PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Décodage de codes polaires sur des architectures programmables
Auteur: Author:	Mathieu Léonardon
Date:	2018
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Léonardon, M. (2018). Décodage de codes polaires sur des architectures programmables [Ph.D. thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/3788/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/3788/
Directeurs de recherche: Advisors:	Yvon Savaria, & Christophe Jégo
Programme: Program:	génie électrique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

UNIVERSITÉ DE MONTRÉAL

DÉCODAGE DE CODES POLAIRES SUR DES ARCHITECTURES PROGRAMMABLES

MATHIEU LÉONARDON
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE ÉLECTRIQUE)
NOVEMBRE 2018

© Mathieu Léonardon, 2018.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

DÉCODAGE DE CODES POLAIRES SUR DES ARCHITECTURES PROGRAMMABLES

présentée par : LÉONARDON Mathieu
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. SAWAN Mohamad, Ph.D., président
M. SAVARIA Yvon, Ph.D., membre et directeur de recherche
M. JÉGO Christophe, Ph.D., membre et directeur de recherche
M. BAGHDADI Amer, Ph.D., membre
M. MULLER Olivier, Ph.D., membre
M. POULLIAT Charly, Ph.D., membre
M. LEROUX Camille, Ph.D., membre
M. CASSEAU Emmanuel, Ph.D., membre externe

DÉDICACE

À mes grands parents.

REMERCIEMENTS

Je remercie vivement les membres de mon jury de soutenance, le professeur Amer Baghdadi, de l'Institut Mines-Télécom Atlantique, le professeur Emmanuel Casseau, de l'Université Rennes 1, le docteur Olivier Muller, de L'Institut National Polytechnique de Grenoble et Charly Poulliat, de l'Institut National Polytechnique de Toulouse, pour leurs lectures attentive et leurs retours bienveillants sur mes travaux. Merci à Mohamad Sawan, de Polytechnique Montréal, d'avoir présidé ce jury.

Merci à Yvon pour m'avoir donné la possibilité de réaliser cette thèse en cotutelle entre la France et le Canada. Ce fut une expérience humaine formidable. Merci pour nos nombreuses discussions, pour ton suivi assidu de mes travaux, pour ton soutien humain. Merci à Christophe d'avoir cru en moi. Merci de ton soutien sans faille à tous les points de vue. Tu sais créer des relations fortes, basées sur la confiance, cela est très précieux. Merci à Camille d'avoir partagé autant et de t'être investi à ce point. Merci de ton exigence permanente et de m'avoir tiré vers le haut. Merci enfin de ta sympathie, de ta bonne humeur et pour tous les moments que nous avons passés ensemble. Merci Adrien pour ton travail exceptionnel. Merci d'être à l'origine de ce fantastique projet qu'est AFF3CT. J'ai énormément appris à ton contact et je suis très fier de ce que nous avons réalisé ensemble. J'espère que nous continuerons encore longtemps sur ce chemin.

Merci à Ali d'avoir été si enthousiaste pour travailler avec moi, pour nos nombreuses discussions. Je te souhaite beaucoup de réussite dans tes projets futurs. Merci à Mickaël pour nos franches rigolades et pour nos débats animés. Merci à Érika de les avoir supportés. Merci à David avec qui je n'ai que d'excellents souvenirs. Merci à Anders, Céline, Fanny, Guillaume, Juliette, Laurent, Luce, Rodrigo et à toutes les personnes qui ont rendu notre séjour à Montréal si agréable. Merci à Thibaud de m'avoir mis le pied à l'étrier, merci à Fanny et toi pour votre amitié. Merci aux collègues actuels ou passés de l'équipe CSN au sein de laquelle on se sent bien, Antoine, Ali, Baptiste, Guillaume B., Guillaume D., Imen, Jean-Baptiste, Jonathan, Mariam, Nicola, Nicolas, Olivier, Vincent, Yann, Yassine. Merci à ceux qui se reconnaîtront pour les parties endiablées de Teeworlds.

Merci à mes parents pour votre soutien durant toutes ces années. Je suis fier d'être votre fils.

Merci à ma famille et à mes amis proches, en particulier à Jasmine pour son désintéressement.

Enfin, et surtout, merci Aline sans qui jamais je n'aurais pu réaliser tout ceci.
Merci de partager ma vie et de la rendre meilleure.

RÉSUMÉ

Les codes polaires constituent une classe de codes correcteurs d'erreurs inventés récemment qui suscite l'intérêt des chercheurs et des industriels, comme en atteste leur sélection pour le codage des canaux de contrôle dans la prochaine génération de téléphonie mobile (5G). Un des enjeux des futurs réseaux mobiles est la virtualisation des traitements numériques du signal, et en particulier les algorithmes de codage et de décodage. Afin d'améliorer la flexibilité du réseau, ces algorithmes doivent être décrits de manière logicielle et être déployés sur des architectures programmables. Une telle infrastructure de réseau permet de mieux répartir l'effort de calcul sur l'ensemble des nœuds et d'améliorer la coopération entre cellules. Ces techniques ont pour but de réduire la consommation d'énergie, d'augmenter le débit et de diminuer la latence des communications. Les travaux présentés dans ce manuscrit portent sur l'implémentation logicielle des algorithmes de décodage de codes polaires et la conception d'architectures programmables spécialisées pour leur exécution.

Une des caractéristiques principales d'une chaîne de communication mobile est l'instabilité du canal de communication. Afin de remédier à cette instabilité, des techniques de modulations et de codages adaptatifs sont utilisées dans les normes de communication. Ces techniques impliquent que les décodeurs supportent une vaste gamme de codes : ils doivent être génériques. La première contribution de ces travaux est l'implémentation logicielle de décodeurs génériques des algorithmes de décodage "à Liste" sur des processeurs à usage général. En plus d'être génériques, les décodeurs proposés sont également flexibles. Ils permettent en effet des compromis entre pouvoir de correction, débit et latence de décodage par la paramétrisation fine des algorithmes. En outre, les débits des décodeurs proposés atteignent les performances de l'état de l'art et, dans certains cas, les dépassent.

La deuxième contribution de ces travaux est la proposition d'une nouvelle architecture programmable performante spécialisée dans le décodage de codes polaires. Elle fait partie de la famille des processeurs à jeu d'instructions dédiés à l'application. Un processeur de type RISC à faible consommation en constitue la base. Cette base est ensuite configurée, son jeu d'instructions est étendu et des unités matérielles dédiées lui sont ajoutées. Les simulations montrent que cette architecture atteint des débits et des latences proches des implémentations logicielles de l'état de l'art sur des processeurs à usage général. La consommation énergétique est réduite d'un ordre de grandeur. En effet, lorsque l'on considère le décodage par annulation successive d'un code polaire (1024,512), l'énergie nécessaire par bit décodé est de l'ordre de 10 nJ sur des processeurs à usage général contre 1 nJ sur les processeurs

proposés.

La troisième contribution de ces travaux est également une architecture de processeur à jeu d'instructions dédié à l'application. Elle se différencie de la précédente par l'utilisation d'une méthodologie de conception alternative. Au lieu d'être basée sur une architecture de type RISC, l'architecture du processeur proposé fait partie de la classe des architectures déclenchées par le transport. Elle est caractérisée par une plus grande modularité qui permet d'améliorer très significativement l'efficacité du processeur. Les débits mesurés sont alors supérieurs à ceux obtenus sur les processeurs à usage général. La consommation énergétique est réduite à environ 0.1 nJ par bit décodé pour un code polaire (1024,512) avec l'algorithme de décodage par annulation successive. Cela correspond à une réduction de deux ordres de grandeur en comparaison de la consommation mesurée sur des processeurs à usage général.

Mots clefs : Codes polaires, Décodeur Logiciel, ASIP, Annulation Successive, Annulation Successive à Liste

ABSTRACT

Polar codes are a recently invented class of error-correcting codes that are of interest to both researchers and industry, as evidenced by their selection for the coding of control channels in the next generation of cellular mobile communications (5G). One of the challenges of future mobile networks is the virtualization of digital signal processing, including channel encoding and decoding algorithms. In order to improve network flexibility, these algorithms must be written in software and deployed on programmable architectures. Such a network infrastructure allow dynamic balancing of the computational effort across the network, as well as inter-cell cooperation. These techniques are designed to reduce energy consumption, increase throughput and reduce communication latency. The work presented in this manuscript focuses on the software implementation of polar codes decoding algorithms and the design of programmable architectures specialized in their execution.

One of the main characteristics of a mobile communication chain is that the state of communication channel changes over time. In order to address issue, adaptive modulation and coding techniques are used in communication standards. These techniques require the decoders to support a wide range of codes : they must be generic. The first contribution of this work is the software implementation of generic decoders for "List" polar decoding algorithms on general purpose processors. In addition to their genericity, the proposed decoders are also flexible. Trade-offs between correction power, throughput and decoding latency are enabled by fine-tuning the algorithms. In addition, the throughputs of the proposed decoders achieve state-of-the-art performance and, in some cases, exceed it.

The second contribution of this work is the proposal of a new high-performance programmable architecture specialized in polar code decoding. It is part of the family of Application Specific Instruction-set Processors (ASIP). The base architecture is a RISC processor. This base architecture is then configured, its instruction set is extended and dedicated hardware units are added. Simulations show that this architecture achieves throughputs and latencies close to state-of-the-art software implementations on general purpose processors. Energy consumption is reduced by an order of magnitude. The energy required per decoded bit is about 10 nJ on general purpose processors compared to 1 nJ on proposed processors when considering the Successive Cancellation (SC) decoding algorithm of a polar code (1024,512).

The third contribution of this work is also the design of an ASIP architecture. It differs from the previous one by the use of an alternative design methodology. Instead of being based on

a RISC architecture, the proposed processor architecture is part of the class of Transport Triggered Architectures (TTA). It is characterized by a greater modularity that allows to significantly improve the efficiency of the processor. The measured flow rates are then higher than those obtained on general purpose processors. The energy consumption is reduced to about 0.1 nJ per decoded bit for a polar code (1024,512) with the SC decoding algorithm. This corresponds to a reduction of two orders of magnitude compared to the consumption measured on general purpose processors.

Key words : Polar codes, Software decoder, ASIP, Successive Cancellation, Sucessive Cancellation List

TABLE DES MATIÈRES

DÉDICACE iii
REMERCIEMENTS iv
RÉSUMÉ vi
ABSTRACT viii
TABLE DES MATIÈRES x
LISTE DES TABLEAUX xiii
LISTE DES FIGURES xiv
LISTE DES SIGLES ET ABRÉVIATIONS xvii
LISTE DES ANNEXES xx
CHAPITRE 1 INTRODUCTION 1
CHAPITRE 2 LES CODES POLAIRES 6
2.1 Le principe et la construction des codes polaires 6
2.1.1 La chaîne de communications numériques 6
2.1.2 Le canal composite 7
2.1.3 Les codes polaires 8
2.1.4 Le phénomène de polarisation 9
2.2 Les algorithmes de décodage de codes polaires 10
2.2.1 L'algorithme de décodage par Annulation Successive (SC) 10
2.2.2 L'algorithme de décodage SC Liste 14
2.2.3 L'algorithme de décodage SC *Flip* 17
2.2.4 L'algorithme de décodage SC *Stack* 18
2.2.5 Les algorithmes itératifs à sortie souple 18
2.2.6 Les performances de décodage des différents algorithmes 20
2.3 L'élagage de l'arbre de décodage 28
2.3.1 L'élagage de l'algorithme de décodage SC 28
2.3.2 L'élagage de l'algorithme de décodage SCL 29
2.3.3 L'élagage de l'algorithme de décodage SCAN 30
2.4 Synthèse des différents algorithmes de décodage 31
CHAPITRE 3 DÉCODEUR LOGICIEL DE CODES POLAIRES À LISTE 32

3.1 Décodeurs logiciels pour les réseaux cellulaires 32
3.2 État de l'art sur les implémentations logicielles de l'algorithme SCL 34
3.2.1 Vectorisation 34
3.2.2 Déroulage du code source 36
3.3 Généricité et flexibilité d'un décodeur de codes polaires 37
3.3.1 Définitions 37
3.3.2 Généricité 39
3.3.3 Flexibilité 39
3.4 Optimisations de l'implémentation logicielle des décodeurs à liste 48
3.4.1 Algorithme de tri 48
3.4.2 Accélération du contrôle de redondance cyclique 50
3.4.3 Gestion des sommes partielles 51
3.5 Expérimentations : mesures et comparaisons 52
3.5.1 Algorithme complètement adaptatif 53
3.5.2 Comparaison avec l'état de l'art 53
3.6 Résumé des contributions 56
CHAPITRE 4 CONCEPTION D'UN PROCESSEUR PAR LA SPECIALISATION DE SON ARCHITECTURE 58
4.1 Les processeurs à jeu d'instructions spécifique à l'application 58
4.1.1 Principes de base 58
4.1.2 Les processeurs RISC 59
4.1.3 Les processeurs à jeu d'instructions spécifique à l'application 61
4.2 Un ASIP dédié au décodage de codes polaires 64
4.2.1 Paramétrage du processeur de base 64
4.2.2 Ressources calculatoires utilisées dans les implémentations matérielles de l'état de l'art 67
4.2.3 Instructions spécialisées *multi-registres* 69
4.2.4 Instructions spécialisées *simple-registre* 71
4.2.5 Description logicielle de l'algorithme de décodage 72
4.3 Expérimentations et mesures 73
4.3.1 Mesure du gain lié à la spécialisation du processeur 73
4.3.2 Comparaison avec un processeur ARM 75
4.3.3 Comparaison avec un processeur d'architecture x86 76
4.4 Synthèse et limites du modèle architectural 77
CHAPITRE 5 CONCEPTION D'UN PROCESSEUR PAR LA DESCRIPTION DE

SON ARCHITECTURE 79
5.1 Transport Triggered Architectures 79
5.1.1 Principes et caractéristiques des processeurs TTA 79
5.1.2 L'environnement TCE 83
5.2 Transport Triggered Polar Decoders 87
5.2.1 Architecture du décodeur `TT-SC`. 87
5.2.2 Unités fonctionnelles du décodeur `TT-SC`. 89
5.2.3 Description logicielle ciblant l'architecture `TT-SC`. 92
5.2.4 Implémentation de l'algorithme SCAN 94
5.2.5 Généricité et Flexibilité. 97
5.3 Expérimentations et mesures. 97
5.3.1 Architecture `TT-SC`. 98
5.3.2 Architecture `TT-SCAN`. 101
5.4 Synthèse 102
CHAPITRE 6 CONCLUSION 104
RÉFÉRENCES 108
PUBLICATIONS 113
ANNEXES 114

LISTE DES TABLEAUX

Tableau 2.1 Tendances des différents algorithmes concernant leurs pouvoirs de correction, débits et latences. 31

Tableau 3.1 Comparaison de débits et latences des algorithmes SCL adaptatifs pour des représentations à virgule flottante (32 bits) et virgule fixe (16 et 8 bits). Code polaire (2048,1723), L = 32, CRC $\mathsf{c} = 32$ (GZip). 45

Tableau 3.2 Comparaison des débits et des latences des décodeurs proposés avec ceux de l'état de l'art. Représentation des LLR et sommes partielles sur 32 bits. Code polaire (2048,1723), L = 32, CRC GZip $\mathsf{c} = 32$. . . 54

Tableau 3.3 Comparaison des débits, du nombre de cycles d'horloge nécessaires pour décoder une trame, et de l'énergie dépensée par bit décodé, pour des décodeurs proposés sur différentes cibles matérielles. Représentation en virgule fixe sur 8 bits , $\mathsf{E_b/N_0} = 4.5\mathsf{dB}$, CRC GZip $\mathsf{c} = 32$. 55

Tableau 4.1 Impact de chaque amélioration de l'ASIP sur le nombre de cycles d'horloges nécessaires pour décoder une trame, le débit et la surface occupée. La taille des mémoires n'est pas prise en compte pour la surface occupée. Fréquence considérée : 835 MHz. 75

Tableau 4.2 Comparaison de la latence, du débit et de la consommation énergétique de décodeurs SC pour différents processeurs. 77

Tableau 5.1 Comparaison de l'architecture `TT-SC` avec des processeurs à usage général et l'ASIP XTensa proposé dans le chapitre 4. 99

Tableau 5.2 Implémentations sur cible FPGA de décodeurs SC pour un code polaire (1024,512). 100

Tableau 5.3 Implémentations sur cible ASIC de décodeurs SC pour un code polaire (1024,512). 101

Tableau 5.4 Implémentations sur cible FPGA de décodeurs SCAN pour un code polaire (1024,512). 103

Tableau 6.1 Existence, disponibilité d'un système d'exploitation, débits et consommations énergétiques des processeurs pour les différentes architectures considérées. Les intervalles de débits et de consommations énergétiques concernent le décodage de mots de codes polaires dont les tailles varient de $\mathsf{N} = 128$ à $\mathsf{N} = 1024$ et dont le rendement est $\mathsf{R} = 1/2$ 106

LISTE DES FIGURES

Figure 2.1 Modèle simplifié d'une chaîne de communications. 7
Figure 2.2 Représentation en graphe de factorisation de l'encodage de codes polaires. 9
Figure 2.3 Encodage systématique. 9
Figure 2.4 Fonctions élémentaires et séquencement du décodage SC d'un noyau $\mathsf{N} = 2$. 11
Figure 2.5 Arbre de décodage SC. 13
Figure 2.6 Ordonnancement du décodage SC d'un code polaire de taille $\mathsf{N} = 4$. . 13
Figure 2.7 Duplication de l'arbre de décodage dans l'algorithme SCL. 15
Figure 2.8 Concaténation avec un CRC. 17
Figure 2.9 Fonctions élémentaires et séquencement du décodage SCAN du noyau $\mathsf{N} = 2$. 19
Figure 2.10 Performances de décodage de l'algorithme SC pour différentes valeurs de N ($\mathsf{R} = 1/2$). 21
Figure 2.11 Performances de décodage de l'algorithme SC pour différentes valeurs de R ($\mathsf{N} = 2048$). 22
Figure 2.12 Performances de décodage des algorithmes SC, SCL et CASCL pour un code polaire (2048,1723). Un CRC de taille $\mathsf{c} = 16$ est utilisé pour l'algorithme CASCL. 23
Figure 2.13 Impact de la taille du CRC sur les performances de l'algorithme SCL pour un code (2048,1723). 26
Figure 2.14 Performances de l'algorithme SCF en comparaison avec les algorithmes SC et SCL pour un code polaire (1024,512), $\mathsf{c} = 16$. 27
Figure 2.15 Performances des algorithmes itératifs à sortie souple en comparaison avec les algorithmes SC et SCL pour un code polaire (1024,512). . . . 27
Figure 2.16 Élagage de l'arbre de décodage SC. 29
Figure 3.1 Évolution de l'architecture des stations de base. 33
Figure 3.2 Implémentation logicielle des fonctions f et g utilisant la bibliothèque MIPP. 35
Figure 3.3 Déroulage du code source décrivant l'algorithme de décodage SC non élagué d'un code polaire (4,2) systématique. 38
Figure 3.4 Performances de décodage des algorithmes SC et CASCL pour des très grandes tailles de code, $\mathsf{R} = 1/2$, CRC $\mathsf{c} = 32$ (GZip). 40

Figure 3.5 Performances de décodage et débits de l'algorithme CASCL pour différentes valeurs de L d'un code polaire (2048,1024) concaténé à un CRC $\mathsf{c} = 32$ (GZip). 41
Figure 3.6 Impact de l'activation des nœuds d'élagage de l'algorithme CASCL, $\mathsf{N} = 2048$, $\mathrm{L} = 32$, $\mathsf{c} = 32$. 42
Figure 3.7 Effets de l'utilisation des nœuds `SPC4+` dans l'algorithme CASCL pour un FER de 10^{-5}. 43
Figure 3.8 Impact de l'utilisation des nœuds de répétition sur des implémentations quantifiées. 46
Figure 3.9 Performances de décodage et débits des algorithmes de décodage PASCL et FASCL pour un code polaire (2048,1723), $\mathrm{L} = 32$, et CRC $\mathsf{c} = 32$ GZip. 47
Figure 3.10 Méthode de Schreier. 49
Figure 3.11 Extraction des bits d'information avant vérification du CRC. 51
Figure 3.12 Comparaison des débits associés aux méthodes $\mathsf{SCL}_{\mathtt{cpy}}$ et $\mathsf{SCL}_{\mathtt{ptr}}$ pour différentes valeurs de N. 53
Figure 4.1 Étages d'un processeur RISC. 60
Figure 4.2 Structure et fonctionnement du pipeline RISC classique à 5 étages. . 61
Figure 4.3 Méthodologies de conception des ASIP. 63
Figure 4.4 Architecture associée à la fonctionnalité FLIX. 65
Figure 4.5 Nombre d'échecs d'accès à la mémoire cache en fonction de la taille de la mémoire pour le décodage SC d'un code polaire de taille $\mathsf{N} = 1024$. 66
Figure 4.6 Unité matérielle élémentaire (PE : Processing Element). 68
Figure 4.7 Architecture de l'ASIP proposé. 69
Figure 4.8 Implémentations matérielles des instructions spécialisées. 70
Figure 4.9 Instructions *simple-registre*. 72
Figure 4.10 Description logicielle de l'exécution d'une instruction spécialisée. . . 73
Figure 4.11 Description logicielle du pseudo-déroulage. 74
Figure 4.12 Comparaison du nombre de cycles de l'horloge nécessaires au décodage SC entre le Cortex ARM A57 et l'ASIP proposé. 76
Figure 5.1 Schéma illustrant les architectures de processeur TTA. 81
Figure 5.2 Organisation du flot de conception TCE. 85
Figure 5.3 Architecture `TT-SC`. 88
Figure 5.4 Illustration des chargements et sauvegardes non-alignés. 90
Figure 5.5 Unité matérielle de décodage d'un sous-arbre avec un traitement multi-cycles. 91

Figure 5.6 Exemple d'un code source et du code assembleur résultant. 93
Figure 5.7 Illustration du parallélisme d'instructions sur le processeur TT-SC. . . 95
Figure 5.8 Architecture TT-SCAN. 96
Figure 5.9 Organisation de l'unité « LLR PU » réalisant les fonctions élémentaires SC et SCAN. 96
Figure A.1 Représentations des noeuds de décodage 115
Figure A.2 Fonctions F et G de l'algorithme SC 116

LISTE DES SIGLES ET ABRÉVIATIONS

5G	5th generation of cellular mobile communications
ADL	Architecture Description Language
ALU	Arithmetical and Logical Unit
AMC	Adaptive Modulation and Coding
ARM	Advanced Risc Machine
ARQ	Automatic Repeat reQuest
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction set Computer
AVX	Advanced Vector Extensions
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
BI-AWGNC	Binary Input - AWGN Channel
BLLR	Backward LLR
BP	Belief Propagation
BPSK	Binary Phase-Shift Keying
CASCL	CRC-Aided Successive Cancellation List
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DM	Data Memory
EX	Execution
FASCL	Fully Adaptive Successive Cancellation List
FER	Frame Error Rate
FF	Flip Flop
FLIX	Flexible Length Instruction Extensions
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FU	Functional Unit
GCU	Global Control Unit
GNU	GNU's Not UNIX
HARQ	Hybrid Automatic Repeat reQuest
ID	Instruction Decode
IF	Instruction Fetch
IM	Instruction Memory

KNL	Knights Landing
LDPC	Low Density Parity Check Codes
LLR	Log Likelihood Ratio
LLVM	Low Level Virtual Machine
LSU	Load and Store Unit
LTE	Long Term Evolution
LUT	LookUp Table
MAC	Multiple and ACcumulate
MIPP	My Intrinsics Plus Plus library
OS	Operating System
PASCL	Partially Adaptive Successive Cancellation List
PE	Processing Element
PS	Partial Sum
PSN	Partial Sorting Network
PU	Processing Unit
RAM	Random Access Memory
REP	REPetition
RF	Register File
RISC	Reduced Instruction Set Computer
RTL	Register-Transfer Level
SC	Successive Cancellation
SCAN	Soft CANcellation
SCF	Successive Cancellation Flip
SCL	Successive Cancellation List
SCS	Successive Cancellation Stack
SDR	Software Defined Radio
SIMD	Single Instruction Multiple Data
SMT	Simultaneous MultiThreading
SNR	Signal-to-Noise Ratio
SPC	Single Parity Check
SSE	Streaming SIMD Extensions
TCE	TTA-based Co-design Environment
TCEMC	TCE MultiCore
TIE	Tensilica Instruction Extension
TTA	Transport Triggered Architecture
TUT	Tampere University of Technology

VHDL	VHSIC Hardware Description Language
VLIW	Very Long Instruction Word
WB	Write Back

LISTE DES ANNEXES

Annexe A COMPLÉMENTS AU CHAPITRE 1 114

CHAPITRE 1 INTRODUCTION

Le décodage de codes polaires sur des architectures dites programmables constitue le sujet de cette thèse. Deux axes principaux sont développés. Le premier est l'implémentation d'algorithmes de décodage sur des architectures de processeurs généralistes. Le second est la conception d'architectures programmables spécialisées pour une classe d'applications, à savoir le décodage de codes polaires. Dans cette introduction, le contexte des codes correcteurs d'erreurs et de leurs architectures est établi puis le sujet de recherche est présenté et motivé. La structure du manuscrit est ensuite détaillée en récapitulant les différentes contributions contenues dans ces travaux de thèse.

Les codes correcteurs d'erreurs

Une chaîne de communications numériques élémentaire est constituée d'une source, d'un canal de communication et d'un destinataire. Un message est émis par la source à travers le canal jusqu'au destinataire. À moins d'un canal idéal, des perturbations détériorent le message et des erreurs apparaissent. Ces erreurs peuvent être corrigées à l'aide de codes correcteurs d'erreurs. Pour ce faire, de la redondance est ajoutée au message d'origine, en aval de la source. Cette redondance est utilisée, en amont du destinataire, afin de corriger les erreurs potentielles.

Claude Shannon propose en 1948 une formalisation mathématique de l'information [1]. Un des résultats majeurs de ses travaux est l'existence d'une limite théorique à la quantité d'information transmissible sur un canal de communication. Cette limite peut être atteinte à l'aide d'un code correcteur d'erreurs dont la construction reste à déterminer. Dès lors, la communauté scientifique s'empare de cette problématique. À la même époque, John Bardeen, Walter Brattain et William Shockley inventent le transistor. Les circuits électroniques constituent alors l'outil nécessaire à la mise en œuvre de codes correcteurs d'erreurs. Ainsi, de 1948 à nos jours, les chercheurs et ingénieurs mettent au point des algorithmes et des architectures de systèmes de correction d'erreurs de plus en plus complexes, permettant de s'approcher de la limite théorique établie par Claude Shannon.

Les codes polaires sont une des familles de codes correcteurs d'erreurs les plus récentes. Ils sont inventés en 2008 par Erdal Arıkan [2]. Leur sélection pour faire partie de la prochaine norme 5G atteste de leur efficacité et montre la nécessité de concevoir des systèmes performants pour en réaliser les deux étapes : le codage et le décodage. Jusqu'à présent, dans une grande

majorité des systèmes de communications mobiles, le décodage des codes correcteurs d'erreurs est réalisé à l'aide d'architectures matérielles dédiées. Ces architectures sont très efficaces en matière de débit, de latence et de consommation énergétique. Elles souffrent cependant d'un manque de flexibilité.

La description logicielle des algorithmes de décodage et leur exécution sur des processeurs généralistes apparaît comme une alternative prometteuse palliant ce manque de flexibilité. Des travaux récents montrent que de telles implémentations permettent d'atteindre de hauts débits associés à de faibles latences [3, 4]. Cette flexibilité se traduit par une meilleure évolutivité des systèmes et une intégration facilitée. Il est ainsi plus aisé de distribuer le calcul, pour s'approcher d'une infrastructure du type Cloud. Les travaux décrits dans cette thèse ont pour but d'étudier et de proposer des solutions pour le décodage de codes polaires sur des architectures programmables.

Structure du manuscrit de thèse

Le premier chapitre a pour thème les codes polaires. Après une brève introduction portant sur la composition d'une chaîne de communications numériques, les codes polaires et le principe de polarisation sont présentés. Ensuite, les différents algorithmes de décodage de codes polaires sont détaillés. Enfin, les méthodes d'élagage de l'arbre, qui constituent des versions simplifiées des algorithmes de décodage, sont présentées.

Un décodeur logiciel générique et flexible de codes polaires est proposé dans le chapitre 3. Il implémente les algorithmes de décodage à liste. Les concepts de généricité et de flexibilité du décodeur sont définis. La *généricité* du décodeur est sa capacité à supporter une très grande variété de codes polaires, en matière de taille de mot de code, de construction et de concaténation avec des codes détecteurs d'erreurs. Un décodeur générique peut donc s'adapter à un très grand nombre de schémas de codage. La *flexibilité* d'un décodeur est définie comme sa capacité à configurer l'algorithme de décodage dans le but de proposer des compromis entre débits, latences et performances de décodage. La version de l'algorithme, la taille de liste, le format de représentation des données internes sont des exemples de telles configurations. Plusieurs améliorations algorithmiques et d'implémentation sont appliquées à ce décodeur *générique* et *flexible*. Ainsi, il approche, atteint voire dépasse les décodeurs logiciels de l'état de l'art, selon les algorithmes et les niveaux de bruit du canal considéré. Ce décodeur est intégré au sein de la suite logicielle libre AFF3CT [59], le code source est donc disponible. Ceci permet son utilisation par l'ensemble de la communauté, ainsi que la reproduction de l'intégralité des résultats présentés dans ce deuxième chapitre.

Un défaut des décodeurs logiciels de ce type est toutefois l'énergie consommée. En effet, les processeurs considérés sont généralistes. Cela implique une grande complexité matérielle et une forte consommation énergétique. La seconde partie de nos travaux a pour but de proposer deux architectures programmables à faible consommation et haute performance pour le décodage de codes polaires. Le but de ces travaux est de conserver des architectures flexibles, tout en améliorant l'efficacité énergétique. Les chapitres 4 et 5 reposent sur deux méthodologies de conception différentes. Par conséquent, la structure des architectures résultantes ainsi que leurs performances sont radicalement différentes.

Dans le chapitre 4, la méthodologie considérée est celle de la spécialisation d'une architecture programmable de base. Il s'agit donc d'une approche par extension. Cette spécialisation est effectuée à l'aide des outils de la société Tensilica. Elle consiste à configurer une architecture de processeur de base puis à étendre son jeu d'instructions par l'ajout d'unités matérielles dédiées. Les processeurs ainsi conçus atteignent des débits comparables à ceux obtenus sur les processeurs d'architecture ARM, tout en réduisant la consommation énergétique d'un ordre de grandeur.

Dans le chapitre 5, la méthode de conception retenue permet une configuration plus fine du processeur. Cette approche implique la description de l'ensemble de l'architecture programmable. Sa structure ainsi que les modèles matériels des unités élémentaires du processeur sont alors entièrement définis par l'utilisateur. L'architecture programmable obtenue permet d'atteindre de très hauts débits et de faibles latences tout en réduisant fortement la complexité matérielle et donc la puissance dissipée. L'architecture proposée fait partie de la famille des TTA (Transport Triggered Architecture). Les résultats d'implémentations montrent que les décodeurs proposés dépassent les débits des implémentations logicielles sur architecture x86 et réduisent la consommation énergétique de deux ordres de grandeur tout en conservant une généricité et une flexibilité suffisante dans le contexte applicatif considéré.

Contributions des travaux de thèse

Les différentes contributions originales de ces travaux de thèse peuvent se résumer comme suit :

1. La proposition d'un décodeur logiciel pour les algorithmes de décodage de codes polaires à liste. Les caractéristiques principales de ce décodeur logiciel sont sa généricité et sa flexibilité. Les débits obtenus dépassent les débits des implémentations logicielles de la littérature sur des architectures de processeurs x86. Des résultats d'exécution sur architecture ARM sont également présentés, constituant la première référence de la littérature. Ces travaux sont détaillés dans le deuxième chapitre.

2. La proposition de la première architecture ASIP (Application Specific Instruction-set Processor) spécialisée dans le décodage de codes polaires. Cette architecture permet de conserver une grande flexibilité tout en améliorant l'efficacité énergétique. Les débits obtenus sont comparables à ceux de l'architecture ARM. La consommation énergétique est réduite d'un ordre de grandeur. Ces travaux sont présentés dans le troisième chapitre.
3. La proposition de la première architecture du type TTA (Transport Triggered Architecture) pour le décodage de codes polaires. Il s'agit d'un type spécifique d'ASIP. Tout comme les architectures précédentes, elle bénéficie d'une grande flexibilité. Elle permet une exploitation importante du parallélisme d'instructions et une définition fine de la structure. Ainsi, les débits obtenus dépassent ceux obtenus sur les architectures de processeur x86 et la consommation énergétique est quant à elle réduite de deux ordres de grandeur. Ces travaux sont détaillés dans le quatrième chapitre.

Ces différentes contributions ont été valorisées à travers des publications scientifiques :

- Conférences nationales sans acte :
  - M. Léonardon, C. Leroux, and C. Jégo, "Les Codes polaires, Algorithmes et Décodeurs." *CNES CCT TSI Technologies pour la 5G - segment spatial*, 2016.
  - A. Cassagne, M. Léonardon, O. Hartmann, T. Tonnellier, G. Delbergue, V. Giraud, C. Leroux, R. Tajan, B. Le Gal, C. Jégo, O. Aumage, and D. Barthou, "AFF3CT : Un Environnement de Simulation pour le Codage de Canal," *GdR SoC2*, 2017.
- Conférence internationale sans acte :
  - A. Cassagne, M. Léonardon, O. Hartmann, G. Delbergue, T. Tonnellier, R. Tajan, C. Leroux, C. Jégo, B. Le Gal, O. Aumage, and D. Barthou, "Fast Simulation and Prototyping With AFF3CT," in *IEEE International Workshop on Signal Processing Systems (SiPS)*, 2017.
- Conférences internationales avec actes :
  - A. Ghaffari, M. Léonardon, Y. Savaria, C. Jégo, and C. Leroux, "Improving Performance of SCMA MPA Decoders Using Estimation of Conditional Probabilities," in *IEEE International New Circuits and Systems Conference (NEWCAS)*, 2017.
  - M. Léonardon, C. Leroux, D. Binet, J. M. P. Langlois, C. Jégo, and Y. Savaria, "Custom Low Power Processor for Polar Decoding," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018.

    - M. Léonardon, C. Leroux, P. Jääskeläinen, C. Jégo, and Y. Savaria, "Transport Triggered Polar Decoders," in *IEEE International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*, 2018.

- Revues internationales avec comité de lecture :
    - M. Léonardon, A. Cassagne, C. Leroux, C. Jégo, L.-P. Hamelin, and Y. Savaria, "Fast and Flexible Software Polar List Decoders," *Journal of Signal Processing Systems (JSPS)*, accepted for publication, 2019.
    - A. Ghaffari, M. Léonardon, A. Cassagne, C. Leroux, and Y. Savaria, "Toward High Performance Implementation of 5G SCMA Algorithms," *IEEE Access*, accepted for publication, 2019.

CHAPITRE 2 LES CODES POLAIRES

Le but de ce premier chapitre est d'introduire le lecteur au codage polaire, avec une emphase particulière sur les algorithmes de décodage.

Dans la première section, le modèle de la chaîne de communications numériques utilisé tout au long du manuscrit est décrit, puis le principe et la construction des codes polaires sont détaillés. La deuxième section présente les principaux algorithmes de décodage de codes polaires. La troisième section porte sur une simplification algorithmique, l'élagage de l'arbre de décodage.

2.1 Le principe et la construction des codes polaires

2.1.1 La chaîne de communications numériques

Une chaîne de communications numériques telle que conceptualisée par Claude Shannon [1] est représentée en Figure 2.1. Elle représente les principales étapes de la transmission de données numériques depuis une **source** vers un **destinataire** à travers un **canal** de transmission. Ce dernier est le support sur lequel transite l'information. Lors de communications sans fil, il s'agit de l'ensemble constitué des antennes d'émission et de réception et de l'espace libre les séparant. Or les grandeurs physiques associées à ce canal sont continues, tandis que les données à transmettre sont constituées de bits, donc discrètes. Le **modulateur** transforme ce flux binaire en signaux physiques transmissibles par le canal. Si l'on considère toujours les communications sans fils, le modulateur transforme les séquences de bits en formes d'ondes. Au sein du canal de communication, les signaux subissent de nombreuses perturbations comme le bruit thermique des composants électroniques de la chaîne de transmission ou encore les interférences causées par d'autres utilisateurs du canal. Le démodulateur effectue la fonction duale du modulateur, il convertit les signaux physiques en données binaires. Si les perturbations du canal sont trop fortes, il est possible que le modulateur génère une estimation erronée du bit transmis. Ceci est un problème, puisque le but de la chaîne de transmission est de transmettre au destinataire l'exacte réplique de la **séquence d'information**, **b** émise par la source. Lorsque l'information reçue et l'information émise diffèrent, il y a une erreur de transmission. La qualité de la chaîne de transmission est souvent estimée à l'aide de son taux d'erreur binaire (BER), qui correspond à la proportion d'erreurs binaires par rapport au nombre total de bits transmis.

L'ajout d'un encodeur et d'un décodeur de canal dans la chaîne de transmission est un

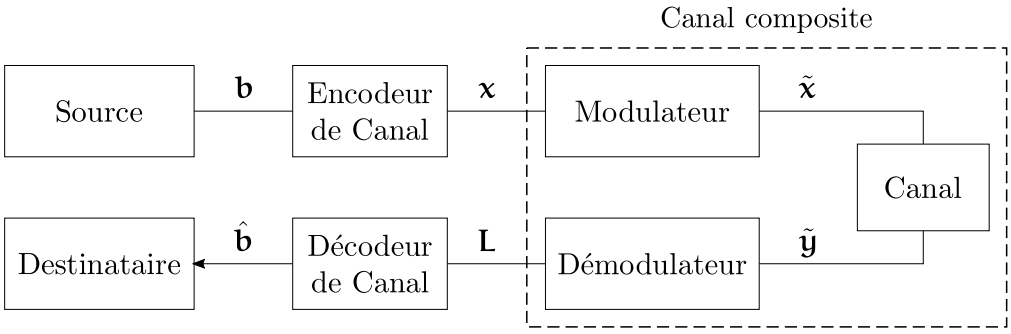


Figure 2.1 Modèle simplifié d'une chaîne de communications.

moyen efficace de réduire ce taux d'erreur. Dans le cas des codes en blocs étudiés dans ce manuscrit, l'encodeur transforme une séquence d'information $\mathbf{b}$ de K bits en un **mot de code** $\mathbf{x}$ de N bits. La taille du mot de code, en nombre de bits, est supérieure à la taille de la séquence d'informations afin d'ajouter de la redondance au message transmis : le rendement du code $R = K/N$ est inférieur à 1. Cette redondance est utilisée par le décodeur de canal, ou parfois le couple démodulateur - décodeur canal, afin d'améliorer l'estimation du message transmis et donc de réduire le taux d'erreur binaire de la chaîne de transmission. Dans ce manuscrit, le décodeur de canal est découplé du démodulateur. L'ensemble modulateur - canal - démodulateur peut être considéré comme une seule entité indépendante dont l'entrée est constituée du mot de code $\mathbf{x}$ et la sortie d'une séquence d'estimations de $\mathbf{x}$ notée $\mathbf{L}$. Cet ensemble est appelé canal composite.

2.1.2 Le canal composite

Un seul et unique modèle de canal composite sera considéré tout au long de ce manuscrit. Il est constitué d'une modulation à changement de phase binaire (BPSK) qui associe respectivement aux valeurs binaires d'entrée $x \in \{0,1\}$ les valeurs réelles $\tilde{x} \in \{-1,1\}$.

Le canal composite considéré est basé sur un canal à bruit blanc additif gaussien à entrée binaire (BI-AWGNC : Binary Input - Additive White Gaussian Noise Channel) tel que défini dans [**?**, section 1.5.1.3]. Ce modèle est classiquement utilisé pour caractériser les performances des codes correcteurs d'erreur. Il consiste en l'addition aux données de sortie du modulateur $\tilde{\mathbf{x}}$ d'une variable aléatoire à valeurs réelles ayant une distribution gaussienne centrée en 0 et de densité spectrale N_0. Afin d'évaluer les performances de correction des codes correcteurs d'erreurs, les taux d'erreurs seront souvent rapportés à cette densité spectrale, ou plus précisément au **rapport signal à bruit** (SNR), noté E_b/N_0, où $E_b = \frac{\mathbb{E}(\tilde{x}^2)}{R}$ est l'énergie moyenne par bit d'information.

Les estimations en sortie du démodulateur sont données sous la forme de rapport de vraisemblance logarithmique (LLR : Logarithmical Likelihood Ratio). Leur signe détermine pour chaque donnée de sortie du canal $\tilde{y}_i \in \tilde{\mathbf{y}}$ la valeur binaire d'entrée $x_i \in \mathbf{x}$ la plus probable. La valeur absolue correspond au degré de fiabilité de l'information. Plus de détails sur le modèle de canal et la définition des LLR sont donnés en Annexe A. L'expression mathématique de L_i est la suivante :

$$L_i = \log\left(\frac{P_r(y_i|x_i = 0)}{P_r(y_i|x_i = 1)}\right) \tag{2.1}$$

2.1.3 Les codes polaires

Les codes polaires [2] sont des codes en blocs linéaires [**?**]. Soient les ensembles $\mathbb{B}^n = \{0,1\}^n$, et une matrice $G \in \mathbb{B}^K \times \mathbb{B}^N$, l'encodage d'un code en blocs linéaire est une application linéaire injective de $\mathbb{B}^K$ vers $\mathbb{B}^N$ qui à un élément $\mathbf{b} \in \mathbb{B}^K$ associe un élément $\mathbf{x} \in \mathbb{B}^N$ tel que $x = \mathbf{b}G$. Les codes polaires peuvent ainsi être définis comme un ensemble de codes en blocs linéaires ayant une matrice génératrice avec une forme particulière.

La matrice génératrice d'un code polaire est elle-même le produit de deux matrices $E \in (\mathbb{B}^K \times \mathbb{B}^N)$ et $F^{\otimes n} \in (\mathbb{B}^N)^2$ tel que $G = EF^{\otimes n}$. La multiplication par la matrice E correspond à l'introduction de "bits gelés", dont la valeur est 0, à la séquence d'information $\mathbf{b}$. Le résultat de l'opération $\mathbf{u} = E\mathbf{b}$ est constitué de K bits $b_i \in \mathbf{b}$ et de $N - K$ bits gelés. Les positions respectives des bits d'informations et des bits gelés sont liées au phénomène de polarisation décrit dans la sous-section suivante. Ces positions déterminent largement le pouvoir de correction du code.

La matrice $F^{\otimes n}$, où $N = 2^n$, est la n-ième puissance de Kronecker du noyau $F = \left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right]$. $F^{\otimes n}$ peut être définie récursivement : $F^{\otimes 1} = F$ et $\forall n \geqslant 1$, $F^{\otimes n+1} = \left[\begin{smallmatrix} F^{\otimes n} & 0_n \\ F^{\otimes n} & F^{\otimes n} \end{smallmatrix}\right]$ avec $0_n \in \mathbb{B}^N$ une matrice nulle. La représentation sous forme de graphes de factorisation (*factor graph*), dans la Figure 2.2, est une représentation graphique du processus d'encodage de codes polaires où les symboles $\oplus$ correspondent à des opérations *ou-exclusif*.

Le processus d'encodage peut également être modifié pour rendre le code systématique. Un code correcteur d'erreur est dit systématique si les bits de la séquence d'information sont présents dans le mot de code. Pour ce faire, il faut modifier la matrice d'encodage comme démontré dans [**?**] : $G_{sys} = EF^{\otimes n}EF^{\otimes n}$. La représentation en graphe de factorisation de ce type de matrice est donnée en Figure 2.3. Lorsque cette matrice d'encodage est utilisée, les bits de la séquence d'information $\mathbf{b}$ se trouvent dans le mot de code $\mathbf{x}$, car si $\mathbf{x} = \mathbf{b}G_{sys}$ alors $\mathbf{b} = E^{-1}\mathbf{x}$.

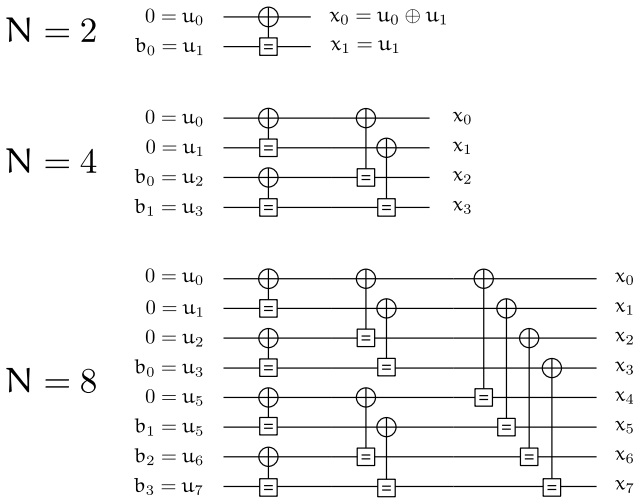


Figure 2.2 Représentation en graphe de factorisation de l'encodage de codes polaires.

