RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

École Nationale Polytechnique



Département d'Électronique

Projet de Fin d'Études

présenté pour obtenir le titre d'INGÉNIEUR d'ÉTAT en Électronique

Implémentation sur Circuit Reconfigurable d'un décodeur de Viterbi à Décision Ferme

Massiva Yamina ZOUAOUI
Abderrezak HADJI

Soutenu publiquement le 25/06/2012 devant un jury composé de :

Président	M. C.Larbes	Professeur	(ENP)
Examinateur	M. S.Ait Cheikh	Maitre de Conférence A	(ENP)
Encadreur	M. M.Taghi	Chargé de Cours	(ENP)
Co-encadreur	M. L.Abdelouel	Chargé de Cours	(ENP)

LDCCP



Projet de Fin d'Études réalisé au

Laboratoire des Dispositifs de Communications
et de Conversions Photovoltaïques
Équipe 03 : Commande des dispositifs.
École Nationale Polytechnique
10 avenue Hacène Badi – El Harrach
BP182–16200 Alger
Algérie

Tél : (+213) 21 52 53 01/03
Fax : (+213) 21 52 29 73
Web : http ://www.enp.edu.dz/

ملخص

يتمحور هذا العمل حول محاكاة ثم تثبيت خوارزميه فكّ الرموز فيتربي (Viterbi) ذو القرار الصعب على بطاقة FPGA. المفكك يستعمل داخل أجهزة استقبال أنظمة الاتصالات الرقمية لكشف و تصحيح الأخطاء الواردة عندما تكون المعلومة مشفرة بمشفر ملفف (Codeur Convolutionnel) قمنا بدراسة مرمزة التفافيّة .

هذه الخوارزمية عبارة عن مفكك قناة و يستعمل لكشف و تصحيح الأخطاء الواردة. افترضنا مشفر ملفف يحتوي قيد طوله $L = 4$ و مردوده $R = {}^{1}/_{2}$ معرف بكثري حدود $g_2 = 13_8$ و $g_2 = 15_8$ أنجزنا تجارب على MATLAB لهذا الطول و لأي طول، كذلك كتابة الخوارزمية بلغة VHDL حيث بينا منه النتائج.

الكلمات المفتاحية : فيتربي، الاتصالات الرقمية، **مشفر ملفف.**

Résumé

Ce travail s'articule autour de la simulation et l'implémentation sur FPGA d'un décodeur de Viterbi à décision ferme.
Il s'agit d'un décodeur utilisé dans les récepteurs des systèmes de communication numériques lorsque l'information est codée par un code convolutionnel. Cet algorithme concerne le décodage de canal et sert donc à la détection et à la correction des erreurs au sein de l'information reçue. Nous avons supposé un codeur convolutionnel de longueur de contrainte $L = 4$ et de rendement $R = {}^{1}/_{2}$ défini par les polynôme générateurs $g_1 = 13_8$ et $g_2 = 15_8$. Nous avons effectué des simulations sur MATLAB pour cette même longueur mais aussi pour une longueur quelconque. Nous avons aussi réalisé un programme VHDL dont nous avons exposé l'avancement et les résultats.

Mots clé : Viterbi, Code convolutionels, Codage de canal.

Abstract

Our project aims to simulate and implement, on FPGA circuit, a hard decision Viterbi decoder. It is used to decode received information in digital communication systems when the information is coded by a convolutionnal code. This algorithm has a canal decoding purpose and is therefore used to detect and correct errors within the received information. A convolutionnal coder of constraint length $\mathrm{L} = 4$ has been studied. MATLAB simulations were performed for this same length, but also for nondescript length. At last, a VHDL program has been written, which we have outlined the progress and results.

Keys words: Viterbi, Convolutionnal encoder, Channel coding.

Remerciements

Nous remercions nos parents qui nous ont beaucoup encouragés le long de ce projet. Et à l'issu de nos études faites au sein de l'École Nationale Polytechnique nous voudrions rendre un hommage tout particulier à : Nos promoteurs M. M.Taghi et M. L.Abdelouel qui nous ont honorés en acceptant de nous encadrer. Tous les enseignants du département d'Électronique de l'École Nationale Polytechnique. Tous les enseignants du département des Sciences Fondamentales. Nous remercions tous ceux qui ont participé de prés ou de loin à l'aboutissement de nos études.

Dédicace

A celle qui ma accompagnée et défendue depuis ma plus tendre enfance et que j'aime plus que tout. A ma *sœur*.

A celle qui a consacré tout son temps, son énergie et sa patience à me donner une bonne éducation. A *ma mère*.

A celui qui ma indiqué le chemin à suivre en me rappelant que le travail et la volonté font de l'être humain ce qu'il est. A mon *père*.

Je dédie aussi ce mémoire à tous mes amis et collègues de travail qui m'ont énormément aidés, je pense tout particulièrement à Celia et Rabah.

Merci.

citation :

« L'erreur n'annule pas la valeur de l'effort accompli. »

de Proverbe africain

Massiva Yamina ZOUAOUI

Dédicace

À mes chers Parents,

Vous avez guidé mes premiers pas et veillé sur moi depuis ma plus tendre enfance, tout en m'indiquant le savoir et les bonnes manières.

Si aujourd'hui je suis devenu ce que je suis, c'est bien grâce à vous. C'est pourquoi je vous dédie humblement ce travail qui n'est autre que le fruit de vos sacrifices.

Maman, Papa je vous dois tous !

À la mémoire

de mes Grands Parents paternels,
de ma Tante et de mon Oncle paternels ;

À mes Grands Parents maternels ;

À mes Tantes et mes Oncles maternels ;

À mes Cousines et mes Cousins paternels et maternels ;

À ma Sœur,

À mes deux Frères ;

À mes amis en tout particulièrement Inès, Hayet, Marwa, Amir, Ridha, Hichem, Hakim, Abdellatif, Noredine, Houssem, Khaled, Zaki...

« Donnez-moi un point fixe et un levier, et je soulèverai la terre ».

Archimède

Abderrezak HADJI

Table des matières

Résumé i

Remerciements iii

Dédicace v

Dédicace vii

Table des matières ix

Table des figures xiii

Liste des tableaux xv

Introduction 1

1 Codeurs Convolutionnels 3
1.1 Introduction . . . 3
1.2 Chaîne de transmission numérique et théorie de l'information . . . 3
1.2.1 Chaîne de transmission numérique . . . 3
1.2.1.1 Canal . . . 4
1.2.1.2 Codeur de source . . . 4
1.2.1.3 Codeur de canal . . . 5
1.2.1.4 Modulation . . . 5
1.2.1.5 Bruits . . . 7
1.2.2 Théorie de l'information . . . 8
1.2.2.1 Entropie . . . 8
1.2.2.2 Capacité de canal . . . 8
1.2.2.3 Canal Symétrique Binaire . . . 10
1.2.2.4 Canal AWGN . . . 10
1.3 Codage de canal et codes correcteurs d'erreur . . . 11
1.3.1 Principes fondamentaux des codes en blocs . . . 12
1.3.1.1 Paramètres du code . . . 12

1.3.1.2 Décodage au maximum de vraisemblance 13
1.3.1.3 Détection et correction d'erreur 13
1.4 Codeurs convolutionnels 14
1.4.1 Fonctionnement 15
1.4.2 Caractéristiques du codeur convolutionnel utilisé 16
1.4.3 Diagramme d'état et ses représentations 18
1.5 Applications des codes convolutionnels 20

2 Décodeur de Viterbi 23
2.1 Introduction 23
2.2 Algorithme de VITERBI 23
2.2.1 Principe de l'algorithme 24
2.2.2 Exemple de décodage 27
2.3 Schéma fonctionnel du décodeur 31
2.3.1 Unité de Calcul des Métriques de Branches 32
2.3.2 Unité de calcul des Métriques de Chemin 33
2.3.3 Unité de Décision 34

3 Architecture proposée 37
3.1 Introduction 37
3.2 Schéma proposé du décodeur 37
3.3 Unité de Calcul des Métriques 38
3.3.1 ACS à deux chemins 38
3.3.2 Papillons 38
3.3.3 Registre d'état 40
3.3.4 Architecture de la MCU 43
3.4 Unité de décision 46
3.4.1 Organisation de la mémoire du Trace Back 46
3.4.2 Fonction de recherche du minimum 48
3.4.3 Module *Trace Back* 48
3.4.4 Architecture de DU 48
3.5 Circuit de contrôle 51
3.5.1 Synchronisation 51
3.5.2 Chemin de données 52
3.5.3 Le circuit de contrôle du Trace Back 52

4 Simulations, tests et résultats 53
4.1 Introduction 53
4.2 Présentation de la carte FPGA 53
4.3 Simulation sous MATLAB® 54
4.3.1 Simulation du décodeur choisi 55
4.3.1.1 Décodage Trace Back 55

4.3.1.2 Décodage Register Exchange . . . 58
4.3.2 Simulation Cassini . . . 60
4.3.3 Simulation CDMA 2000 . . . 62
4.3.4 Simulation UWB (802.15) . . . 64
4.4 Simulation et implémentation sous ISE de Xilinx . . . 66
4.4.1 Codeur Convolutionnel . . . 66
4.4.2 Papillons . . . 67
4.4.3 RAM des métriques et des décisions . . . 72
4.4.4 Détecteur de minimum . . . 74
4.4.5 Trace Back . . . 74
4.4.5.1 Shift Registre . . . 75
4.4.5.2 Pile LIFO . . . 75
4.4.6 Circuit de contrôle . . . 76

Conclusion et perspectives **77**

Bibliographie **79**

A Programmes MATLAB® **81**
A.1 Décodeur $L = 4$, $R = 1/2$, $g_1 = 15_8$ et $g_2 = 13_8$ . . . 81
A.1.1 Trace Back . . . 81
A.1.1.1 Butterfly1.m . . . 81
A.1.1.2 Butterfly2.m . . . 83
A.1.1.3 Butterfly3.m . . . 85
A.1.1.4 Butterfly4.m . . . 87
A.1.1.5 Viterbi_TB.m . . . 88
A.1.2 Register Exchange . . . 92
A.1.2.1 Butterfly1.m . . . 92
A.1.2.2 Butterfly2.m . . . 93
A.1.2.3 Butterfly3.m . . . 93
A.1.2.4 Butterfly4.m . . . 93
A.1.2.5 Viterbi_RE.m . . . 93
A.2 Décodeur L, $R = 1/n$ et G . . . 98
A.2.1 Trace Back . . . 98
A.2.1.1 Treillis.m . . . 98
A.2.1.2 ConvolutionalEncoder.m . . . 99
A.2.1.3 Butterfly.m . . . 100
A.2.1.4 Viterbi_TB.m . . . 102

B Programmes VHDL **107**
B.1 Coder.vhd 107
B.2 Papillons 108
B.2.1 BF000_001.vhd 108
B.2.2 BF010_011.vhd 110
B.2.3 BF100_101.vhd 112
B.2.4 BF110_111.vhd 114
B.3 RAMs 116
B.3.1 RAM des métriques 116
B.3.2 RAM des décisions 117
B.4 Recherche du minimum 118
B.5 Registre à décalage 119
B.6 Trace Back 120
B.7 Viterbi Decoder 123

Table des figures

1.1 Chaîne de transmission numérique. 4
1.2 Diagramme de la modulation QPSK utilisant le code Gray. 6
1.3 Diagramme de Berger. 9
1.4 Canal Binaire Symétrique. 10
1.5 Canal AWGN. 11
1.6 État de l'information dans la chaîne de transmission. 12
1.7 Stratégie de décodage. 13
1.8 Détection et correction d'erreurs. 14
1.9 Codeur convolutionnel $m = 2$ et $R = 1/2$. 15
1.10 Codeur convolutionnel de rendement $R = k/n$. 16
1.11 Registre à décalage dans la forme canonique du codeur. 17
1.12 Schéma du codeur convolutionnel choisi. 17
1.13 Diagramme d'état du codeur convolutionnel. 18
1.14 Représentation en arbre du codeur convolutionnel. 19
1.15 Codeur utilisé dans le GSM. 21

2.1 Illustration de l'opération ACS Taux $R = 1/n$. 26
2.2 État du treillis à l'instant $i = 1$. 28
2.3 État du treillis à l'instant $i = 2$. 28
2.4 État du treillis à l'instant $i = 3$. 29
2.5 État du treillis à l'instant $i = 4$. 29
2.6 État du treillis à l'instant $i = 5$. 30
2.7 État du treillis à l'instant $i = 6$. 30
2.8 Décodage en remontant le treillis. 31
2.9 Illustration des différents modules du VD. 32
2.10 Illustration de la BMU. 33
2.11 Schéma de l'ACS pour $R = 1/n$. 34
2.12 Regroupement des papillons pour un treillis $N = 8$ et $R = 1/2$ 35
2.13 Illustration de la DU. 36

3.1 Schéma bloc du décodeur. 37
3.2 Organigramme appliqué par l'ACS. 39

3.3 Schéma détaillé de l'ACS. 40
3.4 Architecture d'un papillon. 41
3.5 Architecture détaillée du papillon. 42
3.6 Unité de calcul des métriques modifiée (illustration pour un seul papillon). 44
3.7 Architecture finale de la MCU. 45
3.8 Mémoire du Trace Back. 46
3.9 État de la mémoire du TB pour $0 \leq i < L_{TB}$. 47
3.10 État de la mémoire du TB pour $L_{TB} \leq i < 2L_{TB}$. 47
3.11 État de la mémoire du TB pour $2L_{TB} \leq i < 3L_{TB}$. 48
3.12 Organigramme de la fonction de recherche du minimum. 49
3.13 Schéma détaillé du module TB. 50
3.14 Architecture finale de la DU. 51

4.1 Représentation des signaux u et b. 56
4.2 Représentation des signaux r et $\widehat{u}$: Essai 1. 57
4.3 Représentation des signaux r et $\widehat{u}$: Essai 2. 57
4.4 Représentation des signaux u et b. 58
4.5 Représentation des signaux r et $\widehat{u}$: Essai 1. 59
4.6 Représentation des signaux r et $\widehat{u}$: Essai 2. 59
4.7 Représentation des signaux u et b. 60
4.8 Représentation des signaux r et $\widehat{u}$: Essai 1. 61
4.9 Représentation des signaux r et $\widehat{u}$: Essai 2. 61
4.10 Représentation des signaux u et b. 62
4.11 Représentation des signaux r et $\widehat{u}$: Essai 1. 63
4.12 Représentation des signaux r et $\widehat{u}$: Essai 2. 63
4.13 Représentation des signaux u et b. 64
4.14 Représentation des signaux r et $\widehat{u}$: Essai 1. 65
4.15 Représentation des signaux r et $\widehat{u}$: Essai 2. 65
4.16 Test Bench du codeur convolutionnel à sortie parallèle. 66
4.17 Papillon 000-001 sans les bits Flags. 72
4.18 RAM des métriques. 73
4.19 RAM des bits de décision. 74
4.20 Test Bench du registre à décalage. 75
4.21 Test Bench Pile LIFO. 75
4.22 Structure du décodeur. 76

Liste des tableaux

1.1 Autre représentation du codeur convolutionnel. 20

2.1 Paramètres du VD pour certaine normes de communication. 24

3.1 Résultats pour une seule opération ACS 43

Liste des symboles

ACS Add Compare Select

ACSU Add Compare Select Unit

ARQ Automatic Repeat reQuest

AWGN Additive White Gaussian Noise

BMU Branch Metric Unit

BPSK Binary Phase Shift Keying

BSC Binary Symmetric Channel

CDMA Code Division Multiple Access

DU Decision Unit

DVB Digital Video Broadcasting

EGPRS Enhanced General Packet Radio Service

FSK Frequency Shift Keying

GSM Global System for Mobile communications

LAN Local Area Network

LIFO Last In First Out

MAP Maximum A Posteriori

MCU Metrics Calculating Unit

ML Maximum Likelihood

MPSK M-ary Phase Shift Keying

NRZ Non Return to Zero

PMU Path Metric Unit

QAM Quadrature Amplitude Modulation

QPSK QuadriPhase Shift Keying

RAM Random Access Memory

RE Register Exchange

ROM Read Only Memory

SISO Soft Input Soft Output

SNR Signal to Noise Ratio

SOVA Soft-Output Viterbi Algorithm

TB Trace Back

UMTS Universal Mobile Telecommunications System

UMTS Universal Mobile Telecommunications System

VA Viterbi Algorithm

VD Viterbi Decoder

WiFi Wireless Fidelity

Introduction

Cadre général et objectifs

L'histoire des transmissions a été marquée par plusieurs évolutions. Après les pigeons voyageurs, l'information est écrite sur un support et transmise au destinataire ; le courrier est toujours utilisé dans notre vie quotidienne.

Avec l'arrivée de l'électricité, on a pu commencer à transporter des messages sous une forme moins matérielle, à des vitesses beaucoup plus importantes. De nos jours, nous assistons à une forte augmentation des besoins en termes de communication numérique notamment pour les applications mobiles. La communication à distance (téléphone, télévision, satellite, etc...), entre machines et usagers nécessite des canaux de transmission acheminant l'information. Les canaux utilisés sont en général loin d'être parfaits, la différence entre eux est la manière ou la probabilité de modifier une information en une autre.

Parmi les critères de développement des nouveaux standards de communication tels que l'UMTS, la qualité de transmission est devenue un facteur primordial, ceci s'explique par l'augmentation de la gamme des applications potentielles nécessitant une transmission fiable de données, c'est-à-dire le but dans tout système de communication est d'assurer une transmission de donnée sans erreurs. Pour respecter ce cahier de charge plusieurs techniques ont été développées (amélioration des canaux de transmission, augmentation de la puissance d'émission, amélioration du facteur de bruit...) cependant, ces techniques s'avèrent très coûteuses en énergie et technologie, ce qui nous pousse à trouver une solution plus abordable. Le contrôle de l'erreur par le codage est donc indispensable. Pour cela, l'information devra être codée d'une manière à pouvoir détecter les erreurs, ou ce qui est encore mieux de les corriger automatiquement. On a été amené à concevoir des codes détecteurs et correcteurs d'erreurs. Parmi ces codes on peut citer les codes en bloc. Les codes convolutifs ou convolutionnel, introduits en 1955 par Elias[2], peuvent être considérés comme un cas particulier et une alternative au codes en blocs linéaires. Pour décoder et reconstituer l'information original, souvent on a recours à l'algorithme le plus répandu : l'algorithme de Viterbi.

Organisation du mémoire

L'objectif de notre travail consiste à étudier la simulation et réaliser l'implémentation sur Circuit Reconfigurable d'un décodeur de VITERBI à décision ferme. Ce circuit sert à décoder l'information au sein des systèmes de communication numérique afin de corriger les erreurs de transmission.

Le travail présenté dans ce mémoire est organisé en quatre chapitre. Dans le premier chapitre, nous présentons un bref rappel théorique concernant la chaine de transmission numérique, ensuite nous enchainons avec des définitions qui concerne le codage de canal et des codes correcteurs, puis nous parlons des Codeurs Convolutionnels et illustrons leurs fonctionnement par un exemple, afin d'introduire notre travail qui porte sur le décodeur de VITERBI à décision ferme. À la fin nous verrons un exemple d'application des ces codeurs.

Dans le second chapitre nous parlons de l'algorithme de VITERBI et ces différents types (SOVA, SISO) puis nous exposons le schéma fonctionnel du décodeur et ses différents étages fonctionnels qui le constituent (BMU, PMU et DU). Le troisième chapitre décrit en détails notre travail. Nous illustrons l'architecture proposé aussi les modifications apportées au schéma vu dans le deuxième chapitre (MCU et DU).

Dans le quatrième chapitre nous présentons brièvement la carte FPGA utilisée (Virtex2 XC2V3000-5FG676) puis nous exposons les simulations faites sous MATLAB® et ISE de Xilinx.

Chapitre 1

Codeurs Convolutionnels

1.1 Introduction

Les codes convolutifs, introduits en 1955 par Elias[2], peuvent être considérés comme un cas particulier des codes en bloc linéaires, mais un point de vue plus large nous fera découvrir que la structure convolutive additionnelle munit le code linéaire de propriétés favorables qui facilitent à la fois son codage et améliorent ses performances.

Dans ce premier chapitre nous parlerons principalement des codeurs convolutionnels qui constituent une méthode de codage permettant d'effectuer des transmissions sans les erreurs qui peuvent être causées par le bruit du canal de transmission. Nous commencerons par établir quelques définitions essentielles concernant la chaîne de transmission numérique. Puis nous parlerons du codage de canal de son utilité et des codes correcteurs d'erreur. Et enfin nous aborderons les codes convolutionnels qui sont un type très répandu des codes correcteur d'erreur.

1.2 Chaîne de transmission numérique et théorie de l'information

Un signal quelconque ne peut pas être transmis sans modification à travers un canal bruité. Il doit être traité. Il subit donc plusieurs opérations. Il est compressé, codé, modulé avant la transmission mais aussi démodulé, décodé à la réception. Il traverse donc plusieurs étapes constituant la chaîne de transmission numérique.

1.2.1 Chaîne de transmission numérique

C'est l'ensemble d'opérations à appliquer à un signal afin de le transmettre numériquement à travers un canal de transmission.

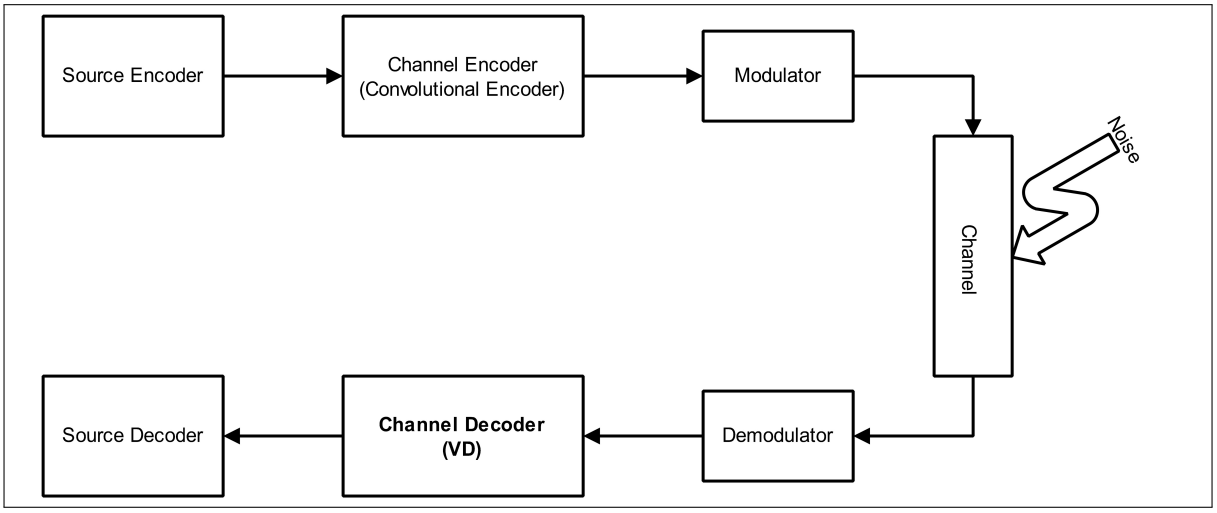


FIGURE 1.1: Chaîne de transmission numérique.

1.2.1.1 Canal

Le canal est le support de transmission de l'information. Il peut être filaire, optique, hertzien...

Le moyen de transmission introduit un nombre d'effets indésirables comme l'atténuation, la distorsion, l'interférence et le bruit. Ces effets indésirables engendrent une incertitude quant à la réception correcte et intègre de l'information envoyée.

La manière dont les symboles transmis sont corrompus est décrite dans les définitions des différents types de canaux :

Canal sans mémoire

Dans un canal sans mémoire, la probabilité d'erreur est indépendante du symbole transmis.

Canal symétrique

Dans un canal symétrique la probabilité qu'un symbole i soit transmis sachant que le symbole j a été reçu est égale à la probabilité de recevoir un i sachant qu'un j a été envoyé. Et ce quelles que soient les valeurs de i et de j.

1.2.1.2 Codeur de source

On donne habituellement à l'information une représentation discrète, probablement en conjonction avec les techniques de suppression de redondance au sein de la donnée. La quantité d'information contenue dans n'importe quel message est définie en terme de

probabilité p que le message soit transmis. L'information H, mesurée en bits, est donnée par[9] :

$$H = p \log_2 \left(\frac{1}{p} \right) \tag{1.1}$$

Par exemple un message avec 1% de chance d'être transmis contiendrait approximativement 6.64 bits d'information. (A ne pas confondre avec le bit informatique qui signifie *binary digit*). S'il y a M messages probables et que la sélection d'un message m a une probabilité de p_m, la quantité moyenne d'information transférée dans les messages est[9] :

$$\overline{H} = \sum_{m=0}^{M-1} p_m \log_2 \left(\frac{1}{p_m} \right) \tag{1.2}$$

et ce dans la condition :

$$\sum_{m=0}^{M-1} p_m = 1 \tag{1.3}$$

Si les messages sont équiprobables, c'est-à-dire $p_m = \frac{1}{M}$, alors l'information moyenne transférée est $\overline{H} = \log_2 (M)$. Par exemple avec 256 messages un code de 8 bits peut être utilisé pour représenter n'importer quel message, et s'ils sont équiprobables, l'information contenue dans n'importe quel message est aussi de 8 bits.

1.2.1.3 Codeur de canal

Globalement, le codage de canal sert à introduire de la redondance dans la séquence transmise afin de détecter et corriger les erreurs dus à la transmission, lors du décodage. Le nombre de bits transmis est donc supérieur au nombre de bits nécessaires pour représenter l'information (Ajout de redondance). Sans cela, le code ne nous permettrai même pas de détecter la présence d'erreur. Par voie de conséquence ne nous donnerais aucun contrôle sur les erreurs introduites. Ce qui veut dire que, théoriquement, une compression incomplète effectuée par le codeur de source comprendrait des propriétés de gestion d'erreur. Néanmoins, en pratique il serait préférable de compresser l'information de la source autant que possible puis de réintroduire de la redondance d'une manière à pouvoir l'exploiter correctement à la réception : c'est le principe de fonctionnement du codeur de canal.

1.2.1.4 Modulation

Le modulateur peut être considéré comme un convertisseur numérique analogique, préparant le flux de bits au canal de transmission qui est analogique. Initialement, le flux de bit est dans une représentation en bande de base, c'est-à-dire que ça fréquence est basse et que le changement de signaux se fait à une fréquence comparable à celle

des symboles numériques représentés. Une représentation appropriée est le format NRZ (Non Return to Zero), qui représente les bits par des niveaux $+V$ ou $-V$.

Même s'il est possible de transmettre ce signal, il est généralement translaté à une fréquence plus élevée. La raison à cela est la possibilité d'utiliser différentes régions du spectre pour différentes transmissions, ainsi que le fait que les hautes fréquences ont des longueurs d'ondes plus petites et nécessitent donc de plus petites antennes.

Un schéma de modulation très utilisé est le BPSK (Binary Phase Shift Keying). Dans ce schéma, on multiplie le signal NRZ en bande de base par une porteuse dont la fréquence est un multiple du débit de bits. Il en résulte que le signal transmis durant l'intervalle de temps correspondant à un bit est soit la porteuse elle même, soit son inverse.

Il est aussi possible d'utiliser une seconde porteuse déphasé de $\frac{\pi}{2}$ de la première, la moduler et l'ajouter à la première. En d'autres termes, si le signal BPSK est $\pm\cos(2\pi f_c t)$ (f_c étant la fréquence de la porteuse et t le temps) le second signal sera $\pm\sin(2\pi f_c t)$ et la résultante sera alors :

$$s(t) = \sqrt{2}\cos\left(2\pi f_c t + i\frac{\pi}{4}\right); \quad i = -3,\ -1,\ +1,\ +3 \tag{1.4}$$

Ceci est connu sous le nom de QPSK (QuadriPhase Shift Keying), est possède l'avantage _par rapport à la modulation BPSK_ de permettre la transmission de deux fois plus de bits durant le même temps et en utilisant la même bande passante[11].

Un diagramme de phase est décrit dans la Figure 1.2. La distribution des valeurs des bits sur les phases implique que chaque porteuse est indépendamment modulée en alternant les bits du flux de données codées.

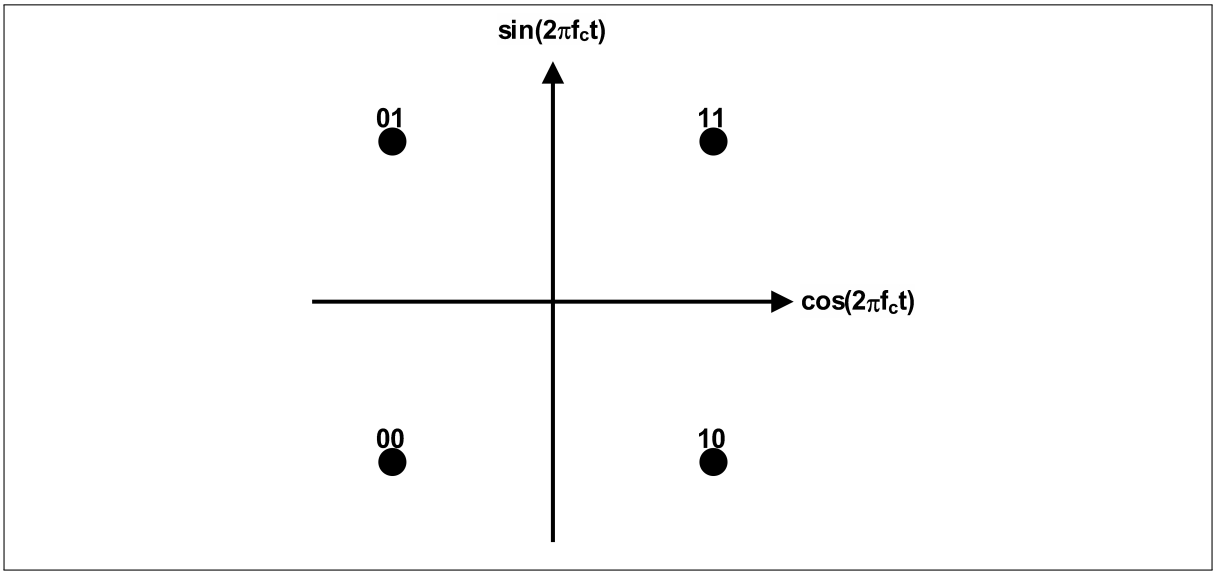


FIGURE 1.2: Diagramme de la modulation QPSK utilisant le code GRAY.

D'autres méthodes de modulations comprennent la FSK, dans laquelle la donnée module la fréquence du signal transmis ; l'avantage de la FSK est la simplicité d'implémentation. Néanmoins, sa résistance au bruit est inférieure à celles des modulations BPSK et QPSK. Dans la MPSK un nombre plus important de phases peut être transmis. En ce qui concerne la modulation QAM, une paire de porteuses en quadrature de phase sont utilisées ; on leur attribue deux amplitudes différentes avant de les additionner pour produire un signal contenant différentes amplitudes et différentes phases. La méthode QAM résiste plus au bruit que son équivalent MPSK, mais la différence d'amplitude peut poser problème dans les systèmes comportant des éléments non linéaires[11].

1.2.1.5 Bruits

Un signal est toujours affecté de petites fluctuations plus ou moins importantes. Ces fluctuations, dont les origines peuvent être diverses sont appelées bruit électrique, bruit de fond, ou tout simplement bruit. On peut citer :

– Le bruit de « friture » dans un récepteur radio entre deux stations ;
– Le bruit de souffle à la sortie d'un amplificateur.

Un signal n'est jamais sans bruit ; simplement il peut être affecté d'un bruit plus ou moins important. Si celui-ci est très faible par rapport au signal, il devient invisible à l'oscilloscope.

Cause externe à la chaine

C'est le bruit qui affecte déjà le signal à l'entrée de la chaîne et qui est amplifié et filtré avec le signal.

• L'antenne du récepteur capte le signal de l'émetteur mais aussi des parasites industriels et le bruit de fond cosmique de tous les astres qui rayonnent des ondes électromagnétiques ;

• La tête de lecture du tourne-disque qui capte la musique inscrite dans le sillon mais aussi les bruits de surface, les vibrations de la platine et les décharges électrostatiques.

Pour ce type de bruit, une conception soignée du produit (mécanique, électronique, blindages, filtrages, etc . . .) nous donne un bruit minimal en sortie.

Cause interne à la chaine

L'agitation thermique des électrons provoque des fluctuations aléatoires de la tension en tout point d'un circuit. C'est le bruit thermique qui existe toujours. On peut le diminuer en choisissant des composants à faible bruit mais on n'arrivera jamais à le supprimer.

Outre ce bruit thermique, il existe d'autres catégories de bruits, toujours associés à la nature discontinue des porteurs de charges.

Les fluctuations étant aléatoires, il est clair qu'en moyenne, $b(t)$ est aussi souvent positive que négative. Nous en déduisons un premier résultat important :

La valeur moyenne d'une tension de bruit est nulle $b(t) = 0$, par contre, si nous élevons cette tension au carré avant de prendre la valeur moyenne, le résultat ne sera plus nul. Nous en déduisons que :

La valeur efficace d'une tension de bruit n'est pas nulle $B_{eff} = \sqrt{\overline{b(t)^2}} \neq 0$. Une tension de bruit sera donc caractérisée par sa valeur efficace.

1.2.2 Théorie de l'information

Shannon démontre en 1948 que l'on peut réduire autant que l'on veut les erreurs et donc transmettre de l'information vierge d'erreur du moment que l'on réduit le débit de transmission. Il y a donc un compromis à faire entre le débit de transmission des bits et le nombre d'erreurs. Le débit maximal pour lequel la transmission se fait sans erreur est appelé Capacité du canal de transmission. Il est possible de réaliser une transmission sans erreur à travers un canal bruité, tant que le débit de transmission n'excède pas la capacité du canal. Pour expliquer ce résultat nous devons mesurer l'information.

1.2.2.1 Entropie

Dans le reste de cette partie nous adopterons les notations ci-dessous :

Considérons une source discrète et sans mémoire. A un instant donné, la source discrète d'information émet le symbole aléatoire discret $X = x_i$ qui peut prendre une des M valeurs possibles x_1, $x_2, \ldots,$ x_M. Les probabilités d'apparition de ces symboles sont : $P_X(x_1)$, $P_X(x_1), \ldots,$ $P_X(x_M)$ avec $P_X(x_i) = \Pr\{X = x_i\}$.

L'information moyenne associée à la variable aléatoire X ou entropie comme définie précédemment, est[11] :

$$I(X) = -\sum_{i=1}^{M} P_X(x_i) \log_2 P_X(x_i) \tag{1.5}$$

1.2.2.2 Capacité de canal

A l'aide du concept d'entropie, nous pouvons modéliser le canal selon le diagramme de Berger[2] ; Figure 1.3 :

Ici, X correspond au symbole d'entrée et R correspond au symbole reçu. Nous considérons aussi que M valeurs d'entrée sont possibles x_1, $x_2, \ldots,$ x_M et N valeurs de sorties r_1, $r_2, \ldots,$ r_N. A l'aide des lois de probabilités conditionnelles nous avons :

$$\Pr_{X|R}(x_i|r_j) = \Pr\{X = x_i|R = r_j\} \tag{1.6}$$

et

$$\Pr_{R|X}(r_j|x_i) = \Pr\{R = r_j|X = x_i\} \tag{1.7}$$

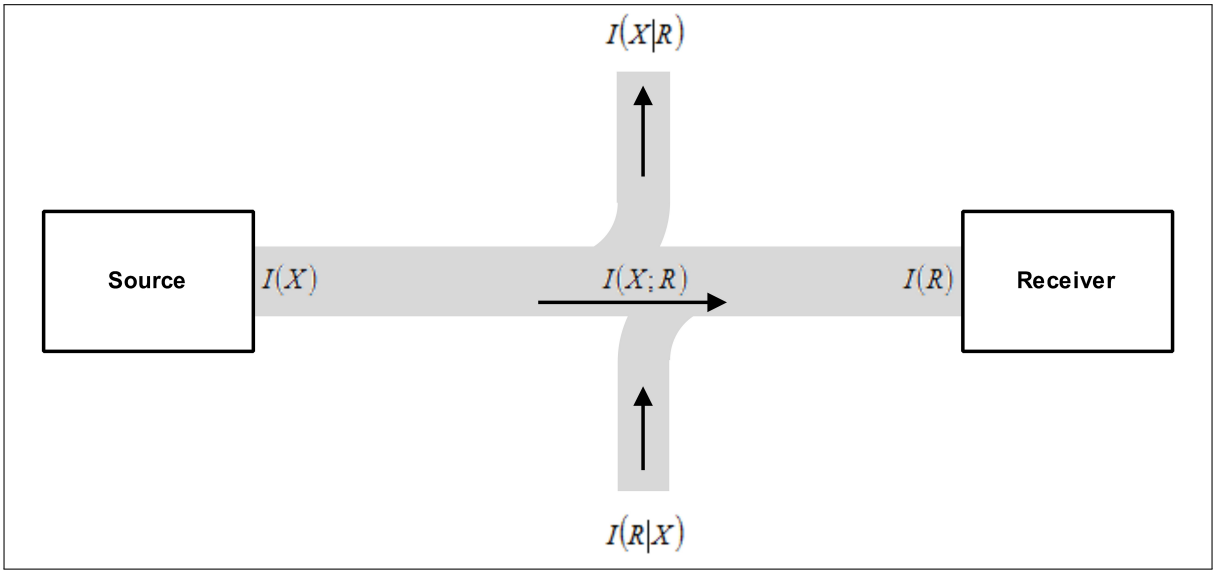


FIGURE 1.3: Diagramme de Berger.

Les entropies conditionnelles sont données par :

$$I\left(X|R\right) = -\sum_{i=1}^{M}\sum_{j=1}^{N} \Pr_{X,R}\left(x_i, r_j\right) \log_2\left(\Pr_{X|R}\left(x_i|r_j\right)\right) \tag{1.8}$$

et

$$I\left(R|X\right) = -\sum_{i=1}^{M}\sum_{j=1}^{N} \Pr_{X,R}\left(x_i, r_j\right) \log_2\left(\Pr_{R|X}\left(r_j|x_i\right)\right) \tag{1.9}$$

Avec ces probabilités conditionnelles, l'information mutuelle :

$$I\left(X;\ R\right) = I\left(X\right) - I\left(X|R\right) = I\left(R\right) - I\left(R|X\right) \tag{1.10}$$

peut être déduite. Elle mesure la quantité d'information transmise à travers le canal de l'entrée à la sortie pour une source d'information donnée.

La capacité C de ce canal est obtenue en maximisant l'information mutuelle $I\left(X;\ R\right)$ tout en respectant les caractéristiques statistiques de la variable d'entrée X, c'est-à-dire en choisissant de manière appropriée les probabilités $\left\{\Pr_X\left(x_i\right)\right\}_{1\leq i\leq M}$. Cela mène à :

$$C = \max_{\left\{\Pr_X\left(x_i\right)\right\}_{1\leq i\leq M}} I\left(X|R\right) \tag{1.11}$$

Si l'entropie d'entrée $I\left(X\right)$ est inférieure à la capacité du canal $I\left(X\right) < C$, alors l'information peut être transmise à travers un canal bruité avec une probabilité d'erreur aussi petite que l'on veut. De cette manière la capacité C quantifie réellement la capacité de transmission d'information du canal.

1.2.2.3 Canal Symétrique Binaire

Un exemple très important de canal sans mémoire est le Canal Symétrique Binaire ou BSC. Le canal de la Figure 1.4 montre le diagramme d'un canal symétrique binaire avec une probabilité d'erreur p.

En calculant l'information $I(X;\ R)$, la capacité du canal symétrique binaire est :

$$C = 1 + p\log_2(p) + (1-p)\log_2(1-p) \tag{1.12}$$

La capacité du canal est égale :

$$C = \begin{cases} 1 & p = 0 \text{ ou } p = 1 \\ 0 & p = 1/2 \end{cases} \tag{1.13}$$

Contrastant avec le canal symétrique binaire qui possède des entrées et sorties discrètes, le canal AWGN est définit par des variables aléatoires réelles continues.

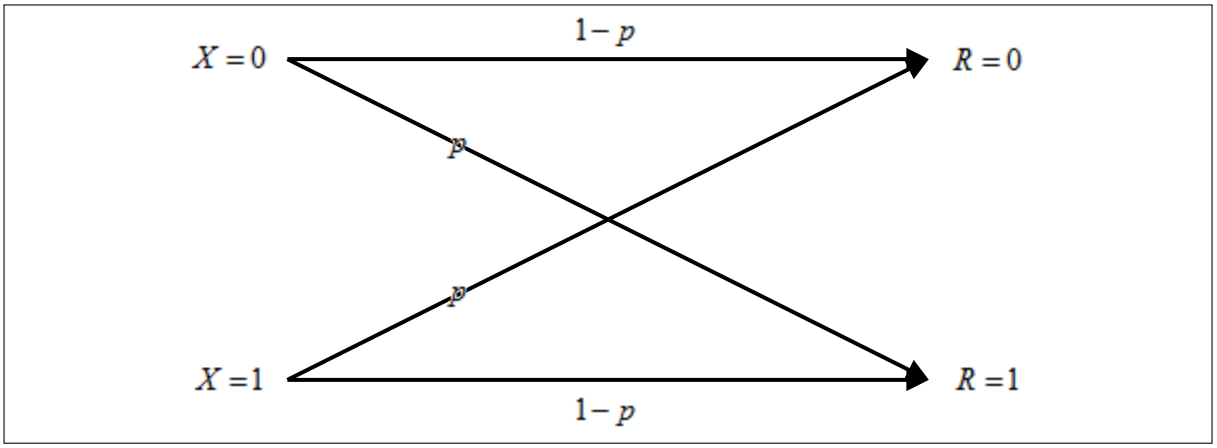


FIGURE 1.4: Canal Binaire Symétrique.

1.2.2.4 Canal AWGN

Jusqu'ici nous n'avons considéré que des symboles à valeurs discrètes. Le concept d'entropie peut être transféré aux variables aléatoires continues réelles, en incluant la dite entropie différentielle. Il en résulte que le canal à entrée et sorties réelles peut aussi bien être caractérisé par l'information mutuelle $I(X;\ R)$ et son maximum par la capacité C.

- Puissance du signal : $S = \mathrm{E}\{X^2\}$;
- Puissance du bruit : $N = \mathrm{E}\{Z^2\}$.

La capacité du canal AWGN est donné par :

$$C = \frac{1}{2}\log_2\left(1 + \frac{S}{N}\right) \tag{1.14}$$

La capacité du canal dépend exclusivement du SNR ; $SNR = \frac{S}{N}$.

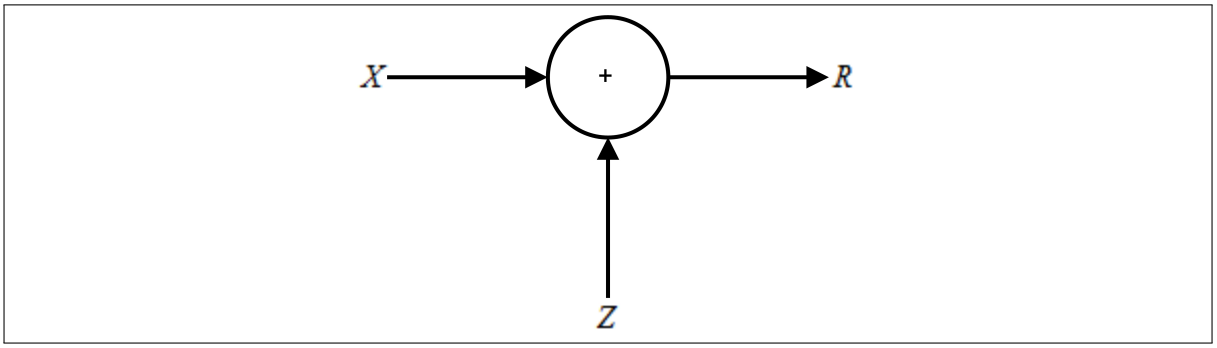


FIGURE 1.5: Canal AWGN.

