de déterminer l'architecture finale de l'unité AGU. On a vu précédemment que le nombre d'adresses exportées par cycle correspondant à la bande passante *Données* ne dépassait pas 4. L'analyse des opérations AGU des instructions montre qu'il est nécessaire d'inclure 4 unités de calcul d'adresses (UCAs), car certaines instructions comportent jusqu'à 4 opérations de calculs d'adresses simultanées. Contrairement aux opérateurs des UFs du chemin de données, l'ensemble des modes d'adressages définis pour le jeu d'instruction par défaut est utilisé, il n'est donc pas nécessaire de faire une sélection des modes d'adressages pertinents. Afin de permettre l'implémentation efficace d'éventuelles structures de données de type FIFO ou file d'attente (non présents dans les algorithmes étudiés mais généralement très utilisés dans les processeurs DSP), la première UCA intègrera, en plus de l'additionneur/soustracteur nécessaire au calcul des modes d'adressages indexés, la logique et les registres supplémentaires nécessaires à l'implémentation de l'adressage modulo.

La Figure 139 montre que le nombre de registres pointeurs nécessaires varie beaucoup selon les fonctions, et que la fonction *search_10i40* en réclame beaucoup plus que les autres. Il peut alors être tentant de réduire le nombre de registres utilisés dans cette fonction pour diminuer le coût du banc de registres. Malheureusement, il n'est pas possible de réduire le nombre de registres sans dégrader significativement la performance de la fonction. On conserve donc 17 registres pointeurs pour le banc de registres, plus un dernier qui est affecté à la gestion de la pile logicielle (**SP**) : **N_Preg=18**. L'influence d'un plus petit nombre de registres sur la performance et le coût matériel est discuté en 5.6.3.

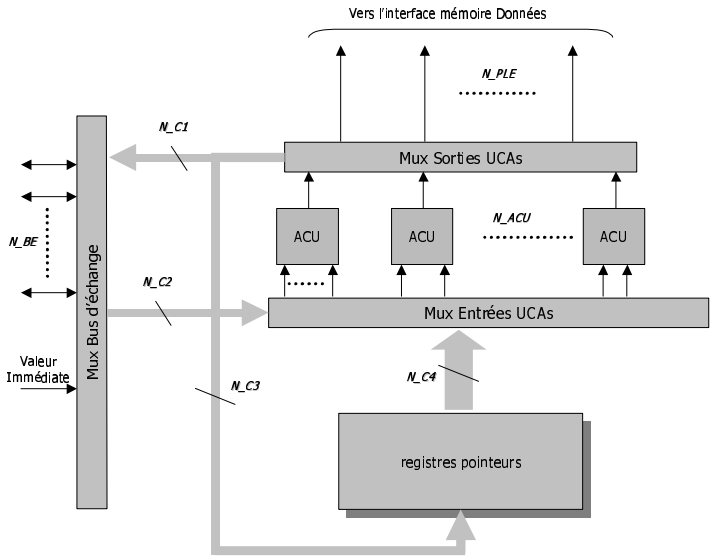


Figure 138 Modèle de l'unité d'adressage

Le nombre de registres pointeurs lus simultanément ne dépassant jamais 4, on fixe **N_C4=4**. Cela s'explique par le fait que la plupart des adressages indexés utilisent un compteur de boucle ou une valeur immédiate comme deuxième opérande au lieu d'un registre pointeur. En modifiant légèrement le code d'une des fonctions critiques, le nombre maximum d'écritures simultanées dans les registres pointeurs peut être facilement ramené à 3 sans perte de performance : **N_C3=3**.

La connexion « N_C2 » permet d'utiliser des valeurs provenant de l'extérieur (immédiats, registres *Ax*, compteurs de boucles) pour effectuer des calculs d'adresses ou des initialisations de pointeurs. Le besoin des fonctions critiques est de deux données au maximum par cycle. Il serait possible de descendre à une seule donnée par cycle, mais les opérations de restitution de contexte lors des appels de fonction (pour les registres pointeurs) prendraient deux fois plus de temps. On conserve donc **N_C2=2**. Pour les mêmes raisons, on fixe aussi **N_C1=2**. L'architecture finale de l'unité AGU est illustrée Figure 139.

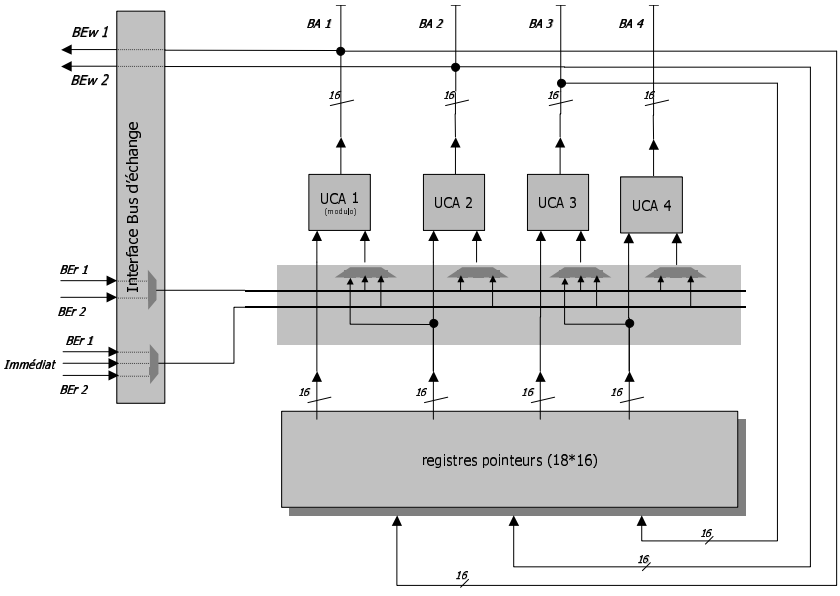


Figure 139 Version finale de l'AGU

5.4.3 Unité PCU

Le décodeur d'instructions mis à part, l'unité de contrôle présente beaucoup moins de degrés de liberté que les autres unités. Le codage des fonctions critiques montre que le nombre de boucles imbriquées ne dépassent jamais 3, on peut donc fixer la profondeur des piles de registres **P_pile_CNTR=3** (cf. schéma de l'unité PCU page 96). Bien que les opérations AGU fassent un usage fréquent des compteurs de boucle comme valeur d'index pour les adressages indirects, un seul compteur exporté par cycle suffit, d'où **N_Cntr=1**.

Le dernier paramètre concerne la profondeur de la pile CP, qui mémorise les adresses de retour lors des appels de fonctions et des interruptions. Il est difficile de déterminer une valeur à la seule étude des fonctions critiques. La sauvegarde matérielle automatique de l'adresse de retour n'a d'intérêt que pour des appels de fonctions de courte longueur. L'algorithme EFR contient quelques fonctions courtes de manipulation de nombre double précision qui pourraient bénéficier de ce mécanisme. Ces fonctions forment les feuilles de l'arbre d'appels de fonction, et ne nécessitent donc pas une profondeur de pile supérieure à 1. Pour les autres fonctions, une sauvegarde classique à l'aide de la pile logicielle suffit amplement.. C'est en fait le nombre maximal d'interruptions imbriquées qui fixera la valeur idéale de la profondeur de la pile matérielle. Ce paramètre dépend de l'architecture du système dans lequel est implanté l'ASIP et ne peut donc être déterminé à ce stade.

5.4.4 Système mémoire et bus d'échanges

L'analyse des fonctions critiques nous permet de fixer les contraintes que devra respecter le système mémoire *Donnée* attaché au processeur. Comme on l'a vu au moment du paramétrage de l'unité CU, la bande passante mémoire requise est de 4 données simple précision par cycle en lecture et de 2 données simple précision par cycle en écriture. Les lectures et écritures doivent pouvoir se faire en parallèle, dans la limite de 4 accès dont 2 maximum en écriture.

En général, on trouve deux types de mémoire dans les DSP : simple et double port. Ces dernières sont utilisées pour stocker des tableaux de données susceptibles d'être accédés deux fois par instruction, comme par exemple le tableau d'échantillons d'entrée d'un filtre FIR sur une architecture Bi-MAC. Notre ASIP étant capable d'émettre quatre requêtes mémoire par cycle, il est théoriquement possible d'adresser jusqu'à 4 fois le même espace mémoire, ce qui nécessiterait l'utilisation de mémoires à quatre ports excessivement coûteuses.

En réalité, l'analyse des accès effectués par les fonctions critiques montre que les tableaux de données sont accédés au maximum deux fois par cycle, la plupart d'entre eux ne l'étant qu'une seule fois. Les doubles accès se trouvent souvent dans les boucles critiques des fonctions, ce qui interdit de modifier le code pour sérialiser les accès et adopter ainsi une solution purement simple port. La première solution envisageable consiste donc à construire le système mémoire à base de mémoires simple et double ports.

Il existe cependant une solution permettant d'émuler le comportement de mémoires doubles ports avec des mémoires simple port. Elle a le principal avantage d'être moins coûteuse à la fois en terme de surface de silicium et de consommation, et offre aussi de meilleurs performances dynamiques (cf. Figure 51 page 71). Le principe consiste à affecter un banc mémoire simple port aux adresses paires du tableau de données, et un autre banc simple port aux dresses impaires. A condition que les accès

aient une parité différente, ce système permet d'accéder à deux données du même tableau dans le même cycle. Ce système est particulièrement utile en traitement du signal où les accès se font très souvent de manière séquentielle, ou par paire de données contiguës dans le cas des architectures Bi-MAC.

Les fonctions critiques de l'EFR sont conformes à ce principe : dans les 12 fonctions étudiées, l'ensemble des accès double ports sont constitués d'adresses paire/impaire. Le système mémoire aura donc tout intérêt à être bâti sur ce principe afin de rendre minimale la quantité de mémoire double ports et réduire le coût matériel du système. L'architecture finale pourrait dans ce cas ressembler à celle de la Figure 140 qui représente un système mémoire à quatre accès par cycle combinant les deux types de mémoire.

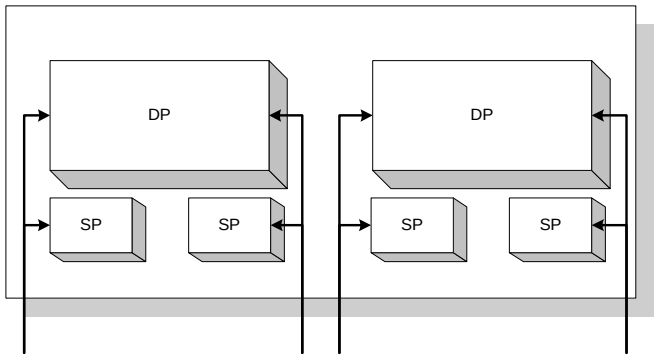


Figure 140 Architecture hybride simple-double port

Il ne reste maintenant plus qu'à déterminer le nombre de bus d'échanges permettant de véhiculer des données simple précision (ou double précision via deux bus) entre les différentes unités. L'examen des besoins en bande passante des différentes unités a montré qu'il fallait 2 bus d'échanges pour permettre d'assurer de bonnes performances. Il faut noter que ces bus ne conditionnent pas vraiment la performance des noyaux critiques (qui ne les utilisent que très peu, mis à part pour accéder au compteur de boucle pour les calculs d'adresse) mais surtout la performance du code de contrôle qui effectue toutes les opérations d'initialisations des différents registres des unités, ainsi que le restitution et sauvegarde de contexte lors des appels de fonctions. 2 bus sont nécessaires surtout pour pouvoir accélérer ce type d'opération : **N_BE=N_L_AGU=N_L_CU=N_L_DMU=N_L_PCU=2**. Un nombre supérieur est de toute façon exclu puisque le système mémoire n'autorise que deux accès simultanés au même espace de données.

5.5 Structure et encodage des instructions

Nous nous intéressons ici à l'impact de la spécialisation du processeur sur la compacité finale du code objet. Dans un premier temps, nous allons définir le format d'encodage des différentes opérations exécutées dans le processeur. A structure d'instruction égale, nous comparons la largeur d'instruction obtenue avec celle du TMS320C62, et proposons une structure d'encodage alternative pour améliorer le taux de compaction du code de l'ASIP.

