ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA
MAITRISE EN GÉNIE ÉLECTRIQUE
M.Ing.

PAR
ABDELKARIM OUADID

PROTOTYPE MICRO-ÉLECTRONIQUE D'UN DÉCODEUR ITÉRATIF POUR DES CODES DOUBLEMENT ORTHOGONAUX

MONTRÉAL, LE 20 DÉCEMBRE 2004

CE MÉMOIRE A ÉTÉ ÉVALUÉ
PAR UN JURY COMPOSÉ DE :

M. François Gagnon, directeur de mémoire
Département de génie électrique à l'École de technologie supérieure

M. David Haccoun, codirecteur de mémoire
Département de génie électrique à l'École Polytechnique de Montréal

M. Claude Thibeault, président du jury
Département de génie électrique à l'École de technologie supérieure

M. Jean Belzile, Membre du jury
Département de génie électrique à l'École de technologie supérieure

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC
LE 10 NOVEMBRE 2004
À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

PROTOTYPE MICRO-ÉLECTRONIQUE D'UN DÉCODEUR ITÉRATIF POUR DES CODES DOUBLEMENT ORTHOGONAUX

Abdelkarim Ouadid

SOMMAIRE

Ce mémoire porte sur le prototypage microélectronique FPGA d'un décodeur itératif doublement orthogonal issu de récents travaux de recherche. Le nouvel algorithme est simple et présente un certain nombre d'avantages par rapport aux codes turbo très prisés actuellement dans le codage de canal. En effet, ces derniers outre la complexité de leur algorithme de décodage, souffrent d'un problème de latence qui les rend inadaptés pour certaines applications, comme la téléphonie par exemple. Le décodeur utilisé, est un décodeur itératif à quantification souple, basé sur le décodage seuil tel que présenté par Massey et amélioré par l'approximation de la probabilité a posteriori (AAPP). Grâce à cette approche, on arrive à concilier complexité, latence, performance en correction d'erreurs, et haut débit de fonctionnement. Le prototype vise à valider les résultats de simulation, ainsi que l'estimation de la complexité et de la fréquence maximale que l'on peut atteindre sur des FPGA Virtex-II XC2V6000 et ceci pour différentes structures du décodeur.

MICROELECTRONIC PROTOTYPE OF AN ITERATIVE DECODER FOR SELF-DOUBLY ORTHOGONAL CODES

Abdelkarim Ouadid

ABSTRACT

In this thesis, a prototype of an iterative decoder using an FPGA technology is analyzed. The proposed algorithm is a result of recent research to find an alternative to Turbo codes. These turbo codes have a high complexity at the decoder and involve large latency due to the interleaving and de-interleaving processes. The iterative soft decision decoder we implemented is based upon the work of Massey and the approximation of the A Posteriori Probability, APP, known as the Approximate A Posteriori Probability, AAPP. Each iteration is made by a modified threshold decoder with a feedback of the estimated symbols. The structure provides a trade-off between complexity, performance in term of error correcting capability and a high through-put. The prototype will help validate simulations results, evaluate the complexity of the system and the maximum frequency of the circuit in a Virtex XC2V6000 device.

REMERCIEMENTS

Je remercie en premier lieu le professeur François Gagnon, mon directeur de mémoire, pour la confiance qu'il a eu en moi en me confiant un sujet aussi riche et stimulant, ainsi que pour ses conseils et son encadrement.

Je remercie aussi le professeur David Haccoun, de l'École Polytechnique de Montréal, pour sa co-direction de mon mémoire et l'aide financière qu'il m'a octroyée. Puisse ce travail être à la hauteur de ses attentes.

Je ne peux qu'exprimer ma gratitude à Monsieur Christian Cardinal pour les heures passées à discuter de l'aspect théorique de sa thèse. Sa générosité en temps et en explications m'a grandement facilité la tâche.

Une pensée va à l'ensemble des étudiants que j'ai côtoyé durant mes études à l'ETS, dans un climat fraternel et studieux.

Ce travail est dédié à toutes les personnes que j'aime et qui me le rende bien. À mon père, ma mère, mes soeurs et mon jeune frère qui ont toujours cru en moi et qui me manquent énormément. Aux familles Chouinard, Robitaille et Giguère qui rendent mon séjour au Québec des plus agréables. À ma chère et tendre Nadia, qui m'a énormément encouragé, supporté et subit les affres des corrections de mes ébauches.

TABLE DES MATIÈRES

Page

SOMMAIRE i

ABSTRACT ii

REMERCIEMENTS iii

TABLE DES MATIÈRES iv

LISTE DES TABLEAUX vii

LISTE DES FIGURES viii

LISTE DES ABRÉVIATIONS ET SIGLES xi

INTRODUCTION 1

CHAPITRE 1 INTRODUCTION AU CODAGE DE CANAL 3

1.1 Introduction 3
1.2 Les systèmes de communications numériques 3
1.3 Théorie de la capacité d'un canal 8
1.4 Le canal discret 9
1.4.1 Le canal discret sans mémoire (DMC) 9
1.4.2 Le canal binaire symétrique (BSC) 10
1.4.3 Le canal binaire à effacement (BEC) 11
1.4.4 Le canal à entrée binaire et bruit blanc additif gaussien 12
1.5 Le codage convolutionnel 13
1.5.1 Structure des codes convolutionnels 13
1.5.1.1 Code convolutionnel de taux $\frac{1}{\nu}$ 13
1.5.1.2 Code convolutionnel de taux $\frac{k}{\nu}$ 14
1.5.2 Représentation polynomiale 15
1.5.3 Codes convolutionnels systématiques et récursifs, RSC 16
1.5.4 Représentation sous forme graphique du processus de codage 17
1.6 Les codes turbo 19
1.6.1 La concatenation 20

1.6.2 L'entrelacement 21
1.7 Algorithmes de décodages 22
1.8 Le décodage itératif 24
1.8.1 Algorithme BCJR 26
1.8.2 Algorithme SOVA 27
1.9 Conclusion 27

CHAPITRE 2 BASES THÉORIQUES DU DÉCODEUR 29

2.1 Introduction 29
2.2 Décodage à seuil en quantification ferme pour les codes convolutionnels 30
2.2.1 Code orthogonal (CSOC) 32
2.2.2 Décodage des codes CSOC 33
2.3 Adaptation du décodage à seuil à des sorties non quantifiées 37
2.4 Décodage itératif, définition des codes convolutionnels doublement orthogonaux au sens large CSO^{2}C-WS 43
2.4.1 Le processus itératif 43
2.4.2 Les conditions de double orthogonalité 45
2.4.3 Les codes convolutionnels doublement orthogonaux au sens large 46
2.5 Les codes convolutionnels doublement orthogonaux au sens strict CSO^{2}C-SS 48
2.6 Conclusion 50

CHAPITRE 3 ARCHITECTURE DU CIRCUIT PROGRAMMABLE FPGA CIBLE ..52

3.1 Introduction 52
3.2 Les circuits FPGA 53
3.2.1 Architecture des FPGA 53
3.2.1.1 Les CLB 53
3.2.1.2 Les IOB 55
3.2.1.3 Les interconnexions 57
3.2.2 Cas des FPGA de la famille Virtex 59
3.3 Outils et méthodologie de développement pour FPGA 63
3.3.1 La saisie du design 63
3.3.2 La synthèse 67
3.3.3 Optimisation, projection et placement-routage 67
3.4 Conclusion 68

CHAPITRE 4 ARCHITECTURE DU DÉCODEUR CSO^{2}C-SS 71

4.1 Introduction 71
4.2 Analyse de l'algorithme du décodeur CSO^{2}C-SS 72
4.2.1 Décomposition de l'algorithme 73

4.2.2 Analyse des opérations de base à effectuer 74
4.3 Architecture proposée 75
4.3.1 Nature de l'information utilisée 77
4.3.2 Les blocs FIFO de mémorisation et de décalage 78
4.3.3 Les blocs FIFOULP 78
4.3.4 Blocs de complément à deux 82
4.3.5 Bloc de sommation 83
4.3.6 Bloc de saturation 85
4.3.7 Assemblage des différents blocs 86
4.4 Codage en VHDL et validation fonctionnelle 88
4.4.1 Hiérarchie des fichiers VHDL et leurs descriptions 88
4.4.2 Validation fonctionnelle et procédures de vérification 95
4.5 Généralisation pour l'ensemble des décodeurs 97
4.5.1 Calcul du nombre de bits de dépassement de capacité 98
4.5.2 Mise en place de la structure d'addition 99
4.5.3 Mise en place du FIFOULP 102
4.6 Conclusion 102

CHAPITRE 5 RÉSULTATS ET ANALYSES 104

5.1 Introduction 104
5.2 Optimisation de l'architecture du décodeur 105
5.2.1 Modification du code HDL des FIFO 105
5.2.2 Amélioration de la fréquence maximale 108
5.2.3 Initialisation du système et sa synchronisation 111
5.3 Étude de la complexité du système 114
5.3.1 Méthode de calcul de la complexité pour les différents blocs 115
5.3.2 Complexité estimée versus la complexité relevée 120
5.3.3 Rapidité du décodeur 123
5.4 Étude des performances du décodeur 125
5.4.1 Procédure de mesure de la performance 125
5.4.2 Résultats des simulations 126
5.5 Conclusion 128

CONCLUSION 130

ANNEXES

1 : CODE VHDL POUR UN DOUBLE DÉCODEUR AVEC J=3 ET PIPELINE 132
2 : CODE C POUR UN GÉNÉRATEUR DE CODE VHDL DU DÉCODEUR CSO^{2}C-SS 152

BIBLIOGRAPHIE 190

LISTE DES TABLEAUX

Page

Tableau I Deux représentations possibles des q bits des symboles. 77

Tableau II Nombre de bits de dépassement en fonction du code 98

Tableau III Compilation des résultats de l'étude de la complexité en nombre de tranches. 121

Tableau IV Compilation des résultats de l'étude de la complexité en nombre de tranches (suite). 122

Tableau V Synthèse de l'étude de la fréquence maximale des décodeurs en Mhz 124

LISTE DES FIGURES

Page

Figure 1.1 Chaîne de communication numérique 4
Figure 1.2 Modèle du canal DMC 10
Figure 1.3 Modèle du canal BSC 10
Figure 1.4 Modèle du canal BEC 11
Figure 1.5 Modèle du canal AWGN 12
Figure 1.6 Représentation en arbre d'un code convolutionnel NRNSC(7,5) de taux 1/2 et de longueur de contraintes 3 18
Figure 1.7 Représentation du code NRNSC(7,5) par un diagramme d'états 19
Figure 1.8 Représentation du code NRNSC(7,5) par un treillis 19
Figure 1.9 Structures de concaténation de codeurs (a) en série, (b) en parallèle, (c) produit 20
Figure 1.10 Principe du décodage Turbo 25
Figure 2.1 Modèle simplifié d'un système de communication numérique avec canal BSC 30
Figure 2.2 Réalisation d'un codeur CSOC avec J=3, m=3 et $\alpha = (0,1,3)$ 33
Figure 2.3 Montage de calcul des syndromes 34
Figure 2.4 Décodeur à seuil en quantification ferme avec rétroaction 36
Figure 2.5 Ensemble codeur, canal BSC et décodeur a seuil pour un code k=2 et R=1/2. 37
Figure 2.6 Modèle d'un système de communication pour un canal non quantifié 38
Figure 2.7 Le canal équivalent gaussien 38
Figure 2.8 Codeur CSOC avec J=4 42
Figure 2.9 Décodeur à seuil a entrée non-quantifiée avec rétroaction et opérateur addmin 43

Figure 2.10 Schéma de principe de M itérations de décodage à seuil pour un code CSO^2C-WS 45

Figure 2.11 Codeur doublement orthogonal au sens strict avec J=3 de taux $R = \frac{J}{2J} = \frac{1}{2}$ 49

Figure 3.1 Diagramme simplifié d'une tranche d'un FPGA Virtex de Xilinx™ 54

Figure 3.2 Schéma simplifié d'un bloc IOB d'un FPGA Virtex de Xilinx™ 56

Figure 3.3 Exemple de lignes d'interconnections dans les FPGA, avec les matrices de connexions programmables (PSM) 58

Figure 3.4 Interconnections directe 59

Figure 3.5 Architecture générale des Xilinx™ Virtex (a) et Virtex-E (b) 60

Figure 3.6 Architecture simplifiée du CLB d'un circuit Virtex 61

Figure 3.7 Nomenclature d'un circuit de la famille Virtex 62

Figure 3.8 Les deux flux de programmation des FPGA 65

Figure 4.1 Diagramme boite noire des blocs de décodage 73

Figure 4.2 Structure bloc du décodeur intermédiaire 76

Figure 4.3 Réalisation d'un bloc FIFO avec des bascules D 78

Figure 4.4 Schéma bloc du FIFOULP 79

Figure 4.5 Exemple d'insertion des addmin au sein d'un FIFO 81

Figure 4.6 Schéma bloc des complément à deux en entrée (a) et sortie (b) du sommateur 83

Figure 4.7 Structure d'addition en cascade (a), dyadique (b) et exemple d'utilisation (c) 84

Figure 4.8 Schéma bloc du sommateur 84

Figure 4.9 Schéma bloc du saturateur 86

Figure 4.10 Structure bloc du décodeur frontal 87

Figure 4.11 Structure bloc du décodeur terminal 87

Figure 4.12 Hiérarchie des fichiers VHDL 89

Figure 4.13 Résultats graphiques d'une simulation du code VHDL 96

Figure 4.14 Organigramme pour le calcul du nombre de bits d'extension nécessaires 99
Figure 4.15 Exemple de décomposition de l'addition en niveaux pour deux entrées (a) et trois entrées (b) 100
Figure 5.1 Mise en place d'un registre à décalage de profondeur 128 bits à l'aide des SRL16 107
Figure 5.2 Principe du Pipeline 109
Figure 5.3 Stratégie de pipelining pour le décodeur intermédiaire 110
Figure 5.4 Stratégie de pipeline pour le sommateur 111
Figure 5.5 Système activ2 d'initialisation des FIFOULP 112
Figure 5.6 Décodeur frontal optimisé 113
Figure 5.7 Exemple de structure du FIFOULP 118
Figure 5.8 Résultats des simulations pour deux itérations de décodage pour le taux J=3 et q=3 127
Figure 5.9 Résultats des simulations pour trois itérations de décodage pour un taux J=3 et q=3 127
Figure 5.10 Résultats des simulations pour deux itérations de décodage pour le taux J=5 et q=3 128

LISTE DES ABRÉVIATIONS ET DES SIGLES

APP	A posteriori probability
AAPP	Approximate a posteriori probability
ASIC	Application specific integrated circuits
AWGN	Additive white gaussian noise
BEC	Binary erasure channel
BPSK	Binary phase shift keying
BRAM	Blockselect RAM
BSC	Binary symmetric channel
CLB	Configurable logic block
CMOS	Complementary metal-oxide-silicon
CSO^2C	Convolutional self doubly orthogonal code
CSO^2C-SS	Convolutional self doubly orthogonal code in the strict sens
CSO^2C-WS	Convolutional self doubly orthogonal code in the wide sens
CSOC	Convolutional self orthogonal code
dB	Decibel
DLL	Delay locked loop
DMC	Discrete memoryless channel
EDIF	Electronic digital interchange format
FIFO	First in first out
FIFOULP	First in first out pour les symboles U, L et P
FPGA	Field programmable gate array
Gbps	Giga bits par seconde
GPS	Global positioning system
HDL	Hardware description language
IIS	Interférences inter symboles
IOB	Input output block

JTAG	Joint test action group
LACIME	Laboratoire de communication et d'intégration de la microélectronique
LRV	Logarithme du rapport de vraisemblance
LSB	Least significant bit
LUT	Look-up table
MAP	Maximum a posteriori
Mbps	Mega bits par seconde
MHz	Mega Hertz
MSB	Most significant bit
NRNSC	Non recursive non systematic code
OFDM	Orthogonal frequency division multiplexing
Perl	Practical extraction and report language
PLD	Programmable logic device
RAM	Random access memory
RF	Radio fréquence
ROM	Read-only memory
S/R	Set/Reset
SISO	Soft input soft output
SNR	Signal to noise ratio
SOC	System on chip
SOVA	Soft output Viterbi decoder
SRAM	Static random access memory
SRL16	Shift left register 16
TCL	Tool command language
TTL	Transistor-transistor logic
VHDL	Very high speed integrated circuits hardware description language
VHDL-AMS	Very high speed integrated circuits hardware description language-analog mixed signal

INTRODUCTION

Le décodage itératif consiste à utiliser des décodeurs en cascades de façon à réaliser avec une succession de décodeurs de coûts faibles, les performances que l'on peut atteindre avec un décodeur très complexe donc coûteux. Les études menées au cours des dernières années ont permis de démontrer que l'on est en mesure de dépasser les limites des décodeurs classiques pour s'approcher de plus en plus de la limite dite de Shannon, imposée par la théorie édictée en 1948.

Les codes turbo présentent une avenue intéressante, en utilisant une structure itérative basée sur un décodeur de type SISO («Soft-Input/Soft-Output»). La technique de décodage, bien qu'étant itérative, reste encore très complexe et surtout introduit une latence qui tend à limiter la longueur de contrainte des codes que l'on peut utiliser et donc leur pouvoir correcteur. Il s'avère donc impossible de profiter pleinement de cette innovation pour les applications où de grandes latences sont inacceptables, la téléphonie par exemple. À partir de ces observations, un travail de recherche a été initié par D. Haccoun, F. Gagnon et C. Cardinal, et a aboutit avec la thèse de doctorat de ce dernier [1] portant sur un algorithme de décodage itératif sans entrelaceurs. Les décodeurs itératifs pour les codes doublement orthogonaux au sens strict utilisent un algorithme de décodage à seuil à quantification souple, et emploient une approximation de la probabilité a priori. Il en ressort une structure parallèle, moins complexe et donnant des résultats intéressants.

Ce mémoire s'applique à présenter la mise en place d'un prototype micro programmable ciblant des FPGA de la société Xilinx^{TM}. La rédaction a été guidée par le souci d'amener graduellement le sujet à terme, en éclaircissant au fur et à mesure toutes les notions nécessaires à son développement. Ainsi, le premier chapitre présente brièvement les notions de codage de canal, des codes convolutionnels et de leur décodage, puis les codes turbo et leur décodage itératif. Le second chapitre constitue un résumé succinct de

la thèse de C. Cardinal et plus précisément présente les codes convolutionnels doublement orthogonaux au sens strict ainsi que leur décodage décrit par un système d'équations. Le chapitre trois passe en revue la technologie microélectronique cible, en mettant l'accent sur l'architecture générale des FPGA de Xilinx^{TM}, des outils de développement ainsi que des structures internes propres à la famille Virtex. Ceci est nécessaire, puisque les performances sont fortement tributaires de l'exploitation ou non de certaines ressources optimisées pour le type d'applications que l'on traite. Il s'en suit un chapitre où on propose une architecture possible et qui est réalisée en un langage de description matérielle («Hardware Description Language, HDL») pour un cas de codeur, ceci avant de décrire la procédure qui permet de générer de façon automatisée le code HDL pour n'importe quel codeur. Le chapitre cinq apporte deux modifications à la description HDL de notre décodeur afin de réduire sa complexité d'une part et d'augmenter sa fréquence maximale de fonctionnement. Dans ce même chapitre, on procède à une étude de la complexité estimée mathématiquement et celle que l'on obtient par des outils spécialisés, ainsi qu'à l'analyse des performances des décodeurs en terme de correction d'erreurs. Enfin, une conclusion présente des idées d'améliorations possibles et des recommandations.

CHAPITRE 1

INTRODUCTION AU CODAGE DE CANAL

1.1 Introduction

Les systèmes de communication modernes qui sont de nos jours en majorité numériques, nécessitent souvent des liaisons où le taux d'erreurs binaires doit être maintenu au strict minimum en conjonction avec la puissance de transmission disponible, afin de ne pas altérer l'intégrité de l'information transmise. On peut penser, par exemple, à des domaines critiques comme la télécommande d'éléments de protection ou de sécurité dans les réseaux électriques, voire des centrales nucléaires ou plus communément les transactions bancaires et les systèmes de positionnement géographique par satellite (Global positioning system, GPS). Afin d'atteindre ce but, plusieurs techniques de codage de canal ont été développées et la littérature ne cesse de s'enrichir. Le présent chapitre vise à introduire ces techniques de codage qui permettent d'atteindre la fiabilité voulue dans les systèmes de communication. On commencera d'abord par décrire les différentes composantes de tels systèmes, puis on s'intéressera à un type de codage particulier, le codage convolutionnel. Ceci nous amènera à présenter les codes turbo et à discuter de leur intérêt. Nous mettrons aussi en place les bases de l'implémentation qui fait l'objet de ce mémoire, en présentant et analysant brièvement les algorithmes de décodage les plus utilisés pour les codes convolutionnels et leur adaptation pour les codes turbo.

1.2 Les systèmes de communications numériques

Les systèmes de communications numériques visent à transférer de l'information entre deux entités (ou plus) par le biais de différent mediums tels les câbles, la fibre optique

ou la propagation de signaux radioélectriques. Généralement, on vise trois impératifs dans l'utilisation et donc la conception de ces systèmes :

- L'efficacité spectrale, qui représente la capacité de la technique de modulation utilisée à transporter l'information dans une bande de fréquences la plus limitée possible.
- L'efficacité de rendement de puissance, qui impose d'utiliser le moins d'énergie possible pour transférer l'information en toute fiabilité.
- La complexité des équipements qui entre en ligne de compte dans le coût final de la solution et de sa date de mise en marché. Plus la solution est complexe, plus les coûts sont élevés et le temps de développement est long.

Il est intéressant de remarquer qu'on ne peut disposer d'un système parfait qui répond à tous les critères. Dans les faits, on optimise notre conception en fonction des critères d'un cahier de charges, en relaxant certaines contraintes secondaires pour en atteindre d'autres jugées primordiales pour l'utilisation finale.

Le schéma synoptique d'une chaîne de communication numérique peut être synthétisée comme à la figure 1.1.

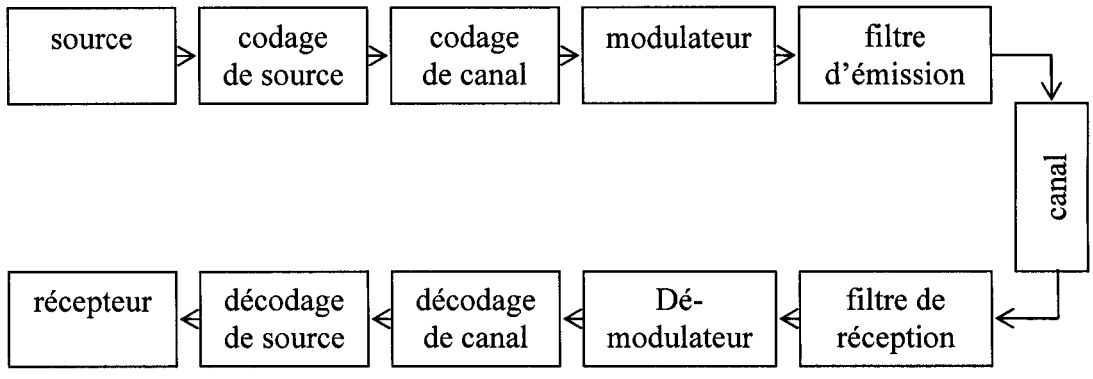

Figure 1. 1 Chaîne de communication numérique

Cette chaîne est composée des éléments suivants :

- La source : elle génère un message qui peut être, soit continu (parole, capteurs, etc.), soit discret (réseaux informatiques, capteurs intelligents, etc.).

- Le codeur de source : il se charge quant à lui de transformer l'information en une suite binaire contenant le moins de bits possibles [2], deux cas sont à distinguer :
  - Si le signal est continu, dans ce cas on procède à un échantillonnage en respectant le critère de Nyquist

    $$f_s > 2B_{signal} \tag{1.1}$$

    où :

    f_s : La fréquence d'échantillonnage en Hz,

    B_{signal} : Largeur de bande du signal en Hz.

    L'étape de quantification qui suit, introduit une perte d'information se traduisant par l'ajout d'un bruit de quantification.
  - Si le signal est discret, on cherche alors à maximiser l'entropie du signal binaire, en enlevant toute redondance d'information des données de la source.

- Le codeur de canal : il vise à protéger l'information des bruits et perturbations qui sont présents sur le canal de transmission. On arrive à cela en transformant k symboles à l'entrée du codeur, en n symboles à sa sortie par ajout de bits de redondance de façon bien contrôlée, d'où n>k. Ces bits ajoutés permettent au décodeur de canal au niveau du récepteur de détecter et/ou corriger un nombre d'erreurs qui dépend du couple codeur/décodeur utilisé. Ce processus nous conduit à améliorer nos rendements d'erreur en utilisant moins de puissance que si on procédait sans codage. Le revers de la médaille est une augmentation assez significative de la complexité du système et de la largeur de bande.

On distingue deux catégories de codes :

- Les codes en blocs qui sont généralement réalisés avec des circuits combinatoires affectant un bloc entier de bits. On citera, par exemple, les codes suivants qui font partie de cette catégorie [2], Bose-Chaudhuri-Hocquenghem (BCH), Reed-Muller (RM), codes cycliques, codes de Hamming, codes de Reed-Solomon (RS), etc.
- Les codes convolutionnels dont les codeurs sont généralement réalisés grâce à de la logique séquentielle [3], et qui sont aussi appelés les codes en treillis ou en arbres.

- Le bloc de modulation : à chaque symbole codé, il associe une forme d'onde spécifique. On procède à la transmission, soit en :
  - Bande de base quand le spectre du signal émis reste centré sur zéro Hz,
  - Bande passante quand on procède à une translation du spectre du signal vers une fréquence centrale, plus compatible avec le canal à utiliser.

Afin d'améliorer notre système, on peut décider de grouper les bits en blocs de taille N et à chaque symbole formé par cette concaténation de bits, on associe une forme d'onde d'une constellation. Cette opération permet d'augmenter significativement l'efficacité spectrale du système. Dans les applications civiles modernes, les modulations en quadrature MQAM et en phase MPSK sont les plus généralement utilisées [2].

Une seconde technique, qui est très utilisée dans les canaux sujets à des perturbations radio fréquences (RF), ainsi qu'aux délais dûs à des multi trajets, consiste à moduler l'information sur plusieurs porteuses qui sont orthogonales entre elles. On nomme cette technique l'OFDM (Orthogonal Frequency Division Multiplexing) [4].

- Le canal de communication, caractérisé par sa réponse en fréquence, se comporte comme un filtre avec un ensemble de défauts qui le rendent non idéal. On assiste ainsi à :
    - Une distorsion du signal suivant les fréquences. L'utilisation de la technique OFDM qui est robuste aux distorsions limite ce problème, et on rétablit le signal transmis par égalisation. Plusieurs modèles sont disponibles pour l'étude de ce phénomène, comme le canal de Rayleigh, le canal de Rice et le canal de Rummler [2].

    - La limitation en bande passante du canal, impose un effet de filtre passe bas. Ceci entraîne des interférences inter symboles (IIS), que l'on corrige par égalisation.

    - Le canal étant un milieu sujet au bruit, ce dernier s'ajoute au signal. Dans sa forme la plus simple, on considérera un bruit blanc gaussien additif.

- Les filtres d'émission et de réception, servent à annuler les IIS. Ce sont des filtres adaptés (matched filter), qui visent à maximiser le rapport signal à bruit (Signal to Noise Ratio, SNR) avant la prise de décision au démodulateur.

On ne s'attardera pas sur les blocs à la réception, étant donné qu'ils font exactement l'opposé des blocs à l'émission. Néanmoins, on tient à préciser que le modèle que l'on présente ici est simplifié, puisqu'il faut d'autres blocs chargés de différentes tâches tout aussi importantes pour le bon fonctionnement du système. On citera par exemple les éléments de synchronisation de bits, de symboles, de trames et de phases, le cryptage pour la confidentialité, la signalisation sur les réseaux, etc.

Il faut noter que pour l'étude des modèles de codage et de décodage de canal, on a recours à une simplification de la chaîne de communication en considérant les blocs de

filtrage d'émission et de réception, de modulation et du canal comme une seule entité que l'on désignera comme canal discret.

1.3 Théorie de la capacité d'un canal

Shannon a démontré que pour un canal à bruit blanc et gaussien, on peut exprimer sa capacité comme suit [5] :

$$C = W \log_2\left(1+\frac{S}{N}\right) \quad \text{(bits/s)} \tag{1.2}$$

avec :

C : Capacité du canal en bits par seconde (bits/s),

W : Largeur de bande du signal (Hz),

S : Puissance moyenne reçue (Watt),

N : Puissance moyenne du bruit reçu (Watt) dans la bande W,

Il est donc théoriquement possible de transmettre à travers un canal à n'importe quel taux de transmission R, à condition que R<C, avec un taux d'erreur aussi faible que l'on veut par l'ajout de complexité de codage.

Si le taux de transmission R est supérieur à la capacité, R>C, alors il est théoriquement impossible d'avoir un code qui permette un taux d'erreurs qui s'approche de zéro. Le théorème de Shannon démontre ainsi que les valeurs S, N et W limitent le taux de transmission et non la probabilité d'erreur. Ainsi si l'on développe l'équation (1.2) en considérant que :

$$N = WN_0 \tag{1.3}$$

où N_0 est la densité spectrale de bruit,

dans un cas limite on aura :

$$R = C \tag{1.4}$$

alors on peut écrire que :

$$\frac{S}{N_0}C = \frac{E_b}{N_0} \tag{1.5}$$

avec E_b l'énergie par bit,

et on obtient :

$$C = W \log_2\left[1 + \frac{E_b}{N_0}\frac{C}{W}\right] \tag{1.6}$$

Or en faisant tendre $\frac{C}{W}$ vers zéro et en sachant que $\ln(1+x)^{\frac{1}{x}} = e$, on arrive à la conclusion suivante :

$$\frac{E_b}{N_0} = \frac{1}{\log_2(e)} = -1.56 \ dB \tag{1.7}$$

Cette valeur constitue la limite de Shannon en dessous de laquelle aucune communication sans erreur n'est possible quelque soit la vitesse de transmission et la technique envisagée.

1.4 Le canal discret

Différents modèles de canaux discrets existent dans la littérature. Selon le système de communication étudié, on choisit l'approche qui rend le plus fidèlement compte des phénomènes à analyser et du niveau de détail que l'on veut atteindre. Nous présentons, dans ce qui suit, les principaux modèles utilisés dans l'étude du codage de canal.

1.4.1 Le canal discret sans mémoire (DMC)

Le canal DMC (Discrete Memoryless Channel) est constitué d'un modulateur M-aire, d'un chemin de transmission et d'un démodulateur Q-aire. On peut représenter ce canal à l'aide du diagramme de la figure 1.2. Les P(j/i) définissent la probabilité de recevoir le symbole j sachant que le symbole i a été transmis avec i allant de 1 à M et j allant de 1 à

Q. On notera que si Q>M, on se trouve dans le cas d'un canal à quantification souple (Soft Decision), c'est-à-dire que le démodulateur en plus de prendre une décision, fournit une indication de fiabilité sur cette dernière.

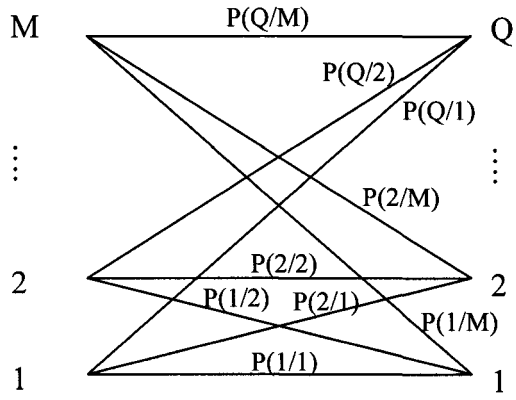

Figure 1. 2 Modèle du canal DMC

1.4.2 Le canal binaire symétrique (BSC)

Le cas le plus simple de canal DMC est le canal BSC (binary symmetric channel). Les alphabets d'entrée et de sortie sont tous deux binaires, et le diagramme du DMC se trouve simplifié comme on peut le voir sur la figure 1.3. On remarque aussi que comme Q=M=2, le démodulateur procède à une prise de décision sur l'information transmise au décodeur. On se trouve alors avec un démodulateur à sorties à quantification ferme ou dure (Hard Decision).

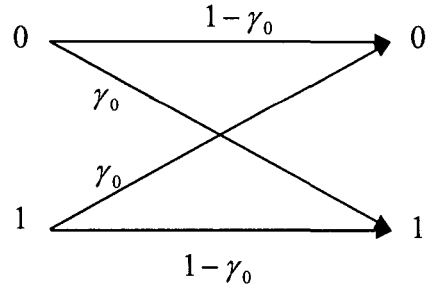

Figure 1. 3 Modèle du canal BSC

Pour la modulation BPSK avec un canal à bruit blanc gaussien additif (AWGN), la probabilité d'erreur par bit est donnée par :

$$\gamma_0 = P_b = Q\left(\sqrt{\frac{2E_s}{N_0}}\right) = Q\left(\sqrt{\frac{2E_b}{N_0}}\right) \tag{1.8}$$

où $Q(\alpha)$, dite fonction d'erreur de la distribution gaussienne, est donnée par (1.9).

$$Q(\alpha) = \frac{1}{\sqrt{2\pi}} \int_{\alpha}^{+\infty} e^{-\frac{\beta^2}{2}} d\beta \tag{1.9}$$

Il est à noter que E_s représente l'énergie par symbole.

1.4.3 Le canal binaire à effacement (BEC)

Le canal BEC (Binary Erasure Channel) est binaire en entrée mais ternaire en sortie. On ajoute en sortie un troisième symbole noté ζ. Le diagramme de transition de ce type de canal est représenté à la figure 1.4. On remarque bien qu'il n'y a pas d'erreur, puisqu'une transition de zéro vers un ou de un vers zéro est impossible. Les transitions possibles sont de zéro vers zéro avec une probabilité 1-q, de un vers un avec une probabilité 1-q, et de un ou zéro vers ζ avec une probabilité de q. Lorsque la sortie du canal est ζ, aucune décision n'est prise, d'où l'effacement.

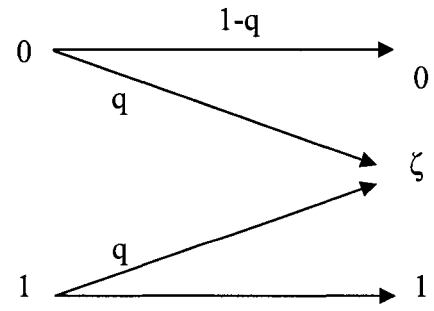

Figure 1. 4 Modèle du canal BEC

1.4.4 Le canal à entrées binaires et bruit blanc additif gaussien

Ce canal est très important analytiquement dans l'étude des techniques de codage de canal et spécialement dans le cas du décodage à entrées souples. Les hypothèses de ce modèle sont :

- la source est binaire et idéale,
- le modulateur associe à chaque élément binaire un signal antipodal (modulation BPSK),
- le canal de transmission affecte les signaux par l'ajout d'un bruit indépendant du signal, stationnaire, gaussien et blanc avec une variance σ^2 et une densité spectrale bilatérale de $\frac{N_0}{2}$,
- le démodulateur avec un filtre adapté est optimal,
- l'échantillonnage est supposé effectué aux instants adéquats.

On représente graphiquement ce canal par le diagramme de la figure 1.5. Les symboles de l'entrée sont binaires, mais les symboles de sortie prennent des valeurs dans l'espace des réels. On exprime la sortie suivant l'équation (1.10).

$$y=\pm 1+n \tag{1.10}$$

où n est une variable aléatoire Gaussienne de moyenne nulle et de variance $\frac{N_0}{2}$.

Ce genre de canal peut se transformer en BSC par l'adjonction d'un quantificateur sur les échantillons à l'entrée du décodeur.

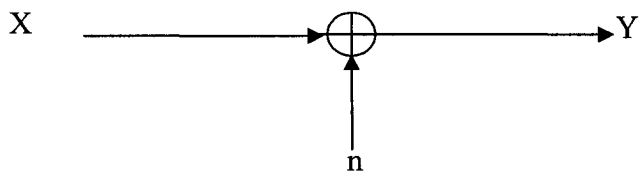

Figure 1. 5 Modèle du canal AWGN

1.5 Le codage convolutionnel

Les codes convolutionnels ont été introduits en 1955 par Elias [3] comme alternatives aux codes en blocs. De ce fait, ils peuvent être considérés comme un cas particulier de ces derniers.

Ce type de codes ne cesse de s'imposer année après année, et constituent actuellement la famille la plus utilisée dans les systèmes de télécommunications fixes et mobiles. Ils s'appliquent, à la différence des codes en blocs, sur une séquence infinie de symboles pour générer une séquence infinie de symboles codés.

1.5.1 Structure des codes convolutionnels

1.5.1.1 Code convolutionnel de taux $\frac{1}{\nu}$

Il est aisé de considérer un code convolutionnel comme un code bloc linéaire décrit par une matrice génératrice sous la forme suivante :

$$\mathbf{G} = \begin{bmatrix} \mathbf{g}_0 & \mathbf{g}_1 & \mathbf{g}_2 & \Lambda & \mathbf{g}_{L-1} & & & & \\ & \mathbf{g}_0 & \mathbf{g}_1 & \mathbf{g}_2 & \cdots & \mathbf{g}_{L-1} & & & \\ & & \mathrm{O} & \mathrm{O} & & & \mathrm{O} & & \\ & & & \mathrm{O} & \mathrm{O} & & & \mathrm{O} & \\ & & & & \mathrm{O} & \mathrm{O} & & & \mathrm{O} \end{bmatrix} \tag{1.11}$$

avec les éléments $g_i, 0 < i \leq L-1$, des vecteurs ligne binaire de dimension ν que l'on écrit :

$$\mathbf{g_i} = [g_{i1} \quad g_{i2} \quad \cdots \quad g_{i\nu}] \tag{1.12}$$

Les éléments non spécifiés dans la matrice génératrice sont tous des zéros, et chaque ligne se trouve en décalant de ν éléments la ligne précédente vers la droite. La matrice

génératrice du code convolutionnel est une matrice semi infinie, puisque la séquence en entrée est d'une taille arbitraire à l'opposé des codes en blocs. On remarque que pour chaque bit en entrée, on génère v bits en sortie d'où le rendement de ce code que l'on appelle aussi taux de codage $R = \frac{1}{v}$. L, est appelé la longueur de contrainte du code convolutionnel, et on définit la mémoire m qui est égale à L+1.

On parle de convolution, puisque dans l'écriture de la séquence de bits **C** générée pour une suite de bits en entrée **U**, on a :

$$\mathbf{C} = \mathbf{GU} \tag{1.13}$$

et en développant le produit matriciel, pour chaque v bit de sortie on obtient un produit de convolution [2].

1.5.1.2 Code convolutionnel de taux $\frac{k}{v}$

L'extension de l'approche du codeur de taux $\frac{1}{v}$ à un codeur de taux $\frac{k}{v}$ s'obtient par la modification de la forme de la matrice génératrice **G** et par suite du schéma du codeur. Dans cette hypothèse, on a k bits en entrée qui parviennent au codeur en parallèle et subissent un décalage temporel. Cette approche nous mène à considérer les éléments $\mathbf{g_j}$ non plus comme des vecteurs lignes mais comme des matrices de dimension $k \times v$.

Un exemple simple peut être proposé en considérant le code convolutionnel suivant de rendement $\frac{2}{3}$ avec une longueur de contrainte $L = 2$ que l'on définit par deux matrices :

$$g_0 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \text{ et } g_1 = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \tag{1.14}$$

Les sorties binaires dépendent non seulement des k entrées binaires, mais aussi de l'état des registres à décalages. Il existe donc 2^k transitions possibles à partir d'un état du

code à un instant donné. La complexité du codeur ne dépend ainsi que de la longueur de contrainte L et de k [2].

1.5.2 Représentation polynomiale

Il est assez intéressant d'avoir recours à la notation polynomiale pour décrire les codes convolutionnels, étant donné qu'une convolution temporelle correspond en fait à une multiplication polynomiale avec addition.

Ainsi, si l'on considère un code convolutionnel de rendement $R = \frac{k}{\nu}$ et longueur de contrainte L, son entrée peut être vue comme une suite infinie de bits sur k entrées différentes que l'on peut écrire sous la forme :

$$u_i(D) = \sum_{l=0}^{+\infty} u_{il} D_l, \ i = 1,...,k \tag{1.15}$$

Avec D symbolisant un retard d'une période (Delay) et u_{ij} le bit présent à l'instant j à l'entrée i.

On exprime de même les ν sorties du codeur :

$$c_j(D) = \sum_{l=0}^{+\infty} c_{jl} D_l, \ j = 1,...,\nu \tag{1.16}$$

Reste à identifier les polynômes générateurs qui s'obtiennent en considérant que s'il y a connexion entre le r$^{\text{ième}}$ élément du registre de l'entrée i et l'additionneur modulo 2 de la sortie j, alors le coefficient du degré r du polynôme g_{ij} vaut un.

On peut enfin obtenir les sorties en fonction des entrées et des polynômes générateurs grâce à la formulation suivante :

$$c_j(D) = \sum_{i=1}^{k} g_{ij}(D) u_i(D), \ j = 1,...,\nu \tag{1.17}$$

Sous forme matricielle compacte en utilisant notre nouvelle notation :

$$\mathbf{C} = \mathbf{GU} \tag{1.18}$$

avec

$$\mathbf{C} = [c_1(D),...,c_v(D)] \tag{1.19}$$

$$\mathbf{U} = [u_1(D),...,u_v(D)] \tag{1.20}$$

$$\mathbf{G} = \begin{bmatrix} g_{11}(D) & \cdots & \cdots & g_{1v}(D) \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ g_{v1}(D) & \cdot\cdot & \cdots & g_{vv}(D) \end{bmatrix} \tag{1.21}$$

1.5.3 Codes convolutionnels systématiques et récursifs, RSC

Le code est dit «systématique», lorsqu'un ensemble des v sorties reproduit fidèlement les k entrées. De même, un code convolutionnel est « récursif» lorsque les polynômes générateurs sont remplacés par les quotients de deux polynômes. Dans ce cas, une partie de la sortie est réinjectée dans les registres à décalage suivant les connexions fixées par les polynômes aux dénominateurs.

Il est possible de transformer un code convolutionnel non récursif, non systématique dont le rendement est $R = \frac{k}{k+1}$ en un code systématique récursif [2].

Pour un codeur de rendement $\frac{1}{2}$ par exemple qui est défini par deux polynômes générateurs $g_1(D)$ et $g_2(D)$ formant la matrice du codeur :

$$\mathbf{G} = [g_1(D) \quad g_2(D)] \tag{1.22}$$

On transformera ce code non récursif non systématique (NRNSC) en RSC en procédant à la modification suivante sur la matrice :

$$\mathbf{G} = \begin{bmatrix} 1 & \frac{g_2(D)}{g_1(D)} \end{bmatrix} \text{ ou } \boldsymbol{G} = \begin{bmatrix} 1 & \frac{g_1(D)}{g_2(D)} \end{bmatrix} \tag{1.23}$$

1.5.4 Représentation sous forme graphique du processus de codage

La représentation graphique d'un codeur convolutionnel s'inspire des caractéristiques markoviennes de la sortie du codeur. En effet, la sortie ne dépend que de l'état actuel des registres du codeur et des variables aléatoires en entrée. Il en résulte aussi l'état suivant du codeur. Ces graphes, équivalents à la représentation polynomiale, sont souvent plus faciles à manipuler pour atteindre des résultats intéressants.

Tout code convolutionnel est représentable par trois graphes équivalents, mais différents :

- l'arbre du code,
- le treillis du code,
- le diagramme d'états.

Nous allons présenter les définitions de ces trois modes de représentations pour le cas d'un code NRNSC (7,5). Il s'agit d'un codeur de taux R=1/2 dont les quatre états du codeur sont :

A=00
B=01
C=10
D=11

Ce sont dans les faits les valeurs que peuvent prendre les bits dans le registre à décalage.

- L'arbre est un graphe de hauteur et de largeur infinie. Un sommet dans l'arbre représente un état possible du codeur. Une arête symbolise une transition d'un état à un autre. L'arbre commence à son sommet par l'état zéro (registres à décalages initialisés à zéro). Chaque chemin dans l'arbre du code est une

séquence possible (un mot de code) à la sortie du codeur convolutionnel. La figure 1.6 montre la représentation en arbre du codeur NRNSC (7,5).

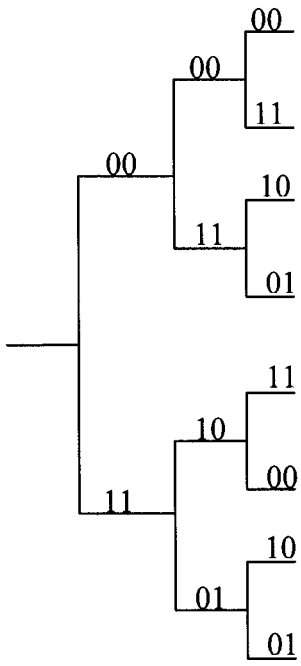

Figure 1. 6 Représentation en arbre d'un code convolutionnel NRNSC(7,5) de taux 1/2 et longueur de contrainte 3

- Le diagramme d'états est un graphe constitué de cercles représentant les différents états du codeur, reliés par des arcs orientés qui représentent les transitions entre les états. C'est un outil utile pour visualiser les états et les transitions, mais il pèche par son manque de dynamisme puisque l'information temporelle disparaît. Il est aussi limité à des codeurs avec un nombre d'états restreints. Pour le cas présenté auparavant, un tel diagramme est représenté à la figure 1.7.

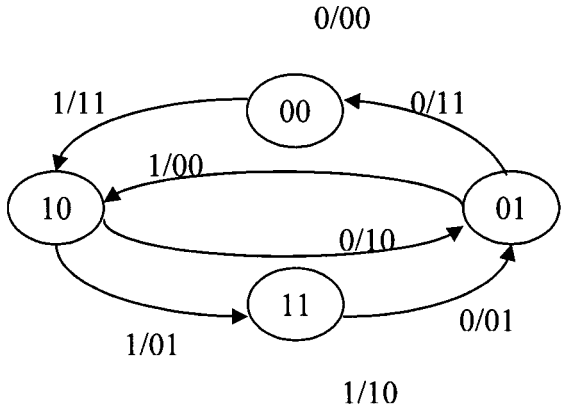

Figure 1. 7 Représentation du code NRNSC (7,5) par un diagramme d'états

- Le diagramme en treillis est l'outil graphique le plus performant pour caractériser un code et pour concevoir des algorithmes de décodage. Il est obtenu en repliant l'arbre sur sa largeur, par fusion des sommets représentant le même état au même instant. La représentation se fait sur deux axes, le premier, l'axe vertical représente les états et l'axe horizontal la variable temps. Les liens qui relient les états à l'instant t-1 et t, indiquent les transitions possibles. Dans l'exemple que l'on exploite, on peut trouver une représentation du diagramme en treillis à la figure 1.8.

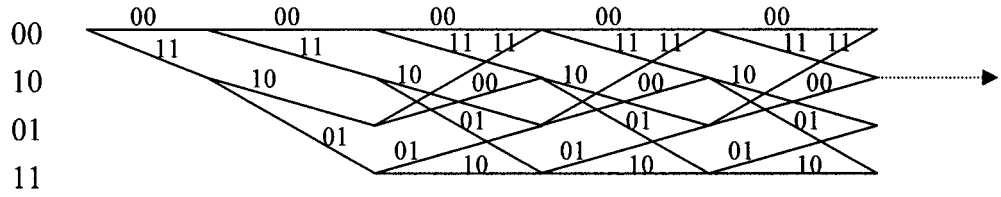

Figure 1. 8 Représentation du code NRNSC(7,5) par un treillis

1.6 Les codes turbo

L'apport le plus important dans le domaine des codes correcteurs d'erreurs de ces dernières années, a été sans conteste le développement des codes turbo [6]. Le principe de cette technique se base sur l'utilisation de deux codeurs convolutionnels, ou plus,

concaténés en parallèle. Les données à coder sont traitées par un codeur, alors que les autres codeurs codent une version entrelacée des données d'origine. Le décodage quant à lui est itératif et est constitué de deux décodeurs montés en série, entre lesquels on intercale un désentrelaceur. Ce processus itératif implique une rétroaction à l'image du système turbo en mécanique, d'où l'appellation [6].

