Titre: Title:	Implémentation et optimisation d'un décodeur à seuil itératif de codes convolutionnels doublement orthogonaux
Auteur: Author:	Ghislain Provost
Date:	2005
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Provost, G. (2005). Implémentation et optimisation d'un décodeur à seuil itératif de codes convolutionnels doublement orthogonaux [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8398/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/8398/
Directeurs de recherche: Advisors:	Mohamad Sawan, & David Haccoun
Programme: Program:	Unspecified

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Implémentation et optimisation d'un décodeur à seuil itératif de codes convolutionnels doublement orthogonaux

Ghislain Provost
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE EN SCIENCES APPLIQUÉES (M.Sc.A.)
(GÉNIE ÉLECTRIQUE)
Avril 2005

Your file Votre référence
ISBN: 978-0-494-47683-3
Our file Notre référence
ISBN: 978-0-494-47683-3

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE

Ce mémoire intitulé:

Implémentation et optimisation d'un décodeur à seuil itératif de codes convolutionnels doublement orthogonaux

présenté par: Ghislain Provost

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

Yvon Savaria, Ph.D., président

Mohamad Sawan, Ph.D., membre et directeur de recherche

David Haccoun, Ph.D., membre et codirecteur

Christian Cardinal, Ph.D., membre

Je dédie ce mémoire à mon frère Jocelyn qui a su me montrer que rien n'est impossible et qu'il suffit de foncer sur l'objectif.

Remerciements

Je tiens à remercier principalement monsieur David Haccoun, et monsieur Mohamad Sawan, pour leur contribution financière. Également monsieur Mohamad Sawan pour la supervision ainsi que le support qu'il m'a apporté au cours de cette maîtrise. J'aimerais souligné l'apport de monsieur Yvon Savaria pour ses critiques constructives de plusieurs éléments de ce mémoire et Christian Cardinal pour ses précieux commentaires sur les aspects de télécommunications. Aussi j'aimerais remercier monsieur Marc-André Cantin pour sa collaboration étroite pour le performant optimiseur que nous avons construit. Finalement je remercie la compagnie Xilinx pour leur généreux don de l'IP générateur de bruit blanc gaussien et la société canadienne en microélectronique (SCM) pour les outils de conception et la plateforme ARM Integrator.

Résumé

Les travaux de recherche présentés dans ce mémoire discutent différents aspects liés à l'implémentation d'un décodeur à seuil itératif. Ce type de décodeur utilisé conjointement à un codeur de codes convolutionnels doublement orthogonaux définit au sens large constitue un performant algorithme de correction d'erreurs. Cet algorithme fut introduit en 1998 par Cardinal et al. [7–9] dans le but de réduire la complexité matérielle requise pour les performants codeurs et décodeurs turbo. L'intérêt pour ce type d'algorithme de correction d'erreurs est très grand dans le domaine des télécommunications. Cet intérêt est principalement nourri par la performance de correction d'erreurs de la famille de décodeurs turbo qui atteint de faibles taux d'erreurs à des rapports signal à bruit proches de la limite de Shannon.

Notre but dans ce mémoire est d'explorer les différentes facettes qu'implique une implémentation matérielle de cet algorithme. Le premier objectif fut de réaliser et de valider l'implémentation VHDL de ce décodeur à seuil itératif. Le module IP (Intellectual Propriety) développé possède une grande configurabilité des différents paramètres que comporte le décodeur à seuil itératif (DSI, Décodeur à Seuil Itératif). Ce IP décrit l'architecture du DSI de manière structurelle, ce qui donne un meilleur contrôle des ressources matérielles utilisées lors de la synthèse logique. Il est possible de construire un DSI à partir de n'importe quel code, de complexité matérielle raisonnable, qu'il soit ou non doublement orthogonaux au sens large et pour un taux de codage égal

à $\frac{1}{2}$. Également un nombre arbitraire d'itérations est implémenté et la largueur des mots de l'architecture est paramétrable.

Une optimisation du délai combinatoire et de la complexité des différents opérateurs qui constitue le DSI fut réalisée. Cette optimisation décrit l'architecture optimale des opérateurs add-min et de l'additionneur ainsi que les particularités d'implémentation selon les représentations binaires complément à deux et signe-amplitude. Cette étude a aussi permis de conclure que l'emploi de la représentation signe-amplitude réduit la complexité matérielle du DSI comparativement à la représentation complément à deux. Il est aussi possible d'approximer le délai combinatoire du décodeur à seuil itératif en utilisant les données récoltées lors de la synthèse des différents opérateurs logiques utilisés dans le DSI.

La vérification du bon fonctionnement d'un algorithme de correction d'erreurs performant peut s'avérer laborieux. La façon la plus représentative de démontrer le bon fonctionnement constitue à mesurer le taux d'erreurs résiduel à différents rapports signal à bruit (SNR, Signal to Noise Ratio). Le temps nécessaire à la version logicielle pour caractériser les performances du décodeur est considérable, il peut se compter en jours. De plus le temps nécessaire pour vérifier les performances de l'IP matérielle à l'aide d'un logiciel de simulation VHDL tel que Modelsim est beaucoup plus important. Afin de s'assurer du bon fonctionnement du décodeur matériel, un prototype permettant de caractériser la performance de correction d'erreurs a été réalisé en utilisant la plateforme "ARM integrator".

En plus de servir à la vérification du bon fonctionnement du DSI, cette plateforme a permis de trouver de meilleurs coefficients de pondération qui maximalise la performance de correction d'erreurs à de faibles valeurs de SNR. Finalement, cet accélérateur a permis de caractériser sur une plage de valeurs SNR étendue, variant entre 2.0 dB et 8.0 dB, le comportement de correction d'erreurs du DSI. Le comportement

du DSI à des SNR relativement élevés n'avait jamais été vérifié auparavant.

Finalement un nouveau critère de sélection des codes doublement orthogonaux est introduit, offrant de nouvelles possibilités quant à l'implémentation du décodeur. Ce nouveau critère couplé à une stratégie du réajustement de l'ordonnancement rend possible une implémentation synchrone fonctionnant à une cadence d'horloge élevée. Ainsi les codes de longueur moindre ne sont pas nécessairement ceux qui permettent une implémentation efficace du décodeur tel qu'énoncé dans les recherches précédentes. Une étude de l'influence de ce nouveau critère sur la longueur du code résultant montre qu'une accélération substantielle du DSI est possible à un faible coût matériel supplémentaire. Le coût matériel qu'introduit ce nouveau critère est mis en relation avec le facteur d'accélération possible qu'on peut atteindre avec ce code. L'introduction de ce critère constitue une innovation importante apportée à l'algorithme. Finalement, une technique de saturation des sorties est introduite dans l'architecture et réduit de manière significative la mémoire nécessaire à l'implémentation de l'algorithme.

Abstract

The research done in this master thesis presents different aspects related to the implementation of an Iterative Threshold Decoder (ITD). This type of decoder used with a Convolutional Self doubly Orthogonal Code defined in the Wide-Sense (CSO^2C-WS) constitutes a powerful error correcting algorithm. This algorithm has been introduced in 1998 by Cardinal et al [7–9] in order to reduce the hardware complexity needed by the turbo encoder and decoder. The research efforts devoted to the turbo decoder by the scientific community is still intensive. This interest is maintained because the turbo decoder achieves a low bit error rate which approaches the Shannon limit.

Our main goal in this research is to explore the different facets of the ITD hardware implementation. The first objective was to create and validate a VHDL ITD implementation. This IP (Intellectual Propriety) module has a great flexibility with respect to various ITD parameters. It describes the ITD implementation in a structural manner to give to the designer full control over the hardware resources required during the logic synthesis. The IP supports any convolutional code, with a reasonable hardware complexity, whether it can be CSO^2C-WS or not for a coding rate of $\frac{1}{2}$. Also an arbitrary number of iterations is implemented and the architecture word lengths is parametrizable.

A detailed analysis of the critical path delay of the ITD operator was performed and was optimized. The implemented ITD was carefully verified. For instance, the

error correcting performance was verified for many values of Signal to Noise Ratio (SNR). It was found that performing these analysis with a software model of the ITD is very time consuming. Thus a hardware implementation of the ITD was developed to accelerate the process. This hardware model was implemented on a ARM integrator platform.

This platform was used to find the weighing factors that maximize the error correcting performance at low SNR. It was also used to verify the ITD error correcting capability behavior over an extended SNR range between 2.0 dB and 8.0 dB.

Finally, a new selection criteria for the CSO^{2}C-WS has been introduced to allow an efficient ITD implementation. This new criteria coupled with the retiming theorem allows to increase considerably the ITD maximum clock frequency. The previous researches, which were trying to find the best code, i.e. with a minimal code length, do not necessarily led to the best decoder implementation. The codes obtained with this new criteria show that a substantial frequency acceleration is accomplished by adding only a small amount of hardware resources. Also a chosen metric, which is the ratio between the supplemental hardware cost and the theoritical acceleration factor, shows the gain evolution in relationship with the new criteria. The criteria introduction is an important research contribution to the ITD hardware implementation.

Table des matières

Liste des figures

Liste des tableaux

Liste des sigles et abréviations

AHB : Advanced High-performance Bus

APB : Advanced Peripheral Bus

ASIC : Application-Specific Integrated Circuit

AWGN : Additive White Gaussian Noise

AWLDT : Automatic Word Length Determination Tool

CAN : Convertisseur Analogique-numérique

CLB : Configurable Logic Block

CMC : Société Canadienne en Micro-électronique (Canadian Microelectronics Corporation)

CMOS : Complementary Metal-Oxide Semiconductor (transistor type)

CPU : Central Processing Unit

CSO^{2}C-SS : Convolutional Self-Doubly Orthogonal Codes in the Strict Sense

CSO^{2}C-WS : Convolutional Self doubly Orthogonal Code defined in Wide-Sense

DCM : Ditital Clock Management

DIMS : Delay-Insensitive Minterm Systhesis

DR-ST : Double-Rail Self-Timed

DSI : Décodeur à Seuil Itératif

ECM : Environnement de caractérisation matériel

FPGA : Field Programmable Gate Array

IOB : Input/Output Block (in FPGA)

IP : Intellectual Propriety

ITD : Iterative Threshold Decoder

LC : Logic Cell

LFSR : Linear Feedback Shift Register

LUT : Lookup Table

LVDS : Low Voltage Differential Signal(ing)

NMOS : Negative-Channel Metal Oxide Semiconductor

PCI : Peripheral Component Interconnect (personal computer bus)

PMOS : Positive-Channel Metal Oxide Semiconductor

RAM : Random-Access Memory

RTL : Register Transfer Level

SNR : Signal to Noise Ratio

TCL : Tool Command Language

TCP/IP : Transmission Control Protocol/Internet Protocol

TEB : Taux d'Erreurs par Bit USB : Universal Serial Bus

VHDL : VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

R : Taux de codage (Coding Rate)

Liste des annexes

Chapitre 1

Introduction

1.1 Motivation

Dans la nouvelle ère de la société du savoir, la communication de l'information constitue le pilier de cette révolution que vit actuellement l'humanité. Ce nouveau mode de vie apporte à la plupart des personnes de nouvelles habitudes telles que le courrier électronique comme principal moyen de communication et la capacité de rejoindre en tout temps une personne grâce à la téléphonie cellulaire. La société du savoir s'appuie principalement sur l'existence d'un réseau de communication performant et fiable. L'efficacité de ce réseau repose en grande partie sur l'existence d'une révolution des cinquante dernières années, celle de la microélectronique. La venue d'Internet permet également d'accéder à une multitude d'information à la portée de la main. Il faut dire que l'accessibilité à cette riche ressource d'information est précieuse, mais faut-il néanmoins être en mesure de pouvoir transmettre celle-ci efficacement et sans erreur.

Découvert en 1993 par Berrou et autres [3], le décodeur turbo est un algorithme de correction d'erreurs très performant capable de corriger des erreurs avec un taux

résiduel très bas à des rapports signal à bruit proches de la limite de Shannon [26]. Cependant l'énorme complexité matérielle de ce type de décodeur constitue un désavantage majeur à la pénétration de cette technologie dans les futurs systèmes de communication.

Ces dernières années, beaucoup d'efforts ont été dépensés afin d'implémenter et d'améliorer un nouvel algorithme conçu par Cardinal et al [8]. Avec cette découverte, une nouvelle classe d'algorithmes de correction d'erreurs [8] réduisant les erreurs subvenues lors d'une transmission a vu le jour. Ce nouveau type d'algorithme propose un compromis intéressant, entre la complexité matérielle et la performance de correction d'erreurs, à celui offert par les puissants décodeurs turbo.

Nous avons entrepris dans ce mémoire l'implémentation de ce nouvel algorithme de corrections d'erreurs. L'objectif principal est la création d'un IP (Intellectual Propriety) paramétrable implémentant le nouvel algorithme de correction d'erreurs soit le décodeur à seuil itératif. Le second objectif consiste en l'étude de la complexité matérielle et du délai du chemin critique afin de déterminer la fréquence maximale d'opération. Dans la section suivante, les principales contributions découlant de ces 2 objectifs initiaux sont énumérées.

1.2 Contributions

Plusieurs contributions sont discutées dans ce mémoire.

1. Création d'un IP paramétrable qui implémente un décodeur à seuil itératif qui supporte n'importe quel code convolutionnel doublement orthogonal au sens large, avec une complexité matérielle raisonnable, et un nombre arbitraire d'itérations pour un taux de codage égal à $\frac{1}{2}$.
2. Première implémentation en matériel du décodeur à seuil itératif.

3. Étude de la complexité matérielle et du chemin critique en fonction des différents paramètres du décodeur à seuil itératif.
4. Identification du facteur limitant le réajustement de l'ordonnancement de l'architecture. Définition d'un nouveau critère de sélection de codes doublement orthogonaux qui repousse les facteurs limitant le réajustement de l'ordonnancement. Identification d'un ensemble de codes où il est possible d'atteindre des débits considérables d'informations traitées à l'aide de ce critère.
5. Présentation d'une nouvelle méthodologie pour la caractérisation de la performance d'un algorithme de correction d'erreurs. Cette approche peut être transposée à différents problèmes d'optimisation.
6. Optimisation des coefficients de pondération et découverte d'un vecteur de coefficient de pondération qui donne de meilleures performances à un SNR (Signal to Noise Ratio) peu élevé. Également un gain généralisé des performances de correction d'erreurs est observé pour la plupart des itérations dont le nombres d'éléments du code utilisé égal à 10.
7. Résultats expérimentaux de la performance du décodeur à seuil itératif à de hauts rapports signal à bruit.

1.3 Organisation du mémoire

L'objectif de ce mémoire est d'étudier les différents aspects qu'implique l'implémentation matérielle de cet algorithme. Dans le prochain chapitre, des notions de télécommunication et de matérielle (électroniques intégrées) préalables à la compréhension des chapitres 3 et 4 sont présentées. La section portant sur les notions de bases de télécommunication, présente de manière détaillée les différentes parties constituant un système de communication. Également, des notions plus spécifiques au décodeur

à seuil itératif forment une partie de ce chapitre. Les équations qui sont à la base du décodeur à seuil itératif sont expliquées et une correspondance est faite avec l'architecture proposée par Cardinal et al. [8]. Les règles de la double orthogonalité au sens large et la méthode de recherche heuristique pseudo aléatoire de codes CSO^2C-WS (Convolutional Self-Doubly Orthogonal Codes in the Wide Sense) respectant la double orthogonalité sont décrites.

Dans la section se rapportant aux notions matérielles du chapitre 2, une introduction aux composants FPGA (Field Programmable Gate Array) et de l'architecture des FPGA de la société Xilinx est réalisée. Des notions théoriques quant au facteur limitant la fréquence maximale d'opération d'un circuit numérique synchrone sont décrites, ce qui permettra de mieux comprendre ce qui empêche d'atteindre une cadence d'horloge élevée pour un décodeur à seuil itératif. La méthode du réajustement de l'ordonnancement et d'opération pipeline, principales techniques qui augmentent la fréquence d'opération, sont introduites au lecteur. Une analyse des différents éléments consommateurs de puissance d'un circuit CMOS (Complementary Metal-Oxide Semiconductor), ainsi que la plateforme "ARM integrator" utilisée pour implémenter toutes les parties matérielles sont présentées. Finalement, une brève explication de la conception de circuit asynchrone, de l'intérêt en ce mode de conception et des causes de la non viabilité d'une implémentation du décodeur à seuil itératif utilisant cette technique sont élaborés.

Au chapitre 3, une approche structurelle VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language) de l'architecture du DSI (Décodeur à Seuil Itératif) est détaillée. La flexibilité de l'IP développée, ainsi que celle des sous-modules sont décrites. Aussi, la reconfigurabilité des sous-modules ainsi que la connectivité de ces sous-modules en fonction du CSO^2C-WS utilisé sont élaborées. Ce chapitre présente l'architecture optimale des opérateurs qui composent le DSI. Une

étude des performances de corrections en fonction des coefficients de pondération et de la résolution binaire utilisée à l'intérieur de l'architecture montre un aperçu de l'évolution du taux d'erreurs en fonction de la complexité matérielle des éléments mémoires dans le décodeur. Également dans ce chapitre est décrit une méthodologie de vérification préliminaire de la fonctionnalité du DSI. Enfin, le dernier point traitera des techniques d'optimisation des ressources mémoires et de diminution du délai du chemin critique au sein du décodeur. Ainsi la technique de saturation permet de diminuer la mémoire consommée de l'implémentation. La technique du réajustement de l'ordonnancement permettant d'augmenter considérablement le débit d'information traité.

Au chapitre 4, une méthodologie efficace et rapide de caractérisation des performances d'un algorithme de correction d'erreurs fut élaborée. Cette méthodologie propose d'implémenter physiquement l'algorithme au sein d'un système de communication afin de diminuer le temps nécessaire à la caractérisation des performances de correction d'erreurs. Les implémentations matérielle et logicielle réalisées sur la plateforme "ARM integrator" sont détaillées. Cet accélérateur fut utilisé afin de vérifier le fonctionnement du décodeur à seuil itératif. Aussi, il a servi à déterminer les coefficients de pondération afin d'obtenir de meilleures performances de correction d'erreurs du décodeur et à vérifier le comportement du pouvoir correcteur à des rapports signal à bruit élevés.

Finalement, le dernier chapitre présente les conclusions des travaux présentés dans ce mémoire et donne quelques pistes possibles de recherches futures.

Chapitre 2

Notions préliminaires

2.1 Notions de télécommunications

2.1.1 Système de communication

Plusieurs éléments sont utilisés dans un système de communication moderne entre la source émettrice d'information (transmetteur) et le destinataire (récepteur). La figure 2.1 montre schématiquement le trajet qu'emprunte l'information dans un système de communication. Le traitement de l'information débute par une opération de codage. Le rôle du codeur est d'ajouter au signal la faculté de correction et/ou de détection des erreurs qui peuvent se produire lors de la transmission de l'information. L'addition de cette faculté de correction et de détection d'erreurs correspond à ajouter une redondance au sein du signal numérique original.

Suite au processus de codage et avant de transmettre cette information sur le canal de transmission, le signal est converti en une forme mieux adaptée à sa transmission. Ce processus se nomme la modulation. Ainsi la modulation traduit les valeurs proprement binaires sous une forme analogique (symbole). À la réception se trouve un

démodulateur qui estime quel signal fut envoyé et converti ce symbole sous forme binaire. Un décodeur est utilisé pour corriger les erreurs ayant pu se produire. Ces erreurs sont causées principalement par des phénomènes non désirables dans le canal de communication.

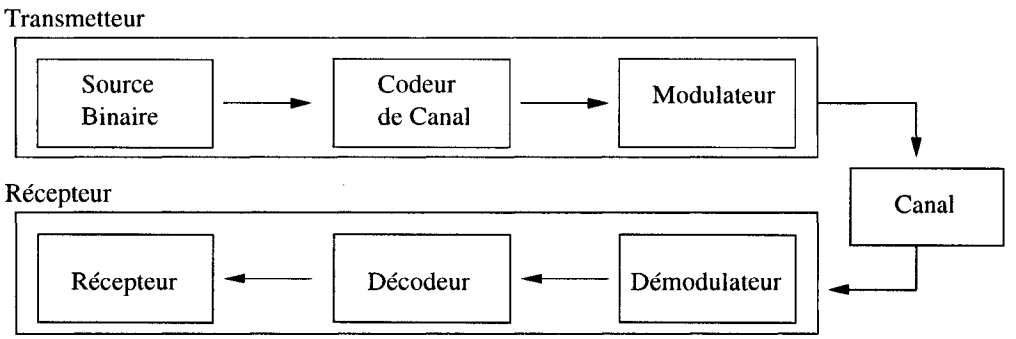


Figure 2.1: Schéma bloc d'un système de communication

2.1.1.1 Source d'information

La source d'information peut être de nature diverse. Par exemple, cette source peut représenter un usager discutant au téléphone. À des fins de conception de circuits numériques, la source d'information est toujours traduite sous forme électrique par l'intermédiaire d'un capteur quelconque. Également, ce signal transformé sous forme électrique est préalablement échantillonné afin de bénéficier de la facilité de conception des circuits numériques d'aujourd'hui. La nature de l'information véhiculée, c'est-à-dire ce qu'elle représente, n'a pas d'importance et n'influence en rien le traitement qu'elle reçoit par les divers modules jusqu'à l'acheminement à l'usager. L'information générée par la source est sérialisée et acheminée au codeur de canal. Pour des fins théoriques, on considère que la source d'information utilise un alphabet dont les bits sont équiprobables. Ainsi, pour une source binaire, la probabilité est de 0.5 pour chacune des valeurs 0 et 1. Dans le cas contraire, la source d'information possède une

redondance inutile et celle-ci doit être compressée afin de maximiser son entropie [26].

2.1.1.2 Principe de codage

Le codage est un mécanisme qui ajoute la possibilité de correction et/ou de détection d'erreurs au signal provenant de la source. Deux principaux types de codage sont utilisés : soit le codage en bloc et le codage convolutionnel. Comme son nom l'indique, le codage en bloc utilise un ensemble de bits de taille prédéfinie pour calculer un nombre déterminé de bits de parité qui seront ajoutés au bloc en question. Le codage convolutionnel calcule les bits de parité au fur et à mesure que les bits d'information lui sont acheminés. Cette propriété découle du fait que ce type de codage utilise une fenêtre fixe temporellement où sont utilisées certaines données pour calculer le bit de parité résultant. D'ailleurs, cette propriété est le principal avantage du codage convolutionnel. C'est également ce type de codage qui est utilisé de pair avec le décodage à seuil itératif dont il est question dans ce mémoire.

Un paramètre important à considérer lors du codage est le taux de codage (R). Ce dernier représente la portion du signal résultant dédiée à l'information véhiculée suite au processus du codage. Par exemple, lors d'un processus de codage où K bits d'information sont utilisés pour générer D bits de parité, le taux de codage est donné par la formule 2.1 :

$$R = K/(K + D) \tag{2.1}$$

Les répercussions de R sont importantes. En principe, tout concepteur de système recherche à maximiser la largeur de bande allouée à l'information utile et ainsi tendre R le plus proche possible de 1. Également, tous cherchent à minimiser le taux d'erreurs, ce qui ce traduit par une augmentation de la proportion de la bande passante consacrée

à la parité et conséquemment à une diminution de R. Il en découle qu'il existe toujours un compromis entre le taux de codage et le taux d'erreur.

2.1.1.3 Canal de communication

Le milieu par lequel transite l'information et la parité une fois codées se nomme canal de communication. Selon le milieu, différents types d'altérations peuvent se produire. Les principaux milieux utilisés comme canal de communication sont les conducteurs métalliques (cuivre) tels qu'utilisée pour les communications téléphoniques standards, l'atmosphère (voie hertzienne) pour les communications cellulaires et satellites et la fibre optique pour les communications terrestres à haut débit. Dans certains pays asiatiques et d'Afrique, où les réseaux câblés sont inexistants, la technologie cellulaire est beaucoup plus répandue car ses coûts d'installation et d'entretien moindres la favorise. Dans un milieu urbain où plusieurs obstacles à la propagation de l'onde sont présents et où le canal de communication n'est pas formé par une ligne directe, un phénomène d'écho apparaît. Ce phénomène est produit par l'addition du même signal ayant parcouru des chemins différents. Dans le cadre de cette maîtrise, la source de bruit considérée est un bruit blanc gaussien de densité spectrale $N_o/2$. Les types d'altérations autres que le bruit blanc gaussien, tel que le phénomène d'écho, ne sont pas considérées.

2.1.2 Codage convolutionnel

La figure 2.2 représente la structure d'un codeur convolutionnel systématique ayant un taux de codage égal à $\frac{1}{2}$. Un code convolutionnel systématique contient l'information sous sa forme originale à la sortie du codeur. Dans le cadre de cette maîtrise, R sera toujours égal $\frac{1}{2}$. La technique de perforation permet d'augmenter R,

mais ne sera pas discutée dans cette maîtrise [10]. Dans la figure 2.2, la structure du codeur convolutionnel est constituée d'un registre à décalage qui agit de fenêtre temporelle. À son entrée, un train binaire $U_i = \{u_i, u_{i+1}, ...\}$ d'information entrant alimente le codeur et à sa sortie le train binaire $V_i = \{v_i, v_{i+1}, ...\}$ de parité est généré par celui-ci. Les deux trains binaires sont multiplexés temporellement avant d'être modulés et transmis sur le canal de transmission.

Certaines informations sont utilisées par le codeur pour calculer les bits de parité. L'opérateur "+" représente simplement un opérateur XOR à plusieurs entrées dont le résultat correspond au bit de parité. Chacun des éléments du vecteur générateur $G = \{\alpha_1, \alpha_2, ..., \alpha_J\}$ est dénoté par une constante α_i où J correspond au nombre d'éléments dans le vecteur et constitue le code convolutionnel. Également, un paramètre important du code convolutionnel employé est le dernier élément α_J que l'on nomme dans la littérature "span" qui est la longueur du code. Cette longueur correspond à la longueur de la fenêtre temporelle nécessaire au codage. Le choix de ces éléments $G = \{\alpha_1, \alpha_2, ..., \alpha_J\}$ a une incidence directe sur la performance du processus de corrections d'erreurs à la réception. La prochaine section introduit les règles qui dictent le choix de ces éléments constituant le code convolutionnel.

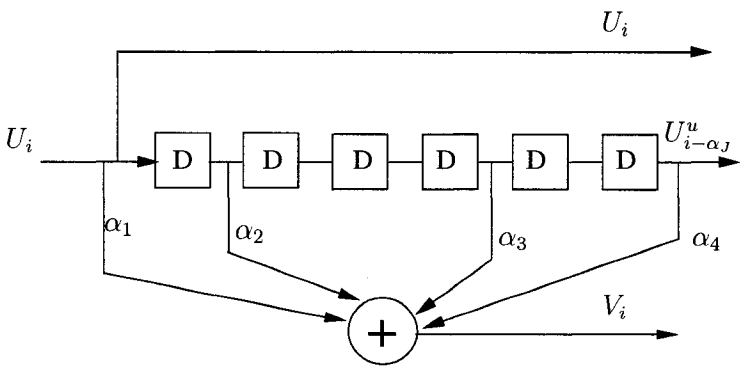


Figure 2.2: Structure d'un codeur convolutionnel où $G = \{0, 1, 4, 6\}$

2.1.3 Définition de la double orthogonalité au sens large

La double orthogonalité dicte les critères nécessaires à la sélection des éléments (α_i) formant un code convolutionnel. Un codeur convolutionnel dont le vecteur générateur respecte ces règles permet à la réception d'obtenir des performances de correction d'erreurs à de faibles SNR lorsqu'il est utilisé conjointement avec un décodeur à seuil itératif. Ainsi, 3 règles doivent être respectées afin qu'un code convolutionnel soit doublement orthogonaux au sens large [7] :

1. les différences $(\alpha_j - \alpha_k)$ sont distinctes.
2. les différences de différences $((\alpha_j - \alpha_k) - (\alpha_q - \alpha_l))$ sont distinctes des différences.
3. les différences de différences $((\alpha_j - \alpha_k) - (\alpha_q - \alpha_l))$ sont distinctes.

Il est à noter que pour toutes les combinaisons des éléments $(j, k, q, l), j \neq k, l \neq q, k \neq l$ et $q \neq j$ sauf pour les permutations inévitables.

Cette définition de la double orthogonalité est définie au sens large. La définition au sens large est en fait une particularité de la définition des codes convolutionnels au sens strict (CSO^{2}C-SS,Convolutional Self-Doubly Orthogonal Codes in the Strict Sense). Il est impossible d'utiliser la structure du codeur convolutionnel présentée à la section 2.1.2 en ayant toutes les différences de différences distinctes. L'adoption de cette architecture cause une répétition inévitable de certaines différences de différences [7–9] d'où la distinction entre les codes convolutionnels définis au sens strict et au sens large.

Également, on peut prouver que la troisième règle inclut la première. Ces 3 règles sont très restrictives et ont un impact important sur la longueur du code résultant. Ainsi, le tableau 2.1 montre différents codes publiés précédemment [8] pour un paramètre J variant de 4 à 10.

La section suivante présente une méthode de recherche heuristique qui permet de trouver des codes convolutionnels doublement orthogonaux définis au sens large.

Tableau 2.1: Exemple de codes convolutionnels doublement orthogonaux définit au sens large [7–9]

Nombre d'éléments (J)	Codes
4	{0,1,4,6}
6	{0,1,17,70,95,100}
7	{0,1,9,71,177,215,228}
8	{0,42,139,332,422,430,441,459}
9	{0,9,21,395,584,767,871,899,912}
10	{0,27,93,503,600,1247,1646,1714,1825,1835}

2.1.4 Méthode de recherche pseudo aléatoire de code CSO^{2}C-WS

La recherche de codes CSO^{2}C-WS fut investiguée par Brice Baechler [2] en l'année 2000. Il propose plusieurs méthodes de recherche de codes respectant les règles de la double orthogonalité au sens large. La méthode qui donne les meilleurs résultats en un temps acceptable emploie une stratégie de recherche pseudo aléatoire. Cette méthode utilise la propriété d'extension des codes doublement orthogonaux au sens large, c'est-à-dire, par exemple pour un code comportant 10 éléments soit J=10 les 9 premiers éléments forment également un code doublement orthogonaux au sens large.

De manière succinte, cette technique cherche à construire successivement un code CSO^{2}C-WS en ajoutant de nouveaux éléments jusqu'à ce que le nombre d'éléments voulus soit atteint par le biais de la propriété d'extension de ces codes. Lors de l'ajout d'un nouvel élément, le respect de la double orthogonalité du code résultant est vérifié. L'élément est retenu seulement si le résultat d'un tirage pseudo aléatoire est positif. Finalement, le processus est répété jusqu'à ce que le nombre d'éléments voulus soit atteint.

Par la suite un code est retenu si sa longueur est moindre que celle du code retenu précédemment. Cette méthode sera utilisée plus loin afin de rechercher une nouvelle

catégorie de codes offrant un potentiel d'implémentation matérielle capable de fonctionner à des débits d'information de quelques dizaines de Méga bits par seconde.

2.1.5 Décodeur à seuil itératif

2.1.5.1 Description de l'algorithme

Utilisé conjointement avec un codeur convolutionnel de codes CSO^2C-WS, le décodeur à seuil itératif (DSI, Décodeur à Seuil Itératif) est placé immédiatement à la suite du canal de communication après un échantillonneur-bloqueur. Les symboles provenant du canal sont quantifiés par ce dernier sur N bits et sont acheminés au DSI. Un algorithme de correction d'erreurs qui utilise des symboles quantifiés sur plusieurs bits se nomme un algorithme à quantification douce (Soft-input). Ainsi, ces valeurs sont introduites dans le décodeur où $y^u_{i+M\alpha_J}$ sont les valeurs quantifiées des symboles d'information, $y^p_{i+M\alpha_J}$ correspond aux valeurs quantifiées des symboles de parité et $\hat{U}_i$ est la valeur du bit d'information estimée à l'instant i. Les indices temporels des symboles d'information et de parité $(i + M\alpha_J)$ indique la présence d'une latence à l'intérieur du processus de correction d'erreurs.

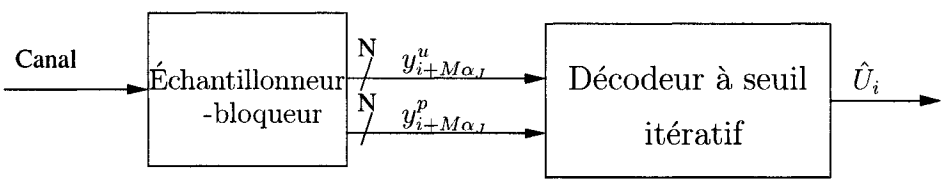


Figure 2.3: Décodeur à seuil itératif

Le fonctionnement du décodeur se fait de manière continue. C'est-à-dire qu'une fois le processus de correction d'erreurs est démarré, ce décodeur à seuil itératif corrige un nouveau bit pour chaque nouveau couple de symboles d'information et de parité. Ce type de fonctionnement est contraire au décodeur de Viterbi. Ce dernier corrige

les erreurs d'un codage en bloc et doit nécessairement recevoir entièrement un bloc de données transmis avant de démarrer le processus de correction des erreurs qui se sont produites. Donc le DSI inventé par Cardinal et al. [7] possède également la même faculté de traitement en continu propre au codeur convolutionnel. Ce type de propriété appliqué au DSI s'appelle décodage symbole par symbole.

Le DSI est constitué de plusieurs éléments de traitement cascadés, que l'on nomme itérations. Les deux équations (2.2) et (2.3) décrivent l'architecture de l'ensemble des itérations du DSI. Ainsi, l'équation (2.2) décrit la première itération tandis que l'équation (2.3) décrit l'ensemble des autres itérations. Les entrées du DSI sont les symboles d'information y_i^u et de parité y_i^p provenant de l'échantillonneur-bloqueur. Le terme $\lambda^{(\mu)}$ du côté gauche de l'équation (2.3) est le résultat de l'itération μ. Du côté droit des équations, le terme $\lambda^{(\mu)}$ de l'itération μ dénote la présence d'une boucle de rétroaction et le terme $\lambda^{(\mu-1)}$ témoigne de l'utilisation du résultat de l'itération précédente $\mu - 1$ par l'itération μ. C'est pour cette raison que l'algorithme est dit itératif. Évidemment à la première itération, on ne peut utiliser les valeurs $\lambda^{(\mu-1)}$ de l'itération précédente et donc l'équation de cette itération diffère des autres. Les valeurs de $\lambda^{(\mu-1)}$ sont remplacées par les valeurs des symboles d'information y_i^u et les indices temporels $i + \alpha_j - \alpha_k$ restent les mêmes. Pour l'ensemble des termes y_i^u, y_i^p, $\lambda^{(\mu-1)}$ et $\lambda^{(\mu)}$ les différents indices temporels nécessaires découlent des indices α_i du CSO^2C-WS utilisés par le codeur.

$$\lambda_i^{(1)} = y_i^u + \sum_{j=1}^{J} \left(y_{i+\alpha_j}^p \diamond \sum_{k=1}^{j-1} \diamond y_{i+\alpha_j-\alpha_k}^u \diamond \sum_{k=j+1}^{J} \diamond \lambda_{i+\alpha_j-\alpha_k}^{(1)} \right) = y_i^u + \sum_{j=1}^{J} \psi(j,i)^{(1)} \quad (2.2)$$

$$\lambda_i^{(\mu)} = y_i^u + \sum_{j=1}^{J} \left(y_{i+\alpha_j}^p \diamond \sum_{k=1}^{j-1} \diamond \lambda_{i+\alpha_j-\alpha_k}^{(\mu-1)} \diamond \sum_{k=j+1}^{J} \diamond \lambda_{i+\alpha_j-\alpha_k}^{(\mu)} \right) = y_i^u + \sum_{j=1}^{J} \psi(j,i)^{(\mu)} \quad (2.3)$$

Où :

1. μ est l'indice d'une itération au sein du décodeur à seuil itératif
2. i représente l'indice temporel
3. p signifie parité
4. u signifie information

2.1.5.2 Opérateur ADD-MIN

L'opérateur "$\diamond$" que l'on retrouve dans les équations (2.2) et (2.3) se nomme l'opérateur ADD-MIN. Il fut utilisé par Cardinal et al. afin de simplifier l'implémentation du décodeur. Cet opérateur provient de la simplification du logarithme d'une somme de plusieurs fonctions exponentielles décroissantes [7]. La fonction de cet opérateur est définie selon l'équation suivante :

$$\sum_{k=1}^{M} \diamond B_k = B_1 \diamond B_2 \diamond ... \diamond B_M = (-1)^{M+1} \times \min_{k=1}^{M} |B_k| \times \prod_{k=1}^{M} signe(B_k) \quad (2.4)$$

Cet opérateur recherche la valeur minimale en valeur absolue parmi M entrées et calcule le signe selon le produit du signe des entrées multipliées par -1 si le nombre d'entrées est pair. Dans la section suivante, un deuxième opérateur qui joue un rôle capital dans la performance requise du décodeur à seuil itératif est présenté.

