Méthodologie de conception système à base de plateformes reconfigurables et programmables

Khemaies Ghali

N° D'ORDRE :

UNIVERSITE PARIS XI
UFR SCIENTIFIQUE D'ORSAY

THESE

Présentée

Pour obtenir

Le GRADE de DOCTEUR EN SCIENCES
DE L'UNIVERSITE PARIS XI ORSAY

PAR

M. Khemaies GHALI

Sujet :

METHODOLOGIE DE CONCEPTION SYSTEME A BASE DE PLATEFORMES RECONFIGURABLES ET PROGRAMMABLES

Soutenue le 01 Mars 2005 devant la Commission d'examen

M. El Mostapha ABOULHAMID	**Rapporteur**
M. Xavier GANDIBLEUX	**Rapporteur**
M. Dominique HOUZET	**Examinateur**
M. Omar HAMMAMI	**Co-directeur**
M. Alain MERIGOT	**Directeur**

AVANT-PROPOS

Le travail présenté dans cette thèse a été réalisé au sein du Laboratoire Electronique et Informatique (LEI) de l'Ecole Nationale Supérieure de Techniques Avancées (ENSTA de Paris). Je remercie Monsieur Alain Sibille, Responsable du laboratoire LEI à l'ENSTA de m'avoir accueilli et donné les moyens pour effectuer mon travail de recherche.

Je tiens tout particulièrement à exprimer ma profonde gratitude à Monsieur Omar Hammami, enseignant chercheur à l'Ecole Nationale Supérieure de Techniques Avancées de Paris, pour son encadrement et son suivi. Son soutien, ses conseils, ses encouragements permanents tout au long de ces années de recherche, son dynamisme et sa disponibilité ont joué un rôle déterminant dans ce travail. Qu'il trouve ici, l'expression de ma profonde reconnaissance.

Je remercie très vivement Monsieur Alain Mérigot, Professeur à l'Institut d'Electronique Fondamental (IEF) du l'Université de Paris XI, d'avoir accepté de diriger ma thèse, et pour avoir supervisé mes travaux. Je tiens à lui exprimer ma gratitude pour ses conseils, ses orientations et ses remarques pertinentes.

Je remercie Monsieur El Mostapha Aboulhamid, Professeur à l'université de Montréal et Monsieur Xavier Gandibleux, Professeur à l'université de Nantes, rapporteurs de cette thèse.

Je remercie également Monsieur Dominique Houzet pour avoir accepté de participer au jury.

Merci à tous mes collègues du laboratoire LEI de l'ENSTA et particulièrement ceux du groupe AVA, notamment Ivan Hermann et Laurent Dorie. Qu'ils trouvent ici l'expression de ma reconnaissance pour leur témoignage de sympathie et d'amitié.

Enfin merci à ma famille et mes amis, pour m'avoir accompagné au cours de ces dernières années.

SOMMAIRE

LISTE DES FIGURES

LISTE DES TABLEAUX

RESUME

Les travaux présentés dans ce mémoire concernent l'exploration de l'espace de conception des architectures SOC pour des applications orientées télécommunication. L'évolution importante des semi-conducteurs a permis l'implémentation de systèmes complets sur une puce. Cette implémentation a été rendue possible par des méthodologies de conception basées sur la réutilisation des composants existants (*IP – Intellectual Property*) qui, combinées ensemble, constituent le système. La différentiation des systèmes est obtenue par l'ajout d'IP propriétaires rattachées au système. L'apport des technologies classiques basées sur le modèle en Y (Y-chart) et les techniques de co-design se sont avérées insuffisantes dès lors que ces IPs initialement sous forme dure (hard IP) donc non modifiables ont étés proposées dans leur version paramétrable (Soft IP), pour garantir un meilleur dimensionnement du système. En effet, la modularité des IPs soft par leurs paramétrisations, créent un espace d'exploration qui s'avère extrêmement important et donc inexploitable par des techniques de conception ad hoc ou interactives.

Le problème posé est l'optimisation mathématique des paramètres de l'ensemble des IPs soft constituant le SOC. Ce problème multidimensionnel en performance est aggravé, dans le cadre des SOC pour systèmes embarqués, par la prise en compte de la consommation d'énergie et de la surface en silicium. Le problème devient alors une optimisation multiobjectifs. Cette thèse propose une résolution de ce problème en plusieurs étapes :

Dans une première étape, des techniques d'exploration pour le dimensionnement d'IP de processeur SuperScalair sont proposées. Ces techniques tiennent compte de trois critères : performance, consommation d'énergie et surface en silicium. Les résultats obtenus par des benchmarks multimédia « MiBench » de taille significative résultent dans un sous ensemble optimal au sens de Pareto, permettant de sélectionner une ou plusieurs solutions efficaces pour les applications cibles.

La seconde étape est une extension du cadre précédent par couplage de l'exploration multiobjectifs avec une implémentation matérielle sur circuits FPGA. Elle permet alors une exploration avec matériel dans la boucle. Le principe poursuivi, à l'inverse des explorations effectuées à des niveaux d'abstraction élevés (SystemC), est qu'une exploration est d'autant plus efficace que les valeurs injectées à l'algorithme d'exploration sont proches de la réalité. L'autre aspect est que l'exploration par simulation des SOC reste problématique, ceci étant dû aux temps prohibitifs de la simulation et que l'exécution directe est toujours plus rapide, donc permet des explorations larges et réalistes. Cette approche est appliquée au processeur LEON v2.0 de l' ESA sur des circuits Xilinx Virtex-II qui, de par leur reconfigurabilité, permet le chargement de nouvelles configurations lors de l'exploration.

Enfin, l'importance des SOC mixtes analogiques/numériques, nous a poussés à nous intéresser à l'optimisation des circuits analogiques et ce, sur le même principe, mais en utilisant des circuits FPAA (*Field Programmable Analog Array*) qui permettent la conception et l'implémentation d'applications sur circuits analogiques re-programmables. Cette possibilité permet de répondre à une fonctionnalité donnée en testant et explorant de nombreuses configurations, en les implémentant physiquement dans un circuit programmable et cela à moindre coût.

La thèse conclut sur les perspectives pouvant découler des contributions de ce travail sur les méthodologies de conception de SOC dans les environnements SOPC.

Mots clés : SOC, SOPC, FPGA, FPAA, algorithme génétique, optimisation multiobjectifs, algorithme NSGA-II, exploration de l'espace de conception, estimation des performances, UMTS, WCDMA, Turbo code.

ABSTRACT

In this document we present a design space exploration methodology for SOC architectures in telecommunication domain. The significant evolution of semiconductors technology has allowed the implementation of complete systems on a single chip. This implementation was made possible by the design methodologies based on the re-use of existing (IP – Intellectual Property) components in the system. Differentiation of the systems being obtained by the addition of IP owners attached to the system. The traditional technologies based on the Y (Y-charts) and the techniques of Co-design proved to be insufficient as they used non-parameterizable Hard IPs for the system. So, for better dimensioning of the system, Soft IPs were proposed which are parameterizable by nature and hence create a huge design exploration space proving extremely useful and not exploitable by ad hoc technique or interactive design. The problem arising is a mathematical optimization problem of the parameters of the whole of IPs software constituting the SOC. This multidimensional problem in performance is worsened when within the framework of SOC for embedded systems of the severe criteria of energy consumption and silicon surface area are also of equal importance. So the problem becomes a multidimensional problem of multi-objective optimization.

This thesis contributes to the resolution of this problem proposing a solution consisting of several stages. In a first stage, the techniques of exploration for the dimensioning of SuperScalar processor IPs are proposed which take account of the three criteria: performance, consumption of energy and silicon surface area. The obtained results on multi-media benchmarks "MiBench" resulted in an optimal subset consisting of Pareto function making it possible to select one or more of the effective solutions for the selected applications. This first stage being realized in the thesis also proposes a second contribution which extends the preceding framework by coupling multi-objective exploration with a physical implementation on FPGA circuits allowing an exploration with physical hardware in the loop. The principle followed is the reverse of explorations carried out has high levels of abstraction (SystemC) is that an exploration is all the more effective since the values injected with the algorithm of exploration are close to reality. The other aspect is that exploration by simulation of the SOC remains problematic due to prohibitory times of simulation and that the direct execution is increasingly faster thus allows broad and realistic explorations. This approach is applied to LEON processor v2.0 "ESA" to Xilinx circuits Virtex-II which from their reconfigurability allow the loading of new configurations during exploration. Lastly, the importance of the mixed analog-digital SOC raised our interest to devise an optimization methodology for the analog circuits based primarily on the same principle. Only difference in this methodology was the usage of FPAA circuits (*Field Programmable Analog Array*) which allows the design and the implementation of applications on reprogrammable analog circuits. This methodology makes it possible to test and explore many configurations by physically implementing them in a programmable circuit at a lower cost.

The thesis concludes with a prospective note on possible future research directions appearing from the contributions of this work on methodologies of SOC design in SOPC environments.

Key-Words: SOC, SOPC, FPGA, FPAA, Reconfigurable Architectures, Genetic Algorithm, Multi-objective Optimization, Design Space Exploration, Performances Estimation, UMTS, WCDMA, Turbo coder.

CHAPITRE 1

CONTEXTE GENERAL DE LA THESE ET MOTIVATIONS

Cette thèse participe aux travaux de recherche du groupe AVA (Algorithmique de Vision et Architectures) au sein du laboratoire LEI (Laboratoire d'Electronique et Informatique) à l'ENSTA (Ecole Nationale Supérieure de Techniques Avancées de Paris) dans le cadre de développement et conception de systèmes embarqués (SOCs).

Plus précisément, ces travaux sur la méthodologie de conception de systèmes embarqués sont motivés par l'évolution technologique et la généralisation du multimédia. L'arrivée de la vidéo sur les terminaux mobiles ne relève plus désormais de la fiction. Les applications multimédias impliquent une grande puissance de calcul dont l'implantation sur un terminal mobile reste un challenge du point de vue contrainte de la consommation d'énergie et celle de la disponibilité des ressources de traitement embarquées dans l'objet mobile.

Aujourd'hui, le choix d'une architecture pour le traitement multimédia embarqué n'est pas trivial et qu'un compromis doit certainement être trouvé entre flexibilité, consommation, performance, coût et la rapidité de conception en terme TTM (Time-To-Market).
Face à cette problématique, nous proposons une méthodologie pour les choix des paramètres de l'architecture basée sur des algorithmes génétiques multiobjectifs.

1.1 Conception des systèmes embarqués

Ce travail s'inscrit dans le contexte de la conception des systèmes embarqués monopuces. Les systèmes monopuces contiendront des réseaux formés de plusieurs processeurs dans le cas d'applications telles que les terminaux mobiles, les processeurs de jeux et les processeurs de réseau. De plus, ces puces contiendront des éléments non digitaux (parti analogique ou RF) et des mécanismes de communication très sophistiqués. Etant donné que tous ces systèmes correspondent à des marchés de masse, ils ont tous été intégrés sur une seule puce afin de réduire les coûts de production. Il est prévu que ces systèmes soient les principaux vecteurs d'orientation de toute l'industrie des semi-conducteurs. Il est donc crucial de maîtriser la conception de tels systèmes tout en respectant les contraintes de mise sur le marché et les objectifs de qualité. Cette architecture de SOC (Figure1.1) est composée de plusieurs couches en vue de maîtriser la complexité.

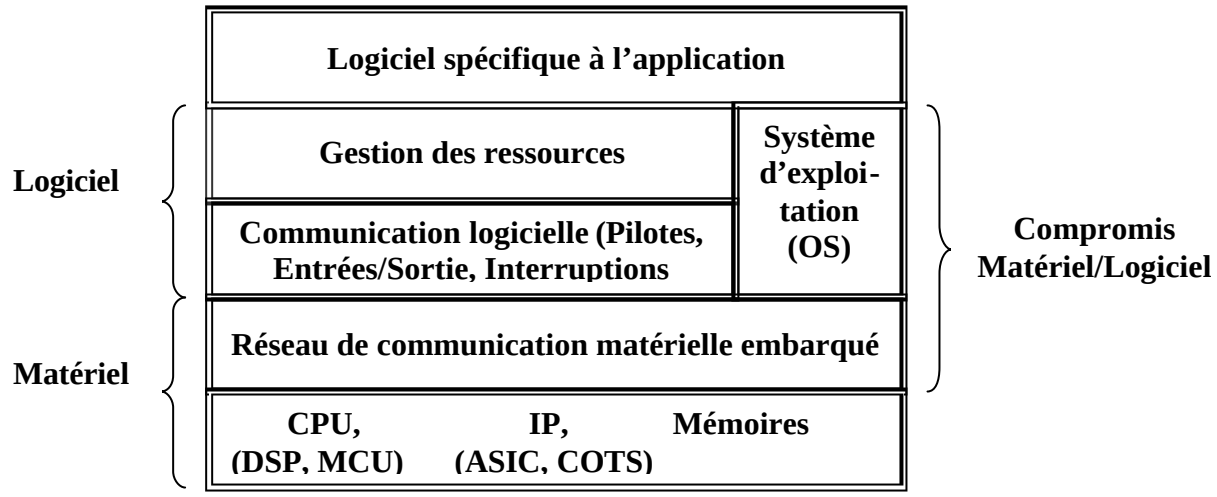


Figure 1.1 - Représentation d'un système embarqué sous la forme des couches

La partie matérielle est composée de deux couches :

- La couche basse contient les principaux composants utilisés par le système. Il s'agit de composants de base tel que des processeurs (DSP, MCU, IP, mémoires).
- La couche de communication matérielle embarquée sur la puce : il s'agit des dispositifs nécessaires à l'interaction entre les composants. Cette couche peut contenir un réseau de communication complexe allant du simple pont (bridge) entre deux processeurs au réseau de communication de paquets. Bien que le réseau de communication lui-même soit composé d'éléments standard, il est souvent nécessaire d'ajouter des couches d'adaptation entre le réseau de communication et les composants de la première couche.

Le logiciel embarqué est aussi découpé en couches :

- La couche basse contient les pilotes d'entrées/sorties et d'autres contrôleurs de bas niveau permettant de contrôler le matériel. Le code correspondant à cette couche est fortement lié au matériel. Cette couche permet d'isoler le matériel du reste du logiciel.
- La couche de gestion de ressources permet d'adapter l'application à l'architecture. Elle fournit les fonctions utilitaires indispensables pour le bon fonctionnement de l'application. La couche de gestion de ressources est une interface entre l'application software et la partie architecture hardware. Bien que la plupart des systèmes d'exploitation (OS) fournissent une telle interface, il sera souvent utile d'en réaliser une spécifique à l'application pour respecter des contraintes de temps réel, taille de la mémoire nécessaire et/ou performances. En effet, les applications embarqués utilisent souvent des structures de données particulières avec des accès mémoire non standard (manipulation de champs de bits ou parcours rapides de tableaux) qui ne seront généralement pas fournis par les OS standard. D'autre part, les couches de gestion de ressources fournies par les OS standard sont généralement trop volumineuses pour être embarqués.

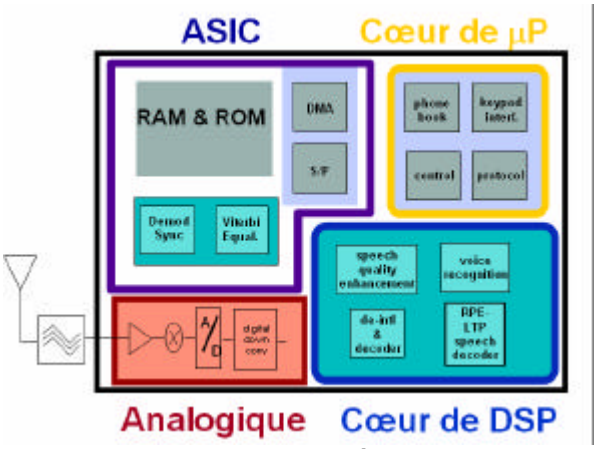


Figure 1.2 - Exemple d'un système sur puce (SOC)

La Figure 1.2 montre un exemple d'un système embarqué, cette architecture est typiquement utilisée par les objets de communication mobile (téléphone mobile, PDA, etc.). Le système est composé principalement de quatre blocs :

- Cœur de µP : Protocole et contrôle, Interface utilisateur
- Cœur de DSP : Calculs lents, Flexibilité
- ASIC : Accélérateurs, Architecture mémoire
- Analogique : A/D, RF, modulation

1.2 Contribution

Aujourd'hui, en conception de système sur puce, l'apport des méthodologies classiques basées sur le modèle en Y « Y-chart » (Figure 1.3) et les techniques de co-design se sont avérées insuffisantes. Dès lors que ces IPs initialement sous forme dures (hard IP), donc non modifiables, ont étés proposées dans leur versions paramétrable (Soft IP) et ce pour un meilleur dimensionnement du système. En effet les softs IPs, par leurs paramètrisations créent un espace d'exploration qui peut s'avérer extrêmement important et donc inexploitable par des technique de conception interactives.

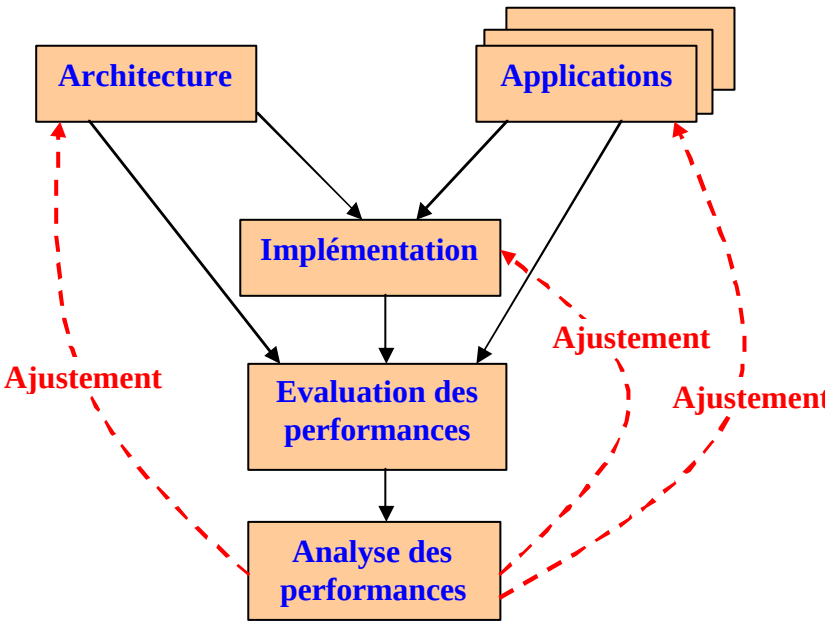


Figure 1.3 – Modèle en Y (Y-chart)

La contribution générale de ce travail est la proposition d'une méthodologie pour l'automatisation du flot de conception suivant le modèle en Y (Figure 1.3). Par l'exploration de l'espace des solutions d'une architecture système sur puce (SOC pour *System On Chip*), pour localiser les configurations optimales par :

- Adaptation et utilisation des algorithmes d'optimisations multiobjectifs pour l'exploration d'espace des solutions pour des architectures SOC
- Exploration architecture et optimisation des systèmes sur puces à base de la simulation
- Exploration et optimisation des systèmes sur puces, à base d'exécution directe, par une implémentation sur FPGA
- Exploration et optimisation multiobjectifs des circuits analogiques à travers des implémentations sur FPAA

Ces travaux intéressent l'exploration de l'espace de conception des architectures SOC pour des applications orientées télécommunication. D'où notre intérêt pour la chaîne de transmission WCDMA employée par la norme UMTS pour la téléphonie mobile de 3ème génération. Dans ce cadre nous avons développé, un turbo coder [54], entièrement en langage C, pour l'utiliser comme une application logicielle cible à notre architecture matérielle à paramétrer.

1.3 Plan du mémoire

La suite de ce mémoire est organisée de la façon suivante :

Dans le second chapitre, nous présentons les deux technologies disponibles sur le marché pour la conception de SOC. Ensuite, nous abordons une étude détaillée de l'état de l'art sur ces architectures ASIC et FPGA et leurs flots de conception.

Le chapitre 3 présente les différentes méthodologies d'optimisation et d'exploration de l'espace d'architecture. En particulier la mise en œuvre des algorithmes évolutionnaires (AE) et leurs aptitudes à attaquer des problèmes d'optimisation mono et multiobjectifs difficiles à résoudre autrement. Nous présenterons en détail l'algorithme NSGA-II et leur efficacité en optimisation multiobjectifs en terme d'optimale de Pareto.

Au cours du quatrième chapitre, nous présentons deux architectures à base de microprocesseur paramétrable à explorer. Le principe de base est l'application de l'algorithme génétique multiobjectifs pour le choix des paramètres de chaque architecture à travers de l'espace d'exploration architectural guidé par une optimisation simultanée de plusieurs grandeurs :

- Cas 1 : Paramétrage d'une architecture à base d'un processeur de type Supercalar : C'est une optimisation en terme de temps d'exécution, surface de silicium et consommation d'énergie. Dans ce cas, l'environnement d'évaluation des performances est le simulateur SimpleScalar.
- Cas 2 : Paramétrage d'une architecture à base d'un processeur IP en VHDL synthétisable à travers des implémentations sur FPGA. C'est une évaluation des performances (temps d'exécution en nombre des cycles) et des ressources (taux d'utilisation des ressources FPGA en terme des BRAMs et slices) par exécution directe sur FPGA.

Dans le cinquième chapitre, nous étudierons en détail la chaîne de transmission WCDMA adoptée par la norme 3GPP pour le téléphone mobile de 3ème génération UMTS. Notre but dans ce chapitre est d'écrire le turbo code en langage C afin de l'utiliser comme une application cible pour l'architecture à base de processeur SuperScalar à paramétrer.

Dans l'avant-dernier chapitre, nous appliquons la méthodologie d'optimisation multiobjectifs, pour l'exploration des circuits analogiques, à travers des implémentations sur une puce programmable dédiée au domaine analogique le FPAA.

Enfin, nous concluons ce document en rappelant les principaux résultats et en ouvrant des nouvelles perspectives d'études.

CHAPITRE 2

SYSTEME SUR PUCE

Un système monopuce, appelé encore SOC (*System-on-Chip*) ou système sur puce, désigne l'intégration d'un système complet sur une seule pièce de silicium. Ce terme est devenu très populaire dans le milieu industriel malgré l'absence d'une définition standard [1]. Certains considèrent qu'un circuit complexe en fait automatiquement un SOC, mais cela inclurait probablement chaque circuit existant aujourd'hui. Une définition plus appropriée de système sur puce serait : « un système complet sur une seule pièce de silicium, résultant de la cohabitation sur silicium de nombreuses fonctions déjà complexes en elles-mêmes : processeur, DSP, mémoires, bus, convertisseurs, blocs analogiques, etc. il doit comporter au minimum une unité de traitement de logiciel (un CPU) et ne doit dépendre d'aucun (ou de très peu) des composants externes pour exécuter sa tâche. En conséquence, il nécessite à la fois du matériel et du logiciel ».

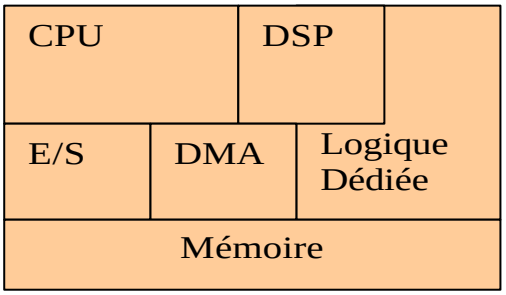


Figure 2.1 - Exemple de système sur puce (SOC)

La Figure 2.1 présente un exemple de système sur puce typique. Il se compose d'un cœur de processeur (CPU), d'un processeur de signal numérique (DSP), de la mémoire embarquée, et de quelques périphériques tels qu'un DMA et un contrôleur d'E/S. Le CPU peut exécuter plusieurs tâches via l'intégration d'un OS. Le DSP est habituellement responsable de décharger le CPU en faisant le calcul numérique sur les signaux de provenance du convertisseur A/N.

Il y a plusieurs raisons pour lesquelles la solution monopuce représente une manière attractive pour implémenter un système. Les processus de fabrication d'aujourd'hui (de plus en plus fins) permettent de combiner la logique et la mémoire sur une seule puce, réduisant ainsi le temps global des accès mémoire. Etant donné que le besoin en mémoire de l'application ne dépasse pas la taille de la mémoire embarquée sur la puce, la latence de mémoire sera réduite grâce à l'élimination du trafic de données entre des puces séparées. Le nombre de broches peut également être réduit. Ces caractéristiques aussi bien que la faible consommation et la courte durée de conception (par rapport à un système-sur-carte) permettent une mise sur le marché, rapidement, de produits plus économiques et plus performants.

2.1 Technologie ASIC, Evolution et Projection

La complexité du silicium se rapporte à l'impact de la graduation de processus, et à l'introduction de nouveaux matériaux ou d'architectures. Bon nombre de phénomènes précédemment ignorés, présentent aujourd'hui un fort impact sur l'exactitude de la conception [2]:

- graduation non idéale de dispositif et seuil de la tension d'alimentation (fuite, gestion de la puissance, innovation de circuit/dispositif, courant fourni)
- les dispositifs à haute fréquence couplés et reliés ensemble (le bruit/interférence, l'analyse et la gestion d'intégrité de signal, accouplement de substrat, retarde la variation due à l'interconnexion)
- manque de stabilité de la fabrication (modélisation statistique de processus, rendement, fuite de la puissance)
- dégradation de la fiabilité (perçage des isolateurs, dissipation de la chaleur, défaut tolérance générale)
- variabilité des processus (caractérisation de la bibliothèque, plateformes analogique et numérique, tolérance d'erreur dans la conception, de réutilisation de design, fiabilité, implémentations prévisibles)

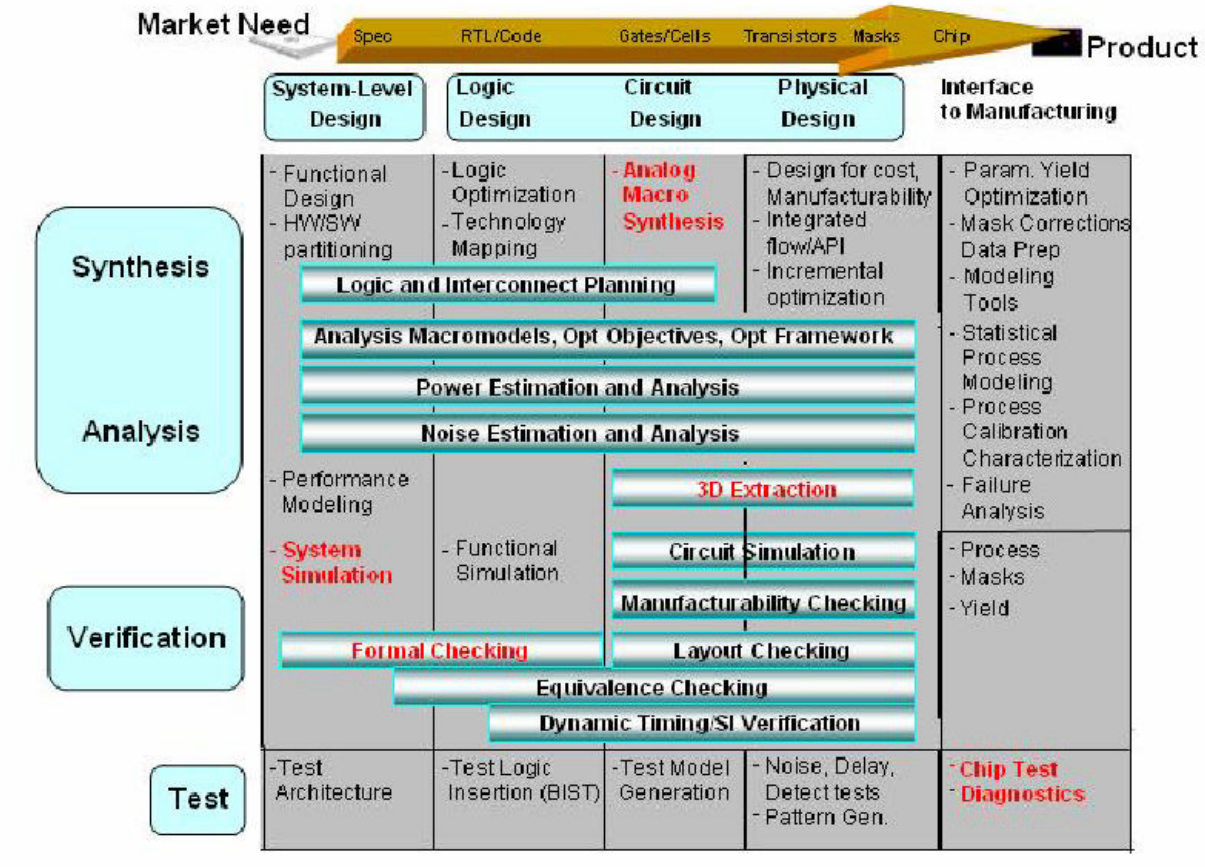


Figure 2.2 - Technologie ASIC [2]

La complexité d'un système sur puce, dépend de l'augmentation exponentielle (suivant la loi de Moore) du nombre des transistors par unité de surface de silicium, stimulés par l'accrue de la demande des consommateurs à des fonctionnalités plus performantes, du

coût plus faible et du temps de conception TTM (*Time-To-Market*) plus court, pour assurer la rentabilité du produit.
Enfin, il y a d'autres complexités additionnelles (tel que environnement système et/ou hétérogénéité du composant), dans la phase de l'intégration du système sur puce. Les spécifications et la validation de la conception deviennent extrêmement difficiles, en particulier en ce qui concerne des contextes de fonctionnement complexe. Des compensations doivent être faites entre tous les aspects de quantité ou de qualité, et tous les aspects de coût :

- réutilisation : supporte la conception hiérarchique, l'intégration hétérogène de SOC tel que la modélisation, simulation, vérification et test des blocs de composant.
- vérification et test : capture des spécifications, réutilisation de vérification pour le SOC hétérogène, vérification au niveau système et de logiciel, vérification pour l'analog/mixed-signal et nouveaux systèmes, auto-test, réutilisation d'essai.
- optimisation coût de cycle de conception : coût de la fabrication en terme de modélisation et de l'analyse, optimisation du système en ce qui concerne des objectifs multiples, testabilité, etc.
- conception des logiciels embarqués : emploi des plateformes adoptées à la méthodologie de conception, logiciel pour vérification/analyse de fonctionnement.
- contrôle de processus industriel de conception : taille de l'équipe de la conception et distribution géographique, gestion des données, collaboration en conception.

2.2 Technologie FPGA, Evolution et Projections

2.2.1 Introduction

Le marché de la conception d'application à base de FPGA intéresse de plus en plus les éditeurs traditionnels de CAO. Jusqu'à une époque récente, la conception d'un système sur puce (SOC) n'était accessible qu'aux sociétés de grande taille, du fait des multiples exigences qu'elle induisait (complexité et diversité des compétences requises, coûts élevés des outils de développement matériel et logiciel, et forts volumes nécessaires pour justifier le coût de conception d'un ASIC). Aujourd'hui, deux événements ont changé la donne : l'avènement des dernières générations de FPGA et la disponibilité des blocs de propriété intellectuelle (IP), utilisable notamment sur ces matrices programmables, mettent la technologie SOC à la portée d'un public nettement plus large. Ainsi, cette technologie, qui ciblait principalement les domaines : public, médical, télécommunication et automobile, devient intéressante pour des applications plus simples et portant sur des volumes moins importants. Et ceci particulièrement depuis que les FPGA sont proposés à un prix très faible (0.0001dollar par porte). De plus, l'offre en propriété intellectuelle pour ces circuits, à présent est très variée et performante. Mentionnons simplement la richesse des cœurs des processeurs : des microcontrôleurs 8 et 16 bits aux microprocesseurs et DSP 32 bits de haute performance, grâce notament à l'architecture VLIW (*Very Large Instruction Word*).

Pour mettre en œuvre ces applications SOC, les concepteurs doivent disposer d'outils de CAO électronique et de développement de logiciels enfouis. Dans les PME, les premiers systèmes sur puce remplacent généralement des produits existants réalisés à l'aide de plusieurs composants (microcontrôleur, périphériques, etc.). Afin de mener à bien leurs projets, ces sociétés ont besoin d'un jeu de blocs IP fonctionnant ensemble et d'un fournisseur capable de garantir cette interopérabilité, et non d'une méthodologie générique de conception SOC. Les cœurs IPs et les périphériques choisis pour ces systèmes sont souvent des copies de composants discrets associés à des outils de développement logiciel, de haute qualité et de prix modique, mais proposés par des firmes différentes.

2.2.2 Limites de FPGA

L'emploi de FPGA est devenu, à juste titre, une solution privilégiée pour produire rapidement des circuits. Pourtant, la complexité croissante des plus grandes matrices engendre de graves problèmes à l'étape de routage. Entre les ASIC et les FPGA de haute complexité, la durée du procédé de placement-routage sous contrainte temporelle varie beaucoup avant de converger vers la solution optimale (Figure 2.3).

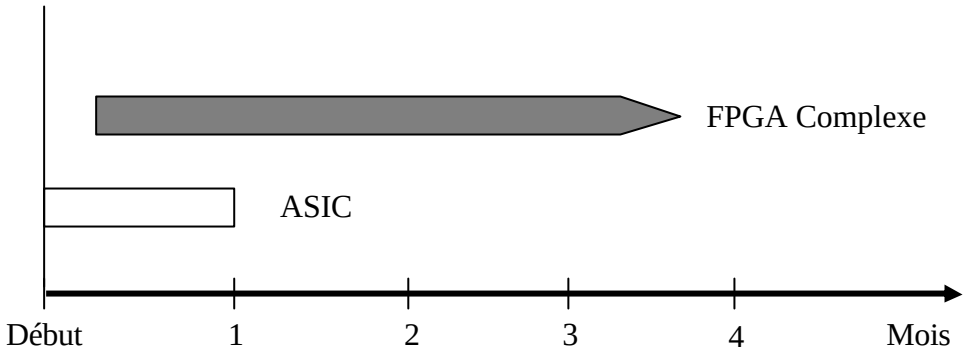


Figure 2.3 - Comparaison des temps de routage entre circuits configurables

Les FPGA ont certes de nombreux atouts, mais les concepteurs de systèmes ont tendance à se tourner, par habitude, vers cette solution sans regarder si elle répond correctement aux objectifs de performances et de coût, voire si elle s'inscrit bien dans le calendrier de développement de leurs projets. Pourtant, les FPGA de forte complexité induisent des limitations sévères qui peuvent pénaliser le bon déroulement d'une conception. Certaines applications, dans le domaine des réseaux et communications notamment, exigent des fonctions complexes et un nombre élevé de broches, ceci assorti de performances toujours croissantes. De ce fait, les plus grands réseaux logiques programmables choisis pour réaliser ces projets sont fréquemment employés au-delà de leurs possibilités physiques, des limitations inhérentes à leurs architectures. Avec de tels FPGA, il faut souvent plusieurs mois avant de converger vers un routage satisfaisant au niveau des performances temporelles (Figure 2.3). Dans ce cas, la convergence vers un timing optimal est presque toujours un procédé long, imprévisible, onéreux, et extrêmement laborieux.

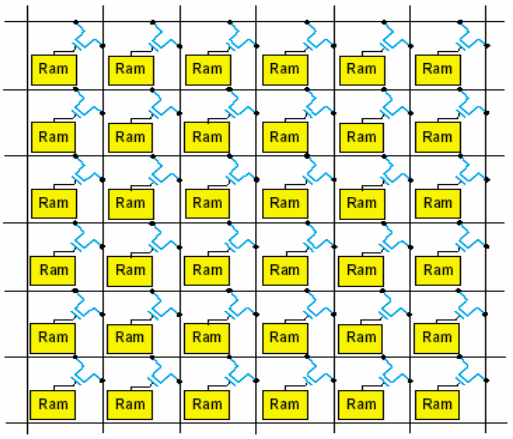


Figure 2.4 - Réseau d'aiguillage typique d'un FPGA

Au niveau des FPGA de forte complexité, la difficulté d'utilisation de ces matrices dans des applications de hautes performances réside dans le principe d'interconnexions programmables

pour le routage des signaux. Et ceci n'est pas entièrement dû au fait que la quantité de ressources de routage disponibles est plus faible dans les FPGA basés sur des points SRAM que dans les technologies concurrentes. Entre en jeu également le retard substantiel, de type R-C, inhérent à ces circuits programmables par SRAM (Figure 2.4 : Chaque point d'aiguillage consomme au moins cinq transistors pour la Ram et un transistor interrupteur de grande taille). Ce délai de propagation rend les performances du composant hautement sensibles au moindre changement dans un chemin de routage. Comme l'illustre la Figure 2.5, le point de congestion des équipotentielles, situé au centre de la puce, est d'une taille beaucoup plus importante pour les circuits les plus complexes d'une famille de FPGA. En outre, quand une interconnexion doit prendre un chemin détourné pour éviter cette congestion, l'opération se traduit par une augmentation conséquente des délais de propagation.

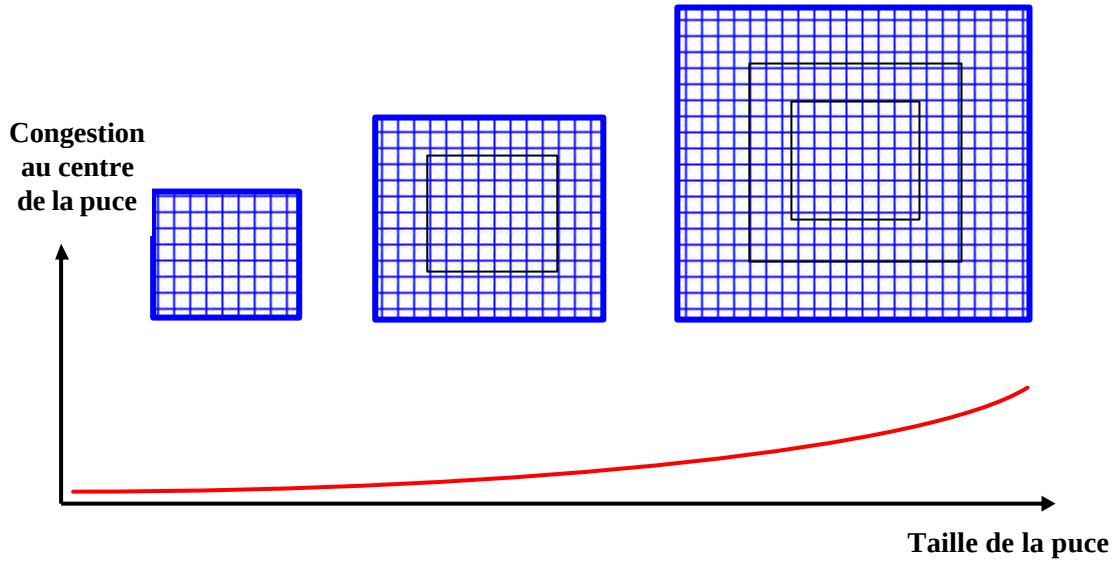


Figure 2.5 - Zones de congestion dans le routage des FPGA

De ce fait, alors les matrices de petite ou moyenne taille permettront de converger rapidement vers une solution de routage temporellement satisfaisante, cette bonne conclusion demande souvent trois ou quatre mois pour les grands FPGA.

2.2.3 Technologie FPGA du Xilinx

Dès l'origine, les FPGA, tels que Xilinx les a inventés, avaient la réputation de mettre à disposition de l'utilisateur une conception rapide, fiable et simple. Les progrès technologiques ont permis d'accéder à des matrices logiques programmables de plusieurs millions de portes. Cette complexité actuelle reste absolument gérable et permet la réalisation d'applications très performantes moyennant une bonne connaissance des ressources offertes et le respect d'une méthodologie de conception.

La Figure 2.6, montre l'architecture simplifier d'un FPGA Xilinx (pour les familles Spartan-IIE, Virtex-E et Virtex-II). Les principales caractéristiques de ces trois familles sont :

- Complexités allant de 1500 à plus de 8 millions de portes.
- Faible consommation.
- Grande souplesse d'utilisation des entrées/sorties avec adaptation d'impédance (Virtex-II) et configuration en mode différentiel (Spartan-IIE, Virtex-E et Virtex-II).
- Fonctions mémoire (distribuée et blocs de Ram).
- Dispositifs de gestion des horloges (DLL et DCM).
- Multiplieurs câblés (Virtex-II)

Et bien d'autres possibilités permettant d'optimiser à la fois la performance et la densité des fonctions logiques et/ou arithmétiques.

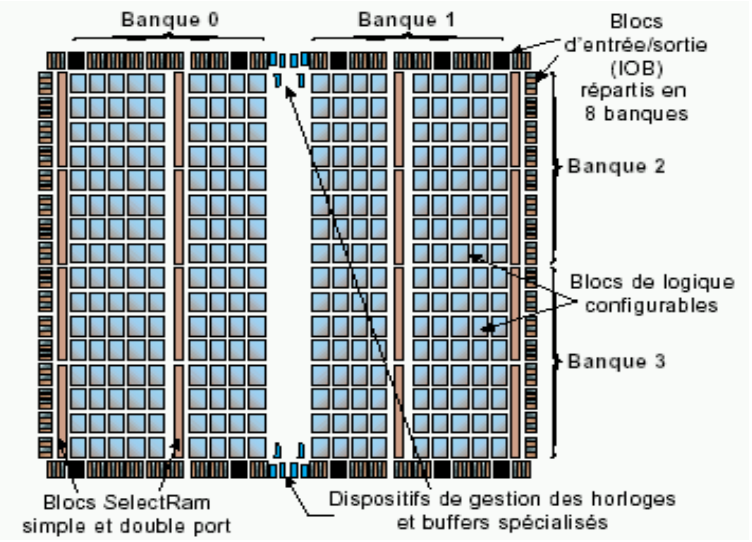


Figure 2.6 - Organisation des FPGA de Xilinx

Nous y voyons qu'un des modules de base est le bloc logique configurable (ou CLB). Lui-même constitué de s*lice*. La Figure 2.7, illustre l'architecture simplifiée d'un slice. La logique combinatoire est implantée grâce aux LUT (*Look-Up Table*) contenues dans chaque slice. Ces LUT peuvent également être configurées comme éléments de mémoire synchrone, simple ou double-port de 16 bits, ou encore comme registres à décalage de 16 bits. Il existe donc trois modes de configuration de ces LUT. Plus précisément, le fonctionnement en mode combinatoire est obtenu en lisant le contenu pointé par les signaux d'entrée (Figure 2.8a). Autrement dit, les LUT sont des mémoires dont le contenu est initialisé lors de la configuration du FPGA. De ce fait, elles permettent à l'utilisateur d'en disposer en mode « élément mémoire » dans chacun des slices si nécessaire (Figure 2.8b). La Figure 2.8c décrit le mode de configuration particulier en registre à décalage de longueur programmable jusqu'à 16 bits. Par ailleurs, une logique supplémentaire utile pour la réalisation des fonctions arithmétiques est disponible dans chaque slice. Grâce à ces éléments, et au style d'écriture adapté, des modules de type accumulateur chargeable en addition/soustraction pourront être implantés à raison de 2 bits par slice.

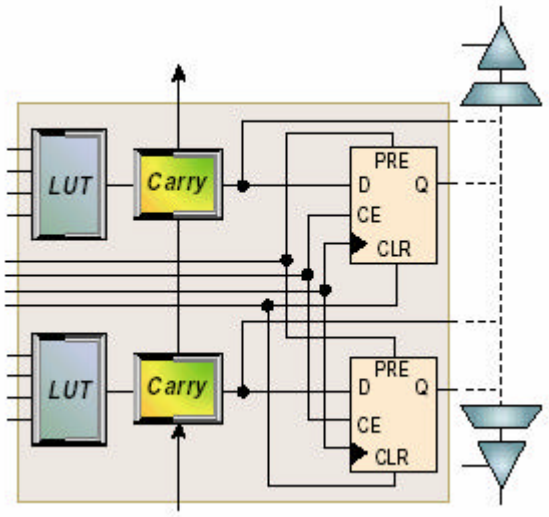


Figure 2.7 - Architecture simplifiée d'un « Slice »

Les bascules dans chaque slice ont aussi des caractéristiques importantes pour le concepteur. En particulier, elles sont initialisées systématiquement à la mise sous tension (par défaut à valeur '0'), et sont utilisables indépendamment de la logique combinatoire disponible dans le même slice. En outre, chaque bascule bénéficie de broches de contrôle telles que : entrée dédiée de validation de l'horloge (*Clock Enable*) permettant d'activer ou de suspendre le fonctionnement de chacune des bascules individuellement, et ceci sans avoir à insérer de la logique combinatoire sur le chemin de l'horloge (entrée de «*set* » et de «*reset* » synchrones ou asynchrones). La polarité des signaux d'horloge, de «*Clock Enable* », de *set* et *reset* est programmable pour chacune des bascules. Autrement dit, ces signaux peuvent être individuellement actifs au niveau haut ou au niveau bas.

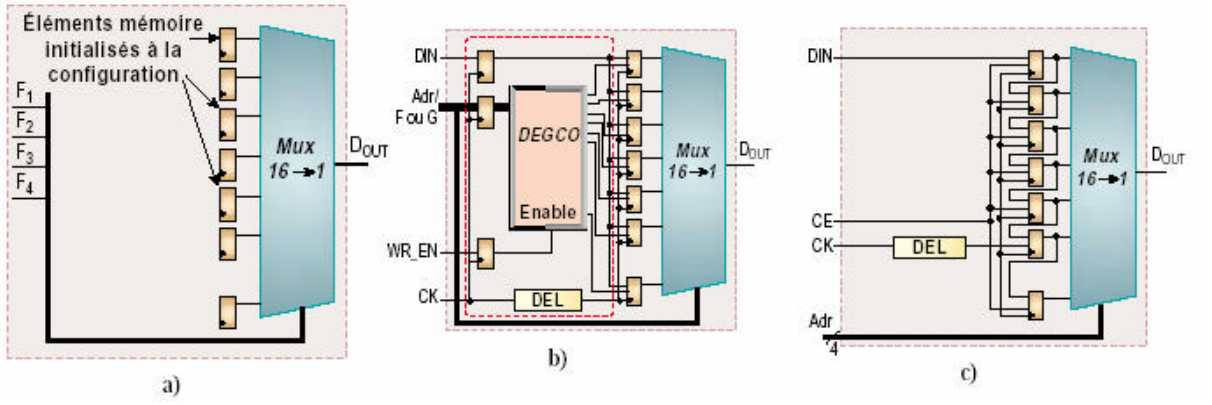


Figure 2.8 - Trois modes de configuration des LUT

L'échange des données avec les circuiteries externes (microprocesseur, DSP, mémoires rapides, convertisseurs A/N et N/A) est souvent un critère important dans la perspective des performances à atteindre. Les blocs d'entrées/sorties disposent de bascules sur les chemins d'entrées/sorties et de contrôle à trois-états. Les outils de synthèse les plus sophistiqués permettent l'utilisation systématique de ces bascules.

Enfin, les FPGA de la famille Virtex-II intègrent des registres doubles sur chacun de leurs trois chemins, autorisant ainsi la réalisation d'interfaces DDR (Double Data Rate) pour la communication avec les mémoires SDRAM du même nom ou autres dispositifs de même type (Figure 2.9).

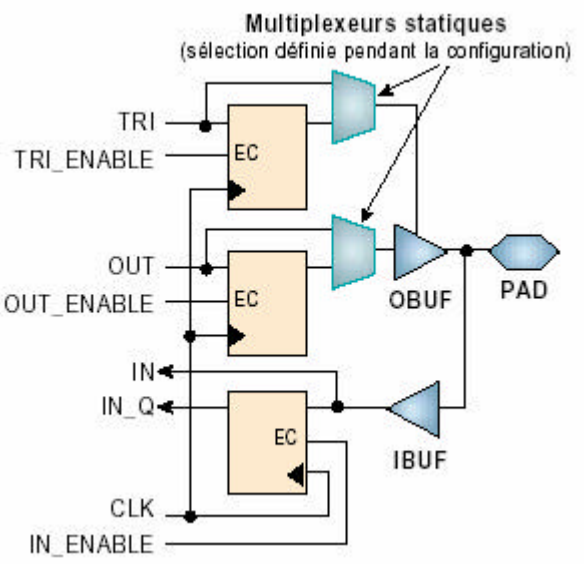


Figure 2.9 - Configuration des blocs d'E/S

La configuration électronique des blocs d'E/S donne la possibilité d'ajuster la raideur des fronts sur les étages de sortie, la sortance, les seuils de communication et les standards de communication (LVTTL, LVCmos, SSTL, PCI, LVDS…). Par ailleurs, la famille Virtex-II offre d'adapter en impédance aussi bien les entrées que les sorties, sans avoir besoin d'insérer les traditionnelles résistances d'adaptation.

2.2.4 La famille FPGA Xilinx Virtex-II

La famille FPGA Xilinx Virtex-II est constituée de onze composants dont les caractéristiques sont résumées par le Tableau 2.1.

Device	System Gates	CLB (1 CLB = 4 slices = Max 128 bits)			Multiplier Blocks	SelectRAM Blocks		DCMs	Max I/O Pads[1]
		Array Row x Col.	Slices	Maximum Distributed RAM Kbits		18 Kbit Blocks	Max RAM (Kbits)		
XC2V40	40K	8 x 8	256	8	4	4	72	4	88
XC2V80	80K	16 x 8	512	16	8	8	144	4	120
XC2V250	250K	24 x 16	1,536	48	24	24	432	8	200
XC2V500	500K	32 x 24	3,072	96	32	32	576	8	264
XC2V1000	1M	40 x 32	5,120	160	40	40	720	8	432
XC2V1500	1.5M	48 x 40	7,680	240	48	48	864	8	528
XC2V2000	2M	56 x 48	10,752	336	56	56	1,008	8	624
XC2V3000	3M	64 x 56	14,336	448	96	96	1,728	12	720
XC2V4000	4M	80 x 72	23,040	720	120	120	2,160	12	912
XC2V6000	6M	96 x 88	33,792	1,056	144	144	2,592	12	1,104
XC2V8000	8M	112 x 104	46,592	1,456	168	168	3,024	12	1,108

Tableau 2.1 - Famille FPGA Virtex-II

Les blocs « SelectRam » dans l'architecture Virtex-II sont des mémoires Ram double port de 18Kbits, configurables de différentes manières entre 18x1Kbit et 512x36 bits. Chaque port est totalement synchrone et indépendant. Les blocs «Selectram » peuvent être mis en cascade pour réaliser des larges zones des stockages enfouies dans le FPGA. Un module multiplieur 18x18 bits est disposé à proximité de chaque bloc « SelectRam » et optimisé pour opérer sur le contenu d'un port de la mémoire.

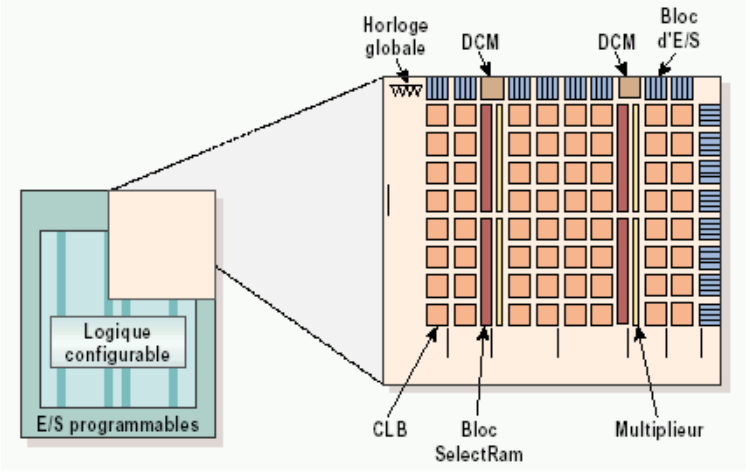


Figure 2.10 - Détail de l'architecture d'un composant Virtex-II

Les fonctions mémoire « SelectRam » et les multiplieurs sont connectés aux matrices d'interconnexions, cette configuration leur ouvrant l'accès aux ressources de routage globales de la matrice FPGA. La Figure 2.10 présente en détail l'architecture d'un composant Virtex-II.
Dans cette famille de Virtex-II, on s'intéresse particulièrement au composant XC2V6000, puisque c'est le composant supporté par la plateforme ADM-XRC-II.

2.2.5 FPGA Virtex-II Pro

Xilinx a mis depuis peu sur le marché, une nouvelle technologie de circuits FPGA intégrants de nouvelles fonctionnalités, ces derniers permettent la réalisation d'architectures qui auparavant s'avéraient très compliquées à entreprendre.

Parmi ces nouveaux circuits, le VIRTEX-II PRO (Figure 2.11) qui est un composant FPGA intégrant un cœur de processeur PowerPC 405 de chez IBM pouvant fonctionner à plus de 300MHz. Afin de rendre le co-design plus simple à réaliser, la famille de composant Virtex-II-Pro possède plusieurs atouts tel que :

- l'intégration d'un ou deux cœurs de processeur powerPC405
- un grand nombre de slices pouvant atteindre 44.096 unités permet d'implémenter des algorithmes dont la complexité les rend gourmand en consommation des ressources logiques
- des blocs de mémoire de 18Kb sont aussi disponible ce qui diminuera l'accès vers des ressources externes et ce qui peut être fatal pour le circuit surtout du point de vue consommation d'énergie
- des multiplieurs 18x18 bits, des DCM (Digital Clock Manager).

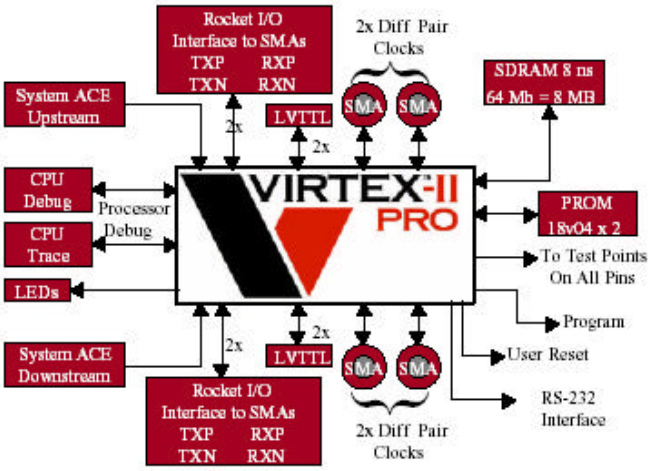


Figure 2.11 - Vue Globale du VIRTEX-II PRO

Pour cela Xilinx fourni des IPs soft (Figure 2.12) écrites en langage VHDL qui permettent d'exploiter le PowerPC405 et de lui fournir tous les périphériques nécessaires à son fonctionnement. Les premières IPs à implémenter autour du processeur sont le PLB « Processor Local Bus » et le « On Chip Memory Controller », la première constitue le bus de communication sur lequel viennent se greffer d'autres IPs, quant au «On Chip Memory Controller » il permet de connecter le PowerPC405 avec les Bram's du FPGA ou bien avec des RAMs externes où sera logé le code à exécuter.

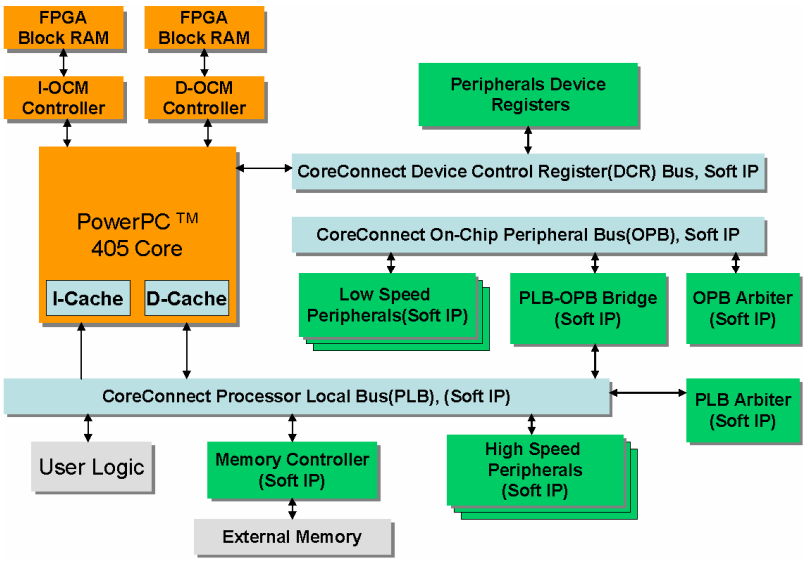


Figure 2.12 - Architecture standard autour du PowerPc405 sur VIRTEX-II PRO

Une application peut se limiter à une configuration avec un PowerPC405, une IP « Processor Local Bus » et un «On Chip Memory Controller », l'utilisateur peut ajouter des bus ou bien des contrôleurs de mémoire externe ou autres IPs, selon les exigences de son architecture.

Device	RocketIO Transceiver Blocks	PowerPC Processor Blocks	Logic Cells	CLB (1 = 4 slices = max 128 bits)		18X 18 Bit Multiplier Blocks	Block SelectRAM+		DCMs	Maximum User I/OPads
				Slices	Max Distr RAM(Kb)		18Kb Blocks	Max Blocks RAM (Kb)		
XC2VP2	4	0	3,168	1,408	44	12	12	216	4	204
XC2VP4	4	1	6,768	3,008	94	28	28	504	4	348
XC2VP7	8	1	11,088	4,928	154	44	44	792	4	396
XC2VP20	8	2	20,880	9,280	290	88	88	1,584	8	564
XC2VPX20	8	1	22,032	9,792	306	88	88	1,584	8	552
XC2VP30	8	2	30,816	13,696	428	136	136	2,448	8	644
XC2VP40	0, 8, or 12	2	43,632	19,392	606	192	192	3,456	8	804
XC2VP50	0 or 16	2	53,136	23,616	738	232	232	4,176	8	852
XC2VP70	16 or 20	2	74,448	33,088	1,034	328	328	5,904	8	996
XC2VPX70	20	2	74,448	33,088	1,034	308	308	5,544	8	992
XC2VP100	0 or 20	2	99,216	44,096	1,378	444	444	7,992	12	1,164

Tableau 2.2 - Ressources du circuit VIRTEX-II PRO/ VIRTEX-II PRO X

La famille VIRTEX-II PRO dispose d'un nombre suffisant de ressources permettant la réalisation d'applications « System On Chip » et ce avec une électronique associée très réduite. Le Tableau 2.2 donne un récapitulatif des ressources dont dispose le composant de la famille VIRTEX-II PRO.

2.2.6 Flot de conception FPGA

Xilinx fournit dans son pack d'outils différents soft permettant la création de systèmes embarqués sur puce, parmi ces softs on dénombre ISE (Integrated Software Environment) et EDK (Embedded Development Kit), tous deux nous offrent la possibilité d'avoir un bitstream pour la programmation des FPGA suivant l'application ciblée.

2.2.6.1 Vue Générale

Le flot de conception standard est composé de plusieurs étapes :

- Conception et synthèse du design
- l'implémentation et la vérification du design

Une vue générale du flot de conception est présentée par la figure suivante :

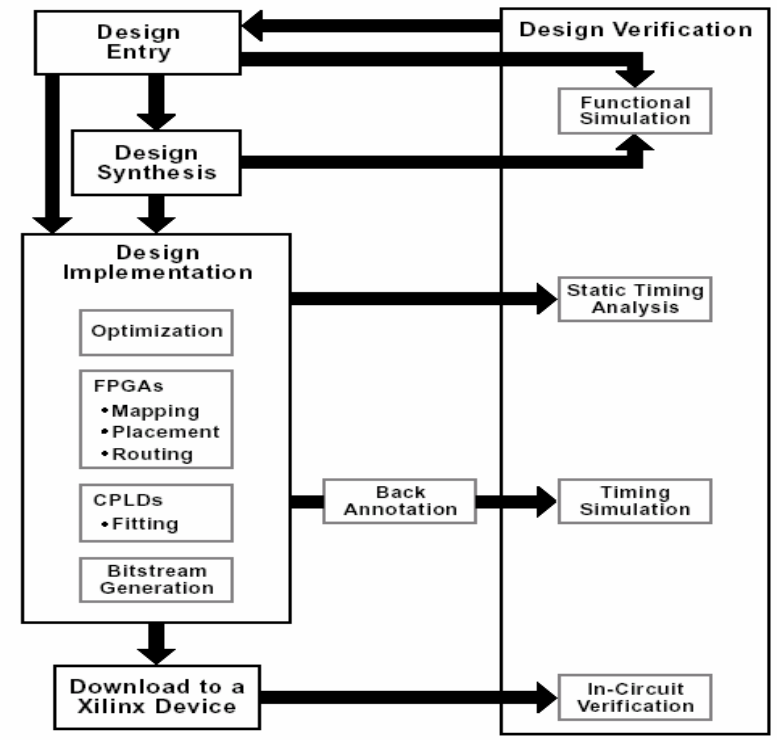


Figure 2.13 - Flot de conception générique

Les outils (Xilinx [45]) utilisés dans notre cas spécifient en détail ce flot générique de la figure 2.13, pour donner un flot plus précis, qui est illustré par la figure suivante :

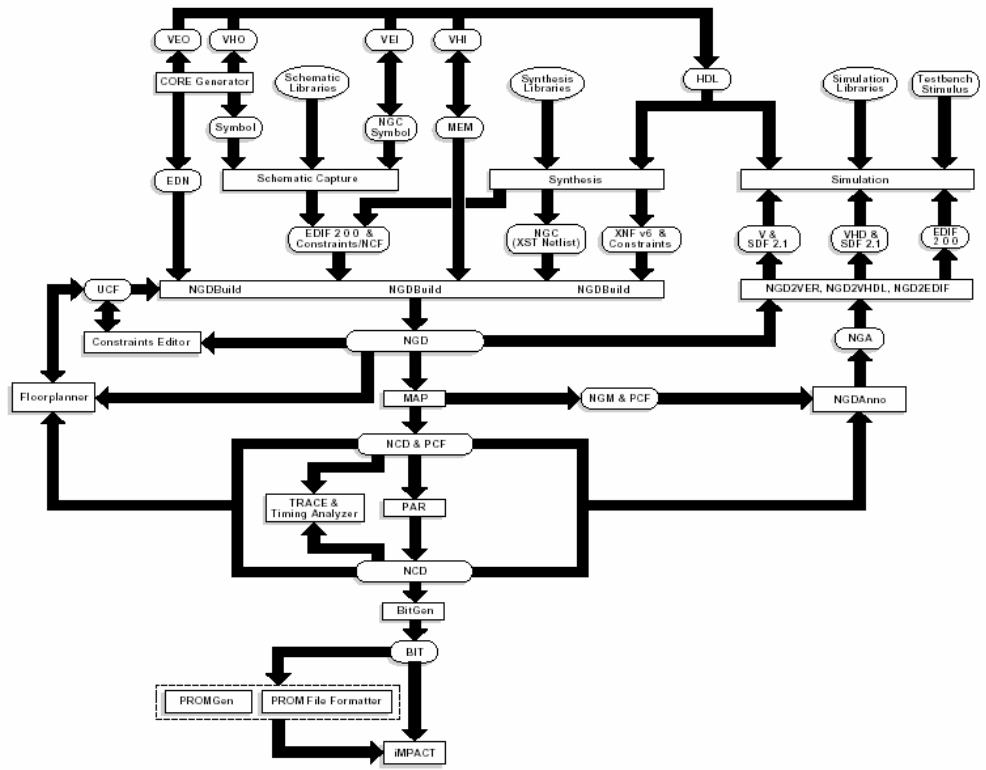


Figure 2.14 - Flot complet de conception FPGA de Xilinx

2.2.6.2 Conception et Synthèse

Un design peut être conçu à l'aide d'un éditeur schématique ou d'un outil de traitement de textes. La conception commence par un concept exprimé par un schéma ou une description fonctionnelle. A partir du design original, une *netlist* est créée, synthétisée puis translatée en fichier NGO (*Native Generic Object*). Ce fichier est utilisé ensuite par un logiciel appelé NGDBuild qui produira alors un fichier NGD (*Native Generic Database*).

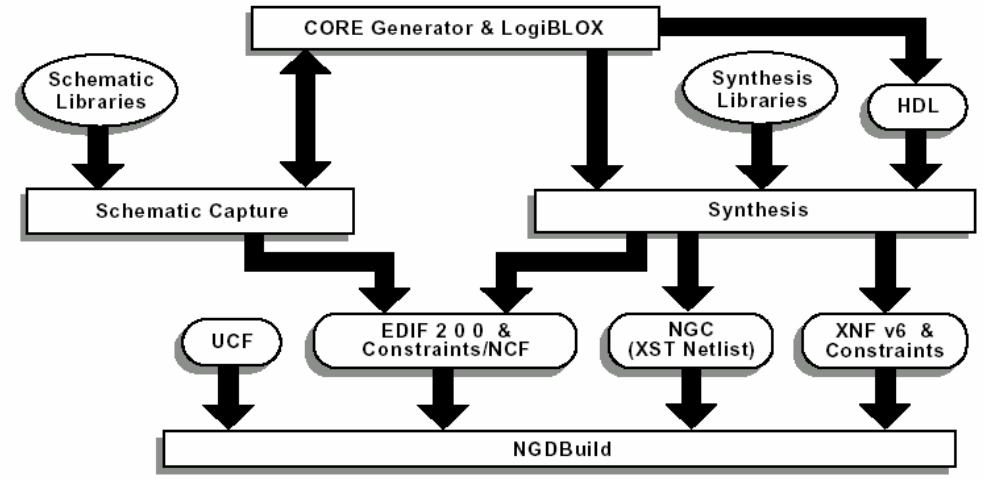


Figure 2.15 - Flot de synthèse sous Xilinx

2.2.6.3 Design Hiérarchique

Une hiérarchie du design est importante dans les deux modes de conception : Schématique et HDL. Et ceci pour plusieurs raisons :

- pour avoir un design bien structuré et facile à déboguer
- combinaison de plusieurs types de conceptions (Schématiques, VHDL, Verilog, etc.)
- design incrémental qui consiste à concevoir, implémenter et vérifier des parties de système individuellement
- réduit le temps d'optimisation et facilite les designs concurrents par une répartition du système en parties qui seront développées simultanément par plusieurs personnes (comme de design modulaire).

2.2.6.4 Schématiques

Les outils schématiques produisent des interfaces graphiques pour décrire le design. Ces outils peuvent être utilisés pour connecter des symboles représentants des instances des composants utilisés dans le design. Le design peut être construit à l'aide des portes individuelles, ou à l'aide de blocks fonctionnels en utilisant des éléments de librairies et des outils comme *Core Generator* et *LogiBLOX*.

2.2.6.5 Les éléments des Librairies

Les primitives et les macros sont les constituants fondamentaux des librairies de composants. Les librairies de *Xilinx* offrent des primitives et des macros de haut niveau.
Une primitive est une partie élémentaire du circuit (portes logiques, Flip-Flops, etc.). Chaque primitive a des propriétés caractéristiques (nom de librairie, symbole et description). Un macro contient plusieurs éléments de librairies qui peuvent être primitives ou des macros.
Deux types de macros sont disponibles à l'utilisation avec les FPGA de Xilinx :

- les *soft-macros* dont les fonctionnalités sont prédéfinies, mais qui sont flexibles du point de vue *mapping*, placement et routage.
- les RPMs (*Relationnaly Placed Macros*) qui possèdent un *mapping* et un placement relatif fixés.

Les macros ne sont pas disponibles pour la synthèse parce que les outils de synthèses possèdent leurs propres générateurs de modules et ne requièrent pas de RPMs.

2.2.6.6 Outil de Génération de Cores

L'outil de conception *CORE Generator* de Xilinx offre des cores paramétrables qui sont optimisés pour les FPGAs de Xilinx. La librairie contient des éléments qui varient entre de simples registres à des applications très complexes (filtres et multiplexeurs DSP).

2.2.6.7 HDL et synthèse

Un langage de description Hardware (HDL) offre la possibilité de description à plusieurs niveaux dans laquelle les portes élémentaires et les *netlists* peuvent être utilisés avec une description fonctionnelle. Cette variation de niveau permet de décrire l'architecture du système à un niveau d'abstraction plus élévée, puis d'une façon incrémentale raffiner l'implémentation du design jusqu'au niveau des portes.
La description HDL offre les avantages suivants : la fonctionnalité du design peut être vérifiée immédiatement ce qui permet d'évaluer les décisions architecturales, une description HDL est lue et comprise plus facilement qu'une *netlist* ou une description schématique et elle constitue une documentation du design et de sa fonctionnalité indépendante de la technologie, et sans oublier qu'une description HDL facilite la manipulation de larges designs par rapport à une description schématique.
Après la création du design en HDL, il faut le synthétiser. Durant la synthèse, les informations comportementales dans le fichier HDL sont traduites en *netlist* structurale, et le design est optimisé pour les composants de Xilinx. L'outil de synthèse utilisé par ISE est XST ou *Xilinx Synthesis Tool*.

2.2.6.8 Simulation fonctionnelle

Après sa conception et sa description, la simulation fonctionnelle teste la logique du design pour détecter et corriger tout dysfonctionnement dû à une mauvaise conception ou à une description erronée. Le logiciel utilisé pour la simulation fonctionnelle est le *ModelSim*.

2.2.6.9 Contraintes

Pour forcer le système à travailler selon certaines limitations temporelles ou de surfaces, des contraintes sont utilisées. Les contraintes peuvent être introduites manuellement, ou à l'aide de l'éditeur de contraintes, *floorplanner* ou *FPGA editor* (outils accompagnant ISE). Pour évaluer le circuit après implémentation sous ces contraintes *Timing Analyser* ou *TRACE* peuvent être utilisés.