1.6.1 La concaténation

Les codes concaténés furent introduits par Elias [3] et Forney [7] comme classe de codes puissants à grand pouvoir de correction. Les premiers décodeurs utilisés avaient une complexité faible grâce à l'utilisation d'un algorithme de décodage séquentiel à entrées/sorties fermes. L'avènement des codes turbo et leur décodage itératif à entrées/sorties souples (Soft Input/ Soft Output, SISO), a montré par la suite que des performances s'approchants de la limite de Shannon peuvent être atteintes [6].

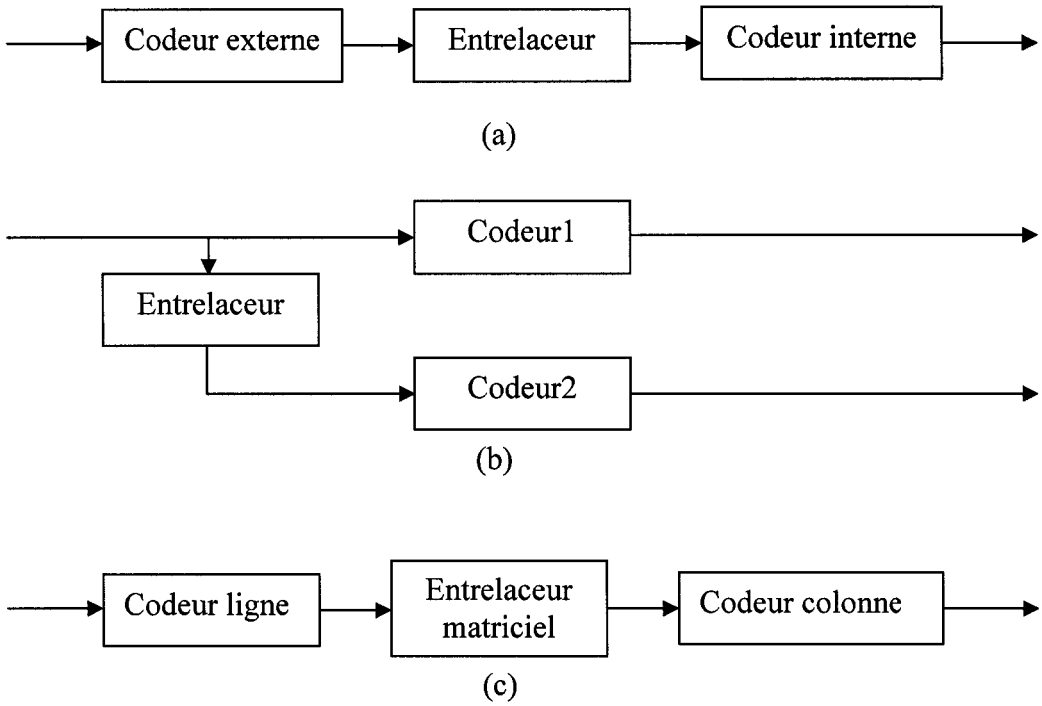

Figure 1. 9 Structures de concaténation de codeurs (a) en série, (b) en parallèle, (c) produit

Il existe deux structures possibles de concaténations de codes :

- Les codes concaténés en série, tel que représenté à la figure 1.9(a). Ils sont formés d'un codeur externe (Outer code) qui ajoute de la redondance aux k bits d'informations, le mot de code résultant est permuté par un entrelaceur avant d'être soumis à un codeur dit interne (Inner Code). Ce dernier ajoute ses propres bits de redondance et transmet l'information à l'élément suivant dans la chaîne de l'émetteur. Il est fréquent d'utiliser un codeur externe de type bloc et un codeur interne de type convolutionnel.
- Les codes concaténés en parallèle tel que montré sur la figure 1.9(b), sont formés de deux codeurs ou plus. L'un des codeurs opère sur la version initiale de l'information tandis que les autres utilisent une version permutée par un entrelaceur. C'est cette architecture qui a été retenue pour les codes turbo, avec deux codeurs convolutionnels systématiques et récursifs. Seul le premier codeur transmet l'information originale, les autres se contentant de transmettre la version codée.

On notera que dans la littérature, on définit une concaténation appelée codes produits (product codes), qui est une concaténation série comme on peut le voir sur la figure 1.9(c). Mais dans ce cas, le code interne est un code en bloc de type (n_C,k_C), le code externe est un code en bloc de type (n_L,k_L) et l'entrelaceur et un entrelaceur matriciel.

1.6.2 L'entrelacement

Le rôle de l'entrelacement et du désentrelacement est de décorréler les erreurs qui pourraient arriver dans un même temps en salve. Cela permet donc au récepteur de considérer les symboles codés subséquemment comme étant statistiquement indépendants. Une hypothèse de base est considérée dans l'utilisation de cette technique, le canal est à mémoire courte ou sans mémoire. L'entrelaceur doit aussi étaler les

symboles sur une longueur de quelques fois la contrainte du codeur [8], d'où l'augmentation de la latence.

On distingue trois types d'entrelaceurs :

- l'entrelaceur en bloc,
- l'entrelaceur convolutionnel,
- l'entrelaceur aléatoire.

1.7 Algorithmes de décodage

L'information qui parvient au récepteur est une version bruitée de celles émise originairement. Le rôle du décodeur de canal est donc d'estimer au mieux l'information que l'on a émise. Il serait ambitieux de vouloir traiter tous les algorithmes mis en place pour le décodage de canal, nous essayerons seulement de présenter les travaux les plus marquants afin de bien situer l'algorithme que l'on doit implanter [1].

Les symboles qui sont émis par le codeur subissent différents processus qui altèrent l'information au travers de toute la chaîne de communication. De ce fait, la séquence qui parvient au décodeur doit être traitée pour corriger et retrouver l'information initiale. Afin d'arriver à cela, plusieurs travaux ont été, et sont encore en cours, pour développer des algorithmes performants.

Les algorithmes de base peuvent être classés en deux catégories :

- les algorithmes de décodage probabilistes,
- les algorithmes de décodage algébriques.

C'est en 1963 que Massey dans son article [9] proposa l'algorithme à logique majoritaire, appelé aussi décodage seuil, et qui se classe dans la seconde catégorie. Ce travail étant la base du décodeur que l'on se propose d'implémenter, on trouvera dans le

chapitre suivant toute la présentation nécessaire à sa compréhension. On soulignera brièvement qu'il s'agit d'un décodeur sous-optimal, mais qui offre l'avantage de la simplicité de réalisation et d'un temps d'exécution connu.

Les algorithmes probabilistes se basent sur le calcul du maximum de vraisemblance à partir de la séquence de symboles reçue et détermine ainsi une estimation de l'information qui a été envoyée. Pour ce faire, on met en place un mécanisme de calcul d'une distance entre la séquence reçue à l'instant t et les séquences possibles à partir de l'état atteint à t-1. On établit alors un calcul de métrique cumulative que l'on cherche à optimiser. Ainsi par exemple pour un canal BSC on a recours à une distance de Hamming et on cherche à avoir le mot de code dont le chemin présente la distance cumulative minimale.

Deux techniques de décodages permettent la mise en place d'un décodage à maximum de vraisemblance. La première, dite algorithme de Viterbi ou aussi algorithme de décodage en treillis, a été proposée en 1967 par Viterbi [10] et c'est une approche qui tire profit de la structure en treillis pour réduire la complexité de l'évaluation du maximum de vraisemblance. C'est un algorithme qui donne d'excellents résultats mais qui ne peut s'appliquer qu'à des codes dont la longueur de contraintes est petite. En effet, le nombre de chemins possibles, et qui sont examinés à chaque intervalle, augmente de façon exponentielle avec la longueur de contrainte. Dans certaines applications, on trouve des décodeurs pour des cas où la contrainte des codeurs vaut au maximum neuf.

Le second type d'algorithmes probabilistes, ceux qui sont dits séquentiels, a été proposé en 1957 par Wozencraft [11] et il a fallu attendre 1963 pour que Fano [12] propose une procédure de décodage. Un autre algorithme dans la même catégorie est connu comme étant le «Stack algorithm» et il a été proposé par Zigangirov [13] et Jelinek [14]. La base du décodage cette fois-ci, est la structure en arbre des codes convolutionnels. Les deux

algorithmes ne sont pas trop différents quant à leur principe et ils souffrent tous les deux de la même limitation, à savoir leur effort de calcul qui est aléatoire. Des techniques usant de tampons à l'entrée et à la sortie du décodeur pour réguler la variabilité du temps de calcul ont été proposées. Néanmoins, ce désavantage a fait que les algorithmes séquentiels ne sont pas très utilisés dans la pratique.

Des approches visant à améliorer l'algorithme de Viterbi ont aussi été développée pour les codes convolutionnels concaténés en parallèles. On retiendra l'algorithme MAP, développé en 1972 par Bahl et al [15] et Mac Adam et al [16], qui améliore les performances de l'algorithme de Viterbi de quelques 3 dB.

Les performances que peuvent atteindre les différents algorithmes de décodage peuvent être améliorées par l'adoption d'un décodage à quantification souple (Soft-decision decoding). En effet, il est prouvé que l'on peut augmenter le gain de 2 dB par l'utilisation d'une quantification souple, comparée à une quantification dure, et ceci pour un niveau de quantification infini [2]. De même, il est démontré que l'utilisation d'une quantification sur huit niveaux équidistants, représenté sur trois bits, engendre une dégradation de 0.2 dB. De telle sorte que l'on atteint en fin de compte une amélioration de 1.8 dB si la quantification se fait sur huit niveaux.

Beaucoup d'algorithmes ont été adaptés pour pouvoir fonctionner avec une quantification souple, ainsi au lieu d'utiliser une distance de Hamming pour le décodeur de Viterbi, on utilise une distance Euclidienne.

1.8 Le décodage itératif

Le décodage itératif consiste à décoder un code par étapes successives à l'aide de décodeurs à faible complexité au lieu d'utiliser un seul décodeur complexe. Dans le cas

des codes turbo, tel que proposé par Berrou et al. [6], on procède à une concaténation en série de deux décodeurs tel qu'on peut le voir sur la figure 1.10.

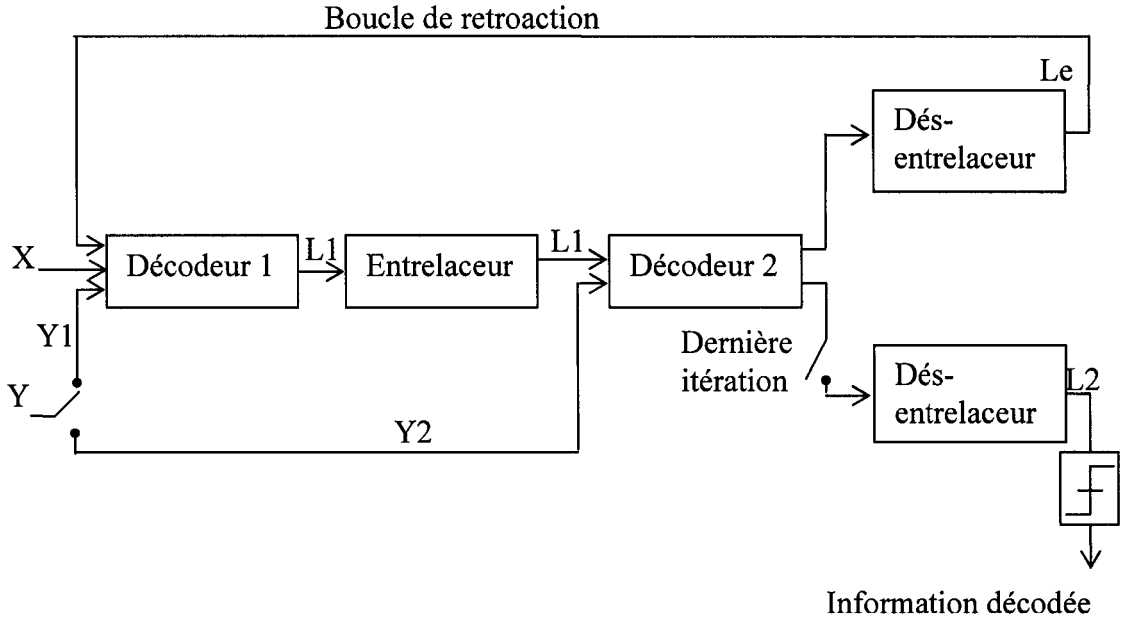

Figure 1. 10 Principe du décodage Turbo

Ainsi, le premier décodeur reçoit du canal la séquence des symboles systématiques et des symboles de parité délivrés par le premier codeur, en plus d'une rétroaction délivrée par le second décodeur. À partir de ces entrées, il génère pour chaque symbole décodé une valeur mesurant son degré de fiabilité appelée Probabilité A Posteriori (PAP). Le même entrelaceur utilisé au niveau du codeur, se charge de remettre en ordre les données délivrées par le premier décodeur pour qu'ils correspondent à la séquence de symboles de parité délivrés par le second codeur. Le second étage de décodage, génère des symboles décodés et les soumet à un désentrelaceur qui les réinjecte au premier décodeur. Grâce à cette boucle, le premier décodeur dispose d'une information supplémentaire qui l'aide à améliorer sa performance itération après itération. Au bout d'un certain nombre d'itérations, la sortie du second décodeur sera quantifiée puis remise en ordre par un désentrelaceur. Il est important de noter que compte tenu de la

nature du flux d'informations qui transite entre ces deux blocs, on doit utiliser un algorithme de décodage par symbole et non un algorithme de décodage par séquence.

Chang et Hancock [17] ont proposé un algorithme minimisant la probabilité par symbole pour canal à interférence inter-symbole (IIS), que l'on nomme Maximum A Posteriori (MAP). En 1972, l'algorithme MAP a été adapté par Bahl et al.[15] ainsi que Mc Adam et al. [16] aux codes correcteurs d'erreurs. Mais en raison de sa trop grande complexité, il a été longtemps ignoré malgré une supériorité de près de 0.3 dB à de faibles rapports signal à bruit et des taux d'erreurs par bits élevés. Une autre simplification a été proposée par Robertson [18] et reprise par Berrou et al. [6] dans les codes turbo. Il existe un second algorithme de décodage minimisant la probabilité d'erreur par symbole. Il s'agit d'une modification de l'algorithme de Viterbi, plus connue sous le nom de algorithme «Soft Output Viterbi Algorithm» (SOVA). On présente, dans ce qui suit, les deux algorithmes les plus utilisés dans le décodage des codes turbo.

1.8.1 Algorithme BCJR

L'algorithme BCJR a été nommé d'après ses auteurs [15], il permet un calcul exact du maximum de vraisemblance logarithmique, symbole par symbole, au sens du MAP pour une chaîne de Markov discrète. Compte tenu de la complexité et de la longueur de la démonstration mathématique à mettre en place pour aboutir à la formulation de l'algorithme, on réfèrera les personnes intéressées à la lecture de l'article de base. Il est possible de résumer les quatre étapes du décodage BJCR .

1. Précalculer les métriques de branche à partir des observations à priori et du canal.
2. Calcul récursif ascendant des métriques de branches (marche avant).
3. Calcul récursif descendant des métriques de branches (marche arrière).
4. Calcul des rapports de vraisemblance logarithmiques à posteriori bit à bit sur toutes les transitions dans la représentation en treillis du code.

Des applications de cet algorithme au décodage itératif ont été initialement proposées dans [15] et repris dans de nombreuses autres publications par la suite.

1.8.2 Algorithme SOVA

L'algorithme de Viterbi à sorties souples, ou pondérées, calcule de bonnes approximations des rapports de vraisemblance logarithmique bit à bit quand on dispose d'une représentation en treillis efficace. L'idée de base est d'associer à chaque décision ferme la fiabilité de cette décision dans l'algorithme de Viterbi de base. On résume les quatre étapes du décodage :

1. Précalculer les métriques de branche à partir des observations à priori et du canal.
2. Calcul récursif ascendant des métriques d'états cumulés et des fiabilités.
3. Remonter le chemin de vraisemblance maximale.
4. Calcul des rapports de vraisemblance logarithmique à posteriori bit à bit en trouvant le chemin concurrent de plus grande métrique.

Comparé à l'algorithme BCJR, l'algorithme SOVA produit des sorties pondérées avec une complexité moindre et une dégradation acceptable des performances en terme d'erreurs binaires.

1.9 Conclusion

Ce chapitre s'est voulu une introduction générale à la problématique du codage de canal, et s'est attaché à faire une présentation des codes convolutionnels sous différents aspects. Les modèles théoriques qui sont pris en compte dans le développement et l'étude des codes correcteurs d'erreurs sont aussi présentés. Par la suite, l'algorithme de décodage à seuil a été situé dans le domaine global des décodeurs pour codes convolutionnels. Ainsi, il apparaît clairement que le décodeur de Viterbi malgré ses performances, ne peut malheureusement pas être appliqué à des codes dont la longueur

de contrainte est élevée en raison du caractère exponentiellement croissant de sa complexité. Ceci limite l'utilisation de ce décodeur à des codes dont la contrainte est inférieure à dix. Or, plus la longueur de contrainte d'un code est élevée, meilleur est son pouvoir correcteur. Les décodeurs séquentiels de Fano et Stack, ont un effort de calcul aléatoire, ce qui les rend inaptes pour des missions ou le temps de réponse doit être connu et être aussi petit que possible. On a aussi présenté la révolution de l'heure que sont les codes turbo, leur principe et la problématique de la trop grande latence introduite au codage et au décodage par les entrelaceurs/désentrelaceurs.

Tous ces constats, ont amené Cardinal et al. à se pencher sur une nouvelle approche qui tirerait profit du décodeur à seuil avec le recourt à plus d'une itération de décodage comme dans le cas des codes turbo. Le chapitre suivant présentera l'aspect théorique de cette approche et les résultats atteints par cette étude.

CHAPITRE 2

BASES THÉORIQUES DU DÉCODEUR DE CODES DOUBLEMENT ORTHOGONAUX

2.1 Introduction

La mise en place de l'algorithme de décodage itératif pour les codes doublement orthogonaux nécessite la compréhension du cheminement qui a mené à cette contribution majeure. L'idée de base, comme on l'a déjà souligné, est de s'affranchir de la latence introduite par les entrelaceurs et désentrelaceurs présents dans les codes turbo, tout en préservant le caractère itératif du décodeur et l'indépendance statistique nécessaire entre les observables. En effet, certaines applications s'accommodent mal des délais qui peuvent s'introduire dans un transfert de données, d'où la nécessité de garder la taille des entrelaceurs dans des limites raisonnables, et de diminuer la contrainte k au niveau du codeur. Or, ceci diminue la capacité correctrice du code. En plus, les algorithmes utilisés dans les décodeurs turbo sont très complexes même en diminuant k.

L'algorithme proposé par Cardinal et al. [1] est basé sur l'algorithme de décodage à seuil. À l'inverse des autres algorithmes qui sont probabilistes, celui imaginé par Massey [9] se base sur une approche algébrique. En effet, on utilise un calcul de syndromes pour décoder, et on utilise une séquence de longueur m pour générer une décision sur le bit reçu. On notera que l'algorithme à seuil est classé comme un algorithme sous-optimal. Il engendre donc une perte en terme de gain de codage, mais c'est l'un des plus simples à implémenter du point de vue complexité matérielle.

Dans ce chapitre, on s'inspire de la thèse de C. Cardinal [1] pour mettre en place tout l'aspect théorique entourant le sujet de la réalisation. Néanmoins, le lecteur cherchant

plus de détails sur certains aspects du développement de ce décodeur est encouragé à se référer à la dite thèse.

Le principe du décodage à seuil en quantification ferme pour les codes convolutionnels est tout d'abord présenté. L'adaptation du décodeur pour le cas d'une quantification souple sera ensuite abordée avec l'utilisation d'une technique d'estimation de la probabilité à posteriori (APP) des bits décodés combinée à une approximation de cette APP connue comme l'approximation de la probabilité à posteriori (AAPP). La mise en place d'un décodage itératif mènera ensuite à considérer les conditions nécessaires pour garder l'indépendance des observables, ce qui permettra de considérer les codes convolutionnels doublements orthogonaux au sens large, puis au sens strict. Ce travail porte principalement sur cette seconde catégorie.

2.2 Décodage à seuil en quantification ferme pour les codes convolutionnels

En quantification ferme, le modèle du canal de communication est simplifié et se réduit à un canal BSC. Ce dernier est sans mémoire, et contient un modulateur BPSK, un canal physique et enfin un récepteur optimal qui donne l'information sur un bit. Le système tel qu'il nous intéresse se trouve illustré à la figure 2.1.

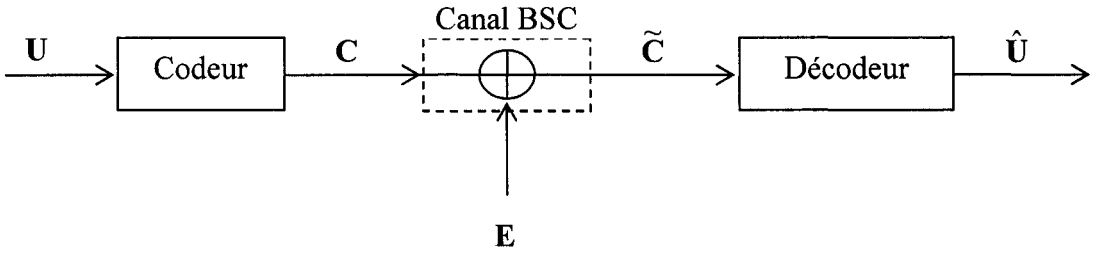

Figure 2. 1 Modèle simplifié d'un système de communication numérique avec canal BSC

L'information issue de la source se présente au codeur sous forme d'une séquence $\mathbf{U} = (u_0, u_1, ...)$ avec $u_i \in \{0,1\}$ et avec l'indice $i = 0,1,2,....$ représentant le temps.

Par simplification et sans perte de généralité, on considère que le codeur est de taux $R = \frac{1}{2}$. Ce dernier, fournit en sortie une séquence codée $\mathbf{C} = (\mathbf{c}_0, \mathbf{c}_1, ...)$ où chaque élément est formé du bit d'information u_i suivi d'un bit de parité p_i. Ainsi, $\mathbf{c}_i = (u_i, p_i)$ avec $i = 0,1,2,.....$ $\mathbf{P} = (p_0, p_1, ...)$ est la séquence de parité qui est fournie par le codeur au moyen des opérations modulo-2.

Les vecteurs de symboles $\mathbf{c}_i = (u_i, p_i)$ transitent par le canal où ils sont affectés par une séquence binaire $\mathbf{E} = (\mathbf{e}_0, \mathbf{e}_1, ...)$. Les vecteurs $\mathbf{e}_i = (e_i^u, e_i^p)$ affectent à la fois le bit d'information et le bit de parité respectivement, avec :

$$e_i^u = \begin{cases} 1 & \text{avec probabilité } \gamma_0 \\ 0 & \text{avec probabilité } (1-\gamma_0) \end{cases}$$

de plus

$$e_i^p = \begin{cases} 1 & \text{avec probabilité } \gamma_0 \\ 0 & \text{avec probabilité } (1-\gamma_0) \end{cases}$$

et γ_0 tel que décrit par (1.8).

En sortie du canal, on se retrouve avec une séquence $\widetilde{\mathbf{C}} = (\widetilde{\mathbf{c}}_0, \widetilde{\mathbf{c}}_1, ...)$. On exprime alors le vecteur $\widetilde{\mathbf{c}}_i$ comment suit

$$\widetilde{\mathbf{c}}_i = \mathbf{c}_i \oplus \mathbf{e}_i \tag{2.1}$$

on en déduit l'expression suivante :

$$\widetilde{\mathbf{c}}_i = (u_i \oplus e_i^u, p_i \oplus e_i^p) \tag{2.2}$$

En observant la séquence $\widetilde{\mathbf{C}}$, le décodeur effectue l'estimation des bits d'information transmis, que l'on note $\hat{\mathbf{U}} = (\hat{u}_0, \hat{u}_1, ...)$ avec $i = 0,1,2,.....$

2.2.1 Code orthogonal (CSOC)

Le but du codeur est d'associer à chaque bit d'information u_i un bit de parité p_i en utilisant un ensemble fini des bits d'information stockés dans ses registres. Supposant un taux de codage $R = \frac{1}{2}$, on spécifie ce code par un vecteur de connexions $\mathbf{g} = (g_0, g_1, ..., g_m)$ avec $g_k \in \{0,1\}$ et $g_k = 1$ si la k^{ième} cellule du codeur est connectée à l'additionneur modulo-2. La variable $g_k = 0$ s'il n'y a pas de connexion.

Pour les besoins de l'algorithme, on a recours à une seconde notation qui considère les J composantes non nulles du vecteur g, avec $J \leq m$ et $g_{\alpha_1}, g_{\alpha_2}, g_{\alpha_3}, ..., g_{\alpha_J}$, les éléments non nuls tel que $\alpha_1 < \alpha_2 < .... < \alpha_J$. Ainsi $\alpha_j = k$ signifie que le j^{ième} composant non nul du vecteur de connexion g traduit une connexion entre la cellule k et l'additionneur. On peut ainsi générer le bit de parité à l'instant i suivant la formule suivante :

$$p_i = \sum_{j=1}^{J} \oplus U_{i-\alpha_j} \tag{2.3}$$

On définit alors que pour qu'un code soit complètement orthogonal [1], il faut que les $\frac{J(J-1)}{2}$ différences positives

$$(\alpha_i - \alpha_j) \; i = 1,2,,...,J \text{ et } j = 1,2,,...,J \text{ et } j > i \tag{2.4}$$

soient distinctes entre elles. Dans l'exemple d'un codeur avec m=3 et J=3 dont on peut voir la réalisation sur la figure 2.2, on constate que l'on peut représenter le codeur par les vecteurs suivants :

$$\mathbf{g} = (1101)$$

ou encore

$$\boldsymbol{\alpha} = (0,1,3)$$

qui sont équivalents. On en déduit alors les différences positives suivantes :

$$(\alpha_2 - \alpha_1) = 1$$

$$(\alpha_3 - \alpha_1) = 3$$

$$(\alpha_3 - \alpha_2) = 2$$

Il s'agit bien d'un cas de code orthogonal.

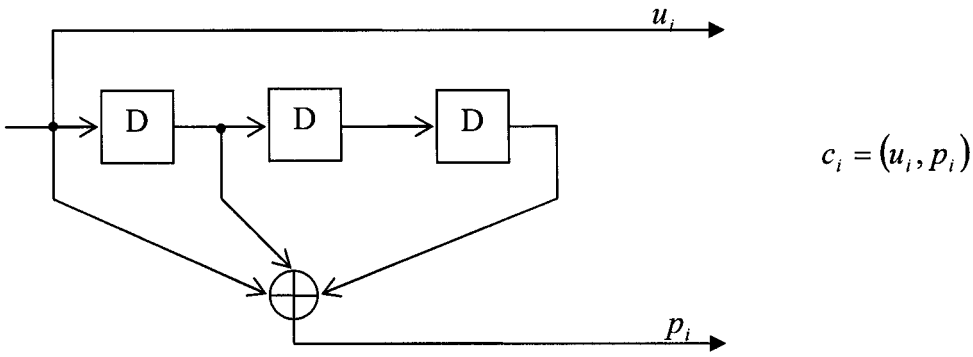

Figure 2. 2 Réalisation d'un codeur CSOC avec J=3, m=3 et $\boldsymbol{\alpha} = (\mathbf{0,1,3})$

2.2.2 Décodage des codes CSOC

L'algorithme de décodage à seuil est constitué d'une réplique du codeur qui réévalue les bits de parité p_i' à partir des bits d'informations reçus $\widetilde{u}_i$. On procède ensuite à une comparaison entre les bits de parités ainsi générés p_i' et les bits de parité reçus $\widetilde{p}_i$ par le biais d'un additionneur modulo-2. Les résultats de la comparaison forment les syndromes S_i, que l'on stocke dans un registre de longueur α_J. En combinant ces bits de syndrome suivant une logique de majorité, on estime le symbole d'erreur $\hat{e}_i^u$ qui a perturbé le bit d'information courant u_i. On obtient une estimation de ce dernier en procédant à l'opération suivante :

$$\hat{u}_i = \widetilde{u}_i \oplus \hat{e}_i^u \tag{2.5}$$

Ainsi si l'on considère un codeur de taux de codage R=$\frac{1}{2}$ avec une mémoire $m = \alpha_J$, le schéma de la figure 2.3 décrit le montage pour le calcul des syndromes.

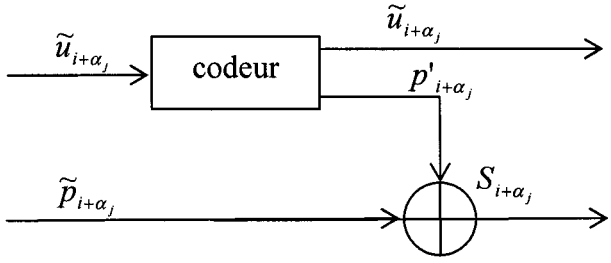

Figure 2. 3 Montage de calcul des syndromes

On est alors en mesure de calculer la valeur du syndrome se trouvant dans la cellule k comme suit :

$$S_{i+k} = \widetilde{p}_{i+k} \oplus \sum_{j=1}^{J} \oplus \widetilde{u}_{i+k-\alpha_j};\ \mathrm{k} = 0,1,...,\mathrm{m} \tag{2.6}$$

d'où en utilisant (2.5) on obtient :

$$S_{i+k} = e^{p}_{i+k} \oplus \sum_{j=1}^{J} \oplus e^{u}_{i+k-\alpha_j};\ \mathrm{k} = 0,1,...,\mathrm{m} \tag{2.7}$$

On remarque qu'il n'y a que les symboles d'erreurs qui interviennent dans cette formule, ce qui implique que la valeur de S_i ne dépend que des erreurs introduites par le canal.

On utilise à ce niveau, un ensemble de J équations de parité $\{A_{i,1}, A_{i,2},..., A_{i,J}\}$ défini à l'instant i par la formule [19] :

$$A_{k,i} \equiv S_{i+\alpha_k} = e^{p}_{i+\alpha_k} \oplus e^{u}_{i} \oplus \sum_{j=1}^{k-1} \oplus e^{u}_{i+\alpha_k-\alpha_j} \oplus \sum_{j=k+1}^{J} \oplus e^{u}_{i+\alpha_k-\alpha_j} \tag{2.8}$$
$$k = 1,2,...,J;\ i = 0,1,2,...$$

Cette équation est dite équation de contrôle de parité [19].

Pour que cet ensemble soit orthogonal au symbole d'erreur e^{u}_{i}, il faut que dans chaque équation ce symbole soit inclus et que les autres symboles d'erreurs n'apparaissent qu'une seule fois dans l'ensemble des J équations. On obtient l'estimation par une logique majoritaire qui s'énonce comme suit :

Choisir $\hat{e}_i^u = 1$ *si et seulement si plus de* $\left\lfloor \frac{J}{2} \right\rfloor$[1] *équations de parités* $A_{k,i}$ $k = 1,2,...,J$ *orthogonales à* e_i^u *sont égales à la valeur 1, sinon choisir* $\hat{e}_i^u = 0$.

Une amélioration est apportée aux équations de parités quand on considère qu'à l'instant *i*, la valeur de l'estimation $\hat{e}^u_{i+\alpha_k-\alpha_j}$ est connue car $(\alpha_k - \alpha_j) < 0$.

Ceci amène à introduire cette valeur dans les $A_{k,i}$ pour limiter l'effet des symboles d'erreurs $e^u_{i+\alpha_k-\alpha_j}$ avec $\alpha_k - \alpha_j < 0$ et donc améliorer la probabilité d'erreur [19], [9]. Ce décodage est alors dit un décodage avec rétroaction (Feedback decoding). L'équation (2.8) devient alors; pour *k*=1,2,…,3 et *i*=0,1,2,…

$$A_{k,i} = e^p_{i+\alpha_k} \oplus e^u_i \oplus \sum_{j=1}^{k-1} \oplus e^u_{i+\alpha_k-\alpha_j} \oplus \sum_{j=k+1}^{J} \oplus (e^u_{i+\alpha_k-\alpha_j} \oplus \hat{e}^u_{i+\alpha_k-\alpha_j}) \tag{2.9}$$

où

$$e^u_{i+\alpha_k-\alpha_j} = \hat{e}^u_{i+\alpha_k-\alpha_j} \tag{2.10}$$

quand la rétroaction est idéale. Dans ce cas de figure, les J équations de parités sont réduites à l'expression :

$$A_{k,i} = e^p_{i+\alpha_k} \oplus e^u_i \oplus \sum_{j=1}^{k-1} \oplus e^u_{i+\alpha_k-\alpha_j} \quad k = 1,2,...,J \;\; i = 0,1,2,... \tag{2.11}$$

Il est alors facile de remarquer que ces équations sont bien orthogonales à e_i^u. L'algorithme utilise alors une nouvelle règle de décision qui s'énonce :

Choisir $\hat{e}_i^u = 1$ *si et seulement si*

[1] $\lfloor * \rfloor$ Représente la partie entière

$$\sum_{k=1}^{J} A_{k,i} > T \quad i = 0,1,2,...$$

$$Choisir\ \hat{e}_i^u = 0\ \ sinon.$$

Avec T le seuil de décision qui vaut $\left\lfloor \frac{J}{2} \right\rfloor$ pour une décision optimale.

La figure 2.4 décrit le schéma de réalisation d'un tel décodeur à seuil en quantification ferme. Un exemple pour un codeur de longueur de contrainte k=2 et de taux $R = \frac{1}{2}$ est présenté à la figure 2.5 avec l'ensemble codeur, canal BSC et décodeur.

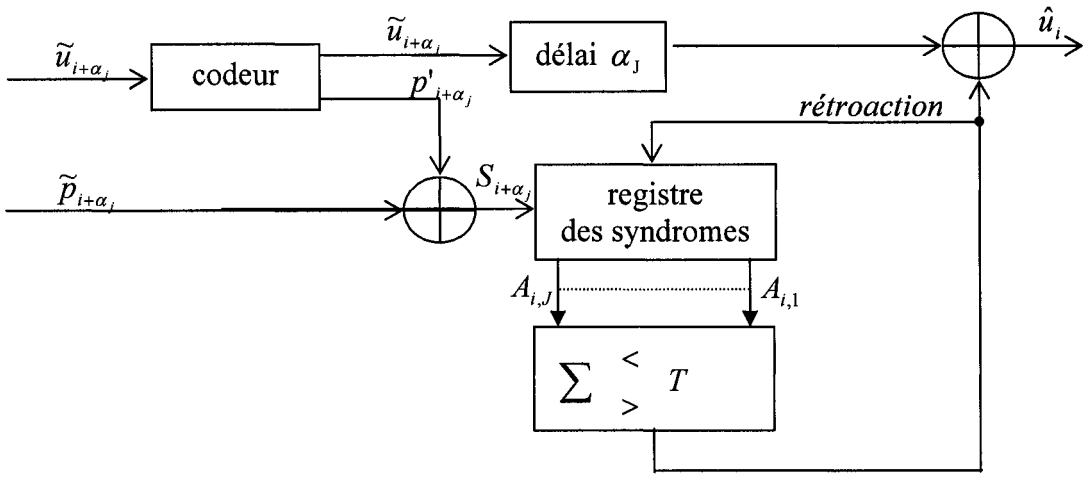

Figure 2. 4 Décodeur à seuil en quantification ferme avec rétroaction

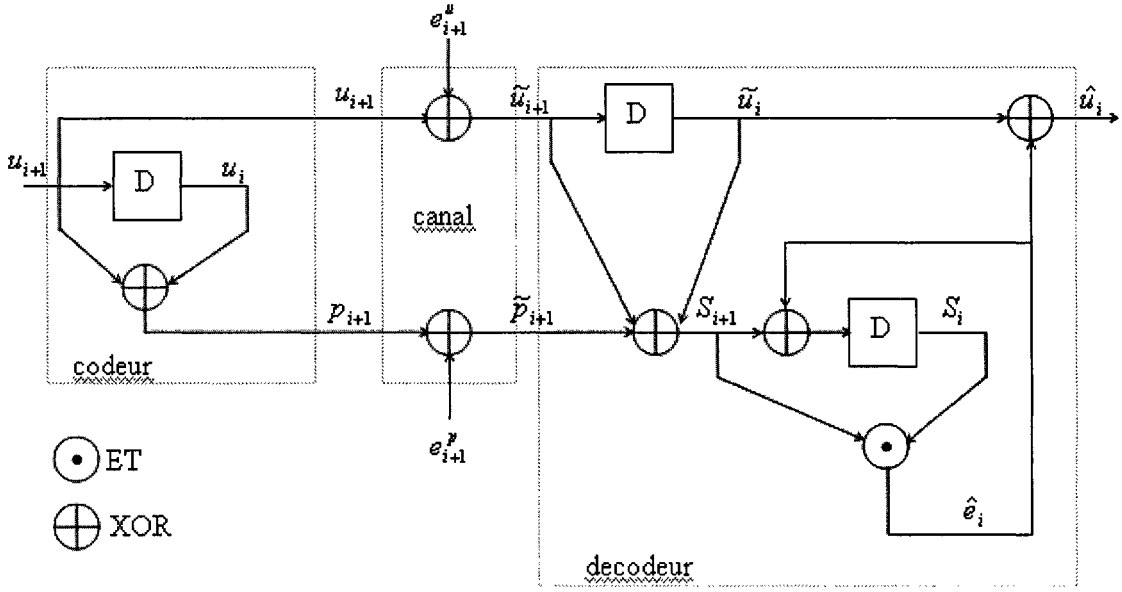

Figure 2. 5 Ensemble codeur, canal BSC et décodeur à seuil pour un code k=2 et R=1/2

2.3 Adaptation du décodage à seuil à des sorties non quantifiées

L'algorithme de décodage à seuil vu jusqu'à maintenant et tel que présenté par Massey [9], est limité au décodeur agissant sur des données en quantification ferme, avec comme conséquence, une perte sur le gain potentiel de codage. L'utilisation d'une entrée pondérée, nécessite une modification du processus. Une approche proposée par Wu [22] [23] permet d'arriver à nos fins par le biais de l'estimation d'une probabilité à posteriori (APP), qui donne une mesure de la probabilité de l'exactitude de la décision portée sur le bit courant. Cette approche améliore nettement le décodage à seuil, mais est handicapée par l'utilisation d'opérateurs non linéaires difficilement implantables au niveau matériel et qui augmentent significativement la complexité du circuit à utiliser.

Une alternative a été proposée par Tanaka et al. [24] et une architecture proposée par Lavoie et al [25] puis simplifiée par Gagnon et al. [26], en utilisant une approximation de la probabilité à posteriori (AAPP). Ceci est fait en remplaçant les opérateurs non linéaires par des opérateurs plus simples, au prix d'une dégradation des performances.

Afin de mettre en place les équations de contrôle nécessaires à notre réalisation, nous devons tout d'abord revoir le modèle du canal de transmission qui ne peut plus être considéré comme BSC, car c'est un canal non quantifié dont on a besoin. Les figures 2.6 et 2.7 présentent notre nouveau model.

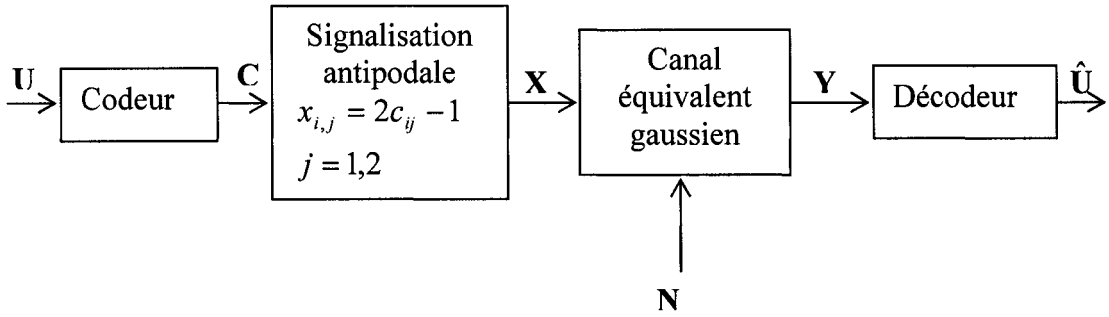

Figure 2. 6 Modèle d'un système de communication pour un canal non quantifié

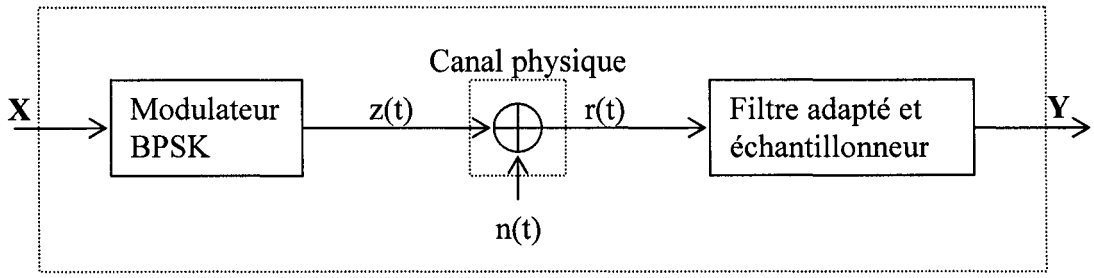

Figure 2. 7 Le canal équivalent gaussien

Les symboles C qui sont issus du codeur sont transformés en une séquence $\mathbf{X} = (\mathbf{x}_0, \mathbf{x}_1, ...)$ de symboles antipodaux où $\mathbf{x}_i = \begin{pmatrix} x_i^u \\ x_i^p \end{pmatrix} = \begin{pmatrix} 2u_i - 1 \\ 2p_i - 1 \end{pmatrix}$.

La séquence obtenue est ensuite modulée BPSK pour fournir un signal $z(t)$ qui est envoyé dans le canal. Ce dernier est du type AWGN avec un spectre de densité de puissance bilatérale $\frac{N_0}{2}(W/Hz)$.

En sortie du canal, le signal $r(t)$ est exprimé par l'équation :

$$r(t) = z(t) + n(t) \tag{2.12}$$

On procède ensuite à une démodulation à l'aide d'un filtre adapté qui fournit au décodeur la séquence $\mathbf{Y} = (\mathbf{y}_0, \mathbf{y}_1, ...)$ avec $\mathbf{y}_0, \mathbf{y}_1, ...$ des variables aléatoires gaussiennes qui s'écrivent :

$$\mathbf{y_i} = \begin{pmatrix} y_i^u \\ y_i^p \end{pmatrix} = \begin{pmatrix} x_i^u + n_i^u \\ x_i^p + n_i^p \end{pmatrix} \tag{2.13}$$

Cette séquence est injectée dans le décodeur qui génère en sortie une séquence d'informations décodées $\hat{\mathbf{U}} = (\hat{\mathbf{u}}_0, \hat{\mathbf{u}}_1, ...)$.

Une seconde modification s'applique à la structure du décodeur à seuil. La structure vue dans la quantification ferme est dite de type I et l'on y utilise J équations de parité $A_{j,i}$ $j = 1,....,J$ et $i = 0,1,2,...$ pour obtenir une décision sur le symbole d'erreur courant e_i^u. Or, dans ce qui suit, on considérera une structure du même algorithme, dite de type II et qui fournit une estimation directe du bit décodé $\hat{u}_i$. Cette variation fait introduire une règle de décision qui est basée, cette fois ci, sur J+1 équations $B_{j,i}$ $j = 0,....,J$ et $i = 0,1,2,...$ que l'on exprime dans les formules suivantes :

$$\begin{aligned} B_{0,i} &= \widetilde{u}_i \\ B_{j,i} &= A_{i,j} \oplus \widetilde{u}_i \end{aligned} \tag{2.14}$$

Il s'agit en fait d'équations de contrôle basées sur une «inversion de parité» [1]. En développant cette présentation mathématique, on aboutit à l'expression :

$$B_{j,i} = \widetilde{p}_{i+\alpha_j} \oplus \sum_{k=1}^{j-1} \oplus \widetilde{u}_{i+\alpha_j-\alpha_k} \oplus \sum_{k=j+1}^{J} \oplus \widetilde{u}_{i+\alpha_j-\alpha_k} \tag{2.15}$$

or, comme $\widetilde{u}_i = u_i \oplus e_i^u$ on obtient :

$$B_{j,i} = u_i \oplus \sum_{k=1}^{j-1} \oplus e^u_{i+\alpha_j-\alpha_k} \oplus e^p_{i+\alpha_j} \oplus \sum_{k=j+1}^{J} \oplus \left(e^u_{i+\alpha_j-\alpha_k} \oplus \hat{e}^u_{i+\alpha_j-\alpha_k} \right) \qquad (2.16)$$
$$j = 1,2,...,J \text{ et } i = 0,1,2,...$$

et

$$B_{0,i} = \widetilde{u}_i = u_i \oplus e_i^u \qquad (2.17)$$

On remarque que u_i à l'instant i est présent dans les J+1 équations $B_{j,i}$ et que chaque symbole d'erreur n'apparaît qu'une seule fois dans l'ensemble des J+1 équations $B_{j,i}$.

Le but de l'algorithme est de minimiser la probabilité d'erreur moyenne, en sortie du décodeur. Pour cela, on cherche à avoir la valeur ξ du bit d'information u_i qui maximise :

$$\Pr\left(u_i = \xi / \{B_{j,i}\}\right) \quad i = 0,1,2,... \text{ et } j = 0,1,2,...,J \qquad (2.18)$$

La règle de décision devient alors :

Choisir $\hat{u}_i = 1$, *si et seulement si*

$$\Pr\left(u_i = 1/\{B_{j,i}\}\right) \geq \Pr\left(u_i = 0/\{B_{j,i}\}\right)$$

$\hat{u}_i = 0$ *si non.*

En introduisant le logarithme sur les deux parties de l'inéquation, et en procédant à une manipulation de la formule, la règle s'écrit :

Choisir $\hat{u}_i = 1$, *si et seulement si*

$$L\left(u_i / \{B_{j,i}\}\right) = \ln\left(\frac{\Pr\left(u_i = 1/\{B_{j,i}\}\right)}{\Pr\left(u_i = 0/\{B_{j,i}\}\right)} \right) \geq 0 \qquad (2.19)$$

$\hat{u}_i = 0$ *si non.*

Avec $L(u_i / \{B_{j,i}\})$ appelé le «logarithme du rapport de vraisemblance» (LRV) de u_i conditionné à l'ensemble $\{B_{j,i}\}$. La valeur absolue du LRV constitue la fiabilité de la décision prise.

Étant donné l'orthogonalité des équations d'inversion de parité, les symboles d'erreurs qui forment les équations $B_{j,i}$ sont indépendants.

On peut alors utiliser le théorème de Bayes pour l'appliquer à (2.19) qui devient alors :

$$L(u_i / \{B_{j,i}\}) = \sum_{j=1}^{J} \ln\left(\frac{\Pr(B_{j,i} / u_i = 1)}{\Pr(B_{j,i} / u_i = 0)}\right) + \ln\left(\frac{\Pr(B_{0,i} / u_i = 1)}{\Pr(B_{0,i} / u_i = 0)}\right) + \ln\left(\frac{\Pr(u_i = 1)}{\Pr(u_i = 0)}\right) \quad (2.20)$$

Le terme $\ln\left(\frac{\Pr(u_i = 1)}{\Pr(u_i = 0)}\right)$ dénoté $L(u_i)$ représente l'information à priori qui vaut en général zéro puisqu'il est équiprobable de transmettre un 1 ou un 0.

Moyennant certaines remarques sur les équations ainsi que des développements mathématiques que l'on retrouvera dans [1] [22] [23], et suite à l'application de la simplification [24] apportée sur des opérateurs non linéaires, on peut exprimer le LRV de la manière suivante :

$$\begin{aligned} L(u_i / \{B_{j,i}\}) \approx \lambda_i &= y_i^u + \sum_{j=1}^{J} \left(y_{i+\alpha_j}^p \Diamond \sum_{k=1}^{j-1} \Diamond y_{i+\alpha_j-\alpha_k}^u \Diamond \sum_{k=j+1}^{J} \Diamond \lambda_{i+\alpha_j-\alpha_k} \right) \\ &= y_i^u + \sum_{j=1}^{J} \psi_{j,i} \end{aligned} \quad (2.21)$$

avec comme approximation que le terme «à priori» de u_i, $L(u_i)$ est nul.

Le terme $\psi_{j,i}$ est la valeur approximative de chaque équation $B_{j,i}$. L'opérateur «$\Diamond$» introduit pour simplifier la complexité de l'algorithme est dit opérateur addmin et on l'exprime comme suit :

$$L(\zeta_1) \lozenge L(\zeta_2) = -Sign(L(\zeta_1))Sign(L(\zeta_2))\min\left[|L(\zeta_1)|,|L(\zeta_2)|\right] \tag{2.22}$$

avec ζ_1 et ζ_2 deux variables aléatoires binaires indépendantes.

