

# Architecture générique de décodeur de codes LDPC

Frédéric Guilloud

▶ **To cite this version:**

Frédéric Guilloud. Architecture générique de décodeur de codes LDPC. Signal and Image processing. Télécom ParisTech, 2004. English. NNT : . pastel-00000806

**HAL Id: pastel-00000806**
**https://pastel.hal.science/pastel-00000806**

Submitted on 6 Sep 2004

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Generic Architecture for LDPC Codes Decoding

Frédéric GUILLOUD

July, 2004

This page intentionally left blank.

*j'ai longtemps hésité ...*
*et puis maintenant il faut se décider ! faut choisir !*
*alors je mets quoi ?*
*d'ailleurs, je mets quelque chose ?*
*très envie de le crier, évidemment ...*
*et puis en même temps pourquoi ?*
*je crois que je n'ose pas ...*
*c'est pas grave, ca ne change rien, en vrai ...*
*comme c'est tout le temps et partout, pourquoi ici plus qu'ailleurs ?*

*finalement je crois que j'ai osé ...*


(c) Frédéric Guilloud, Télécom Paris - July 2004

This page intentionally left blank.

# Remerciements

Je tiens à remercier Emmanuel Boutillon et Jean-Luc Danger pour m'avoir permis de faire cette thèse au sein du département COMELEC de Télécom Paris, et pour m'avoir encadré avec sérieux et clairvoyance. J'ai pu bénéficier de leurs compétences complémentaires et cette thèse est le résultat de cette étroite collaboration. Je tiens particulièrement à remercier Emmanuel Boutillon qui, malgré la distance, a su se montrer très présent aux moments des choix décisifs qui ont jalonné cette thèse, et dont la compétence et la générosité ont grandement contribué aux résultats de ce travail.

J'exprime toute ma gratitude à Michel Jezequel pour avoir accepté de présider le jury et je remercie Marc Fossorier et Jean-Didier Legat pour l'honneur qu'ils m'ont fait en acceptant le rôle difficile de rapporteur. Merci également à David Declercq et François Barbara d'avoir bien voulu examiner ce travail.

Je remercie très chaleureusement Mouhcine Chami sans qui le développement de la plateforme ne serait pas aussi avancé. Son travail efficace sur la mise en oeuvre de la plateforme m'a permis de réaliser les premières simulations sans trop de retard. Je remercie aussi vivement Nicolas Lascar pour son étude complète et les développements qu'il a menés autour de l'encodage des codes LDPC, ainsi que Cheng Ching qui l'a précédé dans cette tâche.

Un grand merci aussi à Julien Viard dont les compétences en programmation orientée objet m'ont permis de gagner un temps précieux lors du développement du logiciel de décodage des codes LDPC. Merci aussi à Christophe Devaucelle qui a pendant son stage de fin d'étude défriché les problèmes d'implantation du décodage des codes LDPC.

Je tiens enfin à rendre hommage à Jean-Claude Bic, responsable du département COMELEC lorsque j'y ai débuté ma thèse, dont je garderai un souvenir ému.

../..

**Et pour ces trois années et demi de bonne ambiance ...**

Merci au groupe Système d'Intégration Analogique Mixte qui a bien voulu héberger un élément numérique dans leur locaux, et notamment à Elizabeth qui n'a pas réussi à m'apprendre l'espagnol, malgré ces trois années à partager le même bureau, mais à qui j'espère avoir appris que la fin normale du repas est constituée de fromage, de fruits et du dessert (au chocolat de préférence !).

Un grand merci à Slim Chabbouh, David Gnaedig pour toutes les discussions que nous avons pu avoir et les divers transports dans la lande bretonne ; merci à Christophe (Tophe), Mouhcine, Nico et Gégé pour les pauses bien sympathiques qui ont agrémenté cette fin de thèse ; merci à Chantal pour sa disponibilité et sa gentillesse ; merci à Van Tam, mon partenaire de printemps, pour avoir soutenu après moi ; merci à Sabeur, Mohammed, Karim, Bilal, Betch, David, Sonia, Richard, Reda, Sébastien, Ioannis, Maryam pour leur contribution à la bonne ambiance de travail qui a baigné le cours de ma thèse.

Je remercie aussi ma famille, et en particulier mes parents, pour leur patience et la compréhension dont ils ont fait preuve tout au long de mes études.

Et puis enfin un très grand merci à mon épouse Raphaëlle, qui m'a soutenu tout au long de ce travail, et qui a su apporter avec talent et enthousiasme les touches finales à ce manuscrit.

A vous tous je le dis de façon certaine : les études, cette fois ci, j'arrête vraiment ; j'ai décidé d'en faire mon métier !

Paris, le 8 juillet 2004

(c) Frédéric Guilloud, Télécom Paris - July 2004

# Résumé

Les codes correcteurs d'erreurs LDPC (Low Density Parity Check) font partie des codes en bloc permettant de s'approcher de quelques fractions de dB de la limite de Shannon. Ces remarquables performances associées à leur relative simplicité de décodage rendent ces codes très attractifs pour les prochaines générations de systèmes de transmissions numériques. C'est notamment déjà le cas dans la norme de télédiffusion numérique par satellite (DVB-S2) qui utilise un code LDPC irrégulier pour la protection de la transmission des données descendantes.

Dans cette thèse, nous nous sommes intéressés aux algorithmes de décodage des codes LDPC et à leur implantation matérielle. Nous avons tout d'abord proposé un algorithme sous-optimal de décodage (l'algorithme lambda-min) permettant de réduire de façon significative la complexité du décodeur sans perte de performances par rapport à l'algorithme de référence dit à propagation de croyance (algorithme BP).

Nous avons ensuite étudié et conçu une architecture générique de décodeur LDPC, que nous avons implantée sur une plateforme dédiée à base de circuits logiques programmables FPGA. Ce décodeur matériel permet avant tout d'accélérer les simulations d'un facteur supérieur à 500 par rapport à une simulation logicielle. De plus, par sa conception entièrement programmable, modulaire et générique, il possède de nombreuses fonctionnalités : Il peut ainsi être configuré pour une large classe de codes, et en conséquence permettre la recherche de codes efficaces ; par la généricité des opérateurs de calcul, il permet aussi l'optimisation de la précision interne des calculs en vue d'une conception ASIC ; et par sa modularité, différents algorithmes de calcul (dits processeur de noeuds) et de séquencement peuvent être testés.

Enfin, notre travail nous a permis de dégager un cadre formel d'analyse et de synthèse des architectures de décodeurs LDPC. Ce cadre englobe à la fois les chemins de données (parallélisme, architecture des processeurs de noeuds) et le mode de contrôle associé aux différents séquencements de décodage. Ainsi, cette formalisation nous permet de classer les différentes solutions de l'état de l'art des décodeurs LDPC, mais aussi de proposer de nouvelles architectures intéressantes non publiées à ce jour.

 (c) Frédéric Guilloud, Télécom Paris - July 2004

This page intentionally left blank.

# Abstract

The Low-Density Parity-Check codes are among the most powerful forward error correcting codes, since they enable to get as close as a fraction of a dB from the Shannon limit. This astonishing performance combined with their relatively simple decoding algorithm make these codes very attractive for the next digital transmission system generations. It is already the case for the next digital satellite broadcasting standard (DVB-S2), where an irregular LDPC code has been chosen to protect the downlink information.

In this thesis, we focused our research on the iterative decoding algorithms and their hardware implementations. We proposed first a suboptimal algorithm named the $\lambda$-min algorithm. It reduces significantly the complexity of the decoder without any significant performance loss, as compared to the belief propagation (BP) algorithm.

Then we studied and designed a generic architecture of an LDPC decoder, which has been implemented on a FPGA based platform. This hardware decoder enables to accelerate the simulations more than 500 times as compared to software simulations. Moreover, based on an all-tunable design, our decoder features many facilities: It is possible to configure it for a very wide code family, so that the research for good codes is processed faster ; thanks to the genericity of the processing components, it is also possible to optimize the internal coding format, and even to compare various decoding algorithms and various processing schedules.

Finally, our experience in the area of LDPC decoders led us to propose a formal framework for analysing the architectures of LDPC decoders. This framework encompasses both the datapath (parallelism, node processors architectures) and the control mode associated to the several decoding schedules. Thus within this framework, a classification of the different state-of-the-art LDPC decoders is proposed. Moreover, some synthesis of efficient and unpublished architectures have been also proposed.


(c) Frédéric Guilloud, Télécom Paris - July 2004

This page intentionally left blank.

# Preamble

**Summary:**

*This preamble aims at giving the particular events which modified the initial subject of my Ph.D., within the European SPRING project. The work and the results presented in this thesis are the main part of my thesis. The first part of my work, which will not be presented in this thesis is briefly presented in this chapter, after some explanations on the particular context of this thesis.*

## The SPRING project

My thesis has been funded by the European Commission (Directorate-General Information Society) within the a European research project named SPRING (project ). The project is coordinated by Schlumberger Industries (France), and three other academic partners are implied the project:

- the *Université Catholique de Louvain* (UCL - Belgium),
- the *Centro Nacional de Microelectronica* (CNM - Spain),
- the *Ecole Nationale Supérieure des Télécommunications* (ENST - France).

This project proposes the creation of a research and training network gathering academic partners (CNM, ENST, and UCL) and industry (Schlumberger) to build and exchange knowledge. The network will focus on transfer of technologies between partners (through Ph.D. thesis) and on the mutual and complementary training of all partners. The project directly contributes to 3 key actions:

- Microelectronics,
- Peripherals, sub-systems and Microsystems,
- Technologies for and management of information processing, communications and networks, including broadband, together with their implementation, and their application.


(c) Frédéric Guilloud, Télécom Paris - July 2004

In the beginning of the project, the RMS Montrouge Technology Center (MTC), a technical research center of Schlumberger located in Montrouge (France) was in charge of the coordination of the project. Coordinators were in particular interested by remote reading applications in the metering business. This topic has been the starting point of my work. For one year, we have been studying some theoretical aspects of the communication scheme used in this application, as described in the next section.

At the end of the first year, the TMC has been closed and the coordination of the project moved to an another Schlumberger entity (OFS), located in Clamart, and working in the oilfield services. After a work on the performance of the demodulation scheme, we decided to study the channel decoder part. Our interest for LDPC codes decoding grew between the transfer of the SPRING project from RMS to OFS (November, 2001), motivated by two main reasons: the first reason is that these codes had very promising performance, compared to turbo-codes; the second reason was that architectures for LDPC codes decoder were a new challenging interest of research in the algorithm architecture adequation area.

Note that this thesis is part of the deliverables of the SPRING project. This is the reason why this thesis will be written in English.

## Contributions not cited in this thesis

As explained in the previous section, two important contributions of my work will not be discussed in this thesis:

1. A first contribution of this work has been published at ICC 2002 (Guilloud, Boutillon, and Danger 2002a). We calculated the bit error rate (BER) of a multiband non coherent on-off keying (OOK) demodulation. The results of this theoretical study fits perfectly with the simulations of the system. The influence of the filter and of the decimation factor on the modulation performance have been performed thanks to these results. We have also been able to optimize the system, by means of other criteria (e.g. system complexity, jammer sensitivity) thus avoiding time consuming simulations.

2. A second contribution is related to the parallel analog to digital conversion. A specific attention has to be paid to the filter banks in hybrid parallel architectures, in order to get a perfect signal reconstruction. In (Guilloud, Boutillon, and Danger 2002c), we proposed to study an alternative and original approach to the design of such converters to relax the filter bank constraint by some digital processing on the converted signal. We showed that for the moment, this original concept remains too sensitive to the inaccuracy of the implementation.

(c) Frédéric Guilloud, Télécom Paris - July 2004

# Contents

**Remerciements** v

**Résumé** vii

**Abstract** ix

**Preamble** xi

**Contents** xvi

**List of figures** xix

**List of tables** xxi

**List of listings** xxiii

**Notations** xxv

**Introduction** 1

**1 Channel coding** 5

1.1 Optimal decoding . . . . . 5

1.1.1 Shannon theorem for channel coding . . . . . 5

1.1.2 Communication model . . . . . 6

1.1.3 Optimal decoding . . . . . 7

1.2 Performance of error correcting codes . . . . . 8

1.2.1 The Shannon bound . . . . . 8

1.2.2 The AWGN capacity . . . . . 10

1.3 Decoding of linear block codes . . . . . 12

1.3.1 Definitions . . . . . 12

1.3.2 Optimal decoding of binary block codes . . . . . 15

1.3.3 The iterative algorithm . . . . . 19

1.4 Conclusion . . . . . 21


(c) Frédéric Guilloud, Télécom Paris - July 2004

**2 Low Density Parity Check codes** **23**
2.1 A bit of History . . . 23
2.2 Classes of LDPC codes . . . 24
2.3 Optimization of LDPC codes . . . 26
2.4 Constructions of LDPC codes . . . 28
2.4.1 Random based construction . . . 28
2.4.2 Deterministic based construction . . . 31
2.5 Encoding of LDPC codes . . . 32
2.5.1 Lower-triangular shape based encoding . . . 32
2.5.2 Other encoding schemes . . . 33
2.6 Performance of BPSK-modulated LDPC codes . . . 34
2.7 Decoding of LDPC codes . . . 36
2.7.1 Scheduling . . . 37
2.7.2 Performance in iterative decoding . . . 40
2.8 Conclusion . . . 40

**3 A unified framework for LDPC decoders** **43**
3.1 Generalized message-passing architecture . . . 43
3.1.1 Overview . . . 43
3.1.2 Shuffle network . . . 45
3.2 Node processors . . . 46
3.2.1 Generic node processor . . . 46
3.2.2 Variable and check node processors . . . 50
3.3 Complexity analysis . . . 54
3.3.1 Computation requirements . . . 54
3.3.2 Message rate . . . 54
3.3.3 Memory . . . 55
3.4 Synthesis . . . 57
3.4.1 Flooding schedule (check way) . . . 58
3.4.2 Horizontal shuffle schedule . . . 60
3.4.3 Vertical shuffle schedule . . . 62
3.4.4 Memory comparison . . . 64
3.5 Existing platforms survey . . . 64
3.5.1 Parallel designs . . . 64
3.5.2 serial design . . . 65
3.5.3 Mixed designs . . . 65
3.5.4 Summary . . . 66
3.6 Conclusion . . . 68

(c) Frédéric Guilloud, Télécom Paris - July 2004

**4 $\lambda-$Min Algorithm** **69**
4.1 Motivations and state of the art . . . 69
4.1.1 APP-based algorithms . . . 70
4.1.2 BP-based algorithm . . . 71
4.2 The $\lambda-$Min Algorithm . . . 71
4.3 Performance of the $\lambda-$Min Algorithm . . . 73
4.3.1 Simulation conditions . . . 73
4.3.2 Algorithm Comparison . . . 74
4.3.3 Optimization . . . 76
4.4 Architectural issues . . . 77
4.4.1 PCP architecture . . . 77
4.4.2 Memory saving . . . 84
4.5 Perspectives . . . 84
4.6 Conclusion . . . 87

**5 Generic Implementation of an LDPC Decoder** **89**
5.1 Overview . . . 89
5.1.1 About genericity . . . 89
5.1.2 Synoptic . . . 90
5.1.3 Architecture . . . 90
5.2 Memory Management . . . 93
5.2.1 Preliminaries . . . 93
5.2.2 Variable node memories . . . 94
5.2.3 Check node memories . . . 97
5.2.4 The memories A and S . . . 99
5.2.5 Migration to FPGA . . . 103
5.3 Shuffle network . . . 105
5.4 Description of the architecture . . . 108
5.5 Universality of the decoder . . . 110
5.5.1 Randomly designed LDPC codes . . . 111
5.5.2 Preprocessing of existing LDPC codes . . . 111
5.6 Conclusion . . . 114

**6 The platform** **115**
6.1 Platform description . . . 115
6.1.1 Overview . . . 115
6.1.2 Intrinsic information processing . . . 118
6.2 Synthesis . . . 119
6.2.1 Illustration of the genericity . . . 119
6.2.2 Synthesis results . . . 122
6.3 Simulations . . . 122

(c) Frédéric Guilloud, Télécom Paris - July 2004

6.3.1 Simulation conditions . . . 122
6.3.2 On the BP algorithm . . . 122
6.3.3 On the $\lambda-$min algorithm . . . 124
6.3.4 Algorithm comparaison . . . 124
6.3.5 Intrinsic information computing . . . 126
6.4 conclusion . . . 129

**Conclusion and Perspectives** **131**

**A Minimum BER achievable by coded BPSK systems** **133**

**B Log Likelihood Ratios and Parity Checks** **135**
B.1 Iterative expression . . . 135
B.1.1 2−variable rule . . . 135
B.1.2 Hardware efficient implementation . . . 136
B.1.3 $n-$variable rule . . . 137
B.2 Expression of the LLR using the tanh rule . . . 140
B.2.1 2−variable rule . . . 140
B.2.2 $n-$variable rule . . . 140

**C Architecture of the node processors** **141**
C.1 Check Node Unit (CNU) . . . 141
C.2 Variable Node Unit (VNU) . . . 144

**D Platform component architecture** **149**

**E Listings** **153**

**F Résumé étendu** **157**
Introduction . . . 157
F.1 Codage de canal . . . 158
F.2 Les codes LDPC . . . 158
F.3 Analyse unifiée des décodeurs LDPC . . . 159
F.4 L'algorithme $\lambda-$Min . . . 160
F.5 Architecture générique de décodeur LDPC . . . 161
F.6 La plateforme de simulation . . . 162
Conclusion . . . 162

**Bibliography** **165**

(c) Frédéric Guilloud, Télécom Paris - July 2004

# List of Figures


1.1 Basic scheme for channel code encoding/decoding. . . . . 6
1.2 AWGN noise power spectral density . . . . . 9
1.3 The Shannon bound. . . . . 10
1.4 Capacity in bit per dimension for the AWGN channel. . . . . 11
1.5 Capacity in bit per dimension for the discrete-input AWGN channel. . . . 12
1.6 Different ways of describing the same code. . . . . 14
1.7 The notations related to the bipartite graph of a code. . . . . 15
1.8 Cycles in graphs. . . . . 16
1.9 Example for calculating total information. . . . . 20
1.10 Update rules in graph node processors. . . . . 22

2.1 Sum-product algorithm in a factor graphs. . . . . 24
2.2 Random constructions of regular LDPC codes . . . . . 29
2.3 Random constructions of irregular LDPC codes . . . . . 30
2.4 Shape of parity check matrices for efficient encoding. . . . . 32
2.5 Gap between AWGN and Bi-AWGN channel capacity. . . . . 35
2.6 Various thresholds for regular or irregular LDPC codes, for various rates . 35
2.7 Distance to the Shannon bound estimation using error probability curves. 36
2.8 Flooding and probabilistic schedule. . . . . 38
2.9 Shuffle schedule. . . . . 39
2.10 Horizontal Shuffle schedule. . . . . 40
2.11 Waterfall and error floor regions in iterative decoding. . . . . 41

3.1 Generalized design for message passing architecture decoders . . . . . 44
3.2 Elementary switch . . . . . 46
3.3 A generic node processor . . . . . 46
3.4 Possible implementations for a generic node processor . . . . . 48
3.5 The master/slave modes for node processors . . . . . 49
3.6 The slow/fast modes for slave node processors . . . . . 50
3.7 Possible positions of the interconnection network . . . . . 52
3.8 Simple parity-check matrix example . . . . . 57



(c) Frédéric Guilloud, Télécom Paris - July 2004

3.9 Illustration of the Flooding schedule (check way) . . . . . 59
3.10 Illustration of the Horizontal shuffle schedule . . . . . 61
3.11 Illustration of the vertical shuffle schedule . . . . . 63

4.1 Shape of the function $f$. . . . . 72
4.2 Algorithm comparison on a regular LDPC code . . . . . 75
4.3 Algorithm comparison on an irregular LDPC code . . . . . 76
4.4 Influence of the offset value on the BER for the 3−min algorithm . . . . . 78
4.5 Comparison between the BP and the Offset 3−min algorithm with (vs $i_{\max}$) 79
4.6 Comparison between the BP and the offset 3−min algorithm (vs $E_b/N_0$) . 80
4.7 Description of a Parity Check Processor . . . . . 80
4.8 Pre-processing block (3−min algorithm) . . . . . 81
4.9 On the fly parity check scheme for $\lambda = 3$ . . . . . 82
4.10 LLR processing with the $\star$−operator ($\lambda = 3$) . . . . . 82
4.11 Synthesis block. . . . . 83
4.12 Order of operations inside a PCP . . . . . 83
4.13 Extrinsic memory saving for $\lambda$−min algorithm . . . . . 85
4.14 k=4 . . . . . 86
4.15 k=5 . . . . . 86
4.16 k=6 . . . . . 86
4.17 k=12 . . . . . 86

5.1 Synoptic of the decoder associated to the flooding scheduling. . . . . 91
5.2 System architecture of the decoder. . . . . 91
5.3 Didactic example for a parity check matrix . . . . . 93
5.4 Dual-Port Block SelectRAM . . . . . 93
5.5 Variable Memory Filling . . . . . 94
5.6 Figure 5.3 after memory banks reordering . . . . . 95
5.7 Reordering and addressing the parity check matrix for $M/P = 3$ and $P = 5$ 95
5.8 Dataflow of variable information. . . . . 96
5.9 The principle of the first accumulation bit access. . . . . 97
5.10 Organization of the check node processor memories $L$ and $P_t$. . . . . 98
5.11 How the information saved in the A-memory are used ? . . . . . 102
5.12 Waksman permutation generator of order $P$ . . . . . 106
5.13 Some permutation generator of order $N \neq 2^r$ . . . . . 107
5.14 Size of the shuffle address word. . . . . 107
5.15 The top-level architecture of the decoder . . . . . 108
5.16 Top-level Finite State Machine . . . . . 109
5.17 Top-level waveforms . . . . . 110
5.18 An illustration of a random parity-check matrix . . . . . 112
5.19 Row permutation for spreading the non-zero entries of $H$. . . . . 112

(c) Frédéric Guilloud, Télécom Paris - July 2004

5.20 Column permutation for spreading the non-zero entries of $H$. . . . . 113
5.21 Using a "NOP" variable. . . . . 113

6.1 The block diagram of the platform for encoding and decoding LDPC codes. 116
6.2 The block diagram of the platform using the Nallatech evaluation board. . 116
6.3 The Ballyinx and the Ballyblue cards from Nallatech . . . . . 117
6.4 Distribution of the intrinsic information $I_n$ . . . . . 119
6.5 Influence of the fixed point coding on the BP algorithm . . . . . 123
6.6 Influence of the intrinsic information coding . . . . . 124
6.7 Influence of the quantization on the $\lambda-$min algorithm ($\lambda = 3$). . . . . 125
6.8 Comparaison between the 3-min and the A-min* algorithm. . . . . 125
6.9 The intrinsic information is computed assuming a constant noise power . 127
6.10 The intrinsic information is computed to have a lower standard deviation 128

A.1 The binary-input gaussian-output channel (Bi-AWGN channel) . . . . . 133
A.2 Minimum BER for coded BPSK modulations. . . . . 134

B.1 An XOR operation on 2 bits $c_1$ and $c_2$. . . . . 136
B.2 Example of architecture for the 2−input LLR operator . . . . . 138
B.3 An XOR operation on $n$ bits $c_i$. . . . . 139

C.1 Parity Check Processor . . . . . 142
C.2 Waves of the PCP component . . . . . 143
C.3 Variable Processor . . . . . 144
C.4 Waves of the VP component for the `IO` state . . . . . 145
C.5 Waves of the VP component for the `ITER` state . . . . . 146
C.6 The `dpram_accu` component . . . . . 147
C.7 Example of a memory conflict (1). . . . . 147
C.8 Example of a memory conflict (2). . . . . 148

D.1 Architecture of the channel emulator component. . . . . 150
D.2 Architecture of the `protocol` component. . . . . 151

F.1 Capacité du canal gaussien pour les modulations binaires . . . . . 159
F.2 Seuils de convergence de différents codes LDPC . . . . . 160
F.3 Gain mémoire entre l'algorithme $\lambda-$min et l'algorithme BP . . . . . 161

(c) Frédéric Guilloud, Télécom Paris - July 2004

This page intentionally left blank.

# List of Tables


1.1 Reliable communications . . . . . 9
1.2 Channel parameters and Intrinsic information . . . . . 18

2.1 Different classes of LDPC codes . . . . . 26
2.2 Summary of the different LDPC encoding schemes . . . . . 34

3.1 Node processor operator as a function of the permutation network position. 53
3.2 Possible combinations for node control . . . . . 53
3.3 Memory requirements for the 3 different schedules . . . . . 64
3.4 State of the art of published LDPC platforms . . . . . 67

4.1 Different algorithm for extrinsic processing . . . . . 73
4.2 Bit node distribution degree for code $\mathcal{C}_2$. . . . . 74
4.3 Check node distribution degree for code $\mathcal{C}_2$. . . . . 75

5.1 Listing of the shuffle network addresses for $P = 3$ . . . . . 100
5.2 Encoding the parity-check matrix in the A-memory . . . . . 101
5.3 Memory size requirements . . . . . 103
5.4 FPGA RAM Block usage for several FPGA. . . . . 104

6.1 FPGA RAM Block usage for several examples. . . . . 121



(c) Frédéric Guilloud, Télécom Paris - July 2004

This page intentionally left blank.

# Listings

E.1 LDPC decoder VHDL package for constant definitions. . . . . 154
E.2 LDPC decoder VHDL package for constant definitions. . . . . 155
E.3 map summary for the synthesis 3400x1700x5. . . . . 156


(c) Frédéric Guilloud, Télécom Paris - July 2004

This page intentionally left blank.

# Notations

### LDPC codes notations

| | |
|---|---|
| $x$ | : Encoder input of length $K$: $x = (x_1, \cdots, x_K)$. |
| $c$ | : Encoder output (sent codewords) of length $N$: $c = (c_1, \cdots, c_N)$ |
| $\mathcal{C}$ | : A channel code, *i.e.* the set of all the codewords: $c \in \mathcal{C}$. |
| $\mathcal{A}_\mathcal{C}$ | : The alphabet of the code $\mathcal{C}$. |
| $y$ | : Decoder input (received codewords) of length $N$: $y = (y_1, \cdots, y_N)$ |
| $K$ | : Number of information bits. |
| $N$ | : Number of variables. |
| $M$ | : Number of parity checks $pc_m$ : $M = N - K$. |
| $R$ | : Rate of the code $\mathcal{C}$: $R = K/N$. |
| $q$ | : order of the Galois field $GF(q)$. |
| $H$ | : a parity check matrix of size $(N - K) \times N$ of the code $\mathcal{C}$ |
| $pc_m$ | : $m$th parity check of the parity check matrix $H$ |
| $vn_n$ | : $n$th variable node, attached to $y_n$ |
| $cn_m$ | : $m$th constraint or check node attached to the single parity check $pc_m$ |
| $d_c$ | : Maximum weight of the parity checks. |
| $\lambda_i$, $\tilde{\lambda}_i$ | : Proportion of non-zero values of $H$ in a column of weight $i$, proportion of variable node of degree $i$. |
| $d_v$ | : Maximum weight of the bits. |
| $\rho_i$, $\tilde{\rho}_i$ | : Proportion of non-zero values of $H$ in a row of weight $i$, proportion of check node of degree $i$. |
| $M_{c\to v}$, $M_{v\to c}$ | : Check-to-variable and variable-to-check messages of a message passing decoder. |


(c) Frédéric Guilloud, Télécom Paris - July 2004

### Mathematical expressions

| | |
|---|---|
| $\hat{.}$ | : stands for the expectation of |
| $(.)^t$ | : stands for transposition of vectors |
| $\|.\|$ | : stands for the cardinality of a set, or for the magnitude of a real value. If the argument is a parity-check equation or a Tanner graph node, it stands for the degree of the argument. |
| $\lceil . \rceil, [.]$ | : ceil and round operators. |

### Architecture notations

| | |
|---|---|
| $P$ | : Number of memory blocks or Number of parity checks that will be computed at the same time. $P$ memory blocks will contain $N/P$ variables. |
| $Q$ | : $Q = M/P$: Number of passes to be done in each iteration. |
| $S_i$ | : Series number $i$, $i \leq d_c$. A vector of $P$ variables that are accessed simultaneously in memory. |
| $i_{\max}$ | : Maximum number of decoding iterations. |
| $\lambda$ | : Number of minimum selected to update the parity checks. |
| $N_b$ | : Number of bits used to code data (magnitude only) at the output of node processors. So data are coded using $N_b + 1$ bits. |
| $w$ | : Number of bits used to code the messages in a message passing decoder. |
| $\Delta$ | : Dynamic range for magnitude values. Data are not clipped if they between $-\Delta$ and $+\Delta$. |
| $D_b$ | : Information bits throughput. |
| $R_e$ | : Edge rate: number of edges processed by clock cycle. |
| MEM | : Total memory size implemented in the decoder architecture. |
| $N_\Pi$ | : Number of elementary multiplexors ($2 \rightarrow 1$) used in the shuffle network. |
| $f_{\text{clk}}$ | : Clock frequency. |

(c) Frédéric Guilloud, Télécom Paris - July 2004

# Introduction

A decade ago at the 1993 IEEE International Conference on Telecommunications, C. Berrou and A. Glavieux presented a new scheme for channel codes decoding: the turbo codes, and their associated turbo decoding algorithm. Turbo codes made possible to get within a few tenth of dB away from the Shannon limit, for a bit error rate of $10^{-5}$.

Beside the major impact that turbo codes have had on telecommunication systems, they also made researchers realize that other capacity approaching codes existed. Hence, the low-density parity-check (LDPC) codes invented in the early sixties by Robert Gallager, have been resurrected in the mid nineties by David MacKay. In fact, when LDPC codes have been invented, their decoding was too complicated for the technology, and so they have been forgotten. Like turbo codes, LDPC codes can get very close to the Shannon limit by the mean of an iterative decoding.

An LDPC code is a linear block code defined by a very sparse parity-check matrix. The decoding algorithm is easy to understand thanks to the factor graph representation of the code: each symbol of the codeword is interrelated with the constraints that the symbols must satisfy in order to form a valid codeword. During the iterative decoding algorithm process, *messages* are exchanged between the symbols and the constraints on the edges of the graph. These message are in fact the probabilities for a symbol or a constraint to be a given value.

The asymptotic performance of LDPC codes, which are nowadays well understood, leads LDPC codes to be used as serious competitors to turbo codes. The choice of an LDPC code as a standard for the second Satellite Digital Video Broadcasting normalization (DVB-S2) makes now the architecture of LDPC decoders a real challenge. Although the decoding algorithm of LDPC codes is easy to understand, the design of an LDPC decoder is a real challenge. It can be split into three part: the node processors, the interconnection network and the memory management. The main bottleneck in the design of an LDPC decoder lies in the memory requirement and in the complexity of the node processors. A first main contribution of this thesis is a global framework which enables an analysis of the state of the art. Our second main contribution is a sub-optimal algorithm and its associated


(c) Frédéric Guilloud, Télécom Paris - July 2004

architecture which make possible to lower the memory requirement without any significant loss of performance. The third main contribution of this thesis is a generic LDPC decoder architecture within a platform for running fast simulations. These original contributions will be proposed throughout this thesis which is organized as in the following.

**The first chapter** is a presentation of the error correcting code decoding in general point of view. The iterative algorithm which will be used for LDPC code decoding is also derived.

**The second chapter** is dedicated to the presentation of the LDPC codes. The different parameters which characterizes the LDPC codes are presented. Then the encoding of LDPC codes and their decoding performance are discussed. As the BPSK will be used throughout this thesis, the particular case of the binary-input additive white gaussian channel modulated capacity is also presented.

**The third chapter** gives an overview of the state of the art architectures within a new unified view of the LDPC decoder architecture. Our contribution is a new theoretical framework for LDPC decoder architectures which tries to encompass all the decoder implementations, and which moreover illuminates new architectures, some of them being of particular interest. These results are recent and have not been published yet. They will hardly be used in the following chapters.

**The fourth chapter** presents a study of the check nodes and its associated memory. From this work, we derived a new sub-optimal algorithm, named the $\lambda-$min algorithm. This new algorithm aims at reducing both the complexity of the node processors and the memory requirements used in non-parallel decoder architectures. Our algorithm has a good performance complexity trade-off since it significantly reduces the complexity of the check node processors, while keeping the bit error rate close to the optimal one. A new architecture implementing the $\lambda-$min algorithm, and more generally the concepts of compacting the edge-memory in non-parallel architecture are also presented.

**The fifth chapter** proposes a generic implementation of an LDPC decoder. We designed and implemented an LDPC decoder which can be tuned for any LDPC code. Moreover, once the synthesis has been performed, the decoder can also decode every LDPC code of a given length and of a given minimum rate.

**In the last chapter,** a complete platform for LDPC codes and architecture studying is presented. This platform is divided into two parts: a software part, which enables both floating and fixed point simulation of any LDPC code, LDPC codes encoding using an efficient low complexity encoder, and random generation of LDPC codes with a given

(c) Frédéric Guilloud, Télécom Paris - July 2004

variable or checks degree distribution; the second part is made of a hardware generic decoder implemented in a Virtex E 1000 FPGA from Xilinx, running on a Nallatech evaluation board. This platform has already been used to validate some quantization effects on the LDPC decoding performance, on the intrinsic information processing, and finally on the $\lambda-$min algorithm as well as its implementation.

**A conclusion and some perspectives** are finally given at the end of this thesis.

(c) Frédéric Guilloud, Télécom Paris - July 2004

This page intentionally left blank.

# Chapter 1

# Channel coding

**Summary:**

*This first chapter introduces the channel code decoding issue and the problem of optimal code decoding in the case of linear block codes. First, the main notations used in the thesis are presented and especially those related to the graph representation of the linear block codes. Then the optimal decoding is discussed: it is shown that under the cycle-free hypothesis, the optimal decoding can be processed using an iterative algorithm. Finally, the performance of error correcting codes is also discussed.*

## 1.1 Optimal decoding

### 1.1.1 Shannon theorem for channel coding

Communication over noisy channels can be improved by the use of a channel code $\mathcal{C}$, as demonstrated by C. E. Shannon (Shannon 1948) for its famous channel coding theorem:

> "Let a discrete channel have the capacity $C$ and a discrete source the entropy per second $\mathcal{H}$. If $\mathcal{H} \leq C$ there exists a coding system such that the output of the source can be transmitted over the channel with an arbitrarily small frequency of errors (or an arbitrarily small equivocation). If $\mathcal{H} > C$ it is possible to encode the source so that the equivocation is less than $\mathcal{H}$."

This theorem states that below a maximum rate $R$, which is equal to the capacity of the channel, it is possible to find error correction codes to achieve any given probability of error. Since this theorem does not explain how to make such a code, it has been the kick-off for a lot of activities in the coding theory community. When Shannon announced his theory in the July and October issues of the Bell System Technical Journal in 1948, the largest communications cable in operation at that time carried 1800 voice conversations. Twenty-five years later, the highest capacity cable was carrying 230000 simultaneous


(c) Frédéric Guilloud, Télécom Paris - July 2004

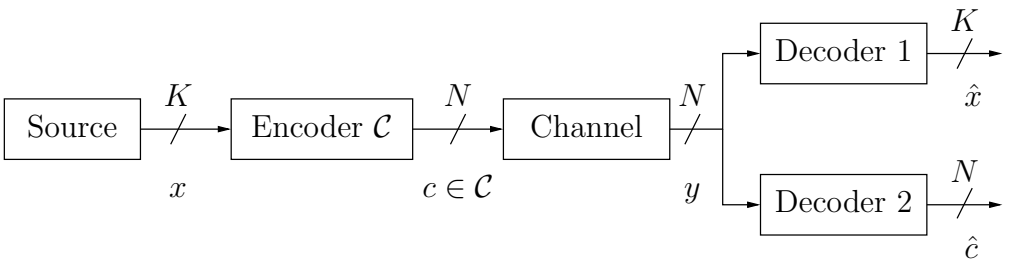



Figure 1.1: Basic scheme for channel code encoding/decoding.

conversations. Today a single optical fiber as thin as a human hair can carry more than 6.4 million conversations (Lucent 2004).

In the quest of capacity achieving codes, the performance of the codes is measured by their gap to the capacity. For a given code, the smallest gap is obtained by an optimal decoder: the maximum a-posteriori (MAP) decoder.

Before dealing with the optimal decoding, some notations within a model of the communication scheme are presented hereafter.

### 1.1.2 Communication model

Figure 1.1 depicts a classical communication scheme. The source block delivers information by the mean of sequences which are row vectors $x$ of length $K$. The encoder block delivers the codeword $c$ of length $N$, which is the coded version of $x$. The code rate is defined by the ratio $R = K/N$. The codeword $c$ is sent over the channel and the vector $y$ is the received word: a distorted version of $c$.

The matched filters, the modulator and the demodulator, and the synchronization is supposed to work perfectly. This is the reason why they do not appear on figure 1.1. Hence, the channel is represented by a discrete time equivalent model.

The channel is a non-deterministic mapper between its input $c$ and its output $y$. We assume that $y$ depends on $c$ via a conditional probability density function (pdf) $p(y|c)$. We assume also that the channel is memoryless: $p(y|c) = \prod_{n=1}^{N} p(y_n|c_n)$. For example, if the channel is the binary-input additive white gaussian noise (BI-AWGN), and if the modulation is a binary phased shift keying (BPSK) modulation with the $0 \rightarrow +A$, $1 \rightarrow -A$ mapping, we have:

$$p(y_n|c_n) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\left(y_n - (-1)^{c_n}\sqrt{E_s}\right)^2}{2\sigma^2}\right) \tag{1.1}$$

where $E_s = A^2$ is the energy of the symbol sent over the channel.

On figure 1.1, two types of decoder are depicted: decoders of type 1 have to compute the best estimation $\hat{x}$ of the source word $x$; decoders of type 2 compute the best estimation $\hat{c}$ of the sent codeword $c$. In this case, $\hat{x}$ is extracted from $\hat{c}$ by a post processing (reverse processing of the encoding) when the code is non-systematic. Both decoders can perform two types of decoding:

**Soft decoding:** The output samples $y_n$ of the channel are not decided: they are the inputs of the decoder. Using the channel specifications, the decoder computes the probability for each $y_n$ to be each one of the code-alphabet element denoted $y_{di} \in \mathcal{A}_{\mathcal{C}}$: $\{\Pr(y_n = y_{di}),\ 0 \leq i \leq |\mathcal{C}|\}$. Then, the decoding process will base its decisions on the value of these probability. The output of the decoder is both the decided word $y_d = (y_{d1}, \cdots, y_{dN}),\ y_{di} \in \mathcal{A}_{\mathcal{C}}$, where $\mathcal{A}_{\mathcal{C}}$ denotes the alphabet of the code symbols, and the probability of each decided symbol $\Pr(y_n = y_{dn})$.

**Hard decoding:** The output samples $y_n$ of the channel are decided: each of them is associated with the most probable code-alphabet element. Then a processing is performed on $y_d = (y_{d1}, \cdots, y_{dN})$ to try to detect and correct the transmission errors. This processing is made without using the knowledge of the probability set $\{\Pr(y_n = y_{di}),\ 0 \leq i \leq |\mathcal{C}|\}$.