2.2.6.9.1 Contraintes de Mapping

Pour spécifier le *mapping* d'un bloc logique dans les CLBs, les contraintes FMAP ou HMAP (pour certaines familles) peuvent être utilisées. Une utilisation excessive de telles contraintes peut rendre le routage difficile ou même impossible.

2.2.6.9.2 Contraintes de Placement

Le placement d'un bloc peut être forcé à une certaine position donnée. La location peut être spécifiée dans la description schématique, dans l'outil de synthèse, ou dans un fichier UCF (*User Constraints File*). Le mauvais placement d'un bloc peut empêcher le design d'être

placé et routé complètement. Typiquement, seuls les blocs d'entrée/sortie requièrent des contraintes de placement pour qu'ils soient dans les bonnes positions par rapport aux pins.

2.2.6.9.3 Contraintes temporelles

C'est pour spécifier les contraintes temporelles pour limiter le temps de propagation des signaux le long des chemins de traitement. L'outil de placement et de routage PAR utilise ces contraintes pour achever une performance optimale durant l'opération de placement et routage.

2.2.6.10 Translation en NetList

C'est le format requis par les outils de développement de Xilinx pour compléter le flot de conception est NGD (*Native Generic Database*). Deux outils sont disponibles pour effectuer la translation d'autres formats en format NGD. Par exemple *EDIF2NGD* traduit un fichier EDIF (*Electronic Data Interchange Format*) en NGD et le programme *NGDBuild* convertie plusieurs formats (XNF, ou EDIF) en NGD. A noter que ce dernier outil, utilise le premier pour la traduction des fichiers EDIF.

2.2.6.11 Implémentation

L'implémentation du design commence avec le *mapping* qui se traduit par l'association des éléments logiques avec les éléments physiques d'un composant donné, et se termine lorsque le design physique est routé totalement et traduit en un série de bits chargeable dans le composant.

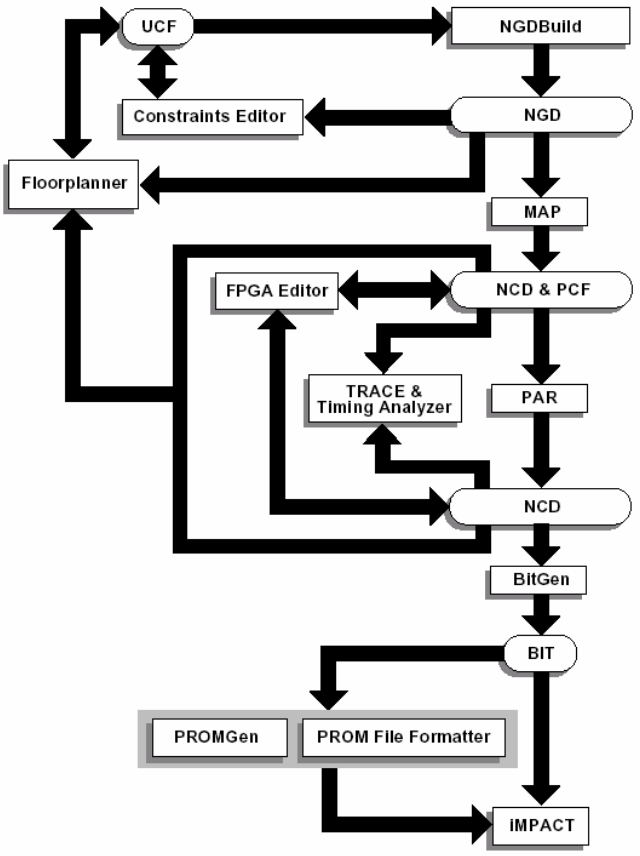


Figure 2.16 - Implémentation

Des contraintes peuvent toujours être ajoutées durant le processus d'implémentation pour que le design satisfasse le cahier de charge. Le processus d'implémentation est illustré par la Figure 2.16.

2.2.6.12 Mapping

L'outil MAP arrange les éléments logiques d'un design dans un FPGA de Xilinx et les associe à des éléments physiques. L'outil MAP prend en entrée un fichier NGD qui contient une description logique du design en terme de composants hiérarchiques utilisés dans le design et de primitives de Xilinx de bas niveau, et d'un nombre quelconque de fichiers NMC (*hard placed and routed macros*), chacun contenant la définition d'un macro physique. MAP associe alors des composants (cellules logiques, cellules d'entrée/sortie, etc) aux différentes parties logiques du design. La sortie du MAP est un fichier de description de circuit d'extension NCD (*Native Circuit Description*). Ce fichier est une représentation physique du design à l'aide des éléments constitutifs du FPGA cible. Le fichier NCD peut être placé et routé.

2.2.6.13 Placement et Routage

L'outil PAR (*Place And Route*) prend à l'entrée une description physique du design (un fichier NCD), le place et le route pour donner un autre fichier NCD, utilisable par BitGen, afin qu'il soit transformé en données de configuration. Le fichier NCD de sortie peut être aussi utilisé comme fichier guide pour refaire le placement routage après avoir effectuer de petits changements au design. L'outil *FPGA editor* permet de placer et router des composants critiques avant le lancement du placeur/routeur automatique, et permet aussi de modifier manuellement le placement et le routage.

2.2.6.14 Génération de flot de bits

L'outil *BitGen* produit une série de bits pour la configuration d'un FPGA de Xilinx. *BitGen* prend un fichier NCD totalement placé et routé pour générer la série de bits de configuration sous la forme d'un fichier d'extension *.bit*. Le fichier *BIT* contient toutes les informations contenues dans le fichier NCD (la logique interne et les interconnections entre les éléments du FPGA), et des informations caractéristiques du composant cible à partir d'autres fichiers associés avec le composant cible. Après la génération du fichier *BIT*, il peut être chargé dans le FPGA à l'aide de l'outil *iMPACT*. Un fichier PROM peut aussi être créé à partir du fichier *BIT*, dans le but de le charger dans une *PROM*, affin d'être utilisé ultérieurement par le FPGA.

2.2.6.15 Vérification

La vérification du design est un processus de test des fonctionnalités et des performances du design. *Xilinx* offre plusieurs méthodes pour ce but :

- la simulation fonctionnelle et temporelle
- un analyse temporelle statique
- la vérification sur le circuit

La procédure de vérification du design se fait normalement comme l'illustre la figure :

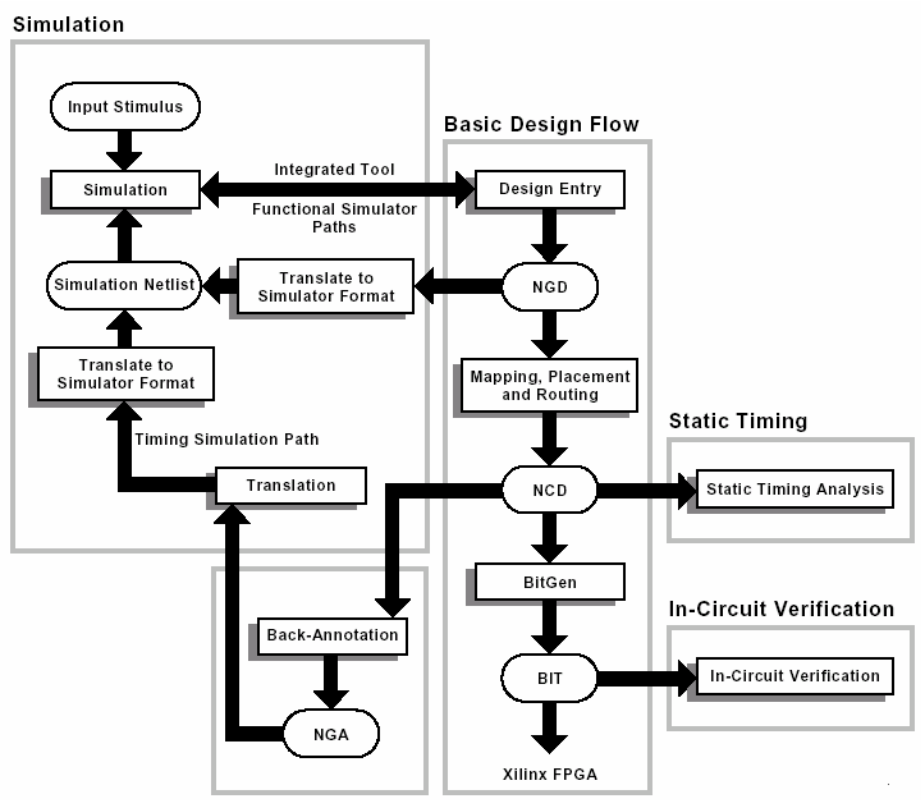


Figure 2.17 - Vérification

Pour vérifier le design, une simulation fonctionnelle ou temporelle peut être exécutée. Un processus de *Back-Annotation* doit avoir lieu avant la simulation temporelle. Avant la phase de la simulation temporelle, la description physique du design doit être traduite en design logique compréhensible par le simulateur. Cette tâche est appelée *Back-Annotation process* en anglais. Un outil appelé *NGDAnno* est chargé de créer une base de données afin qu'elle soit utilisée par l'outil créateur de *netlist*, qui traduit ces informations en format *netlist* compréhensible par le simulateur. La figure suivante illustre le *Back-Annotation process*.

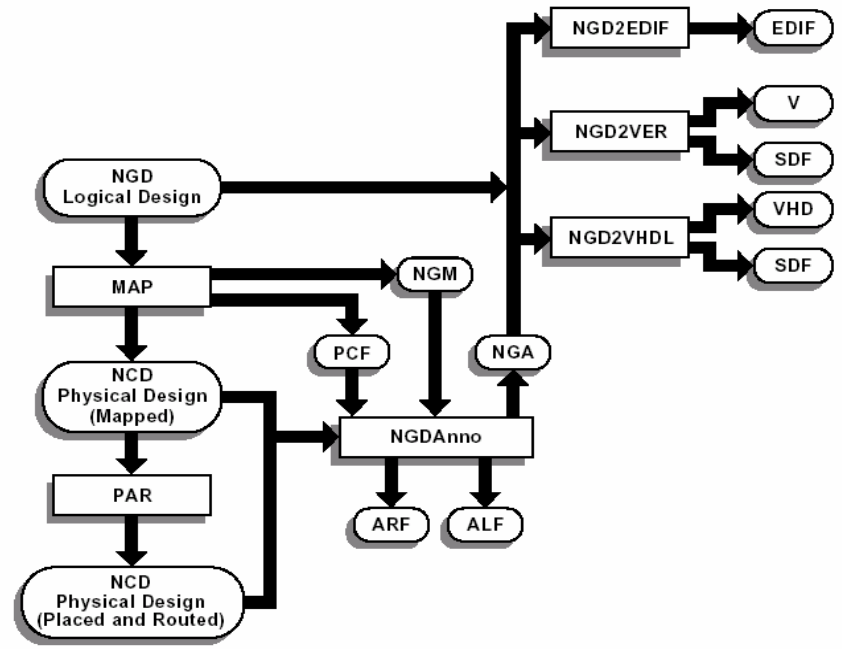


Figure 2.18 - Processus de Back-Annotation

NGDAnno est un outil qui traduit les caractéristiques physiques (délais, temps de propagation des signaux, temps de réponses, etc) trouvées dans un fichier de description de circuit NCD, en fichier de description logique du design NGD (en tenant compte de toutes les caractéristiques physiques du circuit). Le fichier NCD à l'entrée peut être un design totalement ou partiellement placé et routé, ou qui n'est ni placé ni routé (NCD à la sortie de l'outil MAP). Un fichier NGM créé aussi par MAP est une entrée optionnelle pour *NGDAnno* qui fusionne les informations de *mapping* contenues dans ce dernier fichier avec les informations de placement, routage et les informations temporelles provenant du fichier NCD employé. A la sortie, cet outil génère un fichier NGA (Native Generic Annotation) qui est un fichier NGD reconstruit à partir d'un NCD. Ce fichier constituera une entrée pour l'outil de génération de netlist approprié, qui convertira le design du format NGA binaire, en format netlist ASCII. Les *netlist writers* (*NGD2EDIF*, *NGD2VER* ou *NGD2VHDL*) prennent la sortie de l'outil *NGDAnno* et créent une *netlist* de simulation au format spécifié. Un fichier NGD ou NGA peut être une entrée d'un *netlist writer*. *NGD2EDIF* traduit un fichier NGD ou NGA en *netlist* au format EDIF (fichier EDN). *NGD2VER* traduit un fichier NGD ou NGA en netlist au format Verilog (fichier V). Cet outil génère aussi un autre fichier SDF (*Standard Delay Format*) qui contient des informations temporelles qui est utilisable uniquement avec le fichier Verilog créé par le même outil, à partir du même fichier NGD ou NGA. *NGD2VHDL* traduit un fichier NGD ou NGA en *netlist* VHDL (fichier VHD). Si le fichier d'entrée est du format NGA, cet outil génère aussi un fichier SDF spécifique pour ce fichier VHD.

2.2.6.15.1 Simulation basée sur la schématique

La simulation du design c'est le fait de tester le design à l'aide de modèles logiciels. Il est plus efficace de tester le design et ses performances dans les pires conditions. Ce modèle permet de consulter les nœuds internes pour examiner le comportement du circuit, et utiliser les résultats pour changer la description schématique du design. La simulation est faite à l'aide d'outils de partie tierce, qui sont liés au système de développement de *Xilinx*. Les modèles logiciels offerts pour les outils de simulation sont conçus pour présenter les caractéristiques détaillées du design.

2.2.6.15.2 Simulation fonctionnelle

La simulation fonctionnelle ou comportementale détermine si la logique du design est correcte avant la phase d'implémentation. Ce type de simulation peut avoir lieu tôt dans le processus de conception du système. Et puisque les informations temporelles ne sont pas disponibles à ce moment, le simulateur teste la logique en utilisant des délais élémentaires comme unités. Le simulateur utilisé (*ModelSim*) est un simulateur intégré dans l'environnement de développement de *Xilinx* : le passage entre les outils de conceptions (éditeurs HDL ou schématiques) et le simulateur se fait automatiquement sans besoin d'utilisation intermédiaire d'outils de translation.

2.2.6.15.3 Simulation temporelle

La simulation temporelle examine le temps d'exécution du design dans les pires conditions. Ce processus peut avoir lieu après le *mapping*, le placement et le routage du design. A ce moment là, tous les délais du design sont bien connus. La simulation temporelle est très importante parce qu'elle peut vérifier les relations temporelles et détermine les chemins critiques du design dans les pires conditions. Avant la simulation temporelle, il faut passer par le *Back-Annotation process* déjà mentionné.

2.2.6.15.4 Simulation basée sur HDL

Xilinx offre les possibilités d'effectuer des simulations fonctionnelles et temporelles des designs exprimés en HDL à plusieurs niveaux (Figure 2.19):

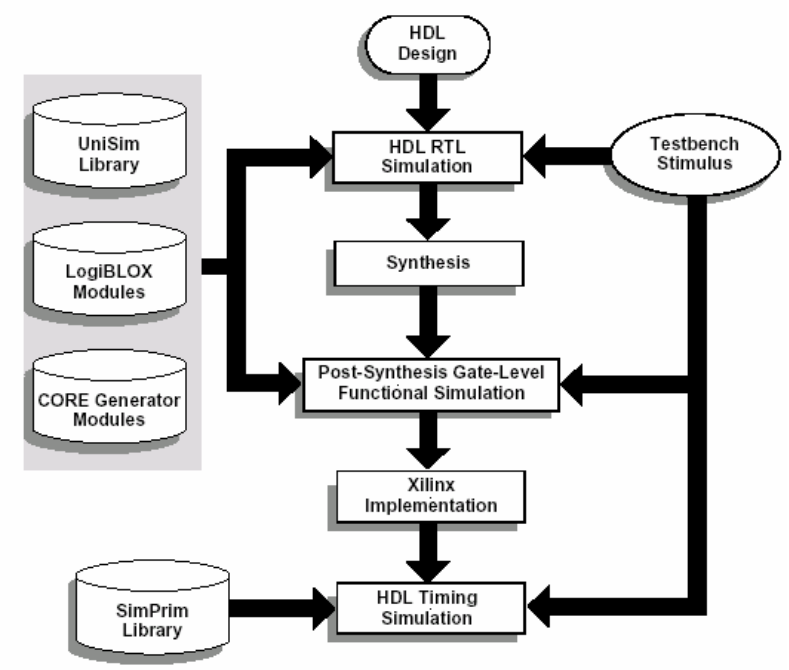


Figure 2.19 - Simulation

Simulation au niveau RTL (*Register Transfer Level*) qui peut renfermer des models *LogiCORE* et des instances des composants de la librairie *UniSim*, au niveau post-synthèse (simulation fonctionnelle), au niveau post implémentation (simulation temporelle après le processus de *Back-Annotation*).

2.2.6.16 Analyse temporelle statique

L'analyse temporelle statique est favorisée dans le cas de vérification rapide d'un design après placement et routage. Elle permet de déterminer les délais des chemins du design. Les deux buts majeurs de l'analyse temporelle sont : la vérification temporelle (vérifier que le design obéit au contraintes temporelles) et le rapportage (décrire le comportement temporel d'une façon technique qui peut être utilisée comme documentation pour évaluer le design). L'outil TRACE (*Timing Reporter And Circuit Evaluator*) est responsable de faire cette analyse.

2.2.6.17 Vérification sur le circuit

La vérification du comportement de design dans l'application cible est considérée comme un test final. La vérification du design sur le circuit test, circuit dans les conditions typiques de fonctionnement. Ce type de test est simple du fait que les composants de Xilinx sont reprogrammables (plusieurs itérations du design peuvent être essayées).
Avant la création du fichier BIT final, il est utile d'utiliser l'option DRC dans l'outil *BitGen* pour évaluer le fichier NCD, et chercher les problèmes qui peuvent empêcher le design de fonctionner correctement sur le composant cible.

2.3 Module ADM-XRC-II

Dans cette cette partie, nous présentons la platforme ADM-XRC-II d'Alpha-Data [43] qu'on a emplyé pour l'exploration architecturale par emulation (*Cf chapitre 4.6).*

2.3.1 Présentation de la plateforme ADM-XRC-II

La plateforme ADM-XRC-II [43] utilise un FPGA de type Xilinx Virtex-II XC2V6000 (Tableau 2.1). Ce type de composant est très utile pour les applications de télécommunications car en plus de sa grande densité d'intégration, sa faible consommation et sa haute fréquence de travail, le Virtex-II possède des entrées/sorties compatibles pour les interfaces LVDS et le PCI. Le FPGA Virtex-II XC2V6000 employé par la plateforme est un boîtier de type FF1152 (boîtier BGA de 35mm x 35mm avec 1152 broches).
Les caractéristiques techniques de ce composant (Tableau 2.1) sont les suivantes :

- Nombre de portes : 6 Millions
- Horloge interne maximale : 420 MHz
- Alimentation du cœur VCCint = 1,5V
- 824 entrées/sorties
- Il possède 16 multiplexeurs d'horloge interne.

La particularité de ce type de technologie (famille FPGA Virtex-II) est d'accepter la reconfiguration dynamique partielle. C'est-à-dire qu'en cours d'exécution d'une application sur le FPGA, nous pouvons reconfigurer une ou plusieurs zone sans devoir reconfigurer entièrement le FPGA. Cette particularité peut s'avérer intéressante pour notre « Plateforme de Prototypage SOC pour Radio Logicielle ». Dans ce cas, nous configurons le FPGA avec une IP de processeur possédant des paramètres déterminés, ensuite au cours de l'exécution nous modifions une partie de ce processeur avec des paramètres différents. Ceci permet un gain de temps au niveau de la phase de la synthèse de design (re-synthétiser et configurer uniquement la partie concernée sur l'FPGA).

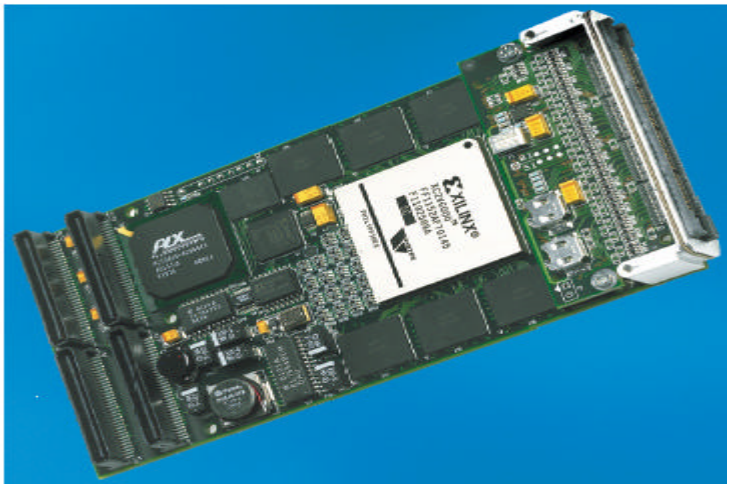


Figure 2.20 - Plateforme de prototypage ADM-XRC-II (Alpha Data)

D'autre part, le module ADM-XRC-II (Figure 2.20) intègre une interface PCI (à travers un connecteur PMC), permettant à un PC, de reconfigurer et/ou communiquer avec le FPGA. Avec ce module, sont fournis tous les drivers et bibliothèques, permettant au PC de communiquer avec celui-ci sous les environnements Linux, Windows et VxWorks.

2.3.2 Spécifications techniques du module ADM-XRC-II

Le module ADM-XRC-II supporte les hautes performances du bus PCI sans avoir à utiliser une IP PCI dans le FPGA. Une interface PCI Hard PLX 9656 [46] gère les communications entre le FPGA et le PC. A travers cette interface PCI, gère aussi les commandes du module tel que l'horloge programmable.

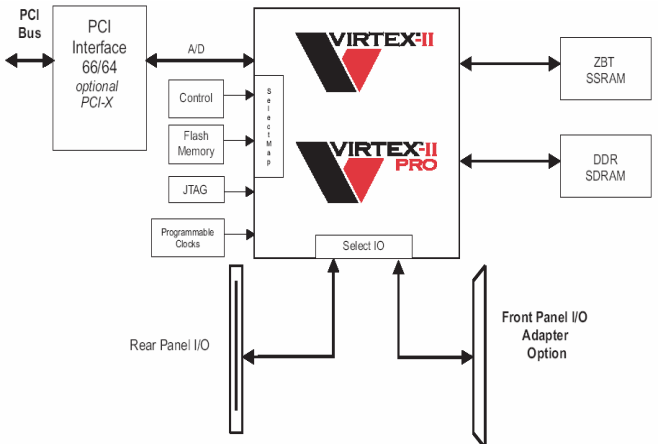


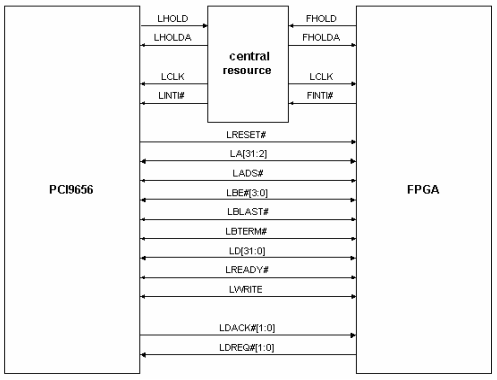
Figure 2.21 - Plateforme ADM-XRC-II

Les caractéristiques techniques sont les suivantes :

- Bus PCI 64bits – 66 MHz
- 6 banques indépendantes de ZBT SSRAM 256K x 32bits
- 64 entrées/sorties via un connecteur PMC
- 146 entrées/sorties via un connecteur XRM
- Sur la carte, une horloge programmable fournit au bus local (entre FPGA et interface PCI) une fréquence variable : 400kHz à 66 MHz
- Une autre horloge programmable fournit une fréquence au FPGA : 0 à 100 MHz
- Possibilité de connecter 256 MO de DDR SDRAM via le connecteur XRM

2.3.3 Signaux du bus local

Le bus local du module *ADM-XRC-II* utilise l'interface PCI 9656 [46] pour fournir des adresses et des données non multiplexées à une fréquence maximale de 66 MHz indépendant des opérations se déroulant sur le bus PCI. Le bus local est plus simple à interfacer qu'un bus PCI et il se rapproche des performances du bus PCI.
Les différents signaux du bus local sont les suivants :

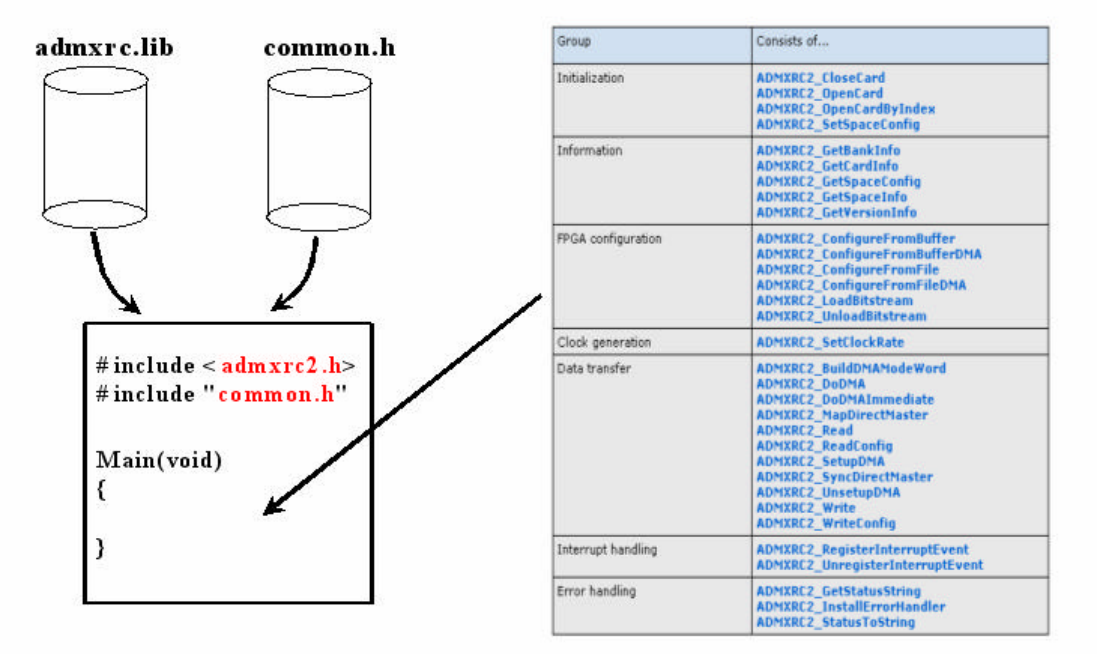


Signal	Actif	Direction	Rôle
FHOLD	Haut	OUT	
FHOLDA	Haut	IN	
LCLK	Haut	IN	Horloge du bus Local
FINTI	Bas	OUT	Interruption
LRESET	Bas	IN	Reset depuis le PCI
LA[31..2]	Haut	IN	Adresses
LADS	Bas	IN	Adresse / Data Start
LBE[3..0]	Bas	IN	Adresses/Bit Enable
LBLASTL	Bas	IN	Fin des Data/Emission
LBTERM	Bas	OUT/TRI	Fin Data/Réception
LD[31..0]	Haut	BIDIR	Bus de données
LREADY	Bas	OUT/TRI	Accept transfert data
LWRITE	Haut	IN	Write/Read
DACK[1..0]	Bas	IN	DMA transfert
DREQ[1..0]	Bas	OUT	Requete transfert DMA

Figure 2.22 - Signaux bus local Plateforme ADM-XRC-II

2.3.4 Fonctions de commande plateforme

Le module *ADM-XRC-II* est fourni avec un driver permettant au PC de communiquer avec le module hôte. A ce driver, nous pouvons utiliser des fonctions C via l'appel des bibliothèques permettant de configurer le FPGA du module et de communiquer avec celui-ci à travers l'interface PCI.



Group	Consists of...
Initialization	ADMXRC2_CloseCard ADMXRC2_OpenCard ADMXRC2_OpenCardByIndex ADMXRC2_SetSpaceConfig
Information	ADMXRC2_GetBankInfo ADMXRC2_GetCardInfo ADMXRC2_GetSpaceConfig ADMXRC2_GetSpaceInfo ADMXRC2_GetVersionInfo
FPGA configuration	ADMXRC2_ConfigureFromBuffer ADMXRC2_ConfigureFromBufferDMA ADMXRC2_ConfigureFromFile ADMXRC2_ConfigureFromFileDMA ADMXRC2_LoadBitstream ADMXRC2_UnloadBitstream
Clock generation	ADMXRC2_SetClockRate
Data transfer	ADMXRC2_BuildDMAModeWord ADMXRC2_DoDMA ADMXRC2_DoDMAImmediate ADMXRC2_MapDirectMaster ADMXRC2_Read ADMXRC2_ReadConfig ADMXRC2_SetupDMA ADMXRC2_SyncDirectMaster ADMXRC2_UnsetupDMA ADMXRC2_Write ADMXRC2_WriteConfig
Interrupt handling	ADMXRC2_RegisterInterruptEvent ADMXRC2_UnregisterInterruptEvent
Error handling	ADMXRC2_GetStatusString ADMXRC2_InstallErrorHandler ADMXRC2_StatusToString

Figure 2.23 - Fonctions de commande Plateforme ADM-XRC-II

Le code source C permettant de piloter le module via le PC doit faire appel à :

- Une librairie «admxrc.lib » correspondant au type de module utilisé, nécessaire pour l'outil de compilation Microsoft *Visual C++*
- Un fichier header « admxrc2.h » et « common.h »

2.4 Méthodologies de Conception SOC et IPs

Les méthodes de conception traditionnelles sont de moins en moins adaptées aux circuits de grandes complexités. Les concepteurs sont amenés à gérer, non seulement un nombre énorme de transistors, mais aussi une incroyable complexité dans le *design*. Les possibilités architecturales pour concevoir de tels systèmes sont extrêmement nombreuses. Par conséquent, l'évaluation des différentes solutions architecturales devient un passage obligé de la conception moderne de circuits intégrés. Ces facteurs, couplés à une demande de plus en plus forte de réduction du temps de mise sur le marché avec des performances de plus en plus pointues, font qu'il n'est plus possible de développer un système en partant de zéro.
La solution à ces problèmes est la réutilisation de blocs déjà conçus et validés. La notion de composants génériques réutilisables s'impose ainsi de plus en plus.

2.4.1 Les composants réutilisables

Nous entendons par composant ou bloc réutilisable un élément d'une bibliothèque dont le concepteur dispose et qu'il peut directement inférer ou instancier sans avoir à le concevoir. L'un des problèmes majeurs concernant la réduction du temps de conception (TTM : Time To Market) réside dans la bonne ou la mauvaise réutilisation des blocs existants [3].

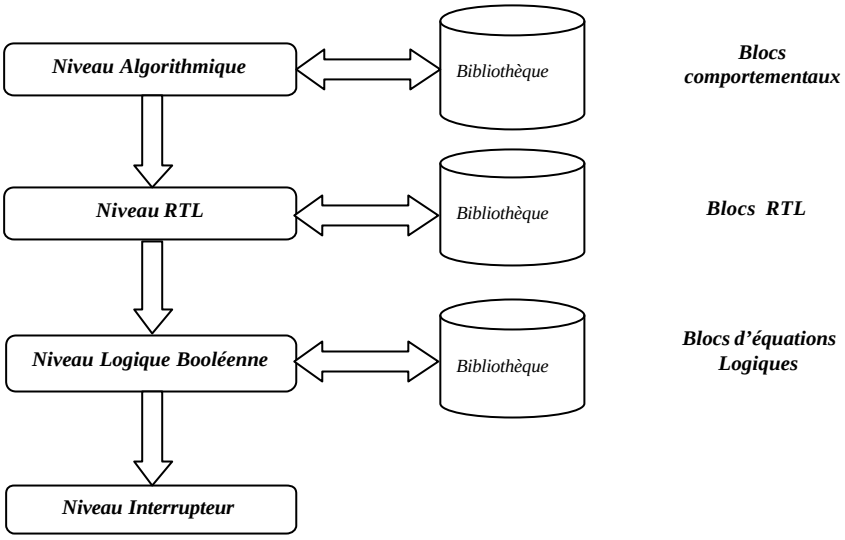


Figure 2.24 - Bibliothèques et niveaux d'abstraction

Les composants se répartissent entre les différents niveaux d'abstraction présentés dans la méthodologie de conception. Ainsi, tout au long du processus on utilise la bibliothèque qui correspond au niveau d'abstraction dans lequel on se trouve (Figure 2.24).

2.4.2 Méthodologie de réutilisabilité

Le concepteur d'une nouvelle architecture doit, au préalable, et tout au long du processus de conception, inspecter la bibliothèque de composants réutilisables à sa disposition pour sélectionner les blocs qui peuvent être intégrés à son application. Pour se faire, il doit être capable de comprendre les caractéristiques du module, de comparer ses performances avant de choisir le module le plus adéquat.
Le module choisi étant en règle générale sous une forme générique, il faut maintenant le personnaliser pour en faire un module spécifique à l'application visée.
La dernière étape consiste en l'intégration du module dans l'architecture globale. L'un des problèmes de l'intégration de composants extérieurs dans un système est le problème de la communication entre les deux ensembles. Pour résoudre cette difficulté, une description commune en VHDL synthétisable (RTL) peut permettre au concepteur d'instancier chaque composant en utilisant les outils de synthèse classique. Ensuite, pour intégrer le composant, le concepteur doit comprendre son interface, c'est-à-dire son protocole de communication avec le monde extérieur, et l'insérer dans le reste du système.

2.4.3 Les composants virtuels IPs

Les efforts actuels visant à standardiser la conception et l'intégration des composants réutilisables permettent de les cataloguer suivant une taxinomie uniforme. De ces travaux se dégage aussi le concept de composants virtuels, les composants IPs (pour Intellectual Property).
Un tel composant correspond à la fois à la spécification d'une fonction plus ou moins complexe mais aussi à un ensemble de services pour :

- l'évaluation de performances de l'*IP* en vue de son intégration dans un système.
- les scripts de synthèse (logique, physique) qui garantissent à l'utilisateur de l'*IP* d'atteindre les mêmes performances physiques que celles obtenues par le fournisseur.
- les fichiers de test, à tous les niveaux d'abstraction, qui permettront de valider l'intégration de l'*IP*.

On trouve trois types d'IP suivant leur flexibilité :

- les *IPs* « softwares » ou souples sont spécifiés au niveau *VHDL RTL* (ou équivalent). Ils sont intégrés lors d'une étape de synthèse logique, ce qui assure à ces composants une indépendance totale vis à vis de la technologie (*ASIC*, *FPGA*).
- les *IPs* « firm » ou semi-durs sont conçus par une synthèse physique au niveau des plans de masse ; Ils reposent sur l'emploi de technologies génériques et sont donc indépendant de la taille des transistors CMOS, chez un même fondeur.
- Les *IPs* « *hardware* » ou durs, sont quant à eux dépendants d'une même technologie.

Si la flexibilité décroît des IPs souples aux IPs durs, la garantie des caractéristiques finales des IPs après layout (performances temporelles, consommation, surface etc.) tend naturellement dans le sens inverse.
Les principaux critères de qualité d'un IP sont donc la qualité et la clarté du code commenté, les scripts de synthèse et de simulation « testbench » ainsi qu'une documentation complète.

2.5 IPs Paramétrables et SOPC

Au niveau matériel, la réalisation d'un système sur une puce s'est démocratisée grâce aux FPGA de forte complexité alliée aux blocs IPs. Par contre, le développement du logiciel embarqué dans ces SOC, comportant souvent plusieurs cœurs configurables, exige une remise à niveau des outils de compilation, d'assemblage et de débogage.
Jusqu'à une époque récente, la conception d'un système sur une puce (SOC) n'était accessible qu'aux sociétés de grande taille, du fait des multiples exigences qu'elle induisait : complexité et diversité des compétences requises, coûts élevés des outils de développement matériel et logiciel, et forts volumes nécessaires pour justifier le coût de conception d'un ASIC. Aujourd'hui, deux événements ont changé la donne : l'avènement des dernières générations de FPGA et la disponibilité de blocs de propriété intellectuelle (IP), utilisables notamment sur ces matrices programmables, mettent la technologie SOC à la portée d'un public nettement plus large.
L'approche SOC (généralement de technologie ASIC) répond aux besoins de performances et d'intégration mais elle présente quelque défauts :

- Elle est peu adaptée à l'évolutivité des systèmes
- Elle reste réservée aux grands volumes de production
- La fabrication et le test sont des étapes longues et coûteuses

L'approche SOPC (technologie FPGA) résoud ces problèmes :

- Développement et prototypage rapides
- Composant reconfigurable en quelques ms et à volonté

Mais

- La densité d'intégration est limitée (quelque Millions de portes)
- La consommation d'énergie est plus importante
- Les performances sont limitées

2.6 Reconfigurabilité dynamique

Longtemps utilisés uniquement pour le prototypage, les circuits reconfigurables de type FPGA (*Field Programmable Gate Array*) ont su s'imposer aujourd'hui comme une alternative entre les solutions dédiées très performantes (ASIC) et les solutions programmables très flexibles (microprocesseurs et DSP). Aujourd'hui, les fabricants de FPGA mettent à la disposition de leurs clients les dernières technologies disponibles, offrant des performances et des capacités permettant l'implémantaion de la plupart des applications courantes. De plus les FPGAs actuels sont de plus en plus aptes à être reconfigurés par des techniques complexes comme la reconfiguration partielle et dynamique voire même l'auto-reconfiguration. Ces

dernières possibilités augmentent de façon conséquente la flexibilité et les possibilités des FPGA.

2.7 Conclusion

Au niveau matériel, la réalisation d'un système sur puce, aujourd'hui, est plus accessible grâce aux FPGA de forte complexité alliées aux blocs IPs. Par contre, le développement du logiciel embarqué dans ces SOC, comportant souvent plusieurs cœurs configurables, exige une remise à niveau des outils de compilation, d'assemblage et de débogage.

A l'heure actuelle, la plupart des éditeurs de logiciels de CAO centrent leurs efforts sur le marché haut de gamme, proposant des outils coûteux à un nombre limité de développeurs, tandis que les fournisseurs d'outils logiciels enfuis s'intéressent avant tout au marché principal. Résultat : il est à présent possible de concevoir et de fabriquer des systèmes matériels complexes comprenant des cœurs configurables hétérogènes, mais on constate souvent l'absence d'outils de développement logiciel prenant en charge tout le potentiel de tels environnements matériels.

Réussir une application à base de FPGA, c'est respecter un certain nombre de règles comme :

- Bien connaître les caractéristiques du FPGA ciblé pour assurer son adéquation avec les besoins du projet
- Respecter une méthodologie de conception et un style d'écriture
- Opter pour des outils de synthèse de qualité
- Maîtriser les outils d'implémantation

Autrement dit, il faut bien comprendre les possibilités offertes par le réseau logique programmable choisi ainsi que les moyens à mettre en œuvre pour en tirer le meilleur profit.

Enfin, il est important de rappeler quelques notions de base sur la méthodologie à adapter dans le cadre de conception d'électronique numérique :

- Eviter les conceptions asynchrones, car elles présentent facilement des défauts de fonctionnement aléatoire. Certaines manifesteront des problèmes de fonctionnement en température, de vieillissement de composant ou d'autres facteurs externes
- Ne pas générer une horloge à partir d'une fonction combinatoire
- Ne pas générer un signal de contrôle à partir d'une fonction combinatoire.

CHAPITRE 3

ALGORITHMES EVOLUTIONNAIRES MONO ET MULTIOBJECTIFS

Les concepteurs sont confrontés quotidiennement à des problèmes de complexité grandissante, qui surgissent dans des secteurs très divers : la recherche opérationnelle, la conception de systèmes électroniques (et tout particulièrement la conception de systèmes sur puce dans le cadre de ce mémoire), etc. le problème à résoudre peut souvent être considéré comme un problème d'optimisation : on définit une (ou plusieurs) fonction objectif, ou fonction coût, que l'on cherche à optimiser par rapport à tous les paramètres concernés. La définition du problème d'optimisation est souvent complétée par la donnée de contraintes : tous le paramètres des solutions retenues doivent respecter ces contraintes, faute de quoi ces solutions ne sont pas réalisables.

3.1 Optimisation difficile

On distingue deux types de problèmes d'optimisation : les problèmes « discrets » et les problèmes à variables continues. Dans le premier cas, on trouve le célèbre problème du voyageur de commerce : il s'agit de minimiser la longueur de la tournée d'un voyageur de commerce, qui doit visiter un certain nombre de villes, avant de retourner à la ville de départ. Un exemple de problème continu est celui de la recherche des valeurs à affectées aux paramètres d'un modèle numérique de processus, pour que ce modèle reproduise au mieux le comportement réel observé.

Des efforts ont longtemps été menés, séparément, pour résoudre ces deux types de problèmes. Dans le domaine de l'optimisation continu, il existe ainsi un arsenal important de méthodes classiques dites d'optimisation globale, mais ces techniques sont souvent inefficaces si la fonction objectif ne possède pas une propriété structurelle particulière, telle que la convexité. Dans le domaine de l'optimisation discrète, un grand nombre d'heuristiques, qui produisent des solutions proches de l'optimum, on été développées ; mais la plus part d'entre elles ont été conçues spécifiquement pour un problème donné.

3.2 Métaheuristiques

La naissance d'une nouvelle classe de méthodes, nommées métaheuristiques, marque une réconciliation des deux domaines : en effet, celles-ci s'appliquent à toutes sortes de problèmes combinatoires, et elles peuvent également s'adapter aux problèmes continus. Ces méthodes, qui comprennent notamment la méthode du recuit simulé, les algorithmes génétique, la méthode de recherche tabou, etc. sont apparues, à partir des années 1980 , avec une ambition commune : résoudre au mieux les problèmes d'optimisation difficile. Elles ont en commun, en outre, les caractéristiques suivantes :

- elles sont, au moins pour une partie, stochastiques : cette approche permet de faire face à l'explosion combinatoire des possibilités ;
- elles sont d'origine combinatoire : elles ont l'avantage, décisif dans le cas continu, d'être directes, c'est-à-dire qu'elles ne recourent pas au calcul, souvent problématique, des gradients de la fonction objectif ;
- elles sont inspirées par des analogies : avec la physique (recuit simulé, diffusion simulée, etc.), avec la biologie (algorithmes génétiques, recherche tabou, etc.) ;

- elles sont capables de guider, dans une tâche particulière, une autre méthode de recherche spécialisée (par exemple, une autre heuristique, ou une méthode d'exploration locale) ;
- elles partagent aussi les mêmes inconvénients : les difficultés de réglage des paramètres même de la méthode et le temps de calcul élevé.

Les métaheuristiques ne s'excluent pas mutuellement : en effet, dans l'état actuel de recherche, il est plus souvent impossible de prévoir avec certitude l'efficacité d'une méthode donnée, quand elle est appliquée à un problème donné.

3.3 Capacité des métaheuristiques à s'extraire d'un minimum local

Pour surmonter l'obstacle des minima locaux, une idée s'est montrée très fructueuse, au point qu'elle est à la base de toutes les métaheuristiques dites de voisinage (recuit simulé, méthode tabou) : il s'agit d'autoriser, de temps en temps, des mouvements de remontée, autrement dit d'accepter une dégradation temporaire de la situation, lors du changement de la configuration courante. C'est le cas si l'on passe de C_n à C_n' (Figure 3.1). Un mécanisme de contrôle des dégradations, spécifique à chaque métaheuristique, permet d'éviter la divergence du procédé. Il devient, dès lors, possible de s'extraire du piège que représente un minimum local, pour partir explorer une autre « vallée » plus prometteuse.

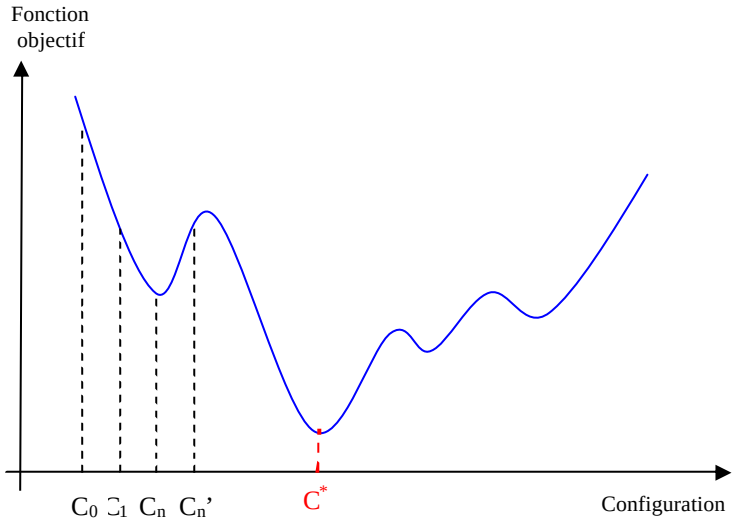


Figure 3.1 – Allure de la fonction objectif d'un problème d'optimisation difficile en fonction de la configuration

Les métaheuristiques « distribuées » (telles que les algorithmes génétiques) ont-elles aussi des mécanismes permettant la sortie hors d'un « puits » local de la fonction objectif. Ces mécanismes (comme la mutation dans les algorithmes génétiques) affectant une solution viennent, dans ce cas, seconder le mécanisme collectif de lutte contre les minima locaux, que représente le contrôle en parallèle de toute une population de solutions.

3.4 Fonction d'objectif et fonction d'adaptation

Les grandeurs à optimiser pour un processeur et dans le cadre de notre travail, peuvent être la durée de l'exécution de l'application, la surface en silicium, la consommation d'énergie, la taille des ressources embarqué (taille du cache, nombre d'unité fonctionnelle, paramètres de pipeline, etc.). Un algorithme d'optimisation nécessite la définition d'une fonction rendant compte de la pertinence des solutions potentielles, suivant les grandeurs à optimiser. Il s'agit de la *fonction d'adaptation f* (ou *fitness function* en terminologie anglo-saxonne). L'algorithme convergera vers un optimum de cette fonction, quelle que soit sa

définition. La pertinence de la solution dépendra donc de la pertinence de la *fonction d'adaptation*. La fonction *f* doit donc traduire en langage mathématique le désire de l'utilisateur.

3.5 Optimisation monobjectif

Dans le cadre d'un *objectif unique*, la définition de la fonction d'adaptation ne pose généralement pas de problème. Par exemple, si l'on se fixe l'objectif de trouver une configuration du processeur dont la durée d'exécution est le minimale, cette fonction sera la durée d'exécution de l'application sur le processeur. Dans cette section nous présentons un bref présentation des méthodes métaheuristiques les plus populaires tel que la méthode de recuit simulé et la méthode de recherche avec tabous. Enfin, dans cadre des algorithmes évolutionnaires, nous étudions en détaillée les algorithmes génétiques.

3.5.1 La méthode du recuit simulé

La méthode du recuit simulé (*Simulated Annealing*) est une récente technique d'optimisation combinatoire multi-variables. Le principe de base de cette méthode est emprunté à la métallurgie : on chauffe un morceau de métal, puis on le laisse refroidir lentement. Le refroidissement lent et régulier du métal permet aux atomes de se stabiliser peu à peu dans une position d'énergie minimale. Le procédé peut être décrit de la manière suivante. On considère un solide en équilibre thermique à une température R. l'état du solide étant caractérisé par sa configuration atomique et son énergie. La probabilité que cette particule soit dans un état d'énergie E, est fournie par la loi de Boltzmann :

$$p = \frac{1}{Z}\exp\left(\frac{-E}{k.T}\right) \qquad (3.1)$$

Où Z représente un facteur de normalisation connu sous le nom de fonction de partition. k est la constante de Boltzmann tandis que le terme exponentiel est connu sous le terme de facteur de Boltzmann. Une modification de la position de ses atomes modifie son état interne et donc, sa probabilité d'existence. Lorsque la température diminue, le caractère probabiliste de la distribution de Boltzmann privilégie les états de moindre énergie.
Ici nous allons minimiser, non pas l'énergie d'un cristal, mais une fonction de coût caractéristique du problème, où une solution est d'autant meilleure que son coût est faible.

Le recuit simulé est une technique d'exploration stochastique qui a actuellement beaucoup de succès. Les algorithmes de recuit simulé font évoluer une seule *pseudo-solution*. Cette caractéristique majeure fait d'ailleurs du recuit simulé une méthode fondamentalement séquentielle [4, 5].
Outre l'analogie avec la physique statistique, cette méthode est construite sur la théorie mathématique des chaînes de Markov qui lui procurent des propriétés de convergence asymptotique. Il est en effet possible de démontrer que la convergence vers l'optimum global est garantie pour un nombre infini d'itérations [6]. Pour des applications pratiques nécessitant des temps de calcul limités, une bonne solution, proche de l'optimum global, peut être obtenue pratiquement indépendamment des conditions initiales et de la taille du problème. L'algorithme explore dans ce cas seulement une fraction de l'espace total de recherche tout en délivrant une bonne solution.

3.5.2 La méthode de recherche avec Tabous

La recherche tabous (*Tabu Search*) est une méthode de recherche locale avancée qui fait appel à un ensemble de règles et de mécanismes généraux pour guider la recherche de manière intelligente [7, 8, 9, 10].
Le principe de base de la méthode tabou est simple : comme le recuit simulé, la méthode tabou fonctionne avec une seule configuration courante à la fois (au départ, une solution quelconque) qui est actualisée au cours des itérations successive. A chaque itération, le mécanisme de passage d'une configuration (s) à la suivante (t), comporte deux étapes :

- On construit l'ensemble des voisins de s, c'est-à-dire l'ensemble des configurations accessibles en un seul mouvement élémentaire à partir de s (si cet ensemble est trop vaste, on applique une technique de réduction de sa taille : par exemple, on a recours à une liste de candidats, ou on extrait aléatoirement un sous-ensemble de voisins de taille fixe) ; soit V(s) l'ensemble (ou le sous-ensemble) de ces voisins ;
- On évalue la fonction objectif f du problème en chacune des configurations apparentent à V(s). la configuration t, qui succède à s dans la suite de solutions construite par la méthode tabou, est une configuration de V(s) en laquelle f prend la valeur minimale. Cette configuration t est adoptée même si elle est moins bonne que s, i.e. si f(t)>f(s). C'est grâce à cette particularité que la méthode tabou permet d'éviter le piégeage dans les minimums locaux de f.

Telle quelle, la procédure précédente est inopérante, car il y a un risque important de retourner à une configuration déjà retenue lors d'une itération précédente, ce qui engendre un cycle. Pour éviter ce phénomène, on tient à jour et on exploite, à chaque itération, une liste de mouvements interdits, dite « liste de tabous » : cette liste, qui a donné son nom à la méthode, contient m mouvements (t => s), qui sont les inverses des m derniers mouvements (s => t) effectués.

La méthode de la recherche tabou existe aussi en version multiobjectifs qui a été proposé par X. Gandibleux et al. en 1996 [83].

3.5.3 Algorithme génétique

Les algorithmes génétiques sont des algorithmes évolutionnaires, les individus soumis à l'évolution sont des solutions, plus ou moins performantes, à un problème posé. Ces solutions appartiennent à l'espace de recherche du problème d'optimisation. L'ensemble des individus traités simultanément par l'algorithme évolutionnaire constitue une population. Elle évolue durant une succession d'itérations appelées générations jusqu'à ce qu'un critère d'arrêt, qui prend en compte a priori la qualité des solutions obtenues, soit vérifié. Durant chaque génération, une succession d'opérateurs est appliquée aux individus d'une population pour engendrer la nouvelle population à la génération suivante. Lorsqu'un ou plusieurs individus sont utilisés par un opérateur, on convient de les désigner comme des parents. Les individus résultant de l'application de l'opérateur sont des enfants. Ainsi, lorsque deux opérateurs sont appliqués en séquence, les enfants engendrés par l'un peuvent devenir des parents pour l'autre.

Dans le chapitre 5.2.2, nous avons employé l'algorithme génétique pour l'exploration des paramètres d'un processeur d'architecture SuperScalar. D'autre part, dans le chapitre 6.5.3, et dans le cadre d'une exploration monobjectif, nous avons utilisé l'algorithme génétique pour l'exploration d'un circuit analogique (filtre passe-bas). Donc pour faciliter la compréhension des parties expérimentales, dans les chapitres suivants, nous présentons dans cette section une étude approfondie de l'algorithme génétique « AG ».

3.5.3.1 Introduction à l'AG

Les algorithmes génétiques sont inspirés de la génétique classique, voici quelques mots de vocabulaire relatifs à la génétique :
Individu : correspond au codage sous forme de gènes d'une solution potentielle à un problème d'optimisation.
Génotype ou *chromosome* : c'est une autre façon de dire « individu ».
Gène : un chromosome est composé de gènes. Dans le codage binaire, un gène vaut soit 0 soit 1.
Phénotype : chaque génotype représente une solution potentielle à un problème d'optimisation. La valeur de cette solution potentielle est appelée phénotype.

L'algorithme génétique fournit des solutions à un problème n'ayant pas de solution analytique ou algorithmique. Selon cette méthode, des millieres de solutions *« génotypes »* plus ou moins bonnes sont générées aléatoirement, puis soumises à une imitation de l'évolution des espèces : les plus adaptés survivent davantage que les autres et la population évolue par générations successives avec des mutations, croisement et reproduction.
La population initiale donne ainsi naissance à des générations successives. Un mécanisme de favorisation des éléments les plus aptes assure que les générations successives sont de plus en plus adaptées à la résolution du problème. Le mécanisme d'évolution et de sélection est indépendant du problème à résoudre : seules varient la fonction qui décode le génotype en une solution possible (tout décodage satisfaisant peut être utilisé, de préférence le plus simple possible) et celle qui évalue l'adaptation (fréquemment en la testant sur quelques centaines de cas).

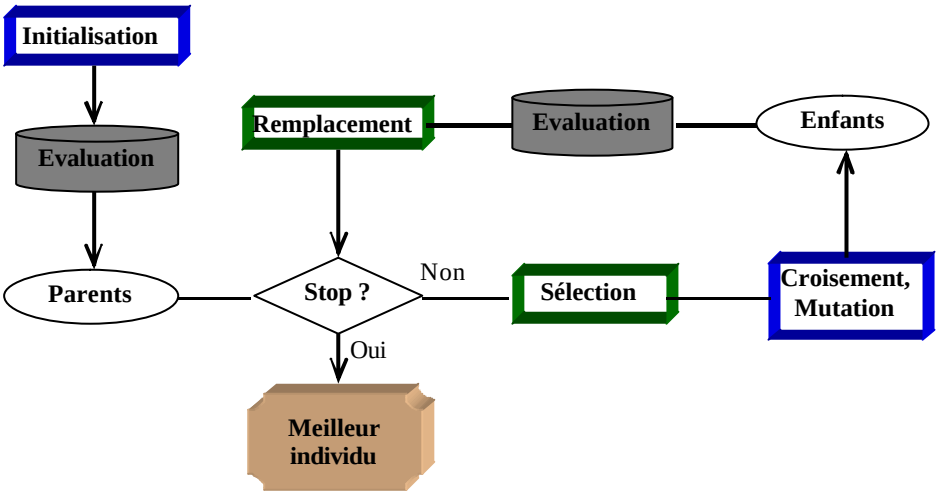


Figure 3.2 - Squelette de l'algorithme évolutionnaire

L'algorithme génétique, n'exige aucune connaissance de la manière pour résoudre le problème; il est seulement nécessaire d'estimer la qualité d'une solution potentielle. Il s'agit d'une approche un peu brutale nécessitant une grande puissance de calcul, mais présentant l'immense avantage pratique de fournir des solutions pas trop éloignées de l'optimal même si l'on ne connaît pas de résolution algorithmique. Il est également léger à mettre en œuvre (le « moteur » est commun, il y a peu de code spécifique au problème à écrire).

3.5.3.2 Algorithme génétique simple « AGS »

Les Algorithmes Génétiques (AGs) représentent une famille assez riche et très intéressante d'algorithmes d'optimisation stochastique fondés sur les mécanismes de la sélection naturelle et de la génétique. Les champs d'application sont fort diversifiés. On les

retrouve aussi bien en théorie des graphes qu'en compression d'images numérisées ou encore en programmation automatique. Les raisons de ce nombre d'applications sont claires. Leur principe est d'opérer une recherche stochastique sur un important espace à travers un ensemble « de population » de pseudo-solutions. Ces algorithmes sont simples et très performants dans leur recherche d'amélioration. De plus, ils ne sont pas limités par des hypothèses contraignantes dans le domaine d'exploration. Ainsi, le mathématicien abordant le sujet n'a guère à se préoccuper de la continuité et de la différentiabilité de la fonction à optimiser. Une remarque importante qui en sort est qu'une exploration pseudo-aléatoire n'implique pas nécessairement une exploration sans direction. En résumé, les algorithmes génétiques diffèrent fondamentalement des autres méthodes selon quatre axes principaux:

1. Les AGs utilisent un codage des paramètres et non les paramètres eux-mêmes
2. Les AGs travaillent sur une population de points au lieu d'un point unique
3. Les AGs n'utilisent que les valeurs de la fonction étudiée, pas sa dérivée ou une autre connaissance auxiliaire
4. Les AGs utilisent des règles de transition probabilistes et non déterministes

Les principaux éléments du jargon pour les AGs qu'on trouve dans la littérature sont les suivants :

- Individu ou Chromosome : une solution potentielle
- Population : un ensemble de chromosomes ou de points de l'espace de recherche
- Environnement : l'espace de recherche
- Fitness ou Fonction d'évaluation : la fonction positive que l'on cherche à maximiser

Ce petit lexique, qui montre une prudente analogie avec la biologie, a naturellement une simple fonction métaphorique. Nous les utilisons dans l'introduction de l'algorithme génétique standard, considéré désormais par plusieurs auteurs sous le nom d'Algorithme Génétique Canonique (AGC), défini par Holland et redynamisé par Goldberg.

3.5.3.3 Algorithme Génétique Canonique « AGC »

Le problème de l'AGC peut être résumé de la manière suivante : on dispose d'une fonction f, non constante, définie sur le cube logique $\{0,1\}^N$ avec N entier naturel, à valeurs dans $\Re_+^*$ et on se fixe pour objectif de localiser l'ensemble f^* des maxima globaux de f, ou à défaut, de trouver le plus rapidement et le plus efficacement possible des points sous-optimaux.

Formellement, il s'agit d'une optimisation sans contrainte du type :

$$\max \quad \{ f(x) / x \in \{0,1\}^N \quad \} \qquad (3.2)$$

Très souvent, en pratique, on s'intéresse plutôt aux minima d'une fonction f (exemple la durée d'exécution d'une application) et, dans ce contexte, les résultats suivants sont utilisés.

Soit f une fonction réelle a une ou plusieurs variables. Les problèmes d'optimisation suivants sont équivalents:

$$\min_x f(x) \qquad (3.3)$$

$$\max_x \{ - f(x) \} \qquad (3.4)$$

De plus, la fonction optimisée par un AG doit avoir des valeurs positives dans l'espace de recherche. Dans le cas contraire, il est nécessaire d'ajouter aux valeurs de f une constante positive adéquate C conformément à l'équivalence des problèmes d'optimisation suivants:

$$\max_x f(x) \qquad (3.5)$$

$$\max_x \{C + f(x)\}, C \in \Re \qquad (3.6)$$

En définitive, l'objectif de l'AG est de régler les différents paramètres d'un problème donné pour obtenir la plus grande valeur du fitness f possible.

3.5.3.4 Principe de l'AGC

L'organigramme de la Figure 3.3 montre succinctement le fonctionnement de l'AGC et indique clairement ses différents opérateurs de base. Ces opérateurs sont inspirés directement des mécanismes de la sélection naturelle et des phénomènes génétiques. Ainsi, le principe de l'AGC (ce qui d'ailleurs justifie son nom) est de faire évoluer une population de façon à adapter les individus à l'environnement en place, comme le ferait un ensemble d'êtres vivants. Ce principe s'apparente convenablement à celui de la *théorie de Darwin* sur l'évolution, selon laquelle « la vie est une compétition et seuls les mieux adaptés survivent et se reproduisent ».

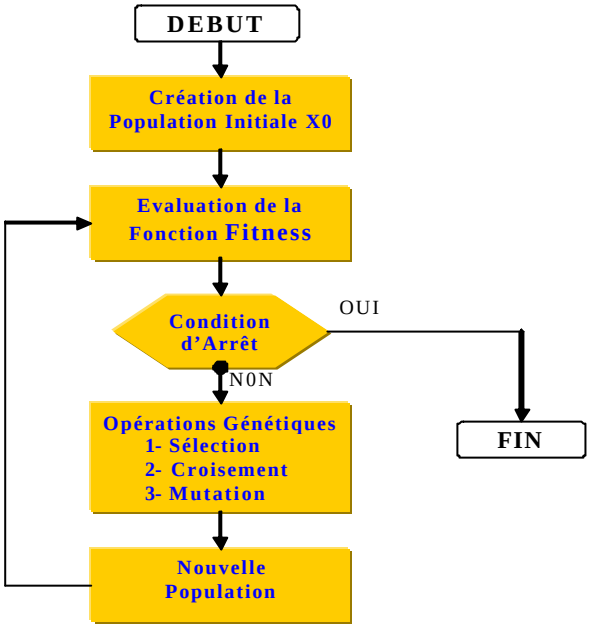


Figure 3.3 - Organigramme de l'AGC

Techniquement, une nouvelle génération s'obtient généralement au bout d'un cycle (Figure 3.3) composé de trois opérateurs standards: Reproduction, Croisement et Mutation.

3.2.4.5 Représentation binaire des individus

C'est l'idée de faire évoluer une population dans un espace de vecteurs binaires. Cette stratégie s'inspire de la transcription *génotype-phénotype* qui existe dans le monde vivant. Dans le cadre des algorithmes génétiques, le génotype est constitué d'une chaîne de symboles binaires ou plus généralement de symboles d'un alphabet à faible cardinalité. Le phénotype est une solution au problème dans une représentation naturelle. Le génotype subit l'action des opérateurs génétiques, tandis que le phénotype ne sert qu'à l'évaluation de la performance d'un individu. Par exemple, si une solution s'exprime naturellement sous forme d'un vecteur de nombres entiers, le phénotype sera ce vecteur. Le génotype sera ainsi une chaîne de symboles binaires qui code ce vecteur. Pour coder l'ensemble des variables entiers d'un problème numérique dans une chaîne de symboles binaires, le plus simple est de convertir chaque variable sous forme binaire, puis ces nombres binaires sont concaténés pour former le génotype.

3.5.3.5 Reproduction

L'opérateur de reproduction directement inspiré de la sélection naturelle consiste à dupliquer chaque individu de la population proportionnellement à sa valeur d'adaptation « sélection proportionnelle ». Il s'agit, en quelque sorte, de réaliser autant de tirages avec

remise (à la manière de la technique du *Bootstrap*), qu'il y a d'éléments dans la population. Chaque élément étant tiré avec une probabilité liée à sa valeur d'évaluation que le statisticien peut bien interpréter comme des probabilités de survie. Ainsi, les individus ayant un fitness grand ont plus de chance d'être sélectionnés pour la génération suivante. On parle alors de sélection proportionnelle.
La probabilité pour l'individu Xi soit sélectionné à partir de la population (X1, X2,…, Xn) est représenté par l'équation suivante :

$$P_{Sélection}\{Xi\} = \frac{f(Xi)}{\sum_{j=1}^{n} f(Xj)} > 0 \qquad (3.7)$$

L'opérateur de reproduction peut être mis en oeuvre sous forme algorithmique de différentes façons. La plus facile est de créer une roue de loterie biaisée (Figure 3.4) pour laquelle chaque individu de la population courante occupe une section de la roue proportionnelle à son adaptation.

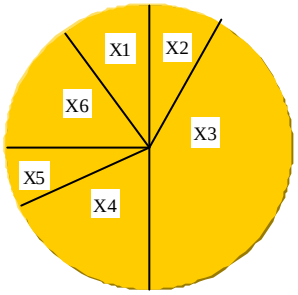


Figure 3.4 - Roue de loterie biaisée

Pour réaliser la reproduction, il suffit de faire tourner la roue biaisée ainsi définie autant de fois qu'on a besoin de descendant. Le groupe ainsi obtenu constitue la nouvelle génération qui est ensuite soumise à d'autres opérateurs génétiques.
Etant donné un fitness positif f (fonction Objectif) et une taille de la population n, une telle reproduction peut être décrite schématiquement par le pseudo code suivant:

```
Début
  Evaluer le fitness f_i de chaque individu X_i
  Calculer F = ?^n_{i=1}(f_i)  /*le fitness total de la population*/
  Pour tout i= 1 jusqu'à n
    pi = f_i/F /*probabilité de sélection de X_i*/
    q_i = 0
    Pour tout j= 1 jusqu'à i
      q_i = q_i + p_j /*q_i fixe la probabilité cumulative de X_i*/
    Fin pour
  Fin pour
  Pour tout i= 1 jusqu'à n
    P1 réel tiré dans [0,1]
    Si p_1 < q_1
      Sélectionner X_1
    Sinon
      Sélectionner X_k, 2 = k = n tel que q_{k-1} < p_1 = q_k
    Fin si
  Fin pour
Fin
```

Algorithme 3.1 - le pseudo code de la roue biaisée

L'inconvénient majeur de ce type de reproduction vient du fait qu'il peut favoriser la domination d'un individu qui n'est pas véritablement le meilleur. De plus, a peut engendrer une perte de diversité par la domination d'un super-individu. Dans l'exemple ci-dessus (Figure 3.4) où la population courante est {X1, X2, X3, X4, X5, X6}. On peut malheureusement obtenir à la génération suivante, une population constituée du même individu. Par exemple {X4, X4, X4, X4, X4, X4}, qui n'est manifestement pas une « bonne » population. Pour pallier à cet inconvénient, on préfère très souvent des méthodes de reproduction plus justes qui n'autorisent en aucun cas la naissance d'un super-individu. Par exemple, la sélection proportionnelle au rang ou d'autres techniques faisant intervenir des notions de voisinage entre chromosomes.
Après la reproduction, un croisement permet de générer les nouveaux individus.

3.5.3.6 Croisement

Relativement à deux individus « *parents* », considérés à cette occasion comme des vecteurs, la dynamique de l'opérateur croisement, inspirée directement du processus de multiplication des chromosomes humains, consiste à générer deux nouveaux individus « *enfants* » de la façon suivante :

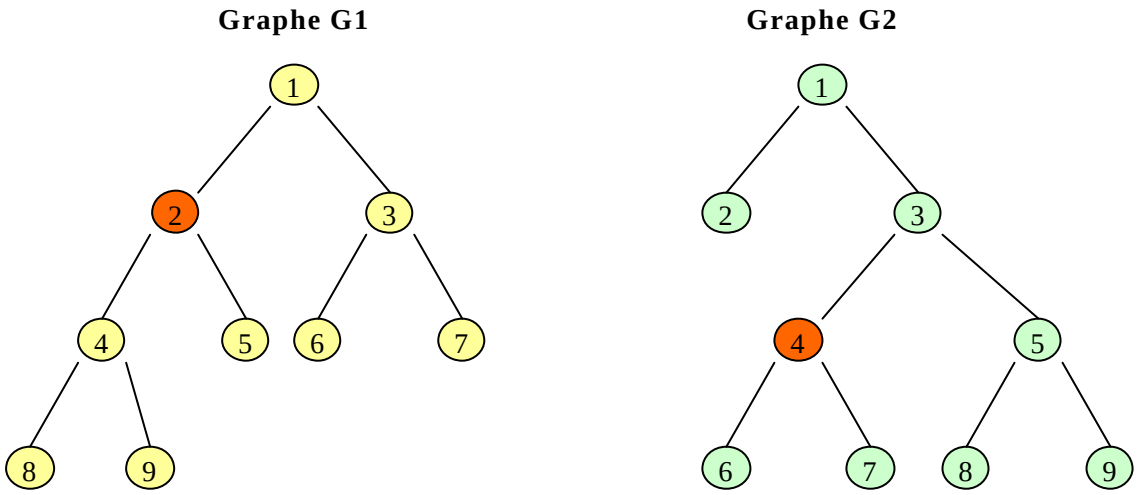


Figure 3.5 - Graphes parents à croiser

Croisement à un site : Un site de croisement est tiré aléatoirement sur l'individu, puis les segments finaux des deux parents sont échangés autour de ce site. Illustrons cette opération dans le cas où notre espace de recherche serait un espace d'arbres dont l'étiquetage des nœuds se fait de gauche à droite. Pour cela, considérons deux graphes G1 et G2 ayant 9 nœuds (Figure 3.5). En supposant que le site de croisement issu d'un tirage au sort est le nœud 2 (respectivement 4) pour le graphe G1 (respectivement G2), on obtient deux enfants G3 et G4 (Figure 3.6).

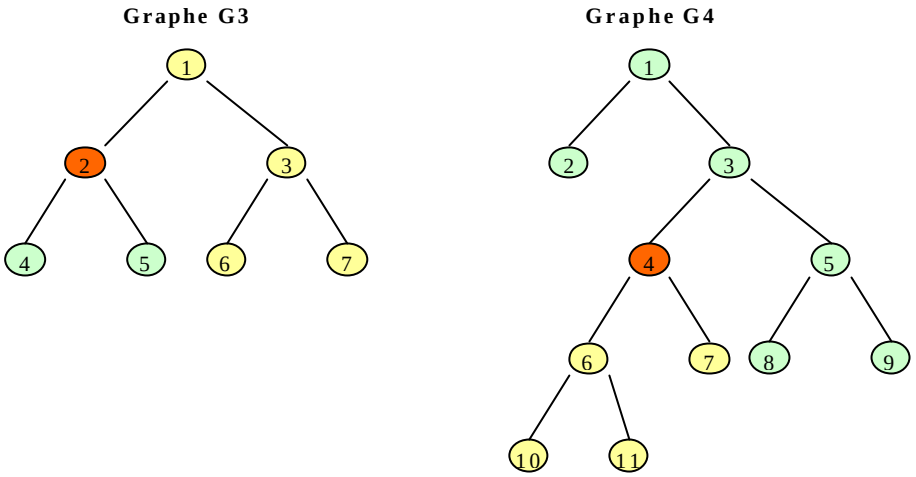


Figure 3.6 - Graphes Enfants issus du croisement

Croisement uniforme : Chaque composante du nouveau vecteur est tirée aléatoirement parmi les composantes des parents ayant le même indice. Généralement le second enfant est construit en prenant le complémentaire du premier enfant comme l'indique la Figure 3.7.