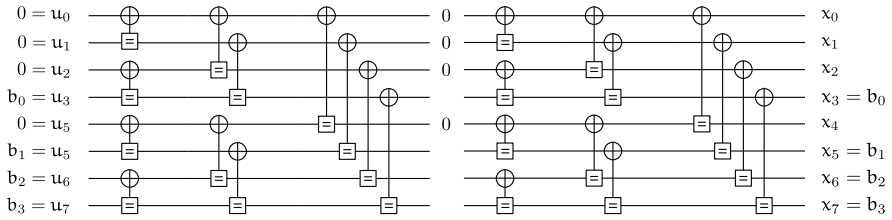


Figure 2.3 Encodage systématique.

2.1.4 Le phénomène de polarisation

La spécificité des codes polaires est liée au phénomène de polarisation. L'ensemble constitué de l'encodeur polaire, du canal composite et du décodeur polaire peut être modélisé comme un ensemble de N canaux de transmission. Chacun de ces canaux transmet un bit. Le terme « polarisation » exprime le fait que ces N canaux ont tendance à se diviser en deux groupes. Un groupe de canaux très fiables, avec une probabilité d'erreur très faible, et un groupe de canaux peu fiables, à probabilité d'erreur forte. Il est possible de montrer que la proportion de canaux très fiables, pour un rendement de code donné, tend vers la capacité du canal telle que définie dans la théorie de l'information [1] lorsque la taille du mot de code N tend vers l'infini. Lors de l'étape d'encodage, les bits d'information sont attribués aux canaux fiables et les bits dits gelés, dont la valeur est connue a priori, sont attribués aux canaux peu fiables. La transformation en produit de Kronecker étant commune à tous les codes polaires, la construction d'un code polaire consiste à déterminer la position des bits gelés dans le vecteur $\mathbf{u}$ pour une taille de code donnée. La méthode de détermination des positions des bits gelés est donnée dans [2] pour les canaux à effacement. Pour le canal AWGN, les premières

tentatives de construction de code polaire furent décrites dans [?, ?], mais les complexités calculatoires associées à ces méthodes sont significatives. Des méthodes simplifiées, utilisées dans la pratique, ont été proposées dans [?,?,?] pour le canal AWGN. La valeur des bits gelés, dans la plupart des cas, est 0. Toutefois, des travaux récents [?,?] montrent que de meilleures performances de décodage peuvent être atteintes en assignant des valeurs dynamiques à ces bits gelés. Ces valeurs dynamiques sont des combinaisons des valeurs des bits précédents. Par exemple, si u_j est un bit gelé, sa valeur peut être définie de la manière suivante : $u_j = \sum_{i=0}^{j-1} \alpha_i u_i$, avec $\alpha_i \in \{0,1\}$. Dans l'ensemble des simulations et des explications contenues dans ce manuscrit, il est cependant considéré que la valeur des bits gelés est 0.

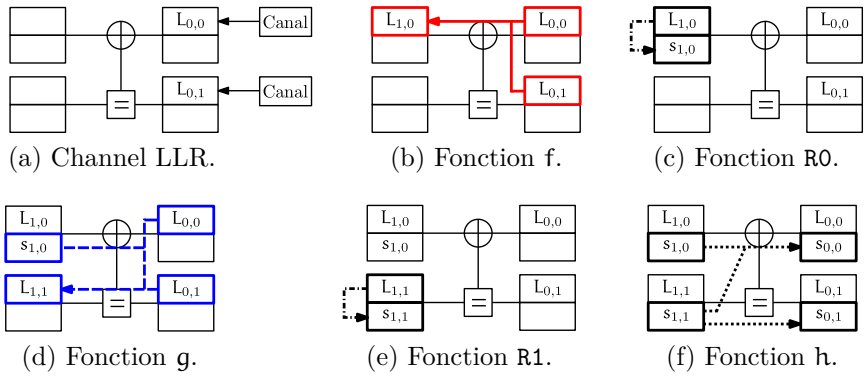
2.2 Les algorithmes de décodage de codes polaires

Le phénomène de polarisation et les codes polaires ont été développés en supposant l'utilisation de l'algorithme de décodage par Annulation Successive (SC : Successive Cancellation). Il s'agit d'un algorithme à sortie dure : les données de sortie sont des bits. Toutefois un deuxième algorithme de décodage par propagation de croyance (BP : Belief Propagation) fut également proposé. Celui-ci est à sortie souple. Cela signifie que les données de sortie sont des estimations représentées typiquement sous forme de LLR. Pour rappel, dans la représentation en LLR, le signe correspond à la décision dure : bit égal à 0 pour un LLR positif, bit égal à 1 pour un LLR négatif. La valeur absolue informe sur la fiabilité de cette décision. Bien que l'algorithme SC soit optimal asymptotiquement lorsque N tend vers l'infini, ses performances sont limitées pour des codes de taille finie. De nombreuses variantes de ces algorithmes ont donc été proposées par la suite afin d'en améliorer les performances de correction. Les algorithmes Annulation Successive par Liste (SCL) [26], Annulation Successive "Flip" (SCF) [?], Annulation Successive par Pile (SCS) [?] sont des évolutions de l'algorithme SC. Par ailleurs, l'algorithme appelé Annulation Souple (SCAN) [?] peut être vu comme une combinaison des algorithmes SC et BP. Chacun de ces algorithmes est décrit dans cette section.

2.2.1 L'algorithme de décodage par Annulation Successive (SC)

2.2.1.1 Le décodage par Annulation Successive sur le graphe de factorisation

Le décodage peut être représenté sous forme de graphe de factorisation. Lors de l'encodage, représenté en Figure 2.2, les bits d'information et les bits gelés sont à gauche du graphe et le parcours de gauche à droite de celui-ci permet de calculer les bits du mot de code. Lors du décodage, opération duale de l'encodage, le sens du parcours est inversé. Les informations du canal viennent de la droite (Figure 2.4a). Elles représentent les estimations des bits du mot de



(a) Channel LLR. (b) Fonction f. (c) Fonction R0.

(d) Fonction g. (e) Fonction R1. (f) Fonction h.

Figure 2.4 Fonctions élémentaires et séquencement du décodage SC d'un noyau $N = 2$.

code. Les étapes du décodage du noyau élémentaire de taille 2 sont détaillées en Figure 2.4.

Les données d'entrée des algorithmes de décodage présentés ici sont donc des LLR, contenus dans le vecteur $\mathbf{L}$ de taille N, sortis du canal composite. Les données de sorties de l'algorithme de décodage sont quant à elles des bits contenus dans le vecteur $\hat{\mathbf{b}}$ de taille K. Durant le processus de décodage de codes polaires, les deux formats de données, les LLR d'une part, et les sommes partielles (PS : Partial Sums) d'autre part, sont utilisés. Les LLR notés $L_{i,j}$ sont les estimations de la valeur du bit à la position (i,j) du graphe de factorisation. Les sommes partielles notées $s_{i,j}$ correspondent à la décision dure de ce même bit du graphe de factorisation. Huit variables sont nécessaires pour le décodage du noyau élémentaire de taille 2, 4 LLR et 4 sommes partielles.

Ces variables apparaissent dans la Figure 2.4 représentant les différentes étapes de l'algorithme de décodage SC. La première étape (a) du décodage est le chargement des LLR du canal L : les LLR $L_{0,0}$ et $L_{1,0}$ prennent la valeur des LLR du canal. Les LLR et les sommes partielles de chaque point du graphe de factorisation sont ensuite calculés par l'intermédiaire des opérations f, R0, g, R1 et h symbolisées dans la Figure 2.4 par des flèches. Elles correspondent aux équations suivantes :

$$\begin{aligned}
\mathtt{f}(L_a,L_b) &= \text{sign}(L_a.L_b).\min(|L_a|,|L_b|)\\
\mathtt{g}(L_a,L_b,\hat{s}_a) &= (1-2\hat{s}_a)L_a + L_b\\
\mathtt{h}(\hat{s}_a,\hat{s}_b) &= (\hat{s}_a \oplus \hat{s}_b,\hat{s}_b)\\
\mathtt{R0}(L_a) &= 0\\
\mathtt{R1}(L_a) &= \begin{cases} 0 \text{ si } L_a \geqslant 0\\ 1 \text{ si } L_a < 0 \end{cases}
\end{aligned} \tag{2.2}$$

La fonction f est appliquée dans l'étape (b). Elle permet le calcul de $L_{0,1}$. L'étape (c) est

l'application de la fonction R0 sur le nœud supérieur. La fonction R0 est appliquée dans l'hypothèse où le bit u_0 (Figure 2.2) est un bit gelé. La fonction g permet ensuite, lors de l'étape (d), le calcul de $L_{1,1}$. L'étape (e) est le calcul de $s_{1,1}$ par l'opération R1 correspondant à un bit d'information. Cette opération est un seuillage du LLR pour obtenir la somme partielle. Lorsque le seuillage est appliqué, l'information de fiabilité est perdue. Après le décodage de la somme partielle $s_{1,1}$, la fonction h est appliquée afin de propager les sommes partielles, lors de l'étape (f).

2.2.1.2 Le décodage par Annulation Successive sur un arbre binaire

La représentation en arbre, comme donnée en Figure 2.5, est une représentation alternative qui permet de mieux représenter certaines simplifications algorithmiques. Les données sont organisées en arbre binaire sur $\log_2(N)+1$ **niveaux**. Dans notre exemple pédagogique, l'arbre possède quatre niveaux. À chaque niveau, un certain nombre de **nœuds** sont attribués. La **racine** de l'arbre numérotée 0 est constituée d'un seul nœud. La racine contient N LLR et N sommes partielles. En parcourant l'arbre, à chaque niveau, le nombre de nœuds double. À l'inverse, les nombres de LLR et de sommes partielles de chaque nœud sont divisés par deux. Les **feuilles** sont les nœuds du dernier niveau de l'arbre (niveau 3 dans la Figure 2.5). Le traitement d'une feuille correspond à l'application des opérations R0 et R1. Chaque niveau d contient 2^d nœuds constitués de 2^{n-d} LLR et de 2^{n-d} sommes partielles, où $n = \log_2(N)$. Les sommes partielles sont propagées par la fonction h. Elles sont propagées jusqu'à un niveau qui dépend de l'indice de la feuille qui vient d'être traitée.

Dans le cas d'un code polaire non systématique, la séquence décodée $\hat{u}$ est composée des sommes partielles contenues par les feuilles de l'arbre de décodage. Lors d'un décodage non systématique, les sommes partielles contenues par les feuilles correspondent à la séquence décodée $\hat{u}$. La Figure 2.6 est une représentation plus synthétique de l'arbre de décodage. Les opérations 12 et 13 de la Figure 2.6 sont donc inutiles puisqu'elles servent seulement à calculer les sommes partielles du nœud racine. Dans le cas d'un code systématique, ce sont les sommes partielles du nœud racine qui correspondent au mot de code décodé. Le parcours de l'arbre est un parcours en profondeur. L'ordonnancement des différentes fonctions est explicité dans la Figure 2.6, où chaque flèche représentant une fonction est numérotée selon l'ordre d'exécution.

2.2.1.3 Les niveaux de parallélisme de l'algorithme de décodage SC

Dans la Figure 2.5, il apparaît que quatre fonctions f sont appliquées sur le nœud racine pour calculer les LLR de la branche de gauche. Ces quatre fonctions sont indépendantes :

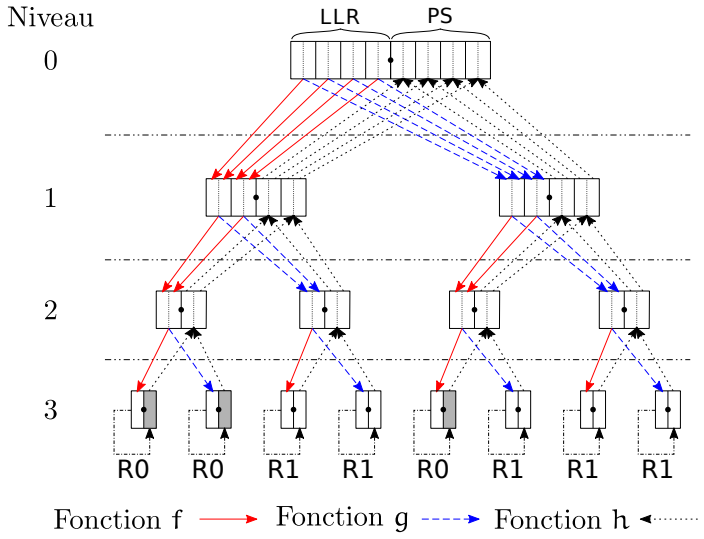


Figure 2.5 Arbre de décodage SC.

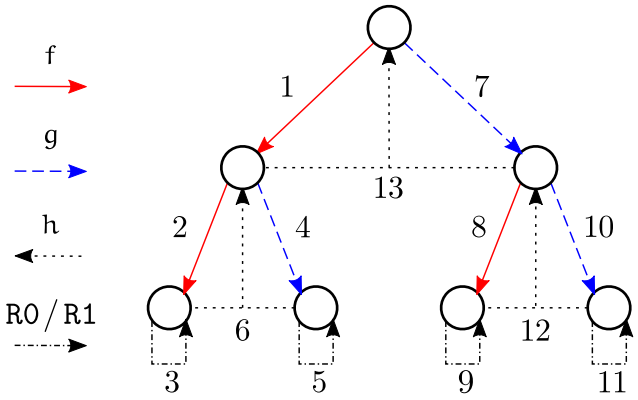


Figure 2.6 Ordonnancement du décodage SC d'un code polaire de taille $N = 4$.

leurs entrées et sorties sont disjointes. Elles peuvent donc être réalisées simultanément. Ce parallélisme existe pour chaque fonction f, g et h d'un nœud donné. Ainsi, le niveau de parallélisme est différent selon le niveau de profondeur dans l'arbre de décodage SC. Le niveau de parallélisme correspond à la taille du nœud cible. Plus on se rapproche des feuilles, plus la taille des nœuds est faible et, donc, plus le parallélisme est faible. Dans les implémentations logicielles ou matérielles de ces algorithmes, ce parallélisme est utilisé pour réduire le temps de décodage et ainsi augmenter son débit. Lorsque ce parallélisme est utilisé, on parle de parallélisme *intra-trame*. Un autre type de parallélisme existe et est appelé parallélisme *inter-trame*. Il désigne la possibilité, dans les implémentations de décodeurs, de traiter plusieurs trames simultanément. Il possède le désavantage, contrairement au parallélisme *intra-trame*, d'augmenter la latence de décodage. Généralement, la sélection de l'un ou l'autre des types

de parallélisme correspond à un compromis entre le débit et la latence de décodage.

2.2.2 L'algorithme de décodage SC Liste

2.2.2.1 L'algorithme de décodage

L'algorithme de décodage par Annulation Successive Liste (SCL) est une évolution de l'algorithme SC [26]. L'algorithme SC présente en effet des performances de correction médiocres pour des codes polaires de petite ou moyenne taille ($N < 8192$). L'algorithme de décodage SCL améliore ces performances substantiellement. Dans l'algorithme de décodage SC décrit précédemment, des décisions dures sont appliquées à chaque traitement d'une feuille correspondant à un bit d'information (fonction `R1`). Si une erreur est obtenue lors de ce seuillage, celle-ci est irréversible. Cela signifie que le décodage du mot de code est un échec. Le principe de l'algorithme de décodage par liste est de retarder la décision dure. Au lieu d'appliquer un seuillage sur la valeur du LLR, les deux possibilités de décodage sont considérées. Pour ce faire, l'arbre de décodage, avec l'ensemble de ses LLR et sommes partielles, est dupliqué. Cette opération est illustrée dans la Figure 2.7.

L'algorithme est représenté en cours d'exécution, à l'étape (i), au cours de laquelle la fonction f est appliquée. L'étape suivante (i+1) consiste en la duplication de l'arbre de décodage afin de former deux chemins. Dans le premier, l'hypothèse est que $\hat{u}_2 = 0$. Dans le second, l'hypothèse est que $\hat{u}_2 = 1$. Le décodage se poursuit en parallèle sur les deux arbres. L'étape suivante (i+2) correspond à l'application de la fonction g. Les deux arbres sont qualifiés de **chemins** de décodage. À chaque feuille de rendement 1, le même procédé est appliqué, doublant le nombre d'arbres à décoder en parallèle comme durant l'étape (i+3).

Il n'est pas possible, en pratique, de dupliquer indéfiniment le nombre d'arbres de décodage, la mémoire nécessaire et le nombre de calcul devient vite insoutenable. Un nombre de chemins maximum L est donc un paramètre de l'algorithme liste. La sélection des chemins qui seront conservés ou éliminés est réalisée à l'aide d'une **métrique** m_j^i associée à chaque chemin. Cette métrique est mise à jour à chaque traitement d'une feuille de l'arbre. Lors des autres étapes, par exemple l'application de la fonction g en étape (i+2), les métriques sont inchangées, $m_j^{i+2} = m_j^{i+1}$. Le détail des calculs de métriques lors du traitement d'une feuille pour un décodage utilisant des LLR est donné dans [**?**]. Deux cas de figure sont possibles : soit la feuille correspond à un bit gelé, soit elle correspond à un bit d'information.

Lorsque la feuille correspond à un bit gelé, cela veut dire que sa somme partielle est égale à 0. Lorsque le LLR de cette feuille est calculé, il y a deux cas de figure. Ou bien celui-ci est positif, auquel cas la métrique est inchangée, car l'estimation du LLR est juste. En revanche, si le

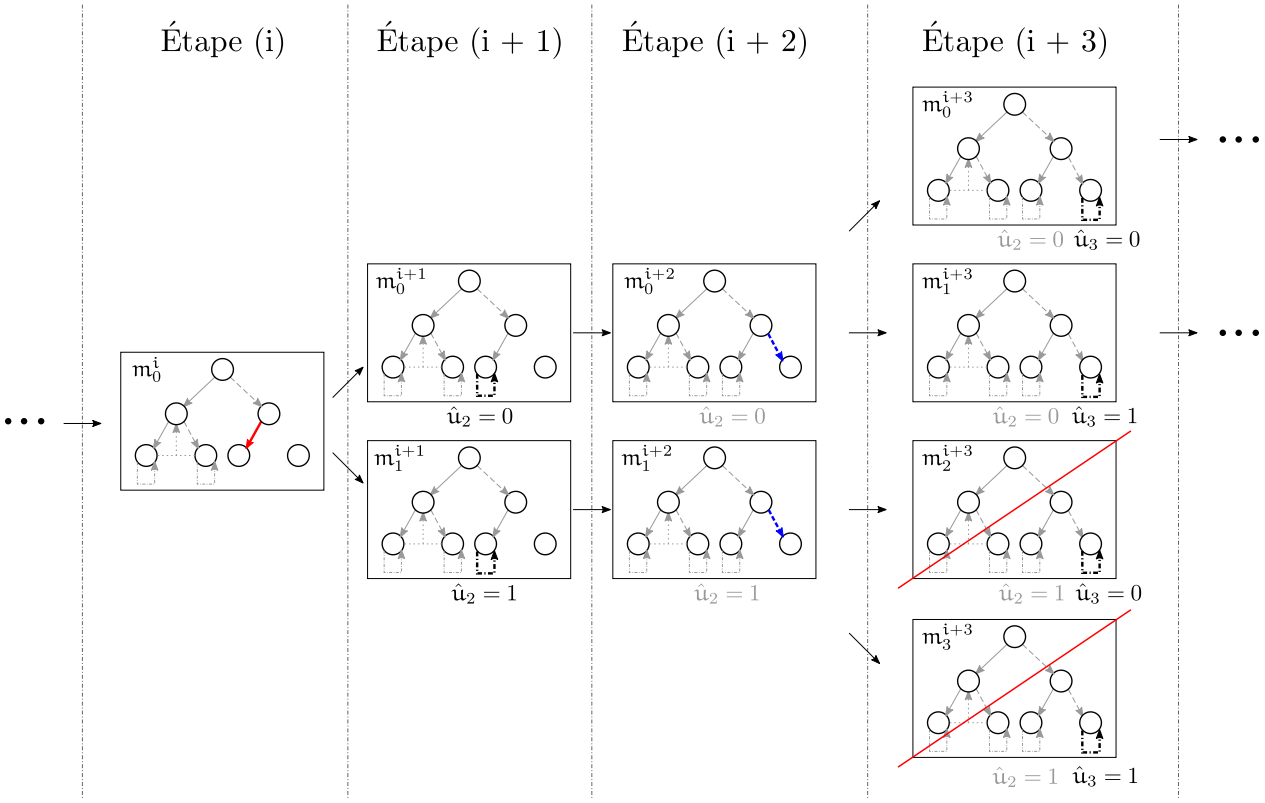


Figure 2.7 Duplication de l'arbre de décodage dans l'algorithme SCL.

LLR est négatif, alors la valeur absolue du LLR lui est ajoutée. On peut interpréter cette addition comme une « pénalité » pour avoir fait une mauvaise estimation. Soit i_0 l'indice correspondant à une étape de traitement d'une feuille de rendement 0, et $L_j^{i_0}$ la valeur de son LLR, alors, pour un chemin donné d'indice j,

$$m_j^{i_0} = \begin{cases} m_j^{i_0-1} & \text{si } L_j^{i_0} \geqslant 0 \\ m_j^{i_0-1} + |L_j^{i_0}| & \text{si } L_j^{i_0} < 0 \end{cases} \tag{2.3}$$

Lorsque la feuille est de rendement 1, deux chemins sont créés à partir de chaque chemin actif, avec chacun une version différente du bit décodé. Le chemin créé dont la valeur du bit est en accord avec la valeur du LLR ne reçoit pas de pénalité. Au contraire, la métrique du second chemin créé est augmentée de la valeur du LLR. Soit i_1 l'étape de décodage d'une

feuille de rendement 1,

$$m_j^{i_1} = \begin{cases} m_j^{i_1-1} & \text{si } L_j^{i_1} \geqslant 0 \text{ et } \hat{u}_j^{i_1} = 0 \\ m_j^{i_1-1} + |L_{i_1}^j| & \text{si } L_j^{i_1} < 0 \text{ et } \hat{u}_j^{i_1} = 0 \\ m_j^{i_1-1} + |L_{i_1}^j| & \text{si } L_j^{i_1} \geqslant 0 \text{ et } \hat{u}_j^{i_1} = 1 \\ m_j^{i_1-1} & \text{si } L_j^{i_1} < 0 \text{ et } \hat{u}_j^{i_1} = 1 \end{cases} \tag{2.4}$$

Une fois les calculs de métriques réalisés, les chemins à conserver sont sélectionnés. Si L chemins étaient actifs au moment du traitement d'une feuille de rendement 1, alors 2L candidats sont générés, avec chacun une métrique. Seuls les L candidats avec la métrique la plus faible sont conservés. Cette opération de sélection est décrite dans la Figure 2.7. Lors de la dernière étape, à droite de la figure, les deux candidats du bas sont éliminés. L'hypothèse était donc que ces deux chemins avaient des métriques de valeurs absolues supérieures à celles des deux candidats du haut. À la fin du parcours de l'arbre de décodage, L chemins seront donc conservés. Ces L chemins correspondent à L mots de codes candidats. Une sélection d'un seul candidat parmi les L doit alors être effectuée. Dans la version originale de l'algorithme, le mot décodé est celui ayant la métrique la plus faible à la fin du décodage. Il est désigné comme étant la sortie de l'algorithme $\hat{\mathbf{b}}$.

2.2.2.2 Concaténation avec un CRC

Les auteurs de [**?**] ont proposé de concaténer un test de redondance cyclique (CRC : Cyclic Redundancy Check) pour discriminer les différents candidats. L'encodage correspondant est représenté en Figure 2.8. Un CRC est un code détecteur d'erreur. Il associe à une séquence binaire quelconque, ici $\mathbf{b}$ de longueur K, une séquence de longueur $K + c$. Après le décodage, la vérification du CRC permet de détecter de potentielles erreurs. Le CRC est construit de manière à ce que la probabilité d'erreur non détectée soit extrêmement faible. Ainsi, à la fin de l'algorithme de décodage, lorsque les L mots de codes candidats en sortie de l'algorithme SCL sont disponibles, une vérification des CRC est effectuée. Si l'un d'entre eux vérifie le CRC, il est alors hautement probable qu'il soit le bon candidat. Dans le cas rare où plusieurs candidats vérifient le CRC, celui ayant la métrique la plus faible parmi eux est sélectionné. Lorsque ce mécanisme de sélection des candidats par la vérification du CRC est appliqué, l'algorithme est appelé « SCL aidé d'un CRC » (CASCL : CRC-Aided SCL).

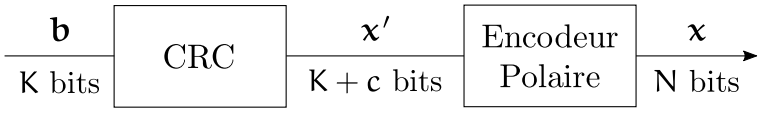


Figure 2.8 Concaténation avec un CRC.

2.2.2.3 Les algorithmes de décodage SCL adaptatifs

L'algorithme CASCL permet d'améliorer significativement les performances de décodage en comparaison de l'algorithme SC. Cette amélioration se fait toutefois au prix d'une augmentation de la complexité calculatoire. Cependant, il est possible de combiner les deux algorithmes, SC et CASCL, afin de se rapprocher de la faible complexité calculatoire du premier tout en conservant les performances de décodage du second [**?**]. Le principe est le suivant : une première tentative de décodage est réalisée avec l'algorithme SC. L'algorithme CASCL n'est ensuite déclenché que si une erreur de décodage est détectée par la vérification du CRC.

Deux variantes de l'algorithme adaptatifs sont possibles. Dans le cas de l'algorithme Partiellement Adaptatif (PASCL) présenté dans [18], lors de l'échec du décodage SC, l'algorithme CASCL est exécuté une seule fois avec comme taille de liste L_{max}. Dans le cas de l'algorithme Complètement Adaptatif (FASCL) original [**?**], l'algorithme CASCL peut être appliqué plusieurs fois. La première tentative de décodage CASCL est réalisée avec , $\mathrm{L} = 2$. Si après décodage, le CRC n'est toujours pas satisfait, l'algorithme est relancé autant de fois que nécessaire en doublant itérativement L jusqu'à atteindre L_{max}.

Ces algorithmes permettent de favoriser des architectures de décodage atteignant de hauts débits. Cependant, ces algorithmes ont également une latence maximum plus grande par rapport à l'algorithme CASCL avec $\mathrm{L} = \mathrm{L}_{max}$. Soit cette latence, $\mathcal{L}_{\mathrm{SCL}}(\mathrm{L}_{max})$, et la latence de l'algorithme SC, $\mathcal{L}_{\mathrm{SC}}$, alors dans le Pire Cas (PC),

$$\begin{cases} \mathcal{L}^{\mathrm{PC}}_{\mathrm{PASCL}} &= \mathcal{L}_{\mathrm{SC}} + \mathcal{L}_{\mathrm{SCL}}(\mathrm{L}_{max}) \\ \mathcal{L}^{\mathrm{PC}}_{\mathrm{FASCL}} &= \mathcal{L}_{\mathrm{SC}} + \sum_{i=1}^{\log_2(\mathrm{L}_{max})} \mathcal{L}_{\mathrm{SCL}}(2^i) \end{cases} \tag{2.5}$$

2.2.3 L'algorithme de décodage SC *Flip*

L'algorithme de décodage par Annulation Successive « Flip » (SCF : Successive Cancellation Flip) [**?**] est une variante de l'algorithme SC. Il vise à améliorer les performances de décodage. Cet algorithme nécessite une concaténation du code polaire avec un CRC pour être appliqué. L'algorithme de décodage SC est appliqué une première fois. L'ensemble des LLR associés

aux décisions dures (les LLR des feuilles) est conservé. Si le mot de code décodé satisfait le CRC, alors le décodage s'arrête. Par contre, si le CRC n'est pas satisfait, alors l'algorithme SC est exécuté une nouvelle fois. La différence est que la décision prise, lors du premier décodage, sur le LLR le moins fiable est inversée. Ce deuxième mot décodé est encore testé à l'aide du CRC. S'il ne satisfait toujours pas le CRC, une autre séquence de décodage est lancée en inversant le bit du deuxième LLR le moins fiable. Ce mécanisme itératif continue jusqu'à un nombre d'essais maximum T.

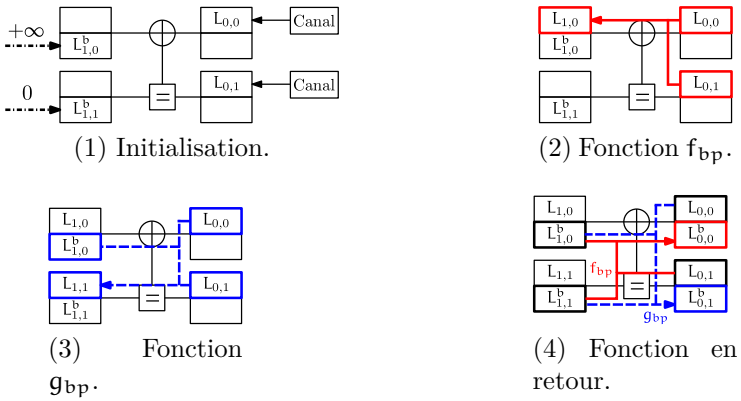
2.2.4 L'algorithme de décodage SC *Stack*

L'algorithme de décodage par Annulation Successive à Pile (SCS : Successive Cancellation Stack) [**?**] est une variante de l'algorithme SCL. Dans l'algorithme SCL, le nombre de candidats L est constant tout au long de l'algorithme (hormis lors des premières duplications). De plus, le séquencement est exactement le même que pour l'algorithme SC dans les différents arbres décodés parallèlement. Les feuilles sont décodées l'une après l'autre, suivant l'ordre établi. Dans l'algorithme SCS, les métriques associées aux différents arbres de décodage sont maintenues dans une pile ordonnée de profondeur T. L'algorithme SC est appliqué sur le chemin associé à la meilleure métrique. A l'arrivée sur une feuille, la métrique de l'arbre est mise à jour. Si la feuille est de rendement 1, un nouveau chemin est ajouté à la pile. À un instant t, seul l'arbre étant le plus haut sur la pile est décodé. Lorsque la dernière feuille de l'un des chemins est décodée, le CRC est testé. Le décodage se termine lorsqu'un mot décodé satisfait le CRC ou que le nombre de candidats testés atteint une valeur déterminée à l'avance D. L'algorithme SCS présente l'avantage de nécessiter moins d'opérations élémentaires pour le décodage des codes polaires. Cependant, le maintien d'une pile et son ordonnancement particulier rendent difficile son implémentation.

2.2.5 Les algorithmes itératifs à sortie souple

Deux algorithmes itératifs à sortie souple ont été proposés. Le premier est appelé algorithme à Propagation de Croyance (BP). C'est le premier dans l'ordre chronologique puisqu'il fut proposé dans l'article original d'Arıkan [2]. Le second fut proposé en 2014 dans [**?**]. Les deux algorithmes ne diffèrent que par l'ordre dans lequel les LLR du graphe de factorisation sont mis à jour durant chaque itération.

La première différence entre ces deux algorithmes et ceux vues précédemment est qu'aucune décision dure n'est faite durant le décodage. L'information reste souple de bout en bout. En conséquence, les sommes partielles utilisées dans l'arbre de décodage SC sont remplacées par des LLR. Deux ensembles de LLR sont donc contenus dans l'arbre de décodage. Les LLR qui



(1) Initialisation. (2) Fonction $\mathtt{f_{bp}}$. (3) Fonction $\mathtt{g_{bp}}$. (4) Fonction en retour.

Figure 2.9 Fonctions élémentaires et séquencement du décodage SCAN du noyau $\mathtt{N} = 2$.

ont remplacé les sommes partielles sont notés $\mathtt{L}^{\mathtt{b}}_{\mathtt{i},\mathtt{j}}$. Ensuite, les fonctions élémentaires sont changées. les fonctions $\mathtt{f}$, $\mathtt{g}$ et $\mathtt{h}$ de l'algorithme SC sont remplacées par les fonctions $\mathtt{f_{bp}}$ et $\mathtt{g_{bp}}$:

$$\begin{aligned} \mathtt{f_{bp}}(\mathtt{L_a},\mathtt{L_b},\mathtt{L_c}) &= \mathtt{f}(\mathtt{L_a},\mathtt{L_b}+\mathtt{L_c}) \\ \mathtt{g_{bp}}(\mathtt{L_a},\mathtt{L_b},\mathtt{L_c}) &= \mathtt{f}(\mathtt{L_a},\mathtt{L_c})+\mathtt{L_b} \end{aligned} \tag{2.6}$$

Le séquencement de ces opérations est illustré en Figure 2.9 pour le noyau de taille $\mathtt{N} = 2$ dans le cas d'un décodage SCAN. La première étape est l'initialisation de la racine et des feuilles de l'arbre. Les LLR $\mathtt{L}$ de la racine prennent la valeur des LLR du canal. Les LLR $\mathtt{L}^{\mathtt{b}}$ des feuilles prennent la valeur $+\infty$ dans le cas d'un bit gelé. et la valeur 0 dans le cas d'un bit d'information. Ensuite, les fonctions $\mathtt{f_{bp}}$ et $\mathtt{g_{bp}}$ sont appliquées dans l'ordre indiqué. Comme évoqué précédemment, les algorithmes SCAN et BP diffèrent du point de vue de leurs séquencements. Le séquencement de l'algorithme SCAN est le même que celui du SC, en remplaçant l'application de la fonction $\mathtt{h}$ du SC par une application conjointe des fonctions $\mathtt{f_{bp}}$ et $\mathtt{g_{bp}}$ comme illustré en Figure 2.94.

Enfin, les algorithmes BP et SCAN sont des algorithmes itératifs. En effet, le parcours de l'arbre peut être effectué plusieurs fois. Ainsi, les valeurs des LLR intermédiaires évoluent d'une itération à l'autre. Ces itérations permettent d'améliorer les performances de décodage. Par ailleurs, l'intérêt des algorithmes de décodage à sorties souples est qu'ils peuvent être utilisés dans un processus itératif si le code polaire est concaténé avec un autre code [**?**] ou bien dans le cadre d'une modulation codée [**?**].

2.2.6 Les performances de décodage des différents algorithmes

Le modèle du canal composite utilisé est celui décrit dans la sous-section 2.1.2. La méthode d'approximation gaussienne est utilisée pour déterminer les positions des bits gelés. Dans la sous-section courante sont présentées plusieurs courbes de performances. Elles permettent de comparer entre eux les algorithmes décrits précédemment. Ces courbes montrent également l'impact des paramètres, communs à tous les algorithmes, ou spécifiques à chacun, sur les performances de décodage. Ces paramètres et leurs effets sont importants car ce sont les leviers utilisés dans les standards de communication afin de s'adapter aux contraintes systèmes. Les axes des différentes courbes sont toujours les mêmes. En abscisse se trouve le rapport signal à bruit noté $\mathsf{E_b}/\mathsf{N_0}$ décrit en sous-section 2.1.2. Sur l'axe des ordonnées se trouve soit le taux d'erreur binaire (BER : Bit Error Rate) dont l'expression est donnée en sous-section 2.1.1, soit le taux d'erreur trame (FER : Frame Error Rate). Le FER est égal au rapport du nombre de trames erronées sur le nombre total de trames émises, sachant qu'une trame est considérée erronée si elle contient au moins un bit erroné. Sauf indication contraire, toutes les simulations ont été réalisées en utilisant le logiciel AFF3CT [1] développé par l'équipe CSN du laboratoire IMS.

2.2.6.1 Impact de la taille du mots de code et du rendement

La théorie de l'information énoncée par Shannon [1] prouve qu'il existe une limite théorique à la capacité d'un canal pour un rendement de code donné. Cette limite est matérialisée dans la Figure 2.10 par la ligne verticale dénommée *Limite de Shannon*. Les codes polaires sont les premiers codes correcteurs d'erreurs qui atteignent théoriquement cette limite. Cependant cette limite n'est atteinte que pour une taille de mot de code infinie.

La Figure 2.10 présente les performances de décodage de l'algorithme SC pour une taille de mot code croissante. S'il apparaît clairement que le taux d'erreur binaire diminue pour un rapport signal à bruit donné, la limite de Shannon est toutefois très éloignée pour les tailles de mot de code présentées.

1. AFF3CT est un logiciel libre permettant la simulation efficace de codes correcteurs d'erreurs : https ://aff3ct.github.io/

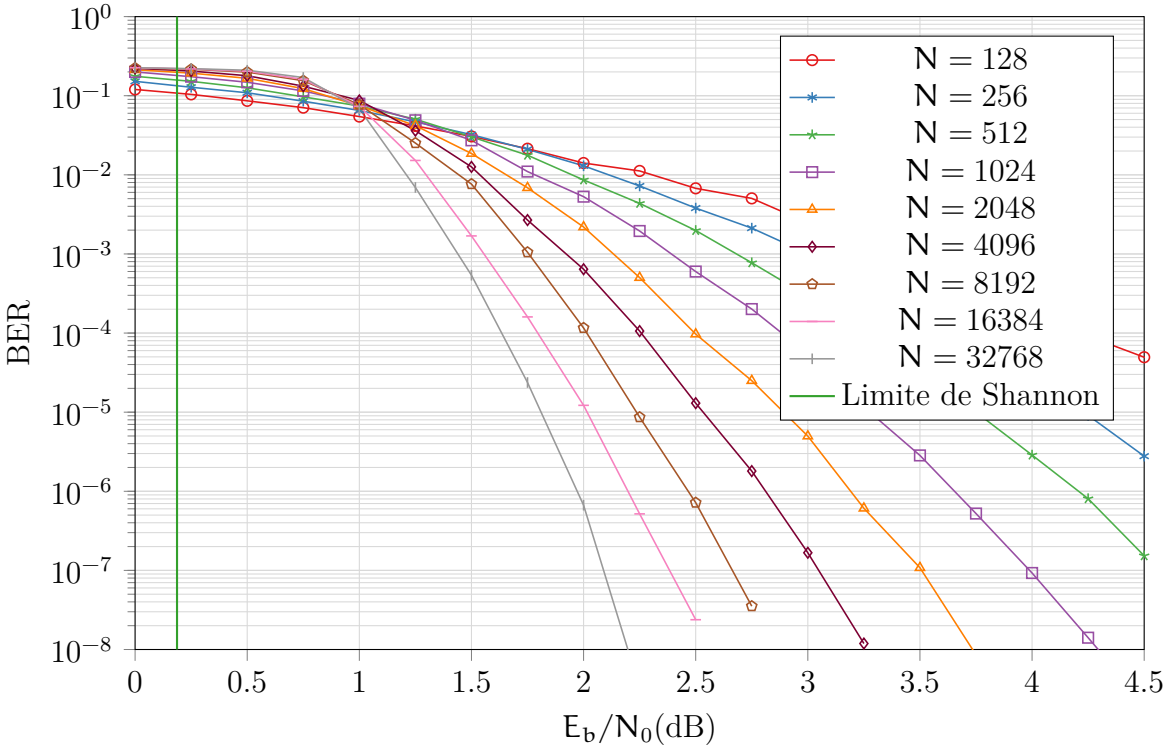


Figure 2.10 Performances de décodage de l'algorithme SC pour différentes valeurs de N (R = 1/2).

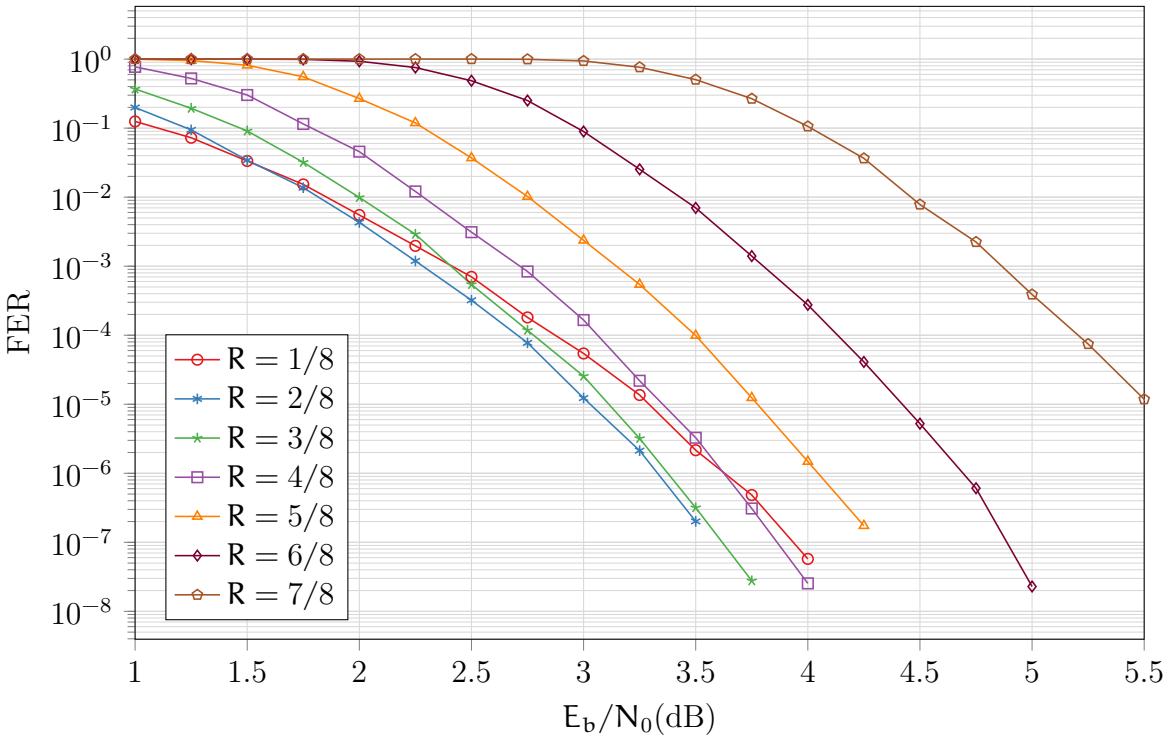


Figure 2.11 Performances de décodage de l'algorithme SC pour différentes valeurs de R (N = 2048).

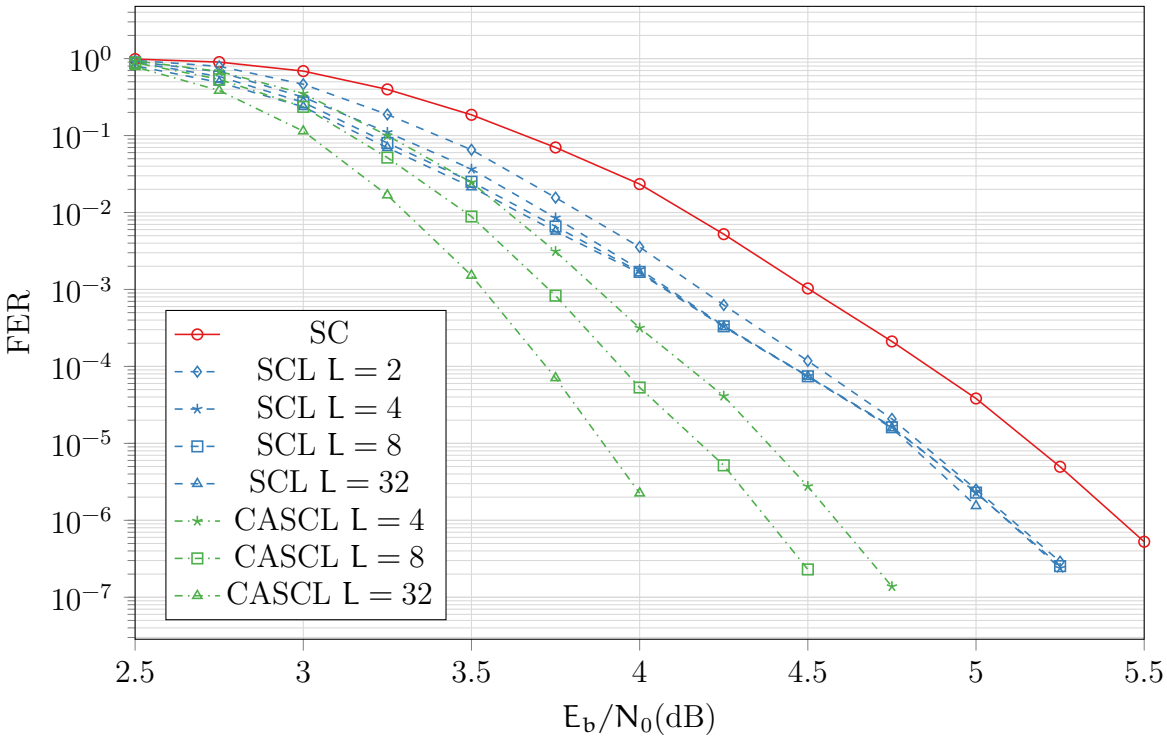


Figure 2.12 Performances de décodage des algorithmes SC, SCL et CASCL pour un code polaire (2048,1723). Un CRC de taille $\mathbf{c} = 16$ est utilisé pour l'algorithme CASCL.

Les codes correcteurs d'erreurs reposent la notion de redondance, quantifiée par le rendement du code. Plus le rendement est faible, plus la redondance est élevée. Ceci est illustré en Figure 2.11, toujours en considérant l'algorithme SC. La tendance observée est bien que plus le rendement est faible, plus le taux d'erreur est faible. Une seule exception apparaît, pour le rendement $R = 1/8$, rendement le plus faible pourtant. Ceci est dû au fait que l'énergie par bit d'information E_b prend en compte le fait que lorsqu'on diminue le rendement, on augmente l'énergie dépensée pour transmettre un bit utile. Deux tendances s'affrontent donc : augmentation de la redondance contre diminution du rapport signal à bruit.