5.5.1 Encodage des opérations

La structure d'une instruction de l'ASIP est rappelée Figure 141. Une instruction est formée d'opérations élémentaires s'exécutant en parallèle sur une des trois unités fonctionnelles du processeur (CU, PCU, AGU). Pour l'encodage, ces opérations sont réparties en 5 classes et encodées selon un format binaire particulier (Figure 142).

```
        [AGU]       op0,   op1,   …,    opN
//      [CU]        op0,   op1,   …     opM
//      [PCU]       op0,   op1,   …     opL    ;
```

Figure 141 Structure d'une instruction générique

Le format MAC est utilisé pour toutes les opérations utilisant les mnémoniques des unités MAC. Ces mnémoniques sont au nombre de 8 et nécessitent 3 bits pour l'encodage. Les opérandes sources peuvent être de trois types : registres *Ax* de l'unité CU, valeur immédiate ou registre *SBx* contenant une donnée lue en mémoire. Le registre destination peut être soit un registre *Ax* soit un registre *SBx* (ou 2 pour les opérations 32 bits). Les opérations de type multiplication/accumulation utilisent trois opérandes sources et une opérande destination, soit au total quatre opérandes à encoder. Or, l'analyse des instructions utilisées montre que le registre destination est le même que le premier registre source dans la quasi-intégralité des cas : il est donc possible de n'encoder que les trois opérandes sources, l'opérande destination étant dans ce cas implicite et égale à l'opérande source numéro 1.

Le nombre d'opérandes est maximale pour les opérations 16 bits, le nombre de registres adressables étant de 20 (contre 10 pour les opérations 32 bits). Une opérande source peut donc prendre 25 valeurs différentes : 20 registres *Ax*, une valeur immédiate et 4 registres *SBx*. 5 bits sont nécessaires pour coder chaque opérande. La largeur du format MAC est donc de 3 +5 +5 +5= 18 bits. 2 bits supplémentaires sont utilisés pour conditionner l'opération en fonction de la valeur du bit T. Trois conditions sont possibles : exécution si le bit T est à 1 (IFT), à zéro (IFF) , ou fonctionnement normal. On note que l'opération « pass(0) » encodée dans ce format est utilisée pour effectuer un transfert de registre via la transparence de l'unité MAC.

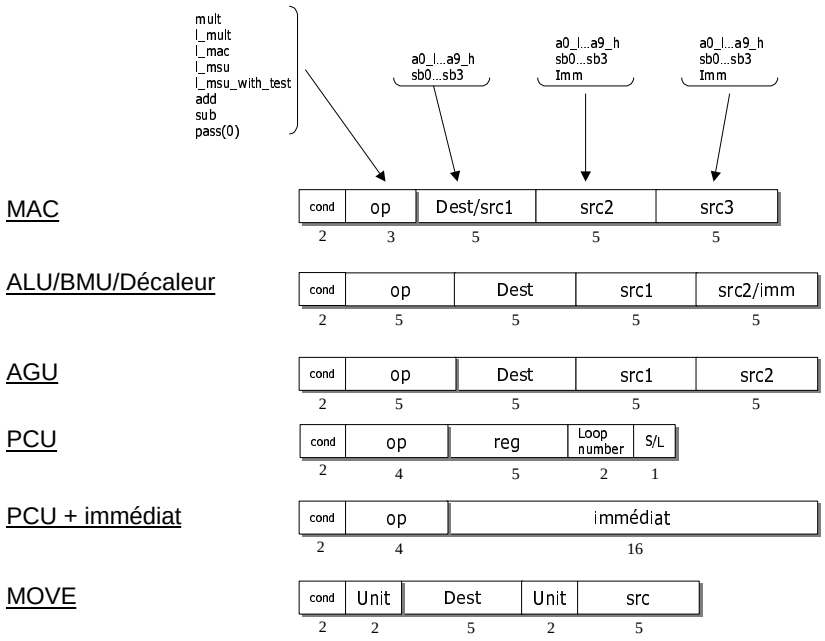


Figure 142 Format d'encodage des opérations élémentaires

Le format ALU/BMU/Décaleur encode toutes les opérations de l'unité correspondante. L'opération « insert » nécessitant 4 opérandes 16 bits, on impose que les opérandes soient passées par l'intermédiaire de deux registres 32 bits, le premier concaténant les deux premières opérandes et le second les deux dernières. Cela permet d'avoir le même nombre d'opérandes sources (2) pour l'ensemble des opérations de ce type. L'encodage des opérandes source et destination suit le même principe que pour le format MAC. Le nombre de mnémoniques à encoder est par contre beaucoup plus élevé : 27, dont 19 pour l'ALU et 8 pour le BMU/Décaleur. On code donc l'opcode sur 5 bits (soit 32 possibilités), ce qui nous laisse 5 opcodes libres. On va les utiliser pour rajouter des opérations de décalage avec opérande immédiate, qui vont nous permettre d'encoder directement l'immédiat sur 5 bits dans le mot d'instruction, à la place de l'opérande *src2*. Comme on le verra plus loin, l'utilisation d'une donnée immédiate nécessite théoriquement de la coder sur un emplacement « opération » du mot d'instruction VLIW. Grâce à ces opcodes supplémentaires, on réduit la taille du code et on permet surtout l'utilisation de deux immédiats dans la même instruction VLIW, ce qui est en temps normal interdit.

Le format AGU encode tous les calculs d'adresses correspondant aux modes d'adressages définis en 3.6.3. A ces 12 opérations de base, on a ajouté trois opérations supplémentaires (« init_pbx », « init_plx » et « set_modulo ») destiné à initialiser les registres spéciaux (adresse de base et longueur) de l'UCA1 pour l'implémentation de l'adressage modulo. On autorise aussi les calculs d'adresses avec une des opérandes sources codant une valeur immédiate sur 5 bits. Cette capacité devrait accroître la densité du code de contrôle en économisant les mots nécessaires au codage des immédiats ; en particulier, tous les adressages relatifs au pointeur de pile, massivement utilisés par les compilateurs, seront codés de manière plus compacte. Pour cela, il est nécessaire de rajouter 6 opcodes supplémentaires, soit un total de 21 opcodes qui seront donc codés sur 5 bits. Les opérandes sources et destination (registres *Px* et compteurs de boucle) nécessitent elles aussi 5 bits de codage.

Les deux formats PCU sont utilisés par les instructions de contrôle de flot (*JUMP*, *CALL*, etc.) et par la fonction d'activation de boucle *Enable_Loop*. Le format MOVE sert à coder tous les transferts de données inter-unités via les bus d'échanges.

La largeur d'encodage des opérations est fixée par le format le plus long , soit ici le format ALU/BMU/Décaleur. Une opération élémentaire sera donc codée sur **22** bits.

5.5.2 Assemblage en une macro-instructions VLIW

La structure d'assemblage des opérations élémentaires en une instruction VLIW a une importance considérable sur la complexité et la souplesse du processeur, ces deux critères évoluant bien évidemment dans le même sens. La méthode la plus triviale pour former un mot d'instruction consiste simplement à abouter les opérations les unes à coté des autres, dans un ordre toujours identique et correspondant à leur unité d'exécution. Si un processeur possède 2 unités MAC, 2 ALUs et un Décaleur, l'opération correspondant à la première « tranche » de l'instruction VLIW sera exécutée sur l'unité MAC1, la deuxième sur l'unité MAC2, la troisième sur la première ALU, etc. Si, pour une instruction donnée, aucune opération MAC n'est sollicitée, les deux tranches correspondantes contiendront l'opération « NOP ». L'avantage de ce type de structure est de rendre très simple l'allocation des ressources puisque la position de l'opération détermine où elle va être exécutée. De même, la complexité du « répartiteur » (l'unité qui redirige les opérations élémentaires du mot d'instruction vers les unités fonctionnelles concernées) est réduite au strict minimum puisqu'elle ne requiert que des connexions directes. L'inconvénient majeur de cette structure est son taux très élevé de NOPs, qui se traduit par une explosion de la taille du code, alors que la tendance en matière de DSP est au contraire à la réduction de la taille du code embarqué. Cette structure n'est donc jamais utilisée et on lui préfère des structures plus souples comme celle du TMS320C62 qui permettent de réduire le taux de NOPs.

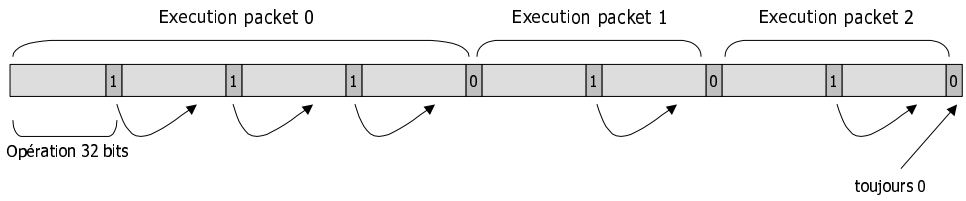


Figure 143 Mot d'instruction du TMS320C62

5.5.2.1 Structure d'encodage de type TMS320C62

Une instruction du C62 a une largeur de 256 bits et est composée de 8 champs de 31 bits codant chacun une opération élémentaire, plus 8 bit de synchronisation (Figure 143). Ces bits déterminent si les différentes opérations composant l'instruction doivent toutes être exécutées en parallèle, ou si elles forment plusieurs paquets d'opérations qui seront exécutées séquentiellement (« execution paquet »). La Figure 143 donne l'exemple d'une instruction formée de 3 paquets d'exécution. Tant que le bit de synchro à la fin d'une opération est à 1, l'opération suivante est considérée comme faisant partie du paquet d'exécution courant. Lorsqu'il est à zéro, l'opération suivante marque le début d'un nouveau paquet.

Pour permettre la comparaison avec le TMS320C62, on va supposer ici que les instructions VLIW de l'ASIP sont construites selon la même structure. Dans ce cas, il faut 1 bit supplémentaire de synchro par opération , plus 4 autres bits pour indiquer l'unité matérielle concernée par l'opération : MAC1, MAC2, ALU/BMU/Décaleur, UCA1, …, UCA4, MOVE et PCU. Parce qu'elle est générique et ne peut donc dépendre de l'implémentation, la syntaxe du jeu d'instructions présenté au chapitre 3 ne permet pas de spécifier explicitement l'unité fonctionnelle sur laquelle l'opération CU doit être exécutée. La phase d'allocation des UFs ne peut donc être explicite et doit être prise en charge soit par les outils de génération de code ou par le programmeur (allocation statique), soit directement par l'unité de contrôle du processeur (allocation dynamique). Comme un ASIP est généralement utilisé pour réduire le coût matériel et la consommation électrique, la méthode statique est la plus appropriée, c'est d'ailleurs celle que l'on retrouve dans l'ensemble des processeurs DSP VLIW. On impose donc que l'allocation des UFs soit faite par les outils de génération de code. On économise ainsi le coût matériel supplémentaire de la logique de contrôle nécessaire à l'implémentation de l'algorithme d'allocation.

Il faut aussi ajouter 2 bits supplémentaires pour coder les deux marqueurs utilisés pour la gestion matérielle des boucles « zéro-cycle ». Une opération complète prendra donc 22+1+4+2=**29** bits. Sachant que le parallélisme maximal constaté dans les fonctions critiques est de **7** opérations par instruction, la largeur finale du mot d'instruction sera donc de **203** bits.

La différence de taille entre une opération du TMS320C62 et une opération de l'ASIP (32 pour 29) est finalement assez faible, ce qui peut faire douter de l'intérêt de la spécialisation d'un processeur pour la réduction de la taille du code. Dans notre cas, il faut tout d'abord noter que toutes les spécialisations visaient un seul but, l'optimisation de la performance, et que tous les paramètres de l'architecture ont été fixés dans ce sens. Une manière simple de réduire la largeur d'encodage aurait par exemple consisté à diminuer le nombre de registre du banc de registres CU ; cela aurait permis d'économiser 3 bits par opération, mais au prix d'une diminution significative de la performance.

Dans la configuration actuelle, il reste beaucoup de « place » dans le jeu d'instructions, puisqu'on pourrait par exemple encoder 12 registres pointeurs, 7 registres *Ax* et plusieurs opcodes supplémentaires sans augmenter la largeur d'une opération, ce qui montre que le taux d'encodage est loin d'être optimal.

On ne doit pas non plus oublier que grâce à son architecture de type « Load/Store », ses mécanismes de boucle « zéro-cycle » et ses modes d'adressages complexes, le jeu d'instructions de l'ASIP encode plus d'opérations utiles par instruction que le TMS320C62, comme en témoignent les écarts significatifs observés sur le nombre d'opérations élémentaires nécessaires pour coder les fonctions critiques (Figure 135 page 190). L'importance de ces écarts ne doit cependant pas être exagéré puisqu'ils sont dus pour une grande partie au pipeline très complexe du TMS320C62, qui s'exprime surtout dans les boucles critiques des fonctions, et beaucoup moins dans les parties « contrôle » qui représente la majorité du code embarqué.