### 1.1.3 Optimal decoding

#### Optimal Word decoding

The aim of the decoder is to find the codeword $\hat{c}$ which is the most probable to have been sent over the channel, based on the channel output $y$, and on the knowledge of the code:

$$\hat{c} = \underset{c' \in \mathcal{C}}{\arg\max} \Pr\left(c = c'|y\right) \tag{1.2}$$

This is the word maximum a posteriori (W-MAP) decoder. The knowledge of the code should appear in the conditional probabilities but will be omitted hereafter to simplify the notations.

Using Bayes' rule, the posterior probabilities $\Pr(c = c'|y)$ are expressed by:

$$\Pr(c|y) = \frac{p(y|c)\Pr(c)}{p(y)} = \frac{p(y|c)\Pr(c)}{\sum_{c \in \mathcal{C}} p(y|c)\Pr(c)} \tag{1.3}$$

If the a priori probabilities $\Pr(c)$ are identical (the source is equally probable), (1.2) can be expressed as:

$$\hat{c} = \underset{c' \in \mathcal{C}}{\arg\max}\, p(y|c = c') \tag{1.4}$$

which is named word maximum likelihood (W-ML) decoding. $p(y|c)$ is called the likelihood function when $y$ is fixed and it is a conditional pdf when $c$ is fixed.

The only way to achieve an optimal W-MAP decoder is to test each codeword, *i.e.* $2^K$ for a binary source for example.

The W-MAP and W-ML decoders are two equivalent and optimal decoders if the source is equally probable. The Viterbi algorithm (VA) (Viterbi 1967; Forney 1973) is an efficient W-ML decoder which eliminates many operations and leads to the best word (frame) error rate (FER), providing that the code has a trellis representation and that the source words are equally likely to happen.

#### Optimal Symbol Decoding

If the symbol (or bit) error rate (BER) is concerned, the bit maximum a posteriori (B-MAP) decoder and the bit maximum likelihood (B-ML) decoders give an estimation of the codeword symbols $c_n$:

$$\hat{c_n} = \underset{c' \in \mathcal{A}_\mathcal{C}}{\arg\max} \Pr\left(c_n = c' | y\right) \tag{1.5}$$

$$\hat{c_n} = \underset{c' \in \mathcal{A}_\mathcal{C}}{\arg\max}\, p(y | c_n = c') \tag{1.6}$$

The BCJR algorithm (Bahl et al. 1974), also called Forward-Backward algorithm is capable of computing a posteriori probabilities of the code symbols, providing that the code has a trellis representation. It is a B-MAP decoder which leads to the best BER. The Soft output Viterbi Algorithm (SOVA) (Hagenauer and Hoeher 1989) is a suboptimal B-MAP algorithm which requires only a forward processing in the trellis.

## 1.2 Performance of error correcting codes

The performance of error correcting codes are compared with each other by referring to their gap to the Shannon limit, as mentioned in section 1.1.1. This section aims at defining exactly what is the Shannon limit, and what can be measured exactly when the limit to the Shannon bound is referred to. It is important to know exactly what is measured since a lot of "near Shannon limit" codes have been discovered now.

The results hereafter are classical in the information theory and may be found in a lot of references. Yet, the first part is inspired by the work of (Schlegel 1997).

### 1.2.1 The Shannon bound

The Shannon capacity of the band-limited AWGN channel is given by (Shannon 1948):

$$C = B \log_2(1 + \mathrm{SNR}) \text{ [bit/s]} \tag{1.7}$$

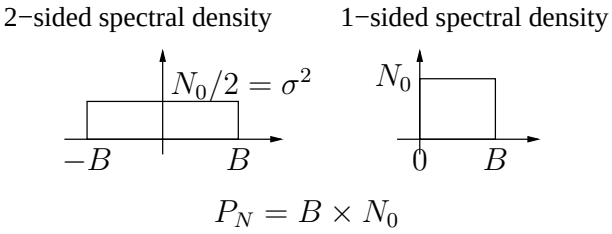



Figure 1.2: AWGN noise power spectral density

Table 1.1: Reliable communications

| | |
|---|---|
| $R_I < C$ | reliable communication: $\lim_{N \to +\infty} \text{FER} = 0.$ |
| $R_I > C$ | unreliable communication. |

where $B$ is the channel bandwidth, and where SNR is the ratio between the transmitted signal power $P_S$ and the channel noise power $P_N$: $\text{SNR} = \dfrac{P_S}{P_N}$. Note that:

$$P_N = N_0 B \tag{1.8}$$

$$P_S = R_I E_b \tag{1.9}$$

where $N_0$ is the one sided noise power spectrum density (figure 1.2) and where $E_b$ is the energy per information bit. The information rate $R_I$ is defined by:

$$R_I = \frac{R \log_2(\mathbb{M})}{T_S} \text{ [bit/s]} \tag{1.10}$$

where $R$ is the code rate, $\mathbb{M}$ is the size of the constellation of the modulation and $T_S$ is the symbol time duration.

Using equations (1.10) to (1.9) into (1.7) yields:

$$C = B \log_2(1 + \frac{P_S}{P_N}) \text{ [bit/s]} \tag{1.11}$$

$$= B \log_2(1 + \frac{R_I E_b}{N_0 B}) \text{ [bit/s]} \tag{1.12}$$

The Shannon theorem can be summarized as in table 1.1. But it can also be stated using the spectral efficiency $\eta$:

$$\eta = \frac{\text{information rate}}{\text{channel bandwidth}} = \frac{R_I}{B} \tag{1.13}$$

Thus, the maximal information rate defines a maximum spectral efficiency $\eta_{\max}$:

$$\eta_{\max} = \frac{R_{I\max}}{B} \tag{1.14}$$

(c) Frédéric Guilloud, Télécom Paris - July 2004

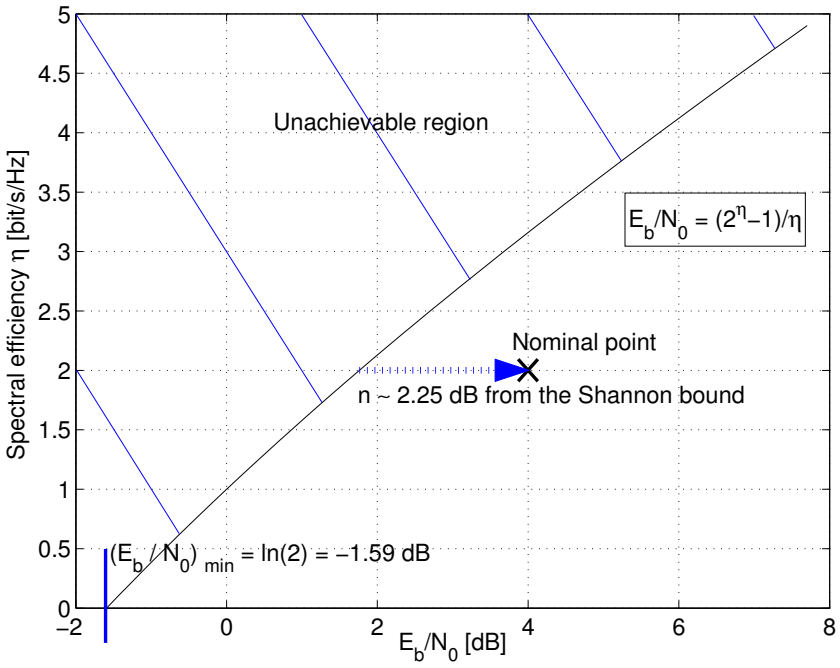



Figure 1.3: The Shannon bound: the spectral efficiency versus $E_b/N_0$ for AWGN channels

The maximal information rate is equal to the capacity (by definition). So, using (1.12), it can be expressed by:

$$R_{I\max} = B\log_2(1+\frac{R_{I\max}E_b}{N_0B}) \tag{1.15}$$

which yields to:

$$\frac{R_{I\max}}{B} = \log_2(1+\frac{R_{I\max}E_b}{N_0B}) \tag{1.16}$$

$$\eta_{\max} = \log_2(1+\eta_{\max}\frac{E_b}{N_0}) \tag{1.17}$$

$$\left(\frac{E_b}{N_0}\right)_{\min} = \frac{2^{\eta_{\max}}-1}{\eta_{\max}} \tag{1.18}$$

Equation (1.18) is called the Shannon bound and is plotted on figure 1.3, where an example of a transmission scheme having a spectral efficiency of 4 [bit/Hz] at an $E_b/N_0 = 4$ dB is found to be about 2.25 dB away from the Shannon bound.

### 1.2.2 The AWGN capacity

The bandwidth efficiency is not very useful to study only the properties of error correcting codes because it takes into account all the signals used in the transmission. According to the Nyquist theorem, a real signal which has a bandwidth $B$ can be sampled at a rate

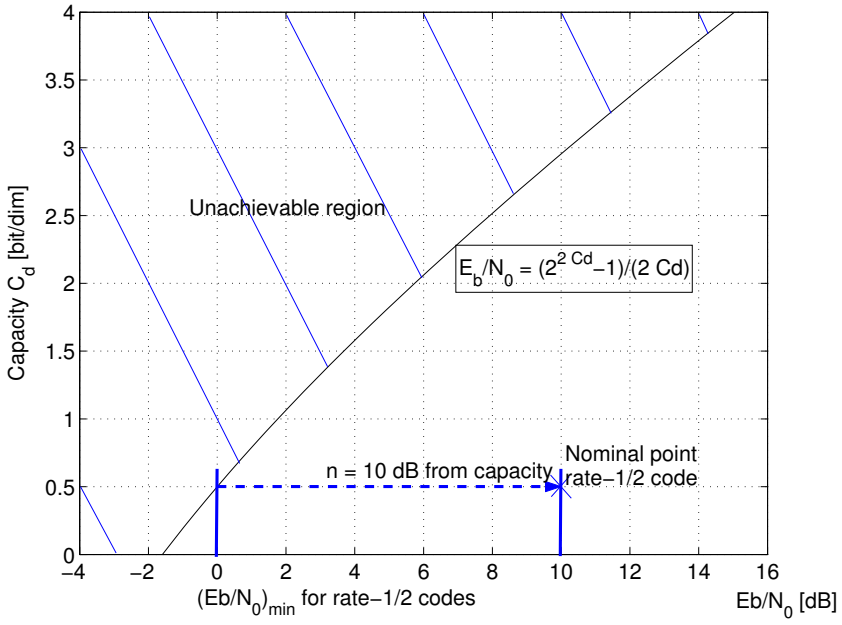



Figure 1.4: Capacity in [bit/dim] as a function of $E_b/N_0$ for the AWGN channel. Example of a 10 dB gap to capacity for a particular rate$-1/2$ code

of $2B$ samples per second without any inter-symbol interference. So the $2B$ samples are "independent" and they are carried on $2B$ signal dimensions [dim] per second. The rate per signal dimension is defined by $R_{Id} = R_I/(2B)$ [bit/dim] and its maximum by the capacity $C_d = C/(2B)$. Thus, (1.12) gives (Wozencraft and Jacobs 1965): [1]

$$C_d = \frac{B}{2B}\log_2(1+\frac{2R_IE_b}{2N_0B}) \tag{1.19}$$

$$= \frac{1}{2}\log_2(1+\frac{2R_{Id}E_b}{N_0}) \tag{1.20}$$

$$= \frac{1}{2}\log_2(1+\frac{2C_dE_b}{N_0}) \tag{1.21}$$

and thus:

$$\frac{E_b}{N_0} = \frac{2^{2C_d}-1}{2C_d} \tag{1.22}$$

This expression is depicted on figure 1.4, where an example of a 10 dB gap to capacity is illustrated.

The capacity in expression (1.21) is by definition the maximization of the mutual information between the input and the output of the AWGN channel over the input channel probabilities. The maximum occurs for a gaussian distribution of the channel

[1] For a complex signal, the Nyquist theorem applies with $B$ samples per second and hence: $C_d = \log_2(1+C_d\frac{E_b}{N_0})$

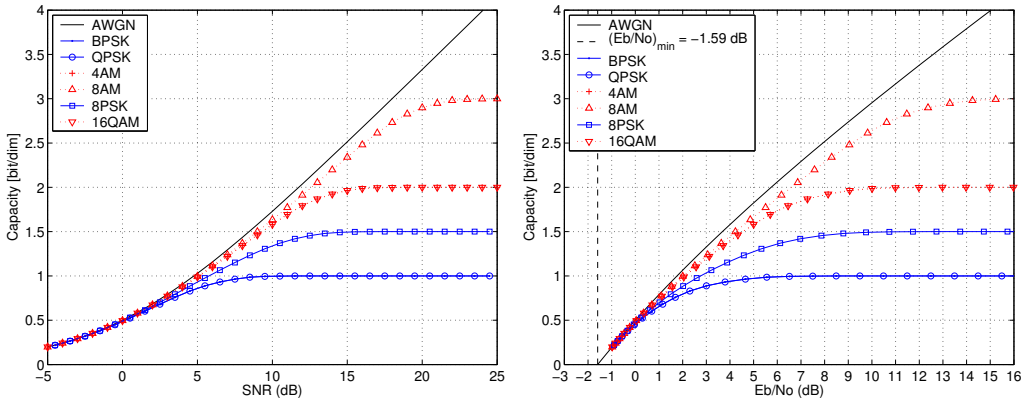



Figure 1.5: Constellation constrained capacity of an AWGN channel as a function of the signal to noise ratio and of $E_b/N_0$: SNR$= 2R\frac{E_b}{N_0}$

input. So for the discrete-input gaussian-output channel, the capacity should not be reached. A pseudo-capacity taking into account the discrete input coming from the binary modulation can be derived so as to separate in the gap to $C$ what is due to the code from what is due to the discrete input. In (Ungerboeck 1982), a constellation-constrained capacity of an AWGN channel is derived:

$$C_d^* = \log_2(\mathbb{M}) - \frac{1}{\mathbb{M}} \sum_{k=1}^{\mathbb{M}} \mathcal{E} \left\{ \log_2 \sum_{i=1}^{\mathbb{M}} \exp\left( - \frac{|a^k + w - a^i|^2 - |w|^2}{2\sigma^2} \right) \right\} \tag{1.23}$$

where the symbols of the input constellation are denoted by $\{a^1, \cdots, a^{\mathbb{M}}\}$ and where $\mathcal{E}$ denotes the expectation over the normally distributed noise variable $w$ with variance $\sigma^2$, if $w$ is real. Based on (Morelos-Zaragoza ), these capacities have been numerically computed and are plotted on figure 1.5.

## 1.3 Decoding of linear block codes

### 1.3.1 Definitions

A linear block code $\mathcal{C}$ defined over the Galois field $GF(q)$ is a $k$-dimension vector subspace of $GF(q)^N$. The code $\mathcal{C}$ can be defined by the list of all the codewords:

$$\mathcal{C} = \{c^{(i)}, i \in \{0, \cdots, 2^K - 1\}\}, \tag{1.24}$$

which is unique. It can be alternatively defined by a vector base $\mathcal{B}_\mathcal{C}$ of $K$ independent codewords, $\{c^{(i)}, i \in \{0, \cdots, K-1\}\}$, which is not unique.

The vector base $\mathcal{B}_\mathcal{C}$ has itself many useful and equivalent representations:

- its generator matrix $G$, whose rows are the vectors of the base $\mathcal{B}_\mathcal{C}$ (so $G$ is an $K \times N$ matrix): $c \in \mathcal{C} \Leftrightarrow \exists u \in GF(q)^K / c = uG$.

- its parity-check matrix $H$, which is a $(N-K) \times N$ matrix with elements in $GF(q)$: $\mathcal{C} = \{c^{(i)} / c^{(i)}.H^t = 0\}$. The parity check matrix is a concatenation of $M = (N-K)$ rows denoted by $pc_m$. Each row $pc_m$ of $H$ is a parity check equation on some bits of the codeword. The bits implied in the parity-check $pc_m$ are the non-zero entries of the $m-$th row of $H$. $H$ is the orthogonal complement of $\mathcal{C}$: $GH^t = 0$.

- its Tanner graph (Tanner 1981). A Tanner graph is a bipartite graph. A bipartite graph is a graph where the elements of a first class can be connected to the elements of a second class, but not to the same class. In a Tanner graph for binary block codes, the elements of the first class are the variable nodes denoted by $vn_n$ and the elements of the second class are the check nodes denoted by $cn_m$. Each variable node $vn_n$ is associated with one code symbol $c_n$ and each check node $cn_m$ is associated with the $m$-th parity check constraint $pc_m$ of $H$. A variable node $vn_n$ is connected to a check node $cn_m$ if and only if $H(m,n)$ has a non-zero entry.

Figure 1.6 resume the different ways of defining a block code with the example of the hamming$-[7,4]$ code in $GF(2)$.

The Tanner graph representation of error correcting codes is very useful since their decoding algorithms can be explained by the exchange of information along the edges of these graphs. The notations related to the Tanner and an important hypothesis will be hereafter detailed.

Let $\mathcal{N}(m)$ be the set of bits which are implied in the $m$-th parity-check constraint: $\mathcal{N}(m) = \{c_n | H(m,n) = 1\}$. Let $\mathcal{N}(m) \backslash n$ denote the same set but with bit $c_n$ excluded. Let also $\mathcal{M}(n)$ be the set of the parity check constraints in which the bit $c_n$ is implied: $\mathcal{M}(n) = \{pc_m | H(m,n) = 1\}$. Let $\mathcal{M}(n) \backslash m$ denote the same set but with parity check $pc_m$ excluded. Hence $c_n$ is implied in $|\mathcal{M}(n)|$ parity check constraints. Let $\phi_{n,k}$ denotes the $k$-th parity check constraint of $\mathcal{M}(n)$ with bit $c_n$ excluded, $k \in \{1, \cdots, |\mathcal{M}(n)|\}$. Figure 1.7 resumes the different notations on a practical example. An important hypothesis related to the Tanner graph representation is the cycle free hypothesis.

**Hypothesis 1 (cycle free graph)** *The bipartite graph of the code $\mathcal{C}$ is cycle free. A graph is cycle free if it contains no path which begins and ends at the same bit node without going backward. When the graph is not cycle free, the minimum cycle length is called the girth of the graph.*

Figure 1.8 depicts an example of cycle free graph and an example of graph with girth 4 and 6. When a bipartite graph is cycle free, it has a tree representation: each variable node and each check node appear exactly once in the tree.

(a)

$$\mathcal{C} = \left\{ \begin{array}{l} 0000000,0001011,0010110,0011101, \\ 0100111,0101100,0110001,0111010, \\ 1000101,1001110,1010011,1011000, \\ 1100010,1101001,1110100,1111111 \end{array} \right\}$$

(b) $G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$ (c) $H = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$

(d)

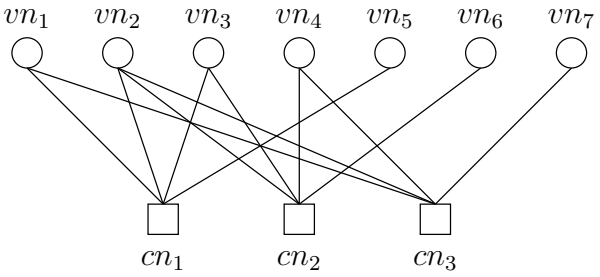



Figure 1.6: Different ways of describing the same code. Example of the hamming$-[7,4]$ code in $GF(2)$: (a) the list of the codewords ; (b) an example of a generator matrix ; (c) an example and of a parity check matrix ; (d) the bipartite graph based on the parity check matrix of (c).

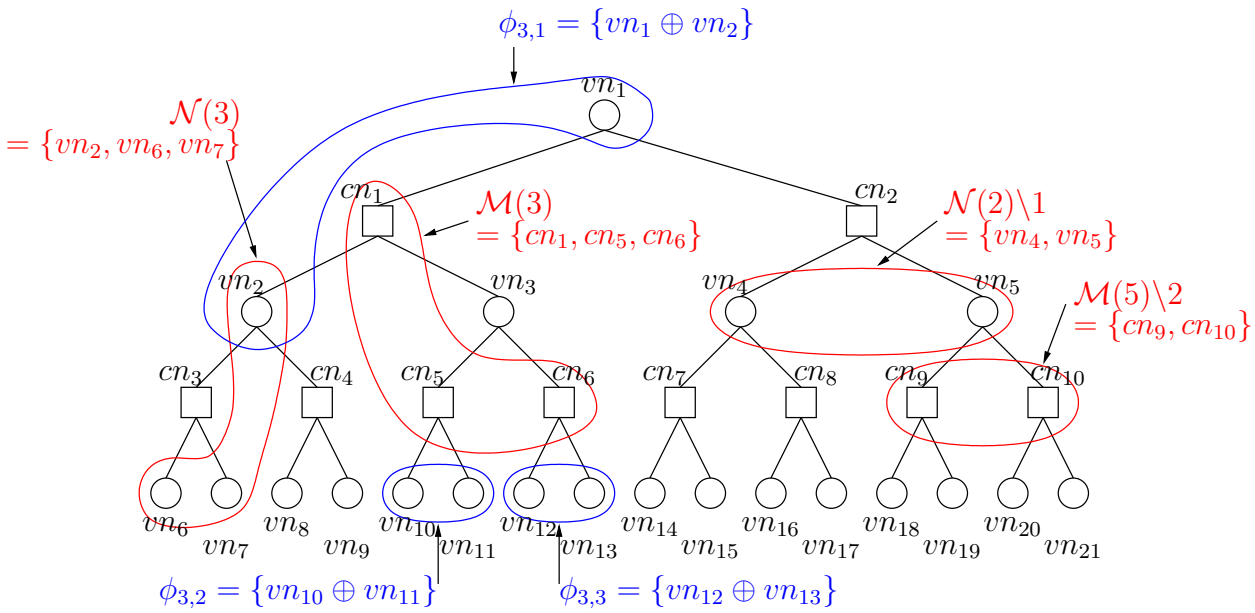



Figure 1.7: The notations related to the bipartite graph of a code.

### 1.3.2 Optimal decoding of binary block codes

As mentioned is section 1.1.3, the best possible FER is reached using the VA: it requires a trellis representation of the code. Although block codes have a trellis representation (Bahl et al. 1974; Wolf 1978), the complexity of their trellis increases exponentially with the size of the code. Since such representations lead to unrealistic algorithm implementations, many suboptimal decoding algorithms have been proposed to reduce the codewords set to be compared to.

In this thesis, only the optimal symbol error rate will be studied. Similarly, it could be reached using the BCJR algorithm with a trellis representation of the code. But here again, the trellis would be too complex for codes with many parity-check equations. Hereafter, we derive equation (1.5) inspired by (Barry 2001), for binary block codes. So the symbols will equivalently be named bits.

$$\begin{aligned} \hat{c_n} = 0 \quad \text{if} \quad & \Pr(c_n = 0|y) > \Pr(c_n = 1|y) \\ \hat{c_n} = 1 \quad \text{if} \quad & \Pr(c_n = 0|y) < \Pr(c_n = 1|y) \end{aligned} \tag{1.25}$$

The received word $y = (y_1, \cdots, y_N)$ can be split into two sets: $y_n$ and $(y_{n' \neq n})$. Then the

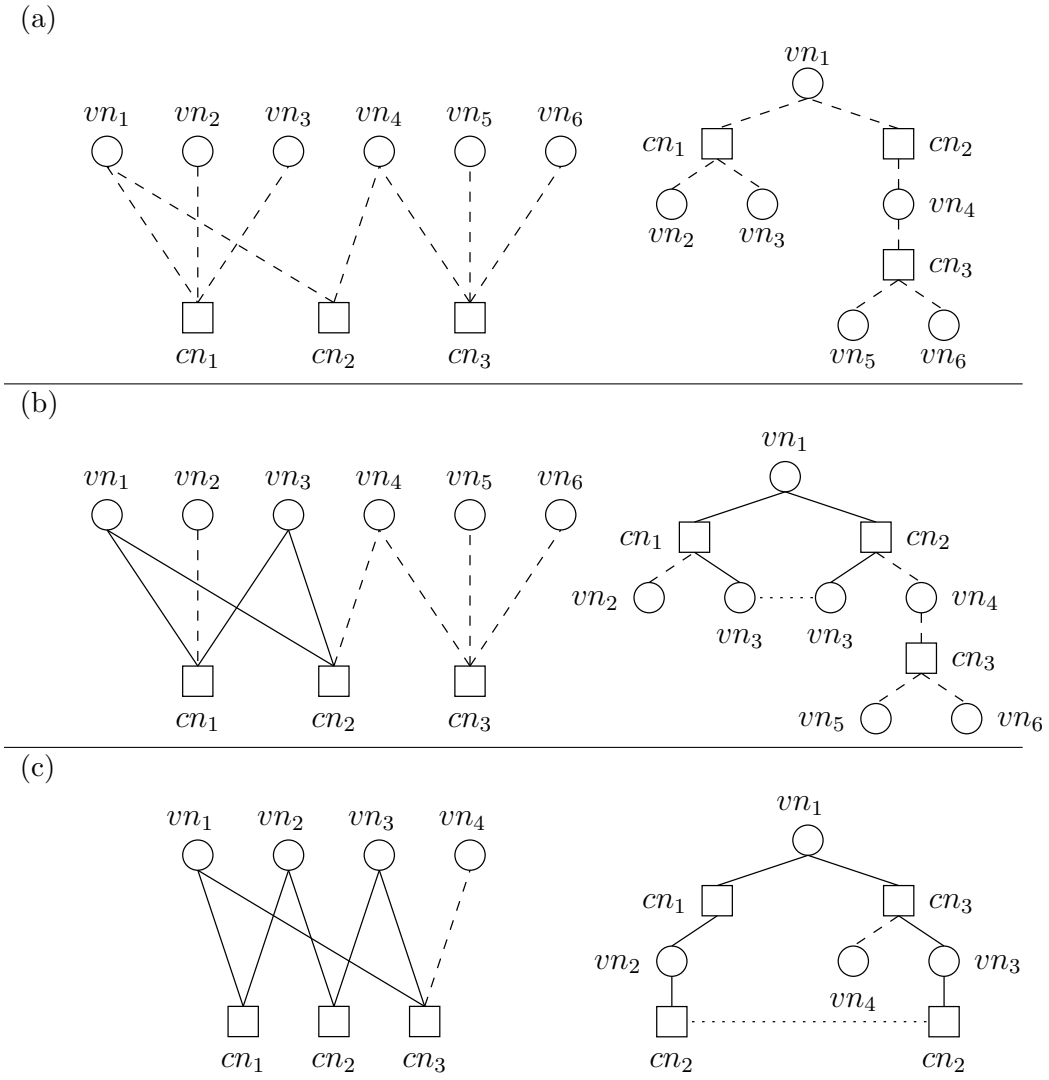



Figure 1.8: Some graph (left side) and trees (right side) (a) without cycles, (b) with a cycle of length 4, (c) with a cycle of length 6. Cycles are depicted with solid lines, whereas others connections are depicted in dashed lines. The dotted lines are links between identical nodes.

(c) Frédéric Guilloud, Télécom Paris - July 2004

probabilities from (1.25) can be modified as in the following:

$$\begin{aligned}
\Pr(c_n|y) &= \Pr\left(c_n|y_n, y_{n'\neq n}\right) && (1.26)\\
&= \frac{p(c_n, y_n, y_{n'\neq n})}{p(y_n, y_{n'\neq n})} && (1.27)\\
&= \frac{p(y_n|c_n, y_{n'\neq n})p(c_n, y_{n'\neq n})}{p(y_n|y_{n'\neq n})p(y_{n'\neq n})} && (1.28)\\
&= \frac{p(y_n|c_n)\Pr\left(c_n|y_{n'\neq n}\right)}{p(y_n|y_{n'\neq n})} && (1.29)\\
&\quad \text{because given } c_n,\ y_n \text{ is independent of } y_{n'\neq n}
\end{aligned}$$

### Log-Likelihood Ratio

From equations (1.25), we have:

$$\hat{c}_n = 0 \Rightarrow \frac{\Pr(c_n = 0|y)}{\Pr(c_n = 1|y)} > 1 \Rightarrow \log\frac{\Pr(c_n = 0|y)}{\Pr(c_n = 1|y)} > 0 \quad (1.30)$$

$$\hat{c}_n = 1 \Rightarrow \frac{\Pr(c_n = 0|y)}{\Pr(c_n = 1|y)} < 1 \Rightarrow \log\frac{\Pr(c_n = 0|y)}{\Pr(c_n = 1|y)} < 0 \quad (1.31)$$

So an equivalent decision rule for optimal BER for binary block codes is to calculate the sign of:

$$\log\frac{\Pr(c_n = 0|y)}{\Pr(c_n = 1|y)} \quad (1.32)$$

which give, using (1.30):

$$\underbrace{\log\frac{\Pr(c_n = 0|y)}{\Pr(c_n = 1|y)}}_{T_n} = \underbrace{\log\frac{p(y_n|c_n = 0)}{p(y_n|c_n = 1)}}_{I_n} + \underbrace{\log\frac{\Pr\left(c_n = 0|y_{n'\neq n}\right)}{\Pr\left(c_n = 1|y_{n'\neq n}\right)}}_{E_n} \quad (1.33)$$

where

- $T_n$ is the overall information of the bit $n$. It is the logarithm of the ratio between the two a-posteriori probabilities on the bit $n$. The sign of $T_n$ enables the estimation of $c_n$ and the magnitude of $T_n$ is the reliability of the decision.
- $I_n$ is the intrinsic information of the bit $n$. It is related to the received value $y_n$ and to the channel parameters. Table 1.2 lists some examples for different channel models when the BPSK modulation is used.
- $E_n$ is the extrinsic information of the bit $n$. It is the improvement of information we gain by considering the fact that the coded symbols respect the parity check constraints. This improvement does not necessarily mean an increase of the reliability $|T_n|$.

(c) Frédéric Guilloud, Télécom Paris - July 2004

Table 1.2: Channel soft value for different channel parameters in the case of a BPSK modulation.

| Channel | Parameter | $I_n$ |
|---|---|---|
| Gaussian | noise variance $\sigma^2$ | $2E_s y_n/\sigma^2$ |
| Binary Symmetric | cross-over probability $p$ | $y_n \log((1-p)/p)$ |
| Laplace | noise $\nu$ | $\frac{\lvert y+A\rvert-\lvert y-A\rvert}{\nu}$ |

**Optimal decoding using cycle free hypothesis**

The probability that $c_n = 1$ is the probability that the parity of all the other bits implied in the parity check is equal to one, so that the parity check equation could be satisfied (even parity).

$$\Pr\left(c_n = 1 | y_{n' \neq n}\right) = \Pr\left(\phi_{n,1} = 1, \cdots, \phi_{n,|\mathcal{M}(n)|} = 1 | y_{n' \neq n}\right) \tag{1.34}$$

Under the assumption of hypothesis 1, the events "$\phi_{n,k} = 1$" for $k \in \{1, \cdots, |\mathcal{M}(n)|\}$ are conditionally independent given $y_{n' \neq n}$. The same statement holds for the bits implied in the parity checks $\phi_{n,k}$. The interpretation of this independence in the graph is that for cycle free graphs, the parity checks constraints $\phi_{n,k}$ are in disjointed trees. So assuming hypothesis 1 and combining equation (1.34) with the expression of the extrinsic information of bit $n$ yield:

$$\begin{aligned} E_n &= \log \frac{\prod_{k=1}^{|\mathcal{M}(n)|} \Pr\left(\phi_{n,k} = 0 | y_{n' \neq n}\right)}{\prod_{k=1}^{|\mathcal{M}(n)|} \Pr\left(\phi_{n,k} = 1 | y_{n' \neq n}\right)} && (1.35) \\ &= \sum_{k=1}^{|\mathcal{M}(n)|} \underbrace{\log \frac{\Pr\left(\phi_{n,k} = 0 | y_{n' \neq n}\right)}{\Pr\left(\phi_{n,k} = 1 | y_{n' \neq n}\right)}}_{E_{n,k}} && (1.36) \end{aligned}$$

$E_n$ is then the sum over $k$ of the $E_{n,k}$ which are the information given by each of the parity-check constraints $\in \mathcal{M}(n)$ on the bit $c_n$. Let $c_{n,k,l}$ be the $l$-th bit implied in the parity check $\phi_{n,k}$ of degree $|\phi_{n,k}|$. Then, applying equation (B.29) of appendix B.2 to the parity check $\phi_{n,k}$ yields to:

$$E_{n,k} = 2 \tanh^{-1} \prod_{l=1}^{|\phi_{n,k}|} \tanh \frac{1}{2} \log \frac{\Pr\left(c_{n,k,l} = 0 | y_{n' \neq n}\right)}{\Pr\left(c_{n,k,l} = 1 | y_{n' \neq n}\right)} \tag{1.37}$$

Hence, the total information of the bit $c_n$ is completely derived, provided the derivation of $\log \frac{\Pr(c_{n,k,l}=0|y_{n'\neq n})}{\Pr(c_{n,k,l}=1|y_{n'\neq n})}$ by:

$$T_n = I_n + \sum_{k=1}^{|\mathcal{M}(n)|} E_{n,k} \tag{1.38}$$

### 1.3.3 The iterative algorithm

The derivation of $\log \frac{\Pr(c_{n,k,l}=0|y_{n'\neq n})}{\Pr(c_{n,k,l}=1|y_{n'\neq n})}$ of equation (1.37) is in fact the same problem as the derivation of $T_n$, but considering the bits $c_{n,k,l}$ at the top of the subtrees resulting from the erasure of bit $c_n$. This recursion has to be processed until the leaves of the tree.

A summary of the operations using the practical example of figure 1.7 is depicted on figure 1.9, where:

$$E_{n,m} = f(T_{n,m'}) = 2\tanh^{-1} \prod_{m'\in\mathcal{N}(m)\setminus m} \tanh \frac{T_{n,m'}}{2} \tag{1.39}$$

and where $T_{n,m}$ denotes the information which is sent by a variable node $vn_n$ to its connected check node $cn_m$. Note also that:

$$T_{n,m} = T_n - E_{n,m} \tag{1.40}$$

The partial results $T_{n,m}$ and $E_{n,m}$ are called messages, since they are transmitted from nodes to nodes.

In this example, the total information $T_1$ of the bit node $vn_1$ is calculated in 4 steps, corresponding to the depth between the leaves of the tree and the variable node considered. For the calculation of $T_3$, some partial results of the calculation of $T_1$ are reused. The others are not replaced by different results but by messages in the opposite way. This means that the calculation of $T_1$ and $T_3$, and by generalization all the $T_n$, can be processed in parallel: the cycle free hypothesis let them be all independent.

Then, if one node of the graph is considered, as the $N$ variables are processed at the same time, this particular node will be implied in all the steps (in fact the even ones for a variable node, and the odd ones for check nodes). But a complex problem would be: when should node $vn_n$ or $cn_m$ process the information $E_{n,m}$ or $T_{n,m}$ ?

In order to simplify the problem, a general processing behaviour for the nodes is to let them process all the messages all the time, *i.e.* at each time when one or more incoming messages on the variable nodes $vn_n$ (resp. check node $cn_m$) has changed, the variable node (resp. check node) process all the possible outgoing messages. The nodes behave like independent local processors: they don't have to process conditionally to the processing of other nodes. The check (resp. variable) node processing is also called check

(c) Frédéric Guilloud, Télécom Paris - July 2004

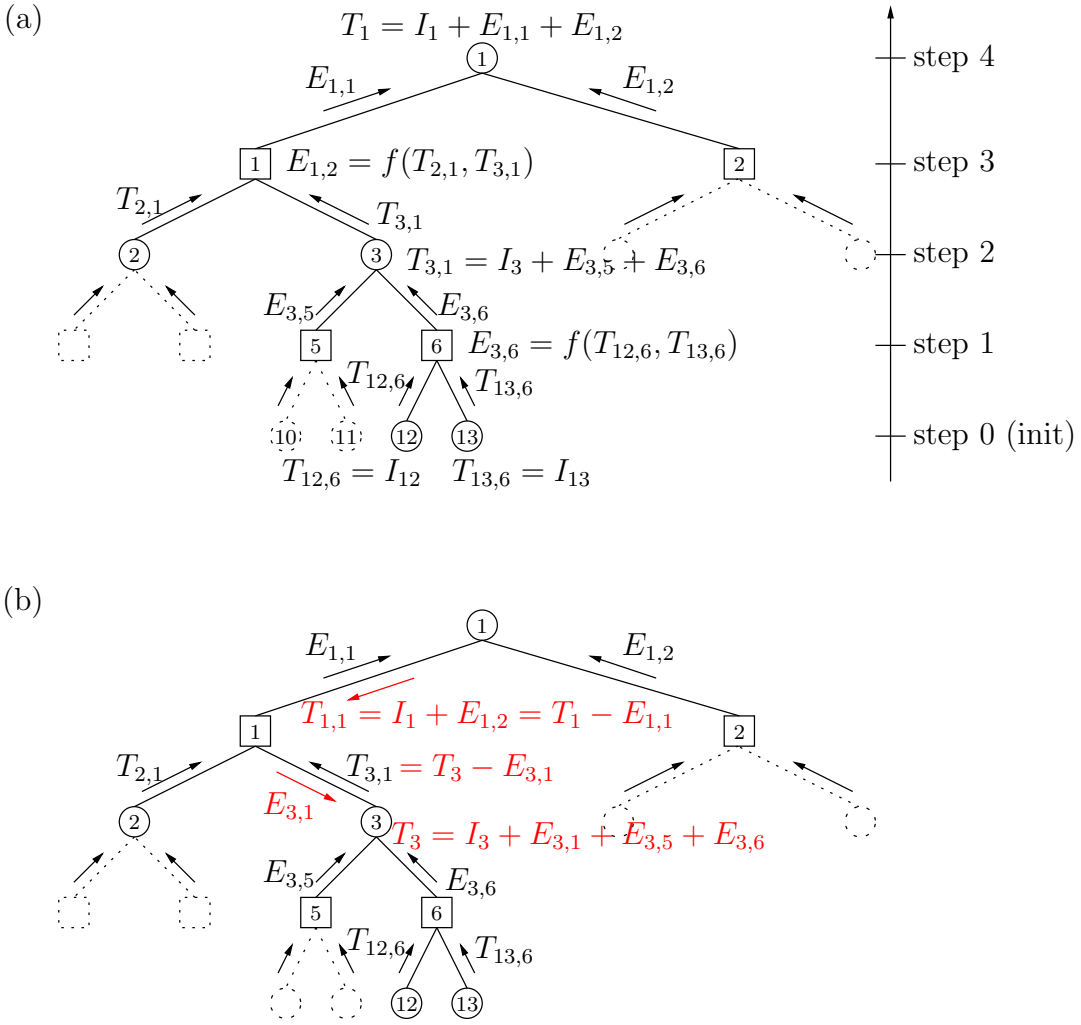



Figure 1.9: Summary of operations on the graph of figure 1.7 for calculating (a) $T_1$ and (b) $T_3$.