2.2.6.2 Impact des paramètres de l'algorithme SCL

La Figure 2.12 présente les performances de l'algorithme SCL. Le code considéré est un code dont la taille du mot de code est 2048 et le nombre de bits d'information est 1723. La notation employée pour désigner un tel code est : (2048,1273). Comme précisé auparavant, l'algorithme SCL permet d'améliorer les performances de décodage des codes polaires par rapport à l'algorithme SC. Plusieurs choses sont à observer. Tout d'abord, l'augmentation de la taille de la liste permet dans tous les cas une meilleure correction d'erreurs. Ensuite, l'utilisation de la concaténation du CRC par l'algorithme CASCL augmente très fortement les performances de décodage. Par exemple lorsque L = 32, à un FER de 10^{-5}, il y a une différence de 1 dB entre les algorithmes SCL et CASCL.

Des tendances plus fines se dégagent. Les courbes de l'algorithme SCL montrent que les gains de performance résultant de l'augmentation du paramètre L deviennent marginaux. En effet, les performances obtenues pour L = 8 et L = 32 sont très proches, alors que la complexité de l'algorithme augmente approximativement linéairement. En revanche, concernant l'algorithme CASCL, l'amélioration des performances due à l'augmentation du paramètre L est continue et significative. Par exemple, pour $E_b/N_0 = 4\text{dB}$, le gain de FER entre L = 8 et L = 32 est de plus d'une décade.

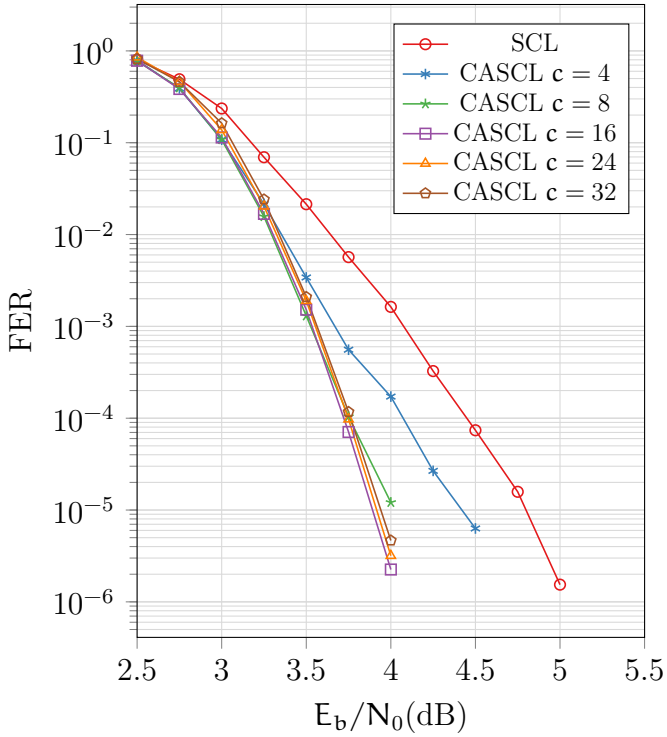
Pour un code polaire concaténé avec un CRC donné, la taille du CRC a également un impact sur les performances de décodage. Là encore, deux phénomènes antagonistes s'affrontent. Tout d'abord, plus le CRC est long, moins il est probable que se produise une validation erronée d'une trame fausse, appelée fausse détection. Ceci apparaît pour des valeurs élevées de E_b/N_0 : les algorithmes pour lesquels c est trop petit sont moins performants. Cependant, sur la Figure 2.13a, jusqu'à $E_b/N_0 = 4\text{dB}$, la version de l'algorithme la plus performante est celle pour laquelle $c = 16$, et non pas $c = 32$. En effet, il est montré dans la Figure 2.8 que le nombre de bits d'information par trame est K mais que le nombre de bits à l'entrée de l'encodeur polaire est $K+c$. Autrement dit, bien que le rendement total du code concaténé soit

$R = \frac{K}{N}$, le rendement du code polaire seul est $R = \frac{K+c}{N}$. Lorsque c augmente, le rendement du code polaire seul augmente également, et ses performances de décodage diminuent. Par ailleurs, il est important de noter que pour une taille de CRC donnée, le polynôme choisi influe également sur les performances comme étudié dans [**?**].

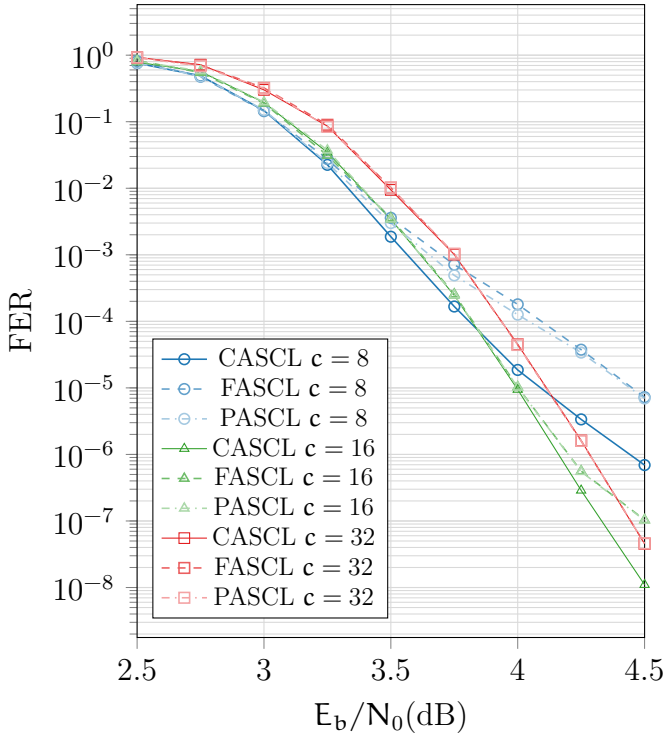
L'impact de la taille du CRC sur les performances de décodage des algorithmes adaptatifs est encore plus important. Celles-ci sont représentées pour un code polaire (2048,1723) et différentes tailles de CRC. En utilisant un CRC de taille $c = 8$ ou $c = 16$ on observe des dégradations importantes des performances entre les algorithmes adaptatifs d'une part et l'algorithme SCL de base d'autre part. La différence entre PASCL et FASCL est en outre négligeable. Ce phénomène s'explique par le fait qu'à des FER très faibles, il devient davantage probable d'obtenir des faux positifs lors du test de CRC durant premier décodage SC dans le cadre des algorithmes adaptatifs. Toutefois, pour un CRC de taille $c = 32$, il n'y a aucune différence de performances pour le code polaire étudié entre les algorithmes CASCL, PASCL et FASCL.

2.2.6.3 Performances de décodages des algorithmes SCAN et SCF

Les Figures 2.14 et 2.15 présentent les performances de décodages de ces deux algorithmes pour un code polaire (1024,512). Les courbes de performances pour l'algorithme SCF sont issues de [**?**] tandis que les courbes de performances de l'algorithme BP sont issues de [**?**]. Les performances du SCF s'améliorent lorsque T augmente mais restent inférieures aux performances de l'algorithme SCL avec $L = 4$. Le même ordre de grandeur est observé pour un algorithme SCAN à 4 itérations. Il est notable de remarquer que les performances du SCAN avec 4 itérations sont bien meilleures que celles de l'algorithme BP pour 50 itérations. En effet, à l'inverse du décodage SCAN, le séquencement du décodage BP ne tire aucun bénéfice du phénomène de polarisation car chaque bit est estimé en parallèle.



(a) Algorithmes non adaptatif.



(b) Algorithmes adaptatifs.

Figure 2.13 Impact de la taille du CRC sur les performances de l'algorithme SCL pour un code (2048,1723).

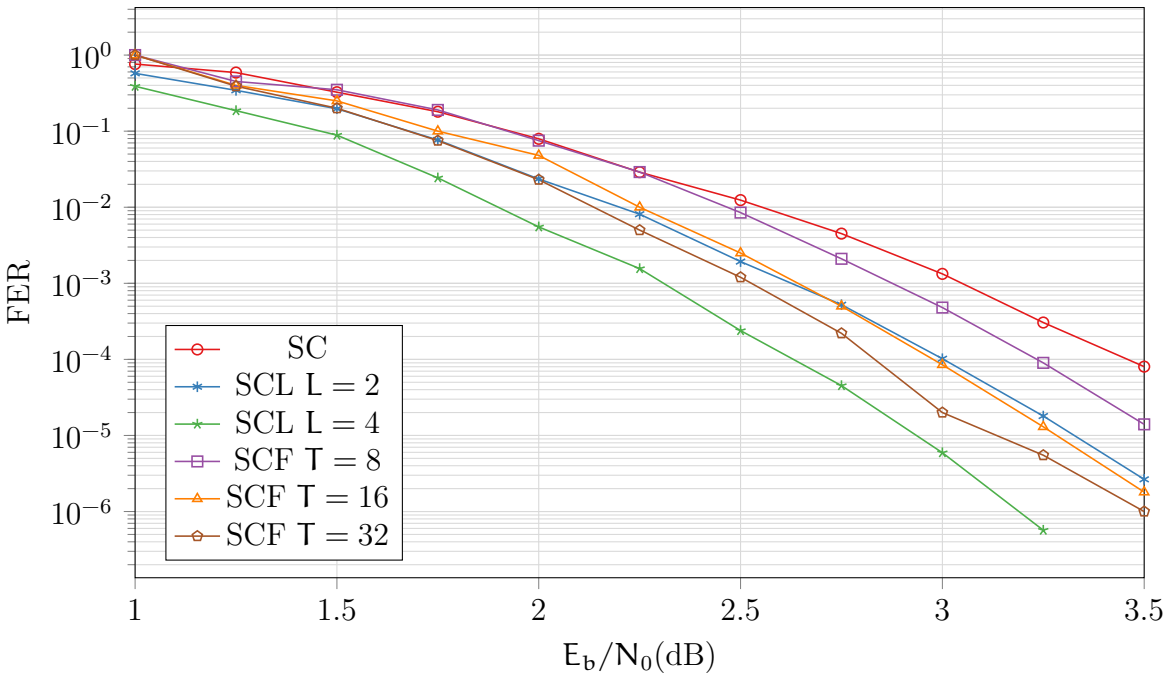


Figure 2.14 Performances de l'algorithme SCF en comparaison avec les algorithmes SC et SCL pour un code polaire (1024,512), $c = 16$.

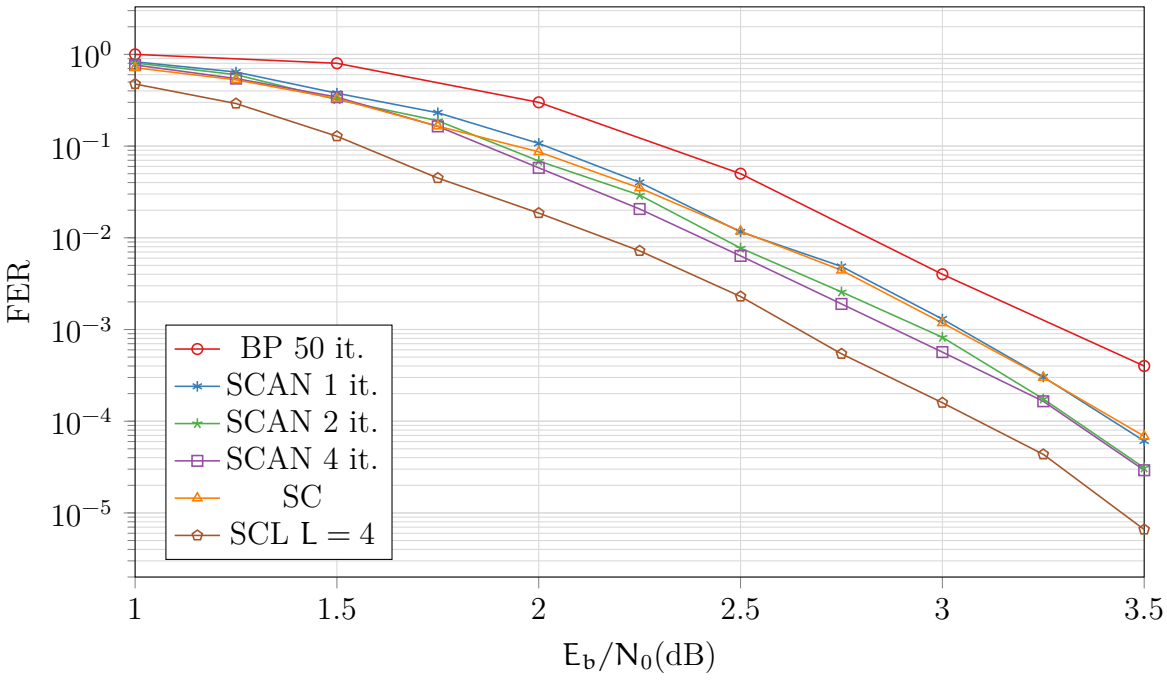


Figure 2.15 Performances des algorithmes itératifs à sortie souple en comparaison avec les algorithmes SC et SCL pour un code polaire (1024,512).

2.3 L'élagage de l'arbre de décodage

2.3.1 L'élagage de l'algorithme de décodage SC

Afin de réduire la complexité calculatoire de l'algorithme de décodage SC, il est possible d'élaguer l'arbre de décodage comme indiqué dans [**?**]. En effet, un sous-arbre dont toutes les feuilles sont de rendement 0 n'a pas besoin d'être parcouru. Comme toutes ses feuilles sont des bits gelés, alors toutes les sommes partielles du sous-arbre sont nulles. Le même élagage est possible pour des sous-arbres dont toutes les feuilles sont de rendement 1. Pour un tel sous-arbre, il suffit d'appliquer la fonction de seuillage `R1` sur chaque LLR de la racine du sous-arbre en question pour obtenir le même résultat que s'il avait été parcouru dans sa totalité.

Deux autres types de sous-arbres peuvent être identifiés [3], et des fonctions spécialisées permettent d'obtenir les valeurs des sommes partielles directement depuis les valeurs des LLR de la racine de ces types de sous-arbres. Soit un sous-arbre dont le nœud racine a une taille M. Si $M-1$ feuilles correspondent à des bits gelés et la dernière à un bit d'information, le sous-arbre correspond à un nœud de répétition (`REP`). Les valeurs des sommes partielles d'un tel nœud sont toutes égales à 0 ou toutes égales à 1. Pour déterminer quelles valeurs prennent les sommes partielles, tous les LLR du nœud considéré sont additionnés. Si le total est supérieur à 0, alors toutes les sommes partielles sont mises à 0, et dans le cas contraire, toutes à 1.

Le dernier type de sous-arbre correspond à un nœud à test de parité unique (`SPC` : Single Parity Check). Plusieurs étapes sont nécessaires au traitement d'un tel nœud. Tout d'abord, la fonction élémentaire `R1` est appliquée sur chaque LLR, comme pour un nœud de rendement 1. La parité de ce premier vecteur de sommes partielles est testée. Si celle-ci est égale à 0, le test de parité est satisfait, et les sommes partielles ne sont pas modifiées. Dans le cas contraire, les LLR sont triés selon leur valeur absolue respective. La somme partielle associée au LLR ayant la valeur absolue la plus faible est inversée. Le vecteur ainsi corrigé respecte cette fois la contrainte de parité. La Figure 2.16 détaille un exemple d'élagage de l'arbre de décodage SC, avec deux nœuds `R0` et `R1` chacun de taille 2 et un nœud `SPC` de taille 4. L'intérêt de l'élagage apparaît ici clairement : le nombre de fonctions appliquées, matérialisées par des flèches, diminue. Qui plus est, le parcours de l'arbre nécessite des calculs, surtout pour des implémentations logicielles. La détermination du branchement correspond, selon le type de description logicielle, soit à un appel récursif de fonction, soit à des calculs d'adresse, puis à de indirections. De plus, diminuer la taille de l'arbre réduit la complexité calculatoire de contrôle.

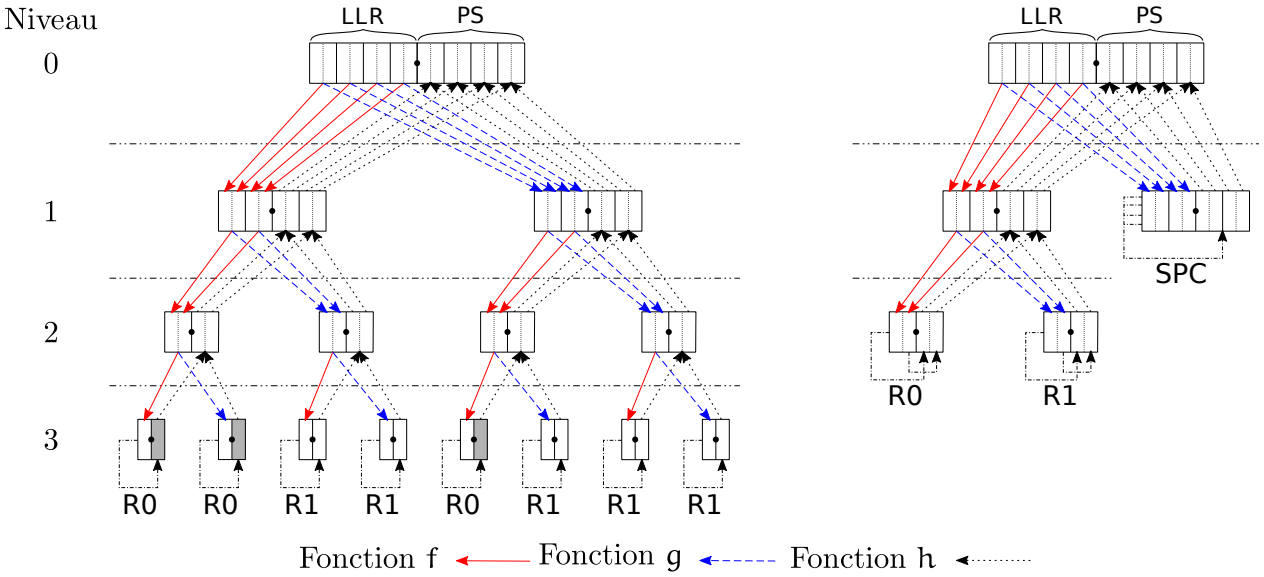


Figure 2.16 Élagage de l'arbre de décodage SC.

2.3.2 L'élagage de l'algorithme de décodage SCL

Le mécanisme d'élagage de l'arbre peut être décliné pour l'algorithme SCL, il est toutefois plus complexe à mettre en œuvre. Dans l'algorithme SCL classique, une duplication du décodeur est nécessaire à chaque fois qu'un bit d'information est estimé. Dans le cadre d'un arbre élagué, pour tous les types de sous-arbres (exceptés ceux de rendement 0), il faut générer plusieurs candidats pour le processus de duplication. Les mécanisme de traitement de chaque type de nœud sont énumérés ci-dessous. Pour chaque type de nœud, la méthode de génération des candidats est exposée, accompagnée du calcul de leurs métriques respectives.

Dans un nœud de rendement 0, comme pour le SC, les sommes partielles de sortie d'un sous-arbre sont toutes égales à 0. Il n'y a pas de nouveaux candidats à générer. Par contre, les métriques doivent être mises à jour. Le traitement pour chaque LLR en entrée du sous-arbre est le même que pour le traitement d'une feuille de rendement 0. S'il est positif, aucune pénalité n'est appliquée à la métrique du chemin courant. S'il est négatif, une pénalité égale à la valeur absolue du LLR est ajoutée à la métrique.

Pour le nœud de répétition, puisqu'un seul bit d'information est présent dans le sous-arbre, seulement deux versions des sommes partielles en sortie sont possibles, soit toutes à 1, soit toutes à 0. Ces deux versions constituent les deux candidats issus de chaque chemin. A chacun de ces candidats doit être affectée une nouvelle métrique. Pour le candidat « tout à 0 », la valeur absolue de chaque LLR négatif du nœud est ajoutée à la métrique du chemin. Pour

le candidat « tout à 1 », la valeur absolue de chaque LLR positif du nœud est ajoutée à la métrique du chemin.

Le traitement du nœud de rendement 1 est plus intensif en calcul. En effet, si le nœud est de taille T, alors le nombre de candidats possibles est 2^{T}. Pour chaque candidat, il faut calculer la nouvelle métrique. Ce calcul est le traitement `R1` appliqué à chaque LLR en entrée du sous-arbre, c'est-à-dire un seuillage. Comme pour tout nœud terminal, il faut ensuite trier les 2^{T} candidats afin d'en sélectionner les L ayant les métriques les plus faibles. Malheureusement, ces traitements deviennent trop intensifs pour une implémentation réaliste. Une diminution drastique de la complexité de ce traitement peut être obtenue par l'utilisation de l'algorithme « Chase » [**?**] sur le sous-arbre. Tout d'abord, le premier candidat est obtenu en appliquant un seuillage sur les T LLR d'entrée à la manière du traitement du nœud de rendement 1 dans l'algorithme SC. Ensuite les T LLR d'entrée sont tout d'abord triés selon leur valeur absolue respective. Les deux LLR parmi les T dont les valeurs sont les plus faibles sont identifiés, notés LLR_1 et LLR_2. Les deuxième et troisième candidats sont générés en inversant la somme partielle associée au LLR_1 et celle associée au LLR_2. Le quatrième candidat est généré en inversant les deux sommes partielles à la fois. La métrique de chaque candidat est calculée en ajoutant à la métrique du chemin considéré la valeur absolue des LLR dont la somme partielle a été inversée. Cet heuristique permet d'atteindre des performances très proches de celles d'une version non simplifiée. Il est possible que des dégradations de performance apparaissent, puisque le nombre de candidats envisagés pour un sous-arbre est réduit par rapport à la version non élaguée de l'algorithme SCL. Aucune dégradation notable n'a néanmoins été observée dans les faits expérimentaux.

Enfin, le traitement des nœuds `SPC` est similaire au traitement du nœud de rendement 1 à la différence qu'au lieu de deux bits potentiellement inversés, ce sont quatre bits qui sont à considérer. De plus, seuls les candidats respectant la contrainte de parité sont générés. A la différence des nœuds de rendement 1, la performance de correction de l'algorithme élagué peut être dégradée. En effet, dans [18], des dégradations importantes ont été observées pour des nœuds `SPC` de taille supérieure à 4.

2.3.3 L'élagage de l'algorithme de décodage SCAN

Tout comme les algorithmes SC et SCL, des élagages peuvent être appliqués à l'algorithme SCAN [53]. À la différence des deux précédents, les nœuds `REP` et `SPC` ne sont pas élagués. De plus, le concept de sous-arbre dans l'algorithme SCAN évolue. Tandis que dans l'algorithme SC, un sous-arbre prenait comme entrée des LLR et produisait en sortie des sommes partielles, dans l'algorithme SCAN, les sorties des sous-arbres sont également des LLR. Dans le cas d'un

rendement 0, les sorties du nœud sont toutes à $+\infty$, puisqu'il est certain que les bits du mots de code sont des 0. Dans le cas d'un rendement 1, les sorties du nœud sont tous à 0 puisqu'en l'absence de redondance, aucune information ne peut être donnée à partir de ce sous-arbre.

2.4 Synthèse des différents algorithmes de décodage

Dans ce chapitre ont été présentés les codes polaires, avec une emphase particulière sur les différents algorithmes de décodage. Le Tableau 2.1 récapitule ces différents algorithmes, en donnant une indication approximative sur leurs performances en termes de correction d'erreur, débit et latence maximum.

Tableau 2.1 Tendances des différents algorithmes concernant leurs pouvoirs de correction, débits et latences.

Algorithme de décodage	**Performances BER & FER**	**Débit** ($\mathcal{T}$)	**Latence max.** ($\mathcal{L}_{\mathbf{worst}}$)	**Sorties souples**
SC	faibles	très élevé	très faible	non
SCF	moyennes	élevé	faible	non
SCL	moyennes	faible	élevée	non
SCS	élevées	faible	élevée	non
CASCL	élevées	faible	élevée	non
PASCL	élevées	élevé	élevée	non
FASCL	élevées	élevé	très élevée	non
BP	faibles	élevé	faible	oui
SCAN	faibles	élevé	faible	oui

Les meilleures performances de décodage sont atteintes par les algorithmes à liste utilisant un CRC concaténés (CASCL, PASCL, FASCL) ainsi que par l'algorithme SCS. Toutefois, la complexité de ces algorithmes a pour conséquence une latence élevée de leurs implémentations. L'algorithme SC est au contraire très peu complexe mais présente des performances de décodage plus faibles. Les algorithmes BP et SCAN sont des algorithmes à sortie souple. Leurs performances brutes sont faibles, mais ils peuvent être utilisés dans des schémas de codages concaténés avec échange d'information souple. Le chapitre suivant porte sur une implémentation logicielle des algorithmes à liste.

CHAPITRE 3 DÉCODEUR LOGICIEL DE CODES POLAIRES À LISTE

Le sujet de ce chapitre est la proposition d'un décodeur logiciel qui implémente les algorithmes de décodage à liste de codes polaires. La section 3.1 traite de l'intérêt potentiel de tels décodeurs logiciels dans les réseaux mobiles cellulaires. La section 3.2 dresse un état de l'art des décodeurs logiciels de codes polaires de l'état de l'art.

Les contributions originales de ces travaux sont présentées dans les sections 3.3, 3.4 et 3.5. La section 3.3 définit les concepts de généricité et de flexibilité des décodeurs avant de montrer comment celles-ci sont mises en œuvre dans notre décodeur et constituent une rupture vis-à-vis de l'état de l'art. La section 3.4 présente trois optimisations originales qui permettent d'améliorer débit et latence de décodage. La section 3.5 présente les résultats d'expérimentation. La section 3.6 dresse une liste détaillée des différentes contributions.

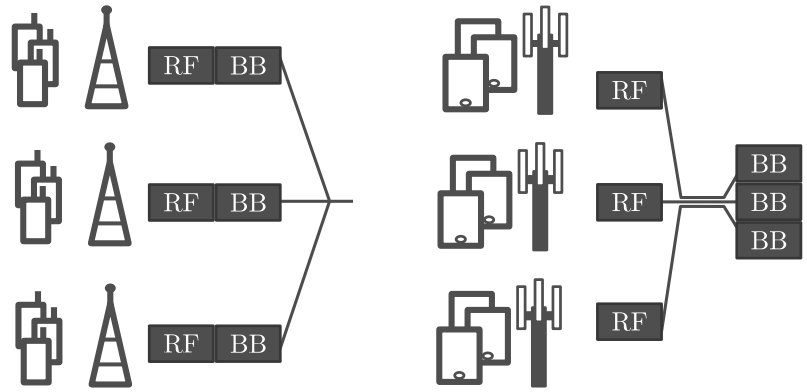
3.1 Décodeurs logiciels pour les réseaux cellulaires

Le principe des réseaux mobiles cellulaires est de diviser le territoire en zones appelées cellules. À chaque cellule sont associés une station de base et un certain nombre de canaux de fréquence pour communiquer avec les terminaux mobiles. Chaque station de base est reliée aux différents réseaux gérant les appels vocaux et les messages textuels ainsi qu'à l'Internet. Au fil de l'évolution des normes, la structure des réseaux mobiles cellulaires évolue, afin d'augmenter les performances de débit et de latence ainsi que le nombre de terminaux connectés. L'objectif est de faire face à la croissance exponentielle du nombre de terminaux.

La virtualisation des réseaux radio mobiles est considérée par les acteurs industriels [5, 6] et académiques [7–9] comme une évolution prometteuse. Cette virtualisation est proposée en particulier pour les traitements effectués classiquement dans les stations de base. Comme illustré dans la Figure 3.1a, dans les premières générations de téléphonie mobile (dites 1G et 2G), toutes les étapes de traitement du signal sont réalisées dans la station de base, à proximité de l'antenne. Une première évolution introduite dans les réseaux 3G est la séparation des traitements fréquence radio (RF : Radio Frequency) d'un coté et des traitements en bande de base (BB : Base Band) de l'autre. Comme représenté dans la Figure 3.1b, chaque unité BB associée à chaque station de base est distante de l'antenne (jusqu'à 40 kilomètres). La connexion repose sur une liaison filaire. Dans ces unités BB sont réalisés les traitements numériques, incluant les étapes d'encodage et de décodage de canal. La cellule RF a pour rôle la conversion depuis la bande de base vers la bande RF. L'avantage de cette évolution

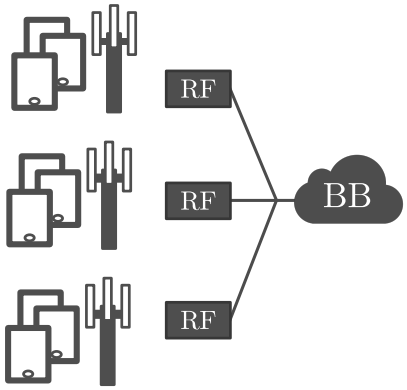
est de pouvoir placer les infrastructures pour le traitement en BB près des centres urbains afin de faciliter la maintenance et de réduire les coûts.

Dans une structure de réseau de type Cloud-RAN (Cloud Radio Access Network), les ressources matérielles de calcul en BB sont partagées entre plusieurs antennes. Des optimisations sont ainsi rendues possibles : i) une meilleure adaptation aux trafics non uniformes, ii) des économies d'énergie, iii) des augmentations de débits et des réductions de la latence et enfin iv) une évolutivité et une maintenabilité accrues [9]. Pour cela, les unités de traitement BB doivent être virtualisées. Il ne doit plus y avoir de support matériel dédié à chaque station de base. Au contraire, les calculs doivent être distribués au niveau du Cloud. Pour ce faire, il est nécessaire que tous les algorithmes exécutés pour le traitement BB soient implémentés en logiciel. Le décodage de canal est une des tâches les plus intensives en calcul [10,11]. Il est donc primordial de disposer d'implémentations logicielles de décodeurs de canal qui soient à la fois efficaces et flexibles.



(a) Stations de base traditionnelles.

(b) Séparation BB / RF.



(c) Cloud-RAN.

Figure 3.1 Évolution de l'architecture des stations de base.

Ce chapitre est divisé comme suit. Tout d'abord, la section 3.2 présente l'état de l'art des implémentations logicielles de l'algorithme SCL sur des processeurs à usage général. Dans cette même section, nous présentons les principales techniques permettant d'atteindre de hauts débits et de faibles latences. Nous décrivons ensuite les implémentations proposées qui ont la propriété d'être flexibles et génériques. Dans un souci de clarté, la section 3.3 définit précisément ces concepts de généricité et de flexibilité qui sont au cœur de l'étude. La section 3.4 détaille les optimisations d'implémentation que nous proposons. Ces dernières permettent d'atteindre des débits et des latences compétitifs tout en conservant des caractères génériques et flexibles. Enfin, la section 3.5 compare les différentes implémentations proposées avec des implémentations de l'état de l'art.

3.2 État de l'art sur les implémentations logicielles de l'algorithme SCL

Les deux principales techniques d'implémentation logicielle permettant d'atteindre de hauts débits et de réduire la latence de décodage des codes polaires sont la vectorisation et le déroulage du code source. Cette section présente ces deux techniques d'implémentation logicielle.

3.2.1 Vectorisation

Les processeurs à usage général (GPPs : General Purpose Processors) actuels sont équipés d'unités de calcul vectoriel SIMD (Single Instruction Mutiple Data). Les architectures de processeurs x86-64 définissent par exemple des jeux d'instructions SIMD nommés SSE (Streaming SIMD Extension) et AVX (Advanced Vector Extensions). Ces jeux d'instructions contiennent des opérations de chargement et de sauvegarde. Ces opérations permettent d'accéder aux données en mémoire et de les déplacer dans le fichier de registres. Ils intègrent également des opérations de calcul, comme les additions, soustractions ou multiplications.

Réduire le temps d'exécution des algorithmes de décodage de codes polaires à l'aide d'instructions SIMD est une technique classique utilisée dans de nombreuses implémentations de l'algorithme SC [3, 4, 12–17] comme de l'algorithme SCL [18–20]. Nous utilisons dans nos travaux une bibliothèque générique et portable implémentant les fonctions élémentaires des codes polaires [15]. Elle est basée elle-même sur la bibliothèque MIPP [21] qui est une encapsulation des instructions SIMD dont la description logicielle est écrite en langage C++.

L'utilisation de cette bibliothèque présente plusieurs avantages. D'une part le code source apparaît clair, lisible et compact, contrairement à ce qui serait obtenu en utilisant un langage assembleur. Un extrait du code est donné dans la Figure 3.2. D'autre part, le code source est portable puisqu'il est compatible avec différentes cibles architecturales (Intel x86, Xeon KNL

```
1 class API_polar
2 {
3   // l'usage de templates permet le support de différents formats
4   // de données : virgule fixe, virgule flottante, nombre de bits
5   template <typename R>
6   // le type "Reg" permet l'accès aux registres vectoriels
7   mipp::Reg<R> f_simd(const mipp::Reg<R> &la,
8                       const mipp::Reg<R> &lb)
9   {
10    // toutes les opérations nécessaires aux fonctions polaires
11    // sont vectorisées (abs, min, sign, neg)
12    auto abs_la  = mipp::abs(la);
13    auto abs_lb  = mipp::abs(lb);
14    auto abs_min = mipp::min(abs_la, abs_lb);
15    auto sign    = mipp::sign(la, lb);
16    auto lc      = mipp::neg(abs_min, sign);
17
18    return lc;
19  }
20
21  template <typename B, typename R>
22  mipp::Reg<R> g_simd(const mipp::Reg<R> &la,
23                      const mipp::Reg<R> &lb,
24                      const mipp::Reg<B> &sa)
25  {
26    auto neg_la = mipp::neg(la, sa);
27    // MIPP supporte également la surcharge d'opérateurs
28    auto lc     = neg_la + lb;
29
30    return lc;
31  }
```

Figure 3.2 Implémentation logicielle des fonctions f et g utilisant la bibliothèque MIPP.

et ARM) à travers l'utilisation de différents jeux d'instructions (SSE, AVX et NEON). Enfin, MIPP permet d'utiliser plusieurs formats de représentation des données, à savoir : la virgule flottante sur des mots de 32 ou 64 bits et virgule fixe sur des mots de 8, 16, 32 ou 64 bits. Plus le nombre de bits utilisés pour représenter une donnée est faible, plus le parallélisme résultant est important. Ainsi, le jeu d'instruction AVX2 permet de manipuler et de réaliser des opérations sur des registres de 256 bits. Si les données manipulées sont représentées sur 8 bits, alors le niveau de parallélisme est de 64. En clair, 64 opérations peuvent être alors réalisées en parallèle. Il est à noter que la versatilité de la bibliothèque MIPP est obtenue sans perte de performance de décodage comme démontré dans [21].

Comme évoqué dans la sous-section 2.2.1.3, dans le contexte du décodage de codes polaires, il existe deux stratégies principales de parallélisation. Le parallélisme est utilisé soit pour décoder plusieurs trames en parallèle (*inter-trame*), soit pour accélérer le décodage de chaque trame prise individuellement (*intra-trame*). Dans les travaux présentés ici, seule la stratégie *intra-trame* est utilisée. En effet cette dernière permet d'obtenir des latences plus faibles que dans le cas d'une stratégie *inter-trame*. Le principe de la stratégie *intra-trame* est d'appliquer simultanément plusieurs fonctions polaires élémentaires ($\mathtt{f}$, $\mathtt{g}$ ou $\mathtt{h}$) sur un ensemble de données contenues dans un nœud de l'arbre. Il est également possible d'utiliser des instructions SIMD pour réaliser les opérations sur les données contenues dans les feuilles. En effet, que ce soit dans les feuilles de type `R1`, `REP` ou `SPC`, il est nécessaire d'effectuer des opérations de seuillage sur un nombre de LLR correspondant à la taille du nœud. Ce seuillage est effectué en parallèle sur tous les LLR du nœud considéré grâce à des instructions SIMD.

Toutefois, le parallélisme *intra-trame* n'est exploitable que dans la partie supérieure de l'arbre. À ce niveau, les nœuds contiennent un nombre de données supérieur au niveau de parallélisme des unités SIMD. Dans le bas de l'arbre, près des feuilles, la bibliothèque MIPP [15] utilise automatiquement les versions séquentielles des implémentations des fonctions élémentaires. Dans le cas de l'algorithme CASCL sur un code polaire de taille (2048, 1723), l'augmentation du débit obtenu par l'utilisation des instructions AVX2 sur un processeur Intel i7-6600K est d'environ 20% pour les décodeurs proposés.

3.2.2 Déroulage du code source

La seconde technique classique que nous retrouvons dans les travaux récents est le déroulage du code source [4, 13, 15, 16]. Il découle de l'observation faite sur les décodeurs implémentant l'algorithme SC que les différents tests nécessaires au parcours de l'arbre de décodage prennent un temps significatif. Le parcours de l'arbre étant une donnée statique de l'algorithme de décodage, il est ainsi possible d'éviter ces tests en « déroulant » le code source avant

la compilation. La Figure 3.3 est une illustration de ce déroulage. D'un côté, dans le code non déroulé de la Figure 3.3a, des appels récursifs potentiellement coûteux de la fonction *DecodeNode* sont réalisés. De plus, des branchements conditionnels sont nécessaires pour traiter des feuilles de types différents (`R0`, `R1`, `REP` ou `SPC`). Au contraire, dans l'implémentation de la Figure 3.3b, ni appel récursif ni branchement conditionnel ne sont nécessaires. Seules les fonctions élémentaires polaires sont appelées.

Il est à noter que l'arbre de décodage que nous prenons en exemple, dans la Figure 3.3c, n'est pas élagué. Lors du décodage d'un arbre élagué, le nombre de tests effectués au cours du parcours de l'arbre augmente en proportion du nombre total d'instructions. Cela résulte du fait que le nombre de fonctions polaires différentes augmente. En effet, les fonctions `REP` et `SPC` sont ajoutées ainsi que des versions différentes des fonctions `f`, `g` et `h` prenant en compte l'existence de nœuds spécialisés dans l'arbre élagué, comme décrit dans [3, 15]. Dans certains cas, le déroulage du code source permet d'augmenter jusqu'à un facteur 2 les débits de décodage [13]. Cette technique de déroulage a été également étendue pour l'algorithme de décodage SCL [18].

Les résultats d'implémentation logicielle au niveau des débits et latences des différents décodeurs reportés dans la littérature [18–20] sont récapitulés et discutés dans la section 3.5, dans le tableau 3.2 accompagnés des résultats des décodeurs proposés.

3.3 Généricité et flexibilité d'un décodeur de codes polaires

Dans la section précédente, un état de l'art des implémentations logicielles de codes polaires a été dressé. Nous allons maintenant détailler les travaux réalisés pour parvenir au décodeur logiciel proposé. En premier lieu, l'originalité de ce décodeur est d'être *générique* et *flexible*.

3.3.1 Définitions

Les termes de généricité et de flexibilité pouvant donner lieu à diverses interprétations, nous proposons d'en donner une définition pour éviter toute ambiguïté. Nous parlerons de la *généricité* comme étant la capacité d'un décodeur à supporter n'importe quel encodage. En effet, dans le contexte de communications mobiles, les paramètres de l'encodeur changent constamment pour s'adapter au canal. Pour ce faire, des méthodes adaptatives de modulation et de codage [22] (AMC : Adaptive Modulation and Coding) sont appliquées. Ainsi, la taille du code polaire, le rendement de codage ou bien encore la position des bits gelés peuvent varier d'une trame à une autre. Par ailleurs, des patrons de poinçonnage sont souvent nécessaires. Enfin, des codes CRC sont concaténés aux codes polaires afin de détecter

Algorithme 1 : Non Déroulé.

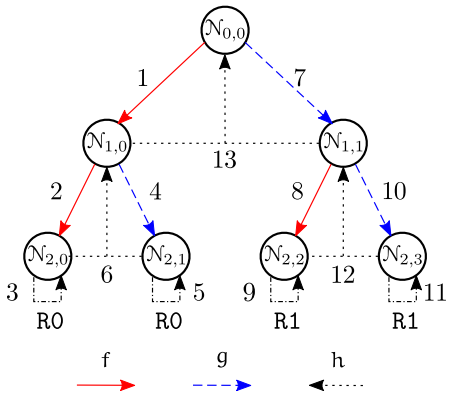
DecodeNode($\mathcal{N}_{0,0}$);
Function *DecodeNode* ($\mathcal{N}_{i,j}$)
  si $i<n$ **alors**
    f($\mathcal{N}_{i,j}$);
    DecodeNode($\mathcal{N}_{i+1,2j}$) ;
    g($\mathcal{N}_{i,j}$);
    DecodeNode($\mathcal{N}_{i+1,2j+1}$) ;
    h($\mathcal{N}_{i,j}$);
  sinon
    si $\mathcal{N}_{i,j}$ *est gelé* **alors**
      R0($\mathcal{N}_{i,j}$);
    sinon
      R1($\mathcal{N}_{i,j}$);
    fin
  fin

(a) Code non déroulé.

Algorithme 2 : Déroulé.

f($\mathcal{N}_{0,0}$);
f($\mathcal{N}_{1,0}$);
R0($\mathcal{N}_{2,0}$);
g($\mathcal{N}_{2,1}$);
R0($\mathcal{N}_{2,1}$);
h($\mathcal{N}_{1,0}$);
g($\mathcal{N}_{0,0}$);
f($\mathcal{N}_{1,1}$);
R1($\mathcal{N}_{2,2}$);
g($\mathcal{N}_{1,1}$);
R1($\mathcal{N}_{2,3}$);
h($\mathcal{N}_{1,1}$);
h($\mathcal{N}_{0,0}$);

(b) Code déroulé.



(c) Arbre décodé.

Figure 3.3 Déroulage du code source décrivant l'algorithme de décodage SC non élagué d'un code polaire (4,2) systématique.

les erreurs et permettre des méthodes de transmission à demande de répétition automatique (ARQ : Automatic Repeat reQuest) ou leur évolution hybride (HARQ : Hybrid ARQ). Un décodeur générique devrait donc supporter toutes les combinaisons possibles de chacun de ces paramètres.

D'autre part, le terme *flexibilité* s'appliquera au paramétrage de l'algorithme et des méthodes d'implémentation du décodeur, indépendamment du schéma de codage. Ces paramètres ne sont généralement pas imposés par la norme de communication. Dans le cas de l'algorithme SCL, les paramètres suivants sont concernés : les variantes de l'algorithme (FASCL, PASCL), le format de représentation des données (virgule flottante ou fixe, nombre de bits), la taille de la liste L ou encore l'élagage de l'arbre de décodage. La flexibilité du décodeur amène des degrés de liberté à l'implémentation permettant des compromis entre performances de décodage, latence et débit, pour un schéma d'encodage donné.

Comme nous allons le détailler dans les sections suivantes, tous les paramètres de généricité et de flexibilité cités sont supportés dans les implémentations de décodeurs proposées. Ils sont déterminés de manière dynamique à l'aide de fichiers de configuration ou des arguments de ligne de commande, lus lors de l'exécution. Aucune compilation du code source n'est nécessaire en cas de changement d'un paramètre. De ce choix découle le fait que nous

n'utilisons pas la technique de déroulage présentée dans sous-section 3.2.2. En effet, cette technique implique la génération d'un code source pour chaque combinaison de paramètres comme proposé dans [13] pour l'algorithme SC.

3.3.2 Généricité

Les implémentations logicielles de décodeur proposées s'appliquent à n'importe quelle taille de mot de code. De plus, les débits des décodeurs étant compétitifs, il est possible d'explorer de grandes tailles de code conjointement à des profondeurs de liste importantes, comme montré dans la Figure 3.4. Dans ce cas, N prend des valeurs allant de 2^8 à 2^{20} et le rendement du code polaire est $\mathsf{R} = 1/2$. Le CRC utilisé est défini dans le norme GZip : sa longueur est $\mathsf{c} = 32$ et son polynôme `0x04C11DB7`. Des simulations de performances de décodage de l'algorithme CASCL pour des codes polaires de grande taille ($\mathsf{N} \geqslant 2^{14}$) sont rares dans la littérature. La seule référence disponible présente les performances de décodage de l'algorithme CASCL pour un code polaire (32768,29504) avec L = 4. L'algorithme SCL a été proposé pour améliorer les performances de décodage des codes polaires pour de petites tailles ($\mathsf{N} < 2^{12}$). Néanmoins, le fort débit du décodeur logiciel proposé permet de montrer que son utilisation pour des codes polaires de grande taille apporte également un gain significatif au niveau des performances de décodage. Dans le cas où $\mathsf{N} = 2^{12}$, l'utilisation de l'algorithme CASCL avec une taille de liste L = 32 apporte un gain d'environ 1.2 dB lorsque le FER est égal à 10^{-5}. Cependant, ce gain diminue lorsque N augmente : 0.75 dB pour $\mathsf{N} = 2^{16}$ et 0.5 dB pour $\mathsf{N} = 2^{20}$. Des simulations ont également été réalisées pour une profondeur de liste L = 128. Elles montrent que le gain par rapport à L = 32 n'est pas significatif.

3.3.3 Flexibilité

Les paramètres de flexibilité sont eux aussi configurables lors de l'exécution du programme. Ils permettent divers compromis entre la latence, le débit et les performances de correction. Ainsi, l'algorithme de décodage peut être ajusté pour un code polaire donné. Dans la suite de cette section, ces paramètres de flexibilité sont détaillés et leurs effets analysés. Les mesures de débit et de latence sont réalisées sur un cœur du processeur Intel i5-6600K d'architecture Skylake. Le jeu d'instructions SIMD AVX2 est exploité. Sa fréquence pour les résultats reportés est 3.9 GHz.