2.1.5.3 Rôle du pondérateur

L'introduction du coefficient de pondération à la sortie de chacune des itérations du décodeur améliore les performances de correction d'erreurs. Ce pondérateur multiplie par un coefficient "a" la sortie $\lambda^{(\mu)}$ de chacune des itérations du décodeur itératif tel que montré par l'équation (2.5). Cette valeur "a", est prédéterminée et reste fixe tout au cours du processus de correction d'erreurs. Sa valeur est définie dans la plage $]0,1]$ et des études précédentes montrent que pour un même coefficient utilisé pour l'ensemble des itérations, la valeur qui minimise le taux d'erreurs est située proche de 0.2 [7].

$$\lambda_i^{(\mu)} = a y_i^u + a \sum_{j=1}^{J} \left(y_{i+\alpha_j}^p \diamond \sum_{k=1}^{j-1} \diamond \lambda_{i+\alpha_j-\alpha_k}^{(\mu-1)} \diamond \sum_{k=j+1}^{J} \diamond \lambda_{i+\alpha_j-\alpha_k}^{(\mu)} \right) = a \left[y_i^u + \sum_{j=1}^{J} \psi(j,i)^{(\mu)} \right] \tag{2.5}$$

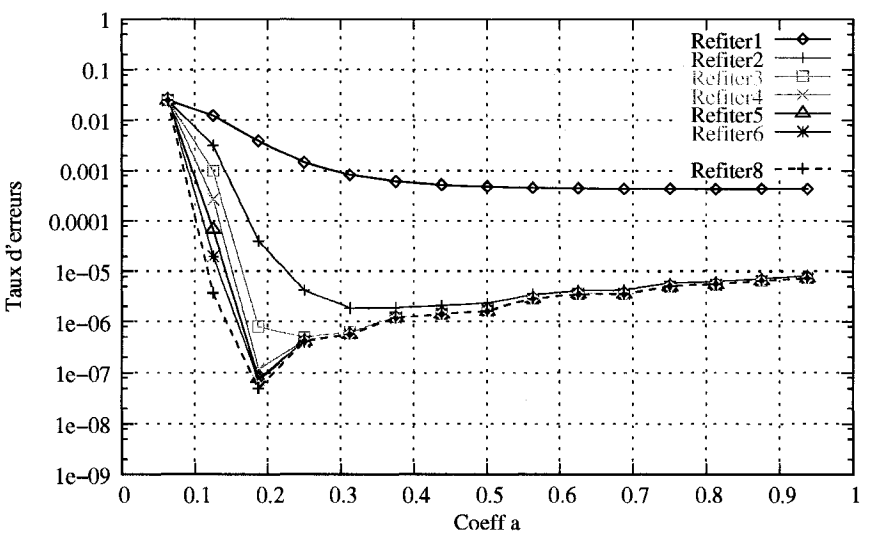


Figure 2.4: Variation du taux d'erreurs en fonction d'un coefficient de pondération uniforme (J=10, E_b/N_o=4.5 dB)

La figure 2.4 montre la variation du taux d'erreurs en fonction du coefficient de

pondération "a". Ces résultats furent obtenus avec une implémentation du décodeur à seuil itératif avec une quantification de 3 bits pour les symboles d'information et de parité et une résolution des mots internes de 8 bits pour le décodeur à seuil itératif. Il est à noter que le coefficient de pondération en abscisse représente une valeur uniforme attribuée pour toutes les itérations. Ce qui est important à retenir de ce graphique est que le taux d'erreurs varie d'un facteur de 2.39×10^5 entre la valeur optimale de (0.1875) et la valeur de (0.0625). Aussi, la dérivée de celle-ci ne varie pas de manière abrupte pour des valeurs supérieures au coefficient de valeur optimale soit 0.1875.

2.1.5.4 Structure d'une itération du DSI

En entrant plus en profondeur dans le système, on voit que le décodeur à seuil itératif se compose de plusieurs unités de base, qui sont des décodeurs (itérations) cascadés et ayant une structure similaire. On peut voir à la figure 2.5 l'interconnexion entre les différentes itérations. Le premier point à remarquer est que chacune des itérations utilise les symboles d'information y_i^u et de parité y_i^p. Également, chacune des itérations utilise le résultat de l'équation précédente sauf naturellement pour la première itération. Toutes ces valeurs doivent être préservées pendant un laps de temps suffisant et elles déterminent la consommation totale de mémoire nécessaire au processus itératif. Le terme "à seuil" signifie la méthode de décision du bit subissant la correction d'erreurs. Cette décision est très simple et si la valeur de sortie de la dernière itération est positive, la valeur du bit correspondant est égale à 1 et vice-versa.

La figure 2.6 montre l'architecture d'une itération du DSI proposée par Cardinal dans sa thèse de doctorat [7]. Ce schéma représente la structure d'une itération intermédiaire, où le code convolutionel employé est $G = \{0, 1, 4, 6\}$ soit J=4. Cette structure se divise en 4 parties matérielles distinctes, soit le registre à décalage conser-

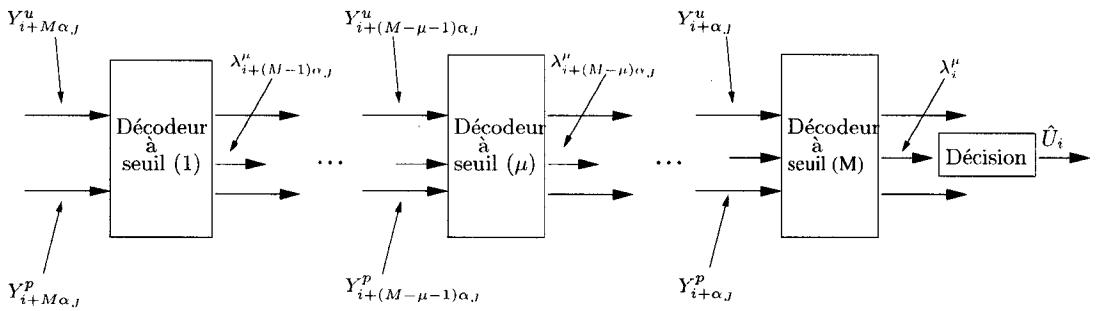


Figure 2.5: Architecture du décodeur à seuil itératif

vant l'information (Z_1), le registre à décalage conservant les résultats de l'itération précédente $\lambda^{(\mu-1)}$ (Z_2), la boucle de rétroaction (Z_3) et finalement le coeur de logique combinatoire qui est composée de J-1 opérateurs add-min, d'un additionneur à J+1 entrées et d'un multiplicateur qui implémente le coefficient de pondération (Z_4). Les différents opérateurs add-min de Z_4 et la boucle de rétroaction de Z_3 forment les différents termes $\psi(i,j)$ de l'équation (2.5). Les termes $\psi(i,j)$ sont tous acheminés à l'additionneur. La boucle de rétroaction provient de la combinaison des termes $y^p_{i+\alpha_J}$ et $\lambda^{(\mu)}$ de chaque terme $\psi(i,j)$. Ainsi, le terme $\psi(i,4)$ ne comprend aucun terme $\lambda^{(\mu)}$ et il est formé à partir des termes $y^p_{i+\alpha_J}$ et $\lambda^{(\mu-1)}$. Le terme $\psi(i,3)$ comprend l'expression $\lambda^{(\mu)}_{i+\alpha_3-\alpha_4} \diamond y^p_{i+\alpha_3}$ qui correspond au signal R_2. Ce terme correspond en fait au terme $\lambda^{(\mu)}_i \diamond y^p_{i+\alpha_4}$ retardé de $\alpha_3 - \alpha_4$ coups d'horloge. $\psi(i,2)$ est également constitué de cette manière en utilisant le signal R_3. Pour constituer R_3, ce terme est exprimé par l'expression $y^p_{i+\alpha_2} \diamond \lambda^{(\mu)}_{i+\alpha_2-\alpha_4}$. Si l'on retarde le terme $\lambda^{(\mu)}_i \diamond y^p_{i+\alpha_4}$ de $(\alpha_4 - \alpha_2)$ cycles d'horloge, soit de $(\alpha_4 - \alpha_3)$ cycles en premier et ensuite de $(\alpha_3 - \alpha_2)$ cycles, on retrouve le terme $y^p_{i+\alpha_2} \diamond \lambda^{(\mu)}_{i+\alpha_2-\alpha_4}$. La boucle de rétroaction est présente afin de réutiliser les termes R_j afin de réaliser une économie de complexité matérielle. Il aurait été possible de construire un décodeur avec la même fonctionnalité qui conserverait les valeurs $\lambda^{(\mu)}$ dans un registre à décalage. Toutefois, cette architecture consommerait plus de mémoire et de ressources matérielles.

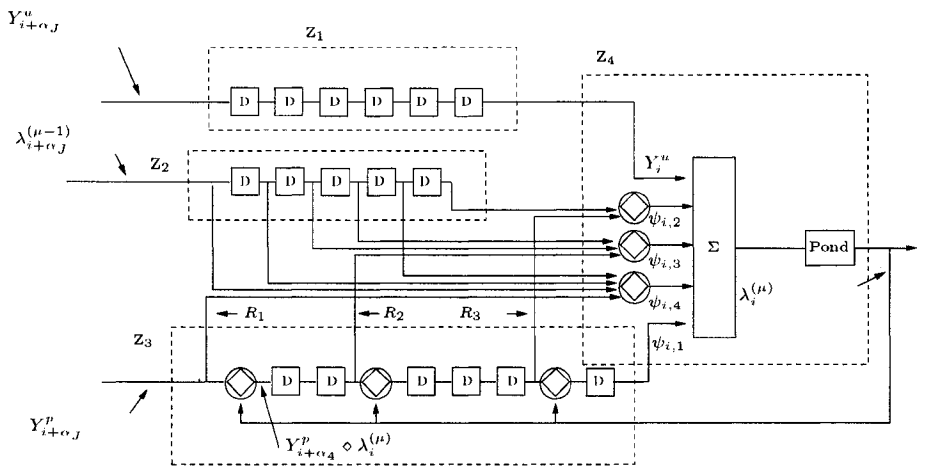


Figure 2.6: Architecture d'une itération (μ) du DSI où G={0,1,4,6}

2.2 Notions matérielles

2.2.1 Technologie FPGA

Il y a environ 15 ans, une nouvelle classe de composants électroniques voyait le jour. Ce nouveau type de composant répondait à la demande croissante de circuit programmable servant principalement comme logique d'interface "glue logic". Depuis l'utilité des FPGA a connue une diversification notable. Aujourd'hui, grâce à la constante réduction d'échelle des transistors, les FPGA possèdent des fonctionnalités plus sophistiquées telles que des quantités de mémoire de quelques méga-octets, des modules de gestion d'horloges et des entrées-sorties de hautes performances qui supportent un nombre impressionnant de protocoles de signalisation tel que PCI (Peripheral Component Interconnect), LVDS (Low Voltage Differential Signal) et Rocket-IO. Également quelques FPGA possèdent un ou plusieurs processeurs intégrés. Tel est le cas pour la famille Virtex-II Pro de Xilinx qui possède 4 processeurs Power-PC. Avec autant de fonctionnalités intégrées, les FPGA sont devenus une catégorie distincte de composants offrant la possibilité d'implémenter des circuits logiques d'une complexité supérieure comparativement aux autres types de circuit programmable. Les principales compagnies fabricants ce type de composant sont Xilinx [21] et Altera [19].

Les FPGA du manufacturier Xilinx se divisent en plusieurs familles tel que Spartan-II et Virtex-II. Les différentes familles se distinguent par les différentes fonctionnalités spécialisées que celles-ci offrent. Toutes les familles possèdent des éléments communs soit des CLB (Configurable Logic Block), de bloc de mémoire RAM (Block RAM) (RAM, Random-Access Memory) et de IOB (bloc d'entrées/sorties, Input/Output Block). Pour chacune de ces familles, un éventail de composants possédant différentes quantités de ces ressources sont disponibles à différents prix.

La principale particularité des FPGA est la simplicité du flot de conception.

Contrairement au ASIC (Application-Specific Integrated Circuit) où un dessin de masques doit être réalisé, le FPGA possède déjà une implémentation physique et les outils de conception utilisent les ressources disponibles de celui-ci et interconnectent ces différentes ressources grâce à un réseau configurable d'interconnexion préexistant. Le FPGA offre la même flexibilité au niveau matériel que le CPU (Central Processing Unit) offre au niveau logiciel. Le FPGA constitue la machine implémentant tout circuit numérique synchrone et le CPU constitue la machine qui peut exécuter n'importe quelle suite d'instructions qu'il implémente. Le CPU offre une structure matérielle disponible pour l'exécution d'instruction, et il suffit à l'utilisateur d'indiquer au processeur quelles instructions exécutées pour accomplir la fonctionnalité voulue. Dans le même ordre d'idée, le FPGA met à la disposition de l'usager des ressources matérielles où les contenus de différentes mémoires et la connectivité des différents éléments sont configurables. Une telle flexibilité apporte plusieurs avantages à ce type de composant.

Avec une capacité d'implémentation matérielle grandissante rendue possible par la réduction de la taille des transistors, il est possible d'implémenter de plus en plus d'applications sur FPGA. La technologie FPGA devient un compromis de plus en plus intéressant au niveau commercial comparativement au traditionnel ASIC. De par leur configurabitlité supérieure et leur simplicité de conception qui est de beaucoup inférieure à celle des ASIC, il constitue une alternative qui réduit les risques financiers encourus et le temps de mise en marché d'un nouveau produit. Également, contrairement au ASIC, il est possible de reconfigurer les composants FPGA une fois le produit rendu chez le consommateur. Cette caractéristique est un avantage majeur pour diminuer les coûts du support à la clientèle en distribuant seulement une mise à jour de la configuration matérielle. Il est également possible de vendre un produit et d'ajouter par la suite des fonctionnalités même lorsque le produit est rendu

chez le client. Également, le principal avantage de ce composant est qu'il est possible de vérifier rapidement la fonctionnalité de l'implémentation et il ne requiert pas la fabrication d'un circuit intégré. Le DSI fut implémenté sur ce type de composants.

En résumé, la technologie FPGA représente une alternative très attrayante pour les compagnies sur plusieurs aspects comparativement au traditionnel ASIC. Néanmoins, il n'y a pas que des avantages. Leur coût par unité est plus élevé que celui des ASIC, ce qui constitue un coût récurrent beaucoup plus important. Le ASIC possède un faible coût de production et son coût global d'ingénierie est élevé. Pour une production de masse, la solution FPGA est beaucoup plus dispendieuse. Également, les ASIC de par leur nature plus compacte pour une même technologie peuvent atteindre des fréquences d'horloge beaucoup plus élevées. Cependant pour les besoins de recherche, l'utilisation d'un FPGA permet de réduire de beaucoup le temps de conception et le temps de livraison d'un premier prototype.

2.2.1.1 Architecture des FPGA Xilinx

Plusieurs familles de composants existent chez la compagnie Xilinx. Tous ont une architecture commune et quelques fonctionnalités spécialisées distinguent les différentes familles. Pour implémenter le décodeur à seuil itératif, un FPGA de la famille Virtex-II fut utilisé (XC2V6000). La figure 2.7 montre la disposition physique d'un FPGA de cette famille. Ainsi 5 composants principaux qui constituent le FPGA sont montrés soit les cellules IOB, les CLB, les blocs de mémoire RAM, les Mult18x18 et les DCM (Ditital Clock Management). Les IOB sont naturellement disposés au pourtour du composant électronique proche des plots. Leur architecture peut être modifiée ainsi que leur alimentation afin de supporter différents protocoles de signalisation. Les CLB sont les blocs de logique reconfigurables (Configurable Logic Block) et constituent les éléments où la logique combinatoire est implémentée. Les composants Block

RAM sont des mémoires de tailles prédéterminées (18 Kbits) où la plage d'adresse et le nombre de bits par adresse sont configurables. Les composants Mult18x18 sont des multiplicateurs rapides câblés. Il existe un multiplicateur pour chacun des blocs de mémoire RAM disponibles. Également, les modules DCM (Digital Clock Manager) sont spécialisés dans la génération et la synchronisation d'horloge.

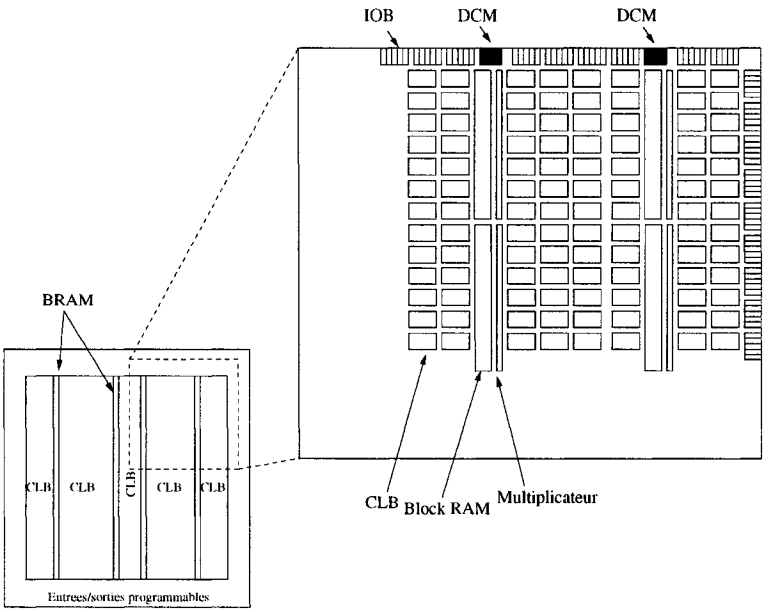


Figure 2.7: Architecture du FPGA Virtex-II

2.2.1.2 Architecture des CLB

Les CLB sont la pierre angulaire de l'implémentation des réseaux de logiques combinatoires. L'architecture d'un CLB d'un composant de la famille Virtex-II est constituée de 4 "slices". Tel que montré sur la figure 2.8, les 4 "slices" d'un CLB sont divisés en deux colonnes où une chaîne de retenue est disponible pour chacune des paires. Cette chaîne de retenue est particulièrement intéressante pour l'implémentation d'additionneurs et de comparateurs rapides au sein du FPGA.

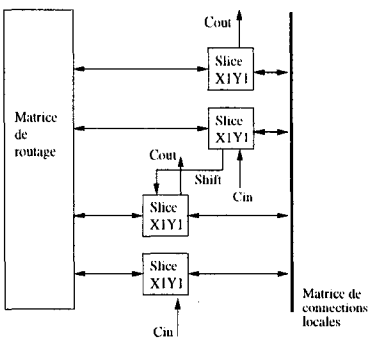


Figure 2.8: Architecture d'un CLB du FPGA Virtex-II

Un "slice" comporte deux structures similaires que l'on nomme cellule logique (LC, Logic Cell). L'architecture d'un "slice" comprend deux LUT (Look-Up Table). Il est bien important de savoir que cet élément LUT est une mémoire 16 x 1 bits qui est programmée avec le contenu approprié pour implémenter la fonction combinatoire désirée. Les ressources combinatoires d'un FPGA sont exprimées selon cette unité. En regardant le contenu du réseau logique d'un CLB, aucun élément ne possède autant de capacité d'implémentation qu'un LUT à l'intérieur d'un CLB. Ce type de FPGA est dit orienté LUT. Également, ce LUT peut être transformé en un registre à décalage d'une longueur de 16 bits (primitive SRL16) ou en une mémoire RAM de 16 x 1 bits (block Select-RAM).

Le restant de la logique disponible d'un LC ajoute des fonctionnalités spécifiques nécessaires à l'implémentation logique de fonction combinatoire rapide et de grande densité. La porte logique MUXCY permet d'implémenter une chaîne de retenu rapide entre "slices" afin de créer localement (c'est-à-dire entre les "slices" sans retourner par la matrice de routage) une chaîne de retenu. Il est possible également d'ajouter un registre à la sortie d'un LUT. Un LUT étant limité à 4 entrées ($\log_2(16)$), il est possible d'utiliser le multiplexeur MUXFX pour implémenter à l'intérieur d'un même "slice" une fonction logique ayant jusqu'à 8 entrées tout en minimisant le délai de

routage.

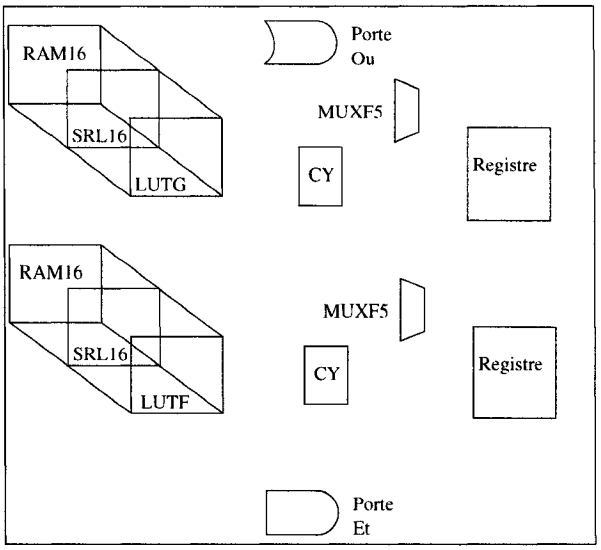


Figure 2.9: Architecture d'un "slice" de la famille Virtex-II

2.2.1.3 Éléments de mémoire

Il existe 3 types de mémoires dans le FPGA. Deux types de mémoire sont situées à l'intérieur des CLB soit un LUT et un registre par LC. Le FPGA XC2V6000 possède en tout 33792 "slices" et donc 2 fois plus de LUT. Il existe également des modules de mémoire de large capacité que l'on nomme bloc de mémoire RAM (block RAM). Pour la famille Virtex-II, les blocs de mémoire RAM possèdent une capacité de 18 Kbits. La configuration de la plage d'adressage et du nombre de bits par adresse peut être sélectionnée parmi quelques configurations déjà existantes. Également avec ces mémoires à 2 ports, il est possible d'implémenter une large gamme de fonctionnalités. Par exemple, ce type de mémoire est très utile pour des applications multi-processeurs où un élément mémoire est partagé entre 2 processeurs.

2.2.1.4 Délai critique

Plusieurs facteurs influencent la fréquence maximale de l'horloge d'un circuit numérique. Tous ces facteurs peuvent être dérivés de l'équation (2.6) [17]. Cette équation met en relation le temps de propagation (t_p) d'un inverseur CMOS (figure 2.11) de différents paramètres. Lors de la conception avec FPGA où l'alimentation (V_{DD}) et la technologie sont fixées (K qui est une constante dépendante de la technologie et V_{TH} qui est la tension de seuil d'un transistor), le facteur principal qui influence le délai de propagation t_p est la charge capacitive (C_L). Au niveau de cellules CMOS, la sortance (fan-out) représente le nombre d'entrées de cellules CMOS connecté à une sortie CMOS. La sortance a une incidence directe sur t_p. Plus la sortance est élevée, plus la charge capacitive est également élevée, ce qui augmente considérable le temps de propagation. Également, le nombre de cellules CMOS utilisées entre deux registres constitue un facteur important qui limite la fréquence maximale d'horloge. Deux techniques permettent d'augmenter la fréquence d'horloge lorsque les différents facteurs influençant la fréquence ont été considérés. Ces deux techniques sont présentées à la prochaine section.

$$t_p = \frac{1.6\ C_L}{K \times (V_{DD} - V_{TH})} \tag{2.6}$$

2.2.2 Optimisation matérielle : réajustement de l'ordonnancement et l'insertion de pipeline

Le réajustement de l'ordonnancement (retiming) et l'insertion de pipeline sont deux notions intimement liées. Tel qu'exposé dans la section précédente, les différents critères qui limitent la fréquence maximale d'opération d'un circuit numérique ont été

énumérés. La technique d'opération pipeline vise à découper le chemin critique à l'intérieur d'une architecture ciblée afin d'augmenter de manière significative la cadence d'horloge. Cette technique de pipelinage consiste à introduire des registres permettant de diviser le délai de la logique combinatoire du chemin critique. L'introduction de ces éléments de mémoire augmente la fréquence d'opération par un facteur égal au nombre de registres introduit dans le chemin critique plus 1. Par contre, l'introduction d'un élément mémoire implique que le résultat de l'opération sera disponible après un nombre de cycles égal au nombre de bascules ajoutées. Pour appliquer correctement la notion d'insertion de pipeline, il faut respecter méticuleusement l'ordonnancement des données afin que le circuit après l'introduction d'étages pipeline respecte toujours la fonctionnalité du circuit original. L'ajout de registres à l'intérieur d'un circuit comportant une boucle de rétroaction est souvent problématique.

La technique du réajustement de l'ordonnancement permet justement, à partir d'un circuit numérique, de diviser le délai critique en utilisant les bascules déjà présentes au sein du design. En appliquant cette technique, le déplacement de registres respectera la fonctionnalité du circuit original. Une règle régie la méthode du réajustement de l'ordonnancement. Celle-ci concerne le déplacement de registres au travers d'un opérateur de logique combinatoire. Pour déplacer les registres à l'entrée d'un opérateur vers la sortie, il suffit de retirer un registre de chaque entrée pour en ajouter un à chaque sortie (2.10). Le résultat de l'opération du réajustement de l'ordonnancement est montré à la figure 2.10. Il est également possible de généraliser la théorie de l'insertion d'étages de pipeline en utilisant le théorème du réajustement de l'ordonnancement. En ajoutant des registres à l'entrée de l'opérateur concerné et si par la méthode du réajustement de l'ordonnancement, il est possible de déplacer ces registres jusqu'au chemin critique afin de réduire le délai de celui-ci, il est également possible d'ajouter des registres directement par l'insertion de pipeline. Cette

méthode du réajustement de l'ordonnancement est plus systématique et moins sujette à l'erreur.

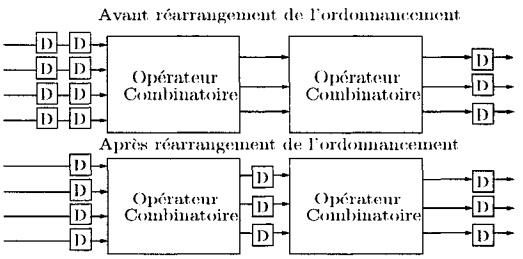


Figure 2.10: Exemple du réajustement de l'ordonnancement

2.2.2.1 Puissance consommée

La puissance consommée d'un circuit CMOS se divise en deux catégories soit : dynamique et statique. La puissance dynamique consommée est plus importante que celle statique. L'équation (2.8) montre les 4 sources principales de consommation de puissance.

$$P_{TOT} = P_{sous-seuil} + P_{fuite} + P_{court-circuit} + P_{commutation} \tag{2.7}$$

La puissance dynamique est principalement caractérisée par $P_{commutation}$ qui est la puissance dépensée pour charger les différentes capacités. $P_{court-circuit}$ constitue la puissance dissipée lors de la commutation d'une cellule CMOS. Les principaux facteurs qui constituent la consommation de la puissance statique est la puissance de consommation sous le seuil ($P_{sous-seuil}$) et la puissance dissipée par le courant de fuite (P_{fuite}) lorsque l'entrée et la sortie d'une cellule CMOS sont stables.

La charge capacitive des circuits numériques constitue le principal élément consommateur de puissance dynamique. Cette consommation est dite dynamique car elle

charge les différents condensateurs parasites lorsque la sortie d'une cellule CMOS transite de l'état $0 \rightarrow 1$. La figure 2.11 montre le circuit d'un inverseur en technologie CMOS. La capacité C_L comprend l'ensemble des capacités qui constituent le circuit, des différentes capacités de ligne et la capacité des entrées à laquelle la sortie de la cellule CMOS est reliée. Aussi, une importante partie de la puissance dynamique dissipée provient de la distribution du signal d'horloge (généralement $\frac{1}{3}$ de P_{TOT}) où le nombre d'entrées de portes logiques connectées à ce signal devient facilement très élevé.

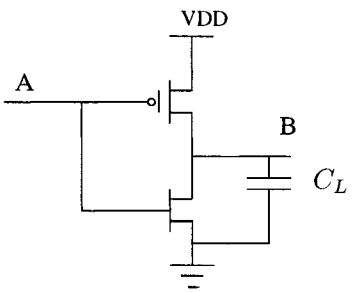


Figure 2.11: Circuit d'un inverseur CMOS

L'équation 2.8 décrit la relation entre la puissance dynamique et la capacité totale C_L d'un circuit électronique, de l'alimentation V_{DD} et de la fréquence de l'horloge opération ($f_{horloge}$) [17].

$$P_{dyn} = \alpha \times C_L \times V_{DD}^2 \times f_{horloge} \tag{2.8}$$

La capacité C_L est un paramètre intrinsèque du circuit et afin de limiter la consommation de puissance dynamique, il est donc nécessaire de diminuer au maximum la valeur de cette capacité. Plusieurs techniques ont été développées afin de réduire la puissance consommée. Une technique utilisée dans les processeurs actuels diminue

l'alimentation du circuit. La puissance consommée diminue par le carré du facteur de réduction de la tension de l'alimentation. Une diminution de la fréquence d'horloge est également une technique employée pour diminuer la consommation en puissance. Les nouveaux ordinateurs portables utilisent cette technique pour prolonger le budget énergétique de la batterie. Également, α représente le taux d'activité de la sortie CMOS. Presque toujours, une sortie CMOS aura un taux d'activité maximal à 50% de la fréquence d'horloge.

La puissance statique dépensée pour un circuit CMOS est principalement due au transistor en mode sous-seuil (subthresold). La commutation non idéale des transistors NMOS (Negative-Channel Metal Oxide Semiconductor) et PMOS (Positive-Channel Metal Oxide Semiconductor), lorsque la tension de porte est moins que la tension de seuil, induit une courant résiduel circulant à l'intérieur du transistor. Également, il existe un courant de fuite causé par la diode de jonction de chaque transistor qui est en mode polarisation inverse "reverse-biased" lorsque les transistors ne conduisent pas. La densité du courant de fuite varie entre 10 et 100 $pA/\mu m^2$. Pour un circuit comptant quelques millions de transistors, d'une surface totale de 0.5 μm^2 et d'une tension d'alimentation de 1.8 V (V_{DD}) [17], on peut s'attendre à obtenir une puissance de consommation statique de l'ordre de 0.09 W selon l'équation (2.9).

$$P_{statique} = I_{statique} \times V_{DD} \tag{2.9}$$

2.2.3 Plateforme ARM integrator

La plateforme ARM integrator offre un environnement approprié pour implémenter de manière rapide un design comportant une partie logicielle et une autre maté-

rielle. Cette plateforme se divise en quatre parties soit l'ordinateur hôte du système, un module central "core module" qui intègre un processeur ARM servant à l'implémentation de la partie logicielle, un module logique "logic module" qui possède un FPGA de haute capacité et une carte mère qui interconnecte les deux parties ensemble. Une gamme de processeurs ARM et de FPGA sont disponibles avec cette plateforme. Présentement un processeur ARM7TDMI et un FPGA de la famille Virtex-II de Xilinx (XC2V6000) sont utilisés. Ce FPGA est le deuxième composant de cette famille de plus grande capacité. Pour communiquer, le processeur ARM utilise un bus qui se nomme AHB (Advanced Hardware Bus) afin de communiquer avec le FPGA. Ce bus est bidirectionnel et permet autant au FPGA qu'au processeur ARM d'initialiser une communication.

Cette plateforme est vraiment intéressante de part la quantité et la diversité des fonctionnalités qu'elle offre. Il est possible de transformer cette plateforme en un ordinateur conventionnel en utilisant le micro-noyau Linux fournis par la Société Canadienne en Microélectronique (SCM) . Également, un port à haut débit d'information (PCI, Peripheral Component Interconnect) peut servir à transférer rapidement et efficacement de l'information au monde externe. Par exemple, une carte réseau PCI peut être utilisée pour communiquer des résultats par Internet. Aussi cette plateforme possède des ports d'entrées-sorties sérielles (RS-232), des ports PS2, quelques diodes électroluminescentes et quelques boutons poussoirs au choix. Il est possible d'utiliser plusieurs modules centraux faisant de cette plateforme un système multiprocesseurs. Plusieurs modules logiques peuvent être empilées afin d'augmenter la capacité d'implémentation matérielle de la plateforme. Il est toujours possible pour les différents modules de communiquer entre elles. L'ensemble des communications entre les différents modules logiques et modules centraux se font par l'intermédiaire de l'AHB.

Une grille de prototypage est disponible sur chacune des modules logiques. Cette grille ajoute la possibilité de souder de la circuitrie sur la carte. Une connectivité déjà établie avec le FPGA permet de communiquer avec celui-ci. Par exemple, il est possible de transmettre des données aux différents IP inclus à l'intérieur du FPGA avec un composant comme le FT245BM de FTDI [20] pour transmettre de l'information par un bus USB (Universal Serial Bus). Ce composant communique avec le FPGA par les différents plots déjà connectés à la grille de prototypage.

Le module logique ne supporte pas intrinsèquement la communication AHB. Pour que le processeur ARM soit en mesure de communiquer avec le FPGA, l'implémentation du protocole AHB doit être fait à l'intérieur du FPGA. Les différents modules matériels nécessaires au AHB sont fournis avec la plateforme ARM integrator. Toutefois, aucun module de banc d'essai VHDL n'est fourni avec la plateforme afin de simuler le bon fonctionnement de l'ensemble de l'implémentation AHB. Également aucun module maître AHB n'est disponible. Ce module est essentiel au FPGA afin d'initialiser une communication avec le processeur ARM. Ce sont ces 2 aspects qui constituent les inconvénients majeurs de cette plateforme. La figure 2.12 montre la configuration de base donnée avec la plateforme afin d'implémenter le protocole AHB.

Jusqu'ici, différentes notions de conception de circuits numériques synchrones ont été présentées. Quels sont les différents avantages qu'une implémentation asynchrone du décodeur à seuil itératif peut donner ? La prochaine section introduit quelques notions de conception asynchrone et donne une conclusion quant à la viabilité d'une conception asynchrone du DSI.

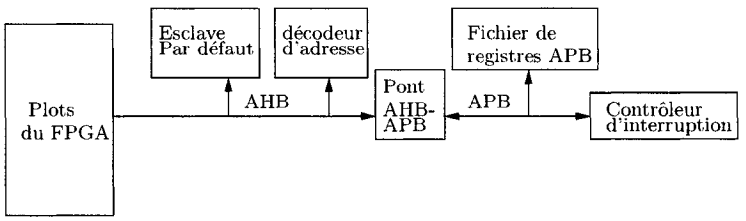


Figure 2.12: Configuration intérieure AHB du FPGA

2.2.4 Notion de circuit asynchrone

Au tout début de la conception de circuits numériques avec des transistors, deux modes de conception étaient présents soit la conception de circuits asynchrones et de circuits synchrones. Avec la disponibilité d'un grand nombre de transistors, la plus grande facilité de conception des circuits synchrones et les pressions économiques ont amenées à favoriser la méthode de conception de circuits synchrones. Dû à la réduction d'échelle des transistors, un nombre toujours croissant de transistors sont disponibles. Également la puissance a crue à un tel point que plusieurs technologies synchrones ont été marginalisés par l'utilisateur d'une technologie moins gourmande énergétiquement. La technologie de conception NMOS consomme une partie non négligeable de puissance statique tandis que la technologie CMOS, malgré qu'elle consomme plus de transistors pour une même fonctionnalité, a une puissance statique consommée de loin inférieure. Pour des circuits de haute performance CMOS d'aujourd'hui (cadence d'horloge élevée), dont la puissance dissipée est proportionnelle à cette fréquence, la dissipation de cette puissance est devenue problématique. Le meilleur exemple est celui du microprocesseur utilisé dans les ordinateurs personnels. Ils sont cadencés à une fréquence d'horloge de 3 GHz. Ces CPU hautes performances dissipent la puissance générée à l'aide de dissipateur de puissance et d'un ventilateur afin de ne pas endommager le circuit intégré.

La conception asynchrone se distingue de celle synchrone par le fait que cette der-

nière n'utilise pas d'horloge pour synchroniser le fonctionnement du circuit. Au lieu d'employer une horloge, elle utilise un protocole de transfert de données ("handshaking"). Cette technique de conception consomme de la puissance uniquement lorsqu'il y a des données à traiter contrairement au circuit numérique synchrone qui consomme de la puissance en tout temps. Comparativement à la technique de masquage d'horloge, la conception asynchrone agit de la même manière mais avec une pleine granularité d'activation et de désactivation du circuit. La consommation de puissance anticipée avec le décodeur à seuil itératif est élevée dû au nombre important d'éléments mémoires nécessaires à l'implémentation. De part l'économie importante reportée dans la littérature [14], il est intéressant de considérer cette méthode de conception de circuit numérique.

La facilité de conception et les avantages des circuits synchrones deviennent de plus en plus discutable avec la disponibilité d'un nombre de transistors grandissant. Premièrement, la conception synchrone demande un temps considérable pour bien équilibrer le délai de chacun des étages pipelines. Également, la fréquence d'horloge doit être égale à l'inverse du pire délai de l'ensemble des étages pipelines. La conception asynchrone offre des avantages qui deviennent de plus en plus importants avec la complexité grandissante des circuits. Des performances plus élevées sont envisagées, car les données sont traitées immédiatement lorsque le module reçoit des données et s'ilne traite aucune données au moment qu'il les reçoit. Avec la logique asynchrone, il n'existe plus de période morte tel que l'on retrouve en logique synchrone. De plus, le délai peut s'avérer plus ou moins long selon les données traitées. Il existe certains protocoles qui gèrent le traitement des données indépendamment du délai de traitement des données et de l'hétérogénéité de l'implémentation, et de ce fait, cette méthode de conception est beaucoup plus indépendante au changement de technologie d'implémentation.

Malgré les différents avantages qu'offrent une implémentation d'un circuit asynchrone, peu de celles des outils sont présentement disponibles et la maturité de ceux-ci est loin des outils disponibles pour la conception de circuits synchrones. Par exemple, l'outil "Balsa" de l'Université de Manchester (Grande-Bretagne) a servi à concevoir les familles de processeurs asynchrones AMULET [12]. Ces processeurs sont des répliques asynchrones des processeurs ARM. Une implémentation matérielle malgré les propriétés de l'asynchrone ne conduit pas toujours à une implémentation moins énergivore. Cet aspect est bien introduit dans l'article [13]. L'implémentation de la fonctionnalité asynchrone conduit à une implémentation consommant plus de surface qu'une implémentation synchrone. Plus le protocole utilisé est indépendant des délais comme le protocole DIMS (Delay-Insensitive Minterm Systhesis), plus la surface consommée est importante. Le tableau 2.2 [13] présente la surface et le nombre de transistors consommés par l'implémentation de 2 opérateurs. Le ratio du nombre de transistors consommés par l'implémentation d'un comparateur 4 bits selon la logique synchrone et la logique asynchrone DIMS est de 6.7. Également, le tableau inclus des résultats de la technique de conception asynchrone DR-ST (Double-Rail Self-Timed) qui est largement connue dans ce domaine.

Pour contrer l'augmentation de la surface en terme de puissance, la technique d'implémentation asynchrone s'avère avantageuse si le taux d'activité global (α) de l'équation (2.8) du circuit en entier est suffisamment faible pour contrer l'accroissement de surface. Pour le DSI, étant donné que les opérations réalisées sont prédéterminées et que l'ensemble de la logique combinatoire est sollicitée à chaque cycle d'horloge (lorsqu'une nouvelle donnée est disponible à l'entrée du décodeur), une implémentation asynchrone du DSI n'est pas avantageuse.