(c) Frédéric Guilloud, Télécom Paris - July 2004

(resp. variable) node update. The update rules are repeated until the total information of each bit is computed. Each repetition is an iteration from the node point of view. It is important to notice that an iteration exist here from the processor point of view. *The scheduling of the different processors does not affect the convergence of the algorithm.*

The update rules are depicted on figure 1.10: the check node processors compute and output check-to-bit messages from the incoming bit-to-check messages, and vice versa for variable node processors. There are 2 possible cases, whether the bit-to-check messages is the $T_{n,m}$ information or the $T_n$ information. In the first case, the total information $I_n$ is not calculated:

**Iterative Algorithm 1 (BP)** *Propagation of $T_{n,m}$ messages from variable node $vn_n$ to check node $cn_m$*

*Initialization:* $E_{n,m}^{(0)} = 0$

*Variable node update rule:* $T_{n,m}^{(i)} = I_n + \sum_{m' \in \mathcal{M}(n) \backslash m} E_{n,m'}^{(i-1)}$

*Check node update rule:* $E_{n,m}^{(i)} = 2 \tanh^{-1} \prod_{n' \in \mathcal{N}(m) \backslash n} \tanh \frac{T_{n',m}^{(i)}}{2}$

*Last Variable node update rule:* $T_n = I_n + \sum_{m \in \mathcal{M}(n)} E_{n,m}^{(i-1)}$

In the second case, the information $E_{n,m}$ have to be subtracted in the check node processor to the message $T_n$ coming from the variable node $vn_n$, but the variable node processors have a more simple task.

**Iterative Algorithm 2 (BP)** *Propagation of $T_n$ messages from variable node $vn_n$.*

*Initialization:* $E_{n,m}^{(0)} = 0$

*Variable node update rule:* $T_n^{(i)} = I_n + \sum_{m \in \mathcal{M}(n)} E_{n,m}^{(i-1)}$

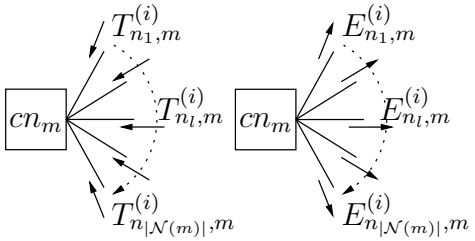
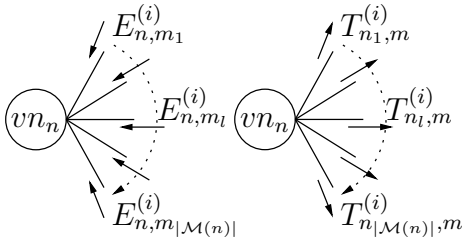
*Check node update rule:* $E_{n,m}^{(i)} = 2 \tanh^{-1} \prod_{n' \in \mathcal{N}(m) \backslash n} \tanh \frac{T_{n'}^{(i)} - E_{n',m}^{(i-1)}}{2}$

## 1.4 Conclusion

Optimal decoding of error correcting codes is possible using an simple iterative algorithm called belief propagation. The only hypothesis to assume is that the graph of the code should not have any cycle. But such an hypothesis is hard to combine with good error correcting codes, *i.e.* which are closed to the Shannon bound. LDPC codes are a class of block codes which can be decoded with the belief propagation algorithm, as described in the next chapter.

(c) Frédéric Guilloud, Télécom Paris - July 2004

Propagation of $T_{n,m}$ messages from variables $vn_n$ to checks $cn_m$

| Check node update rule | Variable node update rule |
|---|---|
|  |  |
| $E^{(i)}_{n,m} = 2\tanh^{-1} \prod_{n' \in \mathcal{N}(m)\setminus n} \tanh \frac{T^{(i)}_{n',m}}{2}$ | $T^{(i)}_{n,m} = I_n + \sum_{m' \in \mathcal{M}(n)\setminus m} E^{(i-1)}_{n,m'}$ |

Propagation of $T_n$ messages from variables $vn_n$ to checks $cn_m$

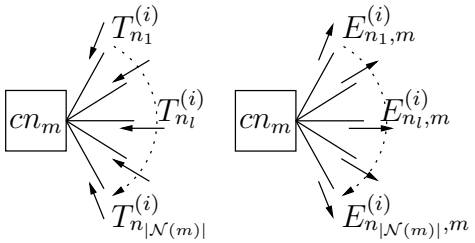
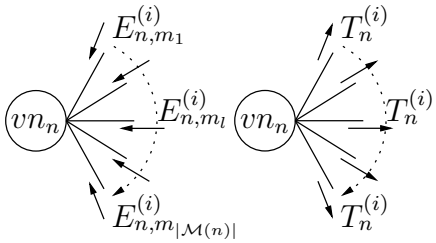
| Check node update rule | Variable node update rule |
|---|---|
|  |  |
| $E^{(i)}_{n,m} = 2\tanh^{-1} \prod_{n' \in \mathcal{N}(m)\setminus n} \tanh \frac{T^{(i)}_{n'} - E^{(i-1)}_{n',m}}{2}$ | $T^{(i)}_{n} = I_n + \sum_{m \in \mathcal{M}(n)} E^{(i-1)}_{n,m}$ |

Figure 1.10: The update rules of the check node and variable node processors at the $i$th iteration.

(c) Frédéric Guilloud, Télécom Paris - July 2004

# Chapter 2

# Low Density Parity Check codes

**Summary:**

*In this chapter, a brief history of LDPC codes is proposed. LDPC codes encompass a wide range of codes, defined by a wide range of parameters. The design of LDPC codes consist in finding a specific set of parameters which suit well to a particular communication scheme. Once the set of parameters is designed, the constructions of a particular parity-check matrix can be built, either by the means of random constructions, or using deterministic constructions. The last 2 sections of this chapter deal with the performance of LDPC codes under iterative decoding with the different possible schedules.*

## 2.1 A bit of History

Low-density parity-check (LDPC) codes were invented by R. G. Gallager (Gallager 1963; Gallager 1962) in 1962. He discovered an iterative decoding algorithm which he applied to a new class of codes. He named these codes low-density parity-check (LDPC) codes since the parity-check matrices had to be sparse to perform well. Yet, LDPC codes have been ignored for a long time due mainly to the requirement of high complexity computation, if very long codes are considered.

In 1993, C. Berrou *et. al.* invented the turbo codes (Berrou, Glavieux, and Thitimajshima 1993) and their associated iterative decoding algorithm. The remarkable performance observed with the turbo codes raised many questions and much interest toward iterative techniques.

In 1995, D. J. C. MacKay and R. M. Neal (MacKay and Neal 1995; MacKay and Neal 1996; Mackay 1999) rediscovered the LDPC codes, and set up a link between their iterative algorithm to the Pearl's belief algorithm (Pearl 1988), from the artificial intelligence community (bayesian networks). At the same time, M. Sipser and D. A. Spielman (Sipser and Spielman 1996) used the first decoding algorithm of R. G. Gallager (algorithm A) to decode expander codes.


(c) Frédéric Guilloud, Télécom Paris - July 2004

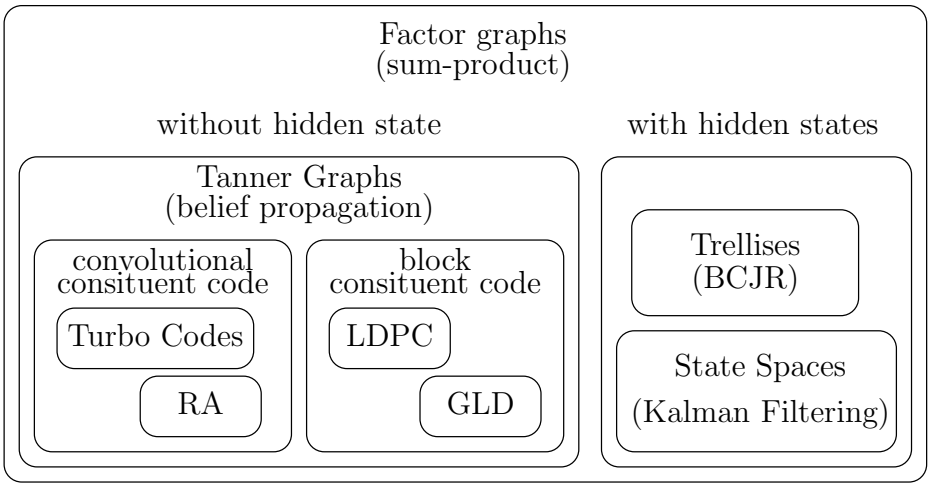



Figure 2.1: 3 classical decoding algorithms as a particular instance of the sum-product algorithm in a factor graph.

The articles of MacKay and Neal have been the kick off of a great work in the field of LDPC codes. Most of the main articles related to the LDPC codes are gathered in a special issue of the IEEE's Transactions on Information Theory (IEEE 2001): irregular codes, density evolution, design of capacity approaching codes, ...

Meanwhile, turbo decoding of turbo codes is shown to be an instance of the Pearl's belief algorithm by McEliece *et. al.* (McEliece, MacKay, and Cheng 1998), collecting under the same model (belief propagation) the last 2 best classes of codes. Graphs are becoming a standard representation of error correcting codes: F. R. Kschischang denotes by factor graphs (Kschischang and Frey 1998) a wide class of graph associated with the sum-product algorithm, which aim at describing many different algorithms by the same formalism (see figure 2.1). This work have its origin in the work of Tanner (Tanner 1981), and N. Wiberg *et. al.* (Wiberg 1996; Wiberg, Loegliger, and Kötter 1995).

Hence, LDPC codes are at the confluence of two major revolutions in the channel coding community: the graph-based code-description, and the iterative decoding techniques.

## 2.2 Classes of LDPC codes

R. Gallager (Gallager 1962) defined an $(N, j, k)$ LDPC codes as a block code of length $N$ having a small fixed number ($j$) of ones in each column of the parity check $H$, and a small fixed number ($k$) of ones in each rows of $H$. This class of codes is then to be decoded by the iterative algorithm described in chapter 1 (see section 1.3.3).

This algorithm computes exact a posteriori probabilities, provided that the Tanner graph of the code is cycle free (see hypothesis 1 page 13). Generally, LDPC codes do have cycles (Etzion, Trachtenberg, and Vardy 1999). The sparseness of the parity check

matrix aims at reducing the number of cycles and at increasing the size of the cycles. Moreover, as the length $N$ of the code increases, the cycle free hypothesis becomes more and more realistic. The iterative algorithm is processed on these graphs. Although it is not optimal, it performs quite well.

Since then, LDPC codes class have been enlarged to all sparse parity check matrices, thus creating a very wide class of codes, including the extension to codes in $GF(q)$ (Davey and MacKay 1998) and irregular LDPC codes (Luby et al. 2001).

### Irregularity

In the Gallager's original LDPC code design, there is a fixed number of ones in both the rows ($k$) and the columns ($j$) of the parity check matrix: it means that each bit is implied in $j$ parity check constraints and that each parity check constraint is the exclusive-OR (XOR) of $k$ bits. This class of codes is referred to as *regular* LDPC codes.

On the contrary, *irregular* LDPC codes do not have a constant number of non-zero entries in the rows or in the columns of $H$. They are specified by the distribution degree of the bit $\lambda(x)$ and of the parity check constraints $\rho(x)$, using the notations of (Luby et al. 1997), where:

$$\lambda(x) = \sum_{i=2}^{d_v} \lambda_i x^{i-1} \tag{2.1}$$

$$\rho(x) = \sum_{i=2}^{d_c} \rho_i x^{i-1} \tag{2.2}$$

$\lambda_i$ (resp. $\rho_i$) denotes the proportion of non-zero entries of $H$ which belongs to the columns (resp. rows) of $H$ of weight $i$. Note that by definition, $\lambda(1) = \rho(1) = 1$. If $\Gamma$ denotes the number of non-zero entries in $H$, $\lambda_i\Gamma$ is the total number of ones in the columns of weight $i$. So $\Gamma\lambda_i/i$ is the total number of columns of weight $i$, and $\sum_i \Gamma\lambda_i/i$ is the total number of columns in $H$. So the proportion of the columns of weight $i$ is:

$$\tilde{\lambda}_i = \frac{\Gamma\lambda_i/i}{\sum_j \Gamma\lambda_j/j} = \frac{\lambda_i/i}{\sum_j \lambda_j/j} \tag{2.3}$$

Similarly, denoting by $\tilde{\rho}_i$ the proportion of rows having weight $i$:

$$\tilde{\rho}_i = \frac{\rho_i/i}{\sum_j \rho_j/j} \tag{2.4}$$

Table 2.1 lists some classes of LDPC codes. For example, the original LDPC code of R. Gallager (Gallager 1963) which is a regular ($N = 20, j = 3, k = 4$) LDPC code is in the class $\mathcal{L}_2(20, 3, 4) = \mathcal{L}_2(20, x^2, x^3)$.

(c) Frédéric Guilloud, Télécom Paris - July 2004

Table 2.1: Different classes of LDPC codes

| Notation | Description |
|---|---|
| $\mathcal{L}_q$ | LDPC codes in $GF(q)$ |
| $\mathcal{L}_2$ | binary LDPC codes |
| $\mathcal{L}_q(M,N)$ | LDPC codes in $GF(q)$ of length $N$ and of rate $1-M/N$. |
| $\mathcal{L}_q(N,\lambda,\rho)$ | LDPC of length $N$ and degree distribution defined by $\lambda(x)$ and $\rho(x)$. |
| $\mathcal{L}_q(N,j,k)$ | regular LDPC code in $GF(q)$ of length $N$ and with $\lambda(x) = x^{j-1}$, $\rho(x) = x^{k-1}$. |

### Code rate

The rate $R$ of LDPC codes is defined by $R \geq R_d \triangleq 1 - \frac{M}{N}$ where $R_s$ is the *design code rate* (Gallager 1962). $R_d = R$ if the parity check matrix has full rank. The authors of (Miller and Cohen 2003) have shown that as $N$ increases, the parity-check matrix is almost sure to be full rank. Hereafter, we will assume that $R = R_d$ unless the contrary is mentioned. The rate $R$ is then linked to the other parameters of the class by:

$$R = 1 - \frac{\sum_i \rho_i / i}{\sum_i \lambda_i / i} = 1 - \frac{j}{k} \tag{2.5}$$

Note that in general, for random constructions, when $j$ is odd:

$$R = 1 - \frac{M}{N} \tag{2.6}$$

and when $j$ is even:

$$R = 1 - \frac{M-1}{N} \tag{2.7}$$

## 2.3 Optimization of LDPC codes

The bounds and performance of LDPC codes are derived from their parameters set. The wide number of independent parameters enables to tune them so as to fit some external constraint, as a particular channel, for example. Two algorithms can be used to design a class of irregular LDPC codes under some channel constraints: the density evolution algorithm (Richardson, Shokrollahi, and Urbanke 2001) and the extrinsic information transfer (EXIT) charts (ten Brink 1999).

### Density evolution algorithm

Richardson et al. (Richardson, Shokrollahi, and Urbanke 2001) designed capacity approaching irregular codes with the density evolution (DE) algorithm. This algorithm

tracks the probability density function (pdf) of the messages through the graph nodes under the assumption that the cycle free hypothesis is verified. It is a kind of belief propagation algorithm with pdf messages instead of log likelihood ratios messages. Density evolution is processed on the asymptotical performance of the class of LDPC codes. It means that a infinite number of iterations is processed on a infinite code-length LDPC code: if the length of the code tends to infinity, the probability that a randomly chosen node belongs to a cycle of a given length tends towards zero.

Usually, either the channel threshold or the code rate are optimized under the constraints of the degree distributions and of the SNR. The threshold of the channel is the value of the channel parameter (see table 1.2) above which the probability tends towards zero if the iterations are infinite (and the code length also). Optimization tries to lower the threshold or to higher the rate as best as possible.

In (Chung et al. 2001) for example, the authors designed a rate$-1/2$ irregular LDPC codes for binary-input AWGN channels that approach the Shannon limit very closely (up to 0.0045 dB). Optimization based on DE algorithm are often processed by the mean of differential evolution algorithm when optimizations are non-linear, as for example in (Hou, Siegel, and Milstein 2001) where the authors optimize an irregular LDPC code for uncorrelated flat Rayleigh fading channels. The Gaussian approximation (Chung, Richardson, and Urbanke 2001) in the DE algorithm can also be used: the probability density function of the messages are assumed to be Gaussian and the only parameters that has to be tracked in the nodes is the mean.

Hence, optimization for LDPC codes can be made under various type of channels, and prove to be good alternatives to actual solutions for these channels:

- partial response channels (Li et al. 2002), (Thangaraj and McLaughlin 2002) (with gaussian approximation), (Varnica and Kavcic 2003);
- frequency selective channel (OFDM, with Gaussian approximation) (Mannoni, Declercq, and Gelle 2002);
- multiple access channel (2 users, Gaussian approximation) (Amraoui, Dusad, and Urbanke 2002) and joint multiple user decoding (de Baynast and Declercq 2002);
- a joint AWGN and Rayleigh fading channel approach, with Gaussian approximation (Lehmann and Maggio 2003);
- bandwidth efficient modulation (Hou et al. 2003);
- minimum shift keying modulations (Narayanan, Altunbas, and Narayanaswami 2003);
- a multiple input multiple output channel with an OFDM modulation (Lu, Yue, and Wang 2004).

(c) Frédéric Guilloud, Télécom Paris - July 2004

**EXIT chart**

Extrinsic information transfer (EXIT) charts (ten Brink 1999) are 2D graphs on which are superposed the mutual information transfers through the 2 constituent codes of a turbo-code. EXIT charts have been transposed to the LDPC code optimization (Ardakani and Kschischang 2002; Narayanan, Wang, and Yue 2002; Ardakani, Chan, and Kschischang 2003).

## 2.4 Constructions of LDPC codes

By constructions of LDPC codes, we mean the construction, or design, of a particular LDPC parity check matrix $H$. The design of $H$ is the moment when the asymptotical constraints (the parameters of the class you designed, like the degree distribution, the rate) have to meet the practical constraints (finite dimension, girths).

Hereafter are described some recipes taking into account some practical constraints. Two techniques exist in the literature: random and deterministic ones. The design compromise is that for increasing the girth, the sparseness has to be decreased yielding poor code performance due to a low minimum distance. On the contrary, for high minimum distance, the sparseness has to be increased yielding the creation of low-length girth, due to the fact that $H$ dimensions are finite, and thus, yielding a poor convergence of the belief propagation algorithm.

### 2.4.1 Random based construction

The first constructions of LDPC codes were random ones (Gallager 1962; MacKay and Neal 1995; MacKay and Neal 1996; Mackay 1999). The parity check matrix is the concatenation and/or superposition of sub-matrices; these sub-matrices are created by processing some permutations on a particular (random or not) sub-matrix which usually has a column weight of 1. R. Gallager's construction for example is based on a short matrix $H_0$. Then $j$ matrices $\pi_i(H_0)$ are vertically stacked on $H_0$, where $\pi_i(H_0)$ denotes a column permutation of $H_0$ (see figure 2.2).

Regular and irregular codes can be also constructed like in (Luby et al. 2001) where the 2 sets of nodes are created, each node appearing as many times as its degree's value. Then a one to one association is randomly mapped between the nodes of the 2 sets, like illustrated on figure 2.3. In (MacKay, Wilson, and Davey 1999), D. MacKay compares random constructions of regular and irregular LDPC codes: small girth have to be avoided, especially between low weight variables.

All the constructions described above should be constrained by the girth's value. Yet, increasing the girth from 4 to 6 and above is not trivial; some random constructions specifically addresses this issue. In (Campello and Modha 2001), the authors generate a parity check matrix optimizing the length of the girth or the rate of the code when $M$ is

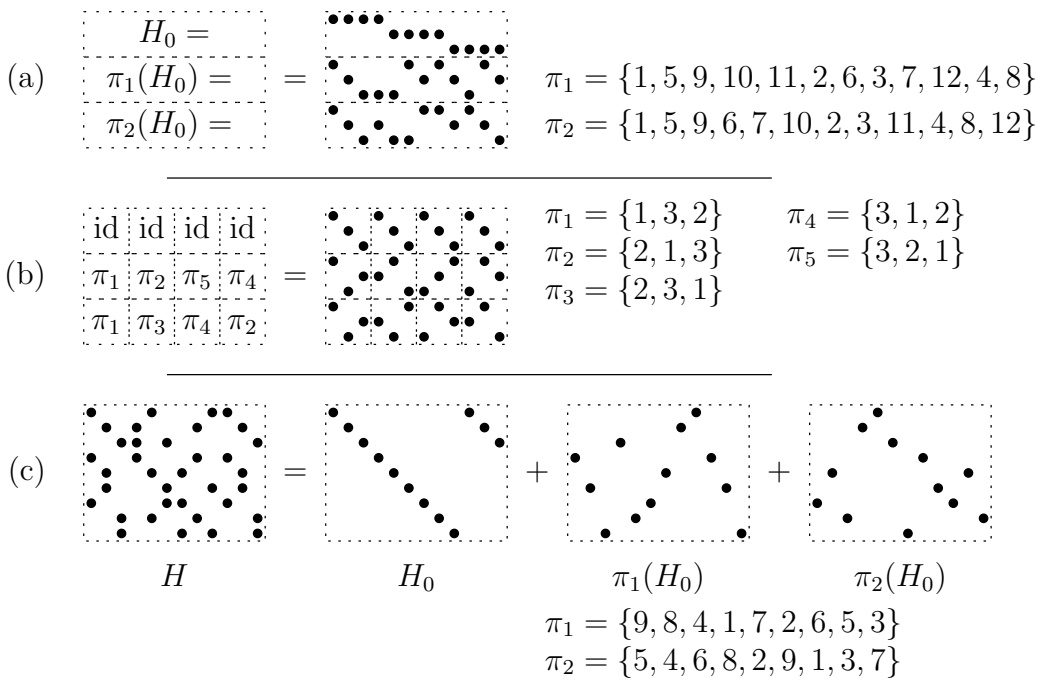



Figure 2.2: Some random constructions of regular LDPC parity check matrices based on Gallager's (a) and MacKay's constructions (b,c) (MacKay, Wilson, and Davey 1999). Example of a regular $(3, 4)$ LDPC code of length $N = 12$. Girths of length 4 have not been avoided. The permutations can be either columns permutation (a,b) or rows permutations (c).

(c) Frédéric Guilloud, Télécom Paris - July 2004

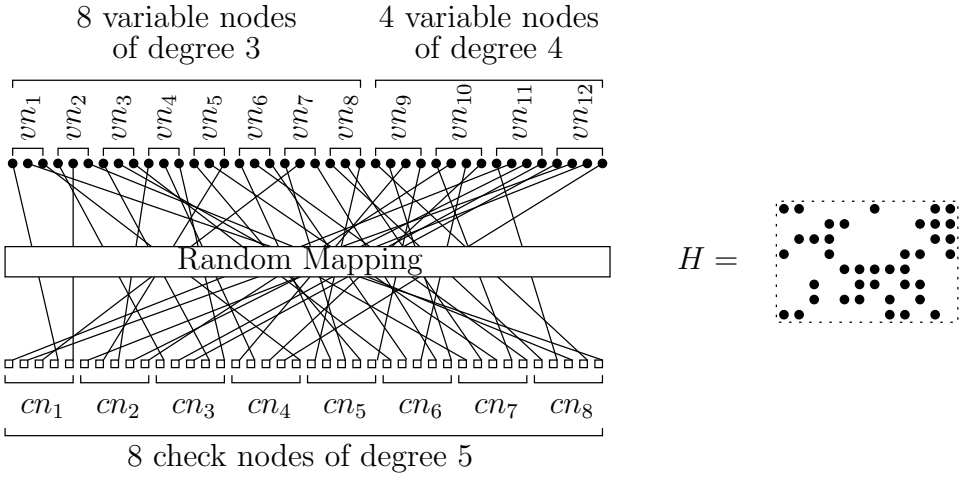



Figure 2.3: Random construction based on (Luby et al. 2001) of an irregular LDPC code of size $N = 12$ and with $(\lambda, \rho) = (\frac{3}{5}x^2 + \frac{2}{5}x^3,\, x^4)$. There are $\tilde{\lambda}_3 = 12\frac{\frac{3/5}{3}}{\frac{3/5}{3}+\frac{2/5}{4}} = 8$ variable nodes of weight 3 and $\tilde{\lambda}_4 = 12\frac{\frac{2/5}{4}}{\frac{3/5}{3}+\frac{2/5}{4}} = 4$ variable nodes of weight 4. The variable nodes are connected to $M = 12\frac{\frac{1}{5}}{\frac{3/5}{3}+\frac{2/5}{4}} = 8$ parity check nodes of weight 5. The mapping is randomly chosen. The size of the example can not lead to a length-4 girth free design.

(c) Frédéric Guilloud, Télécom Paris - July 2004

fixed ($N$ increases). In (Mao and Banihashemi 2001b), the authors study the histogram of cycles length of randomly generated LDPC codes, based on MacKay's construction 2 A and they select the best cycle-length histogram shape. In (Hu, Eleftheriou, and Arnold 2001) the graph of the LDPC code is generated edge by edge so as to avoid small girths. Regular $(N, j = 2, k)$ LDPC codes are built in (Zhang and Moura 2003) with girths of size 12, 16 and 20 on a very structured graph. The girth is very high but each bit is only protected by 2 parity check constraints. An irregular design of a given $(\lambda, \rho)$ profile is proposed in (Sankaranarayanan, Vasic, and Kurtas 2003) based on a regular LDPC code. The parity check matrix is modified so as to obtain the given profile. The authors of (Djurdjevic, Lin, and Abdel-Ghaffar 2003) use a trellis-based algorithm to design an LDPC code with a high girth. Not all the small cycles have the same influence on the performance of the code: in (Tian et al. 2003) only the penalizing small length cycles are removed.

Another original construction in (Prabhakar and Narayanan 2002), implemented in (Verdier, Declercq, and J.-M. 2002), is to use linear congruent sequences to generate the position of non-zero entries in the parity check matrix. The advantage is that the memory required to save $H$ inside the decoder is significantly reduced.

### 2.4.2 Deterministic based construction

Random constructions don't have too many constraints: they can fit quite well to the parameters of the desired class. The problem is that they do not guarantee that the girth will be small enough. So either post-processing or more constraints are added for the random design, yielding sometimes much complexity.

To circumvent the girth problem, deterministic constructions have been developed. Moreover, explicit constructions can lead to easier encoding, and can be also easier to handle in hardware. 2 branches in combinatorial mathematics are involved in such designs: finite geometry and Balanced Incomplete Block Design's (BIBSs). They seem to be more efficient than previous algebraic constructions which was based on expander graphs (Lafferty and Rockmore 2000; Rosenthal and Vontobel 2001; MacKay and Postol 2003). In (MacKay and Davey 2000) the authors designed high rate LDPC codes based on Steiner systems. Their conclusion was that the minimum distance was not high enough and that difference set cyclic (DSC) codes should outperform them, as in (Lucas et al. 2000) where they are combined with the one step majority logic decoding. In (Kou, Lin, and Fossorier 2001), the authors present LDPC code constructions based on finite geometry, like in (Johnson and Weller 2003a) for constructing very high rate LDPC codes. Balanced incomplete block designs (BIBDs) have also been studied in (Ammar et al. 2002; Vasic 2002; Johnson and Weller 2003b; Vasic, Djordjevic, and Kostuk 2003).

The major drawback for deterministic constructions of LDPC codes is that they exist with a few combinations of parameters. So it may be difficult to find one that fits the specifications of a given system.

(c) Frédéric Guilloud, Télécom Paris - July 2004

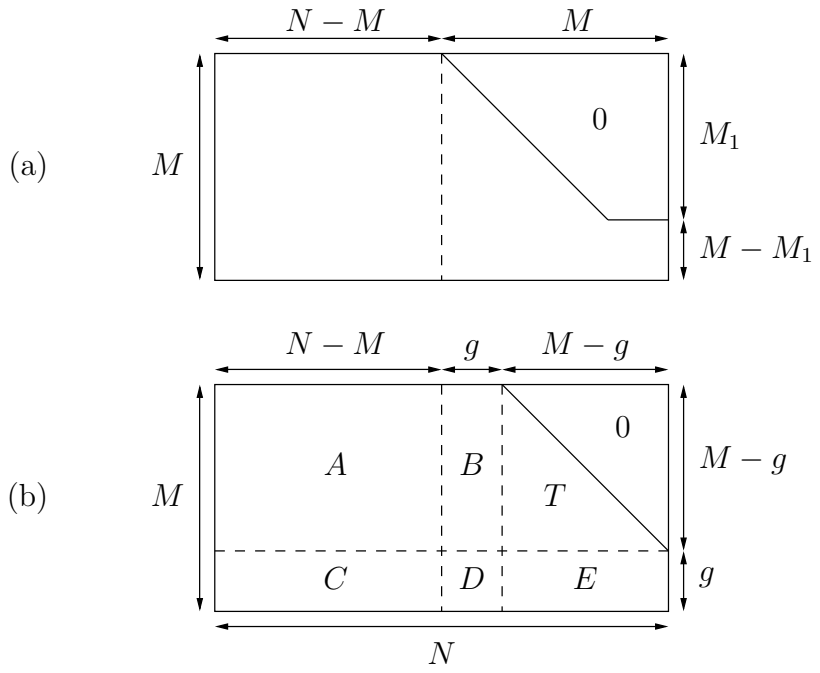



Figure 2.4: Shape of parity check matrices for efficient encoding, by MacKay et al. (MacKay, Wilson, and Davey 1999) (a) and Richardson *et. al.* (Richardson and Urbanke 2001) (b)

## 2.5 Encoding of LDPC codes

The weak point of LDPC codes is their encoding process: a sparse parity check matrix does not have necessarily a sparse generator matrix. Moreover, it appears to be particularly dense. So encoding by a $G$ multiplication yields to an $N^2$ complexity processing. A first encoding scheme is to deal with lower triangular shape parity check matrices. The other encoding schemes are mainly to deal with cyclic parity check matrices.

### 2.5.1 Lower-triangular shape based encoding

A first approach in (MacKay, Wilson, and Davey 1999) is to create a parity check matrix with an almost lower-triangular shape, as depicted on figure 2.4-(a). The performance is a little bit affected by the lower-triangular shape constraint. Instead of computing the product $c = uG^t$, the equation $H.c^t = 0$ is solved, where $c$ is the unknown variable. The encoding is systematic:

$$\{c_1, \cdots, c_{N-M}\} = \{u_1, \cdots, u_{N-M}\} \tag{2.8}$$

The next $M_1$ $c_i$ are recursively computed by using the lower-triangular shape:

$$c_i = -pc_i \times (c_1, \cdots, c_{i-1})^t, \text{ for } i \in \{N - M + 1, \cdots, N - M + M_1\} \tag{2.9}$$

The last $M - M_1$ $c_i$, $i \in \{N - M + M_1 + 1, \cdots, N\}$ have to be solved without reduced complexity. Thus, the higher $M_1$ is, the less complex the encoding is.

In (Richardson and Urbanke 2001) T. Richardson and R. Urbanke propose an efficient encoding of a parity check matrix $H$. It is based on the shape depicted on figure 2.4-(b). They also propose some "greedy" algorithms which transform any parity check matrix $H$ into an equivalent parity check matrix $H'$ using columns and rows permutations, minimizing $g$. So $H'$ is still sparse. The encoding complexity scales in $\mathcal{O}(N + g^2)$ where $g$ is a small fraction of $N$.

As a particular case the authors of (Bond, Hui, and Schmidt 2000) and (Hu, Eleftheriou, and Arnold 2001) construct parity check matrices of the same shape with $g = 0$.

### 2.5.2 Other encoding schemes

#### Iterative encoding

In (Haley, Grant, and Buetefuer 2002), the authors derived a class of parity check codes which can be iteratively encoded using the same graph-based algorithm as the decoder. But for irregular cases, the codes does not seem to perform as well as random ones.

#### Low-density generator matrices

The generator matrices of LDPC codes are usually not sparse, because of the inversion. But if $H$ is constructed both sparse and systematic, then:

$$H = (P, I_M) \text{ and } G = (I_{N-M}, P^t)$$

where $G$ is a sparse generator matrix (LDGM) (Oenning and Moon 2001): they correspond to parallel concatenated codes. They seem to have high error floors (Mackay 1999) (asymptotically bad codes). Yet, the authors of (Garcia-Frias and Zhong 2003) carefully chose and concatenate the constituent codes to lower the error floor. Note that this may be a drawback for applications with high rate codes.

#### Cyclic parity-check matrices

The most popular codes that can be easily encoded are the cyclic or pseudo-cyclic ones. In (Okamura 2003), a Gallager-like construction using cyclic shifts enables to have a cyclic-based encoder, like in (Hu, Eleftheriou, and Arnold 2001). Finite geometry or BIBDs constructed LDPC codes are also cyclic or pseudo-cyclic (Kou, Lin, and Fossorier 2001; Ammar et al. 2002; Vasic 2002). Table 2.2 gives a summary of the different encoding schemes.

(c) Frédéric Guilloud, Télécom Paris - July 2004

Table 2.2: Summary of the different LDPC encoding schemes

| Encoding scheme | Description | Comments |
|---|---|---|
| Generator matrix product | $H \Rightarrow G$ ; $c = uG^t$ | Use sparse generator matrices (LDGM). Bad error floor |
| Triangular system solving | Solve $Hc^t = 0$ using as much back-substitution as much as possible | High complexity post processing |
| Iterative encoding | Solve $Hc^t = 0$ using the some product algorithm | Such iterative encodable codes seem to have weak performance. |
| Cyclic encoding | Multiplications with a shift register | Few constructions |

## 2.6 Performance of BPSK-modulated LDPC codes

### The BPSK capacity

The gap to the AWGN capacity has been presented in section 1.2.2. Hereafter, only BPSK modulations will be used. In this case the constellation size is equal to $\mathbb{M} = 2$ and hence the information rate is equal to the bit rate $R_{Id} = R$. The AWGN capacity constrained by the BPSK input is as farther from the AWGN capacity as the rate increases toward its asymptotic value $R_{max} = 1$. For a fixed capacity, the difference between the $E_b/N_0$ of the both cases is depicted on figure 2.5 in both linear and logarithmic scales. For example, for the AWGN channel, $(E_b/N_0)_{R_{Id}=0.5} = 0$ dB whereas for the BPSK-input constrained AWGN channel, $(E_b/N_0)_{R_{Id}=0.5} = 0.188$ dB. The difference between these two $E_b/N_0$ is used to measure the gap to the capacity of a given class of codes. For example in (Chung et al. 2001), the author designed a class of rate$-1/2$ LDPC codes which are within 0.0045 dB from the capacity $C_d^*$. In (Boutros et al. 2002), the authors designed a class of rate$-1/3$ turbo code which are within 0.03 dB from the capacity $C_d^*$. These designs are optimized using density evolution or its approximation: it is the only algorithm which gives a threshold for the code. Figure 2.6 scatters the threshold of various regular and irregular LDPC classes found in the literature.

### Practical performance

The threshold is usually not known for finite length code designs. In that case, the gap to the capacity is measured between the BER of the code and the BER of the fictitious code of the same rate achieving the Shannon capacity (see figure A.1 in annexe A) for a

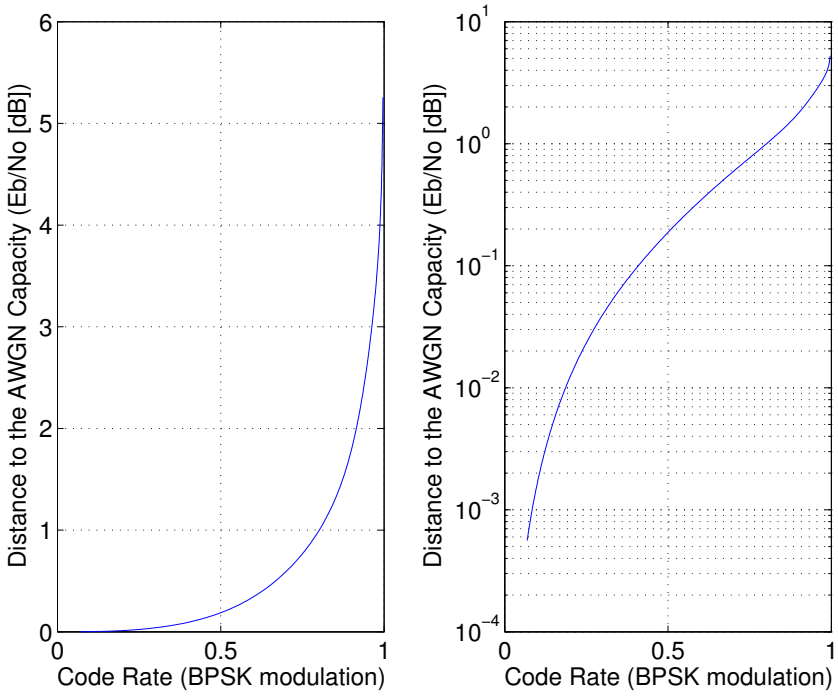



Figure 2.5: Difference between the $E_b/N_0$ of an AWGN channel and of a BPSK input AWGN output channel for a given capacity $C_d$

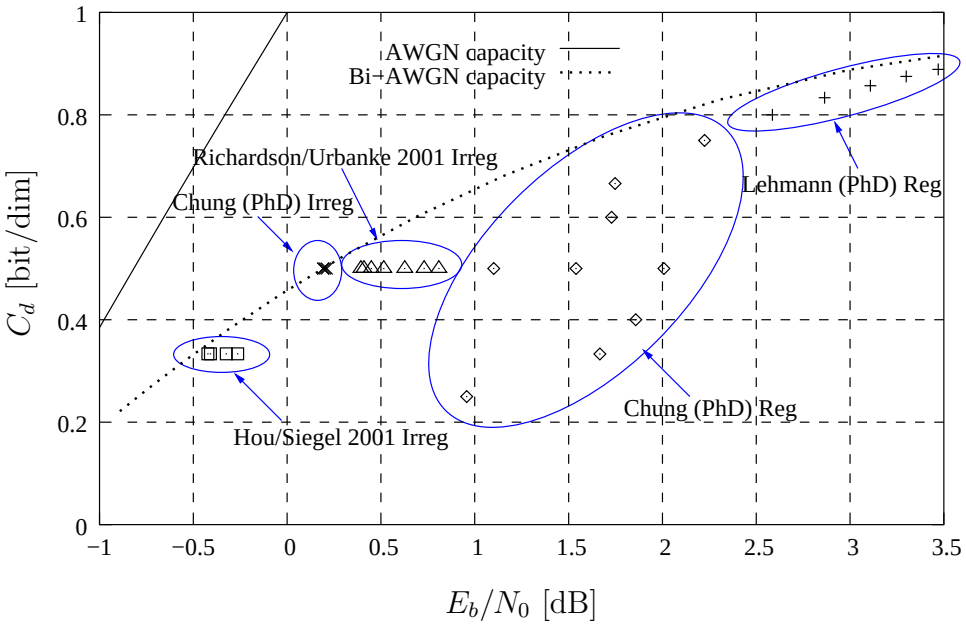



Figure 2.6: Various thresholds for regular or irregular LDPC codes, for various rates

(c) Frédéric Guilloud, Télécom Paris - July 2004

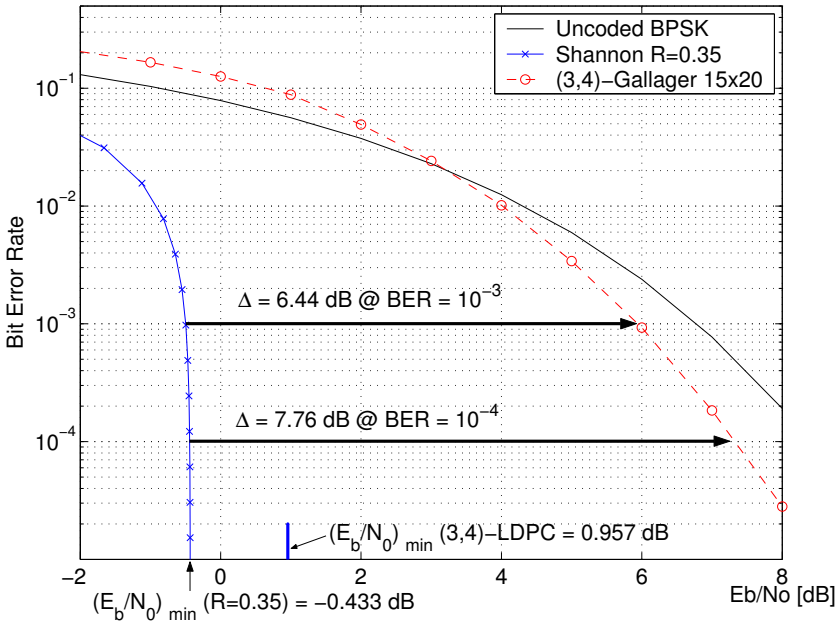



Figure 2.7: An approximate measurement of the distance to the Shannon bound on the error probability graphs

given BER value. The FER curves should be used but for large block codes, the FER can be computationally expensive to estimate. Usually, the gap is measured at a BER of $10^{-5}$, such as in (Nickl, Hagenauer, and Burkert 1997), where the authors designed a turbo-code within 0.24 dB of the Shannon's capacity, at a BER$= 10^{-5}$. Figure 2.7 depicts the gap-to-capacity measurement on the BER curves for the practical example of the regular $(3,4)-$LDPC code of length $N = 20$, with rate $R = 0.35$ ($K = 7$) since $H$ is not full rank, which has been originally designed by Gallager (Gallager 1962). . Such a measurement does not have a very precise meaning as compared to the distance to the Shannon limit. In (Battail and Magalhães De Oliveira 1993), the authors took up again Shannon computations of error probability, without any approximations. They derived exact and approximate FER as a function of the code parameters, for a Gaussian channel. But there is no exact expression for the error probability as mentioned in (Dolinar, Divsalar, and Pollara 1998), where bounds of code performance as a function of the code length and of the code rate are studied.