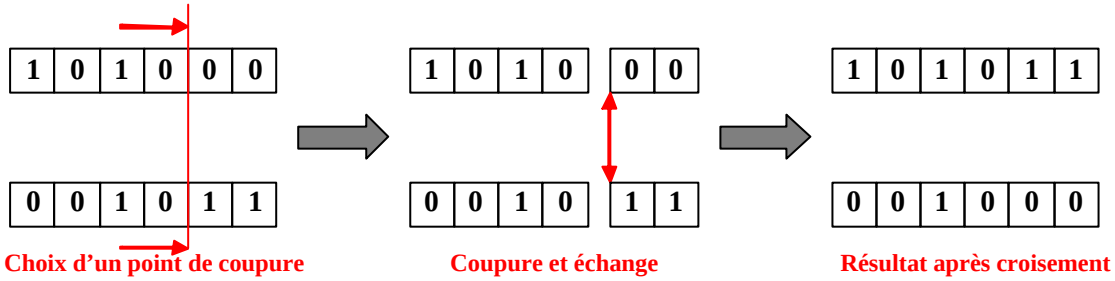


Figure 3.7 – Croisement en « un point » de deux génotypes binaires de 6 bits

Les opérations de croisement que nous venons de décrire sont les plus utilisés dans la littérature des AGs. Il en existe bien d'autres tels que le croisement multi-site ou bien des croisements spécialisés liés à la sémantique du problème traité et/ou à la structure du codage considéré.

3.5.3.7 Mutation

L'opération de mutation consiste à modifier de manière aléatoire, avec une probabilité généralement faible, la valeur d'une composante (gène) de l'individu. Dans le cas du codage binaire de l'AGC, un booléen $w \in \{0,1\}$ choisi aléatoirement est remplacé par son inverse 1-w. Techniquement, la combinaison de la reproduction et du croisement est suffisante pour assurer l'évolution de la population, mais il peut arriver que certaines informations essentielles (composantes dans les cas des vecteurs) disparaissent de la population. Le rôle premier de la mutation est donc de pallier à cet inconvénient. Prise isolément, la mutation constitue une exploration aléatoire de l'espace des solutions. Mais une utilisation parcimonieuse avec la reproduction et le croisement constitue une police d'assurance protégeant la perte prématurée des composantes importantes.
Du rôle secondaire de l'opérateur de mutation dans l'AGC, nous retiendrons que la fréquence de mutation conseillée pour obtenir de bons résultats dans les études empiriques se situe autour d'une mutation tous les 1000 bits (positions). Les taux de mutation sont également

faibles dans les populations naturelles, ce qui nous conduit à penser que la mutation est considéré à juste titre comme un mécanisme d'adaptation secondaire pour les AGs.
Nous illustrons à la Figure 3.8, pour un codage binaire, un exemple de schéma de mutation.

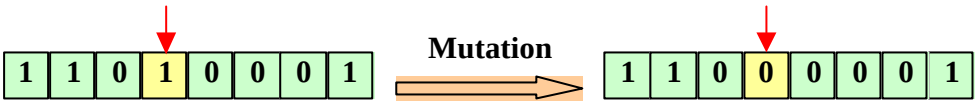


Figure 3.8 - Mutation de gène 1 en 0

Les trois opérateurs (Reproduction, Croisement et Mutation) que nous avons étudiés dans cette partie ont fait la preuve de leur simplicité de mise en oeuvre algorithmique et de leur efficacité pour résoudre un nombre important de problèmes d'optimisation par les AGs.

3.5.3.8 Les paramètres de l'AG

Les opérateurs de l'AGC sont guidés par un certain nombre de paramètres généralement fixés à l'avance et dont dépend très fortement la « bonne » convergence de l'algorithme. Présentons brièvement les principaux paramètres de l'AGC.

- La taille de la population et la longueur du codage de chaque individu. Il est conseillé de prendre comme taille de la population la valeur correspondant à la longueur du codage des individus. En effet, si cette taille est très grande, l'évaluation de tous les individus de la population peut s'avérer trop longue. Par contre, si elle est très petite, l'algorithme peut converger trop rapidement [11].
- La probabilité de croisement Pc dépend de la forme de la fonction d'évaluation. Son choix est généralement expérimental et sa valeur est très souvent prise entre 0.5 et 0.9. Plus elle est élevée, plus la population subit des changements importants. Ainsi, la convergence est très rapide si le taux de croisement est proche de 1.
- La probabilité de mutation Pm : le taux de mutation est généralement faible, le risque d'un taux élevé étant de modifier les meilleurs individus et ainsi, de s'éloigner de l'optimalité.

3.5.3.9 Génération d'une nouvelle population

Les opérateurs génétiques (Sélection, Croisement et Mutation) peuvent constituer une étape directe ou indirecte dans le protocole de création d'une nouvelle génération d'individus. Ainsi, la littérature des AGs offre une collection fort variée de stratégies dont il convient ici d'en citer quelques-unes:

- La nouvelle population est composée uniquement des enfants. On laisse alors disparaître tous les individus de la population courante. L'inconvénient majeur de cette approche est le risque de perdre le meilleur individu lorsqu'on ne le conserve pas automatiquement dans la nouvelle population.
- Les enfants remplacent d'une façon régulière les individus les moins forts de la génération courante.
- La nouvelle génération est constituée des n meilleurs individus de la population intermédiaire formée des enfants et des parents.

3.5.3.10 Elitisme

Une stratégie élitiste consiste à conserver dans la population, d'une génération à l'autre, au moins l'individu ayant la meilleure performance. La performance du meilleur individu de la population courante est ainsi monotone et croissante de génération en génération.

Il apparaît que de telles stratégies améliorent considérablement les performances d'algorithmes évolutionnaires pour certaines classes de problèmes, mais s'avèrent décevantes pour d'autre classes, en augmentant le taux des convergences prématurées.
Choisir une stratégie non élitiste peut être avantageux, mais il n'y a alors pas de garantie que la fonction performance du meilleur individu soit croissante au cours de l'évolution. Cela implique évidement de garder en mémoire la meilleure solution trouvée par l'algorithme depuis le début de l'évolution, sans toutefois que cette solution participe au processus évolutionnaire.

3.5.3.11 Etat de l'art de la convergence de l'AGC

Dans cette partie, on analyse les propriétés de convergence de l'algorithme génétique canonique (AGC) avec la mutation, le croisement et la reproduction proportionnelle appliqués aux problèmes d'optimisation statiques. Il a été démontré au moyen d'analyse de la chaîne de Markov homogène finie qu'un AGC ne convergera jamais à l'optimum global indépendamment de l'initialisation, de l'opérateur de croisement et de la fonction objectif. Mais des variantes de AGCs qui maintiennent toujours la meilleure solution dans la population, avant ou après la sélection, sont montrées pour converger à l'optimum global [12].

3.5.3.11.1 Introduction

Les algorithmes génétiques canoniques (AGC) sont souvent employés pour aborder des problèmes d'optimisation statiques du type:

$$\min\left\{f(x)/\, x \in B^l\right\} \qquad (3.8)$$

$$\text{Avec } 0 < f(x) < \infty \text{ pour tout } x \in B^l = \{0,1\}^l \text{ et } f(x) \neq const.$$

Dans la théorie d'optimisation, on dit qu'un algorithme converge à l'optimum global s'il produit un ensemble de solutions ou des valeurs de fonction dans lesquelles l'optimum global est une valeur limite.

Les chaînes de Markov offrent un modèle approprié pour analyser les AGs. Elles ont été employées dans [13] pour prouver la convergence probabiliste de la meilleure solution dans une population vers l'optimum global sous le choix d'élitiste (le meilleur individu survit avec la probabilité une).

3.5.3.11.2 Chaîne de Markov finie

Une chaîne de Markov finie décrit une trajectoire probabiliste dans un espace d'état fini S de cardinalité |S| = n, où les états peuvent être numérotés de 1 au n. La probabilité de transition $p_{ij}(t)$ de l'état $i \in S$ à l'état $j \in S$ à l'étape t s'appelle la probabilité de transition de i à j à l'étape t. Si les probabilités de transition sont indépendantes de t, $p_{ij}(t) = p_{ij}(s)$ pour tous $i, j \in S$ et pour tout $s, t \in N$, la chaîne de Markov serait homogène.
Les probabilités de transition d'une chaîne de Markov homogène finie peuvent être recueillies dans une matrice de transaction $P = (p_{ij})$. Pour chaque entrée, $p_{ij} \in [0,1]$ et $\sum_{j=1}^{ISI} p_{ij} = 1$ pour tout $i \in S$. Des matrices avec les propriétés ci-dessus s'appellent matrices stochastiques. Avec une distribution initiale P_0 comme vecteur de rangée, la distribution de la chaîne après $t^{ième}$ étape est déterminée par $p^t = p^0 P^t$. Par conséquent, une chaîne de Markov finie homogène est complètement déterminée par la paire (p^0, P). Puisque le comportement de limite de la chaîne de Markov dépend de la structure de la matrice de transition [71, 72].

Définition 1

Une matrice carrée $A: nxn$ est :

(a) non-négative $(A \geq 0)$, si $a_{ij} \geq 0$ pour tout $i, j \in \{1,...,n\}$

(b) positve $(A \succ 0)$, si $a_{ij} \succ 0$ pour tout $i, j \in \{1,...,n\}$

Une matrice non-négative $A: nxn$ est :

(c) primitive, s'il existe $k \in N$ tel que A^k est positive,

(d) Réductible, si A peut être représenté dans la forme suivante (avec matrices carrées C et T) :

$$\begin{pmatrix} C & 0 \\ R & T \end{pmatrix} \qquad (3.9)$$

(e) irréductible, s'elle n'est pas réductible,

(f) stochastique, si $\sum_{j=1}^{n} a_{ij} = 1$ pour tout $i \in \{1,...,n\}$

Une matrice stochastique $A: nxn$ est :

(g) stable, s'il y a des lignes identiques,

(h) à colonne admissible, si elle a au moins une entrée positive dans chaque colonne.

Notez que le produit des matrices stochastiques est encore une matrice stochastique et le produit des matrices positives est également une matrice positive. Les résultats suivants ont installé la base de l'analyse de convergence pour AGs.

Lemme 1

Soient C, M et B, trois matrices stochastiques, avec M est positive et S à colonne admissible. Alors le produit C.M.S est une matrice positive.

Démonstration :

Soit A = C.M et B = A.S. Puisque C est stochastique, donc il existe au moins une entrée positive dans chaque ligne de C. alors, par multiplication des matrices $a_{ij} = \sum_{k=1}^{n} c_{ik}.m_{kj} \succ 0$ pour tout $i, j \in \{1,...,n\}$ c'est-à-dire $A \succ 0$. De même, $b_{ij} = \sum_{k=1}^{n} a_{ik}.s_{kj} \succ 0$ pour tout $i, j \in \{1,...,n\}$, car S est de colonne admissible.

Théorème 1

Soit P une matrice positive stochastique. Alors P^k converge, quand $k \to \infty$, vers une matrice positive stable stochastique :

$P^\infty = 1' p^\infty$,

tel que $p^\infty = p^0 \lim_{k\to\infty} P^k = p^0 P^\infty$ ne contienne pas d'entrées zéro et est unique indépendamment de la distribution initiale.

Théorème 2

Soit P une matrice stochastique réductible, $C: mxm$ est une matrice stochastique primitive et $R, T \neq 0$, alors :

$$P^\infty = \lim_{k\to\infty} \begin{pmatrix} C^k & 0 \\ \sum_{i=0}^{k-1} T^i R C^{k-i} & T^k \end{pmatrix} = \begin{pmatrix} C^\infty & 0 \\ R_\infty & 0 \end{pmatrix} \qquad (3.10)$$

est une matrice stochastique stable avec $P^{\infty} = 1' p^{\infty}$, tel que $p^{\infty} = p^0 P^{\infty}$ est unique indépendamment de la distribution initiale et p^{∞} satisfait : $p_i^{\infty} \succ 0$ pour $1 \le i \le m$ et $p_i^{\infty} = 0$ pour $m \le i \le n$.

3.5.3.11.3 Analyse des algorithmes génétiques avec la chaîne de Markov

L'AGC peut être décrit comme une chaîne de Markov : l'état de l'AGC dépend seulement des gènes des individus de sorte que l'espace d'état soit $S = B^N = B^{l-n}$, où n dénote la taille de la population et l, le nombre de gènes par individu. Chaque élément de l'espace d'état peut être considéré comme un nombre entier dans une représentation binaire. Puisque cette représentation est isomorphe, un élément $i \in S$ est considéré comme dans une représentation binaire ou une représentation entier. La projection $p_k(i)$ représente l'individu k de longueur l bits dans la représentation binaire de l'état i est employé pour identifier les individus dans l'ensemble de la population.

Le changement probabiliste des gènes de chaque individu de la population provoquée par les opérateurs génétiques sont capturés par la matrice de transition P, qui peut être décomposée en un produit des matrices stochastiques $P = C.M.S$, où C, M et S décrivent les transitions intermédiaires provoquées respectivement par le croisement, la mutation et la sélection.

Théorème 3

La matrice de transition de l'AGC avec la probabilité de mutation $p_m \in [0,1]$, probabilité de croisement $p_c \in [0,1]$ et la sélection proportionnelle est une matrice primitive.

Démonstration

L'opérateur de croisement peut être considéré comme une fonction totalement aléatoire dans S, c'est à dire, chaque état de S est représenté avec un certain probabilité dans un autre état de S. Par conséquent, la matrice de croisement C est stochastique. La même propriété est valable pour les autres opérateurs.

Puisque l'opérateur de mutation est appliqué indépendamment, dans la population, à chaque gène/bit, la probabilité de mutation pour que l'état i devient l'état j est représenté par $p_m^{H_{ij}} (1 - p_m)^{N - H_{ij}} \succ 0$ pour tous $i, j \in S$, où H_{ij} dénote ainsi la distance de Hamming entre les représentations binaires de l'état i et j. M est positive.

La probabilité de la sélection ne change pas l'état produit par la mutation. Elle peut être représentée par :

$$s_{ij} \ge \frac{\prod_{k=1}^{n} f(p_k(i))}{\left(\sum_{k=1}^{n} f(p_k(i)) \right)^n} \succ 0 \qquad (3.11)$$

Pour tout $i \in S$, S est une matrice admissible.

D'après le lemme 1, $P = C.M.S$ est positif. Enfin chaque matrice positive est primitive.

Corollaire 1

L'AGC avec le paramètre comme spécifié dans le théorème 3 est une chaîne de Markov ergodique, c'est à dire, il existe une distribution asymptotique unique pour les états de chaîne avec la probabilité non nulle pour tous les états à tout moment, indépendamment de distribution initiale.

Ce résultat découle directement des théorèmes 1 et 3.

La distribution initiale p^0 n'a aucun effet sur le comportement asymptotique de la chaîne de Markov. Par conséquent, de point de vue théorique, l'initialisation de l'algorithme peut être faite arbitrairement et l'opération de sélection effectuée avant le début de la boucle peut être omise: la distribution asymptotique (convergence) demeure la même.
La propriété d'ergodicité détermine le comportement de l'AGC en terme de convergence.

3.6 Optimisation multiobjectifs

Dans les sections précédentes, nous n'avons considéré que les cas où le problème à traiter possédait un objectif unique à optimiser. En pratique, de plus en plus des problèmes exigent la considération simultanée de plusieurs objectifs contradictoires. Il n'existe pas, dans ce cas, un optimum unique ; on recherche, en revanche, une gamme de solutions «optimales au sens de Pareto », qui forme la « surface de compromis » du problème considéré. Ces solutions peuvent être soumises à l'arbitrage final de l'utilisateur [97, 87].

Les algorithmes génétiques, avec un bon choix de leurs paramètres de réglage, sont bien placés pour résoudre un certain calasse des problèmes d'optimisations multiobjectifs [83, 84]. De plus, ce domaine est très dynamique et ne cesse de se développer. Il y a plusieurs méthodes qui ont été mises en œuvre pour le traitement des problèmes d'optimisations multiobjectifs. Ces méthodes se décomposeront en deux familles :

- une première famille comportera les méthodes «agrégatives », le but de ces méthodes est de se ramener à un problème d'optimisation monobjectif en utilisant une fonction objectif équivalente.
- La deuxième famille comportera les méthodes dites « non-agrégatives », dans ces méthodes, il n'y a pas fusion des différentes fonctions objectifs pour se ramener à un problème d'optimisation monobjectif.

3.6.1 Objectifs multiples avec la méthode d'agrégation

Les problèmes d'optimisation doivent souvent satisfaire des objectifs multiples, dont certains concourants. Une méthode classique consiste à définir des fonctions objectifs f_i, traduisant chaque objectif à atteindre, et de les combiner au sien de la fonction d'adaptation. On établit ainsi un compromis, le plus simple est de se ramener à une somme pondérée des fonctions objectifs :

$$f = \sum_i a_i f_i \qquad (3.12)$$

Où les poids α_i sont tels que la fonction d'adaptation f soit bornée dans l'intervalle [0, 1]. On peut classer les objectifs par ordre d'importance mais les poids seront adaptés par tâtonnement jusqu'à l'obtention d'une solution acceptable.
A la place d'une somme, on peut également utiliser un produit du type :

$$f = \prod_i a_i f_i \qquad (3.13)$$

Ou d'autres expressions plus complexes.

Dans cette méthode d'optimisation multiobjectifs, il faut être conscient des effets d'une telle combinaison des objectifs. En effet, deux solutions potentielles dont les fonctions objectifs n'ont pas la même valeur peuvent aboutir à une même valeur de la fonction d'adaptation. De plus, l'algorithme utilisant cette approche ne convergera que vers une seule solution alors qu'il existe peut-être toute une famille de solutions remplissant les mêmes objectifs fixés.

Dans le chapitre 5, nous avons appliqué cette méthode (une somme pondérée) pour l'exploration d'une architecture d'un processeur SuperScalar (*Cf* section 5.2.2) avec l'application turbo code.

3.6.2 Méthodes non-agrégatives

On peut cité quelque méthodes « non-agrégatives » les plus connus dans ce domaine tel que :

- la méthode Vector Evaluate Genetic Algorithm « VEGA » [83]
- la méthode Multiple Objective Genetic Algorithm « MOGA » [83]
- Pareto Archived Evolution Strategy « PAES » [14, 15]
- Strength Pareto Evolutionary Algorithm « SPEA » [16, 17]
- la méthode Niched Pareto Genetic Algorithm « NPGA » [18]
- la méthode Non dominated Sorting Genetic Algorithm « NSGA » [19]

La dernière méthode "NSGA", et en particulier NSGA-II [82] (A Fast and Elitist Multiobjective Genetic Algorithm), c'est celle que nous avons étudier en détail et l'implémenter dans le cadre de ce travail.

3.6.2.1 Pareto Archived Evolution Strategy « PAES »

Knowles et Corne [14] ont suggéré un MOEA simple en utilisant un seul parent et un seul enfant. Dans le *Pareto Archived Evolution Strategy* (PAES), l'enfant est comparé avec son parent respectif. Si l'enfant domine le parent, l'enfant est accepté comme un prochain parent et les itérations continuent. D'autre part, si le parent domine l'enfant, l'enfant est rejeté et une nouvelle solution mutée (un nouvel enfant) est trouvée. Cependant, si l'enfant et le parent ne se dominent pas, le choix entre l'enfant et le parent est fait en les comparant à des archives des meilleures solutions trouvées jusqu'ici. L'enfant est comparé aux solutions dans l'archive, puis voir s'il domine d'éventuels membres de l'archive. Si oui, l'enfant est accepté en tant que nouveau parent et toutes les solutions dominées sont éliminées de l'archive. Si l'enfant ne domine aucun membre de l'archive, le parent et l'enfant sont examinés en terme de proximité avec les solutions de l'archive. Si l'enfant réside dans la région la moins serrée dans l'espace des solutions parmi les membres de l'archive, on l'accepte comme un parent et une copie est ajoutée à l'archive. Si l'enfant et le parent ont le même ordre de proximité, l'un de deux est choisi au hasard. L'étalement (*Crowding*) de l'ensemble des solutions est maintenu en divisant de manière déterministe l'espace entier de recherche dans des sous-espaces de d^n (où d est le paramètre de profondeur et n est le nombre de variables de décision) avec une mise à jour dynamique de ces sous-espaces. D'autres méthodes améliorées ont été proposées, telles que le PESA (*The Pareto-Envelope based Selection Algorithm*), qui emploient la sous population stockée dans chaque cellule de la grille pour piloter la sélection et la diversité des membres de la population [15].

L'algorithme *PAES* emploie clairement les solutions non-dominées. C'est la propriété essentielle pour qu'un *MOEA* progresse vers le véritable optimal front de Pareto. La diversité est également maintenue par l'approche d'étalement (*crowding*) déterministe. Cependant, l'algorithme manque de preuve de convergence, simplement parce qu'un enfant non-dominé, bien qu'il ne soit pas une solution optimale de Pareto, peut remplacer un parent qui est déjà une solution optimale de Pareto (Pareto-optimal solution).

3.6.2.2 Strength Pareto Evolutionary Algorithm « SPEA »

Zitzler et Thiele [16] ont suggéré un MOEA élitiste avec le concept de la non domination. Ils ont proposé de maintenir une population externe à chaque génération, qui regroupe l'ensemble des solutions non dominées. Cette population externe participe aux opérations génétiques. Le fitness de chaque individu dans la population courante et dans la population externe est déterminé suivant le nombre des solutions dominées. D'abord une

population composée de la population externe et courante est construite. Toutes les solutions non dominées dans la population combinée sont assignées à une fitness basée sur le nombre de solutions qu'elle domine. Pour maintenir la diversité dans le contexte de minimisation de la fonction fitness on assigne la valeur dont le fitness le plus important aux solutions non dominées ayant plus de solutions dominées dans la population combinée. D'autre part, des fitness de valeurs plus élevées sont également assignées aux solutions dominées par plus de solutions dans la population combinée. Mais il faut prendre le soin de ne pas assigner à une solution non dominée une valeur de fitness plus mauvaise que celle d'une solution la plus dominée. Cette forme d'attribution des valeurs de fitness garantit que la recherche est orientée vers les solutions non dominées et simultanément la diversité parmi les solutions dominées et non dominées est maintenue. Un mécanisme de classement (*clustering*) regroupe toutes les solutions actuellement non dominées dans un nombre prédéfini de classes (*clusters*) et sélectionne une solution représentative de chaque classe, assurant de ce fait la diversité parmi les membres externes de la population.

Ici aussi, l'attribution de la valeur de fitness basée sur la non domination assure le progrès vers le véritable optimal de Pareto (*Pareto-optimal Set*). L'utilisation de la technique de cluster garantit la diversité de la population externe. Cependant, l'algorithme manque également de preuve de convergence, simplement parce que pendant la procédure de *clustering*, un individu de la population externe appartenant à l'optimal de Pareto (*Pareto-optimal*), peut être remplacé par un individu qui n'appartient pas à l'optimal de Pareto (*non-Pareto-optimal*).

3.6.3 Objectifs multiples avec le critère de Pareto

La difficulté de ces problèmes d'optimisation multiobjectifs vient du fait que, contrairement aux problèmes mono-objectifs, il n'existe pas de relation d'ordre entre toutes les solutions admissibles et donc plus de notion d'optimalité (à moins qu'une solution soit optimale pour toutes les fonctions objectifs, ce qui est rarement le cas). Dans le cas des problèmes multiobjectifs, elles sont remplacées par les notions de dominance et l'optimalité de Pareto. Le concept de l'optimal au sens de Pareto, aide pour la comparaison des solutions relatives à un problème d'optimisation multiobjectifs. En effet, une solution est un optimal de Pareto, si-elle n'est pas dominée par une autre dans le même espace des solutions.

Définition 1 : Dominance entre deux solutions

Une solution A domine une solution B pour un problème de maximisation (resp. minimisation) si :

$$z^q(A) > z^q(B) \quad (\text{resp. } z^q(A) < z^q(B)),\ \forall q \in Q$$

La notation A = B signifie que la solution A domine ou est équivalente à la solution B (c-à-d $z^q(A) = z^q(B),\ \forall q \in Q$).

Définition 2 : Pareto optimalité

$X \in S$ est un optimum de Pareto strict pour un problème de maximisation (resp. minimisation) si :

$$\text{il n'existe pas } X' \in S,\ z^q(X') = z^q(X) \ (\text{resp. } z^q(X') = z^q(X)),\ \forall q \in Q$$
$$\text{et} \quad \exists\, q' \in Q,\ z^{q'}(X') > z^{q'}(X) \ (\text{resp. } z^{q'}(X') < z^{q'}(X))$$

Une solution Pareto optimale est aussi appelée solution non-dominée. Nous ne parlerons donc plus de solution optimale, mais d'un ensemble de solutions Pareto optimales (aussi appelées solutions efficaces).

La figure suivante donne un exemple des ensembles de solutions Pareto optimales L = {S_1, S_2, S_3}. Dans ce cas, l'ensemble des solutions Pareto optimales est défini sur deux fonctions objectifs, f_0 et f_1.

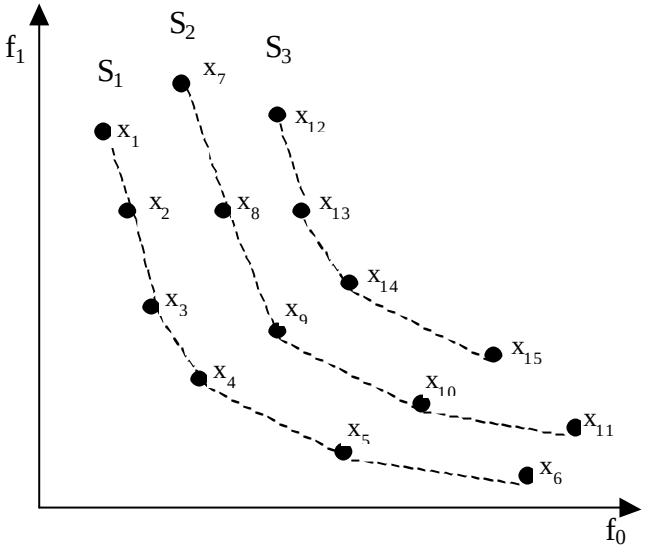


Figure 3.9 - Ensembles de solutions Pareto optimales

La Figure 3.10 représente les relations de domination entre 6 vecteurs objectifs dans un espace à deux dimensions : C est dominé par A, B et E, D est dominé par E, F est dominé par B, D et E.

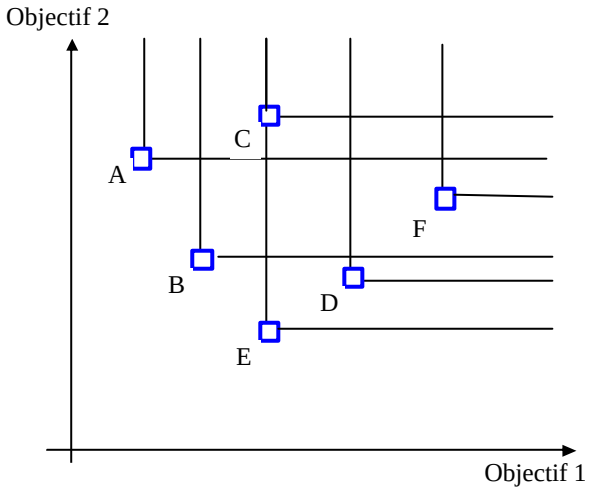


Figure 3.10 – Dominations au sens de Pareto dans un espace d'objectifs de dimension 2

3.7 L'algorithme NSGA-II

L'algorithme NSGA-II [19, 82] est un NSGA modifié, basé sur une classification des individus en plusieurs niveaux.

3.7.1 Classification des individus

Dans un premier temps, avant de procéder à la sélection, on affecte à chaque individu de la population un rang «*rank* » (en utilisant le rang de Pareto). Tous les individus non-dominés, de même rang sont classés dans une catégorie, à laquelle on affecte une efficacité

qui est inversement proportionnelle au rang de Pareto de la catégorie considérée. Le pseudo code de cette procédure de classification est représenté dans l'algorithme 3.2.
La figure suivante présente un exemple de classification en fronts de Pareto.

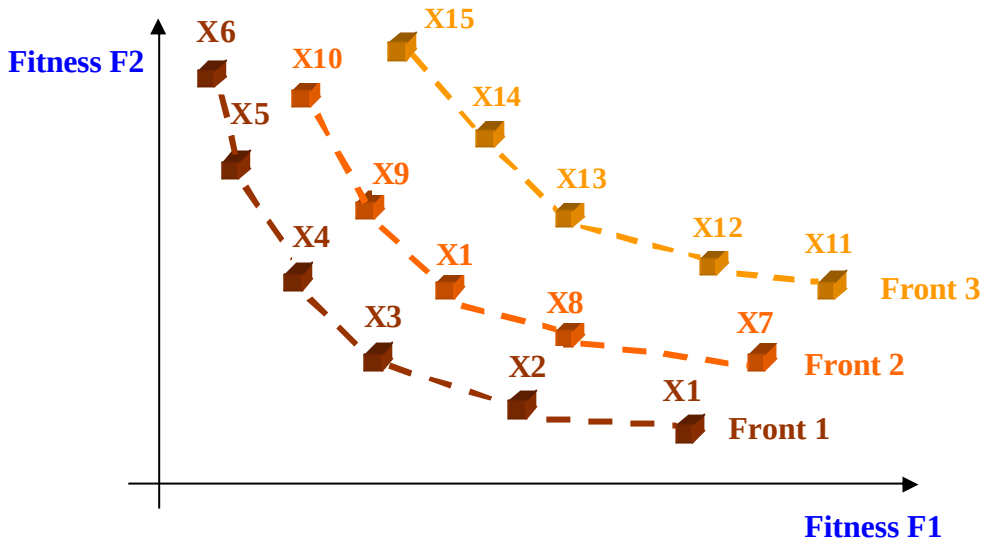


Figure 3.11 - Classification des individus en plusieurs catégories suivant le rang de Pareto (List of Pareto Sets)

```
Début
  P = population() /*Ensemble total de la population*/
  i = 1  /*i est le compteur des fronts de Pareto, initialisé à 1*/
  Répéter jusqu'à P = ∅
    Fi = chercher-ind-non-dominés(P) /*Trouver Fi,  l'ensemble des
                                individus non dominés correspondants
                                au front de Pareto i*/
    P = P \ Fi  /*Supprimer les ind. Non dominés à partir de la
                  population globale P*/
    i = i + 1  /*Incrémenter le compteur de front*/
  Fin Répéter
Fin
```

Algorithme 3.2 - Classification des individus suivant le rang de Pareto

L'algorithme 3.2, permet de classer la population globale (parents et enfants) en plusieurs fronts de Pareto. A partir de la population globale P, on sélectionne tous les individus non dominés pour former le front de Pareto d'ordre 2, les individus sélectionnés sont supprimés de la population P. On reprend la même procédure pour former le front de Pareto d'ordre 2 et ainsi de suite jusqu'à P = ∅.
Dans cet algorithme, pour trouver le premier front de Pareto avec une population de N individus, il nous faut N^2 comparaison. D'autre part, chaque comparaison est appliquée à M fonctions. Dans ce cas la complexité maximale de l'algorithme est de MN^2.
On veut maintenant que les individus dans la catégorie considérée se répartissent de manière uniforme au sein de celle-ci. Ou encore on veut qu'il y ait une diversité de solutions.

3.7.2 Préservation de la diversité

Contrairement à l'algorithme NSGA-II, la méthode MOGA, ne permet pas, dans certains cas, d'obtenir une diversité dans la représentation des solutions.

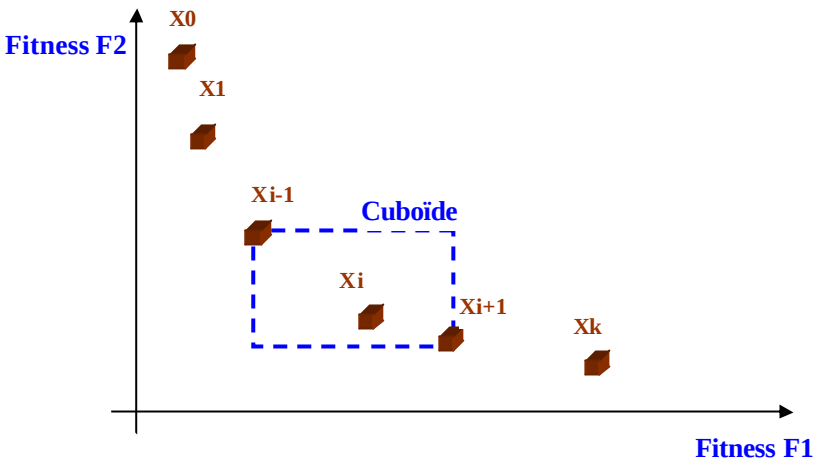


Figure 3.12 - Diversité dans la représentation des solutions.

L'algorithme 3.3, montre la technique (*crowding distance*) utilisée par le NSGA-II pour préserver la diversité des solutions sur le front de Pareto. Cette procédure s'applique sur le dernier front de Pareto pour compléter la taille de la population parents pour la génération suivante.

```
Début
  N <- |I|  /*Nbr. Des solutions dans le front de Pareto I*/
  Pour i = 1 jusqu'à N
    I[i]distance <- 0  /*initialisation de la distance*/
  Fin pour
  Pour m = 1 jusqu'à M /*M est le nombre des fitness*/
    I <- sort(I,m)  /*tri des individus suivant fitness m*/
    I[1]distance <- ∞    # pour les deux points extrêmes
    I[N]distance <- ∞
    Pour i = 2 jusqu'à (N – 1)  /*pour tous les autres points*/
      I[i]distance <- I[i]distance + (I[i+1].m – I[i-1].m)
    Fin pour
  Fin pour
Fin
```

Algorithme 3.3 - Affectation crowding distance

Où I[i].m représente la valeur de la m^{ème} fonction objectif relative au i^{ème} individu dans l'ensemble I. la complexité de cette procédure dépend de la répartition de la population. Quand tous les individus N de la population sont dans le même front I et pour M comparaison (M triages suivant les différentes fonctions objectifs) il nous faut O(MN log N) opération.

3.7.3 Opérateur de comparaison Crowded

L'opérateur de comparaison *crowded* ($<_n$) guide le processus de la sélection avec la répartition uniforme des solutions de Pareto. Un individu de la population a deux attributs :

- rang de non domination (i_{rank}),
- distance crowding ($i_{distance}$)

Soit les deux individus i et j,

$$i <_n j \;\; si \; i_{rank} < j_{rank} \qquad ou \qquad (i_{rank} = j_{rank}) \; et \; (i_{distance} > j_{distance})$$

Avec cette relation pour la comparaison de deux solutions non dominées appartenant à deux fronts de Pareto, on préfère la solution appartenant au front de Pareto d'ordre le plus faible.

Sinon, dans le cas où les deux solutions appartenant au même front de Pareto, on choisit la solution qui a la distance *crowded* la plus élevée.

3.7.4 Boucle principale de l'algorithme NSGA-II

L'algorithme 3.4, représente la boucle principale du NSGA-II. Initialement, il y a la création aléatoire d'une population parent P_0. Chaque individu de cette population est affecté à un front de Pareto adéquat. On applique les opérateurs génétiques (sélection, croisement et mutation) pour générer la population enfant Q_0 de taille N à partir de la population P_0. L'élitisme est assuré par la comparaison de la population courante P_t avec la population précédente P_{t-1}.

```
Début
  Rt <- Pt U Qt  /*Combinaison des pop. parent et enfant |Rt| = 2N*/
  F <- fast-nondominated-sort(Rt)  /*F = {F1, F2,…} ensemble des front
                                     de Pareto tel que ∑|Fi| = 2N  */
  Pt+1 <- ∅  /*initialisation*/
  i <- 1
  Tant que |Pt+1| + |Fi| ≤ N  /*jusqu'à la taille max de la pop. N*/
    Crowding-distance-assignment(Fi)  /*calcul de la distance «crowded»
                                       dans Front Fi*/
    Pt+1 <- Pt+1 U Fi  /*Reconstruction de la population parent en
                         ajoutant ième front de Pareto*/
    i <- i + 1  /*incrémentation vers le front suivant*/
  Fin tant que
  Sort(Fi, <n)  /*Triage des solution Fi dans l'ordre décroissant
                  suivant <n */
  Pt+1 <- Pt+1 U Fi[1:(N - |Pt+1|)]  /*Choisir juste les premiers éléments
                                       Fi pour compléter la population
                                       parent Pt+1 jusqu'à N individus*/
  Qt+1 <- make-new-pop(Pt+1)  /*Application des opérateurs génétique
                                (sélection, croisement, mutation) pour
                                la génération d'une nouvelle pop. Qt+1*/
  t <- t + 1  /*génération suivante*/
Fin
```

Algorithme 3.4 - Boucle principale NSGA-II

La figure 3.13, décrit les différentes étapes d'évolution de l'algorithme NSGA-II. Dans cette procédure, on distingue quatre étapes :

- Génération d'une population enfants Q_t à partir de la population parents Pt par l'application des opérateurs génétique (croisement et mutation). Cette manipulation est suivie par la fusion de deux populations enfants et parents dans une seule population globale R_t.
- Classement des individus de la population globale R_t en plusieurs front de Pareto dans l'ordre croissant (F_1, F_2, F_3, etc.).
- Reconstruction de la population parent pour la génération suivante (P_{t+1}) par la sélection d'individus appartenant aux premiers fronts de Pareto (F_1, F_2 et F_3) tels que :

$$\left|P_{t+1}\right| = \sum_{k=1}^{j} \left|F_k\right| \leq N \qquad (3.15)$$

Avec :

N : La taille de la population, j : Nombre de fronts sélectionnés et

$|F_k|$: Nombre d'individus dans le front de Pareto d'ordre k

- Application de la technique de *crowding distance* sur le front de Pareto F_{j+1} (c'est F_4 dans le cas de la figure 3.13) pour compléter la population P_{t+1} à une taille de N.

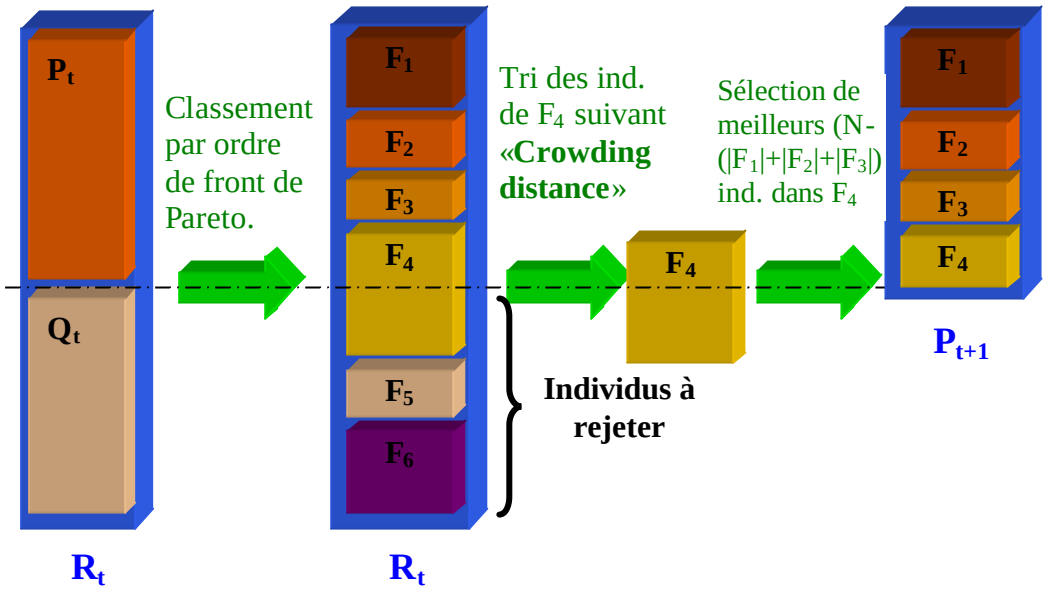


Figure 3.13 – Schéma de l'évolution de l'algorithme NSGA -II

P_t et Q_t dans la figure 3.13, sont respectivement la population parent et enfant de taille N. « Crowding distance » est une mesure de l'encombrement entre les individus de même front de Pareto. La nouvelle population P_{t+1} de taille N est utilisée avec l'application des opérateurs génétiques pour la génération d'une nouvelle population Q_{t+1} de taille N.

3.7.5 Convergence et diversité des solutions des algorithmes multiobjectifs

Au cours de ces dernières années, la recherche sur des algorithmes évolutionnaires a démontré leurs capacités en résolvant les problèmes d'optimisation multiobjectifs, où le but est de trouver, en une seule exécution de l'algorithme, un certain nombre de solutions proches de l'optimale de Pareto. Beaucoup d'études ont été menées sur différent algorithmes évolutionnaires pour montrer leurs aptitudes à progresser les meilleurs solutions, avec une bonne diversité (distribution homogène des solutions sur le front de Pareto) des solutions [83]. Cependant, aucun des algorithmes évolutionnaires multiobjectifs (MOEAs) a une preuve formelle de convergence [83], avec une large diversité, aux véritables solutions en optimales de Pareto [20, 21, 22, 23]. D'après des études qui ont étés menées sur plusieurs algorithmes multiobjectifs [83], le NSGA-II [82] est classé parmi les meilleurs et les plus populaires algorithmes jusqu'à ce jour en terme de convergence et de diversité de solutions.

3.7.6 Implémentation de NSGA-II

L'organigramme de la Figure 3.14, montre l'implémentation de l'algorithme NSGA-II en plusieurs fonctions. Nous avons programmé cet algorithme NSGA-II en langage C++ et en deux versions :

- Une première version sous linux avec la mise en œuvre et l'exploitation de son parallélisme implicite pour l'évaluation des fonctions objectifs sur plusieurs stations de calcul (Figure 3.14). Cette version a été réservée pour l'exploration architecturale à base de la simulation.
- Et une deuxième version sous l'environnement Windows pour l'exploration architecturale à base des exécutions directes de l'application sur FPGA.

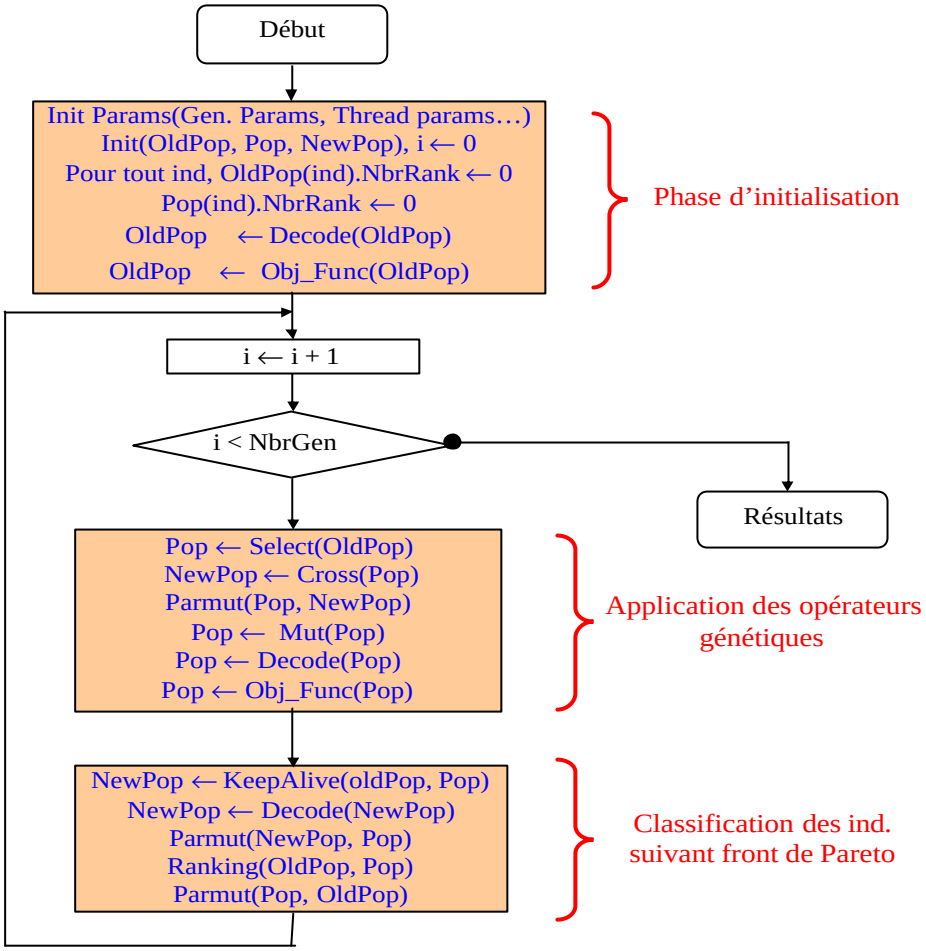


Figure 3.14 - Organigramme de l'implémentation NSGA -II

Dans ce diagramme les variables OldPop, Pop et NewPop sont des classes dans le programme en C++ pour gérer les évolutions des populations parents et enfants suite à l'application des opérateurs génétiques et les autres opérations tels que le codage, l'évaluation des individus, classification des individus par front de Pareto, etc.

3.7.7 Distribution du calcul des performances

Il s'agit de paralléliser l'étape d'évaluation des fonctions objectifs (Figure 3.14 : Obj_Func(OldPop)) pour chaque individu pour accélérer l'exécution de l'algorithme (Figure 3.14). Une station maître gère la boucle principale de l'algorithme NSGA-II (sélection/remplacement et opérateurs génétiques). Au même temps, le calcul des performances (les fonctions objectifs) est assuré par des stations esclaves. Le comportement de l'algorithme parallèle est exactement le même que celui de l'algorithme séquentiel. Avec ce procédé, le facteur d'accélération devient rapidement très proche du nombre de machines lorsque la ratio évaluation/évolution augmente, pour des tailles de population ne dépassant pas quelques multiples du nombre de processeurs disponibles pour le calcul simultané.

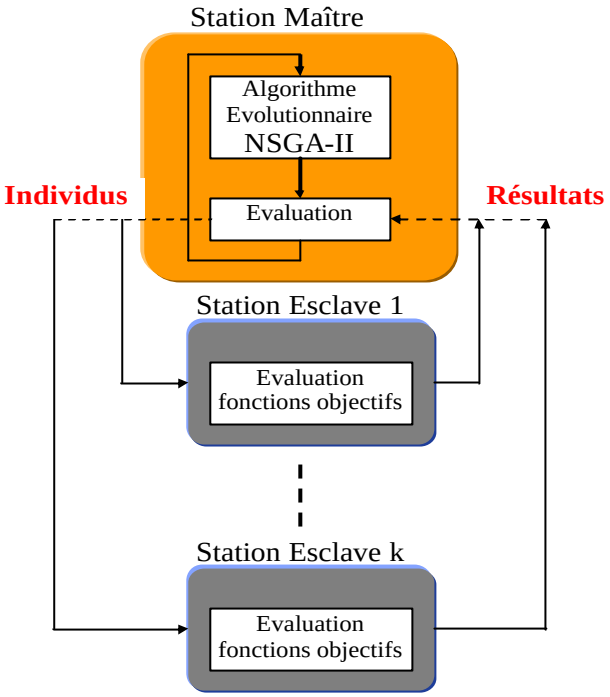


Figure 3.15 - Distribution des calculs de performance à travers le réseau local

Le gain apporté par ce schéma de parallélisation peut, toutefois, se dégrader, lorsque la durée du calcul de la fonction performance varie d'une configuration à l'autre (que ce soit en raison de conditions du calcul différentes ou par l'hétérogénéité des processeurs utilisés). En effet, l'ensemble des processeurs qui ont terminé leurs calculs sont bloqués en attente de la station esclave la plus lente. Une gestion plus fine de la distribution des calculs (à l'aide d'une liste de calculs à effectuer, dans ce cas, la station esclave reçoit une nouvelle configuration à évaluer dès la fin du calcul précédent). Ceci permet de limiter la durée d'attente pour chaque génération de l'algorithme NSGA-II. D'autre part, nous avons implémenté une procédure de contrôle pour détecter et écarter les machines en panne de la liste des stations candidates pour les prochains calculs des fonctions objectifs (si une station de calcul s'arrête pour une raison quelconque, l'ensemble de l'algorithme ne se mettra pas en attente du résultat manquant).

3.8 Conclusion

Dans ce chapitre, nous avons présenté des méthodes d'optimisations mono et multiobjectifs nécessaires à la compréhension de la suite du travail rapporté dans ce mémoire.

Le parallélisme implicite de l'algorithme NSGA-II nous a permis d'implémenter cette méthodologie sur un réseau locale composé de 100 stations de travail, ce qui nous permet l'évaluation (simulation) de tous les individus (les différentes configurations d'une population) simultanément. Avec cette topologie de parallélisme, on ramène le temps d'évaluation de N configurations en mode séquentielle par une seule machine en un temps plus court équivalent à l'évaluation d'une seule configuration. Ce gain du temps est accessible par l'exploitation du parallélisme implicite de l'algorithme génétique et en particulier le NSGA-II, par l'association de calcul des performances de chaque individu à une station de travail.

Les opérateurs de l'algorithme génétique sont guidés par un certain nombre de paramètres fixés à l'avance. Les valeurs de ces paramètres influence la réussite ou non d'un algorithme génétique. Ces paramètres sont les suivants :

- La taille de la population *N*, et la longueur du chromosome de chaque individu *l*. Si *N* est trop grand, le temps de calcul de l'algorithme peut s'avérer très important, et si *N* est trop petit, il peut converger trop rapidement vers un mauvais chromosome.
- La probabilité de croisement Pc. Elle dépend de la forme de la fonction de fitness. Son choix est en général heuristique (tout comme pour Pm), plus elle est élevée, plus la population subit évidemment de changements importants. Les valeurs généralement admises sont comprises entre 0,5 et 0,9.
- La probabilité de mutation Pm. Ce taux est généralement faible puisqu'un taux élevé risque de conduire à une solution sous-optimale.

Donc pour nous études expérimentales dans la suite de ce mémoire nous avons fixée la valeur de la probabilité de croisement autour de 0,8 et la probabilité de mutation autour de 0,1.

Dans ce qui suit, nous allons appliquer la méthode d'optimisation multiobjectifs NSGA-II à deux types d'explorations architecturales dans le domaine du système sur puce « SOC » : la premier cas illustre une exploration architecturale à base de la simulation, puis comme deuxième cas, nous aborderons l'exploration architecturale à base des exécutions directes sur FPGA.

CHAPITRE 4

EXPLORATION ARCHITECTURALE

De nos jours La pression des enjeux économiques est telle que l'évaluation des performances des systèmes est devenue un domaine incontournable et hautement stratégique. Il devient en effet inconcevable de construire un système quelconque (que ce soit un système informatique, un réseau de communication ou un système sur puce) sans avoir auparavant mené d'analyse de performances : un système sous dimensionné sera inutilisable, tandis qu'un système surdimensionné entraînera un gaspillage financier.

4.1 IPs Processeurs et Microarchitecture

Les IPs (*Intellectual Property*) sont des blocs fonctionnels complexes pouvant être réutilisés dans plusieurs conceptions. On distingue deux types de blocs IPs :

- Hard IP : sont des blocs IPs déjà physiquement implanté, dépendant de la technologie utilisée. Cette famille de blocs IPs est très optimisée.
- Soft IP : sont des blocs décrit en langage de haut niveau (VHDL, Verilog), sont souvent des blocs paramétrable et synthétisable.

Généralement, l'emploi de ces blocs IPs dans un projet implique l'utilisation d'un environnement de développement adapté (co-design, co-vérification,…) tel que Xilinx EDK, IBM Core Blue, Mentor-Altera, etc.

4.2 Evaluation des Performances

Au cours de ce travail, l'évaluation des performances est basée sur le nombre de cycles d'horloge du processeur nécessaire à l'exécution d'une application. Le nombre de cycles d'horloge peut être déterminé par simulation (exécution de l'application sur un simulateur c'est le cas de SimpleScalar) ou par exécution directe (dans ce cas un compteur hardware est implémenté dans l'architecture, c'est le cas de la plateforme FPGA). L'évaluation de nombre de cycles par un compteur intégré dans l'architecture est beaucoup plus précise que l'évaluation à base de simulation ou à base d'outils de modélisation [85, 86].

4.3 Evaluation de la consommation d'énergie

La consommation d'énergie est devenue une issue critique dans la conception de processeurs, particulièrement dans les systèmes embarqués. En conception de processeurs de haute performance et à basse consommation d'énergie, les concepteurs doivent expérimenter des différents niveaux du software et d'architecture et évaluer diverses techniques d'optimisation de la consommation d'énergie. Ces outils d'évaluation de la consommation d'énergie au niveau architecturals deviennent de plus en plus importants avec la complexité croissante de la conception actuelle des *" System-on-a-Chip "* pour fournir tôt dans le cycle de conception des évaluations rapides de la consommation d'énergie. Une fois que la conception d'un processeurs de large complexité aie été achevée au niveau spécification circuit ou porte logique, il peut être trop tardif ou trop cher de retourner et traiter des problèmes de consommation excessifs d'énergie. D'autre part, la couche logicielle prend un

aspect important dans les systèmes embarqués actuels et l'étude de l'impact et de l'optimisation de la couche logicielle et matérielle devrait être supportée par de tels outils.
Avec le développement des appareils du type PDA ou téléphones de troisième génération, capables de délivrer des images et du son en temps réel, les besoins en énergie, en ressources matériels ainsi qu'en puissance de traitement sont sans commune mesure avec ce que les concepteurs ont connu. Alors que la technologie des batteries progresse moins vite que la puissance dissipée par les applications, il faut trouver de nouveaux moyens pour étendre la durée d'utilisation.
Par le passé le logiciel et le matériel étaient développés indépendamment, il apparaît aujourd'hui nécessaire de travailler à la fois sur les ressources du système matériel, ainsi que sur la façon dont est construit intrinsèquement le logiciel afin d'économiser des ressources.

4.3.1 Techniques employées pour la réduction de la consommation d'énergie

Pour étendre l'autonomie de fonctionnement d'un système, seules deux méthodes existent : augmenter la quantité d'énergie embarquée ou diminuer la consommation du système.
La première solution a entraîné de nombreuses recherches dans le domaine des batteries, mais malgré les progrès effectués dans ce domaine, il est toujours difficile d'augmenter la capacité d'une batterie sans en augmenter le poids, le volume et le prix. La seconde solution est complémentaire à la première, et a également donné lieu à de nombreuses recherches dans le domaine de l'électronique et de l'informatique.
Les techniques utilisées pour concevoir un système embarqué à faible consommation, sont regroupées en trois catégories :

- Les techniques matérielles,
- Les techniques logicielles,
- Les techniques de conception hybrides,

La première est la plus répandue, concevoir des composants spécifique pour consommer le minimum d'énergie. La seconde méthode consiste à modifier le logiciel pour diminuer le coût énergétique de son exécution (optimisation du code des applications, adaptation des protocoles de communication). Enfin la troisième technique consiste à réaliser une collaboration entre le logiciel et le matériel a fin d'optimiser la consommation totale de l'appareil. Dans la référence [73], les auteurs ont proposé une méthodologie pour la réduction de la consommation d'énergie. Cette méthode peut être classée dans la deuxième catégorie c'est-à-dire dans la technique logicielle. Puisqu'une grande partie de la consommation d'énergie dans un composant CMOS est dû au régime dynamique. Cette énergie dissipée peut être optimisée en réduisant la tension d'alimentation de l'unité de calcul en question. Pour mettre en œuvre cette technique, l'auteur propose une méthode de gestion de la tension d'alimentation des unités de calcul, basée sur l'ordonnancement de ces unités suivant une contrainte temporelle.

4.3.2 SimplePower

SimplePower [74, 75, 76] est basé sur l'architecture d'un pipeline de cinq étages, composé d'une unité de recherche d'instructions « *fetch stage IF* », l'unité de décodage « *decode stage ID* », l'unité d'exécution «*execution stage EXE* », l'unité d'accès à la mémoire « *memory access stage MEM* » et enfin l'étage de registre d'écriture en retour « *write-back stage WB* ».
Nous avons utilisé le simulateur SimplePower pour l'estimer l'énergie dissipée par le processeur SuperScalar que nous avons adapté dans le cas d'exploration à base de simulation.

4.4 Evaluation de la surface silicium

Dans le cas de l'optimisation de l'architecture à base de simulation, nous avons utilisé deux outils pour l'évaluation de la surface en silicium :

- Outil CACTI [78, 79]: nous avons utilisé cet outil pour estimer la surface en silicium du cache instruction et cache données utilisés par le processeur SuperScalar.
- Outil FUPA [80, 81] : c'est l'algorithme pour estimer la surface en silicium des unités de calcul avec virgule flottante. Avec FUPA on utilise un profiler (appliqué à l'application soft) pour déterminer le pourcentage d'utilisation des instructions à virgule flottante Adds, Mults et Divs.

4.5 Exploration de l'espace architectural à base de simulation

Il s'agit d'implémenter le système (l'application SOC : partie matérielle + partie logicielle) à concevoir sur un ou plusieurs simulateurs. Cette technique permette au concepteur de vérifier le bon fonctionnement du son application et d'estimer leurs performances sans passer par une réalisation réelle du système à étudier. Nous dans cette optique, on s'intéresse au simulateur SimpleScalar dédié au processeur de type SuperScalar.

4.5.1 Le Simulateur SimpleScalar

D'abord expliquons pourquoi SimpleScalar a été choisi comme un outil de base pour notre exploration à base de simulation :

- Premièrement, un élément important a été la puissance et la précision de simulation pour une très large gamme des processeurs.
- Deuxièmement, la disponibilité gratuite d'un outil déjà employé pour réaliser des évaluations d'architectures *SuperScalaires*.
- Troisièmement, toute la partie impliquée dans la génération de code (compilateur C, éditeur de liens, etc.) est fournie dans l'outil, ce qui permet de tester le processeur en question (simulé) en employant un langage structuré (le C en l'occurrence).
- Finalement, la part croissante de chercheurs utilisant SimpleScalar s'étendant de plus en plus, il est possible de trouver plus ou moins facilement de l'aide et du support.

SimpleScalar permet de concevoir un simulateur fonctionnel de processeurs, c'est-à-dire pouvant exécuter un programme instruction après instruction en réagissant de la même manière qu'une version «réelle». En d'autres termes, il permet de simuler le comportement interne du processeur et non seulement la partie interprétation d'un jeu d'instructions données. Ainsi, il est possible de construire des architectures en spécifiant avec précision leur fonctionnement.

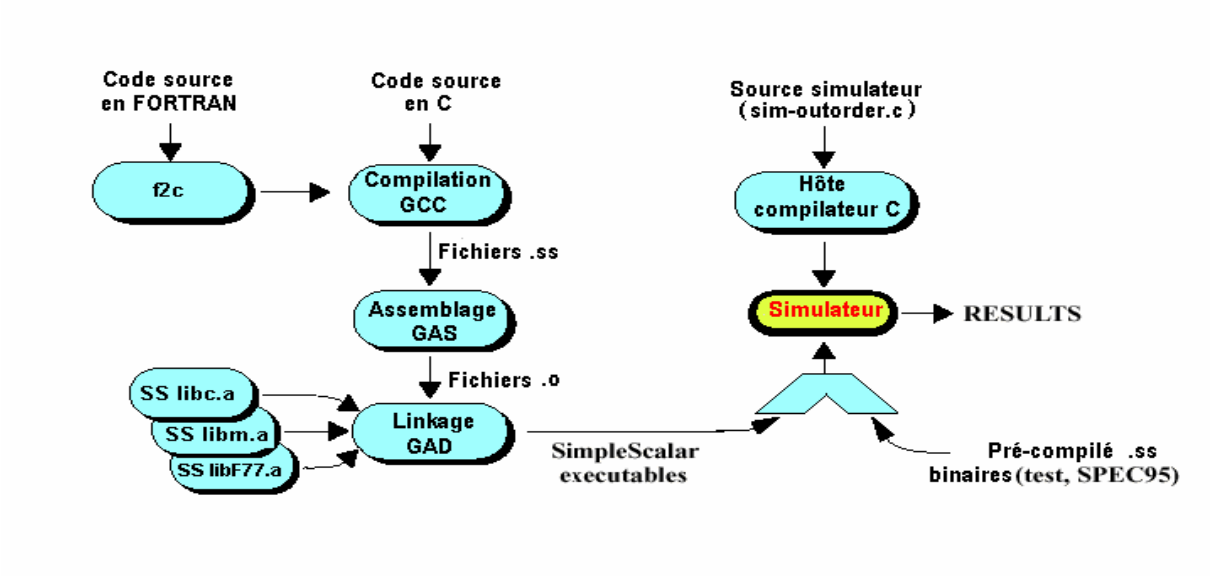


Figure 4.1 - Vue synoptique de SimpleScalar

La simulation produit non seulement des statistiques détaillées sur l'utilisation des ressources internes (le taux de défauts de cache «*Miss-rate :* mots demandés n'existent pas dans le cache », nombre d'accès *TLB*, nombre de sauts mal prédits, etc.) mais fournit également une trace de l'exécution.

D'autre part, SimpleScalar est distribué avec un ensemble d'outils de compilation et un jeu d'instructions prédéfini fortement inspiré de l'assembleur *MIPS*. Ces deux éléments permettent au concepteur de se concentrer sur l'architecture qu'il veut simuler sachant qu'il pourra la tester en lui fournissant des programmes compilés, directement écrits en C ou en Fortran. Dans les grandes lignes, une session de simulation complète ressemble au schéma de la Figure 4.1. On peut voir au sommet de cette figure les fichiers sources C ou Fortran qui sont compilés par une version croisée de GCC (après conversion en C par f2c pour les programmes Fortran) produisant en sortie un code binaire interprétable par le simulateur. Celui-ci se présente sous la forme d'un programme exécutable construit à partir d'une description de l'architecture et d'une description du jeu d'instructions, tous deux décrits par un ensemble de fichiers C. Le simulateur doit être compilé préalablement sur la machine hôte qui va simuler le processeur. Le simulateur ainsi construit charge le code binaire produit par le compilateur croisé et exécute celui-ci. Il produit une trace de l'exécution et des statistiques relatives à l'exécution. La trace peut éventuellement être mise en forme plus tard pour pouvoir être examinée aisément.

L'annexe A, montre un exemple de trace d'exécution et de statistique générée par *SimpleScalar*.

La Figure suivante montre l'architecture générique de pipeline *SuperScalaire* implémentée par le simulateur *SimpleScalar*.

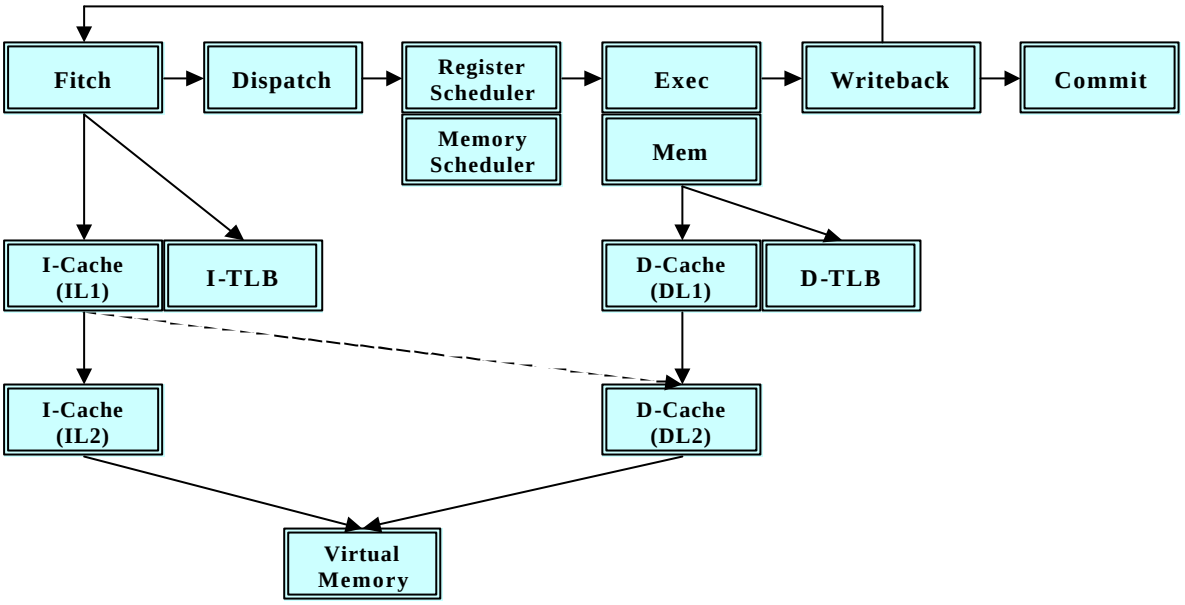


Figure 4.2 - Architecture Pipeline implémentée par SimpleScalar

Dans cette architecture il y a deux caches, I-Cache et D-Cache qui sont respectivement cache d'instructions et cache de données, chaque type de cache est associé à un *TLB* (*Translation Lookaside Buffer*), ou tampon de traduction anticipée, qui est un petit cache spécial de traduction d'adresse.

4.5.2 Paramètres de simulation

Un bon choix du dimensionnement de cache, de pipeline et les unités fonctionnelles, autorise à un calculateur d'atteindre de hautes performances, d'où l'intérêt de simuler ces trois unités de chaque processeur de l'architecture et extraire leurs valeurs optimales.

4.5.2.1 Paramètres caches

Comme le montre la Figure 4.2, le processeur *SuperScalai* comporte deux types de caches : cache instructions (*I-cache*) et cache données (*D-cache*). Le paramétrage de ces deux caches dans le simulateur *SimpleScalar* se sont comme suit :

Bsize : Indique la taille du bloc. En effet, la localité spatiale est naturellement présente dans les programmes. Pour en tirer partie, il faut choisir une taille du bloc de cache qui aie plus d'un mot de long. Lorsqu'un défaut aura lieu, on extraira alors plusieurs mots qui seront adjacents et auront une forte probabilité d'être bientôt demandés. La Figure 4.3, présente un exemple de variation du taux de défauts du cache « *Miss-rate* » en fonction de la taille du bloc [24].

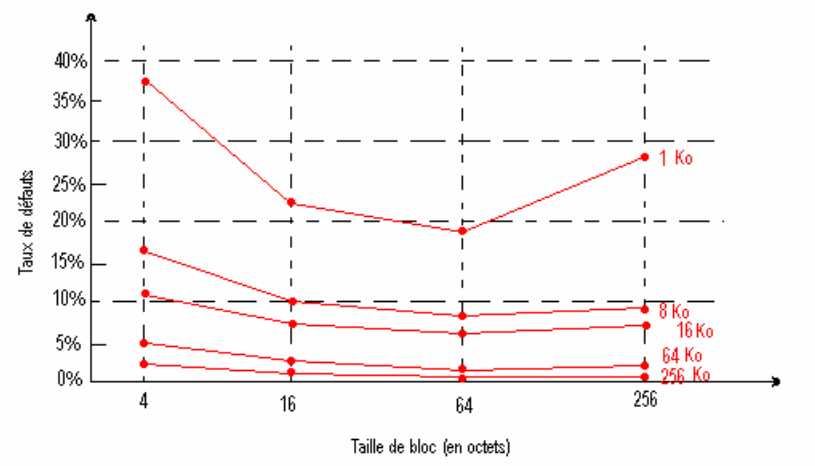


Figure 4.3 - Taux de défauts « Miss-rate » contre taille du bloc

Nsets : Nombre des lignes dans un cache.

Assoc : Indique le mode d'associativité de cache, pour Assoc = 1, dans ce cas c'est un cache à correspondance directe c'est-à-dire le cache est formé d'un seul ensemble des blocs. Pour Assoc = 4, dans ce cas c'est un cache avec associativité de quatre ensemble de blocs, etc.

Et d'autres paramètres de configuration des caches ne sont pas modifiés (chargés avec les valeurs par défaut) au cours des simulations. Pour plus des détails, le lecteur peut se référer au rapport technique du simulateur *SimpleScalar* [25].

4.5.2.2 Paramètres de Pipeline et les unités fonctionnelles

L'architecture du simulateur SimpleScalar adopte un pipeline à cinq étages dont :

- *fetch_ifqsize(taille) :* c'est l'unité de prédiction des instructions à exécuter par le pipeline avec une taille qui prend des valeurs puissance 2, par défaut égale à 4
- *Issue_width(val) :* val prend des valeurs puissance 2, par défaut égale à 4
- *decode_width(val):* c'est l'unité de décodages des instructions dans le pipeline. La variable *val* prend des valeurs puissance 2, par défaut égale à 4

L'unité fonctionnelle est caractérisée par les paramètres suivants :

- *res_ialu(num) :* pour spécifier le nombre d'unités arithmétiques et logiques (*ALU*) intégrées par le processeur, la variable '*num*' par défaut égale à 4 ALUs
- *res_imult(num) :* pour spécifier le nombre d'unités de calcul multiplication/division entier (*MUL*) intégrées par le processeur ,'*num*' par défaut égale à une unités de calcul
- *res_fpalu(num) :* pour spécifier le nombre d'ALU à virgule flottante intégrées par un processeur, 'num' par défaut égale à 4 unités
- *res_fpmult(num) :* pour spécifier le nombre d'unités de calcul multiplication/division à virgule flottante (*MUL*) intégrées par un processeur ,'*num*' par défaut égale à une seule unités

Bien sûr, il y a d'autres paramètres du processeur que nous avons pas modifiés au cours des simulations (chargés par le simulateur avec leurs valeurs par défaut) [25]. Autrement dit, ces paramètres par défaut n'appartiennent pas à l'espace d'exploration.