Tableau 2.2: Tableau comparatif de l'implémentation synchrone et asynchrone d'un additionneur et d'un comparateur 4 bits (NT = nombre de transistors)

Circuits Implémentés		Synchrone	DR-ST	DIMS
Additionneur	mm^2	0.130	0.776	1.163
4 bits	NT	143	973	1297
Comparateur	mm^2	0.141	0.483	0.878
4 bits	NT	127	563	857

Chapitre 3

Conception, complexité matérielle et optimisation de l'architecture du DSI

3.1 Conception du DSI

3.1.1 Conception VHDL

Les choix les plus supportés et répandus présentement pour implémenter une architecture matérielle générique sont les languages VHDL [1] et Verilog [15]. Plusieurs autres méthodologies sont actuellement en développement. Les deux alternatives qui offrent le plus de potentiel sont la synthèse logique d'un algorithme décrit avec les languages SystemC ou C [18]. La synthèse logique de ces languages passe par une étape intermédiaire de traduction vers le language VHDL et ce traducteur constitue l'élément nouveau.

Les besoins anticipés en ressources matérielles du DSI sont importants. Pour des

performances de correction d'erreurs acceptables, soit un taux d'erreurs en dessous de 10^{-6} à des rapports signal à bruit relativement faible $E_b/N_o \leq 4$ dB, J doit être supérieur à 9. La quantité de mémoire nécessaire est proportionnelle à la longueur du code utilisé qui est de 1698 pour J=10. En raison de la longueur importante des codes utilisés, une complexité matérielle importante est anticipée. Un meilleur contrôle des ressources matérielles consommées est exercé avec une approche de description structurelle de l'architecture en VHDL.

L'architecture du DSI est composée d'un ensemble d'itérations similaires. La conception d'une itération est également divisée en 4 parties telle que décrit dans la section 2.1.5.4. Un ensemble d'unités fondamentales sont utilisées dans les différentes parties d'une itération. Au total 4 unités fondamentales sont retrouvées à l'intérieur du DSI soit : l'opération add-min, le registre à décalage, l'additionneur et le multiplicateur. Une telle division structurelle a permis une diminution de la consommation des ressources matérielles en optimisant l'implémentation des unités fondamentales. D'ailleurs cette approche structurelle vise surtout une optimisation au niveau des ressources mémoires. L'interconnexion de ces différents opérateurs permet d'assembler respectivement chacune des 4 parties qui constituent une itération. Ces 4 parties sont le registre à décalage de l'information (Z_1 de la figure 2.6), le registre à décalage des résultats de l'itération précédente (Z_2), la boucle de rétroaction (Z_3) et finalement le noyau de logique combinatoire de l'algorithme (Z_4). À titre de rappel, la première et la dernière itérations comptent 3 parties. Pour la première itération, il n'y a pas de registre à décalage qui contient les résultats de l'itération précédente et la dernière itération ne contient pas de registre à décalage pour les symboles quantifiés de parité.

La configurabilité de l'IP développé est grande. L'IP créé en VHDL supporte l'implémentation de n'importe quel CSO^2C-WS pour un taux de codage $\frac{1}{2}$ et un nombre d'itérations arbitraire. Également, la résolution des mots utilisés dans l'architecture

peut être paramétrisée. Cette flexibilité est disponible seulement avant la synthèse logique du décodeur.

Une telle configurabilité du DSI nécessite la conception d'unités fondamentales flexibles. Le language VHDL permet une telle configurabilité et cache à l'usager, la configuration nécessaire des différents sous-modules utilisés. À partir des paramètres spécifiés par l'usager, les paramètres nécessaires à la configuration des sous-modules sont dérivés. Il suffit à l'usager de spécifier le nombre d'itérations désiré, le CSO^2C-WS et la résolution binaire des différents mots de l'architecture. Pour implémenter n'importe quel code, la connectivité entre les différents modules doit être paramétrisable. Le language VHDL est assez flexible pour supporter ce niveau de configurabilité. Plusieurs fonctions fut développées pour calculer les différents paramètres nécessaires aux sous-modules qui constituent une itération. Chacune des itérations demande une liste précise de paramètres et dans les sections suivantes sont détaillées les formules permettant de calculer ceux-ci.

Dans ce chapitre, une description des différents modules qui composent une itération et de leur configurabilité est réalisée. L'implémentation des différents opérateurs est décrite et une discussion quant à la représentation binaire à utiliser ainsi que l'approche de conception des différents opérateurs est discutée. Des analyses de complexité matérielle et de délai combinatoire sont présentées pour les différents opérateurs et le DSI. Finalement 2 techniques d'optimisation sont discutées permettant de réduire la complexité matérielle et de diminuer le délai du chemin critique.

3.2 Implémentation des différents opérateurs

Plusieurs opérateurs sont nécessaires à l'intérieur du DSI. Selon l'équation 2.5, un additionneur et un nombre plus important d'opérateurs add-min sont requis. Cette

équation représente une itération du DSI et plusieurs itérations sont nécessaires afin d'obtenir des performances de correction d'erreurs satisfaisantes. Le terme "a" est le coefficient de pondération dont le rôle est vital au sein du DSI. Il permet d'obtenir de meilleures performances de correction d'erreurs à de faibles SNR en pondérant les sorties des itérations. Cet opérateur est un multiplicateur à deux entrées où la première entrée est la sortie de l'additionneur et la deuxième entrée est le coefficient de pondération. Dans cette section, la configurabilité des différents opérateurs nécessaires à la création d'un IP générique du DSI est décrite.

3.2.1 Additionneur et opérateur add-min

La configurabilité de l'additionneur et de l'opérateur add-min doit être suffisante pour supporter l'implémentation d'un IP générique du DSI. Afin de supporter n'importe quel CSO^{2}C-WS et pour un taux de codage égal à $\frac{1}{2}$, le nombre de ports d'entrée de l'additionneur et de l'opérateur add-min doivent être paramétrables. Aussi le nombre de bits par port est paramétrable afin d'éviter de mauvaises performances de correction d'erreurs causées par une résolution déficiente des différents opérateurs au sein de l'architecture. La figure 3.1 présente un opérateur générique qui a servi de modèle pour l'implémentation de l'additionneur et de l'add-min.

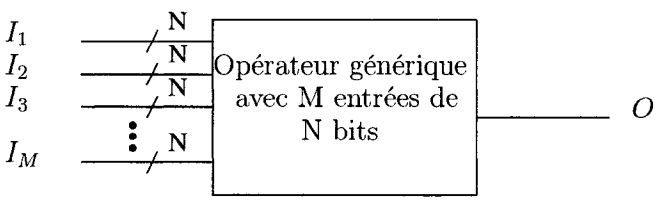


Figure 3.1: Schéma d'un opérateur générique

Plusieurs implémentations génériques de l'opérateur add-min et de l'additionneur ont été réalisées. Une structure arborescente composée d'opérateurs à 2 ports de N bits

fut développée pour implémenter un opérateur avec un nombre de ports arbitraire. Quelques fonctions VHDL sont nécessaires pour calculer par exemple la profondeur logique de la structure arborescente en fonction du nombre d'entrées et ainsi que la connectivité entre les différents opérateurs à 2 ports selon le nombre de ports voulus. Une version constituée d'opérateurs cascadés a été également réalisée, mais naturellement la stucture en arbre possède un délai combinatoire moins important. Au total N-1 opérateurs à deux ports sont nécessaires pour implémenter un opérateur à N ports.

L'implémentation du DSI demande l'adoption d'une représentation binaire pour les différents opérateurs logiques. Deux représentations binaires ont été retenues soit signe-amplitude et complément à deux. La détermination de la représentation binaire qui conduit à une implémentation matérielle et un délai de propagation moindre n'est pas évidente a priori. La représentation complément à deux avantage une implémentation plus efficace de l'additionneur tandis que des nombres exprimés selon la représentation signe-amplitude avantage l'opérateur add-min. L'addition de deux nombres en complément à deux ne demande aucune conversion des nombres négatifs en nombres positifs ou vice-versa tandis que la représentation signe-amplitude nécessite cette opération. Conséquemment l'implémentation d'un additionneur en complément à deux possède un délai et une complexité matérielle moins élevés. Avec la représentation signe-amplique, l'amplitude en valeur absolue est directement disponible sans conversion. Ceci avantage une implémentation plus rapide d'un opérateur add-min avec cette représentation. Une analyse de délai combinatoire et de la complexité matérielle permettra de déterminer la représentation la plus avantageuse à adopter. Différentes implémentations VHDL de l'opérateur add-min et de l'additionneur ont été réalisées afin d'itérer vers une version où le délai et la complexité matérielle sont minimisés.

Une analyse de délais combinatoires pour les différentes représentations sera pré-

sentée dans la section 3.5.2. Une attention particulière est à porter pour l'implémentation de l'opérateur add-min avec la représentation binaire complément à deux. Avec une résolution de 4 bits par port, la représentation complément à deux ne possède aucun équivalent positif de la valeur la plus négative soit "1000". Cette particularité de la représentation complément à deux a un impact non négligeable sur l'implémentation matérielle de l'opérateur. La conversion du nombre le plus négatif vers une valeur positive selon la méthode d'inversion connue, soit l'inversion de tous les bits du mot et l'addition de 1 à celui-ci, donne le même nombre (1000 $\rightarrow$ 0111 $\rightarrow$ 1000). Pour tenir compte de cette particularité, le résultat doit posséder un bit supplémentaire pour contenir l'amplitude du signal (1000$\rightarrow$ 11000 $\rightarrow$ 00111 $\rightarrow$ 01000).

Ainsi, l'implémentation d'un opérateur add-min à N bits par entrées en complément à deux demande des comparateurs de N bits pour tenir compte de la valeur maximale négative tandis que la représentation binaire signe-amplitude nécessite un comparateur avec (N-1) bits. Une complexité matérielle de $\frac{N}{(N-1)}$ fois plus grande est anticipée pour la représentation complément à deux comparativement à la représentation signe-amplitude. Cette augmentation de la complexité matérielle ne sert qu'à implémenter un pas de quantification supplémentaire. Également cette particularité se produit seulement lorsque toutes les entrées sont de valeurs maximales négatives et que le nombre de ports de l'opérateur est impair. Donc un opérateur add-min ayant des entrées de N bits qui utilise une pleine résolution de la représentation complément à deux nécessite N+1 bits pour contenir la sortie.

Une augmentation significative des ressources matérielles est anticipée pour seulement un pas de quantification supplémentaire du côté négatif. Cette augmentation du coût matériel est injustifiable pour le peu de précision ajoutée. Également, un avantage incertain peut être tiré d'une représentation comportant une résolution supérieure pour les nombres négatifs en présence d'un canal où le bruit affecte de manière

similaire des valeurs positives et négatives. En limitant la représentation complément à deux à une version symétrique de 2^N-1 pas de quantification, l'implémentation des comparateurs utilisés nécessite N-1 bits comparativement à N bits pour une version pleine de la représentation. Alors, les valeurs d'une représentation réduite varient entre $-2^N + 1$ et $2^N - 1$. Cette réduction de la représentation est réalisée par le CAN (Convertisseur Analogique-numérique) précédant le DSI.

Ainsi pour une même complexité matérielle anticipée, l'opérateur add-min supporte un nombre de pas de quantification de $\frac{2^N-1}{2^{(N-1)}}$ fois plus grand. Pour une même complexité matérielle, il apparaît évident que la réduction de la représentation binaire complément à deux est une décision avantageuse.

Le tableau 3.2 énumère les différents paramètres à spécifier à l'additionneur et l'opérateur add-min. Le lien entre les différents paramètres est établi. Une des limitations des opérateurs génériques développés est que le nombre de bits par entrée est le même pour tous les ports. Le nombre de bits de sortie doit être spécifié pour l'additionneur.

Tableau 3.1: Liste des paramètres nécessaires pour l'additionneur et l'opérateur add-min

Paramètres	Abbréviation	Calcul
Nombre de ports d'entrée	M	—
Nombre de bits par ports	N	—
Additionneur		
Nombre de bits pour la sortie	NADD	log_2(M)+N
Add-min		
Nombre de bits pour la sortie	N	—

3.2.2 Pondération

La pondération est constituée d'un multiplicateur qui, à chaque coup d'horloge, doit multiplier le résultat de l'additionneur au coefficient de pondération définit dans la plage]0,1]. Une question importante consiste à déterminer la configurabilité exigée de cet opérateur au sein du DSI ? En fait, il suffit d'une implémentation qui multiplie la sortie de l'additionneur par un coefficient de pondération ne comportant que quelques bits de résolution. Le graphique de la figure 2.4 montre que le taux d'erreurs varie lentement en fonction d'un coefficient de pondération supérieur à la valeur optimale. Les résultats présentés dans ce graphique fut générés avec un même coefficient de pondération pour toutes les itérations. Il est possible de voir que la valeur optimale est située proche de 0.2. Les graphiques présentés en Annexe A.4 montrent des simulations à différents rapports signal à bruit avec une résolution supérieure concentrée autour de la valeur 0.2. L'implémentation comprend une résolution équivalente de 7 bits pour le coefficient de pondération. Ainsi, il est possible de voir qu'une variation faible du taux d'erreurs existe dans cette région et qu'un gain marginal peut être réalisé en utilisant une résolution plus grande pour bien positionner le coefficient de pondération proche de l'optimal. Également, comme démontré au chapitre 4 (section 4.10.1) un gain supérieur est réalisé en utilisant des coefficients de pondération non uniformes (figure 4.14). Les valeurs des coefficients de pondération de cette optimisation varie entre 0.25 et 0.9375. Donc, il est plus avantageux d'après ces résultats d'implémenter un coefficient de pondération dont la résolution n'est pas forcément grande mais dont la plage couverte est de [0,1]. Ainsi, une résolution de 4 bits pour le coefficient de pondération est suffisante afin de couvrir la plage visée]0,1] et ainsi d'avoir une valeur située proche de l'optimal à savoir 0,1875.

La résolution de la sortie de l'additionneur est variable en fonction de J et de la

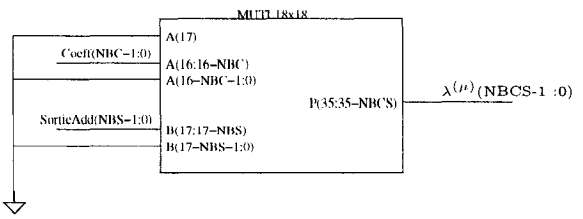


Figure 3.2: Connectivitié du pondérateur au niveau RTL

résolution binaire utilisée à l'intérieur de l'architecture. Une entrée du multiplicateur doit supporter la résolution du port de sortie de l'additionneur et conséquemment le nombre de bits de la sortie du pondérateur doit être paramétrable.

À l'intérieur du FPGA, plusieurs multiplieurs câblés sont disponibles et ces primitives (MULT18x18) peuvent être utilisées pour implémenter le pondérateur. Un multiplieur est nécessaire dans chaque module qui implémente une itération de l'algorithme. Pour implémenter le pondérateur, une encapsulation VHDL fut développée pour transformer la primitive MULT18x18 selon les exigences voulues. Cette primitive est un multiplieur en complément à deux dont les 2 ports possèdent une résoluton de 18 bits (figure 3.2). Le coefficient de pondération (Coeff(NBC-1 :0)) est branché aux bits les plus significatifs du port A (A(16 :16-NBC)) et la sortie de l'additonneur est branchée au port B en commençant par les bits les plus significatifs. La sortie du multiplieur forme le signal $\lambda^{(\mu)}$. L'implémentation VHDL RTL (egister Transfer Level) de cet opérateur est incluse en annexe B.

Tableau 3.2: Liste de paramètres du pondérateur

Paramètres	Abbréviation	Calcul
Nombre de bits pour $\lambda^{(\mu)}$	NADD	$log_2(J+1)$+N
Nombre de bits du Coefficient	NBC	—
Nombre de bits pour la sortie	NBCS	NADD+NBC

3.2.3 Registre à décalage avec des interconnections internes

Le registre à décalage avec des interconnections internes est un registre à décalage standard dont certaines données internes sont acheminées vers un port de sortie. Au niveau VHDL, ce module comporte un port de sortie "taps(2 :0)" (figure 3.3) contenant l'ensemble des données internes demandées. Ce sont ces données internes acheminées au port de sortie que l'on nomme un "tap". La conception de ce registre à décalage est constituée d'interconnexions successives de plusieurs registres à décalage standard. La donnée transitant entre deux registres à décalage est une interconnection interne qui est acheminée au port de sortie "taps(2 :0)". Ce module nécessite une liste de positionnement des différentes données internes voulues. Pour obtenir un registre avec des interconnections internes tel que représenté à la figure 3.3, les indices temporels {0,2,5,6} sont nécessaires. Le dernier indice de la liste correspond à la longueur totale du registre à décalage demandée. La longueur des registres à décalage internes se calcule par la distance entre deux indices de la liste. Ici, 3 registres sont nécessaires et 3 données sont transmises au port de sortie "taps(2 :0)". La longueur du deuxième registre est de 5-2=3. Également la résolution des mots emmagasinés peut être paramétrisée selon le paramètres DW.

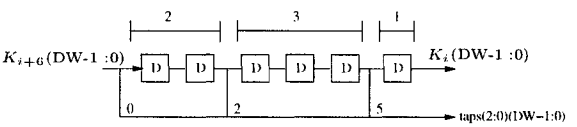


Figure 3.3: Registre à décalage avec des interconnections internes {0,2,5,6}

3.2.4 Registre à décalage

L'implémentation et l'optimisation d'un registre à décalage standard fut réalisée par Martin Dubois [11]. Il a développé une bibliothèque d'IP qui utilisent les différentes

ressources du FPGA Virtex-I pour implémenter un registre à décalage de longueur paramétrables [11]. Tous les IP de registres à décalage développés offre la possibilité permettent de configurer la longueur et la résolution binaire voulues. En résumé, cette bibliothèque fut développée dans l'optique de réduire la consommation en puissance du DSI. Selon les paramètres spécifiés, soit la fréquence cible, la résolution binaire et la longueur du registre voulue, l'IP sélectionne la structure d'implémentation du registre à décalage qui consomme le moins de puissance. Étant donné que la bibliothèque fut développée pour la famille de composants Virtex-I, la bibliothèque utilisée fut conçue pour des blocs de mémoire RAM dont la capacité totale est de 4 Kbits. Avec le Virtex-II, 18 Kbits sont disponibles pour un bloc de mémoire RAM. Cette bibliothèque fut adaptée pour fonctionner avec un FPGA de la famille Virtex-II. Également le modèle de puissance consommée pour l'ensemble des IP de registres à décalage est dérivé à partir d'une table de calcul de Xilinx [23]. Ces structures exploitent les primitives mémoires soit le SRL16, qui est un registre à décalage de 16 bits de long, les bascules contenues dans les CLB et les blocs de mémoire RAM.

3.3 Adaptabilité de l'IP

3.3.1 Génération de plusieurs itérations

L'implémentation du décodeur à seuil itératif comporte la description de 3 types d'itérations soit une description de la première itération, de la dernière itération et d'une itération intermédiaire. Un nombre minimal de 2 itérations peut être implémenté. Elle est alors constituée de modules du premier et du troisième types. Un nombre arbitraire d'itérations peut être implémenté, mais toutefois il faut être conscient que cet algorithme consomme beaucoup de ressources matérielles pour des

longueurs de code importantes, c'est-à-dire pour des longueur de code supérieur à 1000. Pour la première itération, l'équation (2.2) est implémentée. Elle correspond à l'implémentation d'une itération intermédiaire où sont jumelés le registre d'information (Z_1) et le registre ($\lambda^{(\mu-1)}$), (Z_2). L'implémentation de la dernière itération ne possède aucun registre de symbole de parité et de port de sortie et par conséquent un nouveau module fut créé pour l'implémenter.

La connectivité entre les différentes itérations est montrée à la figure 3.4. Il suffit de réaliser la même connectivité entre les différentes itérations décrite en VHDL. Cette connectivité est fixe. Au total 14 registres à décalage sont nécessaires pour un DSI de 8 itérations. Selon la figure 3.4. le symbole de parité et l'entrée du registre ($\lambda^{(\mu-1)}$) (Z_2) sont directement envoyés au noyau de logique combinatoire. L'entrée du registre $\lambda^{(\mu-1)}$ provient directement du noyau de logique combinatoire de l'itération précédente. Il faut découpler deux itérations sucessives avec des bascules si l'on veut que le délai combinatoire du DSI soit indépendant du nombre d'itérations implémentées.

3.3.2 Interconnexion des différents modules d'une itération

La connectivité des différents modules qui constituent une itération se divise en 2 catégories celle qui est dépendante du CSO^{2}C-WS utilisé et celle qui est fixe. Cette connectivité est montrée à la figure 3.4. La connectivité entre le registre ($\lambda^{(\mu-1)}$) et les différents termes $\psi(i,j)$ dépend du code utilisé. Par conséquent, la conception de chacune des parties doit être suffisamment configurable pour supporter l'utilisation de n'importe quel code. De manière simplifiée, les termes add-min de la région (Z_4) nécessitent différentes données provenant de la boucle de rétroaction (Z_3) et du registre à décalage $\lambda^{(\mu-1)}$ (Z_2). Les opérateurs add-min (termes $\psi(i,j)$) utilisent tous une donnée provenant de la boucle de rétroaction et les autres données proviennent du

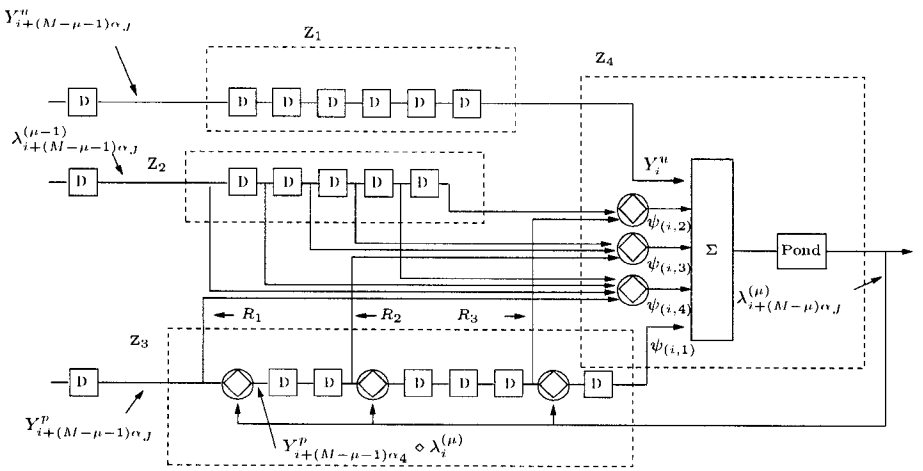


Figure 3.4: Architecture d'une itération (μ) du DSI où J=4 et G={0,1,4,6}

registre à décalage ($\lambda^{(\mu-1)}$). La première interconnexion R_1 de la boucle de rétroaction est dirigée vers l'add-min $\psi(i, J)$, la deuxième interconnexion R_2 est dirigée vers l'add-min $\psi(i, J-1)$ et ainsi de suite jusqu'à R_{J-1} est dirigé vers $\psi(i, 2)$.

Le terme $\psi(i, 1)$ est la sortie de la boucle de rétroaction et celui-ci est connecté au dernier port de l'additionneur. Également la sortie du registre à décalage de l'information est connectée au premier port de l'additionneur et les sorties des opérateurs add-min de Z_4 sont branchées à différentes entrées de l'additionneur. Au total l'additionneur nécessite J+1 entrées. La sortie de l'additionneur est acheminée au pondérateur et la sortie de celui-ci est redirigée vers le registre de rétroaction aux différents opérateurs add-min qui le constitue. Ceci résume les différentes interconnexions des différents modules d'une itération.

3.3.3 Formation de la boucle de rétroaction

La formation du registre de rétroaction demande une liste d'indices temporels des différents termes R_j à la figure 3.5. La boucle de rétroaction est divisée en (J-1) modules (M_i) composés d'un opérateur add-min à 2 ports d'entrée et d'un registre

à décalage. La longueur des registres dépend du code utilisé. La distance en nombre de bascules (Dis_j) entre l'entrée du registre et R_j se calcule en utilisant la formule : $\alpha_J - \alpha_{J-j+1}$. R_1 se positionne à l'indice temporel $\alpha_J - \alpha_J = 0$. D'après le résultat précédent, R_1 est toujours le symbole de parité entrant de l'itération soit $y^p_{i+(M-\mu-1)\alpha_J}$. Deux registres séparent l'entrée et R_2 soit $\alpha_J - \alpha_{J-1} = 6-4 = 2$ et ainsi de suite. Une liste de ces distances (Dis_j) est constituée soit {0,2,5} dans le cas où G={0,1,4,6} et est transmise au registre de rétroaction pour qu'il puisse se configurer. Également, la longueur totale de la boucle de rétroaction est transmise pour calculer la longueur du dernier registre à décalage.

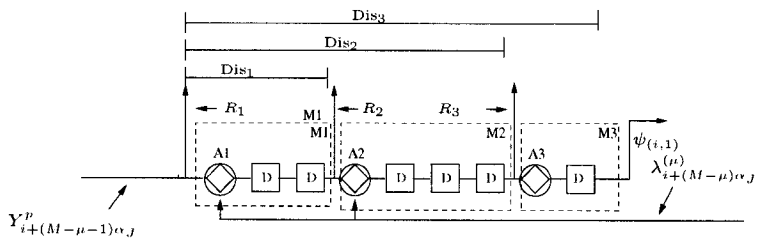


Figure 3.5: Boucle de rétroaction du DSI dont le code est G={0,1,4,6} de l'itération μ

Tableau 3.3: Liste de paramètres pour le IP de la boucle de rétroaction

Paramètres	Abbréviation	Valeur	Calcul
Liste d'indices temp.	—	{0,2,5}	$\text{Dis}_j = \alpha_J - \alpha_{j+1}$
résolution $\lambda^{(\mu)}$	DWFB	8	—
résolution des symboles	DW	8	—

3.3.4 Registre $\lambda^{(\mu-1)}$

Un registre à décalage avec des interconnexions internes est utilisé pour implémenter le registre $\lambda^{(\mu-1)}$. Il suffit d'indiquer quelles sont les données internes nécessaires en ordre croissant pour créer le registre $\lambda^{(\mu-1)}$. Ces différentes données sont utilisées

par les termes $\psi(i,j)$. Par exemple, le terme $\psi(i,3)$ nécessite les termes $\lambda_{i+\alpha_3-\alpha_1}^{(\mu-1)}$ et $\lambda_{i+\alpha_3-\alpha_2}^{(\mu-1)}$ et ces indices temporels découlent du terme $(\lambda_{i+\alpha_j-\alpha_k}^{(\mu-1)})$ et sont calculés à partir de (2.5). Lors de la constitution de la liste des différents indices temporels voulus, ceux-ci doivent être classés en ordre croissant pour que le registre à décalage avec des interconnexions internes puisse être généré adéquatement. La connectivité de ces différentes données internes et des différents ports add-min s'avère variable en fonction du code choisi. Ces indices sont également associés avec un indice de connectivité qui dicte avec quel opérateur add-min celui-ci doit être branché une fois ceux-ci classés. Cette liste classée d'indices et de leur connectivité est constituée par la fonction INPUT_LAMBDA en annexe B. De cette liste, les indices temporels sont transmis au registre à décalage $\lambda^{(\mu-1)}$ et les indices de connectivité sont utilisés pour réaliser les interconnexions des différentes sorties de ce registre aux opérateurs add-min de Z_4. Au total $\frac{J\times(J-1)}{2}$ données proviennent du registre $\lambda^{(\mu-1)}$. La longueur totale de ce registre est de $\alpha_J - \alpha_2$.

3.4 Vérification préliminaire de la fonctionnalité du DSI

Une vérification sommaire de la fonctionnalité de l'IP VHDL fut réalisée. Pour valider le fonctionnement, une version logicielle du décodeur qui constitue un modèle de référence fut modifiée pour donner l'état de certains signaux internes du décodeur à chaque cycle d'horloge. Comme illustré à la figure 3.5, le banc d'essai développé pour la vérification du DSI matériel utilise un logiciel qui caractérise les performances de correction d'erreurs. Il génère les entrées nécessaires au décodeur soit les symboles d'information et de parité et les signaux internes du décodeur.

Ce banc d'essai compare au niveau binaire les signaux internes d'une itération tel que les différentes entrées des opérateurs add-min, de l'additionneur et la sortie des différentes itérations $\lambda^{(\mu-1)}$ de la version logicielle et de la version matérielle. La version logicielle est implémentée en language C et le banc d'essai VHDL est simulé avec Modelsim. Les différents signaux transitent par des fichiers textes. Le temps d'exécution d'un tel processus de vérification est important. Pour une simulation de 5000 cycles d'horloges d'un DSI avec J=10, un temps de simulation de quelques heures est nécessaire. Évidemment devant ce temps considérable, une telle méthodologie de vérification ne peut seulement servir à valider une fonctionnalité partielle du décodeur et on ne peut calculer les performances de correction d'erreurs du DSI. Une méthodologie de vérification beaucoup plus performante sera développée au chapitre 4.

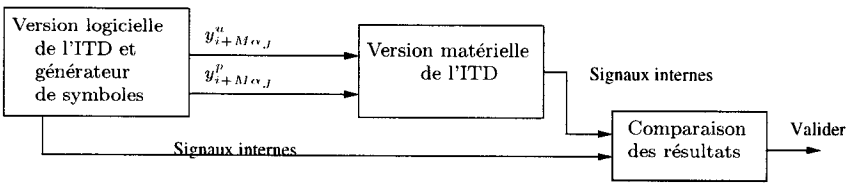


Figure 3.6: Méthodologie de vérification préliminaire du fonctionnement du DSI

3.5 Évaluation de la complexité matérielle

La complexité matérielle est un facteur important qui influence le coût de production d'un circuit intégré et la possibilité d'implémentation d'un algorithme. Premièrement, la résolution binaire nécessaire à l'intérieur de l'architecture est examinée. Ainsi une explication de la résolution binaire théorique est élaborée et une étude d'impact sur le taux d'erreurs de la résolution binaire utilisée pour les différents registres à décalage est présentée. Deuxièmement la complexité matérielle et le délai de pro-

pagation de l'additionneur et de l'opérateur add-min sont explorées en fonction du nombre de ports d'entrée et du nombre de bits par port. La complexité matérielle est décrite par le nombre de LUT utilisé et le délai est exprimé en nanosecondes. Cette implémentation tire profit de la rapidité de la chaîne de retenue disponible dans le FPGA. Ces analyses sont réalisées pour différentes implémentations complément à deux et signe-amplitude des opérateurs. L'influence des différents paramètres du DSI sur la complexité matérielle du réseau combinatoire de l'algorithme est étudiée de manière théorique. Ainsi la représentation binaire la plus avantageuse à utiliser dans l'architecture est expliquée.

3.5.1 Résolutions binaires

Pour mieux comprendre le nombre de bits nécessaires à l'intérieur de l'architecture, le cas où un coefficient de pondération égal à 1 est analysé. Les symboles de parité et d'information provenant du CAN comportent un même nombre de X bits. À la première itération, les symboles d'information quantifiés sont acheminés aux différents opérateurs add-min de Z_4. La sortie du registre de rétroaction, celles des différents opérateurs add-min et du registre d'information sont additionnées. L'additionneur ajoute quelques bits de précision qui représente la partie entière de la représentation binaire d'une itération. Une seule connexion par add-min de Z_4 provient de la boucle de rétroaction. Examinons maintenant la résolution nécessaire pour la boucle de rétroaction. La boucle de rétroaction utilise à son entrée un symbole de parité avec X bits de résolution. L'opérateur add-min A_1 (figure 3.5) compare ce symbole avec la sortie $\lambda^{(\mu)}$. Cette sortie $\lambda^{(\mu)}$, avec un coefficient de pondération égal à 1, possède la même résolution fractionnaire que le symbole de parité. Étant donné que l'opérateur add-min recherche la valeur minimale parmi les entrées et que les symboles de pa-

rité ne possèdent pas de partie entière par définition, le résultat n'aura pas de partie entière. Ce qui implique que l'ensemble des informations acheminées aux opérateurs add-min et à l'additionneur ont une résolution de X bits.

L'usage d'un coefficient de pondération inférieur à 1 augmente la résolution fractionnaire de $\lambda^{(\mu)}$ de quelques bits. Par exemple un coefficient de pondération ≥ 0.25 augmente la résolution fractionnaire de 2 bits. Pour un coefficient entre 0.125 et 0.25, trois bits supplémentaires sont nécessaires. Donc pour des valeurs de coefficients ($\geq$ 0.125) qui minimise le taux d'erreurs, un maximum de 3 bits sont ajoutés à la partie fractionnaire de $\lambda^{(\mu)}$. Cette augmentation de la résolution fractionnaire cause une augmentation de la résolution à l'intérieur de la boucle de rétroaction. Cette résolution plus grande de la boucle de rétroaction requiert davantage de résolution pour les opérateurs add-min de Z_4, de l'additionneur et du pondérateur. Une étude d'impact de la résolution binaire sur le taux d'erreurs peut déterminer la résolution des différentes données emmagasinées à l'intérieur d'une itération. Ici, une étude sommaire est réalisée avec la plateforme de caractérisation des performances de correction d'erreurs décrit au chapitre 4.

Pour réaliser cette étude sommaire de la résolution binaire nécessaire, plusieurs versions du DSI ayant une résolution variant entre 3 et 8 bits fut synthétisées. Cette résolution de 3 à 8 bits signifie que tous les registres à décalage sont implémentés avec cette résolution. Cette étude est réalisée avec l'opérateur saturation ajouté à l'architecture. La première étude montre l'influence de la configuration du CAN utilisée en maintenant une pleine résolution (8 bits) pour les mots de l'architecture. La seconde étude tente de montrer l'influence de la résolution de $\lambda^{(\mu)}$ sur le taux d'erreurs. Une même résolution est utilisée pour $\lambda^{(\mu)}$ allant à l'itération $(\mu + 1)$ et $\lambda^{(\mu)}$ dirigé vers la boucle de rétroaction.

La figure 3.7 présente les performances de correction d'erreurs du DSI qui est

précédé d'un CAN qui implémente une résolution de 3 bits et de 6 bits. Les pas de quantification implémentés pour la version de 3 et 6 bits ont une largueur de 0.36 et 0.045. On voit un gain que l'on peut qualifier de marginal est réalisé en utilisant une configuration d'un CAN avec une précision supérieure à 3 bits. Cette résolution de 3 bits détermine la résolution maximale du CAN pour les symboles de parité et d'information qui sera utilisée dasna les études subséquentes. Ce gain des performances de correction d'erreurs réalisé en utilisant un CAN de 6 bits au lieu de 3 bits est réalisée au détriment d'un coût matérielle 2 fois plus important. Donc une résolution de 3 bits pour le CAN s'avère un choix avantageux à la fois au niveau matériel et en termes de performances de corrections d'erreurs.

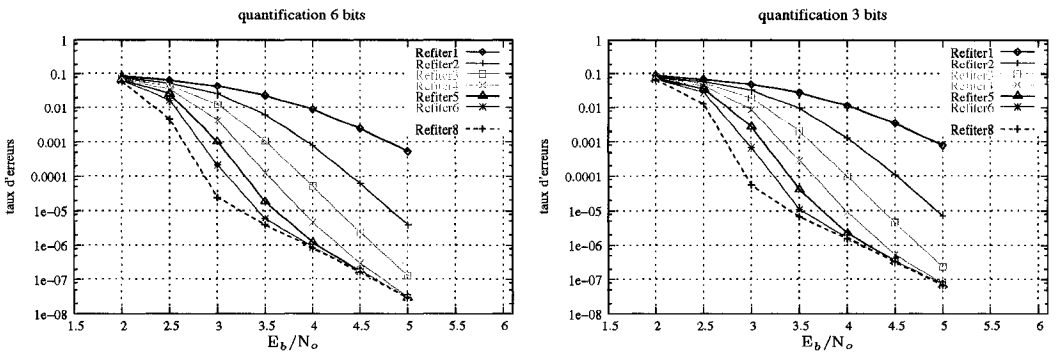


Figure 3.7: Simulations des performances avec un CAN de 3 et 6 bits, un coefficient de pondération de 0.1875 et la résolution internes du DSI est de 8 bits

La prochaine étude tente de démontrer la résolution nécessaire pour le signal λ. La quantification des symboles de parité et d'information est fixée à 3 bits tout au long de cette étude. Le signal λ allant à la prochaine itération et celui dirigé vers la boucle de rétroaction de la même itération est étudié en ayant toujours une même résolution. Dans la figure 3.8, deux simulations avec une résolution binaire du signal λ avec 3 et 6 bits sont présentées. Le coefficient de pondération utilisé est égal à 0.1875. Un coefficient de pondération inférieur à 0.25 équivaut à un décalage d'un

minimum de 2 bits. Donc pour une résolution binaire de 3 bits pour λ, il reste l'équivalent de seulement 1 bits de résolution ce qui explique les mauvaises performances de correction d'erreurs. Il est à remarquer que ce manque de précision affecte surtout les rapports $E_b/N_o \leq 4.5$ dB. En utilisant un coefficient de pondération de 0.9375, on voit à la figure 3.9 que les performances de correction d'erreurs ont augmentées de manière significative, mais que toutefois les performances de corrections d'erreurs restent quasiment inchangées peu importe le nombre de bits de résolution du DSI. Ce dernier cas est normal car pour une valeur du coefficient de pondération proche de 1, le nombre de bits de la partie fractionnaire de λ augmente peu.