## 2.7 Decoding of LDPC codes

Decoding of LDPC codes is processed by applying the optimal iterative decoding algorithm described in section 1.3.3. The optimality is lost since the graph of the code has cycles, but the good performance achieved yields to use it as a good approximation. This algorithm

is called the belief propagation (BP) algorithm.

### 2.7.1 Scheduling

The scheduling of the BP algorithm is the order in which the messages of the graph should be propagated. In the cycle-free case of section 1.3.3, this scheduling does not affect the convergence of the algorithm. But for implementation purpose, two schedules have been distinguished (Kschischang and Frey 1998):

- Two way scheduling: is a serial-oriented schedule, where only the relevant messages are processed and passed, according to the description in figure 1.9.
- Flooding schedule: is a parallel-oriented schedule, where all the nodes are processed according to the description in figure 1.10. A incoming message acts as a trigger for the node processor.

For practical codes, which graphs are not cycle-free, the flooding schedule is used: the behaviour of each node processor is much more simplified. Yet, there is another question: in what order should the node processor compute their output messages ? when should they be activated ? This order, also referred to as schedule, will affect the convergence performance because of the cycles in the graph. Three schedules can be found in the literature:

#### Flooding schedule

The flooding schedule will denote hereafter the classical way of scheduling the BP algorithm. The meaning of flooding might have been changed from the original meaning of (Kschischang and Frey 1998). In this flooding schedule, the nodes of the same type are all updated and then the nodes of the other type are also all updated (see figure 2.8-(a)). The update for a type of node can be made one node at a time (serially) or in parallel : it does not change the output messages.

#### Probabilistic schedule

The authors of (Mao and Banihashemi 2001a) described a probabilistic schedule. The first idea to get rid of auto-confirmation messages induced by the cycles of the graph: they avoid their propagation by sometime not activating node processors, when they should be in the flooding schedule.

Let $g_n$ be the girth (Mao and Banihashemi 2001b) of the variable node $vn_n$: it means the length (number of edges) of the smallest cycle that passes through $vn_n$. Let also $g_{\max}$ be the maximum size of girths $g_n,\ n \in \{1, \cdots N\}$. The smallest number of iterations avoiding the auto-confirmation of information of the variable node $vn_n$ on itself is then $g_n/2$, since one iteration is a 2 edge-long data path. So each variable node $vn_n$ should

(c) Frédéric Guilloud, Télécom Paris - July 2004

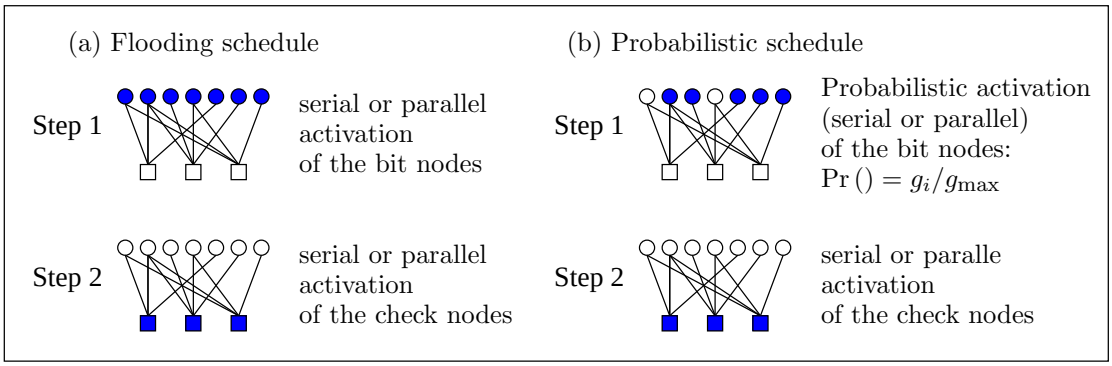



Figure 2.8: (a) Flooding and (b) probabilistic schedule steps at each iteration $i$. Updating node processors are filled

be updated only if iteration $i < g_n/2$. Then it is idled (see figure 2.8-(b)). When more than $g_{\max}/2$ iterations have to be processed, the variable nodes are all updated at iterations $kg_{\max}/2$, $k$ being an integer, and then the same activation rule applies on $vn_n$ by comparing $i \mod (g_{\max}/2)$ to $g_n/2$.

The author of (Mao and Banihashemi 2001a) implemented a slightly different schedule than this one. The activation is probabilistic (hence the name probabilistic shuffle): for iterations $i > 1$, each variable node $vn_n$ is activated with probability $p_n = g_n/g_{\max}$.

### Vertical shuffle scheduling

The authors of (Zhang and Fossorier 2002) proposed a shuffle BP algorithm which converges faster than the BP algorithm. The idea is to update the information as soon as it has been computed, so that the next node processor to be updated could use a more up to date information. This schedule operates along the variables: it means that all the variable node are processed on after the other. So it is also called vertical shuffle since the check node are processed in a shuffle order.

Figure 2.9 illustrates on a example the first steps of an iteration $i$. The scheduling of an iteration $i$ is serial: the check nodes implying the first variable are processed (step 1) and the first variable node is updated (step 2). Then the check nodes implying the second variable are processed (step 3) and the second variable node is updated (step 4) and so on until all the variable nodes have been updated. During an iteration, each check node processor $cn_m$ is activated as much as $|\mathcal{N}(m)|$ times. In (Zhang and Fossorier 2002), the authors have rewritten the updates rules so that the check nodes processors should not re-process the overall calculations each time it should be activated, using a forward-backward strategy. But this solution still requires the saving of the trellis state.

To overcome the low rate implied by the serial processing of the variable nodes, the authors finally suggest to process the shuffle scheduling on groups of $N/G$ variables. Inside

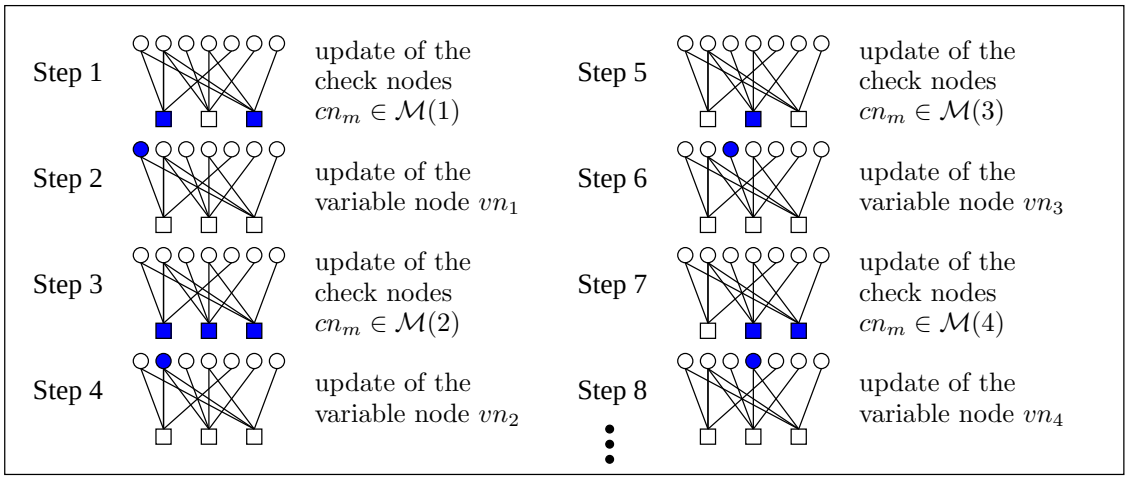



Figure 2.9: First steps of the shuffle schedule at iteration $i$. Updating node processors are filled.

each group, the flooding schedule is used.

The major drawback of this scheduling should be that the girths of the graph are not considered: if applied on small girths, the acceleration of the information could decrease the performance of the decoding algorithm.

### Horizontal shuffle scheduling

The authors of (Yeo, Nikolić, and Anantharam 2001) proposed in a serialized architecture a *staggered* scheduling which consist in processing serially the parity-checks processors. The information sent to the check node under process, say check node $cn_m$, takes into account the information of the previous iteration and the information of the current iteration which have been updated by all the previous check node $cn_{m'}$, $m' < m$. But this scheduling is associated to the APP algorithm and does not perform well under the iterative algorithm.

In (Mansour and Shanbhag 2002b), the authors also designed an horizontal shuffle scheduling. As in the vertical shuffle, the convergence is also accelerated, and the major drawback of this scheduling should also be that the girths of the graph are not considered. Figure 2.10 describes the steps of an iteration $i$: for all the constraints nodes $cn_m$, update the processor and then update all the variable processors $vn_n$ which are connected to $cn_m$. As in (Zhang and Fossorier 2002), the serial processing can be accelerated by grouping some check node constraints and processing for a group of $M/G$ nodes a flooding scheduling.

(c) Frédéric Guilloud, Télécom Paris - July 2004

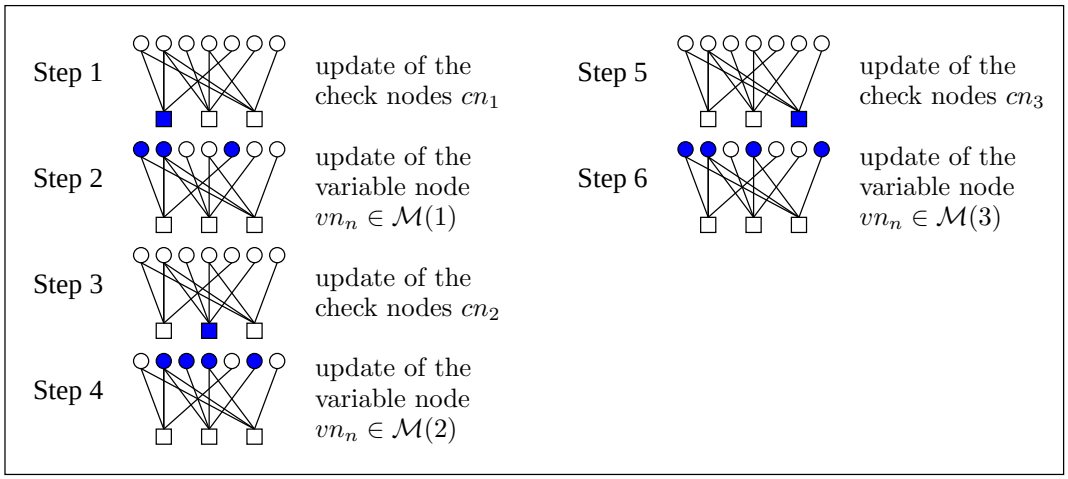



Figure 2.10: Steps of the horizontal shuffle schedule at iteration $i$. Updating node processors are filled.

### 2.7.2 Performance in iterative decoding

The error rate of iteratively decoded codes has a typical shape such as sketched on figure 2.11 in two cases. Three regions can be distinguished on the solid line curve:

- the first region is where the code is not very efficient, below the convergence threshold. Even if the number of iterations is increasing, the performance is not improved;
- the waterfall region is where the error rate as a huge negative slope, which increases as the number of iterations increases.
- the error floor region is where the error rate slope is lower than in the waterfall region. The error floor is due to the minimum hamming distance of the code. For LDPC codes, it is also caused by near codewords (MacKay and Postol 2003), also called pseudo-codewords by (Koetter and Vontobel 2003).

As illustrated by the dashed line curve, there is often a trade-off to be made between the performance of the code in the waterfall region and in the error floor region. The performance achieved by the different scheduling and the different algorithms will be discussed in the next chapter.

## 2.8 Conclusion

LDPC codes have been discovered a long time ago and rediscovered after the invention of the turbo-codes. These two codes are the actors of the revolution of the error correcting codes theory, which combined iterative decoding algorithms and codes based on graphs.

(c) Frédéric Guilloud, Télécom Paris - July 2004

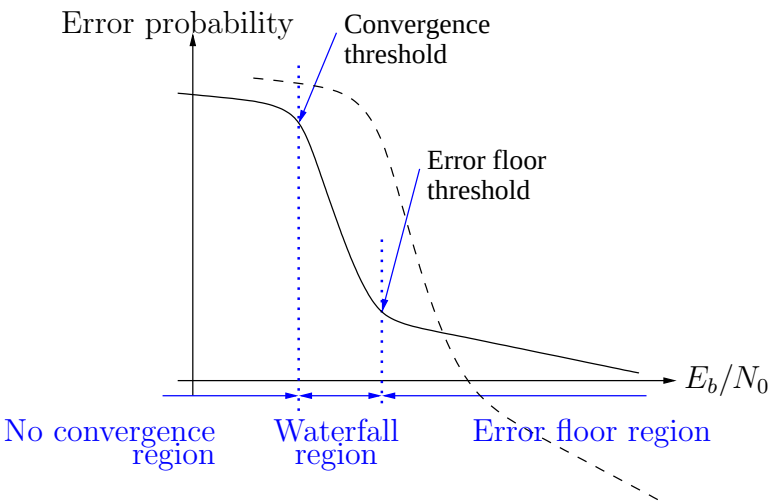



Figure 2.11: Typical regions in an error probability curve of iterative decoding algorithms: the waterfall region and the error floor region are illustrated on a factious performance curve (solid line). The trade off between these two regions is illustrated by the second curve (dashed line) which has lower error floor at the expense of a higher convergence threshold.

An important advantage is that they offer a high degree of freedom for the LDPC code optimization: it is possible to design an LDPC class of codes which fit to some channel specifications. The main drawback in using LDPC codes is their encoding complexity. But some cyclic or pseudo-cyclic encodable LDPC codes may solve this issue, even if the set of parameters is reduced. Note also that another class of channel codes named Repeat-Accumulate codes which can be viewed as LDPC codes have a very simple encoding scheme, as described explicitly through their name. The overall performance of LDPC codes is the major reason why LDPC decoders are being implemented for about four years now. This state of the art is studied in the next chapter.

(c) Frédéric Guilloud, Télécom Paris - July 2004

This page intentionally left blank.

# Chapter 3

# A unified framework for LDPC decoders

**Summary:**

*In this chapter, a unified framework for describing and designing the architecture of LDPC decoders is presented. A set of parameters is defined to characterize the parallelism factor. The generic node processors are classified as a function of their data flow, their control and the position of the interconnection network. The combination of the different instantiation enables to derive an architecture of an LDPC decoder for many decoding schedules. A complexity analysis is also performed: three parameters specifying completely the complexity of an LDPC decoder architecture are proposed. Then a synthesis of this framework is applied on three important examples: a classical one (flooding schedule), a recent one (horizontal shuffling) and also a new one which has not been published yet, implementing a vertical shuffle efficiently. Finally, an overview of the existing LDPC platforms is proposed.*

## 3.1 Generalized message-passing architecture

### 3.1.1 Overview

A generalized architecture of a message passing architecture is depicted on figure 3.1. We will hereafter assume to simplify the notations and without loss of generality that a regular $(j, k)$-LDPC code is concerned. The first descriptions of such architectures can be found in (Boutillon, Castura, and Kschischang 2000; Mansour and Shanbhag 2002a; Zhang and Parhi 2002). It is composed of a direct and a reverse shuffle network, in the center of the figure. On each side of the shuffle networks, the node processors are in charge of the processing of the update rules of the iterative algorithms:


(c) Frédéric Guilloud, Télécom Paris - July 2004

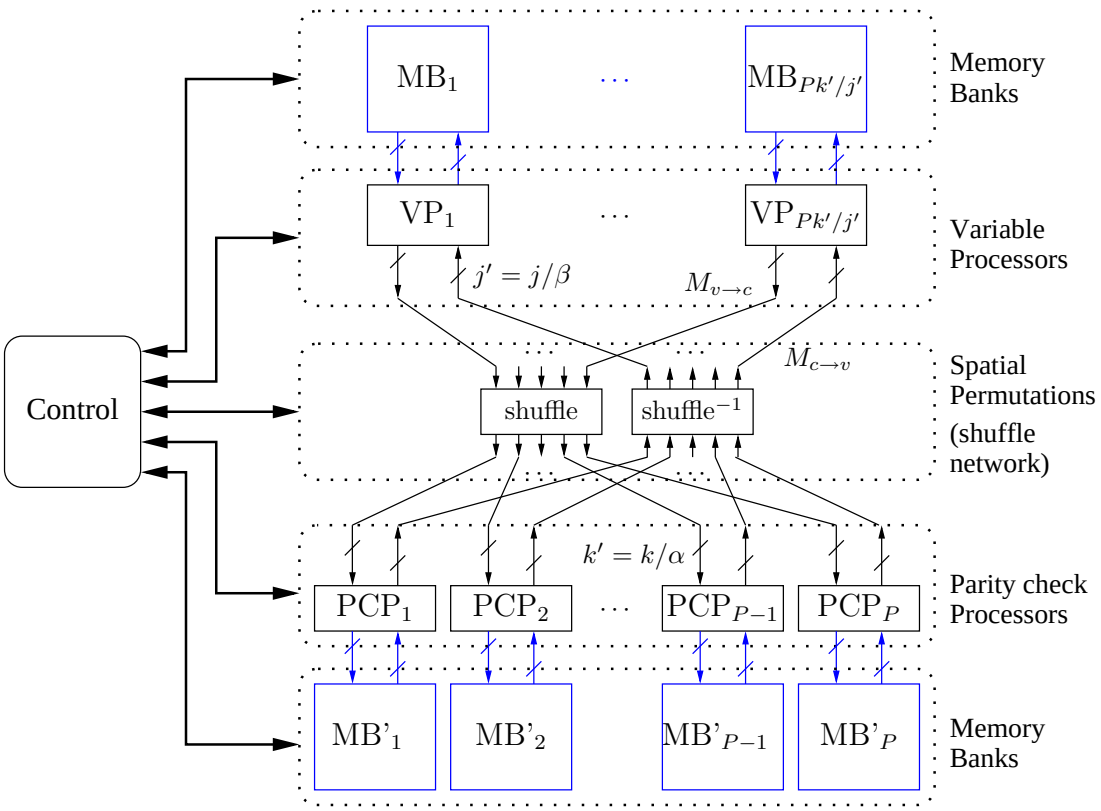



Figure 3.1: Generalized design for message passing architecture decoders

(c) Frédéric Guilloud, Télécom Paris - July 2004

- there are $P$ check node processors, or parity check processors (PCP). Each of them can process $k' = \lceil k/\alpha \rceil$ inputs per clock cycle, $k'$ being a positive integer. If $\alpha = 1$, the $k$ inputs of the PCP are processed at the same time (parallel PCP). If $\alpha = k$, only one input is processed per clock cycle in the PCP (serial PCP).

- There are also $P\lceil \frac{k'}{j'} \rceil$ variable node processors (VP). Each of them can process $j' = \lceil j/\beta \rceil$ inputs per clock cycle, $j'$ being a positive integer. If $\beta = 1$, the $j$ inputs of the VP are processed at the same time (parallel VP). If $\beta = j$, only one input is processed per clock cycle in the VP (serial VP). If the variable processors are based on single port access memories, $j' = 1$ and then there are $k' \times P$ variable processors.

Each of these processor is associated with memory banks which will be described in section 3.4. More details about node processors will be given in section 3.2.

In such an architecture, $P$ check nodes out of $M$ are performed simultaneously. This is repeated $Q$ times, $QP = M$, $Q$ being an integer, to complete an iteration. Varying the parameter $\alpha$, $\beta$ and $P$ values enables to describe all the possible architectures ranging from the whole parallel one ($\alpha = 1$, $\beta = 1$, $P = M$) to the whole serial one ($\alpha = k$, $\beta = j$, $P = 1$), as far as regular $(j, k)-$LDPC codes are concerned. A mixed architecture will refer to neither a parallel nor a serial architecture.

In the following, we denote by $M_{c \to v}$ the message which is sent from the check node processor to the variable node processor through the direct shuffle and by $M_{v \to c}$ the message which is sent from the variable node processor to the check node processor through the reverse shuffle.

### 3.1.2 Shuffle network

The edges of the bipartite graph are wired between the check nodes and the variable node through a permutation network. A message passing decoder architecture features also an interconnection network whose complexity depends on the code itself and on the type of message passing structure.

The particular cases of the whole serial or whole parallel designs do not use any switches: in the parallel case, the switches are hard wired directly, leading to routing problems; in the serial case, the permutation is replaced by a control on the reading addresses to pick up the messages concerned by the current update rule (time shuffling).

For mixed serial-parallel architectures, the interconnection network takes place partly in the time shuffling and partly in the spatial shuffling. The time shuffling is realized by the random bit reading in the memory banks. The spatial shuffling is implemented in a shuffle network. The evaluation of the complexity of the shuffle network is done by using the number ($N_\Pi$) of the elementary switches depicted in figure 3.2. In general, LDPC codes are designed so as to lower the complexity of the shuffle network, using rotations for example.

(c) Frédéric Guilloud, Télécom Paris - July 2004

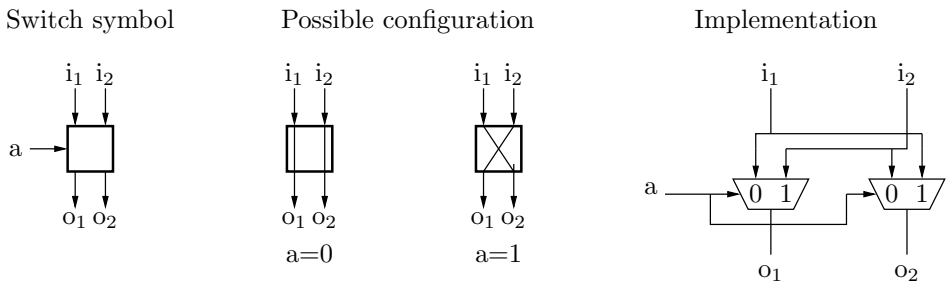



Figure 3.2: Elementary switch

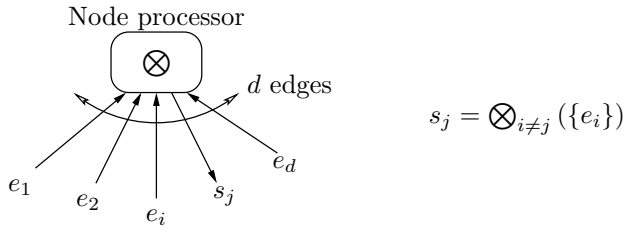



Figure 3.3: A generic node processor

## 3.2 Node processors

### 3.2.1 Generic node processor

#### Data flow

Node processors can be described with the same generic model depicted on figure 3.3. The node processor is defined by its degree $d$, which is the number of edges connected to it, and by the generic commutative operator denoted by $\bigotimes$. The processor has $d$ different input messages denoted $e_i$ and computes $d$ output messages denoted $s_j$ using the $\bigotimes$ operator. The output message on the $i$th edge is a function of the $d-1$ messages input on the $d-1$ other edges:

$$s_j = \bigotimes_{i \neq j} e_i \tag{3.1}$$

There are three main possible implementation of equation (3.1) inside a generic node processor: the *direct* implementation, the *trellis* implementation and the *total sum first* implementation if the $\bigotimes$ operator is invertible. These possible implementations are described below and their associated architectures are depicted on figure 3.4. Note that both parallel and serial architectures are possible.

**Direct implementation:** the $d$ output messages $s_j$ can be computed using $d$ times equation (3.1). So this direct implementation features $d$ times the implementation of equation (3.1). This implementation is only efficient for small values of the degree $d$.

(c) Frédéric Guilloud, Télécom Paris - July 2004

This solution has been already implemented check or variable node processors by (Levine, Taylor, and Schmit 2000; Zhang and Parhi 2003).

**Trellis implementation:** the $d$ output messages $s_j$ can also be computed using a systolic trellis-based implementation. A forward-backward processing is performed to process the $d$ output messages. Such an example is proposed by (Boutillon, Castura, and Kschischang 2000) for the check node processors. Note that a similar solution is also provided by (Mansour and Shanbhag 2002a) to decode parity check equation based on their trellis.

**Total sum first implementation:** if we consider the two expressions (3.1) solved to compute the outputs $s_{j_1}$ and $s_{j_2}$, we see that only one variable has changed (resp. $s_{j_2}$ and $s_{j_1}$). So another possible processing is to replace equation (3.1) by:

$$s_j = \mathrm{inv}_{\otimes}(e_j) \otimes \left( \bigotimes_i e_i \right) \tag{3.2}$$

which is allowed if and only if $\bigotimes$ is an invertible operator: then, $\mathrm{inv}_{\otimes}(e_j)$ denotes the inverse of $e_j$ for the $\bigotimes$ operator. In other words, $e_j \bigotimes \mathrm{inv}_{\otimes}(e_j)$ is equal to the identity. Note that this solution requires the storage of the input meanwhile processing the whole summation. But it is well suited to serial implementations. The total sum first architecture has been proposed for check node processor implementations by for example (Howland and Blanksby 2001a; Yeo, Nikolić, and Anantharam 2001; Blanksby and Howland 2002; Kim, Sobelman, and Moon 2002; Hocevar 2003; Chen and Hocevar 2003) and also for variable node processor implementations by for example (Boutillon, Castura, and Kschischang 2000; Yeo et al. 2001; Howland and Blanksby 2001a; Yeo, Nikolić, and Anantharam 2001; Blanksby and Howland 2002; Kim, Sobelman, and Moon 2002; Mansour and Shanbhag 2002a).

### Control

By control on the generic node processor, we mean to explain how the processor is activated. The processor has $d$ inputs and outputs, so the control of such a processor is in fact the schedule of the input and output messages. We define two modes of control (figure 3.5): the *master* mode and the *slave* mode.

1. In the **master mode**, the processor will first ask for its $d$ input messages at a time $T_1$. After the processing latency, the processor will output its $d$ processed messages.

2. In the **slave mode**, a processor will be asked by a master mode one to output a messages at time $T_1$. It will be asked also to take into account a new incoming messages after a given latency.

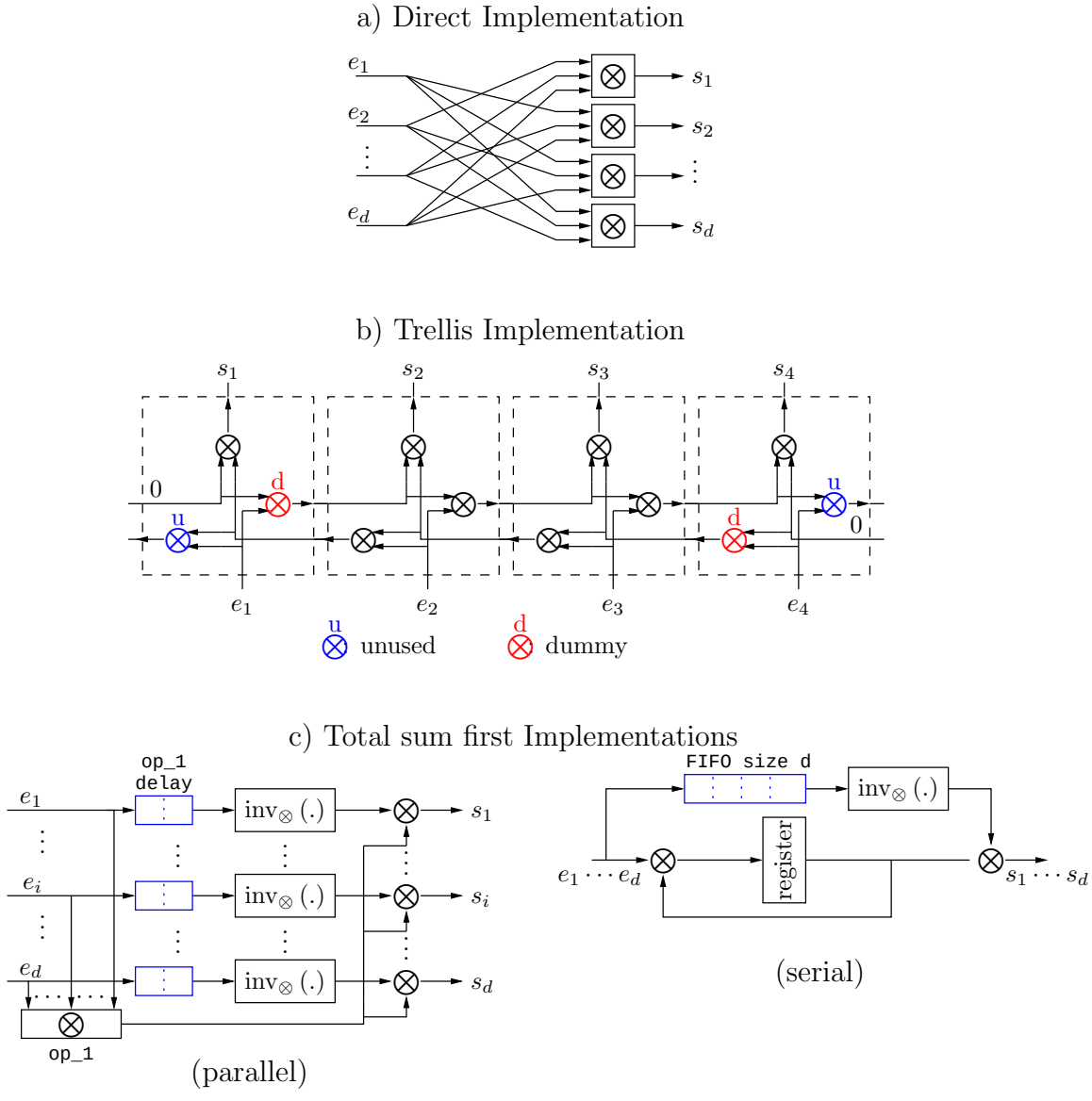



Figure 3.4: Possible implementations for a generic node processor

(c) Frédéric Guilloud, Télécom Paris - July 2004

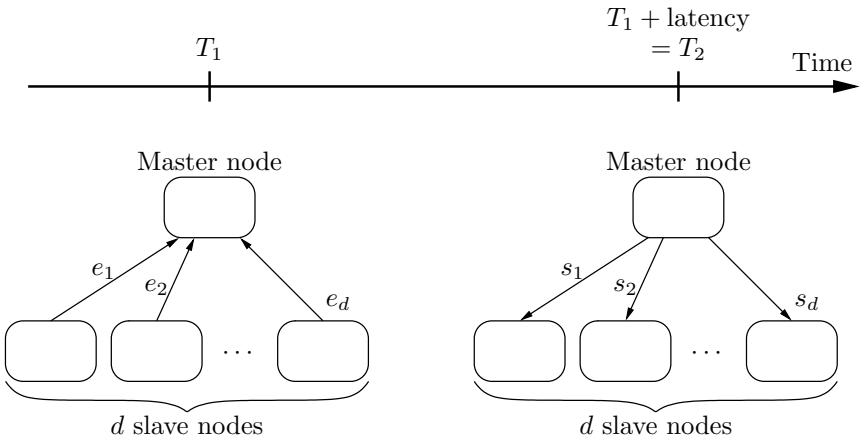



Figure 3.5: The master/slave modes for node processors

From a graph node point of view, the master mode is a parallel mode, and the slave mode is a serial mode. We would rather use the words master and slave so as to avoid any confusion between the control mode and the architecture of the node processor, which can be serial or parallel as well. For example, a master node processor can be serially implemented (one input/output port).

Some more details about the slave mode are now given. We define two kinds of processing for the slave mode: *slow* loop or *fast* loop.

1. In the **slow loop** mode, all the input messages are used to prepare the output of the next iteration.

2. On the contrary, the **fast loop** means that every input message modifies the next output messages of the same iteration. Thus, there is no need to wait for the next iteration to propagate the information, hence the *fast* loop.

So the loop mode indicates how fast the information is updated. Note that the speed can also be in between, when the information is not updated as fast as for the fast loop, but is updated before the end of an iteration, says for example twice within an iteration. The memory requirements are not the same whether the node processor is to be fast-loop slave or slow-loop slave, as depicted on figure 3.6. In the slow loop mode, the information related to the output of the current iteration and to the next iteration are to be saved, whereas in the fast loop, this information is updated within the iteration, in a single memory. This will be illustrated in section 3.4 by some practical examples.

(c) Frédéric Guilloud, Télécom Paris - July 2004

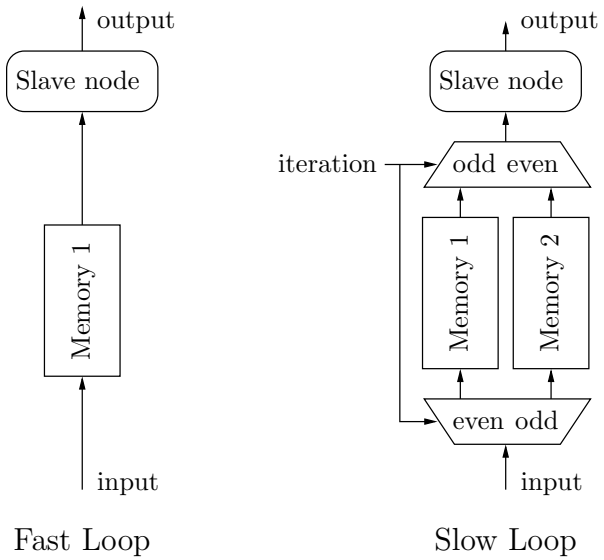



Figure 3.6: The slow/fast modes for slave node processors

### 3.2.2 Variable and check node processors

#### Sign-magnitude processing

For all the different iterative algorithms, the variable update rule is an addition function. It is expressed by (see iterative algorithm 1):

$$T_{n,m}^{(i)} = I_n + \sum_{m' \in \mathcal{M}(n) \backslash m} E_{n,m'}^{(i-1)} \tag{3.3}$$

The check node update rule can be separated into the sign and the magnitude processing, as derived hereafter. The check node update rule is expressed by (see iterative algorithm 1):

$$E_{n,m}^{(i)} = 2 \tanh^{-1} \prod_{n' \in \mathcal{N}(m) \backslash n} \tanh \frac{T_{n',m}^{(i)}}{2} \tag{3.4}$$

We have then from (3.4):

$$\tanh \frac{E_{n,m}^{(i)}}{2} = \prod_{n' \in \mathcal{N}(m) \backslash n} \tanh \frac{T_{n',m}^{(i)}}{2} \tag{3.5}$$

Replacing $T_{n',m}$ by $\operatorname{sign}\left(T_{n',m}\right) \times \left|T_{n',m}\right|$ in (3.5) yields:

$$\operatorname{sign}\left(E_{n,m}^{(i)}\right) = \prod_{n' \in \mathcal{N}(m) \backslash n} \operatorname{sign}\left(T_{n',m}^{(i)}\right) \tag{3.6}$$

$$\tanh \left|E_{n,m}^{(i)}\right| /2 = \prod_{n' \in \mathcal{N}(m) \backslash n} \tanh \left|T_{n',m}^{(i)}\right| /2 \tag{3.7}$$

(c) Frédéric Guilloud, Télécom Paris - July 2004

Let $f(x)$ be defined by:

$$f(x) = -\ln\left(\tanh\left(\frac{x}{2}\right)\right) = \ln\frac{e^x+1}{e^x-1} \tag{3.8}$$

Then, taking the logarithm of the inverse of both side of (3.7) yields:

$$-\ln\tanh\left|E_{n,m}^{(i)}\right|/2 = -\ln\prod_{n'\in\mathcal{N}(m)\backslash n}\tanh\left|T_{n',m}^{(i)}\right|/2 \tag{3.9}$$

$$f\left(\left|E_{n,m}^{(i)}\right|\right) = -\sum_{n'\in\mathcal{N}(m)\backslash n}\ln\tanh\left|T_{n',m}^{(i)}\right|/2 \tag{3.10}$$

$$= \sum_{n'\in\mathcal{N}(m)\backslash n} f\left(\left|T_{n',m}^{(i)}\right|\right) \tag{3.11}$$

$$\text{And because } f\left(f(x)\right) = x, \tag{3.12}$$

$$\left|E_{n,m}^{(i)}\right| = f\left(\sum_{n'\in\mathcal{N}(m)\backslash n} f\left(\left|T_{n',m}^{(i)}\right|\right)\right) \tag{3.13}$$

Note that equation (3.13) can also be written:

$$\left|E_{n,m}^{(i)}\right| = f\left(M_m - f\left(\left|T_{n,m}^{(i)}\right|\right)\right) \tag{3.14}$$

where

$$M_m = \sum_{n'\in\mathcal{N}(m)} f\left(\left|T_{n',m}^{(i)}\right|\right) \tag{3.15}$$

So the iterative algorithms 1 can be written with separate sign and magnitude processing, yielding the following iterative algorithm:

**Iterative Algorithm 3 (BP)** *with the sign-magnitude processing.*

*Initialization:* $E_{n,m}^{(0)} = 0$

*Variable node update rule:* $T_{n,m}^{(i)} = I_n + \sum_{m'\in\mathcal{M}(n)\backslash m} E_{n,m'}^{(i-1)}$

*Check node update rule:* $E_{n,m}^{(i)} = \prod_{n'\in\mathcal{N}(m)\backslash n} sign\left(T_{n',m}^{(i)}\right) \times f\left(\sum_{n'\in\mathcal{N}(m)\backslash n} f\left(\left|T_{n',m}^{(i)}\right|\right)\right)$

*Last variable node update rule:* $T_n = I_n + \sum_{m\in\mathcal{M}(n)} E_{n,m}^{(i-1)}$

*with* $f(x) = \ln\frac{e^x+1}{e^x-1}$

From the iterative algorithm 3, the magnitude processing can be decomposed into 3 steps: first a scale change (the function $f$ in the iterative algorithm 3) puts the messages into a check domain representation, then an addition operation is processed and finally another scale change to get back into the variable domain representation.