Pour comparer les capacités du canal symétrique binaire et du canal AWGN, on suppose que le signal est modulé (BPSK), et que la réception est optimisée à l'aide d'un filtre adapté. Le rapport signal sur bruit (SNR) de la sortie réelle R du filtre adapté est donnée par :

$$\frac{S}{N} = \frac{E_b}{N_0/2} \tag{1.15}$$

Avec E_bl'énergie par bit, et N_0 la densité spectrale de puissance du bruit. Si la sortie R du filtre adapté est comparée avec 0, on obtient le canal symétrique binaire avec une probabilité d'erreur :

$$p = \frac{1}{2}\text{erfc}\left(\sqrt{\frac{E_b}{N_0}}\right) \tag{1.16}$$

Ici, erfc est la fonction d'erreur complémentaire.

1.3 Codage de canal et codes correcteurs d'erreur

Même si la théorie de l'information nous dit qu'il est possible de trouver un code de canal qui nous donne une probabilité d'erreur aussi petite que l'on veut pour un canal donné, il n'en demeure pas moins que d'un point de vue pratique cela ne soit pas si aisé.

Il est généralement difficile de concevoir un bon code de canal. Dans cette section nous parlerons des paramètres utilisés pour décrire un code de canal puis nous aborderons le sujet essentiel nous concernant : les codes convolutionnels (ou convolutifs). Dans les chapitres ultérieurs, nous aborderons l'algorithme de Viterbi pour le décodage de ces dits codes, ainsi que ça simulation et son implémentation.

1.3.1 Principes fondamentaux des codes en blocs

1.3.1.1 Paramètres du code

Les codes de canal sont caractérisés par plusieurs paramètres. Les plus importants sont le taux de codage et la distance de HAMMING minimale. A l'aide de ces paramètres l'efficacité du processus d'encodage peut être évaluée pour un code donné.

Dans ce qui suit nous considèrerons les variables suivantes :
- Mot information à coder $u = (u_0,\, u_1, \ldots,\, u_{k-1})$ de longueur k ;
- Mot code émis $b = (b_0,\, b_1, \ldots,\, b_{n-1})$ de longueur n ;
- Mot reçu à décoder $r = (r_1,\, r_2, \ldots,\, r_{n-1})$ de longueur n ;
- Mot décodé $\widehat{u} = (\widehat{u_0},\, \widehat{u_1}, \ldots,\, \widehat{u_{k-1}})$ de longueur k.

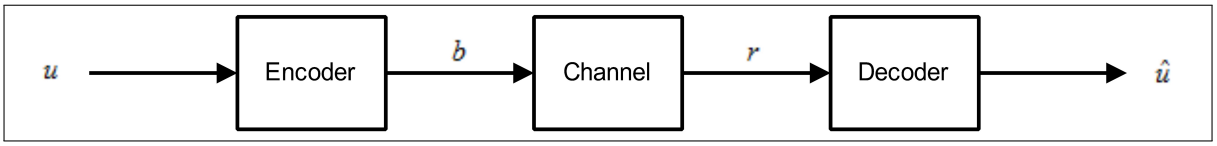


FIGURE 1.6: État de l'information dans la chaîne de transmission.

Le mot décodé $\widehat{u}$ peut être identique à u ou différent, selon que les erreurs soient corrigées ou pas.

Rendement du codage

En partant du principe que tous les symboles u_i de l'information peuvent prendre q valeurs, le nombre de mots informations possibles est donné par :

$$M = q^k \tag{1.17}$$

Sachant que le mot code de longueur n est plus long que le mot d'information de longueur k, le taux auquel l'information transmise à travers le canal est réduite est appelé le taux de codage du code :

$$R = \frac{\log_q (M)}{n} = {}^{k}/_{n} \tag{1.18}$$

Par exemple, si le code consiste uniquement à répéter chaque symbole trois fois, on aura alors $k = 1$, et $n = 3$; ce qui donne un taux de codage de ${}^{k}/_{n} = {}^{1}/_{3} \simeq 0.3333$.

Poids et distance de Hamming

A chaque mot code $b = (b_0,\, b_1, \ldots,\, b_{n-1})$, peut être associée un poids $wt\,(b)$, défini comme le nombre de composantes non nulles $b_i \neq 0$: C'est le poids de HAMMING[1].

$$wt\,(b) = |\{i :\ b_i \neq 0;\ 0 \leq i < n\}| \tag{1.19}$$

La distance entre deux mots b et $\widehat{b}$ est donnée par la Distance de HAMMING[1] :

$$\text{dist}\left(b,\, \widehat{b}\right) = \left|\left\{i :\, b_i \neq \widehat{b}_i;\, 0 \leq i < n\right\}\right| \tag{1.20}$$

La distance de HAMMING traduit le nombre de composantes différentes entre b et $\widehat{b}$, et mesure ainsi la proximité entre b et $\widehat{b}$.

Pour un code consistant en M mots codes $b_0,\, b_1, \ldots,\, b_M$, la distance de HAMMING minimale est donnée par :

$$d = \text{dist}_{\min} = \min_{\forall b \neq \widehat{b}} \left\{\text{dist}\left(b,\, \widehat{b}\right)\right\} \tag{1.21}$$

1.3.1.2 Décodage au maximum de vraisemblance

Il existe plusieurs stratégies de décodage :

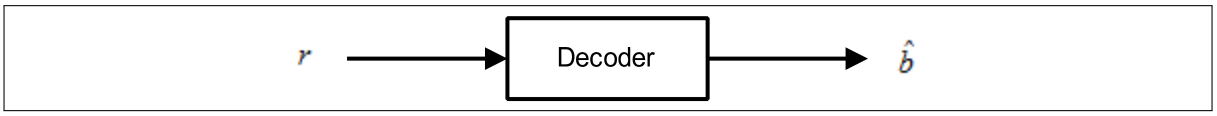


FIGURE 1.7: Stratégie de décodage.

Pour le décodage au minimum de probabilité d'erreur ou *Maximum A Posteriori* (MAP) la règle de décodage est la suivante :

$$\widehat{b}\,(r) = \max\left\{\Pr\left\{b|r\right\}\right\} \tag{1.22}$$

Pour le décodage au maximum de vraisemblance (ML) la règle est la suivante :

$$\widehat{b}\,(r) = \max\left\{\Pr\left\{r|b\right\}\right\} \tag{1.23}$$

Le décodage ML et identique au décodage MAP si tous les mots code sont équiprobables, c'est-à-dire $\Pr\left\{b\right\} = \frac{1}{M}$.

1.3.1.3 Détection et correction d'erreur

En se basant sur la règle de décodage au minimum de distance, nous pouvons évaluer la capacité du code à détecter et corriger les erreurs. Si on considère deux mots codes b et $\widehat{b}$ distant de $d = \text{dist}\left(b,\, \widehat{b}\right)$, où d est la distance minimale de HAMMING. Nous pouvons détecter l'erreur tant que le mot erroné reçu r n'est pas égal à un autre mot code existant. Cette capacité de détection est garantie tant que le nombre d'erreurs est inférieur à la distance minimale de HAMMING. Et ce, pour la simple raison qu'il est nécessaire et non suffisant de changer d composante de b pour arriver à $\widehat{b}$. En conséquence le nombre d'erreurs détectables est :

$$e_{det} = d - 1 \tag{1.24}$$

Pour l'analyse de la capacité de correction d'erreur, nous définissons pour chaque mot b la sphère de correction correspondante de rayon α qui n'est autre que l'ensemble des mots les plus proches de b que de tout autre mot $\widehat{b}$ du codeur. Tous les mots dans cette sphère seront décodés comme un b, c'est-à-dire tous les mots distants de 1, 2, ..., α de b. Ceci est vrai uniquement lorsque les sphères ne possèdent aucun point d'intersection les unes avec les autres, c'est-à-dire si $\alpha < \frac{d}{2}$.

D'où la capacité de correction d'erreur est :

$$e_{cor} = \left[\frac{d-1}{2}\right] \tag{1.25}$$

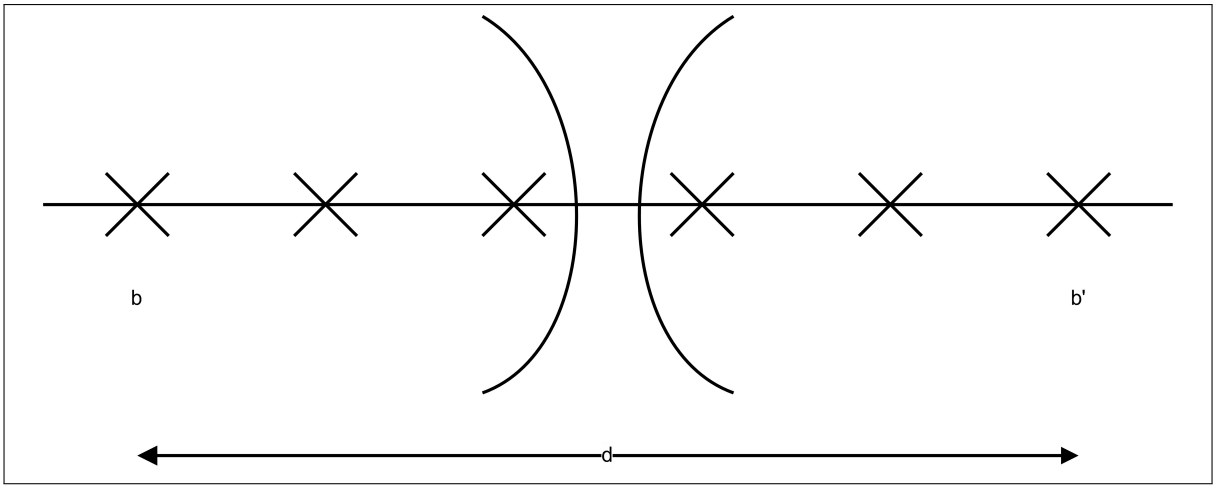


FIGURE 1.8: Détection et correction d'erreurs.

1.4 Codeurs convolutionnels

Aujourd'hui, les codes convolutionnels sont utilisés par exemple dans l'UMTS, et dans les systèmes cellulaires numériques utilisant le GSM. Ils sont aussi utilisés dans les communications par satellite, dans les réseaux sans fil locaux (LAN) et nombre d'autres applications. La principale raison de cette popularité est l'existence d'algorithmes de décodage efficaces. En particulier le très répandu l'algorithme de VITERBI qui est un décodeur utilisant ; en générale ; le décodage ML. Il peut être implémenté avec un niveau de complexité matérielle raisonnable.

1.4.1 Fonctionnement

La manière la plus simple d'introduire les codes convolutionnels est de prendre un exemple simple (voir Figure 1.9).

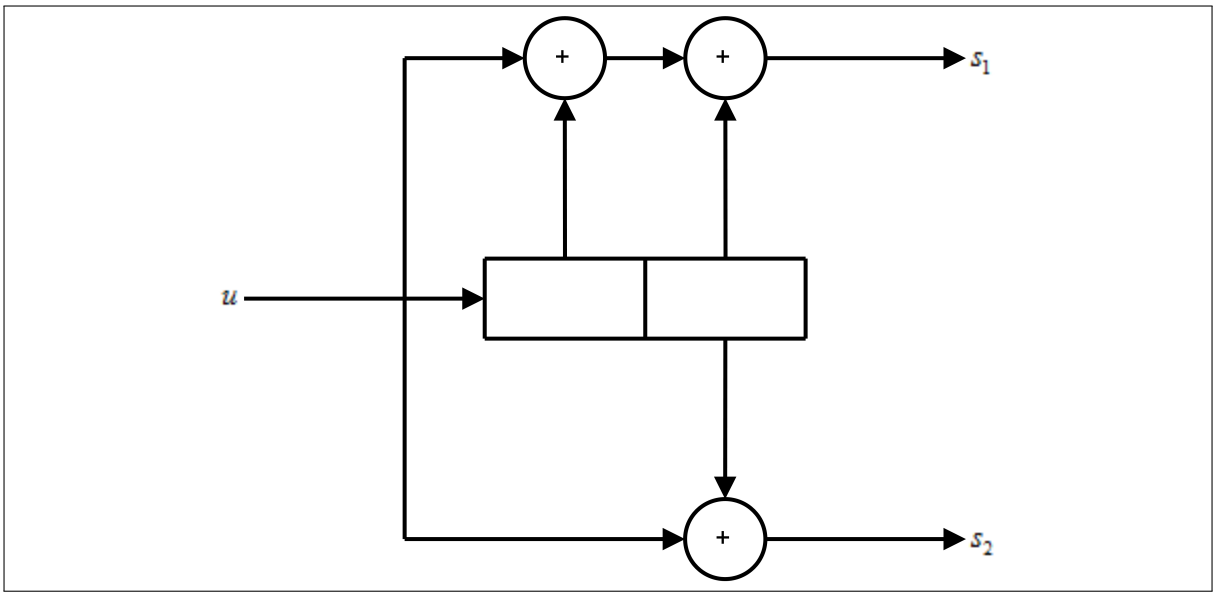


FIGURE 1.9: Codeur convolutionnel $m = 2$ et $R = 1/2$.

L'information entre dans un registre à décalage de longueur $m = 2$ (ce qui correspond à une longueur de contrainte de $L = 3$ et un nombre d'états $N = 2^2 = 4$). La séquence de sortie résulte du multiplexage de deux séquences dénotées s_1 et s_2. Chaque bit de sortie est généré par l'addition modulo 2 de l'entrée actuelle et de certains bits contenus dans le registre. Par exemple, la séquence d'information $u = (1, 1, 0, 1, 0, 0, \ldots)$ sera codée par $s_1 = (1,\ 0,\ 0,\ 0,\ 1,\ 1, \ldots)$ et $s_2 = (1,\ 1,\ 1,\ 0,\ 0,\ 1, \ldots)$ et la séquence de sortie après multiplexage sera $b = (1,\ 1,\ 0,\ 1,\ 0,\ 1,\ 0,\ 0,\ 1,\ 0,\ 1,\ 1, \ldots)$.

Un codeur convolutionnel est un circuit linéaire séquentiel, c'est-à-dire un système invariant dans le temps. Il est donc entièrement caractérisé par sa réponse impulsionnelle. Dans le cas du codeur de la Figure 1.9 les réponses impulsionnelles sont : $g_1 = (1,\ 1,\ 1,\ 0, \ldots)$ et $g_2 = (1,\ 0,\ 1,\ 0, \ldots)$ respectivement. Ces vecteurs générateurs de réponses impulsionnelles sont utiles pour calculer les séquences de sortie pour une séquence d'entrée arbitraire : c'est les polynômes générateurs écrits souvent dans la base 8.

$$s_1 = u \star g_1 \tag{1.26}$$

$$s_2 = u \star g_2 \tag{1.27}$$

Les équations de génération de s_1 et s_2 peuvent être considérées comme la convolution des séquences d'entrée et des générateurs de réponse impulsionnelles $g_1$et g_2. Ceci justifie le nom de codes convolutionnels.

$s_1$et s_2 sont convolutions de la même séquence d'entrée. Cela signifie qu'un seul bit d'information est codé deux fois, le taux d'information utile dans la séquence de sortie est de moitié. En conséquence on dit que le rendement R du codeur est de $R = {}^{k}/_{n} = {}^{1}/_{2}$, où k est le nombre d'entrées du codeur et n le nombre de sorties.

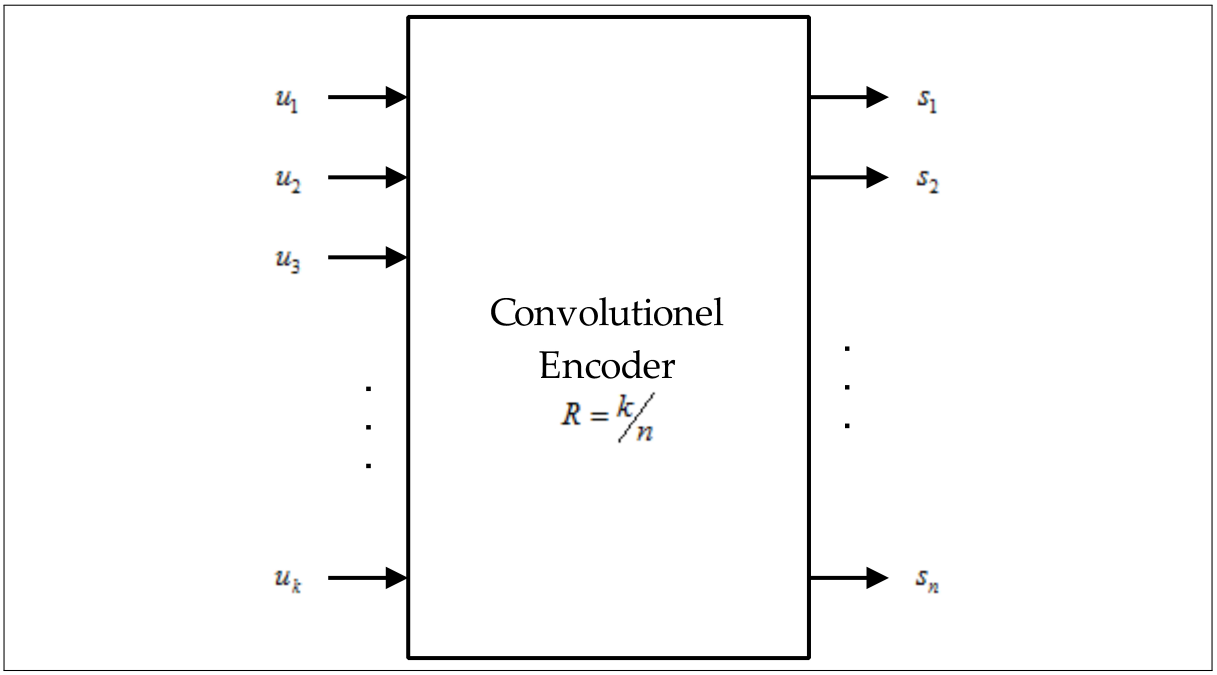


FIGURE 1.10: Codeur convolutionnel de rendement $R = {}^{k}/_{n}$.

Avec les codeurs convolutionnels, il est possible de construire différents circuits résultant en une même séquence de sortie. Le codeur de l'exemple précédent n'a pas de boucle de retour cependant en règle générale un codeur peut avoir des boucles de retour donc une réponse impulsionnelle infinie (voir Figure 1.3.3[2]).

1.4.2 Caractéristiques du codeur convolutionnel utilisé

Pour les besoins de notre sujet nous avons choisi un codeur convolutionnel de rendement $R = {}^{1}/_{2}$ une mémoire de taille $m = 3$ (donc une longueur de contrainte $L = 4$ et un nombre d'états $n = 2^3 = 8$). Les polynômes générateurs de notre codeur convolutionnelles sont $g_1 = (1, 1, 0, 1)$ et $g_2 = (1, 0, 1, 1)$, ou encore en écriture octale : $G = (15_8, 13_8)$.

Le schéma du codeur est celui de la Figure 1.12.

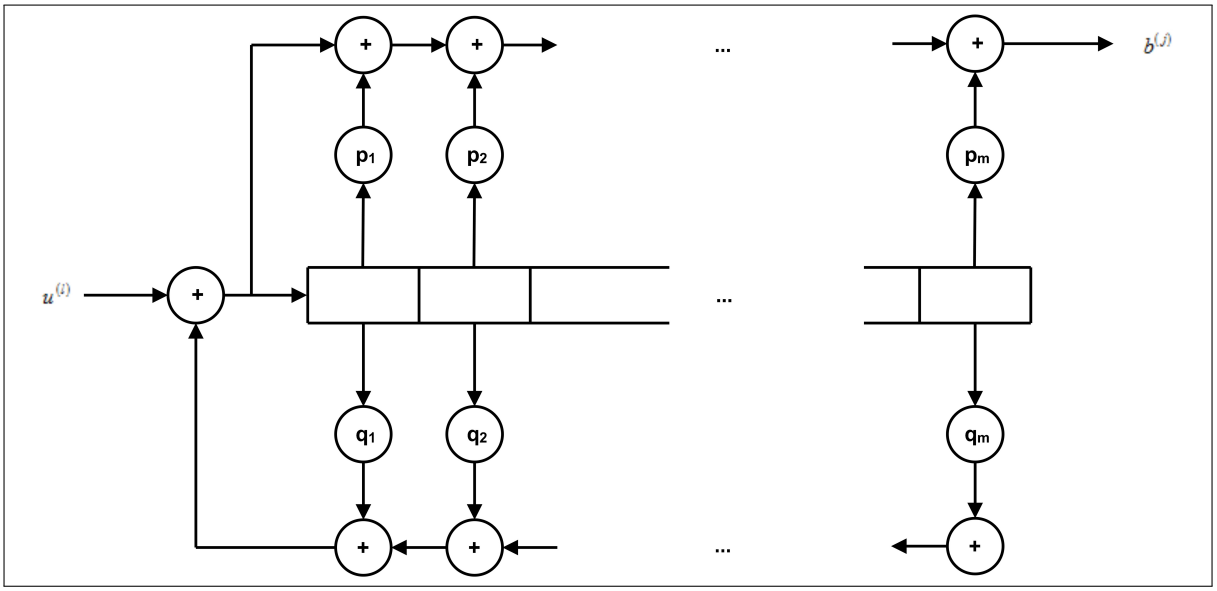


FIGURE 1.11: Registre à décalage dans la forme canonique du codeur.

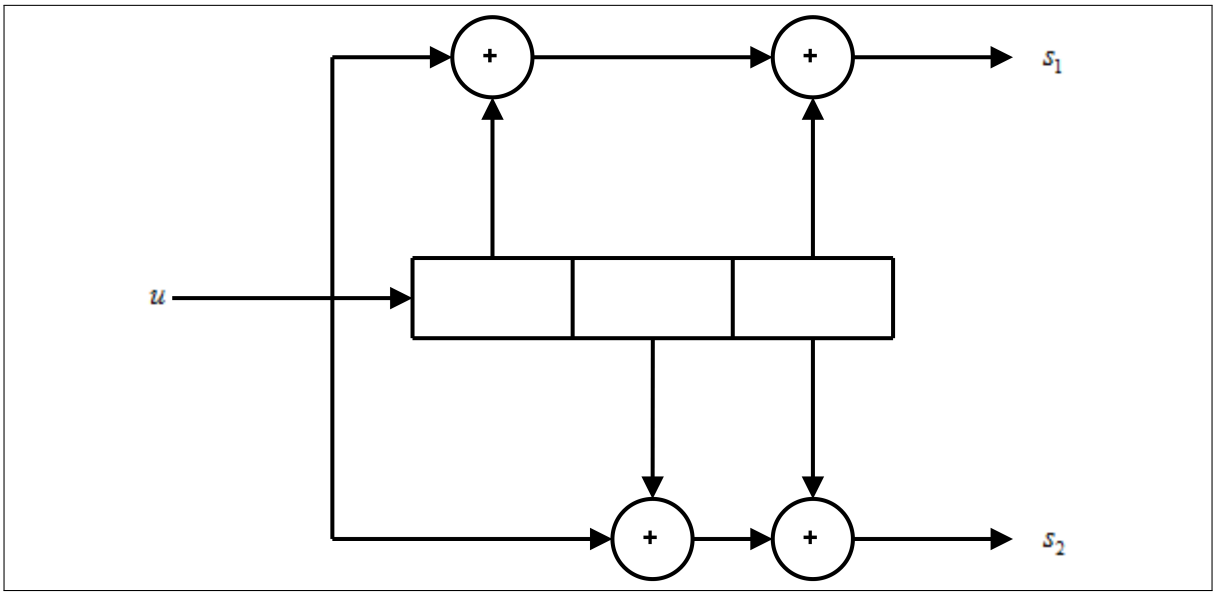


FIGURE 1.12: Schéma du codeur convolutionnel choisi.

1.4.3 Diagramme d'état et ses représentations

Le diagramme d'état du codeur convolutionnel que nous avons choisi est exposé dans le paragraphe précédent et décrit dans la Figure 1.13.

- Chaque case représente un état du codeur, c'est-à-dire le contenu binaire des mémoires du registre à décalage ;
- Chaque branche représente une transition entre deux états et est étiquetée de l'entrée et des deux sorties correspondantes.

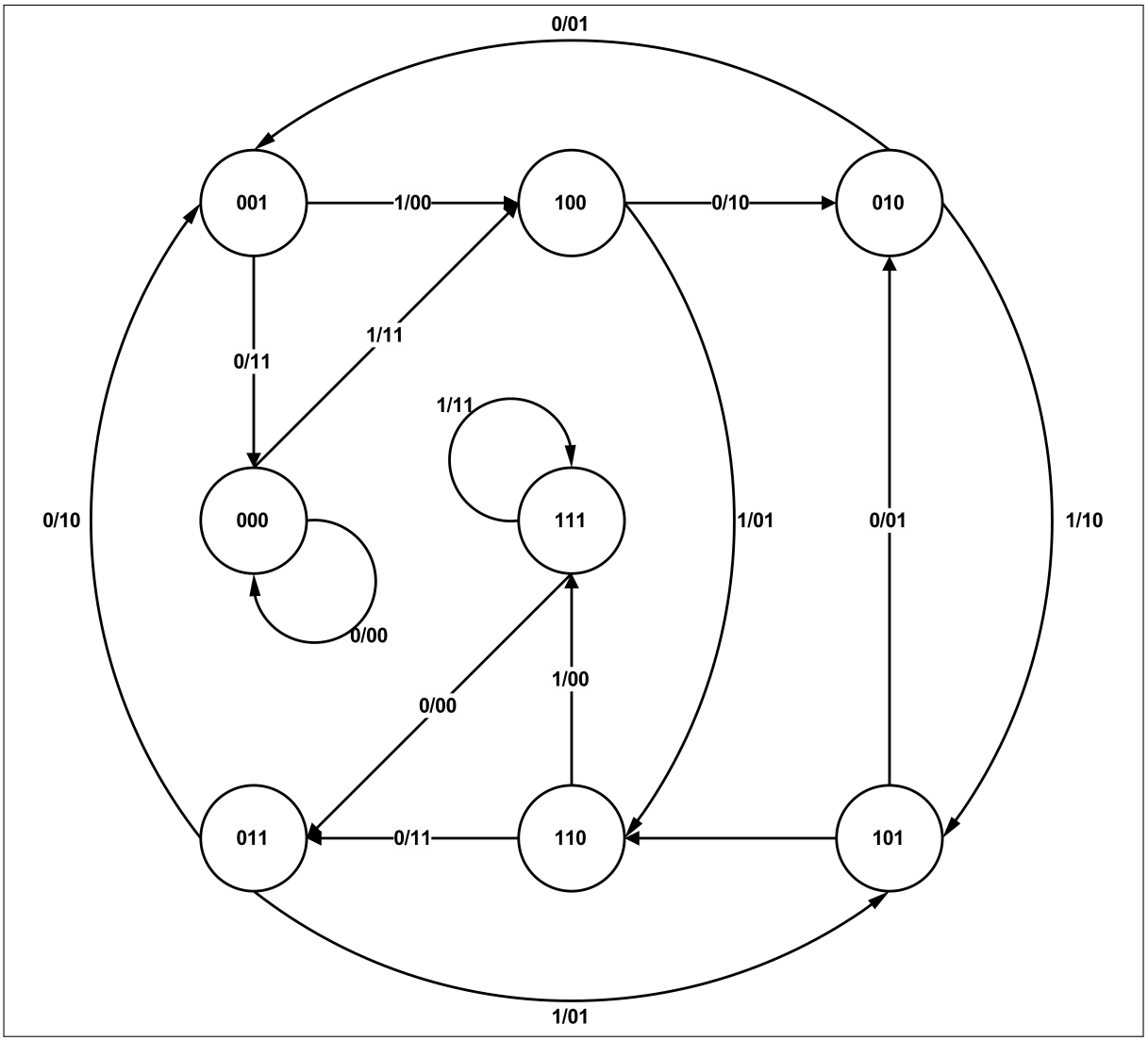


FIGURE 1.13: Diagramme d'état du codeur convolutionnel.

Une autre représentation possible des états du codeur convolutionnel est le diagramme en arbre (Voir Figure 1.14).

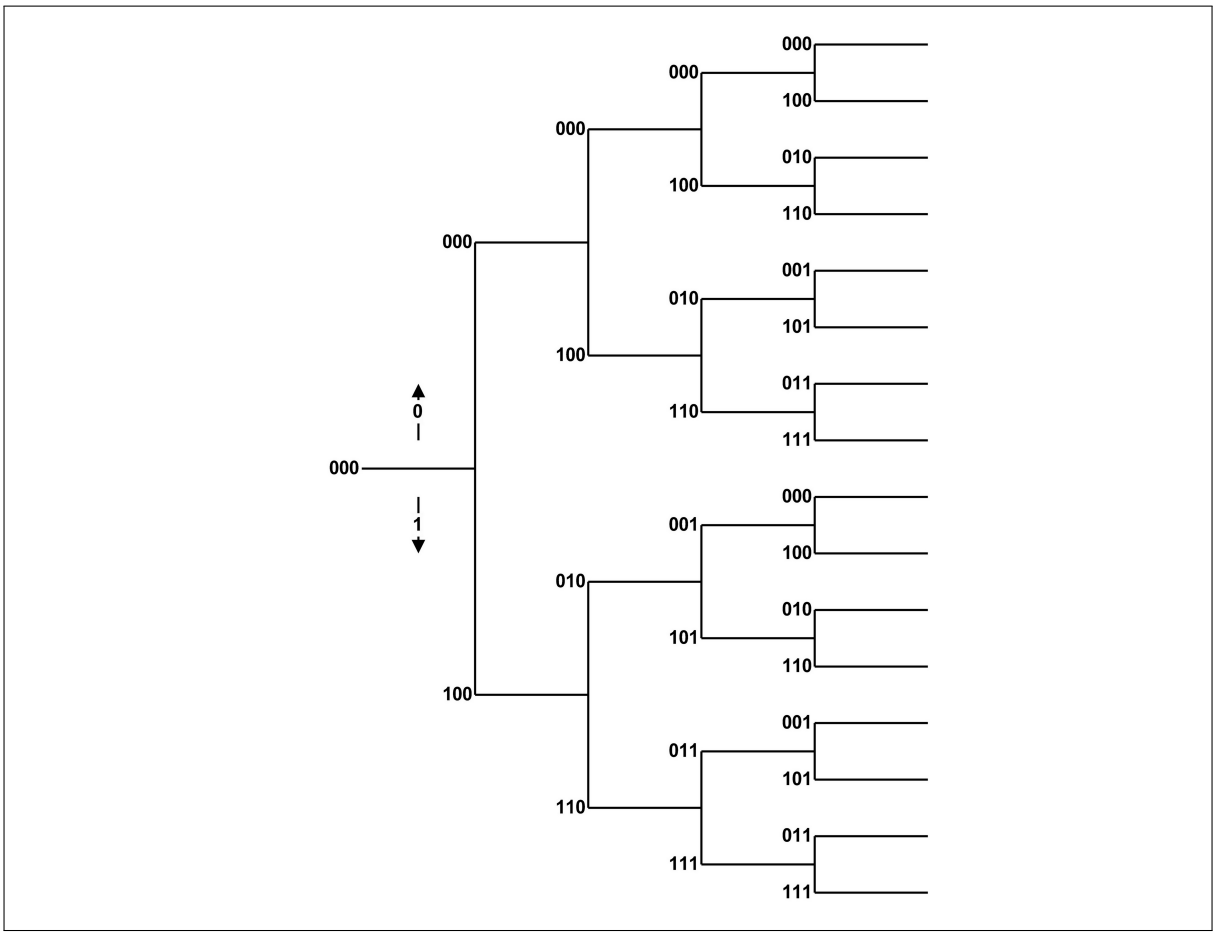


FIGURE 1.14: Représentation en arbre du codeur convolutionnel.

Toujours, en peut aussi représenter le codeur par un tableau qui contient le bit d'entrée, l'état courant du codeur, l'état suivant et le mot code de sortie (Voir Tableau 1.1).

Bit d'entrée	État courant			État suivant			s_1	s_2
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	1	1
0	0	0	1	0	0	0	1	1
1	0	0	1	1	0	0	0	0
0	0	1	0	0	0	1	0	1
1	0	1	0	1	0	1	1	0
0	0	1	1	0	0	1	1	0
1	0	1	1	1	0	1	0	1
0	1	0	0	0	1	0	1	0
1	1	0	0	1	1	0	0	1
0	1	0	1	0	1	0	0	1
1	1	0	1	1	1	0	1	0
0	1	1	0	0	1	1	1	1
1	1	1	0	1	1	1	0	0
0	1	1	1	0	1	1	0	0
1	1	1	1	1	1	1	1	1

TABLE 1.1: Autre représentation du codeur convolutionnel.

1.5 Applications des codes convolutionnels

Dans les communications mobiles, il est nécessaire d'utiliser un codeur de canal afin d'éviter les pertes dues aux erreurs de transmission. Les standards de la téléphonie mobile comme GSM et UMTS distinguent la voix et la communication de données. La transmission de la voix codée est un service à temps réel qui ne permet qu'une petite latence, mais qui tolère les erreurs résiduelles dans le le flux de données. Les erreurs résiduelles pouvant survenir sont masquées efficacement par le décodeur vocal. La transmission de données, quant à elle, requière des taux d'erreurs résiduels très faibles. Les standards GSM et UMTS définissent différents codages de vocaux.

Le codage de canal pour la GSM utilise un codeur convolutionnel ayant une mémoire $m = 4$ (Voir Figure 1.15). Pour terminer le code, quatre zéros (*tail bits*) sont ajoutés. Alors que le standard UMTS utilise un puissant codeur de mémoire $m = 8$ et un taux $R = 1/3$[2].

Le protocole ARQ est une autre application des codes convolutionnels pour la téléphonie mobile. Le principe de l'ARQ est de détecter et de retransmettre les données

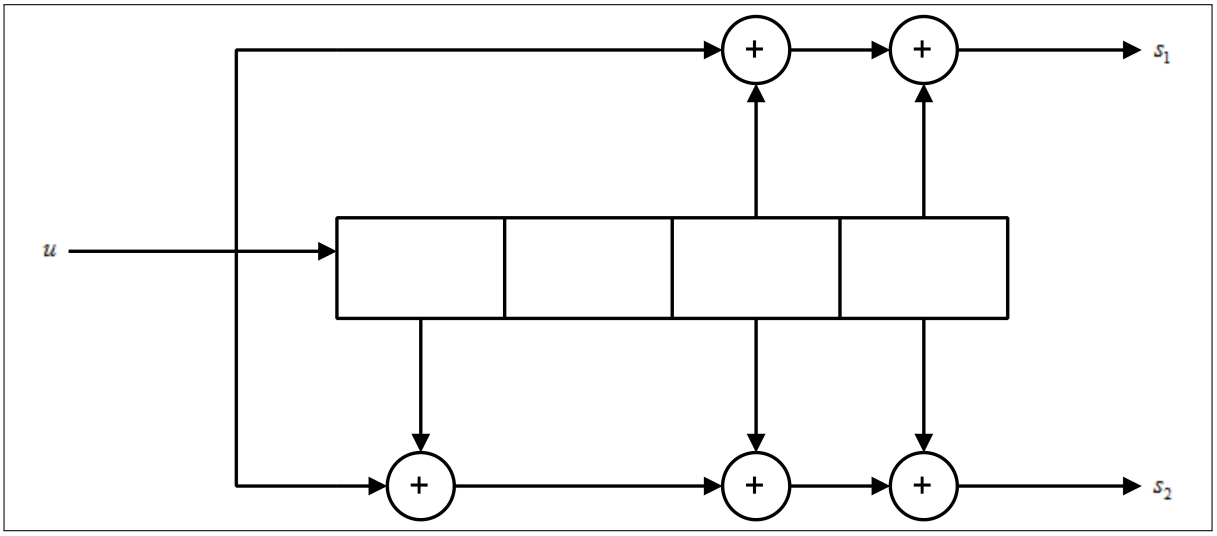


FIGURE 1.15: Codeur utilisé dans le GSM.

corrompues. Ce qui veut dire que le récepteur doit vérifier si l'information reçue est correcte ou erronée. Pour cela, tous les schémas ARQ nécessitent un mécanisme de détection d'erreur[2].

Pour le protocole EGPRS, le codage de canal doit prendre en considération un canal qui varie beaucoup en fonction du temps. Aussi, la modulation et la protection contre les erreurs doivent être ajustables. La donnée transmise est protégée par des codes convolutionnels dont les rendements varient de 0.37 à presque 1.0. Et comme le canal varie très rapidement, le passage d'un schéma de codage et modulation à un autre doit se faire très vite[2].

Chapitre 2

Décodeur de Viterbi

2.1 Introduction

Il existe plusieurs algorithmes de décodage des codes convolutifs. Le plus célèbre est probablement l'algorithme de VITERBI qui repose sur la représentation en treillis des codes [12, 7]. Il permet de trouver, à partir de la séquence des symboles reçus, la séquence d'états dans le treillis la plus probable.

Le décodeur de VITERBI (VD) utilise l'algorithme de VITERBI (VA) pour décoder un flux de bits codés avec un codeur convolutionnel. Il est utilisé dans la CDMA, la GSM, les modems, la communication par satellite (ex. DVB) et le 802.11 Wireless (WiFi 802.11). Le tableau 2.1 résume les paramètres du décodeur pour quelques techniques utilisées dans la communication.

L'algorithme de VITERBI, d'ANDREW VITERBI [1], permet de corriger dans une certaine mesure les erreurs survenues lors d'une transmission à travers un canal bruité.

Son utilisation s'appuie sur la connaissance de la nature du canal, c'est-à-dire la probabilité qu'une information a été modifiée en une autre tout en permettant de simplifier radicalement la complexité de la recherche du message d'origine le plus probable.

2.2 Algorithme de Viterbi

L'algorithme de VITERBI reste la méthode la plus utilisée pour décoder les codes convolutionnels : c'est un décodage dit à maximum de vraisemblance (ou *Maximum Likelihood* en Anglais). Toutefois, l'algorithme de VITERBI reste applicable sur des codes convolutionnels de faible longueur de contrainte ($L \leq 8$). Au-delà de cette limite, sa

1. ANDREA VITERBI devenu après sa naturalisation américaine ANDREW JAMES VITERBI (né à *Bergame* en *Lombardie* le 9 mars 1935) est un ingénieur et entrepreneur américain d'origine italienne. Il est connu pour être l'inventeur de l'algorithme de VITERBI ainsi que pour être l'un des cofondateurs de l'entreprise de télécommunications Qualcomm.

Technologie	Taux R	Longueur de contrainte L	Générateurs g_i
CDMA 2000	$1/4$	9	$g_1 = 765_8$ $g_2 = 671_8$ $g_3 = 513_8$ $g_4 = 473_8$
CDMA IS-95A	$1/2$	9	$g_1 = 153_8$ $g_2 = 561_8$
Cassini	$1/6$	15	$g_1 = 42631_8$ $g_2 = 47245_8$ $g_3 = 56507_8$ $g_4 = 73363_8$ $g_5 = 77267_8$ $g_6 = 64537_8$
UWB (802.15)	$1/3$	7	$g_1 = 171_8$ $g_2 = 165_8$ $g_3 = 133_8$

TABLE 2.1: Paramètres du VD pour certaine normes de communication.

complexité de mise en œuvre impose d'avoir plutôt recours à un algorithme de décodage séquentiel tel que celui de FANO [6].

Dans cette section nous décrivons l'algorithme de VITERBI celui utilisé dans notre travail, mais avant d'entamer cette description, nous allons parler des différents types du VA qui existent.

L'algorithme de VITERBI originel effectue un décodage à sortie ferme, c'est-à-dire qu'il fournit une estimation binaire de chacun des symboles transmis. C'est cet algorithme qui est utilisé dans notre travail ; il s'agit d'un algorithme de décodage à entrée et à sortie fermes.

Des adaptations de l'algorithme de VITERBI telles que celles proposées dans [3], [8] ou [5] ont conduit aux versions à sortie pondérée dites SOVA.

Il existe un troisième algorithme appelé *Forward-Backwad*, il s'agit d'un algorithme à entrée et sortie souples (SISO) : c'est l'algorithme MAP. L'algorithme MAP permet le calcul de la valeur exacte de la probabilité à postériori associée à chaque symbole transmis en utilisant la séquence reçue [10].

2.2.1 Principe de l'algorithme

Un mot code est un chemin faisant partie du diagramme en treillis, possédant un état de départ et un état d'arrivée.

Le décodage ML est basé sur la recherche du mot de code u qui a la plus petite distance du mot reçu. Il est important de connaître la nature du canal de transmission,

car dans le cas d'un canal binaire symétrique, le décodage à maximum de vraisemblance (ML) s'appuie sur la distance de HAMMING, alors que dans celui d'un canal gaussien, il s'appuie sur la distance euclidienne.

Le décodage consiste à trouver le chemin le plus proche, appartenant au treillis, de la séquence reçue à décoder. Il explore le treillis en comparant tous les chemins possibles avec la séquence reçue et sélectionne celui qui est le plus vraisemblable.

L'algorithme de VITERBI permet d'apporter une réduction notable de la complexité de calcul. Il est basé sur l'idée que, parmi l'ensemble des chemins du treillis qui convergent en un état à un instant donné, seul le chemin le plus probable peut être retenu pour les étapes suivantes de recherche. Cette méthode est appelée stockage des chemins survivants.

Soit b un message code transmis sur un canal BSC avec une probabilité d'erreur p et r le message code correspondant reçu. Supposons que b et r sont de même longueur n et z est la distance de HAMMING entre eux, la fonction log-vraisemblance, $\log \Pr(b \mid r)$, est donnée par la formule suivante [4] :

$$\begin{aligned} \log \Pr(b \mid r) &= \log \left[p^z (1-p)^{n-z} \right] && (2.1) \\ &= n \log(1-p) - z \log \left(\frac{1-p}{p} \right) && (2.2) \end{aligned}$$

ou bien :

$$\log \Pr(b \mid r) = -A - Bz \tag{2.3}$$

Où $p \leq 0.5$, ce qui implique que $A,\ B > 0$.

Donc pour maximiser la fonction log-vraisemblance, il faut minimiser la distance de HAMMING.

Il est important de parler de quelques notions de métriques essentielles au décodage par algorithme de VITERBI : Pour effectuer un décodage par algorithme de VITERBI on a besoin de deux métriques :

Métrique de branche

C'est la distance de HAMMING (cas d'un Canal Binaire Symétrique) entre le symbole reçu (symbole de n bits) et tous les symboles possibles de sortie du codeur. Donc, à chaque instant i, on doit calculer 2^n métriques de branches.

Le symbole de sortie correspond à une transition d'un état a_p à l'instant $i-1$, noté $a_p^{(i-1)}$ vers un état a_q à l'instant i, noté $a_q^{(i)}$. Ainsi la notation de la métrique de branche entre le symbole reçu (affecté par le bruit) et le symbole qui correspond à une transition quelconque est :

$$BM\left(a_p^{(i-1)},\ a_q^{(i)}\right)_{p \text{ prend } 2^k \text{ valeurs différentes}} \tag{2.4}$$

Métrique de chemin (ou métrique d'état)

La métrique de chemin d'un état a_q à l'instant i, notée $PM\left(a_q^{(i)}\right)$ est le cumul des métriques de branches des instants précédents conduisant à cet état tout en respectant le critère des chemins survivants.