Dans le cas où l'on aurait plus d'une variable aléatoire indépendante, $\zeta_1, \zeta_2, \zeta_3, ..., \zeta_N$, la formule précédente est généralisée et devient:

$$\sum_{i=1}^{N} \lozenge L(\xi_i) = (-1)^{N+1} \prod_{i=1}^{N} sign(L(\xi_i)) \min_{1 \le i \le N} \left(|L(\xi_i)|\right) \tag{2.23}$$

En fin de compte, pour un codeur CSOC avec J=4 tel que l'on peut le voir sur la figure 2.8, le décodeur à seuil à sortie pondérée avec rétroaction et opérateur addmin est schématisé dans la figure 2.9 (inspirée de [1]).

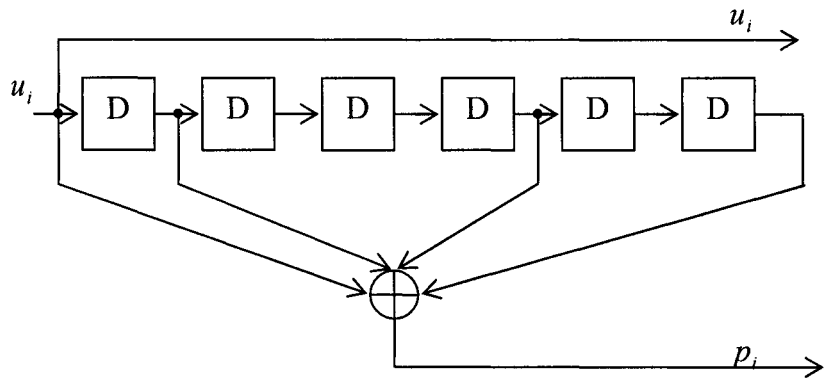

Figure 2. 8 Codeur CSOC avec J=4

La figure représente un codeur décrit par le vecteur suivant **g** = (1100101)

Soit **α** = (0,1,4,6) et donc $m = \alpha_4 = 6$.

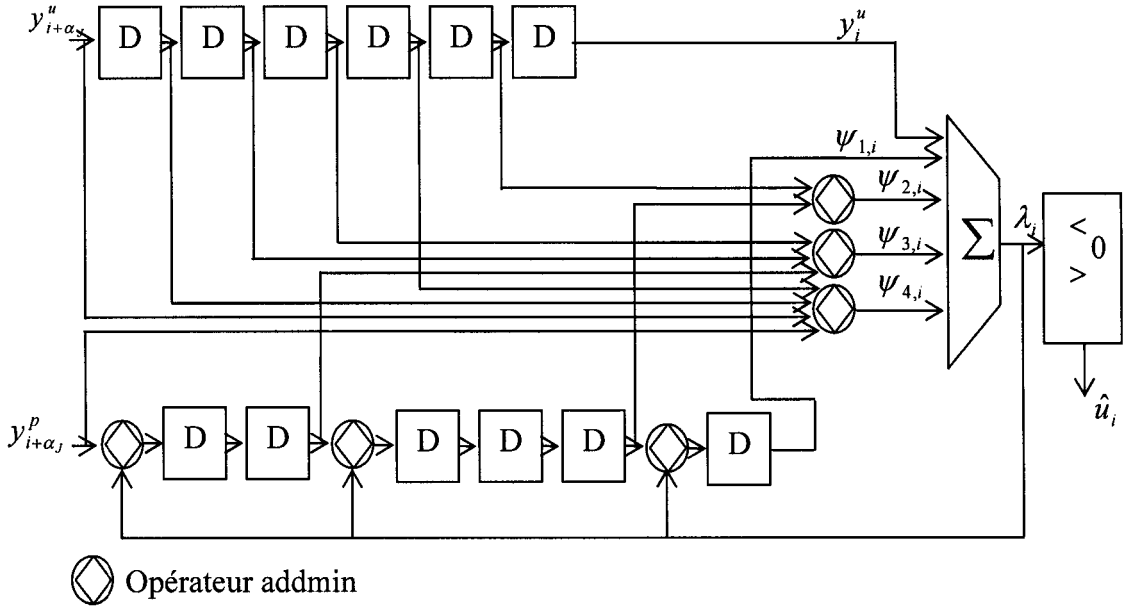

Figure 2. 9 Décodeur à seuil a entrée non-quantifiée avec rétroaction et opérateur addmin

2.4 Décodage itératif, définition des codes convolutionnels doublement orthogonaux au sens large CSO²C-WS

2.4.1 Le processus itératif

Le formalisme présenté met en place un décodeur à seuil à sortie pondérée. Ce décodeur, délivre en sortie une valeur réelle qui correspond au LRV du bit d'information u_i conditionné sur un ensemble fini d'observables à la sortie du canal. On peut donc déduire, que c'est en fait une version sous optimale de l'algorithme MAP (Maximum à posteriori) qui détermine aussi une valeur réelle correspondant au LRV $L(\hat{u}_i) = L(u_i / Y)$. Dans le cas d'un code systématique, on écrit cette valeur sous la forme suivante [23] :

$$L(\hat{u}_i) = L_e(\hat{u}_i) + L_c y_i^u + L(u_i) \tag{2.24}$$

avec

$L(u_i)$ valeur à priori lorsque disponible,

$L_c y_i^u$ valeur provenant du canal,

$L_e(\hat{u}_i)$ valeur extrinsèque.

De la même façon, on peut écrire l'équation du LRV dans le cas du décodage à seuil sous la même forme, ce qui donne :

$$L(\hat{u}_i) = L(u_i / \{B_{j,i}\}) = \tilde{L}_e(\hat{u}_i) + L_c y_i^u + L(u_i) \tag{2.25}$$

Compte tenu des structures similaires des deux décodeurs, il est possible d'appliquer au décodage à seuil un processus itératif [1]. L'idée maîtresse est d'utiliser l'information «à priori» comme étant nulle dans la première itération, et d'utiliser l'information extrinsèque de l'itération n-1 en lieu et place de l'information «à priori» dans l'étape n en plus de l'information issue du canal. On répétera ainsi le processus jusqu'à atteindre une fiabilité jugée acceptable.

La réalisation du processus itératif est assez aisée, puisqu'on n'a qu'à reprendre l'équation (2.21) et remarquer qu'à l'instant *n*, la valeur extrinsèque s'exprime sous la forme :

$$L_e^{(n)}(u_i) = \sum_{j=1}^{J} \psi_{j,i}^{(n)} \tag{2.26}$$

qui combine la sortie du décodeur à l'itération (n-1), $\lambda_i^{(n-1)}$, avec des symboles de parité provenant du canal. Cette valeur extrinsèque sera combinée à l'instant n avec le symbole d'information reçu y_i^u pour donner lieu à une valeur non quantifiée $\lambda_i^{(n)}$.

L'équation de décodage en fonction des itérations s'écrit alors :

$$\lambda_i^{(n)} = y_i^u + \sum_{j=1}^{J} \left(y_{i+\alpha_j}^p \Diamond \sum_{k=1}^{j-1} \Diamond \lambda_{i+\alpha_j-\alpha_k}^{(n-1)} \Diamond \sum_{k=j+1}^{J} \Diamond \lambda_{i+\alpha_j-\alpha_k}^{(n)} \right) \tag{2.27}$$

Le schéma du décodeur peut alors être présenté par une succession de décodeurs comme on peut le voir à la figure 2.10.

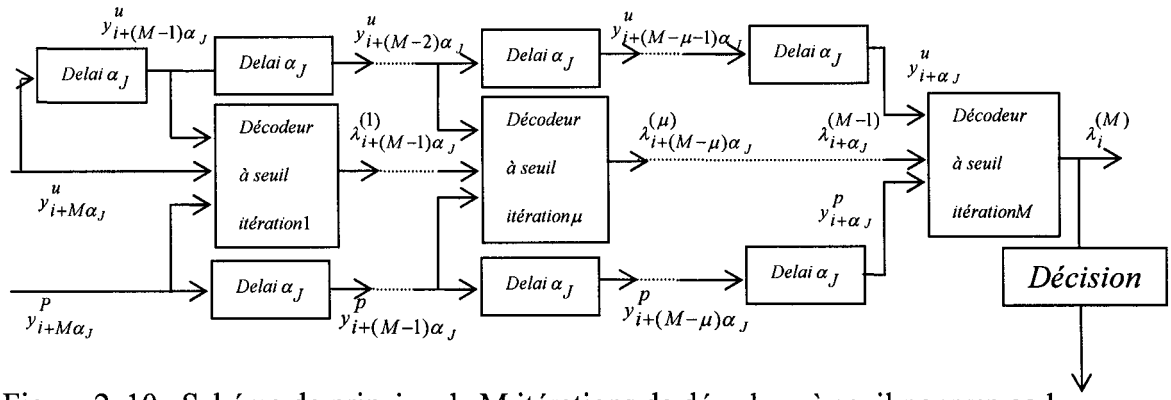

Figure 2. 10 Schéma de principe de M itérations de décodage à seuil pour un code CSO^2C-WS

2.4.2 Les conditions de double orthogonalité

L'intérêt des décodeurs turbo réside dans leur processus de décodage itératif qui a pour condition importante l'indépendance des observables entre les décodeurs qui se succèdent. Pour atteindre leur but, Berrou et al. [6] ont utilisé des entrelaceurs/désentrelaceurs au niveau des codeurs et des décodeurs pour briser toute dépendance qui pourrait survenir. Or, pour diminuer la probabilité d'erreur, il est prouvé [20] [21] qu'il faut utiliser une procédure d'entrelacement adéquate et des entrelaceurs ayant des grandes tailles.

La structure des codes orthogonaux vus jusqu'à maintenant n'assure une indépendance des observables que pour la première itération. Il est donc nécessaire de trouver des codes ayant des propriétés supplémentaires.

Les codes convolutionnels doublement orthogonaux (Convolutional Self Doubly Orthogonal Code) CSO^2C, fournissent une telle structure assurant l'indépendance sur au moins deux itérations [1].

Afin de dégager les règles auxquelles doivent satisfaire ces codes pour être considérés comme doublement orthogonaux, on procédera d'abord à l'écriture de la formule des équations de contrôle pour les deux premières itérations. On obtient donc l'équation (2.28):

$$\lambda_i^{(2)} = y_i^u + \sum_{j=1}^{J} \psi_{j,i}^{(2)} = y_i^u + L_e^{(2)}(\hat{u}_i)$$

$$= y_i^u + \sum_{j=1}^{J} y_{i+\alpha_j}^p \Diamond \sum_{k=j+1}^{J} \Diamond \lambda_{i+\alpha_j-\alpha_k}^{(2)} \Diamond \sum_{k=1}^{j-1} \Diamond \left(\begin{array}{l} y_{i+\alpha_j-\alpha_k}^u + \\ \sum_{\substack{l=1 \\ l\neq k}}^{J} \left(y_{i+\alpha_j-\alpha_k+\alpha_l}^p \Diamond \sum_{\substack{q=1 \\ q\neq j}}^{l-1} \Diamond y_{i,(\alpha_j-\alpha_k)-(\alpha_q-\alpha_l)}^u \Diamond \sum_{q=l+1}^{J} \Diamond \lambda_{i+(\alpha_j-\alpha_k)-(\alpha_q-\alpha_l)}^{(1)} \right) \end{array} \right)$$

On définit donc un code de taux $R = \frac{1}{2}$ comme étant doublement orthogonal si les trois conditions suivantes sont satisfaites [1] :

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 des différences sont distinctes.

Ceci pour toutes les combinaisons des indices (j,k,l,q) avec $j \neq k, \mathrm{l} \neq \mathrm{q}, \mathrm{l} \neq \mathrm{k}, \mathrm{q} \neq \mathrm{j}$.

2.4.3 Les codes convolutionnels doublement orthogonaux au sens large

Les règles établies pour définir un code doublement orthogonal laissent voir que s'il est facile de se conformer aux deux premières contraintes, il est impossible d'avoir des différences des différences $(\alpha_j - \alpha_k)$-$(\alpha_q - \alpha_l)$ qui soient totalement distinctes avec la structure que l'on utilise. En effet, on remarque que des permutations des indices k et q donneront une égalité $(\alpha_j - \alpha_k)$-$(\alpha_q - \alpha_l)$=$(\alpha_j - \alpha_q)$-$(\alpha_k - \alpha_l)$. En outre, on ne peut éliminer les répétitions d'observables du processus d'itération sans engendrer possiblement une propagation d'erreurs.

Cardinal et al. ont été amenés à définir les codes convolutionnels doublement orthogonaux au sens large, CSO^{2}C-WS (Convolutional Self Doubly Orthogonal Codes in the Wide Sense); où l'on garde les répétitions inévitables d'observables à la deuxième itération du processus de décodage. La définition pour les CSO^{2}C-WS est obtenue à partir de celle des CSO^{2}C et s'exprime comme suit :

> *Un code convolutionnel de taux de codage* $R = \frac{1}{2}$ *est doublement orthogonal au sens large, si et seulement si les indices de position* $\{\alpha_j\}$ *sont tel que :*
>
> $(\alpha_j - \alpha_k)$-$(\alpha_q - \alpha_l)$ *sont distincts pour (j,k,l,q) avec* $j \neq k, l \neq q, l \neq k, q \neq l$
>
> *Sauf pour les permutations inévitables.*

On notera que cette définition inclue aussi les conditions 1 et 2 de la définition précédente, puisque la condition 3 implique les deux autres [1].

Pour améliorer le rendement du décodeur, Cardinal et al. ont aussi proposé d'utiliser des coefficients qui donnent une pondération à chaque équation $\psi_{j,i}^{(n)}$, et des valeurs ont été proposées suite à des simulations. Il a été aussi proposé d'utiliser des réseaux de neurones pour atteindre le même objectif, puisque l'opération s'apparente à un problème que l'on peut résoudre par cette approche. Cette approche n'a pas donné de bons résultats [1].

Il n'en demeure que l'implémentation de telles techniques aurait été complexe et nécessiterait un grand effort de développement au niveau matériel. En raison de ces contraintes, une seconde structure plus intéressante a été proposée.

2.5 Les codes convolutionnels doublement orthogonaux au sens strict CSO^{2}C-SS

La structure proposée pour se débarrasser du problème des répétitions indésirables d'observables est une structure parallèle dans laquelle chaque symbole d'information est connecté à un seul symbole de parité d'une séquence de parité. Dès lors, aucune permutation d'indices n'est possible, et on parle alors d'un code code convolutionnel doublement orthogonal au sens strict, CSO^{2}C-SS (Convolutional Self Doubly Orthogonal Codes in the Strict Sense). On possède alors un taux de codage $R = \frac{J}{2J}$ et le code ne se trouve plus définit par un vecteur de longueur J, mais par une matrice α de dimension $J \times J$. Les éléments de la matrice, $\alpha_{j,n}$ représentent la position où l'on va avoir connexion entre le j^{ième} registre à décalage du j^{ième} symbole d'information, vers l'additionneur modulo-2 pour former le n^{ième} bit de parité. Le processus génère les bits de parité suivant l'équation :

$$p_{n,i} = \sum_{k=1}^{J} \oplus u_{k,i-\alpha_{k,n}} \quad n = 1,2,...,J;\ i = 0,1,2,... \tag{2.29}$$

Ainsi, pour un code de taux $R = \frac{3}{6}$ donc J=3, notre codeur est défini par la matrice $\boldsymbol{\alpha}$ qui s'écrit :

$$\begin{bmatrix} 0 & 0 & 5 \\ 1 & 3 & 0 \\ 5 & 2 & 0 \end{bmatrix}$$

On représente alors le codeur par la structure de la figure 2.11.

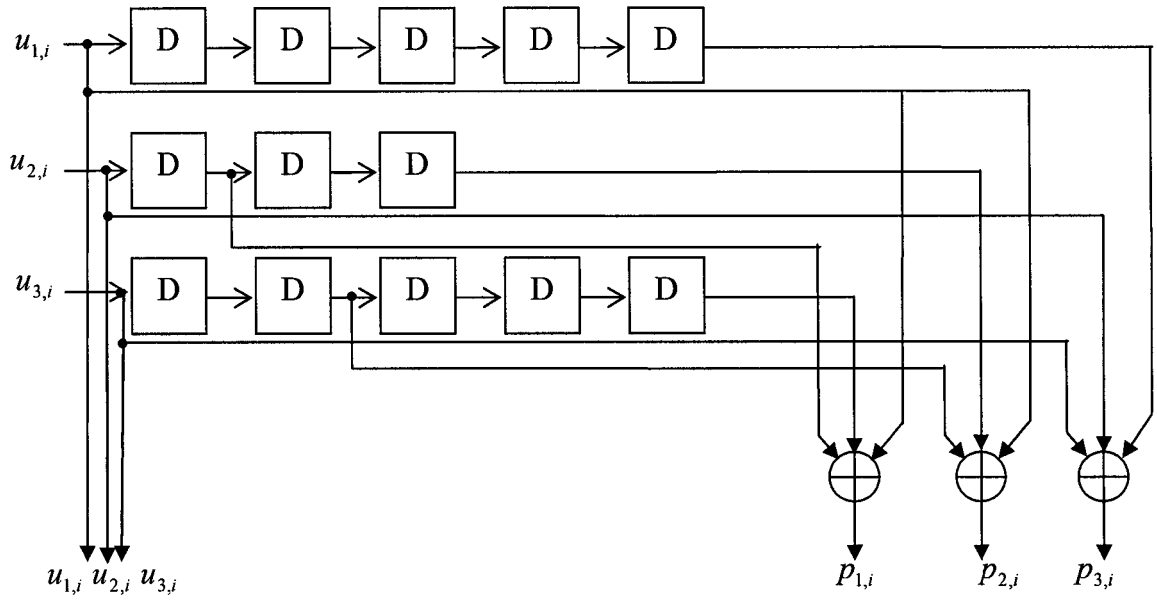

Figure 2. 11 Codeur doublement orthogonal au sens strict avec J=3 de taux $\mathbf{R} = \frac{\mathbf{J}}{\mathbf{2J}} = \frac{\mathbf{1}}{\mathbf{2}}$

L'équation de décodage à la deuxième itération est donnée par (2.30).

$$\lambda_{j,i}^{(2)} = y_{j,i}^{u} + \sum_{n=1}^{J} \left(y_{n,i+\alpha_{j,n}}^{p} \Diamond \sum_{\substack{k=1,k\neq j \\ (\alpha_{j,n}-\alpha_{k,n})<0}}^{J} \Diamond \lambda_{k,(i+(\alpha_{j,n}-\alpha_{k,n}))}^{(2)} \Diamond \sum_{\substack{k=1,k\neq j \\ (\alpha_{j,n}-\alpha_{k,n})\geq 0}}^{J} \Diamond \left(y_{k,(i+(\alpha_{j,n}-\alpha_{k,n}))}^{u} + \sum_{\substack{s=1 \\ s\neq n}}^{J} \left(y_{s,(i+\alpha_{j,n}-(\alpha_{k,n}-\alpha_{k,s}))}^{p} \Diamond \sum_{\substack{l=1,l\neq k \\ (\alpha_{l,s}-\alpha_{k,s})\geq 0}}^{J} \Diamond y_{l,(i+(\alpha_{j,n}-\alpha_{k,n})-(\alpha_{l,s}-\alpha_{k,s}))}^{u} \Diamond \sum_{\substack{l=1,l\neq k \\ (\alpha_{l,s}-\alpha_{k,s})<0}}^{J} \Diamond \lambda_{l,(i+(\alpha_{j,n}-\alpha_{k,n})-(\alpha_{l,s}-\alpha_{k,s}))}^{(1)} \right) \right) \right)$$

$j = 1,2,...,J;\ i = 1,2,...$

On peut alors dire qu'un code convolutionnel systématique de taux $R = \frac{J}{2J}$ est doublement orthogonal au sens strict, si les éléments de la matrice $(\alpha_{j,n})$ satisfont aux conditions suivantes :

1. Les différences $(\alpha_{j,n} - \alpha_{k,n})$ sont distinctes,
2. Les différences de différences, $(\alpha_{j,n} - \alpha_{k,n})$-$(\alpha_{l,s} - \alpha_{k,s})$, sont distinctes des différences,
3. Les différences de différences sont distinctes.

Ceci pour toutes les combinaisons des indices (j,n,s,l) avec $s \neq n, k \neq j, k \neq l$.

Plusieurs travaux ont portés sur l'identification des matrices de codages $(\alpha_{j,n})$ les plus adaptées, et qui répondent aux trois règles de la définition. Actuellement nous disposons d'un ensemble de codes pour J=2,…,30 [27]. D'autres matrices sont en cours d'études.

2.6 Conclusion

L'implémentation d'un algorithme sous forme microélectronique passe toujours par une phase où l'on doit assimiler le but de l'application et ses bases théoriques ou pratiques. Muni de cette connaissance, on est en mesure de comprendre l'impact de nos choix architecturaux sur le rendement que le client demande d'avoir pour son circuit.

Le cas de notre travail, qui s'inscrit dans un cadre académique part d'une thèse qui met en place un processus complexe qu'il a fallu étudier et assimiler pour en extraire les fonctionnalités devant être mises en place. Ainsi, il a été primordial de faire l'analyse de tout le processus qui a mené à l'élaboration de cet algorithme de codage correcteur d'erreur. Notre travail consistera donc à réaliser un montage capable de mettre en œuvre les codes CSO^{2}C-SS et leur décodage récursif qui se base sur les équations (2.29) et (2.30).

Il est intéressant maintenant de présenter la technologie cible qui s'offre à nous pour réaliser notre système. On s'attardera dans le chapitre suivant à introduire le technologie des circuits programmables FPGA et les moyens de les exploiter dans un projet de conception numérique.

CHAPITRE 3

ARCHITECTURE DU CIRCUIT PROGRAMMABLE FPGA CIBLE

3.1 Introduction

Les circuits intégrés sont largement utilisés pour remplacer l'assemblage de plusieurs boîtiers logiques. Le câblage s'en trouve simplifié, l'encombrement et les risques de pannes réduits. Les progrès dans le domaine de l'intégration permettent d'envisager des circuits de plus en plus réduits, plus performants, fiables et non énergivore. On utilise d'ailleurs plus souvent de la technologie «systèmes dans un circuit» (System On Chip, SOC).

En fait, il existe plusieurs technologies pour implanter un algorithme numériquement. Le choix est fait en suivant plusieurs critères dont les plus essentiels sont le volume de production, les performances souhaitées, le délai de mise en marché (Time to Market), ainsi que la maîtrise qu'on a de cette technologie.

Lors des processus visant à développer des prototypes pour valider une approche donnée, on a souvent recourt aux circuits FPGA en raison de leur flexibilité. On s'attardera donc sur les circuits FPGA de la compagnie Xilinx™ qui sont largement adoptés pour le prototypage au sein du LACIME de l'École de technologie supérieure. Après une brève description de l'architecture générale de ces circuits, une attention particulière sera accordée à la dernière famille que propose Xilinx™, à savoir les Virtex. L'intérêt d'une telle présentation se situe au niveau des structures internes que l'on peut exploiter pour optimiser notre implémentation. On détaillera aussi la procédure de développement en utilisant des langages de haut niveau pour ce type de circuits.

3.2 Les circuits FPGA

Les circuits FPGA, à l'inverse des circuits dit prédiffusés et personnalisés, ne nécessitent pas l'intervention du constructeur pour programmer la logique voulue. Ceci réduit énormément le coût du développement pour cette technologie et les rend attrayants pour le prototypage. Inventés par la firme Xilinx™, ces circuits se situent entre les réseaux logiques programmables (PLD) et les prédiffusés. Il s'agit en fait d'un composant standard qui combine la densité et les performances des prédiffusés à la souplesse de la reprogrammation des PLD.

3.2.1 Architecture des FPGA

L'architecture des FPGA est constituée de circuits configurables [29]. Ces derniers se composent d'une matrice de blocs logiques configurables (Configurable Logic Block, CLB) qui permet de réaliser des fonctions combinatoires et séquentielles. Autour de cette matrice de CLB, on trouve des blocs d'entrées/sorties (Input Output Block, IOB) dont le rôle consiste à gérer les entrées/sorties pour réaliser un interfaçage avec le monde externe. Chaque CLB est aussi entouré par quatre matrices de connexions qui permettent de «router» les signaux entre les différents CLB. La programmation du circuit consiste donc à charger une trame de bits dans des cellules de mémoire SRAM pour établir les différentes interconnexions entre les IOB et les CLB ainsi qu'entre les CLB eux même pour réaliser la fonction désirée et assurer la propagation des signaux. Nous allons tâcher de présenter les différentes composantes plus en détails.

3.2.1.1 Les CLB

Les CLB déterminent en partie les performances du FPGA. Chaque bloc peut être vu comme composé d'un ou plusieurs blocs similaires, généralement deux que l'on appelle «tranche» (SLICE). Chacune de ses tranches est formée de deux générateurs de

fonctions à quatre entrées et d'un bloc de mémorisation/synchronisation avec deux bascules D. Quatre autres entrées permettent de réaliser les connexions internes entre les différents éléments du CLB. La figure 3.1 représente le schéma d'une tranche pour la famille Virtex.

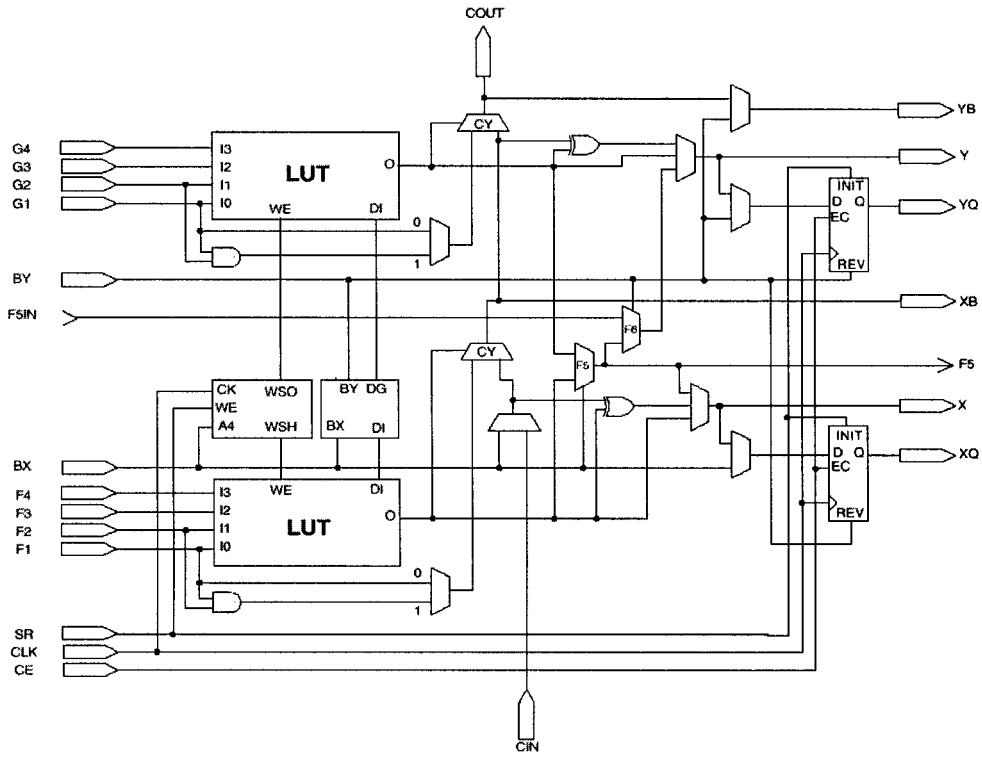

Figure 3. 1 Diagramme simplifié d'une tranche d'un FPGA Virtex de Xilinx™

Les blocs de logique combinatoire possèdent deux générateurs de fonction F et G à quatre entrées indépendantes (F1,..., F4, G1,..., G4) offrant une grande flexibilité de développement. Les deux fonctions sont générées à partir d'une table de vérité câblée inscrite dans une zone mémoire, ainsi les délais de propagation pour chaque générateur sont indépendants de la fonction que l'on réalise. Dans certaines familles de FPGA Xilinx™, une troisième fonctions H est réalisée en combinant les sorties F, G et une troisième variable d'entrées transitant par un bloc secondaire. Les sorties des générateurs

sortent de la tranche soit par la sortie X pour la fonction F, soit Y pour la fonction G. Ainsi on peut réaliser grâce à cet architecture soit :

- deux fonctions indépendantes à quatre entrées distinctes,
- une seule fonction de cinq variables voir jusqu'à neuf variables,
- deux fonctions, l'une à quatre variables et l'autre à cinq variables.

L'intégration de fonctions avec un nombre élevé de variables diminue le nombre de CLB nécessaire ainsi que les délais de propagation des signaux. Ceci augmente la densité et la vitesse du circuit. Les sorties des tranches peuvent s'appliquer à des bascules ou directement vers d'autres tranches ou des CLB.

Il existe une autre entrée appelée «Global Set/Reset» qui se charge de tout initialiser dans le circuit FPGA à chaque mise sous tension, chaque configuration en mettant à un ou à zéro respectivement de toute les bascules en même temps.

Un autre intérêt des CLB, est la possibilité de les configurer en mémoire RAM de 16×2 bits ou 32×1 bits. Les entrées (F1,..., F4, G1,..., G4) deviennent alors des lignes d'adressage pour sélectionner une cellule mémoire particulière. La fonctionnalité des signaux de contrôle se trouve modifiée dans cette configuration. Le contenu de la cellule mémoire adressée se trouve accessible sur les sorties des générateurs de fonction G et F et ils peuvent transiter par X et Y dans les bascules.

3.2.1.2 Les IOB

Les blocs d'entrées/sorties constituent l'interface entre les broches du FPGA et la logique interne qui y est implantée. Ils se trouvent sur toute la périphérie du circuit avec un bloc IOB par broche. Ce bloc, peut être configuré en entrée, en sortie, en signaux bidirectionnels ou en haute impédance. La figure 3.2 présente un exemple simplifié d'un tel bloc.

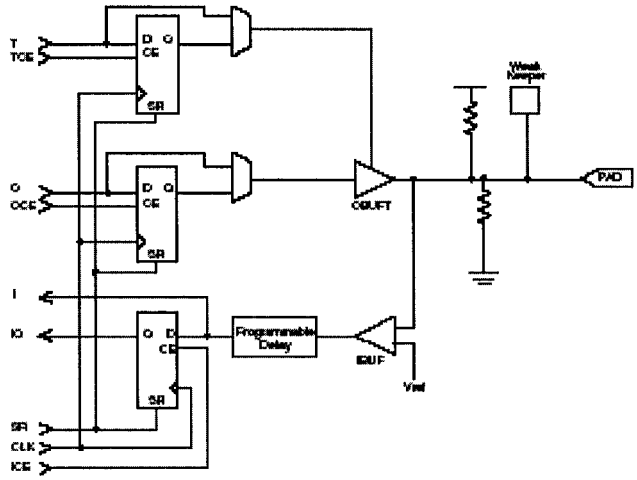

Figure 3. 2 Schéma simplifié d'un bloc IOB d'un FPGA Virtex de Xilinx™

Lors de la configuration en entrée, le signal passe par un tampon (Buffer) qui peut en fonction de sa programmation détecter soit des niveaux TTL ou CMOS. Il peut être routé par la suite, directement vers la logique du CLB, ou bien synchronisé grâce à une bascule de type D qui accepte des changements sur les fronts montants et descendants. On remarque aussi que l'entrée peut être retardée de quelques nanosecondes pour compenser le retard du signal d'horloge qui passe par un amplificateur.

La configuration de l'entrée se fait à l'aide d'un multiplexeur qui est commandé par un bit positionné dans une case mémoire.

En configuration de sortie, il est possible d'appliquer les modifications suivantes au signal :

- inversion ou non du signal avant d'attaquer l'IOB,
- synchronisation du signal sur les fronts montants ou descendants de l'horloge,
- mettre en place un «Pull-up» ou un «Pull-down» pour limiter la consommation des entrées, sorties inutilisées,

- signaux en logique trois états ou deux états. Le contrôle de mise en haute impédance ainsi que la réalisation des lignes bidirectionnelles sont assurés par un signal de commande «Out Enable» pouvant être inversé ou non.

Chaque sortie peut délivrer un courant de 12mA, ouvrant de nombreuses possibilités au concepteur pour connecter au mieux un FPGA avec les périphériques extérieures.

3.2.1.3 Les interconnexions

Les connexions en interne pour les FPGA sont assurées par des segments métallisés comme on peut le voir sur la figure 3.3 et ils influencent de plus en plus les performances des FPGA. Parallèlement à ces lignes, il existe des matrices programmables, dénotées PSM, réparties sur l'ensemble du circuit horizontalement et verticalement entre les divers CLB. Ces matrices se chargent d'établir les connexions entre les diverses lignes par le biais de transistor MOS dont l'état est contrôlé par des cellules en mémoire vive. Le rôle de ce réseau de connexions est de relier efficacement les CLB et les IOB. On distingue dans l'architecture Xilinx™, trois sortes d'interconnexions dépendamment de la longueur et de la destination du lien.

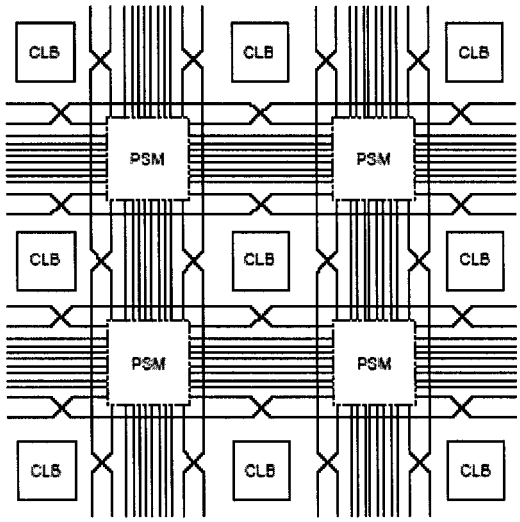

Figure 3. 3 Exemple de lignes d'interconnections dans les FPGA, avec les matrices de connexions programmables (PSM)

Les interconnexions à usage général

Il s'agit d'une grille de segments métalliques verticaux et d'autres segments horizontaux positionnés entre les lignes et les colonnes de CLB et IOB.

Des matrices de commutation situées à chaque intersection ont pour rôle le raccordement des segments entre eux selon diverses configurations pour assurer le passage des signaux d'une voie à l'autre. Ce sont ces interconnexions qui relient n'importe quel CLB aux autres. Pour éviter que les signaux ne soient affaiblis par de fortes demandes de courant, on dispose généralement de mémoires tampons implantées en haut et à droite de chaque matrice.

Les interconnexions directes

Ces interconnexions servent à relier directement les CLB aux IOB pour optimiser l'efficacité en terme de vitesse et d'occupation de surface. Il est aussi possible de relier

directement certaines entrées de CLB avec les sorties d'autres CLB comme on peut le voir sur la figure 3.4.

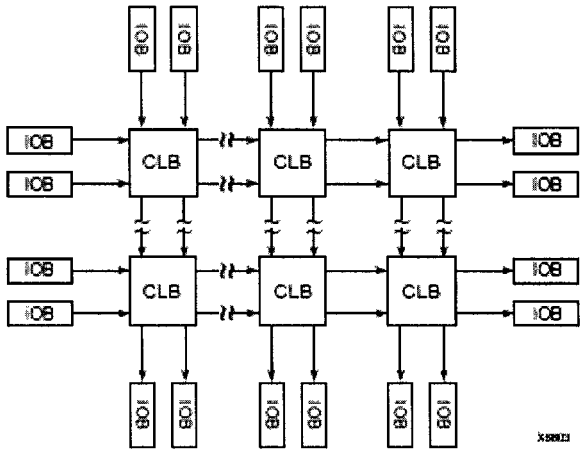

Figure 3. 4 Interconnections directe

Les longues lignes

Ce sont des longues lignes qui parcourent toute la longueur et la largeur du composant pour transmettre avec un minimum de retard les signaux entre les différents éléments. Le but de ces lignes est d'assurer une synchronisation aussi parfaite que possible et d'éviter la multiplicité des points d'interconnexions. Ce sont ces lignes qui véhiculent les signaux d'horloges et de «SetReset».

3.2.2 Cas des FPGA de la famille Virtex

L'évolution des technologies de l'intégration et leur exploitation par les fabricants de FPGA, ont fait passer ces derniers du simple statut de plateforme de prototypage à celui de véritable concurrent des ASIC pour certaines applications. La famille Virtex [30] est commercialisée par Xilinx™ depuis 1998. Le circuit Virtex est constitué, comme on peut le voir sur la figure 3.5 de deux entités configurables :

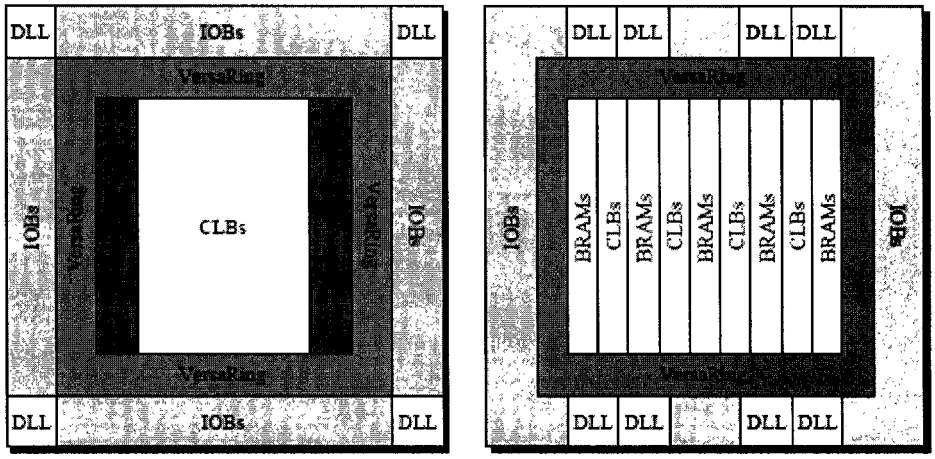

Figure 3. 5 Architecture générale des Xilinx™ Virtex (a) et Virtex-E (b)

- Les CLB qui sont composés de quatre cellules logiques reparties en deux tranches identiques (figure 3.6), et qui servent à la conception du circuit numérique. Chaque cellule logique contient :

  - Une LUT à quatre entrées pour générer les fonctions logiques. Un multiplexeur choisit les sorties des deux LUT d'une tranche pour réaliser n'importe quelle fonction de cinq variables booléennes. De même, un autre multiplexeur combine les quatre LUT d'un CLB en sélectionnant la sortie de l'un des multiplexeurs. Il est intéressant de voir que lors de l'évolution des FPGA Xilinx™, les LUT à trois entrées que l'on trouvait dans les premiers produits (XC4000) ont été abandonnées et remplacées par un mécanisme plus souple. De plus, chaque LUT du Virtex permet la conception d'une mémoire synchrone de 16×1 bits. On peut ainsi joindre les LUT d'une tranche pour réaliser une mémoire synchrone de taille 16×2 bits, 32×2 bits ou 16×1 bits à double accès (Dual-port synchronuous RAM). Une LUT peut aussi servir de registre à décalage de seize bits.

- Un élément de mémorisation avec des signaux d'initialisation (Set et Reset) synchrones ou asynchrones.

- De la logique additionnelle pour la réalisation de circuits arithmétiques performants. Chaque tranche dispose d'une ligne de propagation de retenue pour faciliter la réalisation d'additionneurs, et d'éléments servants pour des multiplicateurs.

- Les IOB pour interfacer les CLB aux bornes du circuit. Le dispositif de routage «VersaRing» offre les ressources nécessaires à l'interconnexion des CLB aux IOB. L'intérêt de ce système est de pouvoir modifier le système implanté sans changer les attributions des bornes donc en gardant la compatibilité au niveau du boîtier.

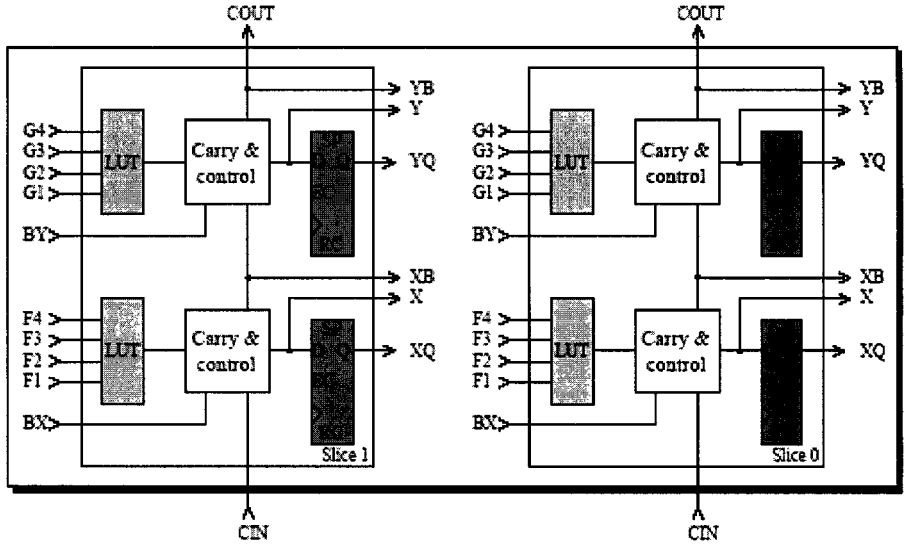

Figure 3. 6 Architecture simplifiée du CLB d'un circuit Virtex

Le circuit Virtex offre aussi la possibilité de stocker de l'information dans deux mémoires (BRAM ou BlockSelectRAM+), organisées en blocs de 4096 bits, situées de part et d'autre de la matrice de CLB. La hauteur de chaque bloc équivaut à celle de quatre CLB, de sorte que si on dispose d'une matrice de CLB de 64 lignes, on aura 16

blocs. Les outils fournis par Xilinx™ permettent de construire avec ces blocs des mémoires synchrones simples ou doubles ports de la taille que l'on désire.

Le composant possède aussi quatre lignes d'horloge disposant chacune d'une DLL (Delay Locked Loop). Cette dernière sert à contrôler le décalage d'horloge (Clock skew) à l'intérieur du FPGA ou entre plusieurs circuits, ainsi que de déphaser, doubler ou diviser une horloge.

Les vendeurs de FPGA classifient les circuits d'une même famille en fonction de leur fréquence maximale de fonctionnement, processus appelé «Speed grading», et le prix est fonction de cette vitesse même si le coût de fabrication est le même. La famille Virtex est caractérisée par un chiffre représentant la vitesse de fonctionnement, -4 étant le circuit le plus lent et -5 ou -6 pour ceux qui sont plus rapides. La figure 3.7 illustre la dénomination utilisée et qui spécifie le type de composant, son «speed grading», le type de boîtier, le nombre de bornes et les températures de fonctionnement.

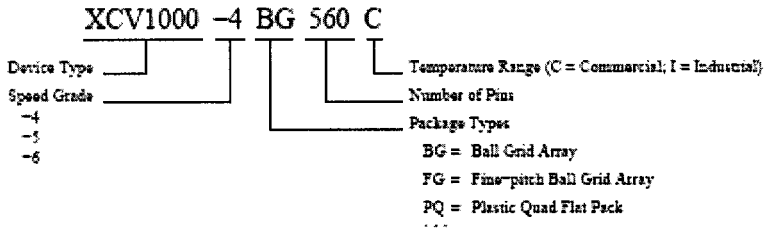

Figure 3. 7 Nomenclature d'un circuit de la famille Virtex

Il est à noter que d'autres FPGA s'inspirant de la famille Virtex ont été développés depuis le début de la commercialisation. Les Virtex-E [31] par exemple disposent de plus de mémoire BRAM qui, comme on peut le voir dans la figure 3.5b est organisée en colonne et est répartie dans la matrice de CLB. Une technologie d'intégration plus fine permet des fréquences de fonctionnement plus élevées (0.18µm pour la famille Virtex-E contre 0.22µm pour la famille Virtex). Une autre famille, la Virtex-E Extended Memory [32] propose des circuits destinés aux applications gourmandes en mémoire, comme le

traitement d'images. La famille Virtex-II [33] quant à elle, offre des primitives nouvelles pour réaliser des opérateurs arithmétiques performants, comme un maximum de 168 multiplicateurs traitant deux nombres de 18 bits codés en complément à deux. Avec l'avenue de la série des Virtex-II pro (Virtex-II Pro, Virtex-II Pro X et Virtex-II Pro EasyPath), on assiste à une augmentation des primitives pour opérations arithmétiques ainsi qu'à l'adjonction de processeur IBM PowerPC 405 en dur et d'émetteurs récepteurs série «RocketIO» qui permettent des débits de l'ordre de 10Gbps. Enfin, la famille Virtex-4 (Virtex-4 LX, DX et FX) utilise la nouvelle architecture de Xilinx™ dite ASMBL (Advanced Silicon Modular BLock) et intègre de nouvelles fonctionnalités pour le calcul arithmétique, un contrôleur de FIFO à l'intérieur du BlockRAM pouvant fonctionner jusqu'à 500 MHz et la partie numérique du control d'accès au medium d'Ethernet 10/100/1000 Mbps.

3.3 Outils et méthodologie de développement pour FPGA

Le développement de circuits logiques numériques à haute vitesse est un processus complexe qui nécessite de respecter une méthodologie spécifique à chaque technologie ciblée. Certaines étapes du développement sont communes à toutes les familles de circuits logiques, alors que d'autres sont spécifiques à une ou plusieurs d'entre elles. Le cas des circuits FPGA suit un schéma qui, de prime abord peut paraître linéaire, mais qui nécessite souvent des reprises d'une partie ou de l'ensemble de la procédure pour atteindre les fonctionnalités et performances souhaitées. La figure 3.8 résume les deux approches possibles pour la conception pour FPGA que l'on va présenter en détails [34].

3.3.1 La saisie du design

La méthode de saisie traditionnelle a été sous forme schématique. Il s'agit d'outils graphiques où le concepteur spécifie les portes logiques ainsi que des modules ou macros de plus grande complexité qu'il veut utiliser. Pour ce faire, il doit d'abord

sélectionner l'outil de travail (exemple : View Draw de ViewLogic), puis une bibliothèque de composants spécifiques au vendeur et à la famille pour laquelle il développe (exemple : Xilinx™ XCR3256XL). Une fois les éléments mis en place sur une page de saisie, il doit interconnecter les entrées et sorties entre elles par le biais de bus et de liaison simple. Il s'ensuit l'ajout et la connexion des tampons d'entrées et sorties et enfin la génération d'une «Netlist». Cette dernière est une représentation compacte sous format texte de l'ensemble du circuit (portes, connexions, tampons,...etc.). On remarque qu'il y a un standard de «Netlist» dans l'industrie à savoir les fichiers EDIF (Electronic Digital Interchange Format), mais que les vendeurs peuvent proposer un format propriétaire (exemple: XNF pour Xilinx Netlist Format). Afin de réaliser le schéma d'une fonction, une page peut contenir approximativement 200 portes logiques, or pour des projets de 10.000, 20.000 ou 50.000 portes, il est clair que la procédure sera fastidieuse, génératrice d'erreurs et peu productive. Un autre point faible de cette approche de saisie est la nécessité de reprendre une énorme partie de la conception en cas de migration d'une famille du composant à une autre ou d'un vendeur à un autre, puisque les éléments qu'on utilise sont intimement associés à une famille d'un vendeur. Une autre problématique se pose quand on s'intéresse aux vérifications fonctionnelles, puisque aucune norme ne régit ce format de fichier d'où l'impossible portabilité de vecteurs de tests d'un outil de simulation à un autre.

Les grandes limitations de la saisie en schématique ont porté le monde du développement de systèmes numériques à adopter une seconde voie, à savoir les langages de descriptions de haut niveau connus sous l'acronyme HDL (Hardware Description Language). Ce sont des langages qui permettent de décrire le circuit sous forme de langage de programmation de haut niveau des blocs constituant le design. Progressivement, les blocs sont décris si nécessaires jusqu'à des niveaux proche des ressources matérielles.