(c) Frédéric Guilloud, Télécom Paris - July 2004

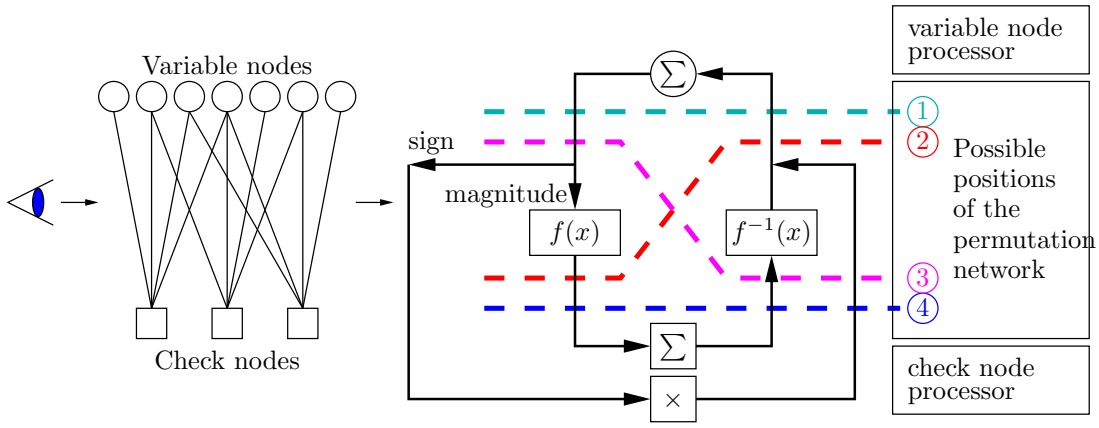



Figure 3.7: The variable and check node processor functions depend on the interconnection network position.

### Variable and check node processor configurations

The combination of the check node and of the variable node update rules, separating the sign and magnitude processing is depicted on figure 3.7, where the 4 possible positions of the interconnection network are symbolized by 4 different dashed numbered lines. The position number 1 is the classical one. It yields to have all the messages being represented in the variable domain. The position number 2 (resp. 3) yields to have the input (resp. output) of the node processors be in their own representation domain, while their output (resp. input) are represented in the other domain. An advantage to these positions is that the node processors have nearly the same architecture, except for the sign processing. The position 2 has been used by (Zhang, Wang, and Parhi 2001). The position number 4 yields to have only check node domain messages between the node processors. This is a new node processor architecture and it fits perfectly with the horizontal schedule of section 3.4.

The operator inside the generic node processor $\bigotimes$ associated with the variable and with the check node processors depends on the choice of the position of the interconnection network. Table 3.1 summarizes the nature of the operator as a function of the position of the interconnection network. The sign processing is independent from the magnitude processing in the check node update rule, so it has been considered as a whole node processor. Note that in the position number 1, it is also possible to use the $\star$-operator, defined in annexe B.1 by the equation (B.6). An implementation of this operator is illustrated on figure B.2.

### Control mode combinations

The combinations between the different controls of the node processors is illustrated in table 3.2. These different combinations are associated to different schedule such as de-

Table 3.1: Node processor operator as a function of the permutation network position.

| | | | Permutation Network Position | | | |
|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | 4 |
| Generic Operator $\otimes$ | Variable Node Processor | | $\sum$ | $f \circ \sum$ | $\sum \circ f^{-1}$ | $f \circ \sum \circ f^{-1}$ |
| | Check Node Processor | Magnitude | $\star$ | $f^{-1} \circ \sum \circ f$ | $f^{-1} \circ \sum$ | $\sum \circ f$ | $\sum$ |
| | | Sign | $\times$ (Sign product) | | | |

Table 3.2: Possible combinations for node control

| | | | Variable | | |
|---|---|---|---|---|---|
| | | | Master | Slave | |
| | | | | Slow loop | Fast loop |
| Check | Master | | Flooding (2 ways) | Flooding (check way) | Shuffle (Horizontal) |
| | Slave | Slow loop | Flooding (var. way) | edge controled | |
| | | Fast loop | Shuffle (Vertical) | | |

(c) Frédéric Guilloud, Télécom Paris - July 2004

scribed in section 2.7.1. The flooding scheduling is divided into three schemes, depending on the order the different processors are activated: the check way flooding schedule means that the check node processors are successively activated; symmetrically, the variable way flooding schedule means that the variable node processors are successively activated. In these two flooding cases, the slave mode is the slow loop one: an iteration has to be completed so as to propagate the information as in the classical flooding schedule. If the slave mode is the fast loop one, the information is updated as the master nodes are activated: it is the shuffle schedule. When both check and variable node processors are controlled in slave mode, the schedule is along the edge. It means that one iteration is controlled by an edge list. This enables to chose exactly the schedule in a given parity-check matrix. More details related to the main schedules are given through the section 3.4.

## 3.3 Complexity analysis

Three parameters will measure the complexity of an LDPC decoder architecture: the edge rate, $R_e$, the amount of memory required and the complexity of the shuffle network, $N_\Pi$. The latter has been defined in section 3.1.2. The other two parameters are described hereafter.

### 3.3.1 Computation requirements

A non-zero entry in the $m$-th row and the $n$-th column of the parity-check matrix corresponds to an edge of the bipartite graph connecting the variable $n$ to the check node $m$. Each connection in the bipartite graph requires the processing of the two messages $M_{v\to c}$ and $M_{c\to v}$. As seen in section 3.2.2, each $M_{v\to c}$ message requires roughly 2 additions and each $M_{c\to v}$ message requires roughly either 2 LUTs and 2 additions when using the $\tanh-$rule, or 3 $\star$-operators. An iteration is over when all the edges of the graph have been so processed. Some schedules may not process all the edges for all iterations, such as in (Mao and Banihashemi 2001a) (see *Probabilistic scheduling* in section 2.7.1). Such algorithms will not be considered hereafter: all the edges are supposed to be processed in the two ways during one iteration. So the number of non-zero entries in the parity-check matrix is proportional to the processing complexity of an iteration. Let $\Gamma$ denotes the number of non-zero entries in the parity check matrix of an LDPC code. For regular $(j,k)-$LDPC codes, we have: $\Gamma = jN = kM$.

### 3.3.2 Message rate

$\Gamma$ is proportional to the amount of elementary operations that are to be processed during one iteration. It is linked to the code and to the BP algorithm. We define the edge rate $R_e$ as a measurement of the processing power; it is defined as the number of edge per

clock cycle that are to be processed to fit the specifications of the decoder. So it depends on $\Gamma$ and on the specifications under which the decoder has to be run:

- the size $K$ of the information bits,
- the information throughput $D_b$ in bit/s,
- the number of iterations $i_{\max}$,
- the clock frequency $f_{\text{clk}}$.

Note that $i_{\max}$ can also be the average number of iterations when input and output buffers are used as in (Martinez and Rovini 2003). According to these specifications, the number of variables (or bits) that have to be process every clock cycle is:

$$\frac{D_b}{R}\ [\text{bit/s}]\ \times 1/f_{\text{clk}}\ [\text{s/cycle}]\ = \frac{D_b}{f_{\text{clk}}R}\ [\text{bit/cycle}] \tag{3.16}$$

Moreover, there are $\Gamma \times i_{\max}$ [edges] to process in order to decode $N$ [bits] , *i.e.*:

$$\frac{\Gamma i_{\max}}{N}\ [\text{edges/bits}] \tag{3.17}$$

So the number of edges processed every clock cycle is equal to:

$$R_e = \frac{D_b}{f_{\text{clk}}R}\ [\text{bit/cycle}]\ \times \frac{\Gamma i_{\max}}{N}\ [\text{edges/bits}]\ = \frac{\Gamma i_{\max} D_b}{K f_{\text{clk}}}\ [\text{edge/cycle}]\ , \tag{3.18}$$

where $K = RN$ is the number of information bits.

If $R_e$ is high, it means that the processing power required is high: a lot of edges per clock cycle should be processed. In that case, a highly parallelized architecture would have to be designed to fit the specifications. Note that $(R_e)_{\max} = \Gamma$: the maximum number of edges that can be processed in one clock cycle is the number of edges itself. $R_e$ is a lower bound on the parallelism factor $P$, which is the number of check node processors running in parallel. $R_e$ is thus also linked to the minimum number of memory blocks which have to be accessed simultaneously. Note that using the notations of the generalized message passing architecture of section 3.1, the edge rate is given by:

$$R_e = P \times \frac{k}{\alpha} \tag{3.19}$$

### 3.3.3 Memory

The memory used in a message passing decoder is specified by its size and also by the number of different blocks required for simultaneous access. The latter one is given by the edge rate $R_e$.

The minimum memory which is needed for LDPC decoders implementing the BP algorithm is used to:

(c) Frédéric Guilloud, Télécom Paris - July 2004

- save the intrinsic information $I_n$ (memory $I$). ($N \times w$ bits) are needed, if we assume that the data is coded using $w$ bits.
- save the variable-to-check messages $M_{v \to c}$ (or equivalently the check-to-variable messages $M_{c \to v}$). ($\Gamma \times w$ bits) are then used to save the messages related to the $\Gamma$ edges.

The minimum memory size is enough for whole parallel decoders ($\alpha = 1$, $\beta = 1$, $P = M$). But mixed architectures ($P < M$), the memory banks associated with the node processors feature additional memory. It is used to avoid to many access to the edge memory. More details about this memory will be given in the practical examples of section 3.4.

(c) Frédéric Guilloud, Télécom Paris - July 2004

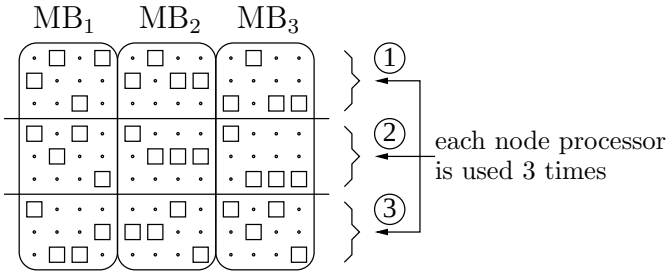



Figure 3.8: Simple $(3, 4)-$parity-check matrix with $(P = 3)$. A square stands for a non-zero entry and a dot for a zero. The memory banks are denoted $\mathrm{MB}_i$.

## 3.4 Synthesis

In sections 3.1 and 3.2, we have proposed respectively a generic message passing architecture and a generic node processor description. These descriptions enable to encompass many the architectures of LDPC decoders which are in the state of the art, nowadays, and even to derive some new interesting architectures. The complexity parameters proposed in section 3.3 can quantify the complexity of these architectures.

In this section, a synthesis of these generic parameters is performed and some practical examples are proposed which illustrate the framework. So an LDPC decoder architecture is specified by:

- the parameters set ($\alpha$, $\beta$, $P$) of the message passing architecture,
- the interconnection network position,
- the control mode of the node processors,
- the architecture of the node processors (data path, serial or parallel implementation).

Theoretically, all the combinations are possible. Since there is a high number of possible combinations, three of the main schedules (flooding (check way), horizontal and vertical shuffle) will be described hereafter, based on a practical example.

We assume first that a mixed architecture ($\alpha$, $\beta$, $P$) is chosen to decode a simple $(3, 4)-$parity-check matrix such as depicted on figure 3.8.

(c) Frédéric Guilloud, Télécom Paris - July 2004

### 3.4.1 Flooding schedule (check way)

To illustrate the flooding schedule (check way), we give hereafter a particular example from the state-of-the-art decoder of (Boutillon, Castura, and Kschischang 2000). The choice of the different parameters of this example is resumed in the table on the top of figure 3.9. In this decoder architecture, $P = 3$ check node processors are controlled in a master mode and perform simultaneously. So there are $P = 3$ check node processors in the decoder architecture. Since $\alpha = 1$ and $\beta = 3$, there are also $P \times \beta/\alpha = 12$ variable node processors. Note that the edge rate is thus $R_e = 12$ [edge/cycle] . As we will see below, there is no need of memory banks MB'$_i$ in the check node processors. But 12 memory banks MB$_i$ are required with the variable node processors. Each of them features the information related to $\frac{Nj'}{Pk'} = N/12$ variables in the code of length $N$.

The check node processor dataflow is a trellis architecture within a parallel implementation. At a time $T_1$, they are fed with $d$ ($d = k = 4$ on the figure) input messages $T_{n,m}$, by the variable node processors. The variable node processors are controlled in a slave mode. After a latency $T_2 - T_1$, the output messages $E_{n,m}$ of the check node processors are fed to the variable node processors. The data flow of the variable node processors is a total-sum first architecture within a serial implementation. The slow-loop mode of the slave control is illustrated by the requirement of 2 memory blocks: the extrinsic memory of the previous iteration $E_{\text{old}}$ and the extrinsic that are being accumulated $E_{\text{acc}}$. An example of one variable node processor and of one check node processor is depicted in the middle of figure 3.9.

During an iteration, all the check node processors are controlled so as to perform successively. For example, at time $T_1$, the check node processor perform the 3rd check node and then at time $T_1'$ it performs the 6th check node of the graph, as depicted at the bottom of figure 3.9

When the iteration is over, the $E_{\text{acc}}$ memory is dumped into the $E_{\text{old}}$ memory and initialized to zero. Another solution is to swap the memories as depicted on figure 3.9 by the dots line. During this next iteration, the $M_{c\to v}$ messages which have been processed in the previous one will be propagated.

Note that in this architecture, the $E_{n,m}$ messages are saved on the variable node side of the shuffle network. They can also be saved on the check node side, as depicted with dots and dashed lines on figure 3.9, such as in (Chen and Hocevar 2003; Guilloud, Boutillon, and Danger 2003b).

For this architecture, the amount of memory required is equal to the sum of $\Gamma w$ ($M_{c\to v}$ memory) and $3Nw$($I$, $E_{\text{acc}}$ and $E_{\text{old}}$ memories), if $w$ bits are used to code the messages.

(c) Frédéric Guilloud, Télécom Paris - July 2004

| Parameter | | value |
|---|---|---|
| Message passing architecture | | ($\alpha = 1$, $\beta = j = 3$, $P = 3$) |
| Shuffle network position | | 1 |
| Variable Node | control | Slave, slow loop |
| | data path | Total sum first, serial |
| Check Node | control | Master |
| | data path | Trellis, parallel |

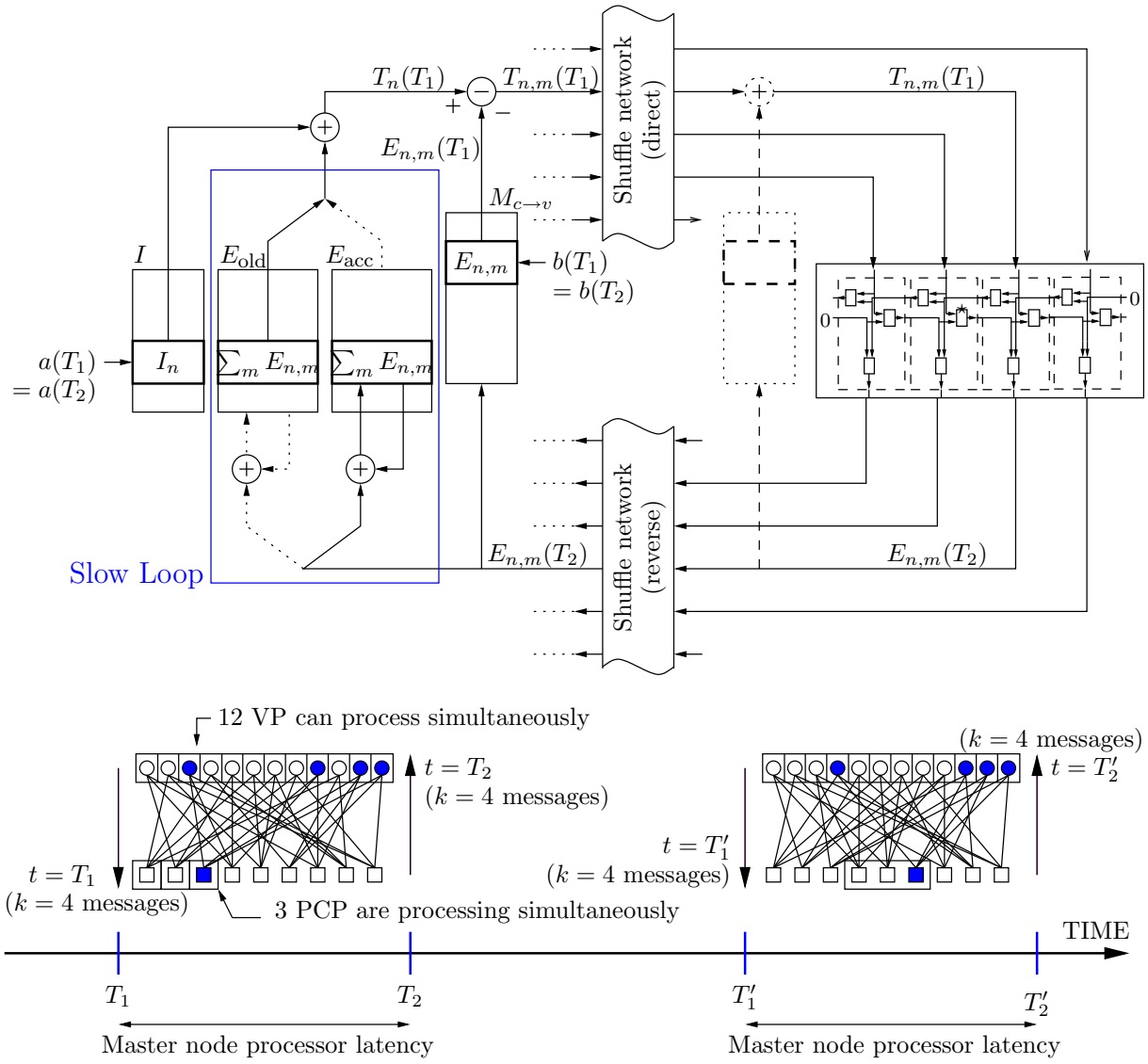



Figure 3.9: Illustration of the Flooding schedule (check way)

(c) Frédéric Guilloud, Télécom Paris - July 2004

### 3.4.2 Horizontal shuffle schedule

The second example illustrates the vertical shuffle schedule. This schedule was first proposed by (Mansour and Shanbhag 2002b) in a specific particular case and is now patent pending (Boutillon, Tousch, and Guilloud 2003) in the United States. The choice of the different parameters of this example is resumed in the upper part of figure 3.10. In this architecture, there are also $P = 3$ check node processor which are controlled in a master mode. Since $\alpha = 4$ and $\beta = 3$, there are also $P = 3$ variable node processors and the edge rate is now $R_e = 3$ [edges/cycle] . As in the flooding schedule, 4 memory banks $\mathrm{MB}_i$ are required with the variable node processors. Each of them features the information related to $\frac{Nj'}{Pk'} = N/3$ variables in the code of length $N$.

The check node processor and the variable node processor data flow are a total-sum first architecture within a serial implementation. At a time $T_1$, the check node processor is serially fed with $d = 4$ input messages $T_{n,m}$, by the variable node processors. The variable node processors are controlled in a slave mode. After a latency $T_2 - T_1$, the output messages $E_{n,m}$ of the check node processors are fed to the variable node processors. The fast-loop mode of the slave control is illustrated by the requirement of only 1 memory block: the $E_{\mathrm{acc}}$ memory block. The sum of the extrinsic messages is updated as soon as a new input message arrives in the variable processor. This schedule is depicted in the middle of figure 3.10, where only the variable processor is detailed. The check node processor is replaced by its generic representation. Note the possibility for the message memory $M_{c\to v}$ to be on both sides of the shuffle network, as in the first example.

As for the flooding schedule, all the check node processors are controlled in a master mode. They perform successively during an iteration. For example, at time $T_1$, the check node processor start to perform the 3rd check node and then at time $T_1'$ it starts to perform the 6th check node of the graph, as depicted at the bottom of figure 3.10.

For the horizontal schedule, the information is propagated as soon as the parity-check has been processed. Only one more kind of memory is then needed: the information that is being updated. The amount of memory required is equal to $\Gamma w + Nw$: avoiding the use of the $E_{\mathrm{old}}$ and the $I_n$ memory saves $2Nw$ bits as compared to the flooding schedule.

(c) Frédéric Guilloud, Télécom Paris - July 2004

| Parameter | | value |
|---|---|---|
| Message passing architecture | | $(\alpha = k = 4,\ \beta = j = 3,\ P = 3)$ |
| Shuffle network position | | 1 |
| Variable Node | control | Slave, fast loop |
| | data path | Total sum first, serial |
| Check Node | control | Master |
| | data path | Total sum first, serial |

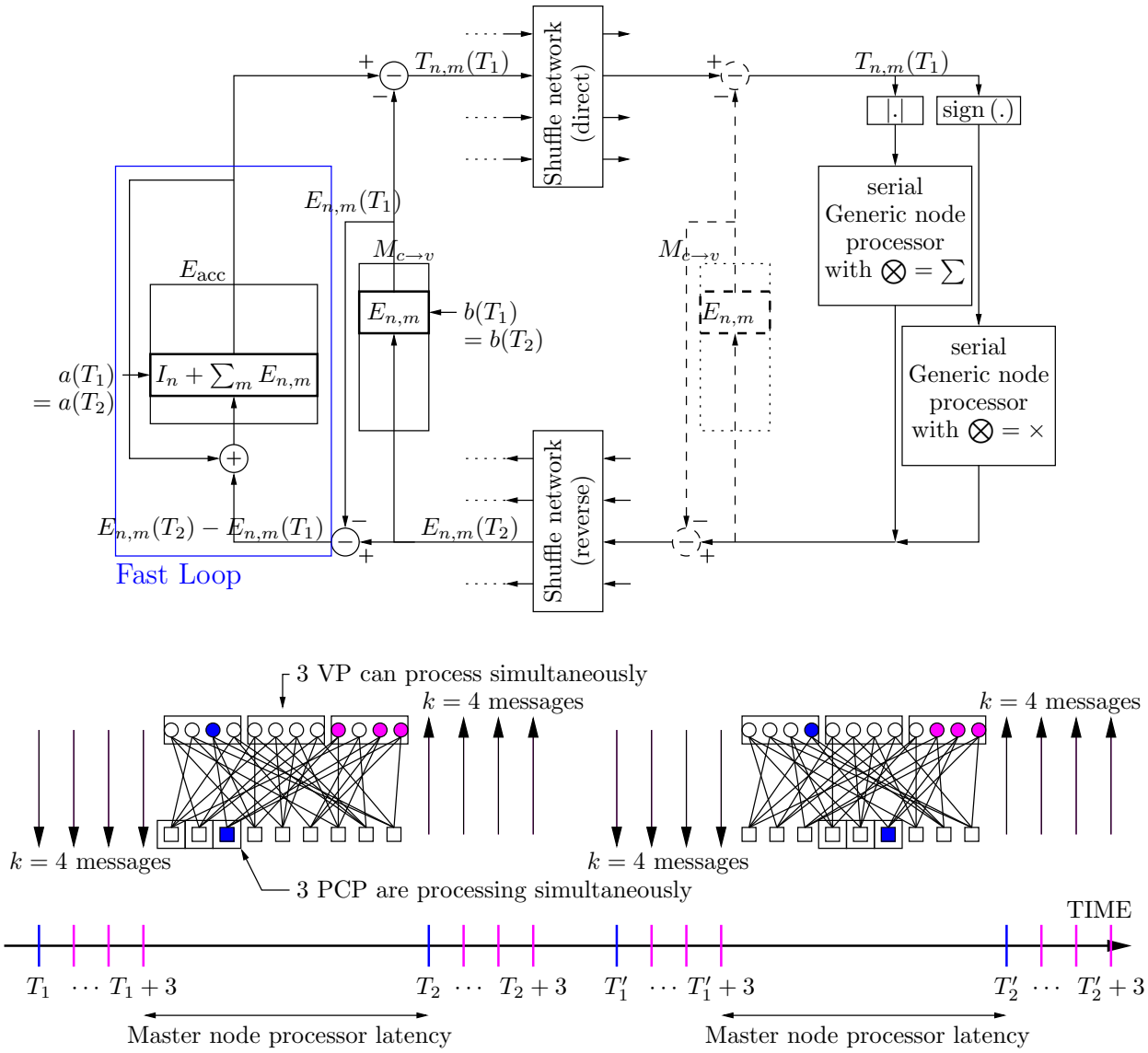



Figure 3.10: Illustration of the Horizontal shuffle schedule

(c) Frédéric Guilloud, Télécom Paris - July 2004

### 3.4.3 Vertical shuffle schedule

The third example illustrates the vertical shuffle schedule. This schedule was first proposed by (Zhang and Fossorier 2002) but we have not been aware of any associated architecture published yet in the literature. The vertical schedule applies the same idea as the horizontal. The difference is that the progression is proceeded along the variable instead of the parity checks. The choice of the different parameters of this example is resumed in the upper part of figure 3.11. In this architecture, there are also $P = 3$ check node processor but they are controlled in a slave mode. Since $\alpha = 4$ and $\beta = 3$, there are also $P = 3$ variable node processors and the edge rate is the same as in the horizontal schedule example. The variable node processors are controlled in a master mode. In this case, there is only one memory $I$ in the memory banks $\mathrm{MB}_i$, whereas in the check node processors, 2 memories are required in the memory banks $\mathrm{MB}'_i$, to save the input messages ($M_{v\to c}$) and also the accumulation of $M_m$, defined by equation (3.15).

The check node processor and the variable node processor data flow are a total-sum first architecture within a serial implementation. At a time $T_1$, the variable node processor is serially fed with $d = j = 3$ input messages $f(E_{n,m})$, by the check node processors. The check node processors are controlled in a fast loop mode. After a latency $T_2 - T_1$, the output messages $f(T_{n,m})$ of the variable node processors are fed serially to the check node processors. The fast-loop mode of the slave control is illustrated by the requirement of only 1 memory block in the check node processor: the $M_m$ memory block. The sum of the input messages is updated as soon as a new input message arrives in the check processor. This schedule is illustrated in the middle of figure 3.11, where the variable node processor has been replaced by its generic representation. Also in the check node processor, the $\bigotimes$ operator has been used to simplify the figure. It should be replaced by the $\sum$ operator for the magnitude processing and by the $\times$ operator for the sign processing. Note also the possibility as in the two previous architectures to place the message memory $M_{v\to c}$ on the both sides of the shuffle network.

On the contrary of the horizontal schedule, all the variable node processors are controlled in the master mode. They perform successively during an iteration. For example, at time $T_1$, the variable node processor start to perform the 6th variable node and then at time $T'_1$ it starts to perform the 9th variable node of the graph, as depicted at the bottom of figure 3.11.

For the vertical schedule, the same number of memory blocks as for the horizontal one are used but the amount of memory required is equal to $\Gamma w + (2 - R)Nw$: only $M$ words are to be saved in the $M_m$ memory, but the intrinsic memory can not be accumulated in the accumulation memory. So $RNw$ more bits are needed as compared to the horizontal schedule.

(c) Frédéric Guilloud, Télécom Paris - July 2004

| Parameter | | value |
|---|---|---|
| Message passing architecture | | $(\alpha = k = 4,\ \beta = j = 3,\ P = 3)$ |
| Shuffle network position | | 4 |
| Variable Node | control | Master |
| | data path | Total sum first, serial |
| Check Node | control | Slave, fast loop |
| | data path | Total sum first, serial |

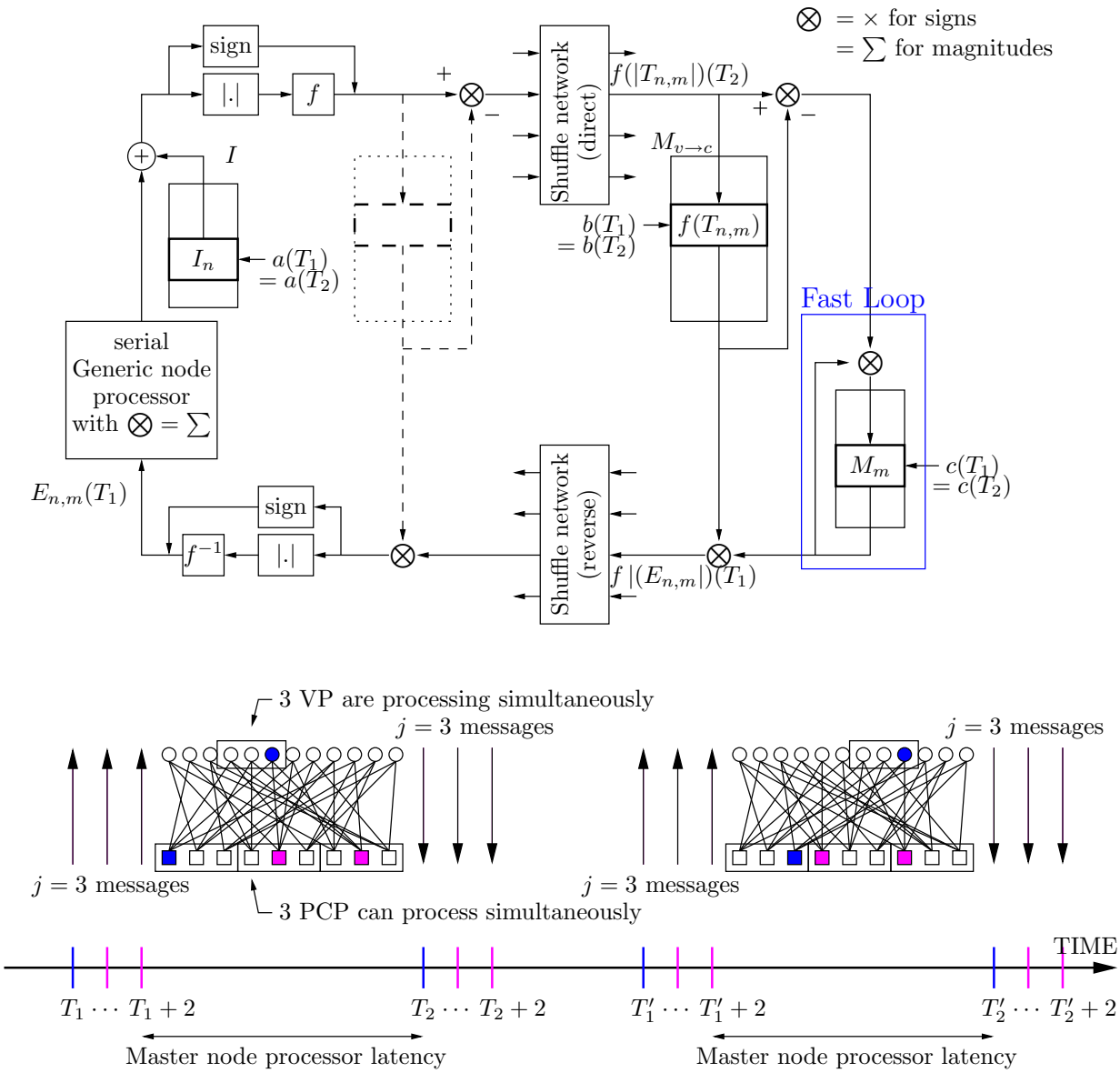



Figure 3.11: Illustration of the vertical shuffle schedule

(c) Frédéric Guilloud, Télécom Paris - July 2004

Table 3.3: Memory requirements for the 3 different schedules

| Schedule | Memory size | Numerical Application | | |
|---|---|---|---|---|
| | | Rate 1/3 | Rate 1/2 | Rate 9/10 |
| Flooding (check way) | $\Gamma w + 3Nw$ | $6Nw$ | | |
| Horizontal | $\Gamma w + Nw$ | $4Nw$ | | |
| Vertical | $\Gamma w + (2 - R)Nw$ | $4.67Nw$ | $4.5Nw$ | $4.1Nw$ |

### 3.4.4 Memory comparison

A memory comparison of the 3 different schedules of the sections 3.4.1, 3.4.2 and 3.4.3 is presented in table 3.3. We assume that an LDPC code of length $N$ has an average variable degree of $j_{\text{average}} = 3$. Then $\Gamma = j_{\text{average}}N = 3N$. We also assume that all the words are coded using the same fixed-point format on $w$ bits. We can see that the horizontal and the vertical schedules have smaller memory requirements than the flooding schedule, check way. The horizontal schedule has the lowest required memory size. The vertical schedule memory size is a function of the code rate: for high code rates, it is possible to use almost the same amount of memory as for the horizontal schedule. However, we can see that the main part of the memory is used to save the edge messages. This issue will be addressed in chapter 4.

## 3.5 Existing platforms survey

We present in this section a survey of the existing and published platforms or designs of LDPC decoders.

### 3.5.1 Parallel designs

The Parallel architectures for message passing decoders achieve the constraints given by the smallest memory size and the highest edge rate $R_e$. Some implementations in ASICs can be found in the literature together with parallel node processors (Howland and Blanksby 2001a; Howland and Blanksby 2001b; Blanksby and Howland 2002). Their node processors are not very complex: the variable node processor have $j = 3, 6, 7, 8$ inputs and their check node processor have $k = 6, 7$ inputs. The average variable node degree is $d_v = 3.25$. So there are $\Gamma = Nd_v = 1024 \times 3.25 = 3328$ edges in the graph. Of course, such an architecture is very fast: a throughput up to $D_b/R = 1$ Gb/s can be reached (information bits and redundant bits), within 64 iterations and at a clock rate of

64 MHz. So the edge rate is:

$$R_e = \frac{\Gamma D_b / Ri_{\max}}{K/Rf_{\text{clk}}} = \frac{3328 \text{ [edges]} \times 1/0.5 \text{ [Gb/s]} \times 64 \text{ [iter]}}{1024/0.5 \text{ [bits]} \times 64 \text{ [MHz]}} \quad (3.20)$$

$$= 3328 \text{ [edges/cycle]} \quad (3.21)$$

$$= \Gamma = (R_e)_{\max} \quad (3.22)$$

This is coherent with this whole parallel architecture. Note that for this numerical example, we assumed that 1 [Gb/s] $= 1024.10^3$ [Gb/s] , since the giga notation is not detailed in the reference. If we assume that 1 [Gb] $= 2^{30}$ [bits] then $R_e$ is a little bit higher than $(R_e)_{\max}$ and if we assume that 1 [Gb] $= 10^9$ [bits] then $R_e$ is a little bit smaller than $(R_e)_{\max}$.

The main difficulty for such a design comes from the routing work of all the wires between check nodes and variable nodes. The messages are coded on 4 bits. So an amount of 3328 edges $\times$ 4 bits $\times$ 2 paths $=$ 26624 wires has to be routed on 5 metal levels. This yielded to lose 50% of the chip surface. In (Thorpe 2002), the authors address the problem of routing complexity by designing random regular LDPC codes with a trade-off between the code performance and the interconnect complexity. If the interconnection is simple, the girth is low and hence the performance is low. On the contrary, if the girth has to be higher, the interconnect complexity should be higher as well. Another drawback of parallel architecture is that the decoder is not tunable. The code is hardwired and can not be changed.

Introducing serial processing can circumvent these drawbacks, as described in the next section.

### 3.5.2 serial design

In (Levine, Taylor, and Schmit 2000; Bhatt, Narayanan, and Kehtarnavaz 2000), there is only one check node processor instantiation and only one variable processor instantiation. Both of them is controlled in the master mode. Three RAM memory blocks are used to save intrinsic information $I_n$, the check-to-variable messages $M_{c\to v}$ and the variable-to-check messages $M_{v\to c}$. In this case, there is no routing congestion. But the price to pay is the long time required to achieve one iteration decoding and which increases as the number of nodes increases. The amount of memory required is also even much more important than in the design example of section 3.4.1 implementing the check-way flooding schedule since it is given by $Nw+2\Gamma w$ instead of $3Nw+\Gamma w$ and because $\Gamma > 2N$.

### 3.5.3 Mixed designs

Mixed designs seem the most popular architectures in the literature for LDPC decoders. For example in (Zhang and Parhi 2002), the authors designed a mixed parallel architecture for a length $N = 9216$ regular $(3, 6)-$LDPC code. A throughput of 56 Mb/s is obtained

for 18 iterations. So one iteration requires:

$$\frac{9216 \text{ bits}}{56 \text{ Mb/s} \times 18 \text{ iter}} \times 56 \text{ MHz} = 512 \text{ cycles} \tag{3.23}$$

yielding an edge rate of:

$$R_e = \frac{3 \times 9216}{512} = 54 \text{ edges / cycles} \tag{3.24}$$

The check node processors are controlled in a master mode, and the variable processors in a slave mode, with slow-loop information propagation.