4.5.3 Représentation chromosomique: codage d'individus

Le codage des différents paramètres du microprocesseur est représenté par un chromosome binaire de longueur 33 bits (Figure 4.4).

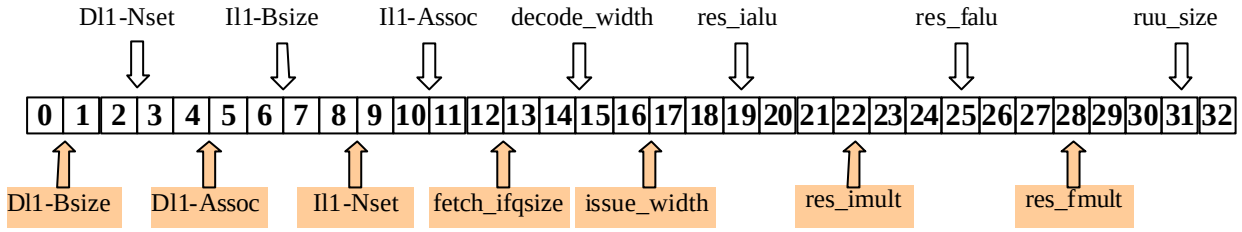


Figure 4.4 - Codage du chromosome appliqué dans l'exploration par simulation

Ce chromosome de 33 bits nous permet de coder 14 paramètres du processeur soit un espace d'exploration de 14 variables.

4.5.4 Décodage de la solution

Le codage des paramètres dans l'AG est représenté par le tableau suivant :

Module	Nom de paramètre	Valeurs possibles
Cache d'instruction (Instruction Cache)	Nombre d'ensemble (nsets)	64, 128, **256**, 512
	Taille de bloc (bsize)	8, 16, **32**, 64 (Octet)
	Associativité (assoc)	**1**, 2, 4, 8
Cache de données (Data Cache)	Nombre d'ensemble (nsets)	64, 128, **256**, 512
	Taille de bloc (bsize)	8, 16, **32**, 64 (Octet)
	Associativité (assoc)	**1**, 2, 4, 8
Pipeline	Fetch:ifqsize, Issue:width, decode:width	1, 2, **4**, 8
ALU Entier	Nombre des unités arithmétique et logique entier (res:ialu)	1, 2, 3, **4**, 5, 6, 7, 8
MULT Entier	Nombre des unités multiplication/division entier (res:imult)	**1**, 2, 3, 4, 5, 6, 7, 8
ALU flottante	Nombre des unités arithmétique et logique flottantes (res:fpalu)	1, 2, 3, **4**, 5, 6, 7, 8
MULT flottante	Nombre des unités multiplication/division flottantes (res:fpmult)	**1**, 2, 3, 4, 5, 6, 7, 8
RUU (Register Update Unit size)	Taille de RUU (ruu:size)	2, 4, 8, **16**, 32, 64, 128, 256 (Instructions)

Tableau 4.1 – Codage des paramètres du processeur SuperScalar

Les valeurs en gras dans le Tableau 4.1, sont les valeurs par défaut des paramètres du processeur à l'entrée du simulateur SimpleScalar. La taille du cache est égale au produit des trois paramètres «nsets, bsize et assoc ». D'où, la taille par défaut de chaque cache (cache instruction et cache données) est égale à 8 KOctet.

4.5.5 Résultats de la simulation

Les applications logicielles cibles que nous avons utilisées dans ce travail sont des benchmarks spécialement développés pour l'évaluation des systèmes embarqués (MiBench [37]).

Le Tableau 4.2 résume toutes les applications disponibles dans MiBench. Pour l'évaluation de notre architecture, nous avons utilisé le coder Jpeg, l'algorithme de Dijkstra, le transformé FFT et le codeur GSM.

Auto./Industrial	Consumer	Office	Network	Security	Telecomm
basicmath	**jpeg**	ghostscript	**dijkstra**	blowfish enc.	CRC32
bitcount	lame	ispell	patricia	blowfish dec.	**FFT**
qsort	mad	rsynth	(CRC32)	pgp sign	IFFT
susan (edges)	tiff2bw	sphinx	(sha)	pgp verify	ADPCM enc.
susan (corners)	tiff2rgba	stringsearch	(blowfish)	rijndael enc.	ADPCM dec.
Susan (smoothing)	tiffdither			rijndael dec.	**GSM enc.**
	tiffmedian			sha	GSM dec.
	typeset				

Tableau 4.2 - MiBench Benchmarks

Nous avons lancé l'exploration de cet espace de solutions par l'algorithme NSGA-II avec les paramètres de réglage suivants :

Nom du paramètre	Valeur
Taille de la population	33
Type de Codage chromosome	Binaire de taille 33 bits
Nombre des variables de l'espace d'exploration	14
Nombre des fonctions objectifs	3
Probabilité de croisement	0,80
Probabilité de mutation	0,015
Nombre des générations	200

Tableau 4.3 – Paramètres de l'algorithme NSGA-II appliqué au MiBench

La Figure 4.5 montre le front de Pareto relatif à l'utilisation de l'application codeur jpeg. Le code source en ANSI C de ce codeur jpeg est disponible dans le benchmark MiBench.
On applique l'exploration architecturale en utilisant le coder jpeg (application soft cible) pour deux images de résolution différentes : (1) image de taille 256x256 pixels et (2) une image de taille 512x512 pixels.

La figure montre bien que dans le cas de traitement de l'image (2) on a besoin davantage de ressources au niveau du processeur et donc plus de consommation d'énergie et un temps d'exécution plus long.

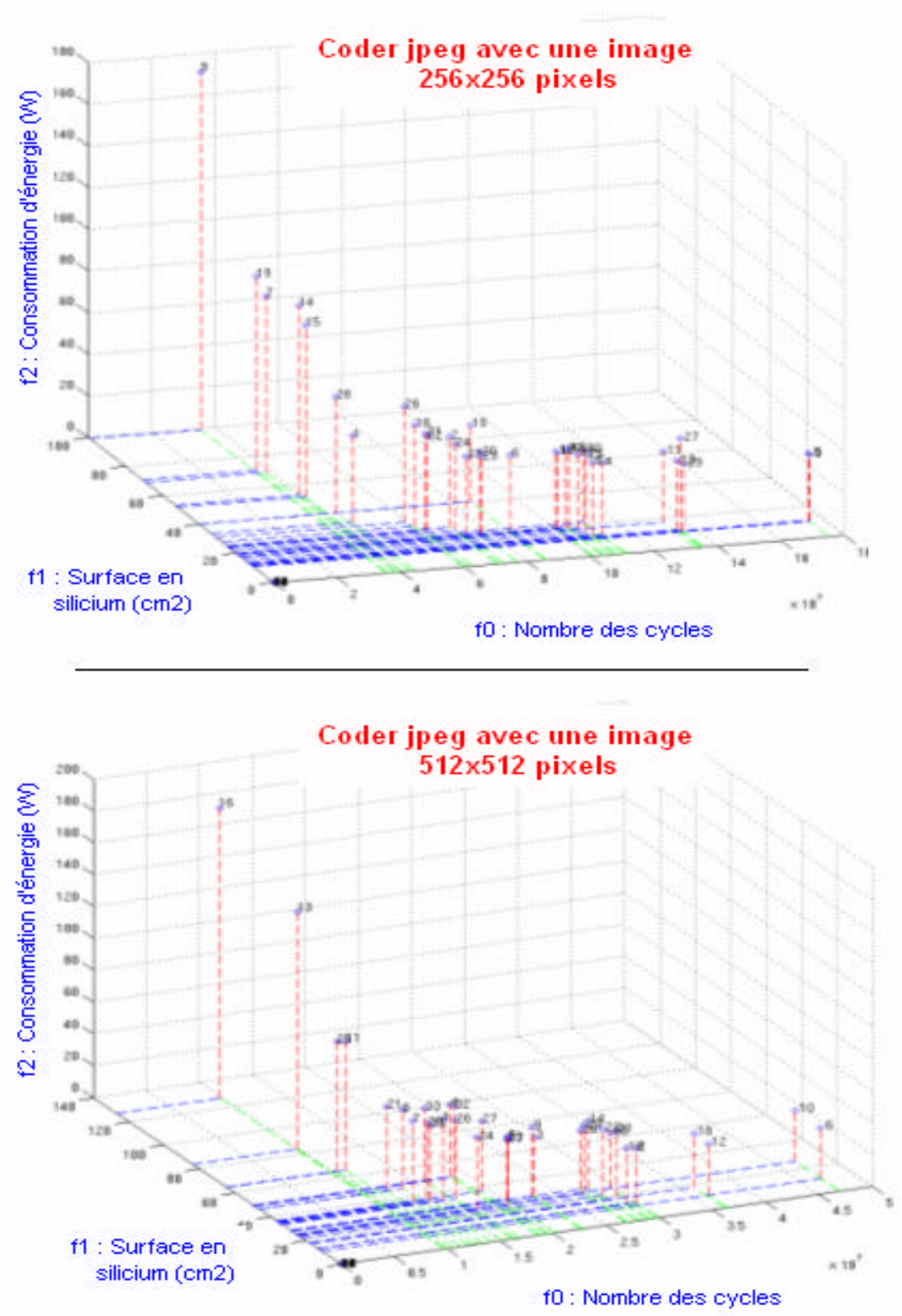


Figure 4.5 – Front de Pareto dans le cas du coder jpeg

La figure 4.6 montre le front de Pareto relatif à l'utilisation de l'application codage de la voix sous la norme GSM. Dans cette exploration nous avons utilisé deux séquences audio de tailles différentes pour montrer l'impact de l'application software sur le dimensionnement de l'architecture hardware.

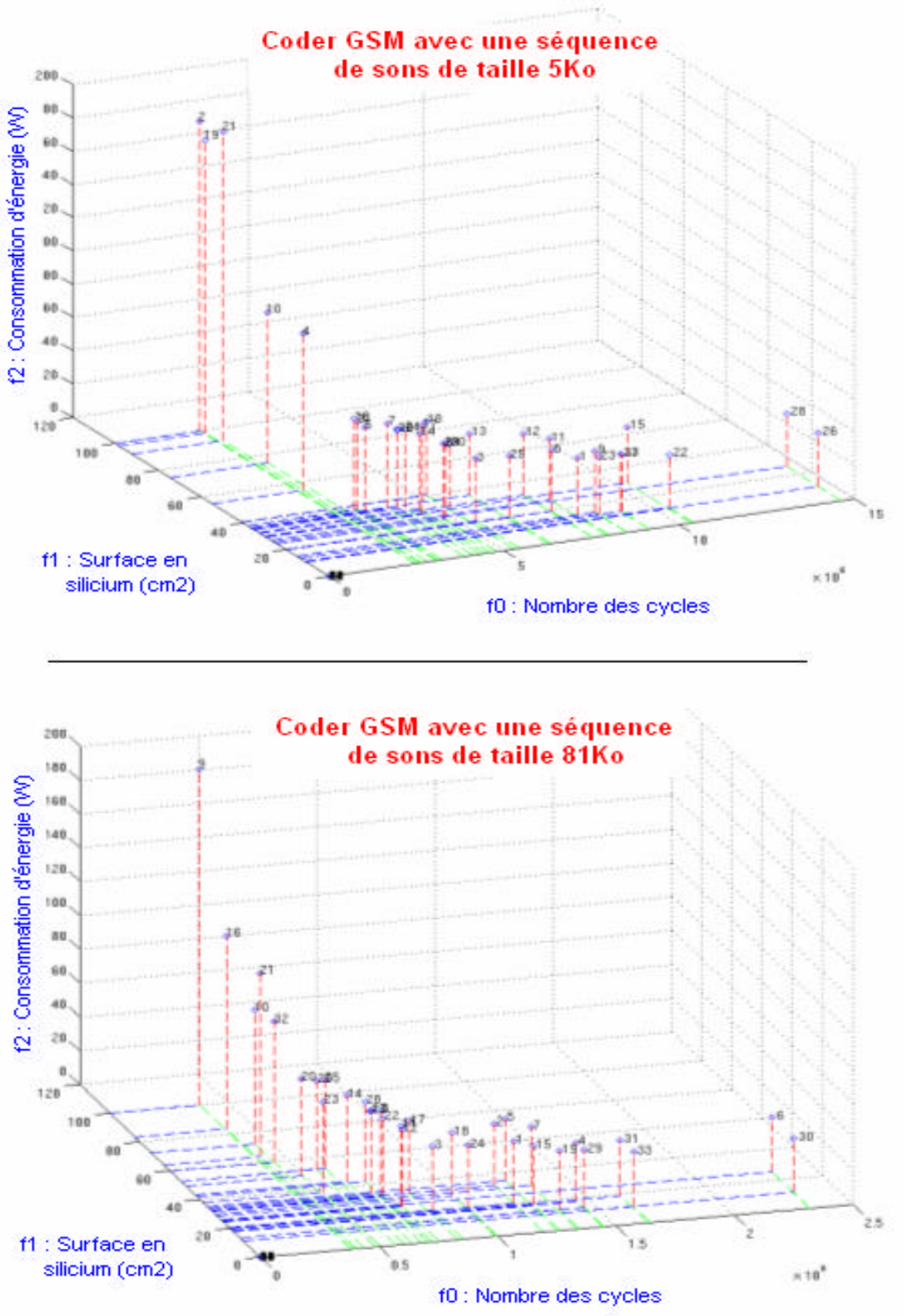


Figure 4.6 – Front de Pareto dans le cas du coder GSM

La figure 4.7 représente le front de Pareto relatif à l'exploration de l'architecture en utilisant la transformée de Fourrier FFT. C'est le même principe que les deux applications précédentes, nous avons lancé deux explorations avec deux tailles différentes de données à l'entrée de FFT.

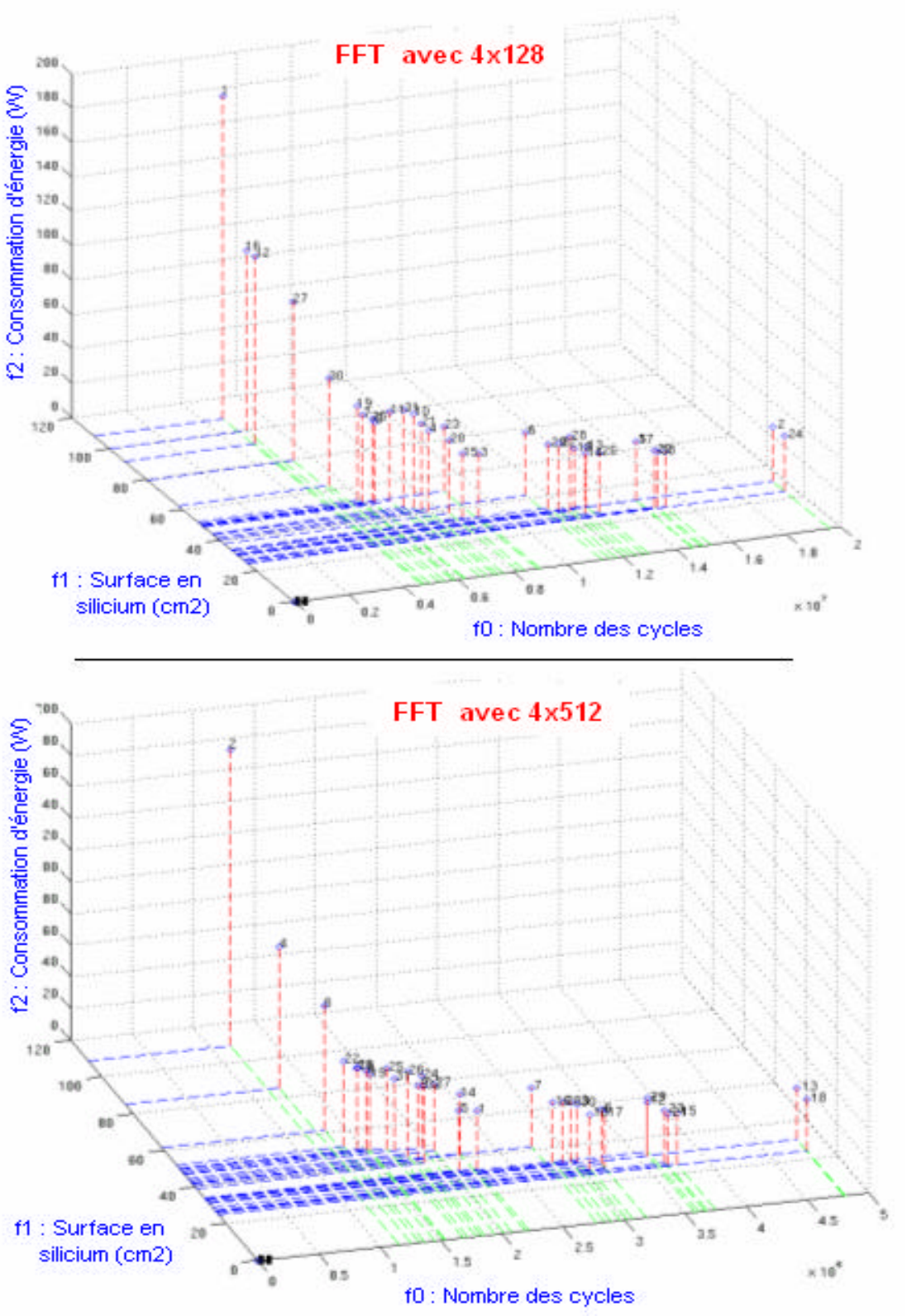


Figure 4.7 – Front de Pareto dans le cas de la FFT

La figure 4.8 montre le front de Pareto relatif à l'exécution de l'algorithme de Dijkstra avec deux configurations de données de tailles différentes.

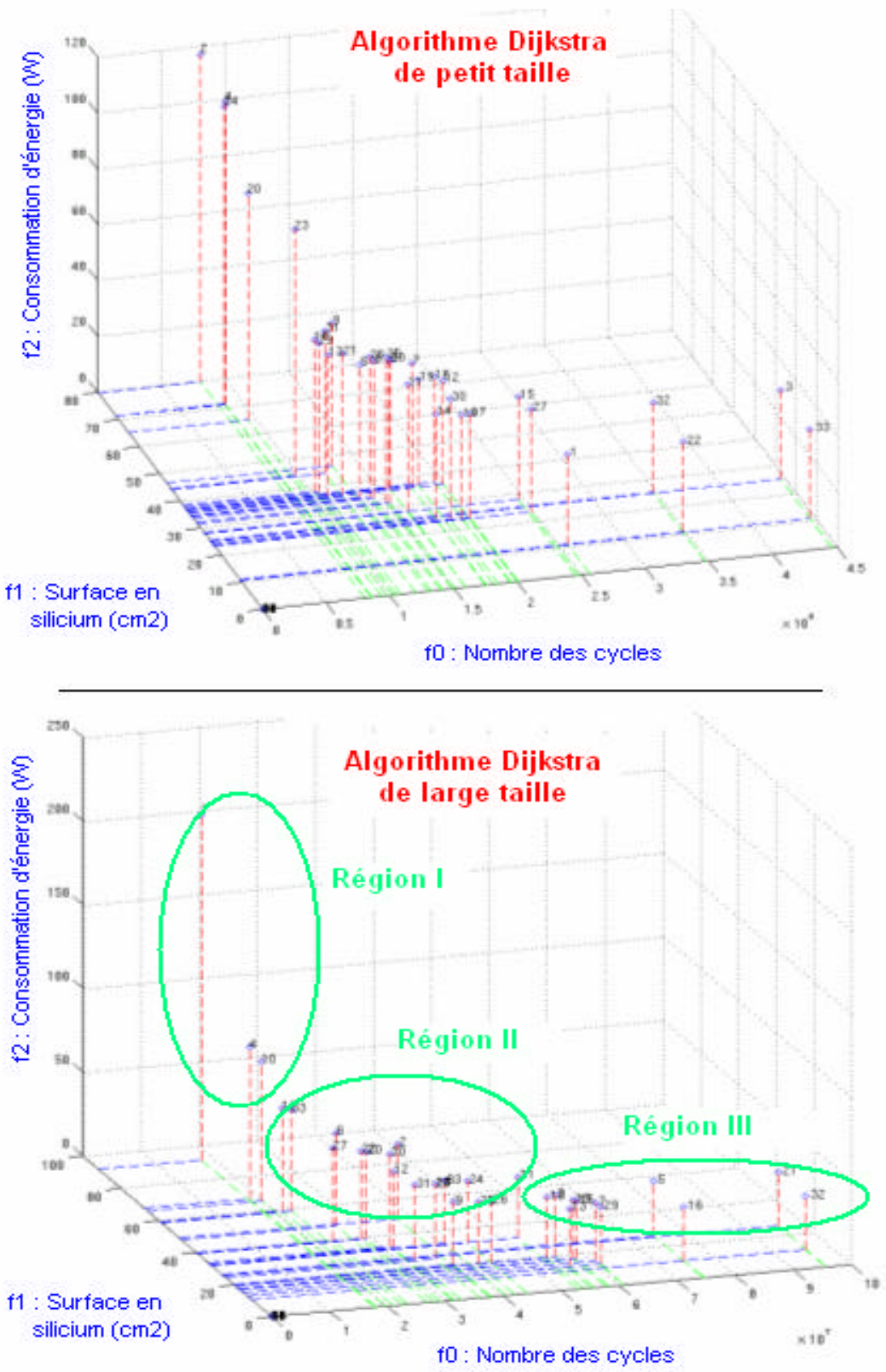


Figure 4.8 – Front de Pareto dans le cas de l'algorithme Dijksta

Toutes les figures présidentes montrent bien l'intérêt de l'utilisation du front de Pareto pour la comparaison et le choix des solutions dans le cas d'un problème multiobjectifs. Pour le concepteur, le front de Pareto est un outil visuel très facile à utiliser pour le choix des paramètres de l'architecture la plus compatible avec son application logicielle à exécuter.

D'après ces résultats, on peut partager le front de Pareto en trois régions différentes suivant l'exigence de l'application (Figure 4.8) :

- Région I: les solutions dans cette région du front de Pareto impliquent un temps d'exécution très rapide mais le prix à payer une consommation d'énergie très élevée et l'allocation d'un maximum de ressources matérielles pour atteindre ces performances.
- Région III : dans ce cas le temps d'exécution est relativement long, mais on gagne en terme de dissipation de la puissance et un minimum de ressources utilisées.
- Région II : représente les solutions modérées en terme des trois objectifs.

En comparaison avec d'autres travaux, ces résultats confirment l'efficacité de cette méthodologie et en particulier en terme d'optimisation multiobjectifs. Ainsi, dans la même boucle, on cherche à optimiser simultanément les trois grandeurs, le nombre de cycles, la surface en silicium et la puissance dissipée d'un microprocesseur paramétrable. Dans la référence [26], les auteurs proposent une méthodologie à base d'algorithme génétique mono-objectif, pour le paramétrage d'un processeur. Cette optimisation est monobjectif, c'est la minimisation de la durée d'exécution (nombre de cycles). D'autre part, dans cet article [26], l'espace des solutions à explorer est plus réduit, seulement sept paramètres de processeur ont été injectés dans l'algorithme génétique contre 14 paramètres à explorer dans le cas de notre méthodologie.

4.6 Exploration à base d'Emulation

Dans cette partie, l'évaluation des performances est basée sur l'exécution directe de l'application. Cette possibilité d'évaluation est rendue possible par l'implémentation du système à évaluer sur un FPGA.

4.6.1 IP de processeur Leon

Le processeur *Leon* [27, 28, 29] emploie une architecture interne de 32 bits (Figure 4.9), entièrement décrite en VHDL et complètement synthétisable. Il a été développé par l'Agence Spatiale Européenne (ESA). Son architecture est de type *SPARC V8* (*Scalable Processor ARChitecture*) [30, 31]. Tout comme l'architecture *RISC* (*Reduced Instruction Set Computer*), l'architecture *SPARC* ne spécifie pas une implémentation matérielle précise mais un standard libre de droits. L'architecture *SPARC* est directement dérivée de l'architecture *RISC*. Elle a été conçue pour s'adapter à différents types des systèmes en terme de performances, prix et flexibilités. D'autre part, dans la norme *SPARC*, Il existe deux versions d'architectures. Une version *SPARCv8* pour une implémentation 32 bits et une autre version *SPARCv9* en 64 bits [32]. Le principal avantage de l'architecture SPARC est l'implémentant des registres du processeur en « *Register Windows* ». Cette architecture permet des économies en nombres d'instructions de type *load/store* en mémoire et offre au compilateur la possibilité de produire un code plus efficace [33]. En effet, comparé à d'autres types d'architectures RISC, le compilateur possède une plus grande flexibilité d'affectation des registres aux variables de l'application. Pour des programmes complexes décrits avec des langages à haut niveaux d'abstraction (type C++), la réduction en terme de nombres d'instructions exécutées est significative.

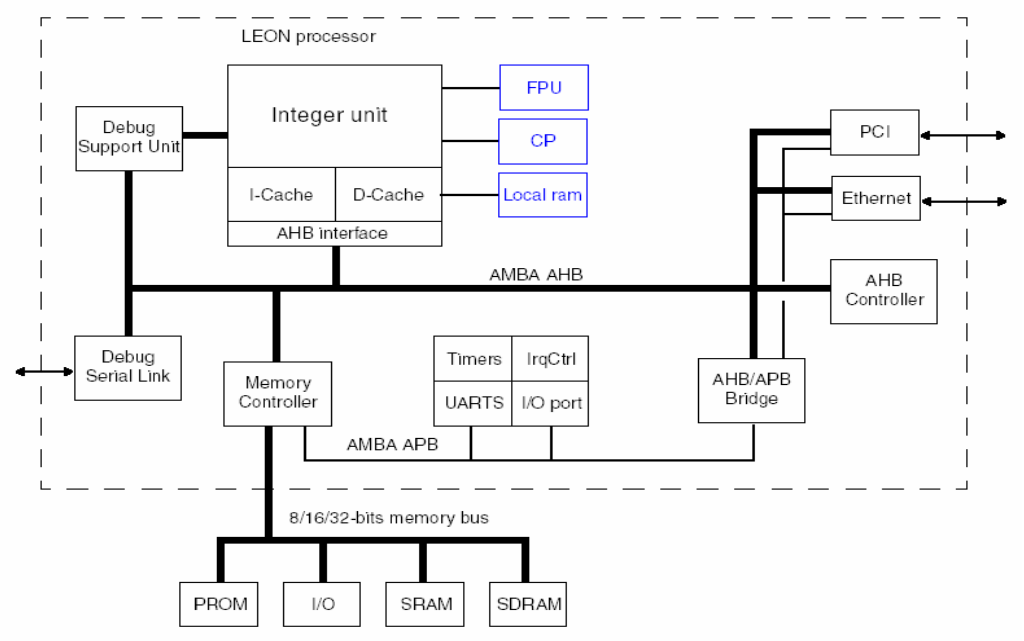


Figure 4.9 - Architecture du processeur Leon

La Figure 4.9 montre l'architecture interne du processeur Leon dont les caractéristiques techniques sont :

- Un cœur de processeur avec 5 étages de pipeline
- Un plage d'adressage de 28 bits
- Intègre un multiplieur configurable (16x16, 32x1, 32x8, 32x16 et 32x32)
- Caches d'instructions et de données séparés.
- Contrôleur d'interruptions
- Fonction '*Power-Down*'
- *WatchDog*
- Contrôleur mémoire flexible (taille du bus d'adresses et de données)
- Gestion du Bus *AMBA AHB/APB* [29, 34]
- Synthétisable et implémentable sur *FPGA/ASICs*
- Langage de description *VHDL* (*VHDL-87*)

Dans le tableau ci-dessous, nous avons, suivant le type de la technologies (*ASIC* ou *FPGA*), un aperçu des ressources et des performances du processeur :

Technology	Area	Timing
Atmel 0.18 CMOS std-cell	35K gates + RAM	165 MHz (pre-layout)
Atmel 0.25 CMOS std-cell	33K gates + RAM	140 MHz (pre-layout)
UMC 0.25 CMOS std-cell	35K gates + RAM	130 MHz (pre-layout)
Atmel 0.35 CMOS std-cell	2 mm2 + RAM	65 MHz (pre-layout)
Xilinx XC2V500-6	4,800 LUT + block RAM	65 MHz (post-layout)
Altera 20K200C-7	5,700 LCELLs + EAB RAM	49 MHz (post-layout)
Actel AX1000-3	7,600 cells + RAM	48 MHz (post-layout)

Tableau 4.4 - Ressources et performances du processeur Leon : ASIC ou FPGA

4.6.2 Caractéristiques configurables du Leon

Puisque le Leon est une IP écrite entièrement en VHDL, cela offre au concepteur de configurer son processeur suivant les besoins de l'application cible.

- Caches de données et d'instructions configurables
- Configuration du FPU

- Configuration du contrôleur de Mémoire
- Configuration des périphériques
- Configuration du bus AMBA [34]

Dans ce cas d'étude, nous nous intéressons plus particulièrement à la configurabilité des caches. En effet, l'espace des solutions architecturales est représenté par les différentes valeurs des configurations de cache d'instruction et de cache de données. Les paramètres de ces deux caches sont les suivants :

- *sets* (1 à 4) : Nombre d'ensembles dans le cache
- *setsize* (1 à 64) : Taille du cache pour chaque ensemble (*set*)
- *linesize* (4 à 8) : Nombre de mots par ligne de cache
- *replace* (lru, lrr ou rnd) : Algorithme de remplacement du cache

Le tableau 4.7, donne plus des détails au sujet de ces paramètres et leurs plages de variations.

4.6.2.1 Utilisation du processeur

Le processeur Leon est disponible sur le site de Gaisler Research [28], avec différentes versions et des nouvelles fonctionnalités et améliorations lui sont régulièrement apportées. La version utilisée au cours de ce travail est le *Leon2 version 1.0.1*. Les fichiers téléchargeables comprennent le modèle source du processeur et ses périphériques décrits en *VHDL*. Le modèle est associé à un '*testbench*' pour une simulation fonctionnelle avec différentes configurations possibles (taille du cache, etc.).

Le Leon doit être utilisé avec un compilateur respectant le jeu d'instruction *SPARCv8* [30, 31]. Le compilateur employé est *LECCS* (*Leon/ERC32 GNU cross-compiler system*) version 1.1.1 [35, 36]. Il permet de compiler les fichiers C/C++ et d'effectuer l'édition des liens. *LECCS* est un outil multi-plateformes basé sur les compilateurs de la famille *GNU* qui sont libres de droits. Le compilateur est basé sur l'outil *GCC 2.95.2* et permet la compilation d'applications simples ou ciblées pour des systèmes complexes temps réels.

4.6.2.2 Utilisation du compilateur

L'outil *LECCS* est utilisable sur diverses plateformes : *Linux*, *Solaris* et *Windows*. La compilation s'effectue sous l'environnement *Windows* et nécessite donc l'installation d'un émulateur *Unix* tel que *Cygwin*. *Cygwin* contient les divers programmes exécutables appelés lors de la compilation et l'édition de liens de l'application.

4.6.2.3 Processeur et mémoires

Le *Leon* peut être configuré pour exécuter des applications depuis différentes sources de mémoires :

- Une *boot-prom* peut être utilisée pour charger l'image compressée du programme au début de la RAM ou lors de l'édition des liens. Le programme peut être dirigé pour résider en début de RAM.
- Le programme peut aussi être directement exécuté à partir du cache. Dans ce cas, dans le flot de conception, il faut initialiser le cache pour qu'il contienne l'application.
- Enfin, une façon plus traditionnelle c'est d'exécuter l'application à partir de la ROM.

4.6.3 Intégration de l'IP Leon sur le FPGA

Pour la mise en œuvre de la plateforme d'émulation, il est nécessaire de prendre en main une IP de processeur Leon, fournit par le groupe *Gaisler research* [27, 28], en libre de droit d'utilisation et entièrement synthétisable. A cette IP nous avons rajouté deux mémoires générées par un des outils de ISE (Xilinx) avec le Core Generator [38]:

- ROM : contenant le code source de l'application
- RAM : contenant les données.

Dans un deuxième temps, nous avons chargé en ROM un petit exemple de produit de matrice permettant de tester le fonctionnement du Leon. Cette partie du travail a donc été de compiler le code C avec le compilateur LECCS [36] fournit avec le Leon et de charger ce code en ROM. Ce travail en amont permettra ensuite d'insérer le processeur Leon sur le FPGA Virtex-II [39, 40, 41, 42] de la plateforme de prototypage ADM-XRC-II [43].

4.6.3.1 Structure Hard

4.6.3.1.1 Structure du Leon

L'IP Leon est constituée d'une arborescence de modules VHDL et par l'appel de packages distingués en 2 groupes :

- Packages contenant des composants utilisés dans l'arborescence du Leon.
- Packages permettant de paramétrer des composants du Leon (Integer Unit, Cache Instruction, Cache Données, Bus Amba [34], Contrôleur de mémoires, etc.).

Dans le cas de la mise en œuvre de la plateforme d'émulation de processeurs, le principe de base est de venir modifier les paramètres de ce processeur et notamment les paramètres du « Cache Instructions » et du « Cache Données ». Par conséquent nous devrons modifier le package qui paramètre le « Cache Instructions » et le « Cache Données », c'est-à-dire le package « *device.vhd* ». Lors de la simulation comportementale (post synthèse), il est indispensable d'insérer le package « *debug.vhd* ». Par contre, lors de la synthèse, il est obligatoire de supprimer le package «*debug.vhd* » pour éviter l'échec de la synthèse. Pour la simulation, les packages « *tblib.vhd* », « *tbgen.vhd* » et « *leonlib.vhd* » sont indispensables.
Le TOP du Leon correspond à « Leon.vhd », ce TOP fait appel à « mcore.vhd » qui, fait appel à son tour à tous les périphériques : Bus AMBA [34], Memory controller, timers, UART et le cœur du Leon le processeur (*proc.vhd*). Ce processeur fait appel de son côté à l'Integer Unit (*iu.vhd*) et au Cache (*cache.vhd*).

Afin de tester le fonctionnement du processeur Leon, il est nécessaire d'ajouter une ROM et une RAM. Etant donné que nous nous intéressons à la simulation, il serait par conséquent nécessaire de créer un petit testbench VHDL, qui est tout simplement la génération d'une horloge et d'un signal de reset pour le Leon.

4.6.3.1.2 Structure Leon – Mémoires

Afin de tester le fonctionnement du processeur Leon, nous avons associé dans un module TOP l'IP Leon à une RAM et à une ROM. La RAM et la ROM sont des modules générés avec un des outils d'ISE Core Generator (Xilinx) [38]. Les signaux de mémoire du processeur Leon sont consignés dans tableau suivant :

Nom Signal	Direction	Fonction	Niveau actif
CLK	Entrée	Horloge Processeur	Front montant
lreset	Entrée	Reset Processeur	Bas
A[27..0]	Sortie	Adresses mémoire	Haut
D[31..0]	Entrée/Sortie	Bus données	Haut
ROMSN[1..0]	Sortie	Sélection ROM	Bas
OEN	Sortie	Validation sortie ROM	Bas
RAMSN[4..0]	Sortie	Sélection RAM	Bas
RAMOEN[4..0]	Sortie	Validation sortie RAM	Bas
RWEN[3..0]	Sortie	Ecriture	Bas

Tableau 4.5 - Signaux de la mémoire du processeur Leon

Etant donné que les mémoires RAM et ROM partagent le même bus de données, les sorties «q » de ces mémoires sont connectées au bus de données via des portes «3 états ». L'une ou l'autre de ces portes est active (sortie mémoire connectée au bus de donnée) lorsque le signal de validation de la mémoire l'est aussi (ROMSN = '0' ou RAMSN = '0').

La structure de notre système est comme suit :

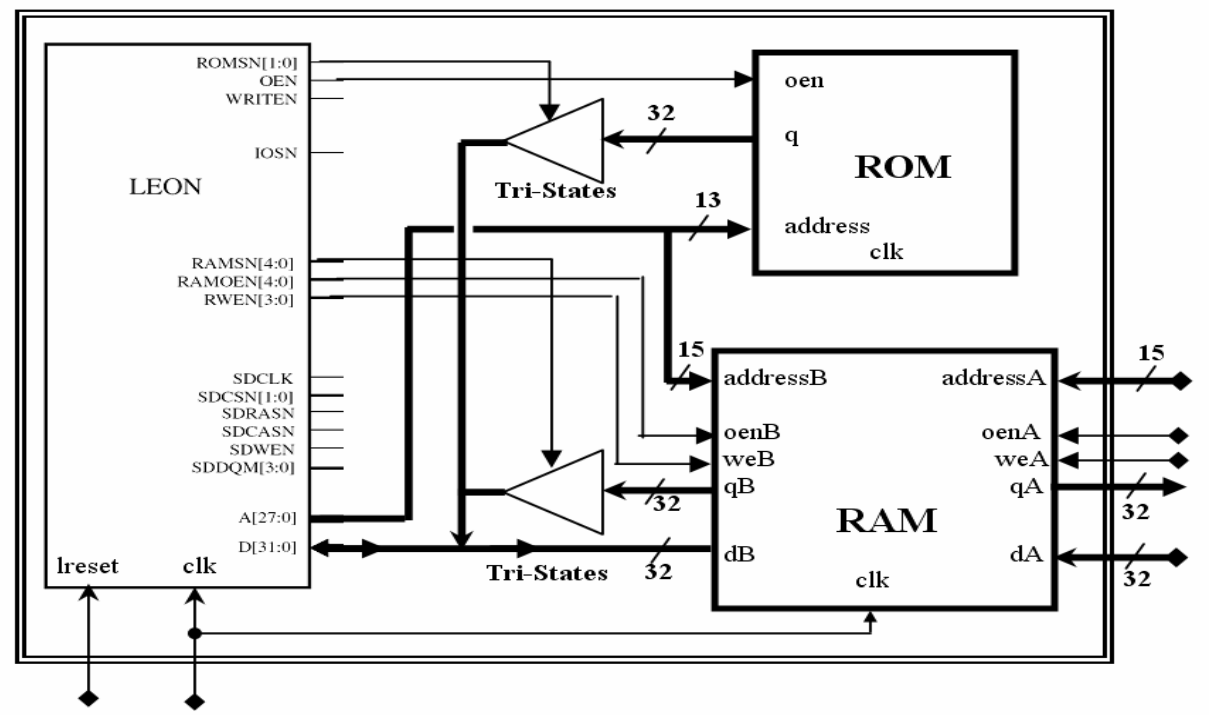


Figure 4.10 - Structure système Leon

L'usage d'une RAM bi-ports permet de tester le contenu de la RAM et donc de retrouver des valeurs stockées par le processeur lors de l'exécution du code C. Par exemple pour vérifier le bon déroulement d'une application en phase de test, nous pourrons :

- Stocker des valeurs pointées dans une zone mémoire de la RAM
- Stocker la valeur d'un Timer Hard (c'est le compteur du nombre de cycles) spécifique au Leon.

Les tailles des mémoires ont été choisies de façon arbitraire, elles ne sont pas du tout fonction de l'application C exécutée par le processeur. Les mémoires utilisent les ressources BRAM du FPGA. Ici, notre cible est un Virtex-II XC2V6000 [41]. Par conséquent, nous avons à notre disposition 144 blocs de BRAM de 18 kBits chacun, soit un total de 2592 kbits.
Les signaux de la commande des mémoires provenant du processeur Leon sont actifs au niveau bas, c'est la raison pour laquelle les signaux de commande des RAM/ROM sont configurés au niveau bas.
Le Tableau 4.6, résume les paramètres de configuration de la RAM et de la ROM.

Mémoire		Paramètres	valeur
RAM (bi-ports)	Port A (coté bus PCI)	Capacité	32767 mots (15bits Addr)
		Longueur du mot	32bits de donnée
		Configuration	Lecture/Ecriture (Read & Write)
		Mode d'écriture	Lecture après écriture (Read after Write)
		Broche Enable	Active niveau haut
		Broche Write	Active niveau haut
	Port B (coté Leon)	Capacité	32767 mots (15bits Addr)
		Longueur du mot	32bits de donnée
		Configuration	Lecture/Ecriture (Read & Write)
		Mode d'écriture	Lecture après écriture (Read after Write)
		Broche Enable	Active niveau bas
		Broche Write	Active niveau bas
ROM		Capacité	8192 (13bits Addr)
		Longueur du mot	32 (32bits donnée)
		Broche Enable	Active niveau Bas

Tableau 4.6 – Paramètres de la RAM et de la ROM

Sélectionner un fichier «.coe » généré à la suite de la compilation de l'application C pour l'initialisation de la ROM. Les composants RAM/ROM ne sont pas synthétisables (sont générées à partir du l'outil Xilinx Core Generator [44]), c'est une netlist qui est appelée durant la phase du NGDBUILD (*Cf. chapitre 2.2.6*).

4.6.3.2 Compilation d'une application pour initialiser la ROM

4.6.3.2.1 Application C

Le choix de l'application est un produit de matrice 50 x 50 (*Cf. Annexe D.1*). Ce code est une boucle infinie, car il servira ultérieurement à mesurer le nombre de cycles d'horloges nécessaires au processeur Leon pour l'exécuter, suivant différentes versions de paramètres du cache Instructions et du cache Données du processeur. Ce code est découpé en 2 parties :

- Initialisation de deux matrices diagonales composées des valeurs 86 pour la matrice A et les valeurs 222 pour la matrice B.
- Calcul du produit de matrice [A] x [B]

Les résultats du produit de matrice sont stockés en mémoire RAM par le biais d'un pointeur (ptra) dont l'adresse de départ est : *ptra=0x40000010*. Le terme de poids fort « 4 » du pointeur correspond à la sélection de la zone mémoire RAM.
Dans le programme, nous avons inséré 3 flags permettant de vérifier le déroulement correct de l'application, la valeur de ce flag est stockée en RAM :

- Flag1 : En début de code → valeur 666 à l'adresse 0 (0x40000000) de la RAM
- Flag2 : En milieu de code → valeur 777 à l'adresse 1 (0x40000004) de la RAM
- Flag3 : A la fin de code → valeur 888 à l'adresse 2 (0x40000008) de la RAM

En début de code, nous avons activé un registre (***registre=0x80000064***) qui correspond à la configuration d'un *Timer Hard*, ce Timer Hard va nous permettre de mesurer la durée d'exécution de l'application. Ce registre correspond au coefficient de multiplication de la valeur du Timer, nous avons donc choisi la valeur 9, ce qui correspond à un coefficient de multiplication= 10. Le Timer est au départ initialisé à (*0xFFFFFF*), et il décomptera durant l'exécution du code. A la fin de l'exécution, nous récupérons la valeur de ce Timer (***registre=0x80000040***) pour la charger à l'adresse 3 de la RAM (*0x4000000C*).

Au niveau de l'adressage de la RAM du côté Leon, nous avons la connexion suivante :

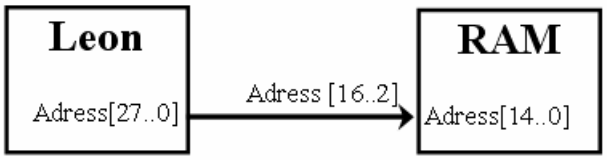


Figure 4.11 - Adressage de la RAM de côté processeur Leon

Les 2 bits d'adresse de poids faibles du Leon ne sont pas utilisés [27] car lors de l'incrémentation d'un pointeur de +1, le bus d'adresse s'incrémente de +4. Par exemple si nous choisissons l'adresse suivante : *« Côté Leon, on choisi l'adresse 1100 (0xC en hexa) => Côté RAM, cela correspond à l'adresse 11 (0x3 en hexa) »*

A présent que nous avons étudié notre application de base qui servira de programme test, nous allons passer à la méthode de compilation et d'initialisation de la ROM.

4.6.3.2.2 Compilation d'une application en C

La compilation d'une application est réalisée par les outils de compilation LECCS fournis avec le Leon [36]. Pour utiliser ces outils sous l'environnement Windows, nous

devons avoir un émulateur Unix (*Cygwin*). La figure suivante montre la chaîne de compilation étape par étape.

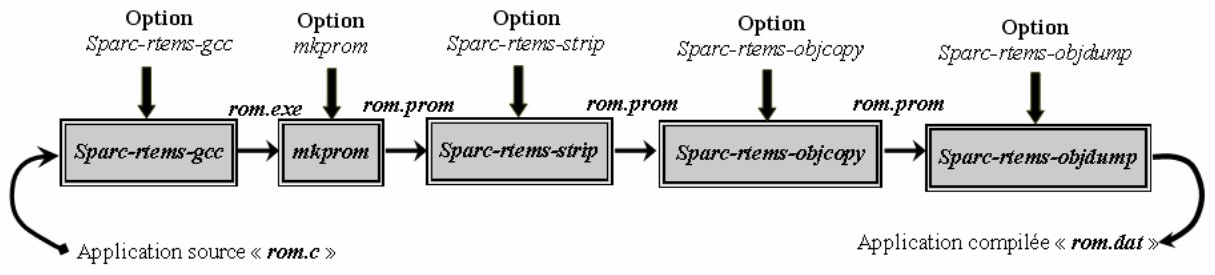


Figure 4.12 - Flot de compilation

L'enchaînement de l'exécution de chacun de ces outils est effectué par le lancement d'un petit programme C.
L'outil *sparc-rtems-gcc :* permet de compiler le code C. L'option « *-qprom* » signifie que l'application peut-être sauvegardée en ROM.
L'outil *mkprom :* permet de générer un fichier de boot pour la ROM en fonction du type de ROM et RAM utilisé.
L'outil *sparc-rtems-strip :* permet de supprimer la table des symboles.
L'outil *sparc-rtems-objcopy :* permet de supprimer les commentaires nécessaire avant la génération du fichier « *rom.dat* ».
L'outil *sparc-rtems-objdump :* permet de récupérer une partie de l'exécutable pour générer d'un fichier *« rom.dat* ».

A l'issue de ce flot de compilation, il nous reste à initialiser la ROM avec le fichier « *rom.dat* ».

4.6.3.2.3 Initialisation de la ROM

La ROM peut-être initialisée de deux façons différentes :

- Soit on effectue une simulation comportementale du système afin de tester rapidement le bon fonctionnement de l'architecture ou de l'exécution du code C *avant synthèse*.
- Soit on se trouve dans la phase de synthèse, place et route.

4.6.1.3.3.2 Initialisation de la ROM pour une simulation comportementale

Dans cette phase, nous pourrons initialiser la ROM en faisant appel à un fichier « *rom.mif* ». Ce fichier est généré au début en fonction du fichier « *rom.coe* » lors de la génération de la ROM avec l'outil Core Generator (Xilinx). Ce fichier «*rom.mif* » peut-être ensuite directement modifié sans passer par le Core Generator.
Ce fichier «*rom.mif* » est généré à partir du fichier «*rom.dat* » issue de la compilation d'un code C de l'application cible. Par conséquent, nous avons réalisé un petit outil en C prénommé « DAT2MIF » permettant de convertir ce type de fichier.

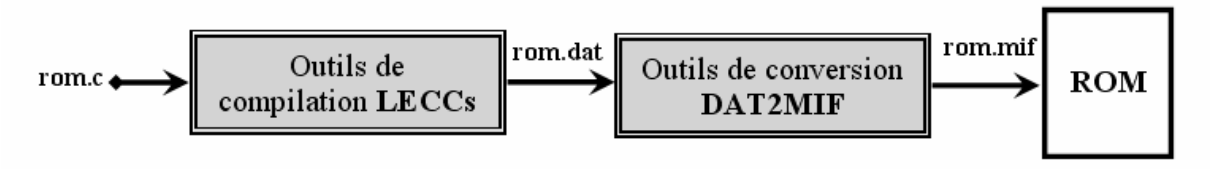


Figure 4.13 - Flot de conversion fichier DAT en MIF

Le principe de fonctionnement est tout simplement un convertisseur de format, la figure ci-dessous montre un exemple de fichier «*.dat* » avec le dual en «*.mif* » :

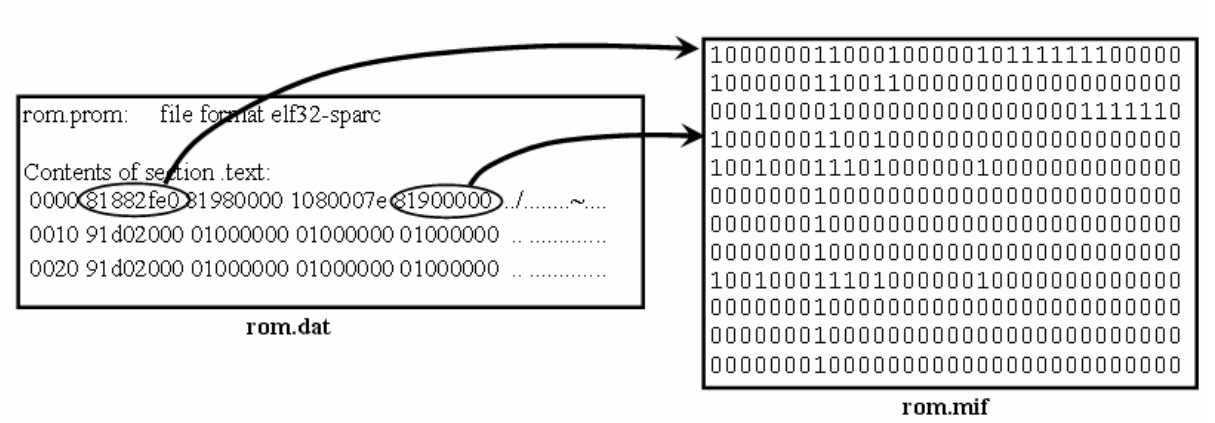


Figure 4.14 - Fonctionnement conversion fichier DAT en MIF

Le fichier «*rom.dat* » a une première colonne qui correspond à la valeur de l'adresse de *start* de la ligne. Ensuite les 4 colonnes suivantes sont les données, pour chacune d'elles, nous incrémentons le compteur d'adresse de 4. La dernière colonne correspond à la valeur des données en ASCII. Dans le fichier «*rom.mif* », le but est de transformer tout simplement la donnée du fichier « *rom.dat* » qui est une valeur hexadécimale en une valeur binaire. Pour se faire, le convertisseur «DAT2MIF » doit lire consécutivement depuis le fichier «*rom.dat* » quatre mots de 32 bits (huit valeurs en hexadécimale). Chacun de ces mots est espacé par un blanc. Par conséquent la colonne adresse n'est pas lue car la taille de son mot n'est pas de 32 bits. Par contre, la colonne adresse permet d'indiquer que les quatre colonnes suivantes sont des données.

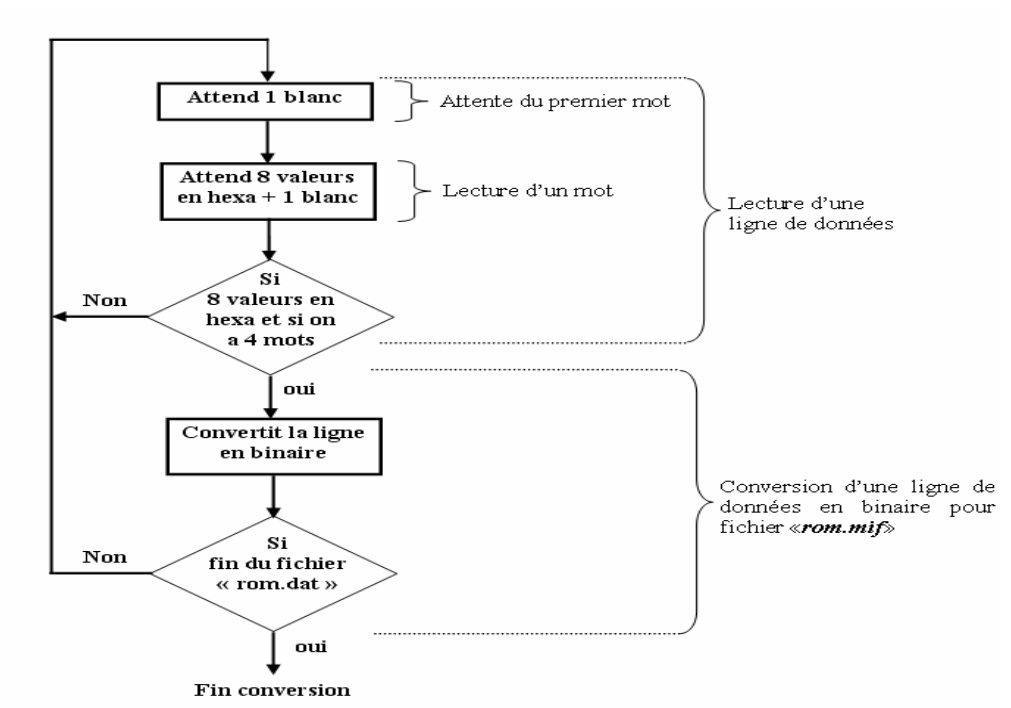


Figure 4.15 - Organigramme convertisseur DAT2MIF

4.6.1.3.3.3 Initialisation de la ROM pour une simulation post synthèse

Pour une simulation post synthèse ou une implémentation sur FPGA, il est nécessaire d'utiliser la netlist de la ROM, c'est-à-dire le fichier « *rom.edn* ». Cette netlist enferme le

contenu de la ROM. Par conséquent nous avons besoin d'un fichier « *rom.coe* » qui permettra d'initialiser la ROM par le biais de l'outil Core Generator (Xilinx).

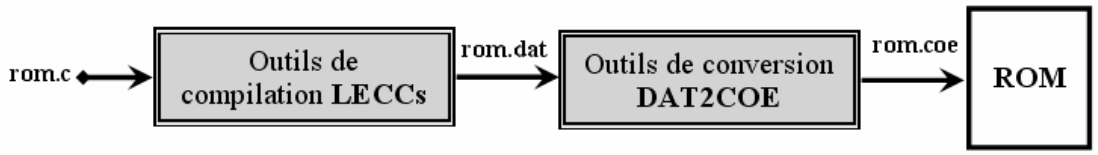


Figure 4.16 - Flot de conversion fichier DAT en COE

L'outil DAT2COE est aussi un convertisseur de format d'un fichier de type « *rom.dat* » en fichier de type « *rom.coe* ». Ce convertisseur est une macro qui fonctionne sous l'environnement « *Microsoft Excel* ». Ci-après, voici un exemple de fichier « *.dat* » avec le dual en « *.coe* » :

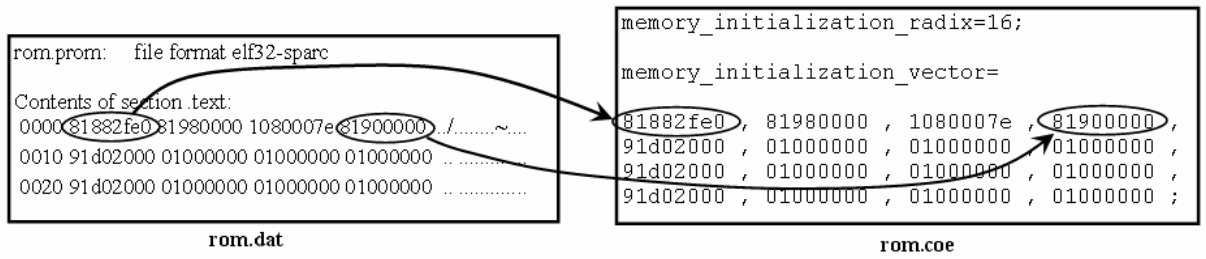


Figure 4.17 - Fonctionnement conversion fichier DAT en COE

La première colonne correspondant aux adresses est systématiquement supprimée, une virgule est insérée entre chaque donnée et la fin du fichier « rom.coe » doit se terminer par un « *;* ». Les instructions suivantes doivent être rajoutées en début de fichier ; la valeur 16 signifie que les données sont en hexadécimal :

```
memory_initialization_radix=16;
memory_initialization_vector=
```

4.6.4 Le processeur Leon dans la bocle d'exploration

4.6.4.1 Objectif

L'objectif de cette partie est de mettre en œuvre un flot automatisé permettant de paramétrer, synthétiser et implémenter une IP de processeur sur FPGA Virtex-II XC2V6000 et de recueillir les performances (nombre de cycles d'horloge et les ressources allouées sur FPGA) de ce processeur en fonction de chaque paramètre. Au début de ce chapitre (*Cf. chapitre 4.1.1*), nous avons vu l'utilisation d'une IP de processeur Leon. Cette IP de processeur sera intégrée dans le flot automatisé illustré par la Figure 4.20.

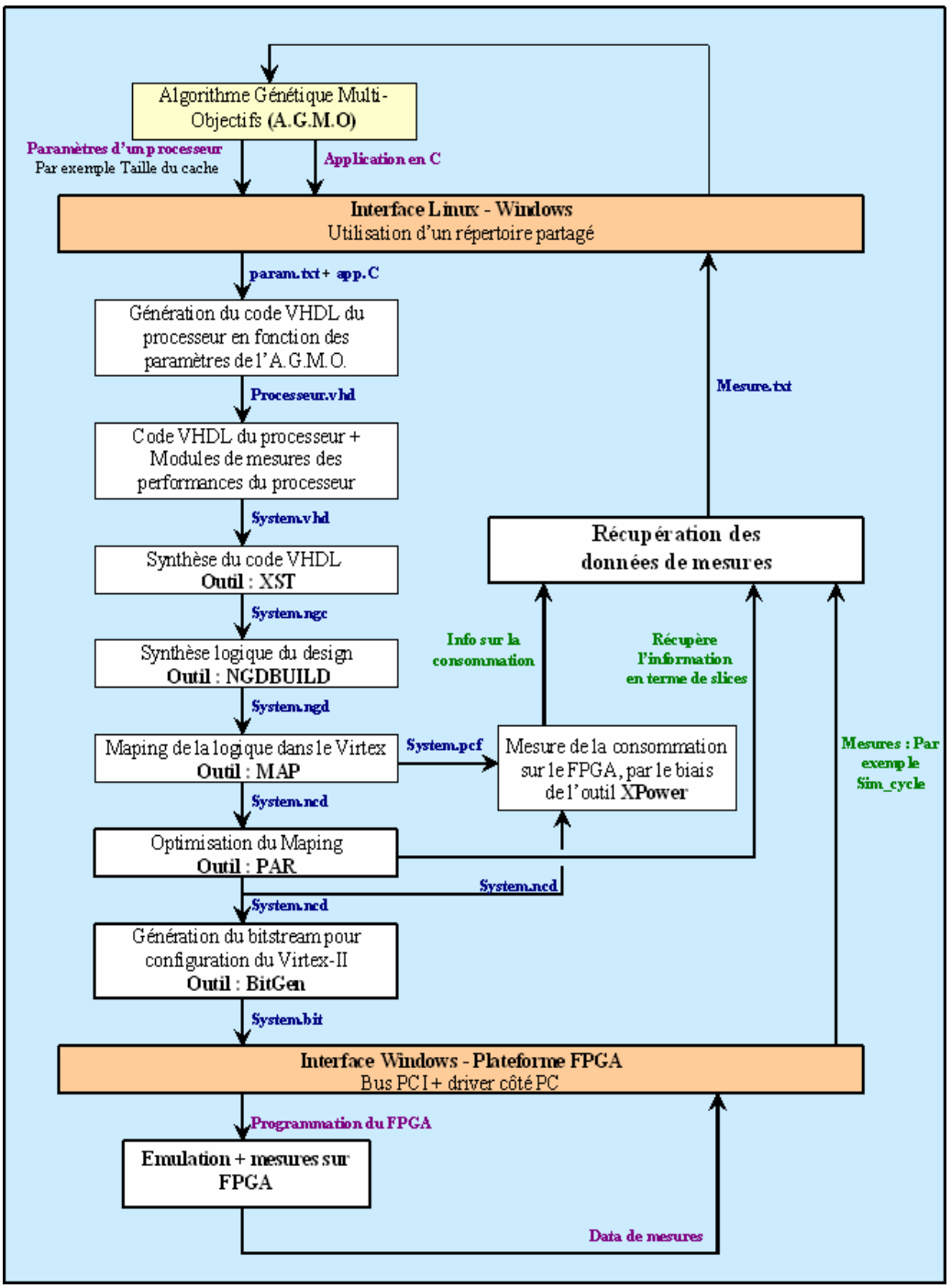


Figure 4.18 – Flot complet de l'évaluation de performances par implémentation FPGA

Par le biais du flot ISE [38, 45], nous allons recueillir les informations en terme de nombre de Slices et de BRAM utilisés dans le FPGA. Sur la plateforme FPGA, nous avons un compteur HARD qui compte le nombre de cycles d'horloges durant l'exécution d'une application C. La valeur du nombre de cycles d'horloges est transmise au PC via le bus PCI.

Dans le flot, nous avons une branche qui permet de mesurer la consommation d'énergie du FPGA. Cette branche utilise l'outil XPower (Xilinx) [38]. Or, pour avoir une mesure fiable, nous devons simuler le design. L'objectif de l'émulation de processeurs est justement de ne plus simuler car les temps de simulation deviennent exponentiellement trop grand en fonction de l'application. Par conséquent, l'outil XPower ne nous permet pas de

recueillir cette information, et il sera donc nécessaire dans un travail ultérieur de trouver une méthode pour recueillir la consommation électrique en régime dynamique.

4.6.4.2 Architecture HARD de la plateforme d'émulation

4.6.4.2.1 Structure fonctionnelle

Le processeur Leon est implanté sur une plateforme de prototypage ADM-XRC-II (*Cf. chapitre 2.3*). Le composant cible est un FPGA Virtex-II XC2V6000, connecté au bus PCI de l'ordinateur via un composant discret PLX 9656 [46]. A ce processeur Leon, sont associés différents modules de mesure. Pour une première approche nous avons un seul module qui est un compteur permettant de mesurer le nombre de cycles d'horloge. A la fin de l'exécution, un signal «*errorn* » généré par le Leon devient actif à « 0 » et stoppe le compteur et permet entre autre d'indiquer au PC que la mesure est terminée.

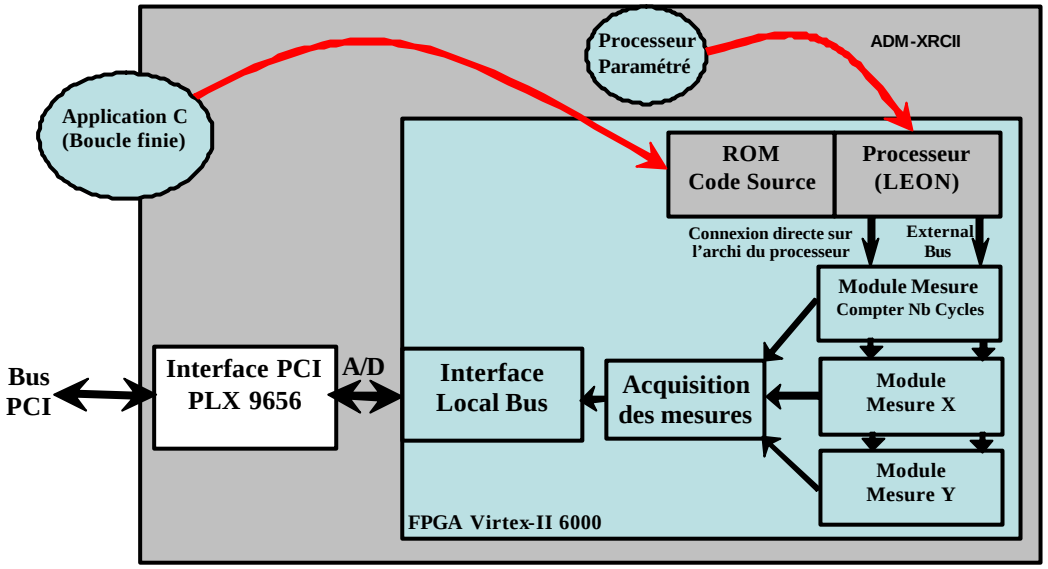


Figure 4.19 - Structure fonctionnelle architecture HARD

Les modules «Acquisition de mesures » et «Interface Local Bus » permettent de transmettre les informations au PC :

- La fin de l'exécution par le biais du signal «errorn ».
- La valeur du compteur HARD (compteur du nombre de cycles d'horloge pendant l'exécution).
- Lire des zones de la mémoire RAM afin de tester le bon déroulement du code C et de lire le Timer Hard interne au Leon (*Cf. partie 4.6.1.2*).

4.6.2.2.2 Structure comportementale

L'architecture est basée sur le processeur Leon et les mémoires (ROM/RAM). A cette architecture, nous insérons un module contrôleur de bus (*ControleurBus.vhd*) permettant au FPGA de communiquer avec le PC en utilisant une IP (*plxdssm.vhd*) fournie avec le module ADM-XRC-II [43]. Cette IP est une machine d'état permettant de gérer le protocole de communication entre le FPGA et le composant discret PLX 9656. L'ensemble de ces modules est appelé par un module TOP (*PlateformeEmulation.vhd*) où nous allons retrouver un compteur HARD permettant de compter le nombre de cycles d'horloge et un banc de registre permettant de détecter le signal «errorn » du processeur (qui indique la fin de l'émulation) et de lire la valeur du compteur HARD.

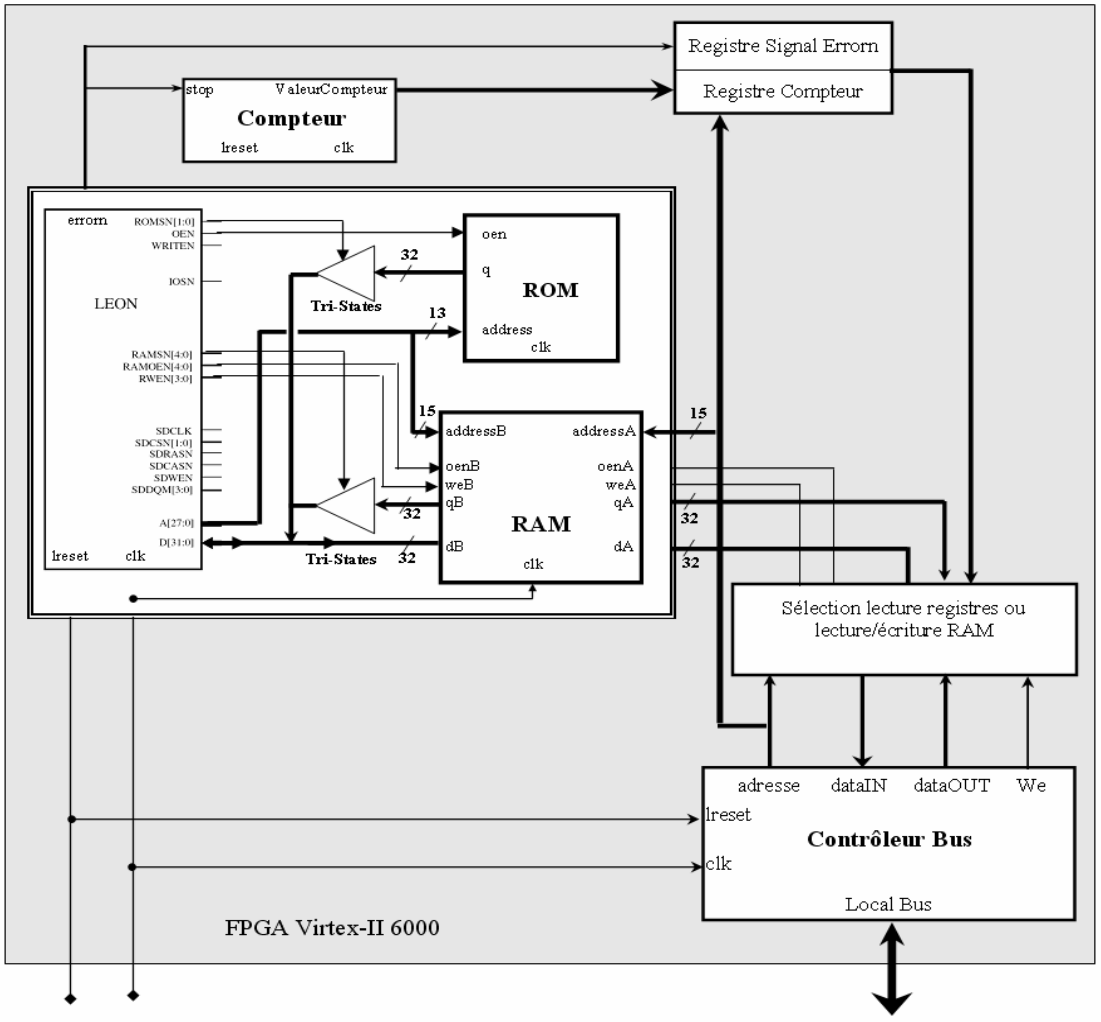


Figure 4.20 - Structure comportementale architecture HARD

Si le bit 15 d'adresse est égal à « 0 » on sélectionne le banc de registre :

- Adresse [14..0] = 0 → On lit la valeur du signal « errorn »
- Adresse [14..0] ? 0 → On lit la valeur du compteur

Si le bit 15 d'adresse est égal à « 1 » on lit/écrit dans la RAM bi-port suivant le signal « We » du contrôleur du bus.