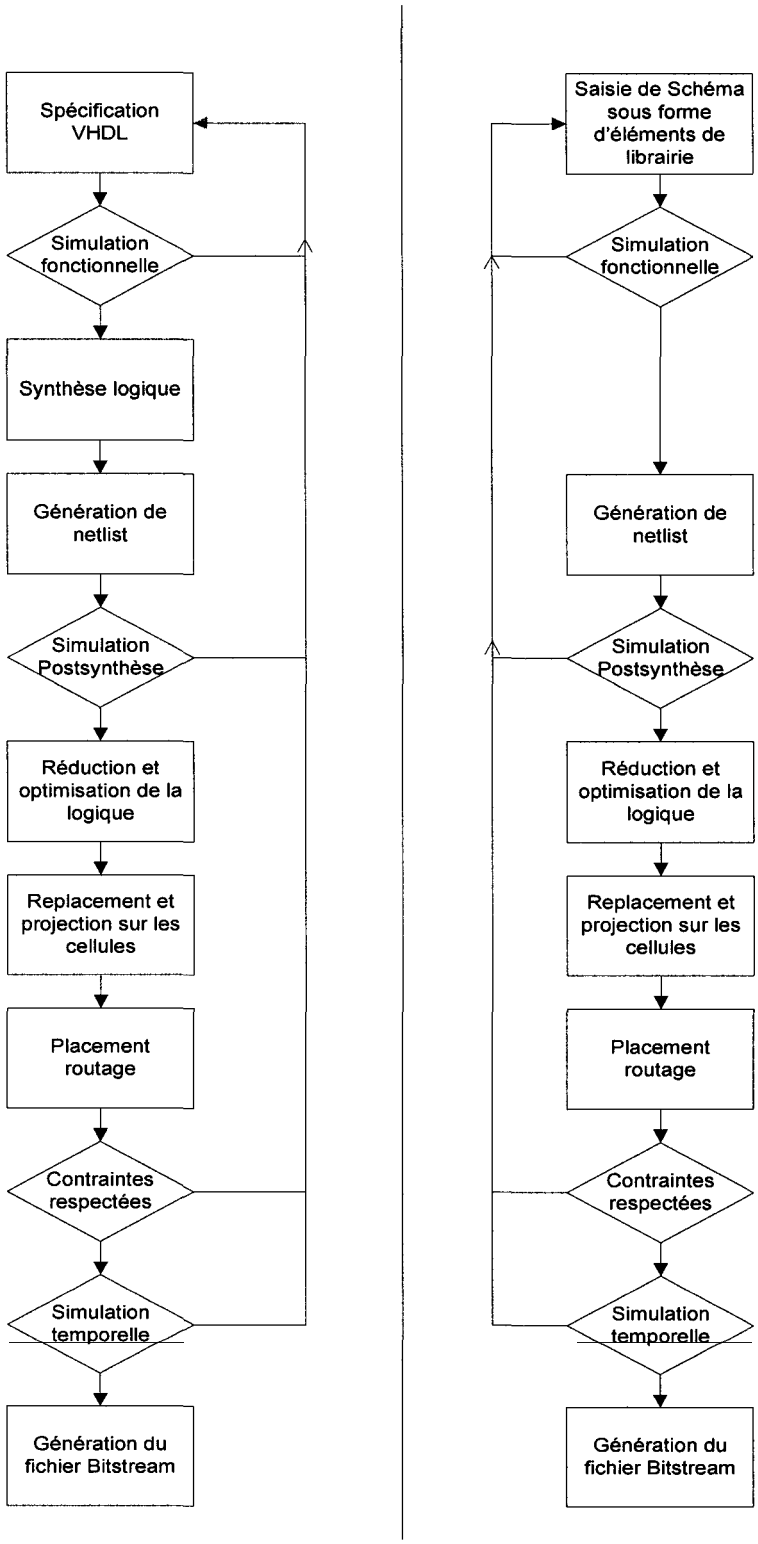

Figure 3. 8 Les deux flux de programmation des FPGA

Il existe deux langages HDL communément utilisés, le VHDL et le Verilog. On n'essayera pas dans ce travail de présenter la bataille qui fait rage dans la communauté des développeurs numériques sur l'avantage de l'un ou de l'autre de ces langages. Le lecteur intéressé par ce débat, peut se référer à un ensemble d'articles qui traitent du sujet, dont la référence [35]. Le langage VHDL étant largement utilisé au LACIME et dans beaucoup d'organismes nord-américains, en raison même de son origine, notre choix c'est donc logiquement porté sur ce langage pour la pérennité du travail.

On peut décrire en VHDL un circuit suivant trois niveaux d'abstraction :

- niveau structurel : décrit le câblage des composants élémentaires,
- niveau flot de données : décrit les transformations d'un flot de données de l'entrée à la sortie,
- niveau comportemental : décrit le fonctionnement par des blocs de programmes appelés processus qui s'échangent des informations par le biais de signaux, et contiennent de la logique séquentielle.

À ce niveau de la description, on utilise des outils de saisie du code, de compilation du code, de déverminage et de simulation. Pour injecter nos vecteurs de vérification, le VHDL offre une approche élégante par le biais de fichiers dits «Test Bench» qui vont isoler la structure à tester et lui appliquer les intrants que l'on spécifie dans le même fichier. Ce même fichier de vérification peut être repris ultérieurement et appliqué. En effet, cette première simulation est dite simulation RTL (Register Transfert Level) et elle ne rend compte que de l'exactitude de l'algorithme et de son bon fonctionnement sans tenir compte des délais de portes et de routages.

3.3.2 La synthèse

Cette étape permet de passer d'un code VHDL proche d'une description en un langage de programmation de haut niveau, à une architecture au niveau transfert de registre (RTL). Le synthétiseur (exemple : Synplify, Leonardo Spectrum, FPGA Express, etc.) se charge alors de déduire quelles portes ou fonctions à utiliser pour rendre compte de la description. Le résultat est un «Netlist» qui est spécifique à un vendeur et à une famille donnée. Lors du paramétrage du synthétiseur, on doit spécifier le vendeur, le modèle de circuit utilisé ainsi que l'ajout de contraintes. Ces dernières sont soit temporelles, soit de nature à minimiser la surface totale du circuit final. On peut aussi imposer les broches qui seront affectées à chaque entrée et sortie. Le synthétiseur procède ensuite à un partitionnement de la logique dans les ressources disponibles et fournit un fichier où il décrit la vitesse atteinte, la complexité du circuit en nombre de tranches et de CLB ainsi qu'en nombre de bascules. On trouve aussi dans ce fichier les chronogrammes des signaux depuis leurs entrées jusqu'à leurs sorties, de telle façon qu'il nous est possible de détecter les chemins critiques et d'essayer de les améliorer par des procédures adéquates.

Il est courant de procéder à ce niveau à une nouvelle vérification pour s'assurer que l'algorithme n'est pas affecté par les délais de porte. Cette vérification est appelée simulation post-synthèse et est réalisée grâce au même fichier que l'on a utilisé au niveau fonctionnel.

3.3.3 Optimisation, projection et placement-routage

Le «Netlist», généré soit par l'outil de synthèse, soit par le logiciel de saisie schématique, est optimisée avant son utilisation. Cette étape se charge de retirer les signaux inutilisés, de simplifier les expressions booléennes, de détecter les signaux

équivalents et surtout de dupliquer la logique dont la sortance est insuffisante pour augmenter cette dernière (problème de FanOut).

Suite à cette optimisation, les équations booléennes doivent être réarrangées pour tenir dans la logique disponible dans les LUT. Ainsi, une équation à huit entrées sera projetée sur trois LUT, puisque chacune de ces tables ne dispose que de quatre entrées possibles.

Une fois la logique bien agencée, l'outil procède à l'étape de placement-routage qui consiste à attribuer les CLB du circuit aux équations qui résultent de la projection, puis à définir l'interconnexion. C'est une opération hautement complexe qui est résolue par un ensemble d'heuristiques. De ce fait, si on procède à une suite de placements et routages, il n'est pas rare de voir une différence dans le résultat final.

L'ensemble de ces trois étapes est géré par le même logiciel (exemple : Xact), et on obtient en final un fichier «Bitstream» qui contient toute l'information pour configurer le FPGA. On procède encore une fois à une simulation dite temporelle, qui prend en compte les délais de portes, mais aussi les délais d'interconnexion et de routage dû aux capacités parasites. La simulation temporelle vérifie que la fonctionnalité n'a pas été modifiée par l'introduction de ces nouveaux délais.

L'étape finale consiste à charger le FPGA avec le fichier «Bitstream» en utilisant une liaison JTAG par exemple et le logiciel Xchecker pour le cas de Xilinx™. Des tests finaux sont effectués en injectant des données et en recueillant les résultats à l'aide d'un analyseur logique ou d'un montage spécifique.

3.4 Conclusion

Le domaine de la conception numérique a connu un grand essor avec l'utilisation combinée de langages de haut niveau (VHDL et Verilog essentiellement), ainsi que des

circuits FPGA. En effet, l'utilisation des HDL permet actuellement un temps de développement court avec une grande souplesse d'utilisation, de vérification et de modification, ce qui augmente significativement la productivité. Au niveau matériel, le coût sans cesse à la baisse des FPGA, leurs hautes densités ainsi que leurs fines granularités, les posent comme concurrent directs des circuits personnalisés qui ne sont rentables que pour des grandes séries.

Néanmoins, la maîtrise d'un outil de développement ne peut dispenser le concepteur de bien connaître tout le processus de développement, mais surtout les forces et les faiblesses du langage utilisé, du synthétiseur ainsi que de la technologie cible. Le but de ce chapitre a été de présenter le processus de développement et les FPGA Xilinx™ que nous ciblons.

Il est intéressant de mentionner les quelques travaux prometteurs dans le domaine de la conception numérique et qui visent à améliorer les langages HDL. On retiendra les travaux du groupe IEEE 1076.1 qui ont abouti à la spécification VHDL-AMS, qui ajoute au standard VHDL 1076-1993 des possibilités de description et de simulation de systèmes analogiques. En outre, le principe de réutilisation du code HDL a porté lui aussi à produire une nouvelle couche dans le langage VHDL pour implanter le concept Objet, le langage Objective VHDL est le fruit de cette évolution. Enfin, la nécessité d'un langage permettant de concevoir et faire interagir aussi bien la partie logiciel que la partie matériel (Codesign) a amené la sortie de nouveaux langages tels que le JHDL basé sur Java, le Handel-C basé sur le C et le plus prometteur à l'heure actuelle, le SystemC basé sur le C et le C++.

Les chapitres qui suivent, traitent de la réalisation du décodeur itératif doublement orthogonal au sens strict. Dans le chapitre quatre, les choix qui ont été faits seront présentés ainsi que la procédure de généralisation mise en place. Le chapitre cinq

détaille des améliorations qui ont été portées à la structure de base, l'étude de la complexité et des performances du décodeur.

CHAPITRE 4

ARCHITECTURE DU DÉCODEUR CSO^{2}C-SS

4.1 Introduction

Les chapitres précédents ont été concentrés sur les aspects théoriques de l'algorithme de décodage et sur la technologie cible qui sera utilisée. Ils constituent la matière première avec laquelle nous allons mettre en place le décodeur CSO^{2}C-SS. Le but de la réalisation est de disposer d'un prototype microélectronique programmable afin de valider le concept. On s'intéresse aussi à l'étude de la complexité du système que l'on désire analyser pour les différentes matrices de décodeur, ainsi que les performances d'erreur atteintes. Il apparaît clairement que la réalisation doit être à la fois assez souple pour représenter l'ensemble des décodeurs dont on possède la structure, mais on doit aussi pouvoir en estimer la complexité et les performances aussi fidèlement que possible. On notera que la structure n'aura pas à être changée en cours de fonctionnement du circuit.

L'ensemble des exigences du projet a amené à se détourner des méthodes éprouvées que l'on trouve dans les études [25] et [36]. En effet, les techniques utilisées dans ces travaux engendrent un surplus de logique non négligeable et surdimensionne le circuit en prenant le cas de la taille extrême du décodeur comme structure de base. Ceci fausse l'estimation de la complexité ainsi que des performances.

Riche de ces remarques, on a orienté le choix vers la réalisation d'un programme dans un langage de haut niveau pour générer à chaque fois le code VHDL de chaque structure. Le recourt à un langage tel que C s'explique par le besoin de dissocier le code du programme qui automatise la génération du code VHDL de celui devant être synthétisé.

Le présent chapitre commence par l'analyse du décodeur CSO^{2}C-SS, puis la proposition d'une architecture et son codage en langage VHDL. Il s'en suit une validation fonctionnelle pour le cas où J=3, avant de présenter la procédure de généralisation à tous les décodeurs possibles.

4.2 Analyse de l'algorithme du décodeur CSO^{2}C-SS

Le décodeur, comme on l'a vu dans le chapitre deux, est constitué d'un systèmes de J équations d'inversion de parité qui sont complètement définies quand on les associe à une matrice, que l'on nommera matrice des connections, $\alpha_{j,n}$. Ainsi, pour un décodeur dont J=3, le taux de codage est de $\frac{3}{6}$, et la matrice des connections est la suivante [1] :

$$\begin{bmatrix} 0 & 0 & 5 \\ 1 & 3 & 0 \\ 5 & 2 & 0 \end{bmatrix}$$

L'équation d'inversion de parité s'exprime pour un premier étage de décodage, avec j=1,…,J:

$$\lambda_{j,i} = y^{u}_{j,i} + \sum_{n=1}^{J} \left(y^{p}_{n,i+\alpha_{j,n}} \Diamond \sum_{\substack{k=1 \\ k\neq j \\ \alpha_{j,n}-\alpha_{k,n}\geq 0}}^{J} \Diamond y^{u}_{k,i+\alpha_{j,n}-\alpha_{k,n}} \Diamond \sum_{\substack{k=1 \\ k\neq j \\ \alpha_{j,n}-\alpha_{k,n}<0}}^{J} \Diamond \lambda_{k,i+\alpha_{j,n}-\alpha_{k,n}} \right) \tag{4.1}$$

que l'on peut aussi écrire sous la forme suivante :

$$\lambda_{j,i} = y^{u}_{j,i} + \sum_{n=1}^{J} \psi_{j,n,i} \tag{4.2}$$

4.2.1 Décomposition de l'algorithme

Afin de réaliser m itérations de décodage, on doit mettre en série m décodeurs CSO^{2}C-SS, suivant la structure de la figure 2.10, pour réaliser l'équation (4.3) à la dernière itération.

$$\lambda_{j,i}^{(m)} = y_{j,i}^{u(m)} + \sum_{n=1}^{J} \psi_{j,n,i}^{(m)} \tag{4.3}$$

avec

$$\psi_{j,n,i}^{(m)} = y_{j,i+\alpha_{j,n}}^{p(m)} \Diamond \sum_{\substack{k=1 \\ k\neq j \\ \alpha_{j,n}-\alpha_{k,n}\geq 0}}^{J} \Diamond \lambda_{k,i+\alpha_{j,n}-\alpha_{k,n}}^{(m-1)} \Diamond \sum_{\substack{k=1 \\ n\neq j \\ \alpha_{j,n}-\alpha_{k,n}<0}}^{J} \Diamond \lambda_{k,i+\alpha_{j,n}-\alpha_{k,n}}^{(m)} \tag{4.4}$$

ceci pour j=1,…,J et i représentant le temps.

Ainsi, on peut schématiser notre premier bloc de décodage par une boite noire, qui est constituée d'un ensemble d'entrées et de sorties comme on le voit sur la figure 4.1.

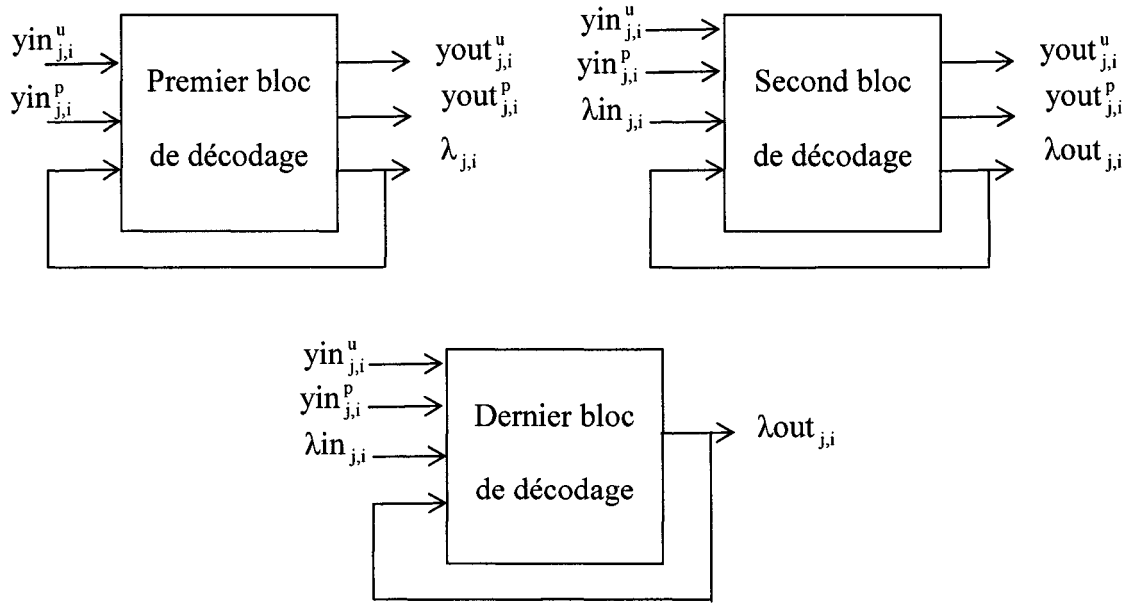

Figure 4. 1 Diagramme boite noire des blocs de décodage

En entrée, on remarque que l'on a besoin des J symboles d'information, et des J symboles de parité. On doit délivrer en sortie, après traitement, les J valeurs approximées $\lambda_{j,i}$, mais aussi les entrées qui sont nécessaires aux calculs dans les (m-1) autres décodeurs.

De la même manière, on représente les (m-2) autres structures de décodages situées entre le premier et le dernier décodeur. La différence que l'on peut relever est le besoin de disposer des J valeurs approximées par le décodeur en aval.

Le dernier bloc à utiliser, quant à lui, dispose d'une représentation similaire, à la figure précédente, mais avec seulement J sorties associées à l'approximation finale obtenue et que l'on va soumettre à une prise de décision.

4.2.2 Analyse des opérations de base à effectuer

L'algorithme utilisé se compose d'un ensemble de trois opérations de base que l'on est en mesure d'extraire facilement pour les analyser. Il est clair que ces opérations sont :

- L'addition : Afin de calculer chaque $\lambda_{j,i}$, on doit procéder à l'addition de (J+1) valeurs. En effet, pour chaque valeur approximée, il faut sommer les J valeurs $\psi_{j,n,i}$ et le symbole d'information $y^{u}_{j,i}$ associé. C'est une des opérations qui nécessite beaucoup d'attention en raison de son impact sur les performances du système.

- L'addmin : Cet opérateur intervient dans le calcul des $\psi_{j,n,i}$. Pour deux opérandes ξ_1 et ξ_2, on l'exprime comme suit :

$$\xi_1 \Diamond \xi_2 = -\mathrm{Sign}(\xi_1)\,\mathrm{Sign}(\xi_2)\,\min(|\xi_1|,|\xi_2|) \tag{4.5}$$

Afin d'évaluer chaque $\psi_{j,n,i}$, on a besoin de (J+1) addmin. Au total, on doit disposer de $J^2(J+1)$ addmin pour chaque bloc de décodage. Puisque son utilisation est fréquente, il faudra réaliser cet opérateur de façon efficace.

- Stockage et décalage : l'algorithme de codage et de décodage est basé entièrement sur la combinaison de ces deux opérations. On doit donc trouver la méthode la plus intelligente pour réaliser cette tâche. De prime abord, on doit stocker les $y^u_{j,i}$, $y^p_{j,i}$ et $\lambda_{j,i}$ à chaque itération. Si on procède de cette façon, on aura besoin de 3*J*Maxi cellules mémoires, Maxi étant la valeur maximale dans la matrice des connections. On doit ensuite les décaler à chaque incrémentation temporelle. Or, une simplification présentée dans [37] et [19], permet de n'utiliser que 2*J* Maxi cellules mémoires. Dans notre cas, ceci est vrai seulement pour le dernier bloc de décodage.

4.3 Architecture proposée

L'architecture proposée tire profit du parallélisme du traitement à réaliser par le décodeur. Les éléments importants qui ont été retenus lors de la conception sont :

- parallélisme,
- simplicité,
- modularité.

C'est à cette fin que le décodeur a été décomposé tout d'abord en une succession de décodeurs élémentaires, chacun intervenant dans une itération. On distingue les trois structures de base que l'on a nommé :

- décodeur frontal,
- décodeur intermédiaire,
- décodeur terminal.

La figure 4.2 donne la structure du décodeur intermédiaire comme exemple. La décomposition a été réalisée sur la base des différences structurelles que l'on a déjà présentées, mais surtout en raison du caractère de sous système global de chacun de ses modules. En effet, chaque module est autonome en terme d'entrées/sorties, de sorte que dans un cas extrême, chacun de ces modules peut être placé dans un FPGA. En outre, la mise en cascade de tous ces modules donne lieu à la réalisation d'un décodeur d'un nombre quelconque d'itérations.

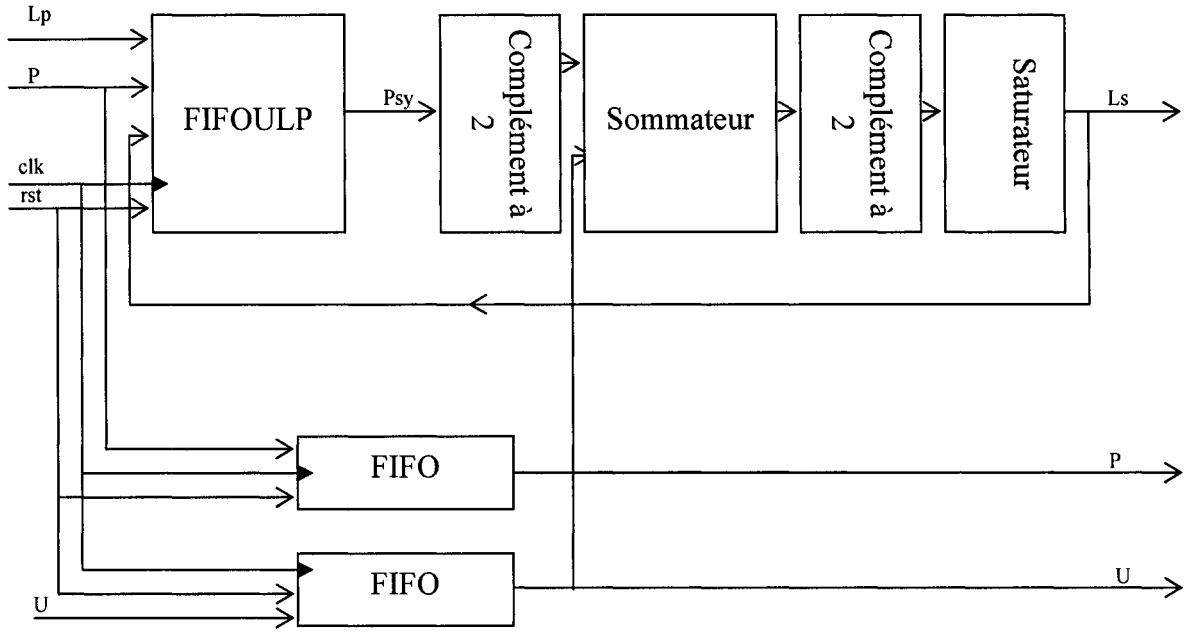

Figure 4. 2 Structure bloc du décodeur intermédiaire

4.3.1 Nature de l'information utilisée

Le décodeur a besoin de signaux en entrée pour fonctionner. L'utilisation des FPGA comme technologie cible impose une logique synchrone, ce qui nous amène à utiliser une horloge pour cadencer le fonctionnement du circuit. De plus, le recours à des éléments de mémorisation nécessite de mettre l'ensemble de ses cellules à un état que l'on connaît en début de fonctionnement, sinon les résultats seront faussés.

Le démodulateur délivre en entrée du décodeur une information sur laquelle il ne prend aucune décision. Ces données sont quantifiées sur un ensemble de q bits dont le bit de poids fort (MSB) correspond à la décision dure, et les q-1 autres bits donnent une indication sur la fiabilité de la décision. Plusieurs représentations sont possibles pour définir ces q bits. Ainsi, si l'on considère que q=3, cas le plus utilisé sur les décodeurs à entrées/sorties souples, on trouve le cas classique des représentations en binaire signé, binaire en complément à deux, mais aussi les deux représentations que l'on résume dans les deux Tableaux I.

Tableau I

Deux représentations possibles des q bits des symboles.

Représentation binaire	**Équivalent décimal**
111	-7
110	-5
101	-3
100	-1
000	+1
001	+3
010	+5
011	+7

Représentation binaire	**Équivalent décimal**
111	7
110	6
101	5
100	4
011	3
010	2
001	1
000	0

On notera ici que l'on est en notation antipodale, c'est-à-dire qu'un zéro binaire représente +1 et un un binaire un -1.

Dans notre étude, après simulation sous Matlab, on a convenu d'utiliser une information sur q bits en binaire signé en raison de la nature des opérations à réaliser. Cette représentation se trouve en effet plus facile à utiliser pour les opérateurs addmin qui sont nombreux, ainsi que pour l'addition moyennant une conversion en complément à deux.

4.3.2 Les blocs FIFO de mémorisation et de décalage

Les blocs de mémorisation et de décalage sont, en fait, des registres à décalage du type premier entré premier sorti (First In First Out, FIFO). Cette structure se charge de mémoriser l'information sur une certaine profondeur. Pour le cas du décodeur CSO^2C-SS, il s'agit de mémoriser des symboles quantifiés sur q bits et ceci pour une profondeur égale à la valeur maximale dans la matrice des connections que l'on nomme Maxi. On peut représenter cette structure par une concaténation de blocs de bascule D de largeur q et de profondeur Maxi, , comme on peut le voir dans la figure 4.3.

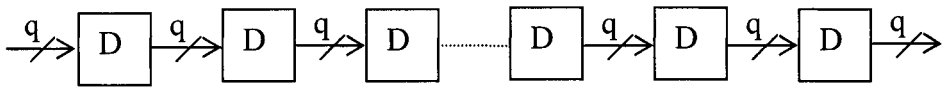

Figure 4. 3 Réalisation d'un bloc FIFO avec des bascules D

La représentation n'est pas complète puisqu'il manque les signaux d'horloge et de remise à zéro.

4.3.3 Les blocs FIFOULP

Ces blocs se chargent de la mémorisation d'un ensemble de données et du calcul, en utilisant les addmin, des différents $\psi_{j,n,i}$ nécessaire pour estimer l'approximation des symboles émis.

Ce système est constitué de 2*J blocs de FIFO de largeur q et de profondeur Maxi. La moitié des FIFO servent à mettre en place un mécanisme de décalage qui permet de prélever des symboles nécessaires au calcul des $\psi_{j,n,i}$. L'autre moitié sert à décaler des données, mais aussi à réaliser des opérations addmin en insérant ces opérateurs entre deux blocs de mémorisation. L'emplacement des addmin, ainsi que les symboles devant être prélevés des FIFO sont définis grâce à la matrice de connections. La figure 4.4 donne une représentation du bloc FIFOULP avec ses entrées et ses sorties.

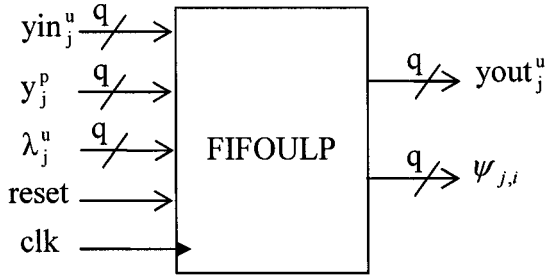

Figure 4. 4 Schéma bloc du FIFOULP

On peut donc clairement voir deux ensembles de J entrées représentant les symboles de parités, ainsi que les symboles d'information ou d'information estimée suivant les cas. Il existe aussi J entrées qui correspondent à une rétroaction des sorties finales du bloc de décodage. En sortie, le bloc fournit les symboles d'informations, retardé de Maxi coups d'horloges ainsi que les J^2 terme $\psi_{j,n,i}$.

La structure interne du FIFOULP dépend de la matrice de connections, et il est difficile de donner une description des étapes nécessaires pour la déduire. On se basera sur l'exemple de la matrice pour J=3 pour expliciter la procédure. Il nous faut tout d'abord écrire les J équations d'inversions de parité qui servent au calcul des estimations des symboles d'information. Pour le cas de J=3 et pour l'estimation du premier symbole, on déduit 4.6 de 4.3.

$$\lambda_{1,i} = y^{u}_{1,i} + \sum_{n=1}^{3} \left(y^{p}_{n,i+\alpha_{1,n}} \Diamond \sum_{\substack{k=1 \\ k\neq 1 \\ \alpha_{1,n}-\alpha_{k,n}\geq 0}}^{3} \Diamond y^{u}_{k,i+\alpha_{1,n}-\alpha_{k,n}} \Diamond \sum_{\substack{k=1 \\ k\neq 1 \\ \alpha_{1,n}-\alpha_{k,n}<0}}^{3} \Diamond \lambda_{k,i+\alpha_{1,n}-\alpha_{k,n}} \right) \tag{4.6}$$

Or, en se référant à la matrice de connexions on identifie les valeurs suivantes :

$$\alpha_{1,1} = 0; \quad \alpha_{1,2} = 0; \quad \alpha_{1,3} = 5;$$

De même

$$\alpha_{11} - \alpha_{21} = -1; \quad \alpha_{11} - \alpha_{31} = -5$$
$$\alpha_{12} - \alpha_{22} = -3; \quad \alpha_{12} - \alpha_{32} = -2$$
$$\alpha_{13} - \alpha_{23} = 5; \quad \alpha_{13} - \alpha_{33} = 5$$

On peut donc en conclure l'expression finale suivante :

$$\begin{aligned} \lambda_{1,i} = {} & y^{u}_{1,i} + y^{p}_{1,i+0} \Diamond \lambda_{2,i-1} \Diamond \lambda_{3,i-5} \\ & + y^{p}_{2,i+0} \Diamond \lambda_{2,i-3} \Diamond \lambda_{3,i-2} \\ & + y^{p}_{3,i+5} \Diamond y^{u}_{2,i+5} \Diamond y^{u}_{3,i+5} \end{aligned} \tag{4.7}$$

De la même manière, on exprime les deux autres symboles estimés que l'on donne dans 4.8 et 4.9.

$$\begin{aligned} \lambda_{2,i} = {} & y^{u}_{2,i} + y^{p}_{1,i+1} \Diamond y^{u}_{1,i+1} \Diamond \lambda_{3,i-4} \\ & + y^{p}_{2,i+3} \Diamond y^{u}_{1,i+3} \Diamond y^{u}_{3,i+1} \\ & + y^{p}_{3,i+0} \Diamond \lambda_{1,i-5} \Diamond y^{u}_{3,i+0} \end{aligned} \tag{4.8}$$

$$\begin{aligned} \lambda_{3,i} = {} & y^{u}_{3,i} + y^{p}_{1,i+5} \Diamond y^{u}_{1,i+5} \Diamond y^{u}_{2,i+4} \\ & + y^{p}_{2,i+2} \Diamond y^{u}_{1,i+2} \Diamond \lambda_{2,i-1} \\ & + y^{p}_{3,i+0} \Diamond \lambda_{1,i-5} \Diamond y^{u}_{2,i+0} \end{aligned} \tag{4.9}$$

Le rôle du FIFOULP est de réaliser le calcul des $\psi_{j,n,i}$, on doit donc exprimer ces J données en les identifiant dans les formules précédentes.

$$\begin{aligned}
\psi_{1,1,i} &= y^{p}_{1,i+0} \Diamond \lambda_{2,i-1} \Diamond \lambda_{3,i-5} \\
\psi_{1,2,i} &= y^{p}_{2,i+0} \Diamond \lambda_{2,i-3} \Diamond \lambda_{3,i-2} \\
\psi_{1,3,i} &= y^{p}_{3,i+5} \Diamond y^{u}_{2,i+5} \Diamond y^{u}_{3,i+5} \\
\psi_{2,1,i} &= y^{p}_{1,i+1} \Diamond y^{u}_{1,i+1} \Diamond \lambda_{3,i-4} \\
\psi_{2,2,i} &= y^{p}_{2,i+3} \Diamond y^{u}_{1,i+3} \Diamond y^{u}_{3,i+1} \\
\psi_{2,3,i} &= y^{p}_{3,i+0} \Diamond \lambda_{1,i-5} \Diamond y^{u}_{3,i+0} \\
\psi_{3,1,i} &= y^{p}_{1,i+5} \Diamond y^{u}_{1,i+5} \Diamond y^{u}_{2,i+4} \\
\psi_{3,2,i} &= y^{p}_{2,i+2} \Diamond y^{u}_{1,i+2} \Diamond \lambda_{2,i-1} \\
\psi_{3,3,i} &= y^{p}_{3,i+0} \Diamond \lambda_{1,i-5} \Diamond y^{u}_{2,i+0}
\end{aligned} \tag{4.10}$$

On remarque bien qu'il s'agit d'opérations addmin effectuées sur des données que l'on doit prélever à des positions précises dans des registres à décalage. La méthode la plus simple de parvenir à nos fins serait d'utiliser un FIFO par symbole, ceci reviendra donc à utiliser 3*J registres à décalage. Or, en recourant à la simplification préconisée dans la littérature [37] et [19], on est en mesure de diminuer le nombre de registres à 2*J. Ceci se fait en réalisant les opérations addmin au sein du FIFO des informations de parités, en leur associant les informations en rétroaction. Ainsi, pour disposer par exemple du symbole $\psi_{1,1,i}$, on utilisera la structure de la figure 4.5.

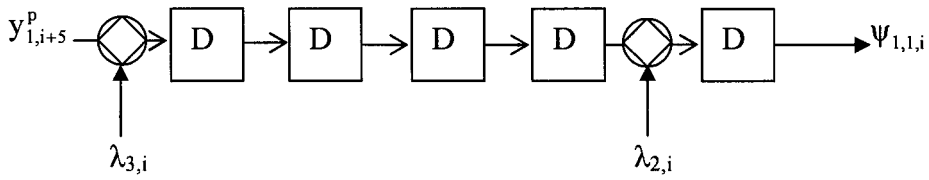

Figure 4. 5 Exemple d'insertion des addmin au sein d'un FIFO

À ce niveau de l'étude, l'unique difficulté qui reste est de savoir où positionner les opérateurs addmin dans le registre à décalage. La solution consiste à identifier la

position du symbole de parité associé comme opérande addmin avec une rétroaction des sorties, et de le considérer comme la référence à partir de laquelle on doit reculer en amont du registre à décalage d'autant d'éléments que nécessaire.

L'opération restante consiste à combiner les symboles de parités avec les symboles d'informations pour former chacun des J^2 symboles $\psi_{j,n,i}$. Il faut aussi réaliser que les symboles de parités ont été altérés, et comme ils sont utilisés dans les autres blocs de décodage, on doit utiliser un autre FIFO où on va les injecter pour garder la synchronisation des signaux. En outre, dans les FIFOULP des modules intermédiaire et terminal, les entrées des symboles reçoivent les valeurs estimées par le décodeur en amont.

4.3.4 Blocs de complément à deux

Les données issues du bloc FIFOULP doivent être sommées. Il est nécessaire de convertir les symboles d'une représentation en binaire signée, à une représentation en complément à deux. C'est cette représentation qui assure des résultats exacts dans le cas d'une somme algébrique. L'opération s'effectue en inversant tous les bits du symbole si le MSB est égal à un et en ajoutant un 1 au LSB. Si le MSB est nul, on ne procède à aucun changement.

Après sommation, les symboles en sortie, seront injectés dans les modules de décodage suivant, mais subiront aussi une rétroaction vers le bloc FIFOULP. Afin de préserver la cohérence des représentations, on doit procéder à une seconde opération de complément à deux.

Le sommateur sera donc encadré de deux blocs de complément à deux que l'on représente dans les figures 4.6(a) et 4.6(b).

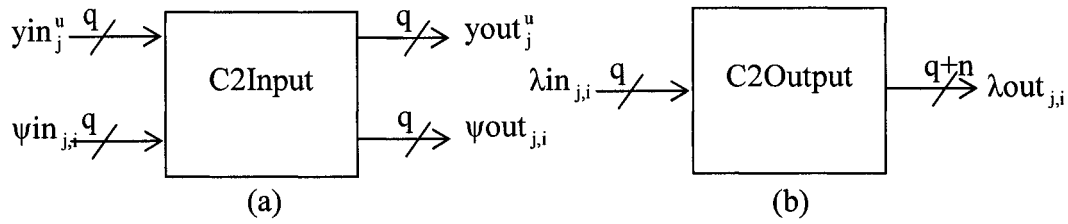

Figure 4. 6 Schéma bloc des complément à deux en entrée (a) et sortie (b) du sommateur

La logique à réaliser étant purement combinatoire, il n'est pas nécessaire d'utiliser des entrées d'horloge et de remise à zéro.

4.3.5 Bloc de sommation

Le sommateur est décomposable en J blocs de sommation identiques. Chaque bloc somme J symboles $\psi_{j,n,i}$ avec le symbole d'information $y^u_{j,i}$ associé, pour fournir en sortie l'information approximée $\lambda_{j,i}$. L'addition de deux symboles binaires peut conduire à un dépassement (overflow) du nombre de bits utilisé pour la quantification. Il est donc nécessaire de procéder à une extension du nombre de bits de quantification pour tenir compte de ce dépassement. Ainsi, pour chaque addition de deux opérandes sur q bits, on doit d'abord les étendre à q+1 bits avant de procéder au calcul. Dans le cas d'une représentation en complément à deux, il suffit de recopier le MSB et de le concaténer comme nouveau MSB du symbole étendu d'un bit.

Le second point à prendre en considération est la structure d'addition, qui permet de disposer en sortie d'un résultat représenté sur un nombre minimal de bits. Il existe deux approches possibles :

- Une addition en cascade comme on peut le voir sur l'exemple de la figure 4.7(a). Cette réalisation a l'inconvénient de manquer de parallélisme. Ceci engendrera un très grand temps de propagation pour la sommation.

- Une addition suivant un arbre dyadique. Cette structure représentée par la figure 4.7(b) est fortement parallèle. Dans le cas où le nombre d'éléments à additionner est impair, l'élément en excès sera simplement étendu d'un bit à chaque niveau où on ne peut l'apparier avec un autre opérande. L'exemple de la figure 4.7(c) est donné pour un cas de trois entrées à sommer.

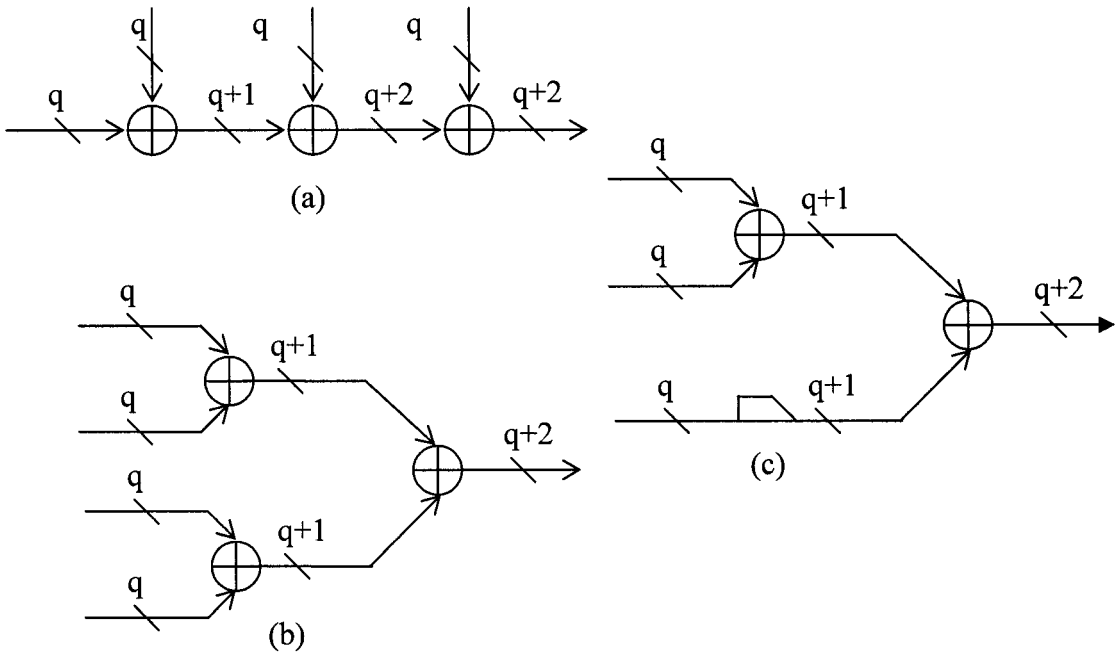

Figure 4. 7 Structure d'addition en cascade (a), dyadique (b) et exemple d'utilisation (c)

Encore une fois, l'addition étant une opération purement combinatoire, on n'utilisera pas d'entrée d'horloge ni de remise à zéro. La figure 4.8 est la représentation du bloc de sommation.

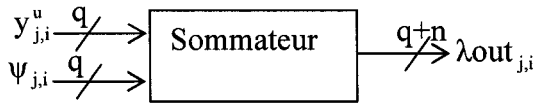

Figure 4. 8 Schéma bloc du sommateur

4.3.6 Bloc de saturation

Les données issues du sommateur sont passées par un bloc de complément à deux qui agit sur J symboles quantifiés sur q+n bits, avec n le nombre de bits ajoutés aux différents étages de l'addition pour tenir compte du dépassement de capacité. Ces symboles vont être utilisés dans des blocs qui ne traitent que des données sur q bits. Il est donc nécessaire de trouver une méthode pour rendre l'ensemble compatible, sans perturber le fonctionnement du décodage. L'idée est de saturer les sorties grâce à un bloc adéquat. Pour cela, on doit détecter si la valeur se situe en dehors de la plage de valeurs permise par les q bits de quantification, auquel cas, on prélève le MSB et on complémente par q-1 bits à un. En fait, le seul problème survient au niveau des addmin, or en saturant les sorties du sommateur, après complément à deux, cela ne change pas le résultat que l'on obtient.

La procédure adoptée consiste à détecter que l'on est en dehors de la plage des valeurs en testant les bits ajoutés hors MSB. En effet, il suffit qu'un de ces bits soit à 1 pour que l'on sature. Si par contre aucun de ces bits n'est égal à 1, donc que l'on n'est pas en dehors de la plage de valeurs permises, il suffit de recopier le MSB et les q-1 bits à partir du LSB.

À titre d'exemple, soit J=3 et q=3, on doit ajouter deux bits pour tenir compte du dépassement. On se retrouve en entrée du saturateur avec des symboles sur 5 bits. Si l'on décide de noter le LSB comme l'élément zéro, et le MSB l'élément quatre, il suffit de tester les bits deux et trois. Si l'un des deux est à un, exemple "10100", il suffit alors de fournir en sortie la valeur "111". Si par contre aucun des deux bits n'est à un, exemple "00010", on devra fournir en sortie du saturateur la valeur "010".

Le saturateur possède donc J entrées sur q+n bits et J sorties sur q bits. Il ne nécessite pas de synchronisation ni de remise à zéro.

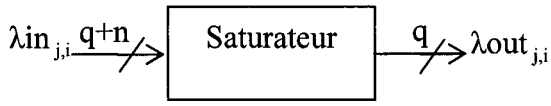

Figure 4. 9 Schéma bloc du saturateur

4.3.7 Assemblage des différents blocs

Il est maintenant possible d'assembler les différents blocs afin de constituer les modules de décodage frontal, intermédiaire et terminal. Les figures 4.10, 4.2 et 4.11 donnent une description détaillée de ces modules en utilisant les blocs élémentaires que nous avons décris. Les signaux ont été renommés en fonction de ce qui a été utilisé dans le code VHDL.

Il est à noter, que les trois structures se ressemblent beaucoup, et que les différences sont minimes. Ainsi, le bloc FIFOULP prend en entrées les symboles de parité, une rétroaction des symboles estimés, un signal d'horloge et de remise à zéro dans les trois figures. Dans le module frontal, on trouve comme entrées additionnelles les symboles d'informations, alors que dans les autres modules ils sont remplacés par les symboles approximés délivrés par les décodeurs en amont. En sortie, on dispose des $\psi_{j,n,i,}$ mais aussi des $y^{u}_{j,i}$ retardés dans le cas du décodeur frontal. Ces données sont ensuite traitées par complément à deux, avant d'être sommées pour donner les $\lambda_{j,i}$. On procède ensuite à un complément à deux sur q+n bits suivi de la saturation avant de réinjecter les symboles approximés dans le FIFOULP et les propager vers le décodeur en aval.

Il est nécessaire d'ajouter un bloc de FIFO dans le module frontal, car comme on l'a déjà présenté, on a modifié les symboles de parité fournis par le démodulateur en procédant à l'intégration des opérateurs addmin entre les mémoires. Ceci annule la simplification de complexité que l'on a voulu réaliser pour les modules de décodage frontal et intermédiaire. Par contre, le module terminal est moins complexe en raison de cette méthode.

Les symboles d'informations sont retardés dans le module frontal par un FIFO intégré au FIFOULP. Ils sont ensuite changés en complément à deux, avant d'être soumis au sommateur, mais aussi orientés comme sorties du module. À chaque module intermédiaire utilisé, ses données sont retardées dans un FIFO, avant d'être sommées et propagées en aval. Une fois dans le décodeur terminal, elles sont retardées et sommées.

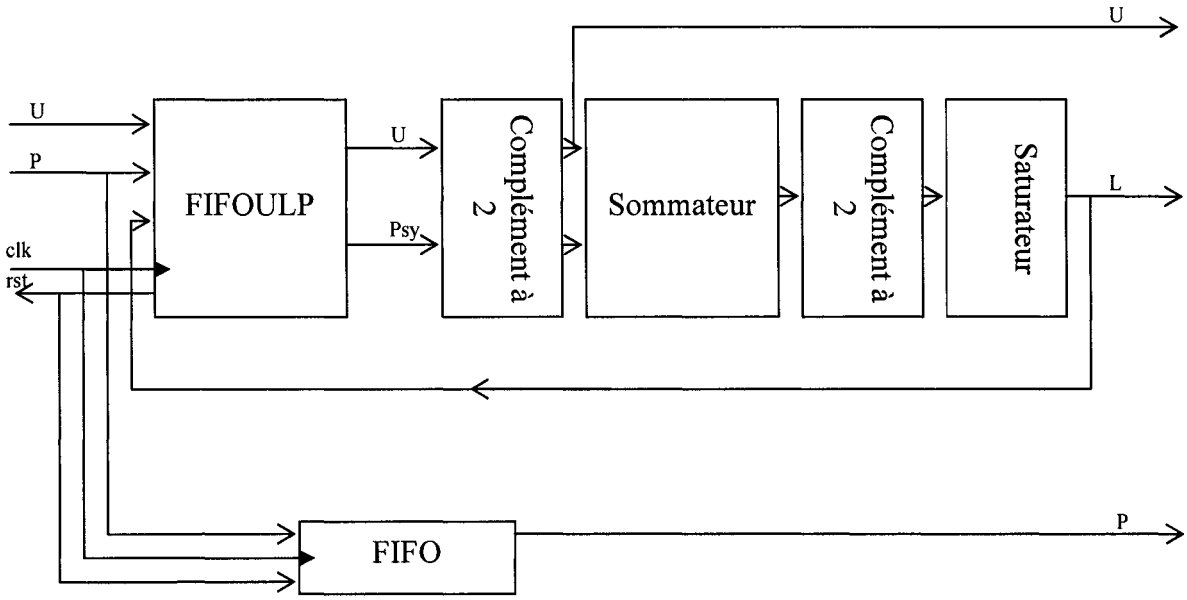

Figure 4. 10 Structure bloc du décodeur frontal

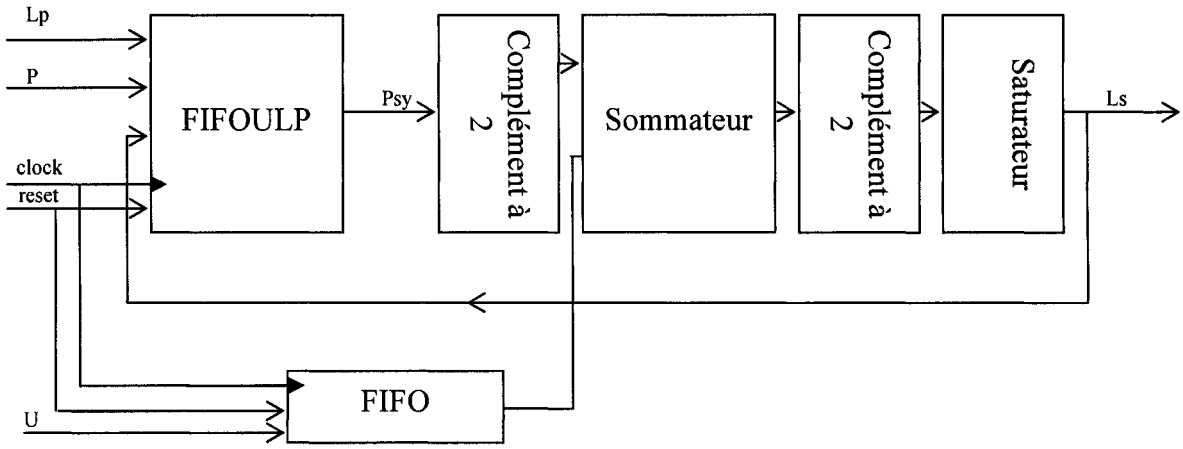

Figure 4. 11 Structure bloc du décodeur terminal

4.4 Codage en VHDL et validation fonctionnelle

Le langage VHDL est un langage très puissant permettant la description de systèmes numériques à divers niveaux d'abstraction. Il dispose aussi de caractéristiques que l'on trouve dans les divers langages de haut niveau.