5.5.2.2 Autres structures d'encodage

Avec la structure d'encodage du TMS320C62, il est en fait très difficile de diminuer significativement la taille du code sans faire chuter de manière importante la performance, puisque les économies

devront se faire sur le nombre de registres opérandes et/ou le nombre d'opcodes codant les opérations. Pour la réalisation matérielle, il est certainement plus intéressant de privilégier d'autres structures d'encodage. Dans les processeurs DSP actuelles, il en existe principalement 2 autres : les structures CLIW (*Configurable Long Instruction Word*) et VLES (*Variable Length Execution Set*). La première est utilisé dans les processeurs *CARMEL* et *R.E.A.L* et repose sur un jeu d'instructions de base de type RISC associé à une table CLIW contant des instructions VLIW configurables (cf. Figure 57 page 78), appelées uniquement dans les noyaux des fonctions critiques. Cette solution a l'avantage d'assurer une compacité de code de type RISC (donc très bonne) tout en permettant d'utiliser quand il le faut la puissance du VLIW. Son premier défaut est lié à la mauvaise interaction avec le compilateur : le cas du compilateur du CARMEL a montré qu'il est très difficile pour un compilateur d'utiliser efficacement ces instructions. D'autre part, le fait d'imposer une structure au mot VLIW (par exemple MAC1 | MAC2 | ALU1 | ALU2 | UCA1 | UCA2) rend cette solution beaucoup moins souple puisqu'on ne peut par exemple pas coder une instruction de type « MAC1 | MAC2 | Décaleur », la structure ne permettant pas l'usage simultané des deux unités MAC et du décaleur. L'analyse des fonctions critiques de l'EFR montre pourtant que les instructions les plus parallèles ont une assez grande disparité en terme d'opérations élémentaires exécutées, ce qui plaide en faveur d'une structure VLIW plus souple.

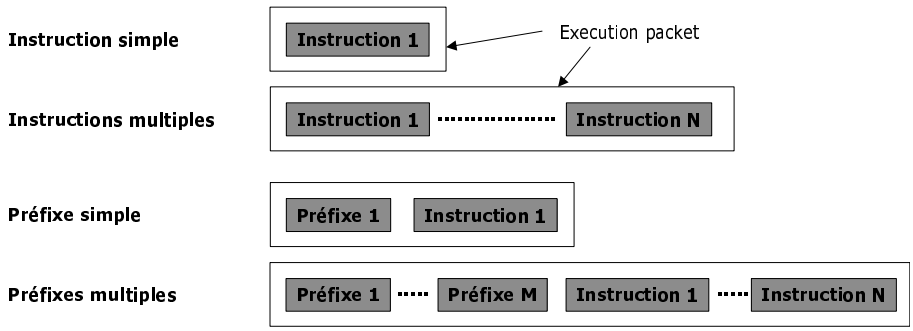


Figure 144 Structure d'encodage VLES

La structure VLES (cf. Figure 144) implémentée dans le récent *StarCore* [82] est plus proche de celle du TMS320C62. Elle repose sur un jeu d'instructions de base dont les opérations sont codées sur 16 bits et peuvent être exécutées en parallèle ou séquentiellement en utilisant des bits de synchronisation selon le même principe que pour le TMS320C62. En plus des mots de 16bits codant des opérations élémentaires, le mot d'instruction VLIW peut contenir un certain nombre de « préfixes » (eux aussi sur 16 bits), qui encodent des informations supplémentaires : marqueurs de boucle, conditionnement des opérations, et registres additionnels. En effet, pour permettre une largeur aussi faible (16 bits), le jeu d'instructions de base est soumis à des restrictions, dont la plus grande concerne l'utilisation des registres : seuls les 8 premiers registres peuvent être adressés (pour ne coder les opérandes que sur 3 bits), les autres registres réclamant obligatoirement un préfixe pour être utilisés.

Cette structure permet d'obtenir un excellent taux d'encodage pour les instructions non parallèles de type contrôle, qui représentent la plus grande partie du code, tandis que les quelques instructions parallèles des noyaux de calculs utilisent des préfixes pour étendre leurs capacités. Les premières estimations montrent d'excellents résultats en terme de compacité de code par rapport aux autres processeurs DSP. Cette solution peut être une alternative intéressante pour l'encodage des instructions

de l'ASIP. En effet, la faible réduction de la largeur des opérations de l'ASIP est principalement du au nombre inadapté de registres et d'opcodes. En définissant un jeu d'instructions de base manipulant moins de registres et d'opcodes, il est certainement possible de réduire considérablement la largeur d'une opération de base et donc de diminuer la taille du code de contrôle, tout en assurant une bonne performance et une utilisation optimale du parallélisme du processeur dans les fonctions critiques grâce à l'usage des préfixes. C'est donc une solution de ce type qu'il faut privilégier pour l'encodage des instructions de l'ASIP.

5.6 Estimation de complexité matérielle

Après avoir étudié les performances et l'encodage du jeu d'instructions de l'ASIP, nous allons maintenant nous intéresser à son coût matériel, afin de vérifier que les réductions appliquées à différents niveaux (nombre de registres, unités fonctionnelles, etc.) ont un effet significatif sur la réduction du matériel embarqué qui justifieraient l'emploi de l'ASIP par rapport à des processeurs plus généraux. Pour ce faire, nous allons comparer quatre processeurs : le TMS320C62, et trois modèles d'ASIP correspondant à trois degrés de spécialisation différents (du Niveau 1 au Niveau 3). Nous ne donnons ici que des résultats bruts, les conclusions qui découlent des comparaisons performance/coût matériel entre ces différents processeurs seront discutées en détail dans la dernière section (5.7).

Les trois versions retenues de l'ASIP privilégient toutes la performance et offrent la même puissance de calcul pour les fonctions critiques. Leurs différences proviennent de leurs différents degrés de spécialisation visant à réduire la complexité matérielle de leurs architectures :

- **Niveau 3** : correspond au degré de spécialisation le plus élevé. C'est le processeur qu'on obtient après réduction du nombre d'unités fonctionnelles (par la fusion des unités ALU et BMU/Décaleur), du nombre de ports de lecture du banc de registres et de la connectivité des multiplexeurs d'entrée et de sortie (cf. Figure 136 page 193).
- **Niveau 2** : correspond au modèle de connectivité « général » de la Figure 137. Par rapport au processeur spécialisé de niveau 3, il n'y a pas de réduction du nombre de ports de lecture du banc de registres ni de réduction de la connectivité du chemin de données. Le banc de registres pointeurs contient aussi un port d'écriture supplémentaire (cf. Figure 137 page 195).
- **Niveau 1** : modélise un processeur DSP de type CARMEL, avec 4 unités en parallèle dans le chemin de données (2 unités MAC, 1 ALU/BMU/Décaleur, et 1 ALU), 10 opérations codées en parallèle dans le mot VLIW, l'implémentation de la logique modulo sur les quatre UCAs, un plus grand nombre de ports de lecture/écriture du banc de registres pointeur. Par rapport au processeur précédent, la gestion d'une unité fonctionnelle supplémentaire et le plus grand nombre d'opérations encodées dans l'instruction oblige à augmenter le nombre de ports de lecture du banc de registres CU, les connexions dans le multiplexeur d'entrée, la largeur du mot d'instruction et donc du répartiteur, et la taille de certains registres du pipeline pour mémoriser l'opcode de l'opération.

Bien que ce dernier processeur puisse être qualifié de général à bien des aspects, il intègre les mécanismes spécialisés pour l'implémentation de Viterbi, les mécanismes de boucle « zéro-cycle », et dispose d'unités dédiées au calcul d'adresses, ce qui le rend plus spécialisé du point de vue traitement

du signal que le processeur TMS320C62. Il est représentatif de ce que l'on trouve dans certains processeurs DSP Bi-Macs spécialement dédiés pour les applications de communication, comme le nouveau *Philips R.E.A.L*, le *CARMEL d'Infineon* ou le *Texas C55x*. Ces processeurs, qui par abus de langage sont encore appelés processeurs DSP généraux, sont du point de vue de la spécialisation à mi-chemin entre les ASIPs et les processeurs réellement généraux comme le TMS320C62.

5.6.1 Méthodologie employée

Notre objectif étant de comparer le coût matériel entre plusieurs processeurs, nous n'avons pas besoin d'une mesure extrêmement précise mais plutôt d'une mesure reflétant l'accroissement ou la réduction de la complexité matérielle entre les différents processeurs. Dans l'absolu, la seule mesure réelle de complexité acceptable requiert le placement/routage du circuit sur une technologie donnée. Cela revient à concevoir et réaliser chaque processeur jusqu'au layout, ce qui est évidemment hors de question. Dès lors, une estimation acceptable de la complexité matérielle consiste à obtenir pour une technologie donnée le nombre de transistors nécessaire à la réalisation de ces processeurs.

Parce que ces processeurs n'ont pas été entièrement réalisés, on ne peut estimer la complexité à la porte logique près. En particulier la réalisation des processeurs réclame l'implémentation d'un certain nombre de blocs combinatoires de contrôle, dont la structure est difficilement estimable sans être passé par l'étape de réalisation. Cependant, en terme de complexité matérielle, ces blocs sont négligeables par rapport aux autres éléments du processeur, et varient peu entre les différents modèles d'ASIP. Notre estimation de complexité tient compte des éléments les plus coûteux pour chaque unité matérielle de l'ASIP : unité CU, unité AGU et unité PCU. Pour les unités CUs et AGU, l'ensemble des éléments matériels de calcul ont été pris en compte dans le calcul de complexité : banc de registre, unités fonctionnelles, unités de calcul d'adresses, multiplexeurs, etc. Pour l'unité PCU, on a pris en compte l'ensemble des éléments matériels présents sur la Figure 71 page 96. En particulier, les tailles des registres du pipeline d'instruction et la complexité du répartiteur ont été précisément modélisées, du fait de leur impact non négligeable sur la complexité totale. Au final, seuls manquent les estimations pour les blocs de contrôle précédemment évoqués ainsi que pour le décodeur d'instructions. On peut donc supposer que les chiffres obtenus donneront une idée relativement précise des différences de complexité entre processeurs.

5.6.1.1 Génération des unités de calcul

Pour l'ensemble des unités de calcul (excepté le banc de registres que nous verrons plus loin), les chiffres obtenus correspondent à une projection de ces unités sur la bibliothèque de cellules standard SXLIB 0.35µm de la chaîne d'outils CAO Alliance développée au sein de notre laboratoire [83]. Pour chaque unité (AGU, CU, PCU), le nombre de transistors obtenu correspond à la somme des contributions de chaque sous-élement matériel la composant : additionneurs /soustracteur, registres, multiplexeurs, comparateur, décaleurs, etc. Ces sous éléments sont réalisés à l'aide des générateurs GENOPTIM correspondants, qui nous permettent d'obtenir rapidement leur description au niveau portes réelles de la bibliothèque. Le nombre de transistors est alors calculé en tenant compte des caractéristiques physiques d'implémentation des portes utilisées. Dans le cas des unités de calcul du chemin de données, on a choisi des architectures privilégiant la performance : additionneur CLA, et multiplieur de Booth.

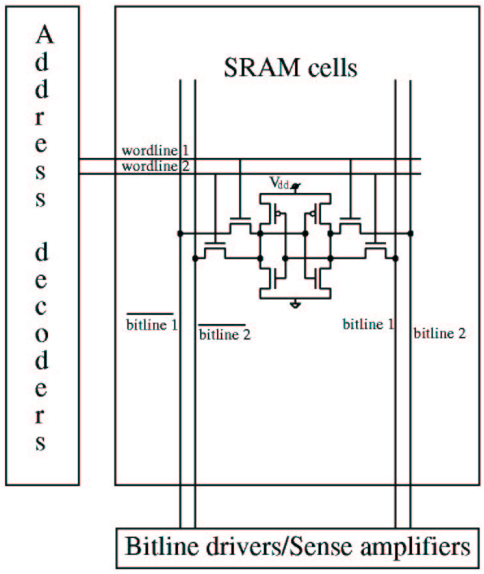


Figure 145 Représentation d'un banc de registre à 2 ports d'accès

5.6.1.2 Modélisation du banc de registres

Pour des raisons de performance et de surface de silicium, il est préférable de ne pas réaliser le banc de registres en cellules standards comme les autres opérateurs. Un banc de registres 32 bits contenant 10 registres et accessible via 8 ports de lecture et 4 ports d'écriture représente environ 42000 transistors (avec la librairie SXLIB), contre 15000 si on utilise une architecture « full-custom » de type SRAM comme celle de la Figure 145. On considère donc que le banc de registres sera plutôt réalisé en « full-custom ».