Pour un codeur de rendement $R = {}^{k}/_{n}$, on a chaque état $a_q^{(i)}$ généré au maximum par 2^k états précédents, notés $a_p^{(i-1)}$. Donc pour calculer la métrique de chemin $PM\left(a_q^{(i)}\right)$, il faut cumuler les métriques de branches précédentes, les plus petites. Pour ce faire, on applique la loi suivante :

$$PM\left(a_q^{(i)}\right) = \min\left\{PM\left(a_p^{(i-1)}\right) + BM\left(a_p^{(i-1)},\ a_q^{(i)}\right)\right\}_{p \text{ prend } 2^k \text{ valeurs différentes}} \tag{2.5}$$

$PM\left(a_q^{(i)}\right)$: métrique de chemin de l'état a_q à l'instant i ;
$PM\left(a_p^{(i-1)}\right)$: métriques de chemin de l'état a_p à l'instant $i-1$;
$BM\left(a_p^{(i-1)},\ a_q^{(i)}\right)$: métrique de branche de la transition de l'état $a_p^{(i-1)}$ vers l'état $a_q^{(i)}$.

Cette opération décrite dans l'équation (2.5) est appelé ACS, illustrée dans la Figure 2.1 pour un codeur de rendement $R = {}^{1}/_{n}$.

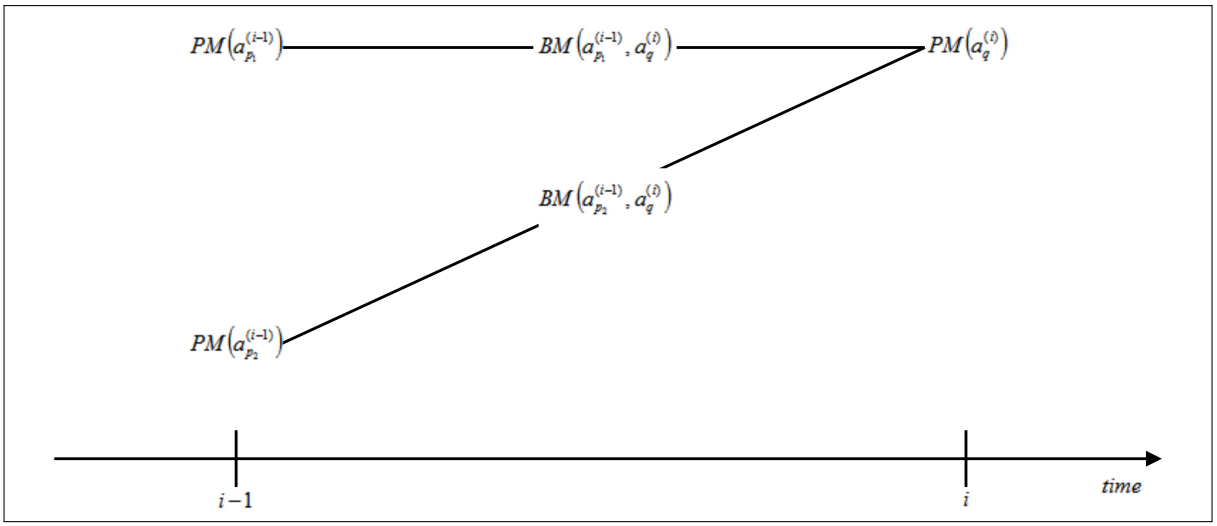


FIGURE 2.1: Illustration de l'opération ACS Taux $R = {}^{1}/_{n}$.

Après avoir vu ces deux définitions des métriques, on va voir étape par étape comment se fait le décodage :

1. Initialisation : Il s'agit d'initialiser le décodeur au même état initial que le codeur ; généralement c'est l'état 0. Il faut aussi attribuer une métrique de chemin nulle à l'état initial.
2. Calcul des métriques de branches : Pour chaque symbole de n bits on calcule les métriques de branches entre le symbole reçu (affecté par le bruit) et les 2^n symboles probable.

3. Calcul des métriques de chemins : Après avoir calculé les métriques de branches, on calcule les métriques de chaque état : On additionne chaque métrique de branche à la métrique d'état précédente. Cet état précédent doit correspondre au même état du début de transition avec laquelle on a calculé la métrique de branche pour avoir la métrique de chemins qui correspond à l'état de la fin de transition.
4. Stockage des survivants : Après l'étape précédente on aura pour chaque état a_q, 2^k métriques de chemin. Parmi ces 2^k métriques de chemins, on doit sélectionner la plus petite et bien sur ce souvenir de la décision appliquée.
5. Décodage : Après avoir exécuté les étapes 2 à 4 pour toutes la séquence reçue (prise symbole par symbole de n bits). On cherche dans la dernière colonne du treillis l'état qui a la métrique de chemins la plus petite. De cet état on remonte le treillis en utilisant les décisions de l'étape 4 jusqu'à l'état initial. Remarquez que la séquence décodée est en inverse, car nous commençons par la fin du treillis.

En résumé pour réaliser le décodage de VITERBI, il faut calculer les métriques de branches puis les accumuler en sélectionnant la métrique de chemin la plus petite pour chaque état et à n'importe quel instant. C'est le stockage des chemins survivants. A la fin il faut sélectionner le chemin qui a la métrique la plus petite, puis remonter le treillis à partir de cet état jusqu'à l'état initial.

2.2.2 Exemple de décodage

Soit la séquence à coder $u = 111001$ par un codeur de longueur de contrainte $L = 4$ (ce qui correspond à une mémoire $m = 3$ et un nombre d'états $N = 2^3 = 8$), un rendement $R = 1/2$ ($k = 1$ et $n = 2$), et les polynômes générateurs $g_1 = 15_8$ et $g_2 = 13_8$, ce qui donne une séquence codée $b = 111010101101$. Ce message b est transmis à travers un canal BSC avec une probabilité d'erreur $p = 0.2$; le message reçu est $r = 111010111101$.

Initialement le codeur et à l'état 000 ; donc il faut que le décodeur démarre du même état avec une métrique de chemin nulle. Le premier symbole reçu est 11 ; il faut donc calculer les métriques de branches entre ce symbole et tous les symboles possibles(c'est-à-dire 00, 01, 10 et 11). Il faut refaire ce même calcul pour les cinq prochaines itérations et pour chaque état, à condition qu'il soit déjà généré, en respectant toujours la minimisation de la métrique de chemins. Les Figures 2.2 à 2.7 illustrent le treillis à chaque instant.

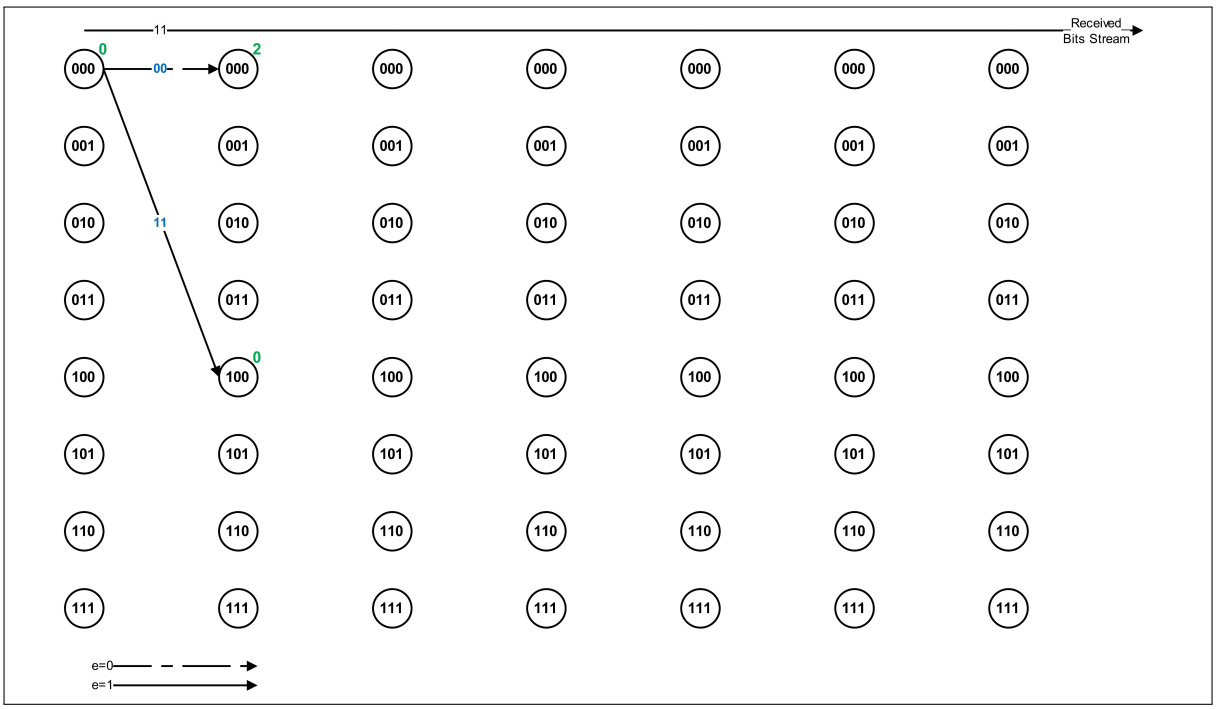


FIGURE 2.2: État du treillis à l'instant $i = 1$.

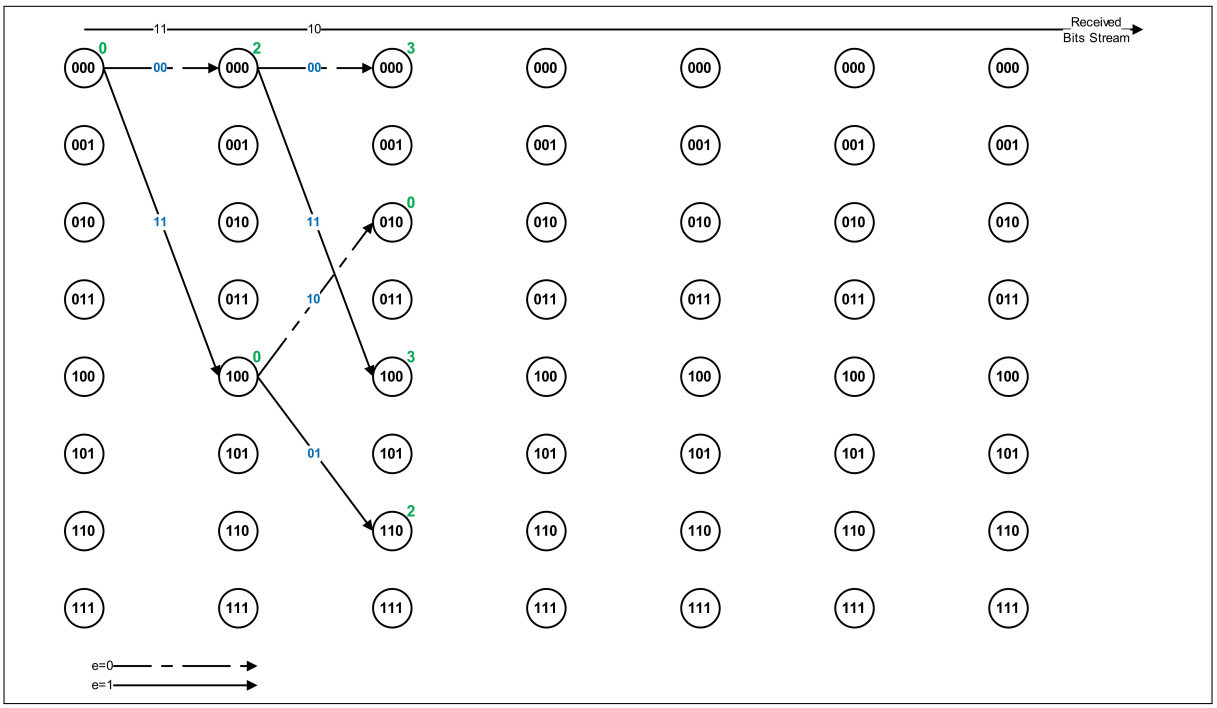


FIGURE 2.3: État du treillis à l'instant $i = 2$.

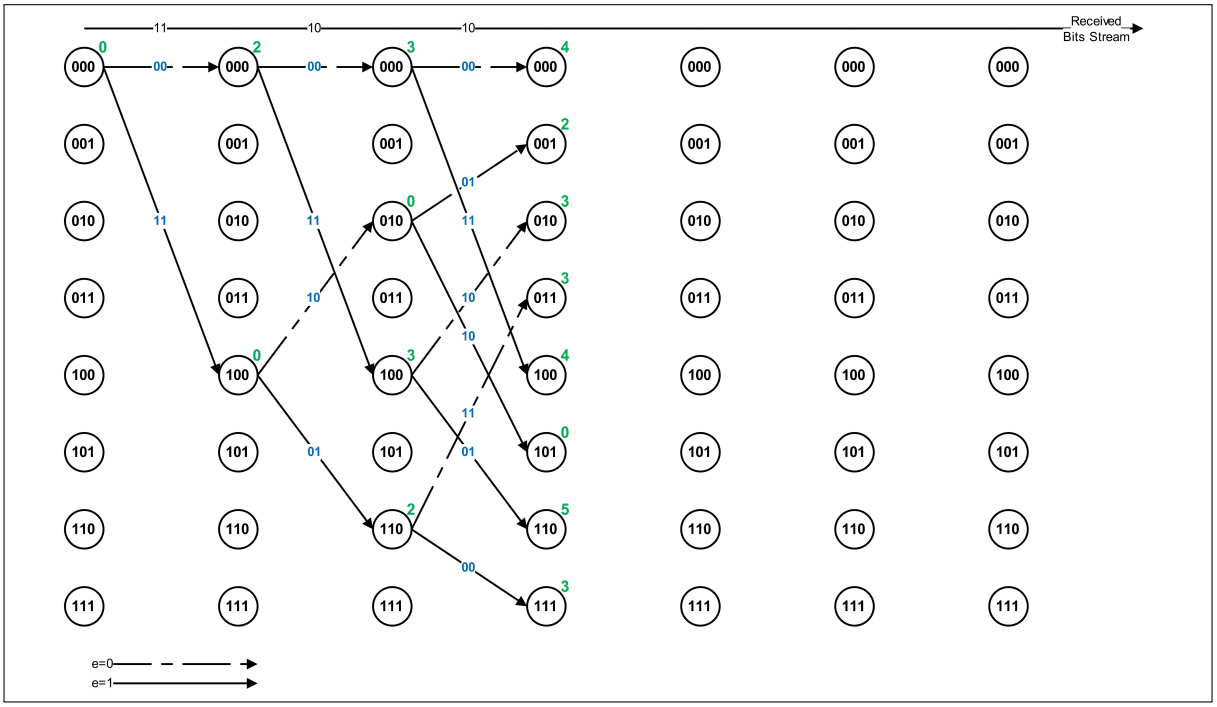


FIGURE 2.4: État du treillis à l'instant $i = 3$.

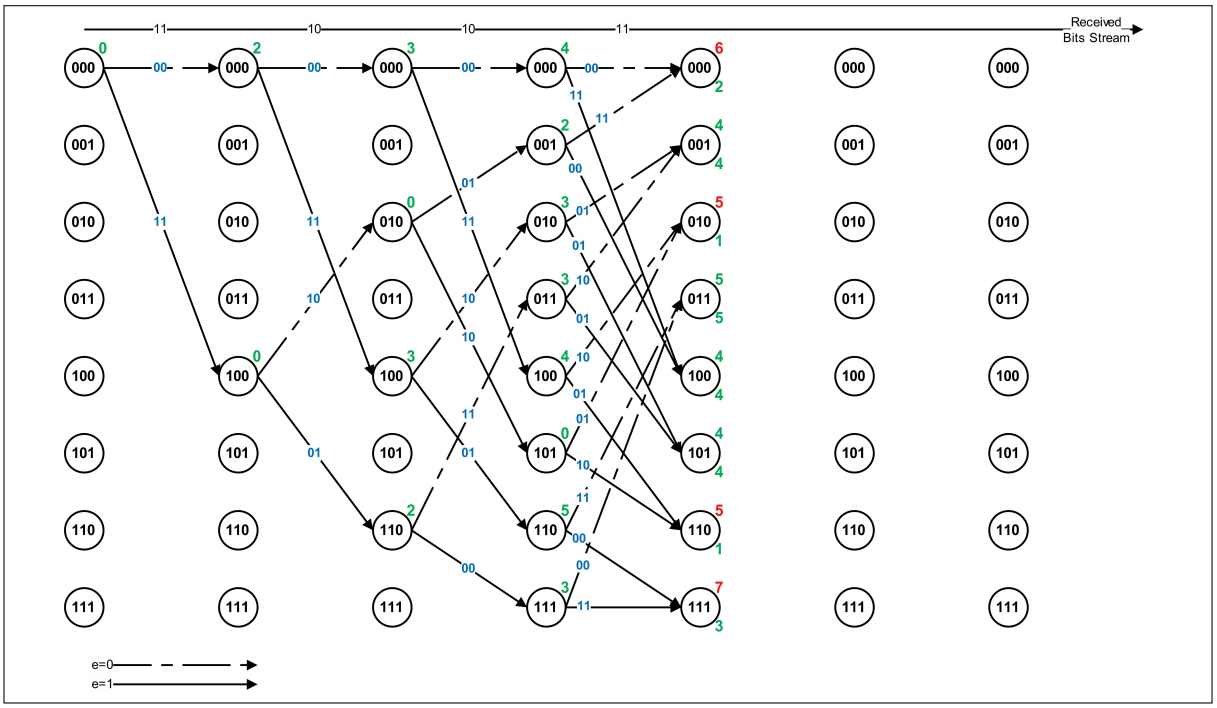


FIGURE 2.5: État du treillis à l'instant $i = 4$.

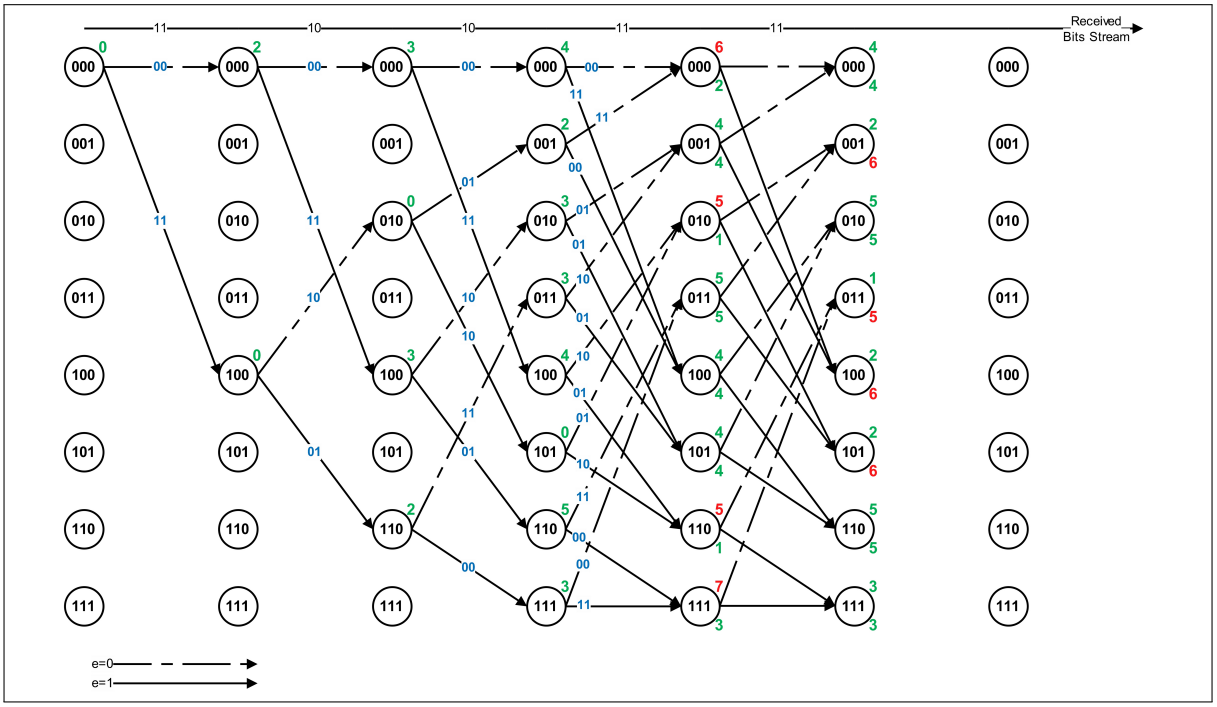


FIGURE 2.6: État du treillis à l'instant $i = 5$.

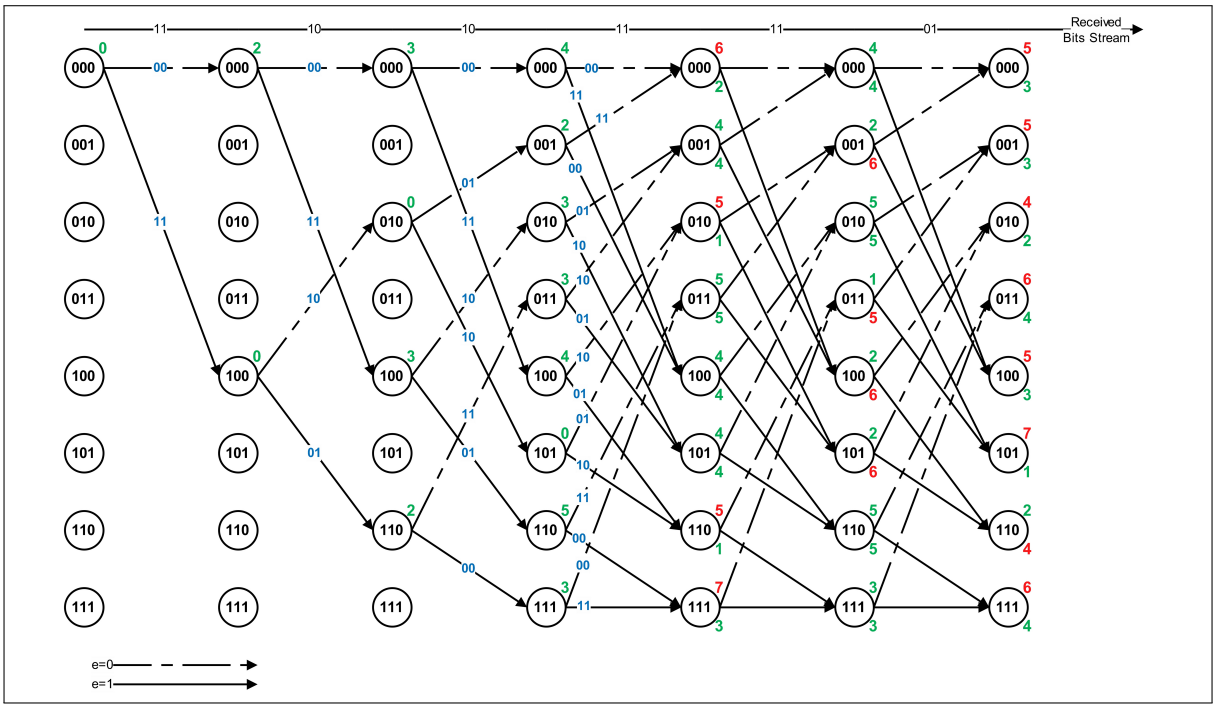


FIGURE 2.7: État du treillis à l'instant $i = 6$.

Après la sixième itération on remonte le treillis en démarrant de l'état qui possède la plus petite métrique de chemin, c'est-à-dire l'état 101. La Figure 2.8 illustre cette opération.

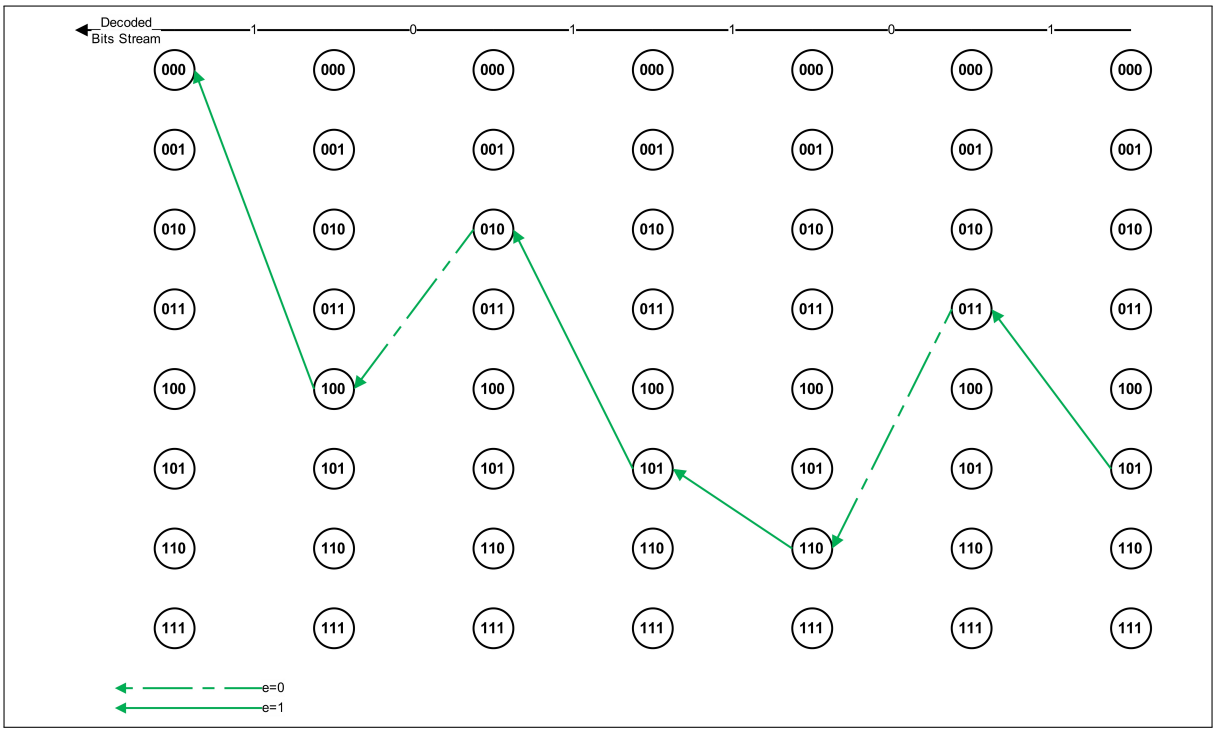


FIGURE 2.8: Décodage en remontant le treillis.

Ainsi la séquence décodée ($\widehat{u} = 101101$) est bien la même que la séquence codée, bien qu'il existe un bit erroné à la huitième position de la séquence codée.

2.3 Schéma fonctionnel du décodeur

Après avoir parlé du principe de l'algorithme de VITERBI, nous allons voir les modules qui constituent le décodeur de VITERBI du point de vue implémentation. Pour réaliser le décodeur de VITERBI il faut souvent implémenter trois modules fonctionnels. Ces trois modules sont représentés dans la Figure 2.9.

D'après la Figure 2.9 on voit bien que le décodeur de VITERBI est un montage en cascade de trois modules :

– **BMU :** C'est l'Unité de Calcul des Métriques de Branches. Elle permet de calculer la métrique de branche c'est-à-dire ; la distance de HAMMING entre le mot code reçu (affecté par le bruit) et le mot code exact correspondant à une transition quelconque du diagramme en treillis. Pour chaque symbole reçu de n bit, elle calcule

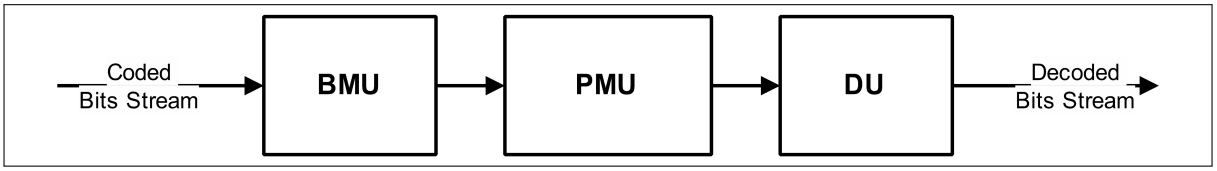


FIGURE 2.9: Illustration des différents modules du VD.

2^n métriques de branches et les fait passer à la PMU pour calculer les métriques de chemins.

- **PMU :** Unité de calcul des Métriques de Chemin, appelée aussi ACSU, constituée principalement des ACS. Ce module additionne les métriques de branches aux précédentes métriques de chemin et sélectionne la métrique la plus petite pour chaque état, puis met à jour les nouvelles métriques. Aussi cette unité délivre à chaque instant i, 2^m bits de décision qui correspondent aux chemins choisis.
- **DU :** C'est l'Unité de Décision. Ce module peut être réalisé par deux approches :
  - *Trace Back* (TB) : Cette approche consiste à remonter le treillis à partir de l'état final qui possède la métrique d'état la plus petite en utilisant les bits de décisions délivrées par la PMU pour tracer le chemin le plus vraisemblable à la séquence reçue.
  - *Register Exchange* (RE) : Cette méthode consiste a allouer pour chaque état un registre qui contient les bits de la séquence décodée à chaque instant. A la fin la séquence décodée se trouve dans le registre qui correspond à l'état qui a la plus petite métrique de chemin. Cette méthode utilise beaucoup de mémoire. Cette mémoire augmente exponentiellement avec la mémoire du code, car pour un codeur de mémoire m, on alloue 2^m registres d'une certaine longueur.

2.3.1 Unité de Calcul des Métriques de Branches

Pour chaque symbole reçu de n bit, la BMU calcule 2^n métriques de branches. Il existe plusieurs types de métriques ; celle qui nous intéresse est la distance de HAMMING.

Distance de Hamming

La distance de HAMMING, définie par RICHARD HAMMING, est utilisée en informatique en traitement du signal et dans les télécommunications. Elle joue un rôle important en théorie algébrique des codes correcteurs. Elle permet de quantifier la différence entre deux séquences de symboles ou tout simplement elle calcule le nombre de bits différents entre deux séquences binaires de même longueur. Cette distance est utilisée dans le cas d'une transmission dans un canal BSC.

Donc pour implémenter la BMU, on doit réaliser une fonction qui calcule la distance de HAMMING entre le symbole reçu et le symbole attendu, stocké dans une ROM. Cette

fonction est réalisée par des XOR et un additionneur. La Figure 2.10 illustre le schéma fonctionnel de la BMU pour $n = 2$.

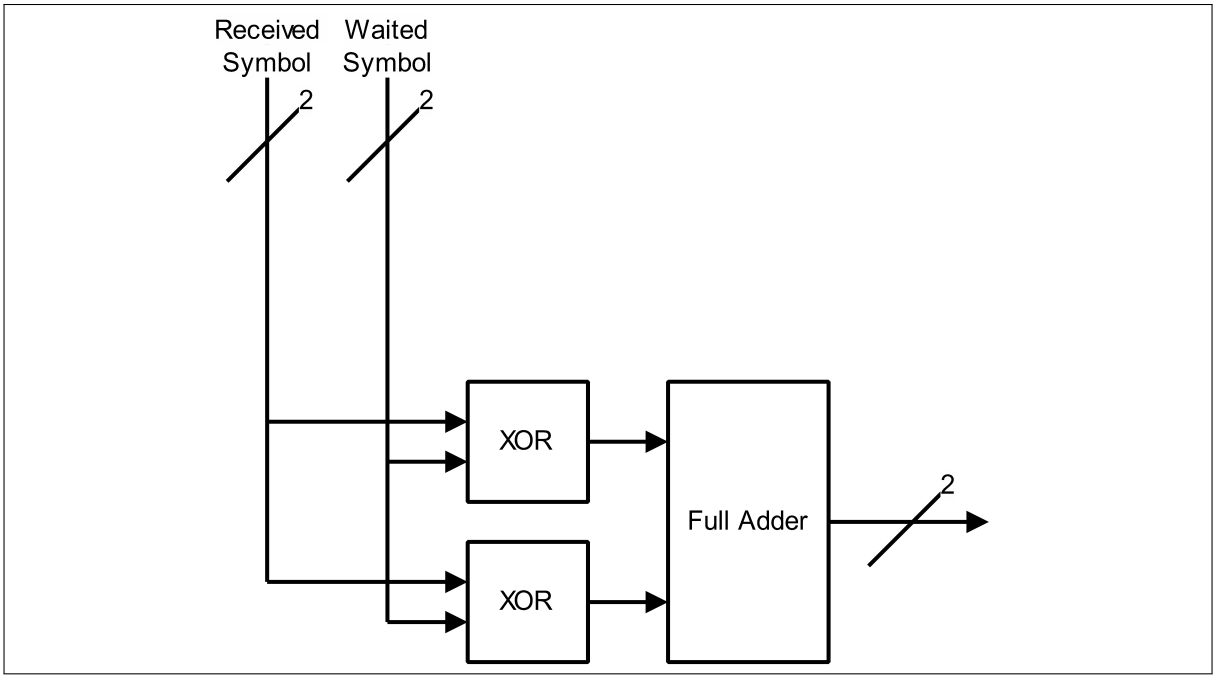


FIGURE 2.10: Illustration de la BMU.

2.3.2 Unité de calcul des Métriques de Chemin

Cette unité calcule et sélectionne le chemin le plus vraisemblable parmi les 2^k chemins entrants dans chaque état. Elle fournit également à chaque instant i, k bits de décisions renseignant nous sur les chemins choisis. Cette principale fonction est réalisée par le sous-module ACS.

L'ACS est constitué de 2^k additionneurs pour calculer toutes les métriques d'état possibles, un comparateur qui sélectionne la position de la métrique d'état la plus petite (sélection des survivants) et un multiplexeur qui fournit en sortie la métrique la plus petite qui sera stockée dans la RAM, pour être utilisé dans la prochaine itération. La Figure 2.11 illustre le schéma bloc de l'opération ASC pour $R = 1/n$.

Donc pour réaliser la PMU (ou ACSU), on doit réaliser toutes les ACS du treillis. Souvent on regroupe les ASC qui possèdent les mêmes états de départ et utilisent les mêmes métriques de branches pour n'en faire qu'un module appelé papillon (*Butterfly*), La Figure 2.12 illustre le regroupement des papillons pour un treillis de $N = 2^3 = 8$, et un $R = 1/2$.

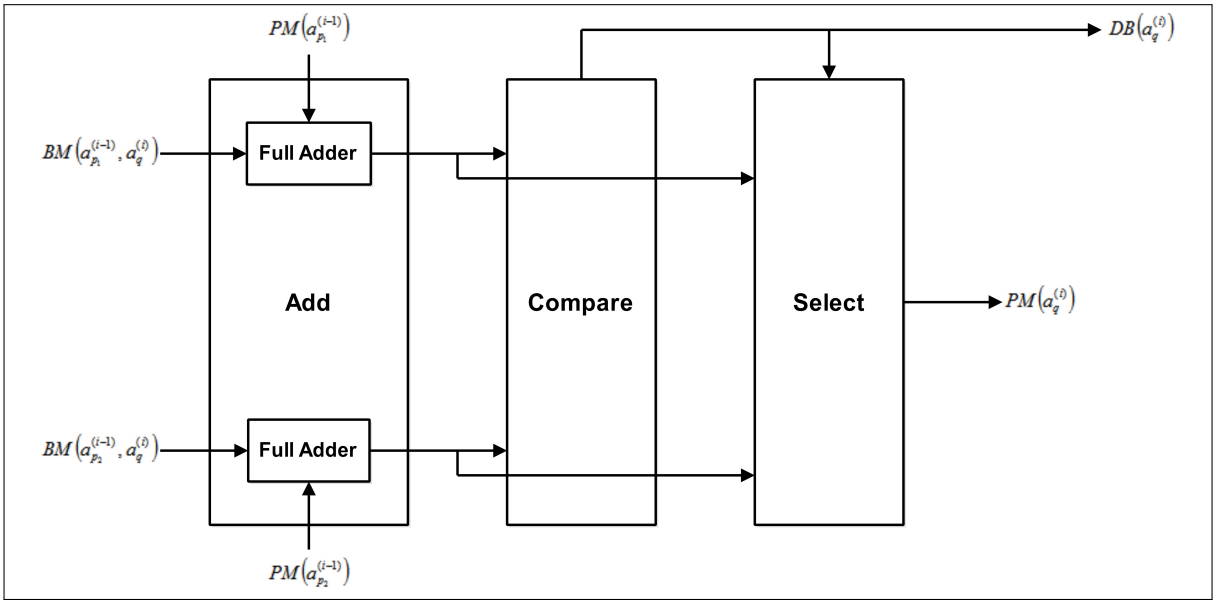


FIGURE 2.11: Schéma de l'ACS pour $R = 1/n$.

2.3.3 Unité de Décision

Dans cette unité on génère la séquence décodée. Dans notre travail, pour cette unité on a choisie l'approche *Trace Back.* Donc pour réaliser la DU on doit implémenter une fonction qui recherche la position du minimum et donne en sortie l'état correspondant, et un module qui n'est autre qu'un registre à décalage (décalage à gauche), qui prend en entrée les bits de décision, et délivre en sortie une séquence de bits, c'est la séquence décodée. Ce registre à décalage doit être initialisé à l'état calculé par la fonction de recherche de du minimum. On remarque dans la Figure 2.8 que la séquence décodée est générée en inverse ; donc on doit ajouter une pile LIFO dans la DU pour inverser la séquence. La Figure 2.13 illustre le schéma bloc de cette unité.

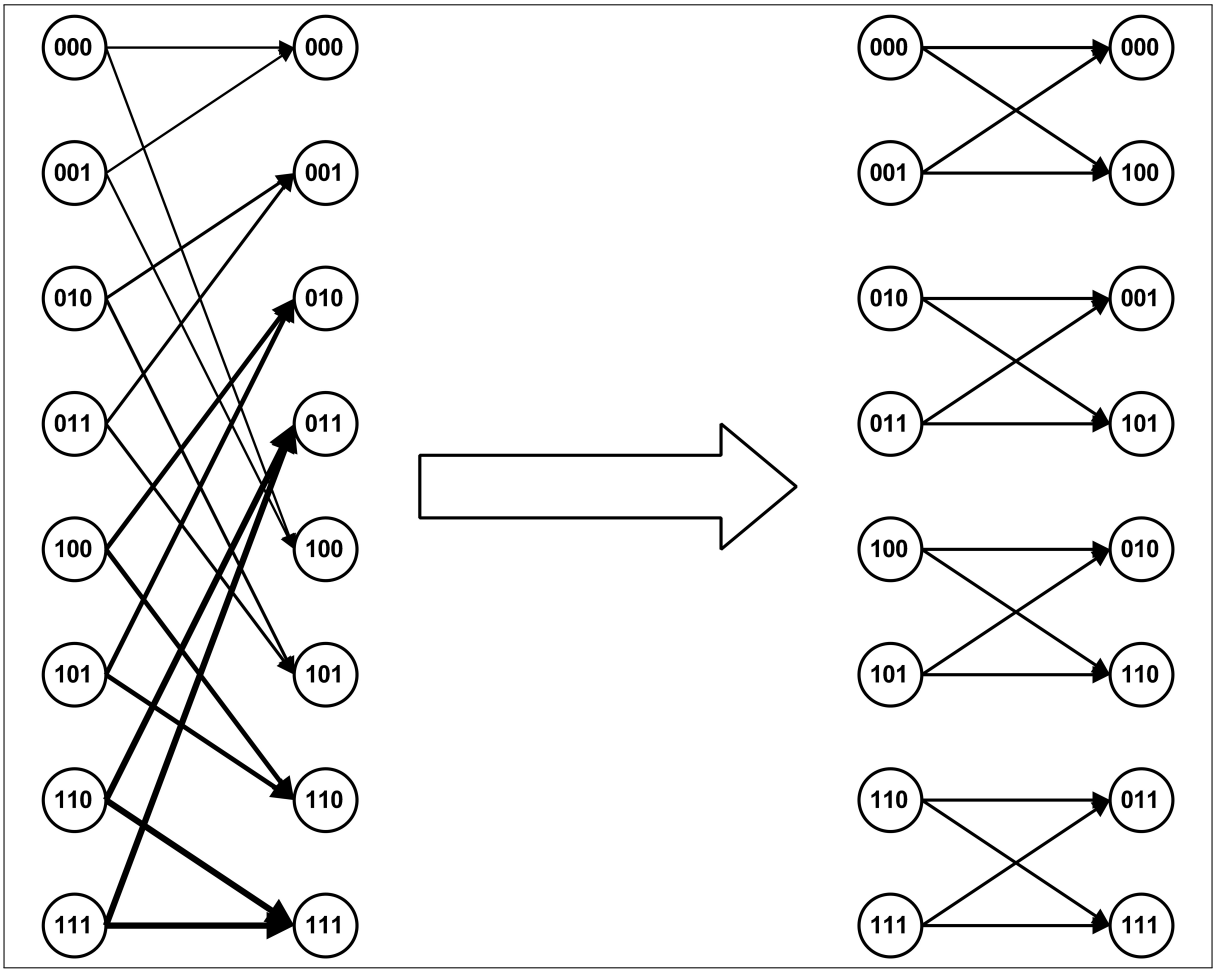


FIGURE 2.12: Regroupement des papillons pour un treillis $N = 8$ et $R = 1/2$

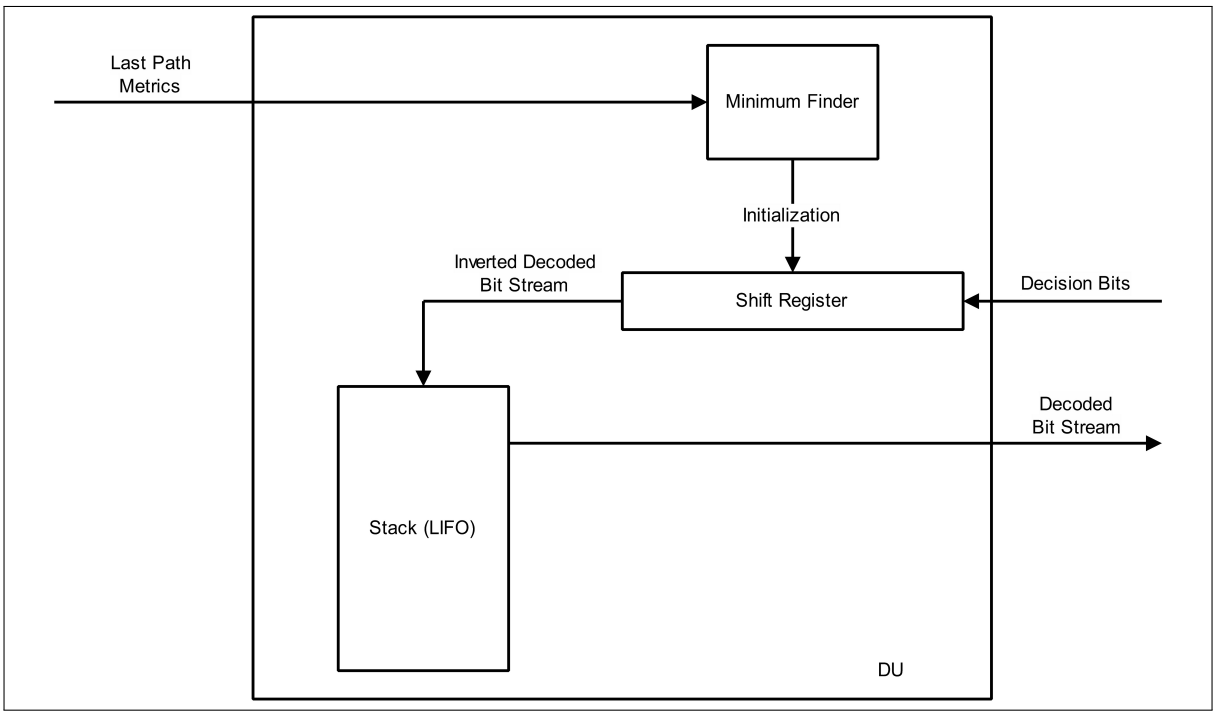


FIGURE 2.13: Illustration de la DU.