La première étape a été de bien définir notre architecture en la décomposant en modules, blocs et fonctions, et leur description détaillée en termes d'opérations à réaliser et d'interfaces entrées/sorties. On est en mesure maintenant de tirer pleinement profit de la capacité du langage pour produire un code facilement paramétrable et réutilisable.

4.4.1 Hiérarchie des fichiers VHDL et leur description

Le code VHDL a été décomposé en un ensemble de fichiers reliés suivant une hiérarchie bien définie que l'on présente dans la figure 4.12.

Lors de la phase de compilation, pour la simulation et pendant la synthèse, il est primordial que cette structure soit respectée. Les fichiers sont classés par ordre de préséance du haut vers le bas, et c'est dans cet ordre que l'on va les passer en revue.

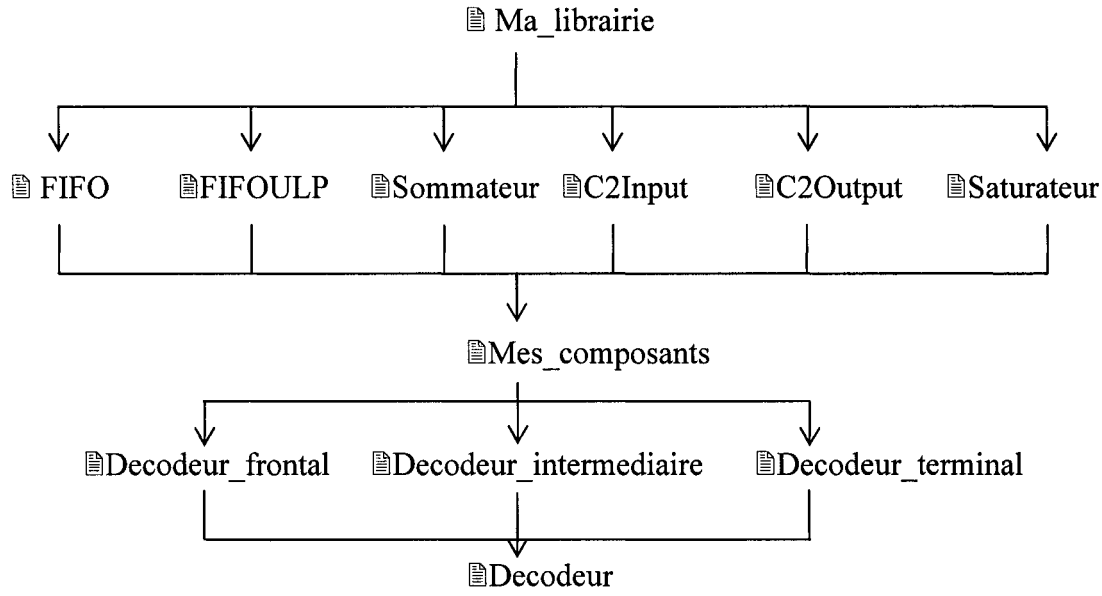

Figure 4. 12 Hiérarchie des fichiers VHDL

✓ ***Ma_librairie.vhdl***

Le VHDL offre la possibilité d'utiliser des bibliothèques qui regroupent la description d'éléments communs à un ensemble de fichiers du projet. Dans notre cas, on a rassemblé des données utiles pour toute l'architecture, à savoir :

- la taille de la quantification q,
- la profondeur des FIFO Maxi,
- la fonction pour l'addition,
- des fonctions addmin,
- une fonction pour le complément à deux,
- une fonction pour la saturation.

L'utilisation de la déclaration de la taille de quantification dans la bibliothèque au lieu d'utiliser des *generic* est intéressante dans la mesure où il n'y a plus qu'à changer la valeur dans un seul fichier, pour que la modification se répercute dans tout le projet.

L'utilisation de *generic* quant à elle, détermine des constantes locales que l'on doit changer dans chaque fichier.

L'utilisation des fonctions est aussi justifiée par la grande souplesse qu'on en tire. En effet, l'autre moyen dont on dispose pour arriver au même résultat, serait de déclarer des entités que l'on utilisera. Or, cette approche nécessite de déclarer des entrées/sorties à largeur fixe, ce qui n'est pas le cas pour les fonctions qui ont besoin simplement de savoir le type du signal à utiliser. On est donc en mesure de créer une seule fonction qui s'adaptera à n'importe quelle largeur de bus d'entrée, et ceci, par simple appel.

L'autre point fort que l'on a utilisé, est la possibilité de surcharger les fonctions. Ceci revient à écrire plusieurs de ces dernières, ayant le même nom mais qui opèrent de façon différente et sont capables de le faire suivant le type et le nombre de valeurs qu'on leur assigne. Ainsi, pour la fonction addmin, on a besoin de fonctions addmin manipulant un nombre d'entrées, allant de deux à J. On a donc conçu une fonction addmin à deux entrées, que l'on réutilise pour obtenir des fonctions surchargées manipulant jusqu'à J éléments. On donne dans ce qui suit un exemple de code VHDL pour l'opérateur addmin sur deux opérandes, puis sur trois opérandes.

```
function ADM  ( l, r : echantillon  ) return echantillon is
begin
        if (l(l'high-1 downto 0) < r(r'high-1 downto 0)) then
                return((l(l'high) xor r(r'high))&l(l'high-1 downto 0));
        else
                return((l(l'high) xor r(r'high))&r(r'high-1 downto 0));
        end if;
end;
```

```
function ADM  ( l, r,s : echantillon  ) return echantillon is
begin
                return(ADM(l,ADM(r,s)));
end;
```

Pour rendre la librairie visible dans un fichier VHDL, il suffit de l'ajouter comme pour les librairies standard en écrivant :

use work.ma_library.all;

✓ ***FIFO.vhdl***

Ce fichier contient la description de l'entité FIFO. Il y a J entrées de type *std_logic_vector* de largeur q. On définit tout d'abord un nouveau type, TU, constitué d'un tableau de Maxi éléments *std_logic_vector* de largeur q. On déclare ensuite J signaux de type TU et on réalise J *process* qui procèdent à des décalages. Pour J=3, on a Maxi=5 et on donne l'exemple d'un *process* de fifo réalisant le décalage comme suit :

```
Process (clk)
        Begin
                If (clk'event and clk='1') then
                        If(reset='1') then
                                for i in fifoU1'range loop
                                        fifoU1(i)<=(others=>'0');
                                end loop;
                        else
                                fifoU1(4 downto 1)<=fifo(3 downto 0);
                                fifoU1(0)<=Ue1;
                        end if;
                end if;
        end process;
```

✓ ***C2Input.vhdl et C2Output.vhdl***

Ces deux fichiers contiennent respectivement J^2+J et J conversion en complément à deux. On fait simplement appel à la fonction déclarée dans la librairie et dont on donne le code.

```
function C2 (a: std_logic_vector) return std_logic_vector is
begin
        if (a(a'high)='1') then
                return(a(a'high)&("0"-a(a'high-1 downto 0)));
        else
                return(a);
        end if;
end C2;
```

C'est le code qui est recommandé par Xilinx™ dans la note d'application [38]. On notera que la synthèse est plus ou moins optimale suivant l'outil que l'on utilise et ce d'après la même note [38].

✓ ***Sommateur.vhdl***

Le fichier de sommation contient la déclaration de deux entités. Une première appelée addition et qui somme les J $\psi_{j,n,i}$ et leur $y^{u}_{j,i}$ associé. Cette structure est réalisée en faisant appel à la fonction ADD qui se charge de procéder à l'extension de bits pour tenir compte du dépassement de capacité. Ce code s'est inspiré de la note d'application [38] de Xilinx™.

Cette fonction est codée de la façon suivante :

```
function ADD (a,b: std_logic_vector) return std_logic_vector is
begin
        return ((a(a'high)&a)+(b(b'high)&b));
end ADD;
```

On remarquera que rien dans la fonction ne la destine à être utilisée avec une taille de vecteur spécifique, au contraire elle est générale.

La seconde entité que l'on trouve dans le même fichier représente la structure globale qui va instancier J composants de l'entité addition précédente.

✓ ***Saturateur.vhdl***

Le saturateur est constitué de J entrées de largeur q+n et de J sorties de largeur q. Il s'agit ici encore d'utiliser la fonction décrite dans la librairie de base. Cette fonction est donnée en exemple pour une entrée et une sortie pour J=3 et n'importe quelle taille de quantification.

```
function sature (a: std_logic_vector) return std_logic_vector is
begin
        if ((a(a'high-1) or a(a'high-2))='1') then
                return (a(a'high)&"11");
        else
                return (a(a'high)&a(N-2 downto 0));

        end if;
end sature;
```

✓ ***Composants.vhdl***

L'utilité de ce fichier est de rassembler toutes les déclarations de composants devant intervenir dans les trois modules de décodages. Ainsi, au lieu d'écrire trois fois les mêmes *components*, on les rassemble dans un seul fichier que l'on va rendre visible dans les fichiers concernés grâce à la déclaration suivante :

use work.Composants.all;

✓ ***Fichiers Decodeur_frontal (intermediaire, terminal).vhdl***

Ces fichiers contiennent le code décrivant les différents modules. En fait, il s'agit d'utiliser les blocs entrant dans la réalisation de chaque module et de les relier avec les signaux adéquats. Il s'agit de l'utilisation d'un niveau de description dit structurel du VHDL.

✓ ***Decodeur.vhdl***

L'entité décodeur, est le bloc supérieur qui englobe l'ensemble des modules. Il est constitué au minimum d'un décodeur frontal et d'un décodeur terminal pour le cas de deux itérations de décodage. Dans le cas où on a m itérations, il suffit d'utiliser un bloc frontal, m-2 blocs intermédiaires et enfin un bloc terminal. Le code VHDL donné en exemple, utilise l'instruction *generate* très utile dans ce cas pour J=3.

```
instdecfron : decodeur_frontal port map(clk,rst,U1,U2,U3,P1,P2,P3,U1(0),U2(0),U3(0),
                    P1(0),P2(0),P3(0),L1(0),L2(0),L3(0));
decintergen:for i in 1 to N-2 generate
    instdecint : decodeur_intermediaire port map (clk, rst, U1(i-1),U2(i-1),U3(i-1),
                    P1(i-1),P2(i-1), P3(i-1),L1(i-1),L2(i-1),L3(i-1),U1(i),U2(i),U3(i),
                    P1(i),P2(i),P3(i),L1(i),L2(i),L3(i));
            end generate;
```

instdecter : decodeur_terminal port map (clk,rst,U1(N),U2(N),U3(N),P1(N),P2(N),
P3(N),L1(N),L2(N),L3(N),L1,L2,L3);

La structure ainsi présentée, est intuitive et nous permet d'avoir une certaine souplesse pour généraliser la description VHDL pour n'importe quelle matrice de connections décrivant le CSO^{2}C-SS. On tient à faire remarquer, qu'un ensemble d'éléments additionnels peuvent être nécessaires pour interfacer le décodeur au démodulateur. En effet, si ce dernier fournit des données en série, on doit utiliser un circuit qui va les transposer en parallèle. On peut aussi avoir besoin d'adapter le rythme de l'horloge du système, à celui du fonctionnement du décodeur par le biais de diviseur de fréquence. Enfin, il est nécessaire de porter une décision sur l'information approximée en fin de décodage. Cela est spécifié dans l'algorithme de base par une comparaison par rapport au zéro, si $\lambda_{j,i} > 0$ alors on prend comme décision que la sortie associée vaut zéro, sinon elle vaut un. Dans une réalisation numérique, cela consiste tout simplement à prélever les MSB des différentes sorties quantifiées du décodeur terminal.

4.4.2 Validation fonctionnelle et procédures de vérification

Lors du développement des différents blocs et modules, il nous est apparu judicieux de procéder à des vérifications dès les premières phases du codage en VHDL. Ainsi, chaque fonction a été utilisée dans une entité, et un banc d'essai validant tous les cas d'entrées possibles a été réalisé. La même procédure a été reprise pour les blocs élémentaires avec des bancs simulant toutes les combinaisons possibles des entrées.

Au niveau des modules, il est clair qu'essayer toutes les combinaisons possibles est une opération qui demandera beaucoup de temps et de puissance de calcul. On s'est donc concentré sur la simulation des cas extrêmes que l'on qualifierait d'effet de bord. On a ainsi testé le cas où toutes les entrées sont au maximum positif, au minimum négatif, une alternance de ces deux cas, et quand tous les symboles sont nuls. Pour compléter les

vérifications, on a généré un ensemble de données aléatoires sous Matlab qui ont été ensuite utilisées dans le banc d'essai pour comparer les sorties fournies par le code VHDL avec celles calculées sous Matlab. C'est une procédure aléatoire qui permet de déceler des comportements que l'on n'aurait pas prévu. Finalement, une simulation avec une injection d'erreurs par inversion de quelques bits d'information et de parité issus du codeur, permet de valider le pouvoir correcteur de notre réalisation comme on peut le voir sur l'exemple de simulation dans la figure 4.13.

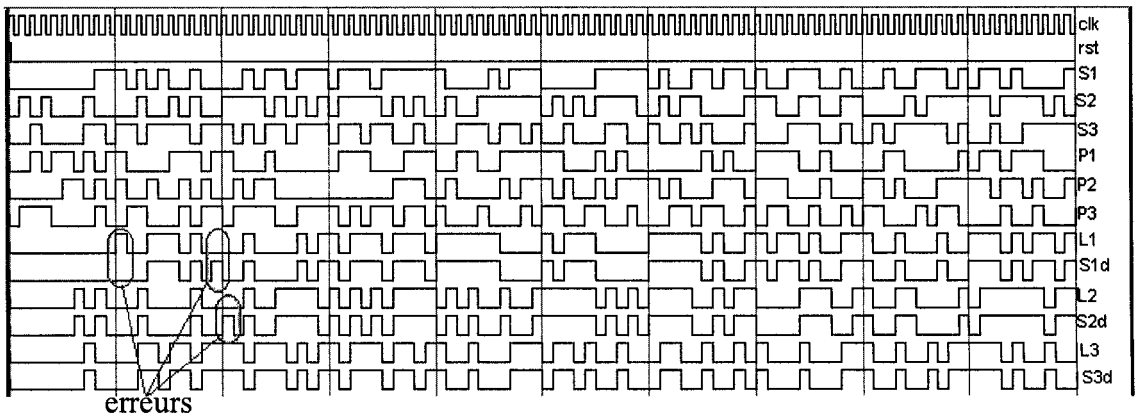

Figure 4. 13 Résultats graphiques d'une simulation du code VHDL

On peut clairement distinguer les signaux d'information Si, de parité Pi, la sortie du décodeur après le décodage Li que l'on compare avec les signaux d'entrées respectifs décalé de Maxi coups d'horloge. Les cercles représentent les bits en erreurs et leur correction.

On tient par contre à signaler que les simulations ont été réalisées avec un modèle de canal en VHDL qui se limite à concaténer q-1 bits aux différents bits. C'est-à-dire que la fiabilité reste la même pour l'ensemble des symboles.

4.5 Généralisation pour l'ensemble des décodeurs

Le code VHDL que l'on a présenté jusqu'à maintenant et dont en trouve l'intégralité dans l'annexe 1, a été réalisé pour valider l'architecture proposée. On a donc utilisé la matrice des connections pour J=3 qui offrait l'avantage d'être générale, sans être trop complexe et lourde à manipuler. Une fois l'architecture validée, on a procédé à sa généralisation pour l'ensemble des taux de codage possibles. Pour ce faire, on a choisi de concevoir un programme qui réalise tous les calculs nécessaires pour extraire la structure du décodeur. Le code a été réalisé en langage C et on peut en trouver copie en annexe 2.

Le principe du générateur de code VHDL est assez simple. En premier lieu, on doit fournir au programme, en ligne de commande, les données suivantes :

- taille du codeur J,
- nombre de bits de quantification q,
- chemin du fichier texte où la matrice des connections est entrée un élément par ligne, et ceci de gauche à droite et de haut en bas de la matrice,
- le nombre d'itérations m que l'on souhaite.

Le programme récupère ces données, affecte J, q et m à des variables internes, puis alloue de l'espace mémoire où seront copiées les données du fichier texte. Une fois cette opération terminée, on passe par pointeur la matrice α à une fonction qui calcule le maximum dans chaque ligne, puis le maximum dans la matrice que l'on a nommé Maxi. Arrivé à cette étape, on appelle différentes fonctions qui sont chargées de générer le code VHDL. Il existe une fonction par fichier VHDL que l'on crée, et l'opération de base est l'accès en écriture à ce fichier grâce à la fonction C *fprintf()*. La procédure de génération est simple et consiste simplement en des calculs et des boucles avec des accès en écriture à un fichier. Néanmoins, il existe certaines fonctions dont on doit expliquer les algorithmes utilisés et que l'on a développées sur la base de remarques et de tests.

4.5.1 Calcul du nombre de bits de dépassement de capacité

Il est important de déterminer le nombre de bits, n, à utiliser pour définir les sorties des sommateurs, entrées/sorties du complément à deux en aval du sommateur et les entrées du saturateur. On a donc procédé à l'étude de plusieurs cas de décodeur et des n bits d'extension nécessaires que l'on présente dans le Tableau II.

Tableau II

Nombre de bits de dépassement en fonction du code

J de	**Extensions**
2 à 3	+2
4 à 7	+3
8 à 15	+4
16 à ...	+5

Après analyse de ces cas on en a déduit l'algorithme dont on représente l'organigramme dans la figure 4.14.

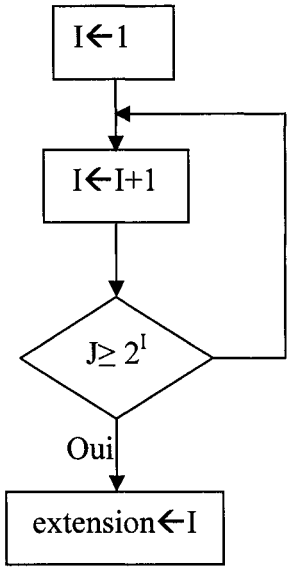

Figure 4. 14 Organigramme pour le calcul du nombre de bits d'extension nécessaires

4.5.2 Mise en place de la structure d'addition

Comme on l'a présenté auparavant, l'additionneur possède une structure en arbre dont on doit automatiser la représentation. En fait, il faut définir un certain nombre de signaux intermédiaires, leurs largeurs, puis les connecter.

On a adopté une décomposition en niveaux, où les signaux d'entrées correspondent au niveau zéro. Après, pour chaque addition dyadique, ou extension de signe pour un signal isolé, on augmente d'un niveau. Ainsi, pour les deux exemples que l'on montre dans la figure 4.15, on a respectivement un niveau et deux niveaux.

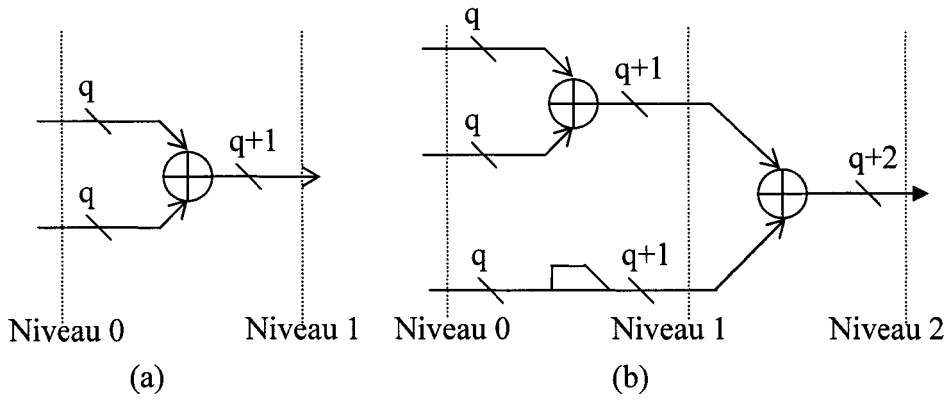

Figure 4. 15 Exemple de décomposition de l'addition en niveaux pour deux entrées (a) et trois entrées (b)

On a procédé à une étude par récurrence pour déduire l'algorithme dont on fournit le pseudo code.

```
T←1;
N←0;
R←J+1;
Faire
    Si R modulo 2 =0
        N←R/2;
    Sinon
        N←R/2+1;
    Fin Si;
    Pour I=1 à N pas 1 faire
        Ecrire(signal S[T][I]:std_logic_vector(q+(T-1)downto 0);)
    Fin Pour;
    T←T+1;
    R←N;
Tant que (N≠2);
```

Une fois les signaux définis, on doit procéder à la mise en place des connections. On utilise alors un second algorithme, dont les étapes sont décrites par pseudo code.

```
T←1;
N←1;
R←J+1;
Tant que (R≠1)faire
    Pour I=0 à 2*(R-1)pas 2
        Si (R≠2)
            Écrire S[N][T]<=ADD(S[N-1][I+1],S[N-1][I+2]);
            T←T+1;
        Sinon
            Écrire SE<=ADD(S[N-1][I+1],S[N-1][I+2]);
            T←T+1;
        Fin Si;
    Fin Pour;
    Si (R≠2)
        Écrire S[N][T]<='0'&S[N-1][R];
        T←T+1;
    Fin Si;
    R←T-1;
    T←1;
    N←N+1;
Fin Tant que;
```

Ainsi, on a procédé à l'élaboration du calcul automatisé de la meilleure structure en arbre pour un bloc d'addition. Le reste de la fonction est constitué de simples opérations d'écriture vers le fichier VHDL de la description des entités addition et sommateur.

4.5.3 Mise en place du FIFOULP

Il s'agit sans doute de la structure la plus complexe et celle qui a nécessité le plus d'effort pour la réalisation en VHDL pour le cas de J=3, mais aussi qui a été la plus délicate à automatiser.

La procédure d'automatisation se base sur la remarque que chaque colonne de la matrice des connexions contient au moins un élément égal à zéro, et que les autres valeurs traduisent l'emplacement où l'on doit placer les rétroactions. Ainsi, dans la matrice J=3, on peut affirmer que la première colonne décrit les connexions vers le FIFO du premier élément de parité, et que les éléments de la ligne deux et trois dans cette même colonne donnent la position où l'on doit placer la rétroaction des symboles estimés $\lambda_{2,i}$ et $\lambda_{3,i}$. On doit simplement s'assurer que la position n'est pas nulle et qu'elle ne se situe pas en tête du FIFO, auquel cas la seconde opérande -outre la valeur en rétroaction- ne sera plus écrite comme un élément FIFOP(i) mais comme une entrée directe du bloc P_{ei}. Il peut arriver que l'on trouve deux rétroactions différentes qui aboutissent au même emplacement du FIFO, dans ce cas on doit utiliser un opérateur addmin qui agira sur autant d'opérandes, ce qui élimine le recours à des signaux intermédiaires que l'on serait obligé d'utiliser pour connecter deux addmin à deux opérandes.

4.6 Conclusion

L'algorithme du décodeur CSO^{2}C-SS a été analysé pour en déduire une proposition d'architecture répondant aux nécessités du projet. On devait présenter un système répondant à un ensemble de critères, dont les plus importants sont la simplicité, la modularité, le parallélisme, généralisable au niveau de la structure ainsi qu'au niveau du nombre de bits de quantification et du nombre d'itérations de décodage, et enfin qui permet de bien estimer la complexité de la réalisation. L'approche proposée découle d'une réflexion et d'une concertation de plusieurs mois. Ainsi, une réalisation pour le

cas d'un décodeur de taille J=3 pour une quantification sur q bits a été le premier travail réalisé pour assimiler les étapes d'analyse et de description du décodeur. Il s'en est suivi une procédure de généralisation à d'autres décodeurs en utilisant le langage C après que l'architecture de base ait été totalement validée.

Deux remarques sont à faire en conclusion de ce chapitre. Premièrement, il a été nécessaire de réaliser une simulation de base sous Matlab du décodeur pour J=3 afin de visualiser son fonctionnement, le flux de données, mais surtout les plages dynamiques dans lesquelles les symboles évoluent. Il est d'ailleurs possible de réaliser un modèle sous Simulink, puis de passer en VHDL grâce à des outils automatisés comme on peut le voir dans [39][40]. Néanmoins, le code obtenu ne sera pas à notre avis de bonne qualité, mais présentera un point de départ que l'on pourra optimiser. Deuxièmement, on a remarqué que la majorité des outils de conception et simulation numérique, disposent d'un interpréteur de script Tcl. Il aurait peut être été plus facile d'utiliser ce langage de plus haut niveau que le C, ce qui peut s'avérer plus attrayant en terme de temps de développement. Un second langage que l'on rencontre aussi dans l'industrie est le Perl, qui n'a pas été utilisé en raison de la nécessité de disposer d'un interpréteur spécifique sur l'ordinateur, contrairement à l'interpréteur Tcl qui est disponible sur les stations de développement numérique et au fichier exécutable qui sont des binaires ne nécessitant pas d'interpréteur.

CHAPITRE 5

RÉSULTATS ET ANALYSES

5.1 Introduction

La réalisation telle qu'on l'a présentée dans le chapitre précédent, peut être grandement raffinée. Bien que parfaitement fonctionnel, le système reste encore non optimisé pour une programmation de FPGA. Si certains éléments semblent être bien gérés par les outils de synthèse, tels les additionneurs et les comparateurs par exemple, d'autres blocs du système doivent faire l'objet de plus de rigueur.

Le but de ce chapitre est de présenter les améliorations que l'on a apportées à l'architecture pour la rendre plus performante et moins complexe. On a choisi délibérément de ne pas inclure cette approche dans le chapitre de base puisque la conception se fait en deux étapes : la génération du code HDL pour la simulation et celui pour la synthèse. Si le but du premier type est de prouver un concept ou de spécifier une architecture, il est rare que le code qui en découle donne par synthèse un «Netlist» optimisé.

Deux modifications importantes seront apportées au décodeur. La première vise à diminuer la complexité des FIFO par une réécriture du code de ces derniers. La seconde modification visera à augmenter la vitesse de fonctionnement du circuit final. Dans ce chapitre, on procédera à une étude comparative entre l'estimation théorique de la complexité et celle qui sera délivrée par les synthétiseurs. Le chapitre se termine par la présentation des résultats finaux sur la performance en erreur du décodeur.

5.2 Optimisation de l'architecture du décodeur

5.2.1 Modification du code HDL des FIFO

Le code VHDL pour décrire un FIFO tel qu'on l'a présenté précédemment, conduit à une architecture dont la complexité est excessive et la consommation de puissance importante. En décrivant la structure avec un signal de remise à zéro synchrone, les différents synthétiseurs n'utilisent pas des ressources adaptées pour ce type de réalisation. En effet, ces derniers infèrent une bascule D par élément à mémoriser. Quoique le recours à des bascules soit optimum du point de vue fréquence de fonctionnement maximale, il reste que les FPGA Xilinx™ de la famille Spartan et Virtex, disposent d'éléments plus intéressants dans leur structure interne. Ces circuits programmables offrent la possibilité d'utiliser la mémoire des LUT comme registres à décalage sur 16 bits (Shift Left Register 16, SRL16). De plus, on est capable de combiner plusieurs SRL16, en série ou en parallèle, pour disposer de registre FIFO de n'importe quelle profondeur et n'importe quelle largeur [41][42].

Ainsi, pour les Virtex-II, chaque CLB est constitué de quatre tranches, et chaque tranche contient deux bascules D et deux LUT. Si l'on désire par exemple un FIFO de profondeur cinq et de largeur trois, comme dans le cas du décodeur J=3, cela nécessitera dans un cas trois LUT en SRL16 soit 1.5 tranche, et dans un second cas quinze bascules D donc 7.5 tranches, d'où un rapport de cinq.

La nouvelle description en VHDL découle de la structure même de ces SRL16 [43]. Chaque LUT dans les Virtex II est configurable comme un registre à décalage sur une profondeur maximale de seize bits, dont l'opération d'écriture est synchronisée par un signal d'horloge. On dispose au besoin d'une entrée pouvant servir à inhiber le signal d'horloge. Il est possible de lire en sortie de la LUT n'importe lequel des seize bits en mémoire, grâce à un bus d'adresses sur quatre bits. Il est important de retenir que l'on ne

peut mettre à zéro ou à 1 n'importe quel bit mémorisé. Ceci explique pourquoi l'outil de synthèse n'est pas capable d'utiliser cette structure quand on utilise le code VHDL non modifié.

On notera aussi que la lecture des SRL16 est asynchrone, mais que l'on dispose d'une bascule pour pouvoir synchroniser cette sortie. Si on veut réaliser par exemple un FIFO sur dix bits, alors l'adresse contenue dans le bus de lecture pointera sur le neuvième bit de la LUT, et le dixième élément sera mémorisé et synchronisé par la bascule D associée à la LUT. De ce fait, en plus de synchroniser le signal, on utilise un élément qui a un temps de réaction horloge-à-sortie (clock-to-out) nettement inférieur à celui des cellules mémoires. La figure 5.1 représente le chaînage qui est utilisé dans un CLB pour disposer d'un registre à décalage d'un bit sur une profondeur de 128.

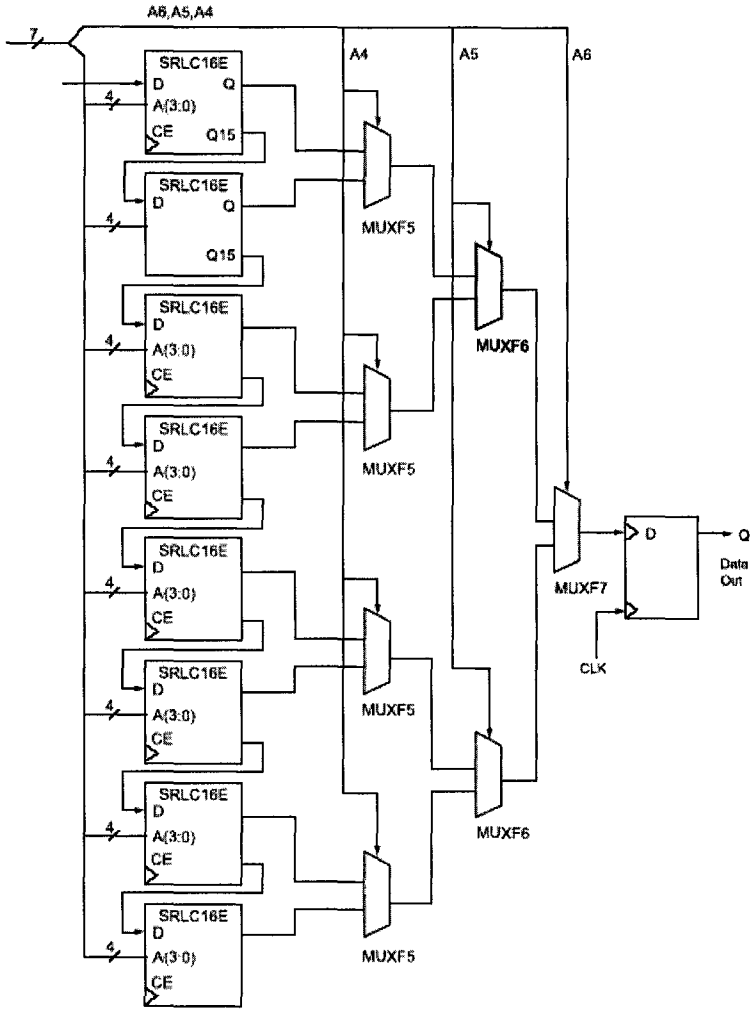

Figure 5. 1 Mise en place d'un registre à décalage de profondeur 128 bits à l'aide des SRL16

Après plusieurs tests, on a pu conclure qu'avec une bonne écriture du code VHDL, on est en mesure d'avoir une architecture optimisée. Tous les synthétiseurs sont capables d'inférer des SRL16 à partir du nouveau code sans ajout de directives propres à chaque outil. Ainsi le code suivant [38] donne un exemple de la nouvelle façon de coder un registre à décalage.

```
Process(clk)
begin
        if (clk'event and clk='1')
                Y<=Y(62 downto 0)&Input;
        End if;
end process;
```

Ceci génère un registre à décalage de profondeur 64. Il faut remarquer que le synthétiseur se charge toujours de synchroniser la sortie en mettant une bascule à la fin. Il suffit donc d'enlever le signal de remise à zéro dans notre projet et de le remplacer par un autre mécanisme afin d'initialiser les registres.

5.2.2 Amélioration de la fréquence maximale

La fréquence maximale à laquelle un circuit numérique synchrone peut fonctionner est déduite du chemin critique. En effet, c'est le long de ce chemin que le temps de propagation des signaux est le plus élevé. Afin de bien fonctionner, le circuit doit être cadencé par une horloge dont la fréquence doit tenir compte de ce délai. Dans le cas contraire, les éléments sur le chemin critique auront un fonctionnement erratique.

L'idée de base pour augmenter cette fréquence est de limiter au mieux les délais de propagation à travers la logique combinatoire, puisque c'est cette variable qui est la plus facile à modifier sur les FPGA, et c'est elle qui fixe le chemin critique. La technique utilisée s'appelle «pipelining», elle consiste à subdiviser une opération combinatoire prenant beaucoup de temps, en une succession d'opérations élémentaires plus rapides que l'on isole entre elles par des bascules comme on peut le voir sur la figure 5.2.

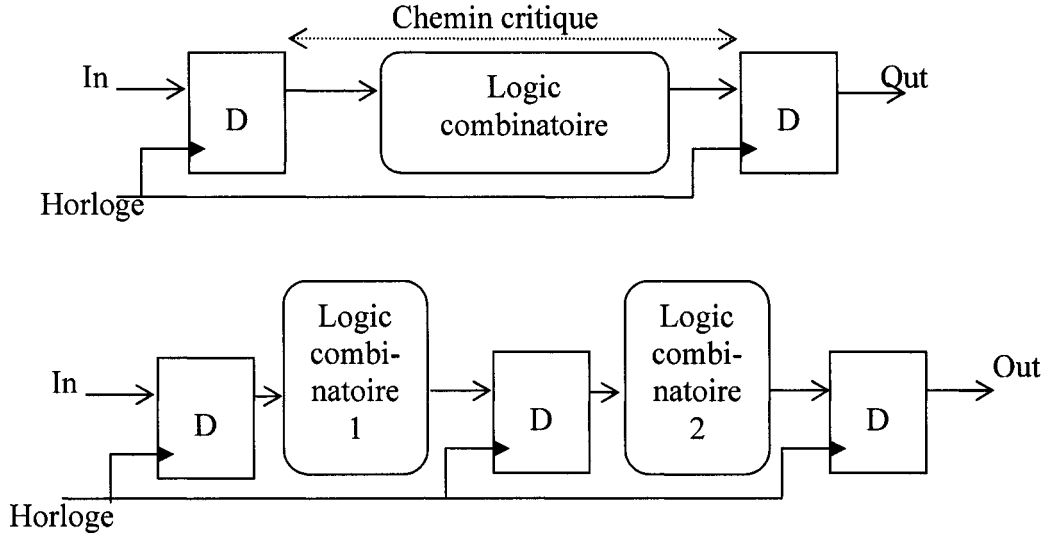

Figure 5. 2 Principe du Pipeline

Cette opération ne doit pas être utilisée à outrance, car elle conduit à une augmentation de la latence en sortie et de la complexité du système. En outre, il est nécessaire de vérifier le bon synchronisme des signaux après son introduction.

Dans notre architecture, il est clair que la succession de blocs combinatoires induit une dégradation de rapidité et que notre chemin critique passe par ces blocs. Le recours à un outil de synthèse a confirmé nos observations et orienté notre choix pour la mise en place des «pipelines». On a donc mis en place des banques de bascules comme on peut le voir dans la figure 5.3 pour le décodeur intermédiaire.

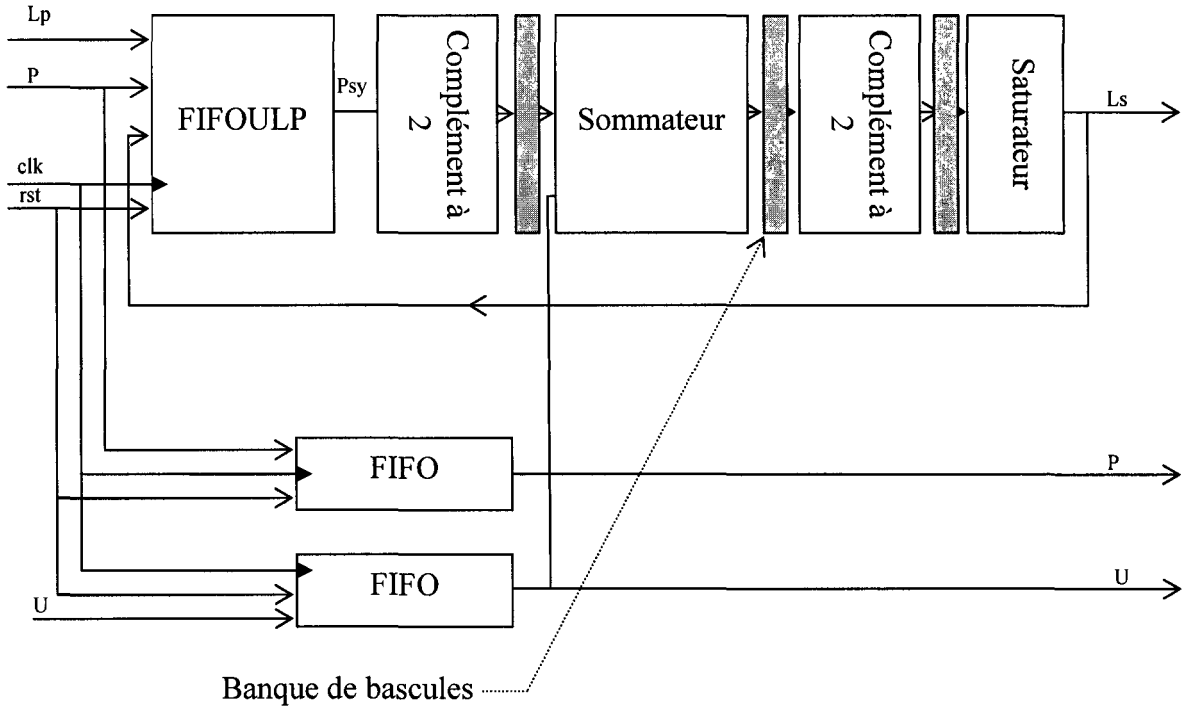

Figure 5. 3 Stratégie de pipelining pour le décodeur intermédiaire

Bien que ceci ait amélioré significativement les fréquences maximales estimées par le synthétiseur, il reste un chemin critique reliant les étages d'addition. Même si le FPGA dispose de ressources spéciales pour une addition, représentées par les chemins de propagation de retenues, la mise en cascade de plusieurs additionneurs n'est pas recommandée puisqu'on a un cumul de délais de propagation. Aussi, après chaque addition on a placé une unité de pipeline, et on a procédé à la synchronisation des signaux qui ne subissent que des extensions de signe. La figure 5.4 donne un exemple d'une entité d'addition pour J=2.

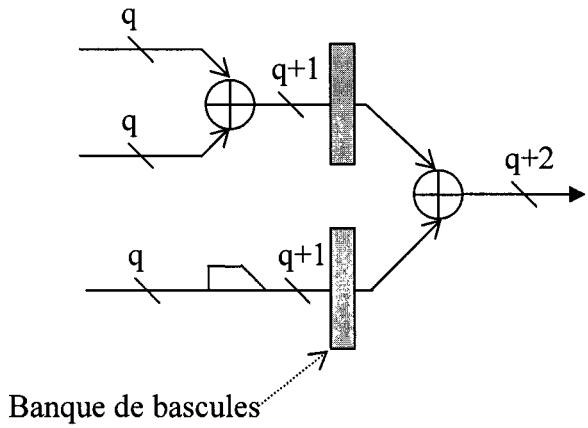

Figure 5. 4 Stratégie de pipeline pour le sommateur

5.2.3 Initialisation du système et sa synchronisation

Le problème qui surgit est relié à la modification apportée au FIFO. En effet, en éliminant le signal de mise à zéro, on doit trouver un autre mécanisme pour atteindre le même résultat, c'est-à-dire mettre notre système au départ dans un état que l'on connaît. L'idée de base est d'écraser les valeurs qui sont dans les mémoires des LUT par les valeurs en entrées. Cette opération est simple à mettre en place pour le bloc FIFO, mais pour le FIFOULP on doit inhiber pendant une certaine période la rétroaction. Les valeurs en rétroaction au début du fonctionnement des modules sont totalement aléatoires et en les soumettant à un calcul addmin on n'aura jamais de bons résultats. Il faut donc forcer ces valeurs à l'élément neutre pour l'opérateur addmin pendant un nombre de coups d'horloge égal à la profondeur des FIFOULP, Maxi. De cette façon, ces premières valeurs en entrées du FIFOULP vont venir charger les mémoires. On utilise donc un signal qui valide les données en entrées des trois modules de décodage que l'on appelle InOk. Une fois que ce signal passe au niveau haut, un compteur piloté par l'horloge est lancé et compte jusqu'à la profondeur des registres. Tant que la valeur du compteur est inférieure à cette dernière, sa sortie reste à l'état bas. Une fois la valeur atteinte, la sortie passera à l'état haut et restera indéfiniment à cet état. Si par contre l'état du signal InOk passe à l'état bas, le compteur est réinitialisé. La sortie du compteur

que l'on nomme OutOk va servir à la fois d'entrée à un multiplexeur et d'entrée InOk pour le module de décodage en aval. Le multiplexeur va associer aux signaux de rétroaction soit la valeur de sortie du saturateur quand le signal OutOk est à l'état haut, soit l'élément neutre dans le cas contraire. La figure 5.5 donne un schéma bloc du système ainsi mis en place et que l'on nomme activ2.

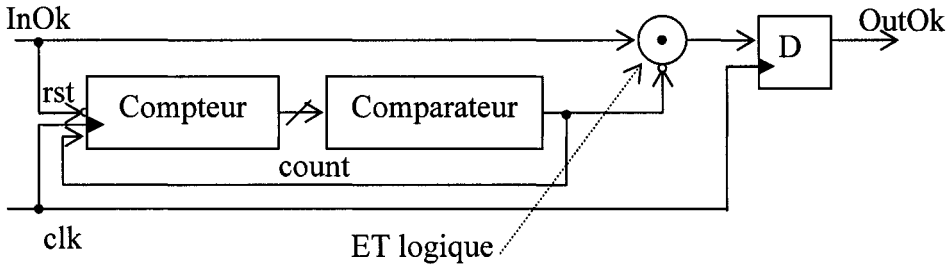

Figure 5. 5 Système activ2 d'initialisation des FIFOULP

L'ajout d'éléments de «pipeline» amène à une désynchronisation entre les différents flux de données. Ainsi, les valeurs estimées en sorties des décodeurs vont être en retard par rapport aux données en entrées et en sorties. Ce retard est égal au nombre de niveaux de «pipelines» mis en place y compris ceux dans le sommateur. Afin que la rétroaction soit combinée avec les bonnes données en entrées, on ne peut simplement les retarder. La solution consiste à faire fonctionner les banques de bascules à une fréquence multiple de celle des FIFO. Le coefficient de multiplication est égal au nombre de niveaux de «pipelines» que l'on a inséré. Pour ce faire, deux scénarios sont possibles :

- utiliser un multiplicateur de fréquence par le biais des ressources spéciales du FPGA, et donc avoir une horloge de base à basse fréquence,
- générer un signal validant les blocs FIFO dans le cas où l'horloge de base est à haute fréquence.

Bien que la première approche soit séduisante, elle possède l'inconvénient de nécessiter un code HDL non portable à d'autres familles de circuits programmables d'une part, et d'avoir deux domaines d'horloges dans notre architecture. C'est pour ces raisons que l'on a opté pour la seconde méthode.

La figure 5.6 présente le diagramme bloc du décodeur frontal dans sa version optimisée après ajout de toutes les modifications. Le décodeur intermédiaire est similaire, à la différence que le signal CE («Chip Enable») valide trois FIFO et l'activ2. On notera que les signaux de remise à zéro doivent alimenter toutes les banques de bascules ainsi que le sommateur et l'unité activ2.

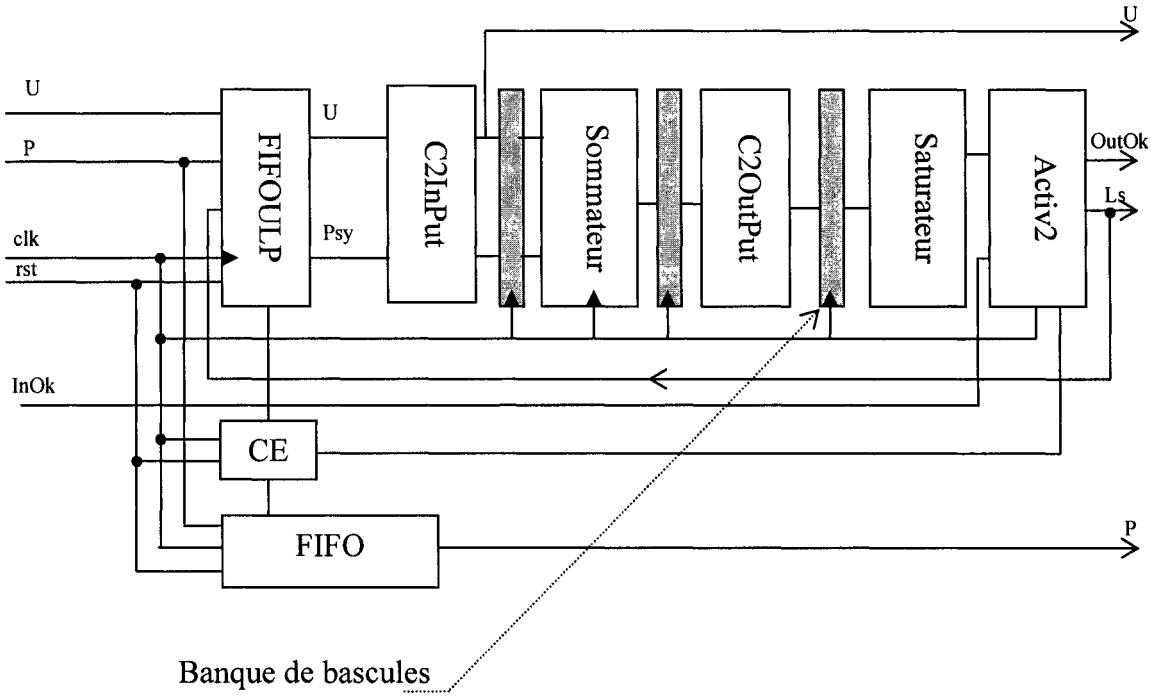

Figure 5. 6 Décodeur frontal optimisé

L'architecture ainsi modifiée se trouve optimisée à trois niveaux :

- la diminution de la complexité en utilisant les ressources spécifiques à la technologie cible,
- la diminution du nombre de bascules D, dans les FIFO et FIFOULP, qui limite la consommation de courant du circuit, puisque ces cellules devaient commutées ensemble en même moment d'où un appel de courant considérable,
- l'utilisation de la technique du «pipeline» qui améliore significativement la fréquence maximale à laquelle on peut faire fonctionner le décodeur. Ceci par

contre introduit des bascules D et la consommation de courant augmente avec la fréquence de fonctionnement.

Le générateur de code VHDL fournit en Annexe 2 intègre ces modifications, et permet de choisir entre une structure avec ou sans «pipeline».