### 3.5.4 Summary

The formalism proposed in this chapter enables to describe also the existing platforms published in the literature. They have been summed up in table 3.4. The entries of the table are described below:

**Target** describes the chip target of the decoder. Synthesis means that the design has not been further.

**Authors** refers to the corresponding references in the bibliography.

**Architecture** is divided into four entries:

1. **decoder** specifies whether the decoding is parallel, serial or mixed and gives also the parameters $(P, \alpha, \beta)$;
2. **data path** is for the node processor data path (Direct, trellis or Total sum);
3. **control** is for the node processor control (Master, slave slow-loop or slave fast-loop);
4. **perm. pos.** is for the position of the permutation network, which defines the node processor operators;

**LDPC Code Type** gives the main parameters of the LDPC code.

**Data (bits)** is the number of bits of the fixed-point format implemented in the decoder;

**clk (Mhz)** is the clock rate.

**rate ($10^6$ bits/sec)** is the throughput of the decoder (information bits and redundant bits).

**i_max** is the maximum number of decoding iterations.

(c) Frédéric Guilloud, Télécom Paris - July 2004

Table 3.4: State of the art of published LDPC platforms

| Target | Authors | architecture | | | | | | | | LDPC code type | | | | Data (bits) | clk (MHz) | rate (Mb/s) | i_max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | decoder | | | data path | | control | | perm. po | N | M | R | profile | | | | |
| | | P | alpha | beta | check | var. | check | var. | | | | | | | | | |
| synthesis | Levine et. al. 2000 | 1 | serial 1 | 1 | parallel (no LLR) - control unknown | | | | | 1200 | 900 | 0.25 | j=3; k=4 | 8 | 50 | 4 | 10 |
| DSP - TMS320C6201 | Bhatt et. al. 2000 | | serial (?) | | serial (?) | | | | 1 | 200 | 100 | 0.5 | j=3; k=6 | 16 | 200 | 0.133 | |
| ASIC 0.6um 7.5x7 sq.mm) | Blanksby et. al. 2002 | M | parallel 1 | 1 | Total sum | | Master | | | 1024 | 512 | | j=3,6,7,8 ; k=6,7 | 4 | 64 | 1000 | 64 |
| FPGA Xilinx Virtex-E 2600 | zhang et. al.2002 | 3k | mixed 1 | 1 | trellis | direct | master | master | 3 | 9216 | 4608 | | j=3; k=6 | 5 | 56 | 54 | 18 |
| ASIC 0.18um (3.1x4.2 sq.mm) | Mansour et. al. sept 2003 | 64 | mixed 1 | 1 | trellis | Total sum | master | slave fast-loop | 1 | 2048 | 1024 | | | 4 | 125 | 1600 | 1 |
| ASIC 0.18um (9.41 sq.mm) | Mansour et. al. dec. 2003 | 64 | mixed 1 | 1 | | | | | | 2304 | 1536 | 2/3 | irregular | 5 | 200 | 192 | 10 |
| FPGA Xilinx Virtex-II 8000 | Chen. Y. et. al. 2003 | 24 | mixed 1 | j | Total sum | | master | slave slow-loop | | 8088 | 4044 | 0.5 | irregular | 6 | 44 / 212 | 80 / 376 | 25 |

(c) Frédéric Guilloud, Télécom Paris - July 2004

## 3.6 Conclusion

In this chapter, we have laid out a framework which unifies the architectures of LDPC decoders. This framework features a generic description of the message passing architecture and a generic description of the node processors. It also features three parameters which describe completely the complexity of the decoder. The combinations of the different generic parameters of these models enables to describe the state of the art LDPC decoder architectures. Moreover, we also found a new interesting architecture associated to the vertical shuffle schedule, which has not been published yet. Finally, a survey of the published platforms using the BP algorithm has been made and described as instantiations of our generic framework. From this framework, it appears that a major milestone of LDPC decoder designs is the management of the memory: even for parallel decoders, a considerable amount of memory is required for storing the edge messages. A solution to this issue would be to search toward approximations of the BP algorithm.

(c) Frédéric Guilloud, Télécom Paris - July 2004

# Chapter 4

# $\lambda-$Min Algorithm

**Summary:**

*In the previous chapter, the architectures of LDPC codes decoder based on the BP algorithms have been studied. It appeared that despite some different scheduling, the amount of memory required for such algorithms is very important because all the edge information ($M_{c\to v}$ or $M_{v\to c}$ messages) have to be saved until the next iteration. Some simplifications can also be made to lower the complexity of the BP algorithm, and particularly the complexity of the check node update rule. A trade-off is then to be decided between the simplifications of the algorithm, and the loss of performance.*

*In this chapter, a new algorithm, named $\lambda-$Min algorithm, for updating extrinsic information is proposed as a performance-versus-complexity trade off between the BP algorithm and the BP-based algorithm. Simulation results show that good performance can be achieved, and which can even be improved by the addition of an offset. The architecture of a check node processors implementing the $\lambda-$min algorithm is also studied.*

*The material of this chapter has been published partly in (Guilloud, Boutillon, and Danger 2003b) and partly in (Guilloud, Boutillon, and Danger 2003a) and is patent pending (Boutillon, Tousch, and Guilloud 2003) in the United-States.*

## 4.1 Motivations and state of the art

There are 2 classes of suboptimal iterative algorithm: hard one and soft one. The most famous hard decoding algorithms are based on the Gallager's algorithm A (Gallager 1963) and the various bit-flipping algorithms. These algorithm are very fast and can be implemented for optical communications for example (Djordjevic, Sankaranarayanan, and Vasic 2004, and references therein). Iterative hard decoding algorithms will not be addressed here.


(c) Frédéric Guilloud, Télécom Paris - July 2004

Iterative soft decoding however performs better than the hard one. The aim of this chapter is to reduce both the complexity of the node processors and the amount of memory needed to store the edge messages. A new algorithm called the λ−min algorithm is proposed as a solution to this issue. Note that in the following, only the flooding schedule is considered.

### 4.1.1 APP-based algorithms

The first idea for solving the memory bottleneck of the BP algorithm is simply to get rid of the edge memory. This idea is applied in an algorithm used in the staggered scheduling, studied by (Yeo, Nikolić, and Anantharam 2001) but first developed by (Fossorier, Mihaljević, and Imai 1999) where it is denoted by *APP algorithm* and *APP-based algorithm*. We denote this algorithm APP-variable because the approximation appears in the variable node update rule:

**Iterative Algorithm 4 (APP-variable)** *The edge message $E_{n,m}^{(i-1)}$ of the previous iteration is removed from the variable node update rule of the iterative algorithm 1*

$$\left[\begin{array}{ll} \textit{Initialization:} & E_{n,m}^{(0)} = 0 \\ \textit{Variable node update rule:} & T_n^{(i)} = I_n + \sum_{m \in \mathcal{M}(n)} E_{n,m}^{(i-1)} \\ \textit{Check node update rule:} & E_{n,m}^{(i)} = 2 \tanh^{-1} \prod_{n' \in \mathcal{N}(m) \backslash n} \tanh \frac{T_{n'}^{(i)}}{2} \end{array}\right.$$

Symmetrically, the same approximation can also be made in the check node update rule. We denote this algorithm as the APP-check algorithm.

**Iterative Algorithm 5 (APP-check)** *The edge message of the previous iteration $E_{n,m}^{(i-1)}$ is removed from the variable node update rule of the iterative algorithm 1.*

$$\left[\begin{array}{ll} \textit{Initialization:} & E_{n,m}^{(0)} = 0 \\ \textit{Variable node update rule:} & T_{n,m}^{(i)} = I_n + \sum_{m' \in \mathcal{M}(n) \backslash m} E_{n,m'}^{(i-1)} \\ \textit{Check node update rule:} & E_{n,m}^{(i)} = E_m^{(i)} = 2 \tanh^{-1} \prod_{n \in \mathcal{N}(m)} \tanh \frac{T_{n,m}^{(i)}}{2} \end{array}\right.$$

During the first iterations, these two algorithms perform well. But the performance is not much improved if the number of iterations is higher. In fact the convergence behaves as if the girth of the graph would be 2 instead of 4, with the loop $(cn_m \rightarrow vn_n \rightarrow cn_m)$ for the APP-variable iterative algorithm, and with the loop $(vn_n \rightarrow cn_m \rightarrow vn_n)$ for the APP-check iterative algorithm.

(c) Frédéric Guilloud, Télécom Paris - July 2004

### 4.1.2 BP-based algorithm

Another approximation for the BP algorithm exist in the literature: the BP-based algorithm from (Fossorier, Mihaljević, and Imai 1999). The approximation used is analog to standard approximation of the Max-Log-MAP of (Hagenauer, Offer, and Papke 1996):

**Iterative Algorithm 6 (BP-Based)** *Only the magnitude part of the check node update rule differs from the iterative algorithm 3:*

*Initialization:* $E_{n,m}^{(0)} = 0$

*Variable node update rule:* $T_{n,m}^{(i)} = I_n + \sum_{m' \in \mathcal{M}(n) \setminus m} E_{n,m'}^{(i-1)}$

*Check node update rule:* $E_{n,m}^{(i)} = \prod_{n' \in \mathcal{N}(m) \setminus n} sign\left(T_{n',m}^{(i)}\right) \times \min_{n' \in \mathcal{N}(m) \setminus n} \left|T_{n',m}^{(i)}\right|$

*Last variable node update rule:* $T_n = I_n + \sum_{m \in \mathcal{M}(n)} E_{n,m}^{(i-1)}$

There is an important simplification in the BP-based algorithm since the check node update is replaced by a selection of the minimum input value. The memory and the complexity reduction is significant since no more LLR are to be computed and since only two messages need to be saved for each parity-check equation. Moreover, there is no need to estimate the noise variance $\sigma^2$ to compute the intrinsic information $I_n$, thanks to the use of the min operator only. But this complexity reduction is made at the expense of a substantial loss in performance.

In (Chen and Fossorier 2002), an improvement is made to the BP-based algorithm by using a correction factor in the check node update rule. It is denoted by *offset* BP-based algorithm when the correction factor is subtracted to the minimum value, or *normalized* BP-based algorithm when it is multiplied by the correcting factor. The result of the check node update equation, which is over-estimated for the BP-based algorithm, is then closer to the result obtained with the BP algorithm. However, the offset or the normalized BP-based algorithm do not seem to achieve good performance for irregular LDPC codes with long length (Chen 2003).

## 4.2 The $\lambda-$Min Algorithm

In the BP algorithm, the magnitude processing is run using the function $f$ defined in equation (3.8) by: $f : x \mapsto -\ln\tanh(|x/2|)$. It is depicted on figure 4.1. The $f-$function shape yields $f(x)$ to be large when $x$ is small, and to be small when $x$ is large. So $\sum_x f(x)$ is mainly dictated by the smallest values of $x$. The proposed algorithm called $\lambda-$min algorithm takes into account in the check node processing the $\lambda$ inputs which have the minimum magnitude.

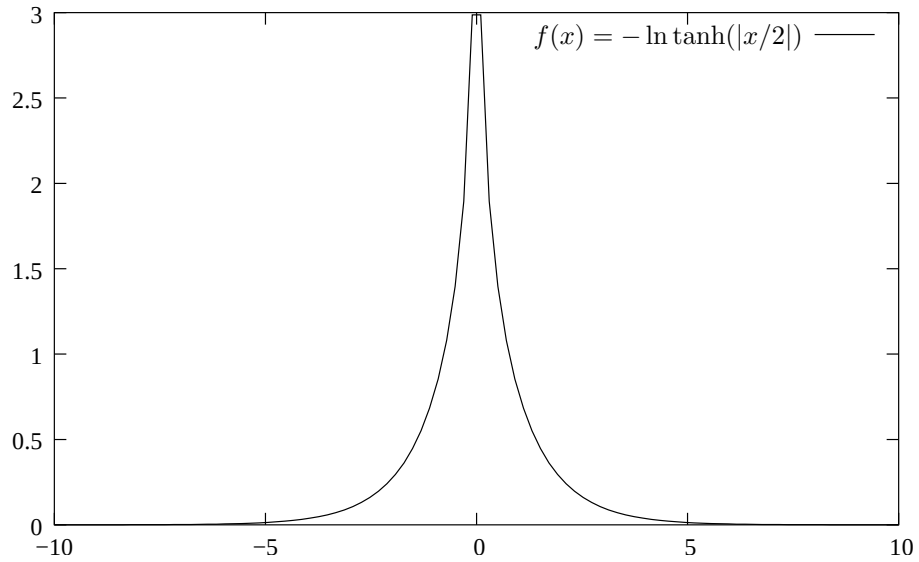



Figure 4.1: Shape of the function $f$.

Let $\mathcal{N}_\lambda(m) = \{n_0, \cdots, n_{\lambda-1}\}$ be the subset of $\mathcal{N}(m)$ which contains the $\lambda$ bits of the parity check $cn_m$ which LLR have the smallest magnitude. The $\lambda-$min algorithm is then:

**Iterative Algorithm 7 ($\lambda-$min)** *Only the magnitude part of check node update rule differs from the iterative algorithm 3:*

*Initialization:* $E_{n,m}^{(0)} = 0$

*Variable node update rule:* $T_{n,m}^{(i)} = I_n + \sum_{m' \in \mathcal{M}(n) \backslash m} E_{n,m'}^{(i-1)}$

*Check node update rule:*
$$E_{n,m}^{(i)} = \prod_{n' \in \mathcal{N}(m) \backslash n} sign\left(T_{n',m}^{(i)}\right) \times f\left(\sum_{n' \in \mathcal{N}_\lambda(m) \backslash n} f\left(\left|T_{n',m}^{(i)}\right|\right)\right)$$

*Last variable node update rule:* $T_n = I_n + \sum_{m \in \mathcal{M}(n)} E_{n,m}^{(i-1)}$

*with* $f(x) = \ln \dfrac{e^x + 1}{e^x - 1}$

Two cases occur:

1. if the bit $vn_n$ belongs to the subset $\mathcal{N}_\lambda(m)$, then the $E_{n,m}^{(i)}$ are processed over the $\lambda - 1$ values of $\mathcal{N}_\lambda(m) \backslash n$
2. if the bit $vn_n$ does not belong to the subset $\mathcal{N}_\lambda(m)$, then the $E_{n,m}^{(i)}$ are processed over the $\lambda$ values of $\mathcal{N}_\lambda(m)$.

So the check node update rule is processed only $\lambda+1$ times: for the $\lambda$ bits which belong to $\mathcal{N}_\lambda(m)$ and one more processing which is sent to all the other bits. Note that the 2−min

Table 4.1: Comparison between the outputs of the BP, BP-based and $\lambda-$min algorithms ($\lambda = \{2, 3, 4\}$)

| inputs | Outputs | | | | |
|---|---|---|---|---|---|
| (LLR values) | BP-based | 2−min | 3−min | 4−min | BP |
| 0.26296 | -0.31502 | -0.31502 | -0.08775 | -0.02555 | -0.00088 |
| 0.31502 | -0.26296 | -0.26296 | -0.07342 | -0.02138 | -0.00074 |
| -0.57686 | 0.26296 | 0.04085 | 0.04085 | 0.01190 | 0.00041 |
| -0.59992 | 0.26296 | 0.04085 | 0.01146 | 0.01146 | 0.00040 |
| -0.67982 | 0.26296 | 0.04085 | 0.01146 | 0.00334 | 0.00035 |
| 0.85523 | -0.26296 | -0.04085 | -0.01146 | -0.00334 | -0.00029 |
| 1.04061 | -0.26296 | -0.04085 | -0.01146 | -0.00334 | -0.00024 |
| 1.22983 | -0.26296 | -0.04085 | -0.01146 | -0.00334 | -0.00021 |

algorithm ($\lambda = 2$) differs only from the BP-based algorithm by the approximation of $E_{n,m}^{(i)}$ in the case where $n \notin \mathcal{N}_2(m) = \{n_0, n_1\}$.

Some LLR computed with different algorithms on the same channel input are given in table 4.1, where the input are listed in the ascending order of their magnitude, for a check node processor of degree 6. We can observe that all the approximations of the BP algorithm are over-evaluated. Of course, when $\lambda$ increases, the approximation is improved.

## 4.3 Performance of the $\lambda-$Min Algorithm

### 4.3.1 Simulation conditions

A C++ program has been written for simulating various kind of matrices with various kind of algorithm.

There is no encoding program so the decoder generates noisy samples from the all-zero codeword. The received bit are decided by comparing the total information to zero. The bit is considered to be wrong The Random Number Generators for the simulation of the AWGN channel comes from (Press, Teukolsky, and Vetterling 1992), to be portable and secure on the randomness point of view.

All the simulations are specified with:

- the $(E_b/N_0)_{\mathrm{dBstart}}$, $(E_b/N_0)_{\mathrm{dBstop}}$ and $(E_b/N_0)_{\mathrm{dBstep}}$;
- the maximum number of iterations $i_{\max}$. The syndrome is computed at each iteration. If the syndrome is equal to zero, the iterations are stopped.
- the maximum number of errors (bit or word) to be reached before increasing the $(E_b/N_0)_{\mathrm{dB}}$ value.

Table 4.2: Bit node distribution degree for code $\mathcal{C}_2$. Actual distribution is computed for a length $N = 2000$ code. Another constraint is that the number of different bit degree should be a multiple of 4.

| $\lambda(x) = \sum \lambda_i x^{i-1}$ (specified) | | $\lambda(x) = \sum \lambda_i x^{i-1}$ (actual for $N = 2000$) | | |
|---|---|---|---|---|
| $i$ | $\lambda_i$ | $i$ | $\lambda_i$ | Number of bits |
| 3 | 11.5897 % | 3 | 11.8721 % | 624 |
| 4 | 17.8648 % | 4 | 18.2648 % | 720 |
| 8 | 16.1923 % | 8 | 16.6413 % | 328 |
| 9 | 6.20557 % | 9 | 6.39269 % | 112 |
| 20 | 13.574 % | 20 | 13.6986 % | 108 |
| 22 | 3.25932 % | 22 | 3.34855 % | 24 |
| 32 | 7.01215 % | 32 | 7.30594 % | 36 |
| 67 | 14.4807 % | 67 | 15.2968 % | 36 |
| 74 | 0.24826 % | 74 | 0 % | 0 |
| 81 | 2.92047 % | 81 | 2.05479 % | 4 |
| 101 | 6.65273 % | 101 | 5.1243 % | 8 |

- the maximum number of words that are to be generated for each $(E_b/N_0)_{\text{dB}}$.

Unless specified, all the simulations ends when 500 erroneous codewords are detected or when a maximum of $10^5$ codewords have been generated. A bit is said to be wrong if the intrinsic information $I_n$ is negative, and it is said to be right if it is positive. Since only floating point operations are considered in this section, the zero case has an extremely low probability.

#### Codes used for simulations

The code $\mathcal{C}_1$ is taken from the MacKay's online database (Mackay ). It is a regular $(5, 10)-$LDPC code of length $N = 816$. The code $\mathcal{C}_2$ has been randomly designed based on the variable degree distribution from Urbanke's online program (Urbanke ). It is an irregular LDPC code of length $N = 2000$ and code rate $R = 0.85$. The distribution variable degree are listed on the tables 4.2 and 4.3. As the length of the code is not high enough, the actual edge distribution degree is not exactly the same as the specified on.

### 4.3.2 Algorithm Comparison

An comparison between the $\lambda-$min algorithm, the BP-based and the BP algorithm is depicted on figure 4.2 for a regular code $\mathcal{C}_1$ and on figure 4.3 for an irregular code $\mathcal{C}_1$. Two conclusions can be made for this comparison:

Table 4.3: Check node distribution degree for code $\mathcal{C}_2$. The constraints are to have only 2 different distribution degree and that the number of different check node degree should be a multiple of 4.

| $\rho(x) = \sum \lambda_i x^{i-1}$ (specified) | |
|---|---|
| $i$ | $\rho_i$ |
| 42 | 100 % |

| $\rho(x) = \sum \rho_i x^{i-1}$ (actual) | | |
|---|---|---|
| $i$ | $\rho_i$ | Number of checks |
| 52 | 43.5312 % | 132 |
| 53 | 56.4688 % | 168 |

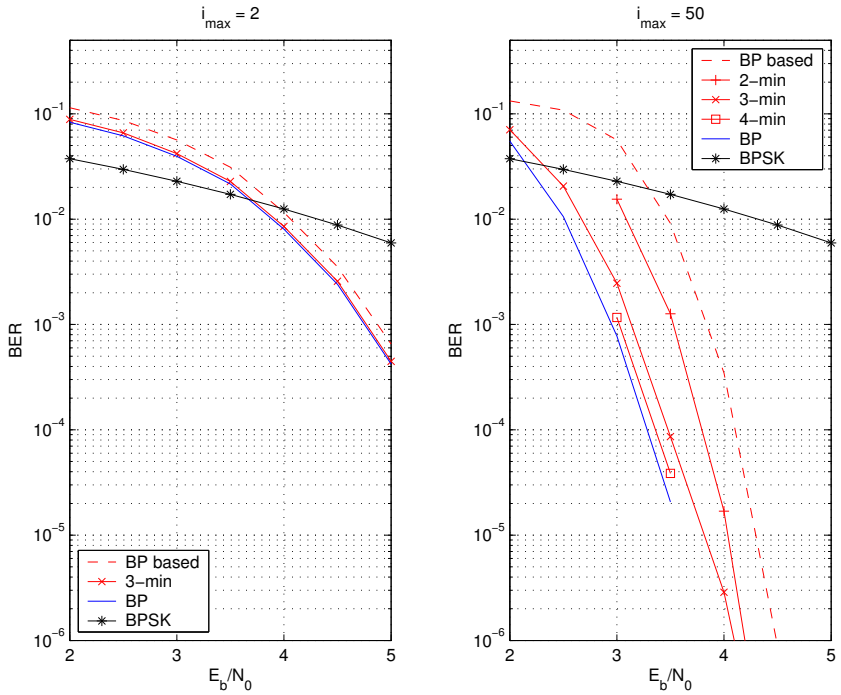



Figure 4.2: Performance of $\lambda-$Min algorithm with $\lambda = \{2, 3, 4\}$ for code $\mathcal{C}_1$.

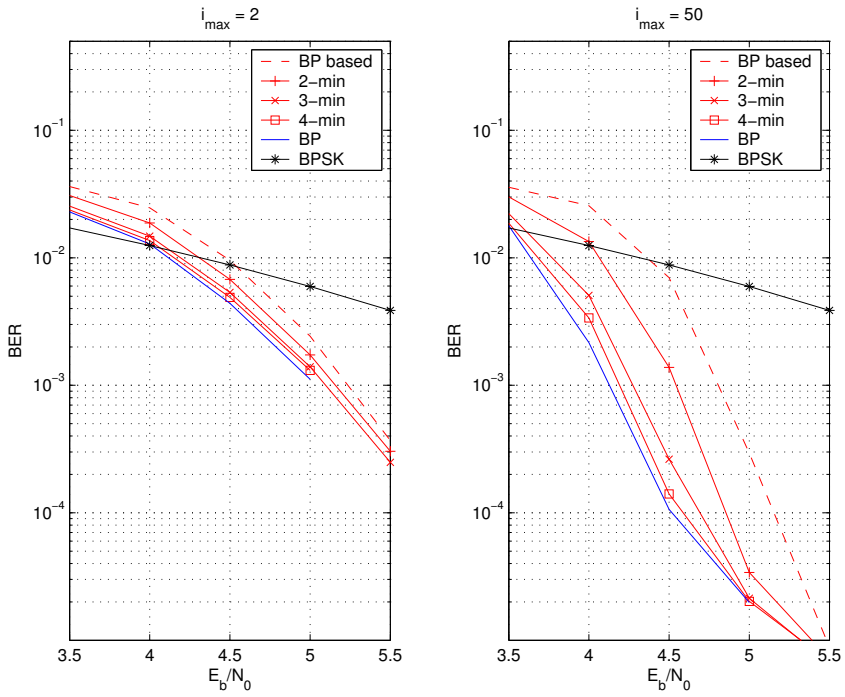



Figure 4.3: Performance of $\lambda-$Min algorithm with $\lambda = \{2, 3, 4\}$ for code $\mathcal{C}_2$.

1. First, the performance loss for all the cases is increasing with the maximum number of iterations. And the differences between the suboptimal algorithm performance is also increasing.

2. The performance of the $\lambda-$min algorithm is very closed to the performance of the BP algorithm for $\lambda \geq 4$. It is as more noticeable that for the code $\mathcal{C}_2$, the check nodes are of degree 52 and 53. So the performance of the $\lambda-$min algorithm is not really a function of the proportion between $\lambda$ and the maximum check node degree.

### 4.3.3 Optimization

As shown in the previous section, the $\lambda-$min algorithm improves the performance compared to the BP-based algorithm but there is still a degradation compared to the BP algorithm. This degradation can be reduced by the addition of an offset, as proposed by (Chen and Fossorier 2002) in the case of BP-Based algorithm: the offset compensates the over estimation of extrinsic information.

**Iterative Algorithm 8 (Compensed $\lambda-$min)** *Only the magnitude part of check node update rule differs from the iterative algorithm 7:*

$$
\begin{aligned}
&\textit{Initialization:} && E_{n,m}^{(0)} = 0 \\
&\textit{Variable node update rule:} && T_{n,m}^{(i)} = I_n + \sum_{m' \in \mathcal{M}(n) \backslash m} E_{n,m'}^{(i-1)} \\
&\textit{Check node update rule:} && E_{n,m}^{(i)} = \prod_{n' \in \mathcal{N}(m) \backslash n} sign\left(T_{n',m}^{(i)}\right) \\
& && \times \max\left(\alpha f\left(\sum_{n' \in \mathcal{N}_\lambda(m) \backslash n} f\left(\left|T_{n',m}^{(i)}\right|\right)\right) - \beta, 0\right) \\
&\textit{Last variable node update rule:} && T_n = I_n + \sum_{m \in \mathcal{M}(n)} E_{n,m}^{(i-1)} \\
&\textit{with } f(x) = \ln \frac{e^x + 1}{e^x - 1} \textit{ and } 0 < \alpha \leq 1, \beta > 0 &&
\end{aligned}
$$

The optimization of the value of the parameters $\alpha$ and $\beta$ can be proceeded by using the density evolution algorithm (Chen and Fossorier 2002). The parameters are then optimized for the threshold of the code. For a practical case (finite code length and finite number of iterations), the optimization can be made to get the best bit error rate for example. The optimal value of $\beta$ can then be simply found through simulations. Figures 4.4 depicts the evolution shape of the BER as a function of $\beta$, for several SNRs and for several max iteration number. The maximum number of word errors is only 50, within a maximum of $10^5$ transmitted codewords. One can note that the offset 3−min algorithm outperforms the BP algorithm. This is explained by a faster convergence; when the number of iterations increases (figure 4.5), the two algorithm perform almost identically. A comparison between the BP, the 3−min and the $3 - min$ with an offset $\beta = 0.35$ is depicted on the figure 4.6. On the BER curves (left side), we can observe that the 3−min outperforms the BP algorithm when an optimized offset is chosen. But the error floor on the FER (right side) which is introduced by the 3−min algorithm does not outperform the BP algorithm beyond $\frac{E_b}{N_0} = 4$ [dB] .

## 4.4 Architectural issues

This section is related to the architecture of the parity check node processor (PCP) for an efficient implementation of the $\lambda$−min algorithm.

### 4.4.1 PCP architecture

#### Description

The implementation is based on a serial parity check processor using the $\star$ operator, defined in annexe B.1 by the equation (B.6), with message propagation as in iterative algorithm 2. A synoptic of the architecture of the PCP is depicted on figure 4.7. It is assumed that each PCP has to process $Q$ successive parity checks, as described in the mixed decoder architecture of section 3.1.

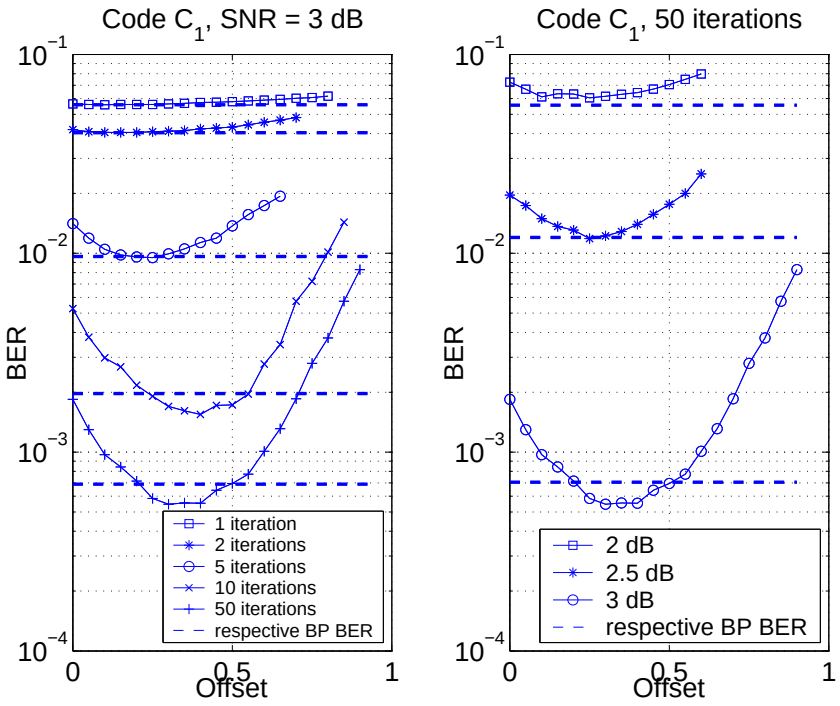



Figure 4.4: Influence of the offset value on the BER for the 3−min algorithm, for several SNRs and several max iteration number, for $\mathcal{C}_1$. The BER obtained with the BP algorithm is represented with a dashed line.

(c) Frédéric Guilloud, Télécom Paris - July 2004

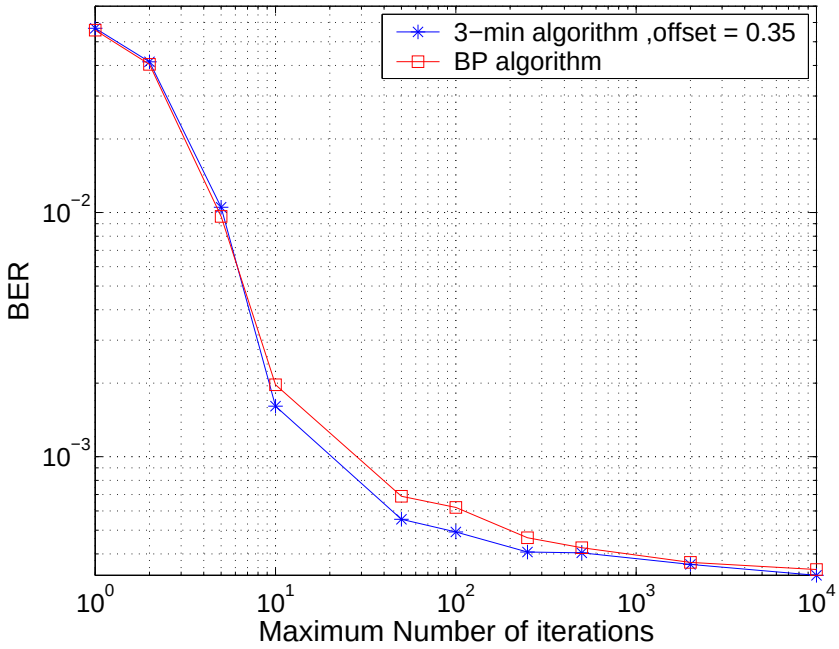



Figure 4.5: Code $\mathcal{C}_1$: comparison between BP algorithm and offset 3−min algorithm as a function of the number of iterations for a fixed SNR of 3 dB.

Each PCP in iteration $i$ works in 5 steps which are pipelined:

1. *Synthesis block #1:* The edge message information is uncompressed from the FIFO memory and sent to the subtractor.

2. *Generation of the $T_{n,m}^{(i)}$ values:* during the first $|\mathcal{N}(m)|$ clock cycles, the PCP associated to the parity check $cn_m$ received serially the $T_n^{(i)}$, $n \in \mathcal{N}(m)$, from an external memory. At cycle $n$, the $E_{n,m}^{(i-1)}$ of the previous decoding iteration are retrieved from an internal memory of the PCP and subtracted to $T_n^{(i)}$ in order to generate $T_{n,m}^{(i)}$ according to (1.40).

3. *Preprocessing:* The pre-processing block is depicted on the figure 4.8 (where $\lambda = 3$). It features a serial sorting of incoming $|T_{n,m}^{(i)}|$ value. Every clock cycle, the current value of $|T_{n,m}^{(i)}|$ is compared to the $\lambda$ previous lowest ones, which are stored with their index in a register file of size $\lambda$. According to the result of the comparators, the new value is inserted in the sorting order in the register and the highest value is forgotten. When all the $|\mathcal{N}(m)|$ input values have been preprocessed, 3 sets of data are output:

   - the set $\Omega_\lambda^{(i)} = \{|T_{n_0,m}^{(i)}|, \cdots, |T_{n_{\lambda-1},m}^{(i)}|\}$ which features the $\lambda$ lowest magnitude of $|T_{n,m}^{(i)}|$.

(c) Frédéric Guilloud, Télécom Paris - July 2004

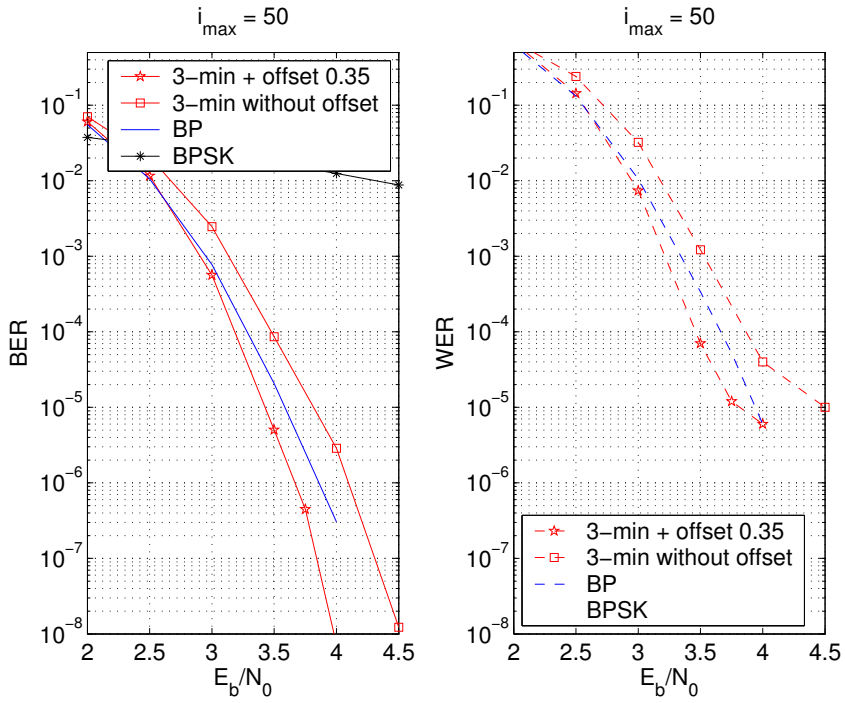



Figure 4.6: Code $\mathcal{C}_1$: comparison between BP algorithm and offset 3−min algorithm as a function of the $\frac{E_b}{N_0}$ for $i_{\max} = 50$. Note $5.10^5$ words have been generated for the last 2 points.

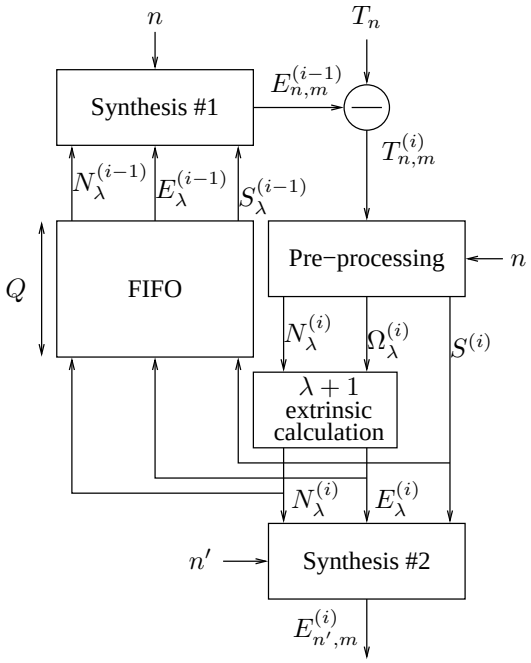



Figure 4.7: Description of a Parity Check Processor

(c) Frédéric Guilloud, Télécom Paris - July 2004

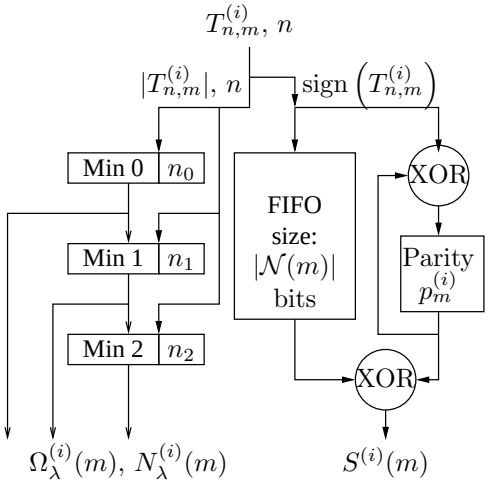



Figure 4.8: Pre-processing block (3−min algorithm)

- the set $N_\lambda^{(i)} = \{n_0, \cdots, n_{\lambda-1}\}$ which features the index $n$ associated to the set $\Omega_\lambda^{(i)}$.
- the set $S^{(i)}$ which features the signs of all the $E_{n,m}^{(i)}$. The incoming signs of $T_{n,m}^{(i)}$ have been saved in a FIFO memory and an XOR loop computed the parity $p_m^{(i)}$. Then the set $S^{(i)}$ is computed by an XOR operation between $p_m^{(i)}$ and the signs saved in the memory.

4. *Processing:* The processing can be designed in two different ways: the values can be either processed on-the-fly, or processed once and then a synthesis block generates the extrinsic information of each variable:

   - *On-the-fly extrinsic generation:* the extrinsic values are computed on the fly, one at each clock cycle, from the $|\mathcal{N}_\lambda(m)|$ values stored in the register file. Figure 4.9 depicts an example of a possible realization for $\lambda = 3$. When the current cycle corresponds to a value of $\mathcal{N}_\lambda(m)$, the corresponding boolean $t_i$ is set to select $+\infty$ in order to bypass the $\star$ operator. Meanwhile, the messages are compressed and sent to the FIFO memory to be saved for the next iteration.

   - *Pipelined extrinsic generation:* the extrinsic values are computed once with the $\star$−operator for example, as depicted on figure 4.10. Then they are saved and uncompressed by the synthesis block #2.

5. *FIFO saving:* a word of the FIFO saves:

   - the index of the $\lambda$ lowest extrinsic magnitude : $\mathcal{N}_\lambda^{(i)}(m)$
   - the set $E_\lambda^{(i)}(m)$ of the $\lambda + 1$ LLR computed

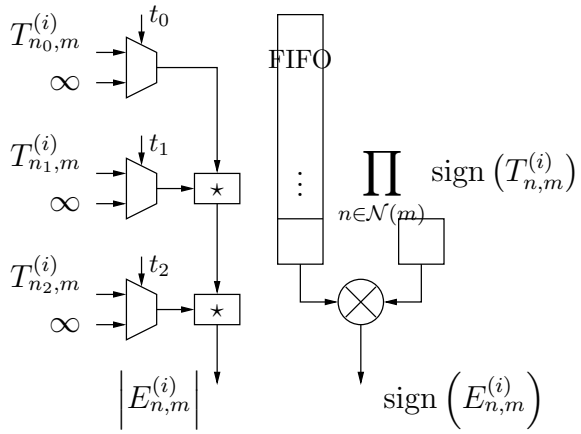



Figure 4.9: On the fly parity check scheme for $\lambda = 3$. When the reading phase is over, $\lambda$ values are saved and are addressed via $t_0, \cdots, t_{\lambda-1}$ by the writing phase for on-the-fly LLR computing.