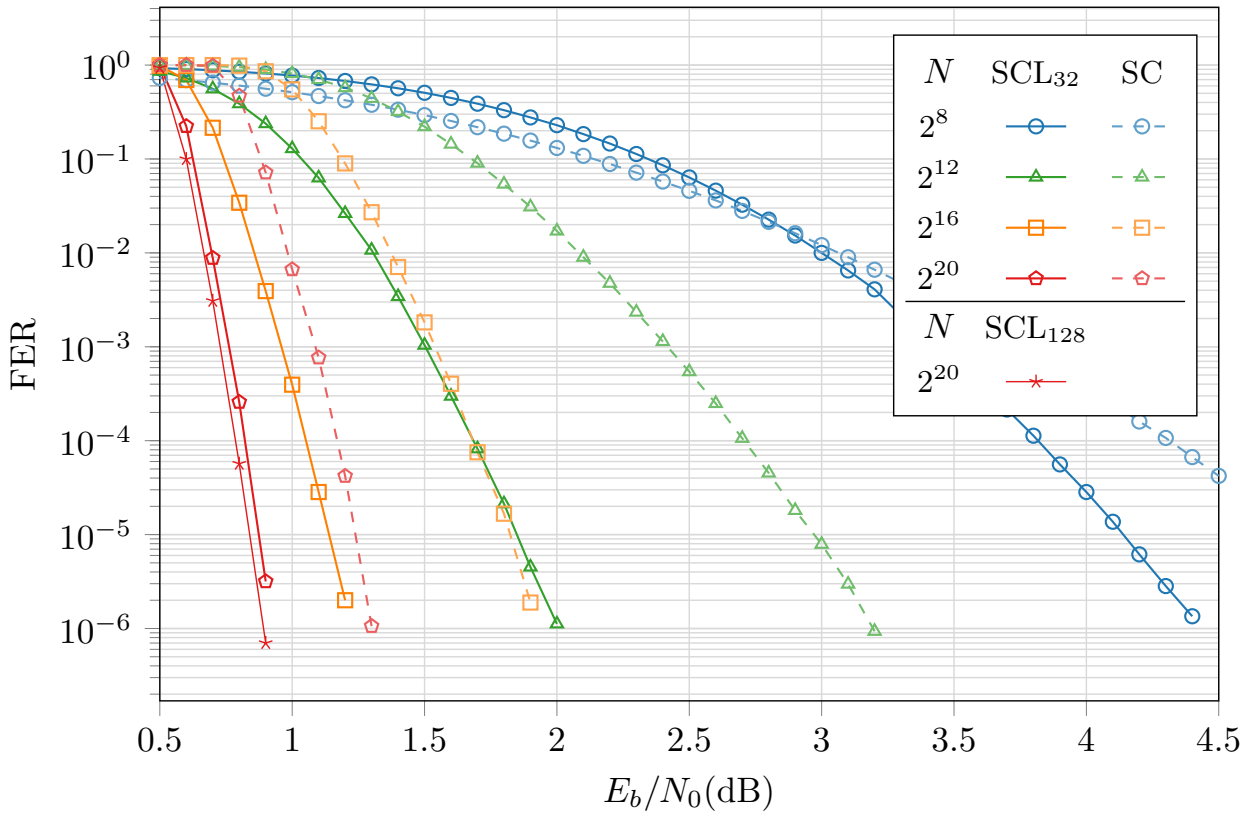


Figure 3.4 Performances de décodage des algorithmes SC et CASCL pour des très grandes tailles de code, $\mathsf{R} = 1/2$, CRC $\mathsf{c} = 32$ (GZip).

3.3.3.1 Profondeur de la liste

La profondeur de la liste impacte directement le pouvoir de correction et la complexité calculatoire de l'algorithme. La Figure 3.5 représente les performances au niveau FER et les débits obtenus pour l'algorithme de décodage CASCL d'un code polaire (2048,1024). La complexité calculatoire augmente linéairement. Le débit est divisé approximativement par deux lorsque la profondeur de liste L est doublée. Le seul cas ne respectant pas cette tendance est celui pour lequel $\mathrm{L} = 1$ qui correspond à l'algorithme SC. Ce dernier est bien plus rapide puisqu'il ne nécessite pas d'effectuer les calculs associés à l'algorithme SCL comme le tri, la génération des candidats, le calcul du CRC. Le FER évolue également avec L. À partir de $\mathrm{L} \geqslant 4$ et $\mathsf{E_b/N_0} = 2\mathsf{dB}$, les performances au niveau FER diminuent continûment, de 4.10^{-3} à 10^{-5}.

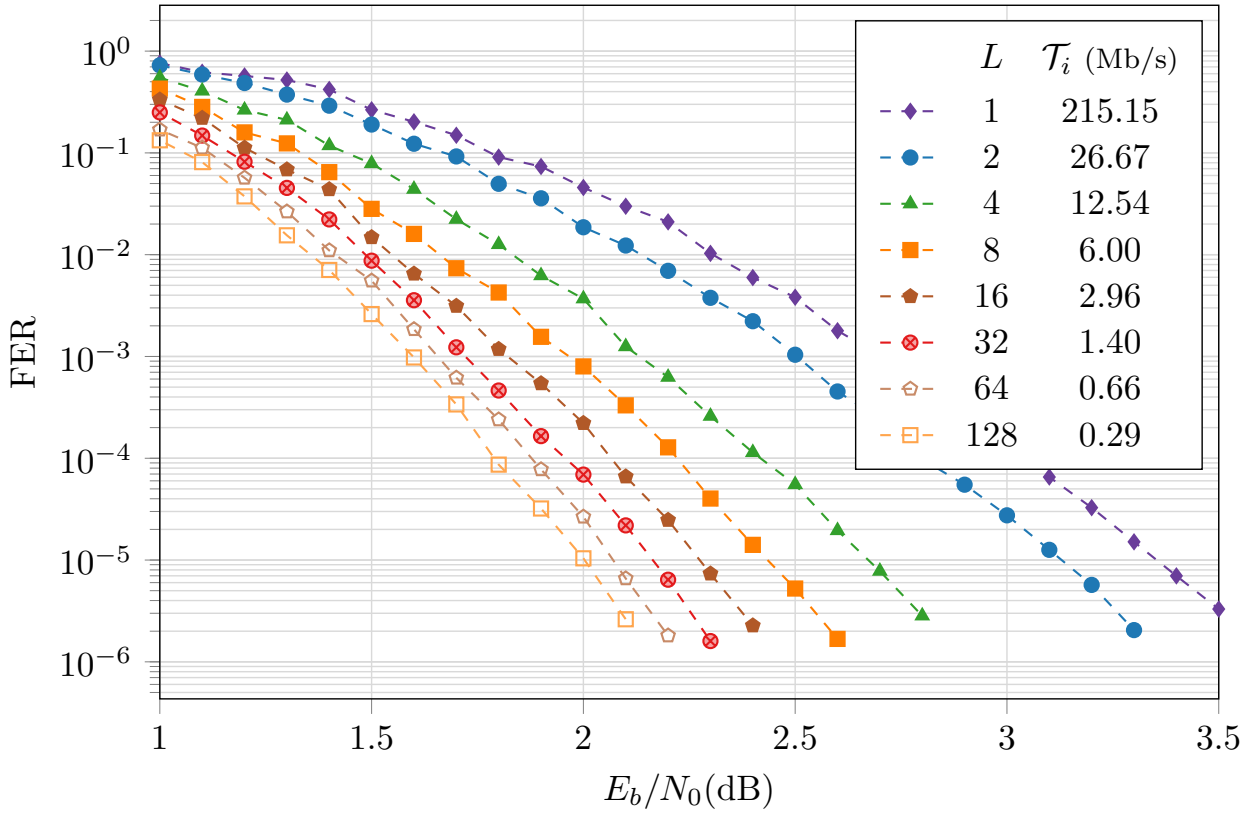


Figure 3.5 Performances de décodage et débits de l'algorithme CASCL pour différentes valeurs de L d'un code polaire (2048,1024) concaténé à un CRC $\mathsf{c} = 32$ (GZip).

3.3.3.2 Paramétrage fin de l'élagage

L'élagage tel que défini dans la sous-section 2.3.2 est paramétrable très finement dans les implémentations de décodeurs polaires proposées. Tout d'abord, chaque type de nœud (`R0`, `R1`, `REP` et `SPC`) peut être activé ou désactivé séparément. Cette caractéristique est utile pour analyser l'impact de l'utilisation de chaque type de nœud sur le débit de décodage.

À ce stade, il est important de faire la distinction entre le débit codé et le débit d'information. Soit $\mathcal{T}_\mathsf{F}$, le nombre de mots de code décodés par seconde. Dans ce cas, la valeur du débit d'information est $\mathcal{T}_\mathsf{i} = \mathsf{K}\mathcal{T}_\mathsf{F}$, soit le nombre de bits d'informations décodés par seconde, tandis que la valeur du débit codé est $\mathcal{T}_\mathsf{c} = \mathsf{N}\mathcal{T}_\mathsf{F}$.

Pour explorer l'impact de l'utilisation des types de nœuds sur le débit, il apparaît plus pertinent d'utiliser le débit codé. En effet, si le débit d'information était utilisé dans la Figure 3.6, le rendement du code « biaiserait » les débits. En effet, les débits des codes à haut rendement seraient supérieurs à ceux des codes à faible rendement.

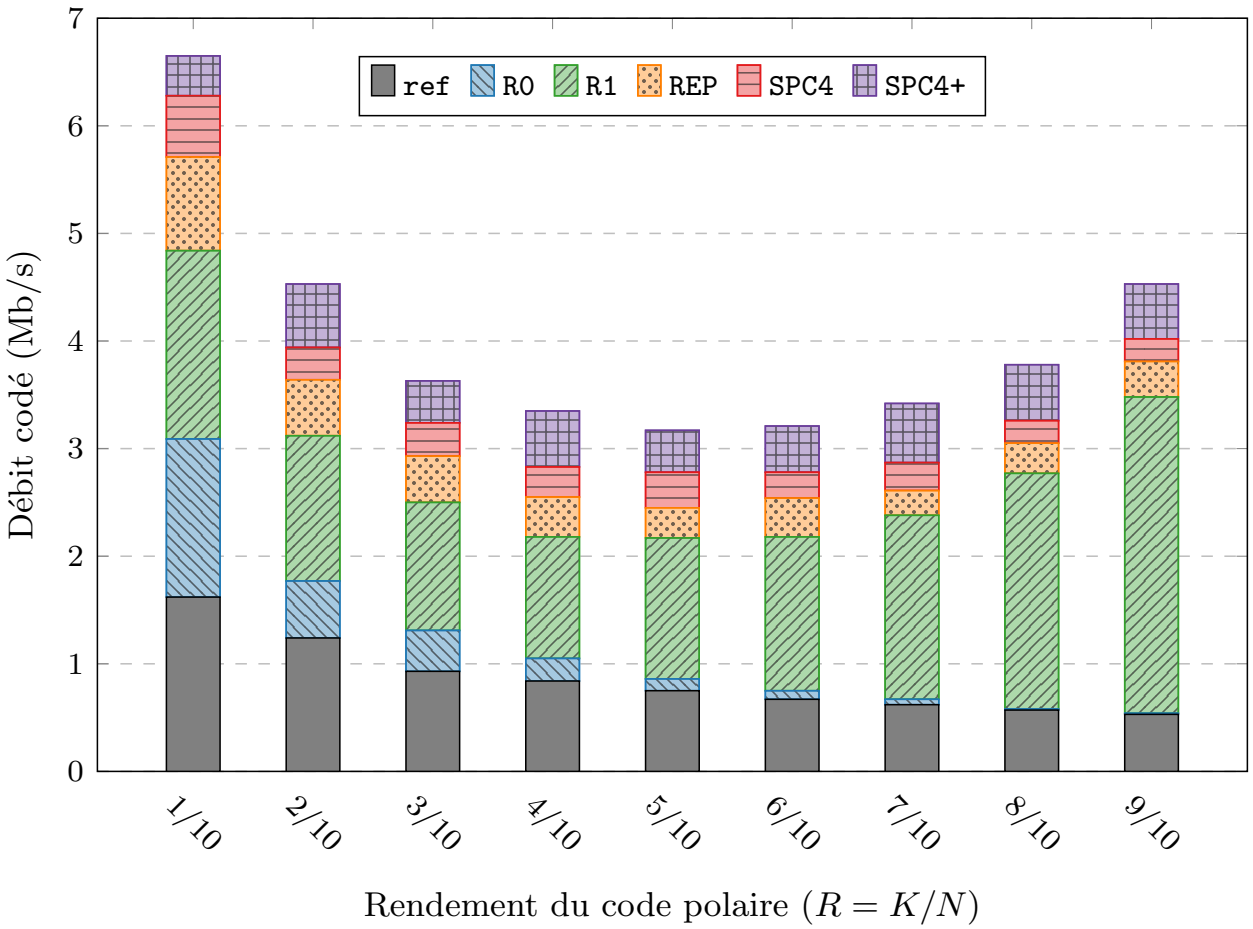


Figure 3.6 Impact de l'activation des nœuds d'élagage de l'algorithme CASCL, N = 2048, L = 32, c = 32.

Le débit codé de l'algorithme non élagué (`ref`) diminue lorsque le rendement augmente. Cela s'explique par le fait que les feuilles de rendement 1, plus nombreuses dans des codes à haut rendement, nécessitent plus de temps pour être décodées que les feuilles de rendement 0. En effet, dans l'algorithme SCL, le traitement d'une feuille de rendement 0 correspond à la mise à jour de métriques. Par ailleurs, le traitement d'une feuille de rendement 1 correspond à la génération de candidats, à la mise à jour de leurs métriques, au tri de celles-ci et à des duplications d'arbres de décodage.

Dans la Figure 3.6, les aires hachurées en diagonales (▨,▨) représentent l'amélioration des performances de décodage lorsque les nœuds `R0` et `R1` de tailles supérieures à un sont utilisés pour l'élagage. De manière attendue, l'élagage `R1` favorise une augmentation plus significative du débit pour les codes à haut rendement. Inversement, l'élagage `R0` s'avère plus efficace pour

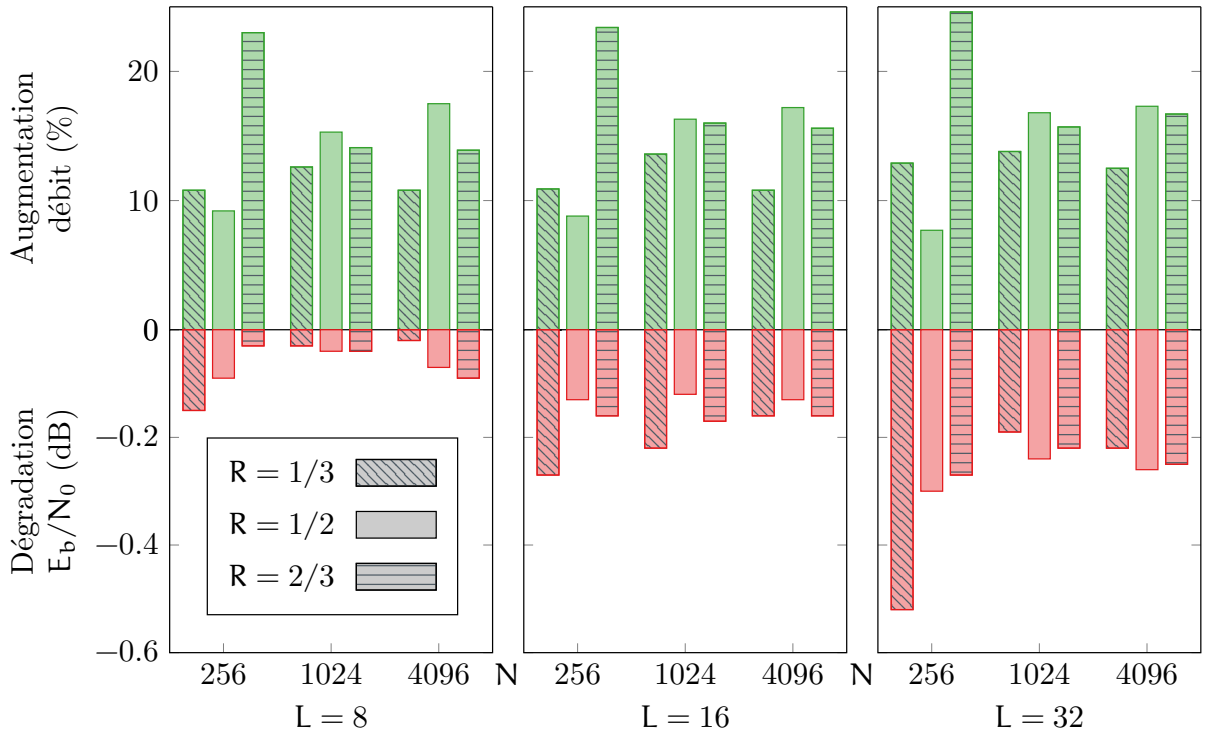


Figure 3.7 Effets de l'utilisation des nœuds `SPC4+` dans l'algorithme CASCL pour un FER de 10^{-5}.

des codes à faible rendement. Une même tendance est observée pour l'élagage des nœuds `REP`. Ces derniers sont plus efficaces pour des codes de faibles rendements. En effet, les nœuds `REP` sont plus abondants dans les codes à faibles rendements. En revanche, la tendance est moins claire pour les nœuds `SPC`.

Il est observé dans [3] que lorsque la taille des nœuds `SPC` n'est pas limitée à 4, les performances de décodage peuvent être dégradées. Dans les implémentations logicielles proposées, le choix a été fait de donner la possibilité de paramétrer finement la taille des nœuds activés pour chaque type de nœuds. En conséquence, leur taille est limitée à 4 dans les aires étiquetées `SPC4` dans la Figure 3.6. Les aires étiquetées nœuds `SPC4+` correspondent aux débits atteints lorsque la taille des nœuds `SPC` n'est plus limitée à 4.

Dans nos expérimentations, la dégradation au niveau des performances de décodage due à l'utilisation des nœuds `SPC4+` n'est pas systématique mais dépendante des caractéristiques du code polaire considéré. La Figure 3.7 illustre le compromis entre le débit du décodeur et les performances de décodage lorsque les nœuds `SPC4+` sont utilisés. La partie supérieure

représente le débit du décodeur et la partie inférieure indique la dégradation induite pour les différents codes polaires considérés. Les augmentations de débits sont comprises entre 8% et 20% selon les cas, c'est-à-dire selon la profondeur de la liste, de la taille du mot de code et du rendement. La partie inférieure de la Figure 3.7 illustre pour sa part la dégradation des performances de décodage. Pour une profondeur de liste L = 8, en dehors d'une configuration particulière ($\mathsf{N} = 256$, $\mathsf{R} = 1/3$), la dégradation du SNR pour un FER de 10^{-5} est inférieure à 0.1 dB. Si L = 32 alors la dégradation est généralement légèrement supérieure à 0.2 dB. Encore une fois, le cas du code polaire ($\mathsf{N} = 256$, $\mathsf{R} = 1/3$) est problématique avec une dégradation d'environ 0.5 dB.

En conclusion, l'élagage de l'arbre permet des augmentations significatives du débit des décodeurs logiciels de l'algorithme SCL. Toutefois le traitement des nœuds `SPC4+` décrit dans [18] peut provoquer une dégradation des performances de décodage qui dépend des paramètres de l'algorithme. Des tests complémentaires seraient nécessaires pour quantifier exhaustivement leur impact. La forte généricité et la grande flexibilité du décodeur logiciel proposé facilitent l'exploration de ce compromis entre performance de décodage et débit. Il est important de noter que l'activation des nœuds d'élagage est effectuée dynamiquement. Aussi, il serait possible de développer un décodeur logiciel qui adapterait dynamiquement son élagage, en temps réel, en fonctions des contraintes systèmes.

3.3.3.3 Représentation en virgule fixe

Il existe deux formats de représentation des nombres en base 2 : la représentation en virgule flottante et la représentation en virgule fixe. La représentation en virgule flottante offre une plus grande dynamique. Quant à la représentation en virgule fixe, elle simplifie les opérations arithmétiques. L'avantage des décodeurs de canal est que les données manipulées sont des données de type probabiliste. À ce titre, ces algorithmes sont souvent robustes vis-à-vis d'une diminution du nombre de bits utilisés pour représenter les données et par conséquent d'une représentation en virgule fixe. L'avantage de représenter les données à l'aide d'un nombre réduit de bits est que cela permet aux unités SIMD d'effectuer davantage d'opérations en parallèle. L'objectif est donc de réduire le nombre de bits afin d'augmenter le parallélisme et par conséquence les performances de débit et de latence.

Des implémentations logicielles quantifiées de l'algorithme SC ont déjà été proposées [12] dans la littérature. Cependant, les décodeurs proposés constituent, à la connaissance de l'auteur, la première implémentation logicielle de l'algorithme SCL permettant de représenter les LLR et les métriques de chemin sur 8 ou 16 bits. Pour que la représentation en virgule fixe n'introduise pas de dégradation, des précautions doivent être prises. Tout d'abord, pour

des représentations sur 8 bits, les LLR et les métriques de chemin sont saturés entre -127 et +127 après chaque opération. De plus, les métriques de chemin sont normalisées après chaque opération de duplication et de sélection de candidats. Cela signifie que la valeur la plus faible des métriques de chemin est soustraite à chacune d'entre elles. La Figure 3.8 montre les performances de l'algorithme CASCL pour différentes représentations des LLR : 8 bits à virgule fixe, 16 bits à virgule fixe et 32 bits à virgule flottante. Les courbes pour 32 bits et 16 bits sont confondues et montrent que la représentation sur 16 bits ne dégrade pas les performances. En revanche, des dégradations apparaissent pour la courbe pour 8 bits, étiquetée `REP2+`. Après analyse, il s'avère que ces dégradations sont dues au décodage des nœuds de répétition. En effet, lors de ce traitement, comme détaillé en sous-section 2.3.2, il faut sommer tous les LLR du nœud considéré. Or pour un nœud de répétition de grande taille, cette addition sur 8 bits peut induire un débordement. La Figure 3.8 montre que lorsque les nœuds de répétition de taille supérieure à 8 sont désactivés (`REP8-`), ces débordements n'ont plus lieu et les performances sont similaires à celles d'un décodage effectué au format 32 bits à virgule flottante.

Tableau 3.1 Comparaison de débits et latences des algorithmes SCL adaptatifs pour des représentations à virgule flottante (32 bits) et virgule fixe (16 et 8 bits). Code polaire (2048,1723), L = 32, CRC c = 32 (GZip).

Décodeur	**Quantif.**	$\mathcal{L}_{\mathtt{PC}}$	**3.5 dB**		**4.0 dB**		**4.5 dB**	
			$\mathcal{L}_{\mathtt{moy}}$	$\mathcal{T}_i$	$\mathcal{L}_{\mathtt{moy}}$	$\mathcal{T}_i$	$\mathcal{L}_{\mathtt{moy}}$	$\mathcal{T}_i$
PA-SSCL	32-bit	635	232.3	7.6	41.7	42.1	7.4	237.6
	16-bit	622	219.6	8.0	40.1	43.8	6.6	267.5
	8-bit	651	232.4	7.6	41.2	42.6	6.5	268.3
FA-SSCL	32-bit	1201	67.2	26.1	8.5	207.8	5.1	345.5
	16-bit	1198	68.7	25.6	7.7	225.7	4.3	408.7
	8-bit	1259	71.8	24.4	7.7	227.3	4.1	425.9

Dans le tableau 3.1 sont listés la latence « pire cas » ($\mathcal{L}_{\mathtt{PC}}$), la latence moyenne ($\mathcal{L}_{\mathtt{moy}}$) et le débit d'information moyen ($\mathcal{T}_{\mathtt{i}}$, en Mb/s) des algorithmes adaptatifs, PASCL et FASCL, pour les différentes représentations de l'information. Le processeur sur lequel les tests ont été réalisés est un processeur Intel i7-6600K. Dans le cas de la représentation sur 8 bits, les nœuds répétitions de taille supérieure à 8 sont désactivés afin d'effectuer les mesures de vitesse pour performances de décodage identiques. Les représentations à virgule fixe réduisent dans la majorité des cas la latence moyenne, surtout dans les régions à fort SNR. Ceci est dû au fait que l'accélération apportée par la représentation en virgule fixe sur une nombre réduit de bits est plus importante pour l'algorithme SC que pour l'algorithme SCL. Or, dans le cas

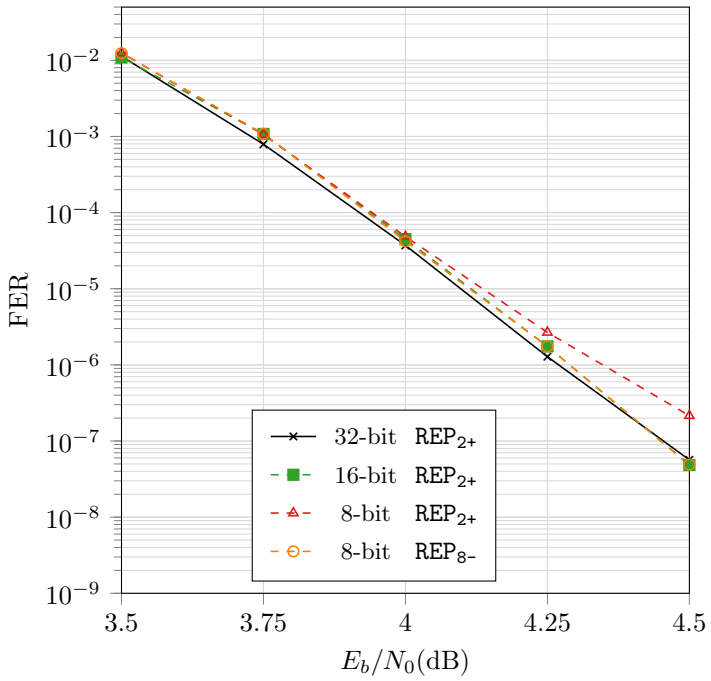


Figure 3.8 Impact de l'utilisation des nœuds de répétition sur des implémentations quantifiées.

des algorithmes adaptatifs, plus le SNR est élevé, plus il est probable que le décodage SC suffise et qu'il n'y ait pas besoin de réaliser les décodages SCL. Ainsi, un débit de 425.9 Mb/s est atteint pour une représentation sur 8 bits des LLR dans le cas de l'algorithme FASCL. Cela correspond à une augmentation de 80 Mb/s en comparaison avec la représentation sur 32 bits. Les décodeurs décrits montrent donc qu'il est possible d'implémenter les différents algorithmes à liste en représentant les données sur 8 bits sans dégrader les performances. Cette exploitation de la représentation en virgule fixe permet d'accélérer significativement le décodage des implémentations logicielles des algorithmes à liste.

3.3.3.4 Les différentes variantes algorithmiques implémentées

La Figure 3.9 montre les deux versions adaptatives de l'algorithme SCL, à savoir les algorithmes partiellement et complètement adaptatifs (PASCL et FASCL). Des débits différents sont observés suivant le rapport signal à bruit $\mathsf{E_b/N_0}$ considéré. Pour de faibles valeurs de SNR, l'algorithme PASCL est plus avantageux. En revanche, l'algorithme FASCL prend l'avantage pour des valeurs intermédiaires, avant que les débits de chaque version se rapprochent pour les valeurs de $\mathsf{E_b/N_0}$ les plus élevées. L'explication de cette observation est la suivante : à faible SNR, la probabilité d'échec de décodage est très forte. La plupart du temps, l'algorithme adaptatif doit déclencher un décodage avec $\mathrm{L} = \mathrm{L}_{max}$ (ici $\mathrm{L}_{max} = 32$).

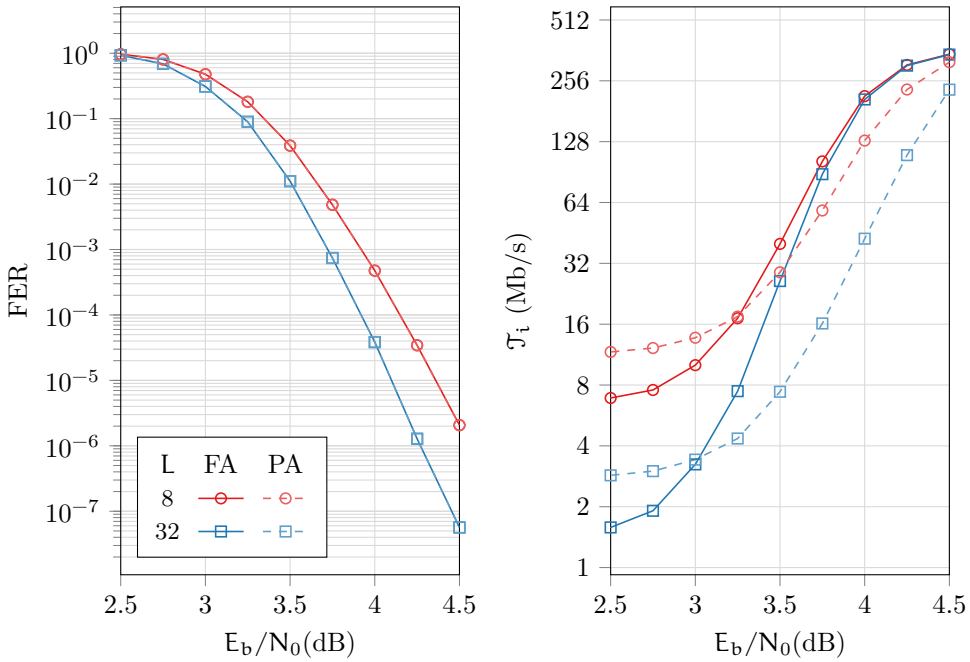


Figure 3.9 Performances de décodage et débits des algorithmes de décodage PASCL et FASCL pour un code polaire (2048,1723), L = 32, et CRC $\mathbf{c}$ = 32 GZip.

Pour l'algorithme PASCL, dans ce cas précis où le décodage avec L = 32 est nécessaire, deux décodages successifs seront déclenchés : le décodage SC puis le décodage SCL avec L = 32. Pour l'algorithme FASCL, des exécutions intermédiaires de l'algorithme seront effectuées, pour L = {2,4,8,16}. En résumé, dans le cas le plus fréquent à faible SNR, l'algorithme PASCL est plus rapide que l'algorithme FASCL. À fort SNR, le cas le plus probable est une réussite du décodage. Lorsque l'algorithme SC est suffisant pour décoder la trame, les algorithmes FASCL et PASCL auront des performances similaires. C'est pourquoi les débits des implémentations des deux algorithmes sont proches pour les valeurs les plus fortes de E_b/N_0. Pour des valeurs intermédiaires de SNR, le FASCL présente des débits supérieurs à ceux de l'algorithme PASCL. En effet, l'exécution de l'algorithme SCL avec des valeurs intermédiaires de L est alors pertinent. Cependant, dans tous les cas et comme décrit dans la sous-section 2.2.2.3, la latence « pire cas », autrement dit le temps maximum nécessaire pour le décodage d'une trame, est plus élevée pour l'algorithme FASCL. Ce fait est vérifié par la mesure comme reporté dans le Tableau 3.1.

La section 3.3 a permis de définir les concepts de généricité et de flexibilité du décodeur. Des contributions propres aux décodeurs proposés et liées à leur flexibilité ont également été introduites :

— la représentation sur 8 et 16 bits en virgule fixe des données pour les algorithmes à liste,

— la configuration dynamique de l'élagage de l'arbre de décodage,

— le support de l'algorithme FASCL.

Cependant, ces implémentations doivent être compétitives par comparaison avec les décodeurs de la littérature du point de vue du débit et de la latence. Pour ce faire, des améliorations permettant d'atteindre de hauts débits et de faibles latences sont présentées dans la section suivante.

3.4 Optimisations de l'implémentation logicielle des décodeurs à liste

Cette section 3.4 présente trois contributions originales. La première porte sur l'utilisation nouvelle d'un algorithme de tri des métriques et des LLR. La seconde concerne l'accélération du contrôle de redondance cyclique. Enfin, la gestion des sommes partielles est l'objet d'une contribution propre.

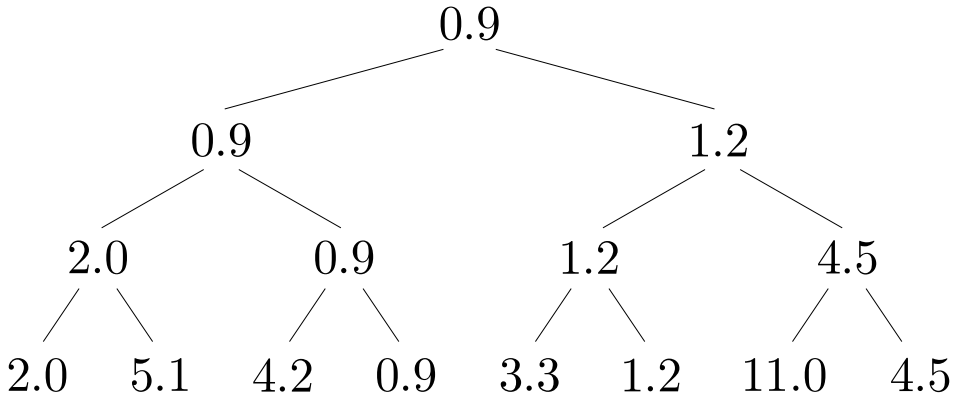
3.4.1 Algorithme de tri

Des algorithmes de tri partiel sont appliqués durant plusieurs étapes lors de l'exécution de l'algorithme SCL. Ainsi, les L métriques de chemin les plus faibles doivent être identifiées pour sélectionner les candidats lors des étapes de duplication (étapes (i+1) et (i+3) de la Figure 2.7). De plus, les 2 LLR ayant les valeurs absolues les plus faibles doivent être identifiés lors du traitement des nœuds `R1`. Enfin, les 4 LLR les plus faibles doivent être retenus lors du traitement d'un nœud `SPC`.

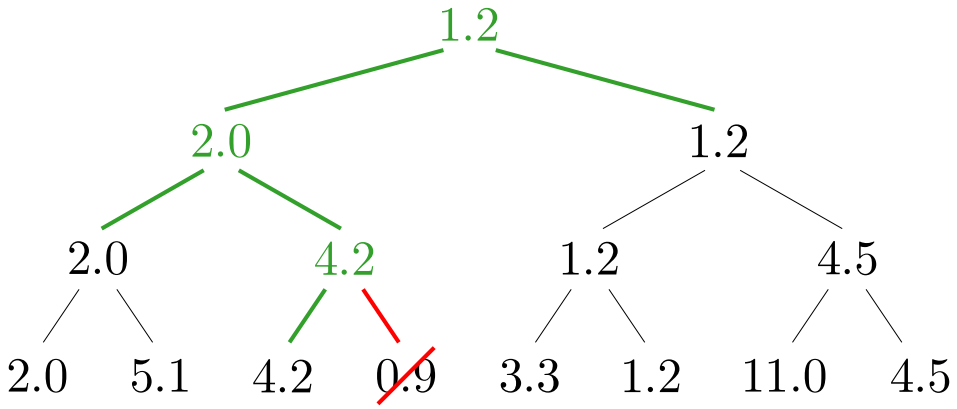
Dans [18], la méthode de tri des LLR dans les nœuds `REP` et `SPC` n'est pas précisée. Schreier [23] donne une méthode pour identifier les deux plus petits (ou les deux plus grands) éléments d'un ensemble. Cette technique est également détaillée dans [24]. Il a été prouvé que cette méthode est la méthode optimale en termes de nombres de comparaisons deux-à-deux à effectuer. Ce nombre est égal à $\mathsf{N} + \log_2(\mathsf{N}) - 2$

Elle nécessite deux étapes complémentaires, présentées en Figure 3.10. La première étape est l'identification du plus petit élément par la traversée d'un arbre binaire. Durant la seconde étape, cet élément est éliminé, et la branche sur laquelle il se trouvait est rejouée. Le second élément le plus petit est ainsi identifié.

Cette méthode a été testée expérimentalement et ses performances ont été comparées à d'autres algorithmes potentiels : *Batcher's merge exchange*, tri par bulle, tri *quick-sort*, tri



(a) Identification du plus petit élément.



(b) Identification du second plus petit élément.

Figure 3.10 Méthode de Schreier.

par tas implémenté dans la bibliothèque standard. Les mesures ont été réalisées sur un processeur Intel i7-4790HQ, pour différents rendements, tailles de codes et profondeur de liste. La méthode de Schreier s'est révélée dans tous les cas le plus rapide. Elle l'était également dans les cas testés pour les nœuds de type `SPC`. En effet, l'utilisation d'autres algorithmes de tri n'apporte pas de gain significatif.

Le tri des métriques dans [18] est effectué à l'aide d'un réseau de tri partiels (PSN : Partial Sorting Network) décrit dans [25]. Les unités de calcul SIMD sont utilisées pour accélérer l'exécution de ce tri. Dans un souci de simplicité de la description logicielle, le choix a donc été fait de n'utiliser que la méthode de Schreier pour réaliser les différents tris nécessaires pour l'exécution de l'algorithme SCL. Dans cette étude, nous n'avons pas implémenté les PSN pour le tri des métriques. En effet, pour que ce réseau de tri soit générique vis à vis de L ainsi que de l'élagage de l'arbre, une méthode devrait être conçue afin de créer un PSN pour n'importe quelle profondeur de liste L, et pour chaque type de nœud différent. Nous pensons que l'utilisation de PSNs pourrait augmenter le débit de nos décodeurs. Par défaut, la

méthode de Schreier est également utilisée pour le tri des métriques. Les mêmes algorithmes alternatifs ont été testés mais aucune amélioration significative n'a permis d'amélioration du débit.

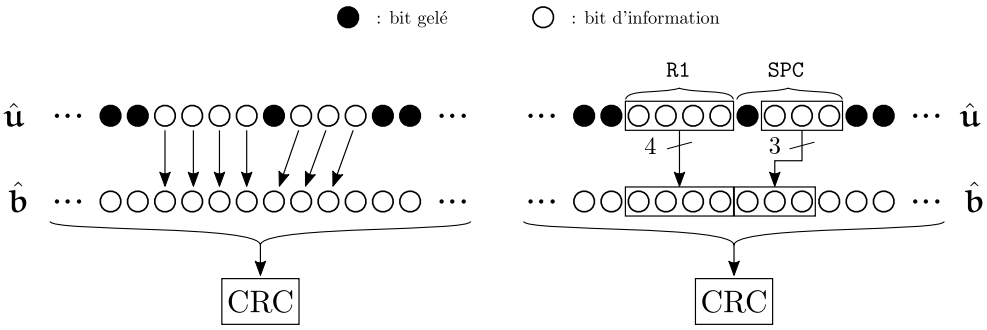
3.4.2 Accélération du contrôle de redondance cyclique

Par profilage de l'exécution d'un décodeur SCL adaptatif, il est possible d'observer que beaucoup de cycles d'exécution sont alloués aux vérifications de CRC. En effet, durant chaque décodage (y compris le premier décodage SC), il est nécessaire de réaliser une vérification du CRC. De plus, la complexité calculatoire d'une vérification de CRC est en $\mathsf{O}(\mathsf{N})$ tandis que celle du reste de l'algorithme SCL est en $\mathsf{O}(\mathsf{N} \log \mathsf{N})$. En conséquence, pour les valeurs de N considérées ($\mathsf{N} < 8192$), le temps nécessaire à la vérification du CRC n'est pas négligeable vis à vis de celui nécessaire pour le décodage de code polaire proprement dit.

C'est pourquoi des méthodes d'implémentation efficaces doivent être utilisées afin de réduire le temps de vérification du CRC. Pour ce faire, nous proposons que les bits soient empaquetés pour être traités 32 par 32. À l'initialisation, les sommes partielles sont chacune stockées sur un entier. Pour vérifier le CRC, 32 sommes partielles sont lues puis stockées dans un seul entier de 32 bits. Une table de conversion est utilisée pour stocker des séquences calculées à l'avance du CRC. La lecture de ces séquences permet de réduire la complexité calculatoire globale.

Après chaque décodage SC ou SCL au sein des décodeurs adaptatifs, des mots de code candidats $\hat{\mathbf{u}}$ de N bits sont produits. Toutefois, le CRC est calculé sur les K bits d'information de $\hat{\mathbf{b}}$ parmi les N bits du mot de code. Il est donc nécessaire de réaliser l'extraction des bits d'information. L'implémentation naïve représentée dans la Figure 3.11a consiste à réaliser cette extraction bit par bit. Pour chaque bit, un test est réalisé pour savoir s'il s'agit d'un bit gelé ou d'un bit d'information. Le profilage des décodeurs montre que cette opération d'extraction représente une portion non négligeable du temps total de traitement du CRC. Or, il est possible de l'accélérer en utilisant l'emplacement des bits gelés et la connaissance des nœuds d'élagage. En effet, la présence d'un nœud de type `R1` de taille 4 correspond à la présence de 4 bits d'information consécutifs dans le vecteur $\hat{\mathbf{u}}$ comme représenté en Figure 3.11b. La fonction (`std::copy`) de la bibliothèque standard C++ permet de réaliser une extraction parallèle de plusieurs bits. Cette technique peut être étendue aux nœuds `SPC` à condition d'exclure le bit gelé du nœud en question.

Au cours de l'exécution des algorithmes adaptatifs, une vérification du CRC doit être effectuée une fois à la fin du décodage SC, et L fois à la fin de chaque décodage SCL, afin de tester chaque candidat. En conséquence, l'accélération des vérifications de CRC est capital pour



(a) Extraction séquentielle. (b) Extraction parallèle.

Figure 3.11 Extraction des bits d'information avant vérification du CRC.

ces algorithmes adaptatifs. Les mesures effectuées le vérifient. Pour l'algorithme FASCL sur un code polaire (2048,1723) associé un CRC de taille $\mathsf{c} = 32$ à $\mathsf{E_b}/\mathsf{N}_0 = 4.5\mathsf{dB}$, le gain de performance d'exécution lié à l'utilisation des techniques présentées ci-dessus est de 63%, portant le débit de 262 Mb/s à 426 Mb/s.

3.4.3 Gestion des sommes partielles

La gestion des sommes partielles dans les décodeurs proposés est détaillée dans cette sous-section. Les sommes partielles sont des décisions dures. Cela signifie que leurs valeurs donc binaires. Cependant, dans les implémentations logicielles proposées, afin d'utiliser efficacement les instructions SIMD, les LLR et les sommes partielles sont stockés sur des entiers de même dimension, notée Q, en octets. Selon la représentation choisie, 32, 16 ou 8 bits, Q peut donc prendre respectivement les valeurs 4, 2 ou 1. Dans les précédentes implémentations logicielles de l'algorithme SCL [18–20], un emplacement mémoire de Q octets est alloué à chaque niveau de l'arbre de décodage comme décrit dans [26]. La taille de l'emplacement mémoire associé à un niveau de l'arbre est égal à la taille des nœuds du niveau en question. Soit $\mathfrak{n} = \log_2(\mathsf{N})$, le nombre total de niveaux est $\mathfrak{n}+1$ et la taille d'un nœud au niveau $\mathfrak{l}$ est égale à $2^{\mathfrak{n}-\mathfrak{l}}$. L'empreinte mémoire pour les sommes partielles d'un arbre de décodage est, quant à elle, égale à :

$$\sum_{\mathfrak{l}=0}^{\mathfrak{n}} 2^{\mathfrak{n}-\mathfrak{l}} = 2\mathsf{N} - 1 \tag{3.1}$$

Pour l'algorithme SCL dans lequel L arbres de décodage sont nécessaires, l'empreinte mémoire totale est donc $\mathsf{L}(2\mathsf{N}-1)$. Cependant, dans l'algorithme SC, les sommes partielles d'un nœud donné de l'arbre de décodage ne sont utilisées qu'une seule fois au cours du décodage. Par conséquent, ces emplacements mémoire peuvent être réutilisés. Ainsi, l'empreinte mémoire

est réduite de $2\mathsf{N}-1$ à N dans l'algorithme SC [27]. De même, il est possible de réduire cette empreinte mémoire de $\mathrm{L}(2\mathsf{N}-1)$ à $\mathrm{L}\mathsf{N}$ pour l'algorithme SCL.

Lors des étapes de duplication et de sélection de l'algorithme SCL, étapes $(\mathtt{i}+1)$ et $(\mathtt{i}+3)$ de la Figure 2.7, il est nécessaire d'affecter les sommes partielles d'un arbre de décodage original à un nouvel arbre. Deux méthodes sont envisageables pour réaliser cette affectation. La première, notée $\mathsf{SCL}_{\mathtt{cpy}}$, consiste à systématiquement copier les N sommes partielles sauvegardées dans l'emplacement mémoire du premier arbre de décodage vers l'emplacement mémoire du second. La seconde, présentée dans [26], est de réaliser cette affectation à l'aide de pointeurs. Ainsi, tant que les sommes partielles de chaque arbre ne diffèrent pas, elles utilisent le même emplacement mémoire. Cette méthode est notée $\mathsf{SCL}_{\mathtt{ptr}}$.

La méthode $\mathsf{SCL}_{\mathtt{ptr}}$ semble plus efficace puisqu'elle évite des copies inutiles. Cependant, nos expérimentations ont montré que les calculs qui sont nécessaires à la gestion des pointeurs pénalisent cette méthode pour des codes de petite et moyenne tailles. La Figure 3.12 représente les débits des implémentations de l'algorithme SCL associés aux deux méthodes, pour différentes valeurs de N, un rendement constant, $\mathsf{R}=1/2$ et une profondeur de liste $\mathrm{L}=32$. Ces courbes montrent que la méthode $\mathsf{SCL}_{\mathtt{cpy}}$ permet d'atteindre des débits plus élevés pour $\mathsf{N}<8192$. En revanche, la méthode $\mathsf{SCL}_{\mathtt{ptr}}$ est plus avantageuse pour des tailles de codes supérieures. La Figure 3.12 montre également l'impact du format utilisé pour la représentation des LLR et des sommes partielles. Pour de très grandes valeurs de N, la représentation sur 8 bits est plus efficace, puisqu'elle occupe moins d'espace mémoire et que les échecs d'accès à la mémoire cache sont plus rares.