5.3 Étude de la complexité du système

La complexité désigne les ressources nécessaires à la mise en place de l'algorithme sur un circuit. On peut la mesurer en différentes unités, tels en nombre de portes logiques ou de transistors par exemple. Dans le cas des FPGA de la famille Xilinx™, il est commun d'estimer le nombre de tranches qui seront utilisées. C'est entre autres cette indication que la majorité des synthétiseurs fournissent. Il faut par contre noter que cette évaluation ne reste valable qu'au sein d'une même famille de produits, en raison de la différence d'architecture quand on change de famille.

Le calcul de la complexité est conduit essentiellement pour s'assurer que le synthétiseur ne met pas en place une logique dont on ne veut pas d'une part, mais surtout pour choisir le circuit qui pourra contenir notre algorithme. Étant donné que l'on utilise comme cible la famille Virtex-II, on dispose de CLB contenant :

- deux LUT capable de servir pour générer aussi bien une fonction logique à quatre entrées et une sortie, qu'une mémoire de 16 bits,
- deux bascules D,
- un ensemble d'éléments pour le routage et la mise en place de fonctions spéciales.

Il faut aussi tenir compte du fait que le calcul que l'on réalise peut différer des résultats des synthétiseurs, et même les résultats diffèrent d'un synthétiseur à l'autre. En effet, chaque produit utilise des algorithmes qui lui sont propres et des degrés d'optimisation

différents. En outre, un ensemble de ressources sont utilisées pour répondre à certaines contraintes que l'on peut imposer, comme par exemple le nombre de sortances de certaines portes logiques et les ressources pour le routage. Tout ceci contribue grandement à rendre un calcul exact peu probable.

5.3.1 Méthode de calcul de la complexité pour les différents blocs

Certains éléments sont faciles à synthétiser, et il est donc aisé de trouver une formulation mathématique de leur complexité. D'autres blocs, comme les FIFO, sont plus ardus quant à l'estimation des ressources qui leur seront nécessaires. On va veiller à décrire toutes les méthodes nécessaires pour parvenir à une estimation valable.

- ***Fonction addmin***

La fonction addmin est appliquée sur deux opérandes de taille q bits, pour fournir en sortie une valeur de la même taille. Elle nécessite donc q LUT par opérateur, soit $\frac{q}{2}$ tranches.

- ***Fonction complément à deux***

Elle est appliquée sur des éléments en entrées du bloc de sommation de taille q et d'autres en sorties du même bloc de taille q+n, avec n le nombre de bits nécessaires pour tenir compte du dépassement de capacité. La fonction utilise l'ensemble des bits de l'opérande moins le MSB pour calculer son résultat. On peut donc estimer :

  - une complexité de $\frac{q-1}{2}$ tranches pour chaque opérande en entrée du sommateur,

- une complexité de $\frac{q+n-1}{2}$ tranches pour chaque opérande en sortie du sommateur.

- ***Fonction d'addition***

Cette fonction réalise tout d'abord une extension de signe par recopie du MSB, avant de procéder à l'addition. Ainsi, si l'on dispose de deux opérandes sur q bits en entrée, la sortie de l'additionneur sera de q+1 bits. De ce fait, et compte tenu des résultats dans la note d'application [38], la complexité sera de $\frac{q+1}{2}$ tranches.

- ***Fonction de saturation***

Le saturateur doit réaliser un OU exclusif sur n bits, puis avec le résultat activer q-1 multiplexeurs. Donc on utilisera $\frac{q-1}{2}$ tranches pour réaliser les multiplexeurs, ainsi qu'un nombre de LUT pour réaliser les OU exclusifs. On aboutit à la formule suivante pour la complexité de la saturation $\frac{q+\left\lfloor \frac{n}{4} \right\rfloor}{2}$ tranches, avec $\lfloor * \rfloor$ désignant la partie entière.

- ***Bloc FIFOP***

Les FIFOP sont implantés en utilisant les LUT ainsi que des bascules pour la synchronisation et l'amélioration des performances en vitesse. Chaque LUT ainsi que la bascule qui lui est associée peut contenir un registre à décalage de largeur unitaire et de profondeur maximale seize bits. Il suffit de calculer le nombre de LUT et de bascules nécessaires pour atteindre la profondeur voulue, et de multiplier le résultat par le nombre

de bits de quantification q. Un algorithme est proposé pour ces calculs et l'organigramme qui suit le présente.

Si $Max \leq 16$

$y \leftarrow 1$

Sinon

$y \leftarrow Modulo\ (\frac{Max}{16}) + 1$

Si Reste $(\frac{Max}{16}) \leq 3$

$x \leftarrow Reste\ (\frac{Max}{16})$

Sinon

$y \leftarrow y + 1$

$x \leftarrow 1$

Fin Si

Fin Si

Le résultat de cet algorithme, contenu dans les variables x et y, permet d'estimer la complexité en nombre de tranches qui vaut :

$$\frac{y}{2} + \frac{x - y}{2} \text{ tranches si } x \geq y$$

ou

$$\frac{y}{2} \text{ tranches sinon.}$$

- ***Bloc FIFOULP***

Ce bloc est constitué aussi de LUT et de bascules, donc assujetti à la même démarche pour calculer sa complexité que le bloc FIFOP. L'unique différence réside dans la nécessité de procéder par étapes, en isolant les registres à décalages qui se trouvent de

part et d'autres des addmins. Ainsi comme on peut le voir dans le cas de la figure 5.7, il faut appliquer le même algorithme vu précédemment aux registres un, deux et trois. Il ne restera plus qu'à sommer les valeurs ainsi obtenues en les multipliant par le nombre de bits de quantification pour avoir la complexité totale.

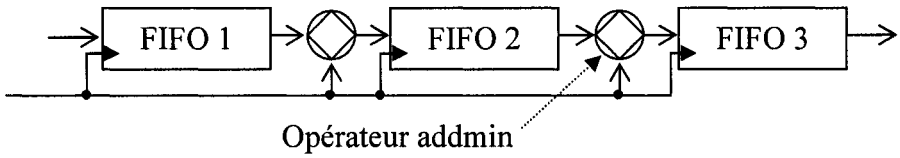

Figure 5. 7 Exemple de structure du FIFOULP

- ***Bloc de sommation***

Le bloc de sommation est composé de J ensembles identiques qui additionnent J+1 éléments binaires sur q bits, pour fournir un résultat sur q+n bits; n étant le nombre de bits nécessaires pour tenir compte du dépassement de capacité. Si on désigne le nombre d'addition dyadiques dans un niveau k par B_k, et n_k le nombre de bits d'extensions pour tenir compte du dépassement de capacité à ce niveau, la complexité du sommateur est de $J * \sum_k B_k * (q + n_k)$ LUT.

- ***Bloc d'initialisation***

Il est constitué des éléments suivants :

- une mémorisation de l'état du signal InOk, qui est synthétisé en une bascule D,
- un sommateur et un comparateur qui ont pour taille le nombre de bits nécessaires pour représenter la profondeur des FIFO et que l'on désigne par s, d'où une complexité de $\frac{s}{2}$ tranches,
- J multiplexeurs de taille q, qui sont réalisés en $J * \frac{q}{2}$ tranches,

- un élément de mémorisation de l'état du compteur de taille s, donc de complexité $\frac{s}{2}$ tranches.

On obtient donc une complexité totale égale à $\left[\frac{1}{2}+\frac{3s}{2}+J*\frac{q}{2}\right]$ tranches.

- ***Blocs de pipelines et synchronisation***

Afin de réaliser le pipeline autour du sommateur, on doit disposer de $J*(J+1)+q$ bascules sur les entrées du sommateur et $J*(q+n)$ bascules en sorties. En outre, entre le bloc de complément à deux et le saturateur, on a positionné $J*q$ bascules. En reprenant les mêmes notations que pour le sommateur, à la différence prêt que B_k désigne le nombre d'addition dyadiques ainsi que des extensions de signes pour les symboles isolés, la complexité du pipeline du sommateur est de $J*\sum_k B_k*(q+n_k)$ bascules.

La synchronisation est obtenue par le biais du bloc CE. Ce dernier est constitué d'un compteur qui génère une impulsion d'après un cycle égal aux niveaux de «pipeline». Sa complexité est similaire à celle du bloc activ2 et que l'on exprimé comme suit :

$$C_{synchro}=\left[\frac{1}{2}+\frac{3s}{2}\right] \text{ tranches} \qquad (5.1)$$

- ***Modules de décodage***

Les modules supérieurs que sont les décodeurs frontal, intermédiaire et terminal, sont constitués de l'ensemble des éléments dont on a décrit la complexité. Afin de calculer leurs complexités, il suffit de multiplier chaque formule obtenue précédemment par le

nombre d'occurrences de la fonction ou du bloc. On obtient les trois équations qui suivent :

$$C_{decodeur\_frontal} = C_{FIFOP} + C_{FIFOULP} + J*(J-1)*C_{addmin} + C_{C2input} + C_{C2Output} + C_{sommateur} + C_{activ2} + C_{Pipeline} + C_{synchronisation} \quad (5.2)$$

$$C_{decodeur\_intermédiaire} = 2*C_{FIFOP} + C_{FIFOULP} + J*(J-1)*C_{addmin} + C_{C2input} + C_{C2Output} + C_{sommateur} + C_{activ2} + C_{Pipeline} + C_{synchronisation} \quad (5.3)$$

$$C_{decodeur\_terminal} = 2*C_{FIFOP} + C_{FIFOULP} + J*(J-1)*C_{addmin} + C_{C2input} + C_{C2Output} + C_{sommateur} + C_{activ2} + C_{Pipeline} + C_{synchronisation} \quad (5.4)$$

En combinant les équations précédentes, on obtient la complexité totale du décodeur en nombre de tranches pour m itérations suivant la formule (5.5).

$$C_{decodeurl} = C_{décodeur\_frontal} + (m-2)*C_{décodeur\_intermédiaire} + C_{décodeur\_terminal} \quad (5.5)$$

5.3.2 Complexité estimée versus la complexité relevée

On a procédé à la comparaison pour certains cas de décodeur des résultats de la complexité estimée à l'aide de l'approche proposée, et celle qui a été délivrée par un outil de synthèse (Synplify de Synplicity™). Les Tableaux III et IV résument les résultats obtenus pour un Virtex-II XC2V6000.

Tableau III

Compilation des résultats de l'étude de la complexité en nombre de tranches.

	J=3 q=3		J=3 q=4		J=5 q=3	
Entité	CE	CR	CE	CR	CE	CR
FIFO	4,5	4,5	6	6	52,5	60
FIFOULP	25,5	25,5	34	34	181,5	200
Addmin	1,5	1,5	2	3,5	1,5	1,5
C2inPut	1	1	1,5	1,5	1	1
C2outPut	2	2,5	2,5	3,5	2,5	3,5
addition	6,5	6,5	8	8,5	11,5	11,5
sommateur	19,5	19,5	24	25,5	57,5	57,5
Saturateur	1,5	1	2	1,5	1,5	1
Activ2	10	8,5	11,5	10	19	12,5
pipeline	45	45	57	57	130	130
synchro	3,5	2,5	3,5	2,5	5	3,5
decof	178	176	227	303,5	755	613
decoint	140,5	138,5	180,5	227	572,5	555
decot	136	134	174,5	221	520	495

Tableau IV

Compilation des résultats de l'étude de la complexité en nombre de tranches (suite).

	J=5 q=4		J=8 q=3		J=8 q=4	
Entité	CE	CR	CE	CR	CE	CR
FIFO	70	80	2004	2016	2672	2688
FIFOULP	242	266	4183,5	4555,5	5578	6136
Addmin	2	3,5	1,5	1,5	2	3,5
C2inPut	1,5	1,5	1	1	1,5	1,5
C2outPut	3	4	3	4	3,5	5
addition	14	15	19,5	20	23,5	25
sommateur	70	75	156	160	188	200
Saturateur	2	1,5	2	1,5	2,5	2
Activ2	21,5	14	31	18,5	35	20
pipeline	162,5	162,5	352	352	436	436
synchro	5	3,5	5	3,5	5	3,5
decof	960,5	920	8019	7463	10498	10184
decoint	743	769	8923	9156	11842	12266
decot	673	689	6919	7140	9169,5	9577,5

Avec les abréviations suivantes :

- CE: Complexité estimée,
- CR: Complexité relevée,
- synchro: bloc CE de synchronisation,
- decof: decodeur frontal,
- decoint: decodeur intermédiaire,
- decot: decodeur terminal.

Il est aisé en parcourant les résultats de tirer les conclusions suivantes :

- la complexité est principalement due aux blocs FIFO et FIFOULP,

- pour des éléments simple, il est possible de parvenir à une estimation correcte de la complexité,
- plus la taille des intrants augmentent, comme pour les addmin par exemple, plus l'écart se creuse entre l'estimé et la valeur observée,
- la valeur estimée est généralement supérieure à celle qui est fournie par le synthétiseur. Mais pour les structures complexes l'inverse peut survenir,
- le cumul des erreurs sur les éléments de bases, induit une erreur plus grande au niveau des modules comme les décodeurs frontal, intermédiaire et terminal.

Malgré les écarts relevées, et qui sont dus comme on l'a déjà précisé, aux algorithmes utilisés, ainsi qu'à des contraintes comme le nombre maximal de sortances, le dédoublement de logique, etc.; l'estimation peut donner une bonne idée sur la taille du circuit nécessaire et les ressources qui vont être utilisées. Dans la pratique, il sera judicieux de majorer la complexité estimée de 10%. Il est aussi possible d'avoir un ordre de grandeur de la complexité en estimant seulement la complexité des FIFO et FIFOULP.

5.3.3 Rapidité du décodeur

Les outils de synthèse permettent de disposer d'une évaluation approximative de la fréquence maximale à laquelle le circuit peut fonctionner. Au delà de cette dernière, les résultats que l'on obtient peuvent devenir faux. Le Tableau V résume les fréquences maximales pour le décodeur frontal, intermédiaire et terminal, ainsi que pour le double décodeur tel qu'estimé par le logiciel Synplify de Synplicity™. Les mesures confrontent l'architecture avec et sans pipeline pour deux taux de codage (J=3 et J=5) et une quantification sur trois puis sur quatre bits.

Tableau V

Synthèse de l'étude de la fréquence maximale des décodeurs en MHz

Sans pipeline	J=3	J=3	J=5	J=5
Avec pipeline	q=3	q=4	q=3	q=4
décodeur frontal	125	83	76	73
	295	210	289	230
décodeur central	114	84	78	73
	295	210	273	203
décodeur terminal	124	83	79	73
	295	210	288	230
double décodeur	65	47	49	43
	295	210	247	176

On remarque que la fréquence des différents blocs de décodage et sensiblement la même, mais que la rapidité du circuit diminue nettement dans le cas du double décodeur. Ceci s'explique par l'augmentation des délais sur le chemin critique qui devient la somme des délais sur les chemins critiques des décodeurs frontal et terminal.

Le passage d'une quantification sur trois bits à une quantification sur quatre bits fait chuter la fréquence maximale possible. En effet, la complexité additionnelle augmente le chemin critique, et fait introduire de la logique et des ressources supplémentaires.

L'ajout du pipeline bien qu'il améliore grandement la fréquence de l'horloge, n'apporte pas une amélioration notable au circuit puisque l'on augmente la complexité ainsi que la latence. De plus, le débit binaire à la sortie doit être divisé par le nombre de niveaux de pipeline en raison de la stratégie de synchronisation avec le bloc CE. À titre d'exemple,

pour J=3 et q=3, le débit binaire de l'architecture sans et avec pipeline est respectivement de 195Mbps et 221Mbps. Soit une augmentation de 13% du débit binaire, mais au prix d'une complexité non négligeable.

5.4 Étude des performances du décodeur

5.4.1 Procédure de mesure de la performance

Les performances d'un décodeur sont représentées par les courbes de probabilité d'erreurs Pe en fonction du rapport signal à bruit, $\frac{E_b}{N_0}$. On est ainsi capable de positionner notre réalisation par rapport aux limites édictées par Shanon d'une part, ainsi que par rapport à d'autres systèmes de décodages correcteurs d'erreurs. Pour tracer ces courbes, on doit générer des entrées à différents niveaux de SNR, et comparer les bits en sorties à ceux qui ont été émis initialement. Le rapport entre le nombre de bits en erreurs après décodage et ceux qui ont été transmis, donne la probabilité d'erreur. Le nombre de bits à générer pour disposer d'une estimation fiable est obtenu par la méthode de Monte Carlo. Néanmoins, pour les canaux statiques, il est communément admis que l'on doit au moins observer une centaine d'erreurs ou bien atteindre 10^7 bits émis pour que l'estimé soit considéré comme acceptable.

Les simulations ont été effectuées en générant un ensemble de symboles aléatoirement. Ils sont ensuite transposés en BPSK, bruités par un canal gaussien avant d'être quantifiés sur 2^q niveaux équidistants. L'ensemble de ces opérations est réalisé sous Matlab et les symboles initiaux sont sauvegardés dans un fichier, ainsi que les symboles quantifiés dans un second fichier. Le banc de test en VHDL accède aux données du second fichier, procède au décodage et enregistre les résultats dans un troisième fichier. L'opération finale consiste à utiliser une fonction de Matlab, qui compare les données dans le premier et le dernier fichier, en tenant compte de la latence et fournit alors le taux

d'erreurs binaires. Il faut noter que des outils de co-conception existent, et ils permettent d'automatiser toute l'opération que l'on vient de décrire [40] [44]. En répétant cette opération pour différentes valeurs de SNR, on peut tracer les courbes de performance du décodeur par interpolation. Les mêmes fichiers peuvent servir pour tester différentes itérations de décodage en modifiant le fichier de test en VHDL, il faut seulement veiller à ce que le nombre d'erreurs observées ou le nombre total de bits décodées soit respecté.

5.4.2 Résultats des simulations

Les simulations ont été réalisées pour des taux de codage de J=3 et J=5 en raison des longueurs de contraintes respectives. Ces dernières rendent les calculs moins exigeants en terme de puissance machine par rapport à des taux de codage plus élevé, ainsi que des itérations de décodage dépassants trois. De plus, le but étant de valider le concept par comparaison à l'étude de Cardinal et al [1], on a relevé point par point les courbes de performances qu'il a obtenus par des simulations sans quantifications. Ainsi, dans les courbes de performances des figures 5.8, 5.9 et 5.10, on trace les valeurs de référence et ceux résultants de notre banc d'essai. On remarque qu'avec une quantification sur trois bits, les courbes différent seulement en raison du bruit introduit par la quantification, et ceci pour les deux taux de codage. De plus, l'ajout d'un niveau de décodage pour porter le nombre d'itération à trois dans le cas de J=3 améliore significativement les performances tout en restant conforme aux remarques précédentes. Il est donc clair que les résultats des simulations répondent aux attentes et confirment les études préliminaires.

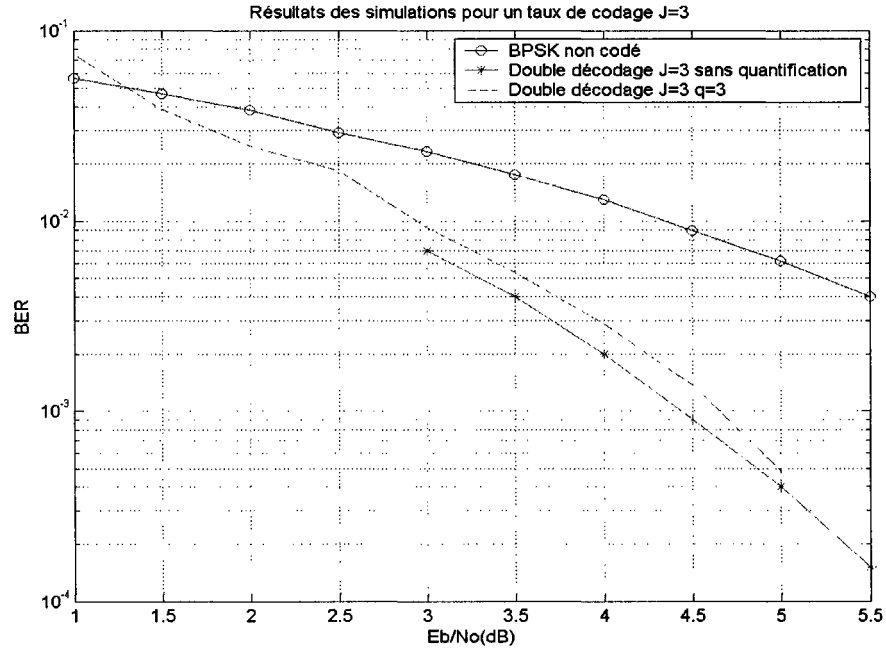

Figure 5.8 Résultats des simulations pour deux itérations de décodage pour J=3 et q=3

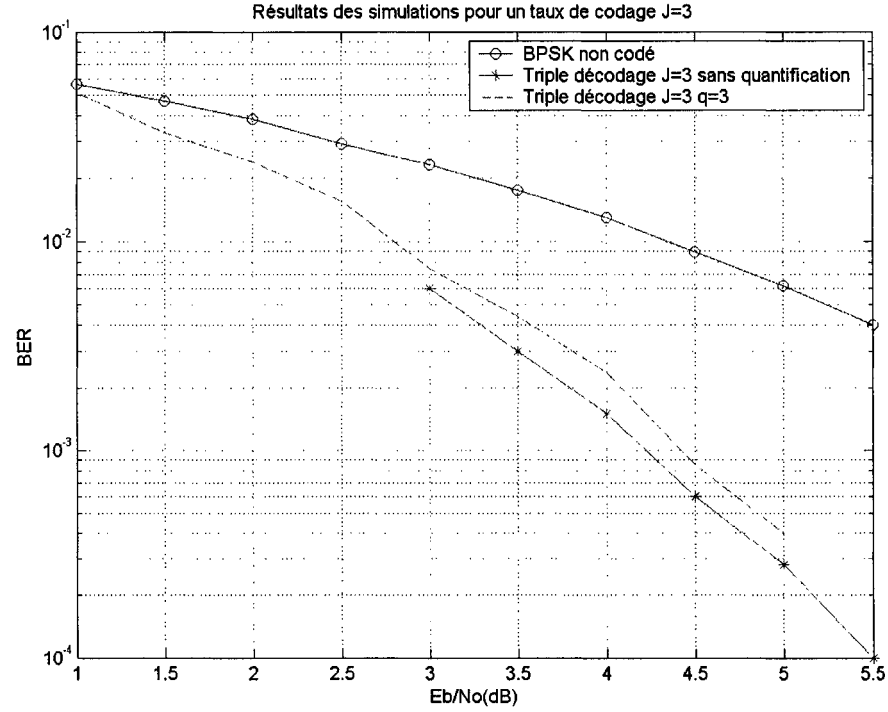

Figure 5.9 Résultats des simulations pour trois itérations de décodage pour J=3 et q=3

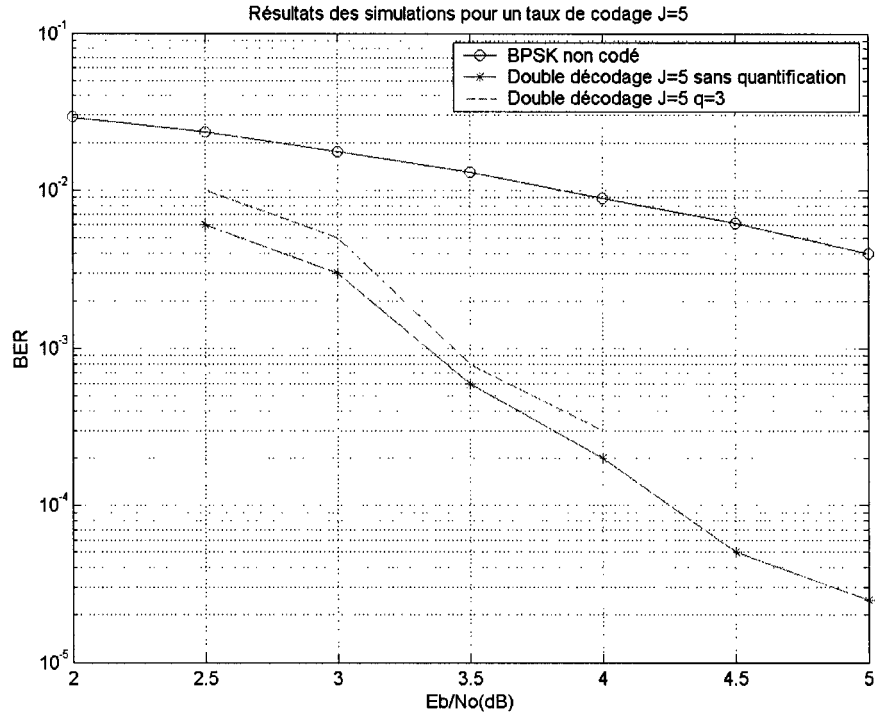

Figure 5.10 Résultats des simulations pour deux itérations de décodage pour J=5 et q=3

5.5 Conclusion

En se referant aux notes d'applications de notre technologie cible, il a été possible de profiter de l'avantage qu'apportent certaines ressources spéciales mises en place pour un créneau comme celui des codes convolutionnels.

Deux autres améliorations peuvent encore être apportées au niveau du bloc gérant l'initialisation des registres à décalages. La première idée consiste à utiliser des portes à trois états (TRI-STATES) pour réaliser le multiplexage. En effet, ces ressources sont rarement utilisées et il faut adapter le code VHDL pour en tirer profit en contribuant à économiser J LUT. La seconde technique, s'inspire de la note d'application [45] qui suggère de remplacer les compteurs par des générateurs de séquences pseudo aléatoires. La séquence est prévisible et se répète au bout d'un cycle donné, ce qui la rend plus

avantageuse par rapport au compteur en terme de complexité. Reste que cette approche est difficilement généralisable ce qui explique qu'on ne l'a pas intégré au code VHDL du décodeur.

L'étude de la complexité de l'architecture démontre que le nombre de tranches occupées par le système augmente fortement avec l'augmentation du taux de codage. Ceci est dû aux tailles des FIFO et FIFOULP qui sont une fonction croissante de J, ainsi que du nombre même de ces registres à décalage. De même, la vitesse de fonctionnement diminue avec l'augmentation de J puisque le chemin critique devient plus long. Il faut aussi noter que la puissance consommée par le circuit devrait augmenter en fonction de J.

Les simulations VHDL de la performance d'erreur viennent confirmer les résultats obtenus par simulations dans [1]. En raison de la nature de la technologie cible, il est facile d'affirmer que les tests qui seront menés sur circuits corroboreront nos conclusions.

CONCLUSION

Le prototypage microélectronique du décodeur de canal de type itératif a été une opération complexe nécessitant une continuelle amélioration des choix d'architectures en fonction de l'algorithme et des ressources disponibles sur la famille de circuits que l'on ciblait. Le décodeur que l'on a conçu devait reproduire aussi fidèlement que possible les performances établies par simulation, tout en présentant une complexité raisonnable. Le but du prototype est de pouvoir estimer la complexité et les performances de n'importe quelle taille de décodeur CSO^2C-SS. Ceci pour les codes déjà disponibles ainsi que pour ceux qui vont être développés dans le futur. On s'est donc basé sur les équations de décodage que l'on a transposé en architecture numérique capable de réaliser les mêmes calculs, mais avec certaines limitations. Ainsi, l'information en entrées devant être numérisée, le choix d'un nombre de bits de quantification ainsi que la représentation qu'on donne à ce symbole binaire est une limitation que l'on porte sur l'algorithme de base qui influence énormément les résultats et qu'il fallait prendre en considération.

Le décodeur est basé sur un algorithme très parallèle, qu'il fallait exploiter au maximum pour atteindre de bonnes performances en fréquence de fonctionnement. Certes il aurait été possible de prendre un tout autre chemin et avoir pour stratégie le partage des ressources. Ceci est faisable en adoptant un développement similaire à celui d'un processeur dédié au décodage. La structure serait alors une combinaison de mémoires, registres, unités spécialisées dans les opérations de base et un mécanisme pour gérer le flux de données. Mais cette architecture aurait été plus difficile à concevoir et aurait sacrifié l'avantage que l'on peut tirer du parallélisme.

Il aurait été possible aussi de produire un programme VHDL qui gèrerait tout le processus de généralisation et serait capable d'extraire la bonne architecture à chaque fois qu'on lui fourni une matrice de codage différente. Cette approche est diamétralement opposée à celle que l'on a suivi, car on a jugé moins complexe de scinder les opérations.

En effet, il faut distinguer deux codes:

- un premier code visant à identifier toute la structure du décodeur,
- un second code qui constitue le décodeur, pour la simulation et la synthèse.

En recourant au langage C pour la première étape, on a gardé cette frontière bien claire et on a simplifié la maintenance de l'ensemble.

Le mémoire a été décomposé de façon à amener le lecteur à cheminer depuis les éléments de base du codage de canal et son utilité, jusqu'aux résultats finaux donnant une idée sur la complexité et les performances du décodeur. Mais pour bien appréhender l'ensemble et surtout la partie théorique, il est impératif de se référer à la thèse de doctorat de C. Cardinal. La conclusion du dernier chapitre présente certaines idées qui peuvent faire l'objet d'un travail supplémentaire pour fin d'amélioration, mais il faut que cela soit étudié au cas par cas. L'étape finale serait de réaliser les tests sur circuit, puis d'intégrer le projet comme système de codage de canal dans une application concrète telle la communication sans fil ou le projet en cours sur les radio logiciels. Il serait aussi intéressant de faire une étude de la puissance consommée et son évolution en fonction de J et de α, ainsi que de réaliser une comparaison de la latence et la complexité d'un décodeur CSO^2C-SS versus un code turbo présentant la même courbe de performance.

ANNEXE 1

Code VHDL pour un double décodeur CSO^2C-SS avec J=3 et pipeline

CODE VHDL POUR UN DOUBLE DÉCODEUR CSO^{2}C-SS AVEC J=3 ET PIPELINE

Package ma_library

```
Library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all ;
use ieee.std_logic_unsigned.all;

package ma_library is
      constant nombre_de_bit_de_quantification : integer :=3;
      constant reset : std_logic_vector :="011";
      constant saturation_1 : std_logic_vector :="101";
      alias N : integer is nombre_de_bit_de_quantification;
      subtype echantillon is std_logic_vector(N-1 downto 0);
      subtype saturation is std_logic_vector (N-3 downto 0);
      function ADM  ( l, r : echantillon  ) return echantillon;
      function ADM  ( l, r,s : echantillon  ) return echantillon;
      function ADD (a,b: std_logic_vector) return std_logic_vector;
      function C2 (a : std_logic_vector) return std_logic_vector;
      function sature (a: std_logic_vector) return std_logic_vector;
end ma_library;

package body ma_library is
--fonction Addmin par surcharge
      function ADM  ( l, r : echantillon  ) return echantillon is
      begin
        if (l(l'high-1 downto 0) < r(r'high-1 downto 0)) then
            return((l(l'high) xor r(r'high))&l(l'high-1 downto 0));
        else
            return((l(l'high) xor r(r'high))&r(r'high-1 downto 0));
        end if;
      end;

--fonction Addmin par surcharge
      function ADM  ( l, r,s : echantillon  ) return echantillon is
      begin
            return(ADM(l,ADM(r,s)));
      end;

--Calcul du complement a deux
      function C2 (a: std_logic_vector) return std_logic_vector is
      begin
        if (a(a'high)='1') then
            return(a(a'high)&("0"-a(a'high-1 downto 0)));
        else
                  return(a);
        end if;
      end C2;
```

```
--Fonction pour la saturation
      function sature (a: std_logic_vector) return std_logic_vector is
      begin
        if ((a(a'high-1) or a(a'high-2))='1') then
            return (a(a'high)&"11");
        else
            return (a(a'high)&a(N-2 downto 0));
        end if;
      end sature;

--Fonction pour l'addition avec extention du MSB
      function ADD (a,b: std_logic_vector) return std_logic_vector is
      begin
               return ((a(a'high)&a)+(b(b'high)&b));
      end ADD;

end ma_library;
```

Entité FIFOP

```
Library ieee;
use ieee.std_logic_1164.all;
use work.ma_library.all;

entity FIFOP is
port(
      clk,CE                  : in std_logic;
      E1,E2,E3                : in echantillon;
      S1,S2,S3                : out echantillon);
end FIFOP;

Architecture AFIFO35 of FIFO35 is

      Type FIFO_tableau is array (4 downto 0) of echantillon;
      Signal fifo1,fifo2,fifo3: FIFO_tableau;
      Begin

      process(clk)
      begin
        if(clk'event and clk='1') then
         if(CE='1') then
            for i in fifo1'high downto 1 loop
                  fifo1(i)<=fifo1(i-1);
            end loop;
                  fifo1(0)<=E1;
         end if;
        end if;
      end process;

      process(clk)
      begin
        if(clk'event and clk='1') then
         if(CE='1') then
```

```
            for i in fifo2'high downto 1 loop
                  fifo2(i)<=fifo2(i-1);
            end loop;
                  fifo2(0)<=E2;
         end if;
        end if;
      end process;

      process(clk)
      begin
        if(clk'event and clk='1') then
         if(CE='1') then
            for i in fifo3'high downto 1 loop
                  fifo3(i)<=fifo3(i-1);
            end loop;
                  fifo3(0)<=E3;
         end if;
        end if;
      end process;

            S1<=fifo1(fifo1'high);
            S2<=fifo2(fifo2'high);
            S3<=fifo3(fifo3'high);
      end;
```

Entité FIFOULP

```
Library ieee;
use ieee.std_logic_1164.all;
use work.ma_library.all;


entity FIFO35_UP is
port(
      clk,CE                  : in std_logic;
      Ue1,Ue2,Ue3             : in echantillon;
      Pe1,Pe2,Pe3             : in echantillon;
      L1,L2,L3                : in echantillon;
      Us1,Us2,Us3             : out echantillon;
      Psy11, Psy12,Psy13,
      Psy21, Psy22,Psy23,
      Psy31, Psy32, Psy33     : Out echantillon);
end FIFO35_UP;

Architecture AFIFO35_UP of FIFO35_UP is

      Type FIFO_tableauPU is array (4 downto 0) of echantillon;
      Type tableauS is array (1 to 19) of echantillon;
      signal fifoU1,fifoU2,fifoU3 : FIFO_tableauPU;
      Signal fifoP1,fifoP2,fifoP3: FIFO_tableauPU;
      Signal S : tableauS;
      Begin
```

```
--premier fifo pour l'information
U1 :  process(clk)
      begin
       if(clk'event and clk='1') then
        if (CE='1') then
         for i in fifoU1'high downto 1 loop
                  fifoU1(i)<=fifoU1(i-1);
         end loop;
                  fifoU1(0)<=Ue1;
        end if;
       end if;
      end process;

--Second fifo pour l'information
U2 :  process(clk)
      begin
       if(clk'event and clk='1') then
        if (CE='1') then
         for i in fifoU2'high downto 1 loop
                  fifoU2(i)<=fifoU2(i-1);
         end loop;
                  fifoU2(0)<=Ue2;
        end if;
       end if;
      end process;

--Troisième fifo pour l'information
U3 :  process(clk)
      begin
       if(clk'event and clk='1') then
        if (CE='1') then
         for i in fifoU3'high downto 1 loop
                  fifoU3(i)<=fifoU3(i-1);
         end loop;
                  fifoU3(0)<=Ue3;
        end if;
       end if;
      end process;

--premier fifo pour la parité
P1 :  process(clk)
      begin
       if(clk'event and clk='1') then
        if (CE='1') then
                  fifoP1(4)<=ADM(fifoP1(3),L2);
                  fifoP1(3)<=fifoP1(2);
                  fifoP1(2)<=fifoP1(1);
                  fifoP1(1)<=fifoP1(0);
                  fifoP1(0)<=ADM(Pe1,L3);
        end if;
       end if;
      end process;
```

```
--second fifo pour la parité
P2 :  process(clk)
      begin
       if(clk'event and clk='1') then
        if (CE='1') then
                  fifoP2(4)<=fifoP2(3);
                  fifoP2(3)<=ADM(fifoP2(2),L3);
                  fifoP2(2)<=ADM(fifoP2(1),L2);
                  fifoP2(1)<=fifoP2(0);
                  fifoP2(0)<=Pe2;
        end if;
       end if;
      end process;

--troisième fifo pour la parité
P3 :  process(clk)
      begin
       if(clk'event and clk='1') then
        if (CE='1') then
                  fifoP3(4)<=fifoP3(3);
                  fifoP3(3)<=fifoP3(2);
                  fifoP3(2)<=fifoP3(1);
                  fifoP3(1)<=fifoP3(0);
                  fifoP3(0)<=ADM(Pe3,L1);
        end if;
       end if;
      end process;

--Affectation des sorties
                  Us1<=fifoU1(fifoU1'high);
                  Us2<=fifoU2(fifoU2'high);
                  Us3<=fifoU3(fifoU3'high);
--Psy11
                  S(1)<=fifoP1(4);
--Psy12
                  S(2)<=fifoP2(4);
--Psy13
                  S(3)<=Pe3;
                  S(4)<=Ue2;
                  S(5)<=Ue3;
--Psy21
                  S(6)<=fifoP1(3);
                  S(7)<=fifoU1(3);
--Psy22
                  S(8)<=fifoP2(1);
                  S(9)<=fifoU1(1);
                  S(10)<=fifoU3(3);
--Psy23
                  S(11)<=fifoP3(4);
                  S(12)<=fifoU3(4);
--Psy31
                  S(13)<=Pe1;
                  S(14)<=Ue1;
```

```
                    S(15)<=fifoU2(0);
--Psy32
                    S(16)<=fifoP2(2);
                    S(17)<=fifoU1(2);
--Psy33
                    S(18)<=fifoP3(4);
                    S(19)<=fifoU2(4);

--cacul de l'ADDMIN
                    Psy11<=S(1);
                    Psy12<=S(2);
                    Psy13<=ADM(S(3),S(4),S(5));
                    Psy21<=ADM(S(6),S(7));
                    Psy22<=ADM(S(8),S(9),S(10));
                    Psy23<=ADM(S(11),S(12));
                    Psy31<=ADM(S(13),S(14),S(15));
                    Psy32<=ADM(S(16),S(17));
                    Psy33<=ADM(S(18),S(19));
end;
```

Entité C2INPUT

```
Library ieee;
use ieee.std_logic_1164.all;
use work.ma_library.all;

entity C2Input is
port(
En1,En2,En3,En4,En5,En6,En7,En8,En9,En10,En11,En12: in echantillon;
So1,So2,So3,So4,So5,So6,So7,So8,So9,So10,So11,So12: out echantillon
      );
end C2Input;

Architecture Arch_C2Input of C2Input is

 begin
      So1<=C2(En1);
      So2<=C2(En2);
      So3<=C2(En3);
      So4<=C2(En4);
      So5<=C2(En5);
      So6<=C2(En6);
      So7<=C2(En7);
      So8<=C2(En8);
      So9<=C2(En9);
      So10<=C2(En10);
      So11<=C2(En11);
      So12<=C2(En12);
end;
```

Entité C2OUTPUT

```
Library ieee;
use ieee.std_logic_1164.all;
use work.ma_library.all;

entity C2Output is
port(
      En1,En2,En3 : in std_logic_vector(N+1 downto 0);
      So1,So2,So3 : out std_logic_vector(N+1 downto 0)
     );
end C2Output;

Architecture Arch_C2Output of C2Output is
begin
      So1<=C2(En1);
      So2<=C2(En2);
      So3<=C2(En3);
end;
```

Entité Sommateur

```
Library ieee;
use ieee.std_logic_1164.all;
use work.ma_library.all;

entity Sommateur_J3 is
port(
      clk, rst                : in std_logic;
      Ue1,Ue2,Ue3             : in echantillon;
      Psy11,Psy12,Psy13,
      Psy21,Psy22,Psy23,
      Psy31,Psy32,Psy33       : in echantillon;
      L1,L2,L3                : out std_logic_vector (N+1 downto 0));
end Sommateur_J3;

Architecture Arch_Sommateur_J3 of Sommateur_J3 is

signal int1,int2,int3,int4,int5,int6 :std_logic_vector (N downto 0);
begin

      process(clk)
      begin
       if clk'event and clk='1' then
        if rst='1' then
               int1<=(others=>'0');
        else
               int1<=ADD(Ue1,Psy11);
       end if;
       end if;
      end process;
```

```
    process(clk)
    begin
     if clk'event and clk='1' then
      if rst='1' then
                int2<=(others=>'0');
      else
                int2<=ADD(Psy12,Psy13);
      end if;
     end if;
    end process;

    process(clk)
    begin
     if clk'event and clk='1' then
      if rst='1' then
                int3<=(others=>'0');
      else
                int3<=ADD(Ue2,Psy21);
      end if;
     end if;
    end process;

    process(clk)
    begin
     if clk'event and clk='1' then
      if rst='1' then
                int4<=(others=>'0');
      else
                int4<=ADD(Psy22,Psy23);
      end if;
     end if;
    end process;

    process(clk)
    begin
     if clk'event and clk='1' then
      if rst='1' then
                int5<=(others=>'0');
      else
                int5<=ADD(Ue3,Psy31);
      end if;
     end if;
    end process;

    process(clk)
    begin
     if clk'event and clk='1' then
      if rst='1' then
                int6<=(others=>'0');
      else
                int6<=ADD(Psy32,Psy33);
    end if;
end if;
```

```
end process;

L1<=ADD(int1,int2);
L2<=ADD(int3,int4);
L3<=ADD(int5,int6);

end;
```

Entité Pipe

```
library ieee;
use ieee.std_logic_1164.all;
use work.ma_library.all;

entity pipe is
generic (taille : integer :=3);
port (clk, rst          : in std_logic;
      IN1               : in std_logic_vector(taille-1 downto 0);
      OUT1              : out std_logic_vector(taille-1 downto 0));
end pipe;

architecture arch of pipe is
begin
      process(clk)
      begin
       if clk'event and clk='1' then
        if rst='1' then
                  OUT1<=(others=>'0');
        else
                  OUT1<=IN1;
        end if;
       end if;
      end process;
end;
```

Entité Activ2

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE WORK.ma_library.all;

entity activ2 is
port (
      IN1,IN2,IN3       : in std_logic_vector (N-1 downto 0);
      InOK              : in  std_logic;
      CLK               : in  std_logic;
      CE                : in std_logic;
      OUT1,OUT2,OUT3 : out std_logic_vector (N-1 downto 0);
      OutOK    : OUT STD_LOGIC);
end activ2;
```

```
architecture arch_activ2 of activ2 is
signal sel: std_logic;
begin

      process(CLK)
      variable count :  unsigned(2 downto 0) ;
      begin
       if rising_edge(CLK) then
        if (CE='1') then
         if ( InOK='0') then
          sel <= '0';
          count := (others => '0');
         elsif InOK = '1' then
                        if count<5 then
                              count:= count+1 ;
                              sel <= '0';
                        else
                              count:=count;
                              sel<='1';
                        end if;
          end if;
        end if;
      end if;
      end process;

            OUT1<=IN1 when sel='1' else reset;
            OUT2<=IN2 when sel='1' else reset;
            OUT3<=IN3 when sel='1' else reset;
            OutOK<=sel;
end;
```

Entité Saturation

```
Library ieee;
use ieee.std_logic_1164.all;
use work.ma_library.all;

entity Saturation is
port(
      En1,En2,En3: in std_logic_vector(N+1 downto 0);
      So1,So2,So3: out echantillon
      );
end Saturation;
Architecture Arch_Saturation of Saturation is
begin
      So1<=sature(En1);
      So2<=sature(En2);
      So3<=sature(En3);

end;
```

Entité Décision

```
Library ieee;
use ieee.std_logic_1164.all;
use work.ma_library.all;

entity decision is
port(
      L1,L2,L3 : in std_logic_vector (N+1 downto N-1);
      S1,S2,S3 : out std_logic);
end decision;

Architecture Arch_decision of decision is
begin
      S1<=L1(L1'high);
      S2<=L2(L2'high);
      S3<=L3(L3'high);
end;
```

Entité CE

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

entity CE is
port (
       CLK,rst    : in  std_logic;
       CE   : OUT STD_LOGIC);
end CE;

architecture arch_CE of CE is
signal sel: std_logic;
begin
  process(CLK)
  variable count :  unsigned(1 downto 0) ;
  begin
   if rising_edge(CLK) then
    if (rst='1') then
          sel <= '0';
          count := (others => '0');
    elsif rst = '0' then
          if count<3 then
                  count:= count+1 ;
                  sel <= '0';
          else
                  count:=(others => '0');
                  sel<='1';
          end if;
       end if;
     end if;
   end process;
```

```
      CE<=sel;
end;
```

Package composants

```
library ieee;
use ieee.std_logic_1164.all;
use work.ma_library.all;

package composants is

component FIFO35_UP
port(
      clk,CE                  : in std_logic;
      Ue1,Ue2,Ue3             : in echantillon;
      Pe1,Pe2,Pe3             : in echantillon;
      L1,L2,L3                : in echantillon;
      Us1,Us2,Us3             : out echantillon;
      Psy11, Psy12,Psy13,
      Psy21, Psy22,Psy23,
      Psy31, Psy32,Psy33      : Out echantillon);
end component;

component Sommateur_J3
port(
      clk, rst                : in std_logic;
      Ue1,Ue2,Ue3             : in echantillon;
      Psy11,Psy12,Psy13,
      Psy21,Psy22,Psy23,
      Psy31,Psy32,Psy33       : in echantillon;
      L1,L2,L3                : out std_logic_vector (N+1 downto 0));
end component;

Component FIFO35
port(
      clk,CE                  : in std_logic;
      E1,E2,E3                : in echantillon;
      S1,S2,S3                : out echantillon);
end component;

component C2Input
port(
En1,En2,En3,En4,En5,En6,En7,En8,En9,En10,En11,En12: in echantillon;
So1,So2,So3,So4,So5,So6,So7,So8,So9,So10,So11,So12: out echantillon
      );
end component;

component C2Output
port(
      En1,En2,En3             : in std_logic_vector(N+1 downto 0);
      So1,So2,So3             : out std_logic_vector(N+1 downto 0)
);
end component;
```

```
component Saturation
port(
      En1,En2,En3              : in std_logic_vector(N+1 downto 0);
      So1,So2,So3              : out echantillon
      );
end component;

component activ2
 port (
      IN1,IN2,IN3              : in std_logic_vector (N-1 downto 0);
      InOK                     : in  std_logic;
      CLK                      : in  std_logic;
      CE                       : in  std_logic;
      OUT1,OUT2,OUT3           : out std_logic_vector (N-1 downto 0);
      OutOK                    : out std_logic);
end component;

component pipe
generic (taille : integer :=3);
port (
      clk, rst          : in std_logic;
      IN1               : in std_logic_vector(taille-1 downto 0);
      OUT1              : out std_logic_vector(taille-1 downto 0));
end component;

component CE
port (
       CLK,rst          : in  std_logic;
       CE               : OUT STD_LOGIC);
end component;

end composants;
```