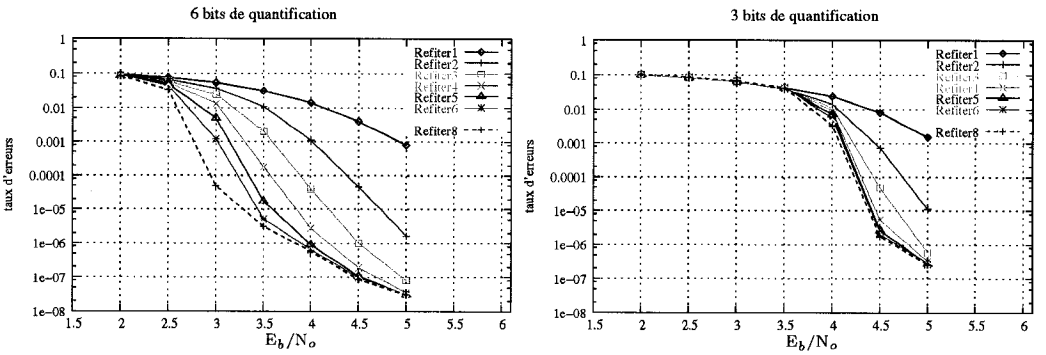


Figure 3.8: Simulations des performances avec λ d'une résolution de 3 et 6 bits et d'un coefficient de pondération de 0.1875

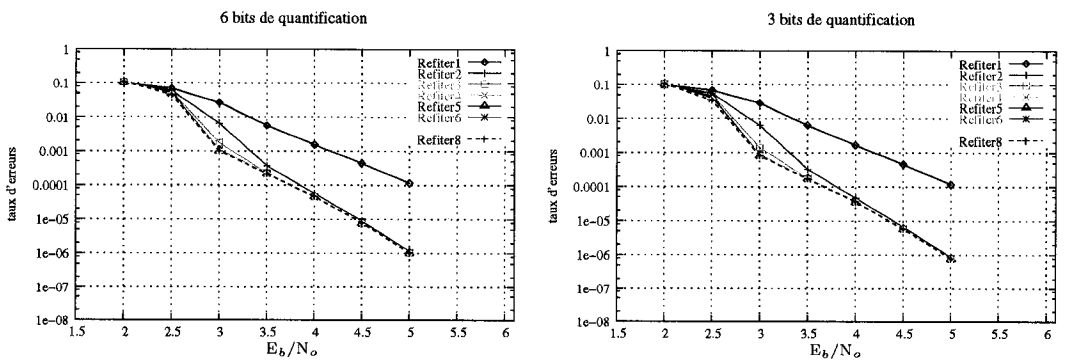


Figure 3.9: Simulations des performances avec λ d'une résolution de 3 et de 6 bits et d'un coefficient de pondération de 0.9375

Finalement le graphique 3.10 utilise le vecteur de coefficients de pondération obtenu par l'optimisation des coefficients de pondération au chapitre 4. Ces coefficients permettent d'obtenir de bonnes performances de correction d'erreurs à des rapports signal à bruit faibles. On voit sur le graphique 3.9 qu'en utilisant une résolution de 4 bits, les performances de corrections d'erreurs sont de beaucoup supérieures aux résultats montrés dans les 2 graphiques précédents pour un seul bit ajouté en utilisant des coefficients de pondération appropriés. Une étude plus détaillée des coefficients de pondérations choisis permettrait d'optimiser davantage la consommation de mémoire de l'algorithme.

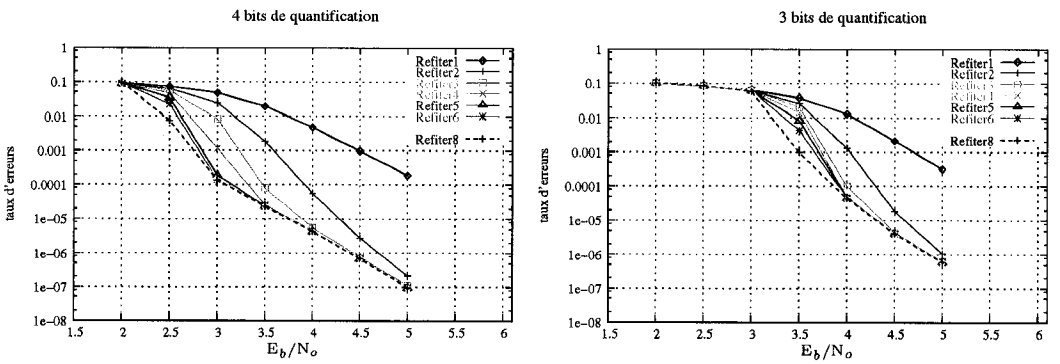


Figure 3.10: Simulations des performances avec λ d'une résolution de 3 et de 4 bits et avec le vecteur de coefficients de pondération obtenus au chapitre 4

3.5.2 Optimisation de la structure de l'opérateur add-min et de l'additionneur

Une version VHDL générique de l'additionneur et de l'add-min fut développée pour implémenter le DSI. Une analyse de délai et de la complexité matérielle est effectuée pour l'implémentation de ces opérateurs. Tous les délais sont extraits suite au processus de placement et routage à l'intérieur du FPGA. Pour chacun des graphiques 3.12, le nombre de ports de l'additionneur étudié varie entre 2 à 16, ce qui permet

d'extraire la complexité matérielle et le délai d'un DSI où $J < 16$. Le nombre de bits pour chacun des ports varie entre 2 et 8 bits. Plusieurs versions VHDL de ces opérateurs ont été développées pour converger vers un opérateur qui exploite efficacement les ressources du FPGA afin de minimiser le délai et la complexité matérielle de ceux-ci. Également des versions complément à deux et signe-amplitude de ces opérateurs ont été développées afin de déterminer quelle représentation est la plus avantageuse.

L'implémentation d'un opérateur générique doit respecter le plus possible les résultats théoriques attendus de la variation du délai du chemin critique et de la complexité matérielle en fonction du nombre de bits par ports et du nombre de ports demandés. Le premier objectif est qu'il faut essayer d'atteindre une implémentation des opérateurs où la synthèse logique de ces opérateurs respecte le plus possible les résultats théoriques attendus pour le délai total combinatoire et la consommation de ressources matérielles. Théoriquement le délai combinatoire devrait varier par paliers en fonction du nombre d'entrées et linéairement en fonction du nombre de bits par entrées. Ces paliers indiquent une augmentation de la profondeur de la structure arborescente. La consommation de ressources matérielles devrait croître de manière linéaire en fonction du nombre de ports d'entrée et du nombre de bits par port de manière à former une surface plane dans un graphique en 3 dimensions. En pratique, il se peut et même il est fort probable que la synthèse logique (qui génère la complexité matérielle) ne génère pas des résultats qui va de pair avec la théorie. Ceci est dû au caractère pseudo aléatoire tous comme le processus de placement et routage et à la performance de l'optimiseur logique. D'ailleurs le processus de placement routage est plus succeptible de générer une évolution des délais non conformes à la théorie comparativement au processus de synthèse logique qui influence la complexité matérielle résultante.

L'implémentation des opérateurs à plusieurs ports suit une structure arborescente composée de plusieurs opérateurs à deux ports. L'implémentation sur FPGA d'un

opérateur à deux ports doit exploiter la chaîne de retenue rapide (primitive MUXCY) qui se propage entre les CLB d'un même "slice" afin de minimiser le délai. Également, il faut que la chaîne de retenue d'un opérateur à 2 ports génère le résultat sans être brisée par l'intermédiaire de LUT. Un exemple d'une implémentation déficiente est montré à la figure 3.11. Ces LUT placés entre la sortie de l'opérateur et la chaîne de retenue sont des MUX générés par une opération conditionnelle tel que "IF". Dans la figure 3.11, ces différents LUT sont générés par une condition sur le signe et l'amplitude du signal.

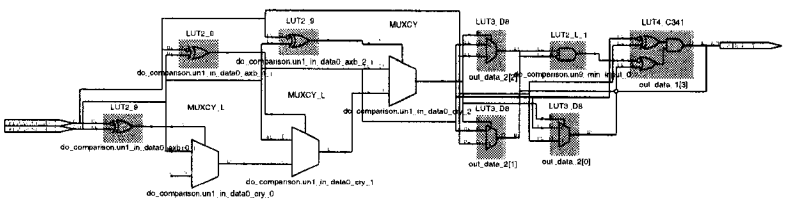


Figure 3.11: Exemple d'une chaîne de retenue non optimale

Ces exigences imposées à l'implémentation RTL ont une implication importante sur la description de l'opérateur en language VHDL. Les opérateurs à deux ports ne doivent pas convertir les ports d'entrées et le résultat en fonction de leur signe. Plus précisément l'opérateur add-min à plusieurs ports ne doit pas être composé d'opérateurs add-min à 2 ports mais plutôt de comparateurs. Tel qu'illustré à la figure 3.12, le traitement du signe est également séparé du processus de comparaison. La raison de séparer les 2 traitements est qu'une opération conditionnelle est nécessaire pour implémenter l'opérateur add-min selon la représentation signe-amplitude. Cette condition traite le cas où l'amplitude minimale trouvée est zéro et que le signe est négatif. La valeur zéro en représentation signe-amplitude est représentée par tous les bits à l'état 0. La valeur "- 0" soit "1000...0" n'est pas supportée. Cette condition additionnelle sur le signe du résultat crée une étape conditionnelle où est générée plusieurs multiplexeurs

(figure 3.11) à la fin de l'opérateur add-min signe-amplitude. Cette condition oblige la séparation du processus de traitement du signe et de comparaison et indique qu'il est préférable d'utiliser des comparateurs plutôt que des opérateurs add-min signe-amplitude à deux ports d'entrée pour la constitution de la structure arborescente. La figure 3.12 montre l'implémentation qui favorise un délai et une complexité matérielle moindre de l'opérateur add-min à plusieurs ports. On voit que cet opérateur inclut une étape de conversion des entrées et de la sortie, qui réalise la conversion de la représentation binaire utilisée à l'intérieur (signe-amplitude pour l'opérateur add-min) vers la representation binaire utilisée à l'extérieur.

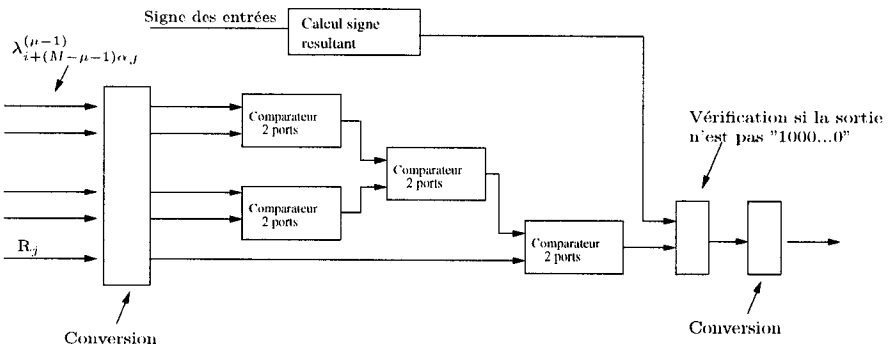


Figure 3.12: Implémentation d'un opérateur add-min à plusieurs ports

Un même raisonnement s'applique à l'implémentation de l'additionneur. Cet additionneur optimal utilise la même structure de l'opérateur add-min à la figure 3.12. L'additionneur utilise une représentation binaire complément à deux pour implémenter le noyau de l'opérateur et des étages de conversion sont utilisés dans le cas où un opérateur signe-amplitude est demandé. Également cette approche permet de diminuer la complexité matérielle de l'implémentation.

Dans le graphique qui suit, une analyse de délai et de complexité matérielle pour l'implémentation signe-ampligne des opérateurs add-min et additionneur génériques sont montrés. On peut remarquer que l'analyse de délai de l'additionneur respecte

moins l'évolution en palier que l'analyse de délai réalisée pour l'opérateur add-min. Par contre, on voit que l'analyse de complexité matérielle forme pratiquement une surface plane dans le cas des deux opérateurs. Les résultats qui se dégagent de cette analyse de la complexité matérielle est que l'additionneur consomme en moyenne 49% plus de LUT que l'add-min pour un même nombre de ports et de bits par port. À partir des 2 figures (3.13), il est également possible d'extrapoler la complexité matérielle d'un décodeur où J $\leq$ 15.

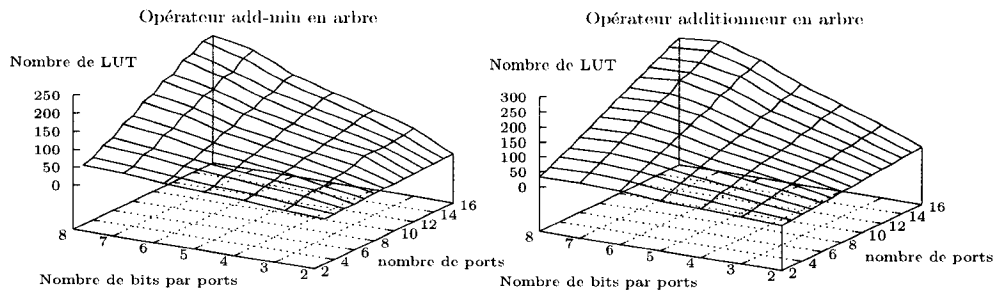


Figure 3.13: Ressources matérielles en nombre de LUT pour l'additionneur et l'add-min en fonction du nombre d'entrées et du nombre de bits par port

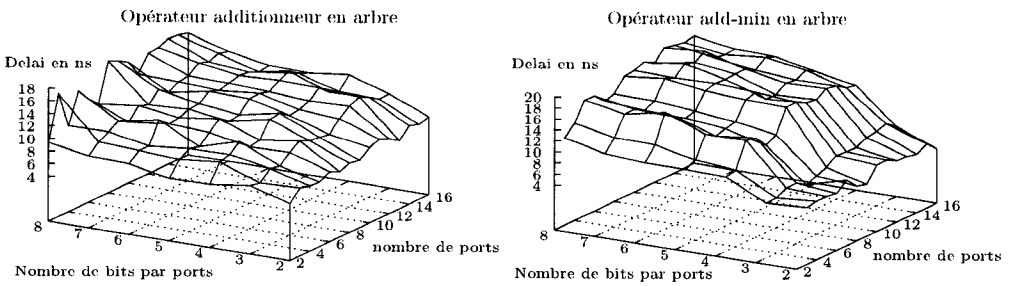


Figure 3.14: Délai en nanosecondes pour l'additioneur et l'add-min en fonction du nombre d'entrées et du nombre de bits par port

3.5.3 Influence des différents paramètres

Le paramètre J a une influence importante sur la complexité matérielle de tous les opérateurs logiques utilisés dans une itération. Au total J-1 add-min sont utilisés

dans la zone Z_4 où chacun d'eux possède un nombre d'entrées variant entre 2 et J. La complexité matérielle d'un opérateur add-min est proportionnelle à son nombre d'entrées. Au total la complexité matérielle des opérateurs add-min de Z_4 est $\frac{(J)\times(J-1)}{2}$ fois celle d'un opérateur add-min à 2 ports d'entrées. Cette formule découle de la somme de J-1 opérateurs add-min ayant un nombre de ports variant entre 2 et J. Également l'additionneur possède J+1 ports d'entrées et par conséquent celui-ci a une complexité matérielle J fois supérieure à un additionneur à 2 ports. Aussi pour la boucle de rétroaction, J-1 opérateurs add-min à deux ports sont nécessaires. La prédominance des opérateurs add-min favorise l'utilisation de la représentation binaire qui demande un minimum de ressources matérielles pour cet opérateur. Donc une représentation binaire signe-amplitude est favorisée pour l'implémentation du décodeur à seuil itératif. Cette complexité est étudiée de manière théorique en supposant que le résolution est la même pour l'ensemble des opérateurs de l'architecture. Tel que discuté à la section 3.5.1, le nombre de bits voulus pour l'architecture reste un compromis entre le coût matériel de l'implémentation et les performances de correction d'erreurs obtenues avec celle-ci.

Tableau 3.4: Complexité théorique des différents opérateurs du DSI où N est égal au nombre d'itérations

Opérateur	Formules
Addmin à 2 ports	$\mathrm{N}\times\left(\frac{(J)\times(J-1)}{2}+(J-1)\right)$
Additionneur à 2 ports	$\mathrm{N}\times\mathrm{J}$
Pondérateur	N
Saturation	N

3.6 Optimisation de l'architecture matérielle

3.6.1 Technique de saturation

Une optimisation importante de la quantité de mémoire consommée est réalisée avec la technique de saturation. Cette économie de mémoire est réalisée en éliminant la partie entière de $\lambda^{(\mu)}$. L'élimination de cette partie de $\lambda^{(\mu)}$ est possible due aux propriétés de l'opérateur add-min. Cet opérateur cherche en valeur absolue la valeur la plus petite parmi M entrées. Les valeurs de $\lambda^{(\mu)}$ sont acheminées à la boucle de rétroaction de la même itération (μ) et aux différents opérateurs add-min du noyau de logique combinatoire de l'itération suivante ($\mu+1$). Les valeurs de $\lambda^{(\mu)}$ sont comparées aux symboles de parité de la boucle de rétroaction qui ne possèdent pas de partie entière. Également les différents add-min de Z_4 de l'itération ($\mu + 1$) utilisent tous une donnée (R_j) provenant du registre de rétroaction, de sorte que les données $\lambda^{(\mu)}$ sont comparées avec des valeurs n'ayant toujours pas de partie entière. Puisque $\lambda^{(\mu)}$ est toujours comparé à des valeurs n'ayant aucune partie entière, la valeur maximale de λ qui sera retenue par l'opérateur add-min est limitée à sa partie fractionnaire. La description VHDL de cet opérateur est ajoutée en annexe B et le schéma bloc de l'implémentation est montré à la figure 3.15.

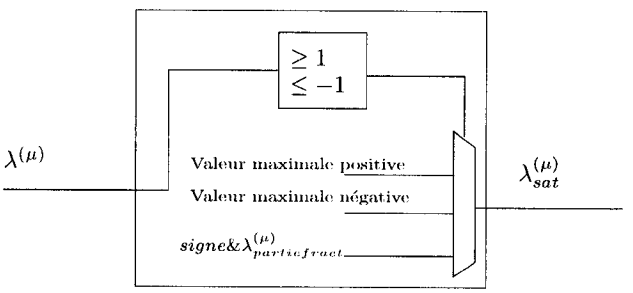


Figure 3.15: Implémentation de l'opérateur de saturation

La figure 3.16 compare la simulation de la version logicielle du décodeur quantifié

avec l'opérateur saturation et une version également quantifiée mais sans saturation. On voit que les performances de correction d'erreurs sont les mêmes pour des résultats ($\geq 10^{-6}$). Sous ce seuil, les variations enre les 2 simulations sont dues à leur caractère pseudo aléatoire. On peut conclure que la saturation n'enlève aucune précision au processus de correction d'erreurs.

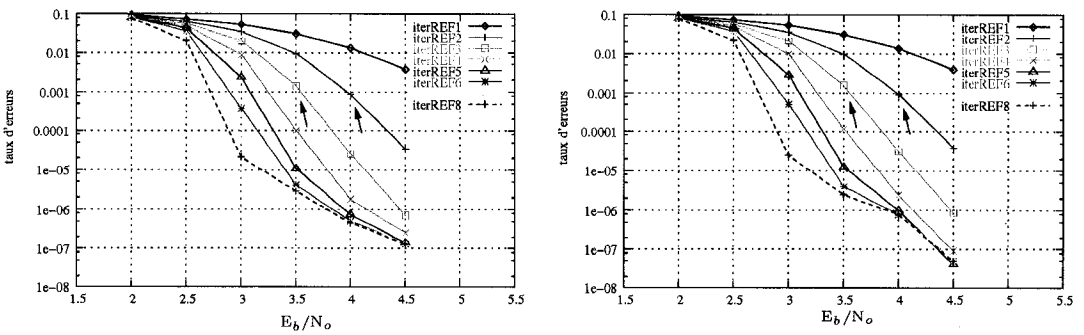


Figure 3.16: Comparaison d'une simulation avec et sans l'opérateur saturation

3.6.2 Délai et complexité matérielle du décodeur à seuil itératif

Le délai du chemin critique du décodeur à seuil itératif est présenté dans cette section. La figure 3.17 reporte les délais obtenus après placement et routage d'un décodeur à seuil itératif de 3 itérations à l'intérieur du FPGA Virtex-II XC2V6000. Cette figure montre le délai en fonction de J pour un décodeur où une représentation binaire signe-amplitude de 8 bits est utilisée. 2 types de décodeur sont implémentés, l'un possède des opérateurs add-min et additionneur avec une structure linéaire et l'autre avec une structure arborescente. On peut observer à la figure 3.17 que le délai reporté pour les décodeurs à structure arborescente dont J=4 et 5 donne un délai plus important comparativement au décodeur employant des opérateurs à structure linéaire. Toutefois, le décodeur avec des opérateurs avec une structure arborescente

montre une évolution beaucoup plus lente du délai combinatoire en fonction de J. Dans le graphique de droite est tracé le nombre de LUT qui forme le chemin critique en fonction du paramètre J. On remarque que dans le cas où J= 4 et 5, le nombre de LUT est moins important pour la structure arborescente que pour la structure linéaire. Donc les délais supérieurs observés dans le graphique de gauche pour J=4 et 5 sont causés par l'inefficacité du processus de placement et routage.

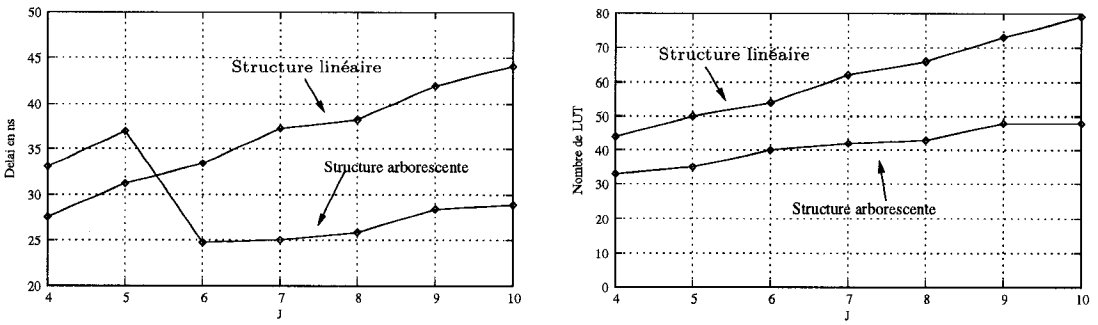


Figure 3.17: Délais des DSI en fonction du paramètre J

Dans le tableau suivant sont détaillées les ressources totales consommées par une implémentation du DSI de 8 itérations où J=10 avec 8 bits pour l'ensemble des mots utilisés. Ce tableau peut servir de référence pour donner une idée des ressources consommées par un tel décodeur. Il faut également rappeler que la quantité de mémoire nécessaire à l'implémentation est de beaucoup inférieure à la quantité de mémoire disponibles à l'intérieur de l'ensemble des blocs de mémoire RAM, car le IP implémentant un registre à décalage avec cette primitive utilise un bloc de mémoire RAM par registre à décalage demandé dont seulement la quantité de mémoire spécifiée pour ce registre est utilisée.

Tableau 3.5: Compilation des ressources matérielles utilisées pour un DSI de 8 itérations, où J=10 et avec une résolution de 8 bits pour l'ensemble des mots implémenté sur un XC2V6000 de Xilinx

Entité matérielle	Quantité	*%d'utilisation*
Nombre de portes équivalentes(10^6)	5.788	96.5
Block RAM	72	50
MULT18x18s	8	5
Slices used.	27171	80
4 inputs LUTs	31512	44
Slices register	7473	4

3.6.3 Stratégie du réajustement de l'ordonnancement

Le délai du chemin critique du DSI est non négligeable (figure 3.17) et empêche le DSI de fonctionner à une fréquence d'horloge élevée. Étant donné que l'ensemble de l'architecture fonctionne avec une même horloge, la fréquence maximale d'opération représente également le débit maximal d'information traité que peut atteindre cette implémentation. Afin de réduire le chemin critique, la méthode du réajustement de l'ordonnancement doit être appliquée à cette architecture. Dans cette section, une stratégie du réajustement de l'ordonnancement est présentée. Cette stratégie s'avère une manière simple de réduire le délai critique de l'algorithme.

La méthode du réajustement de l'ordonnancement cherche à déplacer des bascules disponibles des différents registres de l'architecture vers l'intérieur du noyau de logique combinatoire. En examinant chacune des entrées du noyau de logique combinatoire, il est possible d'aller chercher autant de bascules nécessaires du registre à décalage $\lambda^{(\mu-1)}$ pour les différentes entrées des opérateurs add-min de Z_4. Cette observation est également vraie pour le registre d'information connecté à la première entrée de l'additionneur. Par contre, un nombre limité de bascules sont disponibles pour les connexions R_j provenant du registre de rétroaction. Tel qu'illustré à la figure 3.18, le nombre de bascules disponibles entre deux opérateurs add-min de la boucle de

rétroaction constitue le nombre maximum de bascules qui peut être déplacées vers le coeur de logique combinatoire. À partir de cette observation, le chemin critique où un nombre limité de bascules disponibles pour le réajustement de l'ordonnancement passe par l'opérateur add-min $\psi(i, J-1)$, l'additionneur, le pondérateur, la saturation et l'add-min à 2 ports situé dans la boucle de rétroaction (figure 3.18). Ce chemin ne passe pas par l'opérateur add-min $\psi(i, J)$, car celui-ci peut être réordonnancé comme on le veut en ajoutant des bascules à chacune des entrées (D_{ajout}).

Les différentes distances D_j de la figure 3.18 représentent le nombre de bascules disponibles pour les signaux R_j et $\psi(i, 1)$. Chacun des signaux R_j et $\psi(i, 1)$ ne nécessite pas le même nombre de bascules lors du processus de réordonnancement. Afin de diminuer le délai combinatoire critique, chacun des opérateurs add-min possédant une connexion R_j, soit les add-min $\psi(i, 3)$ à $\psi(i, J-1)$, sont transformés selon la structure arborescente déséquilibrée de la figure 3.19. L'opérateur add-min ainsi conçu traite seulement la donnée provenant de R_j avec un opérateur placé à la fin de la structure. Cette stratégie réduit le délai total du chemin critique à réordonnancer. Aucune stratégie n'est applicable à l'additionneur car seulement deux entrées $\psi(i, J)$ et Y_i^u peuvent fournir une quantité illimitée de bascules.

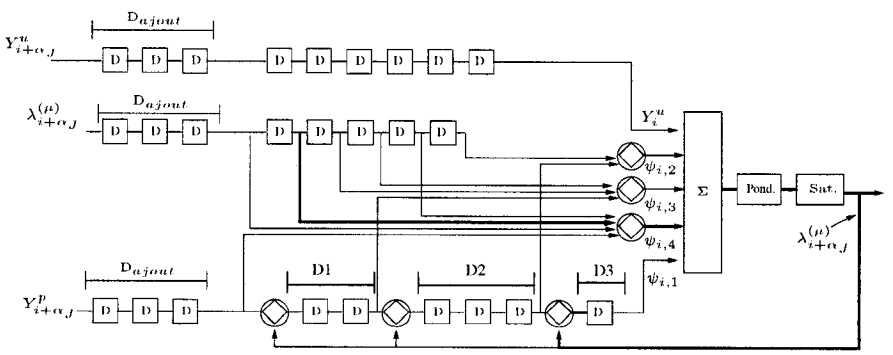


Figure 3.18: Représentation du facteur limitant le réajustement de l'ordonnancement du DSI

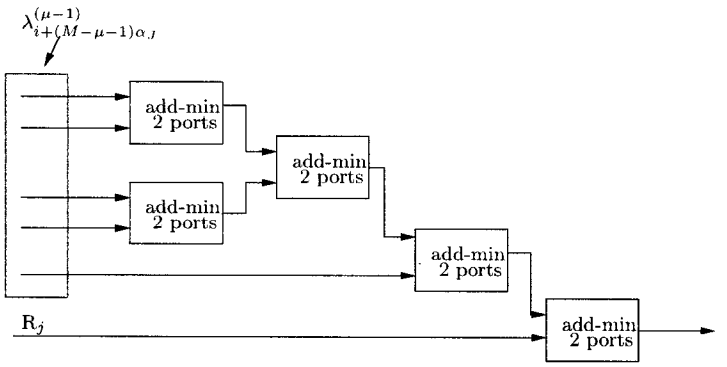


Figure 3.19: Architecture arborescente déséquilibrée d'un opérateur add-min à 6 entrées

3.6.3.1 Recherche de code

Afin de repousser les limites du réajustement de l'ordonnancement, une recherche de nouveaux codes, qui sont également doublement orthogonaux au sens large, où une distance minimale entre les différents indices α_i a été imposée. La méthode de recherche de code utilisée est l'algorithme de recherche présenté à la section 2.1.4. La figure suivante présente la longueur minimale des codes trouvés en fonction de la distance minimale spécifiée entre deux indices soit $(\alpha_i - \alpha_{i-1})$ $\forall i \varepsilon \{2, 3, .., J\}$. Cette figure montre la tendance de l'évolution de la longueur du code pour des codes où J={6,7,8}. Ce qui est remarquable, c'est que pour une distance minimale $D_i \leq 20$ pour J=6, ce qui représente 20 étages pipeline à l'intérieur du noyau combinatoire, le longueur des codes trouvés ne varie pas de manière significative. L'ensemble des codes trouvés sont donnés en annexe B.4.

Dans le graphique 3.21, une étude du facteur d'accélération théorique (D_{min}) de la fréquence d'horloge divisée par l'augmentation de la longueur du code trouvé, soit $\gamma = \frac{D_{min}}{(span - span_{ref})/span_{ref}}$, est présentée. Cette métrique montre dans quelle région est située le gain théorique (γ) de la fréquence d'opération par rapport à l'accroissement de mémoire consommée. On voit que pour la région où D_{min} varie entre 8 et 20, un

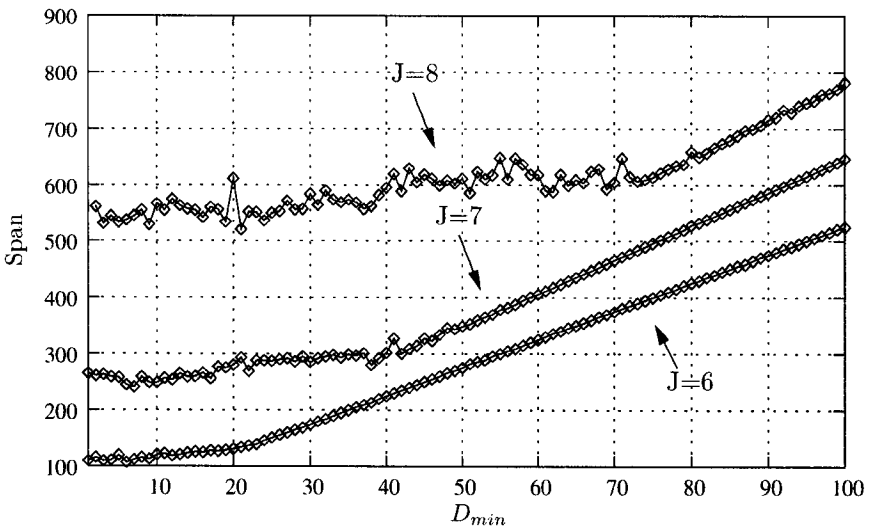


Figure 3.20: Variation de la longueur des codes trouvés en fonction de la distance minimale entre les indices α_i

gain minimal de 47 est trouvé. Également, on voit qu'une asymptote converge vers un gain de 22, mais toutefois le nombre d'étages pipeline est trop important pour être considéré.

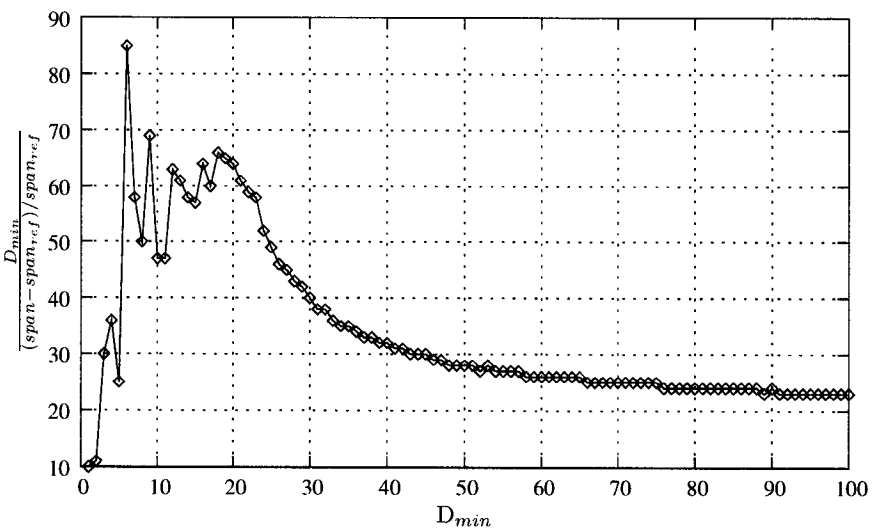


Figure 3.21: Gain γ en fonction de D_{min} des codes trouvés où J=6 par la méthode de recherche pseudo aléatoire

Chapitre 4

Accélérateur de caractérisation des performances de correction d'erreurs

Depuis les années 1990, de nouveaux algorithmes de correction d'erreurs ont été découverts. La caractérisation de ces algorithmes de correction d'erreurs est en majorité réalisée de manière logicielle. Ce chapitre présente une méthodologie d'accélération de la caractérisation d'un algorithme de correction d'erreurs. Cette nouvelle méthodologie tire profit du potentiel d'accélération qu'offre une implémentation matérielle. Grâce à la facilité de conception des FPGA et à leur grande capacité d'implémentation, ce concept d'accélération de calcul s'avère un moyen facile et peu coûteux de résoudre différentes difficultés d'optimisation.

Comme montré précédemment sur la figure 2.4, le taux d'erreurs peut atteindre de faibles valeurs. Ces valeurs demandent un grand nombre de bits transmis afin d'obtenir une mesure précise du taux d'erreurs. Une caractérisation des performances de corrections d'erreurs d'un DSI où J=10 avec 8 itérations pour un SNR donné

nécessite 7.44 heures lorsqu'exécuté sur un SUN SPARC V440. Cette caractérisation ne compte que 100 millions de bits transmis. La vérification de la fonctionnalité du DSI conçu en VHDL avec un logiciel de vérification matérielle tel que Modelsim demande un temps beaucoup plus important.

L'optimisation d'une architecture décrite avec plusieurs variables peut demander une puissance de calcul considérable. Lorsque le nombre de variables augmente, le temps de calcul peut devenir très important. Plusieurs solutions existent pour réduire le temps de calcul nécessaire. L'une des solutions qui est très populaire et qui devient en quelques sortes la solution de facto est la grappe de calcul Linux. Cette grappe (Cluster) est un ensemble d'ordinateurs conventionnels branchés par un réseau performant (gigabits) où chaque ordinateur forme un noeud de calcul. Depuis l'année 2000, plusieurs ordinateurs de ce type ont enrichis le palmarès de 500 ordinateurs les plus puissants de la planète [25]. D'ailleurs plusieurs grandes compagnies offrent des produits de ce type. La figure 4.1 montre l'évolution du nombre de grappes de calcul dans le palmarès depuis 1993. La grappe de calcul est une solution économiquement attrayante comparativement aux ordinateurs conventionnels pour résoudre différents problèmes scientifiques. Néanmoins pour utiliser cette solution, il faut que le problème soit facilement divisible. Est-ce que l'utilisation d'une grappe de calcul Linux constitue toujours la solution la moins dispendieuse pour obtenir la puissance de calcul nécessaire ?

L'avancement de la microélectronique durant la dernière décennie, la réduction de la taille des transistors et le déploiement de circuits reconfigurables ont apporté de nouvelles possibilités. Ces évolutions ont réduit la barrière entre ce qui est réalisé en matériel et en logiciel. L'un des exemples les plus frappant est le modem utilisé par les ordinateurs personnels pour se connecter à un fournisseur Internet. Quelques années auparavant, le processus de décodage était complètement réalisé en matériel.

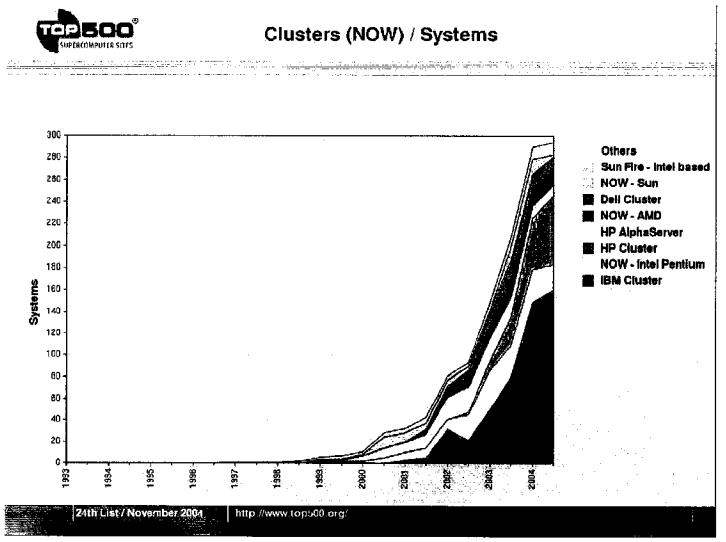


Figure 4.1: Statistique sur le nombre de grappes de calcul des 500 ordinateurs les plus puissant au monde

Avec l'amélioration de la capacité de calcul des processeurs X86 grâce à la réduction constante d'échelle du transistor, le processus de décodage a été transféré au processeur. Ce transfert est dû à une augmentation plus importante de la puissance de calcul des processeurs comparativement aux besoins de l'usager moyen.

Dans le domaine des télécommunications, le contraire se produit présentement. Quelques recherches récentes ont été réalisées afin de créer des modules de propriété intellectuelle (IP, Intellectual Propriety) qui diminuent le temps de simulation nécessaire à la résolution de plusieurs problèmes. Dans [4], les auteurs ont développé un IP générant un bruit blanc gaussien additif afin de simuler un canal de communication. La caractérisation d'un puissant algorithme de correction d'erreurs demande quelques millions et quelques fois quelques milliards de bit transmis. Une solution matérielle-logicielle fut développée dans ce mémoire afin de diminuer le temps de calcul nécessaire à la caractérisation du DSI.

En plus de servir à la validation de l'implémentation matérielle du DSI, une pla-

teforme d'accélération sert non seulement à la validation, mais aussi à trouver un ensemble de coefficients de pondération minimisant le taux d'erreurs à la dernière itération. Aussi, elle fut utile pour analyser le comportement du DSI à de hauts SNR. Nous présentons dans ce qui suit la méthode de validation faite en utilisant la plateforme. Nous décrirons de manière détaillée le flot de communication de l'ECM (Environnement de Caractérisation Matériel) et les implémentations logicielles et matérielle de la solution développée. La section 4.10 présente de nouveaux résultats quant à l'optimisation des coefficients de pondération. La dernière section présente la performance de l'algorithme de correction d'erreurs à des SNR encore inexplorés.

L'annexe A contient l'ensemble des programmes du serveur TCL (Tool Command Language), du programme ARM et du code VHDL de l'implémentation du système de communication. Également une table d'adressage des différents registres APB (Advanced Peripheral Bus) utilisés pour la configuration de l'ECM est donnée. Finalement une liste des frontières utilisées pour les simulations présentées sont données afin que les résultats puissent être reproduits.

4.1 Structure logicielle-matérielle de l'ECM

La figure 4.2 montre un diagramme bloc du flot des communications du processus d'optimisation logicielle-matérielle proposé. Au niveau supérieur se trouve un logiciel d'optimisation AWLDT (Automatic Word Length Determination Tool) développé dans la thèse de doctorat de Marc-André Cantin qui optimise les différents paramètres du décodeur implémenté [5] [6]. Au plus bas niveau se trouve la plateforme ARM integrator qui contient un système de communication complet où le DSI est implémenté dans la partie réception. La communication entre les deux paliers est gérée par l'intermédiaire d'un serveur TCL situé sur l'ordinateur hôte de la plateforme.