[84] propose un modèle simple d'estimation de la surface d'un banc de registre exprimée en nombre de transistors. Ce modèle se base sur les caractéristiques suivantes : 4 transistors par point mémorisant, 2 transistors par port d'accès, 2 transistors pour un « bitline driver » et 15 pour un « sense amplifier » ; enfin, pour chaque ligne, N+9 transistors pour le décodage (N étant le nombre de bits d'adresses nécessaires pour accéder aux mots). La formule de calcul du nombre total de transistors d'un banc de registres de largeur *width* comportant *Nreg* registres, *Nread* ports de lecture et *Nwrite* ports d'écriture est alors la suivante :

*#Trans = 4*Nreg*width*	*// Points mémorisants*
+ 2(Nread+Nwrite)*Nreg*width*	*// Ports Lecture/Ecriture*
+ (LOG2(Nreg)+9)(Nread+Nwrite)*Nreg*	*// Decodeurs de ligne*
*+ (PE16*2+PL16*15)*width*	*// « Bitline drivers »+ « sense amplifiers »*

5.6.2 Résultats

La Figure 146 compare les complexités matérielles des unités AGU et CU duTMS320C62 et des trois versions de l'ASIP. Comme il n'est pas possible d'estimer précisément l'unité de contrôle du TMS320C62 par manque d'informations sur son architecture interne, on se contente donc de comparer la complexité des unités de calcul « adresses » et « données ». En fait, le TMS320C62 ne

possède pas d'unités de calcul d'adresses dédiée, les calculs sur les adresses et les données sont indifférenciés et effectués dans le chemin de données. Les registres pointeurs, les registres servant au calcul et ceux utilisés pour le comptage des boucles logicielles sont tous dans les deux « clusters » de registres A et B qui font office d'unités de mémorisation centrale. La séparation du banc de registres en deux bancs séparés est sans aucun doute motivé par des raisons de performance et d'économie matérielle. En effet, le même nombre de registres et de ports d'accès dans un seul banc occuperait environ 70% de surface en plus, pour un temps d'accès de l'ordre du double.

Par rapport à un processeur ASIP de niveau 1, on constate que la complexité matérielle est sensiblement identique, bien qu'elle ne provienne pas des mêmes éléments pour les deux processeurs : dans le TMS320C62, plus de la moitié de la surface est occupée par le banc de registre, contre environ 40% pour l'ASIP, qui compense par des unités fonctionnelles plus nombreuses et surtout par une connectivité plus grande nécessaire pour implémenter l'architecture à accès mémoire directe. On notera au passage le très faible ajout de complexité occasionné par les mécanismes dédiés à Viterbi : seulement 3% de l'unité CU complète.

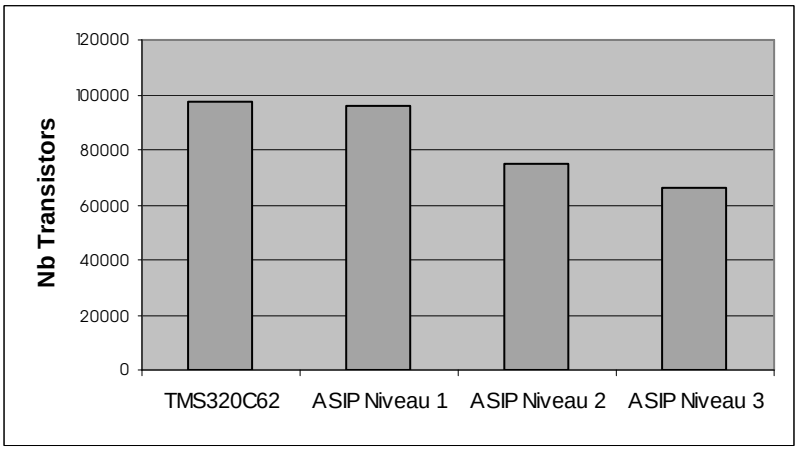


Figure 146 Complexité Matérielle des unités AGU et CU

Pour la comparaison des trois versions d'ASIP, nous avons aussi modélisé la taille de l'unité PCU qui prend en compte entre autres le répartiteur d'opérations et les registres du pipeline d'instruction et des mécanismes « zéro-cycles ». La première chose qui frappe est l'importance considérable de l'unité de contrôle dans la complexité totale du processeur : environ 40% quelque soit la version de l'ASIP, dont 19% du au répartiteur et 16% du aux registres de pipeline. La taille de ces deux éléments est principalement influencé par le nombre d'opérations maximum encodées dans une instruction VLIW. On le constate aisément en comparent la taille de l'unité PCU entre les versions 1 et 2 de l'ASIP, qui chute de 30% pour une diminution du parallélisme d'instructions de 10 à 7. La suppression d'une unité fonctionnelle dans le chemin de données permet de réduire de 20% le coût de l'unité CU, à travers la réduction du nombre de ports d'accès au banc de registres, la diminution de la connectivité en entrée des UFs et bien sûr la suppression de l'unité elle-même. Quant à l'unité AGU, la réduction du nombre de ports d'accès aux registres pointeurs et la suppression des capacités d'adressage modulo pour les 3 dernières UCAs permettent de sauvegarder 30% du matériel. Au final, la réduction matérielle totale entre la version 1 de l'ASIP et la version 2 est de l'ordre de 25%.

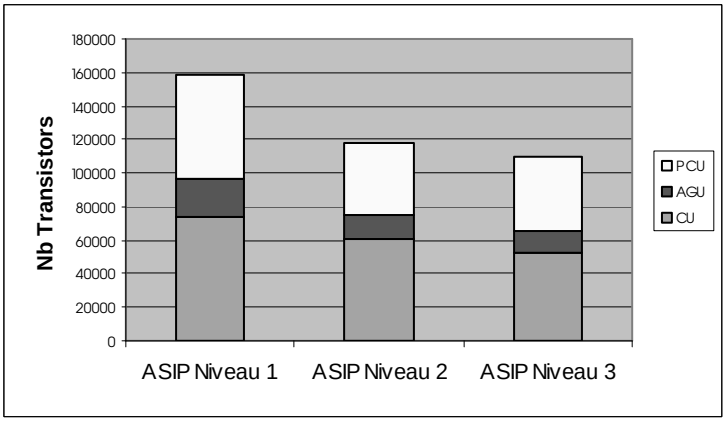


Figure 147 Complexité matérielle des trois ASIPs

La réduction supplémentaire entre la version 2 et la version 3 est due à la réduction du nombre de ports de lecture du banc de registres (grâce au partage des ports entre le unités fonctionnelles) et à la suppression d'un port d'écriture dans le banc de registres pointeurs. Par rapport à la version 1, cela représente un gain supplémentaire de 5%.

5.6.3 Compromis Performance/Coût matériel

Bien que le cheminement de cette présentation donne l'illusion d'un processus de spécialisation séquentiel effectué en une seule passe, on rappelle que la méthodologie de conception proposée est de nature itérative : programmation des fonctions, simulation, estimation du coût du processeur et de ses performances, et en fonction des contraintes du cahier des charges, raffinement de l'architecture et du jeu d'instructions jusqu'à l'obtention d'une version finale du processeur. Il existe bien sûr une multitude de solutions différentes des trois versions proposées précédemment, qui correspondent à des processeurs privilégiant avant tout la performance. En particulier, il est possible de « sacrifier » une certaine partie de la puissance de calcul pour pousser plus loin la réduction du coût du processeur. Nous donnons ici deux exemples de tels compromis.

L'analyse de fonctions critiques a montré qu'il était possible de réduire encore plus la complexité du chemin de données en diminuant le nombre de ports d'accès au banc de registres CU (5 ports de lecture et 4 ports d'écriture), ainsi que le nombre d'écritures dans le banc de registres pointeurs. Cette réduction représente une diminution supplémentaire de 2% du coût matériel, au prix d'une très légère perte de performance, mais surtout d'un ajout de contraintes supplémentaires pour le jeu d'instructions, puisque les ports de lecture devront être encore plus partagés. Vu le faible gain potentiel, cette solution n'a pas été retenue.

Parce que la fonction « search_10i40 » requiert beaucoup plus de registres pointeurs que toutes les autres fonctions, on peut essayer de diminuer ce nombre de registres. Il apparaît possible d'économiser 5 registres (soit 13 registres) en sacrifiant environ 3% de la charge totale de calcul de l'EFR, ce qui représente une perte relativement importante. Or, l'économie matérielle réalisée n'est que de 1%, et la diminution du nombre de registres pointeurs n'a aucune influence sur la largeur d'encodage. La perte est encore plus importante si on tente de réduire le nombre de registres *Ax*. Au vu des faibles gains réalisés, on a donc décidé de conserver un nombre de registres optimal.

5.7 Conclusion

Le tableau ci-dessous résume les différences entre processeurs concernant les critères de performance, de coût matériel, de compacité de code, et d'interaction avec le compilateur. Même si certains de ces chiffres sont approximatifs, ils renseignent sur les forces et faiblesses des différentes solutions et permettent de dégager un certain nombre de conclusions sur l'intérêt d'un modèle de processeur configurable.

	TMS320C62	*ASIP niveau 1*	*ASIP niveau 2*	*ASIP version 3*
Performance EFR (MIPS)	1	0.79	0.79	0.79
Performance Viterbi (MIPS)	1	0.45	0.45	0.45
Performance GSM (MIPS)	1	0.63	0.63	0.63
Complexité matérielle	100	98	74	68
Qualité interaction Compilateur	Moyenne	Bonne	Assez bonne	Mauvaise

Performance

Le gain en performance obtenu par l'usage d'un processeur spécialisé ASIP par rapport à un processeur général dépend complètement de la structure des algorithmes implantés. Lorsqu'ils ont la même structure que les noyaux de calcul courants de type « produit scalaire », ces algorithmes offrent très peu de place pour un degré de spécialisation supplémentaire, et les seules améliorations possibles consistent à redimensionner certains éléments du processeur selon les besoins globaux de l'application. Par exemple, l'analyse des fonctions critiques a montré que pour obtenir une performance maximale sur l'EFR, l'ASIP devait contenir au moins 18 registres et permettre un parallélisme d'exécution de 7 opérations élémentaires par instruction. L'ASIP niveau 1, qui a les mêmes caractéristiques que le processeur CARMEL d'Infineon (même architecture, même unités fonctionnelles, même bande passante) respecte ces contraintes et permet donc d'obtenir une performance optimale pour l'application. A l'inverse, le « vrai » processeur CARMEL ne contient que 12 registres pour un parallélisme de 8 opérations par instruction. Sa performance est donc en deçà de celle de l'ASIP à cause du manque de registres, tandis que sa complexité matérielle est plus grande car le répartiteur et les registres de pipeline manipulent 8 opérations en parallèle au lieu de 7.

De manière plus générale, la performance de ce genre d'algorithmes sur les processeurs DSP est limitée par deux facteurs : le nombre de ressources matérielles générales de calcul du processeur (nombre d'unités MAC, nombre de registres, bande passante), et le parallélisme intrinsèque de l'application. Pour ces algorithmes, la spécialisation consiste simplement à dimensionner correctement ces différentes ressources en fonction du parallélisme de l'application. Comme les processeurs DSP du commerce sont conçus en tenant compte des caractéristiques des algorithmes DSP les plus populaires dont l'EFR fait parti, les différences de performance entre un ASIP et un processeur du commerce possédant le même nombre d'unités MAC (principal facteur de limitation du parallélisme) seront relativement faibles.

La spécialisation de l'architecture et du jeu d'instructions devient vraiment intéressante dès qu'un des algorithmes de l'application sort du cadre traditionnel « produit scalaire » caractérisant la plupart des algorithmes DSP. C'est le cas de l'algorithme de Viterbi, pour lequel l'intégration dans le jeu

d'instructions de nouveaux opérateurs dédiés, associés à des mécanismes matériels très peu coûteux ajoutés au chemin de données, permet d'accroître considérablement la performance du processeur. Si l'algorithme en question représente une charge de calcul significative sur l'ensemble de l'application visée, il devient alors très intéressant de spécialiser le processeur, car les « MIPS » économisés permettront au choix de réduire la fréquence de fonctionnement ou d'adjoindre des fonctionnalités supplémentaires au code embarqué. Pour Viterbi, l'ajout de quelques instructions et d'un peu de matériel a ainsi permis de gagner 55% de performance en plus par rapport au TMS320C62.

Complexité matérielle

A performance égale, la spécialisation permet de réduire le coût matériel de l'ASIP par rapport à un processeur du commerce. A cet égard, elle peut s'avérer très efficace : l'ASIP niveau 3 requiert 30% de matériel en moins par rapport au TMS320C62, pour une performance supérieure de 37%. Et il est même possible de réduire encore plus en renonçant au surplus de performance, par exemple en diminuant le parallélisme d'opérations ou en diminuant le nombre de registres. La différence marquée par rapport aux processeurs du commerce s'explique du fait que ces processeurs ont souvent des ressources matérielles surdimensionnées pour pouvoir exécuter un grande nombre d'algorithmes, dont les besoins quantitatifs en ressources matérielles sont très variés. Au contraire, l'ASIP peut se permettre de n'intégrer que le nombre de ressources indispensable aux quelques algorithmes critiques de son application cible. De ce point de vue, le fait de disposer d'un modèle de processeur dont le nombre de ressource est configurable permet d'obtenir rapidement un processeur à faible coût matériel.