Chapitre 3

Architecture proposée

3.1 Introduction

La structure et le principe de fonctionnement du décodeur de VITERBI ont été décrits de façon détaillée dans le chapitre précédent. Dans ce chapitre non seulement nous parlerons de l'architecture proposée, mais nous passerons aussi en détail tous ces composants qui constituent chaque unité en définissant l'unité de contrôle qui synchronise et gère les unités du décodeur.

Dans notre travail, le décodeur est capable de décoder une séquence de bit codée avec un codeur convolutionnel de rendement $R = 1/2$, une mémoire $m = 3$ (ce qui donne un nombre d'état $N = 2^3 = 8$ et une longueur de contrainte $L = 4$) et des polynômes générateurs $g_1 = 15_8$ et $g_2 = 13_8$.

3.2 Schéma proposé du décodeur

L'architecture en bloc du décodeur de VITERBI est souvent celle montrée à la Figure 2.9. Dans notre travail nous avons assemblé la BMU et la PMU pour n'en faire qu'une seule unité appelée Unité de Calcul des Métriques (*Metrics Calculating Unit* ou MCU). La Figure 3.1 illustre le schéma en bloc de notre décodeur.

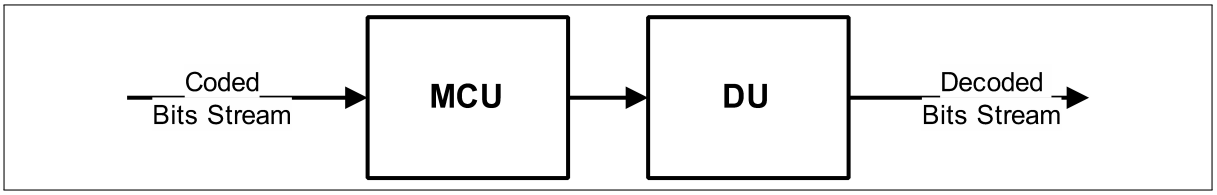


FIGURE 3.1: Schéma bloc du décodeur.

Pour cette architecture l'unité de calcul des métriques de branches est incluse dans

l'unité de calcul des métriques de chemins. Plus précisément le calcul des métriques de branches se fait à l'intérieur des papillons. Chaque papillon peut calculer les deux métriques de branches dont il a besoin.

3.3 Unité de Calcul des Métriques

Cette unité calcule les métriques de branche ainsi que les métriques de chemins et sauvegarde les chemins survivants à chaque instant et pour chaque état en stockant les bits de décision. Aussi doit-elle stocker les métriques de chaque état à chaque instant pour pouvoir les utiliser lors de la prochaine itération. Pour inclure la BMU dans la PMU, qui n'est autre que quatre papillons, il faut que les papillons soient capables de calculer les métriques de branches.

3.3.1 ACS à deux chemins

L'ACS à deux chemins (2-ways ACS) est l'élément de base pour réaliser chaque état du treillis. Sa principale fonction est de calculer la métrique de chemin $PM\left(a_q^{(i)}\right)$ de l'état $a_q^{(i)}$ en partant de deux métriques de chemins $PM\left(a_{p_1}^{(i-1)}\right)$ et $PM\left(a_{p_2}^{(i-1)}\right)$ qui correspondent aux états $a_{p_1}^{(i-1)}$ et $a_{p_2}^{(i-1)}$ respectivement. Donc l'ACS effectue deux additions simultanément. Il additionne une métrique de chemin de l'instant précèdent d'un état quelconque avec la métrique de branche qui correspond à la transition de cet état vers l'état futur. Les deux additions qu'effectue l'ACS sont données par les relations suivantes :

$$NPM_1\left(a_q^{(i)}\right) = PM\left(a_{p_1}^{(i-1)}\right) + BM\left(a_{p_1}^{(i-1)},\ a_q^{(i)}\right) \tag{3.1}$$

$$NPM_2\left(a_q^{(i)}\right) = PM\left(a_{p_2}^{(i-1)}\right) + BM\left(a_{p_2}^{(i-1)},\ a_q^{(i)}\right) \tag{3.2}$$

Après avoir calculé ces deux métriques, l'ACS doit effectuer une comparaison entre elles afin de choisir la plus petite métrique. Aussi, pour sauvegarder les chemins survivants, cette décision doit être stockée dans une mémoire (la RAM de décision qui se trouve dans la DU). La Figure 3.2 est l'organigramme appliqué par l'ACS pour effectuer les trois opérations. Le schéma détaillé de l'ACS est illustré dans la Figure 3.3.

3.3.2 Papillons

Comme nous l'avons décrit précédemment, le papillon est tout simplement la combinaison de deux ACS possédant les mêmes états de départ et le même état d'arrivée. La Figure 3.4 illustre l'architecture d'un papillon dans le cas général.

Notre architecture de papillon est un peu modifiée de l'architecture connue. Avant d'effectuer l'opération ACS, chaque papillon adresse la ROM afin de lire les deux symboles

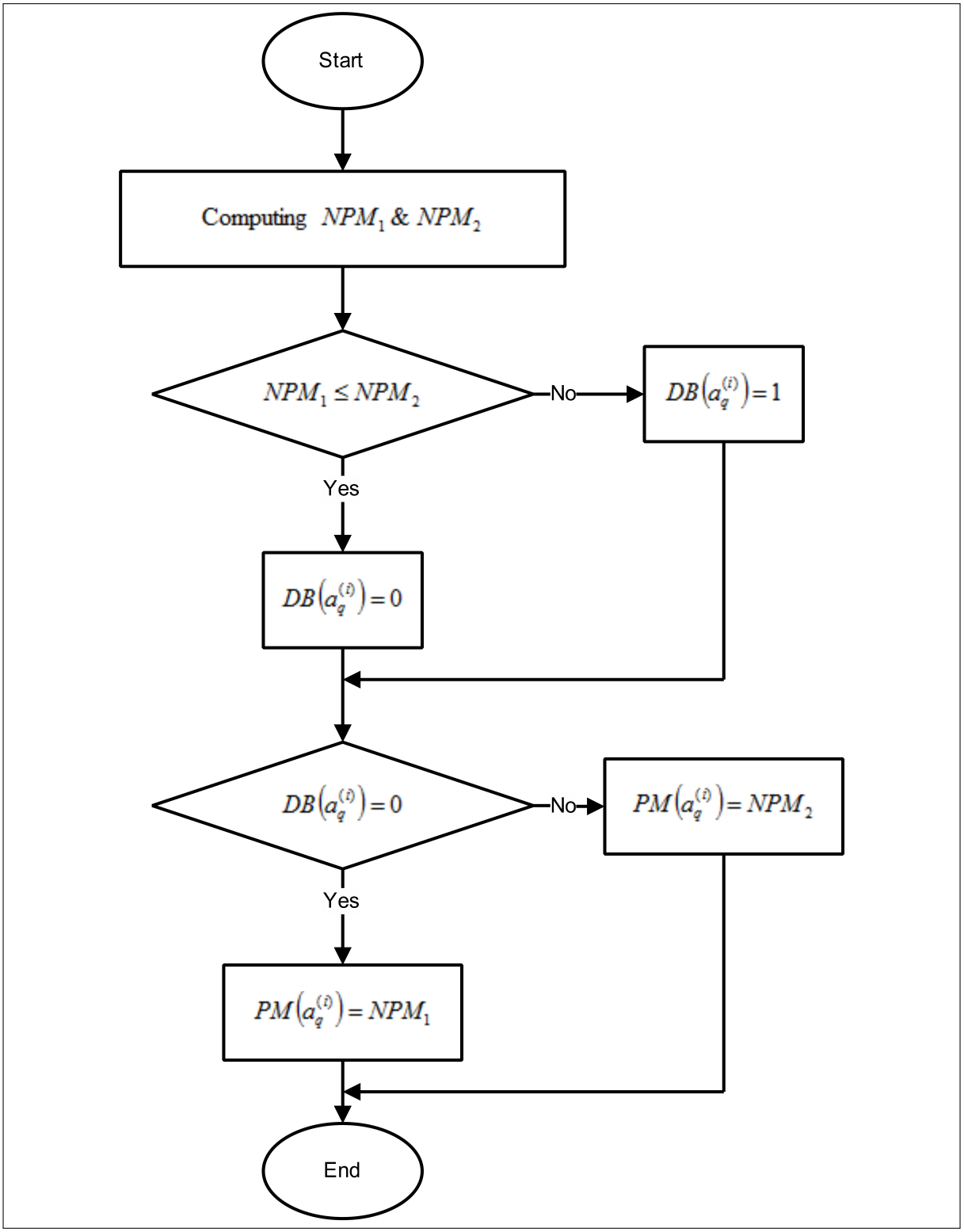


FIGURE 3.2: Organigramme appliqué par l'ACS.

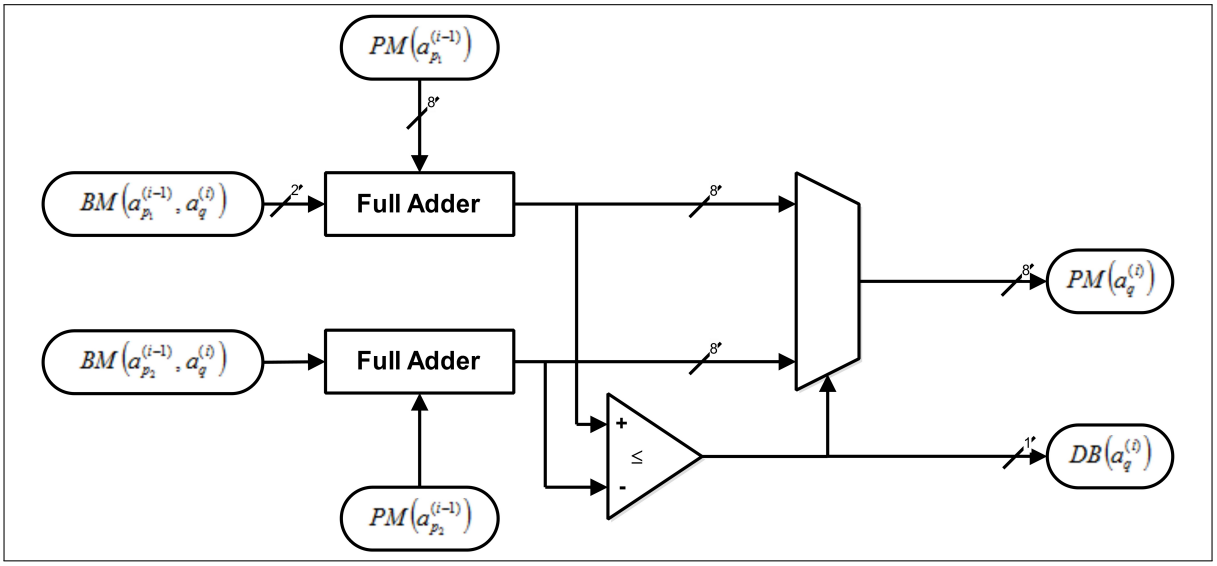


FIGURE 3.3: Schéma détaillé de l'ACS.

attendus. Pour calculer les deux métriques de branches essentielles à l'opération ACS. De ce fait, ce papillon effectue non seulement l'opération ACS, mais calcule au préalable les métriques de branches. La Figure 3.5 illustre l'architecture détaillée du papillon.

En ce qui concerne le fonctionnement des papillons un problème survient lors des premières itérations _exactement $i \leq 2$_ car il existe des états qui ne sont pas encore valides. Le décodeur démarre pour état initial 000, donc tous les autres états ne sont pas valides. Donc le papillon qui effectue le calcul des nouvelles métriques de chemins pour deux états de départ quelconque doit retourner une métrique nulle tant qu'il n'existe pas au moins un état valide à l'entrée de ce même papillon. Pour régler ce problème nous avons eu recours à un registre d'état (*Flag Register*) à l'intérieur de la MCU. Ce registre est de taille 8 bits (un bit pour chaque état). Chacun des bits informe que l'état qui correspond à sa position est actif ou non. Donc l'opération ACS s'effectue seulement sur les états qui sont valides, pour ce faire on doit tester les bits du registre à chaque symbole reçu.

3.3.3 Registre d'état

Après avoir parlé d'un seul élément qui constitue la MCU, nous allons examiner le rôle et l'interconnexion du registre d'état. Après l'utilisation du registre d'état, le multiplexeur de la Figure 3.2.1 doit être modifié pour faire une sélection non seulement avec le bit issu du comparateur (bit de comparaison) mais aussi avec les bits flag. Comme nous l'avons déjà cité avant, un état à l'instant i est généré au maximum par deux états de l'instant $i-1$. Il se peut que l'un de ces deux états ne soit pas valide, donc même le bit de décision va être généré en fonction des bits flags. Le Tableau 3.1 résume toutes

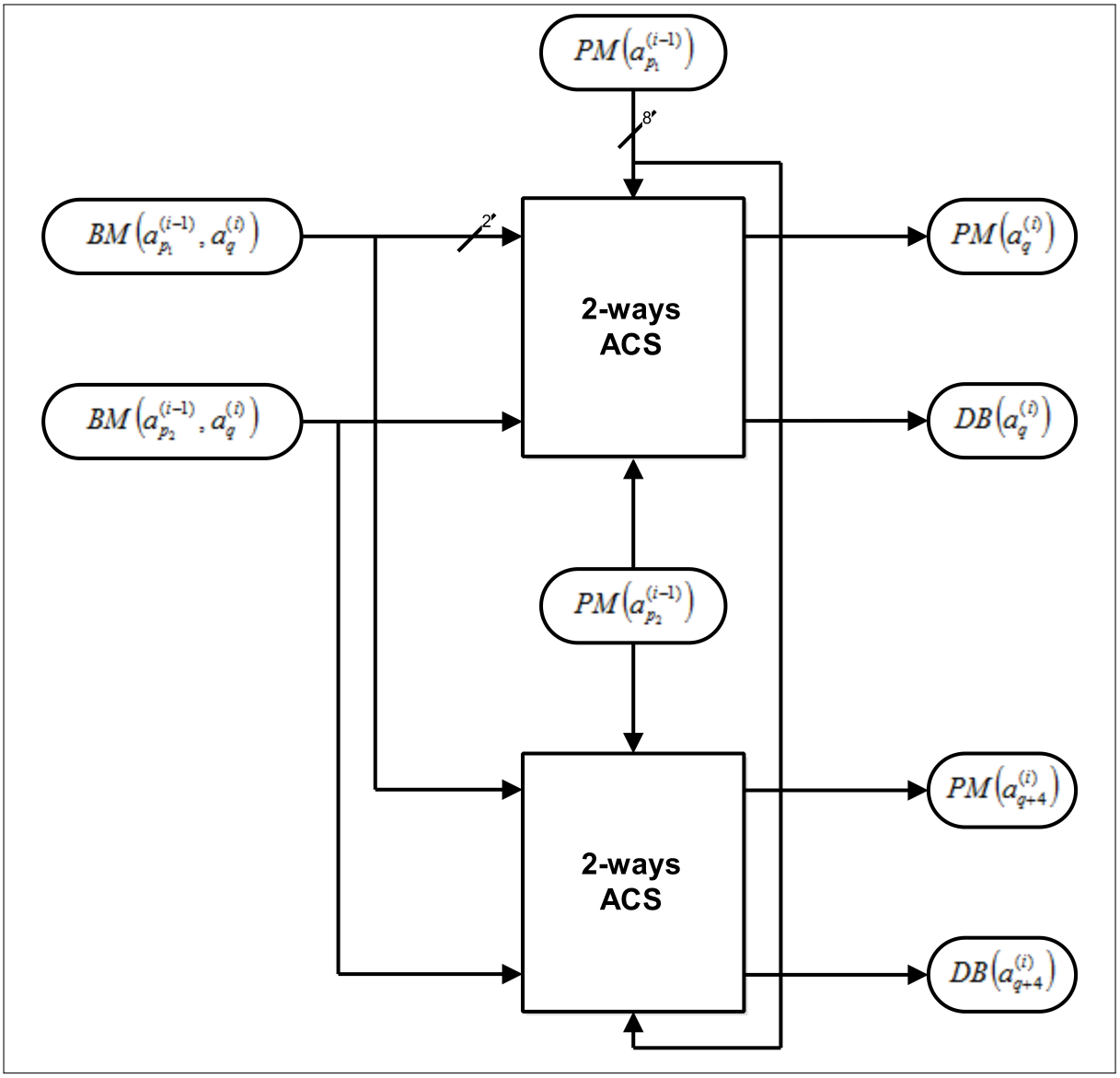


FIGURE 3.4: Architecture d'un papillon.

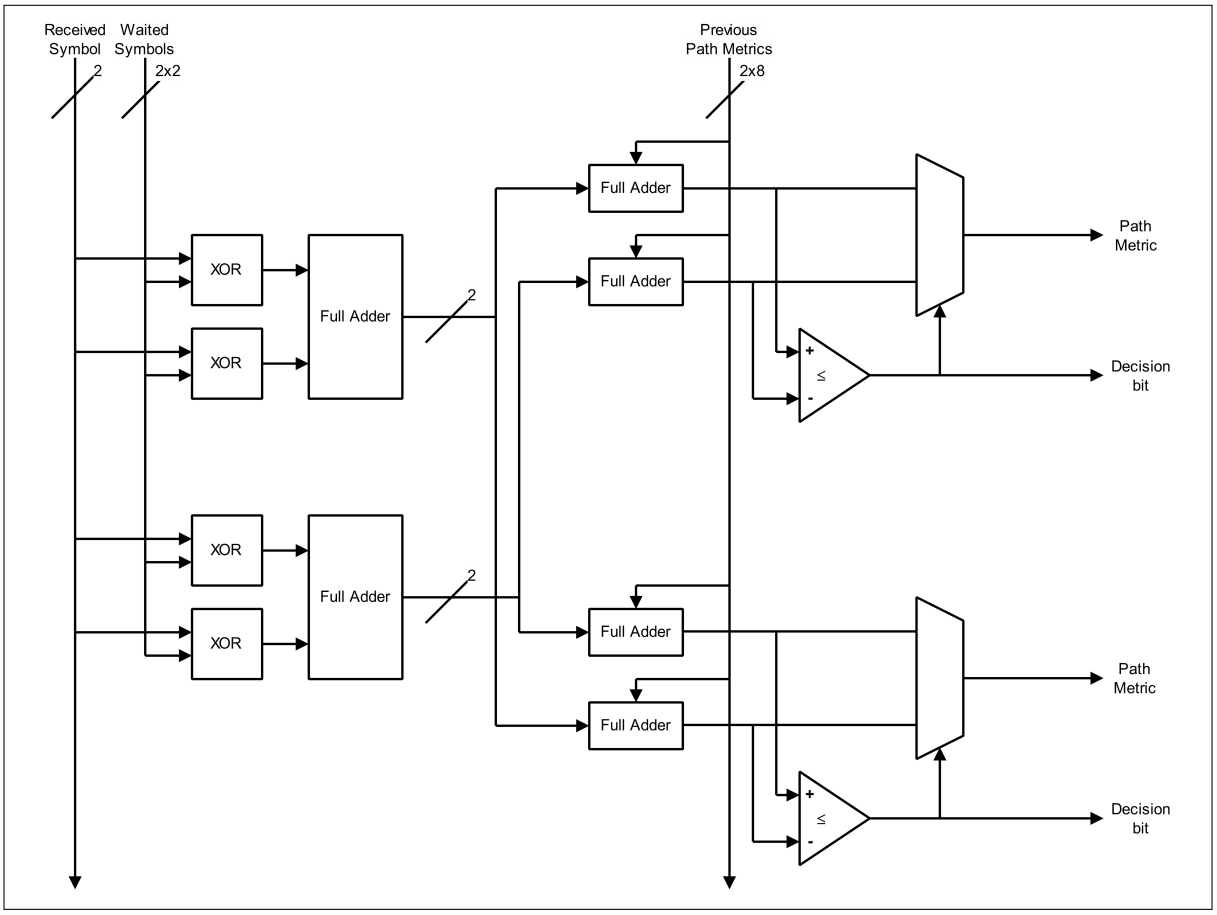


FIGURE 3.5: Architecture détaillée du papillon.

les possibilités et les résultats pour une seule opération ACS. Nous devons aussi ajouter un élément dans la MCU qui, en fonction des bits d'état, génère un bit d'état informant sur la validité de l'état généré.

$Flag\left(a_{p_1}^{(i-1)}\right)$	$Flag\left(a_{p_2}^{(i-1)}\right)$	Bit de comparaison	$PM\left(a_q^{(i)}\right)$	$DB\left(a_q^{(i)}\right)$	$Flag\left(a_q^{(i)}\right)$
0	0	X	0	0	0
1	0	X	NPM_1	0	1
0	1	X	NPM_2	1	1
1	1	0	NPM_1	0	1
1	1	1	NPM_2	1	1

TABLE 3.1: Résultats pour une seule opération ACS

$Flag\left(a_{p_1}^{(i-1)}\right)$: bits informant que l'état $a_{p_1}^{(i-1)}$ est valide ou non ;
$Flag\left(a_{p_2}^{(i-1)}\right)$: bits informant que l'état $a_{p_2}^{(i-1)}$ est valide ou non ;
$PM\left(a_q^{(i)}\right)$: métrique de chemin pour l'état $a_q^{(i)}$;
NPM_1 : New Path Metric 1 ;

$$NPM_1\left(a_q^{(i)}\right) = PM\left(a_{p_1}^{(i-1)}\right) + BM\left(a_{p_1}^{(i-1)},\ a_q^{(i)}\right) \tag{3.3}$$

NPM_2 : New Path Metric 2 ;

$$NPM_2\left(a_q^{(i)}\right) = PM\left(a_{p_2}^{(i-1)}\right) + BM\left(a_{p_2}^{(i-1)},\ a_q^{(i)}\right) \tag{3.4}$$

$DB\left(a_q^{(i)}\right)$: bit de décision pour l'état $a_q^{(i)}$;
$Flag\left(a_q^{(i)}\right)$: bit informant que l'état $a_q^{(i)}$ sera valide ou non.
La Figure 3.6 illustre toutes les modifications apportées à l'unité de calcul des métriques pour un seul éléments ACS.

3.3.4 Architecture de la MCU

Dans cette section nous assemblerons les quatre papillons, la mémoire de stockage des métriques de chemins et le registre d'état pour avoir l'architecture finale de l'unité de calcul des métriques. Tout ceci est illustré dans la Figure 3.7.

Il y'a lieu de noter que pour des contraintes de mémoire nous devons limiter la taille des métriques de chemins. Dans notre travail nous avons attribué une taille de 8 bits pour les métriques de chemins, ce qui donne une métrique maximale égale à 255. Cette métrique maximale est le cumul de 127 fois environ la métrique de branche maximale ($255 \simeq 127 \times 2$), ce qui donne une longueur minimale de 127 bits à transmettre.

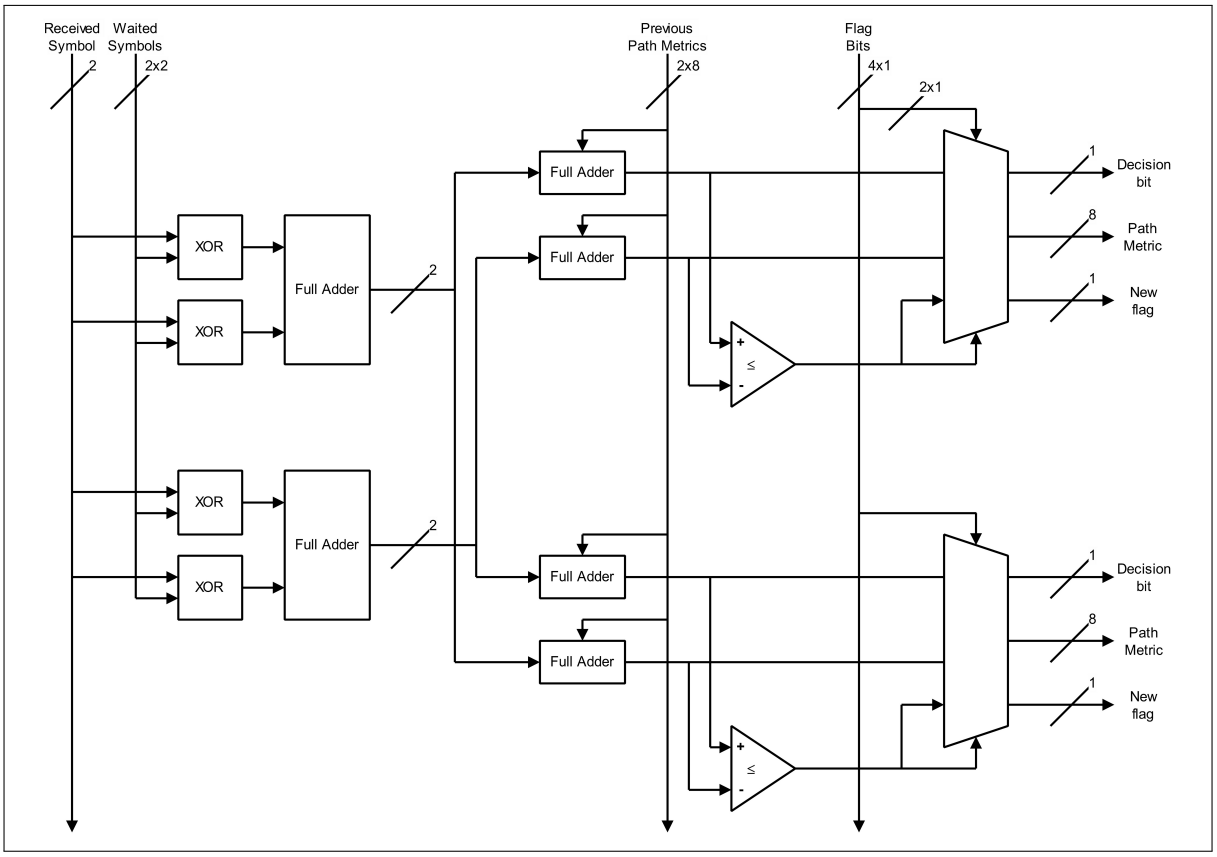


FIGURE 3.6: Unité de calcul des métriques modifiée (illustration pour un seul papillon).

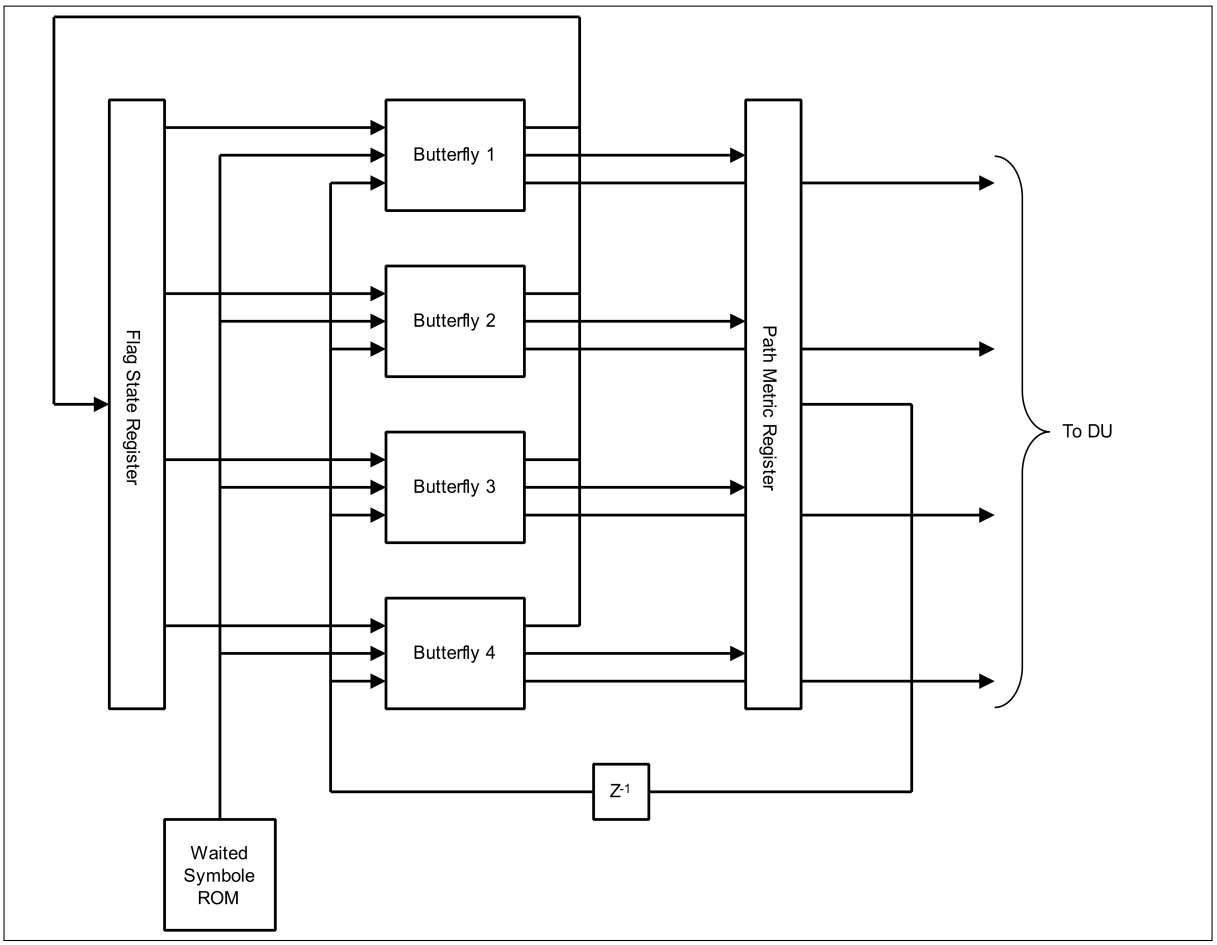


FIGURE 3.7: Architecture finale de la MCU.

3.4 Unité de décision

Cette unité utilise les bits de décision générés par l'unité de calcul des métriques pour décoder la séquence reçue. Il y'a lieu de noter que cette unité ne travaille pas en continu, mais seulement après avoir reçu un nombre de symboles de n bits bien déterminé : C'est la longueur ou la profondeur du *Trace Back* notée L_{TB}. Le décodeur de VITERBI reçoit un flux de symbole continu et il génère un flux de bits de décision qui est aussi continu ; donc on doit diviser le flux des décisions en paquets de longueur $L_{TB} \times N$ qui seront stockés dans la mémoire du TB pour être utilisés une fois le décodage lancé.

3.4.1 Organisation de la mémoire du Trace Back

En ce qui concerne la mémoire du TB, nous ne pouvons utiliser qu'une seule mémoire (RAM) de taille $L_{TB} \times N$ bit, mais un problème se posera sans doute :

Il est clair que le temps du décodage est beaucoup plus grand que le temps nécessaire pour générer une décision : De ce fait, si nous utilisons une seule mémoire, les décisions essentielles, au décodage courant, seront écrasées par les nouvelles décisions des symboles reçus au moment du décodage.

Pour régler ce problème nous avons utilisé deux mémoires de taille $n \times L_{TB}$. La particularité de ces deux mémoires, c'est quand l'une d'elles est en écriture (pour sauvegarder les nouvelles décisions) la deuxième est en lecture (pour servir au décodage courant), sauf pour le premier paquet reçu, la deuxième mémoire sera en mode repos. La Figure 3.8 illustre la mémoire utilisée.

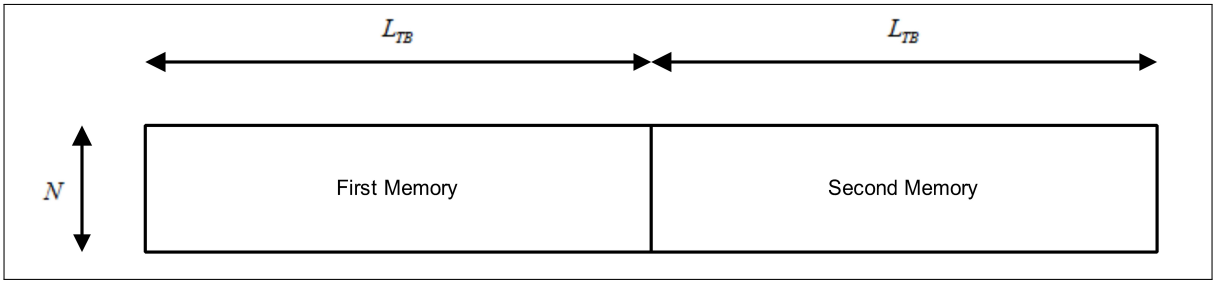


FIGURE 3.8: Mémoire du Trace Back.

Pour bien comprendre le cheminement du flux des décisions dans la mémoire du *Trace Back*, nous décrivons ci-après le contenu de la mémoire pour une longueur de données $3L_{TB}$ en trois étapes :

1. $0 \leq i < L_{TB}$:
   Pour chaque symbole reçu on génère 8 bits de décision (8 états). Ce flux de bits sera stocké dans la première mémoire tandis que la deuxième sera en repos (*Wait*) ; (voir Figure 3.9).

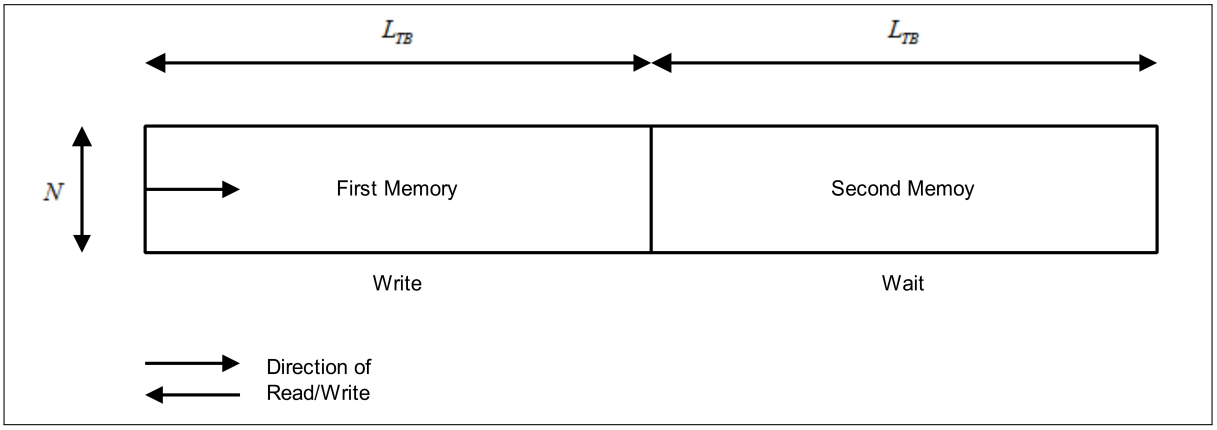


FIGURE 3.9: État de la mémoire du TB pour $0 \leq i < L_{TB}$.

2. $L_{TB} \leq i < 2L_{TB}$:
   La première mémoire est pleine, donc le processus de décodage est lancé. Le décodeur reçoit toujours des symboles et génère des bits de décision qui seront stockés dans la deuxième mémoire. La première mémoire est maintenant en lecture ; (voir Figure 3.10).

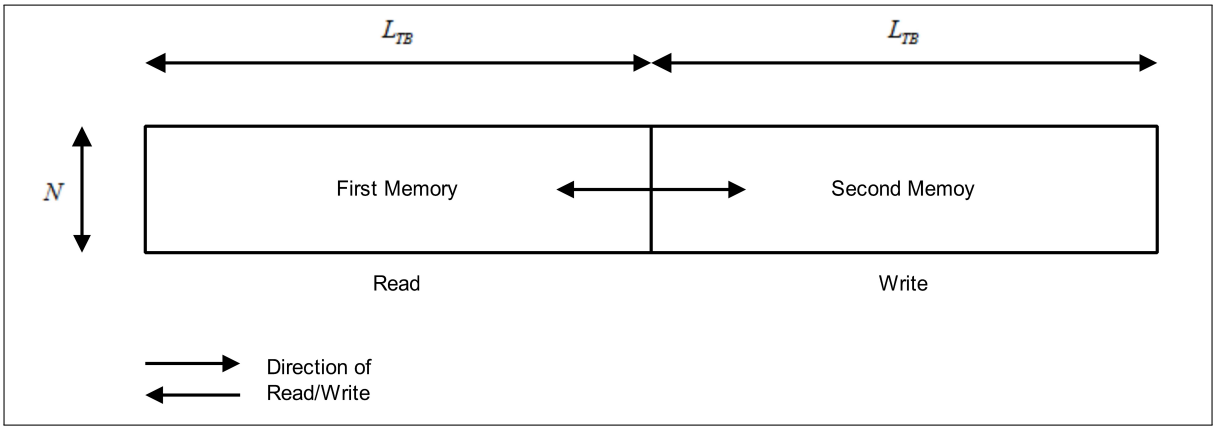


FIGURE 3.10: État de la mémoire du TB pour $L_{TB} \leq i < 2L_{TB}$.

3. $2L_{TB} \leq i < 3L_{TB}$:
   Cette fois c'est la deuxième mémoire qui contient les bits de décision, donc le processus de décodage ; une fois lancé ; utilise cette RAM. La première RAM est en écriture pour sauvegarder les décisions du troisième paquet de données (voir Figure 3.11).

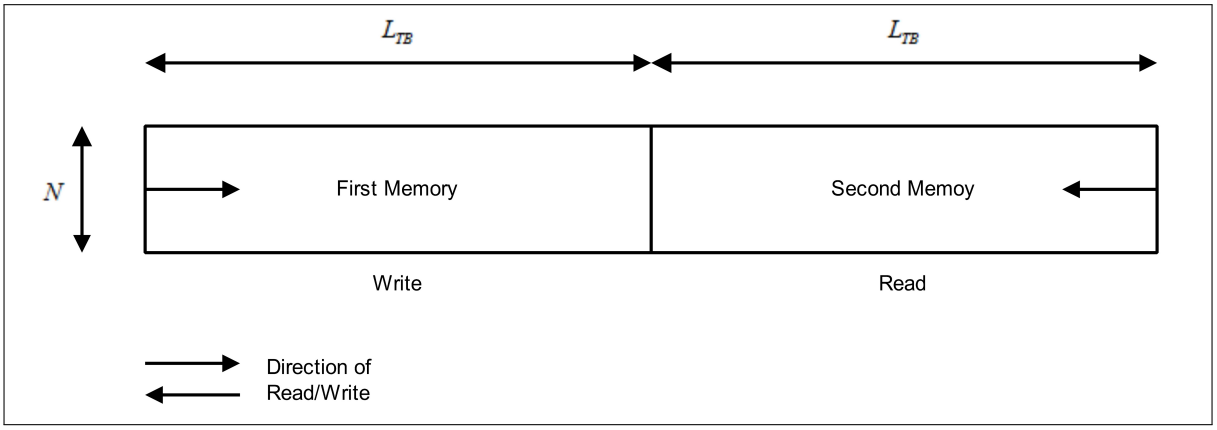


FIGURE 3.11: État de la mémoire du TB pour $2L_{TB} \leq i < 3L_{TB}$.

3.4.2 Fonction de recherche du minimum

Cette fonction a pour but de recherche la position du minimum dans le registre de stockage des métriques de chemin. Puis initialiser le registre du *Trace Back* avec cette position. Pour réaliser cette fonction on applique l'organigramme de la Figure 3.12.

3.4.3 Module *Trace Back*

Cet élément et la pièce maitresse du décodage. Il comporte le registre du *Trace Back* (registre à décalage gauche), et un compteur à décrémentation (de $L_{TB} - 1$ à 0) pour parcourir tous les paquets de décision.

Le registre est initialisé par la fonction de recherche du minimum. En combinant le contenu du registre avec le compteur du *Trace Back* on calcule l'adresse avec laquelle on doit adresser la mémoire du *Trace Back* (celle qui est en mode lecture) pour avoir le bit de décision. Ce bit de décision sera l'entrée du registre TB, et ça sortie c'est le bit de la séquence décodée. A chaque instant on stocke le bit de sortie dans la pile LIFO. Une fois le Trace Back terminé on doit décharger la pile pour avoir la séquence décodée dans l'ordre. La Figure 3.13 est le schéma détaillé du module TB.

3.4.4 Architecture de DU

Après avoir exposer l'architecture de chaque module constituant la DU, nous les assemblerons pour n'en faire qu'une unité : l'Unité de Décision. La Figure 3.14 illustre l'architecture finale de la DU.

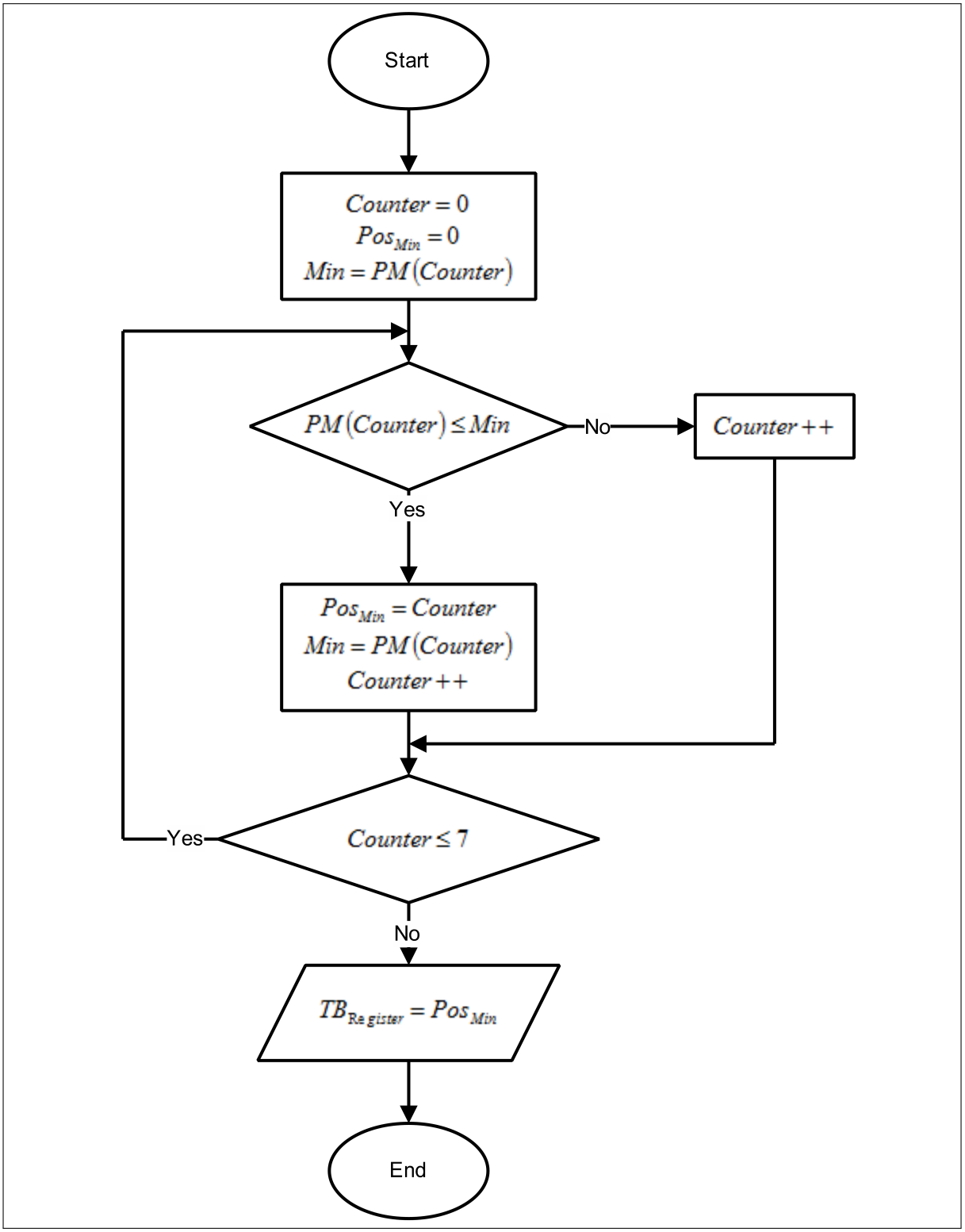


FIGURE 3.12: Organigramme de la fonction de recherche du minimum.