Entité Décodeur frontal

```
library ieee;
use ieee.std_logic_1164.all;
use work.ma_library.all;
use work.composants.all;

entity decodeur_frontal_piped is
port (
      clk,rst,InOK                  : in std_logic;
      Ue1,Ue2,Ue3,Pe1,Pe2,Pe3       : in echantillon;
      Us1,Us2,Us3                   : out echantillon;
      Ps1,Ps2,Ps3                   : out echantillon;
      L1,L2,L3                      : out echantillon;
      OutOK                         : out std_logic
      );

end decodeur_frontal_piped;
```

```
architecture arch_decodeur_frontal_p of decodeur_frontal_piped is

--Déclaration des signaux intermédiaires
signal Ps11,Ps12,Ps13,Ps21,Ps22,Ps23,Ps31,Ps32,Ps33 : echantillon;
signal Uin1,Uin2,Uin3 : echantillon;
signal addin1,addin2,addin3,addin4,addin5,addin6,addin7,addin8,addin9,
       addin10,addin11,addin12 : echantillon;
signal paddin1,paddin2,paddin3,paddin4,paddin5,paddin6,paddin7,paddin8,
       paddin9,paddin10,paddin11,paddin12 : echantillon;
signal addout1,addout2,addout3 : std_logic_vector ( N+1 downto 0);
signal paddout1,paddout2,paddout3 : std_logic_vector ( N+1 downto 0);
signal satin1,satin2,satin3 : std_logic_vector ( N+1 downto 0);
signal psatin1,psatin2,psatin3 : std_logic_vector ( N+1 downto 0);
signal act1,act2,act3: echantillon;
signal Lin1,Lin2,Lin3 : echantillon;
signal enable: std_logic;

--Debut de l'architecture
begin
inst_CE : CE port map(clk,rst,enable);
instFIFO35 : FIFO35 port map (clk,enable,Pe1,Pe2,Pe3,Ps1,Ps2,Ps3);
instFIFO35_UP : FIFO35_UP port map (
      clk,enable,Ue1,Ue2,Ue3,Pe1,Pe2,Pe3,Lin1,Lin2,Lin3,
      Uin1,Uin2,Uin3,Ps11,Ps12,Ps13,Ps21,Ps22,Ps23,Ps31,Ps32,Ps33);
instC2Input   : C2Input port map (
      Uin1,Uin2,Uin3,Ps11,Ps12,Ps13,Ps21,Ps22,Ps23,Ps31,Ps32,Ps33,
      addin1,addin2,addin3,addin4,addin5,addin6,addin7,addin8,addin9,ad
      din10,addin11,addin12);

instpipes4: pipe port map (clk,rst,addin1,paddin1);
instpipes5: pipe port map (clk,rst,addin2,paddin2);
instpipes6: pipe port map (clk,rst,addin3,paddin3);
instpipes7: pipe port map (clk,rst,addin4,paddin4);
instpipes8: pipe port map (clk,rst,addin5,paddin5);
instpipes9: pipe port map (clk,rst,addin6,paddin6);
instpipes10: pipe port map (clk,rst,addin7,paddin7);
instpipes11: pipe port map (clk,rst,addin8,paddin8);
instpipes12: pipe port map (clk,rst,addin9,paddin9);
instpipes13: pipe port map (clk,rst,addin10,paddin10);
instpipes14: pipe port map (clk,rst,addin11,paddin11);
instpipes15: pipe port map (clk,rst,addin12,paddin12);

instSommateur : Sommateur_J3 port map(
      clk,rst,paddin1,paddin2,paddin3,paddin4,paddin5,paddin6,paddin7,
      paddin8,paddin9,paddin10,paddin11,paddin12,addout1,addout2,
      addout3);
instpipes16: pipe generic map (N+2) port map(clk,rst,addout1,paddout1);
instpipes17: pipe generic map (N+2) port map(clk,rst,addout2,paddout2);
instpipes18: pipe generic map (N+2) port map(clk,rst,addout3,paddout3);


instC2Output  : C2Output port map ( paddout1,paddout2,paddout3,
                                    satin1,satin2,satin3);
```

```
instpipes1: pipe generic map (N+2) port map (clk,rst,satin1,psatin1);
instpipes2: pipe generic map (N+2) port map (clk,rst,satin2,psatin2);
instpipes3: pipe generic map (N+2) port map (clk,rst,satin3,psatin3);
instSaturation: Saturation  port
                  map(psatin1,psatin2,psatin3,act1,act2,act3);
instactiv2:activ2 port map(act1,act2,act3,IBE,clk,enable,Lin1,Lin2,
                           Lin3,OBE);
      L1<=Lin1;
      L2<=Lin2;
      L3<=Lin3;

      Us1<=Uin1;
      Us2<=Uin2;
      Us3<=Uin3;

end;
```

Entité Décodeur intermédiaire

```
library ieee;
use ieee.std_logic_1164.all;
use work.ma_library.all;
use work.composants.all;

entity decodeur_ intermed_piped is
port (
      clk,rst,IBE                          : in std_logic;
      Ue1,Ue2,Ue3,Pe1,Pe2,Pe3,Le1,Le2,Le3 : in echantillon;
      Us1,Us2,Us3                          : out echantillon;
      Ps1,Ps2,Ps3                          : out echantillon;
      L1,L2,L3                             : out echantillon;
      OBE                                  : out std_logic
      );
end decodeur_intermed _piped;

architecture arch_decodeur_intermed_piped of decodeur_ intermed_piped
is
--Déclaration des signaux intermédiaires
signal Ps11,Ps12,Ps13,Ps21,Ps22,Ps23,Ps31,Ps32,Ps33 : echantillon;
signal Uin1,Uin2,Uin3 : echantillon;
signal addin1,addin2,addin3,addin4,addin5,addin6,addin7,addin8,addin9,
       addin10,addin11,addin12 : echantillon;
signal addint1,addint2,addint3,addint4,addint5,addint6,addint7,addint8,
       addint9,addint10,addint11,addint12 : echantillon;
signal addout1,addout2,addout3 : std_logic_vector ( N+1 downto 0);
signal addoutt1,addoutt2,addoutt3 : std_logic_vector ( N+1 downto 0);
signal satin1,satin2,satin3 : std_logic_vector ( N+1 downto 0);
signal satint1,satint2,satint3 : std_logic_vector ( N+1 downto 0);
signal act1,act2,act3 :echantillon;
signal Lin1,Lin2,Lin3 : echantillon;
signal CEs: std_logic;
```

```
begin
   inst_CE : CE  port map(clk,rst,CEs);
   inst_P_FIFO35 : FIFO35 port map (clk,CEs,Pe1,Pe2,Pe3,Ps1,Ps2,Ps3);
   inst_U_FIFO35 : FIFO35 port map(clk,CEs,Ue1,Ue2,Ue3,Uin1,Uin2,Uin3);
   instFIFO35_UP : FIFO35_UP port map (
      clk,CEs,Le1,Le2,Le3,Pe1,Pe2,Pe3,Lin1,Lin2,Lin3,
      open,open,open,Ps11,Ps12,Ps13,Ps21,Ps22,Ps23,Ps31,Ps32,Ps33);
   instC2Input   : C2Input port map (
      Uin1,Uin2,Uin3,Ps11,Ps12,Ps13,Ps21,Ps22,Ps23,Ps31,Ps32,Ps33,
      addin1,addin2,addin3,addin4,addin5,addin6,addin7,addin8,
      addin9,addin10,addin11,addin12);
   intpipe1: pipe port map(clk,rst,addin1,addint1);
   intpipe2: pipe port map(clk,rst,addin2,addint2);
   intpipe3: pipe port map(clk,rst,addin3,addint3);
   intpipe4: pipe port map(clk,rst,addin4,addint4);
   intpipe5: pipe port map(clk,rst,addin5,addint5);
   intpipe6: pipe port map(clk,rst,addin6,addint6);
   intpipe7: pipe port map(clk,rst,addin7,addint7);
   intpipe8: pipe port map(clk,rst,addin8,addint8);
   intpipe9: pipe port map(clk,rst,addin9,addint9);
   intpipe10: pipe port map(clk,rst,addin10,addint10);
   intpipe11: pipe port map(clk,rst,addin11,addint11);
   intpipe12: pipe port map(clk,rst,addin12,addint12);
   instSommateur : Sommateur_J3 port map(clk,rst,
      addint1,addint2,addint3,addint4,addint5,addint6,addint7,addint8,
      addint9,addint10,addint11,addint12,addout1,addout2,addout3);
   intpipe13: pipe generic map(N+2)port map(clk,rst,addout1,addoutt1);
   intpipe14: pipe generic map(N+2)port map(clk,rst,addout2,addoutt2);
   intpipe15: pipe generic map(N+2)port map(clk,rst,addout3,addoutt3);
   instC2Output  : C2Output port map (addoutt1,addoutt2,addoutt3,
                   satin1,satin2,satin3);
   intpipe16: pipe generic map (N+2) port map(clk,rst,satin1,satint1);
   intpipe17: pipe generic map (N+2) port map(clk,rst,satin2,satint2);
   intpipe18: pipe generic map (N+2) port map(clk,rst,satin3,satint3);
   instSaturation: Saturation port map (
                                    satint1,satint2,satint3,
                                    act1,act2,act3);
   instactiv2:activ2 port map
                        (act1,act2,act3,IBE,clk,CEs,Lin1,Lin2,Lin3,OBE);
            L1<=Lin1;
            L2<=Lin2;
            L3<=Lin3;
            Us1<=Uin1;
            Us2<=Uin2;
            Us3<=Uin3;

end;
```

Entité Décodeur terminal

```
library ieee;
use ieee.std_logic_1164.all;
```

```
use work.ma_library.all;
use work.composants.all;

entity decodeur_terminal_piped is
port (
      clk,rst,InOK                          : in std_logic;
      Ue1,Ue2,Ue3,Pe1,Pe2,Pe3,Le1,Le2,Le3 : in echantillon;
      L1,L2,L3                              : out echantillon;
      OutOK                                 : out std_logic
      );
end decodeur_terminal_piped;

architecture arch_decodeur_terminal_piped of decodeur_terminal_piped is

--déclaration des signaux
signal Ps11,Ps12,Ps13,Ps21,Ps22,Ps23,Ps31,Ps32,Ps33 : echantillon;
signal Uin1,Uin2,Uin3 : echantillon;
signal addin1,addin2,addin3,addin4,addin5,addin6,addin7,addin8,addin9,
       addin10,addin11,addin12 : echantillon;
signal paddin1,paddin2,paddin3,paddin4,paddin5,paddin6,paddin7,paddin8,
       paddin9,paddin10,paddin11,paddin12 : echantillon;
signal addout1,addout2,addout3 : std_logic_vector ( N+1 downto 0);
signal paddout1,paddout2,paddout3 : std_logic_vector ( N+1 downto 0);
signal satin1,satin2,satin3 : std_logic_vector ( N+1 downto 0);
signal psatin1,psatin2,psatin3 : std_logic_vector ( N+1 downto 0);
signal act1,act2,act3: echantillon;
signal Lin1,Lin2,Lin3 : echantillon;
signal CEs: std_logic;

begin
  inst_CE : CE port map(clk,rst,CEs);
  inst_U_FIFO35 : FIFO35 port map (clk,CEs,Ue1,Ue2,Ue3,Uin1,Uin2,Uin3);
  instFIFO35_UP : FIFO35_UP port map (clk,CEs,Le1,Le2,Le3,Pe1,Pe2,Pe3,
                                      Lin1,Lin2,Lin3,open,open,open,Ps11,
                                      Ps12,Ps13,Ps21,Ps22,Ps23,Ps31,
                                      Ps32,Ps33);
  instC2Input   : C2Input port map (
      Uin1,Uin2,Uin3,Ps11,Ps12,Ps13,Ps21,Ps22,Ps23,Ps31,Ps32,Ps33,
      addin1,addin2,addin3,addin4,addin5,addin6,addin7,addin8,addin9,
      addin10,addin11,addin12);
  instpipes4: pipe port map (clk,rst,addin1,paddin1);
  instpipes5: pipe port map (clk,rst,addin2,paddin2);
  instpipes6: pipe port map (clk,rst,addin3,paddin3);
  instpipes7: pipe port map (clk,rst,addin4,paddin4);
  instpipes8: pipe port map (clk,rst,addin5,paddin5);
  instpipes9: pipe port map (clk,rst,addin6,paddin6);
  instpipes10: pipe port map (clk,rst,addin7,paddin7);
  instpipes11: pipe port map (clk,rst,addin8,paddin8);
  instpipes12: pipe port map (clk,rst,addin9,paddin9);
  instpipes13: pipe port map (clk,rst,addin10,paddin10);
  instpipes14: pipe port map (clk,rst,addin11,paddin11);
  instpipes15: pipe port map (clk,rst,addin12,paddin12);
  instSommateur : Sommateur_J3 port map(clk,rst,paddin1,paddin2,
```

```
              paddin3,paddin4,paddin5,paddin6,paddin7,paddin8,paddin9,
               paddin10,paddin11,paddin12,addout1,addout2,addout3);
  instpipes16: pipe generic map(N+2)port map(clk,rst,addout1,paddout1);
  instpipes17: pipe generic map(N+2)port map(clk,rst,addout2,paddout2);
  instpipes18: pipe generic map(N+2)port map(clk,rst,addout3,paddout3);
  instC2Output  : C2Output port map (paddout1,paddout2,paddout3,
                                     satin1,satin2,satin3);
  instpipes1: pipe generic map(N+2)port map(clk,rst,satin1,psatin1);
  instpipes2: pipe generic map(N+2)port map(clk,rst,satin2,psatin2);
  instpipes3: pipe generic map(N+2)port map(clk,rst,satin3,psatin3);
  instSaturation: Saturation port map (psatin1,psatin2,psatin3,
                                     act1,act2,act3);
  iac2:activ2 port map (act1,act2,act3,IBE,clk,CEs,Lin1,Lin2,Lin3,OBE);

      L1<=Lin1;
      L2<=Lin2;
      L3<=Lin3;
end;
```

Entité double décodeur

```
library ieee;
use ieee.std_logic_1164.all;
use work.ma_library.all;

entity double_decodeur_piped is
port(
      clk,rst,InOK            : in std_logic;
      U1,U2,U3,P1,P2,P3       : in echantillon;
      L1,L2,L3                : out std_logic;
      OutOK                   : out std_logic
      );
end double_decodeur_piped;

architecture arch_double_decodeur_piped of double_decodeur_piped is
component decodeur_frontal_piped
port (
      clk,rst,InOK            : in std_logic;
      Ue1,Ue2,Ue3,Pe1,Pe2,Pe3 : in echantillon;
      Us1,Us2,Us3             : out echantillon;
      Ps1,Ps2,Ps3             : out echantillon;
      L1,L2,L3                : out echantillon;
      OutOK                   : out std_logic
      );

end component;

component decodeur_terminal_piped
port (
      clk,rst,IBE                          : in std_logic;
      Ue1,Ue2,Ue3,Pe1,Pe2,Pe3,Le1,Le2,Le3 : in echantillon;
      L1,L2,L3                             : out echantillon;
```

```
      OBE                                    : out std_logic
      );
end component;

component decision_J3
port(
      L1,L2,L3                : in std_logic_vector (N+1 downto N-2);
      S1,S2,S3                : out std_logic);
end component;

signal Uin1,Uin2,Uin3,Pin1,Pin2,Pin3      : echantillon;
signal Lin1,Lin2,Lin3                     : echantillon;
signal Li1,Li2,Li3                        : echantillon;
signal OBE1                               : std_logic;

begin

   idcf : decodeur_frontal_piped port map (clk,rst,IBE,U1,U2,U3,P1,
                          P2,P3,Uin1,Uin2,Uin3,Pin1,Pin2,Pin3,Lin1,
                          Lin2,Lin3,OBE1);
   idct : decodeur_terminal_piped port map (clk,rst,OBE1,Uin1,Uin2,
                          Uin3,Pin1,Pin2,Pin3,Lin1,Lin2,Lin3,Li1,
                          Li2,Li3,OBE);
   idecision: decision_J3 port map (Li1,Li2,Li3,L1,L2,L3);
end;
```

ANNEXE 2

Code C Pour un générateur de code VHDL du décodeur CSO^2C-SS

CODE C POUR UN GÉNÉRATEUR DE CODE VHDL DU DÉCODEUR CSO^{2}C-SS

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#include<time.h>
#define LONG 10

////////////////////////////////////////////////////////////////////////
// fonction entête pour générer date et résume dans chaque fichier   //
////////////////////////////////////////////////////////////////////////

int entete (int J,int q, FILE *file)
 {
   time_t s;
   struct tm *t;
   char texte[100];
   s= time(NULL);
   t=localtime(&s);
   strftime(texte,100,"%A %d %B %Y à %X %Z",t);
   fprintf(file,"----Projet de maîtrise de Ouadid Abdelkarim\n");
   fprintf(file,"----Génération automatique du code VHDL pour\n");
   fprintf(file,"----un codeur/decodeur doublement orthogonal\n");
   fprintf(file,"----Taux de codage J=%d et quantification sur %d
      bits\n",J,q);
   fprintf(file,"----Fichier générer le :%s\n\n\n",texte);
      }
////////////////////////////////////////////////////////////////////////
//       fonction pour convertir un décimal en binaire                //
////////////////////////////////////////////////////////////////////////
int dec_bin(int n,int q,FILE *file)
{
   int *tab;
   int c,i;
   tab=(int *) malloc(q*sizeof(int));
      for (i=0;i<q;i++)
        {
          c=n%2;
          tab[i]=c;
          n=n/2;
         }


            for (i=q-1;i>=0;i--)
                  fprintf(file,"%d",tab[i]);
   free(tab);
}
////////////////////////////////////////////////////////////////////////
```

```
//  fonction pour calculer le nombre de bit que l'on devra avoir     //
//     a la sortie du sommateur pour gérer l'overflow                //
//////////////////////////////////////////////////////////////////////
int depass (int J)
 {
   int i;
   i=1;
   do
      i++;
      while (J>=pow(2,i));
   return (i);
  }
//////////////////////////////////////////////////////////////////////
// fonction pour Calcul du maximum dans chaque ligne de la matrice   //
//////////////////////////////////////////////////////////////////////
int *maxs (int J, int *tab)
 {
   int *tabl,*Max;
   int i,r;
   tabl=tab;
   Max=(int *) malloc(J*sizeof(int));
   for (i=0;i<J;i++)
         Max[i]=0;
   for (i=0;i<J;i++)
      {
       for (r=0;r<J;r++)
                  {
                  if (Max[i]<tabl[i*J+r])
                        {
                        Max[i]=tabl[i*J+r];
                      }
                  }
        }
   return (Max);
   free(tabl);
   free (Max);
   }
//////////////////////////////////////////////////////////////////////
//     fonction pour calculer la valeur max dans la matrice          //
//////////////////////////////////////////////////////////////////////
int max(int J,int *Maxs)
  {
   int Maxi=0,i;
   int *Max;
   Max=Maxs;
   for (i=0;i<J;i++)
        if (Maxi<Max[i])
             Maxi=Max[i];
   return (Maxi);
   free(Max);
 }
//////////////////////////////////////////////////////////////////////
// Fonction devant générer le code VHDL pour le package             //
```

```
//           de functions, types et constantes                          //
////////////////////////////////////////////////////////////////////////

int mesfonctions (int J,int q,int iter)
  {
   FILE *file;
   int r,s;
   if (file=fopen("ma_library.vhdl","w"))
     {
       entete(J,q,file);
       fprintf(file,"Library ieee;\nuse ieee.std_logic_1164.all;\nuse
             ieee.std_logic_arith.all;\n");
        fprintf(file,"use ieee.std_logic_unsigned.all;\n\n package
             ma_library is\n\t");
       fprintf(file,"constant q: integer :=%d;\n\t",q);
       fprintf(file,"constant N: integer :=%d;\n\t",iter);
       fprintf(file,"constant reset: std_logic_vector :=\"0");
       for(s=1;s<J;s++)
           fprintf(file,"1");
       fprintf(file,"\";\n\t");
       fprintf(file,"subtype echantillon is std_logic_vector (q-1
             downto 0);\n\t");
       fprintf(file,"function ADM(l1,l2:echantillon) return
             echantillon;\n\t");
       for (r=2;r<=J-1;r++)
            {
            fprintf(file,"function ADM(l1,l2");
                        for(s=1;s<r;s++)
                              fprintf(file,",l%d",2+s);
            fprintf(file,":echantillon) return echantillon;\n\t");
            }
       fprintf(file,"function ADD(a,b:std_logic_vector) return
             std_logic_vector;\n\t");
       fprintf(file,"function c2(a:std_logic_vector) return
             std_logic_vector;\n\t");
       fprintf(file,"function sature(a:std_logic_vector) return
             std_logic_vector;\n\t");
       fprintf(file,"end ma_library;\n\npackage body ma_library is\n");
       fprintf(file,"--fonction addition avec extension du MSB\n\t");
       fprintf(file,"function ADD (a,b: std_logic_vector) return
             std_logic_vector is\n\t");
        fprintf(file,"begin\n\t\treturn((a(a'high)&a)+(b(b'high)&b));\n
             \tend ADD;\n");
       fprintf(file,"--fonctions addmin avec surcharge\n\t");
       fprintf(file,"function ADM (l1,l2: echantillon) return
            echantillon is\n\t");
       fprintf(file,"begin\n\t\t if ((l1(l1'high-1 downto 0))");
       fprintf(file," <=(l2(l2'high-1 downto 0)))
            then\n\t\t\treturn((l1(l1'high) xor l2(l2'high))&");
        fprintf(file," l1(l1'high-1 downto
            0));\n\t\telse\n\t\t\treturn((l1(l1'high) xor
            l2(l2'high))&");
       fprintf(file," l2(l2'high-1 downto 0));\n\t\t");
```

```
        fprintf(file,"end if;\n\tend ADM;\n\t");

        for (r=2;r<=J-1;r++)
             {
             fprintf(file,"function ADM(l1,l2");
                        for(s=1;s<r;s++)
                              fprintf(file,",l%d",2+s);
             fprintf(file,":echantillon) return echantillon is
                   \n\tbegin\n\t\t");
             fprintf(file,"return(ADM(l1,ADM(l2");
                        for(s=1;s<r;s++)
                          fprintf(file,",l%d",2+s);
             fprintf(file,")));\n\tend ADM;\n");
             }
        fprintf(file,"--function du complement a deux\n\t ");
        fprintf(file,"function c2 (a: std_logic_vector) return ");
        fprintf(file," std_logic_vector is\n\tbegin\n\t\t");
        fprintf(file," if (a(a'high)='1') then\n\t\t\t return(");
        fprintf(file," a(a'high)&(\"0\"-a(a'high-1 downto 0)));\n\t\t");
        fprintf(file," else\n\t\t\treturn(a);\n\t\tend if;\n\t");
        fprintf(file,"end c2;\n\t");
        fprintf(file,"--function de la saturation\n\t ");
        fprintf(file,"function sature (a: std_logic_vector) return ");
        fprintf(file," std_logic_vector is\n\tbegin\n\t\tif (a(a'high-
                1)");
        for (r=2;r<=depass(J);r++)
             fprintf(file,"or a(a'high-%d)",r);
        fprintf(file,"='1') then\n\t\t\t return(");
        fprintf(file,"a(a'high)&\"");
        for(r=1;r<q;r++)
             fprintf(file,"1");
        fprintf(file,"\");\n\t\t");
        fprintf(file," else\n\t\t\treturn(a(a'high)&a(q-2 downto
                0));\n\t\tend if;\n\t");
        fprintf(file,"end sature;\n\t");
        fprintf(file,"end ma_library;\n\t");
        printf("Fichier ma_library.vhdl cree\n");
          }
       else
          printf("Probleme pour creer le fichier ma_library.vhdl\n");
    return 0;
}




////////////////////////////////////////////////////////////////////
//fonction pour generer le code VHDL du bloc de prise de decision    //
////////////////////////////////////////////////////////////////////

int decision (int J,int q)
```

```
  {
   int i;
   FILE *file;
   if(file=fopen("decision.vhdl","w"))
      {
        entete(J,q,file);
       fprintf(file,"Library ieee;\nuse ieee.std_logic_1164.all;\nuse
                 work.ma_library.all;\n\n");
        fprintf(file,"entity decision is\n\t port (\n\t\t");
        for (i=1;i<=J;i++)
              fprintf (file,"En%d : in echantillon;\n\t\t",i);
        for (i=1;i<=J-1;i++)
              fprintf (file,"So%d : out std_logic;\n\t\t",i);
        fprintf (file,"So%d : out std_logic);\n",J);
        fprintf (file,"end decision;\n\n");
         fprintf (file,"Architecture Arch_decision of decision
                  is\n\tbegin\n\t",J,J);
        for (i=1;i<=J-1;i++)
               fprintf (file,"So%d <= En%d(En%d'high);\n\t",i,i,i);
        fprintf (file,"So%d <= En%d(En%d'high);\n\tend;",J,J,J);
        fclose(file);
        printf("Fichier decision.vhdl cree\n");
      }
else
      printf("Probleme pour creer le fichier decision.vhdl\n");
return 0;
}
////////////////////////////////////////////////////////////////////////
//  fonction pour generer le code VHDL du bloc de pipeline            //
////////////////////////////////////////////////////////////////////////


int pipeline (int J, int q)
 {
  int i;
  FILE *file;
  if(file=fopen("pipe.vhdl","w"))
   {
    entete(J,q,file);
    fprintf(file,"Library ieee;\nuse ieee.std_logic_1164.all;\nuse
           work.ma_library.all;\n\n");
    fprintf(file,"entity pipe is\n\tgeneric (taille : integer
           :=q);\n\t");
    fprintf(file,"port (\n\t\tclk,rst: in std_logic;\n\t\t");
    fprintf (file,"En : in echantillon;\n\t\t");
    fprintf (file,"So : out echantillon);\n");
    fprintf (file,"end pipe;\n\n");
    fprintf (file,"Architecture Arch_pipe of pipe is\n\tbegin\n\t");
    fprintf (file,"process(clk)\n\t\tbegin\n\t\tif (clk'event and
           clk='1') then\n\t\t\t");
    fprintf (file,"if rst='1' then\n\t\t\t\t So<=(others=>'0');
           \n\t\t\telse");
```

```
    fprintf (file,"\n\t\t\t\tSo<=En;\n\t\t\tend if;\n\t\tend
            if;\n\tend process;");
    fprintf (file,"\nend;");
    fclose(file);
    printf("Fichier pipe.vhdl cree\n");
 }
  else
    printf("Probleme pour creer le fichier pipe.vhdl\n");
    return 0;
}
////////////////////////////////////////////////////////////////////////
// fonction pour generer le code VHDL du bloc de complement 2 entrée //
////////////////////////////////////////////////////////////////////////

int c2input (int J,int q)
 {
  int i;
  int inmax;
  FILE *file;
  if(file=fopen("c2input.vhdl","w"))
   {
    entete(J,q,file);
    fprintf(file,"Library ieee;\nuse ieee.std_logic_1164.all;\nuse
            work.ma_library.all;\n\n");
    fprintf(file,"entity c2input is\n\tport (\n\t\t");
    inmax=J*(J+1);
    for (i=1;i<=inmax;i++)
            fprintf (file,"En%d : in echantillon;\n\t\t",i);
    for (i=1;i<inmax;i++)
            fprintf (file,"So%d : out echantillon;\n\t\t",i);
    fprintf (file,"So%d : out echantillon);\n\t",inmax);
    fprintf (file,"end c2input;\n\n");
    fprintf (file,"Architecture Arch_c2input of c2input is\n
            begin\n\t");
    for (i=1;i<inmax;i++)
            fprintf (file,"So%d <= c2(En%d);\n\t",i,i);
    fprintf (file,"So%d <= c2(En%d);\n",inmax,inmax);
    fprintf (file,"end;");
    fclose(file);
    printf("Fichier c2input.vhdl cree\n");
   }
  else
    printf("Probléme pour creer le fichier c2input.vhdl\n");
  return 0;
}



////////////////////////////////////////////////////////////////////////
// fonction pour generer le code VHDL du bloc de complement 2 sortie //
////////////////////////////////////////////////////////////////////////
```

```
int c2output (int J,int q)
 {
  int i;
  FILE *file;
  if(file=fopen("c2output.vhdl","w"))
   {
    entete(J,q,file);
    fprintf(file,"Library ieee;\nuse ieee.std_logic_1164.all;\nuse
            work.ma_library.all;\n\n");
    fprintf(file,"entity c2output is\n\tport (\n\t\t");
    for (i=1;i<=J;i++)
      fprintf (file,"En%d : in std_logic_vector(q+%d downto
                  0);\n\t\t",i,depass(J)-1);
    for (i=1;i<J;i++)
      fprintf (file,"So%d : out std_logic_vector(q+%d downto
                  0);\n\t\t",i,depass(J)-1);
    fprintf (file,"So%d : out std_logic_vector(q+%d downto
            0));\n\t",J,depass(J)-1);
    fprintf (file,"end c2output;\n\n");
    fprintf (file,"Architecture Arch_c2output of c2output
            is\nbegin\n\t");
    for (i=1;i<J;i++)
       fprintf (file,"So%d <= c2(En%d);\n\t",i,i);
    fprintf (file,"So%d <= c2(En%d);\n",J,J);
    fprintf (file,"end;");
    fclose(file);
    printf("Fichier c2output.vhdl cree\n");
   }
  else
    printf("Probleme pour creer le fichier c2output.vhdl\n");
  return 0;
}

////////////////////////////////////////////////////////////////////////
// fonction pour générer code VHDL du bloc pour initialiser les fifos//
////////////////////////////////////////////////////////////////////////

int activ2 (int J,int q,int Maxi)
 {
  int i;
  int inmax;
  FILE *file;
  if(file=fopen("activ2.vhdl","w"))
   {
    entete(J,q,file);
    fprintf(file,"Library ieee;\nuse ieee.std_logic_1164.all;\n");
    fprintf(file,"use ieee.std_logic_arith.all;\nuse
            work.ma_library.all;\n\n");
    fprintf(file,"entity activ2 is\n\tport (\n\t\t");
    fprintf(file,"clk,InOK\t: in std_logic;\n\t\t");
    fprintf(file,"OutOK\t: out std_logic;\n\t\t");
    for (i=1;i<=J;i++)
      fprintf (file,"En%d : in echantillon;\n\t\t",i);
```

```
    for (i=1;i<J;i++)
      fprintf (file,"So%d : out echantillon;\n\t\t",i);
    fprintf (file,"So%d : out echantillon);\n\t",J);
    fprintf (file,"end activ2;\n\n");
    fprintf (file,"Architecture Arch_activ2 of activ2 is\nsignal
            sel:std_logic;\n\t");
    fprintf (file,"begin\n\tprocess(clk)\n\t\tvariable count
            :unsigned(q-1 downto 0);");
     fprintf (file,"\n\t\tbegin\n\t\t\tif (clk'event and clk='1')
            then\n\t\t\t\t\t");
     fprintf (file,"if (InOK='0')
            then\n\t\t\t\t\tsel<='0';\n\t\t\t\t\tcount:=(others=>'0');"
            );
    fprintf (file,"\n\t\t\t\t\telsif (InOK='1') then\n\t\t\t\t\t\tif
            (count<%d) then",Maxi);
    fprintf (file,"\n\t\t\t\t\t\tcount:=count+1;\n\t\t\t\t\t\tsel<='0';
            \n\t\t\t\t\t");
    fprintf (file,"else\n\t\t\t\t\t\tsel<='1';\n\t\t\t\t\tend
            if;\n\t\t\t\tend if;");
    fprintf (file,"\n\t\t\tend if;\n\t\tend process;\n\t");
    for (i=1;i<J;i++)
      fprintf (file,"So%d<=En%d when sel='1' else reset;\n\t",i,i);
    fprintf (file,"So%d<=En%d when sel='1' else reset;\n\t",J,J);
    fprintf (file,"end;");
    fclose(file);
    printf("Fichier activ2.vhdl cree\n");
   }
  else
    printf("Probleme pour creer le fichier activ2.vhdl\n");
return 0;
}

////////////////////////////////////////////////////////////////////////
//            fonction pour générer le code VHDL du saturateur        //
////////////////////////////////////////////////////////////////////////

int saturateur (int J, int q)
 {
   FILE *file;
   int i; int g;
   if(file=fopen("saturation.vhdl","w"))
    {
     entete(J,q,file);
     fprintf(file,"Library ieee;\nuse ieee.std_logic_1164.all;\n");
     fprintf(file,"use work.ma_library.all;\n\n");
     fprintf(file,"entity saturer is\n\tport (\n\t\t");
     for (i=1;i<=J;i++)
       fprintf (file,"En%d : in std_logic_vector(q+%d downto 0);\n
            \t\t",i,depass(J)-1);
    for (i=1;i<J;i++)
      fprintf (file,"So%d : out echantillon;\n\t\t",i);
    fprintf (file,"So%d : out echantillon);\nend saturer;\n\n",J);
    fprintf (file,"Architecture Arch_saturer of saturer is\n");
```

```
    fprintf (file,"begin\n\t");
    for (i=1;i<J;i++)
      fprintf (file,"So%d<=sature(En%d);\n\t",i,i);
    fprintf (file,"So%d<=sature(En%d);\nend;",J);
      printf("Fichier saturation.vhdl cree\n");
   }
  else
    printf("Probleme pour creer le fichier saturation.vhdl\n");
  return 0;
}

////////////////////////////////////////////////////////////////////////
//           Fonction pour générer le code VHDL des fifo sans addmin //
////////////////////////////////////////////////////////////////////////

int fifo (int J,int q,int Maxi)
 {
  FILE *file;
  int i; int g;
  file=fopen("FIFO.vhdl","w");
  if(file==NULL)
     printf("Probleme pour creer le code fifo.vhdl\n");
     else
      {
       entete(J,q,file);
       fprintf(file,"Library ieee;\nuse ieee.std_logic_1164.all;\n");
       fprintf(file,"use work.ma_library.all;\n\n");
       fprintf(file,"entity fifo is\n\tport (\n\t\t");
       fprintf(file,"clk : in std_logic;\n\t\t");
       for (i=1;i<=J;i++)
         fprintf (file,"En%d : in echantillon;\n\t\t",i);
       for (i=1;i<J;i++)
         fprintf (file,"Sor%d : out echantillon;\n\t\t",i);
       fprintf (file,"Sor%d : out echantillon);\nend fifo;\n\n",J);
       fprintf (file,"Architecture Arch_fifo of fifo is\n");
       fprintf (file,"Type Tfifo is array (%d downto 0) of
              echantillon;\n",Maxi-1);
       for (i=1;i<=J;i++)
         fprintf (file,"Signal fifo%d : Tfifo;\n",i);
       fprintf (file,"Begin\n\t");
       for (i=1;i<=J;i++)
          {
            fprintf(file,"pro%d : process (clk) begin\n\t\tif(clk'event
                  and clk='1') then\n\t\t\t",i);
            fprintf(file,"fifo%d(%d downto 1)<=fifo%d(%d downto
                  0);\n\t\t\t",i,Maxi-1,i,Maxi-2);
             fprintf(file,"fifo%d(0)<=En%d;\n\t\tend if;\n\tend
                  process;\n\t",i,i);
          }
        for (i=1;i<=J;i++)
         fprintf(file,"Sor%d<=fifo%d(fifo%d'high);\n",i,i,i);
       fprintf(file,"end;");
       printf("Fichier FIFO.vhdl cree\n");
```

```
   }
return 0;
}

////////////////////////////////////////////////////////////////////////
//      Fonction devant générer le code VHDL pour le FIFOULP          //
////////////////////////////////////////////////////////////////////////

int FIFOULP (int J, int q, int *tabl, int *Maxs, int Maxim)
 {
  FILE *file;
  int i=0,r=0,t,p=0,f,Maxi=0,s=0;
  int *tableau,*Max, *ss, *psy;
  /*Matrice du codeur et des maximums   pour chaque ligne*/
  tableau=tabl;
  Max=Maxs;
  Maxi=Maxim;
  psy=(int *) malloc (J*J*sizeof(int ));
  ss=(int *) malloc(J*sizeof(int));
  /*écriture de l'entité dans le fichier VHDL*/
  file=fopen("FIFOULP.vhdl","w");
  if (file==NULL)
     printf("Probleme pour creer le fichier FIFOULP.vhdl\n");
  else
   {
    entete(J,q,file);
    fprintf(file,"Library ieee;\nuse ieee.std_logic_1164.all;\n");
    fprintf(file,"use work.ma_library.all;\n\n");
    fprintf(file,"entity FIFOULP is\n\tport (\n\t\t",J);
    fprintf(file,"clk : in std_logic;\n\t\t");
    for (i=1;i<=J;i++)
          fprintf(file,"Ue%d : in echantillon;\n\t\t",i);
    for (i=1;i<=J;i++)
          fprintf(file,"Pe%d : in echantillon;\n\t\t",i);
    for (i=1;i<=J;i++)
          fprintf(file,"L%d : in echantillon;\n\t\t",i);
    for (i=1;i<=J;i++)
      for (r=1;r<=J;r++)
        if (i==J && r==J)
              fprintf(file,"psy%d%d : out echantillon);\n",i,r);
         else
              fprintf(file,"psy%d%d : out echantillon;\n\t\t",i,r);
    fprintf(file,"end FIFOULP;\n\n",J);
    fprintf(file,"Architecture AFIFOULP of FIFOULP is\n\t",J,J);
    /*Calcul du maximum dans la matrice*/
     for (i=0;i<J;i++)
         {
              ss[i]=0;
          }
    /*Calcul des Psy*/
     s=0;
     for (i=0;i<J;i++)
         for (t=0;t<J;t++)
```

```
            {
                s++;
                for (r=0;r<J;r++)
                     if (r!=i && tableau[i*J+t]-tableau[r*J+t]>=0)
                       {
                       s++;
                       }
                psy[i*J+t]=s;
             }
        fprintf(file,"type TPU is array (%d downto 0) of
                echantillon;\n\t",Maxi-1);
        fprintf(file,"type TS is array (1 to %d) of echantillon;\n\t",s);
        fprintf(file,"signal S:TS;\n\t");
        for (i=1;i<=J;i++)
             fprintf(file,"signal fifoU%d : TPU;\n\t",i);
        for (i=1;i<=J;i++)
             fprintf(file,"signal fifoP%d : TPU;\n\t",i);
        fprintf(file,"Begin\n\n\t",i);
        for (i=1;i<=J;i++)
          {
             fprintf(file,"U%d : process (clk) begin\n\t\t",i);
             fprintf(file,"if (clk'event and clk='1') then\n\t\t\t");
             fprintf(file,"for i in %d downto 1 loop\n\t\t\t\t fifoU%d(i)
                <=fifoU%d(i-1);",Maxi-1,i,i);
             fprintf(file,"\n\t\t\tend loop;\n\t\t\t\tfifoU%d(0)<=Ue%d;\n
                \t\tend if;\n\tend process;\n\n",i,i);
          }
    /* Calcul des addmin devant intervenir dans le fifoP*/
        s=0;
        for (i=1;i<=J;i++)
          {
             fprintf(file,"P%d : process (clk) begin\n\t\t",i);
             fprintf(file,"if (clk'event and clk='1') then\n\t\t\t");
             for (t=Maxi-1;t>0;t--)
                {
                 for (f=0;f<J;f++)
                  {
                   if (tableau[f*J+(i-1)]!=0 && tableau[f*J+(i-1)]==Maxi-t)
                      {
                        ss[s]=f+1;
                        s++;
                       }
                  }
                p=0;
                if (s==0)
                 fprintf(file,"fifoP%d(%d)<=fifoP%d(%d);\n\t",i,t,i,t-1);
                else


                  if (s==1)
                     fprintf(file,"fifoP%d(%d)<=ADM(fifoP%d(%d),L%d);\n\t\
                           t\t",i,t,i,t-1,ss[0]);
                  else
```

```
     {
      while (s>1)
         {
          if (p==0)
            {
             fprintf(file,"S%d_%d_%d<=ADM(fifoP%d(%d),L%d);
                    \n\t\t\t",i,t-1,s-1,i,t-1,ss[s-1]);
             p=1;
             }
          else
           fprintf(file,"S%d_%d_%d<=ADM(S%d_%d_%d,L%d);
                  \n\t\t\t",i,t-1,s-1,i,t-1,s,ss[s-1]);
           s--;
          }
          p=0;

          fprintf(file,"fifoP%d(%d)<=ADM(S%d_%d_%d,L%d);
                  \n\t\t\t",i,t,i,t-1,s,ss[s-1]);
      }
    s=0;
    for (r=0;r<J;r++) ss[r]=0;
   }

for (f=0;f<J;f++)
 {
  if (tableau[f*J+(i-1)]!=0 && tableau[f*J+(i-1)]==Maxi)
     {
      ss[s]=f+1;
      s++;
     }
  }
p=0;
if (s==0)
   fprintf(file,"fifoP%d(%d)<=Pe%d;\n\t\t\t",i,0,i);
else
  if (s==1)
    fprintf(file,"fifoP%d(%d)<=ADM(Pe%d,L%d);\n\t\t\t"
           ,i,0,i,ss[0]);
  else
     {
      while (s>1)
         {
           if (p==0)
             {
   fprintf(file,"fifoP%d(%d)<=ADM(Pe%d,L%d",i,t,i,ss[s-1]);
              p=1;
              }
           else
              fprintf(file,",L%d",ss[s-1]);
               s--;
               }
            p=0;
            fprintf(file,",L%d);\n\t\t\t",ss[s-1]);
```

```
          }
  fprintf(file,"\n\t\t");
  fprintf(file,"end if;\n\tend process;\n\n\t");
  s=0;
  for (r=0;r<J;r++) ss[r]=0;
}
fprintf(file,"\n");
/*calcul des S(i)*/
fprintf(file,"\n\t");
s=0;
for (i=0;i<J;i++)
    for (t=0;t<J;t++)
    {
        fprintf(file,"--signaux pour psy%d%d\n\t",i+1,t+1);
        s++;
        if (tableau[i*J+t]==Maxi)
            fprintf(file,"s(%d)<=Pe%d;\n\t",s,t+1);
        else
          fprintf(file,"s(%d)<=fifoP%d(%d);\n\t",s,t+1,Maxi-1-
                  tableau[i*J+t]);
        for (r=0;r<J;r++)
             if (r!=i && tableau[i*J+t]-tableau[r*J+t]>=0)
               {
               s++;
               if (tableau[i*J+t]-tableau[r*J+t]==Maxi)
                 fprintf(file,"s(%d)<=Ue%d;\n\t",s,r+1);
               else
                 fprintf(file,"s(%d)<=fifoU%d(%d);\n\t",s,r+1,
                         Maxi-1-tableau[i*J+t]+tableau[r*J+t]);
               }
     }
fprintf(file,"\n\t");
/*Les sorties*/
for (i=0;i<J;i++)
    for (r=0;r<J;r++)
    {
        if (i==0 && r==0)
         {
           if (psy[0]==1)
                fprintf(file,"psy11<=S(1);\n\t");
           else
                {
                fprintf(file,"psy11<=ADM(S(1), ");
                for (f=0;f<psy[0]-2;f++)
                   fprintf(file,"S(%d), ",f+2);
                fprintf(file,"S(%d));\n\t",psy[0]);
                 }
           }

       else
            {
            s=(psy[i*J+r]-psy[i*J+r-1]);
```

```
              if (s==1)
                fprintf(file,"psy%d%d<=S(%d);\n\t",i+1,r+1,psy[i*J+r]);
               else
                {
              fprintf(file,"psy%d%d<=ADM(S(%d),",i+1,r+1,psy[i*J+r-1]+1);
                for (f=1;f<s-1;f++)
                    fprintf(file,"S(%d),",psy[i*J+r-1]+f+1);
                fprintf(file,"S(%d));\n\t",psy[i*J+r]);
                   }
                }
            }
     fprintf(file,"\n");
     fprintf(file,"end;\n");
     free(tableau);
     free(psy);
     free (Max);
     free (ss);
     free(file);
     printf("Fichier FIFOULP.vhdl cree\n");
     }
     return 0;
     }

////////////////////////////////////////////////////////////////////////
//       Fonction devant generer le code VHDL pour le sommateur       //
////////////////////////////////////////////////////////////////////////

int sommateur (int J,int q, char *c)
 {
  int i,n,r,t;
  char pipe;
  FILE *file;
  file=fopen("addition.vhdl","w");
  if (file==NULL)
     printf("Probleme pour cree le fichier addition.vhdl\n");
  else
   {
    entete(J,q,file);
    pipe=toupper(*c);
    fprintf(file,"Library ieee;\nuse ieee.std_logic_1164.all;\n");
    fprintf(file,"use ieee.std_logic_arith.all;\nuse
            ieee.std_logic_signed.all;\n");
    fprintf(file,"use work.ma_library.all;\n\n");
    fprintf(file,"entity addition_J%d is\n\t",J);
    fprintf(file,"port(\n\t\t");
    if (pipe=='P')
         fprintf(file,"port(\n\t\tclk:in std_logic;\n\t\t");
    for (i=1;i<=J+1;i++)
            fprintf(file,"S0%d\t:in echantillon;\n\t\t",i);
    fprintf(file,"SE\t:out std_logic_vector (q+%d downto
            0));\n",depass(J)-1);
    fprintf(file,"end addition_J%d;\n\n",J);
```

```
fprintf(file,"Architecture Arch_addition_J%d of addition_J%d
        is\n\t",J,J);
t=1;
n=0;
r=J+1;
do
 {
  if (r%2==0)
   n= r/2;
  else
   n=r/2+1;
  for (i=1;i<=n;i++)
   fprintf(file,"signal S%d%d : std_logic_vector (q+%d downto
        0);\n\t",t,i,t-1);
  t=t+1;
  r=n;
  }
while (n!=2);
fprintf(file,"begin\n\t");
n=1;
t=1;
r=J+1;
while (r!=1)
 {
  for (i=0;i<=2*r/2-2;i+=2)
    {
     if (r!=2)
      {
       if(pipe=='P')
         {
          fprintf(file,"process(clk) begin\n\t\t");
          fprintf(file,"if (clk'event and clk='1')then\n\t\t\t");
          fprintf(file,"S%d%d<=ADD(S%d%d,S%d%d);\n\t\t",n,t,n-
              1,i+1,n-1,i+2);
          fprintf(file,"end if;\n\t end process;\n\t");
          }
       else
         fprintf(file,"S%d%d<=ADD(S%d%d,S%d%d);\n\t",n,t,n-1,i+1,n-
              1,i+2);
         t=t+1;
      }
     else
      {
       fprintf(file,"SE<=ADD(S%d%d,S%d%d);\n\t",n-1,i+1,n-1,i+2);
       t=t+1;
       }
      }
     if (r%2!=0)
       {
        if(pipe=='P')
         {
          fprintf(file,"process(clk) begin;\n\t");
          fprintf(file,"if (clk'event and clk='1' then\n\t\t");
```

```
            fprintf(file,"S%d%d<='0'&S%d%d;\n\t",n,t,n-1,r);
            fprintf(file,"end if;\n\t end process;\n\t");
           }
          else
            fprintf(file,"S%d%d<='0'&S%d%d;\n\t",n,t,n-1,r);
            t=t+1;
           }
       r=t-1;
       t=1;
       n=n+1;
       }
fprintf(file,"end;\n\n");
//creation du bloc de sommation
fprintf(file,"Library ieee;\nuse ieee.std_logic_1164.all;\n");
fprintf(file,"use ieee.std_logic_arith.all;\nuse
    ieee.std_logic_signed.all;\n");
fprintf(file,"use work.ma_library.all;\n\n");
fprintf(file,"entity Sommateur_J%d is\n\t",J);
fprintf(file,"port(\n\t\tclk\t:in std_logic;\n\t\t");
for (i=1;i<=J;i++)
 fprintf(file,"Ue%d\t:in std_logic_vector (q-1 downto 0);\n\t\t",i);
for (i=1;i<=J;i++)
  for (r=1;r<=J;r++)
    fprintf(file,"Psy%d%d\t:in std_logic_vector (q-1 downto
          0);\n\t\t",i,r);
    for (i=1;i<J;i++)
       fprintf(file,"L%d\t:out std_logic_vector (q+%d downto
                0);\n\t\t",i,depass(J)-1);
    fprintf(file,"L%d\t:out std_logic_vector (q+%d downto
                0));\n",J,depass(J)-1);
    fprintf(file,"end Sommateur_J%d;\n\n",J);
    fprintf(file,"Architecture Arch_Sommateur_J%d of Sommateur_J%d
          is\n",J,J);
    fprintf(file,"component addition_J%d\n\t",J);
    fprintf(file,"port(\n\t\tclk:in std_logic;\n\t\t");
    for (i=1;i<=J+1;i++)
    fprintf(file,"S0%d\t:in std_logic_vector (q-1 downto
          0);\n\t\t",i);
    fprintf(file,"SE\t:out std_logic_vector (q+%d downto
          0));\n",depass(J)-1);
    fprintf(file,"end component;\n\t");
    fprintf(file,"begin\n\t");
    for (i=1;i<=J;i++)
     {
      fprintf(file,"inst_addition%d : addition_J%d port map
          (clk,Ue%d,\n\t",i,J,i);
      for (r=1;r<=J;r++)
        fprintf(file,"Psy%d%d,",i,r);
        fprintf(file,"L%d);\n\t",i);
     }
    fprintf(file,"end;");
    fclose(file);
    printf("Fichier addition.vhdl cree\n");
```

```
      }
   return 0;
      }

////////////////////////////////////////////////////////////////////////
//   fonction pour générer code VHDL de la library de composantes    //
////////////////////////////////////////////////////////////////////////

int composants (int J, int q, char *c)
 {
  int i,r;
  int inmax;
  char pipe;
  FILE *file;
  pipe=toupper(*c);
  file=fopen("composants.vhdl","w");
  if(file==NULL)
   printf("Probleme pour creer le fichier composants.vhdl\n");
  else
   {
    entete(J,q,file);
    fprintf(file,"Library ieee;\nuse ieee.std_logic_1164.all;\nuse
      work.ma_library.all;\n\n");
    fprintf(file,"package composants is\n");
    fprintf(file,"component fifo is\n\tport (\n\t\t");
    fprintf(file,"clk : in std_logic;\n\t\t");
    for (i=1;i<=J;i++)
      fprintf (file,"En%d : in echantillon;\n\t\t",i);
    for (i=1;i<J;i++)
      fprintf (file,"Sor%d : out echantillon;\n\t\t",i);
    fprintf (file,"Sor%d : out echantillon);\nend component;\n\n",J);
    fprintf(file,"component activ2 is\n\tport (\n\t\t");
    fprintf(file,"clk,InOK\t: in std_logic;\n\t\t");
    fprintf(file,"OutOK\t: out std_logic;\n\t\t");
    for (i=1;i<=J;i++)
      fprintf (file,"En%d : in echantillon;\n\t\t",i);
    for (i=1;i<J;i++)
      fprintf (file,"So%d : out echantillon;\n\t\t",i);
    fprintf (file,"So%d : out echantillon);\n",J);
    fprintf (file,"end component;\n\n");
    fprintf(file,"component c2output is\n\tport (\n\t\t");
    for (i=1;i<=J;i++)
     fprintf (file,"En%d : in std_logic_vector(q+%d downto
            0);\n\t\t",i,depass(J)-1);
    for (i=1;i<J;i++)
     fprintf (file,"So%d : out std_logic_vector(q+%d downto
            0);\n\t\t",i,depass(J)-1);
    fprintf (file,"So%d : out std_logic_vector(q+%d downto
            0));\n",J,depass(J)-1);
    fprintf (file,"end component;\n\n");
    fprintf(file,"component c2input is\n\tport (\n\t\t");
    inmax=J*(J+1);
    for (i=1;i<=inmax;i++)
```