Consommation

Aujourd'hui, dans le domaine de la conception de circuits, la réduction de la surface de silicium des cœurs de processeurs n'est plus prioritaire, surtout dans le domaine de l'embarqué où la consommation est le critère numéro un. Comme nous ne disposons pas pour l'instant de modèle de consommation de l'ASIP, on ne peut estimer précisément les différences de consommation entre les différents processeurs. Cependant, on peut raisonnablement supposer que le fait de diminuer le matériel de 25% a déjà un impact bénéfique sur la consommation du processeur. De même, l'accroissement de la performance de l'ASIP de 37% (lié à une meilleure exploitation du parallélisme) permet dans l'absolu de réduire la fréquence de fonctionnement et donc la consommation du même facteur. Dans l'optique d'une exploration de l'espace de configuration orientée vers la réduction de consommation, les choix d'architecture ne seront forcément les mêmes que pour la réduction du coût matériel ou l'optimisation de performance. On peut par exemple imaginer qu'il peut être bénéfique d'accroire le nombre de registres internes au processeur afin de diminuer les accès mémoire. De même, la réduction de la largeur du mot d'instruction permet de réduire les activités de transition dans le décodeur et le répartiteur qui sont les deux plus gros blocs combinatoires du processeur.

Compacité du code

La structure d'encodage des mots d'instruction VLIW fait qu'il est très difficile de réduire de manière significative la largeur d'un mot d'instruction en jouant sur la largeur des opérations élémentaires la composant. La largeur d'une opération dépend essentiellement du nombre de registres et d'opcodes différents du jeu d'instructions. Pour économiser un seul bit, il est souvent nécessaire de sacrifier de nombreux registres ou opcodes, dont l'absence peut avoir des effets désastreux sur la performance.

Une autre méthode efficace consiste à limiter le parallélisme d'opérations au sein d'une instruction, mais là encore tout dépend des contraintes en performance de l'application. Actuellement, la meilleure stratégie d'encodage pour jeu d'instructions DSP VLIW semble être l'encodage à longueur variable (ex : encodage VLES du processeur StarCore) qui permet de réduire la taille des opérations élémentaires utilisés dans le code de contrôle (cf. 5.5.2). Cette méthode peut être considérée comme une spécialisation de l'encodage, mais au niveau « traitement du signal » et non au niveau de l'application elle-même, puisqu'elle se base sur des propriétés structurelles générales à l'ensemble des applications DSP.

Interaction avec le compilateur

Vu le faible gain matériel obtenu par la réduction de la connectivité du chemin de données et le partage des ports du banc de registres (ASIP niveau 3) par rapport à une version plus générale (ASIP niveau 2), il paraît peu probable que ce type de spécialisation ait encore un intérêt pour des processeurs VLIW. A l'heure où les méthodologies de développement logicielles se basent de plus en plus sur l'emploi de compilateurs pour générer une grande partie du code, le fait de détruire la régularité de l'architecture en réduisant les connexions alourdit considérablement la complexité des algorithmes de génération de code et risque de détériorer la qualité du code généré. Détruire la régularité du chemin de données signifie en quelque sorte revenir au principe des processeurs conventionnels à architecture hétérogène, pour lesquels l'expérience a montré qu'il était quasi-impossible d'obtenir des compilateurs réellement performants. La spécialisation de paramètres ne détruisant pas la régularité comme le nombre de registres, le nombre et le type d'unités fonctionnelles ou la largeur des interconnexions, permet déjà d'obtenir des réductions matérielles substantielles sans pour autant complexifier le compilateur. On peut donc **restreindre la spécialisation** de notre modèle générique d'ASIP à ce type de paramètres et négliger la réduction de la connectivité, ce qui simplifiera à la fois la finalisation du compilateur mais aussi la réalisation matérielle à l'aide de générateurs paramétrables. Dans le cas du processeur spécialisé GSM, cela revient à choisir l'« ASIP niveau 2 ».

Intérêt du modèle configurable

Au vu des remarques précédentes, on constate que la spécialisation de notre modèle de processeur présente un intérêt certain pour respecter les besoins spécifiques d'une application tout en minimisant les autres types de contraintes. La définition par l'utilisateur de paramètres architecturaux et logiciels tels que le nombre de registres, la bande passante mémoire ou le nombre d'unités fonctionnelles générales, associé à la possibilité d'inclure des instructions et du matériel spécialisé permet d'obtenir un processeur plus performant et plus « économique » qu'un DSP général. De plus, en limitant la spécialisation à des paramètres généraux ne détruisant pas la régularité, l'architecture VLIW du modèle et le jeu d'instructions générique permettent une bonne interaction avec un compilateur reciblable tel qu'il a été défini au chapitre 4. Associé à une méthodologie d'implémentation matérielle indépendante de la technologie et semi-automatique, ce modèle de processeur devrait permettre une réduction substantielle du temps de développement des processeurs ASIP.

5.8 Travaux futurs et perspectives

Dans cette thèse, nous avons cherché à traiter l'ensemble des problématiques liées au concept de processeur configurable : l'architecture et le jeu d'instructions du processeur, mais aussi la méthodologie de conception, les outils logiciels et la phase de réalisation matérielle. Nous aurions pu focalisé notre recherche sur un seul de ces sujets, mais l'interaction architecture/méthodologie/logiciel est si forte dans le cas des processeurs ASIP qu'il nous a semblé préférable de privilégier une approche globale. La contrepartie de cette approche est que certains sujets n'ont été que succinctement abordés, et que certaines idées émises mais n'ont pu faute de temps être validées par une réalisation matérielle et/ou logicielle. Nous allons maintenant faire le point sur les travaux restant à faire et proposer d'autres axes de recherche.

5.8.1 Compilateur et phase d'exploration

Le prototype de compilateur réalisé dans le cadre de cette thèse a avant tout servi à étudier les problèmes posés par l'interaction entre notre processeur configurable et un compilateur. Les phases de génération de code n'ayant pas directement à voir avec ce problème ont été réalisées au plus vite et sont sans aucun doute perfectibles. En outre, il est nécessaire d'adjoindre des algorithmes de production de code spécialisés « VLIW » afin de tirer parti du parallélisme matériel du processeur. De tels algorithmes existent dans la littérature et doivent permettre d'obtenir un compilateur réellement performant.

L'étape d'analyse de complexité que nous proposons dans la première partie de la phase d'exploration repose sur une annotation du code C par des fonctions de comptage ; cela représente un travail assez fastidieux et ne reflète pas totalement fidèlement le comportement de l'application sur le processeur de référence. On gagnerait grandement à utiliser le compilateur dès cette phase : l'application cible serait compilée sur un premier modèle de référence du processeur, et les décisions concernant la sélection des opérateurs et des fonctions critiques seraient prises après simulation du code assembleur sur le modèle de processeur. Cela permettrait entre autres de supprimer la phase d'annotation de code, mais aussi de tenir compte de paramètres matériels « fins » du processeur comme les instructions multi-cycles dues au pipeline. Dans l'hypothèse où le compilateur gère correctement le parallélisme d'instructions, on pourrait adjoindre au processeur de référence plusieurs unités fonctionnelles virtuelles « universelles » capables d'exécuter n'importe lequel des opérateurs, et pouvoir ainsi déterminer non seulement le type mais aussi le nombre de chaque opérateur à intégrer dans le processeur. Ce qui reviendrait en fait à supprimer l'étape d'analyse de complexité.

Avec l'accroissement constant de la qualité des algorithmes de compilation, on peut espérer que dans un avenir proche la programmation manuelle en assembleur des DSPs sera devenue inutile. Dans ce cas, la phase d'étude et de codage des fonctions critiques, à l'issue de laquelle on détermine les différents paramètres du processeur, pourra être remplacée par une phase de compilation de l'application complète (fonctions critiques + code de contrôle), suivie d'une phase d'analyse du code compilé permettant d'extraire les valeurs des paramètres. Cela sous-entend que le compilateur soit capable d'exploiter au mieux chaque modification ou ajout dans l'architecture ou dans le jeu d'instructions. La phase d'exploration deviendra alors très rapide, l'utilisateur n'ayant plus qu'à spécifier les paramètres du processeur, compiler l'application et simuler le code obtenu pour tester

l'impact des choix architecturaux et logiciels. Cette solution, proche de l'idéal « presse-bouton », est ce vers quoi tendent l'ensemble des projets portant sur la compilation reciblable.

Enfin, afin de répondre aux contraintes des applications embarquées, il est indispensable de prévoir dans la phase d'exploration un module d'évaluation de consommation qui permette d'estimer l'impact des choix matériels et logiciels selon ce critère. La structure de la phase d'exploration présentée dans le chapitre 4, basée sur le partitionnement fonctions critiques/code de contrôle et sur le codage manuel en assembleur, devra sans aucun doute être modifiée, les exigences et méthodes d'optimisations liées aux critère « faible consommation » et « performance » pouvant être très différentes.

5.8.2 Langage de description de processeur

En l'état actuel, le passage d'informations aux outils concernant la valeur des paramètres du processeur et du jeu d'instructions se fait de manière statique. Si l'on décide par exemple d'ajouter une unité fonctionnelle et un ensemble d'instructions supplémentaires au modèle, il faut inclure ces informations dans le code source de chaque outil, et les recompiler. Afin d'aller plus loin dans l'automatisation du flot de conception, il peut être intéressant d'utiliser un seul fichier de configuration, décrit dans un langage particulier dit de « description de processeur », grâce auquel toutes les informations relatives à l'état du modèle (architecture et jeu d'instructions) sont transmises à tous les outils logiciels impliqués dans les phases d'exploration et de réalisation. De tels langages ont déjà été proposés dans le cadre de projets portant sur la compilation reciblable pour pouvoir décrire les différents processeurs cibles (cf. 4.4.2.1) et peuvent servir de support pour la description de notre processeur, à condition d'adapter les outils logiciels déjà existants pour qu'ils prennent en compte le format correspondant.

5.8.3 Réalisation matérielle

Dans ce travail, nous nous sommes avant tout intéressés à la première phase de la méthodologie (exploration d'architecture), le problème de la réalisation matérielle n'ayant été que très succinctement abordé. La méthode de réalisation matérielle que nous proposons basée sur l'emploi de générateurs paramétrables doit permettre la réalisation rapide de l'essentiel du chemin de données du processeur. Pour la partie contrôle, l'usage d'un fichier VHDL modifié à la main en fonction des paramètres du modèle est possible mais loin d'être satisfaisant. Là encore, l'usage d'un langage de description de processeur peut être utile. On peut concevoir un outil de synthèse matérielle lisant le fichier de description du processeur, générant la partie de chemin de données en lançant les générateurs avec la bonne valeur des paramètres, et générant le fichier VHDL de contrôle en fonction des caractéristiques du modèle. Cette solution combinant générateurs et synthèse automatique de la partie contrôle est parfaitement envisageable : elle a d'ailleurs déjà été mise en œuvre dans le projet de processeur configurable *MetaCore* [47].

5.8.4 Architectures SIMD et partitionnées

L'architecture des processeurs DSP a de plus en plus tendance à diverger en deux catégories : des architectures double précision (16/32 ou 24/48 bits), mono-scalaires ou faiblement parallèle (pas plus de deux MACs), destinées aux applications embarquées de type Telecom ; et des architectures fortement parallèles de type VLIW et/ou SIMD, quadruple précision (8/16/32/64 bits) utilisées pour

des applications multi-canal ou des applications de type traitement d'images. Le modèle de processeur que nous avons proposé appartient sans aucun doute à la première catégorie puisqu'il ne gère que la double précision et que la structure du jeu d'instructions se prête mal au SIMD.

Dans l'hypothèse où l'on souhaite s'intéresser au deuxième champ d'application, nous pensons qu'il est préférable de spécifier un autre modèle de processeur de type SIMD multi-précision, avec une architecture et une structure de jeu d'instructions spécialement adaptés, plutôt que d'étendre le modèle actuel. L'exemple de l'implémentation de l'EFR sur le processeur TigerSharc *d'Analaog Devices* montre clairement qu'un jeu d'instructions fortement SIMD et multi-précision donne de piètres résultats sur des algorithmes mono-dimensionnels, comme ceux de l'EFR et de la plupart des algorithmes utilisés en télécommunication.

Une autre technique architecturale est très utilisée dans les architectures VLIW fortement parallèles : le partitionnement (« *clustering* ») des unités de calcul et des bancs de registres en différents *clusters* répartis en parallèle dans le chemin de données. Cette technique sert principalement à réduire le coût lié à l'usage de bancs de registres multi-ports. Le partitionnement des bancs de registres permet de diminuer la consommation et la surface et offre de meilleurs performances, au prix d'une plus grande complexité pour la génération de code. Dans l'optique d'un modèle de processeur à architecture fortement parallèle, cette technique semble être incontournable.