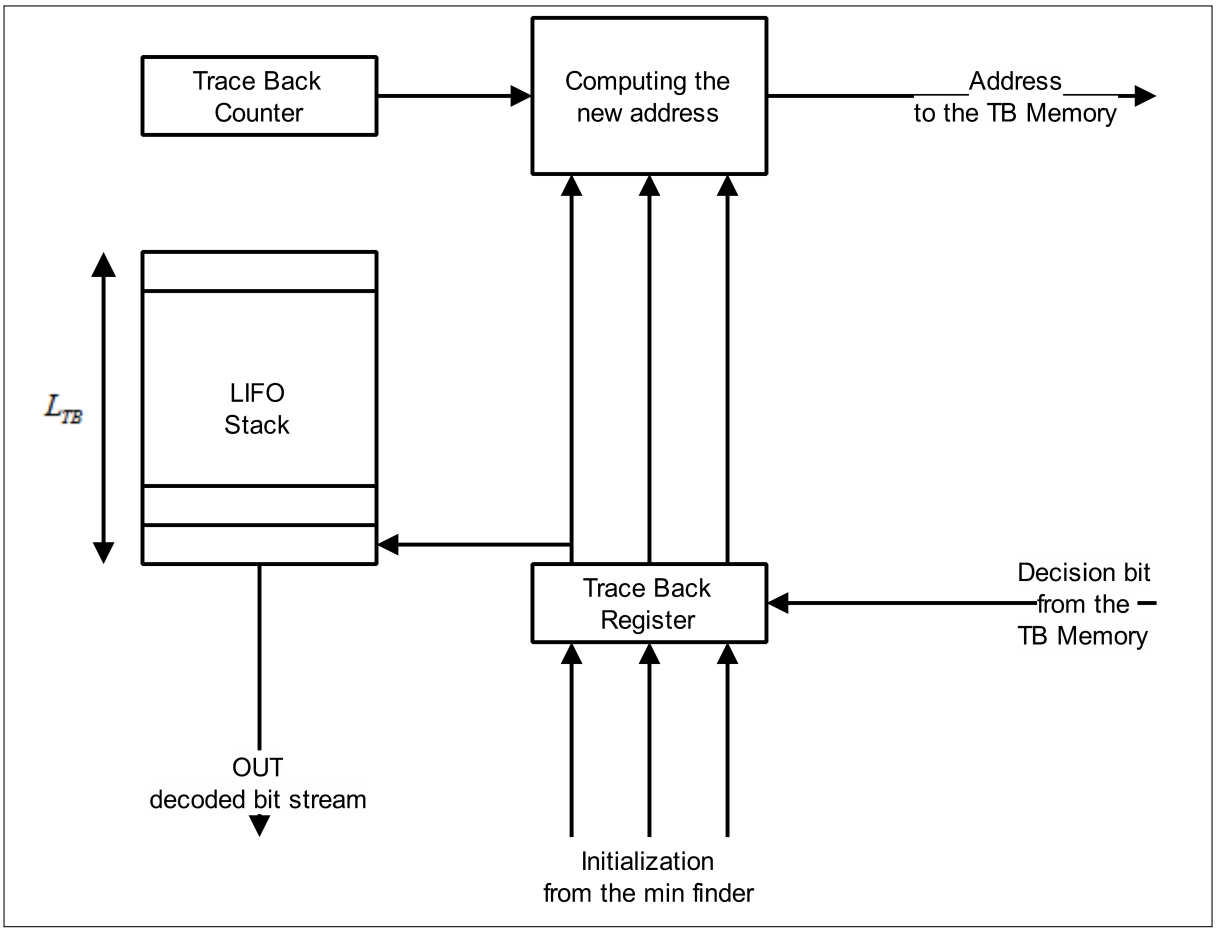


FIGURE 3.13: Schéma détaillé du module TB.

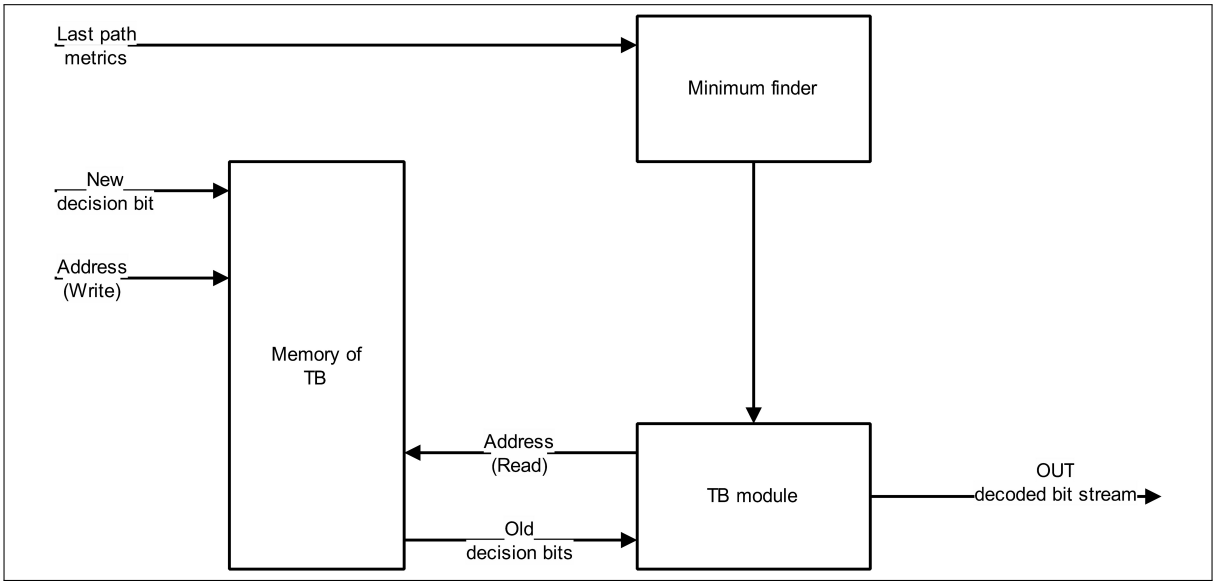


FIGURE 3.14: Architecture finale de la DU.

3.5 Circuit de contrôle

Dans un système logique, l'unité de contrôle ou séquenceur commande et contrôle le fonctionnement d'un système. Une unité de contrôle est un circuit logique séquentiel qui réalise un automate fini, plus précisément une machine de MOORE ou de MEALY, qui génère des signaux de contrôle pour piloter les différents modules de traitement de la donnée.

3.5.1 Synchronisation

Afin de faciliter la synchronisation nous avons décidé de faire transiter toute information d'adressage ou de signaux de contrôle par le circuit principale. Ainsi lorsque les papillons sont prêts à envoyer les métriques de chemin calculées à la RAM des métriques, ils active leur sortie *ready* et cette sortie est considérée par la machine d'état comme une condition de passage à l'étape suivante qui est l'écriture en RAM de ces mêmes métriques de chemin. Une horloge principale permet donc de gérer chacune des horloges internes des différents blocs. Ainsi l'horloge et les Flags donnent la cadence à laquelle se font les opérations.

Des entrée asynchrones permettent de remettre à zéro tous les blocs à n'importe quel moment et ce même si un processus de décodage est en cours.

3.5.2 Chemin de données

C'est aussi le circuit de contrôle qui définie le chemin que suit la donnée dans l'architecture du décodeur. En effet c'est dans cette partie que sont définies les connexions entre différents blocs de traitement.

3.5.3 Le circuit de contrôle du Trace Back

Le processus de Trace Back constitue lui même un circuit de contrôle partiel, synchronisant uniquement les deux blocs suivants :

- Le registre à décalage
- La pile LIFO

Mais au niveau du circuit de contrôle global du décodeur de VITERBI, cette unité ne constitue rien d'autre qu'un composant qui constitue son architecture.

Chapitre 4

Simulations, tests et résultats

4.1 Introduction

A la constante évolution des applications vers des systèmes miniaturisés à hautes performances, les systèmes reconfigurables apportent une réelle réponse s'appuyant sur le développement d'architectures parallèles adaptées.

Dans ce chapitre nous détaillons le décodeur Viterbi développé pour un premier temps sur MATLAB, où nous exposons les différents résultats de la simulation. Ensuite nous explorons la possibilité d'implémentation du décodeur sur circuit FPGA afin de mettre à profit leurs avantages pour envisager l'utilisation en parallèle des quatre papillons lors du calcul des métriques pour accélérer le processus.

4.2 Présentation de la carte FPGA

Le circuit FPGA utilisé est un XC2V3000 4FG676 de la famille Virtex II de Xilinx, il est intégré autour d'une carte multimédia Celoxica RC203E.

Architecture du Virtex II

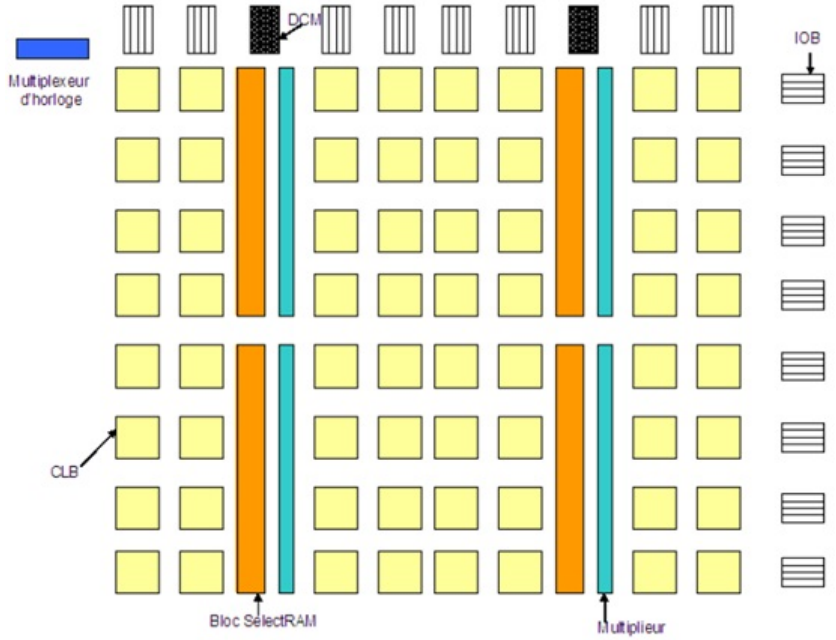
Le FPGA Virtex 2 est principalement composé d'une matrice de CLBs. Chaque bloc configurable est composé de 4 éléments identiques appelés slices et de deux buffers. Les slices sont identiques et contiennent :

- Deux générateurs de fonctions (composés de LUTs à quatre entrées).
- Deux éléments de mémorisation (bascules D).
- Des portes logiques.
- Des multiplexeurs.
- Une chaîne de cascade horizontale (portes OR).

Chaque bloc est connecté à une matrice de commutation pour accéder aux ressources de routage général.

Les blocs mémoires sont de 18kb assignables de 16k * 1bit à 512*36bits. Chaque mémoire possède deux ports synchrones et indépendants l'un de l'autre. A chaque bloc mémoire est associé un multiplieur 18*18 optimisé pour des opérations sur le bloc mémoire. Chaque multiplieur et bloc mémoire est relié à quatre matrices de commutation pour accéder aux ressources de routage.

Les blocs DCM (Digital Clock Multiplexer) et le multiplexeur d'horloge global fournissent des fonctionnalités d'horloges évoluées. Au total il y a 12 DCM sur l'ensemble du FPGA.



4.3 Simulation sous MATLAB®

Pour la simulation MATLAB® nous l'avons partagé en deux parties :

- Un décodeur de paramètres fixes :
  Il s'agit du décodeur choisi pour notre travail. Le décodeur en question peut décoder un flux de bits codés avec un codeur convolutionnel de mémoire $m = 3$, un rendement $R = 1/2$ et des polynômes générateurs $g_1 = 13_8$ et $g_2 = 15_8$. Concernant l'approche de décodage nous avons programmé les deux méthodes (*Trace Back* et *Register Exchange*). Pour chaque approche, nous avons divisé le programme en deux parties :
  - Programme principal « Viterbi_TB.m ou Viterbi_RE.m » : Il contient le codeur convolutionnel, un générateur d'erreurs et l'unité de décision (Voir Annexe A.1.1.5 ou A.1.2.5).
  - Papillons : Pour effectuer le calcul des métriques. Nous avons programmé le quatre papillons séparément (Voire Annexe A.1.1.1 à A.1.1.4).

– Un décodeur paramétrable par l'utilisateur :
  Pour ce décodeur une liberté est offerte à l'utilisateur pour choisir les paramètres. L'approche de décodage utilisée est le Trace Back. Le programme est constitué comme suit :
  – Une fonction « Treillis.m » qui génère une structure selon les paramètres d'entrée. Cette structure contient tous les états, les prochains états et les mots codes pour chaque transition (Voire Annexe A.2.1.1).
  – Une fonction « ConnvolutionalEncoder.m » qui fait appel à la fonction « treillis » pour coder le flux de bits selon les paramètres du codeur (Voire Annexe A.2.1.2).
  – Une fonction « Butterfly.m » qui calcule les métriques (Voire Annexe A.2.1.3).
  – Programme principale « Viterbi_RE.m » : Il fait appel aux trois fonctions précédentes, en plus il génère des erreurs dans la séquence codée avant de lancer le décodage (Voire Annexe A.2.1.4).

4.3.1 Simulation du décodeur choisi

4.3.1.1 Décodage Trace Back

Pour cette simulation nous avons utilisé une longueur de Trace Back $L_{TB} = 15$, mais d'autres simulations ont été réaliser. Le décodeur a été évaluer pour différentes longueur de Trace Back. Nous avons eu, avec la longueur $L_{TB} = 7$, des résultats satisfaisants.

La séquence à coder générée aléatoirement est la suivante :

$$u = 11100110001011111011011011 10$$

La séquence codée correspondante est :

$$\begin{aligned} b \;=\; & 11\,01\,00\,00\,10\,00\,01\,11\,10 \\ & 11\,11\,10\,10\,10\,00\,11\,00\,01 \\ & 10\,11\,01\,10\,11\,01\,10\,00\,00 \end{aligned}$$

Le signal à transmettre et le signal codé transmis sont représentés sur la Figure 4.1

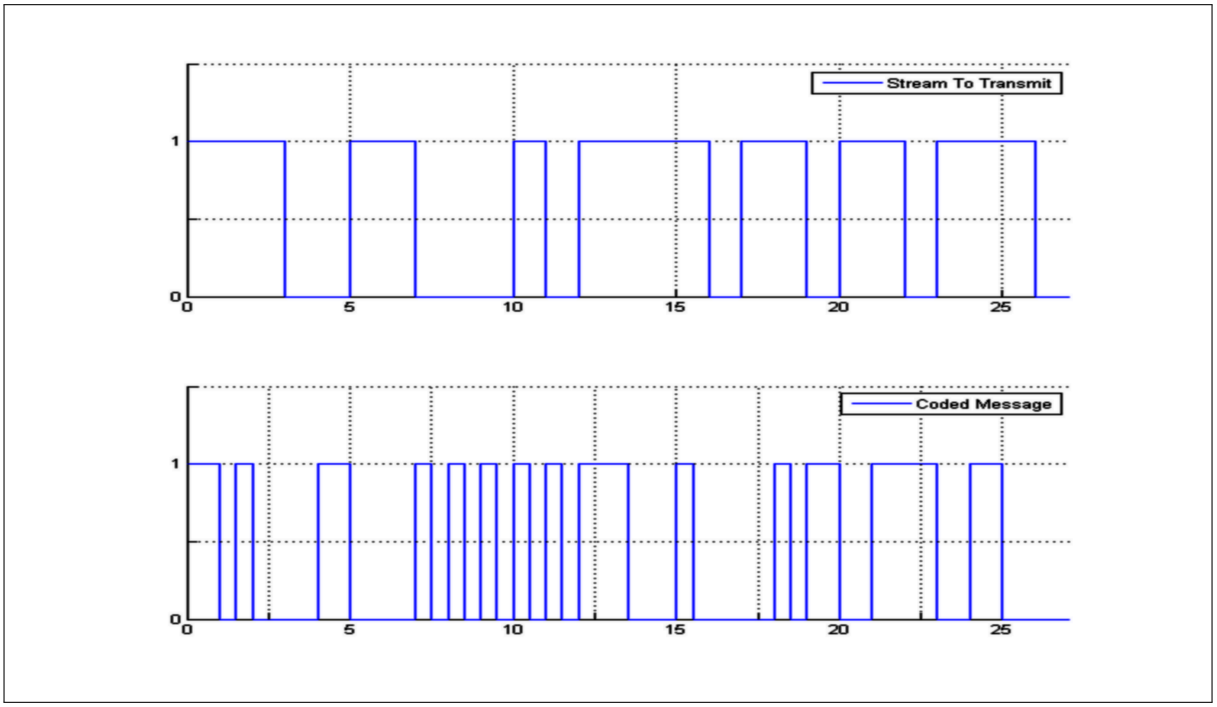


FIGURE 4.1: Représentation des signaux u et b.

Le signal codé reçu (avec erreur) et le signal décodé des deux essais sont représentés sur les Figures 4.2 et 4.3.

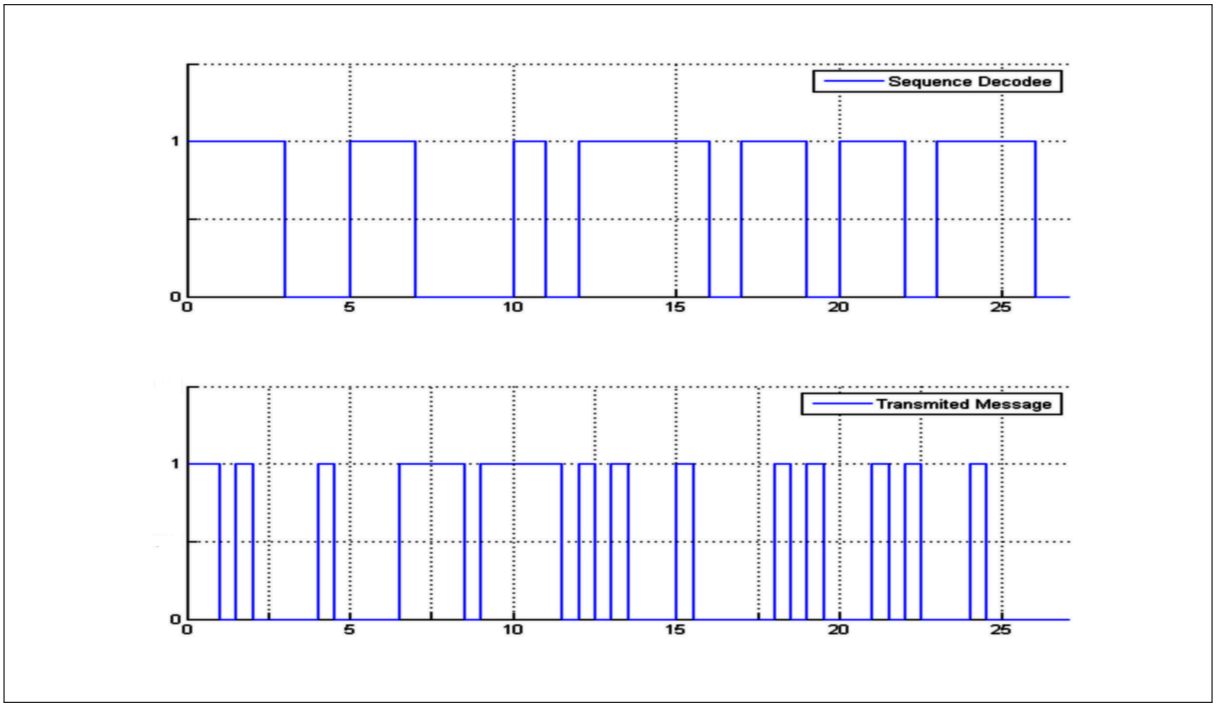


FIGURE 4.2: Représentation des signaux r et $\widehat{u}$: Essai 1.

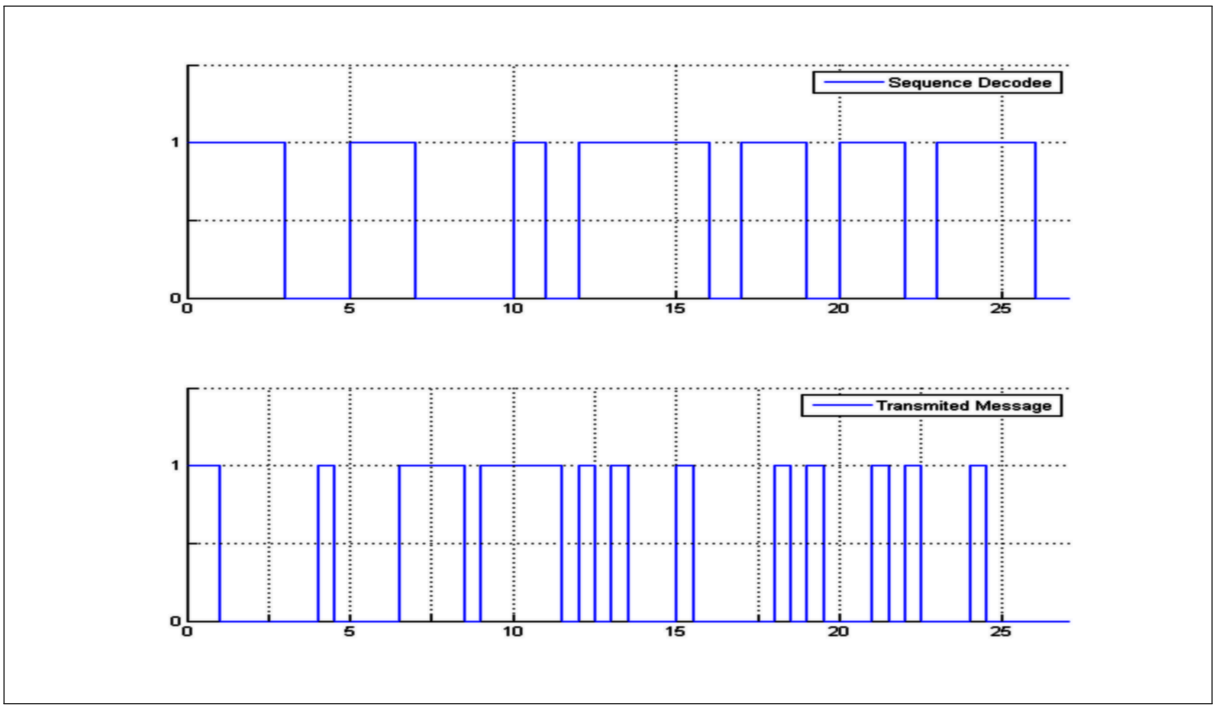


FIGURE 4.3: Représentation des signaux r et $\widehat{u}$: Essai 2.

Nous remarquons que la séquence décodé est la même que l'information originale bien que le signal codé reçu est différent du message codé transmis, pour les deux essais.

4.3.1.2 Décodage Register Exchange

Pour cette simulation nous avons utilisé 8 registres (Chacun correspond à un état) d'une longueur qui varie selon la séquence à décoder. Le décodeur a été évaluer pour des tailles fixes des registres, nous avons eu des résultats moins satisfaisants.

La séquence à coder générée aléatoirement est la suivante :

$$u = 11100110001011110110110 1110$$

La séquence codée correspondante est :

$$\begin{aligned} b \;=\; & 11\,01\,00\,00\,10\,00\,01\,11\,10 \\ & 11\,11\,10\,10\,10\,00\,11\,00\,01 \\ & 10\,11\,01\,10\,11\,01\,10\,00\,00 \end{aligned}$$

Le signal à transmettre et le signal codé transmis sont représentés sur la Figure 4.4

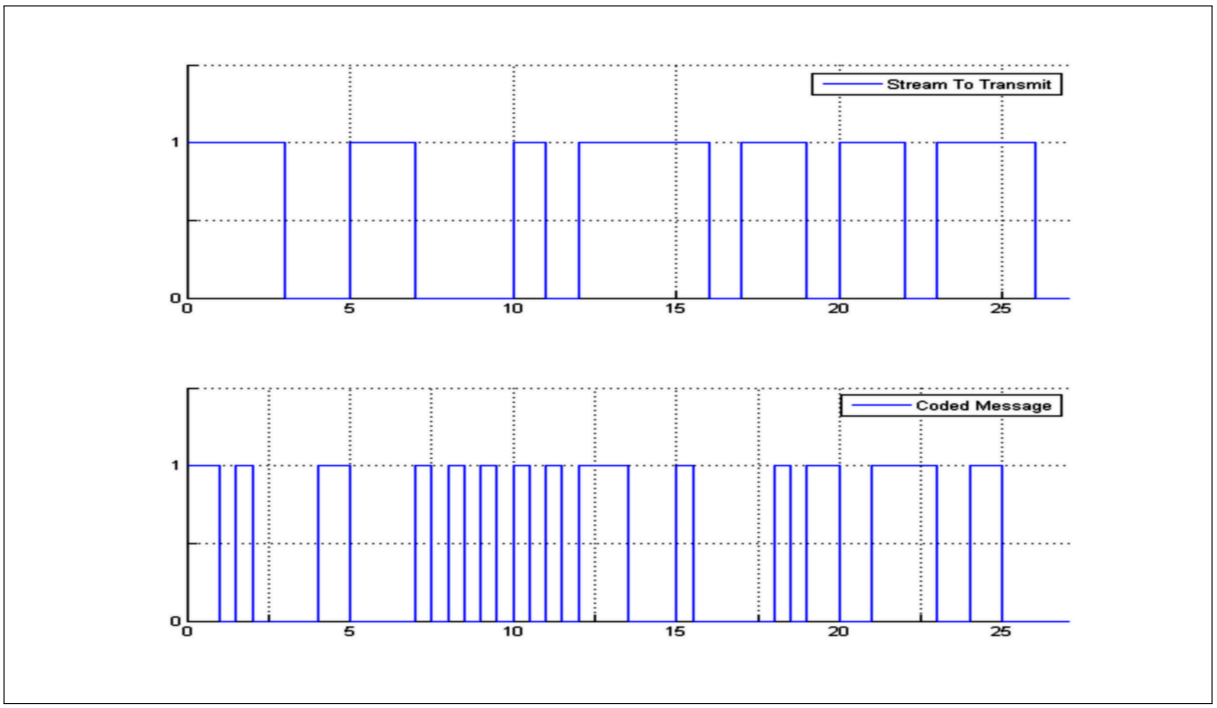


FIGURE 4.4: Représentation des signaux u et b.

Le signal codé reçu (avec erreur) et le signal décodé des deux essais sont représentés sur les Figures 4.5 et 4.6.

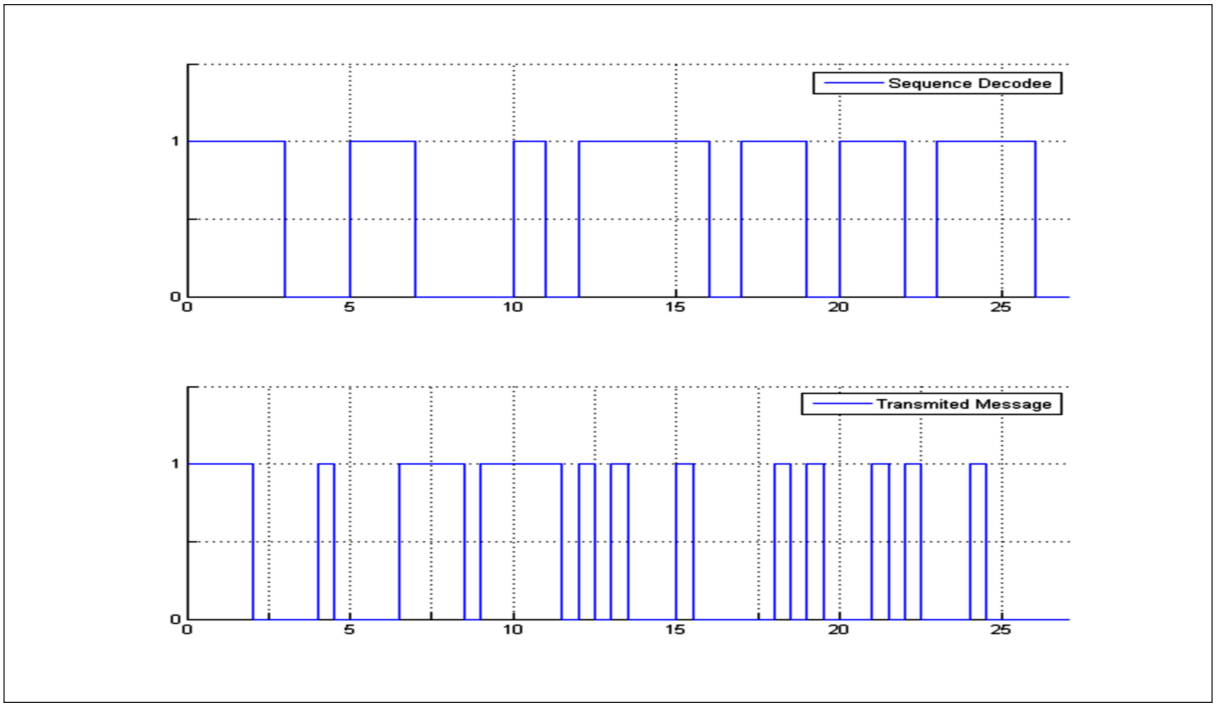


FIGURE 4.5: Représentation des signaux r et $\widehat{u}$: Essai 1.

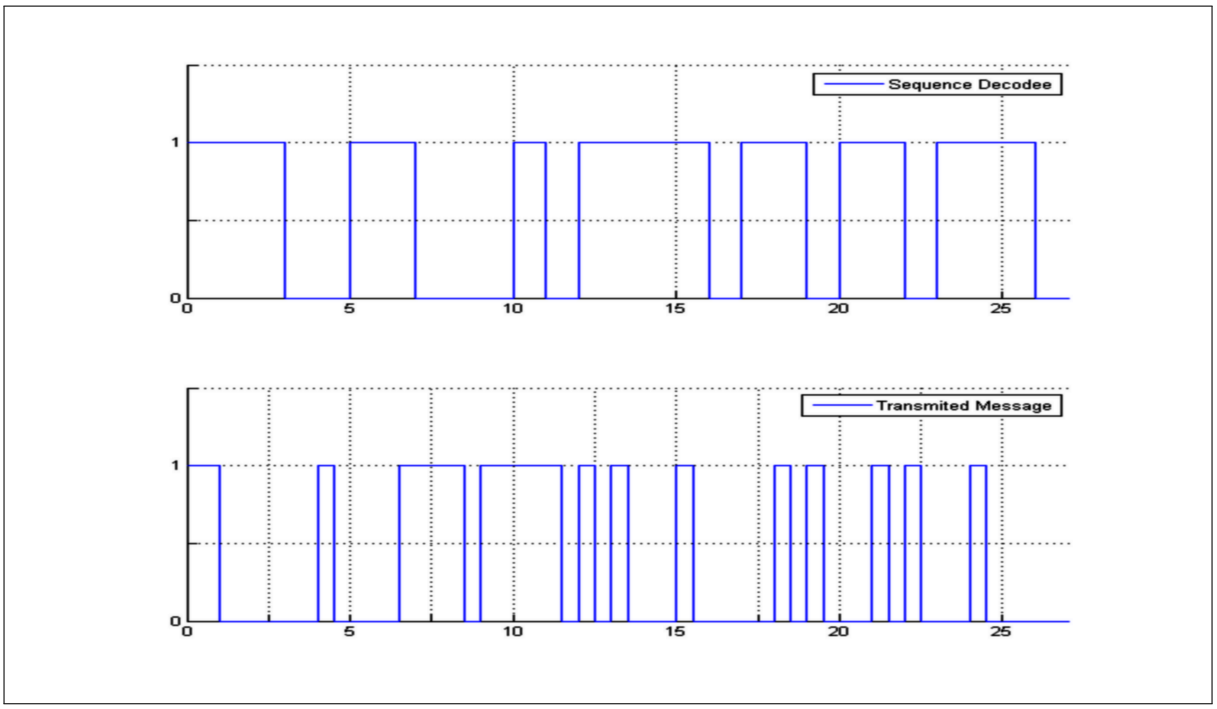


FIGURE 4.6: Représentation des signaux r et $\widehat{u}$: Essai 2.

Nous remarquons que la séquence décodé est la même que l'information originale bien que le signal codé reçu est différent du message codé transmis, pour les deux essais.

4.3.2 Simulation Cassini

Les paramètres du codes Cassini sont donnés dans le Tableau 2.1. La longueur du Trace Back choisis est $L_{TB} = 15$ et la séquence d'information est :

$$u = 110010111010100$$

La séquence codée correspondante est :

$$\begin{aligned} b \quad = \quad & 111111\ 111000\ 001001\ 010101\ 011010 \\ & 101111\ 001110\ 111100\ 110011\ 011111 \\ & 001100\ 000100\ 000111\ 010111\ 110101 \end{aligned}$$

Le signal à transmettre et le signal codé transmis sont représentés sur la Figure 4.7

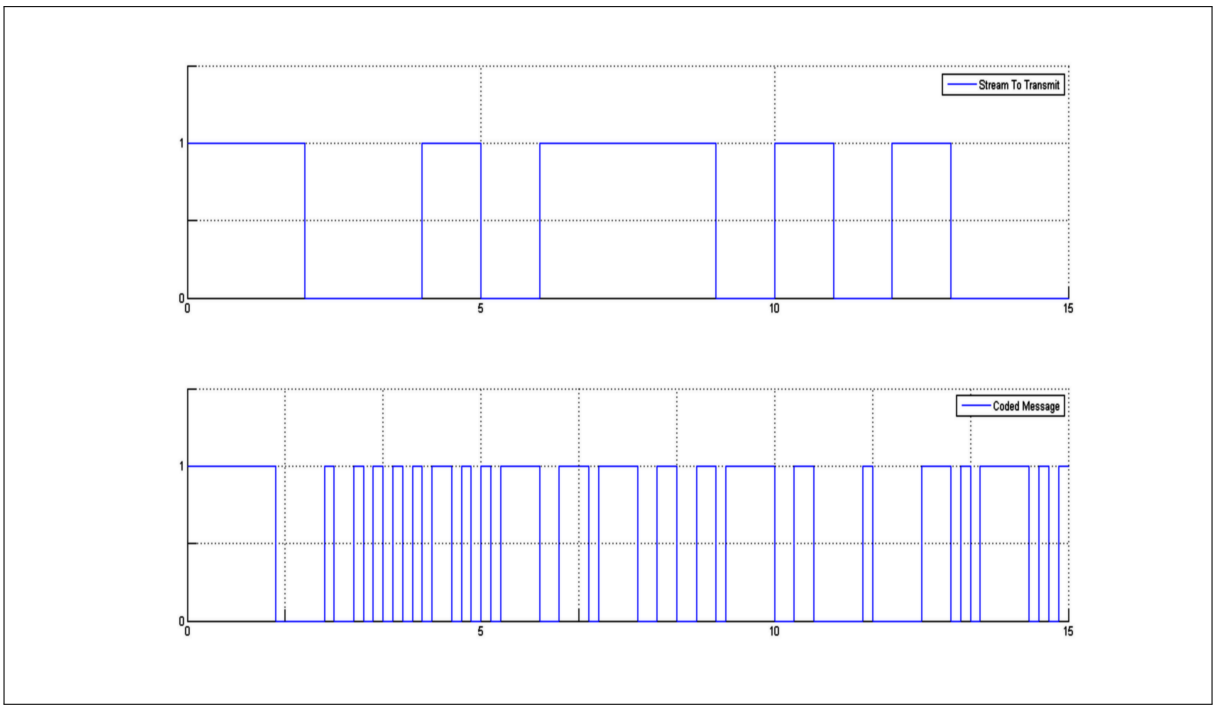


FIGURE 4.7: Représentation des signaux u et b.

Le signal codé reçu (avec erreur) et le signal décodé des deux essais sont représentés sur les Figures 4.8 et 4.9.

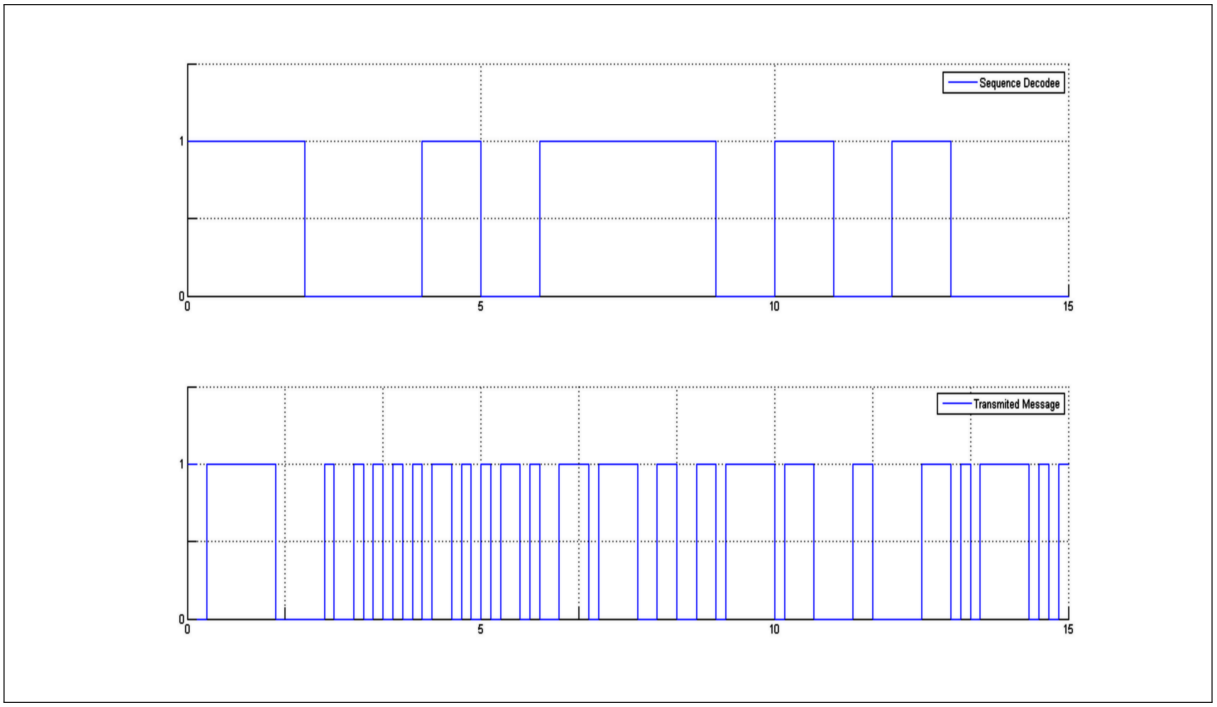


FIGURE 4.8: Représentation des signaux r et $\widehat{u}$: Essai 1.

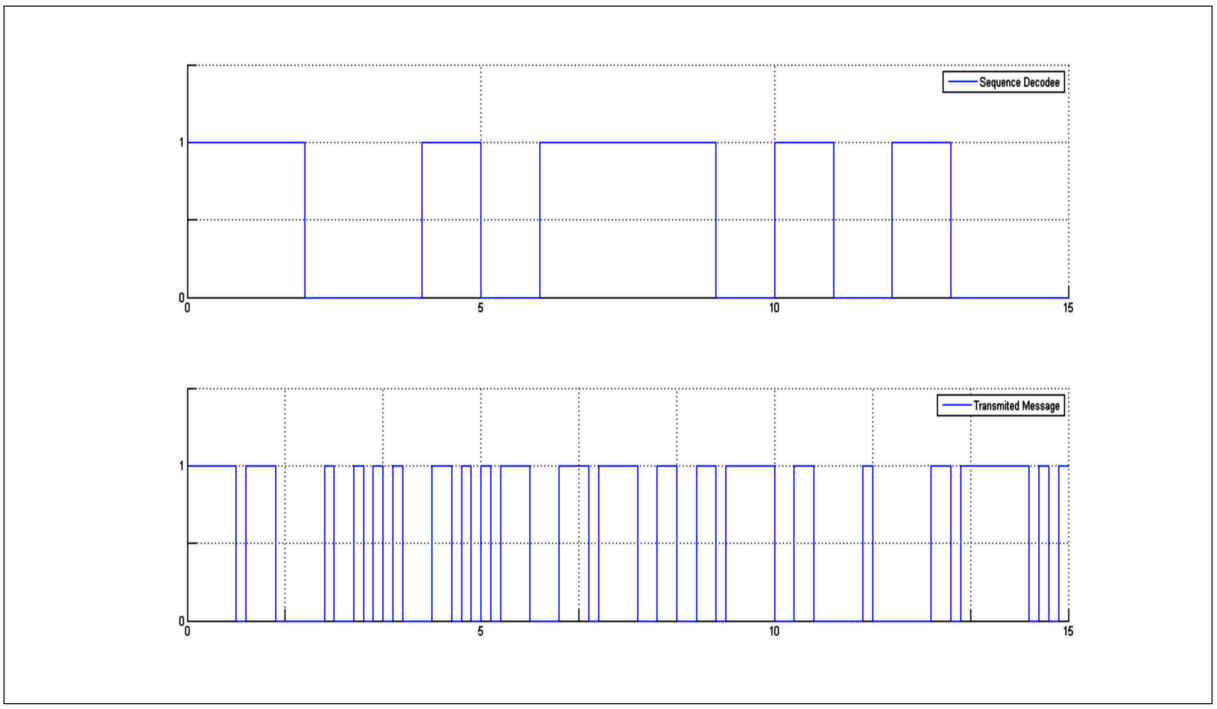


FIGURE 4.9: Représentation des signaux r et $\widehat{u}$: Essai 2.

Nous remarquons que la séquence décodé est la même que l'information originale bien que le signal codé reçu est différent du message codé transmis, pour les deux essais.