Trois optimisations originales permettant d'augmenter le débit et de réduire la latence des implémentations des algorithmes de décodage de codes polaire à liste sont détaillées dans la section 3.4 : nouvel algorithme de tri, accélération du traitement du CRC, et nouvelle méthode de gestion des sommes partielles.

3.5 Expérimentations : mesures et comparaisons

Les débits et les latences des implémentations proposées sont détaillées dans cette section. Tous les résultats proposés ont été obtenus grâce à l'outil AFF3CT [28]. Le code source de ce logiciel est ouvert et disponible en ligne. Ainsi, tous les résultats sont facilement reproductibles. La cible matérielle est un CPU Intel i5-6600K d'architecture Skylake. Le jeu d'instructions SIMD AVX2 est exploité. Sa fréquence pour les résultats reportés est 3.9 GHz. La compilation a été effectuée sur un OS Linux avec le compilateur C++ de GNU (version 5.4.0). Les options de compilation qui ont été appliquées sont les suivantes : `-Ofast`

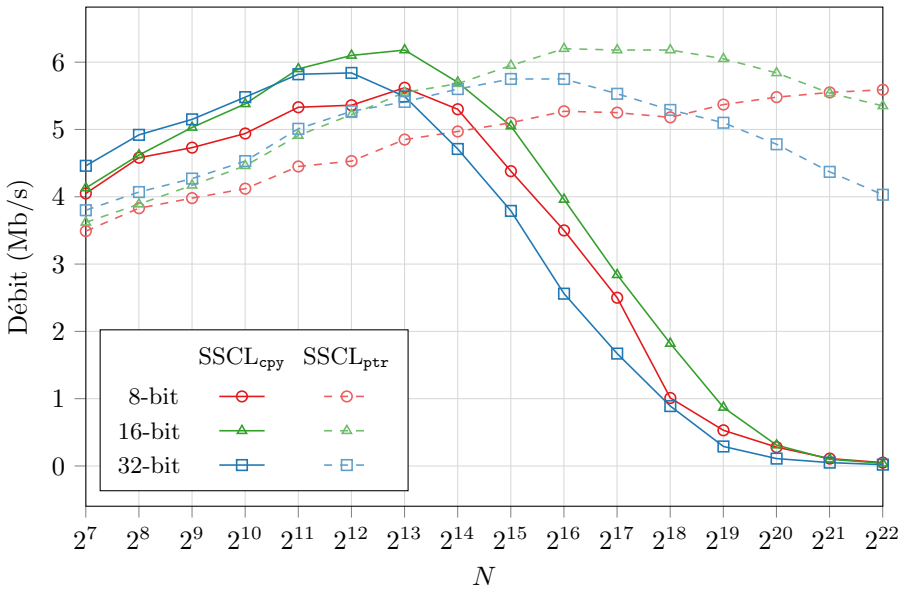


Figure 3.12 Comparaison des débits associés aux méthodes $\mathsf{SCL}_{\mathtt{cpy}}$ et $\mathsf{SCL}_{\mathtt{ptr}}$ pour différentes valeurs de N.

`-march=native -funroll-loops`.

3.5.1 Algorithme complètement adaptatif

L'algorithme FASCL permet d'atteindre les débits les plus élevés. Comme il est montré dans le Tableau 3.1, l'implémentation de cet algorithme avec une représentation sur 8 bits des LLR et des sommes partielles permet d'atteindre un débit de 425 Mb/s pour un code polaire (2048,1723) et le CRC GZip de taille 32 bits lorsque $\mathsf{E_b/N_0} = 4.5\mathsf{dB}$. Ce débit est pratiquement deux fois supérieur à celui obtenu avec l'algorithme PASCL. Le rapport signal à bruit pour lequel la différence entre PASCL et FASCL est la plus grande (380%), est dans le domaine où le FER est compris entre 10^{-3} et 10^{-5}. Il s'agit du domaine ciblé pour les communications sans fils comme le norme LTE ou le futur réseau 5G. Dans ces conditions, le débit de l'algorithme FASCL est d'approximativement 230 Mb/s tandis que celui de PASCL est d'approximativement 40 Mb/s. Toutefois, rappelons que la latence dans le pire cas est plus élevée dans le cas de l'algorithme FASCL.

3.5.2 Comparaison avec l'état de l'art

Les débits et les latences des décodeurs proposés sont listés et comparés avec ceux de certains décodeurs de l'état de l'art dans le tableau 3.2. Pour tous les décodeurs, les nœuds SPC4+ ne sont pas utilisés afin de ne pas impacter les performances de décodage. La latence donnée dans le Tableau 3.2 est la latence « pire cas » et le débit est le débit d'information moyen. La

Tableau 3.2 Comparaison des débits et des latences des décodeurs proposés avec ceux de l'état de l'art. Représentation des LLR et sommes partielles sur 32 bits. Code polaire (2048,1723), L = 32, CRC GZip $\mathsf{c} = 32$.

Cible	**Algorithme**	**Version**	$\mathcal{L}_{\mathbf{PC}}$ (µs)	$\mathcal{T}_i$ (Mb/s)		
				3.5 dB	**4.0 dB**	**4.5 dB**
i7-4790K	CASCL[1]	[20]	1572	1.10	1.10	1.10
i7-2600	CASCL[1]	[19]	23000	0.07	0.07	0.07
i7-2600	CASCL[1]	[18]	2294	0.76	0.76	0.76
i7-2600	CASCL[1]	Ce travail	4819	0.37	0.37	0.37
i5-6600K	CASCL[1]	Ce travail	3635	0.48	0.48	0.48
i7-2600	CASCL	[19]	3300	0.52	0.52	0.52
i7-2600	CASCL	[18]	433	4.0	4.0	4.0
i7-2600	CASCL	Ce travail	770	2.3	2.3	2.3
i5-6600K	CASCL	Ce travail	577	3.0	3.0	3.0
i7-2600	PASCL	[19]	≈ 3300	0.9	4.90	54.0
i7-2600	PASCL	[18]	≈ 433	8.6	33.0	196.0
i7-2600	PASCL	Ce travail	847	5.5	31.1	168.4
i5-6600K	PASCL	Ce travail	635	7.6	42.1	237.6
i7-2600	FASCL	Ce travail	1602	19.4	149.0	244.3
i5-6600K	FASCL	Ce travail	1201	26.1	207.8	345.5

[1] Version non élaguée de l'algorithme CASCL.

première version, CASCL, est l'implémentation de l'algorithme CASCL sans aucun élagage tandis que les versions notées CASSCL, FASSCL et PASSCL utilisent l'élagage de l'arbre. Le débit du décodeur CASSCL proposé (2.3 Mb/s) est seulement divisé par deux par comparaison avec l'implémentation spécifique déroulée de l'algorithme CASSCL décrite en [18] (4.0 Mb/s). Elle est approximativement 4 fois plus rapide que l'implémentation générique proposée en [19] (0.52 Mb/s) et 2 fois plus rapide que celle proposée en [20]. Ce résultat est obtenu grâce aux améliorations algorithmiques proposées dans la section 3.4. De plus, les implémentations proposées présentent une flexibilité et une généricité bien plus grandes que celles proposées dans [19, 20]. En effet, les représentations quantifiées, la possibilité de définir des patrons de poinçonnage et la possibilité d'utiliser les algorithmes adaptatifs n'étaient pas offertes. En utilisant la même cible matérielle (i7-2600), l'implémentation de l'algorithme PASSCL présente des débits proches de l'implémentation spécifique déroulée de [18]. Enfin, le débit obtenu pour l'algorithme FASSCL est lui bien meilleur (jusqu'à 244 Mb/s pour $\mathsf{E_b}/\mathsf{N}_0 = 4.5\mathrm{dB}$).

3.5.2.1 Performances sur différentes cibles matérielles

Tableau 3.3 Comparaison des débits, du nombre de cycles d'horloge nécessaires pour décoder une trame, et de l'énergie dépensée par bit décodé, pour des décodeurs proposés sur différentes cibles matérielles. Représentation en virgule fixe sur 8 bits , $\mathsf{E_b/N_0} = 4.5\mathsf{dB}$, CRC GZip $\mathsf{c} = 32$.

Target	**Freq. (MHz)**	**Algo.**	$\mathcal{T}_i$ **(Mb/s)**			**# Cycles par trame**			$\mathcal{E}_b$ **(nJ/bit)**		
			3.5 dB	**4.0 dB**	**4.5 dB**	**3.5 dB**	**4.0 dB**	**4.5 dB**	**3.5 dB**	**4.0 dB**	**4.5 dB**
i7 4790	3592	CASCL	2.34	2.38	2.24	2645	2600	2763	6325	6218	6607
		PASCL	5.80	33.88	207.96	1067	183	30	2552	437	71
		FASCL	19.43	194.66	372.30	319	32	17	762	76	40
Ryzen 2700X	4230	CASCL	2.68	2.72	2.56	2675	2635	2800	5149	5074	5391
		PASCL	6.55	38.76	237.78	1094	185	30	2107	356	58
		FASCL	20.93	209.37	406.15	342	34	18	659	66	34
Cortex A73	2360	CASCL	1.23	1.28	1.20	3310	3175	3382	1475	1415	1507
		PASCL	3.08	18.90	100.75	1319	215	40	588	96	18
		FASCL	10.64	83.58	137.37	382	49	30	170	22	13
Cortex A15	2000	CASCL	0.74	0.74	0.70	4639	4639	4923	3365	3365	3571
		PASCL	1.81	11.06	61.86	1899	312	56	1378	226	40
		FASCL	6.23	52.77	88.31	553	65	39	401	47	28
Cortex A53	1840	CASCL	0.48	0.48	0.45	6560	6560	7064	1353	1353	1457
		PASCL	1.17	7.32	42.77	2701	433	74	557	89	15
		FASCL	4.45	39.70	68.32	712	80	46	147	16	10
Cortex A7	1400	CASCL	0.28	0.29	0.27	8615	8318	8934	1071	1034	1111
		PASCL	0.71	4.26	21.89	3397	566	110	423	70	14
		FASCL	2.41	19.40	30.93	1001	124	78	124	15	10

La description logicielle des décodeurs proposés est portable, grâce en particulier à l'utilisation du langage C++11 et de la bibliothèque MIPP [21] pour ce qui concerne les instructions SIMD. Ainsi, en plus d'être efficace comme nous l'avons déjà vu sur des architectures de processeur de type x86, le décodeur bénéficie également d'optimisations pour l'exécution sur des processeurs du marché de l'électronique embarquée, à savoir des architectures ARM. Des résultats d'exécution sur plusieurs cibles sont récapitulés dans le Tableau 3.3. Les métriques prises en compte sont le débit d'information ($\boldsymbol{\mathcal{T}_i}$), le nombre de cycles d'horloge nécessaire pour décoder une trame et l'énergie dépensée par bit d'information décodé ($\boldsymbol{\mathcal{E}_b}$). Le Tableau 3.3 présente des résultats d'exécution sur un seul cœur de processeur, sans utiliser le mode multifil simultané (SMT : Simultaneous MultiThreading).

Les deux premières cibles matérielles considérées sont deux processeurs à usage général pour station de travail, un Intel i7-4790 et un AMD Ryzen 2700X. Leurs architectures respectives sont de type x86-64 et tous deux possèdent les instructions SIMD AVX2. Les fréquences affichées sont celles mesurées au moment de l'exécution du programme, tout comme la puis-

sance nécessaire pour le calcul de ($\mathcal{E}_b$). Le logiciel utilisé pour ces mesures est HWiNFO [29]. La puissance mesurée est celle du cœur de processeur seulement, excluant la consommation d'énergie de la mémoire principale et de la mémoire cache de niveau 3.

Par ailleurs, les résultats d'exécution des implémentations logicielles sur quatre versions différentes de processeurs ARM sont présentés. Les architectures ARM sont avantageuses du point de vue de la consommation énergétique pour l'algorithme de décodage SC [16]. Nous effectuons ici des mesures afin de vérifier que c'est aussi le cas pour l'algorithme SCL. Deux cartes d'évaluation ont été utilisées pour cela. La carte Hikey960 embarque la puce Kirin960 de Hisilicon. Il s'agit d'une puce à 8 cœurs, 4 Cortex A73 et 4 Cortex A53 qui correspondent à la microarchitecture ARMv8-A. La valeur de puissance consommée n'est pas mesurée, car aucun capteur ne permet une telle mesure sur cette carte. Nous utilisons donc les valeurs de consommation données par le constructeur [30]. La carte Odroid XU4 embarque la puce Samsung Exynos 5422 à 4 cœurs Cortex A15 et 4 cœurs Cortex A7 de microarchitecture ARMv7-A. La puissance considérée pour chacun des cœurs est issue de [31, 32]. Tous les cœurs ARM possèdent le jeu d'instruction SIMD NEON.

Grâce à une fréquence de fonctionnement plus élevée, les processeurs de station de travail atteignent de plus hauts débits que les processeurs ARM. Nous observons également que malgré des fréquences de fonctionnement proches, le Cortex A73 présente un débit bien plus important que le Cortex A15. Cela est sans doute lié au changement de microarchitecture entre les deux générations. En effet, il y a une diminution d'environ 30% du nombre de cycles nécessaires pour décoder une trame. Un écart du même ordre de grandeur est observé entre les Cortex A53 et A7, qui sont des cœurs à faible consommation énergétique.

Du point de vue du nombre de cycles d'horloge nécessaires au décodage d'une trame, il est important d'observer que le Cortex A73 est compétitif par rapport aux processeurs de station de travail, le i7 et le Ryzen. En conséquence, puisque sa puissance est bien inférieure ($\approx 2W$ en charge contre $\approx 15W$), l'efficacité énergétique est elle 3 à 4 fois supérieure. Les Cortex A7 et A53 sont proches en termes d'efficacité énergétique tandis que le Cortex A15 s'avère être en retrait.

3.6 Résumé des contributions

L'infrastructure Cloud-RAN et plus généralement l'attrait grandissant pour le domaine des radios logicielles (SDR : Software Defined Radio) motivent l'étude et le développement de descriptions logicielles de décodeurs canal comme celles décrites dans ce chapitre.

Voici résumées les différentes contributions originales proposées.

(i) Au contraire des implémentations des algorithmes à liste logicielles précédentes, celles-ci sont les premières à permettre une représentation en virgule fixe sur une nombre réduit de bits des LLR.

(ii) Deux méthodes différentes de gestion des sommes partielles sont présentées. Leurs avantages et inconvénients sont étudiés.

(iii) Une méthode d'extraction des bits d'information, nécessaire avant les vérifications de CRC dans les différents algorithmes, est présentée. Elle permet d'augmenter significativement le débit des décodeurs.

(iv) Un algorithme de tri [23] très adapté aux algorithmes considérés est utilisé pour la première fois.

(v) Contrairement aux implémentations à haut débit et faible latence proposées dans les travaux précédents, les décodeurs proposés supportent l'algorithme FASCL.

(vi) Un type d'élagage était présenté dans des travaux précédents comme dégradant fortement les performances de décodage. Le paramétrage fin de l'élagage dans les décodeurs proposés a permis de montrer que ce fait est discutable suivant le code polaire considéré.

(vii) Les implémentations sur architecture ARM des algorithmes de décodage de codes polaires à liste sont les premières de la littérature.

Ces contributions ont fait l'objet d'un article en cours de revue [60] pour publication dans le *Journal of Signal Processing Systems*.

Des architectures programmables permettant l'implémentation efficace d'algorithmes de décodage de codes polaires sont détaillées dans la suite de ce manuscrit. L'objectif recherché est de conserver la souplesse et la flexibilité apportée par une description logicielle des algorithmes de décodage, tout en améliorant le débit, la latence, et surtout l'efficacité énergétique des décodeurs résultants.

CHAPITRE 4 CONCEPTION D'UN PROCESSEUR PAR LA SPECIALISATION DE SON ARCHITECTURE

Les chapitres 4 et 5 de ce manuscrit présentent deux architectures de processeurs ASIP spécialisées dans le décodage de code polaire. Ces deux architectures ont été conçues selon des méthodologies de conception différentes. La première méthodologie développée dans ce chapitre correspond à la spécialisation d'un processeur de base. Dans la section 4.1, le concept d'ASIP est introduit et la méthodologie utilisée pour la première architecture est décrite. Dans la section 4.2, la conception de l'ASIP est détaillée. Enfin les résultats d'implémentation et les performances de l'architecture en termes de débit, latence, complexité matérielle et consommation énergétiques sont présentés et discutés dans la section 4.3. La section 4.4 dresse une synthèse des résultats obtenus et pointe les limites de ce modèle de conception.

4.1 Les processeurs à jeu d'instructions spécifique à l'application

4.1.1 Principes de base

Afin de tendre vers un réseau virtualisé de type Cloud-RAN, les implémentations logicielles des fonctions de traitement du signal dans les infrastructures de communication radio sont encouragées. Dans le chapitre précédent, de telles implémentations sont présentées pour les algorithmes de décodage polaire à liste. De hauts débits peuvent être atteints. Malgré les contraintes que cela apporte, la flexibilité et la généricité de ces décodeurs sont très importantes. En utilisant des processeurs qui visent des systèmes électroniques embarqués, ces implémentations peuvent également gagner en efficacité énergétique.

Toutefois, les processeurs à usage général incluent plusieurs unités matérielles destinées à exécuter efficacement de nombreuses et diverses applications. Mais toutes les unités matérielles ne sont pas utilisées dans la plupart des applications. Par exemple, dans les algorithmes de décodage de canal, toutes les unités de calcul à virgule flottante sont inutiles puisqu'une représentation des données internes en virgule fixe est privilégiée. Ces unités matérielles consomment alors de l'énergie inutilement. Le profilage d'un décodeur polaire montre que la majeure partie du temps d'exécution est passée à réaliser un ensemble restreint de fonctions élémentaires. De plus, une part significative des instructions exécutées correspond à des opérations de sauvegarde et de chargement de données dans les registres.

Ces observations poussent à envisager la conception d'un processeur programmable qui exclurait les unités matérielles inutiles des processeurs à usage général, tout en intégrant des unités

de calculs spécialisées dans la réalisation efficace des fonctions élémentaires de décodage de codes polaires. Une telle architecture doit permettre de conserver une grande flexibilité nécessaire à la virtualisation des réseaux, tout en garantissant un niveau de performance se rapprochant des architectures dédiées. Ce type de processeurs entre dans la catégorie des processeurs à jeu d'instructions spécifique à l'application ASIP.

L'architecture ASIP proposée dans ce chapitre est basée sur un processeur de type RISC (Reduced Instruction Set Computer). Dans une première sous-section, les principes des architectures RISC sont présentés. Le concept d'ASIP est ensuite introduit. Les méthodologies de conceptions d'ASIP peuvent être divisées en deux groupes que nous détaillons. Le flot de conception de l'outil logiciel retenu pour concevoir ce premier ASIP sera ensuite présenté.

4.1.2 Les processeurs RISC

Dans le domaine de l'embarqué, les processeurs utilisés sont souvent de type RISC. Cette classe de processeurs a été introduite dans [33]. Par une analyse statistique et quantitative des applications traitées par les architectures de processeurs, les auteurs ont abouti à une microarchitecture de processeurs composée de 5 étages. Une unité matérielle est associée à chacun des étages, comme montré dans la Figure 4.1.

— Le premier étage est l'étage de **chargement de l'instruction** (IF : Instruction Fetch) depuis la *mémoire d'instruction* (IM : Instruction Memory). A chaque cycle d'horloge, une nouvelle instruction est chargée dans un registre spécialisé, nommé registre d'instruction. L'adresse en mémoire de l'instruction à charger est déterminée par le pointeur d'instruction, registre incrémenté automatiquement à chaque cycle d'horloge. L'instruction détermine l'opération à effectuer par le processeur dans les étages suivants. Il peut s'agir d'instructions arithmétiques et logiques, d'instructions de chargement et de sauvegarde depuis et vers la mémoire, ou bien d'instructions de branchements et de sauts. Ces dernières permettent de se déplacer dans la mémoire d'instructions. L'instruction contient également des informations sur les registres qui doivent être lus ou écrits, ainsi que des adresses mémoire dans le cas d'opérations de chargement ou de sauvegarde.

— Le deuxième étage correspond au **décodage de l'instruction** (ID : Instruction Decode) et à la lecture de la *file de registres* (RF : Register File) spécifié(s) par l'instruction. Divers signaux de contrôle, qui seront utilisés dans les étages suivants, sont générés, selon l'instruction décodée.

— Le troisième étage est l'étage d'**exécution** (EX : Execution) dans lequel l'unité arithmétique et logique (ALU : Arithmetic and Logic Unit) effectue des opérations arithmétiques (+,-,*,/) et logiques (AND, OR, XOR,...). Ces opérations peuvent servir à

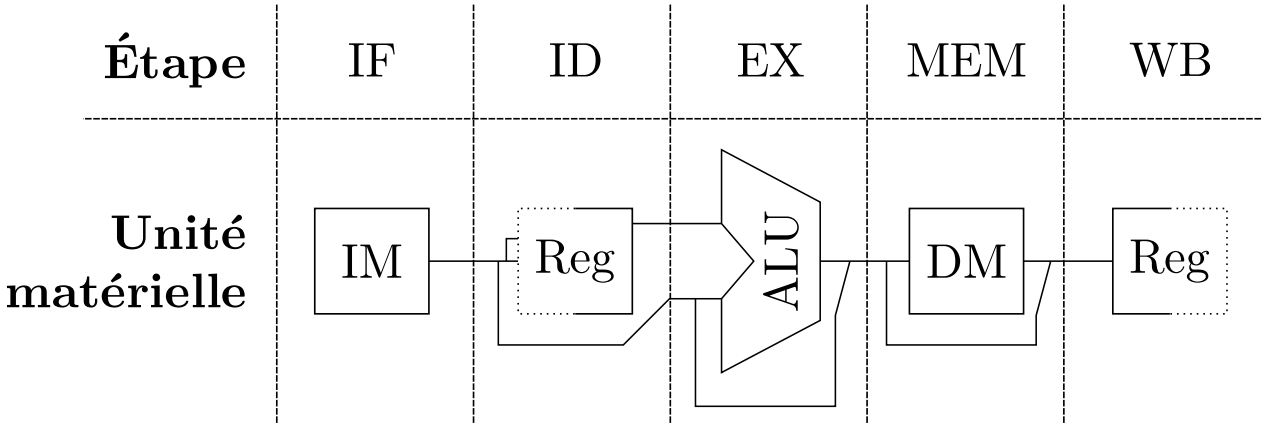


Figure 4.1 Étages d'un processeur RISC.

effectuer des calculs d'adresses relatives ou à réaliser des opérations sur deux opérandes. Ces deux opérandes peuvent être deux registres ou bien un registre et une valeur immédiate intégrée dans l'instruction elle-même.

— Le quatrième étage est l'étage d'**accès à la mémoire**. S'il s'agit d'un chargement, une donnée est lue dans la *mémoire de donnée* (DM : Data Memory) à l'adresse spécifiée dans un des registres. S'il s'agit d'une sauvegarde, la valeur d'un deuxième registre est écrite dans la DM.

— Le cinquième étage est l'étage d'**écriture différée** (WB : write-back). Cet étage permet d'écrire le résultat dans la file de registres, que ce soit le résultat d'une opération effectuée par l'ALU ou bien une donnée lue dans la DM.

L'ajout de registres de pipeline comme illustré dans la Figure 4.2 permet d'isoler chacun des étages. Ainsi à chaque cycle d'horloge, une instruction est transmise d'un étage au suivant. Si une instruction est initiée à chaque cycle d'horloge, alors la performance du processeur sera 5 fois supérieure à celle d'un processeur sans registre de pipeline.

Nous venons d'introduire l'architecture RISC telle que définie dans [33]. Cette dernière fait figure de référence dans le domaine des architectures de processeur. En effet, cette organisation en pipeline à cinq étages, à quelques différences près, se retrouve actuellement dans de nombreux processeurs. L'architecture de base des processeurs XTensa exploités dans ce chapitre est une architecture de type RISC. Les processeurs XTensa constituent une gamme de processeurs proposée par Tensilica, filiale de la société Cadence Design Systems.

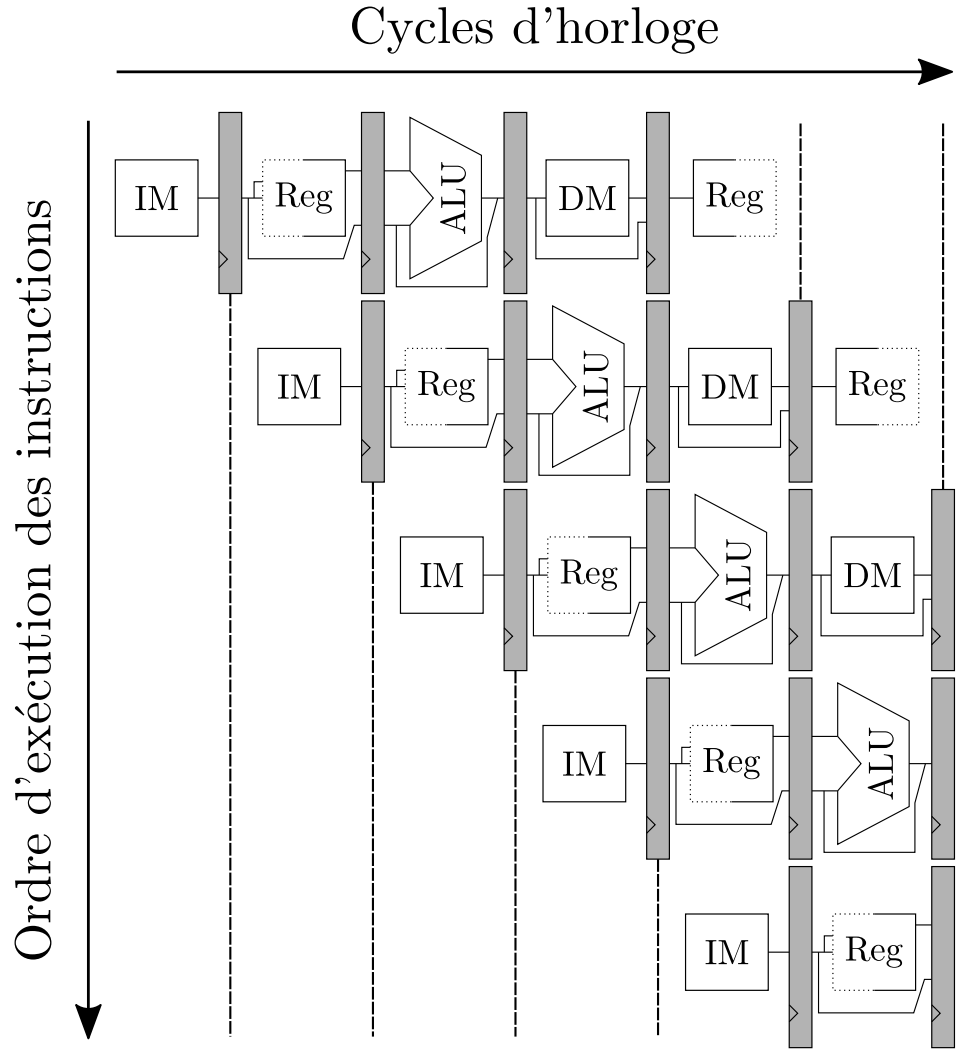


Figure 4.2 Structure et fonctionnement du pipeline RISC classique à 5 étages.

4.1.3 Les processeurs à jeu d'instructions spécifique à l'application

Les méthodologies de conception des ASIP peuvent être divisées en deux familles. La première famille correspond à un flot de conception basé sur la spécification complète du processeur, représentée dans la Figure 4.3a. Le modèle du processeur est décrit dans un langage de description architecturale (ADL : Architecture Description Language) [34]. Ces langages de description permettent de définir précisément l'architecture d'un processeur. Ils peuvent être de différents types. Les ADLs structurels décrivent les composants des processeurs ainsi que leurs interconnexions. Les ADLs comportementaux décrivent quant à eux le comportement du jeu d'instructions du processeur. Des langages de description mixtes permettent quant à eux de décrire simultanément la structure et le comportement du processeur. Le flot de conception utilisé pour concevoir le processeur décrit dans le chapitre 5 utilise des ADLs

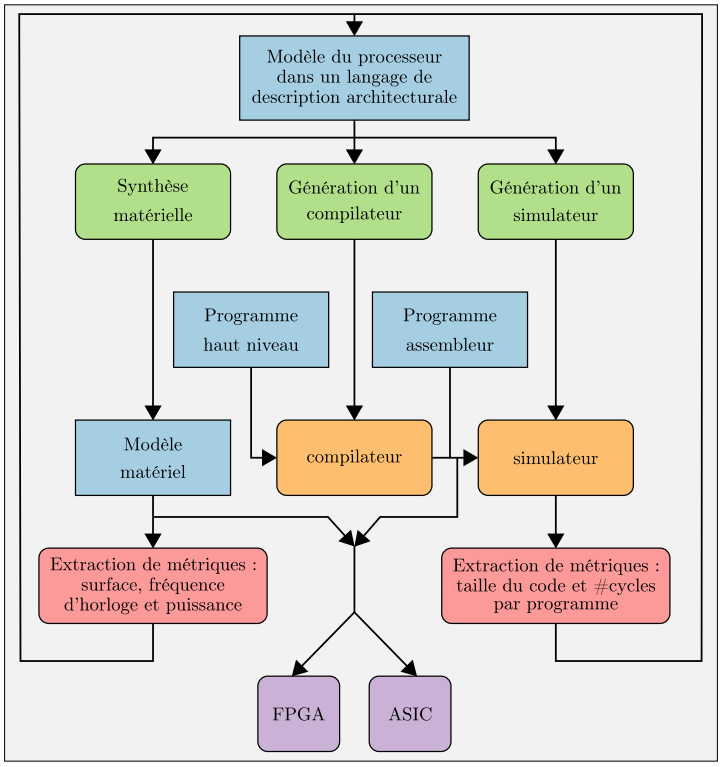
comportementaux et structurels afin de générer le modèle matériel du processeur.

La deuxième famille de méthodologies représentée dans la Figure 4.3b correspond à l'approche utilisée par les outils de Tensilica pour concevoir l'ASIP proposé dans ce chapitre. Une architecture peu complexe en terme de surface occupée et à faible consommation constitue la brique de base du processeur. Dans le cas des outils de Tensilica, cette base est un processeur RISC doté d'un pipeline de 5 étages, tel que présenté dans la sous-section 4.1.2. Afin de spécialiser le processeur, deux axes de conception sont disponibles. Premièrement, les principales unités matérielles du processeur RISC désignées dans la Figure 4.1 sont paramétrables. Des caractéristiques telles que la profondeur et la largeur de la file de registre et / ou l'interface avec les mémoires sont modifiables. Comme indiqué sur la Figure 4.3b, certaines fonctionnalités peuvent être ajoutées ou supprimées : unité de traitement reposant sur des représentations en virgule flottante (FPU : Floating Point Unit), unité d'accélération des multiplications (MUL) ou des divisions (DIV)... Le second axe est la possibilité d'étendre le jeu d'instructions. Grâce à un langage de description matérielle propre aux outils de Tensilica (TIE : Tensilica Instruction Extension) [35], il est possible de spécialiser le processeur pour le décodage de codes polaires en ajoutant des unités matérielles dédiées aux fonctions élémentaires $\mathtt{f}$, $\mathtt{g}$ et $\mathtt{h}$.

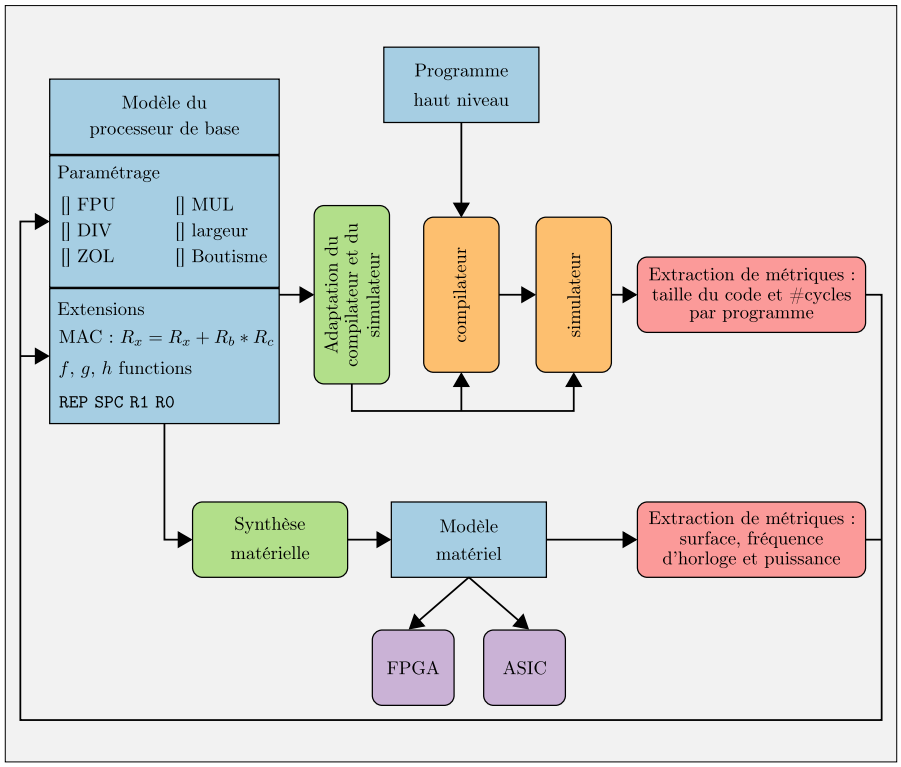
Un des objectifs de la conception de processeurs par les méthodologies de conception que nous venons de présenter est de réduire le temps de conception des systèmes. Pour cela, des étapes clés dans la conception des processeurs sont automatisées. En effet, les suites logicielles de conception d'ASIP permettent la génération automatique du compilateur, du simulateur, d'un outil de profilage et d'un outil de débogage.

La génération du compilateur demeure critique. En effet, l'effort à déployer pour créer un compilateur associé à un processeur spécifique est considérable. En revanche, les compilateurs associés aux outils de conception d'ASIP sont capables de s'adapter aux changements de structure du processeur. Ils proposent des optimisations spécifiques à la structure. Ils doivent également être capables de s'adapter au nombre de registres et les utiliser efficacement. Nous verrons dans le cas de notre processeur spécialisé dans le décodage de codes polaires que l'augmentation du nombre de registres à usage général favorise l'augmentation du débit de décodage. Le compilateur doit également être capable de gérer des modifications du pipeline et permettre un parallélisme d'instructions.

Le simulateur et l'outil de profilage générés permettent la validation fonctionnelle de l'architecture et du programme associé. Le nombre de cycles nécessaires à l'exécution du programme est obtenu à l'aide du simulateur. L'outil de profilage fournit des informations importantes sur la durée de l'exécution de sous-parties du programme afin de cibler les sections les plus



(a) Flot basé sur la spécification complète du processeur.



(b) Flot basé sur la particularisation d'un processeur de base.

Figure 4.3 Méthodologies de conception des ASIP.

consommatrices en temps. Ces métriques déterminent d'éventuelles modifications à apporter pour concevoir un nouveau modèle du processeur. L'outil de conceptions d'ASIP générera alors de nouveau un compilateur, un simulateur et un outil de profilage pour extraire les nouvelles métriques. Des itérations successives de ce flot permettent d'améliorer progressivement les performances recherchées de l'architecture et du programme considérés. Ce processus serait difficilement réalisable sans une telle suite d'outils automatisés.

Une fois l'architecture du processeur déterminée, le modèle matériel du processeur est généré. Ce modèle matériel peut être synthétisé et implémenté sur différentes cibles, ASIC ou FPGA. Il est alors possible d'extraire les métriques d'implémentations : fréquence de fonctionnement, surface occupée, puissance dissipée. De nouveau, des itérations sont possibles afin d'améliorer progressivement les performances. Pour le processeur XTensa conçu dans le cadre de cette étude, la génération du modèle RTL n'a pas été possible par défaut de licence suffisante de l'outil de conception d'ASIP. Seules des estimations de fréquence, de surface et de puissance dissipée sont fournies par l'outil. Cette absence de modèle matériel a été une des raisons pour lesquelles nous avons retenu par la suite d'autres méthodologies de conception telles que celle décrite dans le chapitre 5.

4.2 Un ASIP dédié au décodage de codes polaires

4.2.1 Paramétrage du processeur de base

4.2.1.1 Architecture de base

La philosophie de conception de l'ASIP proposé est de réaliser l'ensemble des fonctions arithmétiques élémentaires pour le décodage de codes polaires à l'aide d'unités matérielles spécialisées. À l'inverse, les opérations de contrôle sont effectuées par l'architecture de référence du processeur XTensa. C'est la raison pour laquelle, l'architecture de référence a été simplifiée. Ainsi, les unités de calcul évoluées (unité de calcul en virgule flottante, MAC, ...) ont été supprimées.

La fonctionnalité FLIX correspond à la possibilité d'ajouter des ALU supplémentaires à la structure pipeline principale des processeurs XTensa. Cela permet au processeur d'exécuter plusieurs instructions en parallèle. Il est possible de configurer les instructions réalisées par chaque niveau pipeline. Notre choix s'est porté sur une configuration nommée FLIX3 prédéfinie par l'outil de conception. Ainsi, trois instructions faisant partie du jeu d'instruction de base peuvent être appliquées simultanément sur trois couples de données de 32 bits. Ce parallélisme d'instructions permet de réaliser plus rapidement les fonctions de contrôle et de calcul d'adresse. La fonctionnalité FLIX est détaillée dans la Figure 4.4. La profondeur de la

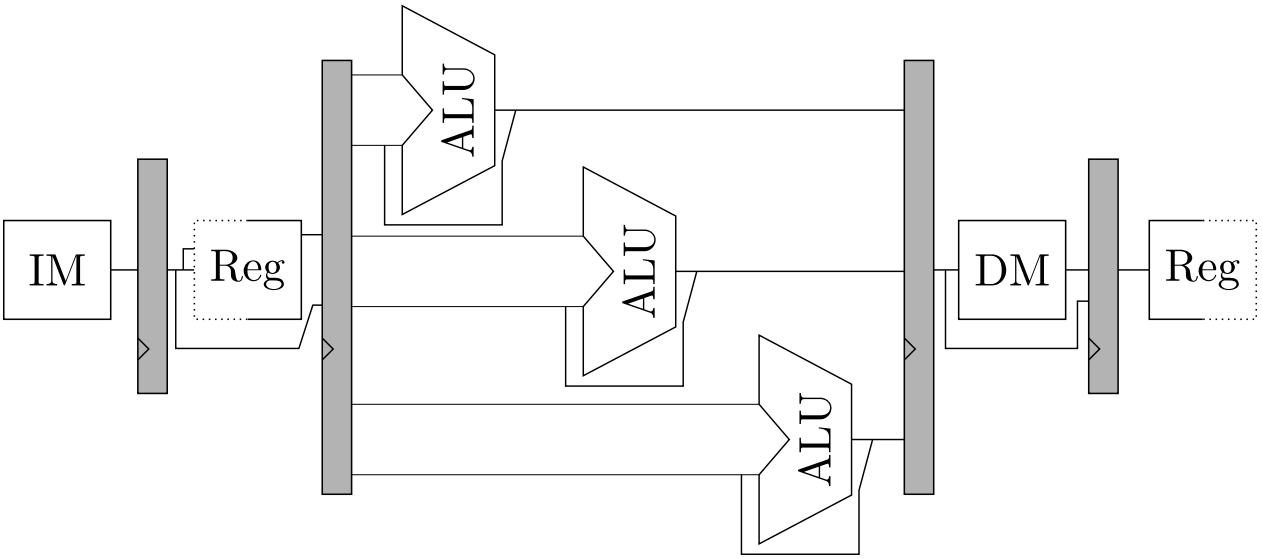


Figure 4.4 Architecture associée à la fonctionnalité FLIX.

file de registres a été augmentée de 32 à 64 données de 32 bits. Cette modification permet une meilleure exploitation de l'augmentation du parallélisme d'instructions.

4.2.1.2 Format de représentation des données internes

Il est démontré dans [3] que des LLR représentés sur 6 bits permettent d'atteindre les performances de décodage d'implémentations en virgule flottante. Cependant, il est plus simple dans une implémentation logicielle de représenter ces données sur 8 bits puisque les langages de programmation définissent de tels types de données. Comme montré dans [27], 2^{n-d} LLR doivent être stockés à un niveau d de l'arbre de décodage. La taille de la mémoire permettant de stocker les LLR est donc de $2^{n+1} - 1$ octets.

Les sommes partielles sont des valeurs binaires. Toutefois, la manipulation des sommes partielles dans le langage de description logicielle est plus aisée si un entier de 8 bits est utilisé pour représenter chaque somme partielle. Il serait possible d'utiliser un entier pour représenter 8 sommes partielles, ce qui réduirait l'empreinte mémoire de celles-ci. Cela nécessiterait en contrepartie des opérations de masquage supplémentaires et cela introduit plus d'irrégularités au niveau des accès aux données.

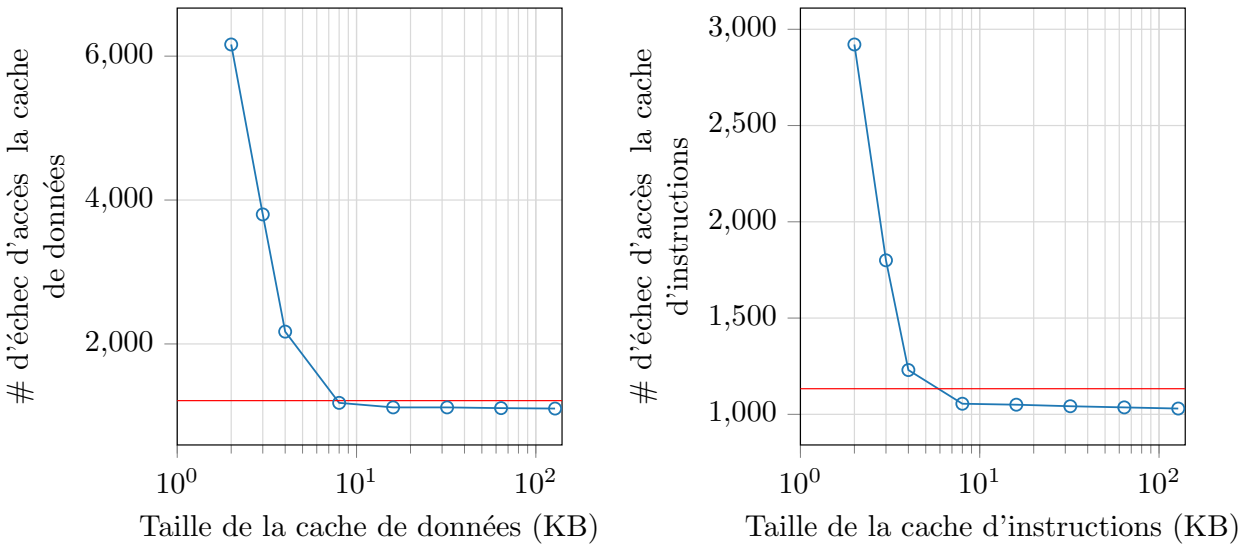


Figure 4.5 Nombre d'échecs d'accès à la mémoire cache en fonction de la taille de la mémoire pour le décodage SC d'un code polaire de taille $\mathsf{N} = 1024$.

4.2.1.3 Configuration de la mémoire cache

Nous avons fait le choix d'augmenter au maximum le parallélisme des instructions élémentaires nécessaires au décodage de codes polaires. La taille maximum des registres du processeur XTensa, qui est donc la taille utilisée dans l'ASIP proposé, est de 512 bits. Afin de pouvoir charger et sauvegarder des données depuis et vers la mémoire cache en un seul cycle d'horloge, la largeur choisie pour une ligne de mémoire cache est également de 512 bits. La taille de ces mémoires ainsi que leur associativité est également configurable. Des expérimentations ont été réalisées afin de sélectionner les valeurs. Il en résulte qu'une associativité à 4 voies pour les mémoires d'instructions et de données permet d'atteindre de meilleures performances.

Le scénario envisagé pour la conception du processeur est celui des canaux de contrôle du standard 5G tel que défini dans [36]. La taille du plus grand code polaire à décoder est $\mathsf{N} = 1024$. Les mémoires du processeur ont donc été dimensionnées en conséquence. Bien que le plus petit nombre d'échec d'accès à la mémoire cache soit atteint pour la taille de cache la plus grande (128 Ko), une mémoire d'instruction de 8 Ko ainsi qu'une mémoire de données de 8 Ko sont suffisantes pour atteindre un nombre d'échecs de cache seulement 10 % plus grand que la valeur optimale. Les mesures réalisées sont reportées dans la Figure 4.5

4.2.2 Ressources calculatoires utilisées dans les implémentations matérielles de l'état de l'art

Les instructions spécialisées choisies pour étendre le jeu d'instructions de l'ASIP s'inspirent des unités de traitement développées dans la littérature pour les circuits dédiés au décodage de codes polaires SC. Dans [27] plusieurs architectures furent proposées. La première architecture est l'architecture dite « papillon ». Lors du décodage, les fonctions f et g doivent être appliquées $\mathsf{N}/2$ fois à chaque niveau de l'arbre. Au total, il est nécessaire d'appliquer $\frac{\mathsf{N}}{2}\log\mathsf{N}$ fois les fonctions f et g lors du décodage d'un mot de code.