Le logiciel AWLDT fonctionne sur un hôte distant. Les valeurs des différents paramètres sont envoyées au serveur TCL en utilisant un socket TCP/IP (Transmission Control Protocol/Internet Protocol). Le serveur TCL écrit les différents paramètres reçus dans un fichier de configuration prédéterminé et signale au processeur ARM de lancer la simulation. Par la suite, le processeur ARM lit ces différents paramètres et configure le fichier de registres matériels de la simulation et démarre celle-ci. Le processeur ARM interroge à intervalle régulier le FPGA afin de détecter la fin de la caractérisation des performances de correction d'erreurs. Une fois la simulation matérielle terminée, les résultats contenus dans le fichier de registres APB sont transférés par le processeur ARM dans un fichier de résultats sur l'hôte de la plateforme et sont communiqués au AWLDT par le serveur TCL. Toutes ces opérations constituent une boucle du processus d'optimisation.

Ce flot de communication est utilisé lors du processus d'optimisation des paramètres. L'ECM fut aussi utilisé pour valider la fonctionnalité du DSI et vérifier de manière expérimentale les performances de correction d'erreurs sur une plage SNR étendue. Ces deux dernières utilités de la plateforme ne demandent pas de serveur TCL intermédiaire et de logiciel d'optimisation et par conséquent, les différents paramètres de la simulation sont gérés directement par le processeur ARM.

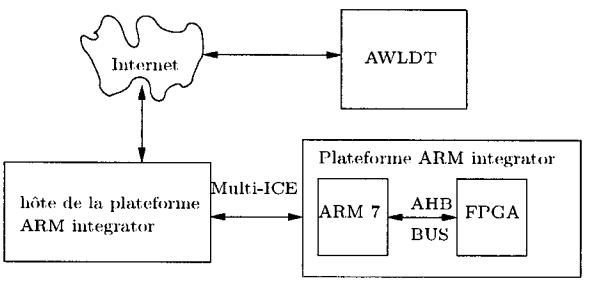


Figure 4.2: Flot des communications

Les communications entre les 3 paliers logiciels sont présentées de manière dé-

taillée. Chacune des implémentations logicielles exécutent sucessivement la même série d'actions. Le processus démarre avec le logiciel AWLDT (S1) avec la transmission des différentes commandes par un socket au serveur TCL. Celui-ci attend la commande "START" du logiciel avant de transmettre l'information reçue pour la caractérisation des performances de correction d'erreurs au processeur ARM. Une fois cette commande reçue, le processeur ARM écrit les différents paramètres dans le fichier "config.txt". La synchronisation entre le serveur TCL et processeur ARM passe par l'intermédiaire de 2 fichiers (interface.lck et platform.lck). Une fois tous les paramètres écrits, le processeur ARM écrit dans le fichier "interface.lck" la chaîne de caractères "START". Ainsi, le processeur ARM qui lit à intervalle régulier le contenu du fichier, identifie la commande envoyée et écrit dans le fichier platform.lck "WAIT" pour indiquer que la simulation est en cours d'exécution. Le processeur ARM démarre la configuration du fichier de registres APB, la configuration du pseudo CAN et par la suite la simulation. La serveur TCL attend que le processeur ARM écrit un mot autre que "WAIT" pour indiquer que la simulation est terminée. Une fois la simulation terminée, le serveur TCL envoie l'information au AWLDT pour que celui-ci puisse transmettre une nouvelle directive jusqu'à ce que les critères d'arrêt du AWLDT soient atteints.

4.2 Implémentation logicielle

4.2.1 Logiciel d'optimisation

Le logiciel (AWLDT) contrôle le processus d'optimisation et par conséquent les différents paramètres du DSI. L'objectif de l'optimisation est de trouver la meilleure combinaison de coefficients de pondération qui maximise la différence entre la com-

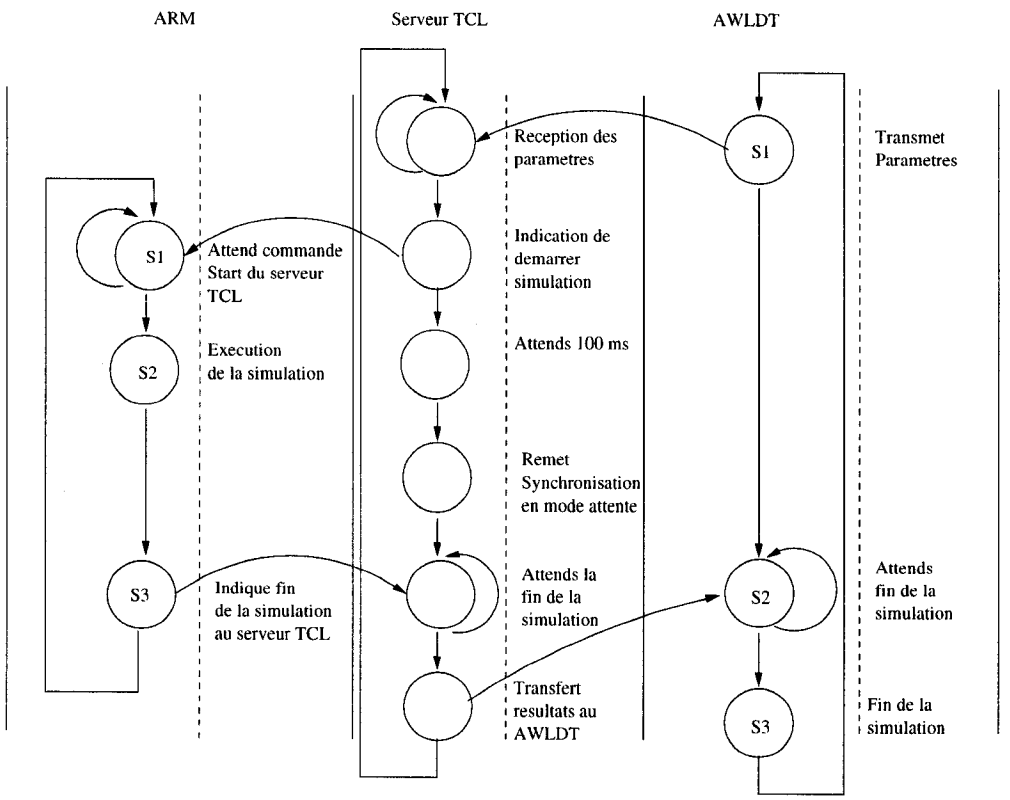


Figure 4.3: Diagramme du communication entre les différents paliers de la plateforme matérielle-logicielle

binaison trouvée et le résultat de la meilleure performance de correction d'erreurs obtenue des recherches précédentes. Aucune quantification n'est considérée au sein de l'architecture et donc tous les mots possèdent une résolution maximale de 8 bits. Le pseudo CAN est configuré pour un mot de 3 bits pour les symboles d'information et de parité acheminés au DSI. Ce problème complexe d'optimisation comporte 8 variables dont chacune d'elles possède 15 possibilités. Au total, il existe $15^8 (\approx 2.56 \times 10^9)$ combinaisons de coefficients de pondération possibles pour un DSI ayant 8 itérations. Il faut 7.44 heures à un SUN SPARC V440 pour déterminer le taux d'erreurs d'une simulation de 100 millions de bits simulés à un certain SNR. Il faut également considérer qu'une caractérisation nécessite 7 simulations à des SNR variant à intervalles

réguliers entre 2.0 dB et 5.0 dB. Alors une optimisation globale nécessiterait 2.56×10^9 années avec la solution logicielle et 655066 années avec la plateforme de caractérisation développée.

La recherche d'une solution optimale n'est évidemment pas praticable. Pour converger vers un minimum local dans un temps acceptable, un puissant outil d'optimisation logiciel est nécessaire. Le logiciel AWLDT procède à la recherche d'un minimum local par une descente de gradient. L'AWLDT répète les 4 mêmes étapes afin de trouver un minimum local :

1. Exécution du décodeur (S_1)

   La première étape consiste à transmettre les valeurs des différents paramètres au serveur TCL situé sur l'ordinateur hôte de la plateforme ARM integrator. L'ARM lance ensuite l'exécution d'une caractérisation.

2. Obtenir le TEB_{obt} (S_2)

   Une fois l'exécution de la caractérisation terminée, la seconde étape consiste à recueillir les TEB_{obt} (Taux d'Erreurs par Bit) obtenus de chaque itération du DSI et des 7 niveaux SNR considérés.

3. Calcul de la métrique DEV_{obt} (S_3)

   En se basant sur les TEB_{obt} obtenus à l'étape 2, la troisième étape consiste à calculer la métrique de déviation (DEV_{obt}) par l'équation 4.1 :

$$DEV_{obt} = \max_{\forall E_b/N_o} \left[\frac{100 \times (TEB_{ref} - TEB_{obt})}{TEB_{ref}} \right] \tag{4.1}$$

où TEB_{ref} est le résultat obtenu de la meilleure combinaison trouvée dans les recherches antérieures, avec un coefficient de pondération de 0.1875 pour l'ensemble des itérations. Les étapes 1 à 3 sont répétées pour chacun des paramètres à optimiser en variant d'une unité chacun des paramètres à tour de rôle.

4) Procédure de maximisation (S_4)

   En se basant sur les DEV_{obt} obtenus à l'étape 3, la procédure choisit la combinaison des coefficients de pondération qui maximise la DEV_{obj} calculée en 3. Cette combinaison correspond à la direction où la valeur du gradient est la plus importante. Ainsi la métrique de l'équation (4.1) calcule la déviation du taux d'erreurs entre la meilleure solution trouvée dans les recherches publiées précédemment et essaie de maximiser cette différence afin de s'éloigner le plus possible de ce point.

Finalement ces 4 étapes sont répétées jusqu'à ce qu'un minimum local soit trouvé tel qu'illustré à la figure 4.4.

4.3 Serveur TCL et synchronisation des différents niveaux

Le serveur TCL sert d'intermédiaire entre le logiciel d'optimisation et la plateforme ARM integrator. Afin de communiquer adéquatement avec le AWLDT, il utilise un protocole particulier qui lui permet de spécifier au serveur TCL des commandes en particulier. Ces commandes sont dictées au serveur TCL soit de démarrer une simulation, de terminer le processus d'optimisation ou de spécifier la valeur d'un certain paramètre. Le code source de ce serveur est inclus en annexe A. Une fois que l'AWLDT a spécifié tous les paramètres et la communication est terminée, le serveur TCL écrit

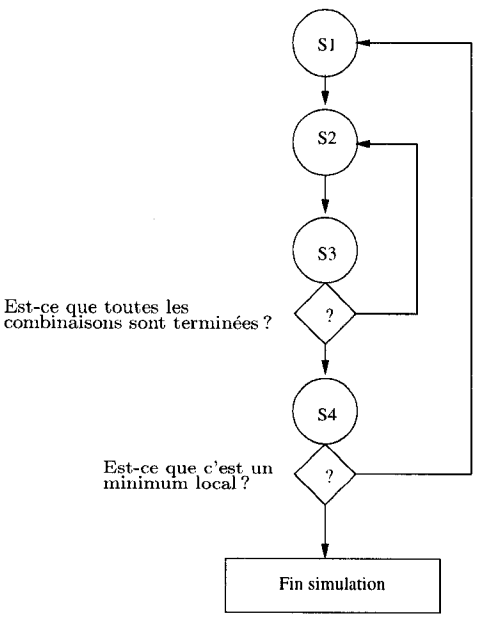


Figure 4.4: Diagramme bloc du processus d'optimisation

dans un certain ordre les paramètres de la simulation dans le fichier config.txt.

4.4 Implémentation matérielle

4.4.1 Architecture AHB, APB et fichier de registres

L'implémentation fournie avec la plateforme est montrée à la figure 4.5. L'AHB est connecté à l'APB par l'intermédiaire d'un pont. Le fichier du registre APB contient initialement les différents registres de configuration des diviseurs de fréquences du module logique. Le code source de chacun des modules est donné avec la plateforme. Le meilleur moyen d'interconnecter l'ECM à l'environnement déjà existant est de le brancher directement au fichier du registre APB. De cette manière, on évite d'implémenter inutilement le protocole APB à l'intérieur de celui-ci. Il suffit d'ajouter les ports d'entrées et de sorties au fichier de registres APB et les registres nécessaires

à la configuration de l'ECM. Au total 831 registres sont nécessaires pour implémenter l'ensemble des paramètres reconfigurable de l'ECM. L'énumération des différents registres est faite dans le fichier "logic.h" qui se trouve en annexe A.

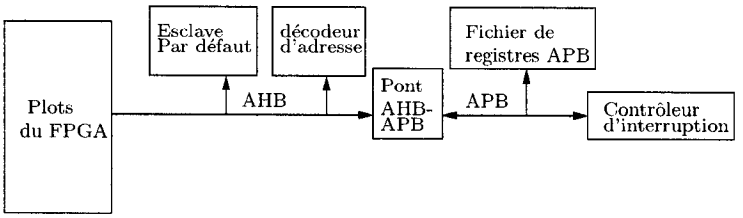


Figure 4.5: Connectivité AHB et APB

4.4.2 Implémentation matérielle d'un système de communication

Le système de communication implémenté en matériel à l'intérieur du FPGA contient tous les blocs nécessaires afin de pouvoir caractériser les performances de correction d'erreurs du DSI. L'environnement de caractérisation matérielle a été implémenté à l'intérieur du FPGA sur le module logique de la plateforme ARM integrator. Le système de communication est composé essentiellement d'un générateur pseudo aléatoire comme source binaire, d'un codeur convolutionnel CSO^{2}C-WS, de 2 traducteurs bit-canal, de 2 canaux de communication matériels, d'un quantificateur et finalement d'un DSI. Comme illustré à la figure 4.6, le système de communication contient un générateur pseudo aléatoire binaire (signal b de la figure 4.6) qui a été implémenté en utilisant un LFSR (Linear Feedback Shift Register). La source binaire possède une entropie maximale donc les valeurs 0 et 1 sont équiprobables. Le flot de données est enchainé avec un codeur de canal CSO^{2}C-WS. Pour chaque bit d'information généré par la source, un bit de parité est également généré. Le taux de codage (R) est donc égal à $\frac{1}{2}$. Le traducteur bit-canal adapte la valeur binaire des bits

d'information et de parité selon la signalisation antipodale utilisée dans le canal de communication. Le canal de communication émule un canal analogique avec 16 bits de précision. L'effet perturbateur introduit dans le canal de communication est généré par un module de bruit blanc gaussien qui est additionné à la valeur du canal de communication (provenant des traducteurs bit-canal). Les terminaisons des deux canaux de communication sont connectées à un quantificateur qui agit comme un CAN paramétrisable. Une fois quantifiée, les bits d'information et de parité sont acheminés au DSI. Le module générateur de séquences retardées compte un LFSR pour chacune des itérations du DSI. Le but de ce module est de reproduire une séquence identique à celle générée par la source binaire mais en phase avec la séquence corrigée à la sortie de chaque itération du DSI. Pour ce faire, chaque LFSR est programmé avec un mot d'initialisation retardé de manière à compenser la latence du DSI et des différents modules précédents celui-ci. Alors, l'information corrigée de chaque itération est comparée avec les séquences de référence et le nombre total d'erreurs par itération est comptabilisé.

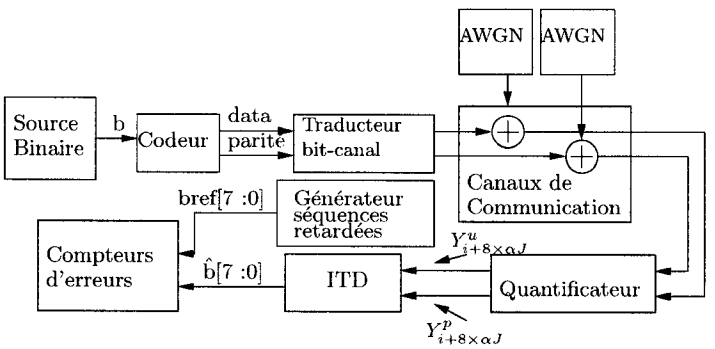


Figure 4.6: Diagramme bloc du système de communication implémenté

4.4.3 Quantificateur

La fonction de ce quantificateur est d'émuler le comportement d'un CAN avec un pseudo canal analogique. Ce CAN est en fait un LUT où le contenu peut être programmé comme souhaité. L'implémentation de ce module est réalisée en utilisant 512 Kbits ($2^{16}\times$ 8 bits) de mémoire RAM distribuée parmi 32 primitives bloc de mémoire RAM. Le nombre de pas de quantification, la largeur des pas et l'assignation binaire pour chacun d'eux doivent être paramétrisés avant chaque simulation. Comme illustré à la figure 4.7, les 5 bits les plus significatifs provenant du canal de communication sélectionne l'un des blocs de mémoire RAM. Chacune des adresses est associée à une donnée de 8 bits. En choisissant adéquatement le contenu des blocs de mémoire RAM, il est possible de réaliser n'importe quelle configuration du CAN. Finalement, le multiplexeur M2 relie la sortie du bloc de mémoire RAM sélectionné avec la sortie du quantificateur.

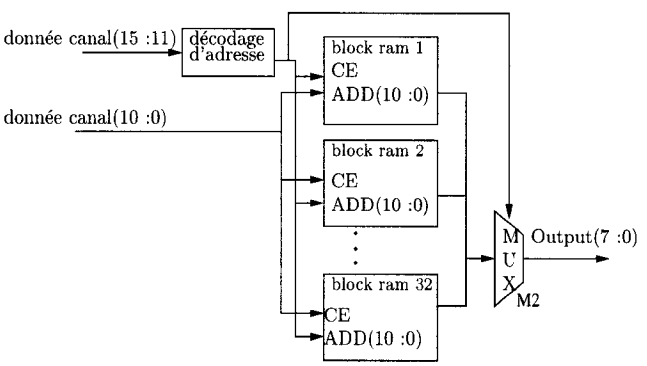


Figure 4.7: Implémentation du quantificateur

La figure 4.8 illustre l'implémentation d'un bloc de mémoire RAM utilisé à l'intérieur du pseudo CAN. Durant la simulation, chacun des 2 ports disponibles sur ces mémoires sont branchés aux canaux de communication d'information et de parité. Avant de lancer une simulation, le contenu de ces blocs de mémoire RAM doit

être programmé. Seulement le port A est utilisé lors de la programmation du pseudo CAN. Trois registres de configuration sont nécessaires à la programmation des blocs de mémoire RAM. Le premier registre contient l'adresse que l'on veut programmer et le second contient les 8 bits de contenu à mettre à l'adresse visée. Un bit du registre de contrôle est nécessaire pour activer l'écriture des blocs de mémoire RAM et sélectionner l'adresse de programmation (adresse ARM) du multiplexeur M1. En somme 46 secondes sont nécessaires pour programmer les 65535 octets des blocs de mémoire RAM.

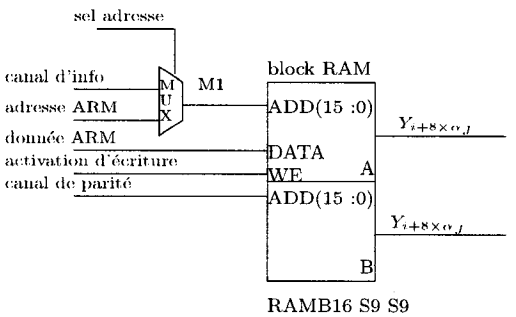


Figure 4.8: Connectivité des blocs de mémoire RAM du pseudo CAN

4.4.4 Implémentation du DSI utilisée

Une version paramétrable après synthèse du DSI fut implémentée. Les coefficients de pondération et la résolution des mots peuvent être changés dynamiquement. La figure 4.9 montre l'architecture du décodeur. À titre de rappel, le coefficient de pondération est défini dans un intervalle de]0,1]. Le nombre de bits qui est utilisé pour le coefficient de pondération est de 4 ce qui donne une résolution de $\frac{1}{16}$. Ici, le nombre d'itérations implémenté est de 8. La résolution de tous les registres à décalage est de 8 bits.

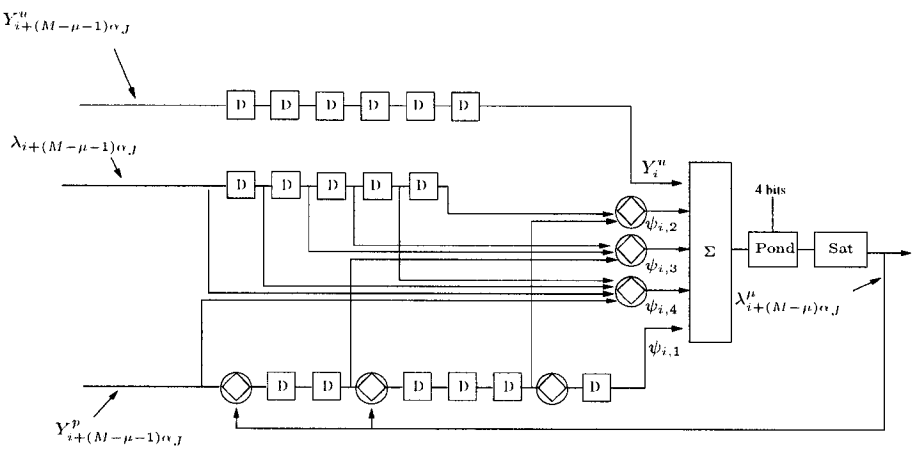


Figure 4.9: Implémentation d'une itération du DSI utilisée dans l'ECM où G={0,1,4,6}

4.5 Programme ARM et circuit de contrôle de l'ECM

Le programme exécuté sur le processeur ARM et la circuitrie de contrôle du fonctionnement de l'ECM sont intimement liés. Le rôle du processeur ARM est de configurer adéquatement la simulation par l'intermédiaire du fichier de registres APB. Il procède premièrement à la programmation du contenu du pseudo CAN. Par la suite, il programme le nombre de bits à simuler et les coefficients de pondération du DSI. La résolution des LFSR utilisée permet un maximun de $2^{32}(\approx 1 \times 10^9)$ bits transmis. Cette résolution de 32 bits fut utilisée pour l'optimisation des coefficients de pondération. Une version avec 40 bits soit $2^{40}(\approx 1.1 \times 10^{12})$ bits transmis fut développée pour étudier la performance de correction d'erreurs sur une plage SNR étendue. Également, toutes les caractérisations ont été réalisées avec un nombre de bits simulé suffisamment grand pour compter un minimum de 100 événements erreurs. Un des paramètres les plus importants est le SNR pour le signal de bruit blanc. L'IP utilisé accepte une valeur de SNR située entre 0 et 15.9 dB. Pour éliminer la limitation imposée par le module AWGN (Additive White Gaussian Noise), il est possible d'adapter l'ampli-

tude de la signalisation antipodale choisie en changeant les valeurs qui traduisent le 0 et 1 binaire par des valeurs appropriées. La plage par défaut de la caractérisation s'effectue entre 2.0 dB et 5.0 dB avec un pas de 0.5 dB. La plage SNR étendue varie entre 2.0 dB et 8.0 dB.

L'implémentation matérielle est complètement autonome une fois que le processeur ARM lance la simulation. Pour lancer la simulation, l'ARM doit activer le bit LOAD et START contenu dans le registre de contrôle APB à la figure 4.10. Lorsqu'il est activé, le bit LOAD est utilisé pour charger les mots d'initialisation appropriés dans les différents LFSR de la source binaire et du générateur de séquences retardées. Le bit START doit être activé en même temps que le bit LOAD, car ce signal est connecté aux différentes entrées d'activation (chip enable) des modules utilisés dans l'implémentation matérielle. Pour compléter le démarrage de la simulation, l'ARM a besoin d'exécuter un dernier accès d'écriture au registre de contrôle sans le bit LOAD activé en maintenant le bit START activé.

Au démarrage de la simulation, un premier bit est émis par la source binaire. Avant de rejoindre la fin de la dernière itération, ce bit doit traverser tout le système de communication. La latence totale dépend de la latence précédent le DSI qui est une valeur fixe de 7 cycles et la latence du DSI. Dans le cas où J est égal à 8 et 9, la longueur des codes utilisés sont de 459 pour J=8 et 912 pour J=9. La latence du DSI pour une itération est égale à la latence du code plus 1 soit 459+1=460 et 912+1=913. Par exemple pour la dernière itération où J=9, la latence totale jusqu'à la fin de la dernière itération du DSI est de $(8 \times 913 + 7)$ cycles d'horloges.

Une fonction fut développée pour calculer les mots d'initialisations nécessaires pour les différents LFSR utilisés. Cette fonction imite le fonctionnement du LFSR pour un nombre de cycle précis. Ainsi une implémentation utilisant un LFSR de 40 bits dont l'équation génératrice $(X^{40} + X^{21} + X^{19} + X^2 + X^0)$ est employée.

La fonction calcule le mot d'initialisation, pour le LFSR qui génère la séquence de référence pour la première itération du DSI, en exécutant le fonctionnement du LFSR pour $(\alpha_J + 1) + 7$ cycles d'horloge. Un mot de départ, qui initialise l'état logiciel du LFSR ainsi émulé, est choisi par l'usager. Les mots d'initialisation destinés aux itéations suivantes et à la source binaire sont calculés en avançnt de (α_J+1) cycles d'horloge le mots d'initialisation trouvé pour la première itération. Le code C se trouve en annexe A dans le fichier "logic.c".

Un compteur d'erreurs par itération est nécessaire. À cause de la latence totale, ces compteurs doivent démarrer seulement après que le premier bit émis atteint la fin de l'itération correspondante. Comme la latence totale d'une itération dépend du code utilisé, un registre par itération est nécessaire pour contenir la valeur de la latence totale entre la source binaire et la fin de l'itération. À la figure 4.10, le signal dont le nom est "Latence 8^e itération" contient le nombre de cycle de latence programmé de la dernière itération dans le fichier de registre APB. Ce signal est comparé avec le compteur principal afin d'activer le compteur du nombre de bits erronés résiduels de la dernière itération.

Pour arrêter la simulation, la sortie du compteur principal est comparée avec le registre contenant le nombre de bits à simuler. Lorsque le nombre de bits transmis devient supérieur au nombre de bits demandés, le signal "CS" est désactivé afin d'arrêter l'ensemble de la simulation. L'arrêt de la simulation active un bit du registre de status qui indique au processeur ARM que la simulation est terminée.

4.6 Extension de la perforation

Les canaux d'information et de parité sont séparés afin de permettre l'intégration future de la perforation dans le système. La principale raison de consommer davantage

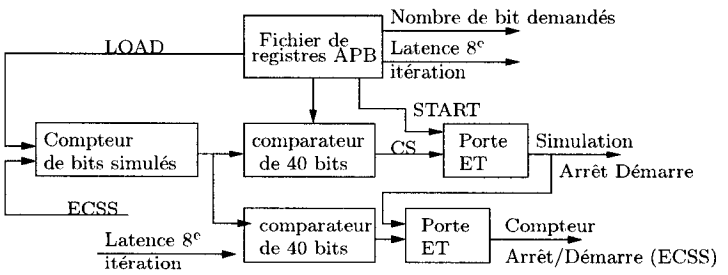


Figure 4.10: Circuit de contrôle de l'ECM

de ressources matérielles pour implémenter deux canaux de communication est principalement due à une limitation de fonctionnalité des modules de gestion d'horloge. La technique de perforation change le taux de codage selon l'équation (2.1). Le taux de codage (R) est égal au nombre de bits d'information transmis (K) divisé par le nombre total de bits transmis ($K+D$) soit $R = K/(K+D)$. Dans un intérêt pratique, on cherche un taux de codage où le nombre de bits de parité est égal à $R = K/(K+1)$. Au niveau matériel, la perforation correspond à diviser l'horloge du canal de parité par K. La seule primitive qui permet de diviser la fréquence d'une horloge dans un FPGA de la famille Virtex-II est un module DCM (Digital Clock Management). Cette primitive divise uniquement l'horloge par un facteur entier. Cette limitation constitue la raison principale d'implémenter 2 canaux de communication.

4.7 Étallonage de E_b/N_o

Les modules traducteurs bit-canal traduisent les bits d'information et de parité qui proviennent du codeur de canal en une valeur de 16 bits selon une forme de signalisation antipodale. Une valeur de 16 bits est associée pour les valeurs binaires 0 et 1. Ces deux valeurs de 16 bits ont été étalonnées avec la source de bruit blanc gaussien (AWGN, Additive White Gaussian Noise) afin d'obtenir le rapport E_b/N_o

voulu. La valeur négative de la signalisation antipodale est 0xFC00 et la valeur positive est 0x0400. Le module AWGN génère un signal de bruit gaussien où la puissance est spécifiée par une entrée indiquant le SNR voulu. La signalisation de référence qu'utilise l'AWGN n'est pas la même que celle utilisée dans cette implémentation et donc une calibration doit être effectuée. Afin d'avoir une valeur de E_b/N_o pour le système qui varie de [2.0,8.0] dB avec les amplitudes de signalisation spécifiées plus haut, la valeur du SNR spécifiée au module AWGN se situe dans la plage [5.0,11.0] dB. Il suffit d'un décalage de 3 dB entre la valeur voulue et la valeur programmée au module AWGN pour calibrer le système. Par exemple si l'on veut 2 dB pour un E_b/N_o du système, il suffit de programmer un SNR de 5 dB au module AWGN. Pour plus d'information sur le IP AWGN se référer à [22].

4.8 Facteur d'accélération

Le facteur d'accélération représente le gain de vitesse d'exécution globale pour la caractérisation des performances du DSI avec la plateforme ARM integrator comparativement à la version logicielle exécutée sur un SUN SPARC V440. Le facteur d'accélération reporté dans [16] est égal à 3908. Dans cet article, la plateforme ARM integrator communique avec un logiciel d'optimisation par l'intermédiaire d'un lien TCP/IP et d'un serveur TCL. Par contre pour l'étude réalisée avec une plage SNR étendue, le facteur d'accélération est encore plus grand car il n'y a pas de temps dépensé pour établir différentes communications avec un logiciel d'optimisation. Dans ce cas, le facteur d'accélération est de 4821 avec un ECM fonctionnant à 18 MHz. Avec une amélioration de la fréquence maximale d'opération de l'DSI, il serait possible d'avoir un facteur d'accélération encore beaucoup plus important. Avec une implémentation fonctionnant à 100 MHz, ce qui est très réalisable avec la technologie FPGA

utilisée, un facteur d'accélération de 26784 serait possible. Ce facteur d'accélération peut s'avérer très utile pour caractériser des DSI plus performants tel que J égal à 15.

4.9 Vérification des performances du décodeur à seuil itératif

Dans cette section, les performances du DSI implémenté sur la plateforme "ARM integrator" sont comparées avec celles obtenues avec la version logicielle. Évidemment, il est à considérer que la caractérisation des performances de correction d'erreurs est un processus pseudo aléatoire où les résultats peuvent variés d'une expérience à l'autre. Dans le graphique 4.11, une comparaison est faite entre les 2 simulateurs. Pour comparer sur un même point d'égalité, la version logicielle a été modifiée afin de représenter les effets de quantification qui sont présents dans l'implémentation matérielle. Les entrées de parité et d'information sont quantifiées selon le même pas de quantification utilisé pour la simulation matérielle. Également une quantification est réalisée à l'intérieur du code C du simulateur logiciel pour montrer l'effet de truncation qui se produit pour le signal λ. Cette truncation est réalisée avec une fonction de la librairie SystemC [24].

Les performances de correction d'erreurs ont été réalisées avec un décodeur dont le code est G={0,27,93,503,600,1247,1646,1714,1825,1825} soit J=10. Les versions logicielle et matérielle donnent sensiblement le même résultat. Dans la figure 4.11, le graphique de droite fut obtenu avec la version logicielle. Le résultat sont très semblable pour la partie supérieure (taux d'erreurs $\geq 10^{-6}$). La partie inférieure est plus susceptible au processus pseudo aléatoire. Pour la simulation logicielle, 100 millions de bits transmis ont été utilisés pour la caractérisation. La graphique 4.12 montre diffé-

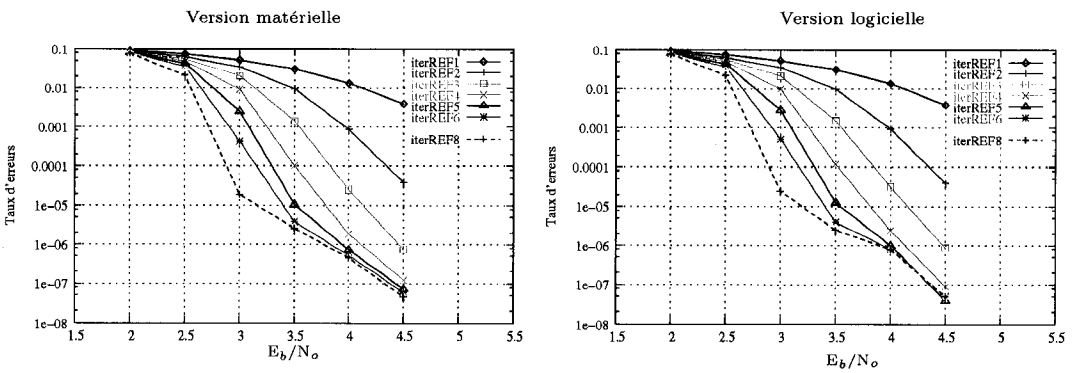


Figure 4.11: Comparaison des résultats entre le simulateur logicielle et matérielle pour 100 millions de bits transmis, J=10 et un coefficient de pondération uniforme de 0.1875

rents points obtenus avec l'accélérateur matérielle pour 100 millions de bits transmis. Il est possible de remarquer que les résultats obtenus de la version logicielle sont dans la zone pseudo aléatoire pour un E_b/N_o et une itération donnés. Il est possible de conclure que les processus de caractérisation en version logicelle et matérielle sont équivalents.

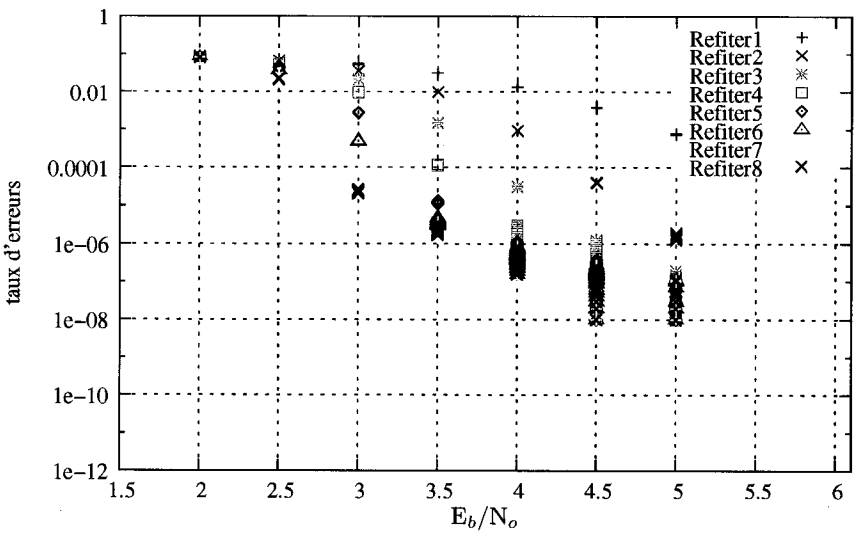


Figure 4.12: Variance du processus pseudo aléatoire de la caractérisation des performances de correction d'erreurs

4.10 Optimisation logicielle-matérielle des coefficients de pondération

Le coefficient de pondération a une influence importante sur les performances de correction d'erreurs du DSI. Dans les recherches antérieures, il a été démontré de manière expérimentale que la valeur du coefficient de pondération qui maximise le potentiel de correction d'erreurs est située proche de 0.2 [7]. Dans cette étude, une même valeur pour l'ensemble des itérations du coefficient de pondération est considérée. La raison principale de considérer seulement des coefficients constants a permis de diminuer la puissance de calcul nécessaire à l'optimisation. Afin d'accélérer le processus de caractérisation du décodeur, il est possible d'accélérer de manière significative les simulations en exploitant une implémentation matérielle. Ainsi avec un tel accélérateur de calcul, il est possible d'éliminer la contrainte imposée de recherche d'un coefficient constant et de trouver un vecteur de coefficients qui constitue un minimum local parmi l'ensemble des solutions possibles.

4.10.1 Résultat de l'optimisation des coefficients de pondération

La figure 4.14 montre les résultats de l'optimisation obtenus avec la plateforme. Le processus d'optimisation débute avec un coefficient de pondérateur de 0.1875 pour l'ensemble des itérations (figure 4.13). 100 millions de bits ont été transmis pour chacune des 7 simulations nécessaires à la caractérisation où le SNR varie de 2.0 dB à 5.0 dB. Au total 148 boucles ont été nécessaires afin de trouver un vecteur de coefficient de pondération qui maximise la déviation de l'équation (4.1). Le vecteurs trouvé est : {0.3125,0.25,0.25,0.3125,0.3125,0.3125,0.3125,0.9375}. Cette optimisation

a nécessité 1.97 heure, mais il faudrait 321.16 jours avec la version logicielle exécutée sur un SPARC V440 pour réaliser le même calcul. L'amélioration la plus marquée est située à 2.5 dB.