4.6.4.2.2 Package de configuration du cache Leon

La configuration du « Cache Instruction » et du «Cache Données » du processeur Leon est effectuée par l'intermédiaire du package « *device.vhd* ». Le flot d'émulation doit donc modifier ce package (*Cf. Annexe E : Package device.vhd*) de la manière suivante :

```
constant cache_config : cache_config_type := (
    isets => 1, isetsize => 4, ilinesize => 8, ireplace => rnd, ilock
=> 0,
    dsets => 1, dsetsize => 4, dlinesize => 8, dreplace => rnd, dlock
=> 0,
    dsnoop => none, drfast => false, dwfast => false, dlram => false,
    dlramsize => 1, dlramaddr => 16#8F#);
```

Dans le Tableau 4.7 est noté la fonctionnalité des différents paramètres avec leur plage de variation. Les valeurs de la configuration par défaut du processeur sont mises en gras:

Cache concerné	Paramètre	Plage de variation	Fonctionnalité
Cache Instructions	isets	**1** ; 2 ; 4	Nombre d'ensembles dans le cache
	isetsize	1 ; 2 ; **4** ; 8 ; 16 ; 32 ; 64	Taille du cache pour chaque ensemble
	ilinesize	2 ; 4 ; **8**	Nombre de mots par ligne de cache
	ireplace	LRU ; LRR ; **RND**	Algorithme de remplacement du cache
Cache Données	dsets	**1** ; 2 ; 4	Nombre d'ensembles dans le cache
	dsetsize	1 ; 2 ; **4** ; 8 ; 16 ; 32 ; 64	Taille du cache pour chaque ensemble
	dlinesize	2 ; 4 ; **8**	Nombre de mots par ligne de cache
	dreplace	LRU ; LRR ; **RND**	Algorithme de remplacement du cache

Tableau 4.7 - Différents paramètres avec leur plage de variation du cache

Si la politique de remplacement (*ireplace* ou *dreplace*) est sélectionnée sur l'option LRR, le paramètre de « isets » ou « dsets » doit obligatoirement être égal à « *2* » [27].

A présent que la structure Hard est analysée, nous avons étudier la partie soft côté PC permettant de paramétrer le processeur Leon via le package « *device.vhd* », synthétiser le code VHDL, configurer le FPGA et communiquer avec la plateforme.

4.6.4.3 Structure Soft côté PC

4.6.4.3.1 Organisation des fichiers

Afin de gérer le flot d'émulation, une application C tournant sur PC permet d'ordonnancer les différentes tâches :

- Paramétrer le processeur Leon
- Lancer la synthèse du design Leon + modules de mesures (*PlateformeEmulation.vhd*)
- Configurer le FPGA de la plateforme ADM-XRC-II
- Recueillir les informations en terme de nombre de cycles d'horloge provenant de la plateforme, et en terme de ressources (BRAM et Slices) du FPGA via les résultats de placement-routage du flot ISE [38].

Une arborescence de répertoires permet de cloisonner les différents fichiers utiles au flot de conception géré par l'application C du PC :

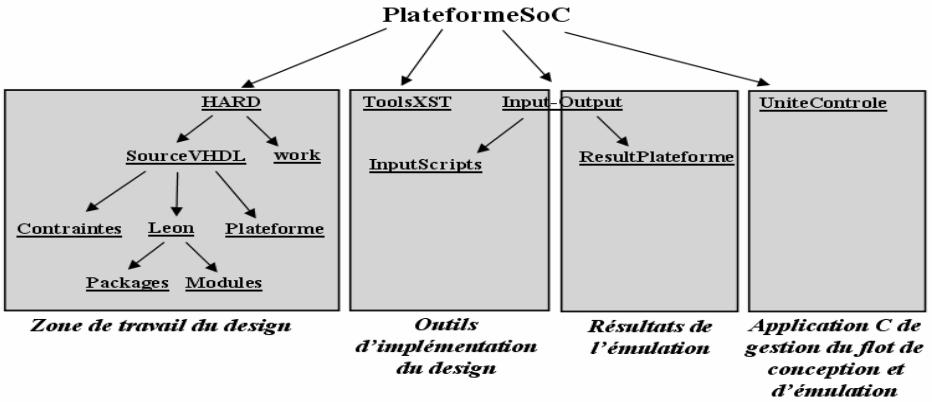


Figure 4.21 - Organisation des fichiers flot de conception

Les différents répertoires sont découpés en quatre groupes :

- Le principal est le répertoire correspondant à « *l'unité de contrôle* », c'est dans ce répertoire que se trouvent tous les fichiers permettant de contrôler l'ensemble de la plateforme d'émulation : Paramètres du processeur, exécution du flot de synthèse, implémentation du design sur la plateforme ADM-XRC-II et récupération des mesures.
- Le second, est le répertoire concernant la partie « *HARD* ». Nous allons retrouver tous les fichiers VHDL de la plateforme d'émulation :
  - Processeur Leon avec ses modules caches Instructions et Données, son package de configuration des caches « *device.vhd* ».
  - Les fichiers VHDL relatifs à la plateforme d'émulation en générale (module de communication, module TOP).
  - Le fichier de contraintes pour le placement-routage, la définition des contraintes de Timing, d'entrées/sorties du FPGA.

  Dans ce même répertoire, nous allons avoir le répertoire « *work* » où nous aurons tous les fichiers intermédiaires qui seront générés durant l'exécution du flot de synthèse et d'implémentation Xilinx.
- Le troisième répertoire concerne les différents outils Xilinx nécessaires à la synthèse et à l'implémentation sur la plateforme FPGA. Chacun de ces outils font appel à des scripts qui détermineront les options de chacun des outils.
- Le quatrième répertoire concerne la génération de fichiers de mesures de performances. Ces fichiers sont générés en fonction des résultats de l'émulation.

Nous allons étudier chacune de ces parties de façon à comprendre l'ensemble du flot de conception. La structure de base restera la même dans le cas de l'émulation avec l'algorithme exhaustif (*Cf. partie 4.6*), seul l'algorithme de recherche des paramètres changera (l'algorithme exhaustif sera remplacé par NSGA-II).

4.6.4.3.2 Unité de contrôle

L'unité de contrôle nous permet par le biais d'une fenêtre graphique de sélectionner les différents paramètres du cache qui sont définis dans la partie 4.6.2.2. Ci-dessous, nous avons la fenêtre interface, pour sélectionner les différents paramètres du cache, il suffit de cliquer sur l'onglet «*cache config* ». Nous avons ensuite une deuxième fenêtre qui s'ouvre dans laquelle nous allons choisir les différents paramètres :

Figure 4.22 - Fenêtre graphique de la sélection des différents paramètres du cache

Dans le choix des paramètres :

- Dans le cas de la Figure 4.22 où nous avons une seule case cochée pour chacun des paramètres du cache (minimum obligatoire), nous aurons une seule émulation du processeur.
- Le nombre d'émulations augmentera de façon arithmétique si nous sélectionnons plusieurs cas d'un seul paramètre (exemple ci-dessus : modification du « isetsize »)
- Le nombre d'émulations augmentera de façon géométrique en fonction du nombre de cas choisis pour chacun des paramètres : Cas de la Figure 4.22 où nous sélectionnons toutes les possibilités, nous aurons 35721 émulations (configurations possibles).

Une fois les paramètres sélectionnés, l'outil lance le flot de synthèse et de placement-routage par le biais de commandes en ligne (ANNEXE E.1). Chacune de ces commandes lance un outil du flot de conception Xilinx. Chaque outil fait appel à un script définissant les options de chacune des étapes. Le schéma ci-après montre le flot :

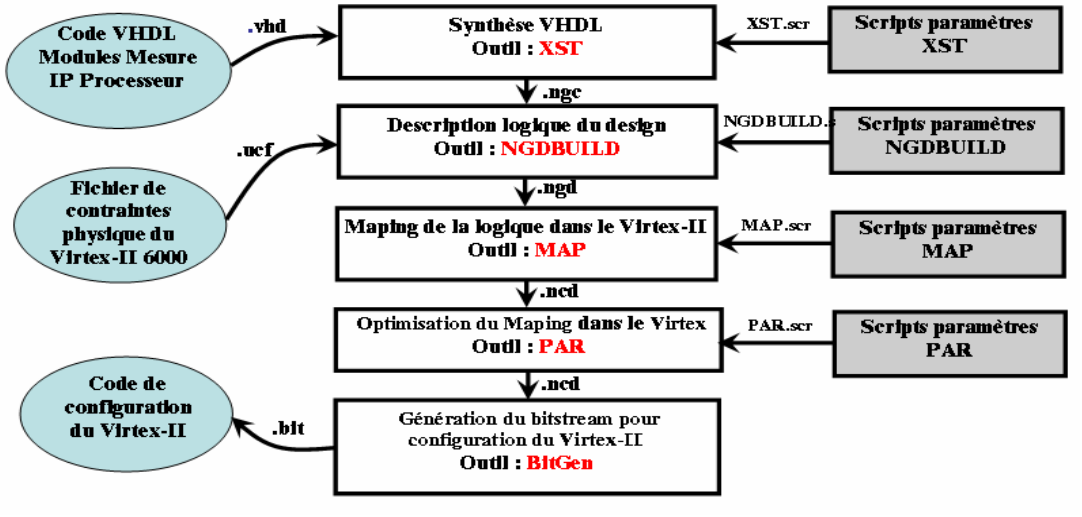


Figure 4.23 - Flot ISE Xilinx

Chacun de ces outils est classé dans le répertoire « ToolsXST ». Chaque outil utilise un script, classé dans le répertoire « InputScripts » (Figure 4.23). Le code VHDL en entrée de l'outil XST de synthèse correspond à tous les modules VHDL de la plateforme d'émulation, l'outil de la synthèse XST lit un fichier qui contient la liste et le nom de chemin de tous les modules VHDL du projet à synthétiser nommé « *PPOP.prj* », qui se trouve dans le répertoire HARD. En Annexe E.2 figure la liste de tous les fichiers VHDL de la plateforme. Après la phase de synthèse, un fichier «*PPOP.ngc* » est généré et qui sera utilisé par l'outil suivant NGDBUILD… et ainsi de suite jusqu'à la phase de génération du fichier « *PPOP.bit* » permettant de configurer le FPGA. Tous les fichiers intermédiaires générés par chacun des outils sont stockés dans le répertoire « *work* » du répertoire « *HARD* ».

A la suite des étapes du flot de synthèse ISE [38], l'outil va lancer par le biais des instructions ci-après la configuration du Virtex-II XC2V6000 de la plateforme ADM-XRC-II

```
status=ADMXRC2_ConfigureFromFile(card,"C:/PlateformeSOC/HARD/work/ppop
.bit");
 if (status != ADMXRC2_SUCCESS) {
       printf ("Failed to load the bitstream '%s': %s\n",
       " HARD/work/PPOP.bit",   ADMXRC2_GetStatusString(status));
       ret = -1;
       goto done;
 }
```

Algorithme 4.1 – Procédure pour la configuration de FPGA avec le fichier « .bit »

Ensuite l'outil attend la fin de l'exécution du processeur en lisant le registre fpgaSpace[0] correspondant au signal «errorn » du processeur. Cette boucle d'attente de signal «errorn » est représentée par le pseudo code suivant :

```
dataIn = fpgaSpace[0];   // Attente signal errorn = 0 : Adresse 0(fin
calcul de processeur Leon)
 while((dataIn != 0) &(j!=860)){
       dataIn = fpgaSpace[0];
       if((i==32760) & (j < 860)){
             i=0;
             j++;
       }
       i++;
 }
```

Algorithme 4.2 – Boucle d'attente du signal errorn

A la fin d'une émulation, l'outil génère dans le répertoire « Input-Output/ResultPlateforme/ » un fichier « *FitnessOut.txt* » dans lequel se trouve les performances du processeur en fonction d'une configuration donnée. Dans l'exemple ci-après, nous allons retrouver :

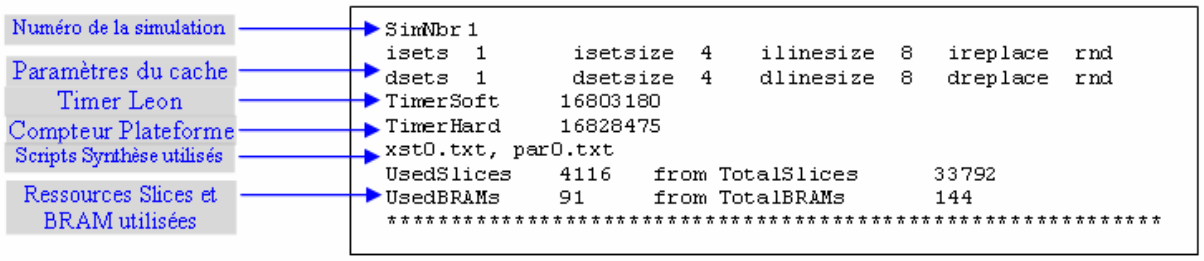


Figure 4.24 – Exemple de résultat pour une cofiguation de processeur Leon

A chaque nouvelle émulation, l'outil complète la liste dans le fichier « *FitnessOut.txt* ». Dans l'exemple ci-après, nous faisons varier le paramètre «dsetsize », la valeur du Timer Soft, Timer Hard, Slices et BRAM :

```
SimNbr 1
 isets 1       isetsize 4    ilinesize 8   ireplace rnd
 dsets 1       dsetsize 4    dlinesize 8   dreplace rnd
TimerSoft      16803180
TimerHard      16828475
xst0.txt, par0.txt
UsedSlices     4116    from TotalSlices      33792
UsedBRAMs      91      from TotalBRAMs       144

************************************************************

SimNbr 2
 isets 1       isetsize 4    ilinesize 8   ireplace rnd
 dsets 1       dsetsize 16   dlinesize 8   dreplace rnd
TimerSoft      17494750
TimerHard      17519991
xst0.txt, par0.txt
UsedSlices     4107    from TotalSlices      33792
UsedBRAMs      79      from TotalBRAMs       144
```

Figure 4.25 – impact de la configuration sur les performances

La fonctionnalité des deux Timer (Hard et Soft) permet de vérifier le bon déroulement de l'application tournant sur le processeur. Si Timer Hard et Timer Soft sont sensiblement équivalents c'est qu'il n'y a pas eu de problème dans l'exécution. Les données en terme de ressources « *BRAMs* et *Slices* » du Virtex-II sont données par le biais de l'outil de placement-routage PAR (Xilinx).

4.6.4.3.3 Scripts de Synthèse et de placement routage

L'optimisation d'un processeur en terme de ressources matérielles, peut-être réalisée en jouant sur un paramètre autre que celui du cache. Elle va être réalisée en modifiant les scripts de synthèse et de placement-routage des outils respectifs XST et PAR. Les options de ces outils sont actuellement non modifiables par interface graphique. Par conséquent, nous choisissons un paramètre de cache par appel de fichier. Un exemple de script pour l'outil de synthèse XST est donné en Annexe E.3, celui de l'outil de placement-routage PAR est donné en Annexe E.4.

Pour le script de synthèse, nous allons jouer sur quatre options de l'outil XST [38, 47] qui sont :

- *Opt_Mode* : Objectifs d'optimisation → *2 options choisies*
- *Opt_Level* : Effort d'optimisation → *2 options*
- *Slice_Utilization* : Pourcentage de remplissage des slices → *2 options*
- *Slice_Utilization_Maxmargin* : ? de remplissage des slices → *2 options*

Etant donné que nous jouons sur 4 paramètres de synthèse, nous allons avoir pour un paramètre de cache 16 synthèses différentes, l'arborescence ci-dessous montre les différents cas possibles de synthèse :

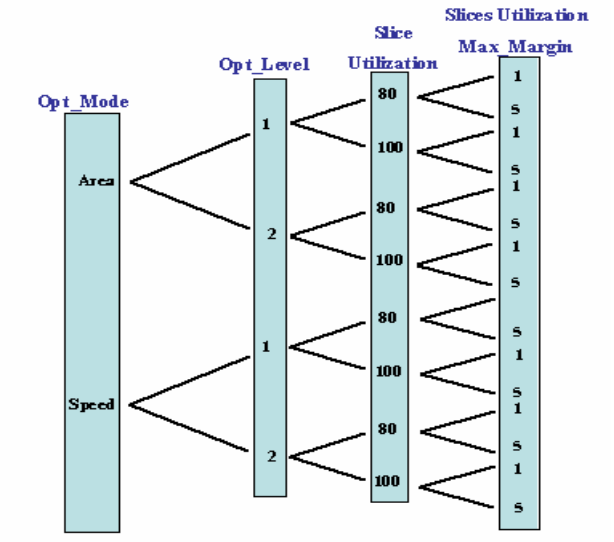


Figure 4.26 - Arborescence des options de Synthèse

Pour le script de placement-routage l'outil PAR [38], nous allons jouer sur deux paramètres :

- *Ol* : Niveau d'effort pour le placement routage → *5 options choisies*
- *n* : Nombre d'itérations pour l'optimisation du placement-routage → *3options*

Le placement-routage est réalisé en aval de la synthèse. D'où, pour chaque paramètre de cache et pour chaque paramètre de synthèse, nous aurons 15 placement-routages.

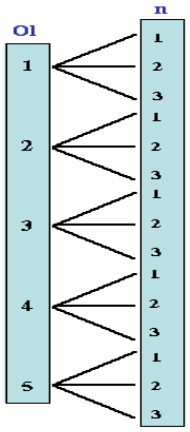


Figure 4.27 - Arborescence des options de placement-routage

Pour un paramètre de cache donné, nous allons pouvoir influencer le nombre de ressources en jouant uniquement avec les paramètres de synthèse et de placement-routage. Dans notre cas, si nous utilisons toutes les combinaisons de synthèse et de placement-routage vues précédemment, nous aurons pour un seul paramètre de cache : *240* implémentations et donc *240 émulations* possibles.

4.6.5 Résultats de l'émulation

4.6.5.1 Exploration exhaustive

La Figure 4.28, montre le résultat de l'exploration exhaustive de l'espace de paramétrage du cache instructions et du cache données pour le processeur Leon. Dans ce cas, nous avons trois objectifs à évaluer en fonction de chaque configuration du cache:

- Durée d'exécution en terme de nombre des cycles d'horloge du processeur.
- Le nombre de BRAM nécessaire pour chaque configuration du cache (ressources FPGA).
- Et le nombre de slices (ressources FPGA).

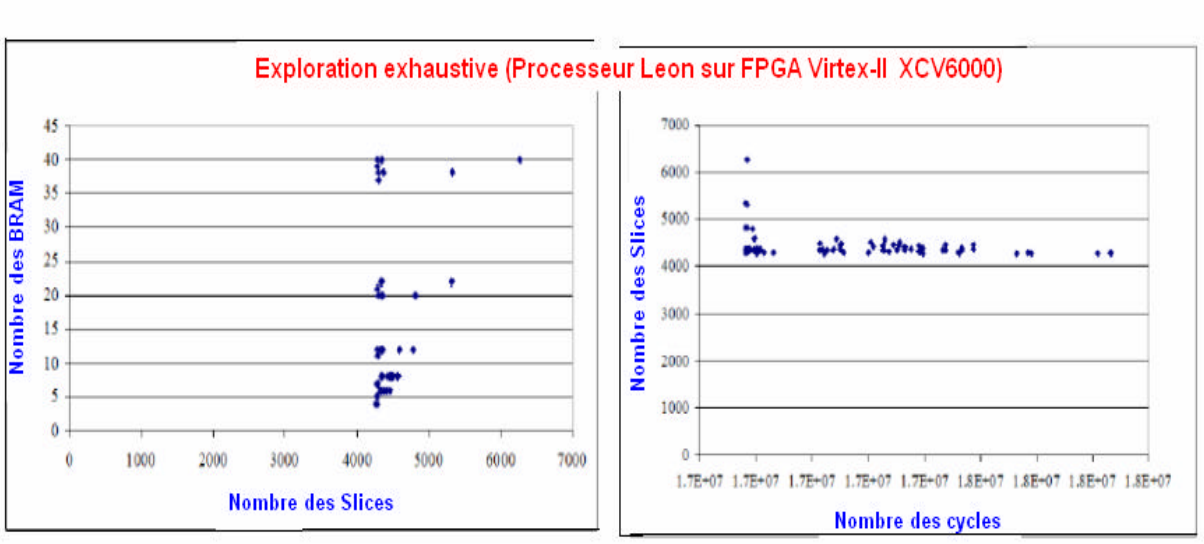


Figure 4.28 – Résultat de l'exploration exhaustive du processeur Leon sur FPGA

La Figure 4.28 montre l'existence d'un phénomène « d'hystérésis » dans la variation des performances (nombre de cycles dans ce cas) en fonction des ressources (nombre de BRAMs et des Slices). Ainsi, on constate qu'il y a une variation significative du nombre de cycles

(suite au changement de la configuration des caches) nécessaires à l'exécution de l'application, pour la même quantité de ressources utilisées par le FPGA. L'existence de ce phénomène s'explique par la granularité de l'allocation des ressources FPGA.

4.6.5.2 Exploration avec l'algorithme NSGA-II

Dans ce cas nous avons trois objectifs à optimiser :

1) Nombre de cycles d'horloge du microprocesseur
2) Nombre de BRAMs (ressource FPGA)
3) Nombre de Silices (ressource FPGA)

Nous avons lancé l'exploration de cet espace de solutions avec l'algorithme NSGA-II en utilisant les paramètres de réglage suivants :

Nom du paramètre	Valeur
Taille de la population	32
Type de Codage chromosome	Binaire de taille 18 bits
Nombre des variables de l'espace d'exploration	13
Nombre des fonctions objectifs	3
Probabilité de croisement	0,80
Probabilité de mutation	0,027
Nombre des générations	5

Tableau 4.8 – Paramètres de l'algorithme NSGA-II appliqué au FPGA

Le codage des différents paramètres du microprocesseur Leon et les options des outils de conception FPGA Xilinx (XST et PAR) est assuré par un chromosome binaire de longueur 18 bits (Figure 4.29).

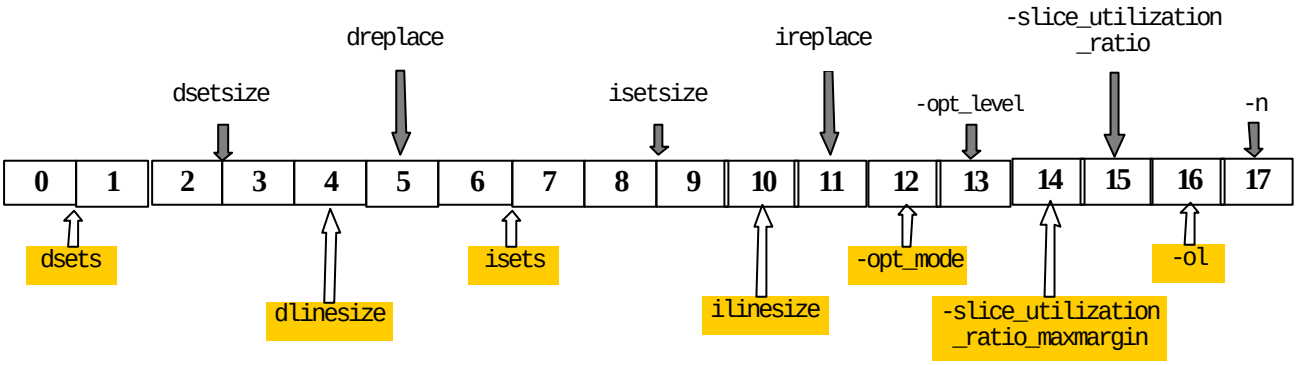


Figure 4.29 - Codage de chromosome

En l'Annexe E.5, il y a plus de détails au sujet des valeurs possibles pour chaque paramètre.

Le Tableau 4.9, montre le résultat (chaque individu de la population avec les valeurs des fonctions fitness et le rang du front de Pareto) de l'exploration par l'algorithme NSGA-II à la 5^{ième} génération.

Numéro Ind.	Nombre Cycles (f0)	Nombre BRAM (f1)	Nombre Slices (f2)	Front Pareto
0	231666912	76	4536	1
1	235174128	68	4285	1
2	233702704	72	4996	1
3	231666832	76	5391	2
4	230015104	82	7356	3
5	275851936	87	4034	3
6	233753168	71	3961	1
7	257922624	63	4296	1
8	221717104	83	5471	1
9	221744608	90	4988	1
10	235083696	69	4266	2
11	221781648	85	4960	1
12	229950192	82	4523	1
13	221883840	76	5217	1
14	221717984	105	4186	1
15	272065024	75	4013	2
16	234670464	69	3940	1
17	235378640	69	4600	3
18	221766800	93	7210	2
19	244069456	68	4256	1
20	221781056	93	4144	1
21	221849952	82	5820	1
22	221749408	86	4577	1
23	221781024	85	5687	2
24	224212672	79	5902	2
25	227573888	83	6276	3
26	230743712	76	6102	2
27	272946752	87	6010	4
28	226280032	74	4556	1
29	230743712	76	6102	2
30	221694368	108	11108	1
31	224296608	84	6405	3

Tableau 4.9 – Valeurs des fitness de chaque individu à la dernière génération du NSGA-II

La Figure 4.30 représente le résultat de l'exploration en terme de deux premiers fronts de Pareto.

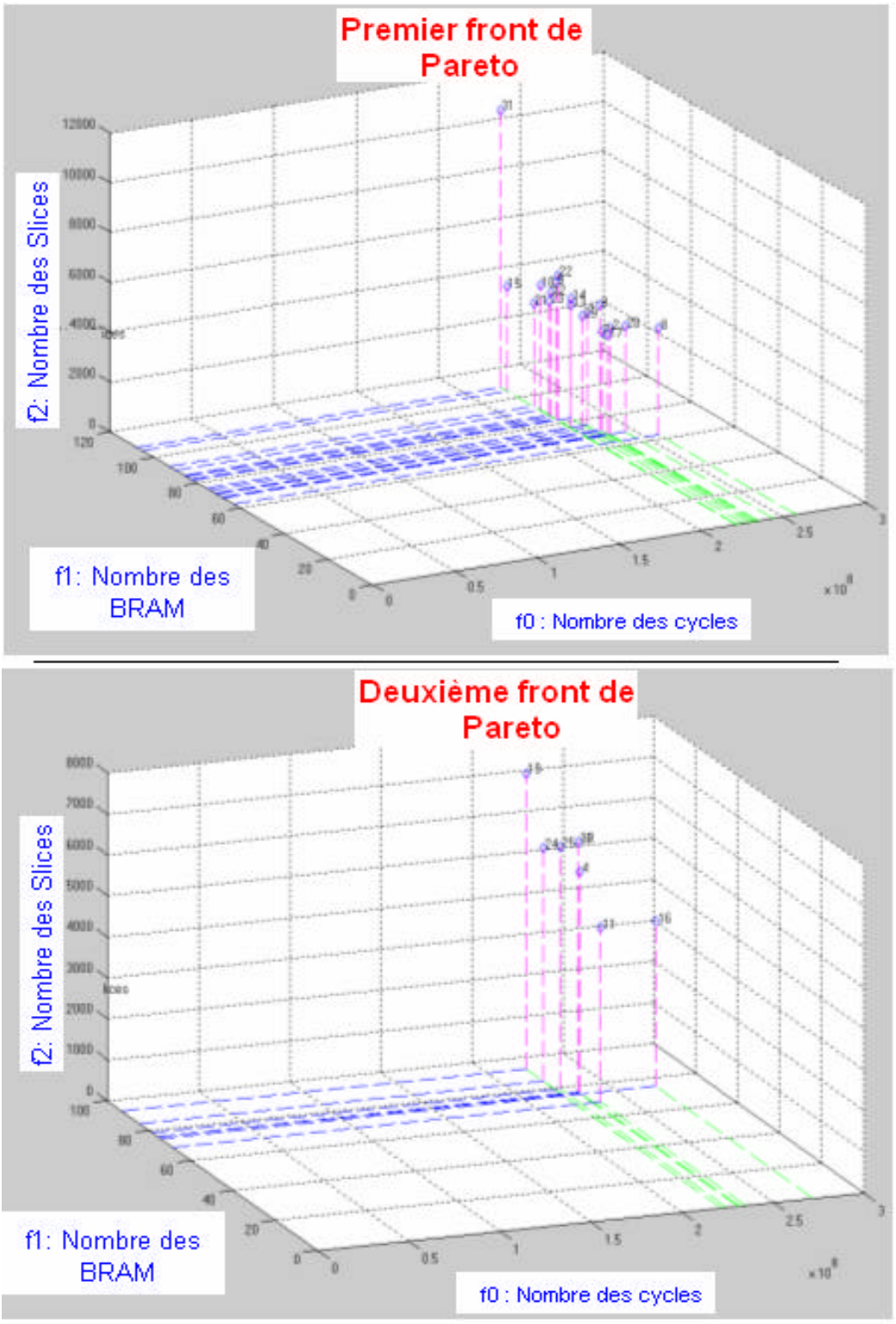


Figure 4.30 – Résultat de l'exploration multiobjectif du processeur Leon sur FPGA

La Figure 4.30 offre au concepteur un précieux outil pour la comparaison et le choix des solutions, pour la sélection des paramètres de l'architecture la plus compatible à l'application software à exécuter (en terme de performance et de ressources utilisées sur le FPGA).

Dans cette exploration, la détermination des performances (les fonctions objectifs) est basée sur l'exécution réelle de l'application sur FPGA. Cette technique est plus efficace que l'évaluation à base de simulation, en terme de précision dans les valeurs des performances. La majorité des travaux réalisés par d'autres équipes de recherche dans ce domaine, emploie des

simulations pour évaluer les performances. Comme l'exemple de la référence [102], propose une méthode d'exploration à base de simulation. Les auteurs ont exploité la dépendance des paramètres de l'architecture pour réduire l'espace des solutions à explorer.

4.7. Conclusion

Cette étude montre beaucoup d'intérêt en particulier pour des applications embarquées SOC. Les solutions obtenues dans le front de Pareto de premier ordre sont idéales pour l'espace des solutions microarchitecture du processeur exploré. Aussi, les résultats confirment bien l'efficacité de cette méthodologie et en particulier la maîtrise de l'optimisation à la fois des trois grandeurs : (1) le nombre de cycles, (2) la surface en silicium et (3) la puissance dissipée d'un microprocesseur au lieu d'une optimisation monobjectif, en générale le nombre de cycles [26].

Le résultat est un ensemble de solutions optimales (front de Pareto de premier ordre, c'est une surface de compromis) représentée sous forme d'une courbe 3D (*Sim_cycle, Aera, Power*) et qui offre à l'utilisateur beaucoup de degrés de liberté et de lisibilité pour le choix d'une solution encore plus personnalisé à ses besoins.
Aussi, la convergence de cette méthodologie vers l'optimal de Pareto est rapide (de l'ordre de 50 générations).

L'outil XPower (Xilinx) [38], permet de mesurer la consommation d'énergie du FPGA. Or, pour avoir une mesure fiable, nous devons simuler le design. L'objectif de l'émulation de processeurs est justement de ne plus simuler car les temps de simulation deviennent exponentiellement trop grands en fonction de l'application. Par conséquent, l'outil XPower ne nous permet pas de recueillir cette information, et il serait donc nécessaire dans un travail ultérieur de trouver une méthode pour recueillir la consommation électrique en régime dynamique.

Pour une configuration donnée du système, nous avons la possibilité d'influencer la taille des ressources FPGA en jouant uniquement sur les options de synthèse et de placement-routage. Par conséquent, avec le choix de ces options qui sont relatives aux outils de conception, l'espace des solutions à explorer devient encore plus large.

Dans ce chapitre nous avons préposé des exécutions directes de l'application pour l'évaluation des performances, rendue possible par des implémentations du système à évaluer sur un FPGA.

Nous avons constaté que les fronts de Pareto (Figure 4.30) dans le cas des implémentations sur FPGA, ne sont pas homogènes à cause de la granularité des cellules sur l'agglomération du FPGA.

CHAPITRE 5

APPLICATION RADIO-LOGICIELLE

Dans ce chapitre, nous nous intéressons à l'étude de la chaîne de transmission WCDMA utilisée par la norme UMTS pour le téléphone mobile de 3ème génération. Notre but est d'écrire le coder turbo code en langage C pour l'utiliser comme application cible. Dans ce cas, l'architecture à explorer, pour supporter le turbo code, est celle du processeur SuperScalar.

5.1 Chaîne de transmission W-CDMA

Les systèmes de télécommunication de deuxième génération, comme le GSM (*Global Système for Mobile communications*), ont permis aux communications vocales de s'affranchir de la traditionnelle paire de cuivre et de gérer efficacement la mobilité de leurs utilisateurs.
A l'heure actuelle, le nombre de téléphones mobiles dans certains pays dépasse celui des lignes fixes et le taux de pénétration y excède parfois 70% de la population. Mais les capacités en transmission de données de systèmes de deuxième génération sont encore limitées. Les futurs systèmes de troisième génération devront être à même d'offrir des services hauts débit permettant de transmettre images et vidéos ainsi qu'une connexion Internet haut débit. Ces systèmes de télécommunication de troisième génération sont connus sous le terme «UMTS » (*Universal Mobile Telecommunication System*).
Le WCDMA (*Wideband Code Division Multiple Access*) en est la principale interface air. Elle sera utilisée tant en Europe qu'en Asie et cela dans la même bande de fréquence, autour de 2 GHz. Le large marché couvert par le WCDMA et ses multiples capacités multimédia vont sans nul doute créer de nouvelles opportunités pour les constructeurs, les opérateurs ainsi que pour les fournisseurs de contenus ou d'applications.

5.1.1 Le WCDMA

Dans les différents forums de normalisation, la technique WCDMA s'est révélée être celle qui a été adaptée le plus largement pour l'UMTS. Le 3GPP (*3rd Generation Partenarship Project*) prend en charge la réalisation de sa spécification. Il regroupe de nombreux organismes de normalisation tant en Europe que dans le reste du monde (Japon, Corée, Chine et Etats-Unis). Au sein du GPP, le WCDMA est appelé UTRA (*Universal Terrestrial Radio Access*) FDD (*Frequency Division Duplex*) et TDD (Time Division Duplex). Le terme WCDMA étant employé pour couvrir à la fois le mode FDD et TDD.

5.1.2 Les interfaces air et l'allocation du spectre

Les études sur les systèmes de troisième génération ont vu le jour en 1992. Durant l'une de ses réunions que le WARC (*World Administrative RADIO Conférence*) de l'UTU (*International Telecommunication Union*) a identifiée les fréquences autour de 2 GHz comme celles qui ont été attribuées aux futurs systèmes appelés IMT-2000 (*International Mobile Telephony 2000*). Dans cette structure, plusieurs interfaces air ont été définies, basées aussi bien sur la technologie CDMA (*Code Division Multiple Access*) que sur la technologie TDMA (*Time Division Multiple Access*).

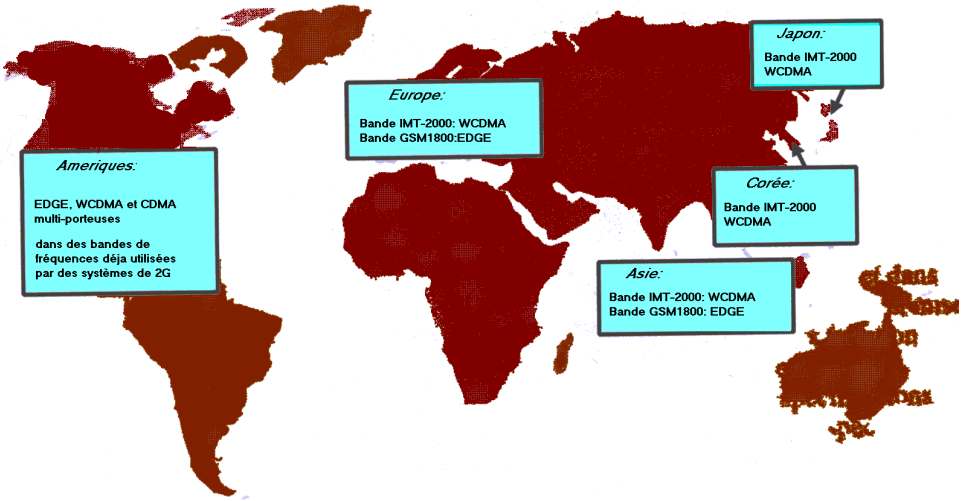


Figure 5.1 - Les interfaces air suivant les zones géographiques

En Europe et dans la plupart des pays asiatiques, les bandes de fréquences IMT-2000 d'une largeur de 60MHz (1920-1980MHz et 2110-2170MHz) seront disponibles pour le WCDMA FDD. Par contre, le spectre pour le WCDMA TDD varie selon les pays. En Europe, il est prévu de réserver 25 MHZ pour des licences TDD, dans les bandes 1900-1920MHz et 2020-2025MHz, alors que la bande 2010-2020MHz est réservée pour des applications différents, une pour le sens montant (*Uplink*) et l'autre pour le sens descendant (*Dowenlink*), séparées par des intervalle fréquentiel. Tandis que, les systèmes TDD utilisent la même bande de fréquence pour les deux sens, *Uplink* et *Dowenlink*.

5.1.3 Le WCDMA et les interfaces air de deuxième génération

Le GSM et l'IS-95 (la norme des systèmes *cdmaOne*) sont les deux principales interfaces air de 2ème génération.
Pour mieux comprendre les origines des différences entre les systèmes 2G et 3G, il est important de connaître les nouveaux besoins des systèmes de troisième génération :

- Haut débit (allant jusqu'à 2Mbit/s),
- Débit variable (bande passante à la demande),
- Multiplexage de services ayant une qualité de service différente (voix, vidéo, données, etc.) sur une même connexion,
- Délai faible pour les services temps réel, plus important et variable pour certains services de données,
- Qualité d'un taux d'erreur trame de 10% à un taux d'erreur bit de 10^{-6},
- Coexistence des systèmes 2G et 3G,
- *Handovers* inter-systèmes,
- Support de trafics *Uplink* et *Downlink* asymétriques,
- Forte efficacité spectrale,
- Coexistence des modes FDD et TDD.

Les Tableau 5.1 et 5.2, résument les principales différences entre les interfaces air WCDMA, GSM et IS-95.

	WCDMA	GSM
Espacement entre porteuses	5 MHz	200 KHz
Fréquence de contrôle de puissance	1500 Hz	2 Hz (ou moins)
Contrôle de qualité	Algorithmes de gestion des ressources radio	Plan de fréquences
Diversité de fréquence	Récepteur Rake	Saut de fréquence
Transmission de données	En fonction de la charge	En fonction du nombre de time slot disponibles (en GPRS)
Diversité de transmission Downlink	Supportée, augmente la capacité Downlink	Non supportée par la norme, peut être implémentée

Tableau 5.1 - Principales différences entre les interfaces air WCDMA et GSM

	WCDMA	IS-95
Espacement entre porteuses	5 MHz	1,25 MHz
Débit chip (débit de la séquence de code)	3,84 Mcp/s	1,2288 Mcp/s
Fréquence de contrôle de puissance	1500 Hz (uplink et downlink)	Uplink: 800Hz Downlink: lent
Synchronisation des stations de base	Pas nécessaire	Nécessaire
Handover inter-fréquence	Oui	Possible
Algorithmes de gestion des ressources radio	Oui	Pas nécessaires pour la voix
Transmission de données	En fonction de la charge	Mode circuit
Diversité de transmission Downlink	Supportée, augmente la capacité Downlink	Non supportée par la norme

Tableau 5.2 - Principales différences entre les interfaces air WCDMA et IS-95

5.1.4 Principaux paramètres de WCDMA

Présentons maintenant brièvement les principaux paramètres du WCDMA :

- Le WCDMA est un système d'accès multiple par répartition de code utilisant une modulation par séquence directe (DS-WCDMA, *Direct Sequence Wideband Code Division Multiple Access*). Cela signifie que les bits correspondants aux données utilisateurs sont étalés sur une large bande passante en multipliant ces données par une séquence pseudo-aléatoire de bits (appelée *chips*) provenant des codes d'étalement CDMA. Afin de pouvoir supporter des débits très élevés (jusqu'a 2 Mbit/s), le WCDMA utilise des transmissions à facteur d'étalement variable et à multiple codes. Ce système est illustré dans la Figure 5.2.
- Le débit *chips* (débit binaire de la séquence de code) de 3.84 Mc/s donne une bande passante par porteuse de l'ordre de 5 MHz supportant des débits utilisateur importants. Elle a en outre un impact bénéfique sur les performances du système. En fonction de licence qui lui a été attribuée, l'opérateur peut utiliser plusieurs porteuses pour augmenter la capacité de son réseau. L'espacement des porteuses peut être choisi par pas de 200 KHz, entre 4,4 et 5 MHz, selon le niveau d'interférence entre les porteuses.
- Le WCDMA permet de supporter des débits utilisateur variable et de proposer aux utilisateurs de la bande passante à la demande (BoD, *Bandwith on Demand*). A chaque utilisateur est attribué une trame d'une durée de 10ms, durant laquelle le débit est constant. Cependant, ce débit peut varier d'une trame à l'autre. La Figure 5.2 présente également cette fonctionnalité. Cette allocation dynamique de la capacité est contrôlée par le réseau afin d'obtenir un débit optimal pour les services par paquet.

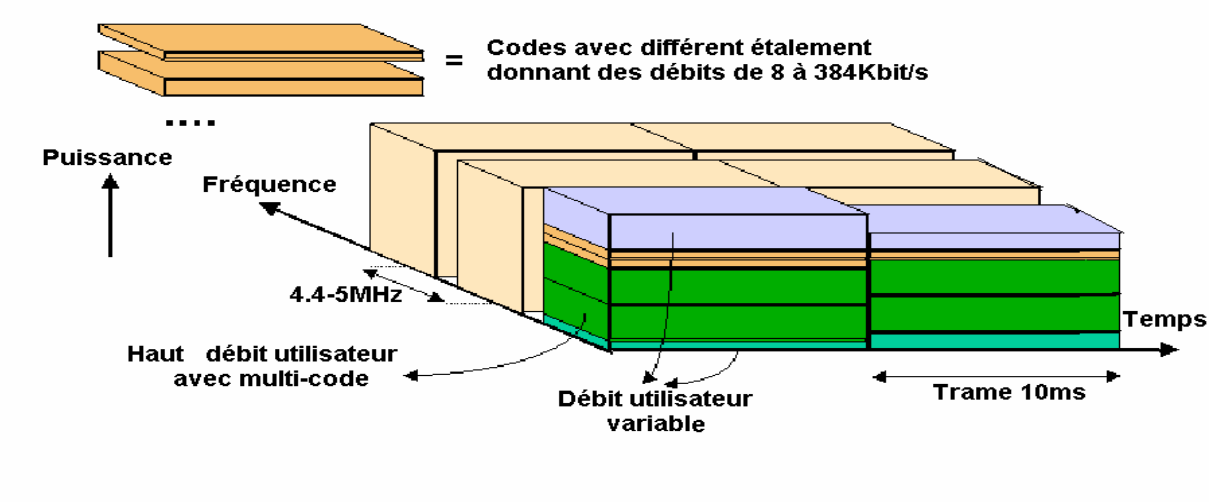


Figure 5.2 - Allocation de la bande passante en WCDMA dans l'espace temps-fréquence-code

Le WCDMA fonctionne en deux modes FDD (*Frequency Division Duplex*) et TDD (Time Division Duplex). Dans le premier cas, deux bandes passantes de 5 MHz sont utilisées, l'une pour le sens montant (*Uplink*), l'autre pour le sens descendant (*Downlink*) alors que dans le mode TDD, une seule bande passante de 5 MHz est utilisée pour le deux sens.

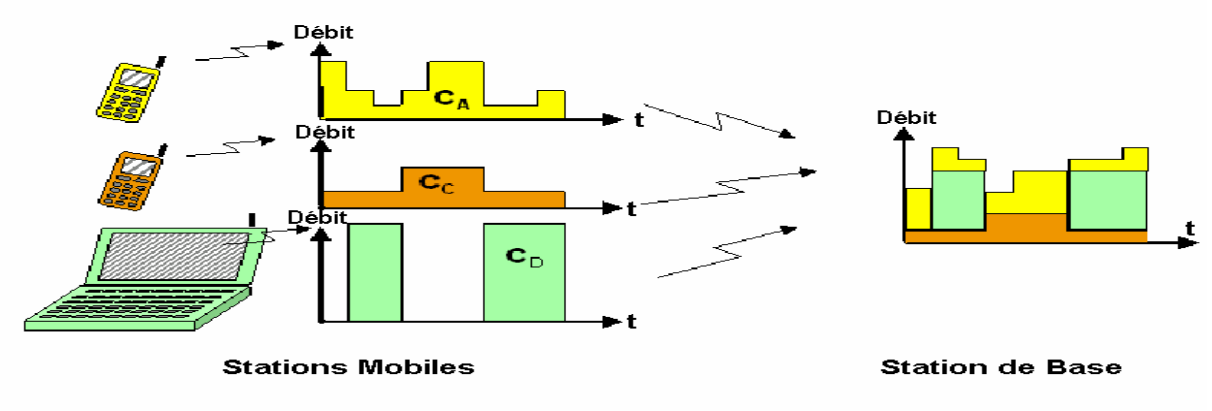


Figure 5.3 - Allocation dynamique des ressources

- Contrairement à l'IS-95, le WCDMA ne nécessite pas de synchronisation des stations de base. Il n'y a donc pas besoin de référence globale de temps tel que le GPS. Le déploiement des stations de base micro ou *indoor* en est donc facilité.
- L'interface air WCDMA a été conçue de telle façon que certaines fonctionnalités avancées du CDMA, comme la détection Multi-Utilisateurs et les antennes adaptatives, peuvent être déployées par l'opérateur afin d'accroître la capacité et/ou la couverture de son réseau.
- Le WCDMA a été conçu afin qu'il puisse être déployé conjointement avec le GSM. Les *handovers* entre le GSM et le WCDMA sont possibles et l'introduction du WCDMA pourra ainsi se faire progressivement.

5.1.5 Canaux de transport et canaux physiques

Au niveau du réseau d'accès UTRA, les données générées par les couches hautes sont transmises à l'interface air par des canaux de transport (*transport channel*) qui s'appuient sur différents canaux physiques (*physical channel*). La couche physique doit pouvoir supporter

différents débits afin d'offrir des services de bande passante à la demande et d'être à même de multiplexer plusieurs services sur une même connexion. Dans cette section nous présentons la correspondance qui existe entre les canaux de transport et les canaux physiques.
Chaque canal de transport comporte un indicateur TFI(*Transport Format Indicator*) à chaque instant où des données sont sensées arriver en provenance des couches hautes et à destination du canal du transport. La couche physique combine les informations des indicateurs TFI des différents canaux de transport avec un indicateur TFCI(*Transport Format Combination Indicator*). Ce dernier est transmis par le canal physique de contrôle (*physical control channel*) afin d'informer le récepteur des canaux de transport qui sont actifs au niveau de la trame courante. Il existe cependant une exception lors de l'utilisation du BTFD (*Blind Transport Format Detection*). Dans ce cas, l'indicateur TFCI n'est pas traité au niveau des canaux dédiés au sens descendant. Sinon, dans le cas d'un fonctionnement normal, l'indicateur TFCI est décodé par le récepteur et les indicateurs TFI correspondants à chaque canal de transport actif de la connexion transmis aux couches hautes. Dans la Figure 5.4, deux canaux de transport s'appuient sur un seul canal physique. A la réception, les blocs de transport sont restitués aux couches hautes ainsi que des indications relatives à d'éventuelles erreurs pour chaque bloc. Les canaux de transport peuvent avoir un nombre de blocs différent et ils ne sont pas nécessairement actifs à chaque instant.

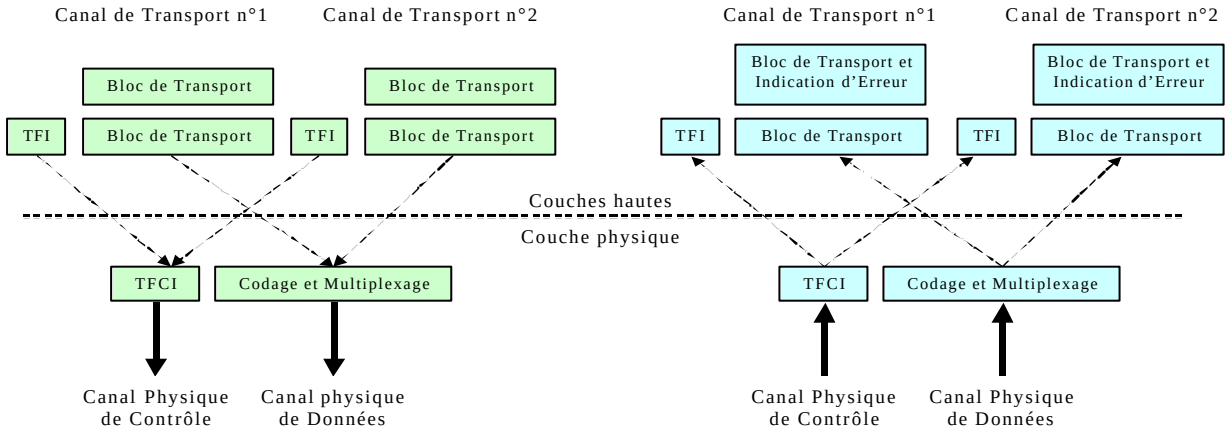


Figure 5.4 - Interface entre la couche physique et les couches hautes

Un canal physique de contrôle et un ou plusieurs canaux physiques de données (*physical data channel*) peuvent être codés et multiplexés par la même entité de traitement. Il en résulte ce que l'on appelle un canal de transport CCTrCh (*Coded Composite Transport Channel*). Il peut exister plus d'un canal CCTrCh pour une connexion donnée mais un seul canal physique de contrôle n'est transmis dans ce cas [48].
L'interface entre la couche physique et les couches hautes provient essentiellement de la méthode de spécification. Elle intervient peu au niveau de l'implémentation du terminal dans lequel toutes ces couches sont gérées par un seul équipement. En ce qui concerne le réseau, cette interface est plus importante. Elle se situe, en effet, entre la station de base et le RNC, c'est-à-dire au niveau de l'interface Iub[49],

5.1.5.1 Canal de transport dédié

Il existe un seul canal de transport dédié, noté DCH (*Dedicated channel*) dans les spécifications de la série 25 de l'UTRA. Il véhicule toutes les informations provenant des couches situées au-dessus de la couche physique et destinée à un utilisateur donné. Cela

inclue aussi bien les données correspondantes au service utilisé que des informations de contrôle issues des couches hautes. La nature des informations transmises n'est pas visible au niveau de la couche physique. Les données utilisateur et les données de contrôle sont ainsi traité exactement de la même manière. Naturellement, les paramètres de la couche physique établis par I'UTRAN peuvent varier pour les données utilisateur ou pour les informations de contrôle.
Le canal de trafic TRCH (*Traffic Channel*) et le canal associé de contrôle ACCH (*Associated Control Channel*), utilisés en GSM, n'existent pas dans la couche physique de I'UTRA. Le canal de transport dédié véhicule depuis le terminal à la fois les données de service, comme les trames de voix, et les informations de contrôle, comme les commandes de *handovers* et les remontées de mesures. En WCDMA, un canal séparé de transport n'est pas nécessaire du fait de la capacité du système à supporter des débits variables et un multiplexage des services.

5.1.5.2 Canaux de transport communs

Six types des canaux de transport communs ont été définis pour 1'UTRA. Ils présentent peu de différences par rapport à ceux définis pour les systèmes de deuxième génération. Notons que les canaux communs ne supportent pas le soft *handover* et que certains d'entre eux peuvent supporter le contrôle de puissance [50].

5.1.5.2.1 Broadcast Channel

Le canal BCH *(Broadcast Channe/)* ou canal de diffusion est un canal de transport utilisé pour transmettre des informations spécifiques au réseau d'accès ou à une cellule donnée. Les données les plus fréquemment nécessaires dans tout type de réseau mobile sont les codes d'accès aléatoires disponibles ou les slots d'accès disponibles au niveau de la cellule, ou encore, les méthodes de diversité en transmission utilisées avec d'autres canaux pour une cellule donnée. Le terminal ne peut pas s'inscrire dans une cellule s'il n'a pas la possibilité de décoder ce canal *broadcast.* C'est pour cela que la puissance de ce canal est généralement élevée afin qu'il puisse être capté par l'ensemble des utilisateurs de la cellule.

5.1.5.2.2 Forward Access Channel

Le canal FACH *(Forward Access Channel)* est un canal de transport du sens descendant qui véhicule des informations de contrôle aux terminaux localisés dans une cellule donnée. C'est par exemple le cas après qu'un message de demande d'accès aléatoire (*Random* Access) a été reçu, de la part d'un terminal, par la station de base. Il est également possible de transmettre des données par paquet sur le canal FACH. Il peut y avoir plus d'un canal FACH par cellule. Un des canaux FACH doit cependant avoir un débit relativement faible afin qu'il puisse être reçu par l'ensemble des utilisateurs de la cellule. Lorsqu'il y a plus d'un canal FACH par cellule, les canaux FACH supplémentaires peuvent avoir un débit plus élevé. Le canal FACH n'utilise pas le contrôle de puissance et les messages transmis doivent comprendre des informations d'identification afin d'assurer leur correcte réception.

5.1.5.2.3 Paging Channel

Le canal PCH *(Paging Channel)* est également un canal de transport du sens descendant assurant le transport des informations nécessaires à la procédure de *paging* lorsque le réseau souhaite initier une communication avec un terminal. Le plus simple exemple est le cas d'un appel voix vers un terminal le réseau transmet alors un message de *paging* grâce au canal PCH sur toutes les cellules de la zone de localisation où est sensé se trouver le terminal que le réseau d'accès souhaite joindre. Ce message de *paging* est alors transmis sur une ou plusieurs cellules, selon la configuration du système et celle des zones de localisation. Les terminaux doivent bien évidemment pouvoir recevoir l'information de *paging* dans toute la

cellule. Notons que la configuration du canal de *paging* affecte directement la consommation d'énergie des terminaux en mode veille. Moins le récepteur du terminal a à écouter le canal PCH pour savoir si un message de *paging* lui est destiné, plus la batterie du terminal aura une durée de vie importante en mode veille.

5.1.5.2.4 Random Access Channel

Le canal RACH *(Random Access Channel)* est un canal de transport du sens montant qui est utilisé pour transporter des informations de contrôle provenant du terminal, telles que les demandes d'établissement de connexion. Il peut être utilisé également pour transmettre un faible quantité de données par paquet du terminal vers le réseau. Pour un fonctionnement correct, le canal RACH doit bien évidemment être reçu par la station de base quelle que soit la localisation du terminal dans la cellule, cela implique que le débit utilisé sur ce canal doit être suffisamment faible, du moins pour les premières procédures d'accès et de contrôle.

5.1.5.2.5 Common Packet Channel

Le canal CPCH *(Common Packet Channel)* est une extension du canal RACH qui permet de transmettre des données utilisateur par paquet dans le sens montant. Le canal correspondant sur le sens descendant est le canal FACH. Au niveau de la couche physique, la principale différence qui réside entre les canaux CPCH et RACH est que le premier utilise le contrôle de puissance, un mécanisme de détection de collision ainsi qu'une procédure de gestion d'état CPCH. La transmission sur le canal CPCH peut être maintenue sur un grand nombre de trames alors que celle s'appuyant sur le canal RACH ne peut excéder une ou deux trames.

5.1.5.2.6 Downlink Shared Channel

Le canal DSCH (*Downlink Shared Channel*) est un canal de transport du sens descendant permettant de transporter des informations utilisateur ou des informations de contrôle dédiées. Il peut cependant être partagé par plusieurs utilisateurs. A certains égards, ce canal est similaire au canal FACH, mais le canal DSCH utilise le contrôle de puissance aussi bien qu'un débit qui peut varier d'une trame à l'autre. Le canal DSCH peut ne pas être reçu dans toute la cellule et peut utiliser les différentes méthodes de diversité de transmission utilisées par le canal associé DCH. Le canal DSCH est toujours associé à un canal DCH du sens descendant.

5.1.5.2.7 Canaux de transports nécessaires

Les canaux RACH, FACH et PCH sont les trois canaux communs de transport nécessaires au bon fonctionnement du système. L'utilisation des canaux DSCH et CPCH reste, quant à elle, optionnelle.

5.1.5.3 Correspondance des canaux de transport et des canaux physiques

Les différents canaux de transport que nous venons d'aborder s'appuient sur différents canaux physiques. La correspondance entre ces canaux de transport et canaux physiques est donnée dans la Figure 5.5.

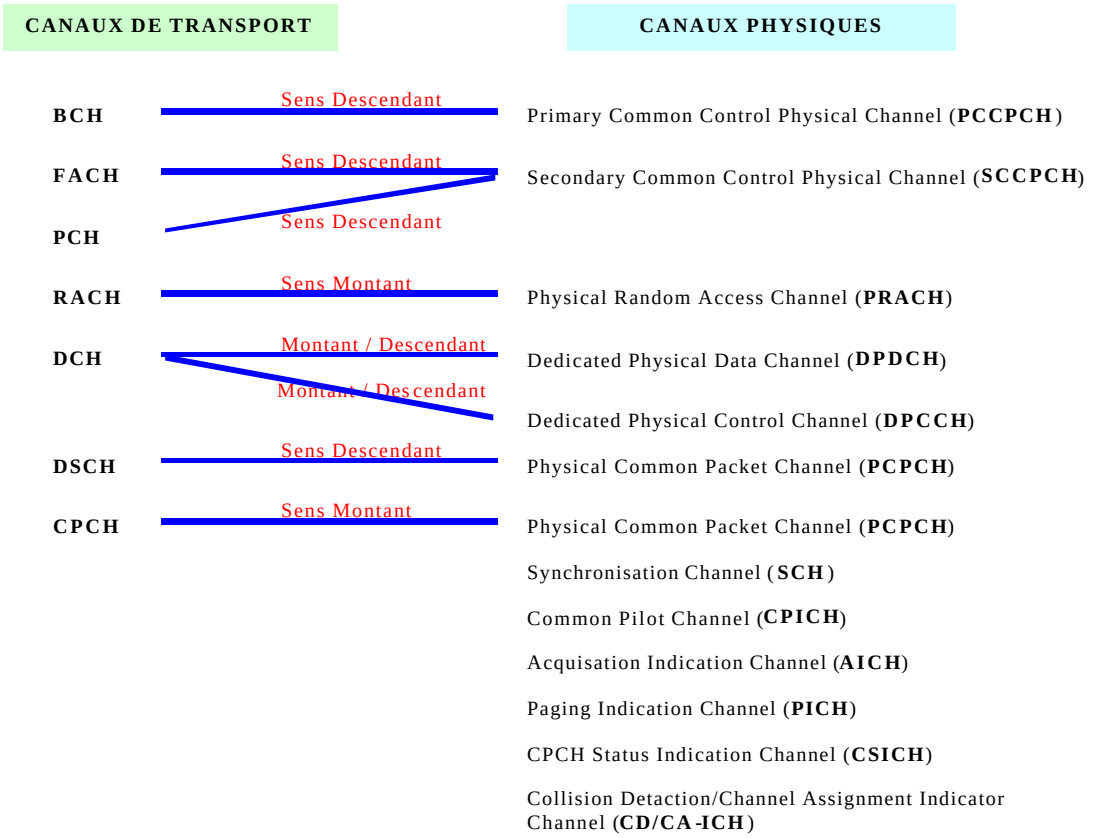


Figure 5.5 - Correspondance entre canaux de transport et canaux physiques

En plus des canaux de transport introduits plus haut, il existe des canaux physiques qui ne véhiculent que des informations propres aux procédures de la couche physique. Il s'agit des canaux SCH (*Synchronisation Channel*), CPICH (*Common Pitot Channel*) et AICH (*Acquisition Indication Channel*) qui ne sont pas visibles du point de vue des couches hautes mais qui sont cependant nécessaires au système et doivent être transmis par chaque station de base. Les autres canaux, CSICH (*CPCH Status Indication Channel*) et CD/CA-ICH (*Collision Detection/Channel Assignement Indication Channel*) ne sont nécessaires que si le canal CPCH est utilisé.

Notons que le canal de transport DCH s'appuie sur deux canaux physiques. Le canal physique DPDCH (*Dedicated Physical Data Channel*) transporte les informations des couches hautes ainsi que les données utilisateur alors que le canal physique DPCCH (*Dedicated Physical Control Channel*) véhicule les informations de contrôle nécessaire à la couche physique. Ces deux canaux dédiés sont essentiels au support de débits variables au niveau de la couche physique. Le débit du canal DPCCH est constant alors que celui du canal DPDCH peut varier d'une trame à l'autre.

5.1.6 Etalement et dés-étalement

La Figure 5.6, décrit le fonctionnement de base de l'étalement et du dés-étalement d'un système DS-CDMA.

Nous avons pris pour exemple, un signal initial BPSK (*Binary Phase Shift Keyng*) de fréquence R. ce signal est donc composé d'une séquence de bits pouvant prendre les deux valeurs suivantes « +1 » et « -1 ». La méthode d'étalement consiste, dans cet exemple, à multiplier chaque bit du signal initial par une séquence de huit bits, chacun de ces huit bits étant appelé chip, Nous voyons dans la Figure 5.6, que le résultat de ce produit est un

nouveau signal de fréquence 8xR. Dans ce cas nous avons utilisé un facteur d'étalement de 8. On remarque que le signal final a l'apparence d'un signal aléatoire tout comme le code d'étalement utilisé. Ce signal large bande sera ensuite transmis sur l'interface air.
En ce qui concerne la procédure inverse, le dés-étalement, nous multiplions, bit par bit, le signal étalé par la même séquence de codes que nous avons utilisée précédemment pour l'étalement. Comme le montre la Figure 5.6, on retrouve exactement le signal initial et cette opération n'introduit aucun déphasage entre le signal initial et le signal final.
La multiplication de la fréquence du signal par facteur 8 engendre un étalement similaire du spectre occupé par le signal résultant. Le dés-étalement permet de restaurer la bande passante initiale, proportionnelle à la fréquence R du signal.

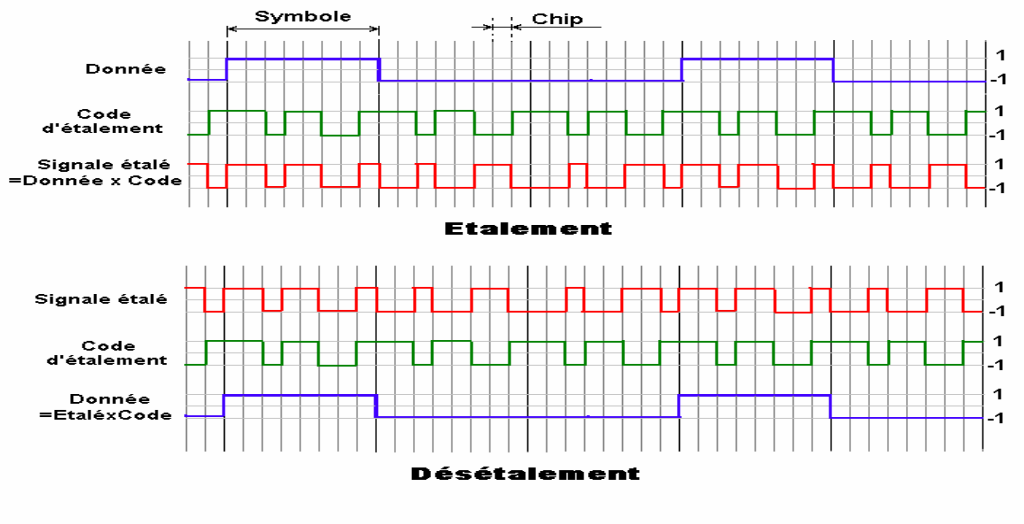


Figure 5.6 - Etalement et désétalement en DS-CDMA

Le fonctionnement de base d'un récepteur à corrélation en CDMA est présenté dans la Figure 5.7. La partie supérieure de la figure montre la réception du signal attendu, c'est-à-dire le signal qui correspond au code d'étalement utilisé. Comme précédemment dans la Figure 5.6, l'étalement parfaitement synchronisé réalisé par le code. Ensuite, le récepteur intègre ou plus précisément additionne le produit des bits du signal reçu avec ceux du code d'étalement.
La partie inférieure de la Figure 5.7 montre l'effet du désétalement quand il est appliqué au signal d'un autre utilisateur pour lequel l'étalement à été effectué avec une autre séquence d'étalement. Le résultat de la multiplication du mauvais signal par le code d'étalement puis son intégration par le récepteur donne une suite des valeurs proches de 0.
Comme nous pouvons le remarquer, l'amplitude du signal attendu est augmentée en moyenne par le facteur 8 par rapport aux autres signaux qui interfèrent. Cette méthode de détection par corrélation permet donc d'amplifier le signal attendu d'un coefficient égal au facteur d'étalement, ici 8. Cet effet est appelé gain de traitement (*processing gain*). C'est un des aspects fondamentaux de tous les systèmes CDMA et des autres systèmes à étalement de spectre. Ce gain de traitement donne une certaine robustesse aux systèmes CDMA face aux interférences qui sont générées par la réutilisation des même porteuses sur des stations de base proches les unes des autres.

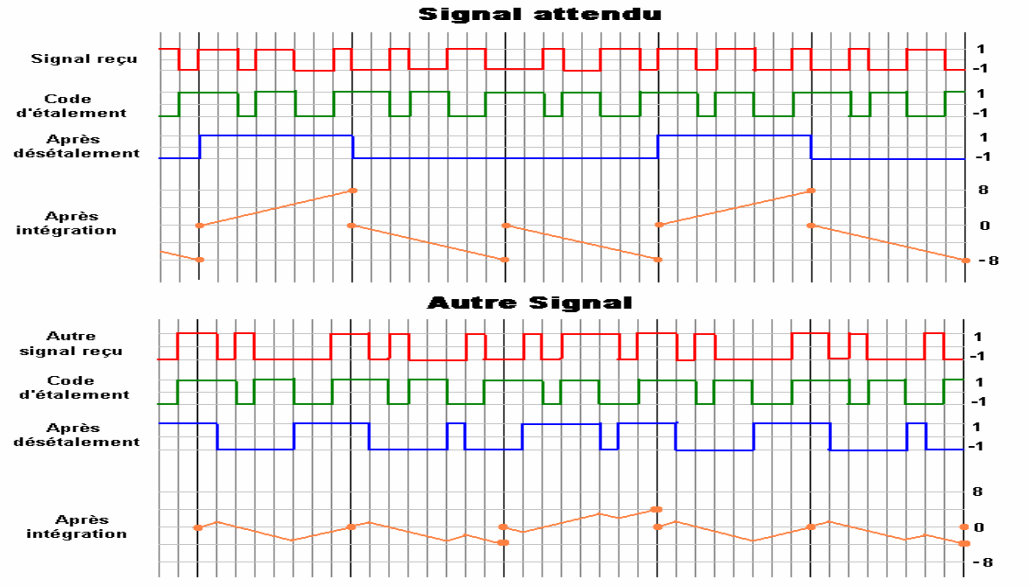


Figure 5.7 - Principe de corrélation du récepteur en CDMA

Prenons maintenant un exemple avec de véritables paramètres. Un service voix de débit 12.2 Kbit /s correspond à un gain de fonctionnement G=25dB,
En effet, $G=10\ log_{10}(3.86\ 10^6/12.2\ 10^3) = 25dB$
Après désétalement, la puissance du signal doit être typiquement supérieure à quelques dB par rapport à la puissance des interférences et du bruit. La densité de puissance nécessaire du signale après désétalement divisée par la densité de puissance des interférences est notée « E_b/N_0 », où E_b est l'énergie ou densité de puissance par bit et N_0 la densité de puissance des interférences et du bruit. Pour le service voix, le ratio E_b/N_0 est de l'ordre de 5dB et le ratio signal sur interférence (SIR, *Signal to Interference Ratio*) est par conséquent de l'ordre de –20 dB (5 dB moins le gain de traitement). En d'autres termes, même si la puissance du signal est inférieure à –20 dB que celles des interférences et du bruit, le récepteur pourra encore détecter et récupérer le signal. Le ratio signal sur interférence est aussi noté « C/I » (*Carrier-to-Interference ratio*). Du fait de l'étalement et du désétalement, ce rapport C/I peut être plus faible en WCDMA par rapport aux autres systèmes GSM et IS-95. Une communication vocale de bonne qualité nécessite, par exemple en GSM, un C/I de 9 à 15 dB.
En effet, un signal large bande peut se trouver sous le niveau du bruit thermique, sa détection est difficile sans en connaître sa séquence d'étalement. C'est pour cette raison que les premiers systèmes d'étalement de spectre ont été développés pour des applications militaires où la nature large bande du signal permet de le cacher sous le bruit thermique ambiant.
Remarque :
Pour une bande passante de canal donnée, nous aurons un gain de traitement plus important pour les faibles débits que pour les hauts débits. Par exemple, le gain de traitement pour un débit de 2Mb/s est inférieur à 2 (3.84/2=1.92), ce qui correspond à 2.8 dB. Dans ce cas, la robustesse des ondes WCDMA aux interférences est clairement compromise.

5.1.6.1 Code d'étalement

Les émissions issues d'une même source sont séparées grâce aux codes de canalisation. Il peut s'agir, par exemple, des différentes connexions provenant d'un même secteur dans le sens descendant et le canal physique dédié d'un terminal dans le sens montant. Les codes d'étalement sont basés sur la technique OVSF (*Orthogonal Variable Spreading Factor*), définie dans [51]. L'utilisation de ces codes OVSF permet de modifier le facteur

d'étalement SF(*Spreading Factor*) et de maintenir l'orthogonalité des différents codes d'étalement même si ces derniers sont de longueur différente. Ces codes sont choisis parmi ceux de « l'arbre des codes » qui est présenté dans la Figure 5.8.