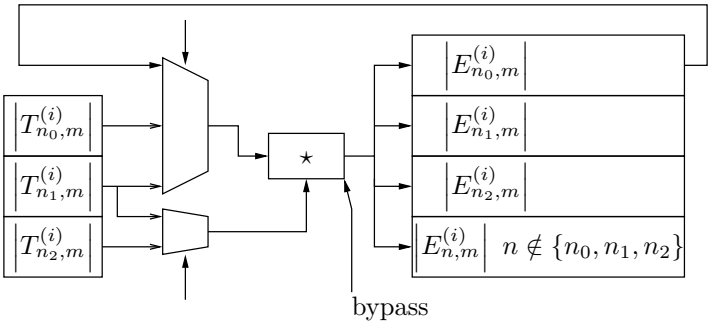



Figure 4.10: How to compute the $\lambda + 1$ different $\lambda-$min results using the $\star-$operator (example of $\lambda = 3$)

(c) Frédéric Guilloud, Télécom Paris - July 2004

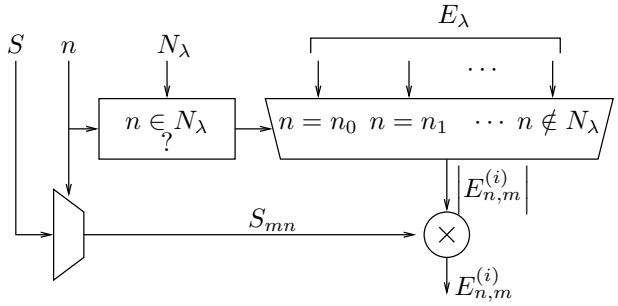



Figure 4.11: Synthesis block.

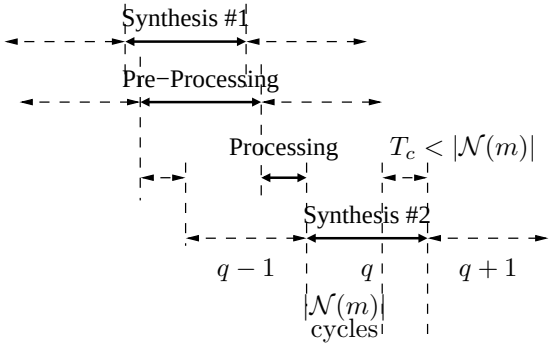



Figure 4.12: Order of operations inside a PCP

- the set $S^{(i)}(m)$ of the $|\mathcal{N}(m)|$ signs of the $M_{c\to v}$ messages

in the stack of size $Q$ so as to recover the $E^{(i)}_{n,m}$ in the iteration $i+1$.

Figure 4.11 depicts the synthesis blocks #1 and #2 of the figure 4.7 which uncompress the extrinsic information of all the bit nodes. The bit node index are compared to the index of the set $\mathcal{N}^{(i)}_\lambda(m)$. The result of the comparison enables to choose which value to be sent among the $\lambda+1$ values of the set $E^{(i)}_\lambda(m)$; It is then multiplied by the corresponding sign of the set $S^{(i)}(m)$.

### Flow of operations

The chronogram depicted in the figure 4.12 describes the flow of operations in a PCP during the processing of the $q^{\text{th}}$ *macrocycles*. Each operation last $|\mathcal{N}(m)|$ cycles, except for the calculation one which can last $T_c = \lambda + 1$ cycles. The non-blocking constraint for a pipe-lined dataflow is that the calculation time should not last more than $|\mathcal{N}(m)|$ cycles, *i.e.* the time needed by the other operations. As far as the irregular codes are considered, one just have to process the parity checks in the ascending order of their weight; otherwise, the $k-$th operation of the macrocycle $q$ might not be over when having to start the $k-$th operation of the macrocycle $q+1$.

(c) Frédéric Guilloud, Télécom Paris - July 2004

### 4.4.2 Memory saving

The λ−min algorithm aims at saving the memory required to store the edge messages. At the expense of a little performance degradation, we will see hereafter that the memory saved is important

Let $N_b + 1$ denotes the number of bit used to code the information ($N_b$ for magnitude and 1 bit for the sign). For the BP algorithm, we have to save $|\mathcal{N}(m)|$ messages $E_{n,m}^{(i)}$ which are coded on $N_b + 1$ bits. Hence, $(N_b + 1)|\mathcal{N}(m)|$ bits have to be saved for the next iteration for each parity check.

The simplification brought by the λ−min algorithm enables to reduce this amount of memory inside the PCP. The synthesis block of figure 4.7 is in charge of the decompression of this memory. The amount of memory needed for the λ−min algorithm for each parity check is detailed as follows:

- $\lambda + 1$ [1] different log-likelihood ratios to compute (magnitude only), *i.e.* $(\lambda + 1)N_b$ bits.
- λ addresses on λ elements of the set $\mathcal{N}_\lambda(m)$ *i.e.*: $\lambda \log_2(|\mathcal{N}_\lambda(m)|)$ bits
- $|\mathcal{N}(m)|$ signs and one parity $p_m^{(i)}$, *i.e.*: $|\mathcal{N}(m)| + 1$ bits.

The ratio between the two memory sizes saving the extrinsic information in the case of the λ−min algorithm and the BP algorithm for the parity $cn_m$ is:

$$\frac{(\lambda + 1)N_b + \lambda \log_2(|\mathcal{N}_\lambda(m)|) + |\mathcal{N}(m)| + 1}{(N_b + 1)|\mathcal{N}(m)|}. \tag{4.1}$$

Figure 4.13 depicts the shape of this ratio as a function of λ and $|\mathcal{N}(m)|$.

For example, in the case of a check $c_m$ with a weight $|\mathcal{N}(m)| = 20$ and for the 2−min algorithm, the memory needed to save the extrinsic memory is as much as 30% of the memory that would be needed with the BP algorithm.

## 4.5 Perspectives

Another algorithm called Approximate-min* or A-min* has been recently published by (Jones et al. 2003):

**Iterative Algorithm 9 (A-min*)** *The check node update rule is approximated for all the incoming messages except for the one with the smallest magnitude for which there is no approximation.*

[1] only if $\lambda > 2$ because if $\lambda = 2$ there is only one LLR to compute

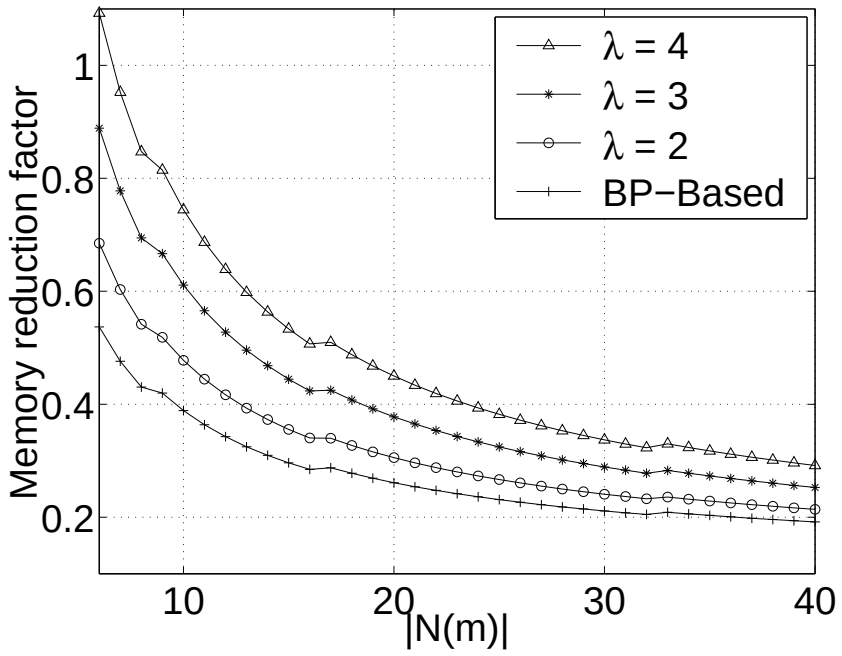



Figure 4.13: Proportion of the memory used to save extrinsic information in the $\lambda-$min algorithm, depending on the parity check degree, for several $\lambda$ value. $N_b = 5$ bits

| | |
|---|---|
| *Initialization:* | $E_{n,m}^{(0)} = 0$ |
| *Variable node update rule:* | $T_{n,m}^{(i)} = I_n + \sum_{m' \in \mathcal{M}(n) \backslash m} E_{n,m'}^{(i-1)}$ |
| *Check node update rule:* | *Let* $n_0 = \arg\min_{n \in \mathcal{N}(m)} \left\lvert \frac{T_{n',m}^{(i)}}{2} \right\rvert$ |
| | *if* $n = n_0$, |
| | $E_{n_0,m}^{(i)} = 2 \tanh^{-1} \prod_{n' \in \mathcal{N}(m) \backslash n_0} \tanh \frac{T_{n',m}^{(i)}}{2}$ |
| | *if* $n \neq n_0$, |
| | $E_{n,m}^{(i)} = 2 \tanh^{-1} \prod_{n' \in \mathcal{N}(m)} \tanh \frac{T_{n',m}^{(i)}}{2}$ |
| *Last Variable node update rule:* | $T_n = I_n + \sum_{m \in \mathcal{M}(n)} E_{n,m}^{(i-1)}$ |

The approximation is quite similar to the APP-check algorithm (iterative algorithm 5), but has remarkable performance. The same check node update rule is applied for all the variable but the one whose reliability is the lowest. For this one, the check node update rule of the BP algorithm is applied. This approximation is also quite similar to the $2-$min algorithm, with the difference that the LLR of all the variable is sent back as the default value, instead of the LLR of the $\lambda$ lowest reliable one.

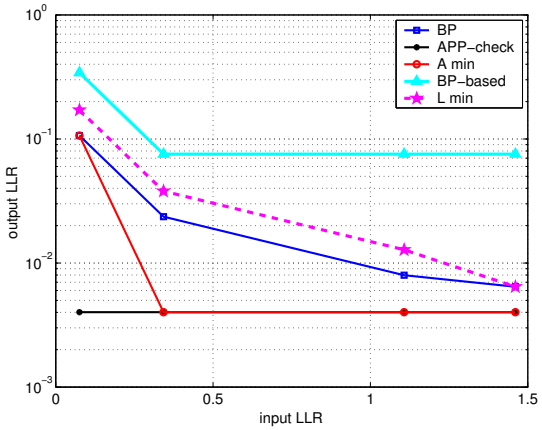



Figure 4.14: k=4

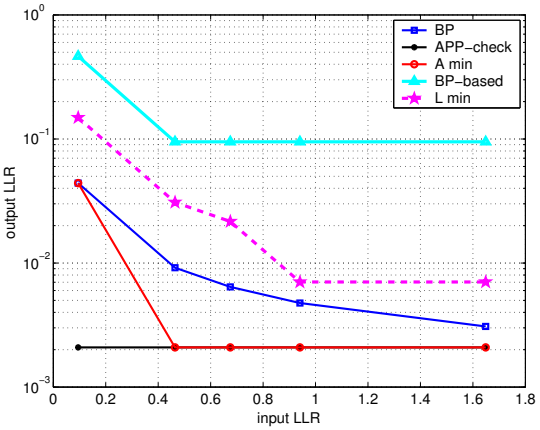



Figure 4.15: k=5

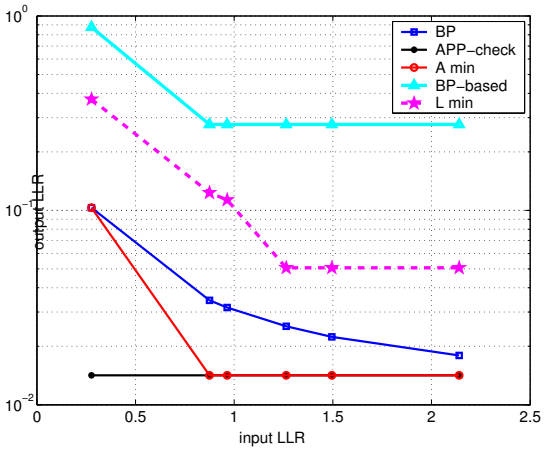



Figure 4.16: k=6

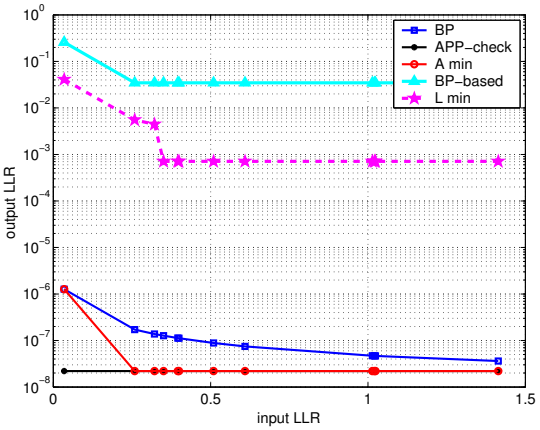



Figure 4.17: k=12

The figures 4.14 to 4.17 depict the value of $E_{n,m}^{(i)}$ versus the value of $T_{n,m}^{(i)}$ respectively for 4 different parity check degree. 4 check node update rules are compared:

- the BP algorithm,
- the APP-check algorithm,
- the $\lambda-$min algorithm,
- the BP-based algorithm.

The comparison is only made for some fixed random values, corresponding to an $\left(\frac{E_b}{N_0}\right)_{\text{dB}} = 0dB$. But it is quite representative of the behaviour of the different check node update rules. On these figures, we can clearly see that the BP-based and the $\lambda-$min algorithms

both over-estimate the extrinsic information whereas the A-min and the APP-check both under-estimate the extrinsic information.

In figure 4.14, the degree is only 4. The 3−min algorithm is then quite competitive since $\lambda$ is almost the same as the degree of the check constraint. But as the degree of the check node is increasing, the difference between the BP and the $\lambda-$Min becomes larger than the difference between the BP and the A-min* algorithm. The power of the A-min* algorithm is that the least reliable bit is perfectly estimated and the worst one almost as well. Note that this better approximation is made at the expense of an increased processing complexity.

## 4.6 Conclusion

In this chapter, a new sub-optimal algorithm, named $\lambda-$Min algorithm, for updating extrinsic information has been proposed. The $\lambda-$Min algorithm offers a performance-versus-complexity trade off between the BP algorithm and the BP-based algorithm: the complexity of the check node processor and the memory required to save the edge messages are reduced, at the expense of no significant performance loss. Moreover, the addition of an offset increases the convergence speed of the $\lambda-$min algorithm: for a given number of iterations, it can outperform the BP algorithm.

An original architecture implementing the $\lambda-$min algorithm in a serial mode has been also discussed. This hardware realization of the $\lambda-$min algorithm enables a reduction up to 75% of the extrinsic memory information, with high rate LDPC codes. This architecture can be very easily transposed to the A-min algorithm, which seems also to be very efficient.

The $\lambda-$min algorithm and its associated architecture have been published in two papers and are patent pending in the USA.

(c) Frédéric Guilloud, Télécom Paris - July 2004

This page intentionally left blank.

# Chapter 5

# Generic Implementation of an LDPC Decoder

**Summary:**

*Simulations for studying the behaviour of LDPC codes under iterative decoding algorithms are time consuming. The time needed for simulating is multiplied by* 10 *when the target BER is a decade lower.*

*In this chapter, a generic implementation of an LDPC code decoder is presented. The aim of this implementation is to run on a FPGA evaluation board so as to simulate various type of LDPC codes faster than it would be on a personal computer. This generic decoder should also be used to compare different architecture implementations of some specific blocks. This generic decoder should be used as a performance meter tool for LDPC codes and for their implementation. So the genericity is an important feature of this implementation and a real challenge for a LDPC code decoder design. The VHDL language has been chosen to describe the architecture.*

*After an overview of the decoder architecture, the management of the different memories is detailed. Then the shuffling problem is discussed and the top-level architecture is described. A more detailed description of the main component of the architecture is given in annexe C. Finally, the genericity of the architecture is discussed.*

## 5.1 Overview

### 5.1.1 About genericity

The main specification of the LDPC decoder that will be described in this chapter is the *genericity*. The meaning of genericity is double:


(c) Frédéric Guilloud, Télécom Paris - July 2004

- The decoder should be generic in the sense that it should be able to decode any LDPC code, providing that they have the same size as the one fixed by the parameters of the architecture. It means that any distribution degree of the variable and check nodes should be allowed, given $N, M$ and $P$.

- The decoder should also be generic in the sense that a lot of parameters and components could be modified: for example the parity-check matrix size, the decoding algorithm (BP, $\lambda-$min, BP-based), the dynamic range and the number of bits used for the fixed-point coding. But the modifications of these parameters should require a new hardware synthesis.

Both of these goals are very challenging since LDPC decoders have been so far always designed for a particular class of parity-check matrix. Moreover, genericity for architecture description increase the complexity level.

Our motivation to design such a decoder is the possibility to run simulations much faster on a FPGA than on a PC. Simulations give, at the end, the final judgement about the comparison of the codes and of their associated architecture.

### 5.1.2 Synoptic

The figure 5.1 depicts the synoptic of the decoder for the flooding scheduling (see section 3.4). There are 2 main phases: the input/output one and the decoding one. The input of a new word to decode and the output of the word that has been decoded are done simultaneously. At the end of each decoding phase, the memories related to the variable information are swapped. The information propagation is the heart of the decoding process. There are 3 phases which are pipelined inside an iteration:

- Information reading (variable node processing)

- Information processing (check node processing)

- Information writing (accumulation, variable node processing)

### 5.1.3 Architecture

A first answer to genericity is to describe a mixed architecture, as detailed in section 3.1. Using the notations of the generalized architecture described in section 3.1, the parameters of the architecture are $\alpha = k$, $\beta = j$, and thus $R_e = P \times k/\alpha = P$. This yields to a serial implementation of the node processors. The serial processing of the node processors relaxes the constraints on the memory organization. The $\lambda-$min algorithm will be implemented to decrease the memory requirements. The flooding scheduling has been chosen for the iterative decoding because we were not aware of the other schedules when we started the design.

(c) Frédéric Guilloud, Télécom Paris - July 2004

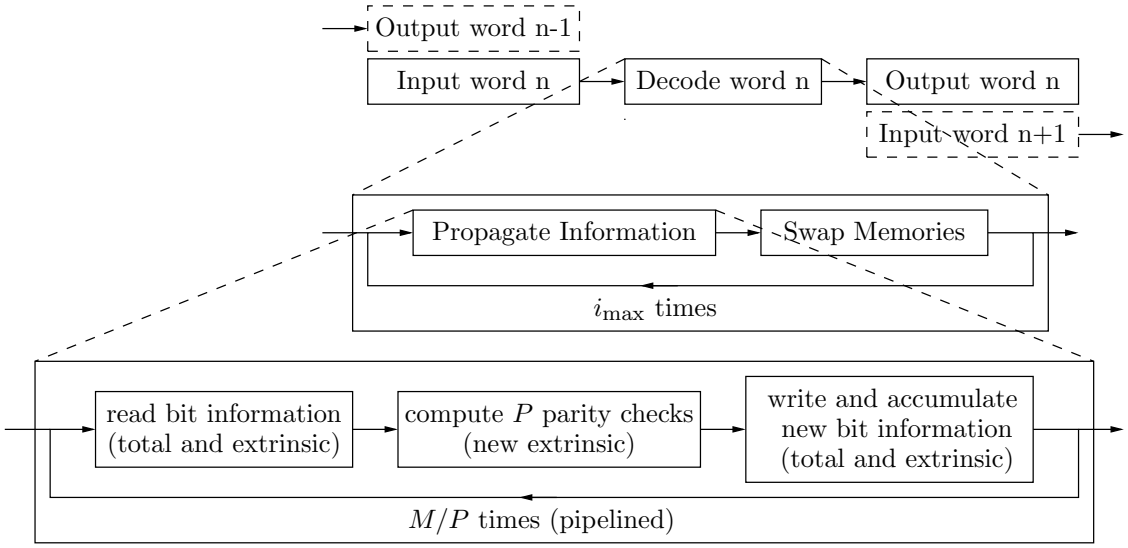



Figure 5.1: Synoptic of the decoder associated to the flooding scheduling.

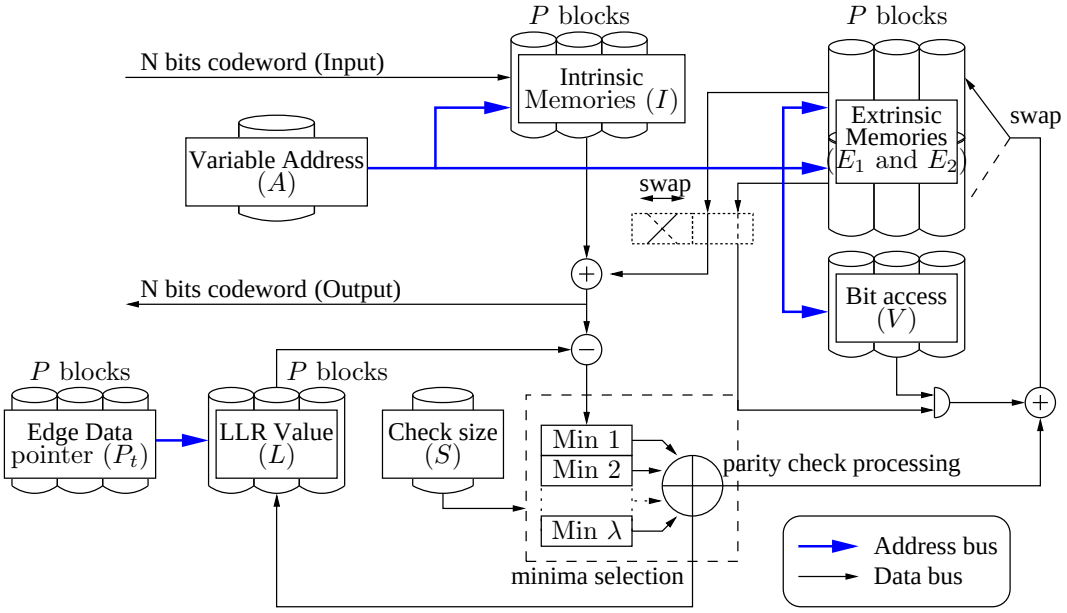



Figure 5.2: System architecture of the decoder: memories are split into $P$ blocks (in this example, $P = 3$).

(c) Frédéric Guilloud, Télécom Paris - July 2004

The figure 5.2 depicts the system-oriented architecture of the decoder with $P = 3$, where 8 different memory blocks are to be defined. These memories can be divided into 3 categories, according to the type of information which is saved in. The information may be related to:

1. **the parity-check matrix** of the LDPC code: this information is used to control the decoder.
   - $A$: memory for variable addresses; $\Gamma/P$ words are saved in the A-memory. Each word is split up into $P+1$ parts: $P$ address for the bits (timing shuffling) and 1 address for the shuffle network (spatial shuffling).
   - $S$: memory for parity check size; this is where the weight of each parity check is saved. This enables to process irregular parity-check matrices.
2. **the variable node processor:** all these memories are split into $P$ blocks, so that it is possible to access $P$ data at the same time:
   - $I$: memory for intrinsic values.
   - $E_1$.`sel` $= 1 | E_2$.`sel` $= 0$: memory for actual extrinsic values that are being accumulated.
   - $E_1$.`sel` $= 0 | E_2$.`sel` $= 1$: memory for former extrinsic values that are being used to process parity check calculation.
   - $V$: memory for the first accumulation flag; one bit for each variable is used in $V$ to remember whether a variable have already been involved in a parity check or not, during each iteration. This enable the first accumulation to be made with an all zero word.

   To be more efficient, the two extrinsic memories are swapped after each iteration. So the actual extrinsic information becomes the former one during the next iteration more easily. This is controlled by the `sel` signal.
3. **the check nodes processor:** all these memories are also split into $P$ block.
   - $P_t$: memory for extrinsic pointer: this is where an address of the L-memory is saved for each bit. A memory word is made of the address and the sign of the bit.
   - $L$: memory for computed LLR values; this is where the pointer of the extrinsic values points.

The memory management will be described in the following example, illustrated by a practical example, and particularly: how the memories are accessed ? How they are organized ?

(c) Frédéric Guilloud, Télécom Paris - July 2004

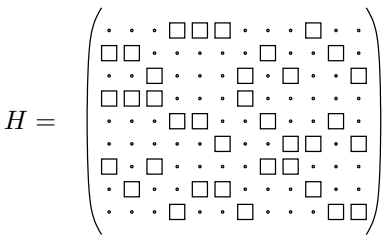


Figure 5.3: Didactic example for a parity check matrix

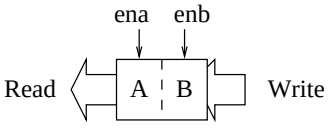



Figure 5.4: Dual-Port Block SelectRAM

## 5.2 Memory Management

As it has been introduced in section 5.1.3 that a lot of different memories is used in the decoder. In this section, a didactic example will explained step by step how the main memories are used.

### 5.2.1 Preliminaries

#### A practical example

The parity check matrix of figure 5.3 will be taken as an example. It is a $(3, 4)-$LDPC code of length $N = 12$. It means that $\alpha = k = 4$. The edge rate $R_e = P$ is set to $P = 3$ for the example. The parity checks are associated with one of the $P$ parity-check processor available. A whole decoding iteration is processed in $Q = M/P = 3$ phases ($q = 0$, $q = 1$, $q = 2$).

#### Memories of the FPGA

Although the design is intended to be generic, it is necessary to know the target implementation, particularly for FPGA implementations, since the resources (memories, ports, clock frequency) can be different. Since memories are a crucial point in the decoder design, it is necessary to take into account their specifications. The FPGA Xilinx Virtex XCV1000-E for example features 96 blocks of True Dual Port RAM. Each RAM block has 4096 bits. All the memories will be assumed to be true dual port memories hereafter, since nowadays almost all the FPGA feature such memories. Figure 5.4 depicts the symbol of a dual RAM block used in this manuscript. By convention, the "A" side will be used for reading and the "B" side for writing. Such dual port RAM can be inferred

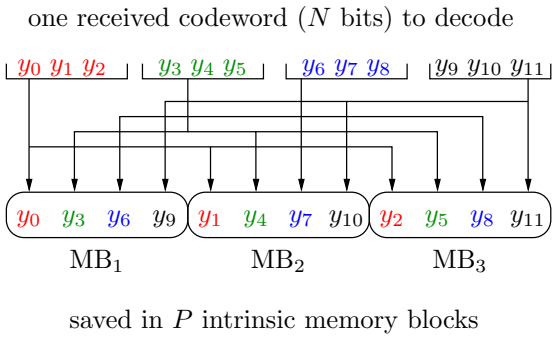



Figure 5.5: Variable Memory Filling

by the synthesis tool. Only the $A$-memories containing the variable address will be used for writing one side "A" and for reading on both sides. Note that despite the specific kind of memory used here, the genericity is kept by the VHDL configurations description: the memory description is embodied in a generic component which is target independent. For other targets, only this elementary RAM block inside the generic memory component would have to be redescribed. Thus we have been able to take advantage of the inference capabilities of the synthesis tool (Leonardo Spectrum).

### 5.2.2 Variable node memories

The variable memories $I$, $E_1$, $E_2$ and $V$ are the memories which are directly linked to the variable processors. The variable or bit information is the intrinsic and the extrinsic information. They are saved in the $I$, $E_1$ and $E_2-$memories. The memory $V$ is the first accumulation flag memory and it will be described at the end of this section.

#### Organization

The variable information are saved in memory following the time of arrival order of the channel output: $P$ variables are waited for and then saved in the $P$ different memory blocks. The way these memories are filled is described in figure 5.5. This method enables to speed up the throughput since $P$ data can written at the same time. The parity-check matrix can be represented using the variable intrinsic memory order instead of the natural order. This yields to the parity-check matrix of figure 5.6 which is the same as figure 3.8 used in section 3.4.

#### Dataflow

The dataflow of information during the different decoding phases is illustrated on figure 5.8.

The first step before decoding is to save the $N$ variables of the received codeword (`STATE` set to `IO`). They are saved in the intrinsic memory $I$. Meanwhile, one of the two

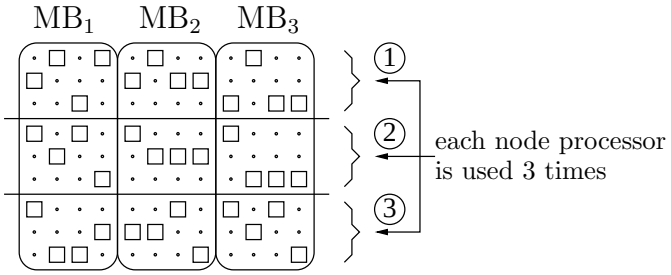



Figure 5.6: Didactic example for a parity check matrix after reordering as in the memory banks

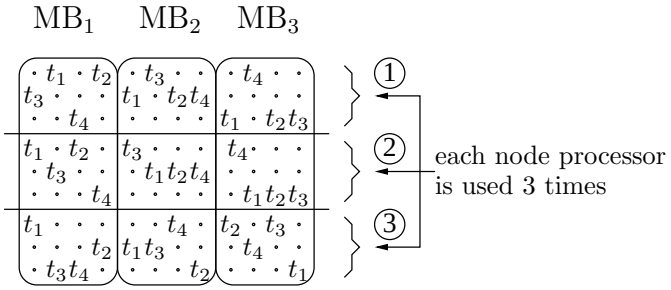



Figure 5.7: Parity check matrix : reordering and addressing into $M/P = 3$ passes and $P = 3$ memory blocks. The non-zero entries of the matrix are replaced by $t_i$, which are the times when the bit are accessed: $t_1 < t_2 < \cdots < t_9$.

extrinsic memory $E_1$ or $E_2$, depending on the `sel` signal value, is initialized to zero. Note also that the last decoded codeword is output in parallel.

As depicted in figure 5.1, the decoding phase consists of the reading and the accumulation of the messages and also of the swap phase. The `sel` signal is complemented during the `SWAP` state. This signal is used to control and exchange the role of the 2 extrinsic memories $E_1$ and $E_2$. The intrinsic memory and the extrinsic memory of the previous iteration are read and their output are summed. Then, the edge message is subtracted as described in the parity-check processor, section 4.4.1. Finally, the output of the parity-check processor is accumulated in the secondary extrinsic memory. The role of the 2 extrinsic memory is then swapped for the next iteration.

The `init_select` signal is used to control whether the accumulation has to be done or not. If not, the accumulation is processed with the all-zero word. It is used to avoid the initialization of the memory before the first accumulations. This information related to each bit is saved in the first accumulation flag memory $V$. The generation of the `init_select` is depicted on figure 5.9. $N/P$ words of one bit are saved in the memory $V$. This memory is depicted as a component with 2 sides `A` and `B`. For each side, the input ports are the data input (`.in`) and output (`.out`), the address (`a` or `b`) and the write enable signal (`.en`). The enable port are also depicted and always set to '1'. Let assume

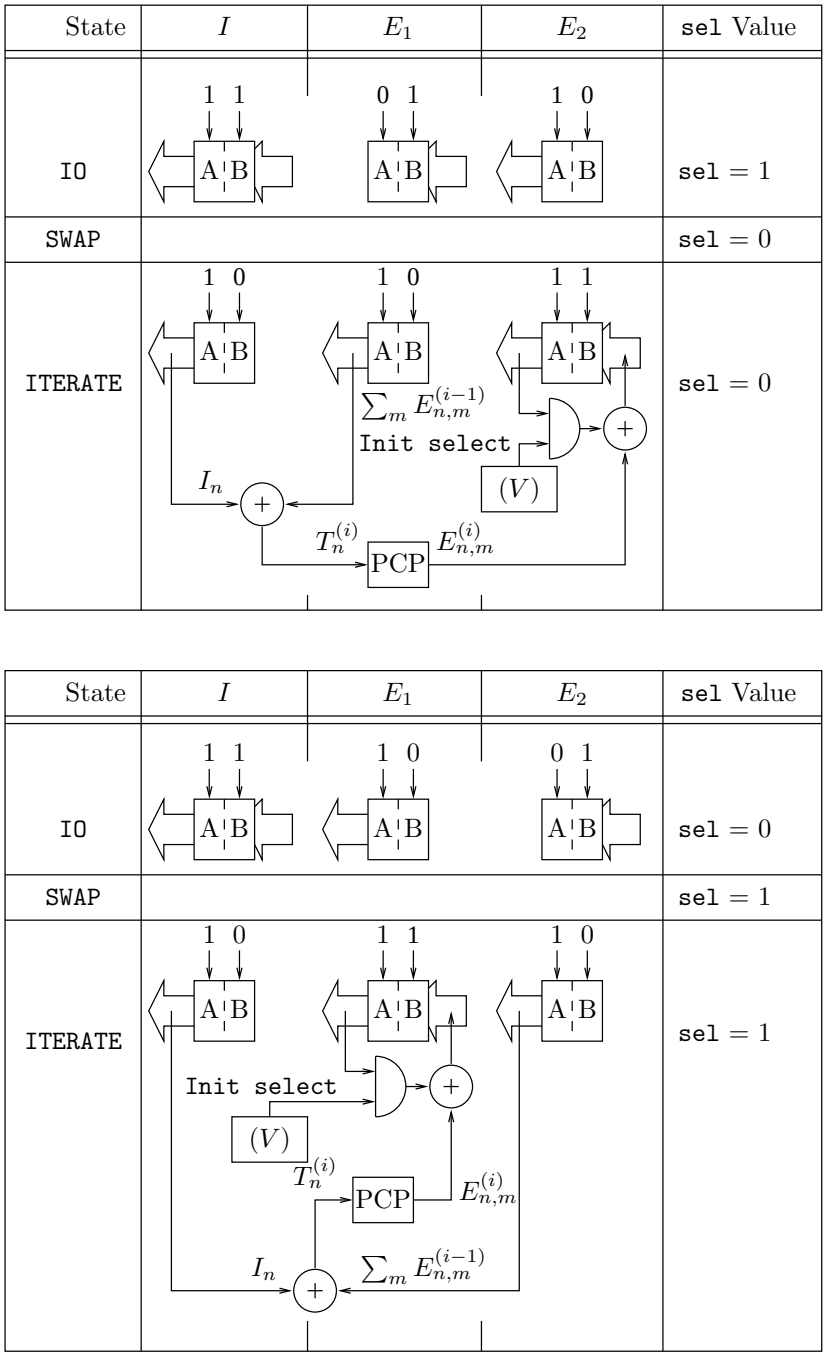



Figure 5.8: Dataflow of variable information depending on the state and the `sel` value changed in the `SWAP` state.

(c) Frédéric Guilloud, Télécom Paris - July 2004

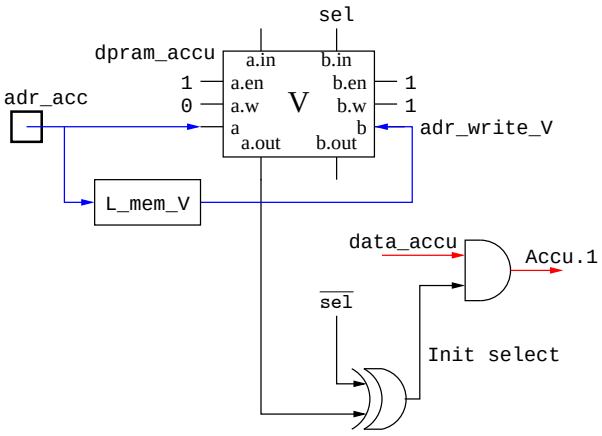



Figure 5.9: The principle of the first accumulation bit access.

that all the memory has been filled with '1', and that the `sel` signal is set to the opposite value '0'. When the address of the variable $vn_n$ is presented on the `adr_acc` wire for the first time in the iteration, the output `a.out` is '1'. After the latency of the memory denoted `L_mem_V`, the value '0' is written at the same address. The `init_select` is equal to '1' `xor` '1' which is '0'. So the first time the address of the variable $vn_n$ is presented, the `init_select` signal forces to zero the previous accumulated value `data_accu`. The next time the same address is presented inside the same iteration, the output of the memory is then '0' and thus the `init_select` signal is set to '1'. Note that at the end of an iteration, all the memory $V$ is filled with '0' values. But as the `sel` signal is complemented in the `SWAP` state, the same principle applies with the complemented values.

### 5.2.3 Check node memories

The check node memories are the $L$ and $P_t$ memories. In section 4.4, the architecture of the $\lambda-$min algorithm has been discussed. In figure 4.7, an architecture of the parity check processor is depicted where a FIFO memory is used to save at iteration $(i)$:

- the set of the $\lambda+1$ results of the $\lambda-$min algorithm is denoted by

$$E_\lambda^{(i)} = \{e_{n_0}, \cdots, e_{n_{\lambda-1}}, e_\lambda\}.$$

  It is saved in the memory $L$. $e_{n_j}$ is the magnitude of the extrinsic information which is sent to the variable which sent the $j$-th minimum. $e_\lambda$ is the default one, which is sent to all the variable which did not sent a message among the minimum set.

- the signs of the extrinsic information $S_{n,m}^{(i)}$, and the set $N_\lambda^{(i)} = n_0, \cdots, n_{\lambda-1}$ of the index of the $\lambda$ least reliable variable of the parity check are saved together in the memory $P_t$. In fact, a convention enables to use only one bit to save the set $N_\lambda^(i)$, as described hereafter.

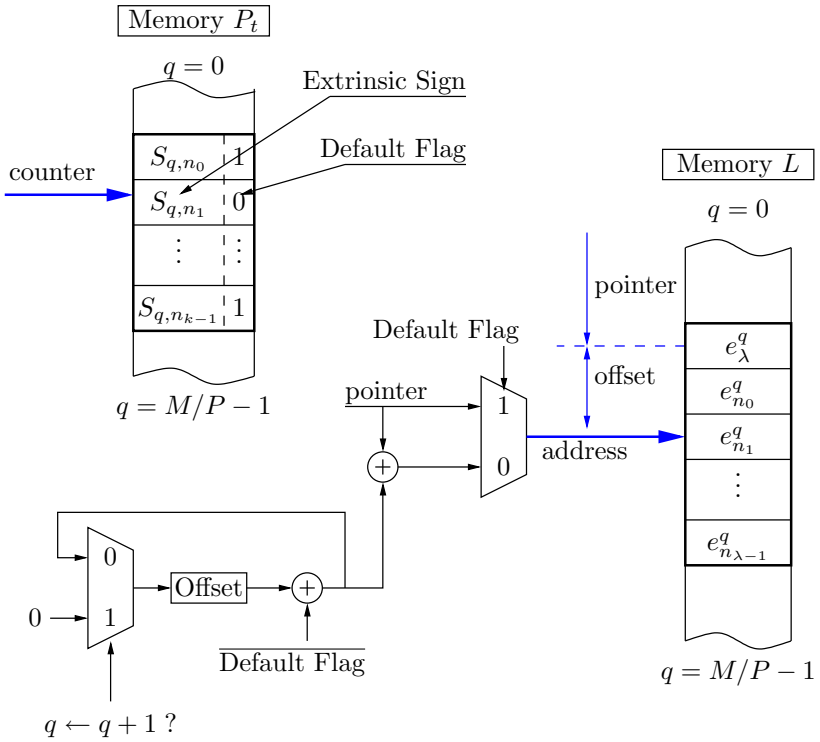



Figure 5.10: Organization of the check node processor memories $L$ and $P_t$.