4.3.3 Simulation CDMA 2000

Les paramètres du codes CDMA 2000 sont donnés dans le Tableau 2.1. La longueur du Trace Back choisis est $L_{TB} = 20$ et la séquence d'information est :

$$u = 110010111010100$$

La séquence codée correspondante est :

$$\begin{aligned} b \;=\; & 1111\ 0011\ 0110\ 0111\ 1111\ 0110\ 1010\ 0101\ 1000\ 1100\ 1110\ 0001\ 1100 \\ & 0100\ 0010\ 1100\ 1010\ 1111\ 1010\ 1100\ 1100\ 0100\ 0000\ 1011\ 1110\ 1000 \end{aligned}$$

Le signal à transmettre et le signal codé transmis sont représentés sur la Figure 4.10

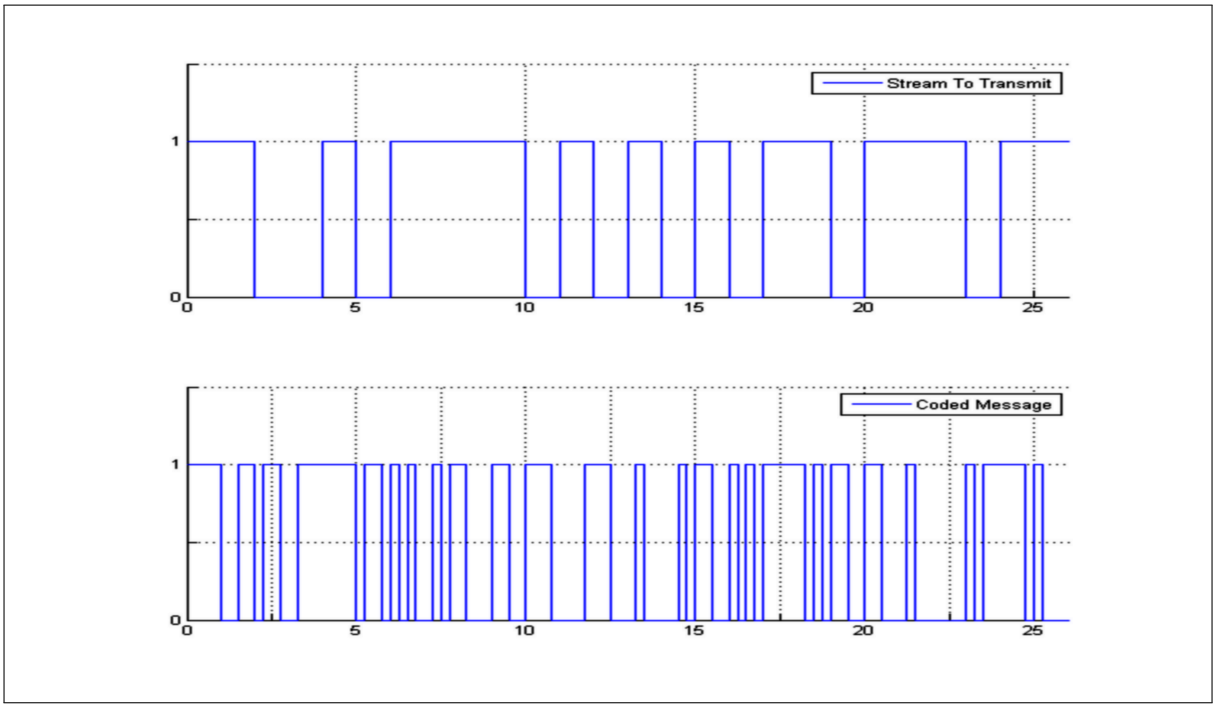


FIGURE 4.10: Représentation des signaux u et b.

Le signal codé reçu (avec erreur) et le signal décodé des deux essais sont représentés sur les Figures 4.11 et 4.12.

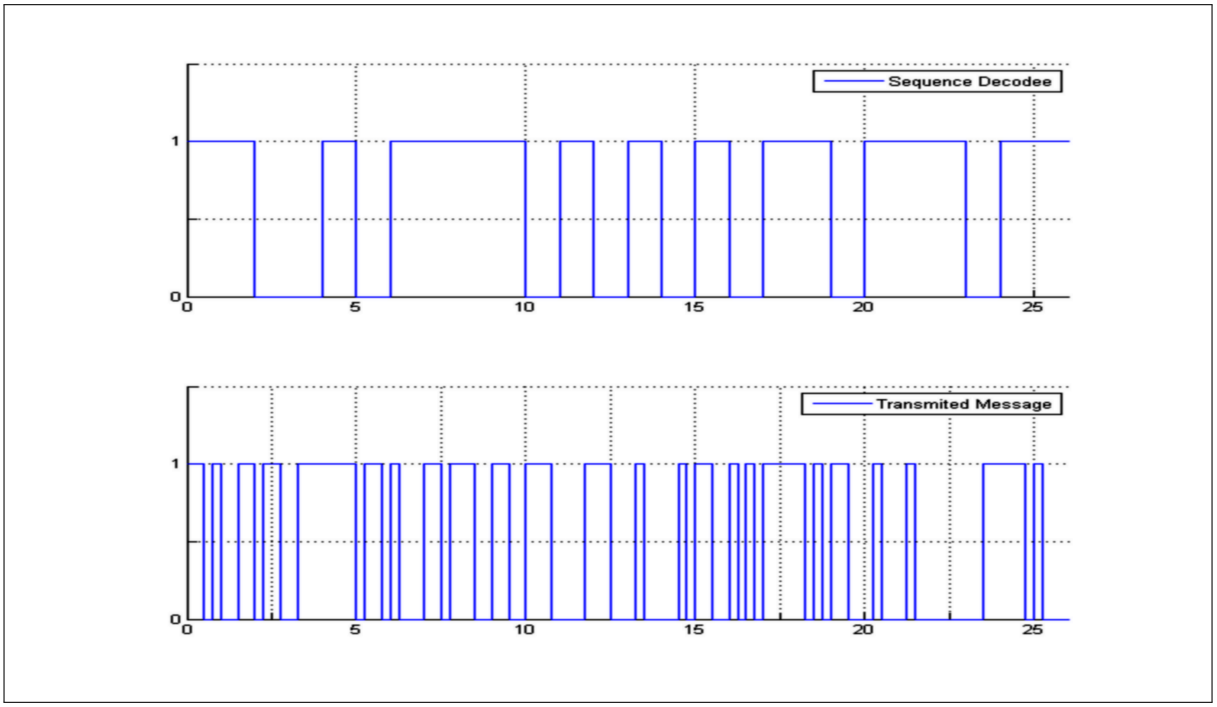


FIGURE 4.11: Représentation des signaux r et $\widehat{u}$: Essai 1.

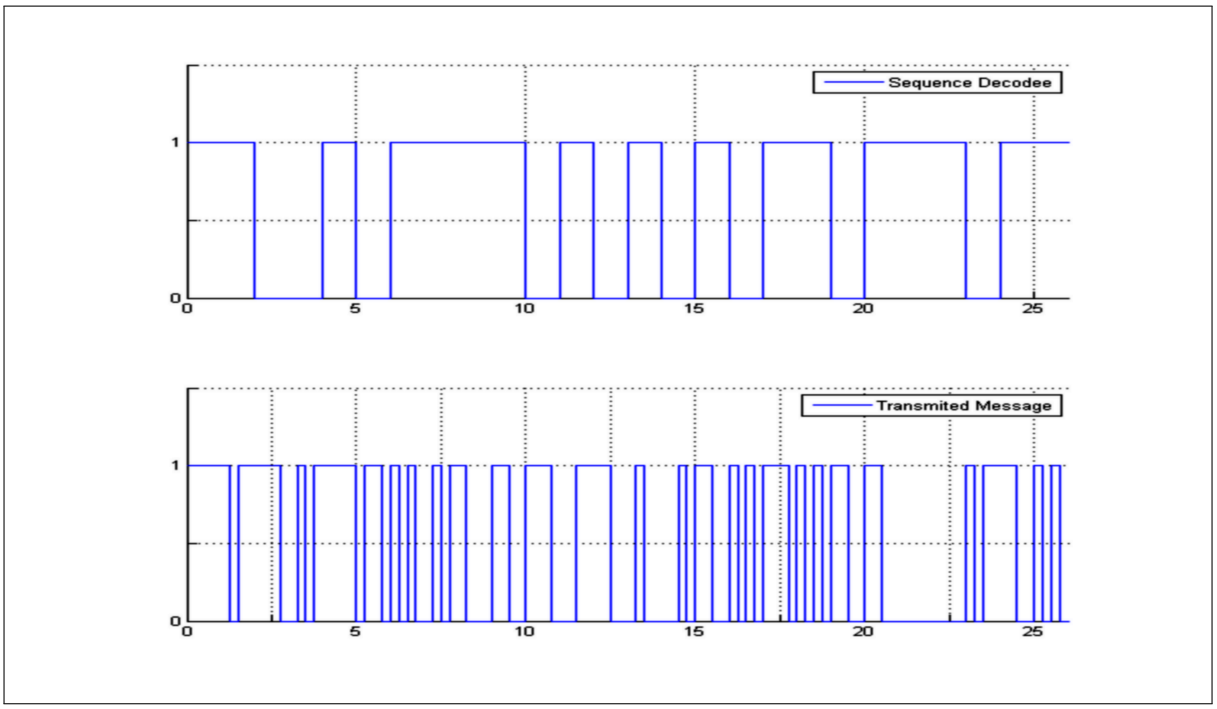


FIGURE 4.12: Représentation des signaux r et $\widehat{u}$: Essai 2.

Nous remarquons que la séquence décodé est la même que l'information originale bien que le signal codé reçu est différent du message codé transmis, pour les deux essais.

4.3.4 Simulation UWB (802.15)

Les paramètres du codes UWB (802.15) sont donnés dans le Tableau 2.1. La longueur du Trace Back choisis est $L_{TB} = 20$ et la séquence d'information est :

$$u = 11001011110101010110111011$$

La séquence codée correspondante est :

$$\begin{aligned} b \;=\; & 111\,001\,001\,010\,000\,101\,110\,011\,100\,010\,001\,110\,111\,100\,101 \\ & 101\,010\,101\,010\,010\,011\,011\,001\,100\,010\,101\,011\,101\,000 \end{aligned}$$

Le signal à transmettre et le signal codé transmis sont représentés sur la Figure 4.13

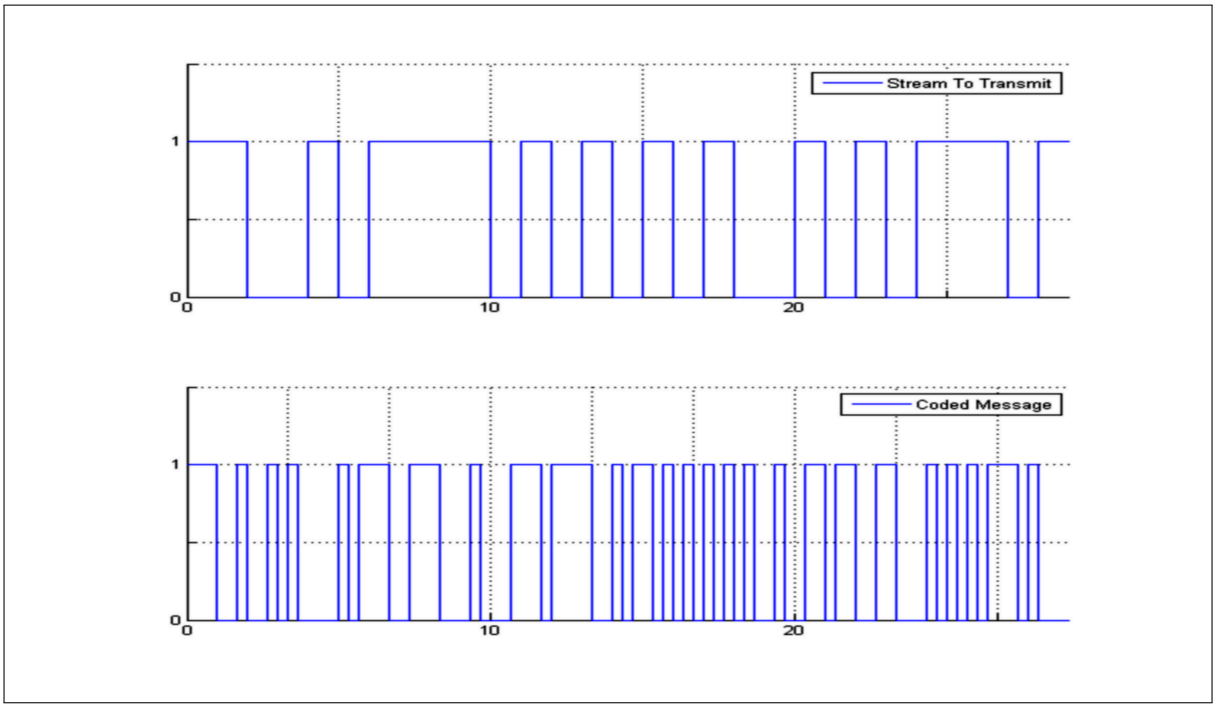


FIGURE 4.13: Représentation des signaux u et b.

Le signal codé reçu (avec erreur) et le signal décodé des deux essais sont représentés sur les Figures 4.14 et 4.15.

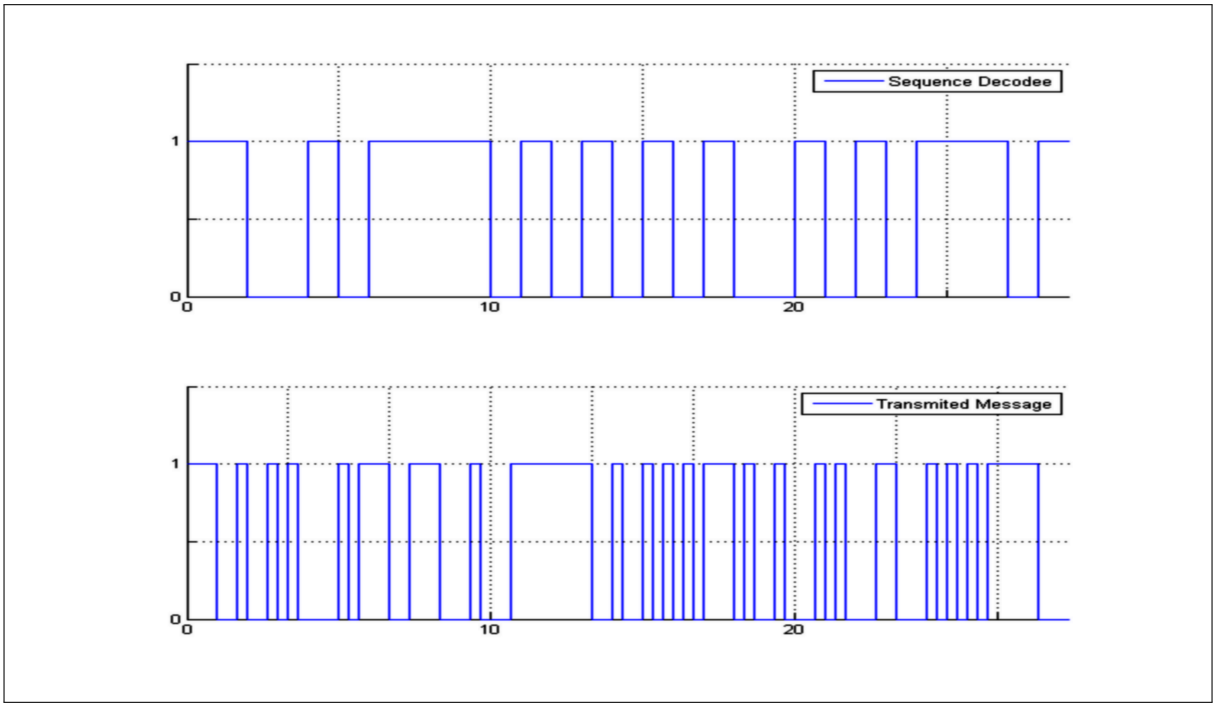


FIGURE 4.14: Représentation des signaux r et $\widehat{u}$: Essai 1.

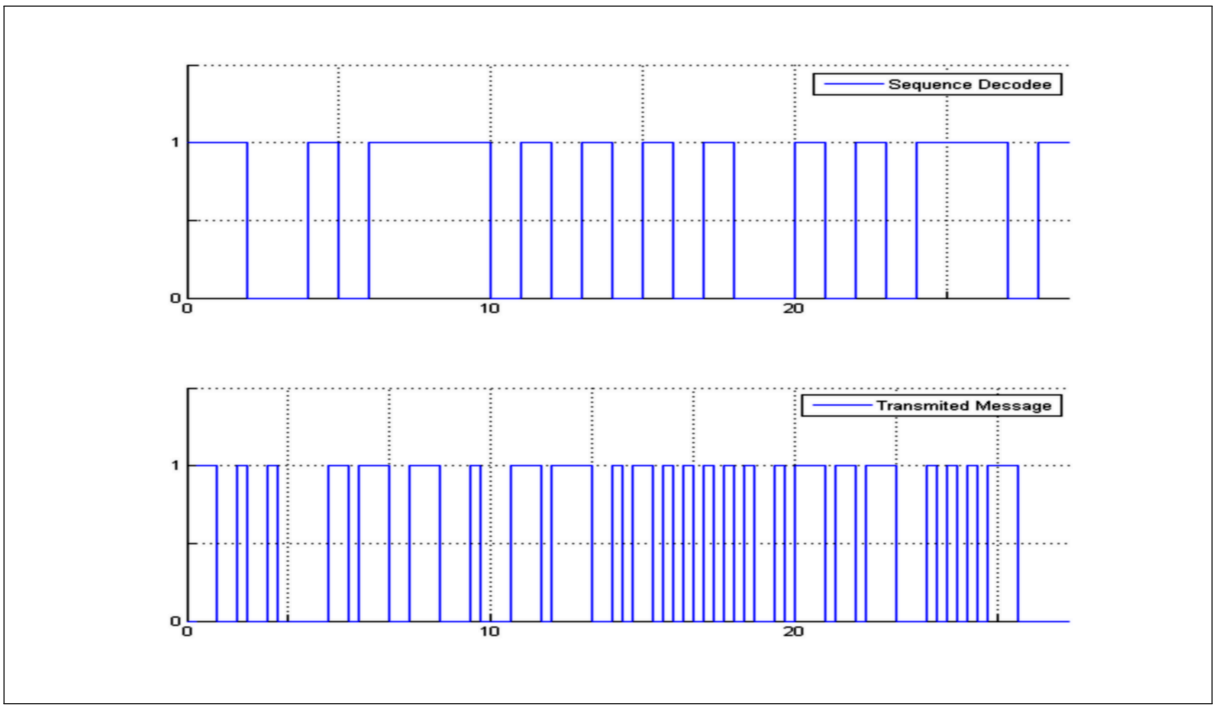


FIGURE 4.15: Représentation des signaux r et $\widehat{u}$: Essai 2.

Nous remarquons que la séquence décodé est la même que l'information originale bien que le signal codé reçu est différent du message codé transmis, pour les deux essais.

4.4 Simulation et implémentation sous ISE de Xilinx

4.4.1 Codeur Convolutionnel

Rappelons que nous avons utilisé un codeur convolutionnel de longueur de contrainte $L = 4$ et de rendement $R = 1/2$. Ce dernier constitue un préambule nécessaire afin de nous fournir la donnée codée. Comme ce codeur n'est pas au centre de nos préoccupations nous l'avons conçu en un seul fichier VHDL. Néanmoins, nous pouvons dire succinctement qu'il est constitué d'un registre à décalage représenté dans le programme par un vecteur nommé *reg* constitué de 4 éléments. Il est à noté que les deux sorties $s_1$et s_2 - qui sont le résultats des deux convolutions décrites dans ce qui précède - sont décrites comme des sorties parallèles à l'intérieur du programme. Il est donc nécessaire de prévoir en dehors ou au sein du codeur convolutionnel un circuit permettant de rendre ces sorties séquentielles.

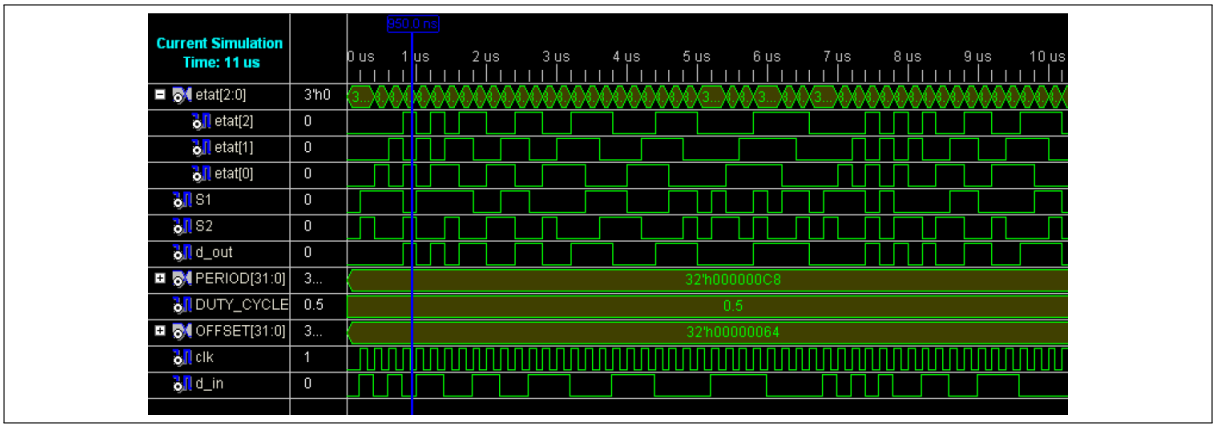


FIGURE 4.16: Test Bench du codeur convolutionnel à sortie parallèle.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.all;
use ieee.std_logic_unsigned.all;
entity ConvoCoder is
  port (
    clk          : in  std_logic;
    d_in         : in  std_logic;
    S1       : out std_logic;
    S2       : out std_logic;
    );
```

```
end ConvoCoder;
architecture Behavioral of ConvoCoder is
  signal reg : std_logic_vector (3 downto 0):="0000";
signal is1: std_logic:='0';
signal is2: std_logic:='0';
begin
  process (clk)
  begin
    if rising_edge (clk) then
      reg(0) <= d_in;
      reg(3 downto 1) <= reg(2 downto 0);
    end if;
  end process;
  is1<=reg(0) xor reg(1) xor reg(3);
  is2<=reg(0) xor reg(2) xor reg(3);
  S1 <= is1;
  S2 <= is2;
end Behavioral;
```

En ce qui concerne la synchronisation du codage, nous avons utilisé l'horloge interne de la carte FPGA utilisée.

Nous avons simulé le comportement de ce codeur sur une carde FPGA Spartan 3E. La fréquence interne de l'horloge étant de $50MHz$, il nous serait impossible de vérifier le bon fonctionnement de notre codeur sans modification de l'horloge. A cette fin, nous avons décider d'introduire au sein de notre programme un diviseur de fréquence utilisé uniquement pour la simulation.

La programme ci-après décrit le circuit de division de fréquence :

```
clk_division : process (clk, clk_divider)
begin
  if (clk = '1' and clk'Event) then
    clk_divider <= clk_divider + 1;
  end if;
  slow_clk <= clk_divider(24);
end process;
```

Et voici la partie de programme qui nous permet d'effectuer cette division.

Nous constaterons dans le programme du codeur joint en Annexe B.1, que nous avons donc remplacé le signal *clk* par le signal *slow_clk* qui représente le signal d'horloge de fréquence inférieure. Nous avons eu la nécessité de diviser la fréquence de l'horloge interne de la carte par 24.

4.4.2 Papillons

Comme nous l'avons expliqué précédemment dans la description de l'architecture. Nous avons placé le calcul des métriques de branches et des métriques de chemin dans

un bloc unique : le bloc MCU (Metrics Calculating Unit). Les calculs de métriques de chemins sont gérées deux par deux par des BFs (Butterflies).

En effet, nous avons décrit en VHDL la structure de quatre papillons :

– BF000_001 : C'est le papillon qui décrit le calcul des nouvelles métriques de chemins des états 000 et 100 à partir des métriques de chemin précédentes des états 000 et 001.

```
entity BF000_001 is
    Port (
      BF1reset : in std_logic;
      clk      : in std_logic;
      BF1en    : in std_logic;
      IF000    : in  std_logic;
      IF001    : in  std_logic;
      RD       : in std_logic_vector(1 downto 0);
      PMs      : in std_logic_vector(2 downto 0);
      PM001    : in  std_logic_vector(7 downto 0);
      PM000    : in  std_logic_vector(7 downto 0);
      OF000    : out  std_logic:='0';          --output Flag
      OF100    : out  std_logic:='0';          --output flag
      BF1r     : out std_logic:='0';           --butterfly ready to
         send flag
      DB000    : out  std_logic;            --decision bit towards new
          state 001
      DB100    : out  std_logic;            --decision bit towards new
          state 101
      Newpm000 : out  std_logic_vector(7 downto 0):="00000000"; --New
         path metric 000
      Newpm100 : out  std_logic_vector(7 downto 0):="00000000"  --New
         path metric 100
    );
end BF000_001;
```

– BF010_011 : C'est le papillon qui décrit le calcul des nouvelles métriques de chemins des états 101 et 001 à partir des métriques de chemin précédentes des états 010 et 011.

```
entity BF010_011 is
    Port (
      BF2reset : in std_logic;            --reset
      clk      : in std_logic;            --clock
      BF2en    : in std_logic;            --enable
      IF010    : in  std_logic;           --input Flag
      IF011    : in  std_logic;           --input Flag
      RD       : in std_logic_vector(1 downto 0);     --received data
      PMs      : in std_logic_vector(2 downto 0);     --Path metric
         select
      PM010    : in  std_logic_vector(7 downto 0);      --Previous
         Path metric 010
```

```
        PM011     : in  std_logic_vector(7 downto 0);      --Previous
            Path metric 011
        OF001     : out  std_logic:='0';          --output Flag
        OF101     : out  std_logic:='0';          --output flag
        BF2r      : out std_logic:='0';           --butterfly ready to
            send flag
        DB001     : out  std_logic;             --decision bit towards new
             state 001
        DB101     : out  std_logic;             --decision bit towards new
             state 101
        Newpm001 : out  std_logic_vector(7 downto 0):="00000000"; --New
            path metric 001
        Newpm101 : out  std_logic_vector(7 downto 0):="00000000"  --New
            path metric 101
        );
end BF010_011;
```

– BF100_101 : C'est le papillon qui décrit le calcul des nouvelles métriques de chemins des états 110 et 010 à partir des métriques de chemin précédentes des états 100 et 101.

```
entity BF100_101 is
    Port (
      BF3reset: in std_logic;
      clk: in std_logic;
      BF3en: in std_logic;
      IF100: in  std_logic;
      IF101: in  std_logic;
      RD: in std_logic_vector(1 downto 0);
      PMs: in std_logic_vector(2 downto 0);
      PM100 : in  std_logic_vector(7 downto 0);
      PM101 : in  std_logic_vector(7 downto 0);
      OF010: out  std_logic;
      OF110: out  std_logic;
      BF3r: out std_logic;
      DB010 : out  std_logic;
      DB110 : out  std_logic;
      Newpm010 : out  std_logic_vector(7 downto 0);
      Newpm110 : out  std_logic_vector(7 downto 0)
      );
end BF100_101 ;
```

– BF110_111 : C'est le papillon qui décrit le calcul des nouvelles métriques de chemins des états 011 et 111 à partir des métriques de chemin précédentes des états 110 et 111.

```
entity BF110_111 is
    Port (
      BF1reset: in std_logic;
      clk: in std_logic;
      BF1en: in std_logic;
```

```
        IF110: in  std_logic;
        IF111: in  std_logic;
        RD: in std_logic_vector(1 downto 0);
        PMs: in std_logic_vector(2 downto 0);
        PM110 : in  std_logic_vector(7 downto 0);
        PM111 : in  std_logic_vector(7 downto 0);
        OF011: out  std_logic;
        OF111: out  std_logic;
        BF4r: out std_logic;
        DB011 : out  std_logic;
        DB111 : out  std_logic;
        Newpm011 : out  std_logic_vector(7 downto 0);
        Newpm111 : out  std_logic_vector(7 downto 0)
        );
end BF110_111;
```

Ainsi que le corroborent les extraits de programmes précédents nous avons utilisé des Flags. Ce sont des signaux que nous avons mis en place afin de vérifier, à chaque étape, la validité de la métrique de branche précédente. En effet, dans les trois premiers étages du treillis, il n'y a respectivement qu'un, deux ou quatre états valides. Et cela doit être pris en considération afin de ne pas effectuer de lecture en RAM des métriques des états non valides.

Ils incluent la BMU (Branche Metrics Unit) mesurant la distance entre la paire de bits reçues et attendues respectivement.

Le Test Bench de l'un des papillons est illustrer sur la Figure 4.17.

Les instructions calculant les métriques de branche effectuent uniquement une sélection au cas par cas, décrite dans l'extrait de programme ci-dessous :

```
process (RD)                    --RD:received
  if (RD ="00") then
    BM00 <= "00";          --BM00 distance entre RD et 10
    BM11 <= "10";          --BM11 distance entre RD et 10
  else
    if( RD ="10" or RD ="01") then
      BM00 <= "01";
      BM11 <= "01";
    else
      BM00 <= "10";
      BM11 <= "00";
    end if;
  end if;
end process;
```

ou encore :

```
 process (RD)                                    --RD Received Data (2
    bits)
begin
  if (RD ="10") then
```

```
    BM10 <= "00";        --BM10 distance entre RD et 10
    BM01 <= "10";        --BM01 distance entre RD et 01
  else
    if( RD ="11" or RD ="00") then
      BM01 <= "01";
      BM10 <= "01";
    else
      BM10 <= "10";
      BM01 <= "00";
    end if;
  end if;
end process;
```

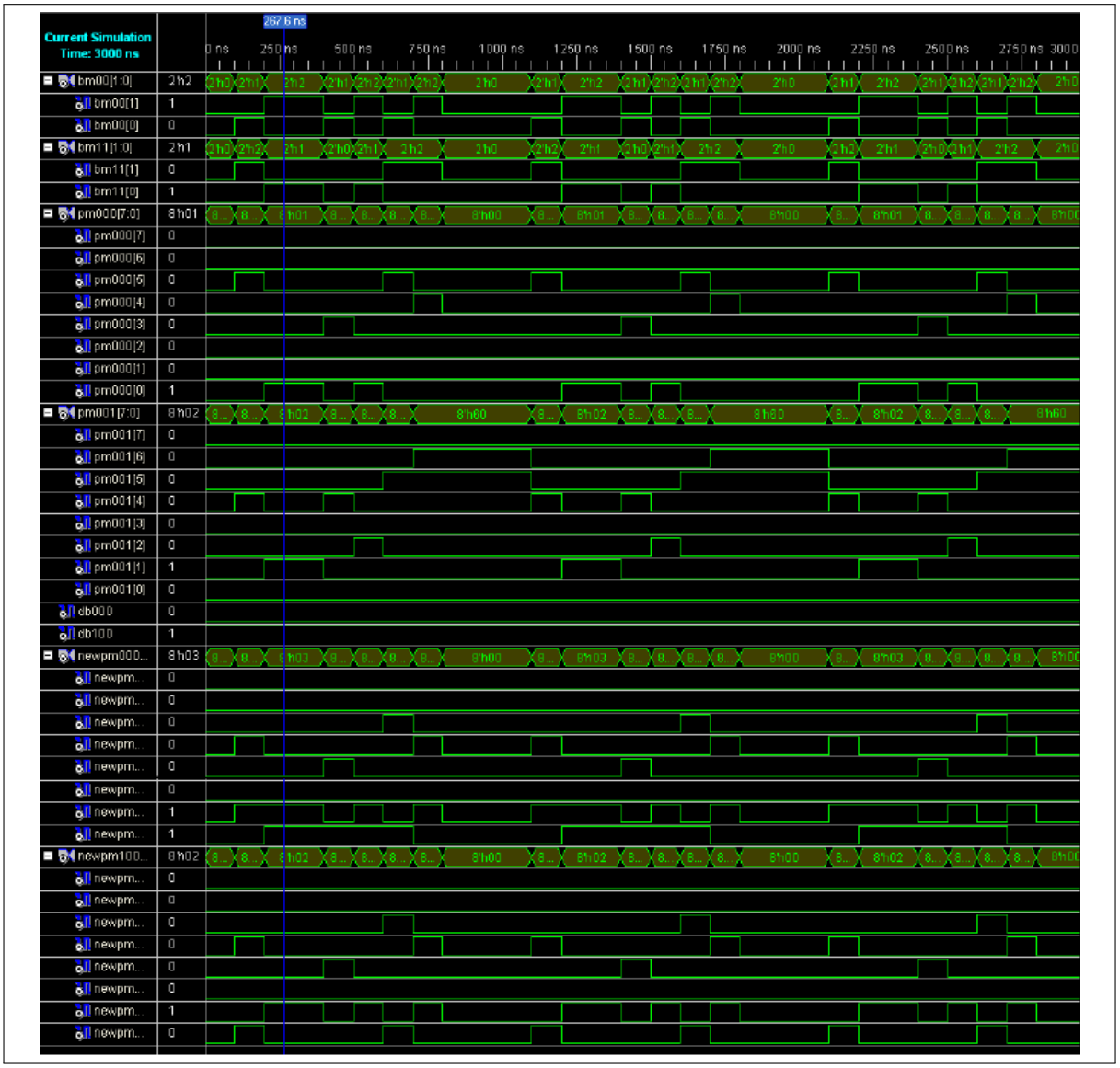


FIGURE 4.17: Papillon 000-001 sans les bits Flags.

4.4.3 RAM des métriques et des décisions

Ainsi que décrit dans l'architecture choisie, nous avons jugé utile de programmer quatre mémoires RAM. Deux d'entre elles sont des RAM pour le stockage des métriques de chemin, et les deux autres sont destinées à stocker les bits de décision. Ces mémoires possèdent deux bus d'adresse, respectivement destinés à la lecture et à l'écriture. Elles sont dotés de deux signaux de contrôle *read* et *write* distincts.

Lorsque le système est remis à zéro, la première RAM de métriques de chemins est activée par un signal *enable*, la seconde reste éteinte. Une fois le treillis plein, cette

dernière est activée et placée en mode écriture afin de recevoir les métriques de chemins suivantes. La première est alors mise en mode lecture est fournira les dernières métriques stockées au *MinFinder* , bloc servant à trouver la métrique de chemin minimale et qui sera décrit plus loin.

L'état d'avancement du treillis est indiqué par un compteur qui, une fois sa valeur maximale atteinte (pour notre cas égale à 15), permet de donner l'ordre d'échanger les RAM de métriques. Ce changement s'effectuant à chaque fois que le compteur atteint un multiple de 15.

Le même modèle de fonctionnement est associé aux deux RAM de décision. A ceci prés que ces RAM ne sont pas connectées au *MinFinder* à la manière des RAM de métriques mais sont plutôt connectées à l'unité de *TraceBack* qui sera aussi décrit plus avant.

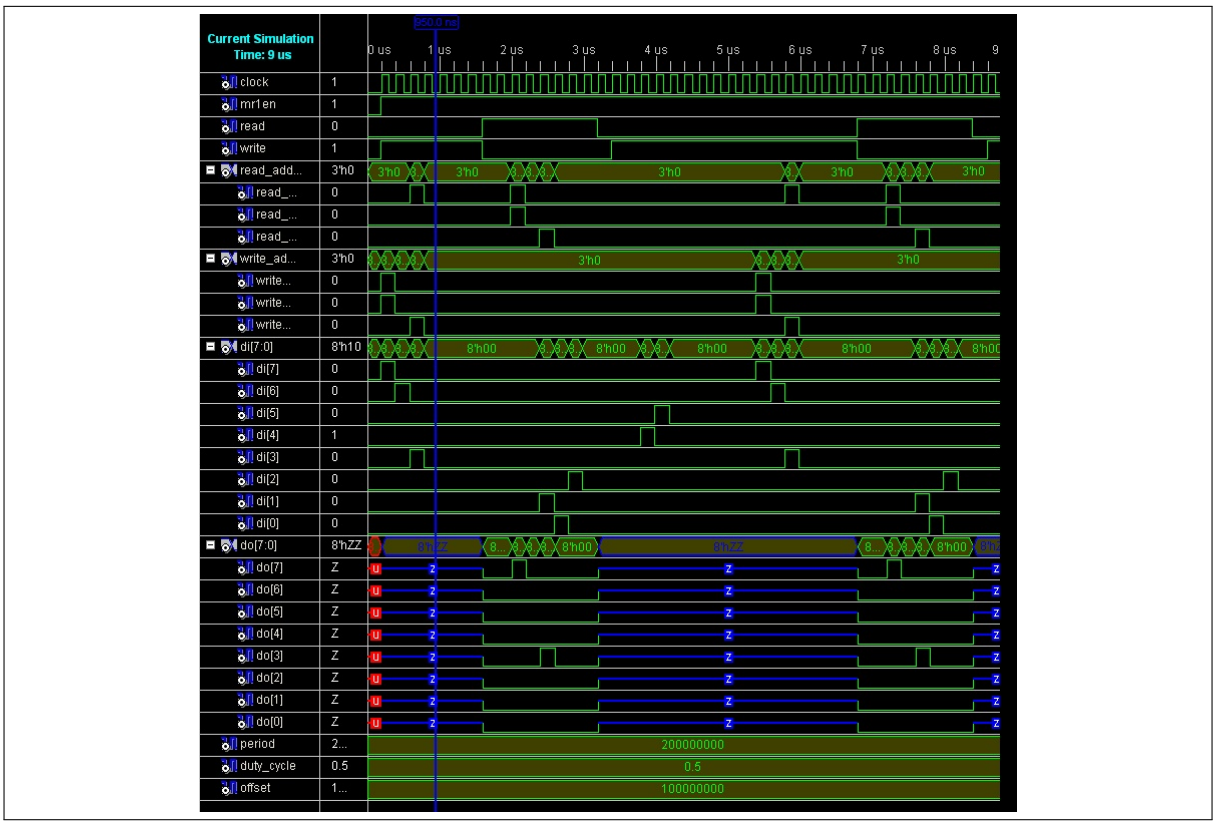


FIGURE 4.18: RAM des métriques.

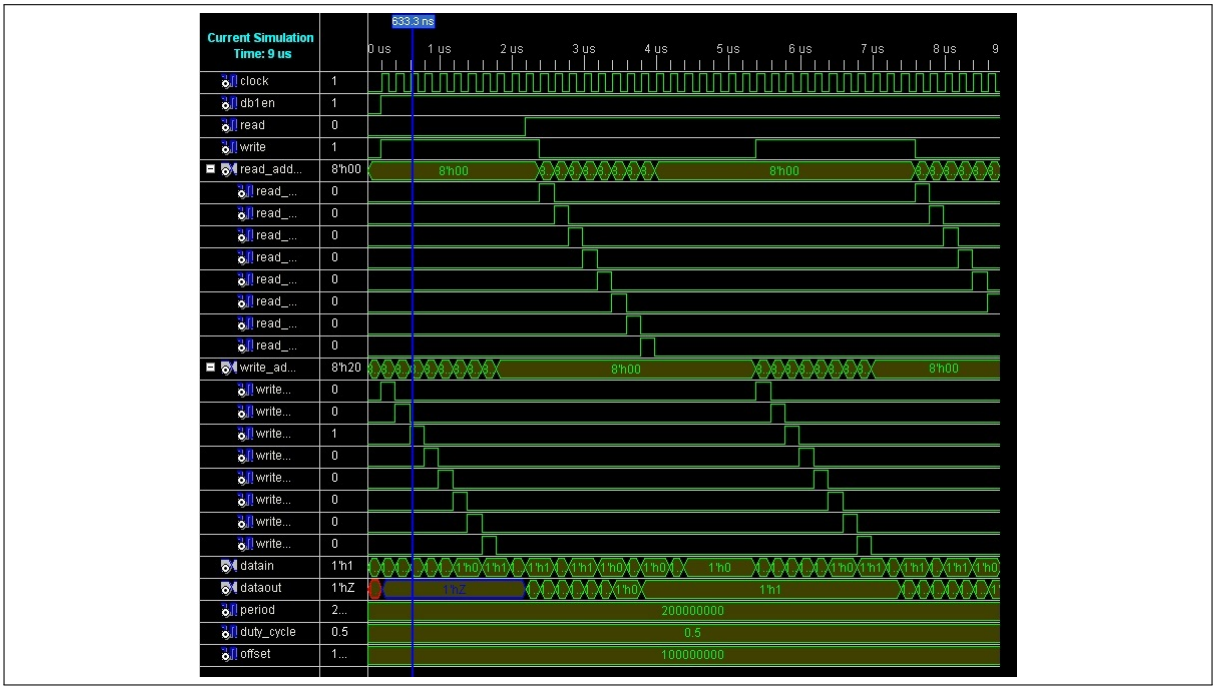


FIGURE 4.19: RAM des bits de décision.

4.4.4 Détecteur de minimum

Ce bloc sert à rechercher la métrique de chemin minimale au sein de la RAM des métriques lorsque le compteur du treillis atteint une longueur égale à 15 ou à un multiple de 15.