.

Conclusion

Nous nous sommes intéressés dans ce travail à l'étude et la spécification d'un cœur de processeur DSP configurable destiné à faciliter le développement de processeurs spécialisés de type ASIP. Nos objectifs étaient les suivants :

- définir un modèle d'architecture configurable et un jeu d'instructions générique permettant l'intégration de structures matérielles et d'instructions spécialisées, et assurant une bonne interaction avec un compilateur.
- proposer une méthodologie de développement d'ASIP définissant les différentes étapes du processus de réalisation et les outils logiciels à mettre en œuvre.
- évaluer la validité du concept en l'appliquant à la spécification d'un processeur spécialisé implémentant une application complexe du domaine DSP embarqué.

Nous avons proposé pour le processeur un modèle d'architecture configurable de type VLIW, qui permet à la fois de faire varier le nombre de ressources matérielles générales et d'intégrer des unités matérielles spécialisées pour accélérer certains algorithmes critiques. L'architecture globale du processeur se décompose en quatre blocs. Pour chacun de ces blocs, nous avons défini un modèle d'architecture associé à un ensemble de paramètres qui permettent d'adapter la structure du processeur aux caractéristiques de l'application. Le contrôle du processeur est assuré par un jeu d'instructions générique, qui autorise un degré de parallélisme variable et qui peut être spécialisé en ne sélectionnant que les instructions utiles pour l'application cible et en ajoutant éventuellement certaines instructions spécialisées.

La méthodologie de développement proposée est composée d'une phase d'exploration permettant de déterminer la valeur optimale des différents paramètres du processeur, et d'une phase de réalisation matérielle. La phase d'exploration est composée d'une première étape d'analyse de complexité qui examine les caractéristiques dynamiques de l'application cible afin de déterminer les fonctions critiques de l'application et les différents types d'opérateurs à inclure dans le jeu d'instructions. La seconde étape de nature itérative permet de tester différentes configurations du processeur afin de déterminer la configuration optimale du modèle. Elle fait appel à deux principaux outils logiciels : un compilateur et un simulateur de jeu d'instruction. Pour chacun de ces outils, nous avons proposé un cahier des charges et une structure permettant de les implémenter, et réalisé un prototype illustrant ces propositions.

Pour la deuxième phase de réalisation matérielle, nous avons proposé une méthode basée sur l'emploi de générateurs de macroblocs paramétrables et portables développés au laboratoire.

Enfin, au travers d'un exemple d'application complexe (les fonctions de base du protocole GSM), nous avons mis en œuvre cette méthodologie pour converger vers un processeur DSP spécialisé.

L'analyse des résultats en terme de performance et de complexité matérielle montre clairement l'intérêt de notre modèle de processeur configurable, qui permet d'obtenir des processeurs dont le facteur performance/coût est meilleur que ceux des processeurs DSP généraux. En autorisant au sein d'une architecture VLIW la variation du nombre de certaines ressources matérielles générales, et l'intégration au processeur et au jeu d'instructions d'unités fonctionnelles et d'instructions utilisateur, la méthodologie de conception permet de converger vers un ou des processeurs dont les caractéristiques collent au plus près des contraintes du cahier des charges.

Principales contributions apportées par cette thèse :

- **Processeur configurable**

  Le concept de processeur VLIW configurable est très récent et n'a pour l'instant donné lieu qu'à une seule réalisation industrielle significative : le processeur JAZZ d'*Improv* [52]. Ce sujet suscite actuellement de nombreuses recherches, comme le projet COCOON de *Philips* dans lequel le modèle de processeur est très proche de celui que nous présentons [85]. En définissant une méthodologie de conception associée à notre modèle et en l'appliquant à une application complexe représentative des futurs besoins en traitements DSP embarqués, nous avons prouvé l'intérêt d'un modèle de processeur VLIW configurable pour la réalisation de processeurs spécialisés. Par ailleurs, la mesure précise des performances et du coût matériel de l'ASIP final obtenu renseigne sur les différences quantitatives que l'on peut attendre par rapport à une implémentation sur un processeur DSP général.

- **Exploration d'architecture, simulateur et compilateur**

  Nous avons défini une phase d'exploration de l'espace de configuration permettant d'extraire les caractéristiques pertinentes d'une application cible pour le paramétrage du modèle de processeur. Cette phase repose sur l'usage d'une bibliothèque d'opérateurs arithmétiques pour la description du code source et sur deux principaux outils logiciels : le compilateur et le simulateur de jeu d'instructions. Pour permettre d'obtenir des mesures précises en performance au cycle et au bit près, nous avons réalisé un simulateur de modèle virtuel capable de simuler n'importe quelle configuration du processeur. Afin d'illustrer les problèmes liés à la prise en compte des différents paramètres du modèle dans la génération de code, nous avons réalisé un prototype de compilateur basé sur l'environnement de développement de compilateurs reciblables SPAM. Nous avons abordé un certain nombre de problèmes liés à l'interaction processeur/compilateur : structure du flot de compilation, utilisation des structures matérielles spécialisées du modèle, et prise en compte d'instructions utilisateur dans le flot de compilation. La version actuelle du compilateur est capable de prendre en compte des instructions utilisateur supplémentaires et de générer du code séquentiel pour le processeur.

- **Comparatif processeur général / processeur spécialisé**

Il est difficile de trouver des comparatifs précis entre processeurs DSP qui portent sur de véritables applications complètes, et non seulement sur quelques petits noyaux de calcul de moins en moins représentatifs des exigences des nouvelles applications. Pour valider et estimer l'intérêt de notre approche, nous avons codé en assembleur un ensemble de fonctions dont la complexité dépasse largement celles des noyaux de calculs habituellement utilisés. Nous avons utilisé différentes techniques de codage permettant d'optimiser du code assembleur pour des architectures VLIW. Les mesures de performance sur ces fonctions nous renseignent sur les caractéristiques matérielles et logicielles des processeurs qui jouent un rôle important dans l'amélioration des performances. Ces chiffres ont permis d'établir une comparaison processeur DSP général/processeur spécialisé à la fois sur des algorithmes DSP classiques (comme ceux du codeur EFR) et des algorithmes plus spécialisés comme Viterbi. Nous avons aussi comparé les processeurs selon le critère de complexité matérielle pour illustrer l'impact de la spécialisation..

Tendances actuelles dans l'industrie

Voilà maintenant plusieurs années que notre laboratoire s'investit dans le domaine des architectures génériques et configurables ayant pour but de favoriser la conception et la réutilisation des « IP cores ». L'intérêt de nos travaux sur les processeurs configurables, qui visent à faciliter le développement de processeurs DSP spécialisés, semble être confirmé par la tendance industrielle actuelle, qui abandonne de plus en plus le concept du processeur DSP unique au profit du concept de modèle de processeur déclinable en plusieurs versions selon le type d'application cible. Le nouveau processeur TMS320C64 de Texas Instruments, qui fait suite au TMS320C62, rompt avec le coté « généraliste non spécialisé » de ce dernier puisqu'il se décline en trois versions : « générale », « réseau », et « 3G wireless », cette dernière incluant des coprocesseurs spécialisés pour l'algorithme de Viterbi et les Turbo codes. Le *Starcore* de Motorola joue lui sur la variation du parallélisme architectural et est disponible en deux versions , la première basée sur une architecture 4 MACs destinée aux applications DSP de type serveur (SC140), et la seconde n'intégrant qu'une seule MAC et visant les applications embarquées. Enfin, le successeur du CARMEL, le CARMEL2000, est un cœur de type VLIW auquel peuvent se raccrocher différents modules matériels commandés par le jeu d'instructions et à choisir parmi une bibliothèque : modules MAC supplémentaires, modules MPEG4, etc.

Les concepts sous-jacents à ces réalisations sont les mêmes que ceux sur lesquels repose notre travail : variation du parallélisme matérielle du processeur selon l'application, et ajout de modules matériels et logiciels spécialisés. Cette tendance est imposée par la diversité toujours croissante des domaines d'applications pris en charge par les processeurs DSP, pour lesquels les contraintes en terme de performance, de consommation et de coût matériel peuvent être radicalement différentes.

A l'heure actuelle, l'utilisateur se contente au mieux de choisir une version particulière d'un processeur dans un « catalogue » prédéfini, et n'a donc que peu d'influence dans le choix des structures matérielles et logicielles de « son » processeur. Pour aller plus loin dans le domaine de la « spécialisation utilisateur », il est nécessaire de donner à l'utilisateur le pouvoir d'agir dans le flot de conception pour « tailler » le processeur aux demandes de son application. C'est pourquoi on voit maintenant apparaître plusieurs projets de processeurs configurables dont les précurseurs sont les

processeurs JAZZ, LX et COCOON (respectivement d'*Improv*, de *ST-Microelectronics/HP* et de *Philips*). Il y'a donc de fortes chances pour que les processeurs DSP configurables aient un rôle très important à jouer ces prochaines années.

Travaux futurs et perspectives

Les principaux travaux à venir visent tout d'abord l'amélioration de la phase d'exploration de l'espace de configuration, qui repose sur l'usage d'un compilateur fiable et performant. Pour cela, il est nécessaire d'intégrer dans le compilateur des algorithmes de génération de code VLIW paramétrables qui puissent gérer les différents paramètres du modèle et tirer parti du parallélisme logiciel et matériel du processeur. De tels algorithmes existent dans la littérature, certains sont même présents dans la bibliothèque SPAM. Une fois le compilateur développé, la phase d'analyse de complexité basée sur l'annotation du code source pourra être supprimée et remplacée par une analyse du code compilé.

Sachant que la validation globale de notre méthodologie passe obligatoirement par la réalisation matérielle d'un processeur, il est nécessaire d'implémenter la phase de réalisation matérielle en la rendant la plus automatique possible. Ceci passe par l'usage d'un langage de description du processeur pour le passage des paramètres du modèle aux différents outils logiciels, par la description en langage GENOPTIM de macrogénérateurs paramétrables permettant la synthèse du chemin de données des différentes unités, et la réalisation d'un outil de synthèse de la partie contrôle.

Enfin, dans l'hypothèse où l'on s'intéresserait à un modèle de processeur configurable destiné aux applications de type serveur ou traitement d'images, il est nécessaire de repenser l'architecture logicielle et matérielle de notre modèle en y intégrant des clusters matériels et des capacités SIMD / multi-précisions.

ANNEXE A

Opérateurs arithmétiques ETSI

Types de données :

typedef short Word16;

typedef long Word32;

Opérateurs:

Word16 add (Word16 var1, Word16 var2); /* Short add */

Word16 sub (Word16 var1, Word16 var2); /* Short sub */

Word16 abs_s (Word16 var1); /* Short abs */

Word16 shl (Word16 var1, Word16 var2); /* Short shift left */

Word16 shr (Word16 var1, Word16 var2); /* Short shift right, 1 */

Word16 mult (Word16 var1, Word16 var2); /* Short mult, 1 */

Word32 L_mult (Word16 var1, Word16 var2); /* Long mult, 1 */

Word16 negate (Word16 var1); /* Short negate, 1 */

Word16 extract_h (Word32 L_var1); /* Extract high, 1 */

Word16 extract_l (Word32 L_var1); /* Extract low, 1 */

```
Word16 round (Word32 L_var1);            /* Round,              1   */
Word32 L_mac (Word32 L_var3, Word16 var1, Word16 var2);   /* Mac,  1  */
Word32 L_msu (Word32 L_var3, Word16 var1, Word16 var2);   /* Msu,  1  */
Word32 L_macNs (Word32 L_var3, Word16 var1, Word16 var2); /* Mac without sat, 1   */
Word32 L_msuNs (Word32 L_var3, Word16 var1, Word16 var2); /* Msu without sat, 1   */
Word32 L_add (Word32 L_var1, Word32 L_var2);    /* Long add,        2 */
Word32 L_sub (Word32 L_var1, Word32 L_var2);    /* Long sub,        2 */
Word32 L_add_c (Word32 L_var1, Word32 L_var2);  /* Long add with c, 2 */
Word32 L_sub_c (Word32 L_var1, Word32 L_var2);  /* Long sub with c, 2 */
Word32 L_negate (Word32 L_var1);                /* Long negate,     2 */
Word16 mult_r (Word16 var1, Word16 var2);       /* Mult with round, 2 */
Word32 L_shl (Word32 L_var1, Word16 var2);      /* Long shift left, 2 */
Word32 L_shr (Word32 L_var1, Word16 var2);      /* Long shift right, 2*/
Word16 shr_r (Word16 var1, Word16 var2);        /* Shift right with round, 2          */
Word16 mac_r (Word32 L_var3, Word16 var1, Word16 var2); /* Mac with rounding,2 */
Word16 msu_r (Word32 L_var3, Word16 var1, Word16 var2); /* Msu with rounding,2 */
Word32 L_deposit_h (Word16 var1);        /* 16 bit var1 -> MSB,     2 */
Word32 L_deposit_l (Word16 var1);        /* 16 bit var1 -> LSB,     2 */
Word32 L_shr_r (Word32 L_var1, Word16 var2); /* Long shift right with round,  3           */
Word32 L_abs (Word32 L_var1);            /* Long abs,              3  */
Word32 L_sat (Word32 L_var1);            /* Long saturation,       4  */
Word16 norm_s (Word16 var1);             /* Short norm,           15  */
Word16 div_s (Word16 var1, Word16 var2); /* Short division,       18  */
Word16 norm_l (Word32 L_var1);           /* Long norm,            30  */
```

ANNEXE B:

Code assembleur de la fonction search_10i40

Remarque : certaines variables utilisées dans les opérations des unités CU et AGU ne correspondent pas à des noms de registres mais aux noms de variables du code source C. Elles sont en fait déclarées comme alias vers des registres du processeur au début du programme, et remplacées par un pré-processeur au début de la lecture du fichier source par le simulateur.