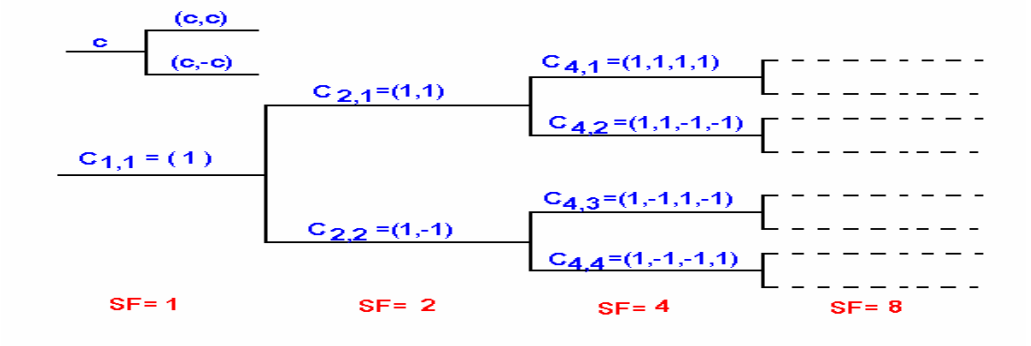


Figure 5.8 - Débit de l'arbre des codes d'étalement

Il existe cependant certaines restrictions quant à l'utilisation des codes d'étalement. Un canal physique ne peut utiliser un code donné que si aucun code de la branche qui en découle n'est utilisé par un autre canal physique, c'est-à-dire si aucun autre canal physique n'utilise un facteur d'étalement plus grand, généré à partir du code que le premier canal souhaite utiliser.

Les fonctionnalités et les caractéristiques du *scrambling* et des codes de canalisation sont résumées dans le tableau suivant :

Fonctionnalités	**Code de *channelisation***	**Code de *scrambling***
Utilisation	*Uplink* : séparation des canaux DPDCH et DPCCH provenant d'un même terminal. *Downlink* : séparation des connexions des différents utilisateurs d'une même cellule.	*Uplink* : séparation des terminaux. *Downlink* : séparation des cellules.
Longueur	Uplink : 4 à 256 chips (1 à 66.7µs) Downlink : 512 chips	Uplink : 10ms = 38400 chips Downlink: 10ms = 38400 chips
Nombre de codes	Nombre de codes égale au facteur d'étalement.	Uplink : plusieurs millions Downlink : 512
Famille de codes	Orthogonal Variable Spreading Factor (OVSF)	10ms : Gold code (long)
Etalement	Oui, Augmentation de la bande passante	Non, Aucune modification de la bande passante

Tableau 5.3 - Fonctionnalités des codes se scrambling et de channelisation

Pour les transmissions provenant d'une même source, qu'il s'agisse d'un terminal ou d'une station de base, il existe un arbre de codes de channelisation pour chaque code de scrambling. Cela signifie que différents terminaux et différentes stations de base peuvent utiliser leurs arbres de codes indépendamment, il n'y a nul besoin de cordonner les ressources en termes d'arbres de codes entre les différents terminaux ou stations de base.

5.1.6.2 Algorithme de génération des codes de canalisation

La méthode de génération des codes de canalisation $C_{ch,SF,canal}$ est défini par les équations suivantes[51] :

$$C_{ch,1,0} = 1$$

$$\begin{pmatrix} C_{ch,2,0} \\ C_{ch,2,1} \end{pmatrix} = \begin{pmatrix} C_{ch,1,0} & C_{ch,1,0} \\ C_{ch,1,0} & -C_{ch,1,0} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

$$\begin{pmatrix} C_{ch,2^{(n+1)},0} \\ C_{ch,2^{(n+1)},1} \\ C_{ch,2^{(n+1)},2} \\ C_{ch,2^{(n+1)},3} \\ \vdots \\ C_{ch,2^{(n+1)},2^{(n+1)}-1} \\ C_{ch,2^{(n+1)},2^{(n+1)}-2} \end{pmatrix} = \begin{pmatrix} C_{ch,2^n,0} & C_{ch,2^n,0} \\ C_{ch,2^n,0} & -C_{ch,2^n,0} \\ C_{ch,2^n,1} & C_{ch,2^n,1} \\ C_{ch,2^n,1} & -C_{ch,2^n,1} \\ \vdots & \vdots \\ C_{ch,2^n,2^n-1} & C_{ch,2^n,2^n-1} \\ C_{ch,2^n,2^n-1} & -C_{ch,2^n,2^n-1} \end{pmatrix}$$

5.1.7 Codes de scrambling

Les émissions issues de différentes sources sont séparées par les codes se scrambling. Au niveau de sens montant, il existe deux types de codes de scrambling : les codes courts et les codes longs. Les codes longs sont tronqués en une longueur correspondant à la durée d'une trame radio de 10ms, soit une longueur de 38 400chips pour un débit chip de 3,84 Mcps. Les codes courts ont, quant à eux, une longueur de 256 chips. Ces deux familles de codes de scrambling possèdent plusieurs millions de codes distincts, ce qui rend inutile la planification des codes au niveau du sens montant [51].
Les codes longs sont appelés « Gold codes ». La séquence de scrambling à valeurs complexes est formée par l'association de deux fois le même code, décalés dans le temps. La Figure suivante représente la configuration du générateur de séquence Scrambling.

Conditions initiales :
$X_n(0) = n_0$, $X_n(1) = n_1$, $X_n(2) = n_2$, …, $X_n(23) = n_{23}$, $X_n(24) = 1$
$Y(0) = Y(1) = Y(2) = \ldots = Y(24) = 1$
Relation récursive :
$X_n(i + 25) = X_n(i + 3) + X_n(i)$ modulo 2 , $i = 0,1,\ldots, 2^{25} - 27$

- $Y(i + 25) = Y(i + 3) + Y(i + 2) + Y(i + 1) + Y(i)$ modulo 2 , $i = 0,1,\ldots, 2^{25} - 27$

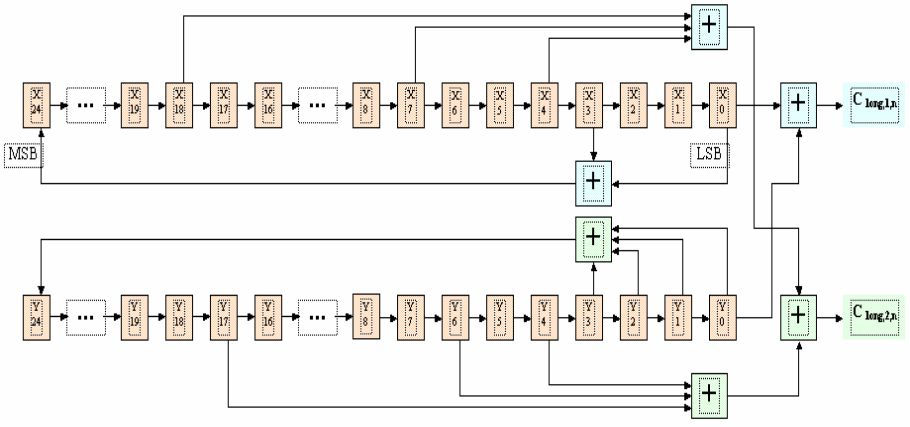


Figure 5.9 - Configuration du générateur de séquence Scrambling

D'où le code de scrambling complexe est généré par la relation suivante :

$$C_{long,n}(i) = c_{long,1,n}(i)\left(1 + j(-1)^{i} c_{long,2,n}(2\lfloor i/2 \rfloor)\right)$$

Avec i = 0,1,…,2^{25}-2 et $\lfloor\rfloor$ représente la partie entière.

5.1.8 Transmission de données utilisateur sur le Canal dédié uplink

La direction uplink utilise un multiplexage I/Q pour les données utilisateur et les informations de contrôle de la couche physique (Figure 5.10). Elles sont véhiculées par le canal DPCCH (Dedicated Physical Control Channel) avec un facteur d'étalement fixe de 256. Les informations de contrôle issues des couches hautes ainsi que les données utilisateur sont, quant à elles, transmises sur le canal DPDCH (Dedicated Physical Data Channel) qui utilise un facteur d'étalement variable, compris entre 4 et 256. La transmission dans le sens uplink peut être constituée d'un ou plusieurs canaux DPDCH avec un facteur d'étalement variable et d'un seul canal DPCCH au facteur d'étalement constant.

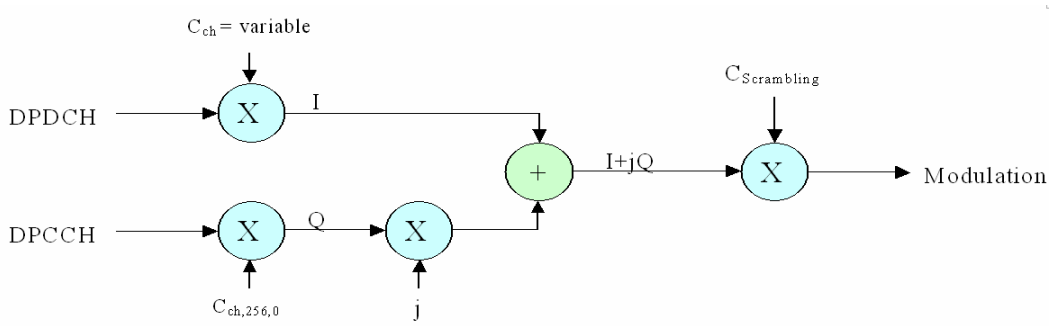


Figure 5.10 - Multiplexage I-Q avec code scrambling complexe

Le débit du canal DPDCH peut varier d'une trame à l'autre. Pour un service à débit variable, le débit du canal DPDCH est indiqué sur le canal DPCCH. Le canal DPCCH est transmis de façon continue et l'information du débit est transmise par l'indicateur TFCI. Par conséquence, si cet indicateur TFCI n'est pas décodé correctement, l'intégralité de la trame sera perdue. Puisque l'indicateur TFCI n'indique que le format de transport de la trame courante, la perte d'un indicateur TFCI d'une trame n'affectera pas les autres trames. La Figure 5.11 présente la structure du canal dédié uplink en détail.

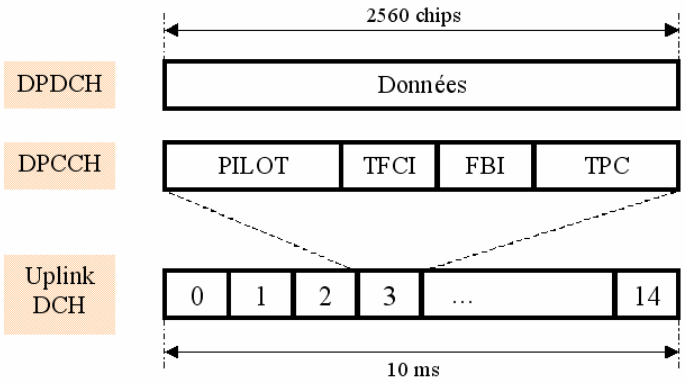


Figure 5.11 - Structure du canal dédié uplink

Le canal DPCCH utilise une structure à 15 slots sur une durée de trame radio de 10 ms. Cela représente une durée de slot de 2 560 chips, soit environ 666 µs. Chaque slot possède quatre

champs utilisés pour transmettre les bits pilot, l'indicateur TFCI(Transport Format Combination Indicator), les bits TPC (Transmission Power Control) et les bits FBI (Feedback Information). Les bits pilot sont utilisés pour l'estimation du canal au niveau du récepteur, les bits TPC, pour véhiculer les commandes du contrôle de puissance downlink. Les bits FBI sont, quant à eux, utilisés quand la diversité de transmission est mise en place sur le sens descendant. L'utilisation de ces bits est décrite dans la spécification technique de 3GPP [50].
Le débit maximum des données utilisateur pour un seul code provient directement du débit maximum du canal qui est de 960 kbps, sans codage du canal et avec un facteur d'étalement de 4. Avec le codage du canal, le débit maximum des données utilisateur pour un seul code est de l'ordre de 400 à 500 kbps.
Lorsque des débits utilisateur plus élevés sont nécessaires, des canaux parallèles doivent être utilisés. Il est possible d'utiliser jusqu'à 6 codes en parallèle ce qui permet d'atteindre un débit de transmission de 5 740 kbps, soit un débit utilisateur de 2 Mbps voire plus si le taux de codage est de ½. Ainsi, il est bien possible d'offrir un débit utilisateur de 2 Mbps, même après retransmission.
Les débits utilisateurs maximums sont donnés dans le Tableau 5.4 avec un taux de codage de ½.

Facteur d'étalement DPDCH	Débit du canal DPDCH (kbps)	Débit utilisateur maximum (kbps)
256	15	7.5
128	30	15
64	60	30
32	120	60
16	240	120
8	480	240
4	960	480
4 (avec 6 codes)	5740	2300

Tableau 5.4 - Débits DPDCH uplink

Le récepteur uplink situé au niveau de la station de base doit réaliser les tâches suivantes lorsqu'il reçoit les signaux d'un terminal
Le récepteur commence à recevoir la trame et désétale le DPCCH puis mémorise le DPDCH en fonction du débit maximum, correspondant au plus petit facteur d'étalement.
Pour certains slots, le récepteur doit:

- estimer le canal à partir des bits pilot du DPCCH
- estimer le ratio SIR à partir des bits pilot de chaque slot
- envoyer la commande TPC au terminal afin qu'il puisse contrôler sa puissance d'émission
- décoder les bits TPC de chaque slot et ajuster la puissance d'émission de la station de base

Tous les deux ou les quatre slots, décoder les bits FBI, s'ils sont présents, et ajuster les phases des antennes utilisées en diversité ou les phases et les amplitudes, selon la méthode de diversité utilisée. Pour chaque trame de 10 ms, décoder l'indicateur TFCI afin d'obtenir le débit et les paramètres permettant le décodage du canal DPDCH. Pour chaque intervalle TTI (Transmission Time Interval) qui correspond à la période d'entrelacement (soit 10, 20, 40 ou 80 ms), décoder les données du canal DPDCH. Les fonctions ci-dessus sont également à réaliser dans le sens descendant, à part les exceptions suivantes :

- Dans le sens descendant, le facteur d'étalement du canal dédié est constant, comme pour les canaux communs. La seule exception est le canal DSCH (Downlink Shared Channel) qui autorise un facteur d'étalement variable.

- Les bits FBI ne sont pas utilisés dans le sens descendant.
- Dans le sens descendant, un canal commun pilot est disponible en plus des bits pilot du canal DPCCH. Les bits de ce canal commun pilot peuvent être également utilisés pour l'estimation du canal.

La transmission dans le sens descendant peut utiliser deux antennes dans le cas de la diversité d'espace. Le récepteur effectue l'estimation du canal à partir des données issues des deux antennes. Par conséquent, il doit réunir les données dés-étalées en provenance des deux antennes.

5.1.9 Chaîne de codage et de multiplexage uplink

Dans le sens montant, les services sont multiplexés de façon dynamique de telle sorte que le flux de données soit continu sauf lorsque aucune donnée n'est à transmettre. Les symboles sur le DPDCH sont envoyés avec une puissance égale pour tous les services. Cela signifie en pratique que le codage du service et le multiplexage du canal doivent ajuster les débits symbole pour les différents services afin d'équilibrer en puissance les différents symboles. La fonction d'adaptation du débit dans la chaîne de multiplexage, présentée dans la Figure 5.12, est utilisée pour ces opérations d'équilibrage entre les services sur un seul canal DPDCH. Il n'existe pas d'emplacement prédéfini pour les différents services au niveau du canal DPDCH uplink. La trame est constituée en fonction de la sortie des opérations d'adaptation de débit et d'entrelacement. Le multiplexage uplink est constitué de onze étapes et est présenté dans la Figure 5.12.

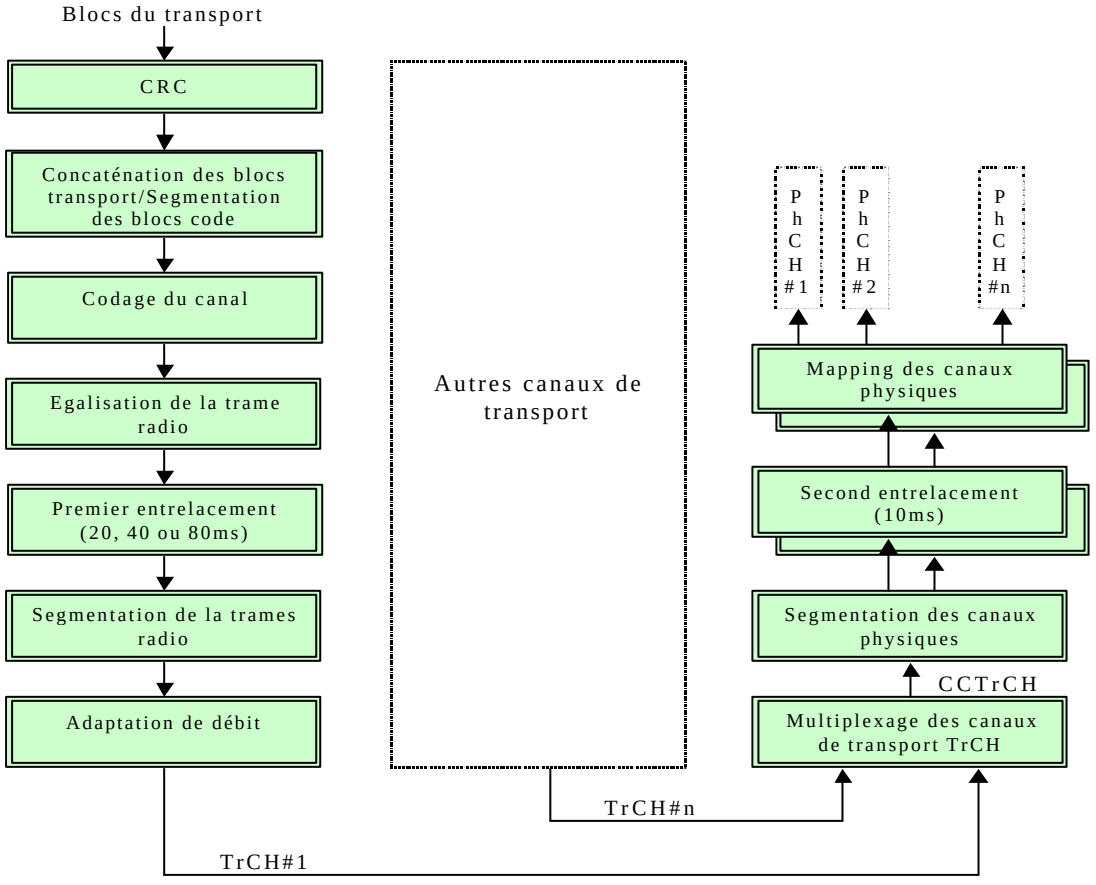


Figure 5.12 - Chaîne de codage et de multiplexage uplink

5.1.9.1 Opération CRC

Après la réception d'un bloc de transport provenant des couches hautes, la première opération est d'ajouter un code CRC. Le CRC (Cyclic Redundancy Check) est utilisé pour la détection d'erreurs réalisée au niveau du récepteur sur les blocs de transport. La longueur de ce code CRC est de 0, 8, 12, ou 24 bits [48]. Plus le code CRC est long, plus la probabilité qu'une erreur ne soit pas détectée est faible. La couche physique fournit aux couches hautes, toujours au niveau du récepteur, le bloc de transport ainsi qu'une information sur la présence ou non d'erreur détectée. Les polynômes générateurs qu'on peut utiliser sont les suivants :

$g_{CRC8}(D) = D^8 + D^7 + D^4 + D^3 + D + 1$

$g_{CRC12}(D) = D^{12} + D^{11} + D^3 + D^2 + D + 1$

$g_{CRC16}(D) = D^{16} + D^{12} + D^5 + 1$

$g_{CRC24}(D) = D^{24} + D^{23} + D^6 + D^5 + D + 1$

Pour utiliser un code polynomial, l'émetteur et le récepteur doivent se mettre d'accord sur le choix d'un polynôme générateur G(x). Pour calculer la somme de contrôle (checksum) d'un bloc de m bits (correspondant au polynôme M(x)), il faut que le bloc soit plus long que le polynôme générateur. Le principe consiste à coller, à la fin du bloc, des bits de contrôle de façon que le bloc de sortie (bloc et bits de contrôle) soit divisible par G(x). Quant le récepteur reçoit le bloc, il le divise par G(x). Si le reste obtenu n'est pas nul, c'est qu'il y a eu une erreur de transmission [52].

L'algorithme de calcul des bits de contrôle est le suivant :

Soit r le degré de G(x), Ajouter r zéros après le bit de poids faible du bloc. Il contient ainsi m + r bits, correspondant au polynôme $x^r M(x)$.

Effectuer la division modulo 2 du polynôme $x^r M(x)$ par G(x).

Soustraire modulo 2 le reste de la division (qui comprend au plus r bits) de la chaîne de bits correspondant au polynôme $x^r M(x)$. Le résultat de cette opération est le bloc de sortie.

5.1.9.2 Concaténation des blocs de transport et Segmentation des blocs code

Après l'ajout d'un code CRC, les blocs de transport sont soit concaténés les uns aux autres, soit segmentés en différant blocs. Cela dépend si le bloc de transport correspond à la taille du bloc de codage disponible défini par la méthode de codage du canal. La concaténation permet de minimiser la taille des overheads et dans certains cas d'améliorer l'efficacité du codage du canal. La segmentation permet d'éviter les blocs de taille trop importante qui compliqueraient sensiblement les opérations qui suivent. Si le bloc de transport assorti de son code CRC dépasse la taille maximum du bloc de codage, il sera divisé en plusieurs blocs de taille inférieure [53].

5.1.9.2.1 Concaténation des blocs de transport

$b_{im1}, b_{im2}, \ldots, b_{imBi}$ sont les bits d'entrés des blocs de transport à concaténer, avec i est le numéro de TrCH, m est le numéro de bloc de transport et Bi est le nombre des bits pour chaque bloc de transport. Le nombre de blocs de transport dans le TrCH i est noté par M_i. Les bits après concaténation sont notés $x_{i1}, x_{i2}, \ldots, x_{iXi}$, avec i est le numéro de TrCH et $X_i = M_i B_i$.

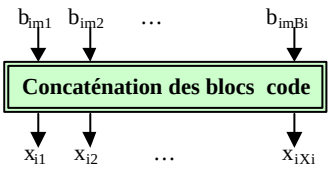


Figure 5.13 - Opération de concaténation des blocs code

Les bits de sortie sont définis par les relations suivantes [48] :

$x_{ik} = b_{i,1,k}$; $k = 1, 2, \ldots, B_i$

$x_{ik} = b_{i,2,(k-Bi)}$; $k = B_i+1, B_i+2, \ldots, 2B_i$
$x_{ik} = b_{i,3,(k-2Bi)}$; $k = 2B_i+1, 2B_i+2, \ldots, 3B_i$
…
$x_{ik} = b_{i,Mi,(k-(Mi-1)Bi)}$; $k = (M_i-1)B_i+1, (M_i-1)B_i+2, \ldots, M_iB_i$

5.1.9.2.2 Segmentation des blocs code

La segmentation de séquence des bits, fournie par l'opération de concaténation, est nécessaire si $X_i > Z$, avec Z est la taille maximale de bloc code autorisée par le coder en question. Après segmentation, tous les blocs code ont la même taille. Le nombre de blocs code de TrCH i est noté C_i. Si le nombre des bits X_i, à l'entrée de segmentation n'est pas multiple de C_i, des bits de bourrage sont ajoutés au début du premier bloc code. Si turbo codage est sélectionné et $X_i < 40$, des bits de bourrage sont ajoutés au début de ce bloc code. Les bits de bourrage sont transmis avec la valeur zéro.
Les tailles maximales de blocs code sont les suivantes :

- Codage convolutif : Z = 504 ;
- Turbo codage: Z = 5114 ;
- Sans codage du canal : Z = non limité ;

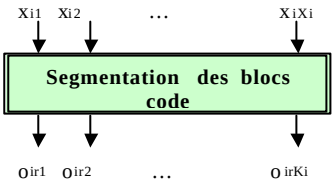


Figure 5.14 - Opération de segmentation des blocs de code

5.1.9.3 Codage du canal pour la transmission de données

Au niveau de l'UTRA, deux méthodes de codage de canal ont été définies. Les codages convolutifs demi-débit et tiers-débit sont généralement utilisés pour les débits utilisateur relativement faibles, équivalents aux débits fournis par les réseaux de deuxième génération actuelle. Pour les débits plus élevés, un codage « Turbo » peut être appliqué. Ce dernier est d'autant plus efficace que les blocs traités sont de taille importante [48].

Type du TrCH	Type de codage	Taux de codage
BCH	Codage convolutif	½
PCH		
RACH		
CPCH, DCH, DSCH, FACH		1/3, ½
	Turbo codage	1/3
	Sans codage	

Tableau 5.5 - Choix de type de codage et taux de codage

La Figure 5.15, montre le bloc de l'opération codage du canal. On note o_{ir1}, o_{ir2}, …, o_{irKi} sont les bits d'entrés, avec i est le numéro de TrCH, r est le numéro de bloc de code et Ki est le nombre des bits pour chaque bloc de code. Le nombre de blocs de code dans le TrCH i est noté par Ci.

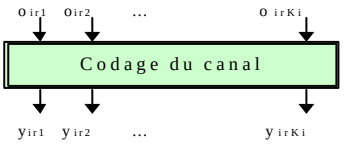


Figure 5.15 - Opération codage du canal

Après codage, les bits sont notés par y_{ir1}, y_{ir2}, …, y_{irYi} , avec Yi est le nombre des bits codés. La relation entre o_{irk} et y_{irk} et entre K_i et Y_i dépend du type de codage du canal, tel que codage convolutif, turbo codage, ou sans codage.
La valeur de Y_i en fonction de type de codage est la suivante :

- Codage convolutif avec taux ½ : $Y_i = 2K_i + 16$; taux 1/3 : $Y_i = 3K_i + 24$
- Turbo codage avec un taux 1/3 : $Y_i = 3K_i + 12$
- Sans codage : $Y_i = K_i$

5.1.9.3.1 Codage convolutif

Codage convolutif avec longueur de contraint égale à 9 et taux de codage égale à ½ et 1/3. La configuration du codeur convolutif est représentée par la Figure suivante [48].

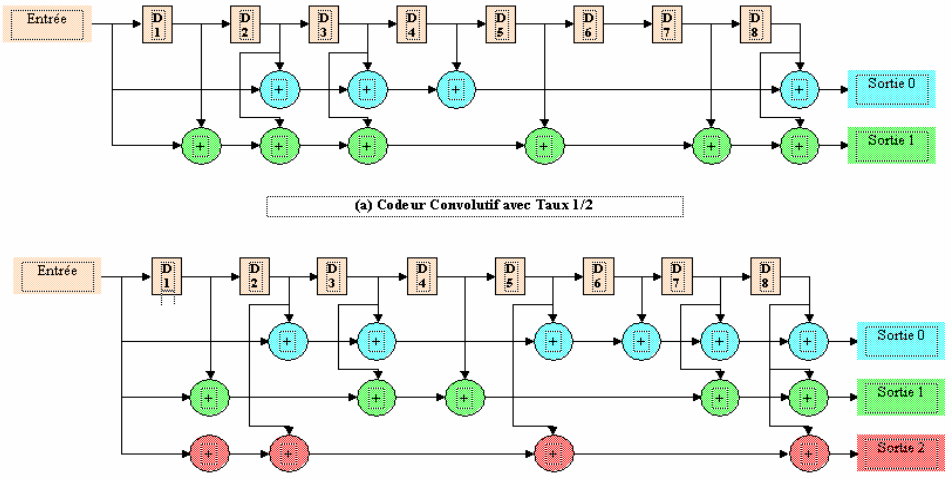


Figure 5.16 - Codeurs convolutifs avec taux de codage de ½ et 1/3

La sortie de codeur de taux ½ prend dans l'ordre les valeurs Sortie0, Sortie1, Sortie0, Sortie1, …,Sortie1. La sortie de codeur de taux 1/3 prend dans l'ordre les valeurs Sortie0, Sortie1, Sortie2, Sortie0, Sortie1, Sortie2,, …,Sortie2

5.1.9.3.2 Turbo coder

Le schéma de turbo coder est un PCCC (Parallel Concatenated Convolutional Code) avec deux encodeurs à huit états et un entrelacer turbo code interne. Le taux de codage de turbo coder est 1/3. La structure de ce turbo coder est illustrée par la figure suivante :

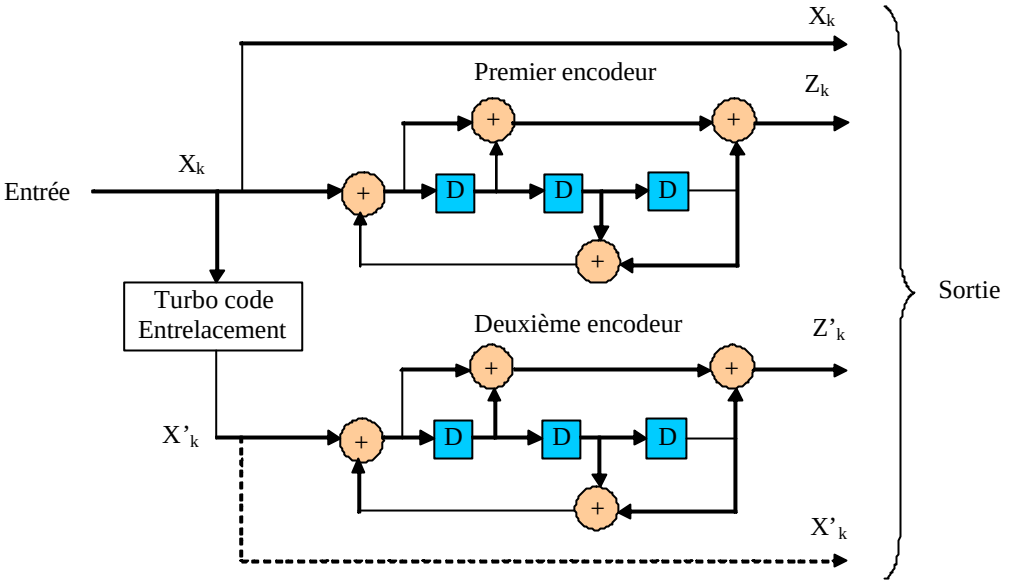


Figure 5.17 - Structure du turbo codeur pour le W-CDMA

La fonction de transfert de PCCC est :

$$G(D)=\left[1,\frac{g_1(D)}{g_0(D)}\right]$$

Avec $g_0(D)=1+D^2+D^3$ Et $g_1(D)=1+D+D^3$

Les registres de décalage des encodeurs huit états doivent être initialisés avec des zéros au débit de l'opération de l'encodage.
La sortie de l'encodeur est de la forme :

$x_1, z_1, z'_1, x_2, z_2, z'_2, ..., x_k, z_k, z'_k,$

Avec $x_1, x_2, x_3, \ldots, x_k$ sont les bits d'entrée du turbo coder, c'est à dire à l'entrée de premier encodeur à huit états et l'entrelacer turbo code interne, et k est le nombre des bits à l'entrée (taille du bloque de données à l'entrée de turbo coder). $z_1, z_2, \ldots, z_k$ et $z'_1, z'_2, \ldots, z'_k$ sont respectivement, les bits de sortie de premier et second encodeur à huit états.
Les bits de sortie de l'entrelacer turbo code interne sont représentés par $x'_1, x'_2, \ldots, x'_k$. Ces bits sont appliqués à l'entrée de deuxième encodeur à huit états.
L'entrelacer turbo code interne est constitué d'une matrice rectangulaire d'entrer avec des permutations intra-ligne et inter-ligne. Les bits d'entrée de l'entrelacer turbo code sont de la forme $x_1, x_2, x_3, \ldots, x_k$, avec k est la taille du bloque de données à encoder ($40 \leq k \leq 5114$)

5.1.9.4 Premier entrelacement

Le premier entrelacement ou entrelacement inter-trame est utilisé lorsque la transmission peut supporter un entrelacement de plus de 10 ms. La longueur de cet entrelacement peut être alors de 20, 40, 60 ou 80 ms. La période d'entrelacement est directement liée à l'intervalle TTI qui indique la périodicité de transfert des données en provenance des couches hautes. Les positions de départ des intervalles TTI pour différents canaux de transport multiplexés sont alignées dans le temps. La relation en temps des différents intervalles TTI est présentée dans la Figure 5.18.

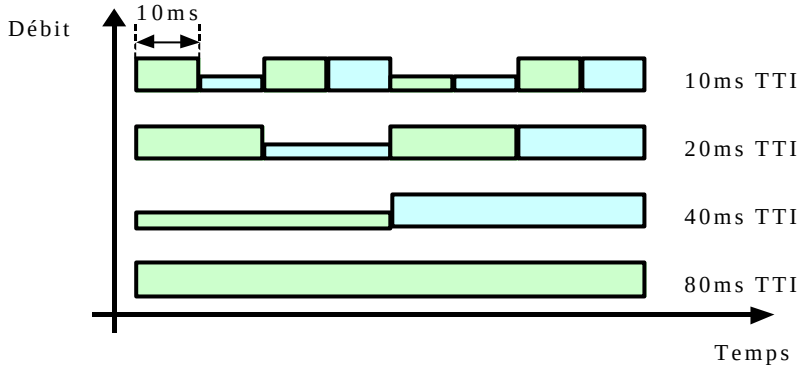


Figure 5.18 - Relation entre différents TTI d'une même connexion

5.1.9.5 Segmentation de la trame radio

Si la fonction d'entrelacement est utilisée, la segmentation de trame répartira les données sur 2, 4 ou 8 trames consécutives en fonction de la longueur d'entrelacement TTI.

5.1.9.6 Adaptation du débit

L'adaptation du débit permet d'adapter le nombre de bits à transmettre au nombre de bits disponibles sur une trame. La fonction d'adaptation du débit doit prendre en compte également le nombre de bits provenant des autres canaux de transport qui sont actifs dans la même trame. L'adaptation du débit sur le sens montant est une opération dynamique qui varie trame après trame, l'adaptation dynamique du débit permet aussi d'ajuster les paramètres des autres canaux afin que tous les symboles de la trame radio soient utilisés. Par exemple, si deux canaux de transport n'ont plus de données à transmettre, la fonction d'adaptation du débit va augmenter le débit symbole pour les autres services afin que tous les symboles soient utilisés, en supposant que le facteur d'étalement reste le même.

5.1.9.7 Multiplexage des canaux de transport

Les différents canaux de transport sont multiplexés. Il s'agit ici d'une simple opération de multiplexage en série effectuée trame après trame. Chaque canal de transport fournit ses données dans des blocs d'une durée de 10 ms.

5.1.9.8 Segmentation des canaux physiques

Dans le cas où plusieurs canaux physiques (plusieurs codes d'étalement) seraient utilisés, une opération de segmentation est nécessaire. Cette opération consiste simplement à répartir uniformément les données sur les codes d'étalement disponibles.

5.1.9.9 Second entrelacement

La seconde opération d'entrelacement réalise un entrelacement de la trame radio par période de 10 ms, elle est appelée « entrelacement intra-trame ». Ce second entrelacement est effectué séparément sur chaque canal physique, dans le cas où plus d'un canal soit utilisé. A la sortie de cette opération, les bits sont directement transmis sur les canaux physiques. Le nombre de bits émis par un canal physique est exactement égale au nombre que le facteur d'étalement de la trame peut transmettre. Si aucun bit n'est à transmettre, le canal physique correspondant n'est pas transmis.

5.2 Simulation

5.2.1 Application cible « Turbo Code r »

Le Turbo Code est un nouveau type de code correcteur d'erreurs introduit avec un algorithme de décodage pratique en 1993 par une équipe de chercheurs de France Télécoms. De nos jours, c'est la technique de codage correcteur d'erreurs la plus puissante. Le principe de base est d'utiliser deux codeurs élémentaires de faible complexité, séparés par une fonction d'entrelacement introduisant de la diversité. La séquence de données est codée deux fois, mais dans des ordres différents, et produit deux séquences de redondance. Deux décodeurs élémentaires sont implémentés en réception, l'un travaillant sur les données dans l'ordre naturel, l'autre sur les données permutées. Chaque décodeur élémentaire produit une information de fiabilité, appelée information extrinsèque, qui est utilisée par l'autre décodeur pour améliorer ses performances. Ce processus itératif est le turbo décodage. La figure suivante représente la structure standard d'un turbo codeur [54].

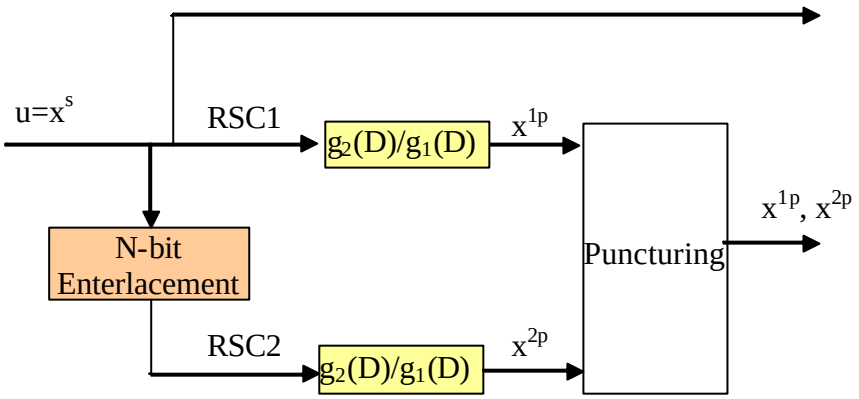


Figure 5.19 - Diagramme d'un turbo encodeur avec deux encodeurs récursifs (RSC) identiques

Comme le montre la Figure 5.19, le turbo encoder est composé de deux encodeurs convolutifs ½, séparés par un entrelacer de N-bit (*N-bit interleaver* ou *permuter*), l'ensemble avec un système de ponctuation optionnel (*puncturing mechanism*). Sans le *poncturing* le taux de codage du turbo coder est de 1/3, pour N bits de données on obtient 3N bits de données codées.

Implémenter un Turbo Coder apporte généralement un gain de 2 à 4 dB par rapport aux solutions de codage correcteur d'erreurs classiques. En fonction des contraintes du système, ce gain pourra être dépensé pour augmenter le débit utilisateur, améliorer la qualité de service, économiser de la puissance, réduire les tailles d'antennes, optimiser la distance de transmission ou la surface des cellules...

5.2.2 Exploration multiobjectifs avec la méthode d'agrégation

Dans le cadre de cette exploration architecturale multiobjectifs, nous avons utilisé la méthode d'agrégation avec une fonction d'adaptation de type : $f = \sum_i a_i f_i$

Pour l'implémentation de l'algorithme génétique nous avons utilisé la bibliothèque GAlib [98].

Dans ce cas d'étude, le turbo code [54] est utilisé comme application soft et le processeur SuperScalar comme architecture Hard à paramétrer. Le codage des paramètres de ce processeur par l'algorithme génétique est représenté par le tableau suivant :

Module	Nom de paramètre	Valeur possible
Cache d'instruction (Instruction Cache)	Nombre d'ensemble (nsets)	64, 128, **256**, 512
	Taille de bloc (bsize)	8, 16, **32**, 64 (Octet)
	Associativité (assoc)	**1**, 2, 4, 8
Cache de données (Data Cache)	Nombre d'ensemble (nsets)	64, 128, **256**, 512
	Taille de bloc (bsize)	8, 16, **32**, 64 (Octet)
	Associativité (assoc)	**1**, 2, 4, 8
Pipeline	Fetch:ifqsize, Issue:width, decode:width	1, 2, **4**, 8
ALU Entier	Nombre des unités arithmétique et logique entier (res:ialu)	1, 2, 3, **4**, 5, 6, 7, 8
MULT Entier	Nombre des unités multiplication/division entier (res:imult)	**1**, 2, 3, 4, 5, 6, 7, 8

Tableau 5.6 – Codage des paramètres du processeur SuperScalar

Notre but est de minimiser le temps d'exécution et la surface en silicium nécessaire à la l'intégration de l'architecture. La durée d'exécution est représentée par un seul objectif (f1) et La surface en silicium est représentée par neuf fonctions objectifs (f2,f3,…,f10). Le tableau suivant résume la répartition des différentes fonctions objectifs :

Objectifs à minimiser	Fonctions objectifs	% de la fonction d'adaptation globale	
Temps d'exécution	f1 = durée/DuréeMax	50%	50%
Surface en silicium	f2 = nbr_dl1_bsize/64	5,55%	50%
	f3 = nbr_il1_bsize/64	5,55%	
	f4 = nbr_dl1_nsets/512	5,55%	
	f5 = nbr_il1_nsets/512	5,55%	
	f6 = nbr_dl1_assoc/8	5,55%	
	f7 = nbr_il1_assoc/8	5,55%	
	f8 = nbr_res_ialu/8	5,55%	
	f9 = nbr_res_imult/8	5,55%	
	f10 = nbr_fetch_ifqsize/8	5,55%	

Tableau 5.7 – Composition de la fonction d'adaptation globale

La fonction d'adaptation globale $f = \sum_i a_i f_i$ doit être incluse dans l'intervalle [0, 1], ce qui implique une normalisation des toutes les fonctions objectifs. Pour la normalisation de f1, nous avons besoin de déterminer la durée d'exécution maximale (DuréeMax implique une exécution avec le minimum de ressources dans le processeur).

5.2.2.1 Représentation chromosomique : codage des individus

Le codage des différents paramètres du microprocesseur est assuré par un chromosome binaire de longueur 20 bits selon la figure suivante :

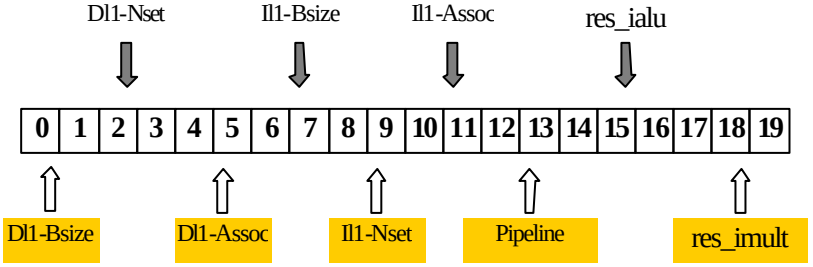


Figure 5.20 - Codage de chromosome pour le processeur SuperScalar

Les paramètres de réglage de l'AG sont enregistrés automatiquement dans un fichier au moment du lancement de la simulation (Tableau 5.8).

Nom du paramètre	Valeur
Taille de la population	20
Nombre des variables	9
Type de Codage chromosome	Binaire de taille 20 bits
Probabilité de croisement	0,70
Probabilité de mutation	0,10
Type de la sélection	Sélection proportionnelle
Nombre des générations	150

Tableau 5.8 – Paramètres de réglage de l'AG

La durée total de cette simulation est d'une heure, 37 minute et 53 secondes.
Cette simulation a été lancée sur la machine uei8 avec une fréquence de processeur 1,4 Ghz et une mémoire centrale SDRAM de taille 768 Mo.
Nous avons lancé deux simulations : (1) avec l'opérateur génétique de croisement en un seul point et (2) avec croisement en deux points.

5.2.2.2 Croisement des individus en deux sites (two points crossover)

Les valeurs suivantes représentent les différentes fonctions objectifs (f1,f2,…,f10) correspondant à l'optimale de la fonction fitness globale ($f = \sum_i a_i f_i$):

Obj.	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10
Valeur	0,30	0,50	0,12	1,00	0,50	0,25	0,12	0,37	0,62	0,50
		0,22								

Tableau 5.9 – Valeurs des fonctions objectifs pour le meilleur individu à la dernière itération de l'AG.

$$f = \sum_i a_i f_i = 0{,}50 * 0{,}30 + 0.50 * 0{,}22 = 0{,}37$$

L'individu (Tableau 5.9) représente la meilleure solution avec une valeur minimale de la fonction d'adaptation globale (f = 0,37). Cette valeur minimale de la fonction d'adaptation globale a été obtenue, à partir de la génération numéro 84.
La Figure 5.21, trace la valeur du meilleur individu à chaque génération. A partir de la génération numéro 84, l'algorithme converge vers la solution minimale en terme de valeur de la fonction objectif globale. Sur la même figure nous avons tracé la variation de la valeur moyenne de la fonction objectif globale, on constate que cette valeur de la moyenne varie beaucoup à chaque génération ce qui implique une grande diversité des solutions au sein de la population à chaque génération de l'AG.

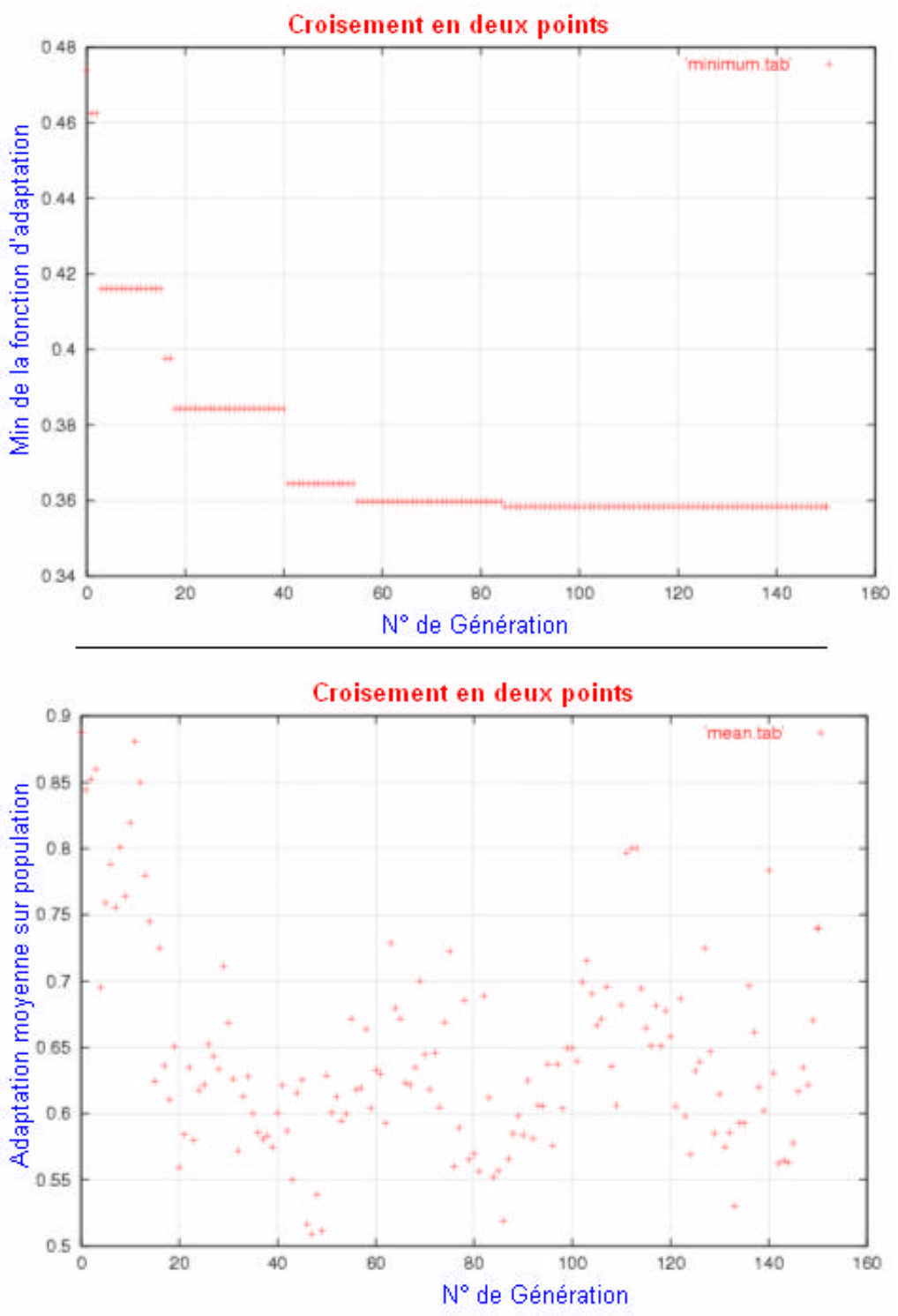


Figure 5.21 - Fonction d'adaptation min et moyenne en fonction du numéro de génération avec deux points de croisement

Le Tableau 5.10 résume l'évolution de la fonction d'adaptation minimale et les valeurs des paramètres à chaque variation de la fitness globale.

Num. génération	1	2	15	17	40	55	84
Fitness	0.462538	0.41609	0.397723	0.384353	0.364536	0.359756	0.358420
Cache_dl1_bsize	32	32	32	8	8	8	8
Cache_dl1_nsets	128	128	128	64	64	64	64
Cache_dl1_assoc	2	1	1	1	1	1	2
Cache_il1_bsize	32	16	16	16	16	16	16
Cache_il1_nsets	512	128	64	128	128	128	128
Cache_il1_assoc	2	2	2	4	1	1	1
Fetch_ifqsize	8	4	4	2	4	4	4
Decode_width							
Issue_width							
Res_ialu	3	5	4	5	4	3	2
Res_imult	5	2	1	1	1	1	5
Sim_cycle	9.399.306	9.419.929	8.634.655	9.396.418	8.947.724	8.968.167	9.153.033

Tableau 5.10 - Convergence vers une solution optimale en fonction du numéro de la génération

D'après le tableau 5.10 et tableau 5.9, la meilleure configuration du processeur est détectée à la génération numéro 84. L'architecture du processeur relative à cette configuration est la suivante :

- Cache de données avec une taille de bloc (dl1_bsize) égale à 8 octets, un nombre des lignes (dl1_nset) égale à 64 lignes et avec une associativité (dl1_assoc) égale à 2. soit un cache de données de taille globale égale à 1024 Octets (1Ko).
- Cache d'instruction avec une taille de bloc (il1_bsize) égale à 16 octets, un nombre des lignes (il1_nset) égale à 128 lignes et avec une associativité (il1_assoc) directe. soit un cache d'instructions de taille globale égale à 2048 Octets (2Ko).
- Un pipeline de 4 unités de recherche d'instruction (Fetch_ifqsize), 4 unités de décodage (Decode_width) et 4 unité d'exécution (Issue_width).
- Sept unités fonctionnelles dont deux unités d'addition/soustraction en nombre entier et 5 unités de division/ multiplication en nombre entier.

5.2.2.3 Croisement des individus en un seul site (One point crossover)

La figure suivante montre le résultat de l'algorithme génétique multiobjectif avec une fonction d'adaptation globale (multiobjectif avec la méthode d'agrégation). Dans ce cas nous avons modifié que le mode de croisement des individus, croisement en un seul point au lieu de deux points dans le cas précédant.

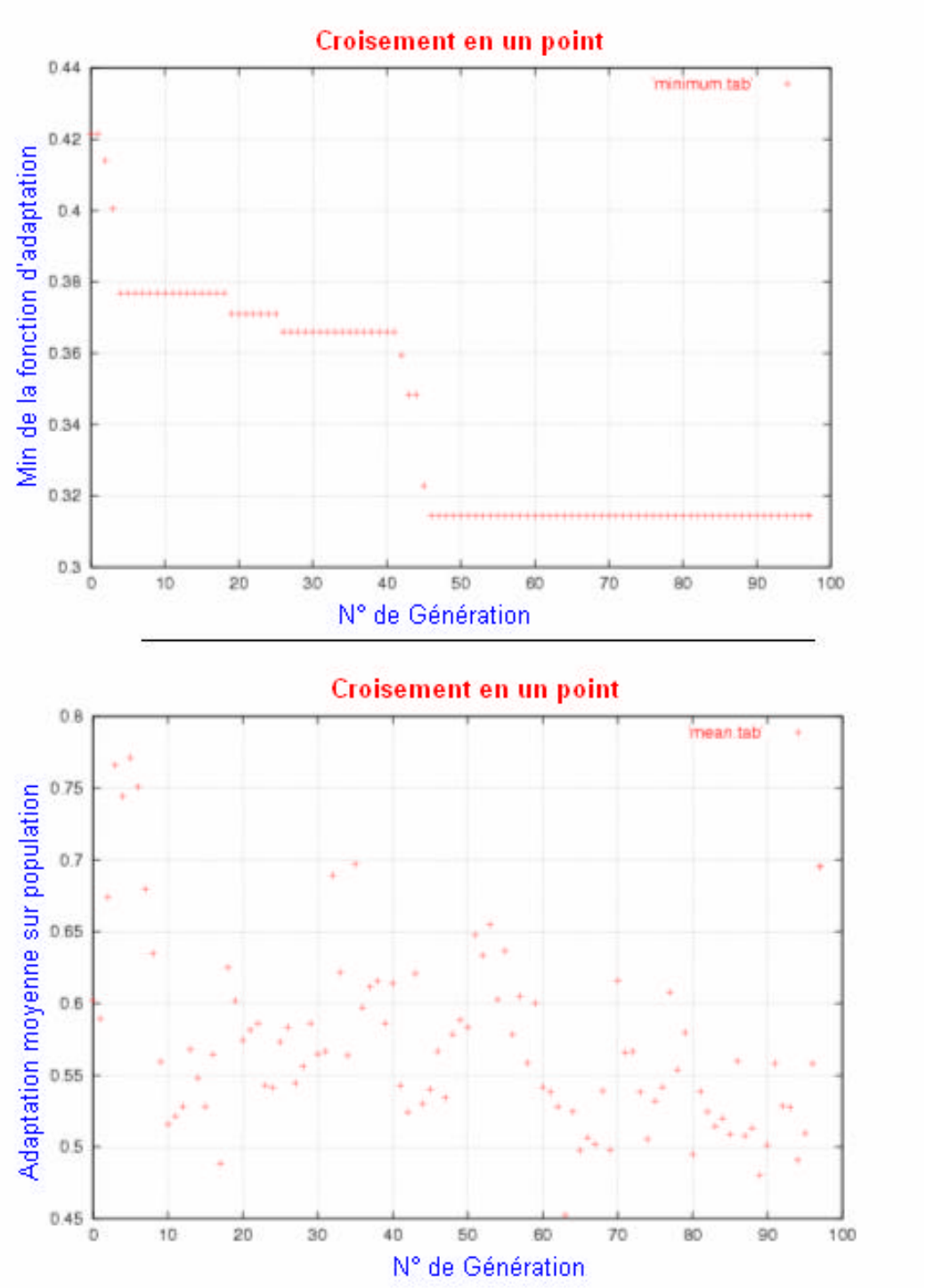


Figure 5.22 - Fonction d'adaptation min et moyenne en fonction du numéro de génération avec un point de croisement

On constate que dans cette deuxième exploration (croisement de deux individus en un seul point), la convergence est beaucoup plus rapide et plus efficace en terme de la valeur de la fonction d'adaptation globale par rapport à celle de l'exploration présidente (exploration avec croisement en deux sites).

5.2.3 Exploration avec l'algorithme NSGA-II

Cette exploration est basée sur des simulations. Le processeur SuperScalar est représenté par un simulateur «SimpleScalar ». Il y a deux objectifs à optimiser (1) la durée d'exécution en terme de nombre des cycles et (2) la surface en silicium.

Le codage des différents paramètres de microprocesseur est assuré par un chromosome binaire de longueur 33 bits, c'est le même schéma de codage que nous avons utilisé dans la section 4.5.3 de chapitre 4.

Les paramètres de configuration avec lesquels est lancé le flot d'exploration sont résumés ci-dessous :

Nom du paramètre	Valeur du paramètre
Taille de la population	48 individus
Longueur du chromosome	33 gènes
Nombre des generations	150 generations
Nombres des fonctions objectifs	2 fitness
Nombres des variables	14 variables
Probabilité de croisement	0.80
Probabilité de mutation	0.015

Tableau 5.11 - Paramètres de l'algorithme NSGA-II appliqué sur un turbo coder

La figure suivante montre le résultat de l'exploration de l'espace des solutions du processeur SuperScalar. L'application soft que nous avons utilisée dans cette exploration est le turbo coder que nous avons développé pour cet effet. La figure montre bien que les 48 solutions (48 individus) sont classées en deux fronts de Pareto.

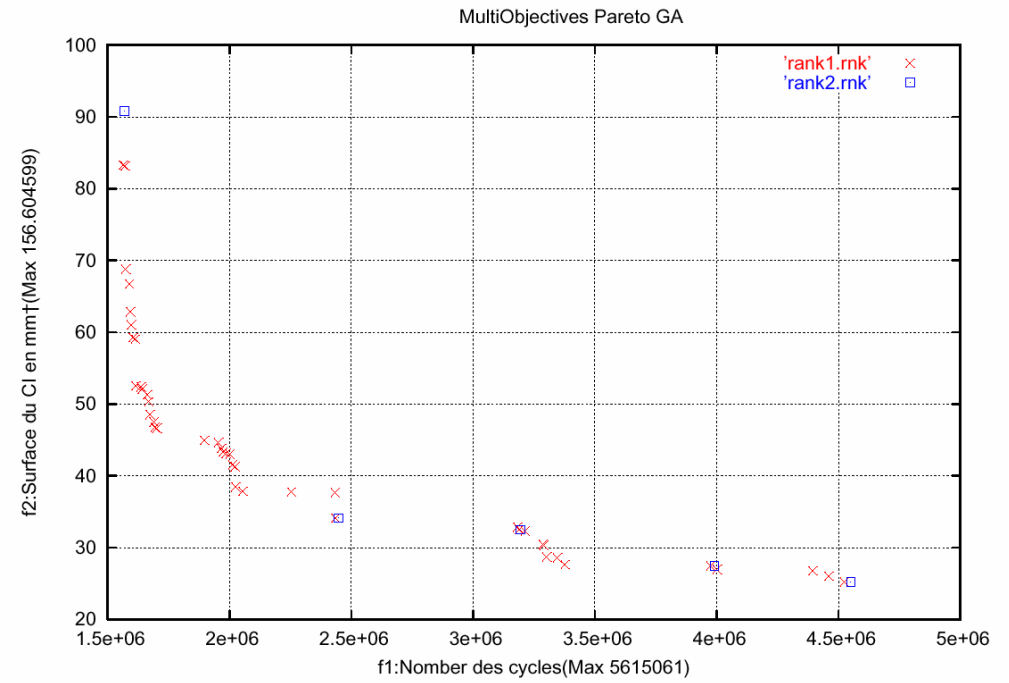


Figure 5.23 - Front de Pareto en 2D, l'application cible est le turbo coder

5.3 Conclusion

Faute de non disponibilité de code source gratuit du turbo coder, nous avons développé nos propre coder pour l'évaluation et l'exploration de l'architecture matérielle. Nous avons fait ce choix vu l'importance de ce codeur dans les systèmes de télécommunication moderne.

Le choix des paramètres d'un algorithme génétique est très important, nous avons montré expérimentalement, comme exemple, la conséquence du choix du type de croisement des individus en un ou deux points. A travers ces deux testes d'exploration à base de l'algorithme génétique multiobjectifs avec la méthode de dérogation, pour le même problème, nous avons vérifié que le croisement en un seul site est plus efficace à celui de croisement en deux sites.

Le résultat de cette exploration, représenté par les Figures 5.21 et 5.22, montre bien l'intérêt de l'utilisation du front de Pareto pour la comparaison et le choix des solutions dans le cas d'un problème multiobjectifs. Pour le concepteur, le front de Pareto est un outil visuel très facile à utiliser pour leur choix des paramètres de l'architecture la plus compatible à son application software à exécuter.

Dans ce qui suit, nous intéressons à l'utilisation de la méthode d'optimisation multiobjectifs, en particulier l'algorithme NSGA-II, pour l'optimisation des circuits analogiques à travers des implémentations sur un FPAA.

CHAPITRE 6

EXPLORATION MIXTE ANALOGIQUE/NUMERIQUE

Les circuits mixtes (Analogique/Numérique) sont dominants dans les systèmes sur puce. Malgré l'existence de techniques de conception et d'optimisation des circuits dans le domaine numérique, les techniques analogiques restent en retard. Les circuits FPAA (*Field Programmable Analog Array*) [55, 56] permettent la conception et l'implémentation des applications sur circuits analogiques re-programmables. Cette possibilité permet de tester de nombreuses configurations pour répondre à une fonctionnalité en les implémentant physiquement dans un circuit programmable avec le moindre coût.

Notre objectif dans ce chapitre consiste à proposer un flot d'exploration pour la conception de circuits analogiques basés sur les techniques d'optimisation multiobjectifs tel que les algorithmes génétiques et en particulier l'algorithme NSGA-II. Cette exploration permet d'évaluer en boucle fermée par exécution directe des configurations potentiellement intéressantes. Les deux critères permettant de les comparer sont : (1) performance et (2) les ressources FPAA utilisées.

6.1. Etat de l'art de la synthèse automatique de circuit analogique

L'outil de synthèse automatique doit réaliser deux étapes dans le flot de conception d'un circuit analogique. La première est d'extraire d'une *netlist* de spécifications la topologie du circuit et la deuxième est de calculer les paramètres de cette topologie (Figure 6.1).

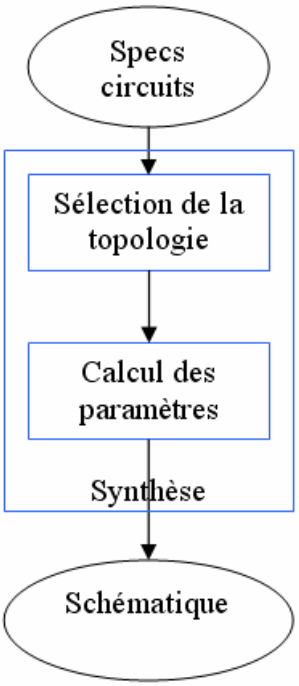


Figure 6.1 - Synoptique des outils de synthèse

6.1.1 Synthèse VHDL-AMS

Le langage VHDL est un standard IEEE (IEEE 1076-1993) pour la modélisation, la simulation et la synthèse des systèmes matériels logiques (HDL - *Hardware Description Language*). Il est aujourd'hui très largement utilisé et est supporté par tous les environnements d'aide à la conception des circuits et des systèmes électroniques (EDA - *Electronic Design Automation*). Le langage VHDL-AMS [57] est aussi un standard IEEE (IEEE 1076.1-1999). Il a été développé comme une extension du langage VHDL pour permettre la modélisation et la simulation des circuits et des systèmes analogiques et mixtes logiques/analogiques [STD1076.1]. VHDL-AMS constitue une extension du VHDL, ce qui signifie principalement:

- Toute description correcte en VHDL, l'est aussi en VHDL-AMS et produit les mêmes résultats de simulation.
- Les extensions apportées dans le VHDL-AMS conservent aussi les principes VHDL: modularité, déclarations avant usage, typage fort des données, flexibilité, extensibilité. Ces principes concernent à la fois la manière dont le langage est défini et la manière dont un modèle a été écrit.

Le langage VHDL-AMS permet de supporter la conception à plusieurs niveaux:

- Modélisation des circuits logiques et analogiques, abstraction possible grâce à des modèles comportementaux de complexités variables (des réseaux de Kirchhoff aux modèles fonctionnels à flot de données).
- Modélisation de systèmes complets (exemple d'une chaîne d'acquisition des données, d'un capteur avec traitement numérique) avec prise en compte de l'environnement (exemple des effets dus à la température).

VHDL-AMS offre en outre, un support de base pour la modélisation des systèmes non électriques (exemple capteurs, actionneurs, éléments mécaniques). Ce principe peut être utilisé pour modéliser et valider un système en tenant compte de son environnement.
Comme pour le VHDL avec XST ou LEONARDO, il faut alors utiliser des synthétiseurs spécifiques. Malheureusement pour l'instant il n'existe pas de synthétiseur VHDL-AMS industriel. VHDL-AMS reste pour l'instant simplement un langage pour la modélisation et la simulation analogique.

6.1.2 Outils Académique

Il existe de nombreux outils académiques de synthèse analogique comme IDAC, OASYS, DARWIN, ARCHEN… [58]. IDAC permet à partir d'une librairie de circuits de sélectionner le module correspondant à la fonctionnalité et aux performances que l'on souhaite. Cette librairie contient une quarantaine de circuits. Il est évident que cette limitation de choix est un obstacle et un frein à l'industrialisation. Il en est de même pour les outils comme OASYS, OASYSN, MIDAS, ARCHGEN [58], qui ont des bibliothèques limitées soient à des filtres, des amplificateurs, ou encore des convertisseurs.
De nouveaux synthétiseurs comme VASE [59] sont apparus mais sont aussi limités par leurs librairies des composants plus souples (représentation par des graphes) que dans l'outil IDAC ou OASYS.

6.1.3 Outils commerciaux: Synopsys Circuit Explorer

Circuit Explorer [60] optimise les différentes solutions analogiques par évaluation logicielle. A partir d'un circuit de base, il génère un certain nombre des solutions suivant les paramètres à optimiser. Ensuite il choisit le circuit optimal par rapport aux critères. Pour cela il associe à chaque solution une courbe de performances faite à partir de modèles HPICE ou SPECTRE (Figure 6.2).

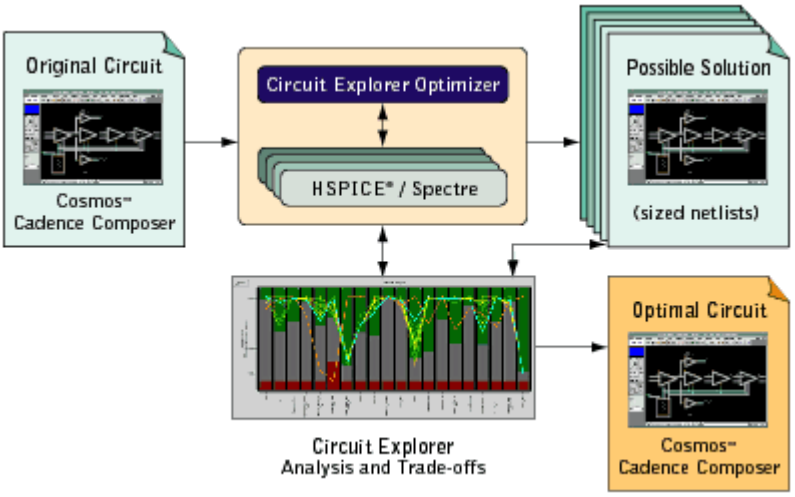


Figure 6.2 - Synoptique de circuit explorer

Néanmoins cet outil est très récent et reste un outil basé sur une validation purement logicielle.

6.1.4 Conclusion

Actuellement il n'existe aucun synthétiseur analogique. VHDL-AMS est simplement un langage pour la simulation et la modélisation. Les outils académiques font de la synthèse ou de la pseudo-synthèse mais ils sont adaptés à certains types de design (synthétiseur d'amplis, de convertisseurs Sigma-Delta, etc.). Seul l'outil Synopsys Circuit Explorer réalise une synthèse analogique mais il est trop récent pour pouvoir le juger. Nous Manquons de recul et d'expériences sur cet outil.

Ainsi nous remarquons le manque d'outil dans ce domaine. Le projet de réaliser un outil permettant d'optimiser les circuits analogiques par exécution directe comblerait vraiment un vide.

6.2. Optimisation sur FPAA

Les articles [58, 61, 62, 63, 64, 70] décrivent des méthodes d'optimisation pour la conception des circuits analogiques par des algorithmes génétiques. Ils s'appuient directement sur une plateforme reconfigurable pour faire évoluer leurs populations.

Les articles [61, 62] s'appuient sur une plateforme reconfigurable appelée PAMA (*Programmable Analog Multiplexer Array*), pour tester en temps réel le circuit. Cette plateforme est constituée par de multiplexeurs qui permettent l'interconnexion de composants discrets par l'intermédiaire d'un bus. Chaque multiplexeur est contrôlé par un registre.

L'article [63] et [58] s'appuient sur un FPTA (*Field Programmable Transistor Array*) pour faire évoluer sa population. Le FPTA est constitué d'un certain nombres de cellules comprenant huit transistors connectables entre eux. Le FPTA permet de réaliser des circuits reconfigurables au niveau transistor.

L'article [64] s'appuie sur le MPAA020, composé de blocs, comprenant un amplificateur et huit capacités. Ce composant permet de concevoir des circuits au niveau amplificateur.

6.2.1 La plateforme PAMA

Cette plateforme PAMA (*Programmable Analog Multiplexer Array*), fait évoluer la population par des modifications de connexions entre les composants. Ces modifications sont téléchargées et testées, au fur et à mesure, sur la plateforme. Chaque circuit ainsi obtenu, est

évalué en comparant l'évolution du signal de sortie à celui du signal de sortie désiré. Cette plateforme est constituée de multiplexeurs qui permettent l'interconnexion des composants discrets par l'intermédiaire d'un bus. Chaque multiplexeur est contrôlé par un registre. Ces registres sont modifiés par les bits de configuration transmis par le PC via un microcontrôleur et une liaison série RS-232.
Pour réaliser l'optimisation de la conception des circuits analogiques par algorithme génétique, nous avons besoin :

- d'implémenter l'AG sur une station de travail (un PC)
- de représenter les chromosomes par les bits de contrôle des *switchs*
- et de définir la fonction de coût comme étant la différence entre le signal de sortie désiré et le signal de sortie obtenu.

Les chromosomes (les configurations) sont envoyés à la plateforme par liaison série. Un PIC réceptionne ces données et les stocke dans le registre de contrôle des multiplexeurs. La topologie du circuit en est aussitôt modifiée. Le circuit est alors évalué avec la configuration en cours. Le signal de sortie est numérisé et mémorisé avant d'être envoyé au PC à travers la liaison série. L'AG n'a plus qu'à utiliser ces données pour générer d'autres chromosomes et les évaluer par la plateforme (Figure 6.3).