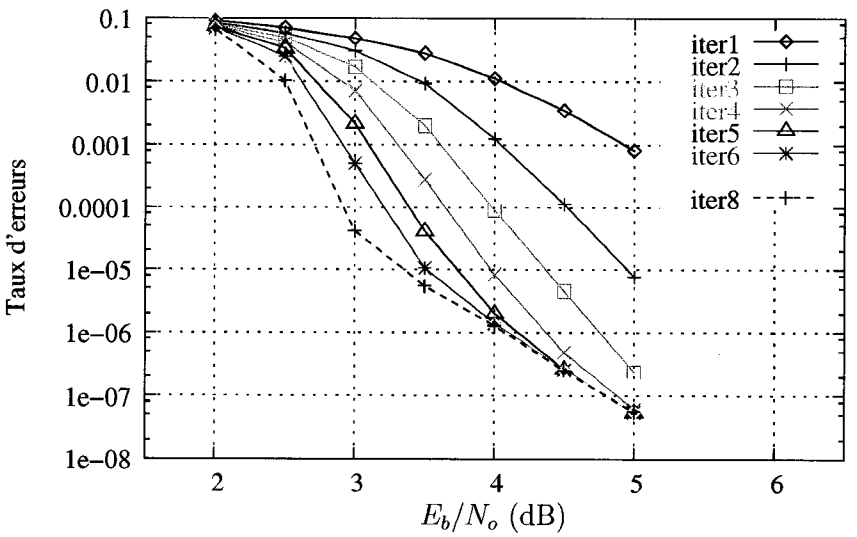


Figure 4.13: Performance du DSI pour un coefficient de pondération uniforme (0.1875)

Il est à noter que l'optimisation s'est effectuée à de faibles SNR. Le critère de l'optimisation est de maximiser la déviation des performances de correction d'erreurs par rapport aux performances de référence montrées à la figure figure 4.13. À partir de 3.5 dB, le taux d'erreurs atteint la saturation du processus itératif où un gain marginal peut peut-être réalisé. Cet observation oriente l'optimisation vers la région à de faibles SNR hors de la saturation du processus itératif. À la dernière itération pour un SNR de 2.5 dB, la performance de correction d'erreurs est améliorée d'un facteur de 70. Également, on observe que les 7^e et 8^e itérations donnent environ la même performance que la 6^e itération dans la plage 3.0 dB et 5.0 dB qui toutes atteignent la saturation du processus de correction d'erreurs. En utilisant 6 itérations au lieu de 8, une économie de 25% en matériel et en puissance est réalisée.

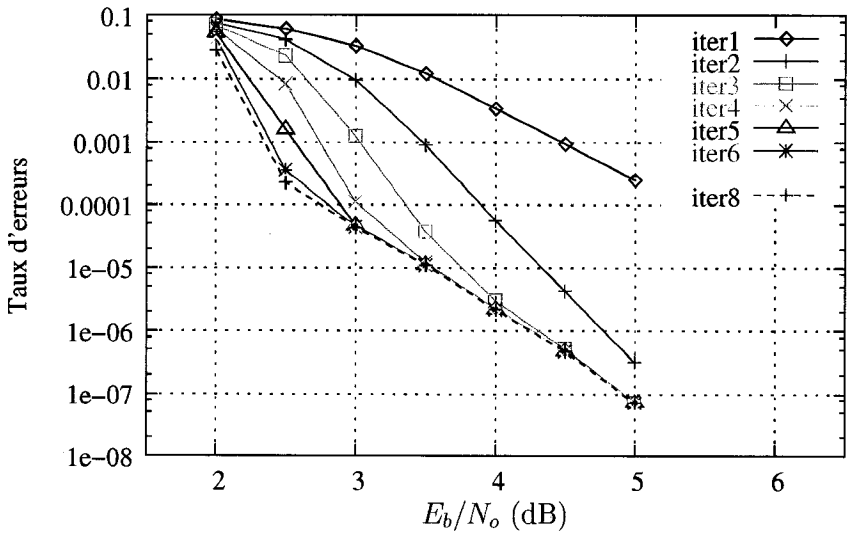


Figure 4.14: Performance de correction d'erreurs du DSI pour J=10

4.11 Étude du comportement du DSI à des SNR de 2 dB à 8 dB

À l'aide de l'ECM, il a été possible de caractériser les performances du DSI à des taux d'erreurs beaucoup plus faibles que ceux étudiés jusqu'à maintenant. L'étude des performances de correction d'erreurs sur une plage E_b/N_o étendue vérifie les performances de correction d'erreurs et la saturation du processus itératif. Les figures 4.15 et 4.16 montrent le taux d'erreurs en fonction de E_b/N_o pour un DSI de J=9. Une même valeur du coefficient de pondération a été utilisée pour l'ensemble des itérations. Une valeur de 0.1875 pour un coefficient de pondération uniforme a été choisi dans la figure 4.15. Cette valeur de 0.1875 est la valeur matérielle la plus près de 0.2 que l'on peut obtenir avec une précision de 4 décimales après le point. Comme prévu, la performance du DSI sature après quelques itérations. Une distinction importante est à noter, c'est le processus itératif qui sature et non les performances de correction d'erreurs. La saturation du processus itératif signifie que l'addition d'une itération

n'apporte plus d'augmentation des performances de correction d'erreurs tandis que la saturation des performances de correction du DSI correspond à l'apparition d'un plancher ou d'une valeur limite du taux d'erreurs qui ne peut être franchi.

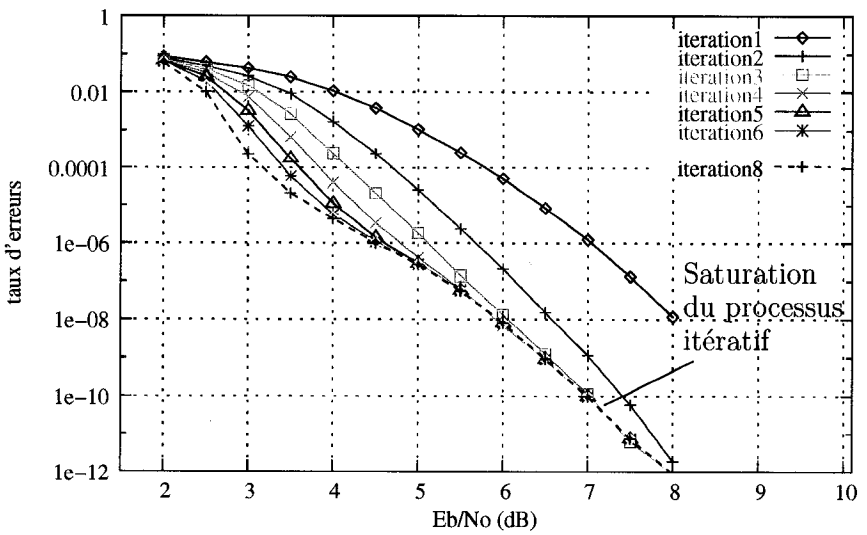


Figure 4.15: Résultats expérimentaux pour 8 itérations du DSI où J=9 (0.1875)

Dans la figure 4.17, les résultats de plusieurs DSI où J varie entre 4 et 9 sont montrés. Grâce à ce graphique, il est possible de confirmer par simulation la tendance de l'évolution de la performance de correction d'erreurs de l'algorithme qui aurait été impossible dans un temps acceptable de vérifier avec la version logicielle. Les résultats présentés à la figure 4.16 ont demandés 2.81 jours de calcul avec l'ECM ce qui correspond à 37.23 années de calcul avec la version logicielle exécutée sur un SUN V440.

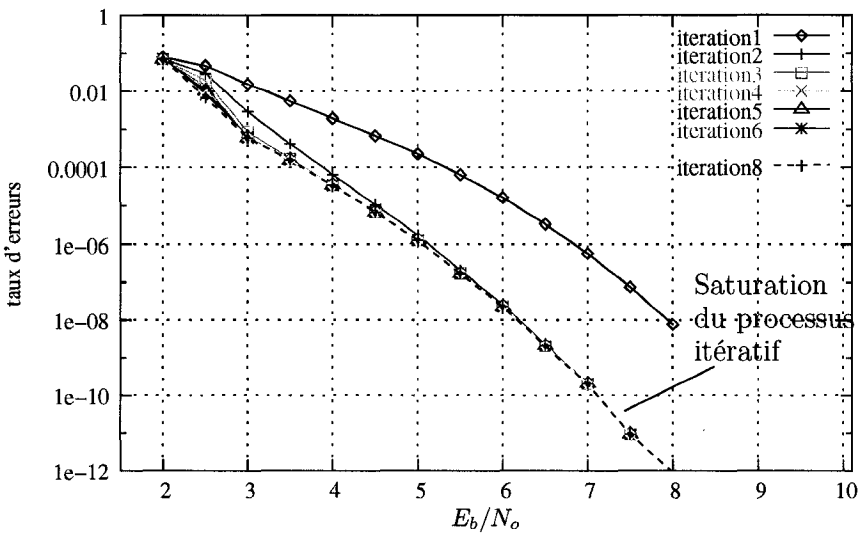


Figure 4.16: Résultats expérimentaux pour 8 itérations du DSI où J=9 (0.9375)

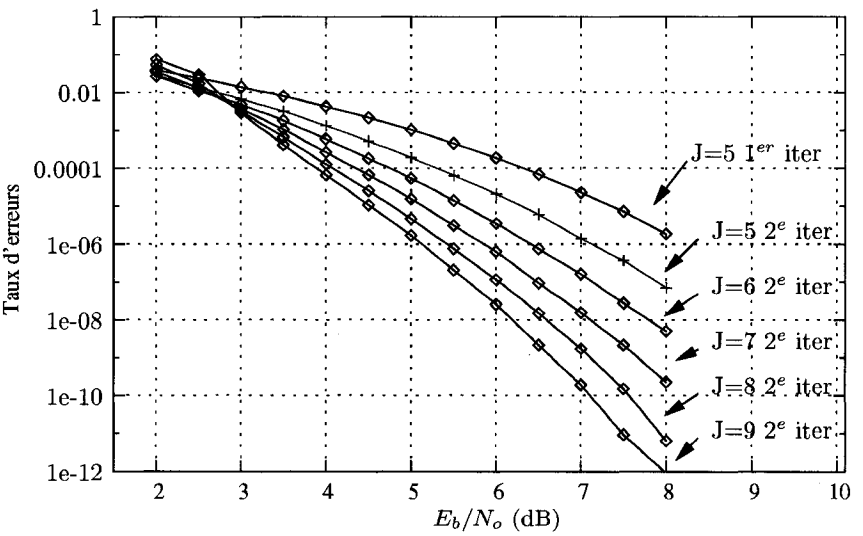


Figure 4.17: Résultats pour différents J de 4 à 9 et le E_b/N_o variant entre 2.0 dB à 8.0 dB

Conclusion

Cette recherche consiste à étudier les différentes facettes de l'implémentation matérielle du décodeur à seuil itératif. Un IP VHDL fut développé et peut implémenter n'importe quel CSO^{2}C-WS pour un taux de codage $\frac{1}{2}$, avec une complexité matérielle raisonnable, et un nombre arbitraire d'itérations. Les détails des différents sous modules ainsi que la flexibilité exigée de ceux-ci par le DSI sont détaillés. Un banc d'essai préliminaire fut développé pour valider la fonctionnalité de l'IP VHDL. Ce banc d'essai compare les signaux internes d'une itération de l'IP VHDL avec les signaux internes générés par un DSI logicielle de référence. Ce banc d'essai a permis de réaliser une vérification préliminaire de la fonctionnalité, mais toutefois ce processus consomme trop de temps pour caractériser les performances de correction du DSI.

Les implémentations de l'opérateur add-min et de l'additionneur furent optimisées pour minimiser les ressources matérielles et le délai combinatoire total. La représentation binaire signe-amplitude favorise une implémentation moins gourmande et plus rapide de l'opérateur add-min tandis que la représentation complément à deux favorise l'implémentation de l'additionneur. Étant donné le nombre beaucoup plus important des opérateurs add-min, la représentation signe-amplitude favorise une consommation moins importante des ressources matérielles d'une itération du DSI. À partir des données extraites de l'étude de la consommation des ressources matérielles des différents opérateurs en fonction du nombre de ports d'entrées et du nombre de bits par

port, il est possible d'approximer le délai et la complexité matérielle d'un quelconque décodeur où $J \leq 15$.

Deux optimisations matérielles ont été ajoutées à l'architecture. La technique de saturation élimine la partie entière de $\lambda^{(\mu)}$. Cette optimisation est possible car $\lambda^{(\mu)}$ est toujours acheminé à des opérateurs add-min dont l'une des entrées ne contient aucune partie entière. Un nouvel opérateur qui implémente la saturation est introduit à la suite de l'additionneur pour limiter la résolution de $\lambda^{(\mu)}$ à sa partie fractionnaire. La deuxième optimisation importante consiste à l'élaboration d'une stratégie du réajustement de l'ordonnancement couplée à la recherche de nouveaux codes qui repousse les limitations du réajustement de l'ordonnancement. Le facteur limitant le nombre de bascules disponibles pour le réajustement de l'ordonnancement est la distance minimale entre deux indices du CSO^{2}C-WS choisi. Une étude de l'accroissement de cette distance minimale en fonction de la longueur du code permet de montrer qu'une accélération importante peut être réalisée à un faible coût matériel.

Finalement, pour accélérer le processus de caractérisation des performances de correction d'erreurs, un système de communication complet fut implémenté en matériel sur la plateforme "ARM integrator". Ce système contient une source de bruit blanc dont la puissance peut être configurée. Ainsi, ce système de communication est utilisé pour évaluer les performances de corrections d'erreurs du décodeur à seuil itératif à différents SNR. Cette méthode d'accélération est principalement possible dû à la présence du composant matériel configurable à des prix abordables. Cette approche d'augmentation de la puissance de calcul nécessaire à la résolution pour différents problèmes s'avère un moyen peu coûteux de résoudre un problème lorsque la puissance de calcul est considérable et que la complexité d'une solution matérielle reste raisonnable.

En plus de servir à la vérification de la fonctionnalité du DSI, cette plateforme a

servi d'accélérateur de calcul pour trouver un vecteur de coefficients de pondération qui minimise localement les performances de correction d'erreurs. Pour ce faire, la plateforme fut interfacée avec un logiciel d'optimisation à l'aide d'un serveur TCL. Le logiciel d'optimisation cherche à maximiser la différence entre la performance de correction d'erreurs de la meilleure valeur publiée précédemment, soit de 0.1875 pour l'ensemble des itérations, et le vecteur de coefficients de pondération recherché. Un vecteur de coefficients de pondération fut trouvé soit {0.3125,0.25,0.25,0.3125,0.3125,0.3125, 0.3125,0.9375} qui permet un gain de la performance de corrections d'erreurs de 70 à 2.5 dB. Un gain généralisé des performances de corrections d'erreurs du DSI est réalisé. De plus cette plateforme fut utilisée pour valider la performance de correction d'erreurs à des SNR encore inexplorés. Au total, la caractérisation à ces SNR a nécessité 2.81 jours de calcul avec la plateforme, ce qui représente un temps équivalent de 37.23 années de calcul avec la version logicielle exécutée sur un SUN V440.

Quant aux travaux fugures, plusieurs voies de recherche restent inexplorées suite à ce mémoire. Entre autres, au niveau architectural, il serait intéressant de modifier l'implémentation actuelle afin d'intégrer la stratégie du réajustement de l'ordonnancement et d'étudier les fréquences d'horloge maximales possibles d'atteindre. Également, il serait approprié avec l'accélération de la fréquence d'horloge de développer une bibliothèque VHDL de registres à décalage qui permet d'intégrer plusieurs registres à décalage à l'intérieur d'un même "block RAM". Ainsi, des décodeurs ayant un J plus élevé pourraient être implémentés.

À l'aide de la plateforme de caractérisation des performances de correction d'erreurs plusieurs optimisations au niveau architectural peut être réalisées. Premièrement, une étude plus élaborée de la résolution binaire nécessaire à l'intérieur d'une itération permettrait de minimiser la consommation des ressources mémoires tout en minimisant la perte de performances de correction d'erreurs. Il serait par exemple

possible de pousser davantage l'analyse avec cet accélérateur de calcul et d'attaquer une optimisation de la largeur non uniforme des pas de quantification du CAN.

Finalement, pour augmenter l'efficacité spectrale dédiée à l'information, la technique de perforation peut être intégrée facilement à l'IP développé. D'ailleurs la plateforme d'accélération de calcul a été conçue pour une intégration future d'un DSI dont la perforation est implémentée.

Bibliographie

[1] ASHENDEN, J. J. *The Designer's Guide to VHDL.* Morgan Kaufmann ; 2 edition, 2002.

[2] BAECHLER, B. Analyse et détermination de codes doublement orthogonaux pour décodage itératif. Mémoire, Département de Génie électrique et Informatique, École Polytechnique de Montréal, Juin 2000.

[3] BERROU, C., GLAVIEUX, A., AND THITIMAJSHIMA, P. Near shannon limit error-correcting coding and decoding : Turbo-codes. In *Proc. IEEE International Symposium on Circuits and Systems* (Mai 2005), vol. 2, pp. 290–294.

[4] BRISO-RODRIGUEZ, C., AND ALONSO-MONTES, J. Multipath hardware and software mobile channel simulator. In *IEEE VTC 2001 Vehicular Technology Conference* (Mai 2001), vol. 4, pp. 3060–3063.

[5] CANTIN, M.-A., SAVARIA, Y., AND LAVOIE, P. A comparison of automatic word length optimization procedures. In *The IEEE Int. Symposium on Circuits and Systems (ISCAS)* (Mai 2002), vol. 2, pp. 612–615.

[6] CANTIN, M.-A., SAVARIA, Y., PRODANOS, D., AND LAVOIE, P. An automatic word length determination method. In *The IEEE Int. Symposium on Circuits and Systems (ISCAS)* (Kobe, Japan, Mai 2001), vol. 5, pp. 53–56.

[7] CARDINAL, C. *Décodage à seuil itératif des codes convolutionnel doublement orthogonaux.* Thèse de doctorat, École Polytechnique de Montréal, Montréal, Juin 2001.

[8] CARDINAL, C., HACCOUN, D., AND GAGNON, F. Iterative threshold decoding without interleaving for convolutional self-doubly orthogonal codes. *IEEE Trans. Commun. 51* (Aout 2003), 1274–1284.

[9] CARDINAL, C., HACCOUN, D., GAGNON, F., AND BATANI, N. Convolutional self doubly orthogonal codes for iterative decoding without interleaving. In *Proceedings. 1998 IEEE International Symposium on* (Aout 1998), p. 280.

[10] DRU, . F. Décodeur à seuil itératif des codes convolutionnels doublement orthogonaux perforés et application aux modulations multiniveaux. Mémoire, Département de Génie électrique et Informatique, École Polytechnique de Montréal, Juin 2001.

[11] DUBOIS, M., SAVARIA, Y., AND HACCOUN, D. On low power shift register hardware realizations for convolutional encoders and decoders. In *Circuits and Systems, 2004. NEWCAS 2004. The 2nd Annual IEEE Northeast Workshop on* (Montréal, Canada, Juin 2004).

[12] FURBER, S., EDWARDS, D., AND GARSIDE, J. Amulet3 : a 100 mips asynchronous embedded processor. In *Computer Design, 2000. Proceedings. 2000 International Conference on* (Septembre 1998), pp. 329 – 334.

[13] KIM, S., AND SRIDHAR, R. Comparison of power consumption among asynchronous design styles with their synchronous counterparts. In *Circuits and Systems, 1994., Proceedings of the 37th Midwest Symposium on* (Aout 1994), vol. 1, pp. 7 – 10.

[14] MARTIN, A., NYSTROM, M., PAPADANTONAKIS, K., PENZES, P., PRAKASH, P., WONG, C., CHANG, J., KO, K., LEE, B., E.OU, PUGH, J., TALVALA, E.-V., TONG, J., AND TURA, A. The lutonium : a sub-nanojoule asynchronous 8051 microcontroller. In *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on* (Kobe, Japan, Mai 2003), pp. 11–23.

[15] PALNITKAR, S. *Verilog HDL (2nd Edition).* Prentice Hall PTR, 2003.

[16] PROVOST, G., CANTIN, M.-A., SAWAN, M., CARDINAL, C., SAVARIA, Y., AND HACCOUN, D. Fast parameters optimization of an iterative decoder using a configurable hardware accelerator. In *to be published in The IEEE Int. Symposium on Circuits and Systems (ISCAS)* (Japan, 2005).

[17] RABAEY, J. M., CHADRAKASAN, A., AND NICOLIĆ, B. *Digital Integrated Circuit, a design perspective second edition.* Pearson Education, Inc, 2003.

[18] Site internet de celoxica pour synthèse avec systemc et c. `http://www.celoxica.com/`, 2005.

[19] Site internet de la compagnie altera :. `http://www.altera.com/`, 2005.

[20] Site internet de la compagnie ftdi :. `http://www.ftdichip.com/`, 2005.

[21] Site internet de la compagnie xilinx :. `http://www.xilinx.com/`, 2005.

[22] Site internet de la documentation du core de bruit blanc gaussien de la compagnie xilinx. `http://www.xilinx.com/ipcenter/awgn/index.htm`, 2005.

[23] Site internet de l'estimateur de puissance de xilinx :. `http://www.xilinx.com/cgi-bin/powerweb.pl`, 2005.

[24] Site internet de systemc. `http://www.systemc.org/`, 2005.

[25] Site internet des 500 ordinateurs les plus puissants au monde. `http://www.top500.org/`, 2005.

[26] WELLS, R. B. *Applied coding and information theory for engineers.* Prentice Hall, 1999.