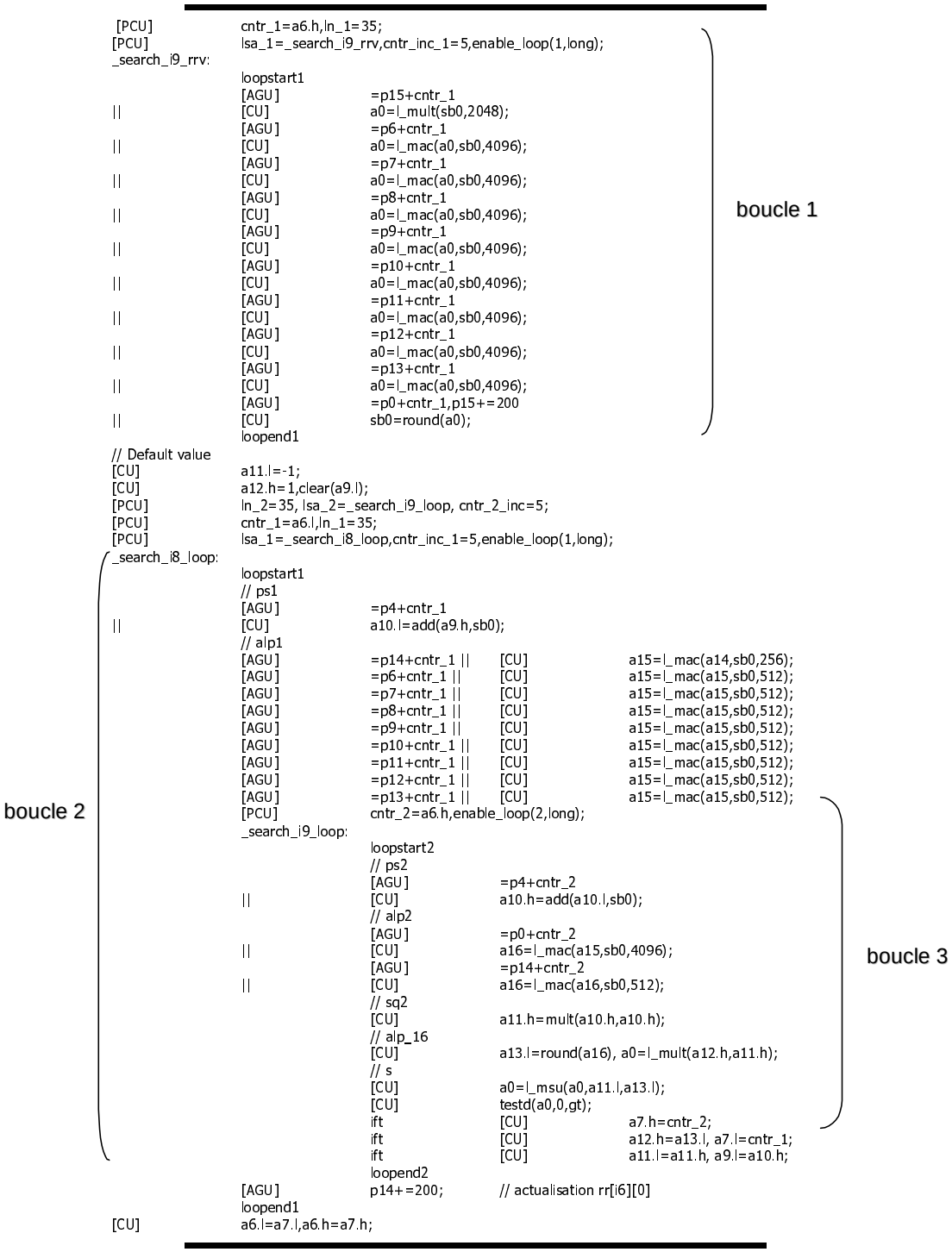


Figure 148 Code référence mono MAC de la fonction de recherche {i8,i9}

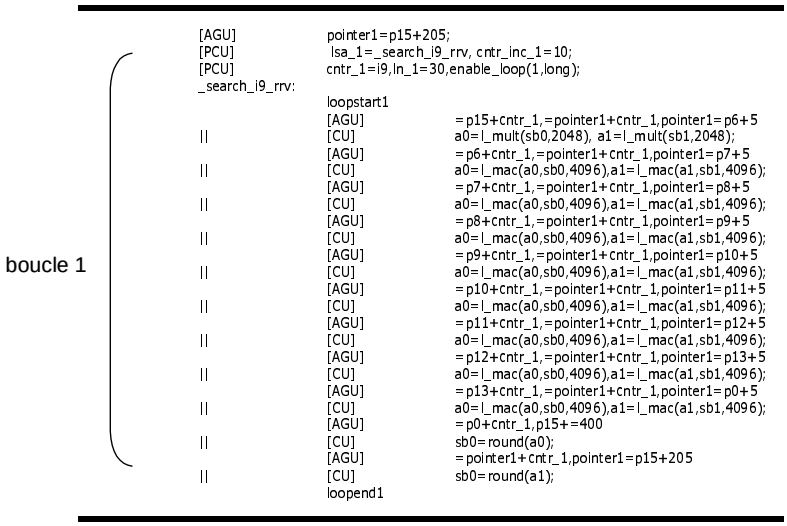


Figure 149 Code optimisé : muti-échantillons+ pipeline logiciel + l-msu-with_test (boucle 1)

```
[AGU]                pointer1=p4+5;
[AGU]                compteur=0;
[PCU]                ln_2=35, lsa_2=_search_i9_loop, cntr_2_inc=5;
[PCU]                cntr_1=a6.l,ln_1=30;
[PCU]                lsa_1=_search_i8_loop,cntr_inc_1=10,enable_loop(1,long);
_search_contrib_i8_loop:
                     loopstart1
                     // ps1 ps1_2
                     [AGU]          =p4+cntr_1,=pointer1+cntr_1,pointer2=p14+205
||                   [CU]           ps1=add(a9.h,sb0),ps1_2=add(a9.h,sb1);

                     // alp1 alp1_2
                     [AGU]          =p14+cntr_1,=pointer2+cntr_1,pointer3=p6+5
||                   [CU]           alp1=l_mac(a14,sb0,256),alp1_2=l_mac(a14,sb1,256);
                     [AGU]          =p6+cntr_1,=pointer3+cntr_1,pointer3=p7+5
||                   [CU]           alp1=l_mac(alp1,sb0,512),alp1_2=l_mac(alp1_2,sb1,512);
                     [AGU]          =p7+cntr_1,=pointer3+cntr_1,pointer3=p8+5
||                   [CU]           alp1=l_mac(alp1,sb0,512),alp1_2=l_mac(alp1_2,sb1,512);
                     [AGU]          =p8+cntr_1,=pointer3+cntr_1,pointer3=p9+5
||                   [CU]           alp1=l_mac(alp1,sb0,512),alp1_2=l_mac(alp1_2,sb1,512);
                     [AGU]          =p9+cntr_1,=pointer3+cntr_1,pointer3=p10+5
||                   [CU]           alp1=l_mac(alp1,sb0,512),alp1_2=l_mac(alp1_2,sb1,512);
                     [AGU]          =p10+cntr_1,=pointer3+cntr_1,pointer3=p11+5
||                   [CU]           alp1=l_mac(alp1,sb0,512),alp1_2=l_mac(alp1_2,sb1,512);
                     [AGU]          =p11+cntr_1,=pointer3+cntr_1,pointer3=p12+5
||                   [CU]           alp1=l_mac(alp1,sb0,512),alp1_2=l_mac(alp1_2,sb1,512);
                     [AGU]          =p12+cntr_1,=pointer3+cntr_1,pointer3=p13+5
||                   [CU]           alp1=l_mac(alp1,sb0,512),alp1_2=l_mac(alp1_2,sb1,512);
                     [AGU]          =p13+cntr_1,=pointer3+cntr_1
||                   [CU]           alp1=l_mac(alp1,sb0,512),alp1_2=l_mac(alp1_2,sb1,512), clear(sq2_2), clear(alp_16_2);
                     [PCU]          cntr_2=a6.h,,enable_loop(2,long)
||                   [CU]           testd(1,0,eq);                          // force le bit T a FALSE pour la premiere occurance
_search_contrib_i9_loop:
boucle 2             loopstart2
                     // ps2
                     [AGU]          =p4+cntr_2
||                   [CU]           ps2=add(ps1,sb0), (ift) alp=alp_16, (ift) sq=sq2,(ift) a7.h=compt2,(ift) a7.l=compt1,(ift) a9.l=ps2;
                     // sq2
                     [AGU]          pointer2=p14+200
||                   [CU]           sq2=mult(ps2,ps2), a1=l_mult(alp,sq2_2), tmp2=ps2_2;
                     // ps2_2 -- alp2
boucle 3             [AGU]          =p4+cntr_2,=p0+cntr_2
||                   [CU]           ps2_2=add(ps1_2,sb0), alp2=l_mac(alp1,sb1,4096),  a1=l_msu_with_test(a1,sq,alp_16_2) ;
                     // sq2_2 -- alp2
                     [AGU]          =p14+cntr_2
||                   [CU]           sq2_2=mult(ps2_2,ps2_2),alp2=l_mac(alp2,sb0,512), (ift) a9.l=tmp2, (ift)sq=sq2_2;
                     // alp16 -- alp2_2
                     [AGU]          =p0+cntr_2
||                   [CU]           alp2_2=l_mac(alp1_2,sb0,4096),alp_16=round(alp2),compt2=cntr_2,(ift) a7.h=compt2,(ift)alp=alp_16_2;
                     // tmp -- alp2_2
                     [AGU]          =pointer2+cntr_2
||                   [CU]           alp2_2=l_mac(alp2_2,sb0,512), a0=l_mult(alp,sq2), compt1=cntr_1, (ift) a7.l=add(compt1,5);
                     // s  -- alp16_2
                     [CU]           alp_16_2=round(alp2_2),a0=l_msu_with_test(a0,sq,alp_16);
                     loopend2

                     ift            [CU]           alp=alp_16,sq=sq2,a7.h=compt2,a7.l=compt1,a9.l=ps2;

                     [AGU]          p14+=400
||                   [CU]           a1=l_mult(alp,sq2_2);
                     [AGU]          pointer1=p4+5
||                   [CU]           a1=l_msu_with_test(a1,sq,alp_16_2);
                     ift            [CU]           a9.l=ps2_2,sq=sq2_2, alp=alp_16_2, a7.h=compt2, a7.l=add(compt1,5);

                     loopend1
```

Figure 150 Code optimisé boucles 2 et 3

Bibliographie

1 **FORWARD CONCEPT**, *DSP Market Bulletin 04/04/2001*, www.fwdconcepts.com

2 **BAJOT Y., MEHREZ H.,** *Les systèmes de traitement numérique du signal*, Rapport LIP6-ASIM, 2001.