Pour cela, des unités matérielles élémentaires (PE : Processing Elements) ont été définies. Un PE permet de réaliser un calcul de f ou un calcul de g. Un tel élément est représenté dans la Figure 4.6. Dans l'architecture dite « papillon », un PE est associé à chaque application des fonctions f et g. Il y a donc au total $\frac{\mathsf{N}}{2}\log\mathsf{N}$ PE. L'application de l'ensemble des fonctions f ou des fonctions g d'un nœud de l'arbre est alors effectuée en un cycle d'exécution.

Cependant, cette architecture est sous-optimale en termes de complexité matérielle. L'ordonnancement de l'algorithme de décodage SC possède les propriétés suivantes :

— Le nombre maximum de fonctions polaires élémentaires pouvant être réalisées simultanément sur un nœud est $\frac{\mathsf{N}}{2}$,

— deux nœuds successifs ne peuvent pas être traités simultanément.

En conséquence, les ressources de calcul allouées au traitement d'un nœud de l'arbre peuvent être utilisées pour le traitements des nœuds suivants. Le niveau de parallélisme le plus élevé est $\frac{\mathsf{N}}{2}$, donc $\frac{\mathsf{N}}{2}$ PE suffisent pour décoder l'ensemble de l'arbre sans impacter la latence de décodage. Dans le cas de l'architecture dite « ligne », $\frac{\mathsf{N}}{2}$ PE sont partagés pour le traitement de tous les nœuds de l'arbre. Sa latence est la même que celle de l'architecture dite « papillon ».

Dans [37] il est proposé de réduire le nombre de PE grâce à l'introduction d'une architecture semi-parallèle. Cette architecture s'inspire d'implémentations pour le décodage de codes LDPC [38]. Dans un décodeur dit « ligne », les N / 2 PE ne sont utilisés simultanément que deux fois durant le décodage d'un mot de code. Cela signifie que le taux d'utilisation des PE est très faible. Il est donc possible de réduire le nombre de PE sans impacter significativement le débit. Cette réduction modifie légèrement l'ordonnancement de l'algorithme de décodage. En effet, les nœuds nécessitant un nombre d'exécutions de fonctions polaires supérieures à P nécessiteront plus d'un cycle d'horloge à décoder. Le rapport du nombre de PE d'une architecture semi-parallèle sur le nombre de PE d'une architecture « ligne » est alors $\mathsf{N}/2\mathsf{P}$. Par exemple, pour $\mathsf{N} = 1024$ et $\mathsf{P} = 64$, le nombre de PE est réduit d'un facteur 16, tandis que le débit est seulement réduit de 2%. Pour $\mathsf{N} = 2^{20}$ et $\mathsf{P} = 64$, le nombre de PE est réduit d'un

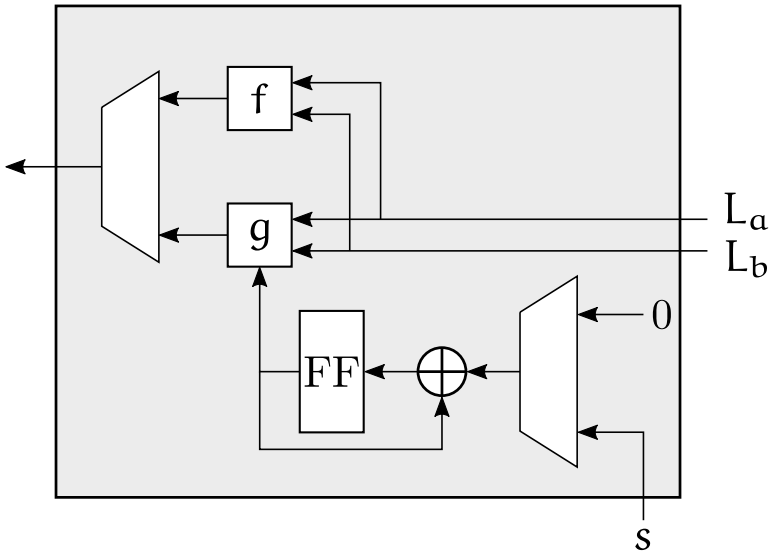


Figure 4.6 Unité matérielle élémentaire (PE : Processing Element).

facteur 8192, et le débit est réduit de moins de 10%. L'architecture semi-parallèle s'avère donc très pertinente.

L'architecture de décodage proposée dans [3] reprend le principe de l'architecture semi-parallèle. Cependant, les PE sont modifiés pour intégrer des unités matérielles de traitement des nœuds spécialisés `R1`, `SPC` et `REP`. Les PE contiennent également une unité particulière `REP-SPC` permettant d'accélérer le traitement de deux nœuds `REP` et `SPC` voisins. Ce motif particulier apparaît régulièrement pour le code polaire (32768,29492) considéré dans la publication. Les unités `REP-SPC` ne sont pas utilisées dans le processeur proposé car elles n'apportent pas de gain significatif pour les tailles de codes traitées dans le cadre du standard 5G, à savoir $\mathsf{N} < 1024$.

Le dernier type de calcul nécessaire dans l'algorithme de décodage SC est le calcul des sommes partielles (fonction $\mathtt{h}$). Dans les implémentations dédiées, les sommes partielles sont stockées dans des registres. Un réseau de portes *ou-exclusif* routées sur ces registres permet leur mise à jour instantanée à chaque traitement d'une feuille de l'arbre de décodage. Ce réseau ne fait pas partie des PE. Dans l'ASIP proposé, les sommes partielles sont stockées dans la mémoire de données, comme les LLR, afin de les rendre accessibles par le processeur comme toute autre variable. La fonction $\mathtt{h}$ fait donc partie des instructions spécialisées conçues pour étendre le jeu d'instructions de l'ASIP.

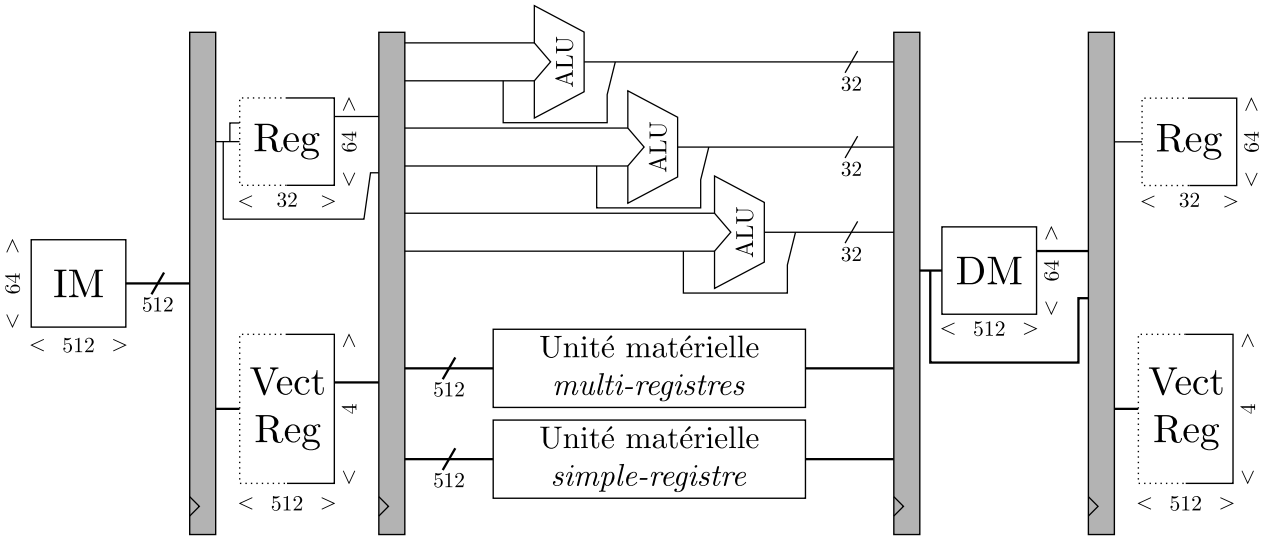
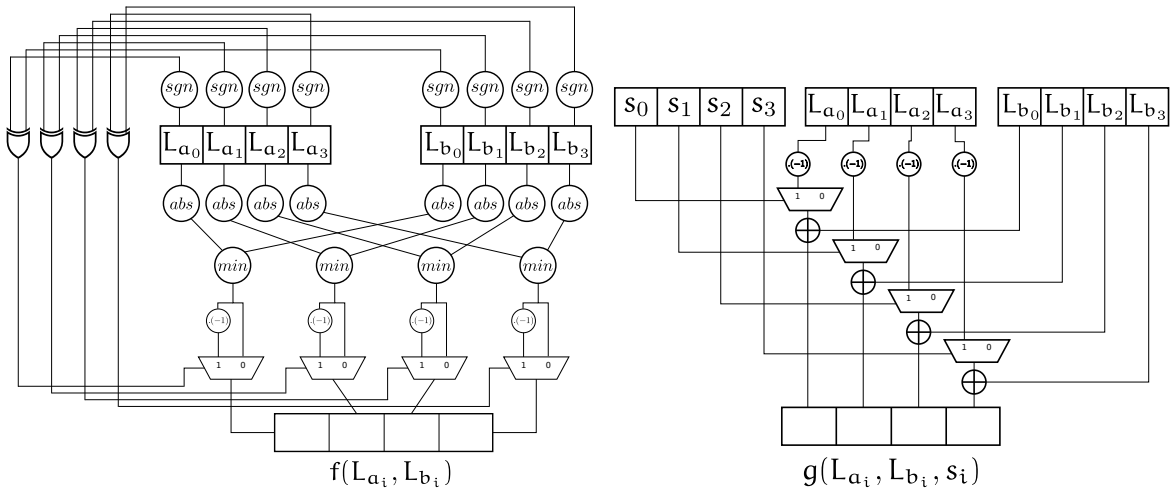


Figure 4.7 Architecture de l'ASIP proposé.

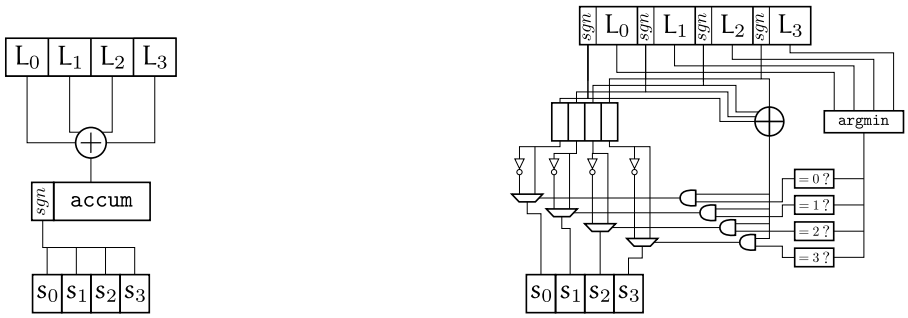
4.2.3 Instructions spécialisées *multi-registres*

Les instructions spécialisées réalisent des fonctions équivalentes à celles des PE des architectures matérielles qui ont été présentées dans la sous-section 4.2.2. Il s'agit des fonctions $\mathtt{f}$, $\mathtt{g}$, `R1`, `R0`, `REP` et `SPC`. Deux types d'instructions spécialisées ont été conçus et ajoutés au processeur proposé comme montré dans 4.7. Les premières sont les instructions *multi-registres* et les secondes les instructions *simple-registre*. À l'image des PE de l'architecture semi-parallèle, leur niveau de parallélisme est fixé ($\mathtt{P} = 64$). Une file de registres vectoriels permet de stocker des données sur 512 bits. Les instructions *multi-registres* lisent et écrivent dans cette file de registres vectoriels.

Ces instructions spécialisées sont décrites à l'aide du langage TIE associé aux outils de Tensilica. L'implémentation de la fonction $\mathtt{f}$ est représentée dans la Figure 4.8a. Les deux opérandes sont les LLR notés $\mathtt{L_a}$ et $\mathtt{L_b}$ et la sortie est notée $\mathtt{f}(\mathtt{L_a},\mathtt{L_b})$. Ces deux entrées et la sortie correspondent à des registres de la file de registres vectoriels. Afin de clarifier le schéma, le niveau de parallélisme dans la Figure 4.8a a été réduit à $\mathtt{P} = 4$. Dans la plupart des décodeurs polaires, pour réduire le chemin critique, les LLR sont représentés au format « signe-amplitude » : un bit est utilisé pour le signe, et le reste pour la valeur absolue. En revanche, dans notre implémentation, les données sont représentées en complément à deux, car il s'agit du mode de représentation utilisé dans les processeurs XTensa. L'architecture associée à la fonction $\mathtt{g}$ est détaillée dans la Figure 4.8b. Cette fonction consiste en une addition simple avec une inversion de signe selon la valeur de la somme partielle $\mathtt{s_a}$. La fonction $\mathtt{h}$ n'est pas représentée,



(a) Fonction f (b) Fonction g



(c) Fonction `REP` (d) Fonction `SPC`

Figure 4.8 Implémentations matérielles des instructions spécialisées.

il s'agit d'un simple *ou-exclusif* entre les sommes partielles d'entrées. Les fonctions `R0` et `R1` sont également simples. La première consiste en une mise à zéro des sorties. La seconde est un seuillage des variables d'entrée. Le seuillage est une simple copie du bit de poids fort.

Comme expliqué dans la sous-section 2.3.2, le traitement d'un nœud de répétition consiste en l'addition de tous les LLR d'entrée pour former la variable notée « `accum` » dans la Figure 4.8c. Puis, une détection du signe de cette variable est effectuée. Elle détermine le vecteur `s` en sortie. Un arbre binaire est parcouru afin de réaliser l'addition nécessaire. Pour éviter tout débordement, la taille du registre de stockage de la variable `accum` est de 32 bits.

L'unité matérielle de traitement des nœuds `SPC` est représentée dans la Figure 4.8d. Il s'agit de l'unité la plus complexe car elle comprend un tri des LLR. Aussi, le parallélisme de cet unité est volontairement limité à 8 dans notre implémentation.

4.2.4 Instructions spécialisées *simple-registre*

Dans les implémentations logicielles, la séquence d'opérations nécessaire à l'exécution des fonctions élémentaires polaires est la suivante :

(i) chargement depuis la mémoire cache vers les registres des variables d'entrée (LLR et / ou sommes partielles),

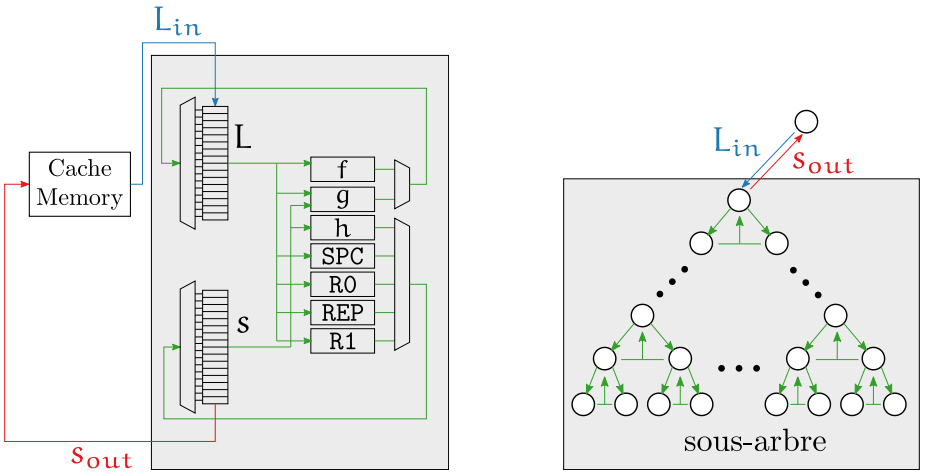
(ii) calcul du résultat de la fonction élémentaire, en une ou plusieurs étapes, dont la sortie est stockée dans un registre,

(iii) sauvegarde de cette variable de sortie dans la mémoire cache du processeur.

Par ailleurs, dans les implémentations matérielles présentées dans la sous-section 4.2.2, le chemin de données part de la sortie de la mémoire, passe par les fonctions combinatoires réalisant les fonctions polaires élémentaires et finit à l'entrée de la mémoire. Ce chemin de données est parcouru en un seul cycle d'horloge. La méthode présentée dans cette sous-section, nommée *simple-registre*, est conçue dans le but de reproduire le fonctionnement des implémentations matérielles de décodeur de codes polaires dans notre architecture de processeur.

La méthode *simple-registre* est appliquée sur les sous-arbres de décodage dont le nœud racine contient 64 LLR. La Figure 4.9b illustre un tel sous-arbre. Ces 64 LLR sont notés $\mathbf{L_{in}}$. Lorsque le programme a généré les LLR de la racine de ce sous-arbre, ceux-ci sont stockés dans un registre dédié. Ce registre est noté $\mathbf{L}$ dans la Figure 4.9a qui décrit l'unité matérielle sur laquelle s'exécutent les instructions *simple-registre*. Le décodage du sous-arbre se fait ensuite intégralement à travers les registres, sans passer par la mémoire cache. Les sommes partielles notées $\mathbf{s_{out}}$ constituent la sortie de l'unité matérielle. Cette méthode reproduit le fonctionnement des implémentations matérielles de l'algorithme SC : l'exécution d'une fonction élémentaire quelconque, comprenant la lecture et l'écriture de la donnée depuis le registre, en un seul cycle d'horloge.

Elle résout également un problème récurrent des implémentations logicielles utilisant du parallélisme : l'alignement des données. En effet, pour pouvoir appliquer une fonction vectorielle des données de deux registres vectoriels, il est nécessaire de les aligner. Cela implique des opérations supplémentaires. Dans l'implémentation *simple-registre*, l'alignement est prévu en amont et implémenté dans l'unité matérielle. Aucune instruction supplémentaire n'est nécessaire.



(a) Unité matérielle *simple-registre*.

(b) Un sous-arbre décodé grâce aux instructions *simple-registre*.

Figure 4.9 Instructions *simple-registre*.

4.2.5 Description logicielle de l'algorithme de décodage

L'ASIP ainsi proposé permet une exécution efficace de l'algorithme de décodage SC de codes polaires. Le décodage est décrit de manière logicielle, en langage C comprenant des instructions spécialisées. À titre d'information, un extrait du code source est donné dans la Figure 4.10. Tout d'abord, les fichiers d'en-tête produits par les outils de conception définissent les types associés aux files de registres vectoriels (`VR512`). Afin de charger une donnée en mémoire cache vers les registres vectoriels, il est nécessaire de créer des pointeurs vers ces variables de type (`VR512`). Les données sont ensuite chargées vers deux de ces registres vectoriels. La fonction spécialisée, ici la fonction `custom_f_64` est ensuite appliquée. Le résultat est stocké dans la mémoire cache.

Lors du développement, il est apparu qu'il était difficile de réduire la durée des opérations liées au parcours de l'arbre et aux tests associés. L'utilisation de l'option FLIX3 a permis une première réduction de ce temps, bien qu'insuffisante. La technique de déroulage présentée en sous-section 3.2.2 permet une réduction plus significative. Dérouler le code améliore la performance au prix d'une réduction de la flexibilité. Il faut en effet générer une version différente du code pour un ensemble donné de paramètres.

Dans l'implémentation logicielle proposée, une alternative au déroulage, nommée pseudo-déroulage, est privilégiée. Tout d'abord, toutes les fonctions élémentaires pour le décodage de codes polaires possèdent le même prototype de fonctions dans la description logicielle.

Le type de fonction à appliquer (`f`, `g`, `h`, `R0`, `R1`, `SPC`, `REP`) est un paramètre de la fonction. Il est accompagné des adresses auxquelles accéder aux données à lire et écrire. De cette

```
1 inline void f_64(signed char* la,
2                  signed char* lb,
3                  signed char* lc)
4 {
5   p512_a = (VR512*) la; // transtyper de char* vers VR512
6   p512_b = (VR512*) lb; // transtyper de char* vers VR512
7   p512_c = (VR512*) lc; // transtyper de char* vers VR512
8
9   v512_a = *p512_a; // chargement des donnees vers VR512
10  v512_b = *p512_b; // chargement des donnees vers VR512
11
12  // execution de l'instruction specialisee
13  v512_c = custom_f_64(v512_a, v512_b);
14
15  // sauvegarde du resultat en cache
16  *p512_c = v512_c;
17 }
```

Figure 4.10 Description logicielle de l'exécution d'une instruction spécialisée.

manière, il est possible de créer un tableau contenant les pointeurs des fonctions à appeler. L'algorithme permettant de déterminer la séquence de fonctions à exécuter prend comme entrée le tableau de bits gelés correspondant au code polaire.

Comme ce tableau est généré par le processeur lors de l'exécution, la généricité du décodeur est donc conservée. Les paramètres N et K ainsi que les indices des bits gelés sont donnés en entrée du programme. Ainsi, le processeur s'adapte automatiquement. Comme cela est détaillé dans la section suivante, ce pseudo-déroulage permet une réduction significative du nombre de cycles d'horloges nécessaires au décodage d'un mot de code.

4.3 Expérimentations et mesures

Afin d'évaluer le processeur conçu, des expérimentations ont été réalisées. Le but est d'une part de quantifier le gain associé aux différentes améliorations du processeur, d'autre part de comparer les performances de débit, de latence et de consommation énergétique de l'architecture proposée avec celles obtenues sur d'autres architectures programmables.

4.3.1 Mesure du gain lié à la spécialisation du processeur

Les codes polaires considérés ont été construits selon le document spécifiant les techniques de modulation et de codage pour le futur standard 5G [36]. La version académique des outils de conception de Tensilica ne permet pas d'effectuer une synthèse logique du processeur ou

```
1 // fonction polaire spécialisée
2 void u_f_64(struct polar_operation& op)
3 {
4     for(int i = 0; i < op.n_elmts; i+=64)
5       F_64(op.arg0 + i, op.arg1 + i, op.arg2 + i);
6 }
7
8 // fonction polaire générique
9 typedef void (*polar_func)(struct polar_operation&);
10
11 // structure passée en argument de la fonction générique
12 struct polar_operation
13 {
14   polar_func   p_func; // pointeur vers la fonction polaire spécialisée
15
16   signed char* arg0; // les arguments des fonctions polaires spécialisées
17   signed char* arg1; // ces arguments sont souvent des adresses
18   signed char* arg2;
19   signed char* arg3;
20
21   int          n_elmts; // le nombre d'éléments traités en parallèle
22   int          off_s; // un offset nécessaire pour l'accès aux PSs
23 };
24
25 // le tableau polar_op_vec est initialisé lors de la construction
26 // du décodeur il contient la séquence des "polar operation"
27 std::vector<struct polar_operation> polar_op_vec;
28
29 // boucle principale du décodage
30 virtual void decode()
31 {
32   const int n_op = polar_op_vec.size();
33   for (int i = 0; i < n_op; i++)
34   {
35     // chaque fonction du tableau "polar_op_vec" est appelée
36     polar_op_vec[i].p_func(polar_op_vec[i]);
37   }
38 }
```

Figure 4.11 Description logicielle du pseudo-déroulage.

d'accéder à sa description RTL. Seules des estimations de la fréquence, de la consommation et de la surface occupée sont disponibles. Nous avons sélectionné la technologie HPM 28nm de TSMC pour ces estimations.

Le Tableau 4.1 présente l'impact des améliorations principales apportées au processeur de base. Un code polaire (32768,29492) est décodé. Il s'agit là du code utilisé durant le développement du processeur. Les mémoires de données et d'instructions sont dimensionnées pour ce code. La première implémentation logicielle de l'algorithme SC sur l'architecture de base du processeur XTensa LX7 nécessite 7.3 millions de cycles d'exécution pour décoder un mot de code. Quant à l'architecture spécialisée pour le décodage de codes polaires combinée à la technique de pseudo-déroulage de la description logicielle, elle permet de réduire le nombre de cycles d'exécution d'un facteur 50. Les instructions *multi-registres* sont responsables d'une grande partie de cette réduction. Elles permettent à elles seules de diminuer d'un ordre de grandeur le nombre de cycles nécessaire au décodage d'un mot de code.

Tableau 4.1 Impact de chaque amélioration de l'ASIP sur le nombre de cycles d'horloges nécessaires pour décoder une trame, le débit et la surface occupée. La taille des mémoires n'est pas prise en compte pour la surface occupée. Fréquence considérée : 835 MHz.

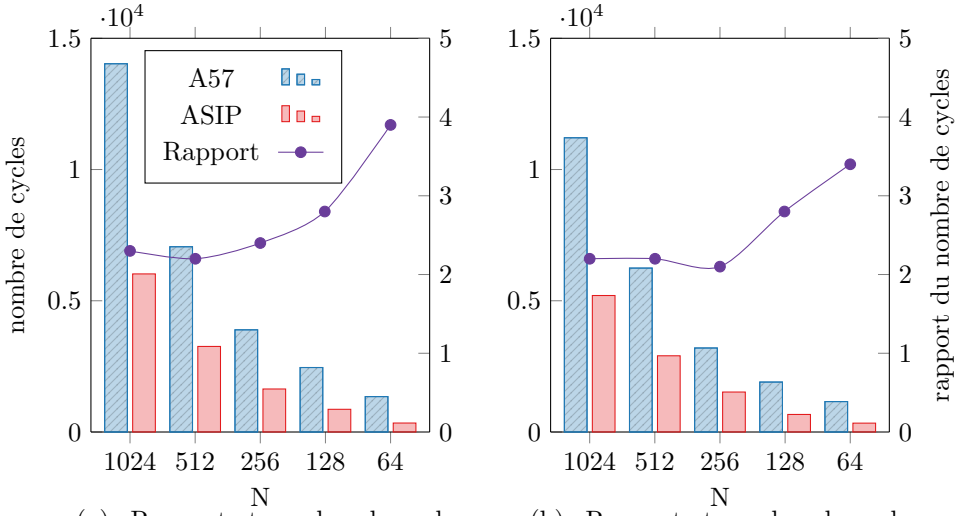
Version du processeur	**# cycles**	$\mathcal{T}_i$ **(Mb/s)**	**Surface (portes)**
Processeur de base	7.30×10^6	3.4	160000
+ instructions *multi-registres* `f`,`g` et `h`	2.64×10^6	9.3	210500
+ instructions *multi-registres* `R0`, `R1`,`SPC` et `REP`	534.00×10^3	46	272000
+ FLIX3	296.00×10^3	83	311900
+ instructions *simple-registre*	192.00×10^3	128	445500
+ pseudo déroulage	151.00×10^3	163	445500

La puissance consommée estimée pour la version complète de l'ASIP est de 111 mW, et la surface occupée est 0.475 mm^2. La fréquence estimée est 835 MHz. Cependant, les instructions spécialisées ne sont pas prises en compte pour l'estimation de la fréquence. En conséquence, un scénario plus réaliste où la fréquence de fonctionnement est 400 MHz est également présentée. Pour cette fréquence, la consommation estimée est 49 mW et la surface occupée 0.374 mm^2.

4.3.2 Comparaison avec un processeur ARM

Le logiciel AFF3CT développé par l'équipe CSN du laboratoire IMS de Bordeaux est utilisé pour exécuter une implémentation optimisée de l'algorithme SC sur un processeur ARM Cortex A57. La méthode de parallélisation *intra-trame* est utilisée, et les LLR sont représentés

sur 8 bits en virgule fixe. La taille des données gérées par les instructions NEON est de 128 bits. Cela signifie que le parallélisme est égal à 16. Les résultats ici reportés ont été fournis par les auteurs de [16]. Dans ces résultats, la consommation de la mémoire RAM n'est pas prise en compte. Le nombre moyen de cycles d'horloges nécessaires à chaque processeur pour décoder une trame est donné dans la Figure 4.12. Le processeur proposé permet une réduction significative du nombre de cycles grâce à l'utilisation d'unités matérielles de calcul spécialisées. La courbe tracée représente le rapport entre les nombres de cycles d'horloge de chaque processeur. Ce rapport est d'autant plus grand que N est faible. Ceci est dû au fait que lorsque N est faible, ce sont les instructions *simple-registre* qui sont le plus utilisées. Or ces instructions sont les plus efficaces. Cependant, ce nombre réduit de cycles d'horloge est contrebalancé par une fréquence réduite comme montré dans le Tableau 4.2. Néanmoins, même dans le scénario le plus pessimiste, avec une fréquence de 400 MHz, le débit atteint par le processeur proposé est similaire à celui obtenu avec le processeur ARM. Qui plus est, l'énergie dépensée par bit décodé est 10 fois inférieure pour l'ASIP.



(a): Rapport et nombre de cycles pour un code de rendement 1/2

(b): Rapport et nombre de cycles pour un code de rendement 1/4

Figure 4.12 Comparaison du nombre de cycles de l'horloge nécessaires au décodage SC entre le Cortex ARM A57 et l'ASIP proposé.

4.3.3 Comparaison avec un processeur d'architecture x86

Le débit, la latence et l'énergie consommée par bit décodé du décodeur logiciel SC inclus dans le logiciel AFF3CT ont également été mesurés sur un processeur Intel i7-4712HQ. La puissance du cœur est évaluée avec l'outil HWiNFO [29]. Seule la consommation du cœur

du processeur, excluant la consommation de la mémoire cache de niveau 3 et de la mémoire externe, est reportée. Cette fois-ci, le niveau de parallélisme est de 32. Ce niveau est atteint à l'aide du jeu d'instructions AVX2. La fréquence du CPU mesurée durant l'exécution est de 3.3 GHz. Les résultats récapitulés dans le Tableau 4.2 montrent que le débit et la latence des implémentations de décodeur SC sur les processeurs i7 sont bien meilleurs que ceux obtenus avec le processeur ASIP proposé ou avec le processeur ARM. Cependant, la consommation énergétique est moins bonne. Une implémentation *inter-trame* permettrait d'augmenter cette efficacité énergétique, au prix d'une augmentation de la latence, comme démontré dans [16].

Tableau 4.2 Comparaison de la latence, du débit et de la consommation énergétique de décodeurs SC pour différents processeurs.

Target	**N**	**Latency [µs]**	**Throughput [Mb/s]**	$\mathcal{E}_b$ **[nJ]**
A57-1.1GHz	1024	13	38	21
	512	6.7	38	21
	256	3.6	35	22
	128	2.1	30	27
i7-3.3GHz	1024	2.3	222	47
	512	1.4	182	57
	256	0.8	155	68
	128	0.5	124	85
ASIP-835MHz	1024	7.2	71	1.6
	512	3.9	66	1.7
	256	1.9	65	1.7
	128	1.0	62	1.8
ASIP-400MHz	1024	15	34	1.4
	512	8.2	31	1.6
	256	4.1	31	1.6
	128	2.1	30	1.7

4.4 Synthèse et limites du modèle architectural

Une première architecture de processeur ASIP spécialisée dans le décodage de codes polaires est présentée dans ce chapitre. Les outils logiciels de Tensilica permettent de spécialiser un processeur de type RISC. Après avoir détaillé la structure générale de ces microarchitectures RISC, nous expliquons comment elles peuvent être configurées et étendues par l'utilisation des outils logiciels de Tensilica. Les avantages et les enjeux liés à l'utilisation de tels outils de conception sont mis en avant.

Dans la deuxième partie du chapitre, nous expliquons comment ces outils ont été mis en œuvre pour notre problématique : le décodage de codes polaires. Tout d'abord les paramètres de l'architecture RISC de base sont modifiés et adaptés. Entre autres, une fonctionnalité permettant le support d'un parallélisme d'instruction est activée. La largeur du bus d'interface avec la mémoire de données est augmentée. Ensuite, des unités matérielles dédiées associées à des instructions spécialisées qui étendent le jeu d'instructions de base sont ajoutées. Enfin, une méthode de description logicielle nommée « pseudo-déroulage » est proposée afin d'améliorer les performances de débit et de latence du processeur spécialisé conçu.

Dans une troisième partie, nous présentons et comparons le débit, la latence et la consommation du décodeur de type ASIP proposé avec les implémentations logicielles de codes polaires de l'état de l'art. Pour un code polaire (1024,512) décodé à l'aide de l'algorithme SC, le débit atteint est d'environ 70 Mb/s, dépassant les performances obtenues avec un processeur du marché de l'électronique embarquée. La consommation énergétique par rapport à ce même processeur est elle réduite d'un ordre de grandeur. Cette contribution a été valorisée à travers une publication scientifique dans le cadre de la conférence ISCAS [61].

Bien que ces résultats soient prometteurs, plusieurs limites concernant l'utilisation des outils et de la méthodologie de conception des processeurs XTensa ont été identifiées. Tout d'abord, le fait de ne pas disposer d'une licence complète empêche la génération du modèle matériel du processeur. Il est donc impossible de générer des résultats exhaustifs de synthèse et d'implémentation. Pour cette raison, les résultats obtenus sont des estimations fournies par l'outil, avec une marge d'erreur importante. De plus, le langage de description matérielle est le langage TIE, spécifique à la suite logicielle de Tensilica. Ceci limite les possibilités de réutilisation des unités matérielles conçues. D'autre part, une analyse attentive du code assembleur généré et des résultats de simulations montrent qu'une part significative de la durée d'exécution est prise par les échanges de données entre la mémoire cache, les registres et les unités matérielles de calcul. La conception et l'ajout des instructions *simple-registre* permettent de diminuer le nombre de ces échanges. Mais cette amélioration est partielle et ne peut pas être appliquée sur l'ensemble de l'arbre de décodage.

Plusieurs outils alternatifs de conception d'ASIP ont été envisagés afin de résoudre ces problèmes. Parmi eux, la famille d'architectures « TTA » a été sélectionnée pour créer le processeur présenté dans le chapitre suivant. Une des raisons de ce choix est la possibilité de contourner les registres et d'échanger les données directement entre la mémoire de données et les unités matérielles.

CHAPITRE 5 CONCEPTION D'UN PROCESSEUR PAR LA DESCRIPTION DE SON ARCHITECTURE

Une nouvelle méthodologie est employée afin de décrire une architecture de processeur efficace pour le décodage de codes polaires. Elle fait partie de la famille des architectures TTA. La section 5.1 de ce chapitre présente la structure générale et les propriétés des processeurs TTA. Une des caractéristiques principales de ces architectures est leur haut degré de parallélisme. Les processeurs TTA se rapprochent des architectures VLIW. Celles-ci sont donc introduites afin de servir de point de comparaison. Un exemple d'architecture TTA est ensuite donné afin d'en illustrer son fonctionnement. La suite logicielle libre utilisée pour concevoir le processeur est nommée TCE. Ces outils permettent l'automatisation d'une grande partie de la conception des processeurs TTA. La seconde partie de la section illustre et décrit le flot de conception et le rôle des différents outils. La section 5.2 détaille la conception de deux processeurs TTA à haute performance pour le décodage de codes polaires. La première architecture supporte l'algorithme SC et la seconde supporte à la fois les algorithmes SC et SCAN. Les expérimentations et les mesures réalisées sur ces deux architectures sont présentées dans la section 5.3.1.

5.1 Transport Triggered Architectures

5.1.1 Principes et caractéristiques des processeurs TTA

Dans le chapitre précédent, l'architecture d'un processeur RISC a été modifiée et son jeu d'instructions étendu afin d'obtenir un ASIP dédié au décodage des codes polaires. Dans ce chapitre, une approche différente est envisagée, qui consiste à définir un processeur de type TTA (Transport Triggered Architecture). Les architectures de type TTA se rapprochent des architectures à très long mot d'instructions (VLIW : Very Long Instruction Word) avec néanmoins une plus grande modularité. Il est ainsi possible de rajouter des unités fonctionnelles personnalisées et de définir précisément la manière dont celles-ci sont interconnectées. Le concepteur de l'architecture est amené à sélectionner chaque unité matérielle et à spécifier précisément leur modèle matériel. La structure et les performances de l'architecture obtenue se rapprochent de celles des architectures matérielles dédiées tout en conservant la programmabilité d'un processeur généraliste. En effet, les fonctionnalités d'un processeur généraliste sont conservées. De plus, les architectures TTA bénéficient d'un écosystème d'outils qui facilitent la conception et le développement de ce type de processeurs. Le programme devant

s'exécuter sur l'architecture est décrit à l'aide d'un langage de programmation haut niveau (C, C++, OpenCL). Des outils de profilage et de débogage sont disponibles et une grande partie du modèle matériel est généré automatiquement. Dans ce chapitre, nous proposons donc une architecture de type TTA pour le décodage de codes polaires.

Dans une architecture de processeur, nous distinguons habituellement deux types de parallélisme. Le premier est le parallélisme de données. Pour exploiter ce parallélisme, les jeux d'instructions de certains processeurs incluent des instructions vectorielles SIMD. C'est le cas des architectures ARM ou x86 actuelles qui implémentent respectivement les jeux d'instructions NEON et AVX utilisés dans le chapitre 3. Les instructions spécialisées de l'ASIP proposé dans le chapitre 4 sont également des instructions SIMD. Ces instructions permettent d'appliquer parallèlement une même opération sur plusieurs données.

Le second type de parallélisme est le parallélisme d'instructions. Contrairement au parallélisme de données, il s'agit d'exécuter des opérations différentes sur plusieurs données d'entrées. Par exemple, effectuer une somme de deux données et, en parallèle, effectuer une opération *ou-exclusif* sur deux autres. Il existe plusieurs façons de concevoir un processeur permettant du parallélisme d'instructions. Le compilateur peut selon les cas être impliqué dans la détection et l'exploitation du parallélisme d'instructions.

Dans les architectures superscalaires, le compilateur n'est pas impliqué dans la gestion du parallélisme d'instructions. Celui-ci est détecté par des unités matérielles spécialisées. Elles permettent de lancer l'exécution en parallèle de plusieurs opérations indépendantes sur les multiples unités fonctionnelles du processeur. Pour cela, les unités ont la capacité d'analyser les dépendances entre les données, de changer dynamiquement l'ordre d'exécution des instructions, ainsi que de spéculer sur les futures instructions du programme à exécuter. Un des avantages de ce type de processeurs est que des programmes séquentiels d'architectures plus anciennes peuvent être accélérés sans nouvelle compilation. Le principal désavantage est une complexité accrue au niveau de l'architecture processeur causée par l'ajout des unités matérielles en charge de la mise en œuvre du parallélisme. Enfin, la surface du circuit résultant augmente, ainsi que sa consommation énergétique [39].

Au contraire, dans les architectures VLIW, l'essentiel de l'effort nécessaire à la mise en œuvre du parallélisme d'instructions est pris en charge par le compilateur. Le compilateur décrit quelles instructions doivent être exécutées en parallèle, et dans quel ordre. L'avantage des architectures VLIW par rapport aux architectures superscalaires est la réduction de la complexité de la logique de contrôle. De plus, les possibilités de parallélisme d'instructions sont plus faciles à identifier par les compilateurs qui ont une vision plus large du programme que les unités matérielles des architectures superscalaires.

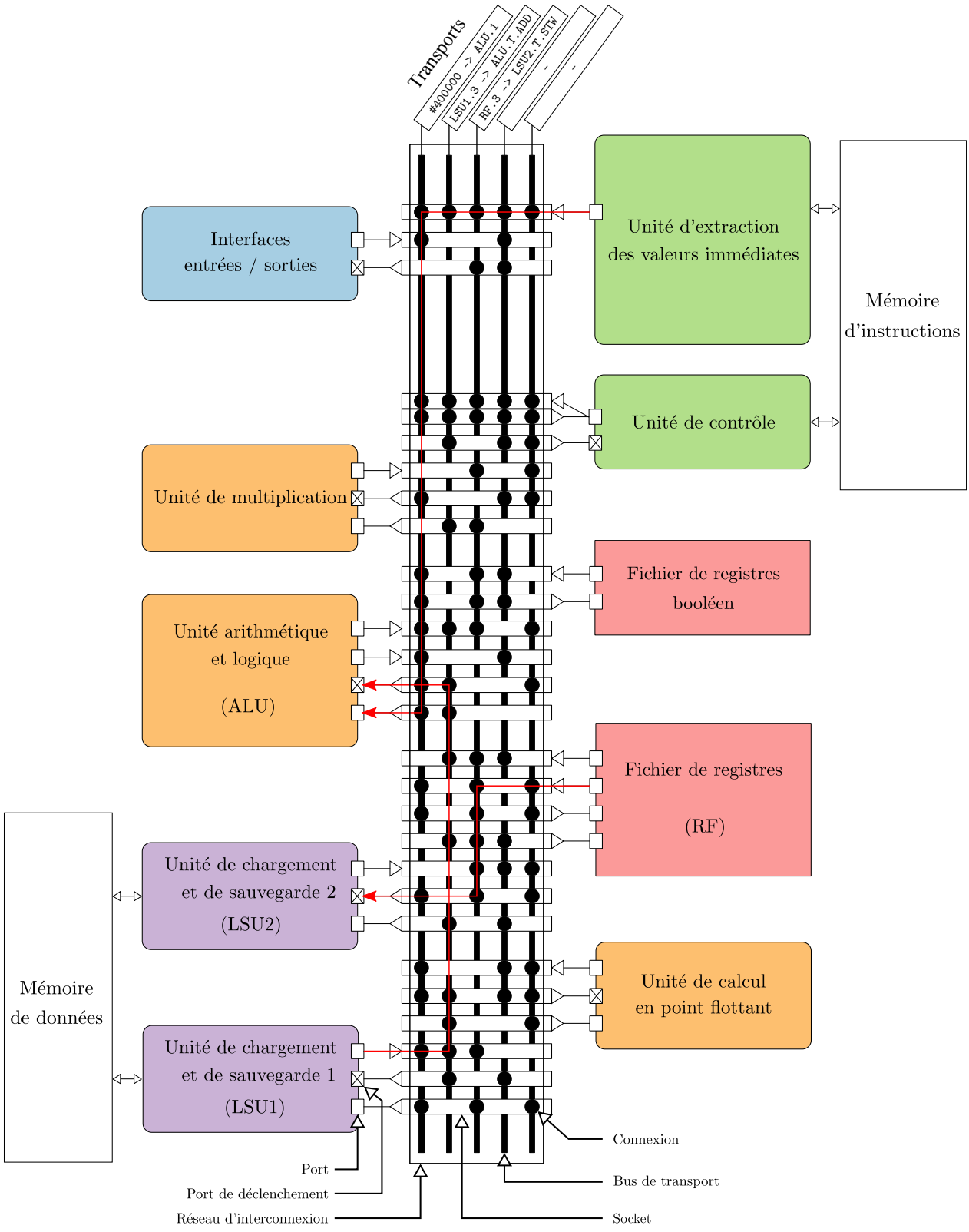


Figure 5.1 Schéma illustrant les architectures de processeur TTA.

L'architecture de processeur proposée dans ce chapitre fait partie de la famille des architectures déclenchées par le transport (TTA : Transport Triggered Architecture). Les TTA sont des architectures modulaires particulières et proches des architectures VLIW [40]. Les TTAs sont inspirées des architectures MOVE [41]. La Figure 5.1 présente un exemple d'architecture TTA [42]. Au centre, nous trouvons les *bus de transport* sur lesquels les données transitent. À ces bus de transport sont connectées les *unités fonctionnelles*, par l'intermédiaire de *sockets* et de *ports*. Les *ports* font partie des *unités fonctionnelles* : ils correspondent à des interfaces avec le monde extérieur. Une *socket* est associée à chaque port. Elle n'est pas forcément connectée à tous les bus de transport. Lors de la définition du processeur, c'est au concepteur de décider quels *bus de transport* doivent être connectés à une *socket*. Le nombre de *connexions* doit être soigneusement dimensionné afin de limiter la complexité du *réseau d'interconnexion*.

La différence principale séparant les architectures VLIW classiques et les architectures TTA sont les instructions utilisées. Dans les architectures VLIW, les instructions correspondent aux opérations à effectuer durant chaque cycle d'instruction. Des registres d'entrée et de sortie peuvent être spécifiés selon le type d'instruction. En revanche, un programme exécuté par un processeur d'architecture TTA contient une séquence de transports de données, depuis un port d'une unité fonctionnelle vers un autre. Les opérations sont exécutées lorsqu'une donnée est transmise vers un port particulier d'une unité fonctionnelle, nommé *port de déclenchement*. Chaque unité fonctionnelle possède un seul port de déclenchement.

Dans le schéma de la Figure 5.1, trois transports sont effectués sur les trois bus de gauche (flèches rouges), tandis que les deux bus de droite sont inutilisés. Les instructions correspondantes sont spécifiées au dessus du réseau d'interconnexion (`#400000 -> ALU.1`, `LSU1.3 -> ALU.T.ADD`, `RF.3 -> LSU2.T.STW`). Le langage utilisé pour décrire les transports est le langage assembleur des architectures TTA, proche du contenu du programme compilé. Dans une instruction TTA, deux ports sont définis : le premier est la source et le second est la destination. Le port est quant à lui spécifié par le nom de son unité fonctionnelle et un identifiant. Lorsque le port de destination est un port de déclenchement (T : Trigger), l'opération déclenchée est également spécifiée. Par exemple, le deuxième transport a pour cible le port de déclenchement de l'unité arithmétique et logique (ALU : Arithmetical and Logical Unit). Il est nécessaire de préciser que l'opération d'addition (ADD) doit être déclenchée : `ALU.T.ADD`.

Augmenter le nombre d'unités fonctionnelles d'une architecture VLIW est problématique car il est nécessaire d'augmenter le nombre de ports d'écritures et de lecture des files de registres afin que les unités fonctionnelles puissent y accéder simultanément. Ceci a pour conséquence une augmentation de la complexité des files de registre et une possible augmentation du

chemin critique. La modularité des architectures TTA résout ce problème.