```
 fprintf (file,"En%d : in echantillon;\n\t\t",i);
for (i=1;i<inmax;i++)
 fprintf (file,"So%d : out echantillon;\n\t\t",i);
fprintf (file,"So%d : out echantillon);\n",inmax);
fprintf (file,"end component;\n\n");
fprintf(file,"component pipe is\n\t port (\n\t\tclk,rst: in
        std_logic;\n\t\t");
fprintf (file,"En : in echantillon;\n\t\t");
fprintf (file,"So : out echantillon);\n");
fprintf (file,"end component;\n\n");
fprintf(file,"component Sommateur_J%d is\n\t",J);
if(pipe=='P')
  fprintf(file,"port(\n\t\tclk\t:in std_logic;\n\t\t");
for (i=1;i<=J;i++)
 fprintf(file,"Ue%d\t:in std_logic_vector (q-1 downto
        0);\n\t\t",i);
for (i=1;i<=J;i++)
  for (r=1;r<=J;r++)
   fprintf(file,"Psy%d%d\t:in std_logic_vector (q-1 downto
        0);\n\t\t",i,r);
for (i=1;i<J;i++)
   fprintf(file,"L%d\t:out std_logic_vector (q+%d downto
        0);\n\t\t",i,depass(J)-1);
    fprintf(file,"L%d\t:out std_logic_vector (q+%d downto
        0));\n",J,depass(J)-1);
    fprintf(file,"end component;\n\n",J);
//FIFOULP
fprintf(file,"component FIFOULP is\n\tport (\n\t\t");
fprintf(file,"clk : in std_logic;\n\t\t");
for (i=1;i<=J;i++)
  fprintf(file,"Ue%d : in echantillon;\n\t\t",i);
for (i=1;i<=J;i++)
  fprintf(file,"Pe%d : in echantillon;\n\t\t",i);
for (i=1;i<=J;i++)
  fprintf(file,"L%d : in echantillon;\n\t\t",i);
for (i=1;i<=J;i++)
  for (r=1;r<=J;r++)
    if (i==J && r==J)
      fprintf(file,"psy%d%d : out echantillon);\n",i,r);
    else
      fprintf(file,"psy%d%d : out echantillon;\n\t\t",i,r);
fprintf(file,"end component;\n");
fprintf (file,"\nend composants;");
fclose(file);
printf("Fichier composants.vhdl cree\n");
    }
    return 0;
  }


////////////////////////////////////////////////////////////////////
// Fonction devant generer le code VHDL pour le decodeur frontal  //
////////////////////////////////////////////////////////////////////
```

```
int decofrontal (int J,int q,int Maxi,char *c)
 {
  FILE *file;
  int i,t; int g;
  char pipe;
  file=fopen("decofrontal.vhdl","w");
  if(file==NULL)
   printf("Probleme pour creer le fichier decofrontal.vhdl\n");
  else
    {
     entete(J,q,file);
     pipe=toupper(*c);
     fprintf(file,"Library ieee;\nuse ieee.std_logic_1164.all;\n");
     fprintf(file,"use work.ma_library.all;\nuse
            work.composants.all;\n\n");
     fprintf(file,"entity decodeur_frontal is\n\tport (\n\t\t");
     if (pipe=='P')
      fprintf(file,"clk,rst,InOK : in std_logic;\n\t\t");
     else
      fprintf(file,"clk,InOK : in std_logic;\n\t\t");
     for (i=1;i<=J;i++)
      fprintf (file,"Ue%d : in echantillon;\n\t\t",i);
     for (i=1;i<=J;i++)
      fprintf (file,"Pe%d : in echantillon;\n\t\t",i);
     for (i=1;i<=J;i++)
      fprintf (file,"Us%d : out echantillon;\n\t\t",i);
     for (i=1;i<=J;i++)
      fprintf (file,"Ps%d : out echantillon;\n\t\t",i);
     for (i=1;i<J;i++)
      fprintf (file,"L%d : out echantillon;\n\t\t",i);
     fprintf (file,"L%d : out echantillon;\n\t\t",J);
     fprintf(file,"OutOK : out std_logic);\nend decodeur_frontal;
            \n\n");
      fprintf (file,"Architecture Arch_decodeur_frontal of
            decodeur_frontal is\n\n");
     /*declaration des signaux*/
     for (i=1;i<=J;i++)
      for(g=1;g<=J;g++)
       fprintf(file," signal Ps%d%d: echantillon;\n ",i,g);
     for(g=1;g<=J;g++)
      fprintf(file," signal Uin%d: echantillon;\n ",g);
     for(g=1;g<=J*(J+1);g++)
      fprintf(file," signal addin%d: echantillon;\n ",g);
     for(g=1;g<=J;g++)
      fprintf(file," signal addout%d: std_logic_vector(q+%d downto
            0);\n ",g,depass(J)-1);
     for(g=1;g<=J;g++)
      fprintf(file," signal satin%d: std_logic_vector(q+%d downto 0);\n
            ",g,depass(J)-1);
      for(g=1;g<=J;g++)
       fprintf(file," signal mux%d: echantillon;\n ",g);
      for(g=1;g<=J;g++)
```

```
 fprintf(file," signal Lin%d: echantillon;\n",g,depass(J)-1);
   if (pipe=='P')
    {
     for(g=1;g<=J*(J+1);g++)
      fprintf(file," signal paddin%d: echantillon;\n ",g);
     for(g=1;g<=J;g++)
      fprintf(file," signal paddout%d: std_logic_vector(q+%d
            downto 0);\n ",g,depass(J)-1);
     for(g=1;g<=J;g++)
      fprintf(file," signal psatin%d: std_logic_vector(q+%d
            downto 0);\n ",g,depass(J)-1);
     }
fprintf(file,"begin\n");
fprintf(file,"instfifo : fifo port map (clk,");
for(g=1;g<=J;g++)
   fprintf(file,"Pe%d,",g);
for(g=1;g<J;g++)
   fprintf(file,"Ps%d,",g);
fprintf(file,"Ps%d);\n",J);
fprintf(file,"instfifoULP : fifoULP port map (clk,");
for(g=1;g<=J;g++)
 fprintf(file,"Ue%d,",g);
for(g=1;g<=J;g++)
 fprintf(file,"Pe%d,",g);
for(g=1;g<=J;g++)
 fprintf(file,"Lin%d,",g);
for(g=1;g<=J;g++)
 fprintf(file,"Uin%d,",g);
for (i=1;i<=J;i++)
 for(g=1;g<=J;g++)
  {
   if (i==J & g==J)
    fprintf(file,"Ps%d%d);\n",J,J);
   else
    fprintf(file,"Ps%d%d,",i,g);
   }
fprintf(file,"instC2Input : C2Input port map (");
for(g=1;g<=J;g++)
 fprintf(file,"Uin%d,",g);
for (i=1;i<=J;i++)
  for(g=1;g<=J;g++)
    fprintf(file,"Ps%d%d,",i,g);
for(g=1;g<J*(J+1);g++)
 fprintf(file,"addin%d,",g);
fprintf(file,"addin%d);\n\t",J*(J+1));
t=1;
if (pipe=='P')
 {
  for(g=1;g<=J*(J+1);g++)
   {
    fprintf(file," intpipes%d:pipe port map (clk,rst,addin%d,
           addin%d);\n ",t,g,g);
    t++;
```

```
  }
}
if (pipe=='P')
 {
  fprintf(file,"instsommateur : sommateur_J%d port map (clk,",J);
  for(g=1;g<=J*(J+1);g++)
   fprintf(file,"paddin%d,",g);
  for(g=1;g<J;g++)
   fprintf(file,"addout%d,",g);
  fprintf(file,"addout%d);\n\t",J);
  }
else
  {
   fprintf(file,"instsommateur : sommateur_J%d port map (",J);
   for(g=1;g<=J*(J+1);g++)
    fprintf(file,"addin%d,",g);
   for(g=1;g<J;g++)
    fprintf(file,"addout%d,",g);
   fprintf(file,"addout%d);\n\t",J);
  }
 if (pipe=='P')
  {
   for(g=1;g<=J;g++)
    {
      fprintf(file," intpipes%d:pipe generic map(q+%d) port map
                  (clk,rst,addout%d,paddout%d);\n ",depass(J)-
                  1,t,g,g);
      t++;
    }
  }

 if (pipe=='P')
  {
   fprintf(file,"instC2Output : C2Output port map (");
   for(g=1;g<=J;g++)
    fprintf(file,"paddout%d,",g);
   for(g=1;g<J;g++)
    fprintf(file,"satin%d,",g);
   fprintf(file,"satin%d);\n\t",J);
  }
 else
  {
   fprintf(file,"instC2Output : C2Output port map (");
   for(g=1;g<=J;g++)
    fprintf(file,"addout%d,",g);
   for(g=1;g<J;g++)
    fprintf(file,"satin%d,",g);
   fprintf(file,"satin%d);\n\t",J);
   }

 if (pipe=='P')
  {
   for(g=1;g<=J;g++)
```

```
          {
           fprintf(file," intpipes%d:pipe generic map(q+%d) port map
                  (clk,rst,satin%d,psatin%d);\n ",depass(J)-1,t,g,g);
           t++;
           }
         }
       if (pipe=='P')
         {
          fprintf(file,"instsaturer : saturer port map (");
          for(g=1;g<=J;g++)
           fprintf(file,"psatin%d,",g);
          for(g=1;g<J;g++)
           fprintf(file,"mux%d,",g);
          fprintf(file,"mux%d);\n\t",J);
         }
       else
         {
          fprintf(file,"instsaturer : saturer port map (");
          for(g=1;g<=J;g++)
           fprintf(file,"satin%d,",g);
          for(g=1;g<J;g++)
           fprintf(file,"mux%d,",g);
          fprintf(file,"mux%d);\n\t",J);
          }
       fprintf(file,"instmux : activ2 port map (clk,InOK,OutOK,");
       for(g=1;g<=J;g++)
        fprintf(file,"mux%d,",g);
       for(g=1;g<J;g++)
        fprintf(file,"Lin%d,",g);
       fprintf(file,"Lin%d);\n\t",J);
       for(g=1;g<=J;g++)
        fprintf(file,"Us%d<=Uin%d;\n\t",g,g);
       for(g=1;g<=J;g++)
        fprintf(file,"L%d<=Lin%d;\n\t",g,g);
       fprintf(file,"\nend;\n");
       fclose(file);
       printf("Fichier decofrontal.vhdl cree\n");
      }
    return 0;
      }

////////////////////////////////////////////////////////////////////////
//   Fonction devant générer code pour le décodeur intermédiaire     //
////////////////////////////////////////////////////////////////////////

int decointerm (int J,int q,int Maxi,char *c)
 {
  FILE *file;
  int i,t; int g;
  char pipe;
  file=fopen("decointerm.vhdl","w");
  if(file==NULL)
   printf("Probleme pour creer le fichier decointerm.vhdl\n");
```

```
else
 {
  entete(J,q,file);
  pipe=toupper(*c);
  fprintf(file,"Library ieee;\nuse ieee.std_logic_1164.all;\n");
  fprintf(file,"use work.ma_library.all;\nuse work.composants.all;");
  fprintf(file,"entity deco_interm is\n\tport (\n\t\t");
  if (pipe=='P')
   fprintf(file,"clk,rst,InOK : in std_logic;\n\t\t");
  else
   fprintf(file,"clk,InOK : in std_logic;\n\t\t");
  for (i=1;i<=J;i++)
   fprintf (file,"Ue%d : in echantillon;\n\t\t",i);
  for (i=1;i<=J;i++)
   fprintf (file,"Pe%d : in echantillon;\n\t\t",i);
  for (i=1;i<=J;i++)
   fprintf (file,"Le%d : in echantillon;\n\t\t",i);
  for (i=1;i<=J;i++)
   fprintf (file,"Us%d : out echantillon;\n\t\t",i);
  for (i=1;i<=J;i++)
   fprintf (file,"Ps%d : out echantillon;\n\t\t",i);
  for (i=1;i<J;i++)
   fprintf (file,"L%d : out echantillon;\n\t\t",i);
  fprintf (file,"L%d : out echantillon;\n\t\t",J);
  fprintf(file,"OutOK : out std_logic);\nend decodeur_interm;\n\n");
  fprintf (file,"Architecture Arch_deco_interm of deco_interm is ");
  /*declaration des signaux*/
  for (i=1;i<=J;i++)
    for(g=1;g<=J;g++)
     fprintf(file," signal Ps%d%d: echantillon;\n ",i,g);
  for(g=1;g<=J;g++)
   fprintf(file," signal Uin%d: echantillon;\n ",g);
  for(g=1;g<=J*(J+1);g++)
   fprintf(file," signal addin%d: echantillon;\n ",g);
  for(g=1;g<=J;g++)
   fprintf(file," signal addout%d: std_logic_vector(q+%d downto 0);\n
          ",g,depass(J)-1);
  for(g=1;g<=J;g++)
   fprintf(file," signal satin%d: std_logic_vector(q+%d downto 0);\n
          ",g,depass(J)-1);
  for(g=1;g<=J;g++)
   fprintf(file," signal mux%d: echantillon;\n ",g);
  for(g=1;g<=J;g++)
   fprintf(file," signal Lin%d: echantillon;\n",g,depass(J)-1);
  if (pipe=='P')
   {
    for(g=1;g<=J*(J+1);g++)
     fprintf(file," signal paddin%d: echantillon;\n ",g);
    for(g=1;g<=J;g++)
     fprintf(file," signal paddout%d: std_logic_vector(q+%d downto
          0);\n ",g,depass(J)-1);
    for(g=1;g<=J;g++)
```

```
   fprintf(file," signal psatin%d: std_logic_vector(q+%d downto
        0);\n ",g,depass(J)-1);
   }
fprintf(file,"begin\n");
fprintf(file,"inst_P_fifo : fifo port map (clk,");
for(g=1;g<=J;g++)
 fprintf(file,"Pe%d,",g);
for(g=1;g<J;g++)
 fprintf(file,"Ps%d,",g);
fprintf(file,"Ps%d);\n",J);
fprintf(file,"inst_U_fifo : fifo port map (clk,");
for(g=1;g<=J;g++)
 fprintf(file,"Ue%d,",g);
for(g=1;g<J;g++)
 fprintf(file,"Us%d,",g);
fprintf(file,"Ps%d);\n",J);
fprintf(file,"instfifoULP : fifoULP port map (clk,");
for(g=1;g<=J;g++)
 fprintf(file,"Le%d,",g);
for(g=1;g<=J;g++)
 fprintf(file,"Pe%d,",g);
for(g=1;g<=J;g++)
 fprintf(file,"Lin%d,",g);
for(g=1;g<=J;g++)
 fprintf(file,"open,");
for (i=1;i<=J;i++)
 for(g=1;g<=J;g++)
  {
   if (i==J & g==J)
     fprintf(file,"Ps%d%d);\n",J,J);
   else
     fprintf(file,"Ps%d%d,",i,g);
  }
fprintf(file,"instC2Input : C2Input port map (");
for(g=1;g<=J;g++)
 fprintf(file,"Uin%d,",g);
for (i=1;i<=J;i++)
for(g=1;g<=J;g++)
 fprintf(file,"Ps%d%d,",i,g);
for(g=1;g<J*(J+1);g++)
 fprintf(file,"addin%d,",g);
fprintf(file,"addin%d);\n\t",J*(J+1));
t=1;
if (pipe=='P')
 {
  for(g=1;g<=J*(J+1);g++)
   {
    fprintf(file," intpipes%d:pipe port map (clk,rst,addin%d,
              paddin%d);\n ",t,g,g);
    t++;
   }
  }
if (pipe=='P')
```

```
 {
  fprintf(file,"instsommateur : sommateur_J%d port map (clk,",J);
  for(g=1;g<=J*(J+1);
   fprintf(file,"paddin%d,",g);
  for(g=1;g<J;g++)
   fprintf(file,"addout%d,",g);
  fprintf(file,"addout%d);\n\t",J);
 }
else
 {
  fprintf(file,"instsommateur : sommateur_J%d port map (",J);
  for(g=1;g<=J*(J+1);g++)
   fprintf(file,"addin%d,",g);
  for(g=1;g<J;g++)
   fprintf(file,"addout%d,",g);
  fprintf(file,"addout%d);\n\t",J);
  }
if (pipe=='P')
  {
   for(g=1;g<=J;g++)
    {
     fprintf(file," intpipes%d:pipe generic map(q+%d) port map
              (clk,rst,addout%d,paddout%d);\n ",depass(J)-1,t,g,g);
     t++;
     }
  }
if (pipe=='P')
  {
   fprintf(file,"instC2Output : C2Output port map (");
   for(g=1;g<=J;g++)
    fprintf(file,"paddout%d,",g);
   for(g=1;g<J;g++)
    fprintf(file,"satin%d,",g);
   fprintf(file,"satin%d);\n\t",J);
   }
else
   {
    fprintf(file,"instC2Output : C2Output port map (");
    for(g=1;g<=J;g++)
     fprintf(file,"addout%d,",g);
    for(g=1;g<J;g++)
     fprintf(file,"satin%d,",g);
    fprintf(file,"satin%d);\n\t",J);
   }
if (pipe=='P')
   {
    for(g=1;g<=J;g++)
      {
       fprintf(file," intpipes%d:pipe generic map(q+%d) port map
              (clk,rst,satin%d,psatin%d);\n ",depass(J)-1,t,g,g);
       t++;
      }
     }
```

```
        if (pipe=='P')
         {
          fprintf(file,"instsaturer : saturer port map (");
          for(g=1;g<=J;g++)
           fprintf(file,"psatin%d,",g);
          for(g=1;g<J;g++)
           fprintf(file,"mux%d,",g);
          fprintf(file,"mux%d);\n\t",J);
         }
       else
         {
          fprintf(file,"instsaturer : saturer port map (");
          for(g=1;g<=J;g++)
           fprintf(file,"satin%d,",g);
          for(g=1;g<J;g++)
           fprintf(file,"mux%d,",g);
          fprintf(file,"mux%d);\n\t",J);
          }
        fprintf(file,"instmux : activ2 port map (clk,InOK,OutOK,");
        for(g=1;g<=J;g++)
         fprintf(file,"mux%d,",g);
        for(g=1;g<J;g++)
         fprintf(file,"Lin%d,",g);
        fprintf(file,"Lin%d);\n\t",J);
        for(g=1;g<=J;g++)
         fprintf(file,"Us%d<=Uin%d;\n\t",g,g);
        for(g=1;g<=J;g++)
         fprintf(file,"L%d<=Lin%d;\n\t",g,g);
        fprintf(file,"\nend;\n");
        fclose(file);
        printf("Fichier decocentral.vhdl cree\n");
       }
  return 0;
}

////////////////////////////////////////////////////////////////////////
// Fonction devant generer le code VHDL pour le decodeur Terminal     //
////////////////////////////////////////////////////////////////////////

int decoterminal (int J,int q,int Maxi,char *c)
 {
  FILE *file;
  int i,t; int g;
  char pipe;
  file=fopen("decoterminal.vhdl","w");
  if(file==NULL)
   printf("Probleme pour creer le fichier decoterminal.vhdl\n");
  else
   {
    entete(J,q,file);
    pipe=toupper(*c);
    fprintf(file,"Library ieee;\nuse ieee.std_logic_1164.all;\n");
```

```
  fprintf(file,"use work.ma_library.all;\nuse
          work.composants.all;\n\n");
  fprintf(file,"entity decodeur_terminal is\n\tport (\n\t\t");
  if (pipe=='P')
   fprintf(file,"clk,rst,InOK : in std_logic;\n\t\t");
  else
   fprintf(file,"clk,InOK : in std_logic;\n\t\t");
  for (i=1;i<=J;i++)
   fprintf (file,"Ue%d : in echantillon;\n\t\t",i);
  for (i=1;i<=J;i++)
   fprintf (file,"Pe%d : in echantillon;\n\t\t",i);
  for (i=1;i<=J;i++)
   fprintf (file,"Le%d : in echantillon;\n\t\t",i);
  for (i=1;i<J;i++)
   fprintf (file,"L%d : out echantillon;\n\t\t",i);
  fprintf (file,"L%d : out echantillon;\n\t\t",J);
  fprintf(file,"OutOK : out std_logic);\nend decodeur_central;\n\n");
  fprintf (file,"Architecture Arch_decodeur_terminal of
          decodeur_terminal is\n\n");
 /*declaration des signaux*/
 for (i=1;i<=J;i++)
  for(g=1;g<=J;g++)
   fprintf(file," signal Ps%d%d: echantillon;\n ",i,g);
 for(g=1;g<=J;g++)
  fprintf(file," signal Uin%d: echantillon;\n ",g);
 for(g=1;g<=J*(J+1);g++)
  fprintf(file," signal addin%d: echantillon;\n ",g);
 for(g=1;g<=J;g++)
  fprintf(file," signal addout%d: std_logic_vector(q+%d downto 0);\n
          ",g,depass(J)-1);
for(g=1;g<=J;g++)
 fprintf(file," signal satin%d: std_logic_vector(q+%d downto 0);\n
          ",g,depass(J)-1);
for(g=1;g<=J;g++)
 fprintf(file," signal mux%d: echantillon;\n ",g);
for(g=1;g<=J;g++)
 fprintf(file," signal Lin%d: echantillon;\n",g,depass(J)-1);
if (pipe=='P')
 {
  for(g=1;g<=J*(J+1);g++)
   fprintf(file," signal paddin%d: echantillon;\n ",g);
  for(g=1;g<=J;g++)
   fprintf(file," signal paddout%d: std_logic_vector(q+%d downto
          0);\n ",g,depass(J)-1);
  for(g=1;g<=J;g++)
   fprintf(file," signal psatin%d: std_logic_vector(q+%d downto 0);\n
          ",g,depass(J)-1);
  }
fprintf(file,"begin\n");
fprintf(file,"inst_P_fifo : fifo port map (clk,");
for(g=1;g<=J;g++)
 fprintf(file,"Ue%d,",g);
for(g=1;g<J;g++)
```

```
 fprintf(file,"Us%d,",g);
fprintf(file,"Ps%d);\n",J);
fprintf(file,"instfifoULP : fifoULP port map (clk,");
for(g=1;g<=J;g++)
 fprintf(file,"Le%d,",g);
for(g=1;g<=J;g++)
 fprintf(file,"Pe%d,",g);
for(g=1;g<=J;g++)
 fprintf(file,"Lin%d,",g);
for(g=1;g<=J;g++)
 fprintf(file,"open,");
for (i=1;i<=J;i++)
 for(g=1;g<=J;g++)
  {
   if (i==J & g==J)
    fprintf(file,"Ps%d%d);\n",J,J);
   else
    fprintf(file,"Ps%d%d,",i,g);
  }
fprintf(file,"instC2Input : C2Input port map (");
for(g=1;g<=J;g++)
 fprintf(file,"Uin%d,",g);
for (i=1;i<=J;i++)
 for(g=1;g<=J;g++)
  fprintf(file,"Ps%d%d,",i,g);
for(g=1;g<J*(J+1);g++)
  fprintf(file,"addin%d,",g);
fprintf(file,"addin%d);\n\t",J*(J+1));
t=1;
if (pipe=='P')
 {
  for(g=1;g<=J*(J+1);g++)
   {
    fprintf(file," intpipes%d:pipe port map (clk,rst,addin%d,
          paddin%d);\n ",t,g,g);
    t++;
   }
  }
if (pipe=='P')
  {
   fprintf(file,"instsommateur : sommateur_J%d port map (clk,",J);
   for(g=1;g<=J*(J+1);g++)
    fprintf(file,"paddin%d,",g);
   for(g=1;g<J;g++)
    fprintf(file,"addout%d,",g);
   fprintf(file,"addout%d);\n\t",J);
  }
 else
  {
   fprintf(file,"instsommateur : sommateur_J%d port map (",J);
   for(g=1;g<=J*(J+1);g++)
     fprintf(file,"addin%d,",g);
    for(g=1;g<J;g++)
```

```
   fprintf(file,"addout%d,",g);
  fprintf(file,"addout%d);\n\t",J);
  }
 if (pipe=='P')
  {
   for(g=1;g<=J;g++)
    {
     fprintf(file," intpipes%d:pipe generic map(q+%d) port map
              (clk,rst,addout%d,paddout%d);\n ",depass(J)-1,t,g,g);
     t++;
    }
  }
 if (pipe=='P')
  {
   fprintf(file,"instC2Output : C2Output port map (");
   for(g=1;g<=J;g++)
    fprintf(file,"paddout%d,",g);
   for(g=1;g<J;g++)
    fprintf(file,"satin%d,",g);
   fprintf(file,"satin%d);\n\t",J);
  }
 else
  {
   fprintf(file,"instC2Output : C2Output port map (");
   for(g=1;g<=J;g++)
    fprintf(file,"addout%d,",g);
   for(g=1;g<J;g++)
    fprintf(file,"satin%d,",g);
   fprintf(file,"satin%d);\n\t",J);
  }
 if (pipe=='P')
  {
   for(g=1;g<=J;g++)
    {
     fprintf(file," intpipes%d:pipe generic map(q+%d) port map
              (clk,rst,satin%d,psatin%d);\n ",depass(J)-1,t,g,g);
     t++;
    }
  }
if (pipe=='P')
  {
   fprintf(file,"instsaturer : saturer port map (");
   for(g=1;g<=J;g++)
    fprintf(file,"psatin%d,",g);
   for(g=1;g<J;g++)
    fprintf(file,"mux%d,",g);
   fprintf(file,"mux%d);\n\t",J);
   }
else
   {
    fprintf(file,"instsaturer : saturer port map (");
    for(g=1;g<=J;g++)
      fprintf(file,"satin%d,",g);
```

```
          for(g=1;g<J;g++)
           fprintf(file,"mux%d,",g);
          fprintf(file,"mux%d);\n\t",J);
         }
     fprintf(file,"instmux : activ2 port map (clk,InOK,OutOK,");
     for(g=1;g<=J;g++)
      fprintf(file,"mux%d,",g);
     for(g=1;g<J;g++)
      fprintf(file,"Lin%d,",g);
     fprintf(file,"Lin%d);\n\t",J);
     for(g=1;g<=J;g++)
      fprintf(file,"L%d<=Lin%d;\n\t",g,g);
     fprintf(file,"\nend;\n");
     fclose(file);
     printf("Fichier decoterminal.vhdl cree\n");
     }
  return 0;
}

////////////////////////////////////////////////////////////////////////
//     Fonction devant generer le code VHDL pour le package           //
//           des blocs de decodage                                    //
////////////////////////////////////////////////////////////////////////

int blocs (int J,int q,char *c)
 {
  FILE *file;
  int i;
  char pipe;
  pipe=toupper(*c);
  if (file=fopen("blocs_lb.vhdl","w"))
   {
    printf("Fichier blocs_lb.vhdl cree\n");
    entete(J,q,file);
    fprintf(file,"Library ieee;\nuse ieee.std_logic_1164.all;\nuse
            ieee.std_logic_arith.all;\n");
    fprintf(file,"use ieee.std_logic_unsigned.all;\nuse
            work.ma_library.all;\n\n package blocs_lb is\n\t");
    fprintf(file,"component decodeur_frontal is\n\tport (\n\t\t");
    if (pipe=='P')
     fprintf(file,"clk,rst,InOK : in std_logic;\n\t\t");
    else
     fprintf(file,"clk,InOK : in std_logic;\n\t\t");
    for (i=1;i<=J;i++)
      fprintf (file,"Ue%d : in echantillon;\n\t\t",i);
    for (i=1;i<=J;i++)
      fprintf (file,"Pe%d : in echantillon;\n\t\t",i);
    for (i=1;i<=J;i++)
      fprintf (file,"Us%d : out echantillon;\n\t\t",i);
    for (i=1;i<=J;i++)
      fprintf (file,"Ps%d : out echantillon;\n\t\t",i);
      for (i=1;i<J;i++)
       fprintf (file,"L%d : out echantillon;\n\t\t",i);
```

```
      fprintf (file,"L%d : out echantillon;\n\t\t",J);
      fprintf(file,"OutOK : out std_logic);\nend component;\n\n");
      fprintf(file,"component decodeur_interm is\n\tport (\n\t\t");
      if (pipe=='P')
       fprintf(file,"clk,rst,InOK : in std_logic;\n\t\t");
      else
       fprintf(file,"clk,InOK : in std_logic;\n\t\t");
      for (i=1;i<=J;i++)
       fprintf (file,"Ue%d : in echantillon;\n\t\t",i);
      for (i=1;i<=J;i++)
       fprintf (file,"Pe%d : in echantillon;\n\t\t",i);
      for (i=1;i<=J;i++)
       fprintf (file,"Le%d : in echantillon;\n\t\t",i);
      for (i=1;i<=J;i++)
       fprintf (file,"Us%d : out echantillon;\n\t\t",i);
      for (i=1;i<=J;i++)
       fprintf (file,"Ps%d : out echantillon;\n\t\t",i);
      for (i=1;i<J;i++)
       fprintf (file,"L%d : out echantillon;\n\t\t",i);
      fprintf (file,"L%d : out echantillon;\n\t\t",J);
      fprintf(file,"OutOK : out std_logic);\nend component;\n\n");
      fprintf(file,"component decodeur_terminal is\n\tport (\n\t\t");
      if (pipe=='P')
       fprintf(file,"clk,rst,InOK : in std_logic;\n\t\t");
      else
       fprintf(file,"clk,InOK : in std_logic;\n\t\t");
      for (i=1;i<=J;i++)
       fprintf (file,"Ue%d : in echantillon;\n\t\t",i);
      for (i=1;i<=J;i++)
       fprintf (file,"Pe%d : in echantillon;\n\t\t",i);
      for (i=1;i<=J;i++)
       fprintf (file,"Le%d : in echantillon;\n\t\t",i);
      for (i=1;i<J;i++)
       fprintf (file,"L%d : out echantillon;\n\t\t",i);
      fprintf (file,"L%d : out echantillon;\n\t\t",J);
      fprintf(file,"OutOK : out std_logic);\nend component;\n\n");
      fprintf(file,"end blocs_lb;\n\t");
      }
     else
       {
       printf("Probleme pour creer le fichier blocs_lb.vhdl!!!!!\n");
       return 1;
       }
return 0;
}
```

```
////////////////////////////////////////////////////////////////////
// Fonction devant generer le code VHDL pour le decodeur itteratif   //
```

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

int decodeur (int J,int q,char *c)
 {
  FILE *file;
  int i;
  char pipe;
  pipe=toupper(*c);
  if (file=fopen("decodeur.vhdl","w"))
   {
    printf("Fichier decodeur.vhdl cree\n");
    entete(J,q,file);
    fprintf(file,"Library ieee;\nuse ieee.std_logic_1164.all;\nuse
            ieee.std_logic_arith.all;\n");
    fprintf(file,"use ieee.std_logic_unsigned.all;\nuse
            work.ma_library.all;\n\nuse work.blocs_lb.all;\n");
    fprintf(file,"entity decodeur is\n\tport (\n\t\t");
    if (pipe=='P')
     fprintf(file,"clk,rst,InOK : in std_logic;\n\t\t");
    else
     fprintf(file,"clk,InOK : in std_logic;\n\t\t");
    for (i=1;i<=J;i++)
     fprintf (file,"U%d : in echantillon;\n\t\t",i);
    for (i=1;i<=J;i++)
     fprintf (file,"P%d : in echantillon;\n\t\t",i);
    for (i=1;i<J;i++)
     fprintf (file,"L%d : out echantillon;\n\t\t",i);
    fprintf(file,"OutOK : out std_logic);\nend decodeur;\n\n");
    fprintf(file,"architecture arch_decodeur of decodeur is\n");
    fprintf(file,"begin\n\ttest1 : if N>2 generate\n\t\t");
    fprintf(file,"Type vecteurs is array (0 to N-2) of
            echantillon;\n\t\t");
    fprintf(file,"Type vecteur is array (0 to N-2) of
            tsd_logic;\n\t\t");
    for (i=1;i<=J;i++)
     fprintf(file,"signal Us%d : vecteurs;\n\t\t",i);
    for (i=1;i<=J;i++)
     fprintf(file,"signal Ps%d : vecteurs;\n\t\t",i);
    for (i=1;i<=J;i++)
     fprintf(file,"signal Ls%d : vecteurs;\n\t\t",i);
    fprintf(file,"signal OutOks : vecteur;\n\t\t");
    fprintf(file,"begin\n\t\t\t");
    fprintf(file,"instdecofront:decodeur_frontal port map(\n\t\t\t\t\t");
    if (pipe=='P')
     fprintf(file,"clk,rst,InOK,\n\t\t\t\t\t");
    else
     fprintf(file,"clk,InOK,\n\t\t\t\t\t");
    for (i=1;i<=J;i++)
     fprintf (file,"U%d,",i);
    fprintf (file,"\n\t\t\t\t");
      for (i=1;i<=J;i++)
       fprintf (file,"P%d,",i);
```

```
fprintf (file,"\n\t\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Us%d(0),",i);
fprintf (file,"\n\t\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Ps%d(0),",i);
fprintf (file,"\n\t\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Ls%d(0),",i);
fprintf (file,"\n\t\t\t\t\t");
fprintf(file,"OutOKs(0));\n\n\t\t\t");
fprintf(file,"boucle: for i in 0 to N-3 generate;\n\t\t\t\t\t");
fprintf(file,"instdecocent deco_interm port map(");
if (pipe=='P')
 fprintf(file,"clk,rst,InOK,\n\t\t\t\t\t\t");
else
fprintf(file,"clk,OutOKs(0),\n\t\t\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Us%d(i),",i);
fprintf (file,"\n\t\t\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Ps%d(i),",i);
fprintf (file,"\n\t\t\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Ls%d(i),",i);
fprintf (file,"\n\t\t\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Us%d(i+1),",i);
fprintf (file,"\n\t\t\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Ps%d(i+1),",i);
fprintf (file,"\n\t\t\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Ls%d(i+1),",i);
fprintf(file,"OutOKs(i+1));\n\t\t\t\t");
fprintf(file,"end generate boucle;\n\n\t\t\t\t");
fprintf(file,"instdecoterminal: decodeur_terminal port map(");
if (pipe=='P')
 fprintf(file,"clk,rst,OutOKs(N-2),\n\t\t\t\t\t");
else
 fprintf(file,"clk,OutOKs(N-2),\n\t\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Us%d(N-2),",i);
fprintf (file,"\n\t\t\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Ps%d(N-2),",i);
fprintf (file,"\n\t\t\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Ls%d(N-2),",i);
fprintf (file,"\n\t\t\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"L%d,",i);
fprintf(file,"OutOK);\n");
```

```
fprintf (file,"\n\t\t\t");
fprintf (file,"end generate test1;\n");
fprintf(file,"test2 : if N=2 generate\n\t\t");
fprintf(file,"Type vecteurs is array (0 to 0) of
      echantillon;\n\t\t");
fprintf(file,"Type vecteur is array (0 to 0) of
      tsd_logic;\n\t\t");
for (i=1;i<=J;i++)
 fprintf(file,"signal Us%d : vecteurs;\n\t\t",i);
for (i=1;i<=J;i++)
 fprintf(file,"signal Ps%d : vecteurs;\n\t\t",i);
for (i=1;i<=J;i++)
 fprintf(file,"signal Ls%d : vecteurs;\n\t\t",i);
fprintf(file,"signal OutOks : vecteur;\n\t\t");
fprintf(file,"begin\n\t\t\t");
fprintf(file,"instdecofront:decodeur_frontal port map(");
if (pipe=='P')
 fprintf(file,"clk,rst,InOK,\n\t\t\t\t");
else
 fprintf(file,"clk,InOK,\n\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"U%d,",i);
fprintf (file,"\n\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"P%d,",i);
fprintf (file,"\n\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Us%d(0),",i);
fprintf (file,"\n\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Ps%d(0),",i);
 fprintf (file,"\n\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Ls%d(0),",i);
fprintf (file,"\n\t\t\t\t");
fprintf(file,"OutOKs(0));\n\n\t\t\t");
fprintf(file,"instdecoterminal: decodeur_terminal port map(");
if (pipe=='P')
 fprintf(file,"clk,rst,OutOKs(0),\n\t\t\t\t");
else
 fprintf(file,"clk,OutOKs(0),\n\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Us%d(0),",i);
fprintf (file,"\n\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Ps%d(0),",i);
fprintf (file,"\n\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"Ls%d(0),",i);
fprintf (file,"\n\t\t\t\t");
for (i=1;i<=J;i++)
 fprintf (file,"L%d,",i);
fprintf(file,"OutOK);\n");
```

```
      fprintf (file,"\n\t\t\t");
      fprintf (file,"end generate test2;\n");
      fprintf (file,"end;");
      }
     else
      {
       printf("Probleme pour creer le fichier decodeur.vhdl!!!!!\n");
       return 1;
       }
 return 0;
}


////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
/////// FIN DE LA ZONE DES FONCTIONS, DEBUT DE LA FONCTION MAIN//////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
int main(int argc, char *argv[])
{
////////////////////////////////////////////////////////////////////////
///////Phase de saisie de données et des calculs préliminaires////////
////////////////////////////////////////////////////////////////////////
if (argc!=6)
      {
        printf("Erreur!!! Vous devez entrer le taux de codage J\n");
        printf("suivi du taux de quantification q,puis le chemin\n");
        printf("du fichier text decrivant la matrice du codeur\n");
        printf("exemple : Vhdlgen 3 3 J3.txt p
                  2//J=3,q=3,fichier=J3.txt\n");
        return 1;
       }
else {
     char c[LONG];
     char reponse;
     char *pipe;
/*indice de codage et de quantification*/
     int J,q,iteration;
/*variables de boucles et Maxi=le maximum dans la matrice de codage*/
     int i=0,t=0;
/*Matrice du codeur et des maximums pour chaque ligne*/
       int *tableau,*Maxs,Maxi;
/*pointeur pour les manipulations sur les fichiers*/
     FILE *file;
/*initialisation des variables taux de codage J et d'échantillonage q*/
     J=atoi(argv[1]);
     q=atoi(argv[2]);
     pipe=argv[4];
     iteration=atoi(argv[5]);
/*accés pour la lecture de la matrice du codeur*/
     if ((file=fopen(argv[3],"r"))!=NULL)
        printf("Fichier du code lu avec succes!!!!\n");
     else
```

```
        {
         printf("Echec!!!\nVerifier le nom du fichier du code et son
                existance\n");
        system("Pause");
        return 2;
        }
/*Allocation d'espace mémoire pour la matrice du codeur*/
     tableau=(int *) malloc (J*J*sizeof(int ));
/*Lecture de la matrice*/
     while ((fgets(c,LONG,file))!= NULL)
                    {
                          tableau[i*J+t]=atoi(c);
                          if (t<J-1)
                              t++;
                          else
                              {
                               t=0;
                               i++;
                              }
                    }
     fclose(file);
/*Affichage pour vérification*/
     printf("********RECAPITULATION********\n");
     printf("Le taux de codage= %d/%d\n",J,2*J);
     printf("Le nombre de bits de quantification=%d\n",q);
/*Affichage de la matrice du codec*/
     printf("La matrice du code est:\n\n");
     for (i=0;i<J;i++)
         {
         printf("\n");
         for(t=0;t<J;t++)
         printf("%d\t",tableau[i*J+t]);
         }
     printf("\n");
     Maxs=maxs(J,tableau);
     Maxi=max(J,Maxs);
/*Validation de la matrice pour génération des fichiers VHDL*/
     do
     {
      printf("Generation des fichiers VHDL?[Y,N]\n");
      reponse=getch();
      }
     while(reponse!='Y'&& reponse!='y'&& reponse!='N'&& reponse!='n');
     if (reponse=='Y' || reponse=='y')
     {
     system("cls");
     mesfonctions(J,q,iteration);
     c2input(J,q);
     c2output(J,q);
     decision(J,q);
     saturateur (J,q);
     sommateur(J,q,argv[4]);
     fifo(J,q,Maxi);
```

```
      FIFOULP(J,q,tableau,Maxs,Maxi);
      activ2(J,q,Maxi);
      pipeline(J,q);
      composants(J,q,pipe);
      decofrontal(J,q,Maxi,pipe);
      decocentral(J,q,Maxi,pipe);
      decoterminal(J,q,Maxi,pipe);
      blocs(J,q,pipe);
      decodeur(J,q,pipe);
      return 0;
      }
      else
      {
       printf("--FIN--");
       return 1;
      }
     }
}
```

BIBLIOGRAPHIE

[1] Cardinal, C. (2001). *Décodage à seuil itérative sans entrelacement des codes convolutionnels doublement orthogonaux.* Thèse de Doctorat à l'École Polytechnique de Montréal, Montréal.

[2] Proakis, J.G. (1995). *Digital Communications* (3e éd). McGraw-Hill.

[3] Elias, P. (1955). Coding for Noisy Channels. IRE Conv. Rec, 4:37-46.

[4] Bahai, A.R.S., Saltzberg B.R.(1999). *Multi-carrier digital communications: theory and applications of OFDM.* New York: Kluwer Academic/Plenum.

[5] Shannon, C.E. (1948).*The Mathematical Theory of Communication.* University of Illinois Press, Urbana, IL.

[6] Berrou, C., Glavieux, A., et Thitimajshima, P. (1993). Near Shannon limit error-correcting coding and decoding: Turbo-codes. *Proc. ICC'93, pp. 1064–1070.*

[7] Forney, G.D.Jr. (1966). *Concatenated Codes.* MIT Press, Cambridge, Massachusetts, USA.

[8] Lee, L.H.C. (1997). *Convolutional Coding: Fundamentals and Applications.* Artech House.

[9] Massey, J.L. (1963). *Treshold decoding.* MIT Press, Cambridge, Massachusetts.

[10] Viterbi, A.J. (1967). Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Transactions on Information Processing,* 13:260-269.

[11] Wozencraft, J.M. (1957). Sequential decoding for reliable communications. *Res. Lab. of Electronics, M.I.T. Technical Report, 325.*

[12] Fano, R.M. (1963). A heuristic discussion of probabilistic decoding. *IEEE Trans. Inform. Theory, vol. 9, pp. 64 74.*

[13] Zigangirov, K.Sh. (1969). Some sequential decoding procedures. *Probl. Peredachi Inform., 2, pp. 13-25.*

[14] Jelinek, F. (1969). A fast sequential decoding algorithm using a stack. *IBM J. Res. and Develop., vol.13, pp.675-685.*

[15] Bahl, L., Cocke, J., Jelinek, F., et Raviv, J. (1974). Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Trans. Inform. Theory, pp. 284-287.*

[16] McAdam, P.L., Welch, L., et Weber, C. (1972). MAP Bit Decoding of Convolutional Codes. *IEEE Intl. Symp. on Information Theory, Asilomar, CA.*

[17] Chang, R.W., Hancock, J.C. (1966). On Receiver Structures for Channels Having Memory. *IEEE Transactions on Information Theory, vol. IT-12, pp. 463–468.*

[18] Robertson, P., Hoeher, P. (1997). Optimal and Sub-Optimal Maximum A Posteriori Algorithms Suitable for Turbo Decoding. *European Transactions on Telecommun., pp. 119-125.*

[19] Lin, S., Costello, D.J.JR. (1983). *Error-control coding: fundamentals and applications*. Englewood Cliffs, NJ: Prentice-Hall.

[20] Robertson, P. (1994). Illuminating the structure of code and decoder of parallel concatenated recursive systematic (Turbo) codes. *IEEE Globecom, pp. 1298-1303.*

[21] Barbulescu, S.A. (1996). *Iterative decoding of turbo codes and other concatenated codes.* Ph.D. Dissertation, School of Electronics Engineering, Faculty of Engineering, University of South Australia.

[22] Wu, W.W. (1975). New convolutionnal codes- Part I. *IEEE Transactions on Communications, Vol. 23, No. 9, pp. 942-956.*

[23] Wu, W.W. (1976). New convolutionnal codes- Part II. *IEEE Transactions on Communications, Vol. 24, No. 9, pp. 946-955.*

[24] Tanaka, H., Furusawa, K., Kaneku, S. (1980). A novel approach to soft-decision decoding of threshold decodable codes. *IEEE Tansactions on Information Theory, Vol. IT-26, No. 2, pp. 244-246.*

[25] Lavoie, P., Haccoun, D., Savaria, Y. (1991). A New VLSI architecture for fast soft-decision threshold decoders. *IEEE Transactions on Communications, Vol. 39, No. 2, pp. 200-207.*

[26] Gagnon, F., Batani, N., Dam, T.Q. (1995). Simplified Designs for AAPP soft decision threshold decoders. *IEEE Transactions on Communications, Vol. 43, No. 2/3/4, pp. 743-750.*

[27] Cardinal, C., Haccoun, D., Gagnon, F. (2003). Iterative threshold decoding without interleaving for concolutionnal self-doubly orthogonal codes. *IEEE Transactions on Communications, Vol.51, No 8 Aug. 2003, pp 1274-1282.*

[28] Ashok, K.S. (1998). *Programmable Logic Handbook: PLDs, CPLDs and FPGAs.* McGraw-Hill Professional.

[29] Xilinx Inc (1999). *XC4000E and XC4000X Series Field Programmable Gate Arrays, Product Specification (Version 1.6).*

[30] Xilinx Inc (2002). *Virtex™ 2.5 V Field Programmable Gate Arrays, Product Specification.*

[31] Xilinx Inc (2002). *Virtex™-E 1.8 V Field Programmable Gate Arrays, Production Product Specification.*

[32] Xilinx Inc (2002). *Virtex™-E 1.8 V Extended Memory Field Programmable Gate Arrays, Production Product Specification.*

[33] Xilinx Inc (2003). *Virtex™-II Platform FPGAs: Complete Data Sheet, Product Specification.*

[34] Parnell, K., Mehta, N. (2003). *Programmable Logic Design Quick Start Book.* Xilinx Inc.

[35] Douglas, J.S. (1996). VHDL & Verilog Compared & Contrasted - Plus Modeled Example Written in VHDL, Verilog and C. *33rd Conference on Design Automation 1996, Las Vegas, Nevada, USA,DAC: 771-776.*

[36] Hazime, B. (1998). *Conception d'un décodeur à seuil programmable pour codes convolutionnels.* Thése de M.Ing à l'Ecole de Technologie Supérieure deMontréal.

[37] Gagnon, F., Batani, N. et Dam, T. (1993). Simplified implementation of a soft decision threshold decoder. Global Telecommunications Conference, IEEE in Houston.

[38] Xilinx Inc. (2000). *Design Tips for HDL Implementation of Arithmetic Functions.* Xilinx Design Notes.

[39] Krukowski, A., Kale, I. (1999). Simulink/Matlab-to-VHDL Route for Full-Custom/FPGA Rapid Prototyping of DSP Algorithms. *Matlab DSP Conference, Tampere, Finland.*

[40] Xilinx Inc. System Generator for DSP. [en ligne] http://www.xilinx.com/ (consulté le 23 Juillet 2004).

[41] Lim, S., Miller, A. (2001). *LFSRs as Functional Blocks in Wireless Applications.* Xilinx application notes XAPP220 (v1.1).

[42] Miller, A., Gulotta, M. (2001). *PN Generators Using the SRL Macro*. Xilinx application notes XAPP211 (v1.1).

[43] Xilinx Inc. (2003). *Using Look-Up Tables as Shift Registers (SRL16) in Spartan-3.* Xilinx application notes XAPP465 (v1.0).

[44] The Mathworks Inc. *Cosimulate and verify VHDL and Verilog using ModelSim,* [en ligne] http://www.mathworks.com/products/modelsim/ (consulté le 23 Juillet 2004).

[45] Xilinx Inc. (1996). *Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators.* Xilinx application notes XAPP052 (v1.1).