```
entity MinFinder is
  port(
       MFen: in std_logic;
       wordin: in std_logic_vector(7 downto 0);
       ReadRam: in std_logic;
       copied: out std_logic;
       done: out std_logic:='0';
       min : out std_logic_vector(7 downto 0):="11111111"
       );
end MinFinder;
```

4.4.5 Trace Back

Cette unité comporte un registre à décalage ainsi qu'une pile LIFO.

Lorsque le détecteur de minimum a transmis à l'unité de Trace Back l'adresse en RAM de la métrique de branche minimale, celle ci est chargée en parallèle dans le registre à décalage du Trace Back.

Le registre à décalage attend alors de recevoir le bit de décision correspondant à l'état chargé et effectue un décalage à gauche. La sortie de ce registre constitue l'information décodée mais cette information est donné dans le sens inverse de sa réception.

La pile LIFO servira, à accumuler les bits sortant du registre à décalage et les remettre dans le bon sens. La sortie de cette pile constitue donc l'information décodée à proprement dite qui doit exactement correspondre à l'information existante avant le codage de canal et la transmission.

4.4.5.1 Shift Registre

La Figure 4.20 illustre le Test Bench du registre à décalage.

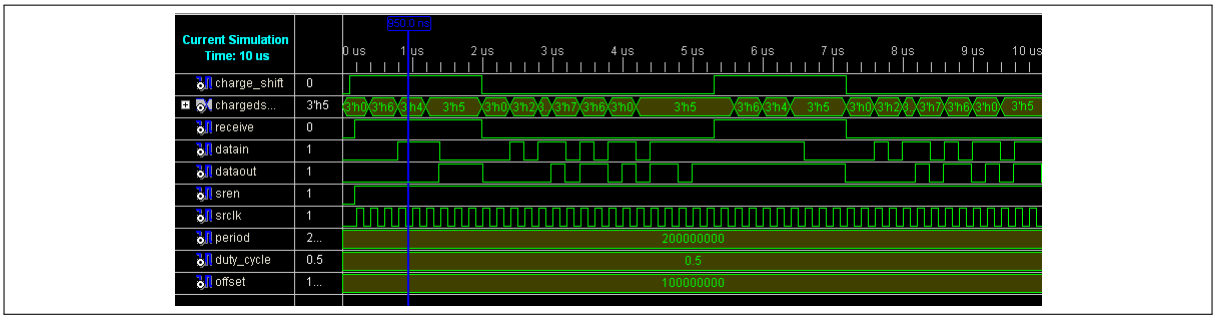


FIGURE 4.20: Test Bench du registre à décalage.

4.4.5.2 Pile LIFO

La Figure 4.21 illustre le Test Bench du registre à décalage.

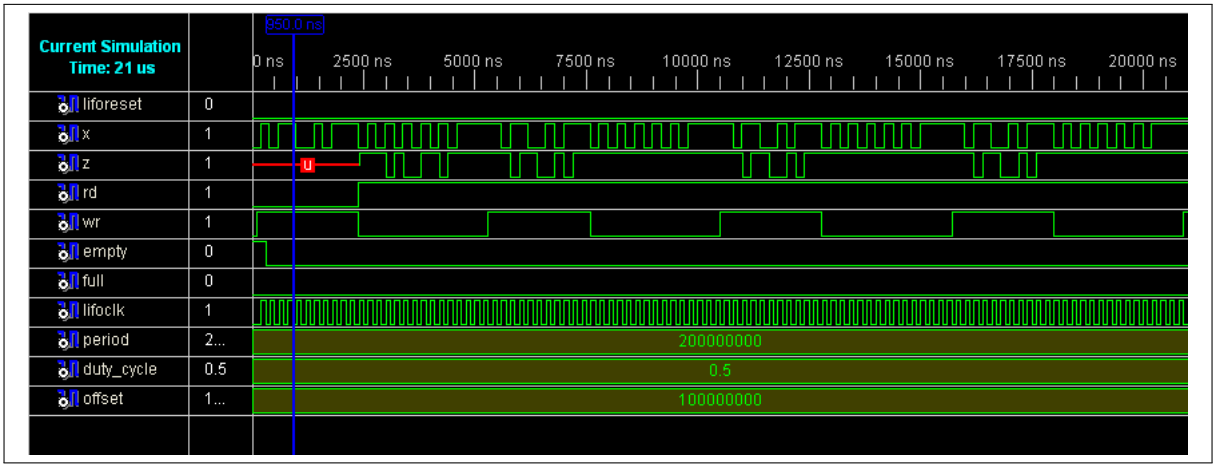


FIGURE 4.21: Test Bench Pile LIFO.

4.4.6 Circuit de contrôle

Le circuit de contrôle a été réalisé sous forme d'un seul fichier décrivant les connexions entres les différents blocs déclarés comme composants décrits dans des fichiers indépendants. Il contient aussi la machine d'état permettant de passer d'une étape à l'autre, comme par exemple de passer du calcul des métriques dans les papillons à l'étape du Trace Back. Il est constitué de plusieurs processus dont le fonctionnement s'enchaine grâce à des signaux de conditions ou Flags.

La structure est celle décrite dans la Figure 4.22.

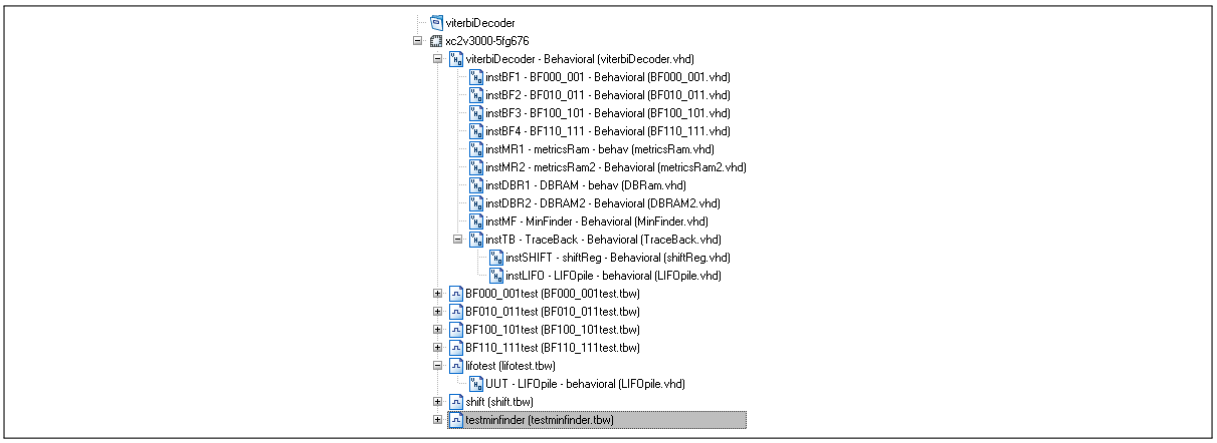


FIGURE 4.22: Structure du décodeur.

Conclusion et perspectives

L'objectif de ce mémoire est la conception et la mise en œuvre d'un décodeur de VITERBI à décision ferme. Nous avons décrit le principe de fonctionnement de celui-ci, ainsi que l'architecture que nous avons décidé d'adopter.

Nous avons effectué après une étude théorique, dans laquelle nous avons rappelé les principes de base de la communication numérique, de la théorie de l'information, de la théorie des codes correcteurs d'erreur et plus particulièrement, des codeurs Convolutionnels. Nous avons aussi étudié la théorie concernant le décodeur de Viterbi et ses variantes. Nous avons traité la stratégie de trace back et celle du registre exchange.

Nos simulations MATLAB ont confirmé le bon fonctionnement de notre décodeur et de son architecture.

A l'aide de ce logiciel, nous avons pu atteindre nos objectifs de simulation. Nous avons en effet simulé le décodeur de VITERBI à stratégie de trace back mais aussi celui à stratégie registre exchange. De plus nous avons réussi à simuler un décodeur dont les paramètres sont définissables par l'utilisateur. Cette dernière simulation à été appliquées pour plusieurs techniques de communication : CASSINI, CDMA 2000, UWB (802.15).

Les applications du décodeur de VITERBI sont très variées et nécessitent l'adaptation des paramètres de ce dernier afin de pouvoir l'adapter à toutes sortes de communications. Par ailleurs il n'existe pas de lois explicites ou d'études indiquant le choix optimal de ces paramètres. D'où l'idée de simuler sur MATLAB un décodeur paramétrable.

Nous avons écrit des programmes VHDL pour le codeur Convolutionnel choisi, et avons aussi implémenté ce dernier sur carte FPGA SPARTAN 3E. Nous avons aussi écrit le programme VHDL du décodeur de VITERBI suivant l'architecture choisie dans notre projet, et nous avons simulé à l'aide de chronogrammes le fonctionnement des différents blocs constituant le décodeur.

De plus nous avons tenu à construire une architecture (en VHDL), permettant de décoder l'information en continue. En effet le jeu des ram des métriques ainsi que des ram des bits de décision permet d'avoir toujours une ram disponible pour l'unité de calcul des métriques, et une autres ram pour l'unité de trace back. Et cela nous permet de ne pas interrompre ou perdre les données reçues lorsque le décodeur est entrain de remonter le treillis.

A l'avenir, il serait pertinent d'apporter des modifications à la logique que nous avons utilisée, car celle-ci peut être optimisée afin de la rendre plus rapide et moins gourmande

en ressources.

Il serait aussi pertinent de penser à construire un décodeur à décision douce qui utilise la distance euclidienne au lieu de la distance de Hamming définie dans ce mémoire.

Bibliographie

[1] Channel coding for telecommunications. *John Wiley and Sons*, 1999.

[2] Jurgen Freudenberger André Neubauer and Volker Kuhn. Coding theory algorithms architectures and applications. *John Wiley and Sons*, 2007.

[3] G. Battail. Pondération des symboles décodés par lálgorithme de viterbi. *Annales des Télécommunications*, 42, n1-2, pp. 31-38, 1987.

[4] Haccoun David Matyas Robert Bhargava, K. Vijay and Nuspl Peter. Digital communication by satellite. *New York N.Y. : J. Wiley and Sons*, 1981.

[5] E. Angui C. Berrou, P. Adde and S. Faudeuil. A low complexity softoutput viterbi decoder architecture. *Proceedings of IEEE ICC, Geneva*, pp. 737-740, 1993.

[6] R. M. Fano. A heuristic discussion of probabilistic decoding. *IEEE Transaction on Information Theory*, IT-9, pp. 64-74, 1963.

[7] G. D. Forney. The viterbi algorithm. *Proceedings of the IEEE*, 3, 1973.

[8] J. Hagenauer and P. Hoeher. A viterbi algorithm with soft-decision outputs and its applications. *IEEE Global Communications Conference, Globecom, Dallas, Texas*, pp. 1680-1686, 1989.

[9] R. Johannesson and K. Sh. Zigangirov. Fundamentals of convolutional coding. *IEEE Press*.

[10] F. Jelinek L. R. Bahl, J. Cocke and J. Raviv. Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Transactions on Information Theory*, IT-20, pp. 284-287, 1974.

[11] J. G. Proakis. Communication systems engineering. *Prentice Hall*, 2nd edition, 2002.

[12] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transaction on Information Theory*, IT-13, pp. 260-269, 1967.

Annexe A

Programmes MATLAB®

A.1 Décodeur $L = 4$, $R = 1/2$, $g_1 = 15_8$ et $g_2 = 13_8$

A.1.1 Trace Back

A.1.1.1 Butterfly1.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Company:         National Polytechnic School (ENP)                        %%%
%%% Engineer:        Abderrezak HADJI                                         %%%
%%%                  Massiva Yamina ZOUAOUI                                   %%%
%%% Create Date:     08/05/20012                                              %%%
%%% Design Name:     Matlab Simulation                                        %%%
%%% Project Name:    Viterbi Decoder (Trace Back)                             %%%
%%% Tool versions:   MATLAB® R2011b(7.13.0.564)                              %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [BD000, BD100, New_PM000, New_PM100, ...
    New_FlagStat000, New_FlagStat100] = ...
    Butterfly1(BM00, BM11, PM000, PM001, FlagStat000, FlagStat001)
% Butterfly 1 (Input States 000 & 001, Output States 000 & 100)
    % Calculate the two New Path Metric
        % 000
            New_PM000_1 = BM00 + PM000;
            New_PM000_2 = BM11 + PM001;
        % 100
            New_PM100_1 = BM11 + PM000;
            New_PM100_2 = BM00 + PM001;
    % Test valid states
        % 000
            if (FlagStat000 == 1) % 000 is valid
```

```
                % 001
                if (FlagStat001 == 1) % 000 & 001 are valid
                    % Compare
                        % 000
                            if (New_PM000_1 <= New_PM000_2)
                                BD000 = 0;
                            else
                                BD000 = 1;
                            end
                        % 100
                            if (New_PM100_1 <= New_PM100_2)
                                BD100 = 0;
                            else
                                BD100 = 1;
                            end
                    % Next Flags
                        New_FlagStat000 = 1;
                        New_FlagStat100 = 1;
                else % 000 is valid & 001 isn't
                    BD000 = 0;
                    BD100 = 0;
                    New_FlagStat000 = 1;
                    New_FlagStat100 = 1;
                end
            else % 000 isn't valid
                if (FlagStat001 == 1) % 000 isn't valid & 001 is
                    BD000 = 1;
                    BD100 = 1;
                    New_FlagStat000 = 1;
                    New_FlagStat100 = 1;
                else % 000 & 001 are not valid
                    New_PM000_1 = 0;
                    New_PM100_1 = 0;
                    BD000 = 0;
                    BD100 = 0;
                    New_FlagStat000 = 0;
                    New_FlagStat100 = 0;
                end
            end
        % Multiplex
            % 000
                if (BD000 == 0)
```

```
                New_PM000 = New_PM000_1;
            else
                New_PM000 = New_PM000_2;
            end
        % 100
            if (BD100 == 0)
                New_PM100 = New_PM100_1;
            else
                New_PM100 = New_PM100_2;
            end
end
```

A.1.1.2 Butterfly2.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Company:         National Polytechnic School (ENP)                        %%%
%%% Engineer:        Abderrezak HADJI                                         %%%
%%%                  Massiva Yamina ZOUAOUI                                   %%%
%%% Create Date:     08/05/20012                                              %%%
%%% Design Name:     Matlab Simulation                                        %%%
%%% Project Name:    Viterbi Decoder (Trace Back)                             %%%
%%% Tool versions:   MATLAB® R2011b(7.13.0.564)                              %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [BD001, BD101, New_PM001, New_PM101, ...
    New_FlagStat001, New_FlagStat101] = ...
    Butterfly2(BM01, BM10, PM010, PM011, FlagStat010, FlagStat011)
% Butterfly 2 (Input States 010 & 011, Output States 001 & 101)
    % Calculate the two New Path Metric
        % 001
            New_PM001_1 = BM01 + PM010;
            New_PM001_2 = BM10 + PM011;
        % 101
            New_PM101_1 = BM10 + PM010;
            New_PM101_2 = BM01 + PM011;
    % Test valid state
        % 010
            if (FlagStat010 == 1) % 010 is valid
                % 011
                if (FlagStat011 == 1) % 010 & 011 are valid
                    % Compare
                        % 001
                            if (New_PM001_1 <= New_PM001_2)
```

```
                        BD001 = 0;
                    else
                        BD001 = 1;
                    end
                % 101
                    if (New_PM101_1 <= New_PM101_2)
                        BD101 = 0;
                    else
                        BD101 = 1;
                    end
                % Next Flags
                    New_FlagStat001 = 1;
                    New_FlagStat101 = 1;
            else % 010 is valid & 011 isn't
                BD001 = 0;
                BD101 = 0;
                New_FlagStat001 = 1;
                New_FlagStat101 = 1;
            end
        else % 010 isn't valid
            if (FlagStat011 == 1) % 010 isn't valid & 011 is
                BD001 = 1;
                BD101 = 1;
                New_FlagStat001 = 1;
                New_FlagStat101 = 1;
            else % 010 & 011 are not valid
                New_PM001_1 = 0;
                New_PM101_1 = 0;
                BD001 = 0;
                BD101 = 0;
                New_FlagStat001 = 0;
                New_FlagStat101 = 0;
            end
        end
% Multiplex
    % 001
        if (BD001 == 0)
            New_PM001 = New_PM001_1;
        else
            New_PM001 = New_PM001_2;
        end
    % 101
```

```
            if (BD101 == 0)
                New_PM101 = New_PM101_1;
            else
                New_PM101 = New_PM101_2;
            end
end
```

A.1.1.3 Butterfly3.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Company:         National Polytechnic School (ENP)                        %%%
%%% Engineer:        Abderrezak HADJI                                         %%%
%%%                  Massiva Yamina ZOUAOUI                                   %%%
%%% Create Date:     08/05/20012                                              %%%
%%% Design Name:     Matlab Simulation                                        %%%
%%% Project Name:    Viterbi Decoder (Trace Back)                             %%%
%%% Tool versions:   MATLAB® R2011b(7.13.0.564)                              %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [BD010, BD110, New_PM010, New_PM110, ...
    New_FlagStat010, New_FlagStat110] = ...
    Butterfly3(BM10, BM01, PM100, PM101, FlagStat100, FlagStat101)
% Butterfly 3 (Input States 100 & 101, Output States 010 & 110)
    % Calculate the two New Path Metric
        % 010
            New_PM010_1 = BM10 + PM100;
            New_PM010_2 = BM01 + PM101;
        % 110
            New_PM110_1 = BM01 + PM100;
            New_PM110_2 = BM10 + PM101;
    % Test valid state
        % 100
            if (FlagStat100 == 1) % 100 is valid
                % 101
                if (FlagStat101 == 1) % 100 & 101 are valid
                    % Compare
                        % 010
                            if (New_PM010_1 <= New_PM010_2)
                                BD010 = 0;
                            else
                                BD010 = 1;
                            end
                        % 110
```

```
                        if (New_PM110_1 <= New_PM110_2)
                            BD110 = 0;
                        else
                            BD110 = 1;
                        end
                    % Next Flags
                        New_FlagStat010 = 1;
                        New_FlagStat110 = 1;
                else % 100 is valid & 101 isn't
                    BD010 = 0;
                    BD110 = 0;
                    New_FlagStat010 = 1;
                    New_FlagStat110 = 1;
                end
            else % 100 isn't valid
                if (FlagStat101 == 1) % 100 isn't valid & 101 is
                    BD010 = 1;
                    BD110 = 1;
                    New_FlagStat010 = 1;
                    New_FlagStat110 = 1;
                else % 100 & 101 are not valid
                    New_PM010_1 = 0;
                    New_PM110_1 = 0;
                    BD010 = 0;
                    BD110 = 0;
                    New_FlagStat010 = 0;
                    New_FlagStat110 = 0;
                end
            end
        % Multiplex
            % 010
                if (BD010 == 0)
                    New_PM010 = New_PM010_1;
                else
                    New_PM010 = New_PM010_2;
                end
            % 110
                if (BD110 == 0)
                    New_PM110 = New_PM110_1;
                else
                    New_PM110 = New_PM110_2;
                end
```

```
end
```

A.1.1.4 Butterfly4.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Company:         National Polytechnic School (ENP)                        %%%
%%% Engineer:        Abderrezak HADJI                                         %%%
%%%                  Massiva Yamina ZOUAOUI                                   %%%
%%% Create Date:     08/05/20012                                              %%%
%%% Design Name:     Matlab Simulation                                        %%%
%%% Project Name:    Viterbi Decoder (Trace Back)                             %%%
%%% Tool versions:   MATLAB® R2011b(7.13.0.564)                              %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [BD011, BD111, New_PM011, New_PM111, ...
    New_FlagStat011, New_FlagStat111] = ...
    Butterfly4(BM11, BM00, PM110, PM111, FlagStat110, FlagStat111)
% Butterfly 4 (Input States 110 & 111, Output States 011 & 111)
    % Calculate the two New Path Metric
        % 011
            New_PM011_1 = BM11 + PM110;
            New_PM011_2 = BM00 + PM111;
        % 111
            New_PM111_1 = BM00 + PM110;
            New_PM111_2 = BM11 + PM111;
    % Test valid state
        % 110
            if (FlagStat110 == 1) % 110 is valid
                % 110
                if (FlagStat111 == 1) % 110 & 111 are valid
                    % Compare
                        % 011
                            if (New_PM011_1 <= New_PM011_2)
                                BD011 = 0;
                            else
                                BD011 = 1;
                            end
                        % 111
                            if (New_PM111_1 <= New_PM111_2)
                                BD111 = 0;
                            else
                                BD111 = 1;
                            end
```

```
                    % Next Flags
                        New_FlagStat011 = 1;
                        New_FlagStat111 = 1;
                else % 110 is valid & 111 isn't
                    BD011 = 0;
                    BD111 = 0;
                    New_FlagStat011 = 1;
                    New_FlagStat111 = 1;
                end
            else % 110 isn't valid
                if (FlagStat111 == 1) % 110 isn't valid & 111 is
                    BD011 = 1;
                    BD111 = 1;
                    New_FlagStat011 = 1;
                    New_FlagStat111 = 1;
                else % 110 & 111 are not valid
                    New_PM011_1 = 0;
                    New_PM111_1 = 0;
                    BD011 = 0;
                    BD111 = 0;
                    New_FlagStat011 = 0;
                    New_FlagStat111 = 0;
                end
            end
    % Multiplex
        % 011
            if (BD011 == 0)
                New_PM011 = New_PM011_1;
            else
                New_PM011 = New_PM011_2;
            end
        % 111
            if (BD111 == 0)
                New_PM111 = New_PM111_1;
            else
                New_PM111 = New_PM111_2;
            end
end
```

A.1.1.5 Viterbi_TB.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%% Company:         National Polytechnic School (ENP)                    %%%
%%% Engineer:        Abderrezak HADJI                                     %%%
%%%                  Massiva Yamina ZOUAOUI                               %%%
%%% Create Date:     08/05/20012                                          %%%
%%% Design Name:     Matlab Simulation                                    %%%
%%% Project Name:    Viterbi Decoder (Trace Back)                         %%%
%%% Tool versions:   MATLAB® R2011b(7.13.0.564)                          %%%
%%% Dependencies:    Butterfly1.m                                         %%%
%%%                  Butterfly2.m                                         %%%
%%%                  Butterfly3.m                                         %%%
%%%                  Butterfly4.m                                         %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [CodedMessage, TransmitedMessage, SequenceDecodee] = ...
    Viterbi_TB(StreamToTransmit, Length_TraceBack)
%% Initialization
    Length_StreamToTransmit = length(StreamToTransmit);
    if Length_StreamToTransmit < Length_TraceBack
        error(['La longueur du message doit ' ...
            'être superieur à la longueur du Trace Back']);
    end
    if any(any(~(StreamToTransmit == '0' | StreamToTransmit == '1')))
        error(message('MATLAB:bin2dec:IllegalBinaryString'))
    end
    DataToTransmit_Vector = zeros(1, Length_StreamToTransmit);
    CodedMessage_Vector = zeros(1, 2*Length_StreamToTransmit);
    TransmitedMessage_Vector = zeros(1, 2*Length_StreamToTransmit);
    Coder_Memory = [0 0 0 0];
    FlagState = [1 0 0 0 0 0 0 0];
    for i = 1 : Length_StreamToTransmit
        DataToTransmit_Vector(i) = str2double(StreamToTransmit(i));
    end
    ROM_BranchMetric = [0 1 1 2; 1 0 2 1; 1 2 0 1; 2 1 1 0];
    RAM_Metrics = zeros(8, Length_TraceBack + 1);
    RAM_Decision = zeros(8, Length_TraceBack);
    i = 1;

%% Main Program
    while (i <= Length_StreamToTransmit)
        % Convolutionel Encoder
            % Shift to right
                Coder_Memory(2:4) = Coder_Memory(1:3);
                Coder_Memory(1) = DataToTransmit_Vector(i);
```

```
        % Calculate S1 and S2
            S1 = xor(xor(Coder_Memory(1), Coder_Memory(2)), ...
                                                    Coder_Memory(4));
            S2 = xor(xor(Coder_Memory(1), Coder_Memory(3)), ...
                                                    Coder_Memory(4));
        % Save S1 and S2
            CodedMessage_Vector(2*i-1) = S1;
            CodedMessage_Vector(2*i) = S2;
        % Transform the Vector's Coded Message to a string
            CodedMessage = sprintf('%d%d ', CodedMessage_Vector);
    % Error Generator
        Err_Pos = randi(50);
        if Err_Pos <= 3
            if Err_Pos <= 1
                S1 = not(S1);
            else
                S2 = not(S2);
            end
        end
        TransmitedMessage_Vector(2*i-1) = S1;
        TransmitedMessage_Vector(2*i) = S2;
        TransmitedMessage = sprintf('%d%d ', TransmitedMessage_Vector);
    % Viterbi Decoder (TB)
        Address = mod(i, Length_TraceBack);
        if Address == 0
            Address = Length_TraceBack;
        end
        % BMU
            % Initialization
                Branch_Metrics = ROM_BranchMetric(2*S1 + S2 + 1, :);
            % Calculate Branch Metric
                BM00 = Branch_Metrics(1);
                BM01 = Branch_Metrics(2);
                BM10 = Branch_Metrics(3);
                BM11 = Branch_Metrics(4);
        % PMU
            % Initialization
                PM000 = RAM_Metrics(1, Address);
                PM001 = RAM_Metrics(2, Address);
                PM010 = RAM_Metrics(3, Address);
                PM011 = RAM_Metrics(4, Address);
                PM100 = RAM_Metrics(5, Address);
```

```
    PM101 = RAM_Metrics(6, Address);
    PM110 = RAM_Metrics(7, Address);
    PM111 = RAM_Metrics(8, Address);
    FlagStat000 = FlagState(1);
    FlagStat001 = FlagState(2);
    FlagStat010 = FlagState(3);
    FlagStat011 = FlagState(4);
    FlagStat100 = FlagState(5);
    FlagStat101 = FlagState(6);
    FlagStat110 = FlagState(7);
    FlagStat111 = FlagState(8);
% Butterfly 1
    [BD000, BD100, New_PM000, New_PM100, ...
        New_FlagStat000, New_FlagStat100] = ...
        Butterfly1(BM00, BM11, PM000, PM001, ...
        FlagStat000, FlagStat001);
% Butterfly 2
    [BD001, BD101, New_PM001, New_PM101, ...
        New_FlagStat001, New_FlagStat101] = ...
        Butterfly2(BM01, BM10, PM010, PM011, ...
        FlagStat010, FlagStat011);
% Butterfly 3
    [BD010, BD110, New_PM010, New_PM110, ...
        New_FlagStat010, New_FlagStat110] = ...
        Butterfly3(BM10, BM01, PM100, PM101, ...
        FlagStat100, FlagStat101);
% Butterfly 4
    [BD011, BD111, New_PM011, New_PM111, ...
        New_FlagStat011, New_FlagStat111] = ...
        Butterfly4(BM11, BM00, PM110, PM111, ...
        FlagStat110, FlagStat111);
% Save
    RAM_Decision(:, Address) = [BD000 BD001 ...
                                BD010 BD011 ...
                                BD100 BD101 ...
                                BD110 BD111]';
    RAM_Metrics(:, Address + 1) = ...
        [New_PM000 New_PM001 ...
         New_PM010 New_PM011 ...
         New_PM100 New_PM101 ...
         New_PM110 New_PM111]';
    FlagState = ...
```

```
                    [New_FlagStat000 New_FlagStat001 ...
                     New_FlagStat010 New_FlagStat011 ...
                     New_FlagStat100 New_FlagStat101 ...
                     New_FlagStat110 New_FlagStat111];
            % DU
            if (mod(i, Length_TraceBack) == 0 || ...
                                              i == Length_StreamToTransmit)
                % Initialization
                    end_i = Address;
                    Last_Minimum = min(RAM_Metrics(:, end_i + 1));
                    CurrentState = dec2bin(find(RAM_Metrics(:,...
                                        end_i + 1)== Last_Minimum) - 1, 3);
                % Decoding
                    for j = end_i:-1:1
                        SequenceDecodee(...
                           floor((i-1)/Length_TraceBack)*...
                           Length_TraceBack + j) = CurrentState(1);
                        Previous_Stat = [CurrentState dec2bin(...
                            RAM_Decision(bin2dec(...
                            CurrentState)+1, j))];
                        CurrentState = Previous_Stat(2:4);
                    end
                % Initialization RAM_Metrics
                    RAM_Metrics(:, 1) = RAM_Metrics(:, end_i + 1);
            end
            % Set up time
                i = i + 1;
    end

%% Compare
    display(['Cmp TransmitedMessage & CodedMessage ' ...
        (dec2bin(TransmitedMessage == CodedMessage))']);
    display(['Cmp SequenceDecodee & StreamToTransmet ' ...
        (dec2bin(SequenceDecodee == StreamToTransmit))']);
end
```

A.1.2 Register Exchange

A.1.2.1 Butterfly1.m

Voir Annexe A.1.1.1.

A.1.2.2 Butterfly2.m

Voir Annexe A.1.1.2.

A.1.2.3 Butterfly3.m

Voir Annexe A.1.1.3.

A.1.2.4 Butterfly4.m

Voir Annexe A.1.1.4.

A.1.2.5 Viterbi_RE.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Company:        National Polytechnic School (ENP)                   %%%
%%% Engineer:       Massiva Yamina ZOUAOUI                              %%%
%%%                 Abderrezak HADJI                                    %%%
%%% Create Date:    08/05/20012                                         %%%
%%% Design Name:    Matlab Simulation                                   %%%
%%% Project Name:   Viterbi Decoder (Register Exchange)                 %%%
%%% Tool versions:  MATLAB® R2011b(7.13.0.564)                          %%%
%%% Dependencies:   Butterfly1.m                                        %%%
%%%                 Butterfly2.m                                        %%%
%%%                 Butterfly3.m                                        %%%
%%%                 Butterfly4.m                                        %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [CodedMessage, TransmitedMessage, SequenceDecodee] = ...
                                               Viterbi_RE(StreamToTransmit)
%% Initialization
    Length_StreamToTransmit = length(StreamToTransmit);
    Length_RAM = 20;
    DataToTransmit_Vector = zeros(1, Length_StreamToTransmit);
    CodedMessage_Vector = zeros(1, 2*Length_StreamToTransmit);
    TransmitedMessage_Vector = zeros(1, 2*Length_StreamToTransmit);
    Coder_Memory = [0 0 0 0];
    FlagState = [1 0 0 0 0 0 0 0];
    for i = 1 : Length_StreamToTransmit
        DataToTransmit_Vector(i) = str2double(StreamToTransmit(i));
    end
    ROM_BranchMetric = [0 1 1 2; 1 0 2 1; 1 2 0 1; 2 1 1 0];
    RAM_Metrics = zeros(8, Length_RAM);
    RE000 = 0;
```

```
    RE001 = 0;
    RE010 = 0;
    RE011 = 0;
    RE100 = 0;
    RE101 = 0;
    RE110 = 0;
    RE111 = 0;
    i = 1;

%% Main Program
    while (i <= Length_StreamToTransmit)
        % Convolutionel Encoder
            % Shift to right
                Coder_Memory(2:4) = Coder_Memory(1:3);
                Coder_Memory(1) = DataToTransmit_Vector(i);
            % Calculate S1 and S2
                S1 = xor(xor(Coder_Memory(1), Coder_Memory(2)), ...
                                                        Coder_Memory(4));
                S2 = xor(xor(Coder_Memory(1), Coder_Memory(3)), ...
                                                        Coder_Memory(4));
            % Save S1 and S2
                CodedMessage_Vector(2*i-1) = S1;
                CodedMessage_Vector(2*i) = S2;
            % Transform the Vector's Coded Message to a string
                CodedMessage = sprintf('%d%d ', CodedMessage_Vector);
        % Error Generator
            Err_Pos = randi(50);
            if Err_Pos <= 3
                if Err_Pos <= 1
                    S1 = not(S1);
                else
                    S2 = not(S2);
                end
            end
            TransmitedMessage_Vector(2*i-1) = S1;
            TransmitedMessage_Vector(2*i) = S2;
            TransmitedMessage = sprintf('%d%d ', TransmitedMessage_Vector);
        % Viterbi Decoder
            Address = mod(i, Length_RAM);
            if Address == 0
                Address = Length_RAM;
            end
```

```
% BMU
    % Initialization
        Branch_Metrics = ROM_BranchMetric(2*S1 + S2 + 1, :);
    % Calculate Branch Metric
        BM00 = Branch_Metrics(1);
        BM01 = Branch_Metrics(2);
        BM10 = Branch_Metrics(3);
        BM11 = Branch_Metrics(4);
% PMU
    % Initialization
        PM000 = RAM_Metrics(1, Address);
        PM001 = RAM_Metrics(2, Address);
        PM010 = RAM_Metrics(3, Address);
        PM011 = RAM_Metrics(4, Address);
        PM100 = RAM_Metrics(5, Address);
        PM101 = RAM_Metrics(6, Address);
        PM110 = RAM_Metrics(7, Address);
        PM111 = RAM_Metrics(8, Address);
        FlagStat000 = FlagState(1);
        FlagStat001 = FlagState(2);
        FlagStat010 = FlagState(3);
        FlagStat011 = FlagState(4);
        FlagStat100 = FlagState(5);
        FlagStat101 = FlagState(6);
        FlagStat110 = FlagState(7);
        FlagStat111 = FlagState(8);
    % Butterfly 1 (State 000 & 100)
        [BD000, BD100, New_PM000, New_PM100, ...
            New_FlagStat000, New_FlagStat100] = ...
            Butterfly1(BM00, BM11, PM000, PM001, ...
            FlagStat000, FlagStat001);
    % Butterfly 2 (State 001 & 101)
        [BD001, BD101, New_PM001, New_PM101, ...
            New_FlagStat001, New_FlagStat101] = ...
            Butterfly2(BM01, BM10, PM010, PM011, ...
            FlagStat010, FlagStat011);
    % Butterfly 3 (State 010 & 110)
        [BD010, BD110, New_PM010, New_PM110, ...
            New_FlagStat010, New_FlagStat110] = ...
            Butterfly3(BM10, BM01, PM100, PM101, ...
            FlagStat100, FlagStat101);
    % Butterfly 4 (State 011 & 111)
```

```
        [BD011, BD111, New_PM011, New_PM111, ...
            New_FlagStat011, New_FlagStat111] = ...
            Butterfly4(BM11, BM00, PM110, PM111, ...
            FlagStat110, FlagStat111);
    % Update the Path Metrics & Flags
        if Address == Length_RAM
            RAM_Metrics(:, 1) = [New_PM000 New_PM001 ...
                                 New_PM010 New_PM011 ...
                                 New_PM100 New_PM101 ...
                                 New_PM110 New_PM111]';
        else
            RAM_Metrics(:, Address + 1) = ...
                                        [New_PM000 New_PM001 ...
                                         New_PM010 New_PM011 ...
                                         New_PM100 New_PM101 ...
                                         New_PM110 New_PM111]';
        end
        FlagState = [New_FlagStat000 New_FlagStat001 ...
                     New_FlagStat010 New_FlagStat011 ...
                     New_FlagStat100 New_FlagStat101 ...
                     New_FlagStat110 New_FlagStat111];
    % Exchange decoded bits betwen Registers
        % RE000
            if BD000 == 0
                New_RE000 = [RE000 0];
            else
                New_RE000 = [RE001 0];
            end
        % RE001
            if BD001 == 0
                New_RE001 = [RE010 0];
            else
                New_RE001 = [RE011 0];
            end
        % RE010
            if BD010 == 0
                New_RE010 = [RE100 0];
            else
                New_RE010 = [RE101 0];
            end
        % RE011
            if BD011 == 0
```

```
                    New_RE011 = [RE110 0];
                else
                    New_RE011 = [RE111 0];
                end
            % RE100
                if BD100 == 0
                    New_RE100 = [RE000 1];
                else
                    New_RE100 = [RE001 1];
                end
            % RE101
                if BD101 == 0
                    New_RE101 = [RE010 1];
                else
                    New_RE101 = [RE011 1];
                end
            % RE110
                if BD110 == 0
                    New_RE110 = [RE100 1];
                else
                    New_RE110 = [RE101 1];
                end
            % RE111
                if BD111 == 0
                    New_RE111 = [RE110 1];
                else
                    New_RE111 = [RE111 1];
                end
        % Update Registers
            RE000 = New_RE000;
            RE001 = New_RE001;
            RE010 = New_RE010;
            RE011 = New_RE011;
            RE100 = New_RE100;
            RE101 = New_RE101;
            RE110 = New_RE110;
            RE111 = New_RE111;
        % Set up time
            i = i + 1;
end
    % DU
        Decoded = [RE000; RE001; RE010; RE011; ...
```

```
                        RE100; RE101; RE110; RE111];
                % Initialization
                    if Address == Length_RAM
                        Address = 0;
                    end
                    Last_Minimum = min(RAM_Metrics(:, Address + 1));
                    CurrentState = RAM_Metrics(:, ...
                                                Address + 1)== Last_Minimum;
                    SequenceDecodee = Decoded(CurrentState, :);
                    SequenceDecodee = SequenceDecodee(...
                                                 2:length(SequenceDecodee));
                    SequenceDecodee = (dec2bin(SequenceDecodee))';
%% Compare
    display(['Cmp TransmitedMessage & CodedMessage ' ...
                            (dec2bin(TransmitedMessage == CodedMessage))']);
    display(['Cmp SequenceDecodee & StreamToTransmet ' ...
                          (dec2bin(SequenceDecodee == StreamToTransmit))']);
end
```

A.2 Décodeur L, $R = 1/n$ et G

A.2.1 Trace Back

A.2.1.1 Treillis.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Company:         National Polytechnic School (ENP)                       %%%
%%% Engineer:        Abderrezak HADJI                                        %%%
%%%                  Massiva Yamina ZOUAOUI                                  %%%
%%% Create Date:     08/05/20012                                             %%%
%%% Design Name:     Matlab Simulation                                       %%%
%%% Project Name:    Viterbi Decoder (Trace Back)                            %%%
%%% Tool versions:   MATLAB® R2011b(7.13.0.564)                             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function MyTreillis = Treillis(L, S, G)
%% Initialization
    m = L-1;
    n = 2^m;
    States = zeros(n, 1);
    NextStates = zeros(n, 2);
    Coder_Memory = zeros(1, L);
```

```
    Outputs = zeros(n, 2);
    Output = dec2bin(0, S);
%% Computing States
    for i = 1:n
        States(i, 1) = i - 1;
    end
    MyTreillis.States = States;
%% Computing Next States
    for i = 1:n
        NextState_0 = ['0' dec2bin(States(i, :), m)];
        NextState_1 = ['1' dec2bin(States(i, :), m)];
        NextStates(i, 1) = bin2dec(NextState_0(1:m));
        NextStates(i, 2) = bin2dec(NextState_1(1:m));
    end
    MyTreillis.NextStates = NextStates;
%% Computing Outputs
for State = 1:n
    % Encoder initialization
        PresentState = dec2bin(MyTreillis.States(State), m);
        for j=2:L
            Coder_Memory(1, j) = bin2dec(PresentState(j-1));
        end
    % Calculate
    for Input = 0:1
        Coder_Memory(1) = Input;
        for Output_i = 1:S
            PolyGene = find(G(Output_i,:) == '1');
            Code = 0;
            for Mem_i = 1:length(PolyGene)
                Code = xor(Code, Coder_Memory(PolyGene(Mem_i)));
            end
            Output(Output_i) = dec2bin(Code);
        end
        Outputs(State, Input + 1) = bin2dec(Output);
    end
end
MyTreillis.Outputs = Outputs;
end
```

A.2.1.2 ConvolutionalEncoder.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%% Company:         National Polytechnic School (ENP)                    %%%
%%% Engineer:        Abderrezak HADJI                                     %%%
%%%                  Massiva Yamina ZOUAOUI                               %%%
%%% Create Date:     08/05/20012                                          %%%
%%% Design Name:     Matlab Simulation                                    %%%
%%% Project Name:    Viterbi Decoder (Trace Back)                         %%%
%%% Tool versions:   MATLAB® R2011b(7.13.0.564)                          %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function BitStream_OUT = ConvolutionalEncoder(L, S, G, BitStream_IN)
%% Initialization
    Length_StreamToTransmit = length(BitStream_IN);
    OUT_Symbolm = zeros(1, Length_StreamToTransmit);
    DataToTransmit_Vector = zeros(1, Length_StreamToTransmit);
    for i = 1 : Length_StreamToTransmit
        DataToTransmit_Vector(i) = str2double(BitStream_IN(i));
    end
    State = 0;
    BitStream_OUT = '';
%% Set the trellis
    MyTreillis = Treillis(L, S, G);
%% Main Program
    for i=1:Length_StreamToTransmit
        OUT_Symbolm(i) = MyTreillis.Outputs(State + 1, ...
                                            DataToTransmit_Vector(i) + 1);
        State = MyTreillis.NextStates(State + 1, ...
                                            DataToTransmit_Vector(i) + 1);
        BitStream_OUT = [BitStream_OUT dec2bin(OUT_Symbolm(i), S) ' '];
    end
end
```

A.2.1.3 Butterfly.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Company:         National Polytechnic School (ENP)                    %%%
%%% Engineer:        Abderrezak HADJI                                     %%%
%%%                  Massiva Yamina ZOUAOUI                               %%%
%%% Create Date:     08/05/20012                                          %%%
%%% Design Name:     Matlab Simulation                                    %%%
%%% Project Name:    Viterbi Decoder (Trace Back)                         %%%
%%% Tool versions:   MATLAB® R2011b(7.13.0.564)                          %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [BD1, BD2, New_PM1, New_PM2, ...
```

```
    New_FlagState1, New_FlagState2] = ...
    Butterfly(BM1, BM2, PM1, PM2, FlagState1, FlagState2)
% Butterfly 1 (Input States a & b, Output States a' & b')
    % Calculate the two New Path Metric
        % a'
            New_PM1_1 = BM1 + PM1;
            New_PM1_2 = BM2 + PM2;
        % b'
            New_PM2_1 = BM2 + PM1;
            New_PM2_2 = BM1 + PM2;
    % Test valid state
        % a
            if (FlagState1 == 1) % a is valid
                % b
                if (FlagState2 == 1) % a & b are valid
                    % Compare
                        % a'
                            if (New_PM1_1 <= New_PM1_2)
                                BD1 = 0;
                            else
                                BD1 = 1;
                            end
                        % b'
                            if (New_PM2_1 <= New_PM2_2)
                                BD2 = 0;
                            else
                                BD2 = 1;
                            end
                    % Next Flags
                            New_FlagState1 = 1;
                            New_FlagState2 = 1;
                else % a is valid & b isn't
                    BD1 = 0;
                    BD2 = 0;
                    New_FlagState1 = 1;
                    New_FlagState2 = 1;
                end
            else % a isn't valid
                if (FlagState2 == 1) % a isn't valid & b is
                    BD1 = 1;
                    BD2 = 1;
                    New_FlagState1 = 1;
```

```
                    New_FlagState2 = 1;
                else % a & b are not valid
                    New_PM1_1 = 0;
                    New_PM2_1 = 0;
                    BD1 = 0;
                    BD2 = 0;
                    New_FlagState1 = 0;
                    New_FlagState2 = 0;
                end
            end
    % Multiplex
        % a'
            if (BD1 == 0)
                New_PM1 = New_PM1_1;
            else
                New_PM1 = New_PM1_2;
            end
        % b'
            if (BD2 == 0)
                New_PM2 = New_PM2_1;
            else
                New_PM2 = New_PM2_2;
            end
end
```

A.2.1.4 Viterbi_TB.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Company:         National Polytechnic School (ENP)                        %%%
%%% Engineer:        Abderrezak HADJI                                         %%%
%%%                  Massiva Yamina ZOUAOUI                                   %%%
%%% Create Date:     08/05/20012                                              %%%
%%% Design Name:     Matlab Simulation                                        %%%
%%% Project Name:    Viterbi Decoder (Trace Back)                             %%%
%%% Tool versions:   MATLAB® R2011b(7.13.0.564)                             %%%
%%% Dependencies:    Trellis.m                                                %%%
%%%                  Butterfly.m                                              %%%
%%%                  ConvolutionelEncoder.m                                   %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [CodedMessage, TransmitedMessage, SequenceDecodee] = ...
    Viterbi_TB(L, S, G, Length_TraceBack, BitStream_IN)
%% Initialization
```

```
    m = L-1;
    n_State = 2^m;
    Length_BitBitStream_IN = length(BitStream_IN);
    FlagState = zeros(1, n_State);
    FlagState(1) = 1;
    New_FlagStats = zeros(1, n_State);
    RAM_Metrics = zeros(n_State, Length_TraceBack + 1);
    RAM_Decision = zeros(n_State, Length_TraceBack);
    New_PathMetrics = zeros(1, n_State);
    BitsDecision = zeros(1, n_State);
    TransmitedMessage = '';
    i = 1;
%% Generate the treillis
    MyTreillis = Treillis(L, S, G);
%% Main Program
    % Coding
        CodedMessage = ConvolutionalEncoder(L, S, G, BitStream_IN);
        CodedMessage_Vector = CodedMessage(not(CodedMessage == ' '));
    % Decoding
        while (i <= Length_BitBitStream_IN)
            % Initialization
                Address = mod(i, Length_TraceBack);
                if Address == 0
                    Address = Length_TraceBack;
                end
                PathMetrics = RAM_Metrics(:, Address);
            % Error generator
                Recived_Symbole = CodedMessage_Vector((i-1)*S + 1 : i*S);
                Err = randi(100);
                if Err <= 20
                    Err_Pos = randi(S);
                    if (Err_Pos == 0)
                        Err_Pos = 1;
                    end
                    Recived_Symbole(Err_Pos) = dec2bin(not(bin2dec(...
                                                Recived_Symbole(Err_Pos))));
                end
                TransmitedMessage = [TransmitedMessage ...
                                     Recived_Symbole ' '];
            for i_State = 1:2:n_State
                % BMU
                    Symbolm1 = dec2bin(MyTreillis.Outputs(i_State, 1), S);
```