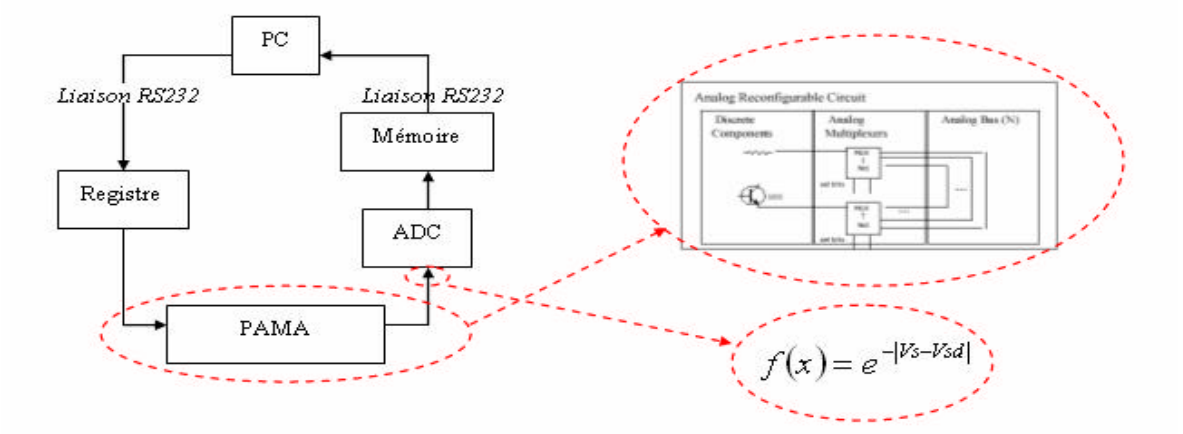


Figure 6.3 - Plateforme PAMA

Pour illustrer la méthode présentée, une réalisation d'un inverseur a été effectuée [61]. La plateforme PAMA dispose de 34 multiplexeurs de type 3-8, d'un bus analogique, de dix transistors et de deux résistances pour cette réalisation. La fonction de coût est la différence entre le signal d'entrée et le signal de sortie. L'objectif est alors de maximiser cette fonction.

Dans un premier test, les paramètres de l'algorithme génétique sont fixés suivant le tableau 6.1.

Nom du paramètre	Valeur
Taille de la population	200 Individus
Probabilité de croisement	0,65
Probabilité de mutation	0,10
Type de Sélection	Roue de loterie (*roulette wheel*)
Type de Codage chromosome	Binaire de taille 27 bits
Nombre des générations	10 (0.015% de l'espace des solutions explorées)

Tableau 6.1 – Paramètres de l'AG appliqués en premier test à la plateforme PAMA

Les composants discrets utilisés dans la PAMA pour réaliser ce circuit analogique sont : trois transistors NPN et une résistance de 4.7k? mise en série avec l'alimentation.
L'exploration de l'espace de solutions (Exécution de l'algorithme) dure une heure sur un Pentium II 233MHz.

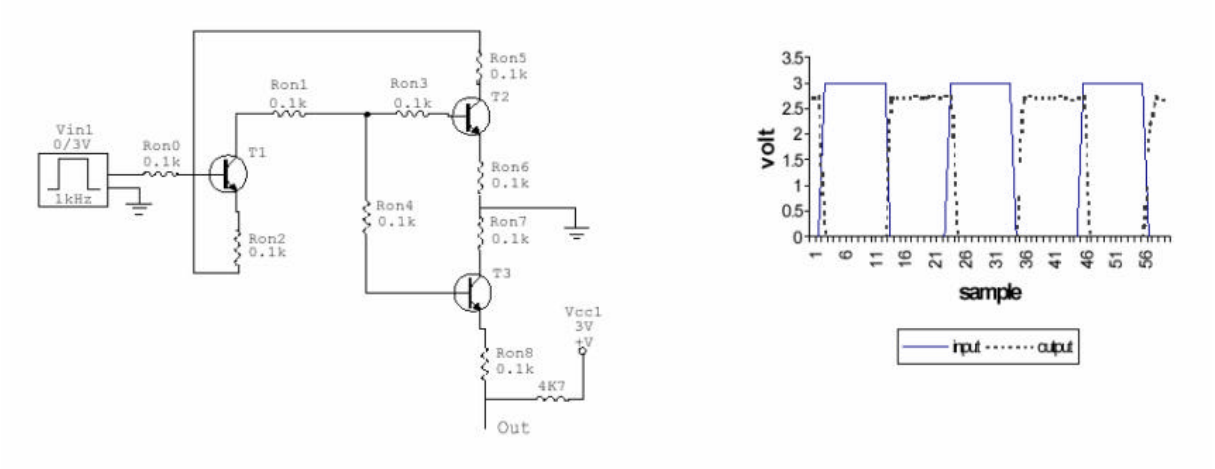


Figure 6.4 - Résultat du premier test avec la plateforme PAMA

A la fin de l'exploration, nous avons obtenu le circuit suivant (Figure 6.4) :

6.2.2 La plateforme avec un FPTA

Chaque cellule d'un FPTA est constituée par 8 transistors et de 24 *switchs* programmables [63, 64] :

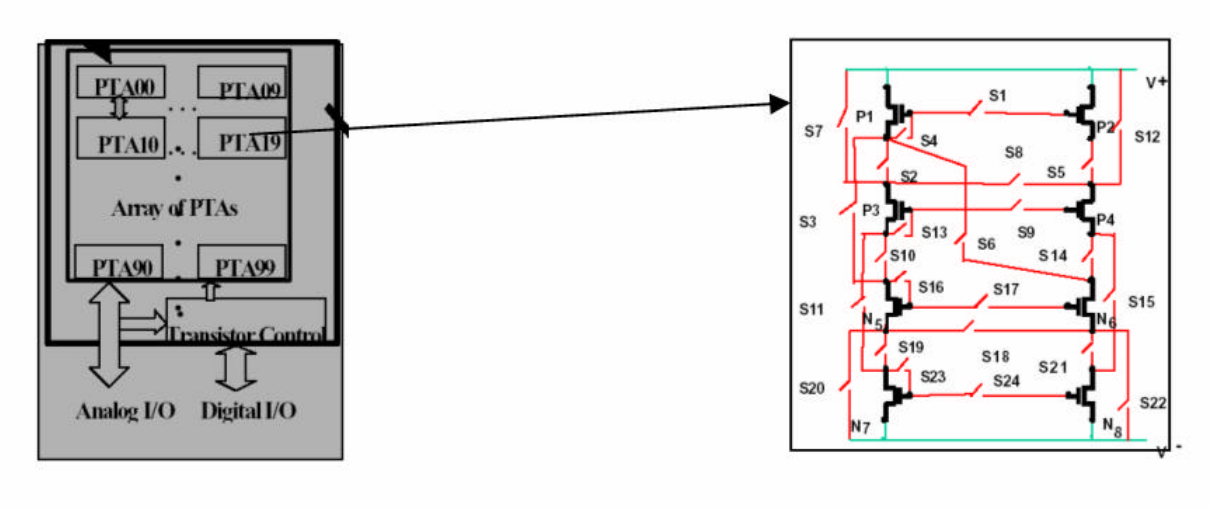


Figure 6.5 - Structure d'un FPTA

L'évolution de la population se traduite par une modification de la topologie du circuit ou les paramètres des transistors. Dans un premier test, le but est de réaliser un amplificateur avec une fonction de transfert continue et connue. Les paramètres sont les états des *switchs* d'une cellule. Le circuit va évoluer par modification de la topologie de la cellule (les paramètres des transistors restent fixes). Ensuite il suffit de mesurer la tension de sortie par rapport à celle d'entrée. Cela est possible avec les deux convertisseurs analogique/numérique et numérique/analogique de la plateforme (Figure 6.6). L'AG a pour paramètres 30 générations et 30 individus.

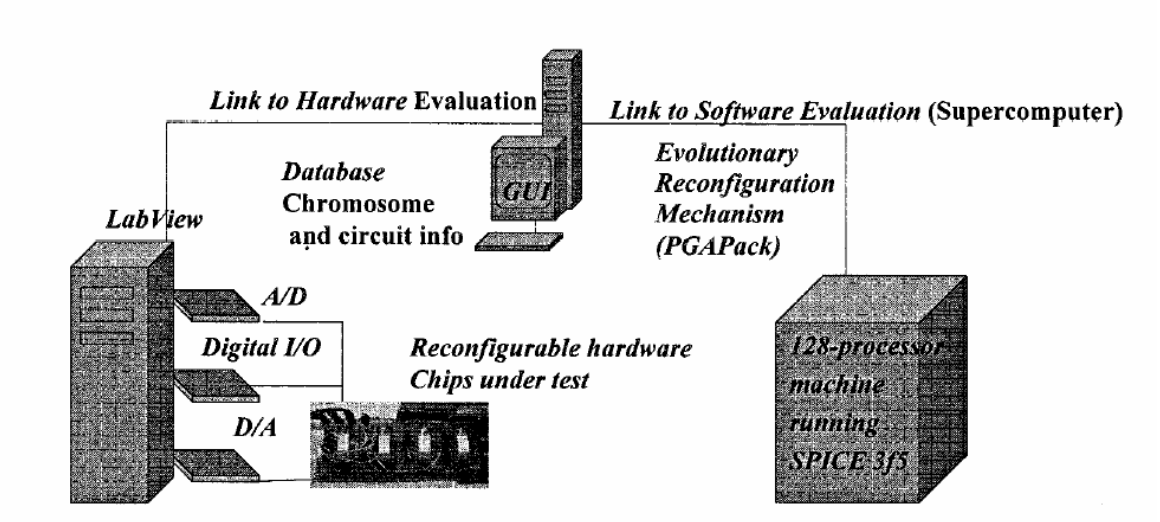


Figure 6.6 - Structure de la plateforme

Dans un second test, il s'agit à partir d'une topologie fixe de paramétrer les transistors (longueur et largeur du canal) pour obtenir une courbe gaussienne du courant de sortie par rapport à la tension d'entré. Des tests ont été effectués sur une population variant de 50 à 512 individus et sur 50 à 86 générations.

6.2.3 La plateforme MPAA020

Cette plateforme est constituée de quatre blocs (Figure 6.7) configurables comprenant un amplificateur et huit capacités [58].

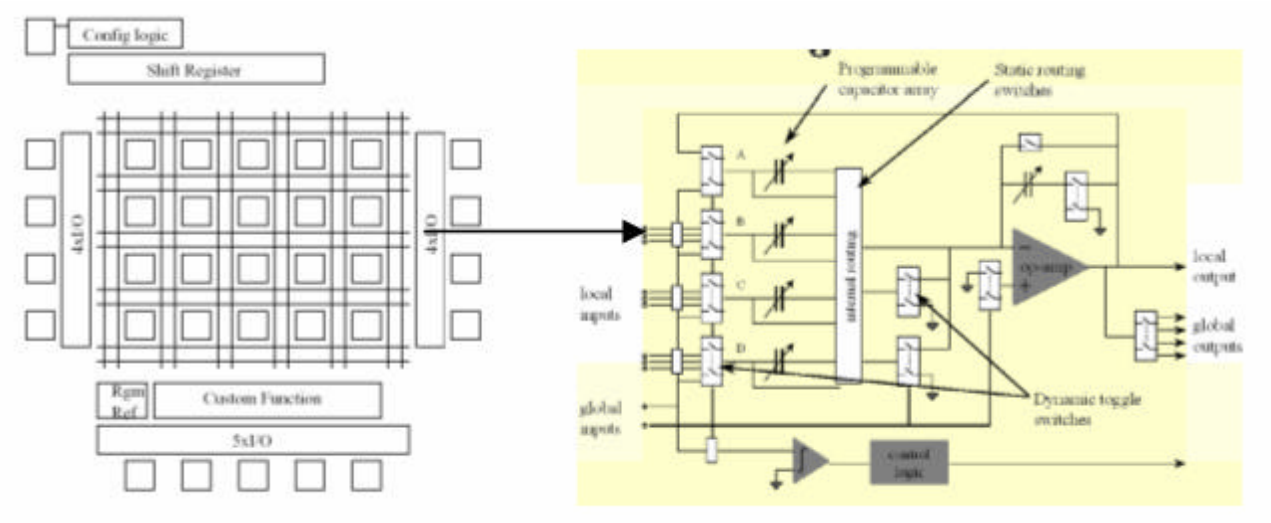


Figure 6.7 - Structure interne du MPAA020

Le PC contient l'AG qui envoie les données de configuration, le signal d'entrée et récupère les signaux de sortie (Figure 6.8)

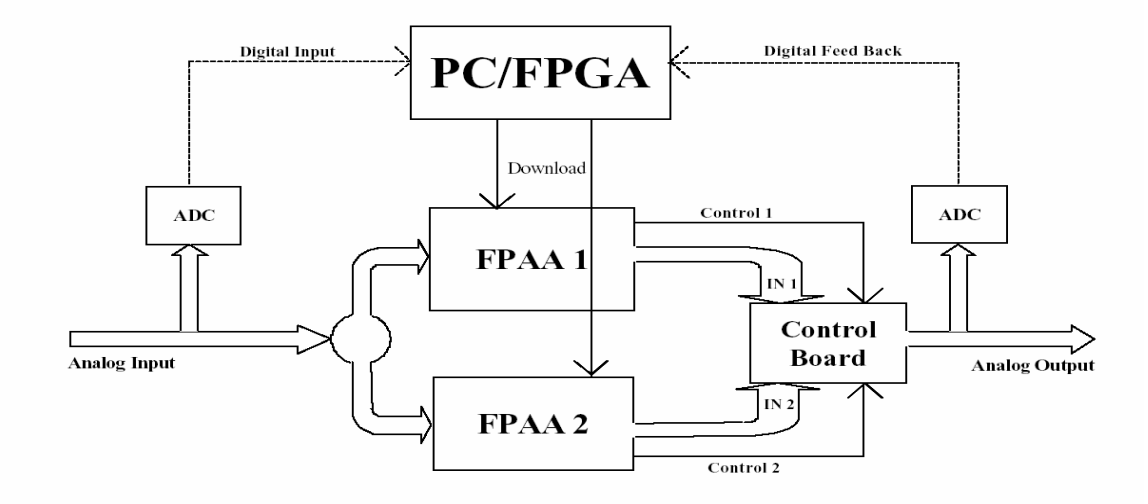


Figure 6.8 - Structure de la plateforme

Des tests ont été faits pour optimiser la fonction de transfert d'un amplificateur.

PAMA réalise des optimisations sur des circuits élémentaires. Le FPTA optimise des circuits simples limités à un amplificateur. Et la plateforme avec le MPAA020 n'optimise que par rapport à la fonction de transfert.

Il est évident alors qu'il n'existe aucune plateforme ou système capable d'optimiser un circuit analogique d'une manière multiobjectifs et en particulier pour les circuits complexes.

6.3 Plateformes Analogiques Reconfigurables dynamiquement

6.3.1 Panorama des techniques utilisées

Il existe différents FPAA. Ils se distinguent par leur granularité, leur domaine temporel, etc.

Fournisseur	Domaine temporel	Bande passante	Granularité
ANADIGM[35]	Discret	2MGHz	Grosse
LATTICE [56]	Discret	1.5MGHz	Grosse
CYPRESS [65]	Continue	10MGHz	Grosse
ZETEX [66]	Continue	12MGHz	grosse
FPTA [67]	Continue	1MGHz	fine

Tableau 6.2 - Différents types de FPAA

Pour notre plateforme, nous utilisons le FPAA AN221E04 d'ANADIGM. Il a l'avantage d'avoir une grosse granularité permettant ainsi de concevoir des circuits complexes de haut niveau.

6.3.2 Le FPAA AN221E04 d'Anadigm

La plateforme FPAA peut être utilisée pour réaliser des fonctionnalités et des circuits analogiques complexes tels que :

- traiter des signaux analogiques provenant d'une sonde, d'un capteur, etc.
- traiter des signaux ultra basse fréquences
- réaliser des systèmes d'asservissement
- réaliser des filtres complexes

Les caractéristiques techniques de cette plateforme FPAA (à base du composant programmable d'Anadigm AN221E04 [55]) sont résumées dans le tableau suivant :

	Fréq. Max Analog	Fréq. Max Numér	Nbre AOP	Nbre comparateur	Nbre Capa	BP	Vmax (v)	Vmin (v)	Nbr. de switch
AN221E04	8MHz	40MHz	8	4	32	2MHz	1.9	-1.9	>9000

Tableau 6.3 - Caractéristiques du FPAA AN221E4

Le circuit AN221E04 est découpé en 4 blocs analogiques configurables (CAB) qui peuvent être connectés entre eux et reliés aux quatre I/O paramétrables et aux deux sorties paramétrables (Figure 6.9).

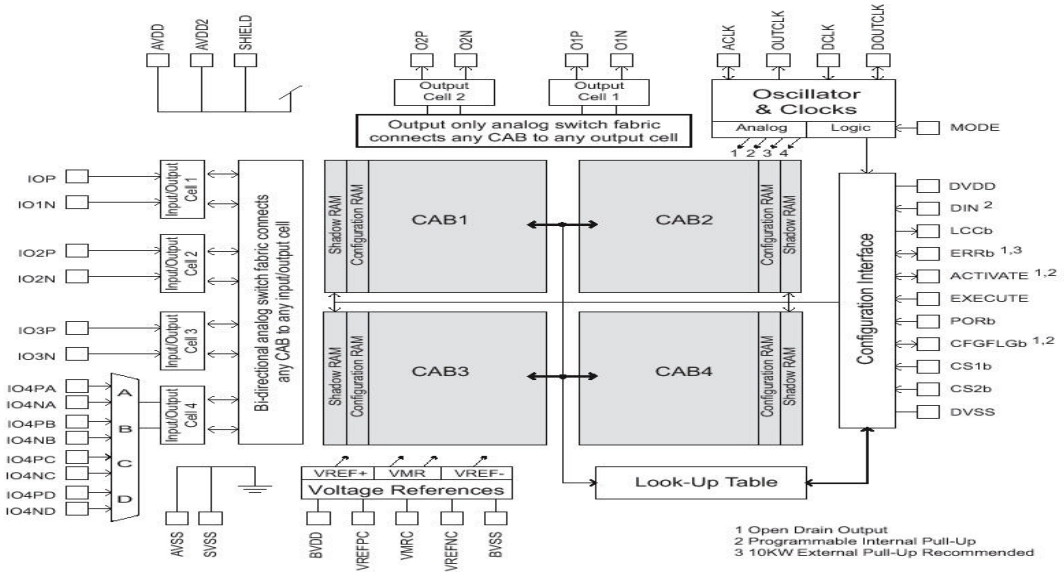


Figure 6.9 - Structure interne du AN221E04

Dans chaque CAB se trouve deux amplificateurs, un comparateur et huit capacités programmables (Figure 6.10). Ces composants sont interconnectés entre eux par des *switchs* dynamiques et statiques. Un *switch* dynamique est une liaison par capacités commutées.

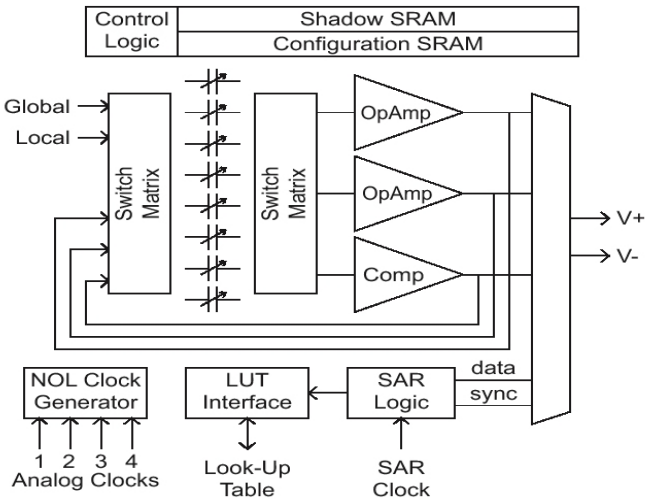


Figure 6.10 - Structure interne d'un CAB

De cette manière, il est facile de réaliser au sein d'un CAB de nombreux designs. Par exemple un filtre passe bas peut être conçu de la façon suivante :

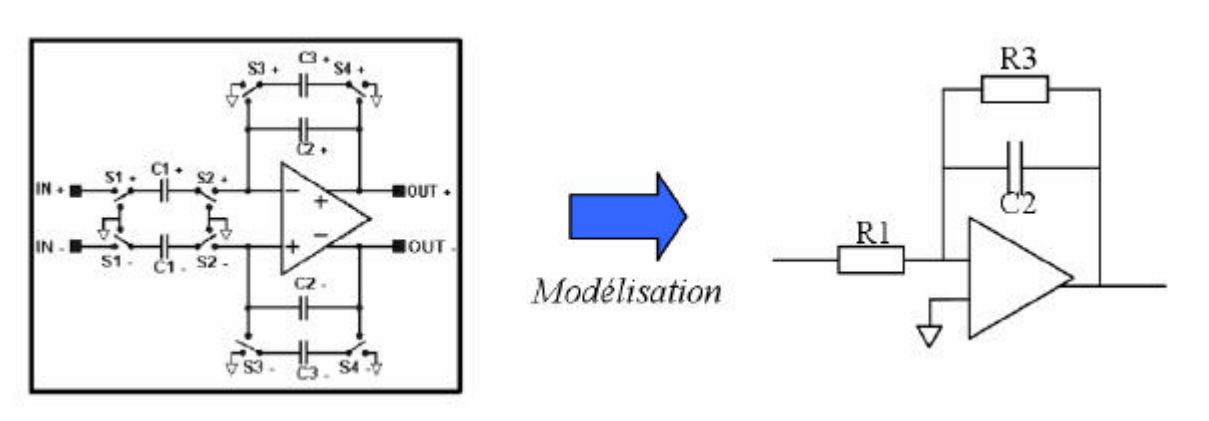


Figure 6.11 - Schéma d'un filtre

Avec $R1 = \frac{1}{C1.fe}$ et $R2 = \frac{1}{C2.fe}$ où f_e représente la fréquence des capacités commutées.

Les blocs d'entrée et de sortie sont paramétrables comme suit :

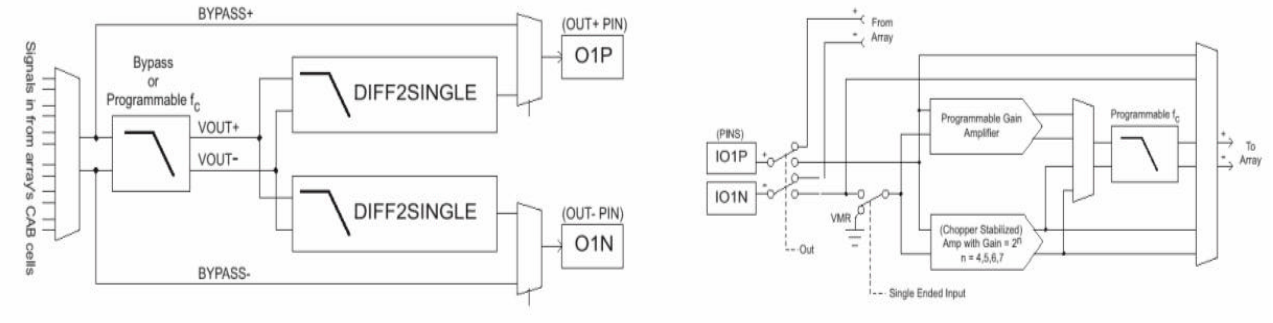


Figure 6.12 - Structure interne des I/O et des Outputs

On peut utiliser des filtres paramétrables en sortie et en entrée (par exemple des filtres anti-repliement ou de lissage). D'autre part, des générateurs de tension et de courant sont aussi utilisables pour les comparateurs, par exemple :

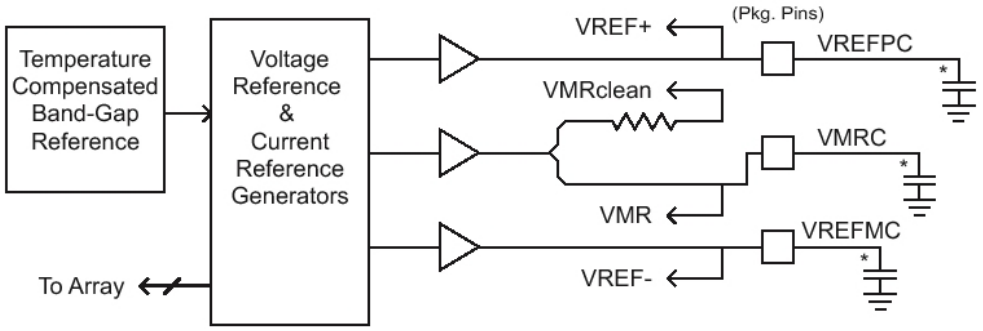


Figure 6.13 - Structure interne de générateurs de tension et de courant

La LUT dans ce composant permet de générer des signaux ou de créer des fonctions de transfert. La LUT est une mémoire de 256 octets qui peut contenir par exemple des valeurs de tension. Il est aussi possible d'utiliser un convertisseur analogique numérique (à approximations successives) mais il est toutefois difficile de récupérer ce signal à cause d'un

signal de synchronisation qui vient modifier le potentiel de référence lorsqu'on mesure le signal.

Le FPAA d'Anadigm permet contrairement aux autres FPAA de concevoir des circuits analogiques complexes de haut niveau. Il a la possibilité d'être reconfiguré partiellement sans obligation d'initialiser le FPAA. Cet avantage est un gain de temps pour l'exploration de circuits sur FPAA. Ce sont les raisons de notre choix de FPAA d'Anadigm dans le cadre de ce travail.

6.4 Plateformes Analogiques Reconfigurables

6.4.1 Couplage FPAA-FPGA-PC

La Figure 6.14 montre le schéma de couplage de la plateforme FPAA à une station de travail (PC). A ces deux éléments, s'ajoute une carte FPGA avec un convertisseur Analogique/numérique pour réaliser une boucle fermée et remonter les informations depuis la carte FPAA vers le PC.

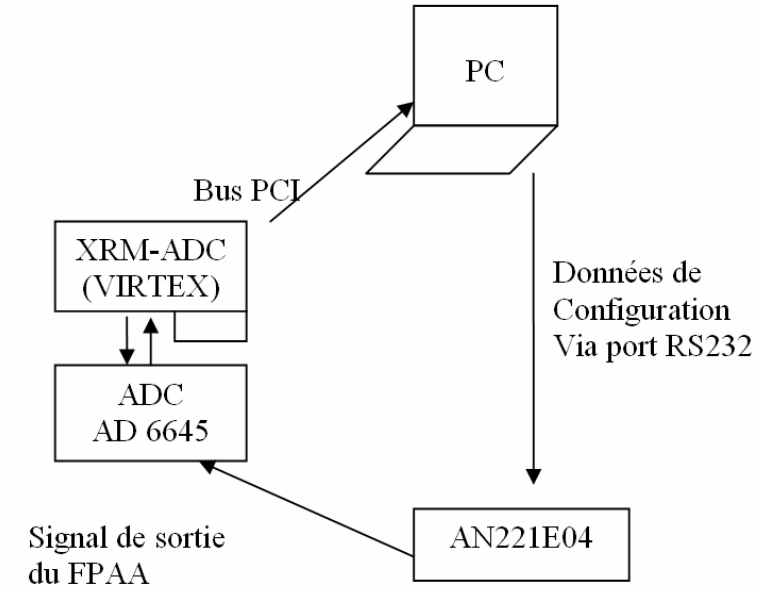


Figure 6.14 - Synoptique du couplage FPAA-FPGA-PC

Le PC permet d'effectuer la configuration du FPAA via le port série. Le convertisseur analogique numérique XRM-ADC de Alpha-Data numérise le signal de sortie du FPAA pour que le circuit du FPAA puisse être évalué. Le FPGA transmet ces données au PC par le bus PCI. Le convertisseur est le AD6649 à approximations successives de 14 bits pouvant fonctionner à 105Msps. Il est connecté à la carte ADM-XRCII contenant un VIRTEX II XC2V6000 (environ 6Milllions de portes) et 6 bancs de mémoires de 512k. Cette carte est connectée sur une carte mère XRM-ADC, à son tour, pouvant se connecter au bus PCI du PC. Ce bus peut fonctionner sur la carte à 50MHz maximum.

6.4.2 Environnement logiciel

6.4.2.1 Logiciel de conception du design analogique

Pour établir la configuration du circuit, nous utilisons *AnadigmDesigner*. Ce logiciel permet de spécifier des circuits analogiques graphiquement (Figure 6.15) puis de télécharger

le fichier de configuration dans la carte FPAA. La figure 6.15 montre cet environnement de travail.

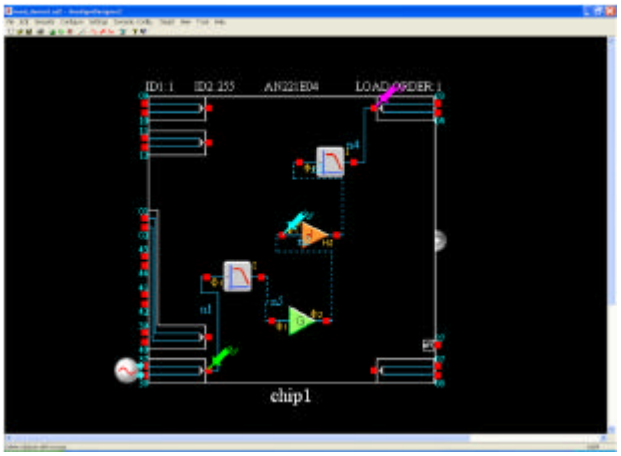


Figure 6.15 - Exemple de design réalisé avec AnadigmDesigner

Il est évident que pour automatiser la conception de circuit, nous ne pouvons pas dessiner manuellement, à chaque fois, le circuit à tester. Par exemple si on a une population de 20 individus et 20 générations, on aurait 400 circuits à dessiner. On perd tout l'intérêt de l'automatisation. Il faudrait trouver une méthode pour récupérer, traduire et télécharger en boucle les données de configuration. Le logiciel génère un code C correspondant au circuit dessiné. Dans ce code C se trouve un tableau qui contient les données transmises au FPAA pour se configurer. Ces données sont les paramètres et la topologie du circuit. Ainsi nous avons les données de configuration qu'on peut modifier et transmettre, indépendamment du logiciel. Nous n'aurons pas besoin à chaque fois de passer par le logiciel pour télécharger un design dans le composant.

6.4.2.2 Logiciel de programmation du FPGA VirtexII

Nous avons utilisé *Xilinx ISE* pour implémenter le design qui contrôle le convertisseur A/N, mémorise les données et les communique au PC via le bus PCI.

6.4.2.3 Environne ment de programmation de l'algorithme génétique

L'algorithme génétique mono-objectif [68] a été réalisé sous l'environnement de programmation VISUAL C++. Les programmes qui permettent de communiquer avec le FPGA et FPAA ont aussi été développés sous VISUAL C++. La carte de conversion et du bus PCI est commandée par un programme C comme le téléchargement des données de configuration du FPAA. Le synoptique des différents programmes est le suivant (Figure 6.16) :

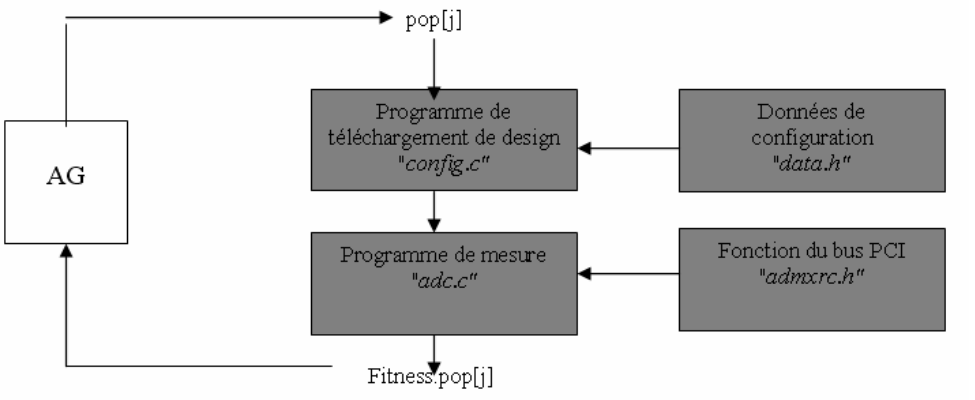


Figure 6.16 - Synoptique des programmes

Les blocs en gris représentent les programmes relatifs à la programmation de la carte FPAA et à la gestion et récupération des données au niveau du convertisseur A/N de coté FPGA.

6.4.3 Technique de téléchargement des circuits dans le FPAA

Les données de configuration sont téléchargées dans une SSRAM. Pour permettre une reconfiguration partielle et dynamique, deux SSRAM sont disponibles sur la carte FPAA. L'une, appelée *« Shadow SRAM »*, reçoit les données ; l'autre appelée *« SRAM de configuration »*, configure le chip.
Les données de configuration peuvent être téléchargées de différentes manières :

- Par un microprocesseur
- Par une EEPROM
- Par un PC via le port série (c'est cette procédure que nous avons utilisée)

Les deux premières méthodes commandent directement les broches de configurations du FPAA tandis que la dernière utilise le PIC présent sur la carte pour commander ces broches.

6.4.3.1 Structure de données de configuration

Les données de la configuration FPAA sont des suites de nombres de 8 bits. Elles commencent toujours avec l'en-tête suivant :

```
/* The header for the configuration stream */
0xD5, //mot de synchronisation
0xB7, // JTAG0
0x22, // JTAG1
0x0 , // JTAG2
0x80, // JTAG3
0x1 , // Device ID
0x5 , // Control
```

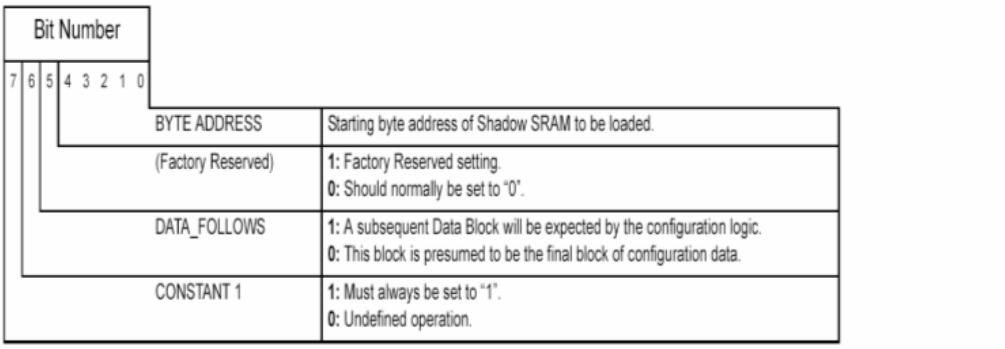
BANK ADDRESS	BYTE ADDRESS
	1F 1E 1D 1C 1B 1A 19 18 17 16 15 14 13 12 11 10 0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00
00	Lower Auxiliary Shadow SRAM Bank
01	Upper Auxiliary Shadow SRAM Bank
02	CAB 1 Lower Shadow SRAM Bank
03	CAB 1 Upper Shadow SRAM Bank
04	CAB 2 Lower Shadow SRAM Bank
05	CAB 2 Upper Shadow SRAM Bank
06	CAB 3 Lower Shadow SRAM Bank
07	CAB 3 Upper Shadow SRAM Bank
08	CAB 4 Lower Shadow SRAM Bank
09	CAB 4 Upper Shadow SRAM Bank

Tableau 6.4 – Plage d'adressage des CABs

Ensuite, nous avons les données qui vont configurer les CAB. Chaque CAB a deux mémoires associées. Ainsi avant d'envoyer les données, il faut spécifier sa mémoire, son adresse et le nombre de données :

```
/* Start of data block */
0xCC, /* Byte address: 12 */
0x0 , /* Bank address:  0 */
0xC , /* Byte count:   12 */
```

Le mot *byte adress* contient aussi une information spécifiant si le bloc de données qui suit sera le dernier ou non. Enfin on termine chaque bloc par le mot 0x2A.
La figure 6.17 montre cette procédure de programmation de FPAA.



Bit Number		
7 6 5 4 3 2 1 0		
	BYTE ADDRESS	Starting byte address of Shadow SRAM to be loaded.
	(Factory Reserved)	**1:** Factory Reserved setting. **0:** Should normally be set to "0".
	DATA_FOLLOWS	**1:** A subsequent Data Block will be expected by the configuration logic. **0:** This block is presumed to be the final block of configuration data.
	CONSTANT 1	**1:** Must always be set to "1". **0:** Undefined operation.

```
11010101    //0xD5 is the required sync header.
10110111    //0xB7 is Least Significant Byte of JTAG ID word. (AN221E04)
00100010    //0x22 is byte 2 of JTAG ID word.
00000000    //0x00 is byte 3 of JTAG ID word.
10000000    //0x80 is Most Significant Byte of JTAG ID word.

00000001    //User assigns any Chip ID except 0xFF.

00000101    //Control Byte - PULLUPS are enabled.
            //ENDEXECUTE - Transfer Shadow SRAM to Configuration
            //SRAM as soon as this download is complete.

11000000    //Constant 1, DATA_FOLLOWS 1, start BYTE address is 0
00000000    //The starting BANK address is 0

00000000    //0x00 byte count field means 256 data bytes follow.

datadata    //The first configuration data byte.
datadata    //The second configuration data byte.
...
datadata    //the 256th configuration data byte.

00101010    //0x2A is the Basic error checking constant expected.
```

Figure 6.17 - Procédure de configuration de circuit FPAA

6.4.3.2 Protocole de communication entre le PIC et le PC

Ce protocole de communication est décrit dans la référence [69]. C'est cette procédure de communication qu'on adopté au cours ce travail.

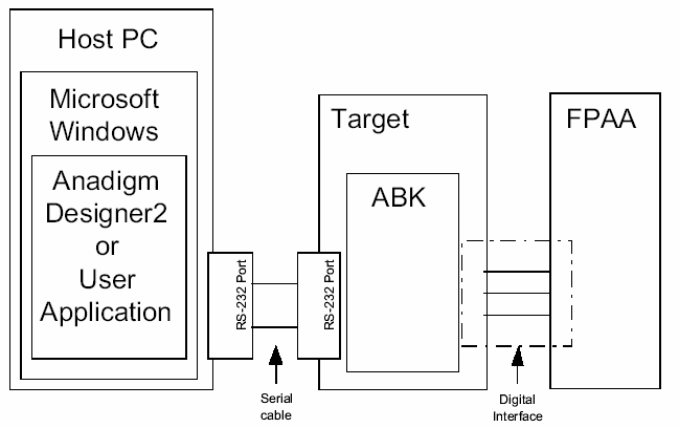


Figure 6.18 - Schéma des composants de communication de l'ABK

La communication entre le PC et le PIC utilise le protocole ABK d'Anadigm. Ce protocole est structuré de la manière suivante :

STX(0x02)	Commande	Paramètre	…	Donnée	…	ETX(0x03)

Figure 6.19 - Schéma des composants de communication de l'ABK

Chaque commande commence avec le caractère STX et se termine avec le caractère ETX. La commande dépend de l'action désirée :

- Configuration complète ou primaire : '0' (paramètre égale au nombre de données à transmettre)
- Configuration dynamique ou partielle : '1' (paramètre égale au nombre de données à transmettre)
- Reset du chip : 'R' (paramètre égale au numéro du chip à reseter)
- Etc.

Les paramètres et les données dépendent de la commande. Ils sont au format ASCII. Par exemple pour une donnée *0xA5* ont transmettra les caractères 'A' et '5'.

Prenons le cas d'une configuration primaire (Figure 6.20) : On transmet d'abord le caractère 0x02 (*char (0x02)*). Ensuite on transmet la commande '0' et le nombre de données de configuration du chip (en base 16). Puis on attend l'initialisation du chip (environ 20 ms). On envoie 10 zéros pour synchroniser les horloges et on envoie les données. On termine par deux zéros et par le caractère 0x03 (*(char (0x03)*).

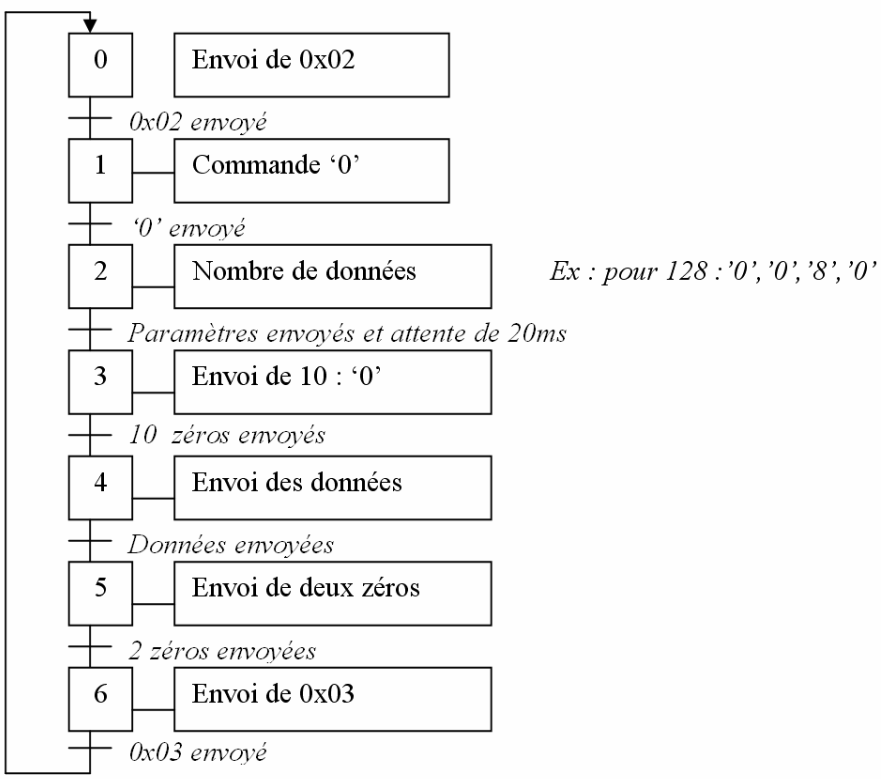


Figure 6.20 - Protocole de communication PC-PIC

Pour configurer complètement le circuit, il faut environ 25 ms et pour configurer partiellement le chip, ce temps est réduit puisqu'on n'a pas besoin d'attendre l'initialisation de 20 ms soit environ 1ms au maximum.

6.4.4 Techniques de mesure

Pour effectuer le choix d'une solution par rapport à une autre, il est nécessaire d'extraire des informations ainsi la carte XRM-ADC d'Alpha-Data permet de mesurer et de convertir un signale analogique en numérique. Le FPGA de la carte ADM-XRC-II mémorise ces données et les envoie au PC via la carte ADC-PMC connectée au port PCI du PC.

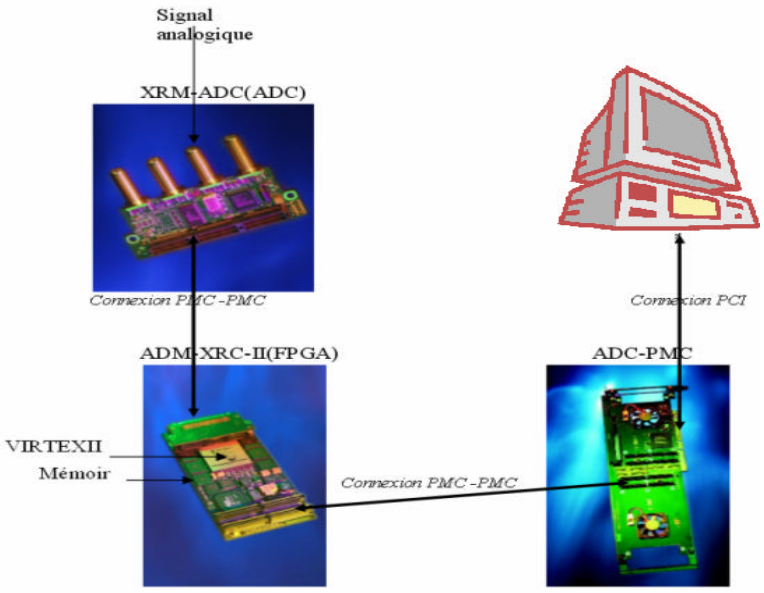


Figure 6.21 - Schéma de connexions des composants pour la mesure du signal

Nous avons créé un design pour la carte FPGA. Ceci nous permet de :

- Gérer les signaux de commande du convertisseur
- Stocker les données du convertisseur dans les mémoires de la carte ADM-XRC-II
- Réagir à une requête d'acquisition du PC
- Envoyer une requête lorsque l'acquisition est terminée
- Transmettre les données au bus PCI

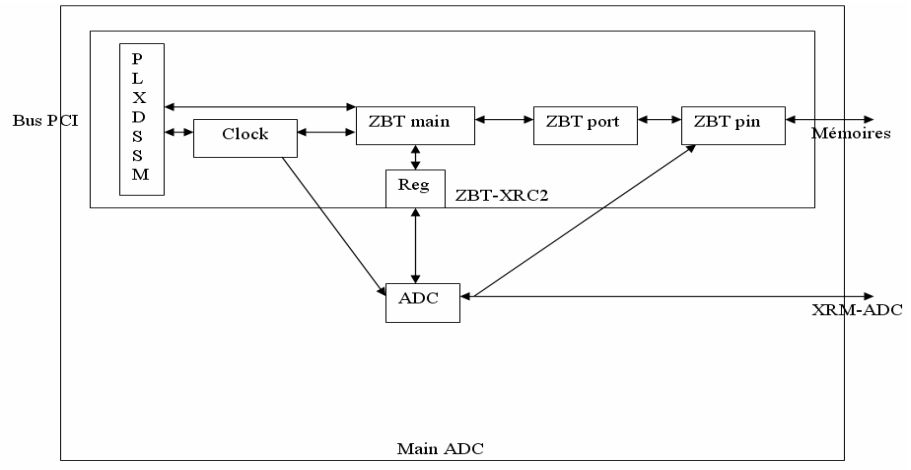


Figure 6.22 - Architecture générale du module FPGA

La figure 6.22 montre l'architecture générale du module implémenté sur le FPGA pour la gestion du convertisseur et l'acquisition de données.

6.4.4.1 Le protocole de communication entre le FPGA et l'ADC

Le convertisseur A/N est contrôlé par un programme logé dans le FPGA (Figure 6.23).

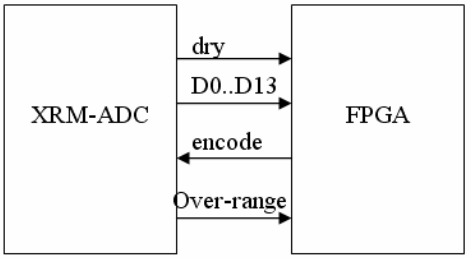


Figure 6.23 - Connections des signaux de protocole entre le FPGA et l'ADC

Ce convertisseur a une résolution de 14 bits et fonctionne jusqu'à 86 MPS. Il a une bande passante de 2 MHz à 25 MHz. Les signaux d'entrée ne peuvent dépasser 1V. Le signal *encode* est l'horloge qui spécifie la vitesse de conversion. Le signal *dry* devient haut quand la conversion est finie et que la donnée est présente sur le bus.

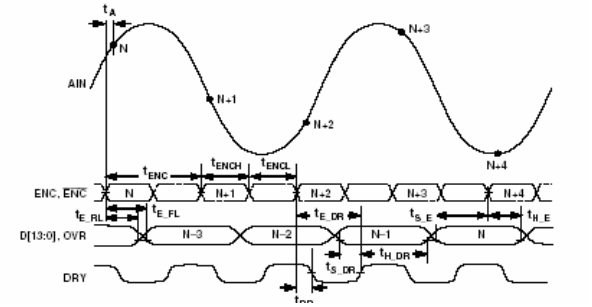


Figure 6.24 - Protocole entre le FPGA et l'ADC

6.4.4.2 L'interface entre le bus PCI et l'ADC

Cette interface permet de commander l'ADC, de capter le signal de demande d'acquisition du PC et de signaler au PC la fin de l'acquisition. Pour capter une demande d'acquisition, l'interface scrute un registre contenant « 1 » pour une acquisition. Une fois captée, l'interface utilise l'horloge du bus PCI de 50 MHz maximum, la régénère en la doublant et la connecte au port *encode* de l'ADC.

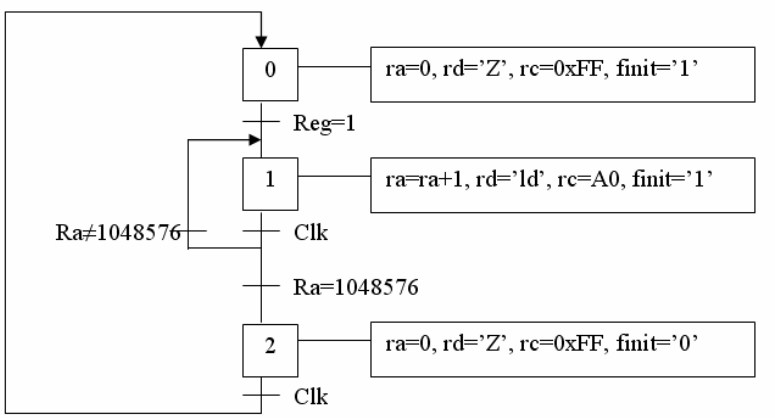


Figure 6.25 - Machine d'état de l'interface

Avec cette procédure (Figure 6.25), l'ADC fonctionne au maximum de sa fréquence et commence à convertir le signal analogique à chaque fois que l'interface reçoit une transition haute du signal *dry* de l'ADC. Mémorisation de donnée présente sur le bus dans un registre tampon jusqu'au front descendant de *dry*, qui est aussi le front montant de l'horloge des SSRAM. La donnée est alors mise en mémoire et l'adresse est incrémentée. Une fois que la mémoire est pleine, l'interface envoie un événement avec le signal *finit*. Le PC comprend ce signal comme une interruption. Cette machine d'états est spécifiée en VHDL et implantée dans le FPGA.

6.4.4.3 Synthèse, Placement et routage

Après synthèse, placement et routage, le module du contrôle FPGA donnera les caractéristiques suivantes (Figure 6.26):

Number of External IOBs	61%
Number of LOCed External IOBs	0%
Number of SLICEs	1%
Number of BUFGMUXs	18%
Number of DCMs	25%
Number of TBUFs	3%
The Average Connection Delay for this design is:	3.127 ns
The Maximum Pin Delay is:	15.699 ns
The Average Connection Delay on the 10 Worst Nets is:	10.604 ns

Figure 6.26 – Ressources utilisées par le module sur FPGA

Nous remarquons qu'il occupe une place infiniment petite dans le FPGA, mais cette place pourra être comblée si on effectue le téléchargement des données de configuration via le bus numérique (cf. technique de téléchargement).

6.4.4.4 Résultat

Nous avons ci-dessous réalisé une acquisition d'un signal sinusoïdal de 1MHz de 0.1V (Figure 6.27) :

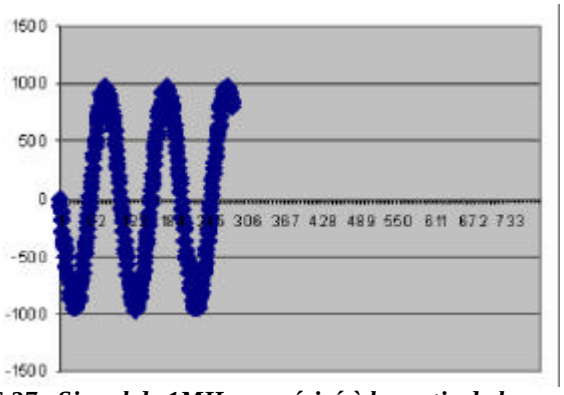


Figure 6.27 - Signal de 1MHz numérisé à la sortie de la carte FPAA

Le but de ce test est de vérifier le bon fonctionnement de la chaîne d'acquisition. Par contre, comme l'ADC a une bande passante de 2MHz à 25MHz, il est impossible d'échantillonner un signal continu et un signal carré (limite de Shannon).

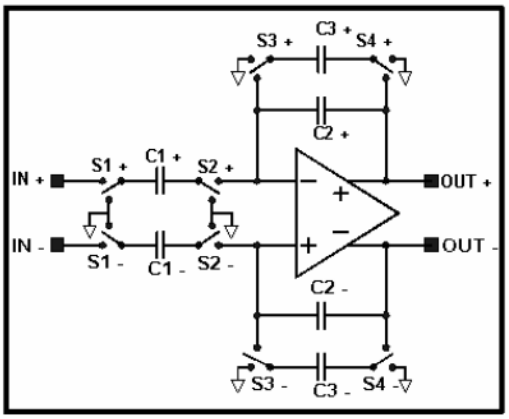
En conclusion, le module réalisé numérise des signaux analogiques à une fréquence de 86 Mps. Cette fréquence est largement suffisante par rapport à la fréquence maximum (40MHz max) des signaux du FPAA. Néanmoins il reste limité à cause de l'ADC à des signaux de bande passante 2 - 25 MHz.

6.5 Cas d'études

Après avoir bien vérifié le bon fonctionnement de la chaîne d'acquisition et les différentes liaisons de communication entre les cartes (FPAA, FPGA) et la station de travail (PC), nous pouvons, maintenant, appliquer notre méthodologie de conception à des circuits analogiques suffisamment complexes tels qu'un filtre passe-bas et un filtre réjecteur.

6.5.1 Filtre passe bas

On souhaite paramétrer le gain et la bande passante du filtre passe-bas pour avoir un signal de sortie d'amplitude de 1V avec un signal d'entrée de 0.7V sous une fréquence de1MHz.



$$f_0 = \frac{f_c}{\pi}\frac{C_3}{(2C_2 + C_3)}$$

$$G = \frac{C_1}{C_3}$$

Figure 6.28 - Schéma et paramètres du filtre passe bas

Dans ce premier cas d'étude, on utilise l'AG simple. Nos chromosomes sont alors composés des trois capacités codées sur 8 bits (1 à 255). Chaque chromosome sera alors codé sur 24

bits. On aura 2^8. 2^8. 2^8 = 4096 configurations possibles. Il faut aussi les localiser dans les données de la configuration de FPAA. Elles se trouvent dans le bloc 3 :

```
/* Start of data block */
0xDE, /* Byte address: 30 */
0x3 , /* Bank address:  3 */
0x14, /* Byte count:   20 */

/* Data bytes for block */
0xA , 0x0 , 0xC , 0x1 , 0x81, 0xC , 0x1 , 0x81,
0xB9, 0x0 , 0x10, 0x0 , 0x0 , 0x0 , 0x0 , 0x3 ,
0x0 , 0x0 , 0x0 , 0x7 ,
```

Figure 6.29 – Données de la configuration FPAA

Les chiffres en bleu (Figure 6.29) représentent les données de la configuration des trois capacités.

Avec une tension d'entrée de 0.15V, les paramètres de l'algorithme génétique sont fixés suivant le tableau 6.5.

Paramètres de l'AG	Valeur
Taille de la population	6 individus
Probabilité de croisement	0,70
Probabilité de mutation	0,15
Type de Codage chromosome	Binaire de taille 88 bits
Nombre des générations	30 générations

Tableau 6.5 – Paramètres de l'algorithme génétique

Le résultat de cette exploration est représenté par la Figure 6.30.

- la première courbe montre la valeur max prise par le meilleur individu à chaque génération
- la seconde montre la valeur moyenne des individus à chaque génération

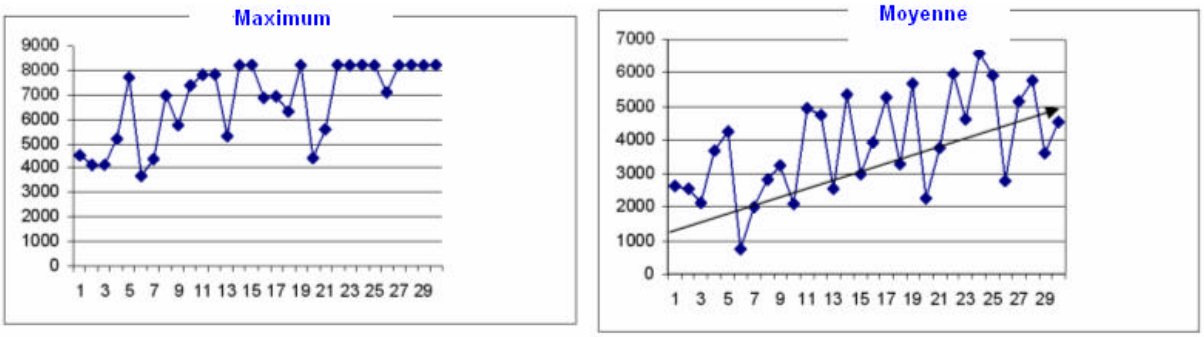


Figure 6.30 - Maximum et moyenne des individus dans la population

La courbe « Maximum » n'est pas strictement croissante, puisque l'AG utilisé dans ce cas n'est pas élitiste (pas de conservation du meilleur individu d'une génération à l'autre).

6.5.2 Filtre réjecteur

Dans ce deuxième cas d'étude, le but est de concevoir un filtre réjecteur (*rejecter filter*) avec le meilleur compromis gain/bande-passante. Donc, c'est un problème d'optimisation multiobjectifs. Les deux objectifs à optimiser sont : (1) le gain du filtre et (2) la bande passante.

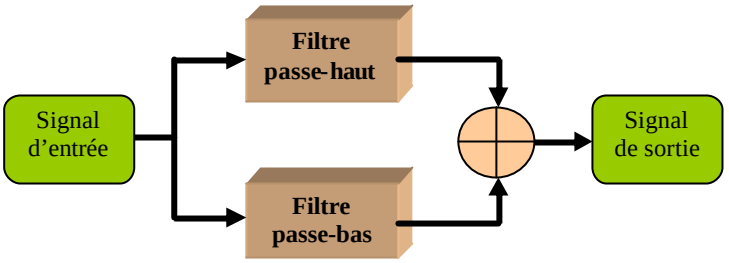


Figure 6.31 – Schéma synoptique d'un filtre réjecteur

Les variables de l'espace d'exploration sont les 22 capacités du filtre réjecteur (Figure 6.32) dont :

- huit capacités au niveau du filtre passe-haut
- huit capacités au niveau du filtre passe-bas
- et 6 capacités au niveau de l'additionner

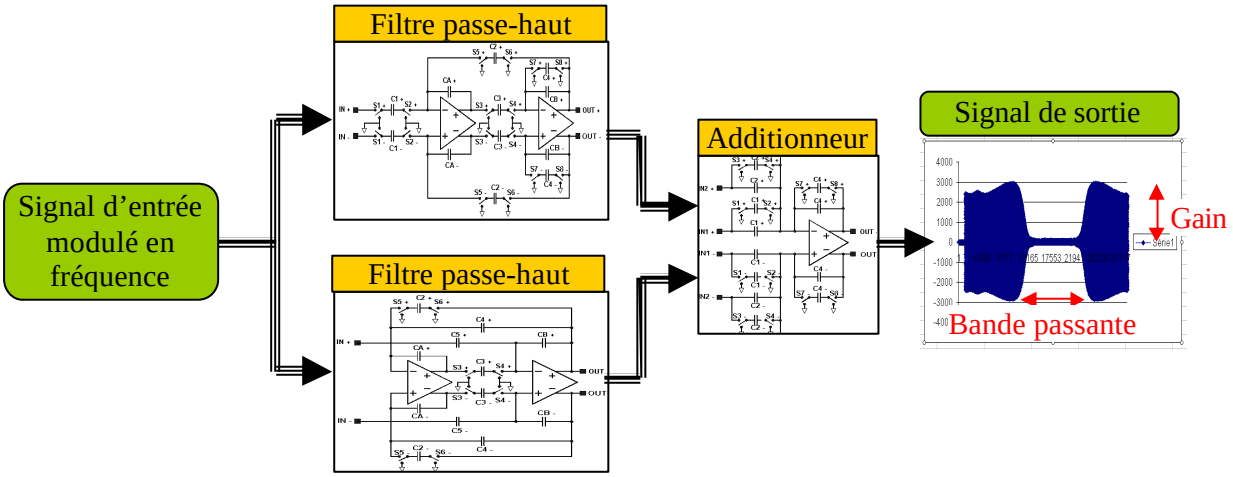


Figure 6.32 – Schéma d'un filtre réjecteur

Chaque capacité peut avoir 16 différentes valeurs possibles (codés sur 4 bits), ce que implique en total un chromosome de longueur 88 bits (4x22 bits), ainsi nous avons un espace des solutions à explorer de taille 2^{88} configurations possibles (Figure 6.33).

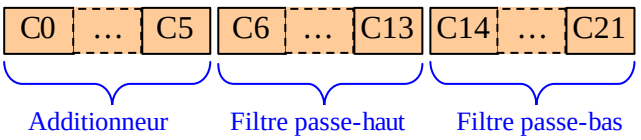


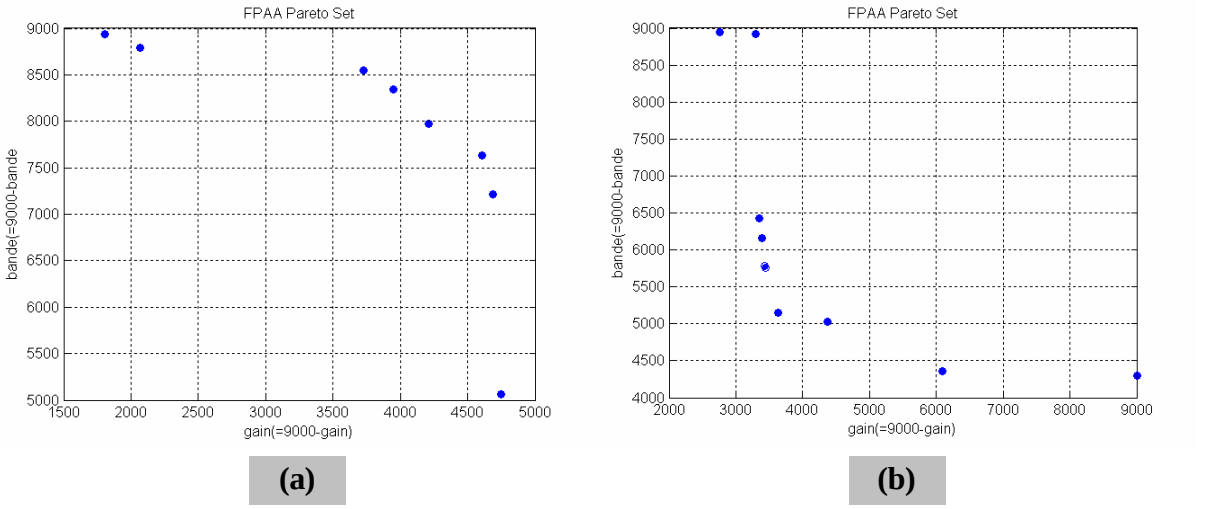
Figure 6.33 - Chromosome

Nous avons lancé l'exploration de cet espace de solution sous l'algorithme NSGA-II avec les paramètres de réglage suivants :

Nom du paramètre		Valeur
Taille de la population		86 Individus
Probabilité de croisement		0,80
Probabilité de mutation		0,10
Type de Codage chromosome		Binaire de taille 88 bits
Nombre des générations	Cas a	10 générations
	Cas b	50 générations

Tableau 6.6 – Paramètres de l'algorithme NSGA-II appliqué au filtre réjecteur

La figure suivante montre le front de Pareto de premier ordre en 2D relatif à l'optimisation de deux objectifs (1) le gain de filtre et (2) la bande passante de filtre réjecteur. Le front (a) a été généré avec les paramètres de réglage de l'algorithme NSGA-II fixés dans le Tableau 6.6 avec un nombre de génération égale à 10 et le front (b) a été généré pour les mêmes paramètres de réglage de l'algorithme NSGA-II que dans le cas (a) mais avec un nombre de générations égale à 50.



(a) **(b)**

Figure 6.34 – Front de Pareto de premier ordre

La Figure 6.34b (cas de 50 générations) montre bien une convergence rapide (par rapport à la Figure a, pour 10 générations) vers l'optimal de Pareto quand on monte en nombre de génération.

6.6 Conclusion

La plateforme avec le FPAA d'Anadigm permet d'optimiser des circuits avec l'algorithme simple et multiobjectifs. Cette plateforme présente l'avantage de réaliser des circuits de haut niveau. La vitesse de fonctionnement s'adapte au circuit à réaliser. L'ensemble de ces avantages montre le potentiel de cette plateforme. Néanmoins, des détails doivent être résolus, comme la mesure de critères adéquats pour l'algorithme multiobjectifs et la notation des composants telle que une *netlist* pour faire évoluer la topologie des circuits.

Pour l'instant cette plateforme reste une introduction à l'exploration de circuits analogiques. Il est prévu d'optimiser un filtre réjecteur par rapport à son gain et sa bande

passante. Ces deux paramètres étant contradictoires, l'algorithme génétique multiobjectifs devrait pouvoir fonctionner et trouver un compromis.

Comme perspective à ce travail, cette plateforme devrait pouvoir optimiser des circuits complexes par rapport aux ressources, aux performances comme le bruit, la consommation, le gain et la bande passante, etc.

CHAPITRE 7

CONCLUSION ET PERSPECTIVES

7.1 Conclusion

Les architectures SOC ont ouvert un large champ de recherche tant au niveau architectural qu'applicatif. La maîtrise de ces architectures est un processus complexe qui ne pourra se faire qu'avec l'apparition d'outils d'exploration et de synthèse travaillant à tous les niveaux de conception.

Nous avons présenté dans ce document une méthode d'exploration de l'espace de solutions des architectures pour mieux choisir les valeurs des différents paramètres permettant d'accéder à un maximum de performance en terme de nombre de cycles d'exécution de l'application toute en réduisant la consommation d'énergie avec un minimum de ressources allouées. Cette méthodologie d'exploration et d'optimisation architecturale est basée sur des algorithmes génétiques multiobjectifs.

Nous avons dans ce document répondu à la problématique présentée en introduction, à savoir s'il était possible de définir une méthode d'exploration des architectures sans passer par des simulations. La méthode que nous proposons s'appuie essentiellement sur l'évaluation des performances à travers des exécutions dans des conditions réelles pour une application donnée. En effet, ce choix d'optimisation multiobjectifs est essentiel afin de concevoir une architecture efficace tant du point de vue des performances temporelles que de la consommation de l'énergie et ressources allouées. Cette analyse a été rendue possible grâce à des études que nous avons menées sur les algorithmes génétiques d'optimisation multiobjectifs et leurs adaptations dans ce domaine d'exploration micro-architecturale.

Nous avons montré dans ce document que notre méthode pouvait être utilisée pour des applications différentes, tant au niveau du domaine d'applications que de leur complexité. Le flot est rapide à mettre en œuvre mais sa vitesse d'exploration dépend bien évidemment des performances du matériel informatique utilisé, de la taille de l'espace architectural à explorer et de la complexité des applications.

Les explorations architecturales que nous avons menées jusqu'à présent sont très riches en informations. Le résultat fourni par cette méthodologie est un ensemble de fronts de Pareto qui nous permet de choisir la bonne configuration, pour une application cible, capable d'attendre les objectifs ciblés en terme de performance temporelle, consommation d'énergie et ressources nécessaires.

Aujourd'hui, avec la complexité des circuits et des applications à mettre en œuvre, les experts ont besoin de ces explorations automatiques pour parvenir à un bon choix de paramétrage de leurs applications. Comme nous l'avons montré dans ce travail, les algorithmes génétiques multiobjectifs sont bien adaptés à ce type d'exploration automatique.

Nos efforts de recherche sur les propriétés théoriques de convergence montrent que des efforts restent à faire en la matière et méritent probablement une thèse dédiée.

7.2 Perspectives

Nous avons développé une méthodologie pour l'adaptation d'une application sur une architecture, dans ce travail nous n'avons pas abordé la question de l'optimisation de plusieurs applications sur une même architecture.

Dans la méthodologie proposée les paramètres ainsi que les intervalles de valeurs de ces paramètres sont fixés a priori alors que certains paramètres sont traditionnellement prédisposés de par leur impact reconnu sur les critères de performance-consommation d'énergie et de surface – d'autres moins évidents, en particulier dans un SOC très complexe devrait pouvoir émerger comme des candidats naturels par la pratique. Il serait donc souhaitable de développer des techniques dans ce sens. Des travaux au sein du laboratoire sont déjà commencés dans cette direction.

Ce travail est basé sur une technique unique d'optimisation qui n'est pas corrélée avec les caractéristiques de complexité du problème. Intuitivement il serait souhaitable d'utiliser des techniques hybrides permettant une meilleure adéquation au problème à traiter soit a priori de manière statique soit de manière dynamique et adaptative.

Il est aussi clairement indispensable de compléter le travail sur les SOC mixtes en se basant sur les nouveaux composants annoncés contenant du FPAA et du FPGA.

Comme perspective à cette méthodologie d'exploration, nous citons les applications multiprocesseurs. C'est le cas d'une thèse en cours au sein de notre laboratoire.

La particularité de la nouvelle génération de FPGA (famille FPGA Virtex-II et Pro) est d'accepter la reconfiguration dynamique partielle. C'est-à-dire qu'en cours d'exécution du FPGA, nous pouvons reconfigurer une ou plusieurs zone du FPGA sans devoir reconfigurer entièrement le FPGA. Cette particularité peut s'avérer intéressante pour une « Plateforme de Prototypage SOC ». Ceci permet un gain de temps au niveau de la phase de synthèse du design et permet de re-synthétiser et de configurer le FPGA avec uniquement la partie concernée.

De nombreuses autres problématiques passionnantes existent qui pourraient faire émerger de nombreux thèmes de recherche.