En effet, le réseau d'interconnexion et les chemins de données disponibles sont connus du programmeur et du compilateur. Nous parlons de « chemin de données exposé » (*exposed datapath*). Comme le programmeur définit la séquence des transports, et non la séquence des opérations à exécuter, il n'est pas nécessaire de dimensionner les ports des files de registre par rapport au pire cas du nombre d'unités fonctionnelles pouvant accéder simultanément à une file de registres. Le programmeur ne peut qu'affecter des transports de données à des bus disponibles.

De plus, dans les architectures VLIW, les mécanismes de dérivation des registres (*register bypass*) sont implémentés matériellement. Dans les architectures TTA, ils sont décrits par le programmeur. Cela permet de réduire l'engorgement des registres et donc la nécessité de ports supplémentaires sur chaque file de registres.

Ces différentes caractéristiques ont guidé notre choix dans la sélection des architectures TTA pour la conception d'architectures programmables de décodage de codes polaires. Elles résolvent en effet le problème identifié en conclusion du chapitre 4, à savoir le nombre d'échanges nécessaires entre les mémoires, les files de registres et les unités fonctionnelles. La modularité des architectures TTA doit nous permettre de définir finement un réseau d'interconnexion permettant des communications directes entre les mémoires et les unités fonctionnelles, mais également entre les unités fonctionnelles elles-même.

Une autre raison justifiant ce choix est la disponibilité d'une suite logicielle libre et mature nommée TCE permettant une conception efficace d'architectures TTA. La suite logicielle offre également un compilateur adaptatif qui permet l'écriture des programmes dans des langages de haut niveau (C / C++). De plus, un contact a été établi avec l'équipe à l'origine de cette suite logicielle, qui nous a apporté un soutien technique efficace au long du développement. Cette suite logicielle est décrite dans la sous-section suivante.

5.1.2 L'environnement TCE

5.1.2.1 Définition de l'architecture et de l'implémentation du processeur

L'environnement TCE (TTA-based Co-design Environment) est une suite d'outils logiciels permettant la description d'architectures TTA, la compilation de programmes exécutables sur ces architectures et la génération de modèles matériels synthétisables et implémentables [43]. Une version améliorée de l'outil, nommée TCEMC (TCE MultiCore), permet le support d'opérations SIMD [44]. Nous l'avons utilisée pour concevoir les décodeurs polaires. La Figure 5.2 présente l'ensemble des outils proposés, les fichiers intermédiaires générés et le flot

de conception.

Le premier outil utilisé est l'éditeur de modèle architectural (**prode**). Il s'agit d'une interface graphique qui facilite la définition de processeurs tels que celui présenté dans la Figure 5.1. Le fichier *.adf* contient le nombre de bus et la largeur de chaque bus, la liste des sockets et de leurs connexions avec les bus de transport et les ports des unités fonctionnelles. Ce fichier liste également les unités fonctionnelles.

Le fichier *.adf* récapitule également les opérations que chaque unité fonctionnelle est capable de réaliser. Chaque opération est décrite par son interface, contenue dans un fichier *.opp* et son comportement, défini par un fichier *.c* ou *.cpp*. L'éditeur de la base de données des opérations (**osed**) permet d'explorer et d'éditer les différentes opérations. Il est également en charge de compiler les modèles comportementaux. L'exécutable (*.opb*) est utilisé pour effectuer des simulations de l'opération, soit de manière individuelle, soit au sein du simulateur du programme complet. L'outil *prode*, quant à lui, associe les unités fonctionnelles aux opérations dans le fichier *.adf*.

Les modèles matériels des unités fonctionnelles sont décrits en langage VHDL ou Verilog. Un grand nombre de modèles matériels et d'opérations de base sont fournis par la suite logicielle TCE : unités de chargement et de sauvegarde, fichiers de registres, ALU, unités SIMD, interfaces d'entrée et de sortie, entre autres. Pour créer des unités fonctionnelles spécifiques, l'utilisateur doit décrire l'interface des opérations, leurs modèles comportementaux et les modèles matériels des unités fonctionnelles dans un langage de description matériel (VHDL ou Verilog). Ces modèles matériels sont réunis dans une base de données de modèles matériels (*.hdb*). Ils sont éditables et consultables à l'aide de l'outil *hdbeditor*. Le concepteur, à l'aide de la base de données, peut réutiliser des modèles matériels déjà conçus par lui même ou d'autres contributeurs.

Le deuxième fichier produit par **prode** est le fichier de définition de l'implémentation *.idf*. Il relie simplement chaque unité fonctionnelle définie dans le fichier *.adf* à son implémentation matérielle contenue dans la base de donnée des modèles matériels.

5.1.2.2 Simulation au niveau architectural.

Le compilateur (**tcecc**) utilise le fichier de description architecturale *.adf* et les modèles des opérations afin de compiler le programme. Ce programme est écrit dans un langage haut niveau (C, C++, OpenCL). Un effort particulier a été effectué par les développeurs de l'environnement TCE afin de rendre ce compilateur efficace. Il est basé sur le projet LLVM [45]. L'environnement TCE bénéficie à ce titre d'optimisations des premiers niveaux

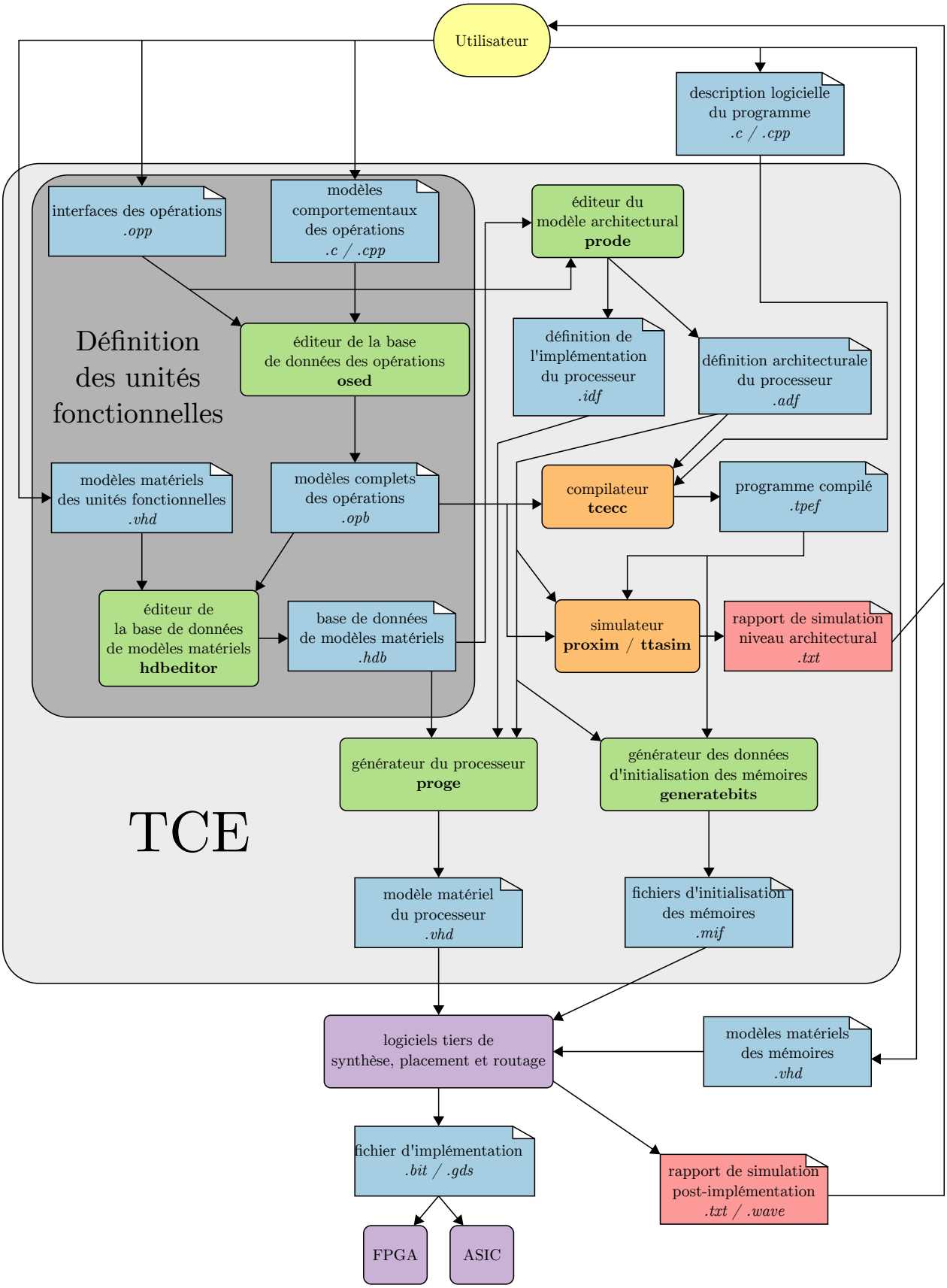


Figure 5.2 Organisation du flot de conception TCE.

de la chaîne de compilation (analyses lexicales, syntaxiques, sémantiques et génération du code intermédiaire). Les dernières étapes d'optimisations spécifiques aux TTAs (dérivation des files de registre, utilisation d'unités fonctionnelles et opérations SIMD) ont été réalisées par les développeurs de l'outil TCE.

Le programme compilé peut ensuite être simulé, soit à l'aide d'un outil en ligne de commandes (**ttasim**), soit par une interface graphique (**proxim**). Cette simulation permet d'obtenir le nombre de cycles nécessaires à l'exécution de l'intégralité ou d'une partie du programme, et ainsi d'avoir un premier retour sur l'adéquation entre l'architecture conçue et l'algorithme à exécuter. D'autres fonctions sont disponibles dans ce simulateur. Un profilage permet notamment à l'utilisateur de connaître le nombre de cycles d'exécutions pris par chaque fonction et ainsi de connaître les portions du programme à accélérer en priorité. Le simulateur donne également des métriques importantes, comme l'engorgement des bus, l'engorgement des sockets ou le taux d'utilisation de chaque opération dans les unités fonctionnelles. Cet ensemble de métriques permet au concepteur d'analyser très finement le fonctionnement du processeur conçu et le déroulement du programme.

5.1.2.3 Génération du processeur.

La dernière étape est la génération du modèle matériel complet du processeur. L'outil de génération du processeur (**proge**) utilise la base de données des modèles matériels des unités fonctionnelles (fichier *.hdb*) ainsi que les fichiers de description de l'architecture et de l'implémentation du processeur (fichiers *.adf* et *.idf*) afin de concevoir l'architecture complète associée au processeur. Elle intègre donc les différents modèles des unités fonctionnelles, le réseau d'interconnexion, mais également l'unité de contrôle, comme illustré dans la Figure 5.1. L'unité de contrôle a pour fonction la lecture du programme, stocké dans la mémoire d'instructions.

L'outil **generatebits** permet quant à lui de générer les contenus d'initialisation des mémoires de données et d'instructions. Les modèles matériels de ces mémoires doivent être fournis par l'utilisateur. En effet, ceux-ci sont en général dépendants de la cible d'implémentation.

L'architecture du processeur, décrite dans un langage de description matérielle (VHDL ou Verilog) et les modèles des mémoires peuvent ensuite être fournis à un logiciel tiers, pour effectuer la synthèse et l'implémentation sur FPGA ou sur ASIC. Les rapports de synthèse ou d'implémentation fournissent alors des métriques concernant la fréquence de fonctionnement, la surface utilisée et la consommation de puissance. Ces métriques peuvent être utilisées par le concepteur afin d'améliorer l'architecture du processeur. Des itérations du flot de conception permettent d'améliorer le processeur jusqu'à l'obtention de performances satisfaisantes.

5.2 Transport Triggered Polar Decoders

Deux architectures de processeurs spécialisées dans le décodage de codes polaires ont été conçues. Elles sont successivement détaillées dans cette section. La première est spécialisée pour le décodage SC. Elle est nommée `TT-SC`. Les algorithmes SC et SCAN sont tous deux supportés par la seconde architecture. Celle-ci est nommée `TT-SCAN`.

5.2.1 Architecture du décodeur `TT-SC`.

Comme illustré dans la Figure 5.3, l'architecture `TT-SC` se compose de trois parties principales. La *Base TTA* contient les unités fonctionnelles de base permettant de réaliser les fonctions d'un processeur généraliste. Elle est constituée d'une unité de chargement et de sauvegarde (LSU : Loading and Storing Unit), d'une ALU et d'une file de registres (RF : Register File). Elle contient également l'unité de contrôle global (GCU : Global Control Unit) qui est destinée à la lecture et au décodage des instructions. La seconde partie est constituée des LSU vectorielles. Elles réalisent les chargements et les sauvegardes des données depuis et vers les mémoires contenant les LLR et les sommes partielles. La troisième partie contient les unités de calcul (PU : Processing Unit), qui sont en charge de réaliser les fonctions élémentaires polaires.

Le niveau de parallélisme de données utilisé dans l'architecture TTA est $\mathsf{P} = 64$. Plus le parallélisme est élevé, plus les débits atteints sont importants. Deux raisons nous ont mené à ce choix de valeur. Premièrement, les architectures matérielles de la littérature utilisent ce niveau de parallélisme [3, 46–48]. Deuxièmement, l'objectif de fréquence de fonctionnement est de 100 MHz. Or, cette fréquence n'est pas atteignable pour un niveau de parallélisme supérieur ($\mathsf{P} = 128$). Diminuer le parallélisme en dessous de $\mathsf{P} = 64$ réduirait le débit de décodage d'une part et la complexité matérielle d'autre part. Ces compromis ne sont pas explorés dans ce manuscrit, priorité étant donnée au débit et à la latence.

Les algorithmes de décodages de codes polaires requièrent un grand nombre d'accès à la mémoire. Les architectures matérielles dédiées de la littérature utilisent en majorité deux mémoires séparées pour accéder aux LLR et deux autres pour les sommes partielles. Pour l'opération f, il est par exemple nécessaire d'effectuer deux lectures en mémoire ainsi qu'une écriture. Lorsque le parallélisme ($\mathsf{P} = 64$) est pris en compte, le total des accès est de 128 lectures pour 64 écritures. Les mêmes lectures et écritures sont nécessaires pour la fonction g, auxquelles il faut ajouter la lecture de 64 sommes partielles. Dans l'architecture `TT-SC`, nous avons choisi d'utiliser une mémoire double port pour stocker les LLR. Cela permet d'augmenter la bande passante, tout en conservant un seul espace de mémoire pour l'ensemble

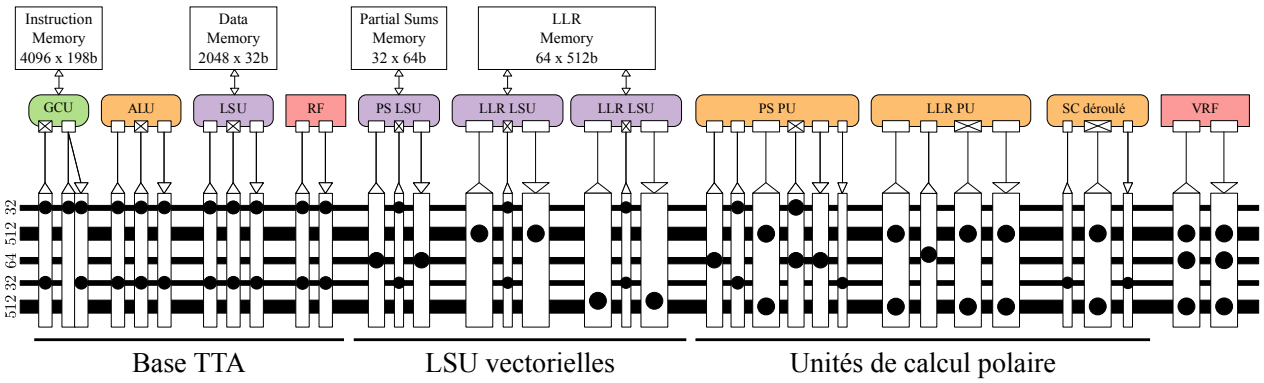


Figure 5.3 Architecture `TT-SC`.

des LLR. A chaque port est associé une LSU. Une LSU permet de lire ou d'écrire 64 données en mémoire par déclenchement.

Les trois unités de calcul sont l'unité de calcul de sommes partielles (PS PU), l'unité de calcul des LLR (LLR PU) et l'unité « SC déroulé ». La PS PU réalise les calculs des fonctions `R0`, `R1` et `h`. La LLR PU réalise les calculs des fonctions `f` et `g`. L'unité « SC déroulé » permet le décodage multi-cycles de sous-arbres. Elle sera détaillée par la suite.

Le nombre d'unités fonctionnelles choisies par le concepteur dans une architecture TTA fait l'objet d'un compromis. Tout d'abord, une unité fonctionnelle ne peut activer qu'une opération à la fois. Augmenter le nombre de FUs est donc nécessaire pour bénéficier du parallélisme d'instructions. Cela permet également d'augmenter la modularité de l'architecture du point de vue du concepteur. À fonctionnalité égale, utiliser un grand nombre de FUs permet de spécialiser chaque FU pour un sous-ensemble d'instructions. Cela simplifie leurs structures. Les modèles matériels sont donc plus simples à décrire, à modifier et à faire évoluer. Il est également plus probable de pouvoir réutiliser une FU simple dans de futures architectures.

En revanche, le nombre de ports augmente avec le nombre de FUs, ainsi que le nombre de connexions nécessaires sur le réseau d'interconnexion. Au niveau de l'implémentation, la densité du réseau d'interconnexion devient rapidement un problème. La congestion peut provoquer une augmentation du chemin critique et donc une diminution de la fréquence d'exécution. Au cours du développement de l'architecture `TT-SC`, il a été nécessaire de fusionner des FUs afin de réduire la congestion et d'augmenter la fréquence d'horloge. Ce fut obtenu au prix d'une réduction de la modularité et d'une augmentation de la complexité des FUs. La PS PU est issue par exemple de la fusion de trois FUs de versions antérieures de l'architecture.

Le réseau d'interconnexion développé possède deux bus de 512 bits pour le transport des LLR, un bus de 64 bits pour le transport des sommes partielles et deux bus de 32 bits pour le transport des données génériques et des adresses. Le choix du nombre de bus utilisés dans notre architecture détermine le parallélisme d'instructions disponible dans l'architecture TTA. Augmenter le nombre de bus de transports des LLRs permettrait d'augmenter les performances de l'architecture lors des exécutions des fonctions f et g, tandis que l'ajout de bus de 64 bits augmenterait les performances lors de la propagation des sommes partielles. Le nombre de cycles d'horloge nécessaire pourrait donc être réduit. Cependant l'augmentation du nombre de bus a un effet très négatif sur la fréquence de fonctionnement de l'architecture synthétisée. Il est impossible de conserver une fréquence de 100 MHz avec un réseau d'interconnexion trop dense.

Dans la sous-section suivante, la structure des FUs connectées à ces bus sont détaillées.

5.2.2 Unités fonctionnelles du décodeur `TT-SC`.

Nous détaillons ici la fonction et l'implémentation matérielle de certaines des FUs. Certaines (GCU, ALU, LSU, RF) sont liées au fonctionnement du base du TTA et à ce titre ne sont pas pertinentes dans le contexte de nos travaux. D'autres (LLR PU et PS PU) sont très similaires aux instructions spécialisées de l'architecture de processeur XTensa décrites dans la sous-section 4.2.3 du chapitre 4.

Les unités fonctionnelles originales spécifiques à l'architecture de processeur TTA sont les unités de chargement et de sauvegarde vectorielles d'une part, et l'unité « SC déroulé » d'autre part.

5.2.2.1 Unités de chargement et de sauvegarde.

Les unités de chargement et de calcul vectoriels permettent d'accéder aux mémoires contenant les LLR et les sommes partielles. Dans l'architecture proposée, la taille de code maximum supportée est $\mathsf{N}_{max} = 1024$. Il s'agit de la taille maximale définie dans le standard 5G [36]. Cette valeur peut être facilement ajustée en augmentant la profondeur des mémoires. Dans l'algorithme SC, N_{max} bits sont nécessaires pour stocker les sommes partielles. Puisqu'il est nécessaire d'accéder à $\mathsf{P} = 64$ sommes partielles en parallèle, une mémoire de 16 x 64-bits est suffisante. Cependant une taille supérieure de mémoire a dû être sélectionnée : 32x64-bits. En effet, une section d'adresse est automatiquement réservée par TCE. Pour des raisons similaires, les LLR sont stockés dans une mémoire de 64x512-bits.

Comme précisé auparavant, l'accès à la mémoire est un enjeu capital dans les architectures

de décodage de codes polaires. Les modèles architecturaux et matériels de LSU proposés par défaut dans les bases de données de TCE ont une latence de 3 cycles d'horloge, ce qui ralentit très fortement l'exécution du décodage. C'est pourquoi, cette latence a été réduite à un cycle d'horloge par la suppression de registres. Cette suppression n'a pas affecté la fréquence maximale de fonctionnement.

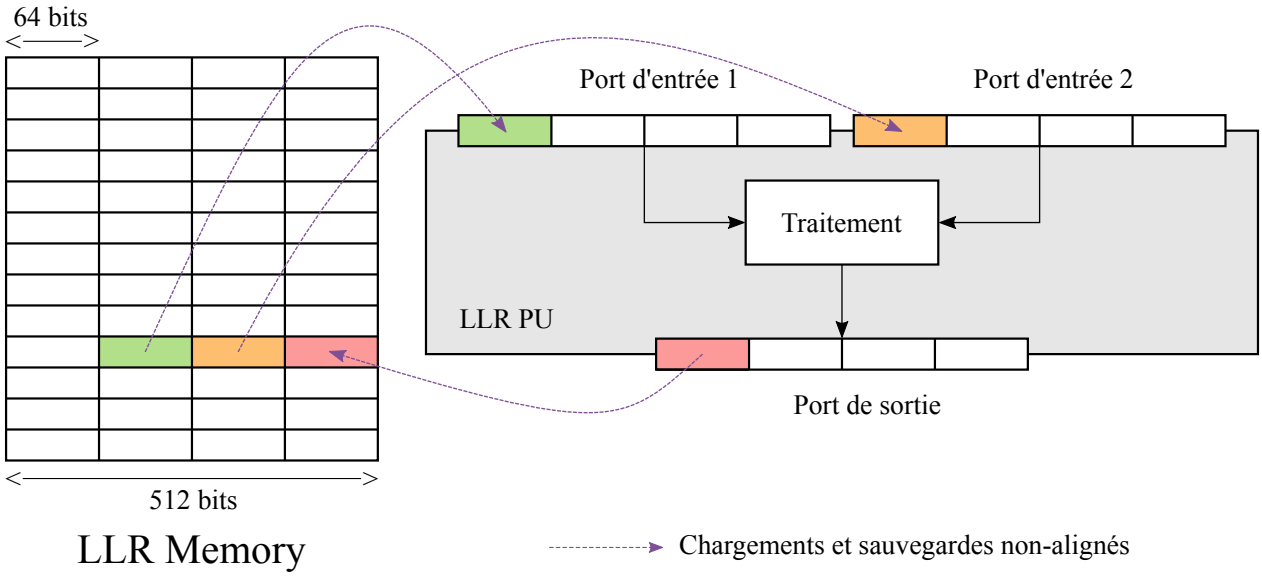


Figure 5.4 Illustration des chargements et sauvegardes non-alignés.

La seconde amélioration apportée à ces LSU est l'ajout d'unités matérielles et d'opérations d'alignement. En effet, pour effectuer parallèlement le traitement de certaines portions de l'arbre de décodage pour lequel le parallélisme est inférieur à P, il est nécessaire de lire et d'écrire des données non-alignées. Les données non-alignées sont des données dont l'adresse n'est pas un multiple de 512, dans le cas des LLR, ou de 64, dans le cas des sommes partielles. De plus, les mots en mémoire sont des mots de 512 bits. Dans les portions de l'arbre à parallélisme inférieur à P, il est nécessaire d'écrire des sous-mots de taille inférieure à 512 bits (jusqu'à 64 bits). Il est donc nécessaire que les mémoires gèrent ces écritures de sous-mots. Par ailleurs, les LSUs doivent être capables d'effectuer le contrôle de ces mémoires. Le processus de chargement et de sauvegarde de données non-alignées est illustré dans la Figure 5.4. Dans celle-ci, des sous-mots de 64 bits sont extraits de la mémoire et positionnés en première position des ports d'entrées de la LLR PU. Le traitement est alors réalisé sur les deux ports d'entrée, le résultat est stocké dans le premier sous-mot du port de sortie. Celui-ci est stocké dans la mémoire, dans le quatrième sous-mot de la ligne de mémoire considérée.

5.2.2.2 Décodage d'un sous-arbre déroulé multi-cycles.

Comme cela a été mis en évidence dans [49], le parallélisme disponible dans le décodage SC varie en fonction de la zone de l'arbre de décodage. Les zones proches du haut de l'arbre disposent d'un très fort parallélisme alors que le bas de l'arbre souffre d'un défaut de parallélisme. Considérons les sous-arbres du bas dont le nœud racine contient 8 LLR. Ce type de sous-arbre étant de taille réduite, il est possible de le dérouler complètement. Il est montré que ce déroulage permet de réduire le nombre de cycles nécessaires au traitement de ce type de sous-arbre. En effet, la durée de cette séquence de décodage avec une architecture semi-parallèle classique serait de 14 cycles. En déroulant l'arbre de décodage et en découpant le chemin combinatoire à l'aide de registres, la latence résultante n'est plus que de six cycles. Cette technique s'inspire de la technique [50]. Dans notre cas, le décodeur déroulé n'est pas pipeliné. Le but des registres est simplement de découper le chemin critique afin de garantir une fréquence de fonctionnement suffisante. Comme il n'y a pas de parallélisme temporel, les registres ne sont pas nécessaires. Le chemin de données qui parcourt le sous-arbre de décodage de bout en bout est un chemin multi-cycles.

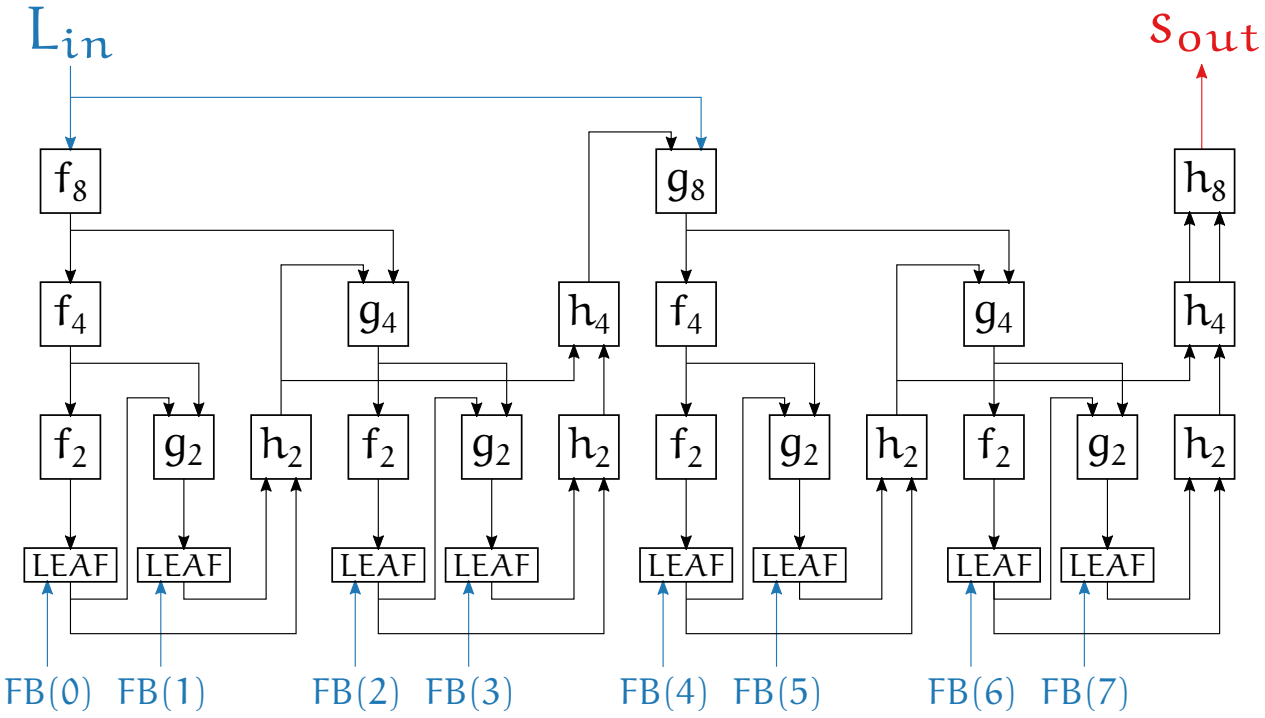


Figure 5.5 Unité matérielle de décodage d'un sous-arbre avec un traitement multi-cycles.

En effet, il est possible de définir des chemins de données multi-cycles dans les processeurs conçus à l'aide de TCE. Par défaut, la latence d'une opération effectuée dans une FU est de un

cycle d'horloge : à un front d'horloge, si une donnée est présentée au port de déclenchement, alors les résultats de l'opération sont disponibles aux ports de sortie au front d'horloge suivant. Dans l'outil **prode**, qui permet de définir l'architecture, il est toutefois possible d'assigner une latence supérieure à une ou plusieurs opérations d'une FU. Si une latence supérieure est spécifiée, alors les données de sortie ne seront disponibles qu'après le nombre spécifié de cycles d'horloge. Le compilateur a connaissance de cette latence spécifiée dans le fichier *.adf*. Il est ainsi en charge de régler les problèmes de disponibilité des données. Du côté du logiciel de synthèse tiers, il faut spécifier que les chemins de données de l'opération en question sont multi-cycles. Leur contrainte de temps est allongée de plusieurs cycles d'horloge. Ainsi, aucun registre n'est assigné et la complexité matérielle de l'unité n'est pas impactée. Cette technique permet de réduire le nombre de cycles nécessaires au décodage du sous-arbre par rapport à la version contenant les registres.

L'unité fonctionnelle « SC déroulé » correspond donc à un sous-arbre de décodage SC déroulé et multi-cycles. Cette unité est détaillée dans la Figure 5.5. Il est important de noter que les bits gelés sont donnés en entrée du sous-arbre. Cela permet d'être générique vis-à-vis du rendement et de la construction du code polaire.

5.2.3 Description logicielle ciblant l'architecture `TT-SC`.

La notion de déroulage de code source a été présentée dans la sous-section 3.2.2. Dérouler le code source permet de réduire le nombre de calculs d'adresse et le nombre d'indirections. Durant la conception de l'architecture `TT-SC`, les indirections sont devenues problématiques. Souvent, celles-ci causaient une sous-utilisation des différents bus. Le traitement d'une indirection nécessite en effet plusieurs cycles d'horloges. De manière récurrente, les bus de transports et les unités fonctionnelles étaient inactives durant ces cycles d'horloge. Le déroulage du code a permis de lever ce verrou. Cependant, le déroulage du code n'est pas sans conséquence. Lorsque l'arbre de décodage est élagué, un code source doit être compilé pour chaque construction du code.

Pour cela, la bibliothèque proposée dans [15] a été utilisée. L'intérêt de cette bibliothèque est de faciliter la génération de codes sources déroulés (en langage C++). Les cibles architecturales visées sont les processeurs x86 et ARM. Cette bibliothèque a donc été étendue afin de permettre la génération d'un code source exploitant les instructions TTA. Ainsi, il est possible de générer le code source déroulé et élagué de n'importe quelle construction de code polaire, quels que soient la taille et le rendement. Par ailleurs, de manière équivalente aux décodeurs logiciels à liste proposés dans le chapitre 3, l'élagage de l'arbre est ajustable, en fonction du type de nœuds utilisés ou de la taille des nœuds activés.

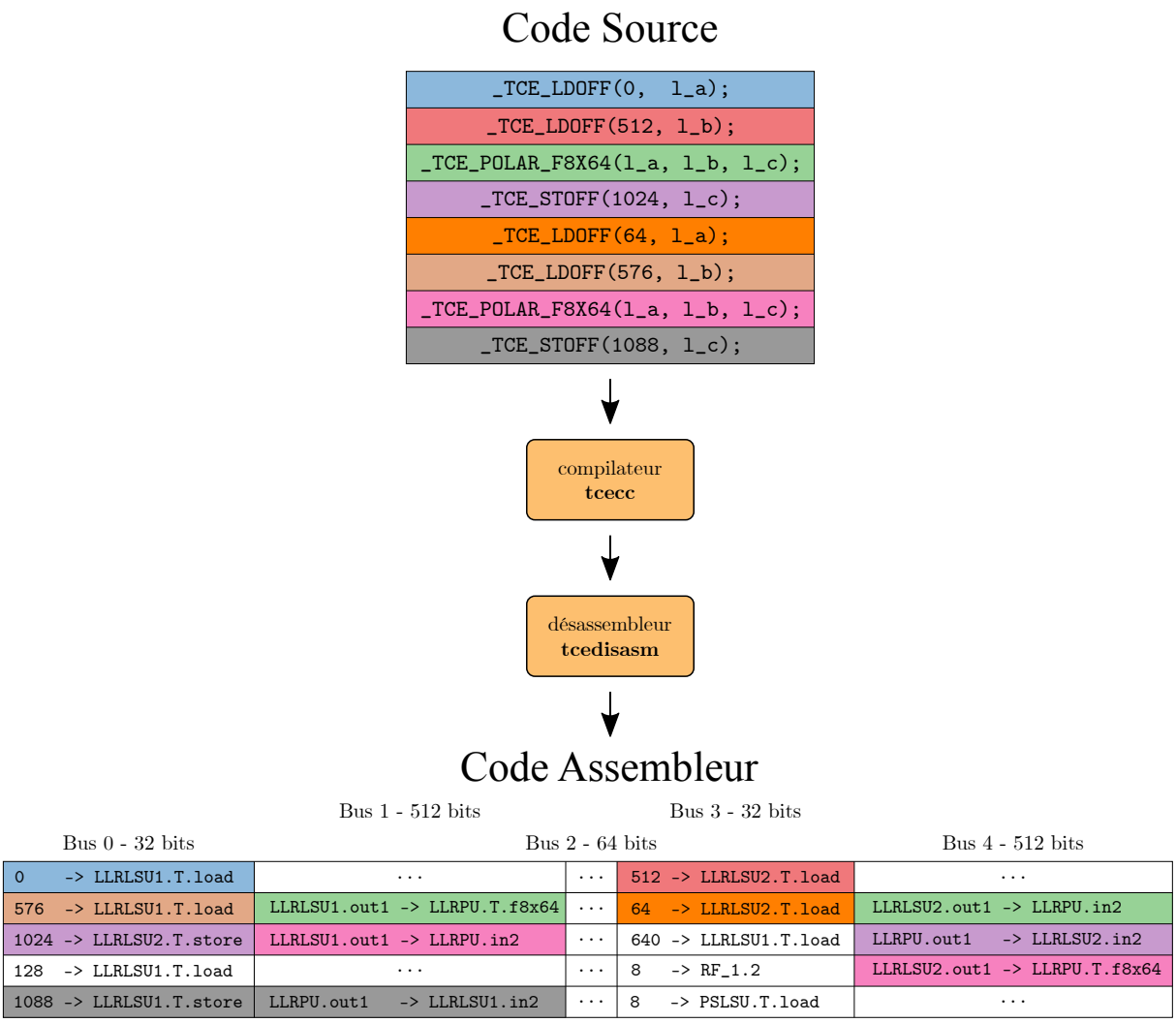


Figure 5.6 Exemple d'un code source et du code assembleur résultant.

La Figure 5.6 contient une illustration du code source. Il s'agit du traitement des premiers nœuds de l'arbre. Des données sont chargées depuis la mémoire (`_TCE_LDOFF`), la fonction f est appliquée (`_TCE_POLAR_F8X64`), puis le résultat est stocké dans la mémoire (`_TCE_STOFF`). Le désassembleur de TCE permet de visualiser le code assembleur correspondant au programme généré par le compilateur. Il apparaît clairement que grâce à la génération du code source déroulé, le compilateur exploite efficacement les possibilités de parallélisme d'instructions offertes par les processeurs TTA. Dans cet exemple, la moyenne du nombre d'unités fonctionnelles déclenchées par cycle d'horloge est de 2. Les bus 0 et 3 sont utilisés à 100%, les bus 1 et 4 sont utilisés à 60% et le bus 2 n'est pas utilisé. En effet, à ce stade de l'algorithme de décodage, aucune opération n'est appliquée sur les sommes partielles. Durant

le décodage d'un mot de code (1024,512), les bus sont occupés à 45% et le nombre moyen d'unités fonctionnelles actives par cycle d'horloge est de 1.4. En analysant finement le code source, il y a principalement deux cas de figure où les bus et les unités fonctionnelles sont peu occupés. Le premier correspond aux sections où les unités « SC déroulé » multi-cycles sont activées. Le second correspond à la propagation des sommes partielles dans l'arbre. En effet, un seul bus est disponible pour les sommes partielles. De plus, une seule LSU est allouée au stockage et au chargement des sommes partielles. Des améliorations de l'architecture seraient donc possibles afin d'augmenter le taux d'utilisation des bus et des unités fonctionnelles. Cela permettrait de réduire le nombre de cycle d'horloges nécessaires au décodage d'un mot de code.

Une illustration de l'exécution d'un programme par le processeur `TT-SC` est donnée dans la Figure 5.7. Les transports de données sont explicités au cours des cinq cycles d'instructions.

5.2.4 Implémentation de l'algorithme SCAN

Afin d'illustrer la modularité et l'évolutivité des processeurs TTA conçus à l'aide de TCE, nous avons développé une seconde architecture nommée `TT-SCAN`. Elle permet le support de l'algorithme SCAN en plus de l'algorithme SC. Cette dernière est décrite dans la Figure 5.8. Comme détaillé dans la sous-section 2.2.5, les sommes partielles sont remplacées par des LLR dans l'algorithme SCAN. Ces LLR sont notés BLLR (Backward LLR). La première unité fonctionnelle ajoutée à l'architecture `TT-SC` pour obtenir l'architecture `TT-SCAN` est l'unité de chargement et de sauvegarde des BLLR. Une mémoire de stockage pour les BLLR a également été ajoutée. La seconde unité fonctionnelle ajoutée est une unité « SCAN déroulé ». Cette unité permet d'accélérer le traitement des niveaux inférieurs de l'arbre de décodage, de manière analogue à l'unité « SC déroulé » pour l'algorithme SC.

De plus, l'unité de calcul des fonctions polaires LLR PU a été adaptée. Les fonctions élémentaires nécessaires au décodage de l'algorithme SCAN sont :

$$\begin{aligned} f_{scan}(L_a,L_b,L_c) &= f(L_a,L_b+L_c) \\ g_{scan}(L_a,L_b,L_c) &= f(L_a,L_c)+L_b \end{aligned} \tag{5.1}$$

Il est possible d'adapter l'unité fonctionnelle LLR PU utilisée dans l'architecture `TT-SC` pour qu'elle puisse supporter ces nouvelles fonctions polaires. L'organisation de l'unité matérielle adaptée est illustrée dans la Figure 5.9. Il est à noter que des multiplexeurs supplémentaires sont nécessaires pour contrôler le paramétrage. Le nombre de cellules logiques nécessaires pour réaliser l'unité fonctionnelle « SC+SCAN » est de 133 sur circuit FPGA. Seules 83 cellules

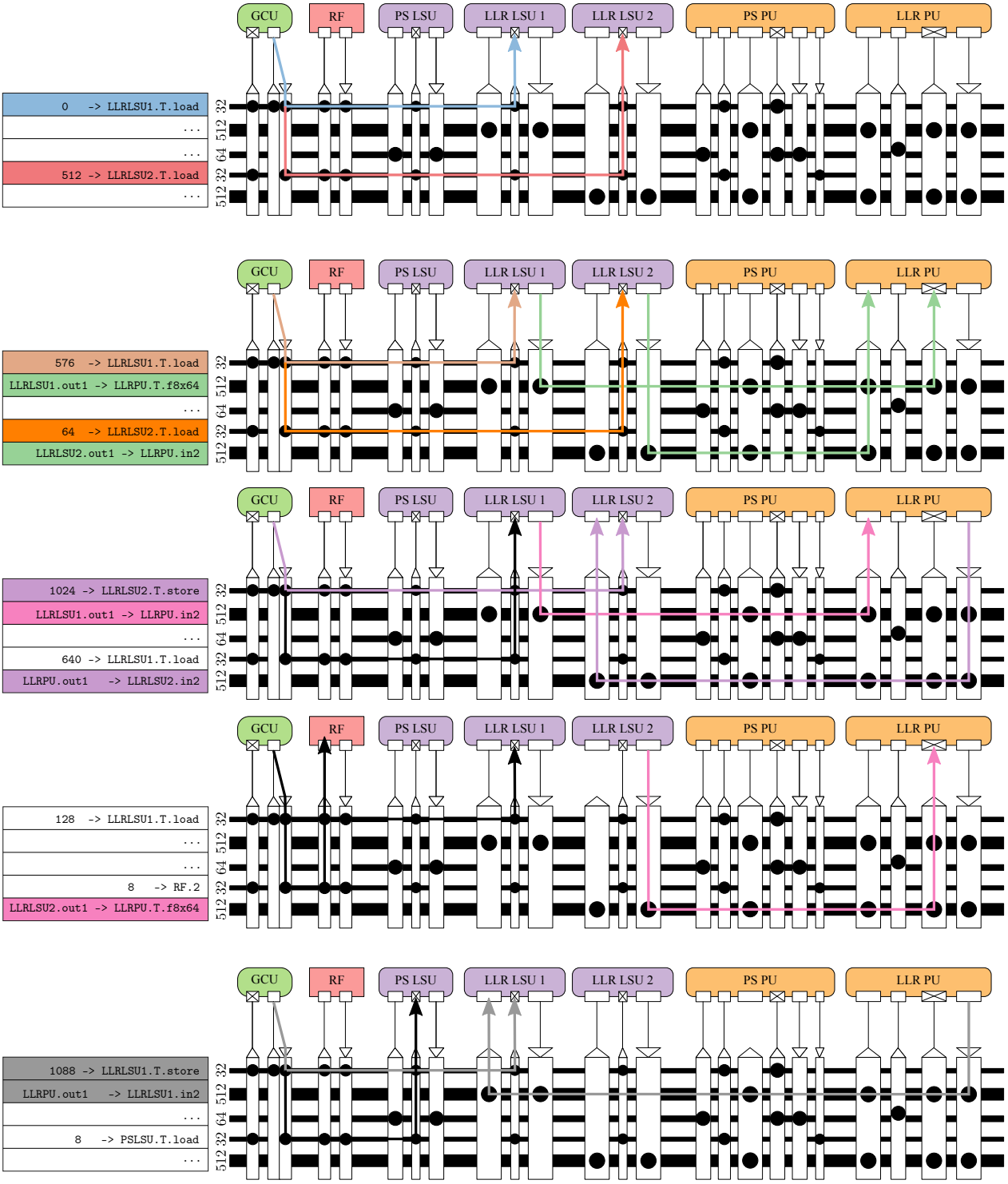


Figure 5.7 Illustration du parallélisme d'instructions sur le processeur `TT-SC`.

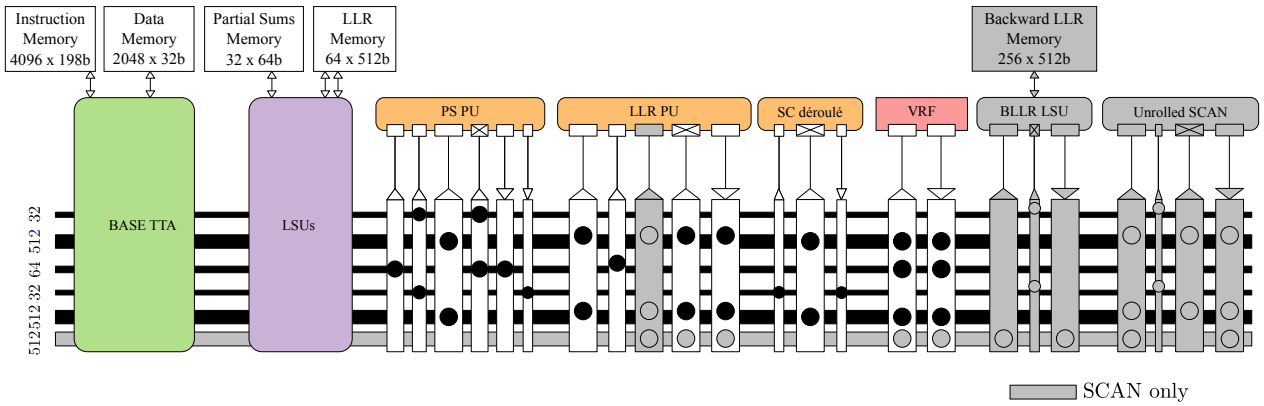


Figure 5.8 Architecture `TT-SCAN`.

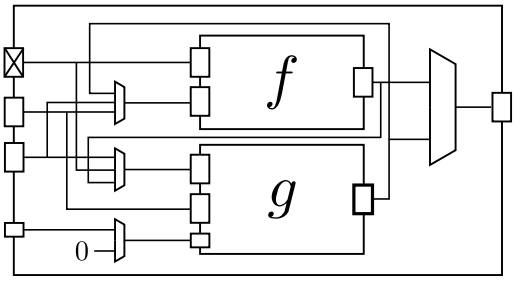


Figure 5.9 Organisation de l'unité « LLR PU » réalisant les fonctions élémentaires SC et SCAN.

logiques sont nécessaires pour l'unité fonctionnelle « SC ». Cela implique une augmentation de 60 %.