```
            Symbolm2 = dec2bin(MyTreillis.Outputs(i_State, 2), S);
            BM1 = sum(not(Recived_Symbole == Symbolm1));
            BM2 = sum(not(Recived_Symbole == Symbolm2));
        % PMU
            PM1 = PathMetrics(i_State);
            PM2 = PathMetrics(i_State + 1);
            FlagState1 = FlagState(i_State);
            FlagState2 = FlagState(i_State + 1);
            [BD1, BD2, New_PM1, New_PM2, ...
                New_FlagStat1, New_FlagStat2] = ...
                Butterfly(BM1, BM2, PM1, PM2, ...
                FlagState1, FlagState2);
            NextState1 = MyTreillis.NextStates(i_State, 1) + 1;
            NextState2 = MyTreillis.NextStates(i_State, 2) + 1;
            New_PathMetrics(NextState1) = New_PM1;
            New_PathMetrics(NextState2) = New_PM2;
            BitsDecision(NextState1) = BD1;
            BitsDecision(NextState2) = BD2;
            New_FlagStats(NextState1) = New_FlagStat1;
            New_FlagStats(NextState2) = New_FlagStat2;
    end
    RAM_Decision(:, Address) = BitsDecision;
    RAM_Metrics(:, Address + 1) = New_PathMetrics;
    FlagState = New_FlagStats;
        % DU
            if (mod(i, Length_TraceBack) == 0 || ...
                                        i == Length_BitBitStream_IN)
                % Initialization
                    end_i = Address;
                    Last_Minimum = min(RAM_Metrics(:, end_i + 1));
                    CurrentState = dec2bin(find(RAM_Metrics(:, ...
                                end_i + 1)== Last_Minimum) - 1, m);
                % Decoding
                    for j = end_i:-1:1
                        SequenceDecodee(...
                           floor((i-1)/Length_TraceBack)*...
                           Length_TraceBack + j) = CurrentState(1);
                        Previous_Stat = [CurrentState ...
                            dec2bin(RAM_Decision(bin2dec(...
                            CurrentState)+1, j))];
                        CurrentState = Previous_Stat(2:L);
                    end
```

```
                        % Initialization RAM_Metrics
                            RAM_Metrics(:, 1) = RAM_Metrics(:, end_i + 1);
                    end
                % Set up time
                    i = i + 1;
            end
%% Compare
            display(['Cmp TransmitedMessage & CodedMessage ' ...
                (dec2bin(TransmitedMessage == CodedMessage))']);
            display(['Cmp SequenceDecodee & StreamToTransmet ' ...
                (dec2bin(SequenceDecodee == BitStream_IN))']);
end
```

Annexe B

Programmes VHDL

B.1 Coder.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.all;
use ieee.std_logic_unsigned.all;
entity ConvoCoder is
  port (
    clk          : in  std_logic;
    d_in         : in  std_logic;
    S1       : out std_logic;
    S2       : out std_logic;
    );
end ConvoCoder;
architecture Behavioral of ConvoCoder is
  signal reg : std_logic_vector (3 downto 0):="0000";
signal is1: std_logic:='0';
signal is2: std_logic:='0';
begin
clk_division : process (clk, clk_divider)
begin
  if (clk = '1' and clk'Event) then
    clk_divider <= clk_divider + 1;
  end if;
    slow_clk <= clk_divider(24);
end process;
  process (clk)
  begin
    if rising_edge (clk) then
      reg(0) <= d_in;
      reg(3 downto 1) <= reg(2 downto 0);
    end if;
  end process;
```

```
  is1<=reg(0) xor reg(1) xor reg(3);
  is2<=reg(0) xor reg(2) xor reg(3);
  S1 <= is1;
  S2 <= is2;
end Behavioral;
```

B.2 Papillons

B.2.1 BF000_001.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.all;
entity BF000_001 is
    Port (
      BF1reset: in std_logic;
      clk: in std_logic;
      BF1en: in std_logic;
      IF000: in  std_logic;
      IF001: in  std_logic;
      RD: in std_logic_vector(1 downto 0);
      PMs: in std_logic_vector(2 downto 0);
      PM001 : in  std_logic_vector(7 downto 0);
      PM000 : in  std_logic_vector(7 downto 0);
      OF000: out  std_logic:='0';
      OF100: out  std_logic:='0';
      BF1r: out std_logic:='0';
      DB000 : out  std_logic;
      DB100 : out  std_logic;
      Newpm000 : out  std_logic_vector(7 downto 0):="00000000";
      Newpm100 : out  std_logic_vector(7 downto 0):="00000000"
      );
end BF000_001;
architecture Behavioral of BF000_001 is
SIGNAL  BM00 :   std_logic_vector(1 downto 0);
SIGNAL  BM11 :   std_logic_vector(1 downto 0);
SIGNAL intpm000_11: std_logic_vector(7 downto 0);
SIGNAL intpm000_00: std_logic_vector(7 downto 0);
SIGNAL intpm100_11: std_logic_vector(7 downto 0);
SIGNAL intpm100_00: std_logic_vector(7 downto 0);
begin
process (RD)
  begin
    BF1R <='0';
    if (RD ="00") then
```

```
      BM00 <= "00";
      BM11 <= "10";
    else
      if( RD ="10" or RD ="01") then
        BM00 <= "01";
        BM11 <= "01";
      else
        BM00 <= "10";
        BM11 <= "00";
      end if;
    end if;
    if (IF000 = '1') then
      OF000 <= '1';
      OF100 <= '1';
      if (IF001 = '1') then
          intpm000_00 <= PM000 + BM00 ;
          intpm100_11 <= PM000 + BM11 ;
          intpm000_11 <= PM001 + BM11 ;
          intpm100_00 <= PM001 + BM00 ;
        if (intpm000_00 < intpm000_11) then
          NEwpm000 <= intpm000_00;
          DB000 <='0';
        else
          NEwpm000 <= intpm000_11;
          DB000 <='1';
        end if;
        if (intpm100_00 < intpm100_11) then
          NEwpm100 <= intpm100_00;
          DB100 <='1';
        else
          NEwpm100 <= intpm100_11;
          DB100 <='0';
        end if;
      else
        intpm000_00 <= PM000 + BM00 ;
        intpm100_11 <= PM000 + BM11 ;
        NEwpm100 <= intpm100_00;
        DB100 <='1';
        NEwpm000 <= intpm000_11;
        DB000 <='1';
      end if;
    else
      if (IF001 = '1') then
        OF000 <= '1';
        OF100 <= '1';
        intpm000_11 <= PM001 + BM11 ;
        intpm100_00 <= PM001 + BM00 ;
        NEwpm000 <= intpm000_11;
        DB000 <='1';
        NEwpm100 <= intpm100_00;
```

```
        DB100 <='1';
      else
        OF000 <= '0';
        OF100 <= '0';
      end if;
    end if;
    BF1r <='1';
end process;
end Behavioral;
```

B.2.2 BF010_011.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.all;
entity BF010_011 is
    Port (
      BF2reset: in std_logic;--reset
      clk: in std_logic;--clock
      BF2en: in std_logic;--enable
      IF010: in  std_logic;--input flag
      IF011: in  std_logic;--input flag
      RD: in std_logic_vector(1 downto 0);--received data
      PMs: in std_logic_vector(2 downto 0);--path metric select
      PM010 : in  std_logic_vector(7 downto 0);--previous path metric
      PM011 : in  std_logic_vector(7 downto 0);--previous path metric
      OF001: out  std_logic:='0';--output flag
      OF101: out  std_logic:='0';--output flag
      BF2r: out std_logic:='0';--bf2 ready
      DB001 : out  std_logic;--decision bit
      DB101 : out  std_logic;--decision bit
      Newpm001 : out  std_logic_vector(7 downto 0):="00000000";--new path
          metric
      Newpm101 : out  std_logic_vector(7 downto 0):="00000000" --new path
          metric
      );
end BF010_011;
architecture Behavioral of BF010_011 is
SIGNAL  BM01 :   std_logic_vector(1 downto 0);
SIGNAL  BM10 :   std_logic_vector(1 downto 0);
SIGNAL intpm001_01: std_logic_vector(7 downto 0);
SIGNAL intpm001_10: std_logic_vector(7 downto 0);
SIGNAL intpm101_10: std_logic_vector(7 downto 0);
SIGNAL intpm101_01: std_logic_vector(7 downto 0);
begin
process (RD)
```

```
begin
  BF2R <='0';
  if (RD ="10") then
    BM10 <= "00";
    BM01 <= "10";
  else
    if( RD ="11" or RD ="00") then
      BM01 <= "01";
      BM10 <= "01";
    else
      BM10 <= "10";
      BM01 <= "00";
    end if;
  end if;
  if (IF010 = '1') then
    OF001 <= '1';
    OF101 <= '1';
    if (IF011 = '1') then
      intpm001_10 <= PM011 + BM10 ;
      intpm101_01 <= PM011 + BM01 ;
      intpm001_01 <= PM010 + BM01 ;
      intpm101_10 <= PM010 + BM10 ;
      if (intpm001_10 < intpm001_01) then
         NEwpm001 <= intpm001_10;
         DB001 <='1';
      else
         NEwpm001 <= intpm001_01;
         DB001 <='0';
      end if;
      if (intpm101_10 < intpm101_01) then
        NEwpm101 <= intpm101_10;
        DB101 <='0';
      else
        NEwpm101 <= intpm101_01;
        DB101 <='1';
      end if;
    else
      intpm001_01 <= PM010 + BM01 ;
      intpm101_10 <= PM010 + BM10 ;
      NEwpm101 <= intpm101_10;
      DB101 <='0';
      NEwpm001 <= intpm001_01;
      DB001 <='0';
    end if;
  else
    if (IF011 = '1') then
      OF001 <= '1';
      OF101 <= '1';
      intpm001_10 <= PM011 + BM10 ;
      intpm101_01 <= PM011 + BM01 ;
```

```
        NEwpm001 <= intpm001_10;
        DB001 <='1';
        NEwpm101 <= intpm101_01;
        DB101 <='1';
      else
        OF001 <= '0';
        OF101 <= '0';
      end if;
    end if;
    BF2r <='1';
end process;
end Behavioral;
```

B.2.3 BF100_101.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.all;
entity BF100_101 is
    Port (
      BF3reset: in std_logic;
      clk: in std_logic;
      BF3en: in std_logic;
      IF100: in  std_logic;
      IF101: in  std_logic;
      RD: in std_logic_vector(1 downto 0);
      PMs: in std_logic_vector(2 downto 0);
      PM100 : in  std_logic_vector(7 downto 0);
      PM101 : in  std_logic_vector(7 downto 0);
      OF010: out  std_logic;
      OF110: out  std_logic;
      BF3r: out std_logic;
      DB010 : out  std_logic;
      DB110 : out  std_logic;
      Newpm010 : out  std_logic_vector(7 downto 0);
      Newpm110 : out  std_logic_vector(7 downto 0)
      );
end BF100_101 ;
architecture Behavioral of BF100_101  is
SIGNAL  BM01 :   std_logic_vector(1 downto 0);
SIGNAL  BM10 :   std_logic_vector(1 downto 0);
SIGNAL  BMready:  std_logic;
SIGNAL intpm010_01: std_logic_vector(7 downto 0);
SIGNAL intpm010_10: std_logic_vector(7 downto 0);
SIGNAL intpm110_10: std_logic_vector(7 downto 0);
SIGNAL intpm110_01: std_logic_vector(7 downto 0);
```

```
begin
process (RD)
begin
  BF3R <='0';
  if (RD ="10") then
    BM10 <= "00";
    BM01 <= "10";
  else
    if( RD ="11" or RD ="00") then
      BM01 <= "01";
      BM10 <= "01";
    else
      BM10 <= "10";
      BM01 <= "00";
    end if;
  end if;
  BMready<='1';
end process;
process
begin
  wait until BMready ='1';
  if (IF100 = '1') then
    OF010 <= '1';
    OF110 <= '1';
    if (IF101 = '1') then
      intpm010_01 <= PM101 + BM01 ;
      intpm110_10 <= PM101 + BM10 ;
      intpm010_10 <= PM100 + BM10 ;
      intpm110_01 <= PM100 + BM01 ;
      if (intpm010_10 < intpm010_01) then
        NEwpm010 <= intpm010_10;
        DB010 <='0';
      else
        NEwpm010 <= intpm010_01;
        DB010 <='1';
      end if;
      if (intpm110_10 < intpm110_01) then
         NEwpm110 <= intpm110_10;
         DB110 <='1';
      else
         NEwpm110 <= intpm110_01;
         DB110 <='0';
      end if;
    else
      intpm010_10 <= PM100 + BM10 ;
      intpm110_01 <= PM100 + BM01 ;
      NEwpm010 <= intpm010_10;
      DB010 <='0';
      NEwpm110 <= intpm110_01;
      DB110 <='0';
```

```
    end if;
  else
    if (IF101 = '1') then
      OF010 <= '1';
      OF110 <= '1';
      intpm010_01 <= PM101 + BM01 ;
      intpm110_10 <= PM101 + BM10 ;
      NEwpm010 <= intpm010_01;
      DB010 <='1';
      NEwpm110 <= intpm110_10;
      DB110 <='1';
    else
      OF010 <= '0';
      OF110 <= '0';
    end if;
  end if;
  BF3r <='1';
end process;
end Behavioral;
```

B.2.4 BF110_111.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.all;
entity BF110_111 is
    Port (
      BF1reset: in std_logic;--reset
      clk: in std_logic;--clock
      BF1en: in std_logic;--enable
      IF110: in  std_logic;--input flag
      IF111: in  std_logic;--input flag
      RD: in std_logic_vector(1 downto 0);--received data
      PMs: in std_logic_vector(2 downto 0);--path metric select
      PM110 : in  std_logic_vector(7 downto 0);--previous path metric
      PM111 : in  std_logic_vector(7 downto 0);--previous path metric
      OF011: out  std_logic;--output flag
      OF111: out  std_logic;--output flag
      BF4r: out std_logic;--BF4 ready
      DB011 : out  std_logic;--new decision bit
      DB111 : out  std_logic;--new decision bit
      Newpm011 : out  std_logic_vector(7 downto 0);--new path metric
      Newpm111 : out  std_logic_vector(7 downto 0) --new path metric
      );
end BF110_111;
architecture Behavioral of BF110_111 is
```

```
SIGNAL  BM00 :    std_logic_vector(1 downto 0);
SIGNAL  BM11 :    std_logic_vector(1 downto 0);
SIGNAL intpm011_00: std_logic_vector(7 downto 0);
SIGNAL intpm011_11: std_logic_vector(7 downto 0);
SIGNAL intpm111_11: std_logic_vector(7 downto 0);
SIGNAL intpm111_00: std_logic_vector(7 downto 0);
begin
-- Computing Branch metrics
process (RD)
  begin
    BF4R <='0';
    if (RD ="00") then
      BM00 <= "00";
      BM11 <= "10";
    else
      if( RD ="10" or RD ="01") then
        BM00 <= "01";
        BM11 <= "01";
      else
        BM00 <= "10";
        BM11 <= "00";
      end if;
    end if;
-- output flag handle
    if (IF110 = '1') then
      OF011 <= '1';
      OF111 <= '1';
      if (IF111 = '1') then
        intpm011_00 <= PM111 + BM00 ;
        intpm111_11 <= PM111 + BM11 ;
        intpm011_11 <= PM110 + BM11 ;
        intpm111_00 <= PM110 + BM00 ;
-- ACS
        if (intpm011_00 < intpm011_11) then
           NEwpm011 <= intpm011_00;
           DB011 <='1';
        else
           NEwpm011 <= intpm011_11;
           DB011 <='0';
        end if;
        if (intpm111_00 < intpm111_11) then
          NEwpm111 <= intpm111_00;
          DB111 <='0';
        else
          NEwpm111 <= intpm111_11;
          DB111 <='1';
        end if;
      else
        intpm011_11 <= PM110 + BM11 ;
        intpm111_00 <= PM110 + BM00 ;
```

```
        NEwpm111 <= intpm111_00;
        DB111 <='0';
        NEwpm011 <= intpm011_11;
        DB011 <='0';
      end if;
    else
      if (IF111 = '1') then
        OF011 <= '1';
        OF111 <= '1';
        intpm011_00 <= PM111 + BM00 ;
        intpm111_11 <= PM111 + BM11 ;
        NEwpm011 <= intpm011_00;
        DB011 <='1';
        NEwpm111 <= intpm111_11;
        DB111 <='1';
      else
        OF011 <= '0';
        OF111 <= '0';
      end if;
    end if;
    BF4R <='1';
end process;
end Behavioral;
```

B.3 RAMs

B.3.1 RAM des métriques

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity DBRAM is
generic( width: integer:=1;
    depth: integer:=256;
    addr: integer:=8);
port( Clock:    in std_logic;
  DB1en:    in std_logic;
  Read:   in std_logic;
  Write:    in std_logic;
  Read_Addr:  in std_logic_vector(addr-1 downto 0);
  Write_Addr:   in std_logic_vector(addr-1 downto 0);
  Data_in:  in std_logic_vector(width-1 downto 0);
  Data_out:   out std_logic_vector(width-1 downto 0)
);
end DBRAM ;
architecture behav of DBRAM is
```

```
-- use array to define the bunch of internal temparary signals
type ram_type is array (0 to depth-1) of
  std_logic_vector(width-1 downto 0);
signal tmp_ram: ram_type;
begin
    -- Read Functional Section
    process(Clock, Read)
    begin
  if (Clock'event and Clock='1') then
      if DB1en='1' then
    if Read='1' then
        -- buildin function conv_integer change the type
        -- from std_logic_vector to integer
        Data_out <= tmp_ram(conv_integer(Read_Addr));
    else
        Data_out <= (Data_out'range => 'Z');
    end if;
      end if;
  end if;
    end process;
    -- Write Functional Section
    process(Clock, Write)
    begin
  if (Clock'event and Clock='1') then
      if DB1en='1' then
    if Write='1' then
        tmp_ram(conv_integer(Write_Addr)) <= Data_in;
    end if;
      end if;
  end if;
    end process;
end behav;
```

B.3.2 RAM des décisions

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity metricsRam is
generic(  width:  integer:=8;
    depth:  integer:=8;
    addr: integer:=3);
port( Clock:    in std_logic;
  MR1en:    in std_logic;
  Read:   in std_logic;
  Write:    in std_logic;
  Read_Addr:  in std_logic_vector(addr-1 downto 0);
```

```
  Write_Addr:    in std_logic_vector(addr-1 downto 0);
  Di:    in std_logic_vector(width-1 downto 0);
  Do:    out std_logic_vector(width-1 downto 0)
);
end metricsRam ;
architecture behav of metricsRam  is

type ram_type is array (0 to depth-1) of
  std_logic_vector(width-1 downto 0);
signal tmp_ram: ram_type;
begin
    -- Read Functional Section
    process(Clock, Read)
    begin
  if (Clock'event and Clock='1') then
      if MR1en='1' then
    if Read='1' then
        -- buildin function conv_integer change the type
        -- from std_logic_vector to integer
        Do <= tmp_ram(conv_integer(Read_Addr));
    else
        Do <= (Do'range => 'Z');
    end if;
      end if;
  end if;
    end process;
    -- Write Functional Section
    process(Clock, Write)
    begin
  if (Clock'event and Clock='1') then
      if MR1en='1' then
    if Write='1' then
        tmp_ram(conv_integer(Write_Addr)) <= Di;
    end if;
      end if;
  end if;
    end process;
end behav;
```

B.4 Recherche du minimum

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity metricsRam is
generic(   width:   integer:=8;
    depth:   integer:=8;
```

```
    addr: integer:=3);
port( Clock:    in std_logic;
  MR1en:    in std_logic;
  Read:   in std_logic;
  Write:    in std_logic;
  Read_Addr:  in std_logic_vector(addr-1 downto 0);
  Write_Addr:   in std_logic_vector(addr-1 downto 0);
  Di:   in std_logic_vector(width-1 downto 0);
  Do:   out std_logic_vector(width-1 downto 0)
);
end metricsRam ;
architecture behav of metricsRam  is

type ram_type is array (0 to depth-1) of
  std_logic_vector(width-1 downto 0);
signal tmp_ram: ram_type;
begin
    -- Read Functional Section
    process(Clock, Read)
    begin
  if (Clock'event and Clock='1') then
      if MR1en='1' then
    if Read='1' then
        -- buildin function conv_integer change the type
        -- from std_logic_vector to integer
        Do <= tmp_ram(conv_integer(Read_Addr));
    else
        Do <= (Do'range => 'Z');
    end if;
      end if;
  end if;
    end process;
    -- Write Functional Section
    process(Clock, Write)
    begin
  if (Clock'event and Clock='1') then
      if MR1en='1' then
    if Write='1' then
        tmp_ram(conv_integer(Write_Addr)) <= Di;
    end if;
      end if;
  end if;
    end process;
end behav;
```

B.5 Registre à décalage

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity shiftReg is
    port( charge_shift: in std_logic;
          chargedState: in std_logic_vector (2 downto 0);
          readAdd     : out std_logic_vector(7 downto 0);
          receive      : out std_logic;
          dataIN       : in std_logic;
          dataOUT    : out std_logic:='0';
          SRen       : in std_logic;
          srclk      : in std_logic
        );
end shiftReg;
architecture Behavioral of shiftReg is
signal countdown : std_logic_vector (4 downto 0);
signal intREGISTER: std_logic_vector (3 downto 0):="0000";
signal readAddint : std_logic_vector(7 downto 0);
begin
process (SRen , charge_shift)
  begin
       receive <='0';
       if (SRen = '1') then
         if (charge_shift='1') then
         intREGISTER(3 downto 1) <= chargedState (2 downto 0);
         countdown <= "01111";
         receive <= '1';
         else if(countdown /= "0" ) then
           if (srclk' event and srclk='1') then
           intREGISTER (0) <= dataIN;
           intREGISTER (3 downto 1) <= intREGISTER (2 downto 0);
           readaddint(7 downto 3) <= countdown (4 downto 0);
           readAddint (2 downto 0) <= intREGISTER (3 downto 1);
           countdown <= countdown - 1;
           end if;
         end if;
         end if;
       end if;
end process;
dataOUT<= intREGISTER (3);
end Behavioral;
```

B.6 Trace Back

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity TraceBack is
        port( clk: in std_logic;
              TBreset: in std_logic;
              TBen: in std_logic;
              Minaddress: in std_logic_vector(7 downto 0);
              Dbaddress: out std_logic_vector(7 downto 0);
              decodedSequence: out std_logic;
              ReadR: out std_logic;
              Din: in std_logic
             );
end TraceBack;
architecture Behavioral of TraceBack is
-------------------------------------------------
------------SHIFT REG COMPONENT------------------
-------------------------------------------------
component shiftReg is
    port( charge_shift: in std_logic;
          chargedState: in std_logic_vector (2 downto 0);
          readAdd     : out std_logic_vector (7 downto 0);
          receive      : out std_logic;
          dataIN       : in std_logic;
          dataOUT    : out std_logic;
          SRen         : in std_logic;
          srclk      : in std_logic
        );
end component;
-------------------------------------------------
------------LIFO COMPONENT-----------------------
-------------------------------------------------
component LIFOpile is
 GENERIC(n: NATURAL:= 14);          -- LIFO size
    PORT(liforeset : IN std_logic;
      X          : IN  std_logic; -- input bit-vector
          Z          : OUT std_logic; -- output bit-vector
          Rd,Wr      : IN  std_logic;          -- read,write control signals
          Empty,Full: OUT std_logic;          -- condition signals
          lifoClk        : IN  std_logic);
end component;
------------signaux internes---------------------
---------------SHift REG-------------------------
signal addout:  std_logic_vector(7 downto 0);
signal SRclk:  std_logic;
signal cCS :  std_logic;
signal CCState : std_logic_vector(2 downto 0);
signal tcDi :  std_logic;
signal tcDo :  std_logic;
signal tSRen :  std_logic;
-------------------------------------------------
signal tclk :  std_logic;
signal countdown: integer:=15;
```

```
signal startshift: std_logic;
signal decisionBits: std_logic;
----------------LIFO-------------------------
signal cliforeset :  std_logic;
signal cX :  std_logic;
signal cZ :  std_logic;
signal cRd :  std_logic;
signal cWr :  std_logic;
signal cEmpty :  std_logic;
signal cFull :  std_logic;
signal lifoclk :  std_logic;
----------Trace Back Process----------------
begin
  process
    begin
    if (TBreset ='1') then
    cliforeset <='1';
    tSRen <= '0';
    else if (TBen = '1') then
    tSRen <= '1';
    cCS <= '1';
    end if;
    end if;
      if (startshift ='1') then
      decisionBits <= Din;
      cliforeset <='1';
      cWr <='1';
      cCS <='0';
        if clk' event and clk ='0' then
        cX <= decisionBits;
        end if;
      end if;
      if (cfull = '1') then
      if (clk ='1') then
        if (cempty ='0') then
        cRd <= '1';
        decodedsequence <=cZ;
          else NULL;
        end if;
        else null;
        end if;
        else null;
      end if;
  end process;
----------------Map--------------------------
instSHIFT: shiftReg port map (charge_shift => cCS ,chargedState => CCState
    ,readAdd => addout,receive => startShift , dataIN => tcDi,dataOUT =>
    tcDo,SRen => tSRen,srclk => clk);
instLIFO : LIFOpile port map (liforeset => cliforeset,X => cX, Z => cZ, Rd
     => cRd, Wr => cWr,Empty => cEmpty,Full => cFull,lifoClk => clk);
```

```
end Behavioral;
```

B.7 Viterbi Decoder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity viterbiDecoder is
    Port ( BitStream : in  STD_LOGIC;
           clock: in std_logic;
           reset: in std_logic;
              DecodedBitStream : in  STD_LOGIC
           );
end viterbiDecoder;
architecture Behavioral of viterbiDecoder is
----------     butterfly 1    --------------------------------
component BF000_001 is
 Port (
    BF1reset: in std_logic;
    BF1en: in std_logic;
    IF000: in  std_logic;
    IF001: in  std_logic;
    RD :in std_logic_vector(1 downto 0);
    PMS :in std_logic_vector(2 downto 0);
    PM000 : in  std_logic_vector(7 downto 0);
    PM001 : in  std_logic_vector(7 downto 0);
    oF000: out  std_logic;
    oF100: out  std_logic;
    BF1r: out std_logic;
    DB000 : out  std_logic;
    DB100 : out  std_logic;
    Npm000 : out  std_logic_vector(7 downto 0);
    Npm100 : out  std_logic_vector(7 downto 0)
      );
end component;
----------     butterfly 2    --------------------------------
component BF010_011 is
 Port (
    BF2reset: in std_logic;
    BF2en: in std_logic;
    IF010: in  std_logic;
    IF011: in  std_logic;
    RD: in std_logic_vector(1 downto 0);
    PMs: in std_logic_vector(2 downto 0);
    PM010 : in  std_logic_vector(7 downto 0);
    PM011 : in  std_logic_vector(7 downto 0);
    oF001: out  std_logic;
```

```
    oF101: out  std_logic;
    BF2r: out std_logic;
    DB001 : out  std_logic;
    DB101 : out  std_logic;
    Npm001 : out  std_logic_vector(7 downto 0);
    Npm101 : out  std_logic_vector(7 downto 0)
      );
end component;
----------      butterfly 3     ---------------------------------
component BF100_101 is
Port (
    BF3reset: in std_logic;
    BF3en: in std_logic;
    IF100: in  std_logic;
    IF101: in  std_logic;
    RD: in std_logic_vector(1 downto 0);
    PMS: in std_logic_vector(2 downto 0);
    PM100 : in  std_logic_vector(7 downto 0);
    PM101 : in  std_logic_vector(7 downto 0);
    oF010: out  std_logic;
    oF110: out  std_logic;
    BF3r: out std_logic;
    DB010 : out  std_logic;
    DB110 : out  std_logic;
    Npm010 : out  std_logic_vector(7 downto 0);
    Npm110 : out  std_logic_vector(7 downto 0)
    );
end component;
----------      butterfly 4     ---------------------------------
component BF110_111 is
Port (
    BF4reset: in std_logic;
    BF4en: in std_logic;
    IF110: in  std_logic;
    IF111: in  std_logic;
    RD: in std_logic_vector(1 downto 0);
    PMs: in std_logic_vector(2 downto 0);
    PM110 : in  std_logic_vector(7 downto 0);
    PM111 : in  std_logic_vector(7 downto 0);
    OF011: out  std_logic;
    OF111: out  std_logic;
    BF4r: out std_logic;
    DB011 : out  std_logic;
    DB111 : out  std_logic;
    Npm011 : out  std_logic_vector(7 downto 0);
    Npm111 : out  std_logic_vector(7 downto 0)
      );
end component;
----------      Metrics RAM1    ---------------------------------
component metricsRam is
```

```
generic( width: integer:=8;
     depth: integer:=8;
     addr:  integer:=3);
port(
    Clock:    in std_logic;
    Enable:   in std_logic;
    Read:   in std_logic;
    Write:    in std_logic;
    Read_Addr:  in std_logic_vector(addr-1 downto 0);
    Write_Addr:   in std_logic_vector(addr-1 downto 0);
    Data_in:  in std_logic_vector(width-1 downto 0);
    Data_out:   out std_logic_vector(width-1 downto 0)
);
end component ;
----------    DB RAM        ------------------------------------
component DBRAM is
generic( width: integer:=1;
     depth: integer:=256;
     addr:  integer:=8);
port( Clock:    in std_logic;
    Enable:   in std_logic;
    Read:   in std_logic;
    Write:    in std_logic;
    Read_Addr:  in std_logic_vector(addr-1 downto 0);
    Write_Addr:   in std_logic_vector(addr-1 downto 0);
    Data_in:  in std_logic_vector(width-1 downto 0);
    Data_out:   out std_logic_vector(width-1 downto 0)
);
end component ;
----------    Metrics RAM1    ------------------------------------
component metricsRam2is
generic(  width:  integer:=8;
    depth:  integer:=8;
    addr: integer:=3);
port( Clock:    in std_logic;
  Enable:   in std_logic;
  Read:   in std_logic;
  Write:    in std_logic;
  Read_Addr:  in std_logic_vector(addr-1 downto 0);
  Write_Addr:   in std_logic_vector(addr-1 downto 0);
  Data_in:  in std_logic_vector(width-1 downto 0);
  Data_out:   out std_logic_vector(width-1 downto 0)
);
end component ;
----------    DB RAM        ------------------------------------
component DBRAM2 is
generic(  width:  integer:=1;
    depth:  integer:=256;
    addr: integer:=8);
port( Clock:    in std_logic;
```

```
  Enable:    in std_logic;
  Read:    in std_logic;
  Write:     in std_logic;
  Read_Addr:  in std_logic_vector(addr-1 downto 0);
  Write_Addr:    in std_logic_vector(addr-1 downto 0);
  Data_in:  in std_logic_vector(width-1 downto 0);
  Data_out:    out std_logic_vector(width-1 downto 0)
);
end component ;
----------      MIN FINDER          ------------------------------
component MinFinder is
  port(
       en: in std_logic;
       wordin: in std_logic_vector(7 downto 0);
       ReadRam: in std_logic;
       copied: out std_logic;
       done: out std_logic:='0';
       min : out std_logic_vector(7 downto 0):="11111111"

       );
end component;
----------      Trace back      ---------------------------------
component TraceBack is
  port( clk: in std_logic;
       TBreset: in std_logic;
       TBen: in std_logic;
       Minaddress: in std_logic_vector(7 downto 0);
       decodedSequence: out std_logic_vector(7 downto 0);
       ReadR: out std_logic;
       Din: in std_logic
      );
end component;
---------- flags managers ------------------------------------
signal cf000 : std_logic;
signal cf001 : std_logic;
signal cf010 : std_logic;
signal cf011 : std_logic;
signal cf100 : std_logic;
signal cf101 : std_logic;
signal cf110 : std_logic;
signal cf111 : std_logic;
signal cBF1R : std_logic;
signal cBF2R : std_logic;
signal cBF3R : std_logic;
signal cBF4R : std_logic;
-------------------------------------------------------------
signal cDBR1en : std_logic;
signal cDBR2en: std_logic;
signal cMR1en : std_logic;
signal cMR2en : std_logic;
```

```
-----------------------------------------------------------------
signal cMRW1     : std_logic;
signal cMRW2     : std_logic;
signal cMRR1     : std_logic;
signal cMRR2     : std_logic;
signal cMRWadd   : std_logic;
signal cMRRadd   : std_logic;
-----------------------------------------------------------------
signal compteur  : unsigned(4 downto 0);
begin
process--viterbi
 begin
  if (reset = '1') then
    process
    --initialisation--
    cBF1en <= '1';
    cBF2en <= '0';
    cBF3en <= '0';
    cBF4en <= '0';
    cDBR1en <= '1';
    cDBR2en <= '0';
    cMR1en <= '1';
    cMR2en <= '0';
    cf000 <= '1';
    cf001 <= '0';
    cf010 <= '0';
    cf011 <= '0';
    cf100 <= '0';
    cf101 <= '0';
    cf110 <= '0';
    cf111 <= '0';
    cMFen <='0';
    cTBen <='0';
    cliforeset <='1';
    cSRen <= '0';
    end process;
  else
    process --PMU
    if (cif010 ='1' or cif011 = '1') then
    cBF2en <= '1';
    end if;
    if (cif100 ='1' or cif101 = '1') then
    cBF3en <= '1';
    end if;
    if (cif110 ='1' or cif111 = '1') then
    cBF4en <= '1';
    end if;
-----------------------------------------------------------
    if (compteur > 1) then
      if (cMR1W = '1') then
```

```
        cMR1R <= '1';
        cMR1Radd<= '000';
        cPM000 <= cData_out;
        cMR1Radd<= '001';
        cPM000 <= cData_out;
        cMR1Radd<= '010';
        cPM000 <= cData_out;
        cMR1Radd<= '011';
        cPM000 <= cData_out;
        cMR1Radd<= '100';
        cPM000 <= cData_out;
        cMR1Radd<= '101';
        cPM000 <= cData_out;
        cMR1Radd<= '110';
        cPM000 <= cData_out;
        cMR1Radd<= '111';
        cPM000 <= cData_out;
        readingdone <='1';
      else
        cMR2R <= '1';
        cMR2Radd<= '000';
        cPM000 <= cData_out;
        cMR2Radd<= '001';
        cPM000 <= cData_out;
        cMR2Radd<= '010';
        cPM000 <= cData_out;
        cMR2Radd<= '011';
        cPM000 <= cData_out;
        cMR2Radd<= '100';
        cPM000 <= cData_out;
        cMR2Radd<= '101';
        cPM000 <= cData_out;
        cMR2Radd<= '110';
        cPM000 <= cData_out;
        cMR2Radd<= '111';
        cPM000 <= cData_out;
        readingdone <='1';
```

```
    if (cBF1r ='1') then
    PM0 <= PM000; PM4 <= PM100;
    DB0 <= DB000; DB4 <= DB100;
    if (cMRW1 = '1') then
    cMRWadd <='000';
    cMR1Di <= PM0;
    cDBR1W <= '1';
    cDBR1Wadd (7 downto 3) <= conv_std_logic_vector(compteur,5);
    cDBR1Wadd (2 downto 0) <= '000';
    cDBR1Di <= DB0;
    cMRWadd <='100';
    cMR1Di <= PM4;
```

```
    cDBR1W <= '1';
    cDBR1Wadd (7 downto 3) <= conv_std_logic_vector(compteur,5);
    cDBR1Wadd (2 downto 0) <= '100';
    cDBR1Di <= DB4;
    else
    cMRWadd <= '000';
    cMR2Di <= PM0;
    cDBR2W <= '1';
    cDBR2Wadd (7 downto 3) <= conv_std_logic_vector(compteur,5);
    cDBR2Wadd (2 downto 0) <= '000';
    cDBR2Di <= DB0;
    cMRWadd <= '100';
    cMR2Di <= PM4;
    cDBR2W <= '1';
    cDBR2Wadd (7 downto 3) <= conv_std_logic_vector(compteur,5);
    cDBR2Wadd (2 downto 0) <= '100';
    cDBR2Di <= DB4;
    END IF;
-------------------------------------------------------------------
    if (cBF2r ='1') then
    PM1 <= PM001; PM5 <= PM101;
    DB1 <= DB001; DB5 <= DB101;
    if (cMRW1 = '1') then
    cMRWadd <='001';
    cMR1Di <= PM1;
    cDBR1W <= '1';
    cDBR1Wadd (7 downto 3) <= conv_std_logic_vector(compteur,5);
    cDBR1Wadd (2 downto 0) <= '001';
    cDBR1Di <= DB1;
    cMRWadd <='101';
    cMR1Di <= PM5;
    cDBR1W <= '1';
    cDBR1Wadd (7 downto 3) <= conv_std_logic_vector(compteur,5);
    cDBR1Wadd (2 downto 0) <= '101';
    cDBR1Di <= DB5;
    else
    cMRWadd <= '001';
    cMR2Di <= PM1;
    cDBR2W <= '1';
    cDBR2Wadd (7 downto 3) <= conv_std_logic_vector(compteur,5);
    cDBR2Wadd (2 downto 0) <= '001';
    cDBR2Di <= DB1;
    cMRWadd <= '101';
    cMR2Di <= PM5;
    cDBR2W <= '1';
    cDBR2Wadd (7 downto 3) <= conv_std_logic_vector(compteur,5);
    cDBR2Wadd (2 downto 0) <= '101';
    cDBR2Di <= DB5;
    END IF;
-------------------------------------------------------------------
```

```
if (cBF3r ='1') then
    PM2 <= PM010; PM6 <= PM110;
    DB2 <= DB010; DB6 <= DB110;
    if (cMRW1 = '1') then
    cMRWadd <='010';
    cMR1Di <= PM2;
    cDBR1W <= '1';
    cDBR1Wadd (7 downto 3) <= conv_std_logic_vector(compteur,5);
    cDBR1Wadd (2 downto 0) <= '010';
    cDBR1Di <= DB2;
    cMRWadd <='110';
    cMR1Di <= PM6;
    cDBR1W <= '1';
    cDBR1Wadd (7 downto 3) <= conv_std_logic_vector(compteur,5);
    cDBR1Wadd (2 downto 0) <= '110';
    cDBR1Di <= DB6;
    else
    cMRWadd <= '010';
    cMR2Di <= PM2;
    cDBR2W <= '1';
    cDBR2Wadd (7 downto 3) <= conv_std_logic_vector(compteur,5);
    cDBR2Wadd (2 downto 0) <= '010';
    cDBR2Di <= DB2;
    cMRWadd <= '110';
    cMR2Di <= PM6;
    cDBR2W <= '1';
    cDBR2Wadd (7 downto 3) <= conv_std_logic_vector(compteur,5);
    cDBR2Wadd (2 downto 0) <= '110';
    cDBR2Di <= DB6;
    END IF;
```

```
if (cBF4r ='1') then
    PM3 <= PM011; PM7 <= PM111;
    DB3 <= DB011; DB7 <= DB111;
    if (cMRW1 = '1') then
    cMRWadd <='011';
    cMR1Di <= PM3;
    cDBR1W <= '1';
    cDBR1Wadd (7 downto 3) <= conv_std_logic_vector(compteur,5);
    cDBR1Wadd (2 downto 0) <= '011';
    cDBR1Di <= DB3;
    cMRWadd <='111';
    cMR1Di <= PM7;
    cDBR1W <= '1';
    cDBR1Wadd (7 downto 3) <= conv_std_logic_vector(compteur,5);
    cDBR1Wadd (2 downto 0) <= '111';
    cDBR1Di <= DB7;
    else
    cMRWadd <= '011';
    cMR2Di <= PM3;
```

```
    cDBR2W <= '1';
    cDBR2Wadd (7 downto 3) <= conv_std_logic_vector(compteur,5);
    cDBR2Wadd (2 downto 0) <= '011';
    cDBR2Di <= DB3;
    cMRWadd <= '111';
    cMR2Di <= PM7;
    cDBR2W <= '1';
    cDBR2Wadd (7 downto 3) <= conv_std_logic_vector(compteur,5);
    cDBR2Wadd (2 downto 0) <= '111';
    cDBR2Di <= DB7;
    END IF;
-------------------------------------------------------------------------
    compteur <= compteur + 1;
    end process;
    process --minFinder
      if (cMR1en = '1' and cMR2en = '1') then
        if cMR1W' event and cMR1W = '0') then
          cMFen <= '1';
          cMR1Radd <='000';
          cwordIn <= cdata_out1;
          cMR1Radd <='001';
          cwordIn <= cdata_out1;
          cMR1Radd <='010';
          cwordIn <= cdata_out1;
          cMR1Radd <='010';
          cwordIn <= cdata_out1;
          cMR1Radd <='011';
          cwordIn <= cdata_out1;
          cMR1Radd <='100';
          cwordIn <= cdata_out1;
          cMR1Radd <='101';
          cwordIn <= cdata_out1;
          cMR1Radd <='110';
          cwordIn <= cdata_out1;
          cMR1Radd <='111';
          cwordIn <= cdata_out1;
        else if (cMR2W' event and cMR2w ='0') then
          cMFen <= '1';
          cMR2Radd <='000';
          cwordIn <= cdata_out2;
          cMR2Radd <='001';
          cwordIn <= cdata_out2;
          cMR2Radd <='010';
          cwordIn <= cdata_out2;
          cMR2Radd <='010';
          cwordIn <= cdata_out2;
          cMR2Radd <='011';
          cwordIn <= cdata_out2;
          cMR2Radd <='100';
          cwordIn <= cdata_out2;
```

```
            cMR2Radd <='101';
            cwordIn <= cdata_out2;
            cMR2Radd <='110';
            cwordIn <= cdata_out2;
            cMR2Radd <='111';
            cwordIn <= cdata_out2;
          else NULL;
          end if;
        end if;
        if (cdone' event and cdone ='1' ) then
        cminadd <= cmin;
        end if;
      end process;
      process -- traceback
      end process;
end process;
instBF1 : BF000_001    port map (IF000 => cIF000,IF001 => cIF001,RD => cRD,
    PMS => cPMS1,OF000 => cOF000,OF100 => cOF100,BF1R => cBF1R,DB000 =>
    cDB000,DB100 => cDB100,NPM000 => cNPM000,NPM100 => cNPM100);
instBF2 : BF010_011    port map (IF010 => cIF010,IF011 => cIF011,RD => cRD,
    PMS => cPMS2,OF001 => cOF001,OF101 => cOF101,BF2R => cBF2R,DB001 =>
    cDB001,DB101 => cDB101,NPM001 => cNPM001,NPM101 => cNPM101);
instBF3 : BF100_101    port map (IF100 => cIF100,IF101 => cIF101,RD => cRD,
    PMS => cPMS3,OF010 => cOF010,OF110 => cOF110,BF3R => cBF3R,DB010 =>
    cDB010,DB110 => cDB110,NPM010 => cNPM010,NPM110 => cNPM110);
instBF4 : BF110_111    port map (IF110 => cIF110,IF111 => cIF111,RD => cRD,
    PMS => cPMS4,OF011 => cOF011,OF111 => cOF111,BF4R => cBF4R,DB011 =>
    cDB011,DB111 => cDB111,NPM011 => cNPM011,NPM111 => cNPM111);
instMR1 : metricsRam   port map (CLOCK,enable =>cMR1en,read => cMR1R,write
    =>cMR1W,read_addr => cMR1Radd,write_Addr => cMR1Wadd,dI => cDI,dO =>cDO
    );
instMR2 : metricsRam2  port map (CLOCK,enable =>cMR2en,read => cMR2R,write
    =>cMR2W,read_addr => cMR2Radd,write_Addr => cMR2Wadd,dI => cDI,dO =>cDO
    );
instDBR1: DBRAM        port map (CLOCK,enable =>cDBR1en,read => cDBR1R,write
     =>cDBR1W,read_addr => cDBR1Radd,write_Addr => cDBR1Wadd,dI => cDI,dO
    =>cDO);
instDBR2:   DBRAM2      port map (CLOCK,enable =>cDBR2en,read => cDBR2R,
    write =>cDBR2W,read_addr => cDBR2Radd,write_Addr => cDBR2Wadd,dI => cDI
    ,dO =>cDO);
instMF  : MinFinder    port map (MFen => cMFen , wordin => cwordin, ReadRam
     => cwordin, copied => ccopied, done => cdone, min => cmin);
instTB  : TraceBack    port map (TBen => cTBen ,Minaddress => cMinaddress,
    decodedSequence => coutput);
end Behavioral;
```