PUBLICATIONS

Embedded Processors Optimization with Hardware in the Loop

K.Ghali, Student, IEEE and O.Hammami, Member, IEEE
ENSTA, 32 Bvd. Victor 75739 Paris Cedex FRANCE
(ghali, hammami)@ieee.org
Publication: 2004 IEEE International Symposium on Industrial Electronics
May 4-7, 2004, Palais des Congres Expositions, Ajaccio, France
Abstract— The design of an embedded microprocessor for a given workload is a tremendous task by itself due to the numerous parameters involved and the ranges of their possible values. If power consumption and area are also to be considered then the problem is even more complicated and requires a suitable framework and methodology for exploring the vast multidimensional space for such a problem. In this paper we propose such a framework based on direct execution on FPGA boards.

A Methodology for the Design of Configurable Computing Machines for Software Radio Handsets

Ghali K., Hammami O., Hermann I., Ecole Nationale Supérieure de Techniques Avancées, Paris- France
Publication: The 46th IEEE International Midwest Symposium On Circuits and Systems
December 27th – 30th, 2003,
Cairo – Egypt
Abstract— The implementation of software radio on configurable computing machines is a natural approach. However, although general computing configurable platforms are readily available on the market software CAD tools are from being able to optimize future software radio systems. State of the art tools for platform based FPGA such as Xilinx Virtex-II Pro tools start from a genetic platform defined by one or several processors connected through various busses and peripherals and the user is expected to tune this genetic platform through simulation runs of the target applications.

Embedded Processor Characteristics Specification Through Multiobjective Evolutionary Algorithms

Ghali K., Hammami O., Ecole Nationale Supérieure de Techniques Avancées, Paris, France
Publication: 2003 IEEE International Symposium on Industrial Electronics (ISIE'03),
Vol. 2, pp. 907–912, IEEE, *June 9-11, 2003, Rio de Janeiro, Brazil*
Rio Othon Palace Hotel
Abstract— The design of a SuperScalar microprocessor for a given workload is a tremendous task by itself due to the numerous parameters involved and the ranges of their possible values. If power consumption and area are also to be considered then the problem is even more complicated and requires a suitable framework and methodology for exploring the vast multidimensional space for such a problem. In this paper we propose such a framework based on multi-objective evolutionary algorithms and demonstrate its use on a significant size example.

WCDMA multiprocessor on chip: design methodology using soft IP cores

Ghali, K., Hammami, O., Ecole Nationale Supérieure de Techniques Avancées

Publication: Proc. SPIE Vol. 4911, p. 120-128, Wireless and Mobile Communications II, Hequan Wu; Chih-Lin I; Jari Vaario; Eds.
Publication Date: 8/2002
Shanghai, China, Oct. 2002.
Abstract— The implementation of the physical layer of W-CDMA on embedded devices requires optimizing the resources required due to the limited space and energy allowed. Although general purpose processors will eventually be embedded they are still lacking performance and more importantly they are not tailored to the computation requirements. We propose ion this paper a methodology based on multiobjective genetic algorithms to tailor soft IP processor cores for the purpose of embedding W-CDMA.

Evaluation des Besoins en calcul pour la Conception SOC pour Mobiles 3G

K. Ghali O. Hammami, ENSTA 32 Bvd Victor 75739 Paris cedex
{ghali,hammami}@ensta.fr
Publication : SOC'2001 : séminaire sur les objets communicants
France Télécom R&D - ZIRST – Meylan, 17 & 18 octobre 2001
Meylan, France, Oct. 2001
Résume— Cette étude propose une méthodologie de conception de systèmes sur puce orientés télécommunications 3G, exploitant l'existence sur le marché de cœurs de processeurs (*Soft IP Core*) pouvant être configurés au niveau micro-architectural. Cette approche maximise la réutilisation et donc favorise la réduction de coût de tout en conservant le contrôle sur la partie intégratrice à travers l'association de plusieurs cœurs pour former un multiprocesseur sur puce. La méthodologie adapte de manière automatique en fonction de l'application les ressources au niveau processeur, ainsi que le nombre des processeurs. Une évaluation de la méthodologie a été effectue par simulation en prenant comme cas d'étude, la transmission montante W-CDMA, de côté terminal.
Mots clés : Conception, méthodologie, multi-processeur, processeur, SOC, W-CDMA.

RÉFÉRENCES

[1] R. Wilson, "*Is SOC really different?*", EE Times, November 08, 1999. URL: http://www.eet.com/article/showArticle.jhtml?articleId=18303185

[2] International Technology Roadmap For Semiconductors, "*Design*", edition 2003.

[3] B. Laurent, *''Conception des blocs réutilisables: Réflexion sur la méthodologie''* Thèse de l'Institut National Polytechnique de Grenoble, Juin 1999.

[4] Theodore W. Manikas, James T. Cain, *"Genetic algorithms and simulated annealing algorithm*", Technical Report 96-101, May 1996

[5] William E. Hart, *"A theoretical Comparison of Evolutionary Algorithms and Simulated Annealing"*, Evolutionary Programming V: Proc. of the Fifth Annual Conf. on Evolutionary Programming, 1996.

[6] S. Kirkpatrick, C. D. Gelatt, Jr., M.P. Vecchi, *"optimization by simulated annealing"*, Science, Number 4598, May 1983.

[7] F. Glover, M. Laguna, *"Tabu Search"*, Kluwer Academic Publishers, Boston, ISBN 0-7923-9965-X, July 1997.

[8] A. Hertz, E. Taillard, D. de Werra, *"A Tutorial On Tabu Search"*, Proc. of Giornate di Lavoro AIRO'95 (Enterprise Systems: Management of Technological and Organizational Changes), 1992.

[9] Glover F., *"Tabu Search, Part I"*, ORSA Journal on Computing 1, 1989.

[10] Glover F., *"Tabu Search, Part II"*, ORSA Journal on Computing 2, 1990.

[11] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, *"A fast and elitist multiobjective genetic algorithm: NSGA-II"*, IEEE Trans. on Evolutionary Computation, April 2002.

[12] Günter Rudolph, "*Convergence Analysis of Canonical Genetic Algorithms*", IEEE Transactions on Neural Networks, 1994.

[13] E. Seneta,*"Non-negative Matrices and Markov Chain*", 2nd edition, New York: Springer, 1981.

[14] Joshua D. Knowles and David W. Corne, *"Approximating the nondominated front using the Pareto archived evolution strategy*", Evolutionary Computation Journal, 8(2):149–172, 2000.

[15] David W. Corne, Joshua D. Knowles, Martin J. Oates, *"The Pareto Envelope-based Selection Algorithm for Multiobjective Optimization*", In Proceedings of the Sixth International Conference on Parallel Problem Solving from Nature (PPSN VI), pp. 839-848, Springer, Berlin, 2000.

[16] E. Zitzler and L. Thiele, *"Multiobjective evolutionary algorithms: A comparative case study and the strength Pareto approach*", IEEE Transactions on Evolutionary Computation, 3(4):257–271, 1999.

[17] E. Zitzler, M. Laumanns, L. Thiele, "*SPEA2: Improving the Strength Pareto Evolutionary Algorithm*", Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH) Zurich, TIK-Report 103, May 2001.

[18] J. Horn, N. Nafpliotis, D. E. Goldberg, "*A Niched Pareto Genetic Algorithm for Multiobjective Optimization*", IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence, ICEC '94, 1994.

[19] K. Deb, S. Agrawal, A. Pratap, T. Meyarivan. *"A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II*", In Proceedings of the Parallel Problem Solving from Nature VI (PPSN-VI), 2000.

[20] M. Laumanns, L. Thiele, K. Deb, E. Zitzler, "*On the Convergence and Diversity-Preservation Properties of Multi-Objective Evolutionary Algorithms*", TIK-Report No.

108, Institut f¨ur Technische Informatik und Kommunikationsnetze, ETH Zürich, June 2001.

[21] G. Rudolph, *"On a multi-objective evolutionary algorithm and its convergence to the Pareto set"*, Evolutionary Computation Proceedings, IEEE World Congress on Computational Intelligence, 1998.

[22] C. H. Papadimitriou, M. Yannakakis, *"The complexity of tradeoffs, and optimal access of web sources"*, In Proceedings of the 41st Annual Symposium on Foundations of Computer Science FOCS'00, 2000.

[23] G. Rudolph, A. Agapie, *"Convergence properties of some multi-objective evolutionary algorithms"*, In Proceedings of the 2000 Congress on Evolutionary Computation (CEC2000), 2000.

[24] D. Patterson, J. Hennessy, "*Organisation et conception des ordinateurs*", Edition DUNOD, Paris 1994.

[25] D. Burger, T. Austin, *"The SimpleScalar Tool Set, Version 2"*, University of Wisconin-Madison Computer Sciences Department Technical Rapport #1342, June 1997.

[26] S. Agarwal, E. Chan, Ben Liblit, C.J. Lin, *"Processor Characteristic Selection for Embedded Applications via Genetic Algorithms"*, University Of California, Berkeley, December 1998.

[27] J. Gaisler, *"The LEON-2 Processor User's Manual"*, Gaisler Research, Version 1.0.10, January 2003.

[28] J. Gaisler, *"The LEON-2 Processor User's Manual: XST Edition"*, Gaisler Research, Version 1.0.15, May 2003.
http://www.gaisler.com

[29] J. Gaisler, *"Leon/AMBA VHDL model description"*, European space agency, leon-2.2, November 2000.

[30] SPARC International, Inc. *"The SPARC Architecture Manual V8"*, Revision SAV080SI9308, 1992.

[31] A. B. Maccabe, J. Vandyke, *"A Laboratory Manual for the SPARC"*, The University of New Mexico, January 1996.

[32] D. L. Weaver, T. Germond, *"The SPARC Architecture Manual, Version 9"*, SPARC International, 2000.

[33] SunSoft, *"SPARC Assembly Language Reference Manual"*, Sun Microsystems, 1995.

[34] ARM, *"AMBA Specification (Rev 2.0)"*, May 1999.
http://www.arm.com

[35] J. Gaisler, *"The LEON/ERC32 GNU Cross-Compiler System"*, Gaisler Research, Version 1.1.1, May 2001.

[36] Gaisler Research, *"LECCS User's Manual"*, Version 1.2.2, October 2003.

[37] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, Richard B. Brown, *"MiBench: A free, commercially representative embedded benchmark suite"*, University of Michigan.

[38] *Development System Reference Guide, ISE 5 / ISE 6*

[39] *Virtex-II Platform FPGAs: Complete Data Sheet*, Xilinx Product Specification, Mars 2004.

[40] *Virtex-II™ Platform FPGAs: DC and Switching Characteristics*, Xilinx Product Specification, Mars 2004.

[41] *Virtex-II™ Platform FPGAs: Detailed Description*, Xilinx Product Specification, Mars 2004.

[42] *Virtex-II™ Platform FPGAs: Pinout Information*, Xilinx Product Specification, Mars 2004.

[43] *ADM-XRC-II PCI Mezzanine Card User Guide*, Version 1.5, ALPHA DATA, 2002.
[44] *"Core Generator guide"*, Xilinx ISE5/6
[45] *"Xilinx 5 Software Manuals"*, 2002 Xilinx.
[46] *PCI 9656 Data Book*, Version 0.90b, February 2002.
[47] *Xilinx Synthesis Technology (XST) User Guide*.
[48] 3GPP Technical Specification 25.212 version 4.0.0, *"Multiplexing and channel coding (FDD)"*, Release 4, 2000-12.
[49] 3GPP Technical Specification 25.302 version 4.0.0, *"Services provided by the Physical Layer"*, Release 4, 2001-03.
[50] 3GPP Technical Specification 25.211 version 4.0.0, *"Physical channels and mapping of transport channels onto physical channels (FDD)"*, Release 4, 2001-03.
[51] 3GPP Technical Specification 25.213 version 3.1.1, *"Spreading and modulation (FDD)"*, Release 1999, 2000-01.
[52] J. Hernandez, R. Joly, *"Résaux"*, Troisième édition, InterEditions, 1997.
[53] Harri Holma et Antti Toskala, *"UMTS Les réseaux mobiles de troisième génération"*, Edition OEM, 2000.
[54] W.E. Ryan, *"A Turbo Code Tutorial"*, New Mexico State University, Box 30001 Dept. 3-O, Las Cruces, NM 88003.
[55] *"Field Programmable Analog Arrays - User Manual AN121E04/AN221E04"*, Anadigm, 2002.
[56] *"Anadigmvortex AN221D04 Evaluation Board User Manual"*, 2003.
[57] Y. Hervé, *"VHDL-AMS, applications et enjeux industriels"*, DUNOD 2002.
[58] R.A. Rutenbar, G.G.E. Gielen, B.A. Antao, *"Computer Aided Design of Analog Integrated Circuits and Systems"*, IEEE Press 2002.
[59] S. Ganesan, *"Synthesis of mixed-signal systems based on rapid prototyping"*, Thesis, University of Cincinnati, Avril 2001.
[60] R.J. Baker, "CMOS *Mixed-Signal Circuit Design"*, IEEE Press 2002. http://www.synopsys.com/products/mixedsignal/hspice/circuit_explorer.html
[61] C.C. Santini, J.F.M Amaral, M.A.C Pacheco, M.M Vellasco, M.H Szwarcman, *"Evolutionary analog circuit design on a programmable analog multiplexer array"*, IEEE International Conference on Field-Programmable Technology (FPT'02), Dec. 2002.
[62] C.C. Santini, R. Zebulumi, M.A.C. Pacheco, M.R. Vellasco, M.H Szwarcman, *"Evolution of analog circuits on a programmable analog multiplexer array"*, Aerospace Conference, IEEE Proceedings, pp. 2301 -2308, March 2001.
[63] A. Stoica, D. Keymeulen, R. Zebulum, A. Thakoor, T. Daud, G. Klimeck, Y. Jin, R. Tawel, V. Duong, *"Evolution of analog circuits on field programmable transistor arrays"*, Evolvable Hardware, pp. 99 -108, July 2000.
[64] P. Andersen, *"Evolvable Hardware: Artificial Evolution of Hardware Circuits in simulation and reality"*, Master's Thesis, University of Aarhus, May 1998.
[65] www.latticesemi.com
[66] www.zetex.com
[67] J. Langeheine, J. Becker, S. Folling, K. Meier, J. Schemmel, *"A CMOS FPTA Chip for intrinsic Hardware Evolution of analog electronic Circuits"*, The Third NASA/DoD Workshop on Evolvable Hardware, July 2001.
[68] R.E. Smith, D.E. Golberg, J.A. Earickson, *"SGA-C: A C language Implementation of a Simple Genetic Algorithm"*, TCGA Report No. 9862, May 1991
[69] *Understanding the Anadigm Boot Kernel(ABK).*
[70] A. Stoica, R. Zebulum, D. Keymeulen, R. Tawel, T. Daud, A. Thakoor, *"Reconfigurable VLSI Architectures for Evolvable Hardware: From Experimental*

Field Programmable Transistor Arrays to Evolution-Oriented Chips", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, February 2001.

[71] G. Cullmann, *"Initiation aux chaînes de Markov, Méthodes et applications"*, Masson 1975.

[72] F. Baccelli, P. Bremaud, *"Chaînes de Markov à temps discret et applications"*, ENSTA, Edition 1995.

[73] N. Chabini1, E.M. Aboulhamid, Y. Savaria, *"Determining Schedules for Reducing Power Consumption Using Multiple Supply Voltages"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2002.

[74] W. Ye, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, *"The Design and Use of SimplePower: A CycleAccurate Energy Estimation Tool"*, DAC 2000, California.

[75] M. J. Irwin, M. Kandemir, N. Vijaykrishnan, *"SimplePower: A Cycle-Accurate Energy Simulator"*, University Park.
http://www.cse.psu.edu/~mdl/

[76] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, W. Ye, *"EnergyDriven Integrated HardwareSoftware Optimizations Using SimplePower"*, ISCA, 2000.

[77] T. Simunic, L. Benini, G. De Micheli, *"Cycle-Accurate Simulation of Energy Consumption in Embedded Systems"*, in Proceedings of DAC'99, New Orleans, Louisiana, USA, 1999.

[78] S.J.E. Wilton, N.P. Jouppi, *"CACTI: An Enhanced Cache Access and Cycle Time Model"*, IEEE Journal of Solid State Circuits 1996.

[79] P. Shivakumar, N.P. Jouppi, *"CACTI 3.0: An Integrated Cache Timing, Power, and Area Model"*, Compaq Western Research Laboratory, Research Report 2001.

[80] M.J. Flynn, S.F. Oberman, "Advanced Computer Arithmetic Design", John Wiley & Sons, New York, 2001.

[81] *FUPA - Floating Point Unit Area*
http://arith.stanford.edu/tools/fupa.html

[82] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, *"A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II"*, KanGAL Report No. 200001, 2002.

[83] M. Ehrgott, X. Gandibleux, *"Approximative Solution Methods for Multiobjective Combinatorial Optimization"*, TOP, Volume 12, pp. 1-88, June 2004.

[84] M. Ehrgott, X. Gandibleux, *"An Annotated Bibliography of Multiobjective Combinatorial Optimization"*, Universitat Kaiserslautern, Report Nr. 62, 2000.

[85] L. Charest, E.M. Aboulhamid, A. Tsikhanovich, *"Designing with SystemC: Multi-Paradigm Modeling and Simulation Performance Evaluation"*, Proceedings of The 11th Annual International HDL Conference, San Jose, CA, March 11-12, 2002, pp. 33-45.

[86] L. Charest, E.M. Aboulhamid, C. Pilkington, P. Paulin, *"SystemC Performance Evaluation Using A Pipelined DLX Multiprocessor"*, Proceedings of Design and Test in Europe Designers' Forum, Paris, March 4-8, 2002, pp. 8-12.

[87] M. Ehrgott, X. Gandibleux, *"A survey and annotated bibliography of Multiobjective combinatorial optimization"*, OR Spektrum, 22:425-460, 2000.

[88] Y. Yuan, C.H. Chan, K.F. Man, R. Mittra, *"A genetic algorithm approach to FSS filter design"*, Antennas and Propagation Society International Symposium, July 2001.

[89] R. Zebulum, A. Stoica, D. Keymeulen, *"Experiments on the evolution of digital to analog converters"*, Aerospace Conference, IEEE Proceedings, March 2001.

[90] N. Paulino, J. Goes, A. Steiger-Garcao, *"Design methodology for optimization of analog building blocks using genetic algorithms"*, Circuits and Systems, ISCAS 2001, May 2001.

[91] R. Costa, N. Paulino, J. Goes, R. Tavares, A.V. Steiger-Garcao, , *"A general-purpose kernel based on genetic algorithms for optimization of complex analog circuits*", Circuits and Systems, MWSCAS 2001, Aug. 2001.

[92] S. Papadopoulos, R.J. Mack, R.E. Massara, *"A hybrid genetic algorithm method for optimizing analog circuits*", Proceedings of the 43rd IEEE Midwest Symposium, Aug. 2000.

[93] Y.T.A. Khalifa, *"Design centering of analog circuit's component values using parallel genetic algorithms*", ICECS 2000. The 7th IEEE International Conference, Dec. 2000.

[94] R. Zebulum, H. Sinohara, M. Vellasco, C. Santini, M. Pacheco, M. Szwarcman, *"A reconfigurable platform for the automatic synthesis of analog circuits*", The Second NASA/DoD, Workshop, July 2000.

[95] N.R. Dhanwada, A. Nunez-Aldana, R. Vemuri, *"Automatic constraint transformation with integrated parameter space exploration in analog system synthesis*", Proceedings of the ASP-DAC '99, Jan. 1999.

[96] N.R. Dhanwada, A. Nunez-Aldana, R. Vemuri, "*Hierarchical constraint transformation using directed interval search for analog system synthesis*", Design Automation and Test in Europe Conference and Exhibition, March 1999.

[97] Y. Collette, P. Siarry, "*Optimisation multiobjectif*", éditions Eyrolles 2002.

[98] M. Wall, *"Galib: A C++ Library of Genetic Algorithm Components"*, Massachusetts Institute of Technology, Version 2.4, August 1996.

[99] C. Zhen; G. Lihui, *"Application of the genetic algorithm in modeling RF on-chip inductors"*, Microwave Theory and Techniques, IEEE Transactions on , Volume: 51, Feb 2003.

[100] M. Taherzadeh-Sani, R. Lotfi, H. Zare-Hoseini, O. Shoaei, *"Design optimization of analog integrated circuits using simulation-based genetic algorithm*", Signals, Circuits and Systems, SCS 2003, July 2003.

[101] M. Murakawa, T. Adachi, Y. Niino, Y. Kasai, E. Takahashi, K. Takasuka, T. Higuchi, "*An AI-calibrated IF filter: a yield enhancement method with area and power dissipation reductions"*, IEEE Journal of Solid-State Circuits, March 2003.

[102] T. Givargis, F. Vahid, J. Henkel, *"System-Level Exploration for Pareto-Optimal Configurations in Parameterized System-on-a-Chip"*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 10, No. 4, Août 2002.

GLOSSAIRE

3GPP	3rd Generation Partnership Project
A	
AAA	Adéquation Algorithme Architecture
ARIB	Association for Radio Industries and Businesses
ARQ	Automatic Repeat Request
ATM	Asynchronous Transfer Mode
AWGN	Additive White Gaussian Noise
ASIC	Application Specific Integrated Circuit
ALU	Arithmetic Logic Unite
AE	Algorithmes Evolutionnaires
B	
BCH	Broadcast Channel
BER	Bit Error Rate
BoD	Bandwith on Demand
BPSK	Binary Phase Shift Keying
C	
C/I	Carrier-to-Interference ratio
CCCH	Common Control Channel
CDMA	Code Division Multiple Access
CRC	Cyclic Redundancy Check
CLB	Configurable Logic Block
CISC	Complex Instruction Set Computer
D	
DCH	Dedicated Channel
DECT	Digital Enhanced Cordless Telephone
DPCCH	Dedicated Physical Control Channel
DPCH	Dedicated Physical Channel
DPDCH	Dedicated Physical Data Channel
DL	Downlink (Forward link)
DS-CDMA	Direct Sequence Code Division Multiple Access
DSCH	Downlink Shared Channel
DSP	Digital Signal Processor
DDR	Double Data Rate
DRAM	Dynamic RAM
E	
EDGE	Enhanced Data rates for GSM Evolution
ETSI	European Telecommunications Standards Institute
EEPROM	Electronically Erasable Programmable Read-Only Memory
EDK	Embedded Development Kit
F	
FACH	Forward Access Channel
FDD	Frequency Division Duplex
FEC	Forward Error Correction
FER	Frame Error Rate
FPLD	Field Programmable Logic Device
FPGA	Field Programmable Gate Array
FPAA	Field Programmable Analog Array

FPTA	Field Programmable Transistor Array
FUPA	Floating Point Unit Area
FFT	Fast Fourier Transformation

G

GPRS	General Packet Radio System
GPS	Global Positioning System
GSM	Global System for Mobile communications
PLD	Programmable Logic Devices
GPP	General Purpose Processor
GA	Genetic Algorithm

H

HDL	Hardware Description Languages

I

IMT-2000	International Mobile Telephony 2000
IP	Internet Protocol
ITU	International Telecommunications Union
IP	Intellectual Property
IOB	Input Output Bloc
ISE	Integrated Software Environment

L

LUT	Look-Up Tables
LECCS	Leon/ERC32 GNU cross-compiler system

M

MAC	Multiple Access Interference
MAI	Medium Access Interference
ME	Mobile Equipment
MSC	Mobile Switching Centre
MIPS	Microprocessor without Interlocked Pipeline Stages

N

NSGA	Non-dominated Sorting Genetic Algorithm
NSGA-II	Elitist Non-Dominated Sorting Genetic Algorithm
NGO	Native Generic Object
NGD	Native Generic Database

O

OVSF	Orthogonal Variable Spreading Factor
OS	Operating System

P

PCH	Paging Channel
PLMN	Public Land Mobile Network
PAES	Pareto-Archived Evolution Strategy
PCI	Peripheral Component Interconnect Bus
PAR	Place And Route
PAMA	Programmable Analog Multiplexer Array

Q

QoS	Quality of Service
QPSK	Quadrature Phase Shift Keying

R

RACH	Random Access Channel
RNC	Radio Network Controller

RNIS	Réseau Numérique à Intégration de Services
RRC	Radio Resource Control
RAM	Random Access Memory
ROM	Read Only Memory
RISC	Reduced Instruction Set Computer

S

SAP	Service Access Point
SCH	Synchronisation Channel
SCMS	Single Chip Multiprocessor Simulator
SIR	Signal to Interference Ratio
SS7	Signalling System #7
SynDEx	Synchronised Distributed Executive
SRAM	Static Random Access Memory
SOC	System on Chip
SoPC	System on Programmable Chip
SRAM	Static RAM
SPEA	Strength Pareto Evolutionary Algorithm
SPARC	Scalable Processor ARChitecture

T

TCP	Transport Control Protocol
TDD	Time Division Duplex
TDMA	Time Division Multiple Access
TF	Transport Format
TFCI	Transport Format Combination Indicator
TFCS	Transport Format Combination Set
TFI	Transport Format Indicator
TMSI	Temporary Mobile Subscriber Identity
TPC	Transmission Power Control
TLB	Translation Lookaside Buffer
TrCH	Transport Channel
TTI	Transmission Time Interval
TX	Transmit
TTM	Time-To-Market

U

UDP	User Datagram Protocol
UE	User Equipment
UMTS	Universal Mobile Telecommunications System
UL	UpLink (Reverse link)
UTRA	UMTS Terrestrial Radio Access
UCF	User Constraints File

V

VOD	Video On Demand
VEGA	Vector Evaluated Genetic algorithm
VLIW	Very Long Instruction Word

W

WCDMA	Wideband Code Division Multiple Access
WTDMA	Wideband TDMA

X

XST	Xilinx Synthesis Tool

ANNEXES

ANNEXE A : Statistique produit par le simulateur sim-outorder

```
sim-outorder: SimpleScalar/PISA Tool Set version 4.0 of December, 2001.
Copyright (C) 2000-2001 by The Regents of The University of Michigan.
Copyright (C) 1994-2001 by Todd M. Austin, Ph.D. and SimpleScalar, LLC.
This version of SimpleScalar is licensed for academic non-commercial use only.

sim: command line: sim-outorder -config
/auto/uei/ghali/moea/prj_moea_kdev/prj_moea_kdev/param_tmp/param-0.par turbo.ss -n

sim: simulation started @ Thu Sep 26 15:59:21 2002, options follow:

sim-outorder: This simulator implements a very detailed out-of-order issue
superscalar processor with a two-level memory system and speculative
execution support.  This simulator is a performance simulator, tracking the
latency of all pipeline operations.

# -config                     # load configuration from a file
# -dumpconfig                 # dump configuration to a file
# -h                    false # print help message
# -v                    false # verbose operation
# -vr                   false # verbose registers operation
-trigger:inst               0 # trigger instruction for verbose operation
# -d                    false # enable debug message
# -i                    false # start in Dlite debugger
-seed                       1 # random number generator seed (0 for timer seed)
# -q                    false # initialize and terminate immediately
# -chkpt                <null> # restore EIO trace execution from <fname>
# -redir:sim            <null> # redirect simulator output to file (non-interactive
only)
# -redir:prog           <null> # redirect simulated program output to file
-nice                       0 # simulator scheduling priority
-max:inst                   0 # maximum number of inst's to execute
-fastfwd                    0 # number of insts skipped before timing starts
# -ptrace               <null> # generate pipetrace, i.e., <fname|stdout|stderr>
<range>
-fetch:ifqsize              1 # instruction fetch queue size (in insts)
-fetch:mplat                3 # extra branch mis-prediction latency
-fetch:speed                1 # speed of front-end of machine relative to execution
core
-fetch:mf_compat        false # optimistic misfetch recovery
-bpred                  bimod # branch predictor type
{nottaken|taken|perfect|bimod|2lev|comb}
-bpred:bimod     2048 # bimodal predictor config (<table size>)
-bpred:2lev      1 1024 8 0 # 2-level predictor config (<l1size> <l2size>
<hist_size> <xor>)
-bpred:comb      1024 # combining predictor config (<meta_table_size>)
-bpred:ras                  8 # return address stack size (0 for no return stack)
-bpred:btb       512 4 # BTB config (<num_sets> <associativity>)
# -bpred:spec_update       <null> # speculative predictors update in {ID|WB}
(default non-spec)
-decode:width               1 # instruction decode B/W (insts/cycle)
-issue:width                2 # instruction issue B/W (insts/cycle)
-issue:inorder          false # run pipeline with in-order issue
-issue:wrongpath         true # issue instructions down wrong execution paths
-commit:width               4 # instruction commit B/W (insts/cycle)
-ruu:size                 64 # register update unit (RUU) size
-lsq:size                 16 # load/store queue (LSQ) size
-lsq:perfect            false # perfect memory disambiguation
-cache:dl1       dl1:64:8:1:l # l1 data cache config, i.e., {<config>|none}
```

```
-cache:dl1lat               1 # l1 data cache hit latency (in cycles)
-cache:dl2       ul2:1024:64:4:l # l2 data cache config, i.e., {<config>|none}
-cache:dl2lat               1 # l2 data cache hit latency (in cycles)
-cache:il1       il1:64:32:2:l # l1 inst cache config, i.e.,
{<config>|dl1|dl2|none}
-cache:il1lat               1 # l1 instruction cache hit latency (in cycles)
-cache:il2                dl2 # l2 instruction cache config, i.e.,
{<config>|dl2|none}
-cache:il2lat               1 # l2 instruction cache hit latency (in cycles)
-cache:flush            false # flush caches on system calls
-cache:icompress        false # convert 64-bit inst addresses to 32-bit inst
equivalents
-mem:lat         18 2 # memory access latency (<first_chunk> <inter_chunk>)
-mem:width                  8 # memory access bus width (in bytes)
-tlb:itlb        itlb:16:4096:4:l # instruction TLB config, i.e., {<config>|none}
-tlb:dtlb        dtlb:32:4096:4:l # data TLB config, i.e., {<config>|none}
-tlb:lat                   30 # inst/data TLB miss latency (in cycles)
-res:ialu                   7 # total number of integer ALU's available
-res:imult                  3 # total number of integer multiplier/dividers
available
-res:memport                2 # total number of memory system ports available (to
CPU)
-res:fpalu                  6 # total number of floating point ALU's available
-res:fpmult                 2 # total number of floating point multiplier/dividers
available
# -pcstat              <null> # profile stat(s) against text addr's (mult uses ok)
-bugcompat              false # operate in backward-compatible bugs mode (for
testing only)

  Pipetrace range arguments are formatted as follows:

    {{@|#}<start>}:{{@|#|+}<end>}

  Both ends of the range are optional, if neither are specified, the entire
  execution is traced.  Ranges that start with a `@' designate an address
  range to be traced, those that start with an `#' designate a cycle count
  range.  All other range values represent an instruction count range.  The
  second argument, if specified with a `+', indicates a value relative
  to the first argument, e.g., 1000:+100 == 1000:1100.  Program symbols may
  be used in all contexts.

    Examples:   -ptrace FOO.trc #0:#1000
                -ptrace BAR.trc @2000:
                -ptrace BLAH.trc :1500
                -ptrace UXXE.trc :
                -ptrace FOOBAR.trc @main:+278

  Branch predictor configuration examples for 2-level predictor:
    Configurations:   N, M, W, X
      N   # entries in first level (# of shift register(s))
      W   width of shift register(s)
      M   # entries in 2nd level (# of counters, or other FSM)
      X   (yes-1/no-0) xor history and address for 2nd level index
    Sample predictors:
      GAg     : 1, W, 2^W, 0
      GAp     : 1, W, M (M > 2^W), 0
      PAg     : N, W, 2^W, 0
      PAp     : N, W, M (M == 2^(N+W)), 0
      gshare  : 1, W, 2^W, 1
  Predictor `comb' combines a bimodal and a 2-level predictor.

  The cache config parameter <config> has the following format:

    <name>:<nsets>:<bsize>:<assoc>:<repl>

    <name>   - name of the cache being defined
    <nsets>  - number of sets in the cache
    <bsize>  - block size of the cache
```

```
    <assoc>  - associativity of the cache
    <repl>   - block replacement strategy, 'l'-LRU, 'f'-FIFO, 'r'-random

    Examples:   -cache:dl1 dl1:4096:32:1:l
                -dtlb dtlb:128:4096:32:r

  Cache levels can be unified by pointing a level of the instruction cache
  hierarchy at the data cache hiearchy using the "dl1" and "dl2" cache
  configuration arguments.  Most sensible combinations are supported, e.g.,

    A unified l2 cache (il2 is pointed at dl2):
      -cache:il1 il1:128:64:1:l -cache:il2 dl2
      -cache:dl1 dl1:256:32:1:l -cache:dl2 ul2:1024:64:2:l

    Or, a fully unified cache hierarchy (il1 pointed at dl1):
      -cache:il1 dl1
      -cache:dl1 ul1:256:32:1:l -cache:dl2 ul2:1024:64:2:l



sim: ** starting performance simulation **

sim: ** simulation statistics **
sim_num_insn                3030266 # total number of instructions committed
sim_num_refs                1198685 # total number of loads and stores committed
sim_num_loads                928371 # total number of loads committed
sim_num_stores          270314.0000 # total number of stores committed
sim_num_branches             302903 # total number of branches committed
sim_elapsed_time                 17 # total simulation time in seconds
sim_inst_rate           178250.9412 # simulation speed (in insts/sec)
sim_total_insn              3094474 # total number of instructions executed
sim_total_refs              1231804 # total number of loads and stores executed
sim_total_loads              957359 # total number of loads executed
sim_total_stores        274445.0000 # total number of stores executed
sim_total_branches           306303 # total number of branches executed
sim_cycle                   3307306 # total simulation time in cycles
sim_IPC                      0.9162 # instructions per cycle
sim_CPI                      1.0914 # cycles per instruction
sim_exec_BW                  0.9356 # total instructions (mis-spec + committed) per
cycle
sim_IPB                     10.0041 # instruction per branch
IFQ_count                   3127507 # cumulative IFQ occupancy
IFQ_fcount                  3127507 # cumulative IFQ full count
ifq_occupancy                0.9456 # avg IFQ occupancy (insn's)
ifq_rate                     0.9356 # avg IFQ dispatch rate (insn/cycle)
ifq_latency                  1.0107 # avg IFQ occupant latency (cycle's)
ifq_full                     0.9456 # fraction of time (cycle's) IFQ was full
RUU_count                  14063554 # cumulative RUU occupancy
RUU_fcount                        0 # cumulative RUU full count
ruu_occupancy                4.2523 # avg RUU occupancy (insn's)
ruu_rate                     0.9356 # avg RUU dispatch rate (insn/cycle)
ruu_latency                  4.5447 # avg RUU occupant latency (cycle's)
ruu_full                     0.0000 # fraction of time (cycle's) RUU was full
LSQ_count                   5530422 # cumulative LSQ occupancy
LSQ_fcount                     2142 # cumulative LSQ full count
lsq_occupancy                1.6722 # avg LSQ occupancy (insn's)
lsq_rate                     0.9356 # avg LSQ dispatch rate (insn/cycle)
lsq_latency                  1.7872 # avg LSQ occupant latency (cycle's)
lsq_full                     0.0006 # fraction of time (cycle's) LSQ was full
sim_slip                    23615395 # total number of slip cycles
avg_sim_slip                 7.7932 # the average slip between issue and retirement
misfetch_count                7715 # misfetch count
misfetch_only_count           1124 # misfetch only count
recovery_count               44829 # recovery count
bpred_bimod.lookups         307502 # total number of bpred lookups
bpred_bimod.updates         302903 # total number of updates
bpred_bimod.addr_hits       264741 # total number of address-predicted hits
```

```
bpred_bimod.dir_hits         265818 # total number of direction-predicted hits
(includes addr-hits)
bpred_bimod.misses            37085 # total number of misses
bpred_bimod.jr_hits           17486 # total number of address-predicted hits for
JR's
bpred_bimod.jr_seen           17522 # total number of JR's seen
bpred_bimod.jr_non_ras_hits.PP           24 # total number of address-predicted
hits for non-RAS JR's
bpred_bimod.jr_non_ras_seen.PP           50 # total number of non-RAS JR's seen
bpred_bimod.bpred_addr_rate    0.8740 # branch address-prediction rate (i.e., addr-
hits/updates)
bpred_bimod.bpred_dir_rate    0.8776 # branch direction-prediction rate (i.e., all-
hits/updates)
bpred_bimod.bpred_jr_rate    0.9979 # JR address-prediction rate (i.e., JR addr-
hits/JRs seen)
bpred_bimod.bpred_jr_non_ras_rate.PP    0.4800 # non-RAS JR addr-pred rate (ie,
non-RAS JR hits/JRs seen)
bpred_bimod.retstack_pushes        17729 # total number of address pushed onto ret-
addr stack
bpred_bimod.retstack_pops        17742 # total number of address popped off of ret-
addr stack
bpred_bimod.used_ras.PP        17472 # total number of RAS predictions used
bpred_bimod.ras_hits.PP        17462 # total number of RAS hits
bpred_bimod.ras_rate.PP    0.9994 # RAS prediction rate (i.e., RAS hits/used RAS)
il1.accesses                3126965 # total number of accesses
il1.hits                    3054241 # total number of hits
il1.misses                    72724 # total number of misses
il1.replacements              72596 # total number of replacements
il1.writebacks                    0 # total number of writebacks
il1.invalidations                 0 # total number of invalidations
il1.miss_rate                0.0233 # miss rate (i.e., misses/ref)
il1.repl_rate                0.0232 # replacement rate (i.e., repls/ref)
il1.wb_rate                  0.0000 # writeback rate (i.e., wrbks/ref)
il1.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
dl1.accesses                1188921 # total number of accesses
dl1.hits                    1068153 # total number of hits
dl1.misses                   120768 # total number of misses
dl1.replacements             120704 # total number of replacements
dl1.writebacks                56911 # total number of writebacks
dl1.invalidations                 0 # total number of invalidations
dl1.miss_rate                0.1016 # miss rate (i.e., misses/ref)
dl1.repl_rate                0.1015 # replacement rate (i.e., repls/ref)
dl1.wb_rate                  0.0479 # writeback rate (i.e., wrbks/ref)
dl1.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
ul2.accesses                 250403 # total number of accesses
ul2.hits                     247631 # total number of hits
ul2.misses                     2772 # total number of misses
ul2.replacements                 13 # total number of replacements
ul2.writebacks                   12 # total number of writebacks
ul2.invalidations                 0 # total number of invalidations
ul2.miss_rate                0.0111 # miss rate (i.e., misses/ref)
ul2.repl_rate                0.0001 # replacement rate (i.e., repls/ref)
ul2.wb_rate                  0.0000 # writeback rate (i.e., wrbks/ref)
ul2.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
itlb.accesses               3126965 # total number of accesses
itlb.hits                   3126932 # total number of hits
itlb.misses                      33 # total number of misses
itlb.replacements                 0 # total number of replacements
itlb.writebacks                   0 # total number of writebacks
itlb.invalidations                0 # total number of invalidations
itlb.miss_rate               0.0000 # miss rate (i.e., misses/ref)
itlb.repl_rate               0.0000 # replacement rate (i.e., repls/ref)
itlb.wb_rate                 0.0000 # writeback rate (i.e., wrbks/ref)
itlb.inv_rate                0.0000 # invalidation rate (i.e., invs/ref)
dtlb.accesses               1202274 # total number of accesses
dtlb.hits                   1202248 # total number of hits
dtlb.misses                      26 # total number of misses
dtlb.replacements                 0 # total number of replacements
```

```
dtlb.writebacks                  0 # total number of writebacks
dtlb.invalidations               0 # total number of invalidations
dtlb.miss_rate              0.0000 # miss rate (i.e., misses/ref)
dtlb.repl_rate              0.0000 # replacement rate (i.e., repls/ref)
dtlb.wb_rate                0.0000 # writeback rate (i.e., wrbks/ref)
dtlb.inv_rate               0.0000 # invalidation rate (i.e., invs/ref)
sim_invalid_addrs                0 # total non-speculative bogus addresses seen
(debug var)
ld_text_base             0x00400000 # program text (code) segment base
ld_text_size                131968 # program text (code) size in bytes
ld_data_base             0x10000000 # program initialized data segment base
ld_data_size                 13156 # program init'ed `.data' and uninit'ed `.bss'
size in bytes
ld_stack_base            0x7fffc000 # program stack segment base (highest address
in stack)
ld_stack_size                16384 # program initial stack size
ld_prog_entry            0x00400140 # program entry point (initial PC)
ld_environ_base          0x7fff8000 # program environment base address address
ld_target_big_endian             0 # target executable endian-ness, non-zero if
big endian
mem.page_count                  59 # total number of pages allocated
mem.page_mem                  236k # total size of memory pages allocated
mem.ptab_misses                 63 # total first level page table misses
mem.ptab_accesses          22846608 # total page table accesses
mem.ptab_miss_rate            0.0000 # first level page table miss rate
```

ANNEXE B: Statistique produit par le simulateur sim-profile

```
sim-profile: SimpleScalar/PISA Tool Set version 4.0 of December, 2001.
Copyright (C) 2000-2001 by The Regents of The University of Michigan.
Copyright (C) 1994-2001 by Todd M. Austin, Ph.D. and SimpleScalar, LLC.
This version of SimpleScalar is licensed for academic non-commercial use only.

sim: command line: sim-profile -iclass turbo.ss

sim: simulation started @ Thu Sep 26 16:12:24 2002, options follow:

sim-profile: This simulator implements a functional simulator with
profiling support.  Run with the `-h' flag to see profiling options
available.

# -config                     # load configuration from a file
# -dumpconfig                 # dump configuration to a file
# -h                    false # print help message
# -v                    false # verbose operation
# -vr                   false # verbose registers operation
-trigger:inst               0 # trigger instruction for verbose operation
# -d                    false # enable debug message
# -i                    false # start in Dlite debugger
-seed                       1 # random number generator seed (0 for timer seed)
# -q                    false # initialize and terminate immediately
# -chkpt               <null> # restore EIO trace execution from <fname>
# -redir:sim           <null> # redirect simulator output to file (non-interactive
only)
# -redir:prog          <null> # redirect simulated program output to file
-nice                       0 # simulator scheduling priority
-max:inst                   0 # maximum number of inst's to execute
-all                  false # enable all profile options
-iclass                true # enable instruction class profiling
-iprof                false # enable instruction profiling
-brprof               false # enable branch instruction profiling
-amprof               false # enable address mode profiling
-segprof              false # enable load/store address segment profiling
-tsymprof             false # enable text symbol profiling
-taddrprof            false # enable text address profiling
-dsymprof             false # enable data symbol profiling
-internal             false # include compiler-internal symbols during symbol
profiling
```

```
# -pcstat              <null> # profile stat(s) against text addr's (mult uses ok)

sim: ** starting functional simulation **

sim: ** simulation statistics **
sim_num_insn                3109145 # total number of instructions executed
sim_num_refs                1227265 # total number of loads and stores executed
sim_elapsed_time                  1 # total simulation time in seconds
sim_inst_rate          3109145.0000 # simulation speed (in insts/sec)

sim_inst_class_prof    # instruction class profile
sim_inst_class_prof.array_size = 7
sim_inst_class_prof.bucket_size = 1
sim_inst_class_prof.count = 7
sim_inst_class_prof.total = 3109144
sim_inst_class_prof.imin = 0
sim_inst_class_prof.imax = 7
sim_inst_class_prof.average = 444163.4286
sim_inst_class_prof.std_dev = 563958.2055
sim_inst_class_prof.overflows = 0
# pdf == prob dist fn, cdf == cumulative dist fn
#          index      count    pdf
sim_inst_class_prof.start_dist
load                  950697  30.58
store                 276568   8.90
uncond branch         128662   4.14
cond branch           188173   6.05
int computation      1500962  48.28
fp computation         64005   2.06
trap                      77   0.00
sim_inst_class_prof.end_dist

ld_text_base             0x00400000 # program text (code) segment base
ld_text_size                 131968 # program text (code) size in bytes
ld_data_base             0x10000000 # program initialized data segment base
ld_data_size                 13156 # program init'ed `.data' and uninit'ed `.bss'
size in bytes
ld_stack_base            0x7fffc000 # program stack segment base (highest address
in stack)
ld_stack_size                16384 # program initial stack size
ld_prog_entry            0x00400140 # program entry point (initial PC)
ld_environ_base          0x7fff8000 # program environment base address address
ld_target_big_endian              0 # target executable endian-ness, non-zero if
big endian
mem.page_count                   59 # total number of pages allocated
mem.page_mem                   236k # total size of memory pages allocated
mem.ptab_misses                  60 # total first level page table misses
mem.ptab_accesses          15714503 # total page table accesses
mem.ptab_miss_rate       0.0000 # first level page table miss rate
```

ANNEXE C : Statistique produit par le simulateur CACTI

```
Votre Commande:  cacti 1024 8 1 0.180000 1 0 0 1

Cache Parameters:
  Number of Subbanks: 1
  Total Cache Size: 1024
  Size in bytes of Subbank: 1024
  Number of sets: 128
  Associativity: direct mapped
  Block Size (bytes): 8
  Read/Write Ports: 1
  Read Ports: 0
  Write Ports: 0
  Technology Size: 0.18um
  Vdd: 1.7V

Access Time (ns): 0.842168
Cycle Time (wave pipelined) (ns):  0.295143
```

```
Total Power all Banks (nJ): 0.305695
Total Power Without Routing (nJ): 0.305695
Total Routing Power (nJ): 0
Maximum Bank Power (nJ):  0.305695

Best Ndwl (L1): 4
Best Ndbl (L1): 1
Best Nspd (L1): 4
Best Ntwl (L1): 1
Best Ntbl (L1): 2
Best Ntspd (L1): 1
Nor inputs (data): 2
Nor inputs (tag): 2

Area Components:

Aspect Ratio Total height/width: 1.040717

Data array (cm^2): 0.001928
Data predecode (cm^2): 0.000038
Data colmux predecode (cm^2): 0.000019
Data colmux post decode (cm^2): 0.000003
Data write signal (cm^2): 0.000002

Tag array (cm^2): 0.000754
Tag predecode (cm^2): 0.000038
Tag colmux predecode (cm^2): 0.000019
Tag colmux post decode (cm^2): 0.000001
Tag output driver decode (cm^2): 0.000032
Tag output driver enable signals (cm^2): 0.000002

Percentage of data ramcells alone of total area (%): 30.899169
Percentage of tag ramcells alone of total area (%): 11.587188
Percentage of total control/routing alone of total area (%): 57.513642

Subbank Efficiency: 42.486358
Total Efficiency: 47.933782

Total area One Subbank (cm^2): 0.002838
Total area subbanked (cm^2): 0.002516

Time Components:
 data side (with Output driver) (ns): 0.777933
 tag side (with Output driver) (ns): 0.842168
 address routing delay (ns): 0
 address routing power (nJ): 0
 decode data (ns): 0.282813
             (nJ): 0.0233149
 wordline and bitline data (ns): 0.195005
            wordline power (nJ): 0.000396848
             bitline power (nJ): 0.0120757
 sense_amp_data (ns): 0.18675
                (nJ): 0.144283
 decode_tag (ns): 0.252858
            (nJ): 0.0151275
 wordline and bitline tag (ns): 0.105285
           wordline power (nJ): 0.000248346
            bitline power (nJ): 0.00608361
 sense_amp_tag (ns): 0.11475
               (nJ): 0.0886035
 compare (ns): 0.295143
         (nJ): 0.0010263
 valid signal driver (ns): 0.0741322
                     (nJ): 0.000455746
 data output driver (ns): 0.113365
                    (nJ): 0.0140793
 total_out_driver (ns): 0
                 (nJ): 0
```

```
 total data path (without output driver) (ns): 0.664568
 total tag path is dm (ns): 0.768036
```

ANNEXE D.1 : Code en C pour le test de Leon

```
#include <stdio.h>
#include <stdlib.h>
int *registre;
int *flag;
int *ptra;
int  A[50][50], B[50][50];
int i,j,k;

void main(){

	ptra = (int*) 0x40000010; /* Allocation memoire pour resultat matrice */
	flag = (int*)malloc(4*sizeof(int)); //Allocation memoire pour Flags

	registre = 0x80000064; //Initialisation registre WaitStates
	*registre = *registre & 0xfffffff1;

	flag = 0x40000000; // Flag debut boucle d'Init
	*flag = 666;        //0x29A or (10 1001 1010)b

	/*============================================================ */
	/*          initialisation Ne deux Mat A et B                */
	for ( i=0;  i<50; i++){
		for ( j=0; j<50;j++){
			*ptra=0;
			ptra++;
			if (i==j){
				A[i][j] = 100;
				B[i][j] = 222;
			}
			else{
				A[i][j] = 0;
				B[i][j] = 0;
			}
		}
	}
	/*============================================================ */
	flag=0x40000004; // Flag Fin boucle d'Init et debut calcul matrice
	*flag =777;         //0x309 or (11 0000 1001)b
	/*============================================================ */
	/*                              Calcul Matrice A et B               */

	ptra=0x40000010;  /*Zone  start  adresse  pour  sauvegarde  en  memoire  du
result matrice A * B */
	for (i=0; i<50;i++){
		for ( j=0; j<50;j++){
			for ( k=0;  k<4;k++){
				*ptra += A[i][k]*B[k][j];
			}
			ptra++;
		}
	}
	/*============================================================ */

	flag=0x40000008; // Flag Fin boucle calcul matrice
	*flag = 888;  //0x378 or (11 0111 1000)b

	flag = (int*) 0x4000000C;
	registre = (int*) 0x80000040;
	*flag = *registre;

	do
```

```
	{

	}
	while(1);
}
```

ANNEXE D.2 : Code en C de l'application exécutée par le Leon

```
#define Dim  80
#define StartMem    0x40000100

int main(){
	register int i,j,k, kk;
	register int *registre, *flag;
	register float Max;
	//int  A[Dim][Dim], B[Dim][Dim];
	register float  *A, *B, *C;

	//ptra=(int*)malloc(16*sizeof(int));//Alloc. Memoire resultat matrice
	//flag=(int*)malloc(4*sizeof(int)); //Allocation memoire pour Flags
	registre = (int*) 0x80000004; //Initialisation registre WaitStates
	*registre = ((*registre) & 0xfffffff0)|0x0000000A;
	registre = (int*) 0x80000064;
	*registre = 9;
	//flag=0x40000000; // Flag debut boucle d'Init
	//*flag =666; //0x29a
	/*============================================================ */
	/*           initialisation Ne deux Mat A et B                */
	A = (float*) StartMem;
	B = A + Dim*Dim;
	C = B + Dim*Dim;
	for ( i=0;  i<Dim; i++){
		for ( j=0; j<Dim;j++){
			C[i*Dim+j] = 0.0;
			if (i==j){
				A[i*Dim+j] = (float) 2.2;
				B[i*Dim+j] = (float) 4.4;
				}
			else{
				A[i*Dim+j] = (float) 3.3;
				B[i*Dim+j] = (float) 5.5;
			}
		}
	}

	/*============================================================ */
	//flag=0x40000004; // Flag Fin boucle d'Init et debut calcul matrice
	//*flag =777;//0x309
	/*============================================================ */
	/*                        Calcul Matrice A et B                 */
	for (i=0; i<Dim; i++){
		for ( j=0; j<Dim; j++){
			for ( k=0;  k<Dim; k++)
				C[i*Dim+j] += (float)A[i*Dim+k]*B[k*Dim+j];

		}
	}
	for (i=0; i<Dim; i++){
		for ( j=0; j<Dim; j++){
			C[i*Dim+j] = (float)((C[i*Dim+j]*C[i*Dim+j])/9.567);

		}
	}
	Max = 0.0;
	for (i=0; i<Dim; i++){
		for ( j=0; j<Dim; j++){
			if(C[i*Dim+j]<0)
				C[i*Dim+j] = -C[i*Dim+j];
			if(C[i*Dim+j]>Max)
```

```
                        Max = C[i*Dim+j];
                }
        }
        for (i=0; i<Dim; i++){
                for ( j=0; j<Dim; j++){
                        C[i*Dim+j]/= Max;
                }
        }
        for (i=0; i<Dim; i++){
                for ( j=0; j<Dim; j++){
                        for ( k=i;  k<Dim; k++)
                                for ( kk=j;  kk<Dim; kk++)
                                        if(C[i*Dim+j] > C[k*Dim+kk]){
                                                Max = C[i*Dim+j];
                                                C[i*Dim+j] = C[k*Dim+kk];
                                                C[k*Dim+kk] = Max;
                                        }
                }
        }
        /*============================================================ */
        //flag=0x40000008; // Flag Fin boucle calcul matrice
        //*flag =888;  //0x378

        flag = (int*) 0x4000000C;
        registre = (int*) 0x80000040;
        *flag = *registre;

        return 0;
}
```

ANNEXE E: Package device.vhd

```
------------------------------------------------------------------------------
--  This file is a part of the LEON VHDL model
--  Copyright (C) 1999  European Space Agency (ESA)
--
--  This library is free software; you can redistribute it and/or
--  modify it under the terms of the GNU Lesser General Public
--  License as published by the Free Software Foundation; either
--  version 2 of the License, or (at your option) any later version.
--
--  See the file COPYING.LGPL for the full details of the license.


------------------------------------------------------------------------------
-- Entity:   device
-- File:     device.vhd
-- Author:   Jiri Gaisler - Gaisler Research
-- Description:     package to select current device configuration
------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use work.target.all;

package device is

------------------------------------------------------------------------------
-- Automatically generated by tkonfig/mkdevice
------------------------------------------------------------------------------

  constant syn_config : syn_config_type := (
    targettech => gen , infer_pads => true,
    infer_ram => true, infer_regf => true, infer_rom => true,
    infer_mult => true, rftype => 2, targetclk => gen,
    clk_mul => 1, clk_div => 1, pci_dll => false, pci_sysclk => false );

  constant iu_config : iu_config_type := (
```

```
    nwindows => 8, multiplier => none, mulpipe => false,
    divider => none, mac => false, fpuen => 0, cpen => false,
    fastjump => true, icchold => true, lddelay => 1, fastdecode => true,
    rflowpow => false, watchpoints => 0, impl => 0, version => 0);

  constant fpu_config : fpu_config_type :=
    (core => meiko, interface => none, fregs => 0, version => 0);

constant cache_config: cache_config_type := (
    isets => 1, isetsize => 4, ilinesize => 8, ireplace => rnd, ilock => 0,
    dsets => 1, dsetsize => 4, dlinesize => 8, dreplace => rnd, dlock => 0,
    dsnoop => none, drfast => false, dwfast => false, dlram => false,
    dlramsize => 1, dlramaddr => 16#8F#);

  constant mmu_config : mmu_config_type := (
    enable => 0, itlbnum => 8, dtlbnum => 8, tlb_type => combinedtlb,
    tlb_rep => replruarray, tlb_diag => false );

  constant ahbrange_config  : ahbslv_addr_type :=
        (0,0,0,0,0,0,0,1,7,7,7,7,7,7,7,7);

  constant ahb_config : ahb_config_type := ( masters => 1, defmst => 0,
    split => false, testmod => false);

  constant mctrl_config : mctrl_config_type := (
    bus8en => false, bus16en => false, wendfb => false, ramsel5 => false,
    sdramen => false, sdinvclk => false);

  constant peri_config : peri_config_type := (
    cfgreg => true, ahbstat => false, wprot => false, wdog => false,
    irq2en => false, ahbram => false, ahbrambits => 11, ethen => false );

  constant debug_config : debug_config_type := ( enable => false, uart => false,
    iureg => false, fpureg => false, nohalt => false, pclow => 2,
    dsuenable => false, dsutrace => false, dsumixed => false,
    dsudpram => true, tracelines => 64);

  constant boot_config : boot_config_type := (boot => memory, ramrws => 0,
    ramwws => 0, sysclk => 33000000, baud => 19200, extbaud => false,
    pabits => 11);

  constant pci_config : pci_config_type := (
    pcicore => none , ahbmasters => 0, ahbslaves => 0,
    arbiter => false, fixpri => false, prilevels => 4, pcimasters => 4,
    vendorid => 16#0000#, deviceid => 16#0000#, subsysid => 16#0000#,
    revisionid => 16#00#, classcode =>16#000000#, pmepads => false,
    p66pad => false, pcirstall => false);

  constant irq2cfg : irq2type := irq2none;


--------------------------------------------------------------------------------
-- end of automatic configuration
--------------------------------------------------------------------------------

end;
```

ANNEXE E.1: Code source du flot de compilation code C

```
/*
================================================================================
===== */
/*                                                                  Bibliotheques
*/
/*
================================================================================
===== */
/* Bibliotheques Generiques */
#include <stdio.h>
```

```
#include <stdlib.h>
#include <windows.h>


/*
================================================================================
===== */
/*                                                Declaration  des  appels  de  fichiers
*/
/*
================================================================================
===== */
void FlotCompilationC();

/*
================================================================================
===== */
/*                                                                  Programme  principal
*/
/*
================================================================================
===== */
void main(void)
{
      printf(" Compilation code C \n LEI-ENSTA Paris \n Septembre 2003\n\n");
      FlotCompilationC();
}


/*
================================================================================
===== */
/*
================================================================================
===== */

void FlotCompilationC(void)
{
//    char carSaisie;

      printf("Phase de lancement du flot de compilation\n");
      /*  printf("Veuillez  entrer  :  NomChemin  &  NomFichier.c  puis  '*'  pour
finir\n");
      printf("Nom de fichier est :");
      do{
      carSaisie=getch();
      printf("%c",carSaisie);
      }while (carSaisie != '*');
      printf("\n");*/


      system("sparc-rtems-gcc -msoft-float -qprom rom.c -o rom.exe");
      system("mkprom  -ramsize  256  -ramcs  1  -romsize  8  -nocomp  -o  rom.prom
rom.exe");
      //system("cp rom.prom rom.prom");
      system("sparc-rtems-strip rom.prom ");
      system("sparc-rtems-objcopy --remove-section=.comment rom.prom ");
      system("sparc-rtems-objdump -s rom.prom > rom.dat ");
      system("sparc-rtems-objcopy -O binary rom.prom rom.bin");
      system("DAT2MIF rom.dat rom.mif ");
      system("cp rom.mif C:\\ADMXRC_SDK4\\apps\\CommLeonPC\\VHDL");
}
```

ANNEXE E.2 : Liste des fichiers VHDL à synthétiser : *PPOP.prj*

```
work ./source_VHDL/Leon/Packages/amba.vhd
work ./source_VHDL/Leon/Packages/target.vhd
work ./source_VHDL/Leon/Packages/device.vhd
work ./source_VHDL/Leon/Packages/config.vhd
```

```
work ./source_VHDL/Leon/Packages/sparcv8.vhd
work ./source_VHDL/Leon/Packages/mmuconfig.vhd
work ./source_VHDL/Leon/Packages/iface.vhd
work ./source_VHDL/Leon/Packages/macro.vhd
work ./source_VHDL/Leon/Packages/ambacomp.vhd
work ./source_VHDL/Leon/Packages/tech_generic.vhd
work ./source_VHDL/Leon/Packages/tech_atc18.vhd
work ./source_VHDL/Leon/Packages/tech_atc25.vhd
work ./source_VHDL/Leon/Packages/tech_atc35.vhd
work ./source_VHDL/Leon/Packages/tech_virtex.vhd
work ./source_VHDL/Leon/Packages/tech_fs90.vhd
work ./source_VHDL/Leon/Packages/multlib.vhd
work ./source_VHDL/Leon/Packages/fpulib.vhd
work ./source_VHDL/Leon/Packages/tech_proasic.vhd
work ./source_VHDL/Leon/Packages/tech_tsmc25.vhd
work ./source_VHDL/Leon/Packages/tech_umc18.vhd
work ./source_VHDL/Leon/Packages/tech_axcel.vhd
work ./source_VHDL/Leon/Packages/tech_virtex2.vhd
work ./source_VHDL/Leon/Packages/tech_map.vhd
work ./source_VHDL/Leon/Tbench/leonlib.vhd
work ./source_VHDL/Leon/Tbench/tblib.vhd
nosort
work ./source_VHDL/Leon/Modules/ahbarb.vhd
work ./source_VHDL/Leon/Modules/ahbram.vhd
work ./source_VHDL/Leon/Modules/ahbstat.vhd
work ./source_VHDL/Leon/Modules/apbmst.vhd
work ./source_VHDL/Leon/Modules/ahbmst.vhd
work ./source_VHDL/Leon/Modules/dcom_uart.vhd
work ./source_VHDL/Leon/Modules/dcom.vhd
work ./source_VHDL/Leon/Modules/dsu.vhd
work ./source_VHDL/Leon/Modules/dsu_mem.vhd
work ./source_VHDL/Leon/Modules/eth_oc.vhd
work ./source_VHDL/Leon/Modules/ioport.vhd
work ./source_VHDL/Leon/Modules/irqctrl.vhd
work ./source_VHDL/Leon/Modules/irqctrl2.vhd
work ./source_VHDL/Leon/Modules/lconf.vhd
work ./source_VHDL/Leon/Modules/sdmctrl.vhd
work ./source_VHDL/Leon/Modules/mctrl.vhd
work ./source_VHDL/Leon/Modules/pci_is.vhd
work ./source_VHDL/Leon/Modules/pci_oc.vhd
work ./source_VHDL/Leon/Modules/pci.vhd
work ./source_VHDL/Leon/Modules/pci_arb.vhd
work ./source_VHDL/Leon/Modules/acache.vhd
work ./source_VHDL/Leon/Modules/dcache.vhd
work ./source_VHDL/Leon/Modules/icache.vhd
work ./source_VHDL/Leon/Modules/cache.vhd
work ./source_VHDL/Leon/Modules/cachemem.vhd
work ./source_VHDL/Leon/Modules/meiko.vhd
work ./source_VHDL/Leon/Modules/fpu_lth.vhd
work ./source_VHDL/Leon/Modules/fpu_core.vhd
work ./source_VHDL/Leon/Modules/fp1eu.vhd
work ./source_VHDL/Leon/Modules/div.vhd
work ./source_VHDL/Leon/Modules/mul.vhd
work ./source_VHDL/Leon/Modules/iu.vhd
work ./source_VHDL/Leon/Modules/mmulrue.vhd
work ./source_VHDL/Leon/Modules/mmulru.vhd
work ./source_VHDL/Leon/Modules/mmutlbcam.vhd
work ./source_VHDL/Leon/Modules/mmutlb.vhd
work ./source_VHDL/Leon/Modules/mmutw.vhd
work ./source_VHDL/Leon/Modules/mmu.vhd
work ./source_VHDL/Leon/Modules/mmu_acache.vhd
work ./source_VHDL/Leon/Modules/mmu_dcache.vhd
work ./source_VHDL/Leon/Modules/mmu_icache.vhd
work ./source_VHDL/Leon/Modules/mmu_cache.vhd
work ./source_VHDL/Leon/Modules/proc.vhd
work ./source_VHDL/Leon/Modules/rstgen.vhd
work ./source_VHDL/Leon/Modules/timers.vhd
work ./source_VHDL/Leon/Modules/uart.vhd
```

```
work ./source_VHDL/Leon/Modules/wprot.vhd           work
./source_VHDL/Leon/Modules/mcore.vhd
work ./source_VHDL/Leon/Modules/leon.vhd            work
./source_VHDL/Plateforme/plxdssm.vhd
work ./source_VHDL/Plateforme/ControleurBUS.vhd     work
./source_VHDL/Plateforme/PlateformeEmulation.vhd
```

ANNEXE E.3 : Exemple d'un script de synthèse pour outil XST

```
# Synthesis script for XST tool (Xilinx) of  platform emulation  of Leon
# Ivan Hermann
# 17/10/2003
# ENSTA Paris
##### Modifs Parameters :
#####   -opt_mode            : area
#####   -opt_level           : 1
#####   -slice_utilization_ratio : 80
#####   -slice_utilization_maxmargin 1
run
#Global options
-ifn C:/PlateformeSOC/HARD/PPOP.prj
-ifmt vhdl
-ofn C:/PlateformeSOC/HARD/work/PPOP.ngc
-ofmt NGC
-case Lower
-opt_mode area
-opt_level 1
-p xc2v6000-6ff1152
#VHDL Source Options
-ent plateformeemulation
#HDL Options(VHDL and Verilog)
#-fsm_extract yes
#-fsm_encoding Auto
-ram_extract yes
-ram_style Auto
-rom_extract yes
-rom_style Auto
-mult_style Auto
-Mux_extract yes
-Mux_style Auto
-decoder_extract yes
-priority_extract yes
-shreg_extract yes
-shift_extract yes
-xor_collapse yes
-resource_sharing yes
-complex_clken yes
#Target Options for Virtex-II
#-bufg 16
#-cross_clock_analysis no
#-equivalent_register_removal yes
-glob_opt AllClockNets
-iob Auto
-iobuf yes
-keep_hierarchy no
-max_fanout 500
-read_cores yes
-register_balancing no
-move_first_stage yes
-move_last_stage yes
-register_duplication yes
-slice_packing yes
-slice_utilization_ratio 80
-slice_utilization_ratio_maxmargin 1
```

ANNEXE E.4 : Exemple d'un script de placement-routage pour outil PAR

```
# Script of Place and Route (PAR) for platform emulation of Leon
```

```
# Ivan Hermann
# ENSTA-Paris
# 17/10/2003
##### Modifs Parameters :
#####   -n  : 1
#####   -ol  : lowest

# Effort Level Options
-gf C:/PlateformeSOC/HARD/work/PPOP.ncd
-ol 1
#-pl std
#-rl std
#-xe 1

# General Options
#-intstyle
#-k
#-nopad
#-p
#-r
#-ub
-w
#-x

# Multi Pass Place and Route (MPPR) Options
-s 1
-t 1
-n 1

C:/PlateformeSOC/HARD/work/PPOP.ncd
C:/PlateformeSOC/HARD/work/multi_routed.dir
C:/PlateformeSOC/HARD/work/PPOP.pcf
```

ANNEXE E.5 : Fichier paramètres à l'entrée de NSGA-II, cas FPGA

```
                    /***** ParamInput.txt *****/
/*************************************************************************
 *             Methodology of Conception SoC Application                 *
 *                    to the Physical Layer of UMTS                      *
 *                          W-CDMA Simulation                            *
 *                                                                       *
 *        Genetic Algorithm for Multiobjective Optimisation              *
 *                           NSGA-II Ver.16/07/2004                      *
 *                                                                       *
 *        Revised by:                                                    *
 *            GHALI Khemaies                                             *
 *            ENSTA-LEI (Paris)                                          *
 *            Email: ghali@ensta.fr                                      *
 *            http://www.ensta.fr/~ghali                                 *
 *************************************************************************/
/*************************************************************************
 *     This is a file to get the input for the NSGA-II program from user   *
 *************************************************************************/
PopSize             32     // taille de la population
NbrGen              50     // nombre de génération
SelType             1     //Selection strategy: for tournament 1 & for roullette
wheel 2
CrossType           1     //Crossover type: 1 for Simple one & 2 for Uniform X-over
and 3 for Real-Coded
PrbCross            0.8    //Cross-over Probability for the Simple or Real Coded
//PrbCross          0.1    //If CrossType==2
//PrbMut            0.1    // the mutation probability (between 0 and %f)
NbrFunc             3      // nombre des function objectif
NbrVar       13     // nombre des variable dans l'espace d'exploration

LimVarInfDef               -80.0  //lower limits for each variable
LimVarSupDef        80.0          //upper limits for each variable
Seed                0.5
```

```
NbrSshByHost 1
HostSelectStrategy  3      // Host Select Methodology:     1=>Best Host Select,
                                                                2=>Random Host
Select and
                                                                3=>in Order Host
Select
//------------------Leon_Data_Cache_Options---------------------//
x0 {
       TargetConf    Data_Cache
       VarName       dsets
       VarNbrBit     2
       VarTabVal     1      2      3      4
}
x1 {
       TargetConf    Data_Cache
       VarName       dsetsize
       VarNbrBit     2
       VarTabVal     2      4      8      16
}
x2 {
       TargetConf    Data_Cache
       VarName       dlinesize
       VarNbrBit     1
       VarTabVal     4      8
}
x3 {
       TargetConf    Data_Cache
       VarName       dreplace
       VarNbrBit     1
       VarTabVal     1      3
}
//------------------Leon_Inst_Cache_Options---------------------//
x4 {
       TargetConf    Inst_Cache
       VarName       isets
       VarNbrBit     2
       VarTabVal     1      2      3      4
}
x5 {
       TargetConf    Inst_Cache
       VarName       isetsize
       VarNbrBit     2
       VarTabVal     1      2      4      8
}
x6 {
       TargetConf    Inst_Cache
       VarName       ilinesize
       VarNbrBit     1
       VarTabVal     4      8
}
x7 {
       TargetConf    Inst_Cache
       VarName       ireplace
       VarNbrBit     1
       VarTabVal     1      3
}
//----------------------Xilinx_XST_Options-------------------------//
x8 {
       TargetConf    XST_Options
       VarName       -opt_mode
       VarNbrBit     1
       VarTabVal     1      2
}
x9 {
       TargetConf    XST_Options
       VarName       -opt_level
       VarNbrBit     1
       VarTabVal     1      2
```

```
}
x10 {
       TargetConf    XST_Options
       VarName       -slice_utilization_ratio_maxmargin
       VarNbrBit     1
       VarTabVal     1      4
}
x11 {
       TargetConf    XST_Options
       VarName       -slice_utilization_ratio
       VarNbrBit     1
       VarTabVal     70     80
}
//-----------------------Xilinx_PAR_Options-------------------------//
x12 {
       TargetConf    PAR_Options
       VarName       -ol
       VarNbrBit     2
       VarTabVal     1      2      4      5
}
x13 {
       TargetConf    PAR_Options
       VarName       -n
       VarNbrBit     1
       VarTabVal     1      2
}
```