3 **LAPSLEY P., BIER J., SHOHAM A., LEE E**., *DSP Processor Fundamentals, Architecture and Features*, IEEE Press

4 **SMITH S,** *The Scientist and Engineer's Guide To Digital Signal Processing*, Second Edition, California Technical Publishing, http://www.dspguide.com/

5 **HARFMAN G**., *Fixed-Point or Floating-Point DSPs ?*, RTC Europe, Vol. II, Number 4, April 1998

6 **YARLAGADDA K**., *The expanding world of DSPs*, Computer Design, Editorial, March 1998

7 **EYRE J., BIER J**., *DSP Processors Hit the Mainstream*, IEEE Computer, August 1998

8 **BERKELEY DESIGN TECHNOLOGY INC.**, *The evolution of DSP Processors*, lecture presented to U.C. Berkeley CS152, April 2000, http://www.bdti.com

9 **BEREKOVIC M., PIRSCH P.**, *An Algorithm-Hardware-System Approach to VLIW Multimedia Processors*, Journal of VLSI Signal Processing, 1998

10 **BILAS A., FRITTS J., SINGH J.P**., *Real-Time Parallel MPEG-2 Decoding in Software*, 11th International Parallel Processing Symposium, Geneva, Switzerland, April 1997

11 **BAGNORDI H**., *Available Instruction-Level Parallelism in Multimedia Applications*, Proc. of the ICSPAT 1997, San Diego, USA

12 **FRITTS J**., *Architecture and Compiler Design Issues in Programmable Media Processors*, PhD Thesis, Princeton University, June 2000

13 **BIER J**., *DSP16xxx Targets Communication Apps*, Microprocessor Report, September 1997

14 **BERKELEY DESIGN TECHNOLOGY INC.**, *Independent DSP Benchmark Results for the Latest Processors*, lecture presented at DSP World, April 2000, http://www.bdti.com

15 **OVADIA B., WERTHEIZER G**., *PalmDSPCore – Dual MAC and Parallel Modular Architecture*, proc. of the ICSPAT, November 1999

16 **TURLEY,** *DSP Screams at 1,600 MIPS*, Microprocessor Report, February 1997

17 **MOERMAN C.M.**, *Instruction Sets in DSP Architectures*, proc. of the ICSPAT99, Orlando FL, 1999

18 **SWEENEY J**., *Superscalar Techniques Applied to Digital Signal Processing*, proc. of the ICSPAT98, Toronto, Canada, 1998

19 **BERKELEY DESIGN TECHNOLOGY INC.**, *DSP Software Optimization Techniques for the Latest Processors*, presented at the Embedded Systems Conference (ESC), September 1999, www.bdti.com

20 **WOLF O., BIER J**., *TigerSHARC Sins Teeth Into VLIW*, Microprocessor Report, December 1998

21 **FRIDMAN J., GREENFIELD Z.**, *The TigerSHARC DSP Architecture*, IEEE Micro, Jan 2000

22 **GREHAN R., BIER J., EYRE J**., *Massana Unveils DSP Coprocessor Core*, Microprocessor Report, November 1999

23 **EYRE J., BIER J**., *Infineon's TriCore Tackles DSP*, Microprocessor Report, April 1999

24 **ROPERS A.**, *DSPstone : TI C54x*, Report IISPS Aachen University of Technology, 1999

25 **PAULIN P., LIEM C., CORNERO M., NACABAL F., GOOSSENS G.**, *Embedded Software in Real-Time Signal Processing Systems: Application and Architecture Trends*, Proc. of the IEEE, Vol 85, March 1997

26 **BERKELEY DESIGN TECHNOLOGY INC.**, *DSP Benchmark Results for the Latest VLIW-Based Processors*, lecture presented at the ICSPAT2000, October 2000, http://www.bdti.com

27 **SAGHIR M., CHOW P., LEE C**., *Application-Driven Design of DSP Architectures and Compilers*, Proc. of the ICASSP94, April 1994

28 **HALAHMI, RONEN S., MISHLOVSKY Y., NAOR A. & al.**, *GSM EFR Vocoder on StarCore 140*, proc. of the ICSPAT99, October 1999

29 **AHO A.V., SETHI R., ULLMAN J.D**., *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986

30 **FREDERIKSEN A., CHRISIANSEN R., BIER J., KOCK P**., *An Evaluation of Compiler-Processor Interaction for DSP Applications*, to appear in Proc. of the Asilomar Conference on Signals, Systems and Computers 2001

31 **BEACH S**., *The Advantages of a Scalable DSP Architecture*, proc. of the ICSPAT2000, October 2000

32 **DEHON A.**, *The Density Advantage of Configurable Computing*, IEEE Computer, Avril 2000

33 **FISHER J., FARABOSCHI P**., *Custom-Fit Processors: Letting Applications Define Architectures*, proc. of the 30th Annual International Symposium on Micro-architecture, Paris, France, December 1996

34 **LEE C., KIN J., POTKONJAK M., GIONE-SMITH W**., *Media Architecture: General Purpose vs. Multiple Application-Specific Programmable Processor*, proc. of the 35th Design Automation Conference, San Francisco, June 1998

35 **FARABOSHI P., BROWN G., FISHER J.& al**, *Lx: A Technology Platform for Customizable VLIW Embedded Processing*, ISCA2000, Vancouver, Canada

36 **SANKARAN J**., *Efficient Implementation of Reed Solomon Decoding on the TMS320C64x DSP*, proc. of the ICSPAT2000, Dallas, USA, October 2000

37 **LUCENT TECHNOLOGIES**, *DSP16210 Datasheet*, www.lucent.com/micro/dsp16000/dsp16000_doc.html

38 **LOU H**., *Implementing the Viterbi Algorithm*, IEEE Signal processing Magazine, September 1995

39 **BEREKOVIC M., STOLBERG H., KULACZEWSKI M., PIRSCH P**., *Instruction Set Extensions for MPEG-4 Video*, Journal of VLSI Signal Processing 23, 27-49 (1999), Kluwer Academic Publishers

40 **SENTIEYS O**., *Estimation et réduction de l'énergie en conception CMOS numérique*, ENSSAT, Université de Rennes I, http://archi.enssat.fr

41 **ACKLAND B., NICOL C.**, *High Performance DSPs – What's Hot and What's Not?*, ISLPED98, August 1998, Monterey, CA, USA

42 **LEKASTAS H., HENKEL J., WOLF W**., *Code Compression for Low Power Embedded System Design*, proc. of the DAC2000, Los Angeles, USA

43 **GONZALES R**., *Xtensa : A Configurable and Extensible Processor*, IEEE Micro, March/April 2000

44 **RAIK-ALLEN G.**, *ARC Cores rides platform divergence*, Red Herring, June 1999, http://www.redherring.com/insider/1999/0604/vcarcores.html

45 **KIEVITS P., VERMEIRE F., WOUDSMA R.**, *Next Generation R.E.A.L DSP for consumer applications*, proc. of the ICSPAT2000, Dallas USA, October 2000

46 **EYRE J., BIER J.**, *Carmel Enables Customizable DSP*, Microprocessor Report, December 1998

47 **YANG J-H. & al.**, *Metacore: An Application-Specific Programmable DSP Development System*, IEEE Transactions on VLSI, April 2000

48 **TULAI A. & al**, *MILIWAT – a core for DSP programmers*, proc. of the ICSPAT1999, Orlando, USA, November 1999

49 **KUULUSA M & al.**, *A Flexible DSP Core for Embedded Systems*, IEEE Design and Test of Computers, October 1997

50 **INFINEON INC.**, *CARMEL 20xx PowerPoint Presentation*, http://www.carmeldsp.com/products/product_overview.html

51 **FARABOSCHI P., BROWN G., FISHER J., DESOLI G., HOMEWOOD F.**, *Lx: A Technology Platform for Customizable VLIW Embedded Processing*, proc. of the ISCA-2000, Vancouver Canada, June 2000

52 **PRIEBE R., USSERY C.**, *A Configurable Platform for Advanced System-On-Chip Applications*, ICSPAT 2000, Dallas, TX, October 2000

53 **PAULIN P., LIEM C., CORNERO M., NACABAL F., GOOSSENS G.**, *Embedded Software in Real-Time Signal Processing Systems: Design Technoogies*, Proc. of the IEEE, Vol 85, March 1997

54 **INFINEON TECHNOLOGIES INC.**, *CARMEL DSP Core Technical Overview Handbook*, *DSP Benchmark*, , http://www.carmeldsp.com

55 **HENNESSY J., PATTERSON D.**, *Architecture des Ordinateurs, une approche quantitative*, Deuxième édition, traduction de Daniel Etiemble, Thomson Publishing

56 **BAUER H., LORENZ D., PETERSEN J.**, *Application of R.E.A.L. DSP Core in GSM Mobile Phones*, proc. of the ICSPAT1999, Orlando USA, October 1999

57 **GSM 06.60 (EN 300 726)**: *Enhanced Full Rate (EFR) speech transcoding*, ETSI, 1998

58 **GSM 06.53 (EN 300 726)**: *ANSI-C code for the GSM EFR speech codec*, ETSI, 1998

59 **GSM 06.54 (EN 300 726)**: *Test sequences for the GSM EFR speech codec*, ETSI, 1998

60 **FAUTH A., VAN PRAET J., FREERICKS M.**, *Describing Instruction Set Processors using nML*, Proc. of the European Design and Test Conference, Mars 1995

61 **MARWEDEL P. GOOSSENS G.**, *Code Generation for Embedded Processors*, Kluwer Academics Publishers, 1995

62 **HADJIYIANNIS G., HANONO S., DEVADAS S.**, *ISDL: An Instruction Set Description Language for Retargetability*, Proc. of the Design Automation Conference, 1997

63 **HANONO S., DEVADAS S.**, *Instruction Selection, Ressource Allocation and Scheduling in the AVIV Retagetable Code Generator*, Proc. of the Design Automation Conference, 1998

64 **PEES S., HOFFMANN A., ZIVOJNOVI'C V., MEYR V.,** *LISA: Machine description language for cycle-accurate models of programmable DSP architectures*. In Proc. of 35th Design Automation Conference, 1999.

65 **MESSE V**., *Production de compilateurs flexibles pour la conception de processeurs programmables spécialisés*, Thèse de doctorat, Université de Rennes I, Mars 1999

66 **WILSON R., FRENCH R., WILSON C., AMASINGHE S. & al**, *SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers*, Rapport technique, Stanford University, Mai 1994

67 **DE ARAUJO G**., *Code Generation Algorithms for Digital Signal Processors*, Thèse de PhD, Princeton University, 1997

68 **SUDARSANAM A**., *Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors*, Thèse de PhD, Princeton University Department of EE, 1998

69 **TJIANG S**., *An Olive Twig*, Rapport Technique, Synopsis Inc., 1993

70 **SPAM RESARCH GROUP**, *SPAM Compiler User's Manual*, http://www.ee.Princeton.edu/spam, 1997

71 **BATTEN D., JINTURKAR S., GLOSSNER J., SCHULTE M., & al**., *Interactions between optimizations and a new type of DSP intrinsic function*, proc. of the ICSPAT1999, Orlando, 1999

72 *SPAM Compiler Release*, http://www.ee.princeton.edu/~spam/release.html

73 **VAUCHER N**., *Méthodologie de conception d'architectures VLSI génériques appliquées au traitement numérique*, Thèse de doctorat, Université Paris VI, Octobre 1997

74 **HOUELLE A.**, *GenOptim : Un environnement d'aide à la conception de générateurs de circuits portables optimisés en performance et en surface*, Thèse de doctorat, Université Paris VI, Octobre 1997

75 **ABERBOUR M.**, *Architecture d'un système hétérogène pour la reconnaissance de formes*, Thèse de doctorat, Université Paris VI, Septembre 1999

76 **CHOTIN R., DUMONTEIX Y., MEHREZ H.***, Use of redundant arithmetic on architec-ture and design of a hight performance dct macro-bloc generator*, In Proc. Design of Circuits and Integrated Systems, november 2000

77 **TEXAS INSTRUMENTS**, *TMS320C62x/C67x Programmer's Guide*, http://www.ti.com/sc/docs/psheets/man_dsp.htm

78 **MINGARDON S., NIGGEBAUM R.**, *Implementation of the Enhanced Full Rate vocoder on the CARMEL DSP Core*, proc. of the ICSPAT, Orlando, November 1999

79 **INFINEON INC.**, *CARMEL DSP Application Library, The Viterbi Equalizer*, http://www.carmeldsp.com

80 **INFINEON INC**., *Implementation of Viterbi Algorithm on CARMEL DSP*, http://www.carmeldsp.com

81 **DU J., FALKOWSKI J., AZIZ A., LANE J**., *Implementation of Viterbi Decoding on StarCore SC140 DSP*, proc. of the ICSPAT, Dallas, October 2000

82 **MOTOROLA**, *SC140 DSP Core Reference Manual*, www.starcore-dsp.com

83 **A. GREINER AND F. PECHEUX**, *A complete set of CAD Tools for teaching VLSI*, Third EuroChip Workshop on VLSI Design Training, September 1992

84 **JOHAN JANSSEN, HENK CORPORAAL**, *Partitioned Register File for TTAs*, MICRO-28, Michigan, November 1995

85 **BEKOOIJ M., BINK A., HOOGERBRUGGE J. & al.**, *COCOON: Core and Compiler Codesign for Embedded DSP***,** proc. of the ICSPAT, Dallas, October 2000