Annexe A

Environnement de caractérisation matérielle

A.1 Fichier Logic.c

```
/*
 * ------------------------------------------------------------
 * Fichier :   logic.c
 * version :   Version 1.0
 *
 * Ce fichier contient le programme du processeur ARM
 * de la plateforme ARM integrator
 *
 * ------------------------------------------------------------
 *

 */


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

// Contient les différentes déclarations nécessaires pour
// l'adresse des différents fichiers de registres APB
// Déclaration des différents masques
// Déclaration de différents constantes et prototypes de fonctions

#include "logic.h"
#include "platform.h"

#define EXIT_FAILURE 1


void loadParameter(void);
```

```
//Frontière servant à la configuration du pseudo-ADC
int boarder [NBRE_BOARDER];
// Assignation binaire de chacun des pas de quantification du pseudo-ADC
int mapping [NBRE_MAPPING];

// Coefficient de pondération de chaque itération
int coeff   [NBRE_ETAGE];

// Quantificateur (MUX) pour chaque itération
// Il y a quatre par itération sauf pour la derniere qui en a 3
int mux     [NBRE_ETAGE][NBRE_MUX];

// Valeur de l'entier pour le SNR spécifié au module AWGN
int ebnoInt;
// Valeur de la partie fractionnaire pour le SNR spécifié au module AWGN
int ebnoFract;


// Tableau contenant les résultats de la simulation
long long result  [NBRE_ETAGE];

// Nombre de cycle pour chaque itération avant de démarrage
// le compteur d'erreurs qui lui est associé
long long latence_count [NBRE_ETAGE];

// Mot d'initialisation pour la source binaire
long long seed;

// mot d'initiation spécifié par l'usaager
long long seedinit=0x0000008000200003;  // long long = 64 bits
```

```
// mot d'initialisation pour chacun des LFSR
long long seed_latence_lfsr [NBRE_ETAGE] ;

// Variable temporaire servant au calcul des mots d'initialisation
long long temp_seed_lfsr [NBRE_ETAGE] ;

// Variable temporaire
int temp ;
int addr ;


// variable contenant la valeur du SNR maintennat
int iteration ;



//
int cyclemax=15 ;
int cyclestart=-1 ;
//int iterebno[7]={0x50,0x55,0x60,0x65,0x70,0x75,0x80} ; // ( cest le bon)


int iterebno[15]={0x50,0x55,0x60,0x65,0x70,0x75,0x80,0x85,0x90,0x95,0xA0,0xA5,
                  0xB0,0xB5,0xC0};

// Constantes de 16 bits pour la signalisation antipodale
int zero_bit_value=0xFC00 ;
int one_bit_value =0x0400 ;

// nombre de bits transmis à simuler
```

```
long long cyclessim;

// Macro retournant les 32 bits supérieurs d'une variable de type long long
#define hi64(a) (((int*) &a)[1])

// Type definitions
typedef volatile unsigned int * vuint;

int main(void)
{
  int i;
  int add;
  char * fileName, * filelckplat;
  char lockstring [20];

  int  count, ltbase, intbase, apdet, intbit;
  FILE * stream, * streamlckplateforme;
  fileName="C :\\ghislain\\serveur\\interface.lck";
  filelckplat="C :\\ghislain\\serveur\\plateforme.lck";

  // Verification de l'ouverture du fichier
  if ((streamlckplateforme = fopen (filelckplat, "w")) == NULL) {
    fprintf (stderr, "Impossible d'ouvrir le fichier \"%s\"  en lecture lock
             plateforme \n", filelckplat);

    exit (EXIT_FAILURE);
}

    // Display the program version
    printf ("\n\nLogic Tile Example2 Test Suite, version 1.0\n");
    printf ("Copyright (C) ARM Ltd 2002. All rights reserved.\n\n");
```

```
// Detect if fitted to Integrator/AP or /CP
apdet = TRUE;
if((*AP_SC_DEC) & EXP_HDR_0) {
    printf ("Integrator/AP detected\n");
    ltbase  = LT_APB_BASE1;
    intbase = LT_INT_BASE1;
}
else if((*AP_HDR_ID & 0x0F) == CP_CMID_REV) {
    printf ("Integrator/CP detected\n");
    ltbase  = LT_APB_BASE2;
    intbase = LT_INT_BASE2;
    apdet = FALSE;
}
else {
    printf("ERROR : Unable to detect Integrator/AP or Integrator/CP\n");
    return TRUE;
}

// Turn off all LEDS
*(vuint)(ltbase + LT_LEDS)     = 0x0F;
*(vuint)(ltbase + IM_LT1_LEDS) = 0xFF;



// Initialisation des Seed des LFSR
seed =seedinit;

// Initialisation des LFSR
calculseedlfsr();
```

```
    // Nombre iteration pour ebno pour tracer un graphique avec 2.0, 2.5 3.0 3.5
    4.0 4.5 5.0
    iteration=cyclestart;


  // charge le mapping, les frontieres et coefficent pour decodeur
  // A partir du fichier sur l'ordinateur Host
  loadParameter();


// Initialisation des poids des valeurs binaires
  add=ltbase +LM_ZERO_BIT_VALUE;
*(vuint)(add)=zero_bit_value;

  add=ltbase +LM_ONE_BIT_VALUE;
*(vuint)(add)= one_bit_value;

while(1) {

   do
   {
   if ((stream = fopen (fileName, "r")) == NULL) {
     fprintf (stderr, "Impossible d'ouvrir le fichier \"%s\"  en lecture.\n",
              fileName);
     //fclose(fileName);
     exit (EXIT_FAILURE);
   }

   fscanf (stream, "%s", &lockstring);
   printf( "%s",lockstring);
   printf ("\n");
```

```
    fclose(stream);
    }while( strcmp( lockstring, "start"));

    // Ajout Cantin
    fprintf(streamlckplateforme, "%s", "wait");

    // Permet de changer le seed a chaque boucle
      if ( iteration == (cyclemax-1))
       iteration=0;
      else
       iteration++;

// Configuration du nombre de bits à simuler pour
// les différents SNR
// Pratique pour diminuer le temps de simluation
// Peut etre configurer dans le programme ARM ou dans
// le fichier config.txt
//  J10
switch(iteration) {

case 0  : cyclessim=100000; break;    // 2.0dB
case 1  : cyclessim=100000; break;    // 2.5dB
case 2  : cyclessim=100000000; break;  // 3.0dB
case 3  : cyclessim=900000000; break;  // 3.5dB
case 4  : cyclessim=0x4FFFFFFFFF; break;  // 4.0dB
case 5  : cyclessim=0x4FFFFFFFFF; break;  // 4.5dB
case 6  : cyclessim=0x4FFFFFFFFF; break;  // 5.0dB
case 7  : cyclessim=0x4FFFFFFFFF; break;    // 5.5dB
case 8  : cyclessim=0x4FFFFFFFFF; break;    // 6.0dB
case 9  : cyclessim=0x4FFFFFFFFF; break;  // 6.5dB
case 10  :cyclessim=0x4FFFFFFFFF; break;  // 7.0dB
```

```
case 11  :cyclessim=0xDFFFFFFFFF; break;  // 7.5dB
case 12  :cyclessim=0xFFFFFFFFF; break;  // 8.0dB
case 13  :cyclessim=0xFFFFFFFFFF; break;  // 8.5dB
case 14  :cyclessim=0xFFFFFFFFFF; break;  // 9.0dB

}

   fprintf(streamlckplateforme, "%s", "wait");

// Verifie la validite des frontiere
  if(verifiefrontiere())
   printf("Probleme frontieres \n");


  // barre le serveur afin d'attendre le resultat

  // Écriture des latence utilisée
   for ( i =0; i< NBRE_ETAGE; i++) {
   // latence_count[i]=(LATENCE+1)*(i+1);
    //latence_count[i]+=DATA_PATH_LATENCE;
    //latence_count[i]+=15;
    latence_count[i]=(LATENCE+1) * (NBRE_ETAGE+2);
   }


// Ecriture du SNR voulue
  ebnoInt = iterebno[iteration];
  ebnoInt = ebnoInt & 0xF0;
  ebnoInt = ebnoInt >>4;

  ebnoFract = iterebno[iteration];
```

```
ebnoFract = ebnoFract & 0xF;

// Ecriture des parametres dans le fichier de registres APB
ecrire_lireReg(0,ltbase);

// Fonction de démarrage de la fonction
DemarreSim(ltbase);

// attend la fin de la simulation
while( !( (*(vuint)(ltbase + LM_INT2)) & 0x01)  );

// Lit les résultats de la simulation du décodeur des registres
LitResult(ltbase);

// Ecrit les résultats dans le fichier résultats
//ecrit resultat sim fichier result.dat
ecritResult();


// Relache le serveur
fprintf(streamlckplateforme, "%s", "fin");

fprintf(streamlckplateforme, "%s", "wait");

fclose(streamlckplateforme);


if (  iteration==cyclemax)
 iteration=0;
}
```

```
} // Fin du main


////////////////////////////////////////////////////////////////////////////////
/////
// Cette fonction lit des différents de configuration de la simluation dans le
fichier
// texte config.txt
//
//
////////////////////////////////////////////////////////////////////////////////
/////

void loadParameter(void)
{

  int i =0,j=0;


  FILE * stream;
  char * fileName="C :\\ghislain\\serveur\\configDec.txt";



  // Verification de l'ouverture du fichier
  if ((stream = fopen (fileName, "r")) == NULL) {
    fprintf (stderr, "Impossible d'ouvrir le fichier \"%s\"  en lecture.\n",
             fileName);
    exit (EXIT_FAILURE);
  }
```

```
  printf ("Acquisition du nombre de cycles de simulation \n");printf ("\n");
  fscanf (stream, "%lld", &cyclessim);
  //fscanf (stream, "%d", &cyclessim);
  printf( "%lld",cyclessim);
  printf ("\n");

  //ebno

printf ("Acquisition ebnoInt \n");printf ("\n");
fscanf (stream, "%d", &ebnoInt);
printf( "%d",ebnoInt);
printf ("\n");

printf ("Acquisition ebnoFract \n");printf ("\n");

fscanf (stream, "%d", &ebnoFract);
printf( "%d",ebnoFract);
printf ("\n");

printf ("Acquisition des coefficients \n");printf ("\n");
for ( i =0; i< NBRE_ETAGE; i++) {

   fscanf (stream, "%d", &coeff[i]);
   printf( "%d",coeff[i]);
   printf ("\n");
}

printf ("Acquisition des frontieres\n");printf ("\n");
for ( i =0; i< NBRE_ETAGE; i++) {
   for ( j=0; j< NBRE_MUX; j++) {
   fscanf (stream, "%d", &mux[i][j]);
```

```
   printf( "%d",mux[i][j]);
   printf ("\n");
   }
}

printf ("Acquisition des frontieres\n");printf ("\n");
for ( i =0; i< NBRE_BOARDER; i++) {

   fscanf (stream, "%d", &boarder[i]);
   printf( "%d",boarder[i]);
   printf ("\n");
}
printf ("Acquisition des mapping memoire \n");printf ("\n");
for ( i =0; i< NBRE_MAPPING; i++) {

   fscanf (stream, "%d", &mapping[i]);
   printf( "%d",mapping[i]);
   printf ("\n");




}
fclose (stream);
}

////////////////////////////////////////////////////////////////////////////////
/////
//
// Affiche le contenu des variables globales de configuration
// Fonction servant au déverminage des registres
```

```
//
//////////////////////////////////////////////////////////////////////////////
/////

void afficheReg( void ) {

  int i;

  // Utilisation avec le quantification version debug
     printf ("Affiche les frontieres\n");printf ("\n");
   for ( i =0; i< NBRE_BOARDER; i++) {

   printf( "%s", "frontiere #");
   printf( "%d",i);printf ("\n");
   printf( "%d",boarder[i]);
   printf ("\n");
   }
   printf ("Acquisition des mapping memoire \n");printf ("\n");
   for ( i =0; i< NBRE_MAPPING; i++) {

   printf( "%s", "mapping #");
   printf( "%d",i);printf ("\n");
   printf( "%d",mapping[i]);
   printf ("\n");
   }

  printf ("Affichage des coefficients \n");printf ("\n");
    for ( i =0; i< NBRE_ETAGE; i++) {

  printf( "%s", "coeff etage #");
```

```
  printf( "%d",i);printf ("\n");
  printf( "%d",coeff[i]);
  printf ("\n");
  }

  printf ("Affichage du nombre de cycles de simulation \n");printf ("\n");
  printf( "%s", "cyclesim #");
  printf( "%d",i);printf ("\n");
  printf( "%lld",cyclessim);
  printf ("\n");
}

////////////////////////////////////////////////////////////////////////////////
/////
//
// Fonction servant à transférer les variables globales vers le fichier de
registre APB
// et vise-verso
//
////////////////////////////////////////////////////////////////////////////////
/////


void ecrire_lireReg(int lire,int ltbase ) {


   int i=0,j=0;
   int tempbas=0,temphaut=0;
   int temp=0;
   int add;
   int cnt_mapping=0, cnt_boarder=0,add_temp=0,condition=0xFFFF;
```

```
// Fichier de debug qui montre le contenu du pseudo-ADC pour
// Chacune des adresses
   char * fileName="C :\\ghislain\\serveur\\BlockRamData.txt";


   long long templong;

   FILE * blockRAMstream ;

// Débarre certains registres
   *(vuint)(ltbase + LT_LOCK)=UNLOCK;

   printf ("Ecrit registres frontieres \n");
   printf ("\n");


// Programmation des frontieres  pour le pseudo-ADC fait avec plusieurs Block
RAM
// ouverture du fichier BlockRamData.txt
if ((blockRAMstream = fopen (fileName, "w")) == NULL) {
   fprintf (stderr, "Impossible d'ouvrir le fichier \"%s\"  en lecture
            BlockRamData.tx \n", fileName);
   exit (EXIT_FAILURE);
   }

if ( iteration ==0 )  {
   cnt_boarder=0; cnt_mapping=0; add_temp;

   // Programme chacune des adresses ( 16 bits)
   for ( add =32768; add <= condition; add++) {
```

```
if ( add & 0x8000 ) {   // nombre negatif ( adresse Block RAM )
if( add < boarder[cnt_boarder]) {

  // Ecriture de l'adresse dans le registre d'adresse du block ram
  add_temp=LM_BOARDERADD_BASE_RAM_ADRESS;
  add_temp+=ltbase;
  *(vuint)(add_temp)=add;

  // Debug
  // Écriture de l'adresse programmé dans le fichier BlockRamData.txt
  fprintf(blockRAMstream, "%s", "0x");
  fprintf(blockRAMstream, "%x", add);
  fprintf(blockRAMstream, "%s", " ");

  // Ecriture de la donnee dans le registre DATA du block ram
  add_temp=LM_BOARDERADD_BASE_RAM_DATA;
  add_temp+=ltbase;
  *(vuint)(add_temp)= mapping[cnt_mapping];

  // Debug
  // Écriture de la donnée programmé dans le fichier BlockRamData.txt
  fprintf(blockRAMstream, "%s", "0x");
  fprintf(blockRAMstream, "%x\n", mapping[cnt_mapping]);

  // Active l'ecriture sur le block RAM
  add_temp=LM_BOARDERADD_BASE_RAM_CONTROL;
  add_temp+=ltbase;
  *(vuint)(add_temp)=WE_BR;

  // Désactive l'écriture
  add_temp=LM_BOARDERADD_BASE_RAM_CONTROL;
```

```
  add_temp+=ltbase;
  *(vuint)(add_temp)=RESET_BR;


}
// condition traitant si la frontière est incluse dans le pas de
// quantification précédent ou suivant
else if ( boarder[cnt_boarder]== add)  {

// Ecriture de l'adresse dans le registre d'adresse du block ram
  add_temp=LM_BOARDERADD_BASE_RAM_ADRESS;
  add_temp+=ltbase;
  *(vuint)(add_temp)=add;

  // Debug
  // Écriture de l'adresse programmé dans le fichier BlockRamData.txt
  fprintf(blockRAMstream, "%s", "0x");
  fprintf(blockRAMstream, "%x", add);
  fprintf(blockRAMstream, "%s", " ");

  // Ecriture de la donnee dans le registre DATA du block ram
  add_temp=LM_BOARDERADD_BASE_RAM_DATA;
  add_temp+=ltbase;
  *(vuint)(add_temp)= mapping[cnt_mapping+1];

   // Debug
  // Écriture de la donnée programmé dans le fichier BlockRamData.txt
  fprintf(blockRAMstream, "%s", "0x");
  fprintf(blockRAMstream, "%x\n", mapping[cnt_mapping]);

  // Active l'écriture sur le block RAM
```

```
  add_temp=LM_BOARDERADD_BASE_RAM_CONTROL;
  add_temp+=ltbase;
  *(vuint)(add_temp)=WE_BR;

  // Désactive l'écriture
  add_temp=LM_BOARDERADD_BASE_RAM_CONTROL;
  add_temp+=ltbase;
  *(vuint)(add_temp)=RESET_BR;

  cnt_mapping++;
  cnt_boarder++;
}
// traitement de la fin de la region zero dans la partie negative
else if ( add==0xFFFF ) {

  // Ecriture de l'adresse dans le registre d'adresse du block ram
  add_temp=LM_BOARDERADD_BASE_RAM_ADRESS;
  add_temp+=ltbase;
  *(vuint)(add_temp)=add;

  // Debug
  // Écriture de l'adresse programmé dans le fichier BlockRamData.txt
  fprintf(blockRAMstream, "%s", "0x");
  fprintf(blockRAMstream, "%x", add);
  fprintf(blockRAMstream, "%s", " ");

  // Ecriture de la donnee dans le registre DATA du block ram
  add_temp=LM_BOARDERADD_BASE_RAM_DATA;
  add_temp+=ltbase;
  *(vuint)(add_temp)= mapping[cnt_mapping];
```

```
    // Debug
    // Écriture de la donnée programmé dans le fichier BlockRamData.txt
    fprintf(blockRAMstream, "%s", "0x");
    fprintf(blockRAMstream, "%x\n", mapping[cnt_mapping]);

    // Active l'écriture sur le block RAM
    add_temp=LM_BOARDERADD_BASE_RAM_CONTROL;
    add_temp+=ltbase;
    *(vuint)(add_temp)=WE_BR;

    // Désactive l'écriture
    add_temp=LM_BOARDERADD_BASE_RAM_CONTROL;
    add_temp+=ltbase;
    *(vuint)(add_temp)=RESET_BR;


    add=-1; // pour commencer partie positive
    condition=0x7FFF;
}

// Traitement de la region zero partie negatives
else {

    // Ecriture de l'adresse dans le registre d'adresse du block ram
    add_temp=LM_BOARDERADD_BASE_RAM_ADRESS;
    add_temp+=ltbase;
    *(vuint)(add_temp)=add;

    // Debug
    // Écriture de l'adresse programmé dans le fichier BlockRamData.txt
    fprintf(blockRAMstream, "%s", "0x");
```

```
  fprintf(blockRAMstream, "%x", add);
  fprintf(blockRAMstream, "%s", " ");

  // Ecriture de la donnee dans le registre DATA du block ram
  add_temp=LM_BOARDERADD_BASE_RAM_DATA;
  add_temp+=ltbase;
  *(vuint)(add_temp)= mapping[cnt_mapping];

  // Debug
  // Écriture de la donnée programmé dans le fichier BlockRamData.txt
  fprintf(blockRAMstream, "%s", "0x");
  fprintf(blockRAMstream, "%x\n", mapping[cnt_mapping]);

  // Active l'écriture sur le block RAM
  add_temp=LM_BOARDERADD_BASE_RAM_CONTROL;
  add_temp+=ltbase;
  *(vuint)(add_temp)=WE_BR;

  // Désactive l'écriture
  add_temp=LM_BOARDERADD_BASE_RAM_CONTROL;
  add_temp+=ltbase;
  *(vuint)(add_temp)=RESET_BR;

}


} else  {


if(add< boarder[cnt_boarder]) {
```

```
// Ecriture de l'adresse dans le registre d'adresse du block ram
add_temp=LM_BOARDERADD_BASE_RAM_ADRESS;
add_temp+=ltbase;
*(vuint)(add_temp)=add;

// Debug
// Écriture de l'adresse programmé dans le fichier BlockRamData.txt
fprintf(blockRAMstream, "%s", "0x");
fprintf(blockRAMstream, "%x", add);
fprintf(blockRAMstream, "%s", " ");

// Ecriture de la donnee dans le registre DATA du block ram
add_temp=LM_BOARDERADD_BASE_RAM_DATA;
add_temp+=ltbase;
*(vuint)(add_temp)= mapping[cnt_mapping];

// Debug
// Écriture de la donnée programmé dans le fichier BlockRamData.txt
fprintf(blockRAMstream, "%s", "0x");
fprintf(blockRAMstream, "%x\n", mapping[cnt_mapping]);

// Active l'écriture sur le block RAM
add_temp=LM_BOARDERADD_BASE_RAM_CONTROL;
add_temp+=ltbase;
*(vuint)(add_temp)=WE_BR;

// Désactive l'écriture
add_temp=LM_BOARDERADD_BASE_RAM_CONTROL;
add_temp+=ltbase;
*(vuint)(add_temp)=RESET_BR;
```

```
}

else if ( boarder[cnt_boarder]== add)  {

  // Ecriture de l'adresse dans le registre d'adresse du block ram
  add_temp=LM_BOARDERADD_BASE_RAM_ADRESS;
  add_temp+=ltbase;
  *(vuint)(add_temp)=add;

  // Debug
  // Écriture de l'adresse programmé dans le fichier BlockRamData.txt
  fprintf(blockRAMstream, "%s", "0x");
  fprintf(blockRAMstream, "%x", add);
  fprintf(blockRAMstream, "%s", " ");

  // Ecriture de la donnee dans le registre DATA du block ram
  add_temp=LM_BOARDERADD_BASE_RAM_DATA;
  add_temp+=ltbase;
  *(vuint)(add_temp)= mapping[cnt_mapping];

  // Debug
  // Écriture de la donnée programmé dans le fichier BlockRamData.txt
  fprintf(blockRAMstream, "%s", "0x");
  fprintf(blockRAMstream, "%x\n", mapping[cnt_mapping]);

  // Active l'écriture sur le block RAM
  add_temp=LM_BOARDERADD_BASE_RAM_CONTROL;
  add_temp+=ltbase;
  *(vuint)(add_temp)=WE_BR;
```

```
    // Désactive l'écriture
    add_temp=LM_BOARDERADD_BASE_RAM_CONTROL;
    add_temp+=ltbase;
    *(vuint)(add_temp)=RESET_BR;

    if ( cnt_boarder!=NBRE_BOARDER-1 ) {
      cnt_mapping++;
    }
    cnt_boarder++;
}
```

```
else { // les dernier cas
    // Ecriture de l'adresse dans le registre d'adresse du block ram
    add_temp=LM_BOARDERADD_BASE_RAM_ADRESS;
    add_temp+=ltbase;
    *(vuint)(add_temp)=add;

    // Debug
    // Écriture de l'adresse programmé dans le fichier BlockRamData.txt
    fprintf(blockRAMstream, "%s", "0x");
    fprintf(blockRAMstream, "%x", add);
    fprintf(blockRAMstream, "%s", " ");

    // Ecriture de la donnee dans le registre DATA du block ram
    add_temp=LM_BOARDERADD_BASE_RAM_DATA;
    add_temp+=ltbase;
```

```
        *(vuint)(add_temp)= mapping[cnt_mapping];

        // Debug
        // Écriture de la donnée programmé dans le fichier BlockRamData.txt
        fprintf(blockRAMstream, "%s", "0x");
        fprintf(blockRAMstream, "%x\n", mapping[cnt_mapping]);

        // Active l'écriture sur le block RAM
        add_temp=LM_BOARDERADD_BASE_RAM_CONTROL;
        add_temp+=ltbase;
        *(vuint)(add_temp)=WE_BR;

        // Désactive l'écriture
        add_temp=LM_BOARDERADD_BASE_RAM_CONTROL;
        add_temp+=ltbase;
        *(vuint)(add_temp)=RESET_BR;
    }
    }
    }
}

// Fermeture du fichier BlockRamData.txt
    fclose(blockRAMstream);

// Écriture des frontières et des assignations ( version debug du
                                                    quantificateur)
/*
 for ( i =0; i< NBRE_BOARDER; i++) {

    //printf( "%d",boarder[i]);
    if( line == 0 ){
```

```
    //hword_write(LM_BOARDERADD_BASE+i<<2,boarder[i]);
    add=ltbase + LM_BOARDERADD_BASE;
    add+=i<<2;
    *(vuint)(add)= boarder[i];

    }else {
    //boarder[i]=hword_read(LM_BOARDERADD_BASE+i<<2);


    add=ltbase + LM_BOARDERADD_BASE;
    add+=i<<2;
    boarder[i]=*(vuint)(add);
    // boarder[i]=*(vuint)(ltbase + LM_BOARDERADD_BASE+i<<2);

    }
  }


     printf ("Ecrit registre de mapping des regions quantification \n");printf (
                "\n");
      for ( i =0; i< NBRE_MAPPING; i++) {

       if( lire == 0 ){
        //hword_write(LM_MAPPINGADD_BASE+i<<2,mapping[i]);
        add=ltbase +LM_MAPPINGADD_BASE;
        add+=i<<2;
        *(vuint)(add)=mapping[i];
//          word_write(add,0xFFFFFFFF);
       } else {

         add=ltbase +LM_MAPPINGADD_BASE;
```

```
            add+=i<<2;
              mapping[i]=*(vuint)(add);

              }
          }

*/


// Écriture ou lecture des coefficients
printf ("Ecriture des coefficient \n");printf ("\n");
    for ( i =0; i< NBRE_ETAGE; i++) {
  // Écriture des coefficient
       if( lire == 0 ){
     add=ltbase +LM_COEFF;
     add+=i<<2;
     *(vuint)(add)=coeff[i];
   }
        else // Lecture des coefficients
     {
     add=ltbase +LM_COEFF;
     add+=i<<2;
     coeff[i]=*(vuint)(add);
     }
  }

  printf ("Ecriture des registres pour les mux de selection ( bits ) \n");printf
          ("\n");
    for ( i =0; i< NBRE_ETAGE; i++) {
  for ( j =0; j< NBRE_MUX; j++) {
```

```
  // Ecriture des quantificateurs des différentes itérations
  if( lire == 0 ){
  add=(i*NBRE_MUX);
  add+=j;
  add=add<<2;
  add+=ltbase;
  add+=LM_MUX;
  *(vuint)(add)=mux[i][j];
  }
  // Lecture des quantificateurs des différentes itérations
  else {
  add=(i*NBRE_MUX);
  add+=j;
  add=add<<2;
  add+=ltbase;
  add+=LM_MUX;
  mux[i][j] = MASK_MUX & ( *(vuint)(add));

  }
}
}

// ecrire la latence utiliser pour les differents comparaison
printf ("Ecriture des latences \n");printf ("\n");
for ( i =0; i< NBRE_ETAGE; i++) {

// Écriture
    if (lire ==0) {

  add=ltbase +LM_LATENCE_COUNT;
  add+=i<<3;
```

```
  *(vuint)(add)=latence_count[i];

  // Partie Haute de latence count =0
     add=ltbase +LM_LATENCE_COUNT_HIGH;
     add+=i<<3;
     *(vuint)(add)=0x0;
    }
// Lecture
else {
  add=ltbase +LM_LATENCE_COUNT;
  add+=i<<3;
  latence_count[i]=*(vuint)(add);
    }
}

// ecriture de ebno

printf ("Ecriture de ebno \n");printf ("\n");
if (lire ==0) {  /// pas oublier latence 6
//byte_write(LM_EBNO, 255);
add=ltbase +LM_EBNO;
*(vuint)(add)= ebnoInt<< NBRE_BIT_EBNO_FRACT |ebnoFract;
//byte_write(LM_EBNO, ebnoInt<< NBRE_BIT_EBNO_FRACT |ebnoFract);
} else {

    add=ltbase +LM_EBNO;
    temp=*(vuint)(add);

    ebnoFract=temp & MASK_FRACT;
    ebnoInt=temp-ebnoFract;
}
```

```
// ecriture des mots d'initialisation  pour chaque lfsr qui represente latence
printf ("Ecriture des seeds pour latence chaques etages \n");printf ("\n");
     for ( i =0; i< NBRE_ETAGE; i++) {

    // Écriture
      if (lire ==0) {
       add=ltbase +LM_SEED_LATENCE_LFSR;
       add+=i<<2;    //3;
       *(vuint)(add)=seed_latence_lfsr[i];


     // Partie haute
       add=ltbase +LM_SEED_LATENCE_LFSR_HIGH;
       add+=i<<2;
       templong = hi64(seed_latence_lfsr[i]);
       temphaut = templong & 0xFF;

       *(vuint)(add)= temphaut;

       }
      else {
        add=ltbase +LM_SEED_LATENCE_LFSR;
        add+=i<<2;
    seed_latence_lfsr[i]=*(vuint)(add);

        // Partie haute
        add=ltbase +LM_SEED_LATENCE_LFSR;
        add+=i<<2;
        temphaut=*(vuint)(add);
        temphaut = temphaut << 31;
```

```
      seed_latence_lfsr[i]=seed_latence_lfsr[i]+ temphaut;

       }
}



// ecriture lfsr de la source binaire
printf ("Ecriture du seed pour source binaire \n");printf ("\n");
if (lire ==0) {
  add=ltbase +LM_SEED;
     *(vuint)(add)=seed;

     // Partie haute
     add=ltbase +LM_SEED_HIGH;
     temphaut = hi64(seed);
     temphaut = temphaut & 0xFF;
     *(vuint)(add)=temphaut;



}
else {
    add=ltbase +LM_SEED;
    seed=*(vuint)(add);

    // Partie haute
    add=ltbase +LM_SEED_HIGH;
    temphaut=*(vuint)(add);
    temphaut= temphaut << 31;
    seed+=temphaut;
}
```

```
// Écriture du nombre de bits à simuler
  //Écriture
if( lire == 0 ) {
  // L'ecriture se fait en 2 cycles
   printf ("Ecriture registre du nombre de cycles de simulation \n");printf (
           "\n");

// Premier 32 bits
  add=ltbase +LM_DECCOUNTERADD;
  *(vuint)(add)=cyclessim ;

  // bit supérieur
  add=ltbase +LM_DECCOUNTERADD_HIGH;
  temphaut = hi64(cyclessim);
  *(vuint)(add)= temphaut;


}
else {
// Lit les 32 premiers bits

add=ltbase +LM_DECCOUNTERADD;
tempbas=*(vuint)(add);

// Lit les bit supérieurs
add=ltbase +LM_DECCOUNTERADD_HIGH;
temphaut =*(vuint)(add);
temphaut = temphaut << 31;

cyclessim=tempbas+temphaut;
```

```
}

printf ("\n");

// Barre le fichier de registres
*(vuint)(ltbase + LT_LOCK)=LOCK;


}


//////////////////////////////////////////////////////////////////////////////////
/////
//
// Fonction servant à transférer les variables globales vers le fichier de
registre APB
// et vise-versa
//
//////////////////////////////////////////////////////////////////////////////////
/////



void DemarreSim(int ltbase) {


  int add;

  add= ltbase + LM_DECCONTROLADD;

// Active les options et le signal LOAD ET START pour charger les mots
```

```
d'initialisation des différents LFSR
*(vuint) add =  (LOAD_SIM |START_SIM |SEL_ERROR |DATA_LFSR |LATENCE_LFSR |
   DATA_NOISE |PARITY_NOISE |SEL_QUANT ) ;

// Démarre la simulation en désactivant le signal LOAD
*(vuint) add =  (  START_SIM |SEL_ERROR |DATA_LFSR |LATENCE_LFSR  |
   DATA_NOISE |PARITY_NOISE |SEL_QUANT ) ;

}



////////////////////////////////////////////////////////////////////////////////
/////
// Fonction qui lit les resultats contenu dans le fichier de registre APB du
nombre
// du nombre de bits en erronés pour chaque itération
//
//
////////////////////////////////////////////////////////////////////////////////
/////



void LitResult(int ltbase) {

int tempbas=0,temphaut=0;
long long tempresult=0;

int add;
int i;
```

```
for ( i =0; i< NBRE_ETAGE; i++) {

// Lecture de la partie base du resultat ( Plage memoire 0x1840)
add=ltbase+LM_DECERRORADD;
add+=(i<<2);
tempbas= *(vuint)(add);


//Lecture de la partie basse
add=ltbase+LM_DECERRORADD_HIGH;
add+=(i<<2);
temphaut= *(vuint)(add);

tempresult = tempbas & MASK_ERROR;

// écrit le résultat dans la variable globale result[i]
result[i]=tempresult;


}
}

////////////////////////////////////////////////////////////////////////////////
/////
// Fonction qui vérifie si les frontières sont toujours croissante
//
////////////////////////////////////////////////////////////////////////////////
/////
```

```
int verifiefrontiere(void) {

int i=0,erreur=0;
// 128 premieres frontieres boarder[0]>boarder[1] ... boarder[126]>boarder[127]
for ( i=0; i< NBRE_BOARDER/2-1; i++) {
  if (boarder[i] > boarder[i+1] )
        erreur=1;
     }
     // Region zero ( poussin écrasé )

     if ( boarder[NBRE_BOARDER/2-1] < boarder[NBRE_BOARDER/2] )
      erreur=1;

     // 128 dernieres frontieres boarder[128]>boarder[129] ... boarder[254]>
     boarder[255]
     for ( i=NBRE_BOARDER/2+1; i< NBRE_BOARDER-1; i++) {
  if (boarder[i] > boarder[i+1] )
        erreur=1;
     }
   return erreur;
}
```

```
//////////////////////////////////////////////////////////////////////////////
/////
// Génère un fichier de données avec le SNR et le taux d'erreurs pour
// chaque itération commençant avec la première itération
// fichier : result.dat
//
// SNR    BER1 BER2 BER3 BER4 BER5 BER6 BER7 BER8
```

```
//
////////////////////////////////////////////////////////////////////////////////
/////

void ecritResult(void)
{

  int i=0;
  FILE * stream;
  // Est utilsé pour calcul le SNR courant
  double temp;
  char *  fileName="C :\\ghislain\\serveur\\result.dat";


  // Verification de l'ouverture du fichier et
  // écrit crée le fichier si c'est le premier SNR caractérisé
  if ( iteration == (cyclestart+1) )  {
  if ((stream = fopen (fileName, "w")) == NULL) {
  fprintf (stderr, "Impossible d'ouvrir le fichier \"%s\"  en lecture lock
           plateforme \n", fileName);
  exit (EXIT_FAILURE);
  }

  }else {
  if ((stream = fopen (fileName, "a")) == NULL) {
  fprintf (stderr, "Impossible d'ouvrir le fichier \"%s\"  en lecture lock
           plateforme 2 \n", fileName);
  exit (EXIT_FAILURE);
  }
```

```
  }

//    Envoie les resultat pour tracer avec gnuplot

  temp=2+0.5*iteration; // SNR courant
  // Écrit le SNR en premier
  fprintf(stream, "%-8.4f", temp);
  fprintf(stream, "%s"," ");

  // Écrit le taux d'erreurs (BER) pour chaque itération à tour de role
  for ( i=0; i<NBRE_ETAGE;i++){
  temp= (double)result[i]/ ( cyclessim- ((LATENCE+1) * (NBRE_ETAGE+2)) );
  fprintf(stream, "%-60.30f", temp);
  fprintf(stream, "%s"," ");

  }
  fprintf(stream, "%s", "\n");
fclose(stream);

}


////////////////////////////////////////////////////////////////////////////////
/////
// Fonction qui calcule les différents mots d'initialisation pour les différents
// LFSR utilisé dans le design
//
//
////////////////////////////////////////////////////////////////////////////////
/////
```

```
void calculseedlfsr ( void ) {

  /* Polynome 32 bits X32+X31+X30+X10+X0 avec des xnor */
  /* Polynome 40 bits X40+X21+X19+X2+X0 avec des xnor */
  // Notes bit le plus signification dans un int represente l'entree d'un LFSR


  long long tempseed=seed;
  long long mask_bit0 =  0x0000008000000000;  // sert au X0
  long long mask_bit2 =  0x0000002000000000;  // sert au X2
  long long mask_bit19 = 0x0000000000100000;  // sert au X19
  long long mask_bit21 = 0x0000000000040000;  // sert au X21

  int  i=0,j=0;
  long long bit0,bit2,bit19,bit21;
  int max =0;

  printf( "%s","SEED demandee");printf("\n");
  printf( "%llx",seed);printf("\n");

  for ( j =0; j< NBRE_ETAGE; j++) {
  temp_seed_lfsr[j]=seed;
  }
  for ( j =0; j<= NBRE_ETAGE; j++) { //allo
  if( j ==0){
  tempseed=seed;
  }
    else {
```

```
tempseed= temp_seed_lfsr[j-1];

}

// configure la latence pour la première itération
if ( j== NBRE_ETAGE) {

max = LATENCE+DATA_PATH_LATENCE; // car sortie pris decision avec registre de
      l'etage suivant

// Latence pour les autres itérations
} else {
max = LATENCE;
}

for (i = 0; i < max; i++) {
// bit 0 equation
bit0  = mask_bit0 &tempseed;
bit0  = bit0 >>39;
bit0  = bit0 & 0x01;

bit2 = mask_bit2 & tempseed;
bit2 = bit2 >>37;

bit19 = mask_bit19 & tempseed;
bit19 = bit19 >>20;

bit21 = mask_bit21 & tempseed;
bit21 = bit21 >>18;

bit0 = bit0^bit2; // xor
```

```
  bit0 = bit0^bit19; // xor
  bit0 = bit0^bit21; // xor

  // Calcul du resultat
  bit0 = ~bit0; // inversion bit xnor
  bit0 = 0x00000001 & bit0;

  tempseed = tempseed<<1;
  tempseed = tempseed |bit0;


}

  temp_seed_lfsr[j]=tempseed;
  if ( j==NBRE_ETAGE)
  seed=tempseed;
  }


  // Changement d'ordre des etages
  for ( j =0; j< NBRE_ETAGE; j++) {
  seed_latence_lfsr[j]=temp_seed_lfsr[NBRE_ETAGE-j-1];
  }


  for ( j =0; j< NBRE_ETAGE; j++) {
  printf( "%s","SEED latence ");printf("%d",j);printf("%s"," ");

  printf( "%llx",seed_latence_lfsr[j]);printf("\n");

  }
```

```
  // mot d'initialisation pour la source binaire
  printf( "%s","SEED actuellement utilisee : ");
  printf( "%llx",seed);printf("\n");
}
```

A.2 Fichier Logic.h

```
/*
 * ------------------------------------------------------------
 * Version
 *
 * Nom de fichier                : logic.h
 * Révision du fichier           : Révision : 1.0
 *
 * ------------------------------------
 *
 */



// Section pour le decodeur
// Par defaut

// Nombre de frontieres ( fin et debut ) d'un pas de quantification du pseudo-
                                ADC
#Define NBRE_MAPPING 255

// Nombre de pas de quantification maximale
#define NBRE_BOARDER 254

// Nombre d'iterations pour implémenter dans l'ITD
#define NBRE_ETAGE 8

// Nombre de quantificateur par itération
#define NBRE_MUX 4

// Latence précédent l'ITD
#define DATA_PATH_LATENCE 7
```

```
// Latence par itération en fonction de J
#define LATENCE 1836 // J+1
// J 4 = 6, 7
// J 5 = 95, 96
// J 6 = 100, Latence =101
// J 7 = 228, 229
// J 8 = 459, 460
// J 9 = 912, 913
// J10 = 1835, 1836


// Adressage des registres du FPGA

// adresse du registre de status du decodeur
#define LM_DECSTATUSADD     0x00000024

// adresse du nombre de cycles de simulation demandée
#define LM_DECCOUNTERADD    0x00000030

// adresse des registres de controls
#define LM_DECCONTROLADD    0x00000028

// adresse de base du nombre d'erreurs totales après simulation
#define LM_DECERRORADD      0x00001840

// defini l'adresse de la deuxieme interruption
#define LM_INT2             0x0000002C

// adresse des coefficients pour chacun des étages
#define LM_COEFF            0x00000080
```

```
// sélection des bits voulus pour chacunes des mux par itérations
// Ordre d'adressage des mux pour chaque itérations
// 100 yu de la premiere itération
// 104 yp de la premiere itération
// 108 lambda qui va prochain itération
// 10C lambda rétroaction

#define LM_MUX              0x00000100

// Adresse des mots d'initialisation pour les differents LFSR
#define LM_SEED_LATENCE_LFSR 0x00000180

// Latence que doit respecter les compteurs d'erreurs avant
// de démarrer a compter
#define LM_LATENCE_COUNT     0x000001C0

// Adresse du mot d'initialisation pour la source binaire
#define LM_SEED              0x00000200

// Registres nécessaires a la programmation du pseudo-ADC

#define LM_BOARDERADD_BASE_RAM_ADRESS   0x0000103C
#define LM_BOARDERADD_BASE_RAM_DATA     0x00001040

#define LM_BOARDERADD_BASE_RAM_CONTROL  0x00001044

// Adresse de programmation du SNR voulu pour le module
// AWGN
#define LM_EBNO              0x00001020
```

```
// Registre de debug
#define LM_BYTE                 0x00001024

// Registre de debug
#define LM_PA_SEE               0x00001028

// Adresse pour la valeur de 16 bits pour le 0 binaire
#define LM_ZERO_BIT_VALUE       0x00001030

// Adresse pour la valeur de 16 bits pour le 1 binaire
#define LM_ONE_BIT_VALUE        0x00001038

// Bit pour la réadaptation dynamique
#define LM_REG_DYN_CONF         0x00001040


// Adresse d'extension pour les registres de 40 bits
// Contient les 8 bits supérieurs des différents mots
// A(39 downto 32)
#define LM_DECCOUNTERADD_HIGH   0x00004030

#define LM_SEED_LATENCE_LFSR_HIGH 0x0004180

#define LM_DECERRORADD_HIGH       0x00005840

#define LM_LATENCE_COUNT_HIGH     0x000041C0

#define LM_SEED_HIGH              0x00004200
```

```
// Section Masque pour ecrire et lire les donnees des registre
// masque de 4 bits pour les coefficients
#define MASK_COEFF 0x0F

// masque de 8 bits
#define MASK_MUX   0xFF  // seulement 8 bits

// masque de 40 bits
#define MASK_CYCLE 0xFFFFFFFF


// 3 bits pour 8 itérations
#define MASK_ETAGE 0x07

// 40 bits max pour les erreurs
#define MASK_ERROR 0xFFFFFFFF

#define CYCLE_FIRST 0x200000

#define MASK_FRACT 0xF


// Section du registre de controle

// bit de chargement des différents mots
#define LOAD_SIM  0x1
// bit de démarrage de la simulaton
#define START_SIM 0x2
// bit de clear n'est plus utilisé
#define CLEAR_SIM 0x4
```

```
// Sélectionne la source binaire ou une source de zéro
#define DATA_LFSR 0x8

// Sélection d'addition de bruit ou non pour l'information
#define DATA_NOISE 0x10

// SÉlection d'addition de bruit ou non pour la parité
#define PARITY_NOISE 0x20

// Sélection entre registre a décalage ou du générateur de séquence
// retardés (debug)
#define LATENCE_LFSR 0x40

// Sélection d'un pseudo-ADC version (debug)
#define SEL_QUANT    0x80

// Activation des registres de compteurs d'erreurs APB (debug)
#define SEL_ERROR    0x100

// Section des shift pour reconstituer les données

// Bit control block ram
// bit d'activation pour écrire les block RAM
#define WE_BR 0x01

// bit de reset des block RAM.
#define RESET_BR 0x00

// Définition du nombre de bit pour la partie fractionnaire
// qui spécifie SNR
#define NBRE_BIT_EBNO_FRACT 4
```

```
// Definition des fonctions


// Prototypes de fonction pour le décodeur
// Lecture des parametres dans le fichier config.txt
extern void loadParameter(void);


extern void ecrire_lireReg (int lire,int);

// Démarrage de la simulation
extern void DemarreSim(int);

// Lecture du fichier de registre pour les résultats dans le fichier
// de registres APB
extern void LitResult(int);

// Affiche a la console de debug les valeurs de registres APB
extern void afficheReg( void );

// Verification de la validite des frontieres du pseudo-ADC
extern int verifiefrontiere(void);

// Écriture des résultats dans le fichier result.dat
extern void ecritResult(void);

// Fonction d'initialisation des frontieres et des mappings pour
// le quantificateur en mode debug
extern void init_frontiere( int ltbase );
```

```
// Fonction qui calcul les mots d'initialisation de la source binaire
// et du générateur de séquences retardées
extern void calculseedlfsr ( void ) ;
```

A.3 Serveur TCL

```
#!/usr/bin/tclsh
#
#
# Programme TCL inspiré de
##########################################
# Moogycode[TM] Mike Tuxford 2001
#     tuxford@earthlink.net moogy@unstable.org
#         http://moogy.unstable.org:8080/
#             irc.fdfnet.net   #Linux
#
# Ce serveur écoute le port 9951 afin de recevoir les parametres du logiciel
d'optimisation
# AWLDT

# Les parametres sont envoyes dans le fornat suivant :

# typeparamtre valeur1 valeur2

# Les types de parametres sont :
# nbrecycle : nombre de bits transmis demandé
# coeff       : coefficient de pondération
# boarder     : ajustement d'une frontiere
# mapping     : valeur d'un pas de quantification
# ebno        : rapport signal a bruit de l'optimisation
# fin         : fin de l'envoie des parametres et lancement
#                 de la simulation
# exit : fin de l'optimisation


set clientHost "127.0.0.1"
set clientPort "9951"
```

```
# Écrit fichier texte pour configuration du decodeur

# fichier de configuration des parametres final
set configDecodeurFile "configdec.txt"

# fichier de données par défaut
set templateconfigDec  "configdecTemplate.txt"

# fichier de synchronisation avec le processeur ARM
set interfacelckname "interface.lck"

# variables globales nécessaires pour configuration
# parametres qui peut etre changer dans le fichier

set nbrecoeff 8
set nbrecycle 50000
set nbreetage 8
set nbremux 4
set midnbreboarder 128
set nbreboarder 256

set midnbremapping 127
set nbremapping 255

set valeurcoeffbase 1
set valeurmuxbase 255

set offsetFrontiere 256

set startsim 0
```

```
# 10 dB par défaut
set ebnoInt 10
set ebnoFract 0

set datawidth 8

# largueur du canal de communication
set datawidthcanal 16

# nombre de bit d'entier dans le canal de communication
set datacanalint 5
# initialisation des differents variables

##########################################
# Mets des valeurs par défauts des différents parametres
#
##########################################

# ecrit les coefficients
set cnt_i 0

for { set cnt_i 0 } { $cnt_i < $nbreetage } { incr cnt_i 1 }  {
   set  coeff($cnt_i) $valeurcoeffbase
}

# ecrit les valeurs des multiplexeurs
for {set cnt_i 0 } { $cnt_i <  $nbreetage  }  { incr cnt_i 1} {
  for { set cnt_j 0 } { $cnt_j <  $nbremux } { incr cnt_j 1 } {
    set index [expr $cnt_j+ (4 * $cnt_i )]
    set mux($index) $valeurmuxbase
```

```
  }
}

# ecrit les frontieres negatives 0 a 127
for {set cnt_i 0} { $cnt_i <  $midnbreboarder   }  { incr cnt_i 1} {
  set  boarder($cnt_i) [ expr (($cnt_i + 129) * $offsetFrontiere )]
}

# ecrit les frontieres positive  128 a 255
for {set cnt_i 0} { $cnt_i <  $midnbreboarder   }  { incr cnt_i 1} {
  set index [expr $cnt_i + $midnbreboarder]
#   set  boarder($index) [ expr (( 1+ $cnt_i) * $offsetFrontiere) ]
#      set temp [ expr ( -1 * ((pow(2,$datawidth)-1)-$cnt_i)*pow(2,$datawidthcanal-
                          $datawidth-$datacanalint-1)+1)]
#      set temp2 [ expr ($temp*100 /36 *  (pow(2,$datawidth)-1) / pow(2,$datawidth)
                          ) ]
    set boarder($index) [expr int( ( -1 * ((pow(2,$datawidth)-1)-$cnt_i)*pow(2,
                $datawidthcanal-$datawidth-$datacanalint-1)+1) *100 /36 *  (pow(
                2,$datawidth)-1) / pow(2,$datawidth) ) ]
#set boarder($index) [ expr int ( $temp2) ]
}


#ecrit le mapping des regions negatives 0 a 126
for {set cnt_i 0} { $cnt_i <  $midnbremapping  } { incr cnt_i 1 } {
  set mapping($cnt_i)  [ expr ($cnt_i +129) ]
}

set mapping(127)  0

#ecrit le mapping des regions positives 128 a 255
```

```
for {set cnt_i 0} { $cnt_i <  $midnbremapping  } { incr cnt_i 1 } {
  set index [expr $cnt_i + $midnbremapping +1]
  set mapping($index)  [ expr $cnt_i + 1 ]
}


##########################################
# Processus de traitement de la commande envoyée par le
# logiciel d'optimisation
##########################################


proc changevariable { data  } {

global coeff
global mux
global boarder
global mapping
global nbrecycle
global startsim
global ebnoInt
global ebnoFract

set line $data

set firstlist [lindex $line 0]

switch -exact -- $firstlist {
    nbrecycle {
    set nbrecycle [lindex $line 1]
    set startsim 0
```

```
}
coeff {
set index [lindex $line 1]
set coeff($index) [lindex $line 2]
set startsim 0
}

mux {
set temp [expr 4 * [lindex $line 1]]
set temp2 [lindex $line 2]
set index [expr $temp2 + $temp  ]
    set mux($index) [lindex $line 3]
set startsim 0
}

boarder {
set index [lindex $line 1]
set boarder($index) [lindex $line 2]
set startsim 0
}
mapping {
set index [lindex $line 1]
set mapping($index) [lindex $line 2]
set startsim 0
}

ebno {
set ebnoInt [lindex $line 1]
set ebnoFract [lindex $line 2]
set startsim 0
}
```

```
    fin {
    # ecrit de le fichier de partir simulation
    set startsim 1
    }
    exit {
    exit
    }
  }
}
```

```
##########################################
# Processus qui écrit le fichier de configuration des parametres
# config.txt qui est lu par le processeur ARM
##########################################
```

```
proc ecrireFichier { } {
  global nbrecycle
  global coeff
  global nbreetage
  global nbremux
  global boarder
  global mapping
  global nbremapping
  global nbreboarder
  global configDecodeurFile
  global templateconfigDec
```

```
global mux
global startsim
global ebnoInt
global ebnoFract

set fileid [open $configDecodeurFile w]
set fileid2 [open $templateconfigDec w]

# ecrit nombre de cycle
puts $fileid $nbrecycle
puts $fileid2 "nbrecycle"

#ecrit le ebno partie entiere
puts $fileid $ebnoInt
puts $fileid2 "ebnoInt"

#ecrit le ebno partie fractionnaire
puts $fileid $ebnoFract
puts $fileid2 "ebnoFract"

# ecrit les coefficients
for {set cnt_i 0 } { $cnt_i < $nbreetage} {incr cnt_i 1} {
  puts $fileid $coeff($cnt_i)

  set  temp "coeff"
  append temp $cnt_i
  puts $fileid2 $temp
}
```

```
# écrit les valeurs des multiplexeurs
for {set cnt_i 0} { $cnt_i <  $nbreetage  }  {incr cnt_i 1} {

  for { set cnt_j 0} { $cnt_j <  $nbremux } {incr cnt_j 1} {
  set index  [expr $cnt_j+ (4 * $cnt_i )]
  puts $fileid $mux($index)
  set temp "mux("
  append temp $cnt_i
  append temp ")("
  append temp $cnt_j
  append temp ")"
  puts $fileid2 $temp
  }
}

# écrit les frontieres pour le quantificateur
for {set cnt_i 0} { $cnt_i <  $nbreboarder  } {incr cnt_i 1 } {

   puts $fileid $boarder($cnt_i)
 set temp "boarder("
 append temp $cnt_i
 append temp ")"
 puts $fileid2 $temp
}

#écrit les frontieres pour le mapping des regions

for {set cnt_i 0} { $cnt_i <  $nbremapping  } {incr cnt_i 1 } {
```

```
      puts $fileid $mapping($cnt_i)
    set temp "mapping("
    append temp $cnt_i
    append temp ")"
    puts $fileid2 $temp

  }
  close $fileid
  close $fileid2
}
```

```
##########################################
# Decloration handler pour socket serveur
# Execute lorsque l'on recoit de l'info sur socket serveur
##########################################

# the server's simple event handler that is invoked when it
# detects an event on the listening socket
proc serverEventHandler {fd address port} {
  # let the client know it's connected by sending back a message
  puts $fd "Simple Server ready"; flush $fd
  # set an event handler for the client's socket
  fileevent $fd readable "echoBack $fd"
  return
}
```

```
# the socket handler which return result of the simulation
```

```
proc netEventHandler {} {
    global fd
    # if an eof is detected then the server has disconnected
    if {[eof $fd] != 0} {
        catch {close $fd}
        puts stdout "Server has disconnected"; flush stdout
        exit
    } else {
        # get the data that the server sent
        catch {gets $fd data; flush $fd}
        # puts the data to the terminal
        puts stdout "Recu : $data"; flush stdout
    }
    return
}
#############################
# Permet de gerer les messages recues du client modifiant les
# parametres de la simulation
#############################

# The event handler that echos what it receives back to the client
proc echoBack {outfd} {
    global svrfd
    global startsim clientHost clientPort
   set filelock [open interface.lck w]
    if {[eof $outfd]} {

   ######################
   # Gestion de fin de connection du client
   # devrait enlever fin de connection
```

```
##########################
    # if the event detected is an End Of File puts a message and exit
    puts stdout "Client disconnected, you don't need me anymore. :)"; flush
    stdout
    # Tcl will clean up open sockets upon exiting but the close is for example
    catch {close $svrfd}
    exit

 } else {

    # get the data from the client socket
    set data [gets $outfd] ; flush $outfd

# change variable generale voulue dans la configuration serveur
puts stdout "Recu : $data"
 changevariable $data

 # Ecrit le fichier
 ecrireFichier

 # demarrage de la simulation
 # si l'usager veut lancer la simulation
if { $startsim ==1 } {

  puts $filelock "start"
  close $filelock
  # attend 100 ms pour que la plateforme ait le temps d'ecrire le fichier lc
  fichier lck
     after 100

   set filelock [open interface.lck w]
```

```
    puts $filelock "wait"


plateforme.lck )
   after 10
    set line "wait"
   #boucle tant que la plateforme n'a pas terminer
       while { $line =="wait"} {
     set fileid2 [open "plateforme.lck"r]
     set line [gets $fileid2]
     close $fileid2
    # puts stdout "attend plateforme"; flush stdout
        #lit le fichier de resultat
    # Envoie les resultats ecrit par la plateforme dans le fichier result.dat
       # Les resultats des 8 etages sont envoyer
    # creer socket vers le client pour envoyer resultat


    if {[catch {socket $clientHost $clientPort} fd]} {
       puts stdout "Failed to connect to server at $clientHost $clientPort"
       # note : flushing stdout is not really necessary but just my SOP.
       # In fact, 'puts "whatever"' is acceptable to Tcl and if no FD
       # is given it will assume stdout
       flush stdout
     #   exit
    } else {
       # put socket in non-blocking mode
       fconfigure $fd -blocking 0
       fconfigure $fd -buffering full
       # assigning socket event handler is last!!!
       # fileevent $fd readable netEventHandler
       # optional, uncomment next line to see the socket's configuration
       #puts stdout "cfg : [fconfigure $fd]"; flush stdout
```

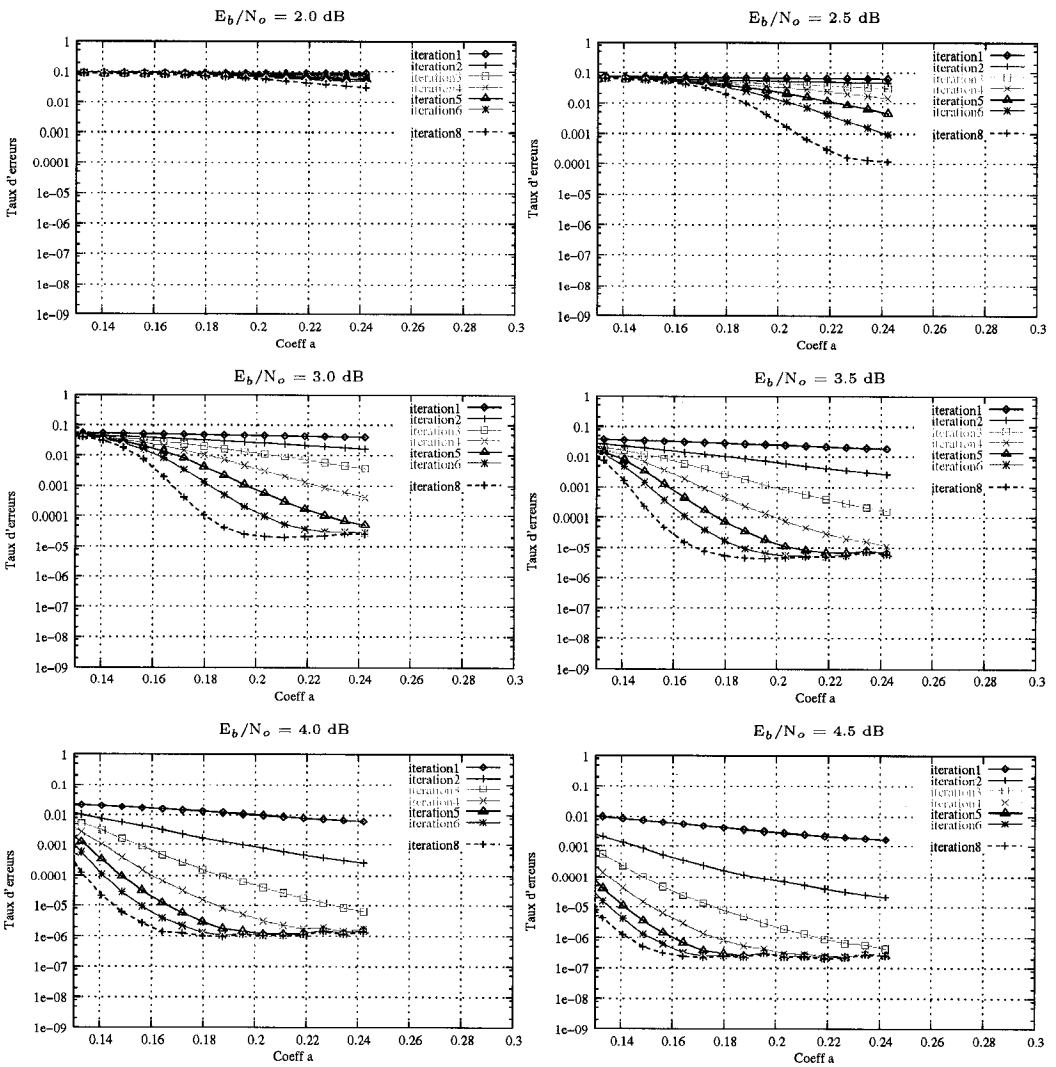
```
    }

    set fileresult [open result.dat r ]
        while { ![eof $fileresult] } {
     gets $fileresult line
     after 100
     puts  $fd "$line\0"
     flush $fd
      puts stdout "Sending : $line"
    }
        close $fd
        close $fileresult

   }
   puts stdout "lis fin simulation"; flush stdout
  } else {
    puts $filelock "wait"
  }
  # puts stdout "pas de mot fin encore"; flush stdout
  close $filelock
   }
   return
}

# a very simple error handler
proc bgerror {args} {
  global svrfd
  catch {close $svrfd}
  puts stdout "bgerror invoked, args were : $args"
  exit
}
```

```
#set clientHost "132.207.108.232"


##########################################
# open the socket
# port 9950 is arbitrary and can be changed, but client must know
# what port we are listening on. High port numbers are always advised
# to avoid conflict with important assigned services ports.

set serverPort 9950
set svrfd [socket -server serverEventHandler $serverPort]

# put server in non-blocking mode
fconfigure $svrfd -blocking 0

# send server startup notice to stdout
puts stdout "Simple Server running...\nListening on port $serverPort"; flush
stdout

# optional, uncomment if you wish to see the socket info
puts stdout "socket configuration :\n[fconfigure $svrfd]"; flush stdout


# DECLARATION POUR CLIENT SOCKET

#    # put socket in non-blocking mode
#    fconfigure $fd -blocking 0
#    # assigning socket event handler is last!!!
#    fileevent $fd readable netEventHandler
```

```
# DECLARATION POUR CLIENT SOCKET

# enter the tcl event loop
vwait __forever__
# an optional might be : vwait tillGatesPlaysFair
```

A.4 Résultats de simulation pour un coefficient de pondération variant entre 0,13 et 0,25

Résultats obtenus avec la plateforme de caractérisation matérielle pour différents rapports signal à bruit avec une grande résoluton autour de l'optimal avec un coefficient de pondération uniforme.