5.2.5 Généricité et Flexibilité.

Les termes de généricité et de flexibilité d'un décodeur ont été définis dans la section 3.3. Les décodeurs TTA de codes polaires sont tout d'abord générique du point de vue de la taille du code, à condition de dimensionner correctement les mémoires. Nous avons par exemple dimensionné nos mémoires pour supporter des tailles de codes allant jusqu'à $\mathsf{N}_{max} = 1024$. Dans ce cas, n'importe quelle taille de code égale à une puissance de deux peut être gérée par le décodeur. Le code source étant déroulé, une version du code doit être générée pour gérer un nombre de bits d'informations données. Il s'agit là du défaut de la technique de déroulage du code présentée dans la section 3.2.2.

Du point de vue de la flexibilité, l'élagage de l'arbre de décodage peut être ajusté grâce à la bibliothèque de génération du code source utilisée comme décrit dans la section 5.2.3. Les variantes algorithmiques utilisées sont pour l'instant l'algorithme SC et l'algorithme SCAN. La quantification est quant à elle fixée à 8 bits. Des degrés de flexibilité pourraient être ajoutés, cela nécessiterait du temps de développement supplémentaire. Par exemple, l'algorithme SCF pourrait être rapidement supporté à condition d'ajouter une FU spécialisée dans le calcul de CRC.

De plus, rappelons que les architectures TTA proposées comprennent toutes des unités de chargement sauvegarde et une ALU génériques. À ce titre, toute variante algorithmique pourrait être décrite logiciellement et exécutée par l'architecture. Suivant la complexité des modifications, les performances de débits et de latence pourraient alors être dégradées.

5.3 Expérimentations et mesures.

Le but de cette section est de présenter les mesures et les expérimentations réalisées afin d'évaluer les deux architectures conçues : `TT-SC` et `TT-SCAN`. Il est important de souligner que les comparaisons avec les décodeurs de la littérature sont délicates. En effet, beaucoup de paramètres changent, tels que la technologie d'ASIC, la tension d'alimentation, les algorithmes supportés par les différentes architectures, la stratégie d'élagage, le format de représentation des LLR, la construction du code polaire. Ces précautions prises, nous allons montrer que les architectures TTAs proposées offrent un compromis pertinent entre les architectures de processeur à usage général et les architectures matérielles dédiées. Nous allons montrer également que ces architectures permettent une amélioration significative du débit et une

réduction de la consommation énergétique par rapport au processeur XTensa détaillée tout au long du chapitre 4.

5.3.1 Architecture `TT-SC`.

L'architecture `TT-SC` a été synthétisée avec la bibliothèque de cellules standard ST 28nm FD-SOI, avec les paramètres 0.9V, 125°C [51]. Les mémoires sont des blocs de SRAM de la même technologie. Le Tableau 5.1 présente les performances de débit, latence et consommation de l'architecture `TT-SC` exécutant l'algorithme SC. Une comparaison est faite avec les performances obtenues sur des processeurs à usage général d'architectures x86 et ARM, ainsi qu'avec celles obtenues sur le processeur XTensa du chapitre 4.

Pour les deux processeurs à usage général, le code source utilisé est généré à l'aide de la bibliothèque introduite dans [15]. Les simulations sont effectuées grâce à la suite logicielle AFF3CT [28]. Le processeur Intel i7-4712HQ présente un débit élevé mais induit une importante consommation énergétique. Le débit mesuré sur le processeur ARM A57 est moins élevé, mais la consommation énergétique est plus faible que celle obtenue sur le processeur Intel. Comme nous l'avons vu dans le chapitre précédent, le débit obtenu sur le processeur spécialisé XTensa est comparable à celui obtenu sur l'architecture ARM, alors que la consommation énergétique est dix fois plus faible. L'architecture `TT-SC` permet d'atteindre un débit plus élevé que les trois autres architectures, tout en réduisant d'un ordre de grandeur la consommation énergétique en comparaison du processeur XTensa. Par exemple, avec un code polaire (1024,512), le débit obtenu par le processeur `TT-SC` est de 352 Mb/s, ce qui est supérieur de 37 % au débit obtenu avec le processeur Intel i7-4712HQ, alors que la consommation énergétique est beaucoup plus faible, avec deux ordres de grandeur de différence. Nous en concluons que l'approche développée dans ce chapitre est pertinente et aboutit à des résultats convaincants.

Les performances de l'architecture `TT-SC` se rapprochent des performances des architectures matérielles dédiées de l'état de l'art. Le Tableau 5.2 présente les performances de plusieurs implémentations de l'algorithme SC sur des cibles FPGA. Les deux références [3, 48] sont des implémentations de la version élaguée de l'algorithme (FAST SC). L'élagage est complet, c'est-à-dire que tous les nœuds spécialisés sont utilisés, en particulier les nœuds `REP` et `SPC`, contrairement à l'architecture `TT-SC` dans laquelle seuls les nœuds `R1` et `R0` sont considérés. Le nombre de cycles d'horloge nécessaire au décodage d'un mot de code est 5 fois plus faible pour les architectures dédiées que pour l'architecture `TT-SC`. Trois facteurs principaux expliquent cette différence. Premièrement, comme il vient d'être expliqué, l'élagage est plus complet dans les architectures dédiées. Cet aspect pourrait être corrigé dans des versions améliorées

Tableau 5.1 Comparaison de l'architecture TT-SC avec des processeurs à usage général et l'ASIP XTensa proposé dans le chapitre 4.

Architecture	N	Latence [µs]	Débit [Mb/s]	E_b [nJ/bit]
i7-3.3GHz (GPP)	1024	2.0	257	41
	512	1.2	210	49
	256	0.7	179	59
	128	0.4	143	73
A57-1.1GHz (GPP)	1024	10.7	48	17
	512	5.3	48	17
	256	2.8	46	17
	128	1.6	41	20
LX7-835MHz (ASIP)	1024	7.2	71	1.6
	512	3.9	66	1.7
	256	1.9	65	1.7
	128	1.0	62	1.8
TT-SC-800MHz (ASIP)	1024	1.4	352	0.14
	512	0.8	313	0.15
	256	0.4	304	0.16
	128	0.2	284	0.17

de l'architecture TT-SC. Deuxièmement, dans l'architecture TT-SC, le chemin constitué d'une lecture en mémoire, d'une opération élémentaire polaire et d'une écriture en mémoire prend trois cycles d'horloge. Dans les architectures dédiées, ces trois opérations sont réalisées en un seul cycle d'horloge. Le compilateur utilise le parallélisme d'instructions pour limiter cette différence, mais ceci n'est pas toujours possible durant le déroulement de l'algorithme de décodage. Troisièmement, comme indiqué dans la sous-section 4.2.2, la propagation des sommes partielles se réalise en parallèle des autres opérations dans les architectures matérielles. Dans l'architecture TT-SC, la propagation des sommes partielles représente environ 20% du temps nécessaire au décodage d'une trame. Cela impacte donc le coût au niveau du nombre de cycles d'horloge.

Le nombre de LUT utilisées par l'architecture TT-SC est inférieur à celui des architectures dédiées. La principale raison est que l'architecture TT-SC ne gère qu'un élagage limité contrairement aux architectures matérielles dédiées considérées. Or, la gestion de l'élagage implique des unités de traitement spécifiques coûteuses en ressources calculatoires. Néanmoins, l'architecture TT-SC est une architecture de processeur programmable complète, avec une ALU généraliste. La présence de cette ALU généraliste et d'une LSU associée permet à l'architec-

Tableau 5.2 Implémentations sur cible FPGA de décodeurs SC pour un code polaire (1024,512).

	TT-SC		[48]	[3]	
Cible	Artix-7	Stratix IV	Stratix IV	Stratix IV	Virtex 6
Cycles d'horloge	1161		222	165	165
Débit info. (Mb/s)	44	39	238	319	217
Fréquence (MHz)	100	90	103	103	70
LUTS	14744	19665	23020	24821	22115
FFs	7354	8359	1024	5823	7941
RAM (Kb)	948		43	36	36
Élagage	R0 & R1		Complet	Complet	Complet

ture TTA de pouvoir potentiellement réaliser n'importe quel algorithme décrit en langage de haut niveau. Si ce nombre réduit de LUT est un fait notable, la versatilité de l'architecture TT-SC a néanmoins un coût. En effet, le nombre de bascules flip-flop utilisées est supérieur. Beaucoup de registres sont nécessaires dans les architectures TTA, dans chaque unité fonctionnelle. De plus, les instructions utilisées par le processeur TT-SC sont très longues : une instruction est stockée sur 198 bits. L'empreinte mémoire de la mémoire d'instructions est donc considérable. Cet élément apparaît dans le Tableau 5.2 en ce qui concerne la quantité de mémoire RAM utilisée. Dans l'architecture proposée, la taille de code polaire maximum est $\mathsf{N}_{max} = 1024$. N_{max} bits sont nécessaires pour le stockage des sommes partielles. Comme il est nécessaire d'accéder à $\mathsf{P} = 64$ données en parallèle, une mémoire 16x64-bit est suffisante. Cependant, une zone de mémoire étant réservée par le compilateur, la taille de mémoire supérieure a été sélectionnée : 32x64-bit. Pour des raisons similaires, une mémoire 64x512-bit est utilisée pour les LLR. De plus, il s'agit d'une mémoire double port, son empreinte mémoire est donc doublée. La mémoire de données de 32 bits est une mémoire 2048x32-bit. Au total, l'empreinte mémoire, comprenant également les différents registres vectoriels et scalaires, est de 948 kbits.

85 % de cette empreinte mémoire est due à la mémoire d'instructions. Nous avons donc étudié des moyens de réduire cette empreinte. Les programmes de décodage utilisent un faible nombre d'instructions différentes. Une optimisation consiste alors à tenter de réduire la taille des instructions à 11 bits seulement en ajoutant un décodeur d'instructions qui permet de convertir des instructions de 11 bits en instructions de 198 bits. La complexité de ce convertisseur est faible, il représente moins de 5% de la surface totale de l'ASIP. Fonctionnellement, l'architecture TT-SC modifiée devient alors équivalente à celle des architectures dédiées. Cette architecture modifiée a été synthétisée dans la technologie ST précédemment présentée. Les

Tableau 5.3 Implémentations sur cible ASIC de décodeurs SC pour un code polaire (1024,512).

	TT-SC-COMP	[46]	[47]	[47][1]
Cible	28nm	28nm	180nm	28nm
Cycles d'horloge	1161	1833	1568	1568
Débit info. [Mb/s]	352	94	49	436
Fréquence [MHz]	800	336	150	1335
Puissance [mW]	48	18	67	5
$\mathbf{E_b}$ [nJ/bit]	0.14	0.19	1.4	0.011
Surface [mm^2]	0.16	0.44	1.71	0.04
Élagage	R0 & R1	First R0	None	None

[1] Les facteurs de mise à l'échelle de 180nm vers 28nm de [47] sont issus de [46].

résultats sont présentés dans le Tableau 5.3. Cette nouvelle version de l'architecture est nommée TT-SC-COMP (pour « compressé »). Ses caractéristiques sont comparées avec des résultats de décodeurs ASIC de la littérature. La comparaison est toutefois difficile compte tenu de la diversité des paramètres d'implémentations. En effet, le décodeur SC proposé dans [46] constitue la base d'un décodeur SCL. Sa puissance et sa surface pourraient donc être réduites si seulement l'algorithme SC était supporté. De plus, une stratégie simpliste d'élagage est utilisée. Ceci explique le débit plus faible que celui obtenu avec l'architecture TT-SC-COMP. Le décodeur SC implémenté dans [47] utilise une technologie ASIC différente et aucun élagage n'est utilisé. Aussi, l'objet du Tableau 5.3 est simplement de montrer que l'architecture TT-SC-COMP présente une complexité matérielle et une consommation énergétique raisonnable lorsqu'elle est comparée à des implémentations similaires. Elle permet d'atteindre plusieurs centaines de Mb/s tout en conservant la flexibilité d'un processeur programmable.

5.3.2 Architecture TT-SCAN.

Les résultats de l'expérimentation de l'architecture TT-SCAN sont présentés dans le Tableau 5.4. La seule implémentation sur circuit FPGA de la littérature est proposée dans [52]. L'architecture TT-SCAN est très proche de celle-ci en terme de nombre de cycles d'horloge, de fréquence et de débit. Elle est toutefois plus complexe au niveau des ressources matérielles, comme en atteste les nombres de LUT et de portes flip-flop utilisées. Cette complexité est due au fait que l'arbre de décodage est élagué, ainsi qu'au niveau de parallélisme qui vaut $\mathsf{P} = 16$ dans [52], contre $\mathsf{P} = 64$ dans l'architecture TT-SCAN. Du point de vue de l'empreinte mémoire, la différence entre l'architecture TT-SC et l'architecture TT-SCAN est due à l'ajout de la mémoire de stockage des BLLR : 256x512-bit.

Une implémentation ASIC de l'algorithme SCAN est proposée dans [53]. Comme pour l'archi-

tecture TT-SCAN, l'arbre de décodage est élagué. Dans cette implémentation, le parallélisme est cette fois P = 64. Dans ce cas, nous constatons que le rapport du nombre de cycles est d'environ 4, ce qui se rapproche de la différence observée entre l'architecture TT-SC et les architectures dédiées au décodage de l'algorithme SC.

5.4 Synthèse

Dans ce chapitre, deux architectures de processeurs à haute performance pour le décodage des codes polaires SC et SCAN sont successivement présentées. Pour ce faire, le modèle architectural TTA est décrit ainsi que la suite logicielle TCE qui propose un flot de conception complet. TCE permet la génération du modèle matériel du processeur, ainsi que la compilation de programmes décrits dans des langages de haut niveau (C / C++). Cette suite logicielle est utilisée afin de concevoir les deux architectures. Plusieurs contributions originales se dégagent de ce travail.

(i) La première architecture de processeur de type TTA spécialisée dans le décodage de codes polaires est proposée.

(ii) Une unité matérielle de décodeur SC déroulé et multi-cycles est introduite pour décoder des parties de l'arbre de décodage.

(iii) Son équivalent est défini pour l'algorithme de décodage SCAN.

(iv) Une unité de calcul supportant les fonctions élémentaires des deux algorithmes SC et SCAN est proposée. Une grande partie des ressources matérielles nécessaires sont mutualisées. L'unité matérielle supportant les deux algorithmes est 60% plus complexe que celle gérant seulement l'algorithme SC.

(v) L'ASIP ainsi conçu présente des débits supérieurs à ceux obtenus sur des architectures de processeurs à usage général, et la consommation énergétique est réduite de deux ordres de grandeur. Les performances de débits et de consommation énergétique se rapprochent des architectures dédiées tout en conservant une meilleure programmabilité.

Les travaux présentés dans ce chapitre ont été valorisés à travers une publication à la conférence ISTC 2018 [62].

L'algorithme SCL permet d'atteindre de meilleures performances que les algorithmes SC et SCAN. Il sera sans doute utilisé dans le cadre du standard 5G. Aussi, de futurs travaux se concentreront sur la conception d'une architecture TTA supportant l'algorithme SCL. De plus, la suite logicielle TCE facilite la conception d'architectures multiprocesseurs et la

Tableau 5.4 Implémentations sur cible FPGA de décodeurs SCAN pour un code polaire (1024,512).

	TT-SCAN	[52]	[53]
Cible	Artix-7	Stratix IV	ASIC 90nm
Cycles d'horloge	10755	10304	2441
Débit info. (Mb/s)	4.5	4.4	217
Fréquence (MHz)	93	90	571
LUTS	20602	3517	-
FFs	8520	1024	-
RAM (Kb)	1079	209	-
Élagage	R0 & R1	Aucun	R0 & R1

description logicielle de programmes adaptés. La conception d'architectures de décodeurs multiprocesseurs est un second axe de recherche à privilégier.

CHAPITRE 6 CONCLUSION

Résumé et comparaison des différentes approches

Les travaux présentés dans ce manuscrit portent sur l'implémentation d'algorithmes de décodage de codes polaires sur des architectures programmables.

Dans le deuxième chapitre, des implémentations logicielles des algorithmes de décodage de codes polaires à liste sont proposées. L'originalité de ces décodeurs logiciels tient à leur forte flexibilité. Cette flexibilité est inédite en comparaison des travaux préalables publiés dans la littérature. Tout d'abord, les données internes de l'algorithme sont représentées en virgule fixe, ce qui diminue l'empreinte mémoire et permet de réduire le temps nécessaire au décodage d'une trame. De plus, l'élagage de l'arbre de décodage est une simplification indispensable pour atteindre de hauts débits et de faibles latences de décodage. Cet élagage, dans les décodeurs proposés, est dynamiquement configurable. La flexibilité de l'élagage permet des compromis intéressants entre le débit, la latence et les performances de décodage. Grâce à différentes optimisations, la flexibilité et la généricité du décodeur sont atteintes sans sacrifier le débit ou la latence de décodage. Une des versions de l'algorithme à liste, l'algorithme FASCL, permet même de dépasser le débit des implémentations de l'état de l'art sur les architectures x86. Des résultats d'implémentation logicielle des algorithmes de décodage de codes polaires à liste sont également proposés sur des architectures ARM. Comme attendu, les débits sont moindres, mais la consommation énergétique est réduite.

Dans le troisième chapitre, une architecture de processeur spécialisée pour le décodage de codes polaires est proposée. Le processeur fait partie de la famille des ASIP. Les outils de la société Tensilica sont utilisés. Les processeurs conçus à l'aide de ces outils sont des processeurs du type RISC. Le jeu d'instructions peut être étendu et la microarchitecture peut être modifiée afin de les rendre plus efficaces pour un type d'application. Ils conservent toutefois la versatilité des architectures RISC classiques et bénéficient d'un écosystème logiciel facilitant leur conception et le développement des programmes les ciblant. Tout comme les implémentations logicielles du deuxième chapitre, le programme est décrit dans un langage de haut niveau (C++). Les expérimentations et les mesures réalisées montrent que la spécialisation de l'architecture RSIC permet d'atteindre des débits comparables à ceux obtenus sur les architectures ARM, tout en réduisant la consommation énergétique d'un ordre de grandeur.

Dans le quatrième chapitre, une architecture de processeur du type TTA est conçue. Configu-

rable plus finement et d'une structure plus modulaire, ce type d'architecture programmable se rapproche des implémentations matérielles dédiées, du point de vue de leur structure comme de celui de leurs performances. Tout comme les architectures du troisième chapitre, elles n'en restent pas moins versatiles et programmables. Elles sont également dotées d'un environnement de conception complet. Cet environnement est une suite logicielle libre développée par l'équipe « Customized Parallel Computing » de l'université technique de Tampere (TUT). Contrairement aux architectures réalisées avec les outils de la société Tensilica, le modèle complet du processeur est produit par l'environnement, pour être utilisé par des logiciels de synthèse tiers. Deux architectures programmables sont proposées. La première supporte l'algorithme de décodage SC seul. La seconde supporte les algorithmes de décodage SC et SCAN. Les deux architectures programmables ont tout d'abord été synthétisées et implantées sur des circuits FPGA afin d'en effectuer la vérification fonctionnelle. Dans un second temps, des résultats de synthèse sur cible ASIC ont été générés pour la première architecture. Ces résultats montrent que cette architecture surpasse les débits obtenus sur les architectures x86, tout en réduisant la consommation énergétique de deux ordres de grandeur.

Dans le Tableau 6.1, les caractéristiques des implémentations logicielles sur les différentes architectures de décodeurs considérées dans ce manuscrit sont récapitulées. Tout d'abord, les processeurs d'architecture x86 et ARM sont disponibles sur le marché. Au contraire, les ASIP sont des architectures spécialisées pour lequel des développements au niveau architectural doivent être engagés afin d'aboutir à une implantation sur cible ASIC. Cependant, les deux ASIP sont plus efficaces énergétiquement. L'architecture `TT-SC` est la plus performante du point de vue du débit et de l'efficacité énergétique. Néanmoins, il s'agit aussi de l'architecture la moins généraliste. Un des symptômes de ce manque de flexibilité est l'absence de système d'exploitation ciblant les architectures TTA. À l'inverse, les processeurs XTensa bénéficient d'un système d'exploitation fondé sur le noyau Linux. En effet, les processeurs XTensa sont basés sur des architectures RISC qui sont des architectures de processeurs classiques. En ce sens, leur flexibilité se rapproche de celles des architectures x86-64 et ARM. L'ensemble des quatre architectures de processeurs offre des compromis marqués entre flexibilité, performance et efficacité énergétique.

Perspectives

Les travaux présentés dans cette thèse ouvrent plusieurs pistes de recherche.

Le décodeur logiciel proposé dans le deuxième chapitre est intégré à la suite logicielle libre AFF3CT de l'équipe CSN du laboratoire IMS de Bordeaux. Le but de ce projet est de proposer à la communauté scientifique des implémentations logicielles efficaces d'algorithmes de

Tableau 6.1 Existence, disponibilité d'un système d'exploitation, débits et consommations énergétiques des processeurs pour les différentes architectures considérées. Les intervalles de débits et de consommations énergétiques concernent le décodage de mots de codes polaires dont les tailles varient de $\mathsf{N} = 128$ à $\mathsf{N} = 1024$ et dont le rendement est $\mathsf{R} = 1/2$

Architecture	**ASIP**	**Système d'exploitation**	$\mathcal{T}_i$ **Mb/s**	$\mathcal{E}_b$ **nJ**
x86-64 (Haswell)	✗	✓	120-260	40-90
ARMv8-A	✗	✓	30-40	10-30
XTensa Polaire	✓	✓	30-70	1-2
`TT-SC`	✓	✗	250-350	0.1-0.2

décodage de codes correcteurs d'erreurs. Les travaux réalisés dans le cadre de cette thèse l'ont grandement enrichi. Dans le futur, une plus grande variété d'algorithmes de décodage de codes polaires pourrait être supportée. Par exemple, les algorithmes SCF et SCS présentés dans le premier chapitre présentent un intérêt certain. Cependant, pour autant que nous le sachions, il n'existe aucune référence dans la littérature reportant les débits et les latences obtenus sur des architectures de processeur à usage général de ces deux algorithmes. Or, la faible complexité calculatoire de l'algorithme SCF semble indiquer de bonnes performances potentielles dans ce domaine, malgré des performances de décodage légèrement en retrait par rapport à celles de l'algorithme SCL. L'algorithme SCS présente quant à lui des performances de décodage égales à celles de l'algorithme SCL. Une étude récente tend à montrer que de hauts débits et de faibles latences pourraient être atteints [54]. Ils proposent également de nouvelles constructions de codes polaires appelés codes polaires multi-noyaux [55,56]. Leur principe est d'utiliser de nouveaux noyaux élémentaires afin de construire la matrice génératrice de codes polaires et non plus seulement les noyaux de taille 2 proposés originellement. L'intégration de ces codes dans AFF3CT et surtout la proposition d'implémentations logicielles efficaces de décodeurs logiciels associés présenteraient un grand intérêt pour la communauté scientifique.

Parmi les architectures programmables considérées dans ces travaux, celles permettant d'atteindre les meilleurs débits, latences et efficacités énergétiques sont les architectures TTA. Un axe de recherche futur envisagé est la conception d'une architecture TTA pour le décodage des algorithmes de codes polaires à liste. En effet, les performances de décodage sont bien plus élevées, au prix d'une plus grande complexité calculatoire. Plusieurs étapes seront nécessaires. Tout d'abord, nos travaux montrent que le temps alloué à la propagation des sommes partielles doit être réduit. Une solution pourrait être de s'inspirer de ce qui est réalisé dans

les architectures matérielles dédiées. Ensuite, l'architecture devra être étendue, afin de gérer le calcul des métriques de chemin et les LLR associés à chaque arbre de décodage. De plus, dans les algorithmes à liste, ces arbres de décodage doivent être parcouru en parallèle. Une méthode pour réaliser ces décodages parallèles pourrait être de concevoir une plate-forme multicœurs. La conception de telles plates-formes peut être facilitée grâce à la suite logicielle TCEMC [44]. Des architectures TTA multicœurs pour le décodage de turbo codes proposées dans la littérature [57] pourraient être une bonne source d'inspiration.

Enfin, au cours de cette thèse, des travaux ont été réalisés afin d'intégrer les codes polaires dans des chaînes de communication plus complexes [63, 64]. Le sujet de ces publications est l'implémentation de la technique d'accès multiple SCMA (Sparse Code Multiple Access). L'articulation de cette technique avec le codage et le décodage de codes polaires a été réalisé. Toutefois, par l'utilisation d'algorithmes de décodage de codes polaires à sorties souples, des échanges d'informations entre le décodeur polaire et le décodeur SCMA seraient possibles au sein d'un processus itératif, améliorant ainsi les performances de décodage de l'ensemble du système. Un autre cas d'application des codes polaires pourrait être l'implémentation de la couche physique de la norme 5G [36], sur des architectures de processeur à usage général comme sur des architectures TTA.

RÉFÉRENCES

[1] C. E. Shannon, “A Mathematical Theory of Communication,” *Bell System Technical Journal*, 1948.

[2] E. Arikan, “Channel polarization : A method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels,” *IEEE Transactions on Information Theory*, vol. 55, n°. 7, 2009.

[3] G. Sarkis, P. Giard, A. Vardy, C. Thibeault et W. J. Gross, “Fast Polar Decoders : Algorithm and Implementation,” *IEEE Journal on Selected Areas in Communications*, vol. 32, n°. 5, p. 946–957, mai 2014.

[4] P. Giard, G. Sarkis, C. Thibeault et W. J. Gross, “Fast Software Polar Decoders,” dans *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, mai 2014, p. 7555–7559.

[5] Ericsson, “Cloud RAN - The Benefits of Virtualization, Centralisation and Coordination,” Rapport technique, 2015.

[6] Huawei, “5G : A Technology Vision,” Rapport technique, 2013. [En ligne]. Disponible : https://www.huawei.com/ilink/en/download/HW_314849

[7] D. Wübben, P. Rost, J. S. Bartelt, M. Lalam, V. Savin, M. Gorgoglione, A. Dekorsy et G. Fettweis, “Benefits and Impact of Cloud Computing on 5g Signal Processing : Flexible Centralization Through Cloud-RAN,” *IEEE Signal Processing Magazine*, vol. 31, n°. 6, p. 35–44, 2014.

[8] P. Rost, C. J. Bernardos, A. De Domenico, M. Di Girolamo, M. Lalam, A. Maeder, D. Sabella et D. Wübben, “Cloud Technologies for Flexible 5G Radio Access Networks,” *IEEE Communications Magazine*, vol. 52, n°. 5, p. 68–76, 2014.

[9] A. Checko, H. L. Christiansen, Y. Yan, L. Scolari, G. Kardaras, M. S. Berger et L. Dittmann, “Cloud RAN for Mobile Networks—A Technology Overview,” *IEEE Communications Surveys Tutorials*, vol. 17, n°. 1, p. 405–426, 2015.

[10] V. Q. Rodriguez et F. Guillemin, “Towards the Deployment of a Fully Centralized Cloud-RAN Architecture,” dans *IEEE International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2017, p. 1055–1060.

[11] N. Nikaein, “Processing Radio Access Network Functions in the Cloud : Critical Issues and Modeling,” dans *ACM International Workshop on Mobile Cloud Computing and Services (MCS)*, 2015, p. 36–43.

[12] P. Giard, G. Sarkis, C. Leroux, C. Thibeault et W. J. Gross, “Low-Latency Software Polar Decoders,” *Springer Journal of Signal Processing Systems (JSPS)*, p. 31–53, juill. 2016.

[13] G. Sarkis, P. Giard, C. Thibeault et W. Gross, “Autogenerating Software Polar Decoders,” dans *IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, 2014.

[14] B. L. Gal, C. Leroux et C. Jégo, “Software Polar Decoder on an Embedded Processor,” dans *IEEE Workshop on Signal Processing Systems (SiPS)*, oct. 2014, p. 1–6.

[15] A. Cassagne, B. Le Gal, C. Leroux, O. Aumage et D. Barthou, “An Efficient, Portable and Generic Library for Successive Cancellation Decoding of Polar Codes,” dans *Springer International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, sept. 2015.

[16] A. Cassagne, O. Aumage, C. Leroux, D. Barthou et B. Le Gal, “Energy Consumption Analysis of Software Polar Decoders on Low Power Processors,” dans *IEEE European Signal Processing Conference (EUSIPCO)*, 2016, p. 642–646.

[17] B. L. Gal, C. Leroux et C. Jégo, “Multi-Gb/s Software Decoding of Polar Codes,” *IEEE Transactions on Signal Processing (TSP)*, vol. 63, n°. 2, p. 349–359, janv. 2015.

[18] G. Sarkis, P. Giard, A. Vardy, C. Thibeault et W. J. Gross, “Fast List Decoders for Polar Codes,” *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 34, n°. 2, p. 318–328, 2016.

[19] ——, “Increasing the Speed of Polar List Decoders,” dans *IEEE Workshop on Signal Processing Systems (SiPS)*, 2014, p. 1–6.

[20] Y. Shen, C. Zhang, J. Yang, S. Zhang et X. You, “Low-latency Software Successive Cancellation List Polar Decoder Using Stage-Located Copy,” dans *IEEE International Conference on Digital Signal Processing (DSP)*, 2016.

[21] A. Cassagne, O. Aumage, D. Barthou, C. Leroux et C. Jégo, “MIPP : a Portable C++ SIMD Wrapper and its Use for Error Correction Coding in 5G Standard,” dans *ACM Workshop on Programming Models for SIMD/Vector Processing (WPMVP)*, 2018.

[22] E. Dahlman, S. Parkvall et J. Skold, *4G : LTE/LTE-Advanced for Mobile Broadband*. Academic press, 2013.

[23] J. Schreier, “On Tournament Elimination Systems,” *Mathesis Polska*, vol. 7, p. 154–160, 1932.

[24] D. Knuth, *The Art of Computer Programming*. Addison-Wesley, 1973, n°. 3.

[25] T. Furtak, J. N. Amaral et R. Niewiadomski, “Using SIMD Registers and Instructions to Enable Instruction-Level Parallelism in Sorting Algorithms,” dans *ACM Symposium on Parallel Algorithms and Architectures*, 2007, p. 348–357.

[26] I. Tal et A. Vardy, “List Decoding of Polar Codes,” dans *IEEE International Symposium on Information Theory (ISIT)*, 2011, p. 1–5.

[27] C. Leroux, I. Tal, A. Vardy et W. J. Gross, “Hardware Architectures for Successive Cancellation Decoding of Polar Codes,” dans *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, mai 2011, p. 1665–1668.

[28] A. Cassagne, M. Léonardon, O. Hartmann, G. Delbergue, T. Tonnellier, R. Tajan, C. Leroux, C. Jégo, B. Le Gal, O. Aumage et D. Barthou, “Fast Simulation and Prototyping with AFF3CT,” dans *IEEE Workshop on Signal Processing Systems (SiPS)*, 2017.

[29] “HWiNFO - Hardware Information, Analysis and Monitoring Tools.” [En ligne]. Disponible : https://www.hwinfo.com

[30] M. Humrick, “HiSilicon Kirin 960 : A Closer Look at Performance and Power.” [En ligne]. Disponible : https://www.anandtech.com/show/11088/hisilicon-kirin-960-performance-and-power

[31] R. Holmgren, “Energy Efficiency Experiments on Samsung Exynos 5 Heterogeneous Multicore Using OmpSs Task Based Programming,” Mémoire de maîtrise, NTNU, 2015.

[32] Y. Benmoussa, “Performance and Energy Consumption Characterization and Modeling of Video Decoding on Multi-core Heterogenous SoC and Their Applications,” Thèse de doctorat, Université de Bretagne Occidentale, 2015.

[33] J. L. Hennessy et D. A. Patterson, *Computer Architecture : a Quantitative Approach*. Elsevier, 2011.

[34] P. Mishra et N. Dutt, *Processor Description Languages*. Elsevier, 2011, vol. 1.

[35] Cadence Design Systems, Inc., San Jose, CA, U.S.A., *Tensilica Instruction Extension (TIE) Language Reference Manual*, 2017.

[36] 3GPP, “TS 38.212, Multiplexing and Channel Coding (Release 15),” Rapport technique, sept. 2017.

[37] C. Leroux, A. J. Raymond, G. Sarkis et W. J. Gross, “A Semi-Parallel Successive-Cancellation Decoder for Polar Codes,” *IEEE Transactions on Signal Processing (TSP)*, vol. 61, n°. 2, 2013.

[38] T. Zhang et K. K. Parhi, “A 54 Mbps (3,6)-Regular FPGA LDPC Decoder,” dans *IEEE Workshop on Signal Processing Systems*, Oct 2002, p. 127–132.

[39] B. R. Rau et J. A. Fisher, “Instruction-Level Parallel Processing : History, Overview, and Perspective,” dans *Instruction-Level Parallelism.* Springer, 1993, p. 9–50.

[40] H. Corporaal, *Microprocessor Architectures : From VLIW to TTA.* New York, NY, USA : John Wiley & Sons, Inc., 1997.

[41] D. Tabak et G. J. Lipovski, “MOVE Architecture in Digital Controllers,” *IEEE Journal of Solid-State Circuits*, 1980.

[42] P. Jääskeläinen, *From Parallel Programs to Customized Parallel Processors.* Tampere University of Technology, 2012.

[43] P. Jääskeläinen, T. Viitanen, J. Takala et H. Berg, “HW/SW Co-design Toolset for Customization of Exposed Datapath Processors,” dans *Computing Platforms for Software-Defined Radio.* Springer, Cham, 2017, p. 147–164.

[44] P. Jääskeläinen, E. Salminen, C. Sanchez de La Lama, J. Takala et J. Ignacio Martinez, “Tcemc : A Co-Design Flow for Application-Specific Multicores,” dans *IEEE International Conference on Embedded Computer Systems : Architectures, Modeling and Simulation (IC-SAMOS)*, 2011, p. 85–92.

[45] C. Lattner et V. Adve, “LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation,” dans *International Symposium on Code Generation and Optimization : Feedback-Directed and Runtime Optimization (CGO).* IEEE Computer Society, 2004, p. 75.

[46] P. Giard, A. Balatsoukas-Stimming, T. C. Müller, A. Bonetti, C. Thibeault, W. J. Gross, P. Flatresse et A. Burg, “PolarBear : A 28 nm FD-SOI ASIC for Decoding of Polar Codes,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems (JESTCS)*, 2017.

[47] A. Mishra, A. J. Raymond, L. G. Amaru, G. Sarkis, C. Leroux, P. Meinerzhagen, A. Burg et W. Gross, “A Successive Cancellation Decoder ASIC for a 1024-bit Polar Code in 180nm CMOS,” dans *Asilomar Conference on Signals, Systems, and Computers (ACSSC)*, 2012.

[48] P. Giard, G. Sarkis, C. Thibeault et W. J. Gross, “A 638 Mbps Low-Complexity Rate 1/2 Polar Decoder on FPGAs,” dans *IEEE Workshop on Signal Processing Systems (SiPS)*, oct. 2015, p. 1–6.

[49] B. L. Gal, C. Leroux et C. Jégo, “A Scalable 3-Phase Polar Decoder,” dans *IEEE International Symposium on Circuits and Systems (ISCAS)*, mai 2016.

[50] P. Giard, G. Sarkis, C. Thibeault et W. J. Gross, “Unrolled Polar Decoders, Part I : Hardware Architectures,” *CoRR*, vol. abs/1505.01459, 2015. [En ligne]. Disponible : http://arxiv.org/abs/1505.01459

[51] STMicroelectronics, “Best-in-Class Standard-Cell Libraries for High-Performance, Low-Power and High-Density SoC Design in 28nmFD-SOI Technology,” 2015.

[52] G. Berhault, C. Leroux, C. Jégo et D. Dallet, “Hardware Implementation of a Soft Cancellation Decoder for Polar Codes,” dans *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, sept. 2015, p. 1–8.

[53] J. Lin, C. Xiong et Z. Yan, “Reduced Complexity Belief Propagation Decoders for Polar Codes,” dans *IEEE Workshop on Signal Processing Systems (SiPS)*, oct. 2015, p. 1–6.

[54] H. Aurora, C. Condo et W. J. Gross, “Low-Complexity Software Stack Decoding of Polar Codes,” dans *IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2018, p. 1–5.

[55] F. Gabry, V. Bioglio, I. Land et J. Belfiore, “Multi-Kernel Construction of Polar Codes,” dans *IEEE International Conference on Communications Workshops (ICC Workshops)*, May 2017, p. 761–765.

[56] V. Bioglio, F. Gabry, I. Land et J. Belfiore, “Minimum-Distance Based Construction of Multi-Kernel Polar Codes,” dans *IEEE Global Communications Conference (GLOBECOM)*, Dec 2017, p. 1–6.

[57] H. Kultala, O. Esko, P. Jääskeläinen, V. Guzma, J. Takala, J. Xianjun, T. Zetterman et H. Berg, “Turbo Decoding on Tailored OpenCL Processor,” dans *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, juill. 2013, p. 1095–1100.

[58] M. P. Fossorier, M. Mihaljevic et H. Imai, “Reduced Complexity Iterative Decoding of Low-Density Parity Check Codes Based on Belief Propagation,” *IEEE Transactions on Communications (TCOM)*, vol. 47, nº. 5, p. 673–680, 1999.

PUBLICATIONS

[59] A. Cassagne, M. Léonardon, O. Hartmann, T. Tonnellier, G. Delbergue, V. Giraud, C. Leroux, R. Tajan, B. Le Gal, C. Jégo, O. Aumage et D. Barthou, "AFF3CT : Un Environnement de Simulation pour le Codage de Canal," GdR SoC2, Rapport technique, juin 2017.

[60] M. Léonardon, A. Cassagne, C. Leroux, C. Jégo, L.-P. Hamelin et Y. Savaria, "Fast and Flexible Software Polar List Decoders," *Springer Journal of Signal Processing Systems (JSPS)*, 2019, accepted for publication.

[61] M. Léonardon, C. Leroux, D. Binet, J. M. P. Langlois, C. Jégo et Y. Savaria, "Custom Low Power Processor for Polar Decoding," dans *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018.

[62] M. Léonardon, C. Leroux, P. Jääskeläinen, C. Jégo et Y. Savaria, "Transport Triggered Polar Decoders," dans *IEEE International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*, 2018.

[63] A. Ghaffari, M. Léonardon, Y. Savaria, C. Jégo et C. Leroux, "Improving Performance of SCMA MPA Decoders Using Estimation of Conditional Probabilities," dans *IEEE International New Circuits and Systems Conference (NEWCAS)*, 2017.

[64] A. Ghaffari, M. Léonardon, A. Cassagne, C. Leroux et Y. Savaria, "Toward High Performance Implementation of 5G SCMA Algorithms," *IEEE Access*, 2019, accepted for publication.

ANNEXE A COMPLÉMENTS AU CHAPITRE 1

Les sorties du canal composite sont des informations souples données par le démodulateur. Ce bruit additif est ajouté à la valeur des données en sortie du modulateur $\mathbf{x}$.

$$y_i = x_i + n_i \tag{A.1}$$

Dans le modèle de chaine de communication considéré, la puissance du bruit du canal AWGN N_0 est supposée connue. Cela permet au démodulateur de calculer la probabilité conditionnelle suivante.

$$P_r(\tilde{y}_i|\tilde{x}_i) = \frac{1}{\sqrt{\pi N_0}} \exp^{-\frac{(\tilde{x}_i^2 - \tilde{y}_i^2)}{N_0}} \tag{A.2}$$

p_i représente la probabilité de recevoir $\tilde{y}_i$ en présupposant la valeur de $\tilde{x}_i$ en sortie du modulateur ($\tilde{x}_i = -1$ ou $\tilde{x}_i = 1$). Les *likelihood ratios* (LRs), sont exprimés en fonction de ces probabilités :

$$l_i = \log\left(\frac{P_r(y_i|x_i = 0)}{P_r(y_i|x_i = 1)}\right) \tag{A.3}$$

Les *log likelihood ratios* (LLR), sont le logarithme népérien des LR !

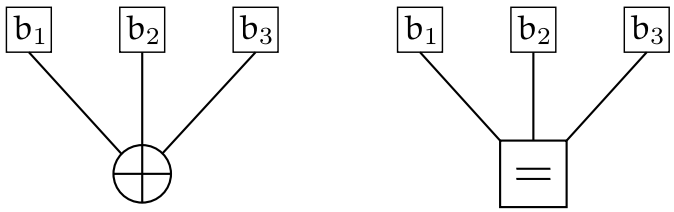
$$L_i = \log\left(\frac{P_r(y_i|x_i = 0)}{P_r(y_i|x_i = 1)}\right) \tag{A.4}$$

Les codes correcteurs d'erreurs peuvent être définis par un ensemble de contraintes de parités. Une représentation graphique de ces contraintes est appelé *graphe de Tanner* comme représenté en Figure A.1.

Le noeud de parité est la représentation graphique de l'équation suivante :

$$b_0 + b_1 + b_2 = 0 \tag{A.5}$$

Tandis que l'encodeur canal associe le vecteur d'entrée $\mathbf{b}$ de longueur K à un mot de code $\mathbf{x}$ de longueur N, le décodeur combine les différents LR ou LLR en sortie du canal composite dans le but de retrouver les bits d'origine. Dans l'exemple de la Figure A.1, les estimations du canal pour les valeurs des bits b_1 et b_2, notés respectivement l_1 et l_2, peuvent être combinées afin de calculer une estimation extrinsèque du bit b_0, notée l_0^e.



Nœud de parité Nœud d'égalité

Figure A.1 Représentations des noeuds de décodage

Sa valeur est par définition :

$$l_0^e = \frac{P(y_1,y_2|b_0=0)}{P(y_1,y_2|b_0=1)} \tag{A.6}$$

Seules deux combinaisons possibles des bits b_1 et b_2 peuvent impliquer que b_0 soit égal à 0 : ou bien $b_1 = 0$ et $b_2 = 0$, ou bien $b_1 = 1$ et $b_2 = 1$. Il est donc possible de calculer $P(b_0|y_1,y_2)$:

$$P(y_1,y_2|b_0=0) = P(y_1|b_1=0)P(y_2|b_2=0) + P(y_1|b_1=1)P(y_2|b_2=1) \tag{A.7}$$

Donc,

$$P(y_1,y_2|b_0=0) = l_1 l_2 + 1 \tag{A.8}$$

De la même manière,

$$P(b_0=1|y_1,y_2) = P(y_1|b_1=0)P(y_2|b_2=1) + P(y_1|b_1=1)P(y_2|b_2=0) \tag{A.9}$$

Donc,

$$P(b_0=1|y_1,y_2) = l_1 + l_2 \tag{A.10}$$

On déduit donc de A.6, A.8 et A.10 :

$$l_0^e = \frac{1 + l_1 l_2}{l_1 + l_2} \tag{A.11}$$

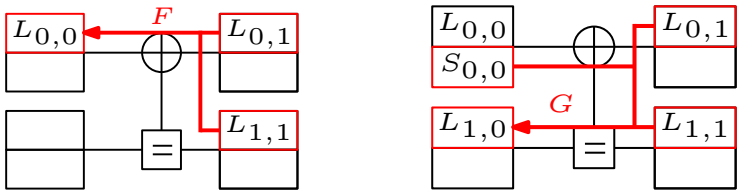
En effectuant la même démarche pour le Nœud d'égalité il est possible d'obtenir :

$$\begin{aligned} P(y_1,y_2|b_0=0) &= P(y_1|b_1=0)P(y_2|b_2=0) \\ P(y_1,y_2|b_0=1) &= P(y_1|b_1=1)P(y_2|b_2=1) \end{aligned}$$

Il est possible d'en déduire l'extrinsèque :

$$l_0^e = l_1 l_2 \tag{A.12}$$

L'algorithme de décodage SC peut être représenté sous la forme de *factor graph* dans lequel les équations A.11 et A.12 sont appliquées, comme montré en Figure A.2. Les fonctions élémentaires F et G sont des combinaisons des équations liées aux nœuds de parité et aux nœuds d'égalités.



(a) F function (b) G function

Figure A.2 Fonctions F et G de l'algorithme SC

$$l_{0,0} = F(l_{0,1},l_{1,1}) = \frac{1 + l_{0,1}l_{1,1}}{l_{0,1} + l_{1,1}} \tag{A.13}$$

$$l_{1,0} = \begin{cases} l_{1,1}l_{0,1} & \text{if } S_{0,0} = 0 \\ l_{1,1}\dfrac{1}{l_{0,1}} & \text{if } S_{0,0} = 1 \end{cases}$$

Cela peut être simplifié :

$$l_{1,0} = G(l_{1,1},l_{0,1},S_{0,0}) = l_{1,1}l_{0,1}^{1-2S_{0,0}} \tag{A.14}$$

Ces équations utilisent des LR. Elles peuvent être modifiées afin de faire apparaître les LLR :

$$L_{0,0} = f(L_{0,1},L_{1,1}) = 2\tanh^{-1}(\tanh(L_{0,1})\tanh(L_{1,1})) \tag{A.15}$$

$$L_{1,0} = g(L_{1,1},L_{0,1},S_{0,0}) = L_{1,1} + (1 - 2S_{0,0})L_{0,1} \tag{A.16}$$

Enfin, f peut être simplifiée [58] :

$$f(L_{0,1},L_{1,1}) \approx \text{sign}(L_{0,1})\text{sign}(L_{1,1})\min(|L_{0,1}|,|L_{1,1}|)) \tag{A.17}$$