Annexe B

Complexité matérielle et implémentation de l'ITD

B.1 Code VHDL des principales fonctions pour configurer l'ITD

```
-------------------------------------------------------------------
-- TITLE :                    function
-- DESCRIPTION : this package contain all the entity declaration for the whole
design
--
--
-- FILE :          function.vhd
-------------------------------------------------------------------
-- CREATION
-- DATE            AUTHOR                PROJECT      REVISION
-- 2003/01/30      Ghislain Provost      XXXXXX       v1.0
-------------------------------------------------------------------
-- MODIFICATION HISTORY
-- DATE            AUTHOR                PROJECT      REVISION
-- COMMENTS
-------------------------------------------------------------------


library ieee;
library STD;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.csoc_types.all;
use std.TextIO.all;
use ieee.std_logic_textio.all;

package functiondeco is

  function do_calculate_input_lambda (J  : integer; CODE  : t_tap_info) return
                                      t_lambda;
```

```
  function calculate_tap_retro ( J        : integer; CODE   : t_tap_info) return
                                    t_tap_info;
  function do_calculate_tap_lambda ( J    : integer; INPUT_LAMBDA  : t_lambda)
                                        return t_tap_info;

end package functiondeco;



package body functiondeco is


  function lireDATAWIDTH return integer is
    file file_datawidth    : text open read_mode is "datawidth.dat";
    variable status_file   : boolean;
    variable L             : line;
    variable temp          : integer;
  begin
    readline(file_datawidth, L);
    read(L, temp, status_file);
    return(temp);
  end lireDATAWIDTH;
-------------------------------------
-- This function return the tap information for lambda register
-- which the first column contain the addmin where the connexion to
-- connected the second one contain which input the input has to be connected
-- the third column indicate the distance from the shift register first
-- element. All the result are sorted in increasing order using the third
-- column
```

```
function do_calculate_input_lambda (J  : integer; CODE  : t_tap_info) return
                                        t_lambda is


  variable temp      : t_lambda ( J*(J-1)-1 downto 0, 2 downto 0) ;  -- Variable
                                    declaration
  variable temp_min  : t_addinput;
  variable temp2     : integer;
begin

  do_jterm  : for cnt_index in 2 to J loop

    do_eachterm  : for cnt_i in cnt_index-1 downto 1 loop

      temp((cnt_index-1)*(cnt_index-2)/2+cnt_i-1, 0) := cnt_index;
      temp((cnt_index-1)*(cnt_index-2)/2+cnt_i-1, 1) := cnt_i-1;
      temp((cnt_index-1)*(cnt_index-2)/2+cnt_i-1, 2) := CODE(J-1)-(CODE(
           cnt_index-1)-CODE(cnt_i-1));


    end loop do_eachterm;
  end loop do_jterm;

  -- faire la recherche et trie des donnees

  do_alljterm  : for cnt_i in 0 to J*(J-1)/2-1 loop
    temp_min(0) := temp(cnt_i, 0);
    temp_min(1) := temp(cnt_i, 1);
    temp_min(2) := temp(cnt_i, 2);
```

```
      do_thesearch  : for cnt_search in J*(J-1)/2-1 downto cnt_i+1 loop
        if temp(cnt_i, 2) > temp(cnt_search, 2) then

          temp_min(0) := temp(cnt_search, 0);
          temp_min(1) := temp(cnt_search, 1);
          temp_min(2) := temp(cnt_search, 2);

          temp(cnt_search, 0) := temp(cnt_i, 0);
          temp(cnt_search, 1) := temp(cnt_i, 1);
          temp(cnt_search, 2) := temp(cnt_i, 2);

          temp(cnt_i, 0) := temp_min(0);
          temp(cnt_i, 1) := temp_min(1);
          temp(cnt_i, 2) := temp_min(2);
        end if;
      end loop do_thesearch;
    end loop do_alljterm;

    return (temp);

  end do_calculate_input_lambda;
-----------------------------------
-- This function return the tap information for the parite
-- register mixed with somes add-min  base on the convolutional code
-- used during the encoding process
-----------------------------------
  function calculate_tap_retro ( J  : integer; CODE  : t_tap_info) return
                                  t_tap_info is

    variable temp  : t_tap_info (J-2 downto 0);
  begin
```

```
    for index in J-1 downto 1 loop
      temp(J-1-index) := CODE(J-1)-CODE(index);
    end loop;

    return (temp);

  end calculate_tap_retro;

-----------------------------------
-- This function only take the third column of the INPUT_LAMBDA vector
-- and paste is into a vector of type t_tap_info
-----------------------------------
  function do_calculate_tap_lambda ( J  : integer; INPUT_LAMBDA  : t_lambda)
                                     return t_tap_info is
    variable temp  : t_tap_info ( J*(J-1)/2-1 downto 0 );
  begin
    for cnt_i in 0 to J*(J-1)/2-1 loop
      temp(cnt_i) := INPUT_LAMBDA(cnt_i, 2);
    end loop;

    return (temp);

  end do_calculate_tap_lambda;
end package body functiondeco;
```

B.2 Code VHDL de l'opérateur saturation

```
--------------------------------------------------------------------
-- TITLE :          saturation
-- DESCRIPTION : This operation truncates the integer par of the lambda signal
-- in the iterative threshold decoder
-- The number output bit (DATA_WIDTH) is not obligated to be equal to DATA_FRACT+
                        1
--
-- FILE :           deco_iteratif.vhd
--------------------------------------------------------------------
-- CREATION
-- DATE             AUTHOR                  PROJECT     REVISION
-- 2003/01/30       Ghislain Provost        XXXXXX      v1.0
--------------------------------------------------------------------
-- MODIFICATION HISTORY
-- DATE             AUTHOR                  PROJECT     REVISION
-- COMMENTS
--------------------------------------------------------------------
```

```
entity saturation is
  generic ( DATA_WIDTH        : natural  := 6;  -- data width input
            DATA_INT          : natural  := 2;  -- integer data width
            DATA_FRACT        : natural  := 4;  -- fractional data width
            DATA_WIDTH_OUT    : natural         -- number of wanted output bits

            );

  port (
    -- input data
```

```
    data_in   : in std_logic_vector ( ( DATA_WIDTH-1 ) downto 0 ) ;

    -- output data
    data_out  : out std_logic_vector ( ( DATA_WIDTH_OUT-1 ) downto 0 )

    ) ;
end entity saturation;


architecture behav of saturation is

  constant zero_int  : std_logic_vector( DATA_INT-1 downto 0) := (others => '0') ;

  constant zero_out       : std_logic_vector( DATA_WIDTH_OUT-1 downto 0)
                                    := (others => '0') ;
  constant zero_out_rest  : std_logic_vector( DATA_WIDTH-(DATA_WIDTH_OUT-1)-
                                   DATA_INT-1 downto 0) := (others =>
                                   '0') ;

  constant all_one_int  : std_logic_vector( DATA_INT-1 downto 0)        := (others
                                  => '1') ;
  constant all_one_out  : std_logic_vector( DATA_WIDTH_OUT-1 downto 0) := (others
                                  => '1') ;

  constant zero_neg_out  : std_logic_vector( DATA_WIDTH_OUT-1 downto 0) := (
                                    others => '0') ;


  constant max_pos  : std_logic_vector( DATA_WIDTH-1 downto 0) := '0'&zero_int(
                                DATA_INT-2 downto 0)&all_one_out(
                                DATA_WIDTH_OUT-2 downto 0)&zero_out_rest;
```

```
  constant max_neg  : std_logic_vector( DATA_WIDTH-1 downto 0) := '1'&all_one_int(
                                        DATA_INT-2 downto 0)&zero_neg_out(
                                        DATA_WIDTH_OUT-3 downto 0)
                                        &'1'&zero_out_rest;  --(all_one_rest);

  constant max_value_pos  : std_logic_vector( DATA_WIDTH_OUT-1 downto 0) :=
                                        '0'&all_one_out(DATA_WIDTH_OUT-2
                                        downto 0);
  constant max_value_neg  : std_logic_vector( DATA_WIDTH_OUT-1 downto 0) :=
                                        '1'&zero_out(DATA_WIDTH_OUT-3
                                        downto 0)&'1';


begin
  data_out <= max_value_pos when signed(max_pos) <= signed(data_in) else (others
              => 'Z');

  data_out <= data_in(DATA_WIDTH-1)&data_in( DATA_WIDTH-DATA_INT-1 downto
              DATA_WIDTH-DATA_INT-(DATA_WIDTH_OUT-1)) when ( ( signed(max_pos) >
              signed (data_in) ) and (signed (max_neg) < signed(data_in)) ) else
              (others => 'Z');

  data_out <= max_value_neg when signed(data_in) <= signed (max_neg) else (
              others => 'Z');

end architecture behav;
```

B.3 Code VHDL du pondérateur

```
----------------------------------------------------------------------
-- TITLE :          ponderator
-- DESCRIPTION : This component allow to multiply the parameter lambda in the
decoder
--
-- FILE :           ponderator.vhd
----------------------------------------------------------------------
-- CREATION
-- DATE             AUTHOR                  PROJECT     REVISION
-- 2004/03/06       Ghislain Provost        XXXXXX      v1.0
----------------------------------------------------------------------
-- MODIFICATION HISTORY
-- DATE             AUTHOR                  PROJECT     REVISION
-- COMMENTS
----------------------------------------------------------------------

library ieee; use ieee.std_logic_1164.all;

library UNISIM;
use UNISIM.VCOMPONENTS.all;



entity ponderator_async_xilinx is
  generic (DATA_WIDTH     : natural;
           DATA_WIDTH_OUT  : natural;
           DATA_WIDTH_COEFF  : natural
           );
```

```
  port (

    in_data              : in  std_logic_vector(DATA_WIDTH-1 downto 0);
    componentdecleff     : in  std_logic_vector(DATA_WIDTH_COEFF-1 downto 0);
    out_data  : out std_logic_vector(DATA_WIDTH_OUT-1 downto 0)
);
end entity ponderator_async_xilinx;



architecture rtl of ponderator_async_xilinx is

  signal A  : std_logic_vector (17 downto 0) := (others => '0');
  signal B  : std_logic_vector (17 downto 0) := (others => '0');
  signal P  : std_logic_vector (35 downto 0) := (others => '0');


  component MULT18X18
    port (P  : out std_logic_vector (35 downto 0);
          A  : in  std_logic_vector (17 downto 0);
          B  : in  std_logic_vector (17 downto 0));
  end component;

--for all : mult use entity unisim.MULT18X18;
begin
-----------------------------------------------------------------------
-- Mapping inputs
-----------------------------------------------------------------------
  gen_mapping_in_complement2  : for i in DATA_WIDTH-1 downto 0 generate
```

```
begin
  A(17-i) <= in_data(DATA_WIDTH-1-i);
end generate gen_mapping_in_complement2;

gen_mapping_in2_complement2  : for i in 17-1-DATA_WIDTH-1 downto 0 generate
begin
  A(i) <= '0';
end generate gen_mapping_in2_complement2;

--        A(17)<='0';

gen_mapping_coeff_complement2  : for i in DATA_WIDTH_COEFF-1 downto 0 generate
begin
  B(17-1-i) <= coeff(DATA_WIDTH_COEFF-1-i);
end generate gen_mapping_coeff_complement2;

gen_mapping_coeff2_complement2  : for i in 17-1-DATA_WIDTH_COEFF-1-1 downto 0
generate
begin
  B(i) <= '0';
end generate gen_mapping_coeff2_complement2;
B(17) <= '0';
-------------------------------------------------------------------------
-- Mapping output
-------------------------------------------------------------------------

gen_mapping_out_complement2  : for i in DATA_WIDTH_OUT-1 downto 0 generate
begin
  out_data(DATA_WIDTH_OUT-1-i) <= P(35-1-i);
end generate gen_mapping_out_complement2;
```

```
  mult   : component MULT18X18
    port map ( P => P,
               A => A,
               B => B );


end architecture rtl;
```

B.4 Codes trouvés permettant un meilleur réajustement de l'ordonnancement

Recherche de code pour J=6		
Dmin	span	Code trouvé
1	110	{0,1,20,24,97,110}
2	116	{0,22,61,82,114,116}
3	110	{0,3,19,26,92,110}
4	111	{0,10,14,25,84,111}
5	120	{0,20,25,36,107,120}
6	107	{0,13,24,34,101,107}
7	112	{0,8,19,36,105,112}
8	116	{0,10,23,37,108,116}
9	113	{0,9,24,34,100,113}
10	121	{0,26,60,83,111,121}
11	123	{0,25,40,52,112,123}
12	119	{0,19,34,59,107,119}
13	121	{0,13,38,56,97,121}
14	124	{0,25,43,66,110,124}
15	126	{0,22,54,80,111,126}
16	125	{0,30,50,72,109,125}
17	128	{0,28,48,70,111,128}
18	127	{0,19,41,59,94,127}
19	129	{0,19,54,77,99,129}
20	131	{0,22,48,75,111,131}
21	134	{0,21,59,83,106,134}
22	137	{0,22,60,85,108,137}
23	139	{0,24,60,89,112,139}
24	146	{0,39,63,88,115,146}
25	151	{0,27,52,83,113,151}

26	156	{0,31,57,89,128,156}
27	160	{0,27,58,86,122,160}
28	165	{0,32,60,89,126,165}
29	169	{0,33,62,97,139,169}
30	174	{0,34,64,100,143,174}
31	180	{0,31,64,98,136,180}
32	184	{0,35,67,100,144,184}
33	190	{0,37,70,104,146,190}
34	195	{0,34,72,107,150,195}
35	200	{0,37,72,110,152,200}
36	205	{0,38,74,113,156,205}
37	209	{0,49,94,134,171,209}
38	213	{0,45,95,136,175,213}
39	220	{0,40,79,122,167,220}
40	225	{0,40,84,125,174,225}
41	230	{0,42,83,128,180,230}
42	235	{0,45,87,131,181,235}
43	240	{0,43,87,134,183,240}
44	245	{0,44,92,137,190,245}
45	250	{0,47,92,140,192,250}
46	255	{0,46,95,143,197,255}
47	260	{0,47,96,146,200,260}
48	266	{0,49,97,148,203,266}
49	270	{0,49,99,152,207,270}
50	275	{0,51,105,155,216,275}
51	282	{0,67,125,177,228,282}
52	286	{0,55,107,160,219,286}

53	289	{0,61,126,180,233,289}
54	296	{0,54,109,166,227,296}
55	300	{0,55,112,170,232,300}
56	305	{0,56,114,173,236,305}
57	309	{0,58,115,175,240,309}
58	315	{0,58,117,179,243,315}
59	320	{0,63,122,182,250,320}
60	325	{0,64,124,185,254,325}
61	330	{0,63,124,188,256,330}
62	336	{0,65,127,190,259,336}
63	340	{0,67,130,194,266,340}
64	346	{0,67,132,196,269,346}
65	350	{0,76,150,216,285,350}
66	355	{0,67,133,203,275,355}
67	360	{0,67,136,206,280,360}
68	366	{0,76,147,215,284,366}
69	370	{0,69,139,212,290,370}
70	375	{0,70,141,215,294,375}
71	381	{0,72,143,217,295,381}
72	386	{0,72,147,220,299,386}
73	390	{0,77,151,224,311,390}
74	395	{0,74,152,227,310,395}
75	400	{0,79,154,230,314,400}
76	405	{0,76,153,233,315,405}
77	410	{0,77,158,236,322,410}
78	415	{0,82,160,239,331,415}
79	421	{0,81,160,245,329,421}

80	426	{0,82,162,248,333,426}
81	430	{0,81,163,250,335,430}
82	436	{0,82,167,250,339,436}
83	440	{0,87,170,254,346,440}
84	446	{0,87,171,256,349,446}
85	451	{0,85,171,259,351,451}
86	456	{0,86,173,262,355,456}
87	459	{0,88,175,265,360,459}
88	465	{0,90,178,269,364,465}
89	471	{0,104,196,285,375,471}
90	475	{0,90,181,275,371,475}
91	480	{0,91,186,278,378,480}
92	486	{0,95,188,280,381,486}
93	490	{0,97,190,284,386,490}
94	495	{0,95,189,287,387,495}
95	500	{0,97,192,290,392,500}
96	505	{0,97,193,293,395,505}
97	510	{0,99,196,296,400,510}
98	515	{0,102,200,299,406,515}
99	521	{0,99,199,301,407,521}
100	525	{0,100,204,305,414,525}

Recherche de code pour J=7		
Dmin	span	Code trouvé
1	265	{0,52,70,100,127,264,265}
2	261	{0,2,11,32,91,196,261}
3	263	{0,12,20,23,76,189,263}
4	259	{0,4,26,51,57,195,259}
5	258	{0,5,16,39,92,222,258}
6	245	{0,7,13,47,83,214,245}
7	241	{0,19,26,63,79,217,241}
8	259	{0,8,22,51,63,205,259}
9	249	{0,9,21,52,81,233,249}
10	249	{0,10,24,35,113,233,249}
11	257	{0,11,42,55,74,229,257}
12	254	{0,18,47,59,79,241,254}
13	265	{0,23,36,58,77,222,265}
14	259	{0,23,37,63,81,235,259}
15	260	{0,22,39,54,104,237,260}
16	266	{0,18,34,67,91,228,266}
17	256	{0,17,38,56,121,210,256}
18	277	{0,19,44,62,92,249,277}
19	275	{0,19,49,69,102,247,275}
20	280	{0,20,41,65,121,229,280}
21	293	{0,21,46,68,98,261,293}
22	269	{0,22,45,75,103,227,269}
23	288	{0,45,97,137,176,265,288}
24	288	{0,35,61,93,117,252,288}
25	287	{0,37,71,96,122,246,287}

26	290	{0,27,59,109,135,256,290}
27	292	{0,47,79,117,144,262,292}
28	286	{0,28,74,107,139,224,286}
29	295	{0,46,82,113,142,258,295}
30	285	{0,32,75,108,138,238,285}
31	292	{0,58,111,158,201,261,292}
32	295	{0,38,72,104,167,213,295}
33	298	{0,45,78,113,159,244,298}
34	293	{0,44,92,135,186,259,293}
35	299	{0,49,84,123,171,227,299}
36	297	{0,51,93,140,194,261,297}
37	301	{0,37,75,116,159,250,301}
38	280	{0,38,78,125,188,229,280}
39	292	{0,48,87,138,181,243,292}
40	302	{0,40,89,134,186,251,302}
41	327	{0,48,99,152,209,286,327}
42	300	{0,42,87,130,184,233,300}
43	309	{0,43,92,140,197,264,309}
44	314	{0,45,95,139,199,247,314}
45	327	{0,45,96,143,199,258,327}
46	323	{0,48,94,143,196,255,323}
47	334	{0,48,100,147,202,273,334}
48	346	{0,52,106,155,203,271,346}
49	344	{0,50,102,151,208,271,344}
50	349	{0,51,104,154,212,276,349}
51	353	{0,53,104,158,217,283,353}
52	360	{0,53,109,161,224,285,360}

53	365	{0,54,111,164,230,289,365}
54	370	{0,63,117,172,229,303,370}
55	379	{0,58,114,169,233,304,379}
56	382	{0,65,121,178,237,313,382}
57	388	{0,58,115,176,239,310,388}
58	395	{0,60,118,179,245,318,395}
59	401	{0,59,120,182,249,323,401}
60	406	{0,60,121,184,251,326,406}
61	412	{0,61,123,187,255,331,412}
62	418	{0,63,125,191,259,335,418}
63	424	{0,63,127,193,263,341,424}
64	430	{0,64,129,196,267,346,430}
65	436	{0,66,131,200,271,350,436}
66	442	{0,66,133,202,275,356,442}
67	448	{0,67,135,205,279,361,448}
68	454	{0,69,137,209,283,365,454}
69	460	{0,70,139,212,287,370,460}
70	466	{0,70,141,214,291,376,466}
71	472	{0,72,143,218,295,380,472}
72	478	{0,73,145,221,299,385,478}
73	484	{0,73,147,223,303,391,484}
74	490	{0,74,149,226,307,396,490}
75	496	{0,75,151,229,311,401,496}
76	502	{0,76,153,232,315,406,502}
77	508	{0,78,155,236,319,410,508}
78	514	{0,78,157,238,323,416,514}
79	520	{0,79,159,241,327,421,520}

80	528	{0,80,164,245,334,425,528}
81	532	{0,82,163,248,335,430,532}
82	538	{0,82,165,250,339,436,538}
83	544	{0,83,167,253,343,441,544}
84	550	{0,84,169,256,347,446,550}
85	556	{0,94,179,265,353,458,556}
86	562	{0,87,173,263,355,455,562}
87	568	{0,87,175,265,359,461,568}
88	574	{0,88,177,268,363,466,574}
89	580	{0,89,179,271,367,471,580}
90	586	{0,91,181,275,371,475,586}
91	592	{0,100,191,283,377,488,592}
92	598	{0,92,185,280,379,486,598}
93	604	{0,93,187,283,383,491,604}
94	610	{0,94,189,286,387,496,610}
95	616	{0,95,191,289,391,501,616}
96	622	{0,96,193,292,395,506,622}
97	628	{0,98,195,296,399,510,628}
98	634	{0,98,197,298,403,516,634}
99	640	{0,108,207,307,409,528,640}
100	647	{0,102,202,305,413,528,647}
101	652	{0,102,203,308,415,530,652}
102	664	{0,102,205,314,419,536,664}

Recherche de code pour J=8		
Dmin	span	Code trouvé
2	561	{0,2,23,32,61,191,415,561}
3	531	{0,14,17,29,95,209,457,531}
4	545	{0,4,13,32,93,207,445,545}
5	533	{0,6,11,30,86,205,456,533}
6	537	{0,11,17,36,88,208,456,537}
7	545	{0,10,18,25,77,192,439,545}
8	555	{0,57,95,137,176,269,547,555}
9	529	{0,9,23,34,101,228,443,529}
10	567	{0,62,72,97,110,212,463,567}
11	555	{0,17,35,46,95,227,404,555}
12	575	{0,12,30,46,79,202,412,575}
13	563	{0,24,43,56,79,229,431,563}
14	557	{0,14,46,61,86,217,469,557}
15	555	{0,59,79,95,110,239,502,555}
16	542	{0,18,34,57,108,212,409,542}
17	560	{0,18,41,74,91,195,467,560}
18	556	{0,57,95,135,179,222,538,556}
19	534	{0,19,50,71,118,209,452,534}
20	612	{0,21,44,64,100,217,465,612}
21	521	{0,24,46,67,105,233,493,521}
22	552	{0,23,53,75,123,214,463,552}
23	551	{0,25,58,90,113,240,442,551}
24	536	{0,30,70,96,139,197,512,536}
25	550	{0,36,62,87,119,230,502,550}
26	553	{0,28,70,96,133,221,489,553}

27	572	{0,29,56,90,128,235,501,572}
28	556	{0,42,70,102,131,286,447,556}
29	557	{0,31,60,94,129,235,512,557}
30	584	{0,36,66,101,145,199,503,584}
31	564	{0,31,73,109,158,190,513,564}
32	590	{0,39,80,112,148,241,528,590}
33	574	{0,37,70,110,149,199,509,574}
34	569	{0,41,78,112,154,242,512,569}
35	574	{0,36,74,109,152,205,512,574}
36	569	{0,41,78,114,165,208,515,569}
37	557	{0,44,90,128,165,236,508,557}
38	562	{0,47,85,129,169,238,510,562}
39	582	{0,47,104,143,184,230,522,582}
40	595	{0,40,85,133,191,237,517,595}
41	620	{0,107,199,279,360,443,579,620}
42	589	{0,42,89,145,193,270,461,589}
43	630	{0,57,106,173,216,270,498,630}
44	605	{0,44,92,142,195,261,514,605}
45	619	{0,51,96,143,207,268,528,619}
46	612	{0,63,112,166,212,293,565,612}
47	599	{0,88,138,199,246,308,544,599}
48	609	{0,80,128,180,233,330,547,609}
49	603	{0,53,102,172,222,284,539,603}
50	612	{0,93,181,257,332,427,562,612}
51	586	{0,58,117,170,244,315,535,586}
52	624	{0,57,110,170,222,300,481,624}
53	611	{0,73,130,183,237,323,499,611}

54	618	{0,57,111,167,232,301,527,618}
55	649	{0,73,129,184,242,307,563,649}
56	611	{0,67,125,191,247,328,513,611}
57	648	{0,68,125,183,247,323,550,648}
58	637	{0,66,142,201,259,338,567,637}
59	619	{0,104,166,233,292,353,529,619}
60	619	{0,63,124,197,277,337,513,619}
61	590	{0,79,142,209,270,352,518,590}
62	588	{0,65,136,198,267,355,501,588}
63	619	{0,102,165,229,300,374,543,619}
64	599	{0,70,135,203,282,346,513,599}
65	610	{0,65,141,207,275,367,522,610}
66	604	{0,103,180,263,336,410,538,604}
67	625	{0,75,142,210,289,370,527,625}
68	629	{0,68,144,213,286,381,470,629}
69	593	{0,70,151,220,299,393,480,593}
70	605	{0,80,150,232,311,397,497,605}
71	648	{0,98,190,273,355,440,577,648}
72	615	{0,77,172,248,320,407,522,615}
73	607	{0,88,168,245,318,419,529,607}
74	610	{0,74,156,236,325,430,531,610}
75	614	{0,82,159,234,327,410,518,614}
76	622	{0,91,171,253,330,434,546,622}
77	629	{0,77,159,245,338,432,549,629}
78	635	{0,78,165,247,340,440,556,635}
79	637	{0,95,174,256,337,438,551,637}
80	659	{0,99,183,266,355,456,579,659}

81	650	{0,82,163,248,338,441,537,650}
82	657	{0,82,168,251,348,439,553,657}
83	667	{0,83,175,259,348,454,571,667}
84	674	{0,85,169,257,350,459,558,674}
85	680	{0,86,175,260,366,457,581,680}
86	688	{0,89,177,263,362,465,580,688}
87	697	{0,92,179,272,373,462,583,697}
88	699	{0,95,183,273,376,470,593,699}
89	706	{0,96,185,276,380,475,599,706}
90	716	{0,94,184,275,383,483,603,716}
91	720	{0,91,183,277,379,488,606,720}
92	734	{0,92,187,283,387,489,609,734}
93	728	{0,93,187,283,383,491,615,728}
94	741	{0,95,189,287,390,506,615,741}
95	746	{0,95,193,289,396,498,618,746}
96	750	{0,97,193,293,395,505,633,750}
97	761	{0,97,195,295,405,511,644,761}
98	763	{0,98,197,298,403,516,645,763}
99	770	{0,99,199,301,407,521,651,770}
100	782	{0,101,201,307,411,525,650,782}
101	784	{0,101,203,307,415,531,663,784}
102	917	{0,105,214,316,419,538,662,917}
103	820	{0,104,207,313,423,541,675,820}

Annexe C

Résultats de la synthèse logique

C.1 Résultats de synthèse pour l'opérateur add-min générique

Nombre de bits	Nombre de ports	Délai(ns)	Nombre de slice	Nombre de LUT
2	3	6,262	5	2
2	4	6,532	6	2
2	5	6,546	8	5
2	6	6,252	8	4
2	7	7,721	10	6
2	8	5,858	12	7
2	9	5,288	14	9
2	10	7,841	15	7
2	11	7,909	18	8
2	12	6,952	18	8
2	13	6,886	21	10
2	14	6,472	21	10
2	15	7,429	24	11
2	16	5,722	25	11
2	17	6,878	27	14
2	18	8,216	28	14
2	19	7,779	31	14
2	20	9,509	32	16
3	3	5,853	10	7
3	4	5,945	14	11
3	5	6,875	21	20
3	6	6,893	25	25
3	7	5,755	26	23
3	8	6,488	34	31
3	9	7,181	36	33
3	10	8,462	43	41

3	11	8,291	51	53
3	12	8,187	54	54
3	13	9,126	54	53
3	14	9,393	61	55
3	15	9,908	63	58
3	16	8,572	68	65
3	17	8,137	80	83
3	18	8,589	84	89
3	19	11,556	82	75
3	20	10,670	95	99
4	3	9,401	16	17
4	4	9,750	20	20
4	5	14,285	26	30
4	6	13,806	31	33
4	7	11,647	37	42
4	8	12,725	42	46
4	9	16,854	49	55
4	10	16,968	53	58
4	11	15,965	60	68
4	12	15,701	64	71
4	13	16,786	71	80
4	14	16,427	78	85
4	15	15,261	82	93
4	16	14,831	87	96
4	17	18,933	96	108
4	18	19,125	100	110
4	19	18,579	107	119

4	20	18,572	112	123
5	3	9,542	21	22
5	4	10.358	26	26
5	5	14,193	35	39
5	6	13,050	40	43
5	7	13,477	49	55
5	8	13,869	55	60
5	9	17,074	64	72
5	10	17,539	69	76
5	11	15,488	80	91
5	12	15,386	84	93
5	13	16,654	93	105
5	14	16,382	100	111
5	15	17,177	107	122
5	16	16,082	114	126
5	17	18,527	125	140
5	18	19,490	129	145
5	19	19,610	139	156
5	20	19,774	145	161
6	3	9,486	24	28
6	4	11,926	30	33
6	5	16,976	41	49
6	6	15,348	47	54
6	7	14,252	57	69
6	8	14,656	64	75
6	9	17,443	76	90
6	10	17,977	81	95

6	11	17,855	96	114
6	12	17,884	101	117
6	13	17,924	110	131
6	14	16,855	115	137
6	15	18,313	127	152
6	16	17,568	134	157
6	17	19,887	148	181
6	18	19,108	158	189
6	19	19,682	167	202
6	20	19,741	173	207
7	3	10,029	32	37
7	4	12,062	39	43
7	5	15,506	52	62
7	6	15,520	59	68
7	7	14,462	75	90
7	8	15,344	83	97
7	9	18,521	99	119
7	10	17,781	107	127
7	11	19,372	120	146
7	12	18,794	128	152
7	13	19,891	144	171
7	14	17,902	147	174
7	15	17,594	163	197
7	16	17,754	174	207
7	17	19,805	191	239
7	18	19,195	197	239
7	19	19,837	216	259

7	20	19,764	223	269
8	3	11,686	35	42
8	4	13,458	43	49
8	5	16,859	58	71
8	6	16,760	66	78
8	7	15,005	83	103
8	8	15,404	92	111
8	9	17,689	110	136
8	10	17,838	119	145
8	11	18,547	135	168
8	12	17,435	142	174
8	13	19,355	163	199
8	14	19,000	170	207
8	15	18,639	182	226
8	16	18,899	190	230
8	17	19,873	215	274
8	18	19,796	224	284
8	19	19,395	241	307
8	20	19,836	248	313

C.2 Résultats de synthèse pour l'additionneur générique

Nombre de bits	Nombre de ports	Délai(ns)	Nombre de slice	Nombre de LUT
2	2	5,981	3	2
2	3	7,701	6	5
2	4	7,261	9	8
2	5	7,126	11	10
2	6	8,007	14	15
2	7	7,278	16	15
2	8	7,83	20	21
2	9	7,559	21	21
2	10	8,817	24	25
2	11	9,407	27	28
2	12	8,718	30	31
2	13	7,704	30	32
2	14	9,286	34	37
2	15	8,759	37	40
2	16	9,283	39	43
3	2	7,741	8	9
3	3	7,414	13	13
3	4	8,92	20	23
3	5	8,592	25	27
3	6	10,006	30	36
3	7	11,059	35	40
3	8	10,369	40	47
3	9	10,176	44	50
3	10	9,882	48	56
3	11	11,622	54	60
3	12	12,903	59	72

3	13	13,627	64	74
3	14	12,832	68	81
3	15	11,803	73	85
3	16	11,858	78	94
4	2	6,893	11	14
4	3	9,648	19	24
4	4	10,126	27	37
4	5	11,42	32	42
4	6	9,824	40	53
4	7	10,181	49	61
4	8	12,731	53	71
4	9	12,993	59	75
4	10	11,896	67	84
4	11	12,056	72	90
4	12	12,707	80	104
4	13	13,985	87	109
4	14	12,582	95	123
4	15	11,924	101	130
4	16	13,824	108	138
5	2	6,815	15	19
5	3	9,017	26	32
5	4	10,006	35	46
5	5	9,574	43	56
5	6	11,115	52	70
5	7	12,497	62	80
5	8	11,39	72	94
5	9	11,694	81	103

5	10	13,509	86	112
5	11	13,601	94	121
5	12	13,096	105	138
5	13	14,162	114	144
5	14	14,803	124	161
5	15	13,718	129	167
5	16	13,024	140	184
6	2	8,039	24	30
6	3	9,439	33	46
6	4	9,368	45	62
6	5	12,783	56	77
6	6	11,394	67	93
6	7	11,685	79	108
6	8	12,268	89	123
6	9	12,42	101	139
6	10	13,596	113	154
6	11	12,991	120	164
6	12	13,416	137	189
6	13	13,86	144	198
6	14	13,596	161	220
6	15	12,881	166	229
6	16	13,185	175	242
7	2	8,293	26	35
7	3	9,731	40	52
7	4	10,635	52	70
7	5	11,598	65	87
7	6	10,687	79	108

7	7	12,546	92	122
7	8	11,948	105	143
7	9	13,593	117	160
7	10	12,406	130	176
7	11	13,366	140	189
7	12	13,507	160	216
7	13	14,755	170	228
7	14	14,205	179	243
7	15	14,72	200	268
7	16	14,31	208	283
8	2	9,357	26	32
8	3	16,352	39	49
8	4	10,517	53	67
8	5	15,277	66	83
8	6	13,347	80	101
8	7	11,808	93	118
8	8	17,716	107	136
8	9	17,02	121	154
8	10	16,102	134	170
8	11	13,477	148	188
8	12	15,941	161	205
8	13	16,087	175	223
8	14	16,785	188	239
8	15	16,599	202	257
8	16	16,161	215	273