The memory $L$ is not exactly used as a FIFO memory, as described in the following.

The memories $L$ and $P_t$ are divided into $P$ independent blocks. Each block is linked to $M/P$ different parity-check processors and is organized as follows:

- the memory $L$ contains $M/P \times \lambda$ words of $Nb + 1$ bits. It is addressed by a pointer and an offset which can be added or not to the pointer, as depicted on figure 5.10. The default LLR $e_{lambda}$ is saved on the first address and then the other $e_i$ are saved on the next addresses following the time scheduling addresses (more details about this scheduling will be given through section 5.2.4). The pointer points the default value $e_\lambda$ and the offset is incremented when the default value is not chosen by the synthesis block.

- The memory $P_t$ contains $M/P \times k$ words for a regular $(j, k)-$LDPC code. Each word is 2 bits long: one bit is the sign of the extrinsic value, and the other bits is the default flag. It indicates whether the variable belongs to the set of minimum (flag set to '0') or not (flag set to '1'). If not, the pointer to the $L$ memory is used without offset. On the contrary, if the flag is set to '0', the offset is incremented and added to the pointer so as to read the next $e_i$ value in the $L$ memory (see figure 5.10).

So during an iteration, the memory $P_t$ is addressed by a counter ranging from 0 to $kM/P - 1$. The output of this memory is split into 2 parts. The first part is the sign of the extrinsic information $E^{(i)}_{n,m}$. It is multiplied by the output of the memory $L$. This memory is addressed either by the value of the current pointer signal or by the value of the sum between the pointer and the offset, depending on the second part of the output of the memory $P_t$ (Default Flag). Each the default Flag is zero, the offset is incremented.

Note that when a parity-check has been processed, the value of $q$ is incremented and the pointer is increased of the value $\lambda + 1$. The degree of the parity check is saved in the memory $S$. Although it is used in the check node processing, it is described in section 5.2.4 because the parity-check matrix is memorized in these two memory blocks.

### 5.2.4 The memories A and S

The memory $A$ is the variable address memory, and the memory $S$ is the parity-check size memory. The memory $S$ contains the degrees of the different parity checks. The memory $A$ contains the non-zero entries of the parity-check matrix. In fact, the data which are saved in $A$ are not simply the index of the non-zero entries of $H$. They are the addresses of the intrinsic and extrinsic memory (time shuffling), and also the addresses of the shuffle network (spatial shuffling). A pre-processing on the parity-check matrix is required to somehow *code* $H$ in the memory A and the memory S. We will assume without any loss of generality that the rows of the parity check matrix are sorted in the ascending way of their degree, as specified by the scheduling of the parity check processor in section 4.4.1. We will also assume in a first time that the parity-check degree distribution has a granularity of $P$. It means that the number of rows with the same weight should be a multiple of $P$. Note that the end of this section will be devoted to the analysis of such constraints: we will show that the decoder is still generic.

So the memory $S$ contains $M/P$ words. The $q-$th word is the degree of the rows $(q-1)P$ to $qP - 1$, which will be denoted $k_q$, $q \in \{0, \cdots, M/P - 1\}$ hereafter.

#### Scheduling access to the edges

For the time shuffling address construction, the following rule has to be respected: as the data are sent serially to the parity check processor: it means that the $k$ bits will be read in $k$ clock cycles in the memories. So at each clock cycle, $P$ variables implied in the $P$ different parity check processor have to be read. These $P$ variables should also be saved in the $P$ different memory banks, since only one read access can be made.

Based on figure 5.6 where the parity-check matrix is written by grouping with each other the variables which belong to the same memory bank, figure 5.7 depicts the time scheduling for reading or writing the information saved in the memory banks. The time $t_i$ is the time when the corresponding variable is accessed. It is written in the place of the non-zero entries of the parity-check matrix, $t_1 < t_2 < t_3 < t_4$. Respecting the rule above means that:

(c) Frédéric Guilloud, Télécom Paris - July 2004

Table 5.1: Listing of the shuffle network addresses for $P = 3$. The order of the input list is changed according to the address of the shuffle.

| input | output | address |
|---|---|---|
| a b c | a c b | 0 |
| | b c a | 1 |
| | c a b | 2 |
| | c b a | 3 |
| | a b c | 4 |
| | b a c | 5 |

1. one can not read 2 bits in the same area (same block) at the same time $t_i$.
2. one can not read 2 bits on the same parity check (same line) at the same time $t_i$.

Then the variable data have to be routed to the ad-hoc parity-check processor. It is implemented by a shuffle network. This shuffle network is a $P \rightarrow P$ MUX which is also addressed to get into the right configuration. More details about the shuffle network are given in section 5.3.

### Organizing $H$ in the memory A

A word of the variable address memory is the concatenation of the $P$ addresses to read or write the variable memory and of the address of the shuffle network. This memory has a total of $\sum_{q=0}^{M/P-1} k_q = M/P \times k$ words for regular $(j,k)-$LDPC codes. The number of bit for each word is the sum of:

- $P \times \log_2(N/P)$ for the memory address part (time shuffling),
- the number of bit dedicated to the address of the shuffle network (space shuffling). If for example the $P!$ permutations are to be possible then $\log_2(P!)$ bits are required. Note that when $P$ is high, coding all the $P!$ permutations is too complex. Then, a restricted number of possible permutations should be implemented.

As an example, the table 5.2 resumes the construction of the data saved in the memory 1 for decoding the parity check matrix of figure 3.8. It is supposed that all the $3! = 6$ permutations are implemented, following the address rule of table 5.1. So $\lceil \log_2(P!) \rceil = 3$ bits are needed to code this address.

### Using the A-memory

Figure 5.11 depicts how the information saved in the A-memory is used during the decoding phases. The two-ports memory is filled with the $kM/P = 12$ words which have

Table 5.2: Encoding the parity-check matrix in the A-memory

| Phase | Time | Label | Number | word value | |
|---|---|---|---|---|---|
| | | | | Integers | Hexa |
| q0 | t1 | bit value | 3 1 2 | 1 0 0 | |
| | | shuffle | 1 2 3 | 4 | |
| | | word value | | | 101h |
| | t2 | bit value | 9 7 8 | 3 2 2 | |
| | | shuffle | 1 2 3 | 4 | |
| | | word value | | | 13Ah |
| | t3 | bit value | 0 4 11 | 0 1 3 | |
| | | shuffle | 2 1 3 | 5 | |
| | | word value | | | 147h |
| | t4 | bit value | 6 10 5 | 2 3 1 | |
| | | shuffle | 3 2 1 | 3 | |
| | | word value | | | 0EDh |
| q1 | t1 | bit value | 0 4 5 | 0 1 1 | |
| | | shuffle | 1 2 3 | 4 | |
| | | word value | | | 105h |
| | t2 | bit value | 6 7 8 | 2 2 2 | |
| | | shuffle | 1 2 3 | 4 | |
| | | word value | | | 12Ah |
| | t3 | bit value | 3 1 11 | 1 0 3 | |
| | | shuffle | 2 1 3 | 5 | |
| | | word value | | | 153h |
| | t4 | bit value | 9 10 2 | 3 3 0 | |
| | | shuffle | 3 2 1 | 3 | |
| | | word value | | | 0FCh |
| q2 | t1 | bit value | 0 1 11 | 0 0 3 | |
| | | shuffle | 1 2 3 | 4 | |
| | | word value | | | 103h |
| | t2 | bit value | 9 10 2 | 3 3 0 | |
| | | shuffle | 2 3 1 | 1 | |
| | | word value | | | 07Ch |
| | t3 | bit value | 3 4 8 | 1 1 2 | |
| | | shuffle | 3 2 1 | 3 | |
| | | word value | | | 0D6h |
| | t4 | bit value | 6 7 5 | 2 2 1 | |
| | | shuffle | 3 1 2 | 2 | |
| | | word value | | | 0A9h |

(c) Frédéric Guilloud, Télécom Paris - July 2004

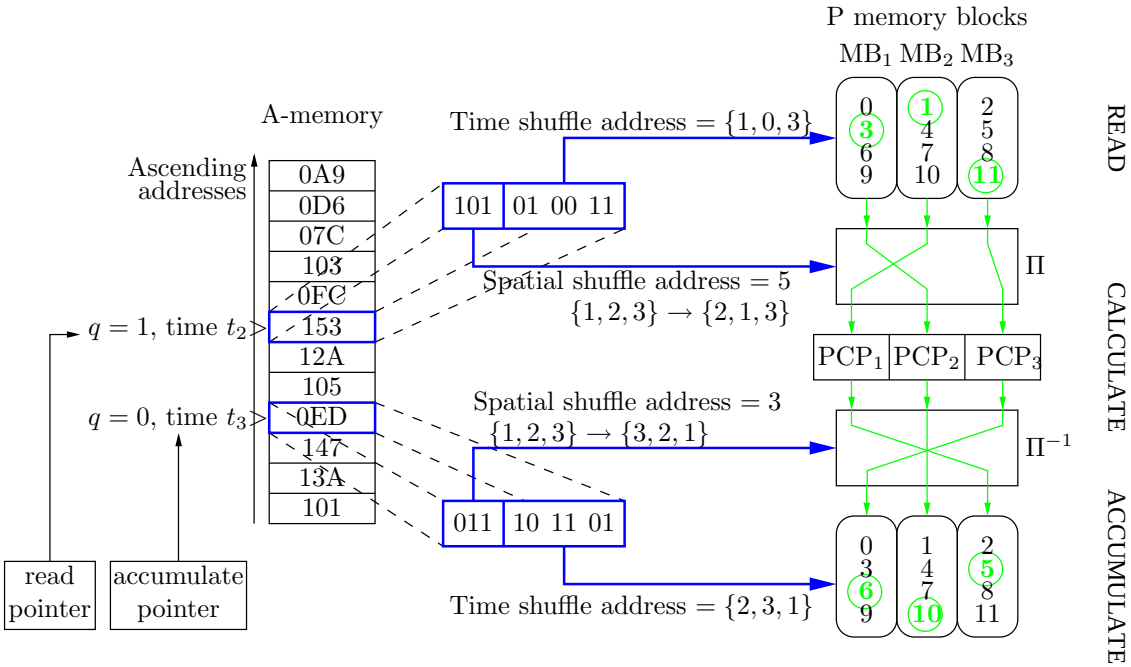



Figure 5.11: The information saved in the A-memory are used for addressing the memory banks and the shuffle network, both during the reading phase and during the accumulation phase.

(c) Frédéric Guilloud, Télécom Paris - July 2004

Table 5.3: Memory size requirements for a regular $(j,k)-$LDPC code of length $N$. The architecture is mixed with parameters $\alpha = k$, $\beta = j$, $P = R_e$.

| Memory | Nb of blocks | Block Size (words) | Word size (bits) | Nb of FPGA Ram Block (Xilinx VirtexE-1000) |
|---|---|---|---|---|
| $I$, $E_1$, $E_2$ | $3P$ | $N/P$ | $N_b + 1$ | $3P \times \left\lceil \frac{N/P \times (N_b+1)}{4096} \right\rceil$ |
| $A$ | 1 | $M.k/P$ | $P.\lceil \log_2(N/P) \rceil + \lceil P\log_2(P) - P + 1 \rceil$ (*) | $\frac{Mk}{4096P}\,(P.\lceil \log_2(N/P) \rceil + \lceil P\log_2(P) - P + 1 \rceil)$ (*) |
| $P_t$ | $P$ | $M.k/P$ | 2 | $P \times \left\lceil \frac{M.k/P \times 2}{4096} \right\rceil$ |
| $L$ | $P$ | $M(\lambda+1)/P$ | $N_b + 1$ | $P \times \left\lceil \frac{M(\lambda+1)/P \times (N_b+1)}{4096} \right\rceil$ |
| $V$ | $P$ | $N/P$ | 1 | $P \times \frac{N/P \times 1}{4096}$ |
| $S$ | 1 | $M/P$ | $\log_2(k_{\max})$ | $\left\lceil \frac{M/P \times \log_2(k\max)}{4096} \right\rceil$ |

(*) see section 5.3 for more information on the shuffle address.

been pre-processed in table 5.2. The dual port RAM block enables this memory to be addressed twice at the same time for reading. In this example, 2 words are read: one address for the reading of phase $q = 1$ and one address for the accumulation of phase $q = 0$.

### 5.2.5 Migration to FPGA

#### Using the FPGA memories

Table 5.3 describes the memory size needed for all the different memory types for a regular $(j,k)-$LDPC code of length $N$ decoder, implementing a mixed architecture with parameters $\alpha = k$, $\beta = j$, $P = R_e$. Note that for irregular matrices, $kM/P$ should be replaced by $\sum_{q=0}^{M/P-1} k_q$. The first column indicates the number of block required by the edge rate $R_e = P$ of the decoder. The second column indicates the number of words needed in one such block, and the third one, the size of this word. Of course, one block might be instanced with more than one FPGA dual block RAM, depending on the required size of one word and on the required number of words. The optimized number of FPGA RAM block required is calculated in the last column of table 5.3 for a Xilinx VirtexE-1000 FPGA, whose RAM blocks have 4096 bits. Optimized means that this number is the lowest, since it does not take into account how the block should be wired to achieve such a number. It is equal to:

$$\text{Nb of Blocks} \times \left\lceil \frac{\text{Block size} \times \text{Word size (bits)}}{4096 bits} \right\rceil \tag{5.1}$$

A numerical application is illustrated in table 5.4. For several FPGA (2 Xilinx Virtex

Table 5.4: FPGA RAM Block usage for several FPGA.

| FPGA | Model | Xilinx | | Altera | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Virtex E 1000 | Virtex II 8000 | Stratix EP1S25 | | | | | | Stratix EP1S80 | | | | | |
| | RAM ressource | 96 x 4k bits | 168 x 18k bits | 224 x 512 bits (a) | | | | | | 767 x 512 bits (a) | | | | | |
| | | | | 138 x 4k bits (b) | | | | | | 364 x 4k bits (b) | | | | | |
| | | | | 2 x 512k bits (c) | | | | | | 9 x 512k bits (c) | | | | | |
| | | | | (a) | (b) | (c) | (a) | (b) | (c) | (a) | (b) | (c) | (a) | (b) | (c) |
| | Examples | 1 | 2 | 3 | 4 | 5 | | | 6 | | | 7 | | | 8 | | |
| DECODER | N | 4096 | 4080 | 18560 | 32800 | 12280 | | | 16000 | | | 36840 | | | 46080 | | |
| | M | 2048 | 816 | 9280 | 6560 | 6140 | | | 3200 | | | 18420 | | | 9216 | | |
| | code rate | 0.5 | 0.8 | 0.5 | 0.8 | 0.5 | | | 0.8 | | | 0.5 | | | 0.8 | | |
| | P | 8 | 8 | 8 | 8 | 8 | | | 8 | | | 8 | | | 8 | | |
| | Nb+1 | 8 | 8 | 8 | 8 | 8 | | | 8 | | | 8 | | | 8 | | |
| | k | 6 | 15 | 6 | 15 | 6 | | | 15 | | | 6 | | | 15 | | |
| | lambda | 3 | 3 | 3 | 3 | 3 | | | 3 | | | 3 | | | 3 | | |
| I, Ei | parallelism (3P) | 24 | 24 | 24 | 24 | 24 | | | 24 | | | 24 | | | 24 | | |
| | Block size (word) | 512 | 510 | 2320 | 4100 | 1535 | | | 2000 | | | 4605 | | | 5760 | | |
| | Word size (bit) | 8 | 8 | 8 | 8 | 8 | | | 8 | | | 8 | | | 8 | | |
| | RAM block usage | 24 | 24 | 48 | 48 | 0 | 72 | 0 | 0 | 96 | 0 | 0 | 216 | 0 | 0 | 288 | 0 |
| V | parallelism (P) | 8 | 8 | 8 | 8 | 8 | | | 8 | | | 8 | | | 8 | | |
| | Block size (word) | 512 | 510 | 2320 | 4100 | 1535 | | | 2000 | | | 4605 | | | 5760 | | |
| | Word size (bit) | 1 | 1 | 1 | 1 | 1 | | | 1 | | | 1 | | | 1 | | |
| | RAM block usage | 8 | 8 | 8 | 8 | 24 | 0 | 0 | 32 | 0 | 0 | 72 | 0 | 0 | 96 | 0 | 0 |
| Pt | parallelism (P) | 8 | 8 | 8 | 8 | 8 | | | 8 | | | 8 | | | 8 | | |
| | Block size (word) | 1536 | 1530 | 6960 | 12300 | 4605 | | | 6000 | | | 13815 | | | 17280 | | |
| | Word size (bit) | 2 | 2 | 2 | 2 | 2 | | | 2 | | | 2 | | | 2 | | |
| | RAM block usage | 8 | 8 | 8 | 16 | 144 | 0 | 0 | 192 | 0 | 0 | 432 | 0 | 0 | 544 | 0 | 0 |
| L | parallelism (P) | 8 | 8 | 8 | 8 | 8 | | | 8 | | | 8 | | | 8 | | |
| | Block size (word) | 1024 | 408 | 4640 | 3280 | 3070 | | | 1600 | | | 9210 | | | 4608 | | |
| | Word size (bit) | 8 | 8 | 8 | 8 | 8 | | | 8 | | | 8 | | | 8 | | |
| | RAM block usage | 16 | 8 | 24 | 16 | 0 | 48 | 0 | 0 | 32 | 0 | 0 | 144 | 0 | 0 | 72 | 0 |
| A | parallelism (1) | 1 | 1 | 1 | 1 | 1 | | | 1 | | | 1 | | | 1 | | |
| | Block size (word) | 1536 | 1530 | 6960 | 12300 | 4605 | | | 6000 | | | 13815 | | | 17280 | | |
| | Word size (bit) | 89 | 89 | 113 | 121 | 105 | | | 105 | | | 121 | | | 121 | | |
| | RAM block usage | 34 | 34 | 43 | 81 | 0 | 0 | 1 | 0 | 0 | 2 | 0 | 0 | 4 | 0 | 0 | 4 |
| S | parallelism (1) | 1 | 1 | 1 | 1 | 1 | | | 1 | | | 1 | | | 1 | | |
| | Block size (word) | 256 | 102 | 1160 | 820 | 767.5 | | | 400 | | | 2302.5 | | | 1152 | | |
| | Word size (bit) | 3 | 4 | 3 | 4 | 3 | | | 4 | | | 3 | | | 4 | | |
| | RAM block usage | 1 | 1 | 1 | 1 | 5 | 0 | 0 | 0 | 1 | 0 | 14 | 0 | 0 | 9 | 0 | 0 |
| | RAM block left | 5 | 13 | 36 | -2 | 51 | 18 | 1 | 0 | 9 | 0 | 249 | 4 | 5 | 118 | 4 | 5 |

(c) Frédéric Guilloud, Télécom Paris - July 2004

and 2 Altera Stratix), a comparison is made on the size of regular LDPC codes that can be implemented, as a function of the memory resources of each FPGA. Two kinds of codes are illustrated: the first one is a regular $(3, 6)$ LDPC code of rate $R = 0.5$ and the second one is a regular $(3, 15)$ LDPC code of rate $R = 0.8$. The main difference between the Virtex and the Stratix families is that for the Stratix series, three different kind of RAM block are available inside a FPGA; they are referred to by the (a), (b) and (c) columns in table 5.4. So the possible code length is higher. Note that these examples are just an overview of what would be possible with nowadays FPGA. It means that an optimization of the memory usage would yield an implementation of longer codes for example. Note also that distributed memories built with the slices of the FPGA are not considered. This would also increase the size of the available memory and thus would enable to implement longer codes too. However, table 5.4 shows that with existing FPGA technologies, it is possible to build such a generic LDPC decoder for codes of a few 10k bits length, with rate $R = 0.5$.

**Throughput**

The information rate is given by (3.18), in section 3.3:

$$
\begin{aligned}
D_b &= \frac{R_e K f_{\text{clk}}}{i_{\max}\Gamma} \text{ [information bit/s]} && (5.2)\\
&= \frac{f_{\text{clk}} PRN}{i_{\max} kM} && (5.3)\\
&= \frac{f_{\text{clk}} PR}{i_{\max} k(1-R)} && (5.4)\\
&= \frac{f_{\text{clk}} PR}{i_{\max} j} && (5.5)
\end{aligned}
$$

So $D_b$ increases as the parallelism rate $P$ and the rate $R$ increases, and decreases as the variable degree increases. In the examples of table 5.4, where $P = 8$, 2 different throughput are possible, if we choose $f_{\text{clk}} = 10$ [MHz] and if assume that $i_{\max} = 10$:

- $R = 0.5$: $D_b = 6.667$ [Mbit/s] ,
- $R = 0.8$: $D_b = 10.667$ [Mbit/s] .

. These numerical applications show that with these existing FPGA technologies, out design make also possible the implementation of a generic LDPC decoder having a throughput a several mega bits per second.

## 5.3 Shuffle network

The shuffle network performs the spatial shuffle which is a part of the whole interconnection network achievement. The other part is the time shuffle, achieved by the random access of the variable information memories.

(c) Frédéric Guilloud, Télécom Paris - July 2004

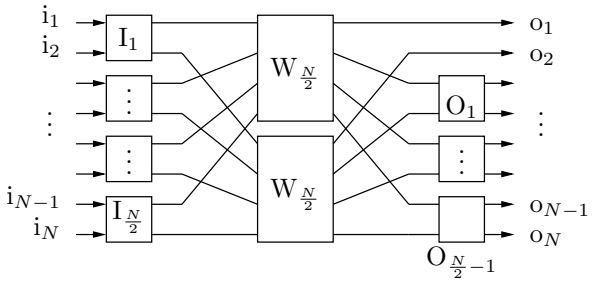



Figure 5.12: Waksman's permutation generator of order $P$: $W_P$; iterative synthesis of $W_P$ using $W_{P/2}$, $P = 2^r$

### Choice

The genericity specification of the decoder yields to have potentially all the spatial permutations achievable. If some permutations are impossible, then some randomly chosen LDPC codes could not be decoded.

A first possible solution is to implement $P$ multiplexors (MUX) ($P \rightarrow 1$) and to code the shuffle address on $P\log_2(P)$ bits. But there's no need to code more than $P!$ permutations since 2 bits can not go to the same serial parity-check processor. Coding the overall permutation requires $\log_2(P!)$ bits. Using Stirling approximation, we can save about:

$$P\log_2(P) - \log_2(P!) \approx P \tag{5.6}$$

bits on the size of the shuffle address.

Of course, here again, only small values of $P$ are acceptable. For large values, a limitation in the number of possible permutation should be accepted.

### Permutation generator

The permutation generator is a device which can output any of the $P!$ permutations of its $P$ inputs. It should be made of 2-input 2-output switches to avoid complex routing. A switch is a device which lets the data go through with or without crossing them. Only one bit is needed to address one switch (figure 3.2).

Hence, the minimum required number of switch is then equal to $\log_2(P!)$. If $N$ is a power of 2, a generic structure exists (Waksman 1968) which requires $P\log_2(P) - P + 1$ switches (figure 5.12). But if $P$ is not a power of 2, it is more difficult to find an efficient generic structure. Figure 5.13 describes non-optimal permutation generator for $N =$ 3, 5, 6, 7.

Figure 5.14 depicts the size of the shuffle address (in bits) for the 3 different cases: the use of $P$ MUX, the lower bound and the Waksman solution, interpolated by our solutions when $P$ is not a power of 2.

(c) Frédéric Guilloud, Télécom Paris - July 2004

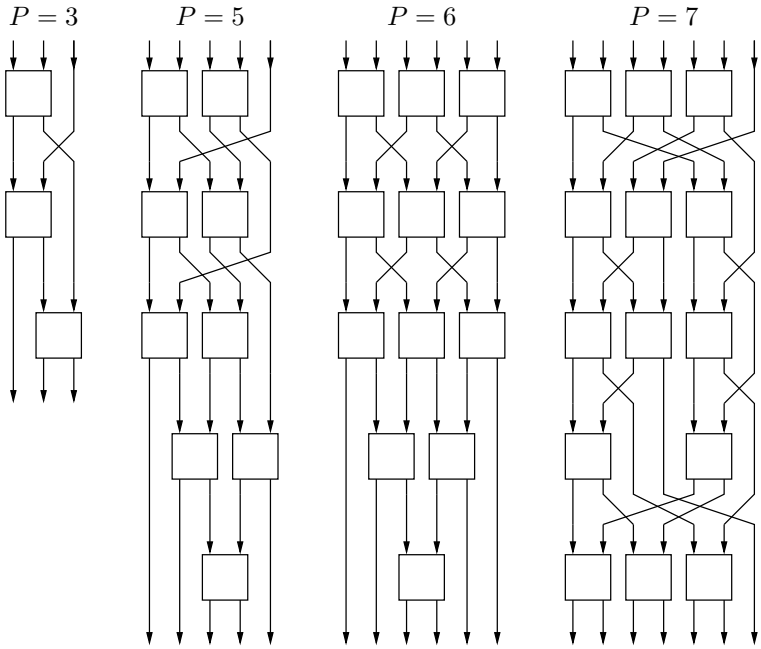



Figure 5.13: Some permutation generator of order $N \neq 2^r$

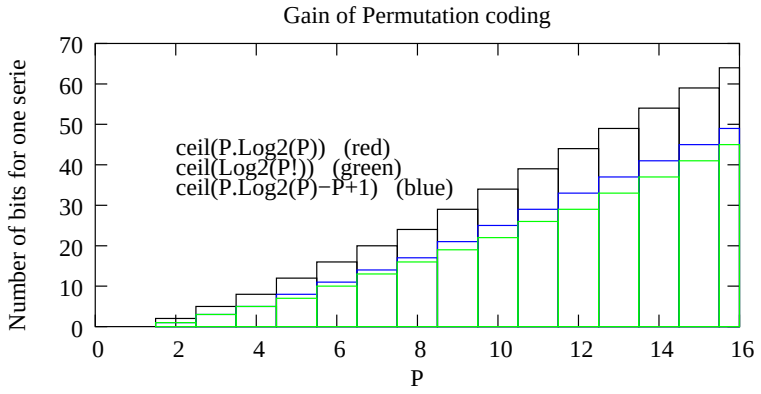



Figure 5.14: The number of bit saved by coding the permutation of the variables increases linearly. The number of bit required when using $P$ MUX is represented by the red bars. The blue bars are for the use of Waksman generator, interpolated to any value of $N$, and the green bars are for the ideal generator.

(c) Frédéric Guilloud, Télécom Paris - July 2004

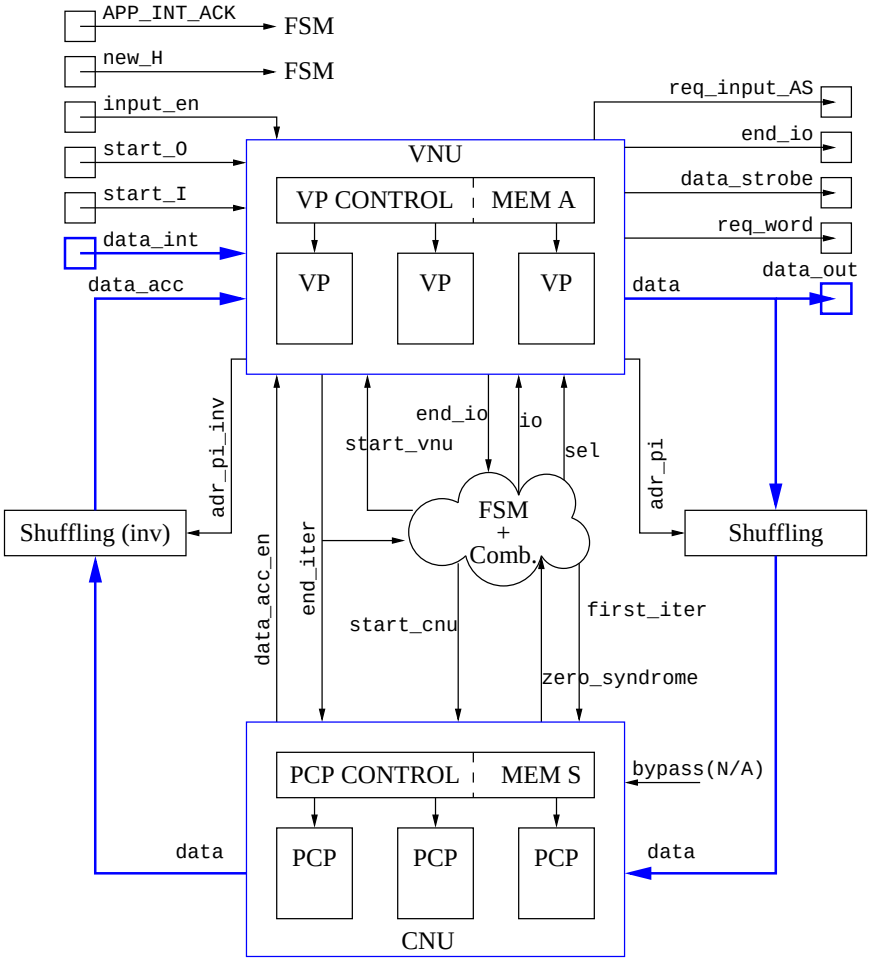



Figure 5.15: The top-level architecture of the decoder

Note that the authors of (Thul, Gilbert, and Wehn 2003) proposed an ring-interleaver. The complexity is reduced as compared to interconnection networks, but at the expense of a high and not constant latency.

## 5.4 Description of the architecture

The architecture of the top-level is illustrated on figure 5.15. It is divided into 2 units

- the Check Node Unit (CNU): it is where the computation of the extrinsic values is made. The CNU features $P$ parity-check processors (PCP). The memories $L$ and $P_t$ are included in the PCP. The memory $S$ is also inside the CNU.

- the Variable Node Unit (VNU): it is where intrinsic and extrinsic information is saved and accumulated. The VNU features $P$ Variable Processor (VP). The mem-

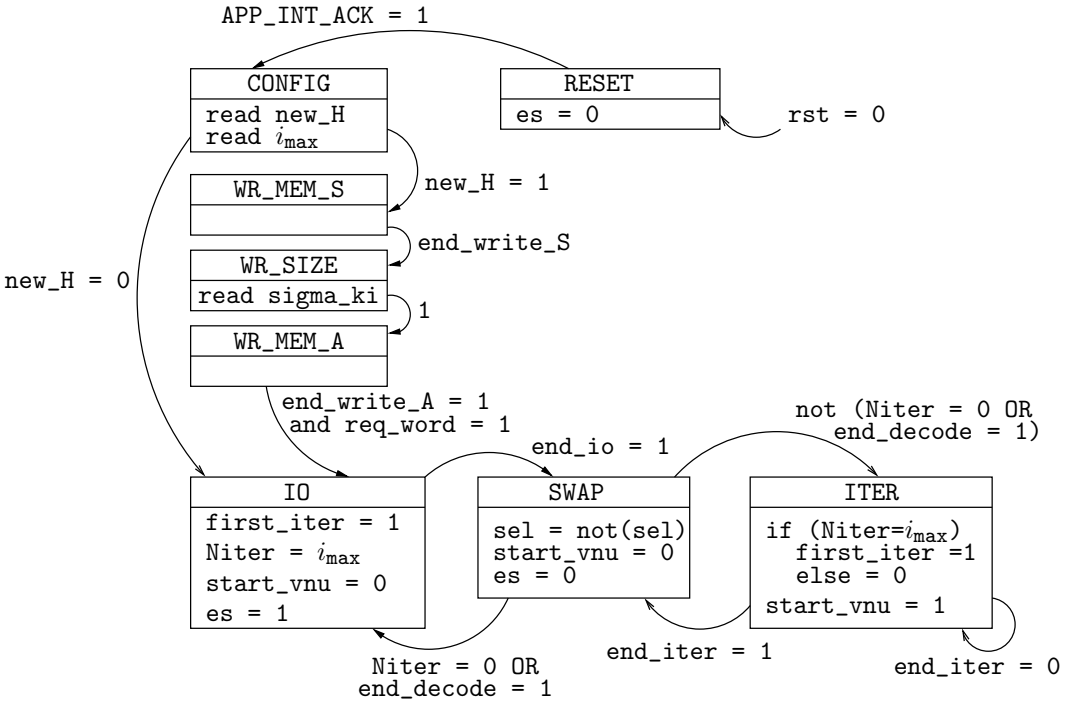



Figure 5.16: Top-level Finite State Machine

ories $I$, $E_1$, $E_2$ and $V$ are inside the VP. The memory $A$ is also inside the VNU.

These 2 units feature a common control for their $P$ processors, avoiding multiple instances of the same control components. The information is exchanged between each others during each iterations through a the permutation network and its inverse.

The detailed architecture of the PCP and of the VP are presented in annexe C, respectively in section C.1 and C.2. The top-level architecture is controlled by the top-level finite state machine (FSM) depicted on figure 5.16. It has 4 main states and a secondary FSM for the configuration control, which will not be detailed here:

1. `RESET` when `rst` is set to 0.

2. `CONFIG` (secondary FSM) when the `APP_INT_ACK` is set to 1 (this signal is considered as the start signal for the decoder). The configuration of the decoder is read at this moment: the maximum number of iterations $i_{\max}$ and the `new_H` signal which specifies if a new parity-check matrix has to be decoded. The input of a new parity-check matrix is also processed by this secondary FSM.

3. `IO` when `new_H` is set to 0 or when a codeword has been decoded. This state correspond to the input and output data transfers.

4. `ITER` when an iteration is being proceeded.

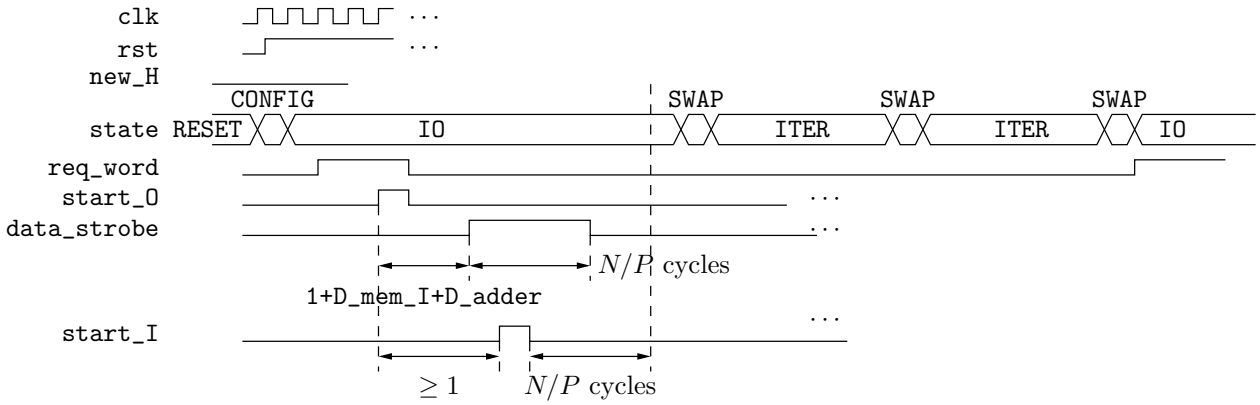



Figure 5.17: Top-level waveforms when the `new_H` signal is set to zero.

5. `SWAP` after IO or ITER; this one clock cycle state swaps the role of the 2 extrinsic memories.

The waveforms of the main signals are depicted on the figure 5.17, where some signals are defined in the finite state machine of the decoder. The `start_O` input enables the decoder to output the result of the decoded word on `data_out`. The `start_I` input enables the decoder to read the word to be decoded on `data_input`. These operations last $N/P$ clock cycles, and the output one has to be done first, so as not to erase the decoded word by the word to decode. The `data_strobe` output port is used to know when the `data_out` one is valid.

## 5.5 Universality of the decoder

A very few generic architectures for LDPC codes have been published in the literature. In fact we have not been aware of such any generic LDPC-decoder platform. Some scalable architecture are described which are a template architecture description for various kind of LDPC codes. The authors of (Kienle, Thul, and Wehn 2003) for example, discussed about implementation issues of scalable LDPC code decoders. This architecture however does not seem to handle more than one code once the synthesis tool has been processed. Another scalable architecture is given by (Verdier and Declercq 2003). The authors designed a highly structured LDPC code decoder which performs as well as randomly generated codes. Here again, not all the degree distributions can be implemented.

In our architecture, before the synthesis has been performed, the set of parameters enables to decode all the LDPC codes. Once the set of parameters $N$,$M$ and $P$ has been decided, all the $(M \times N)$ parity-check matrices can be decoded, whatever the degree distribution can be: parity-check matrices can be either randomly designed according to the architecture, or an existing parity-check matrix can be also decoded using a preprocessing. This preprocessing will prepare the data of the A and S memories, to schedule the

decoding process as described in section 5.2.4.

### 5.5.1 Randomly designed LDPC codes

Two softwares have been written to generate a random LDPC code, either based on a variable node degree distribution or on a check node degree distribution. Both of them can be obtained on-line thanks to (Chung ; Urbanke ) website.

#### Check node degree specification

We will assume here that the check node degree distribution has a granularity of $P$. It means that the number of check node having the same degree is a multiple of $P$. So the first work to do is to adapt the distribution on the required parameters of the parity-check matrix. For the general case, please refer to section 5.5.2. Note however that it is not a hard constraint since $P$ is not very large.

The random generation of the parity-check matrix consist of generating the random addresses of the memory $A$ (Spatial and time shuffle), according to a given distribution profile: first, the check node are sorted in the ascending order; then a permutation address and $P$ memory addresses are randomly chosen. This operation is repeated as many times as the degree of the $P$ check nodes under process. The only constraint is that a variable should not be randomly chosen twice by the same check node. The software tries to have a uniform variable node degree distribution: the variable node degree should be constant, or at least should be made of only 2 successive values. As an illustration, a random rate$-0.5$ LDPC code of length $N = 2048$ is depicted on figure 5.18. The check nodes have degrees 6 ($\rho_6 = 0.6$) or 8 ($\rho_8 = 0.4$), yielding 1368 variables to have a degree 3 and 680 to have a degree 4.

#### Variable node degree specification

The random generation of a parity-check matrix having a given variable node degree specification is processed approximately in the same way as for the check-node degree specification case. The constraint is that the check node degree should be constant, or at least 2 successive degrees, all of them being multiples of $P$.

### 5.5.2 Preprocessing of existing LDPC codes

The first preprocessing step is to sort the parity-check matrices by ascending order of their degree, as specified in the flow of operations of the parity-check processor, section 4.4.1.

The second step is to rearrange the non-zero entries of the parity-check matrix. The number of non-zero entries should be the same in all the regions described by the intersection of a memory bank and a set of $P$ parity-checks processed at the same time. It

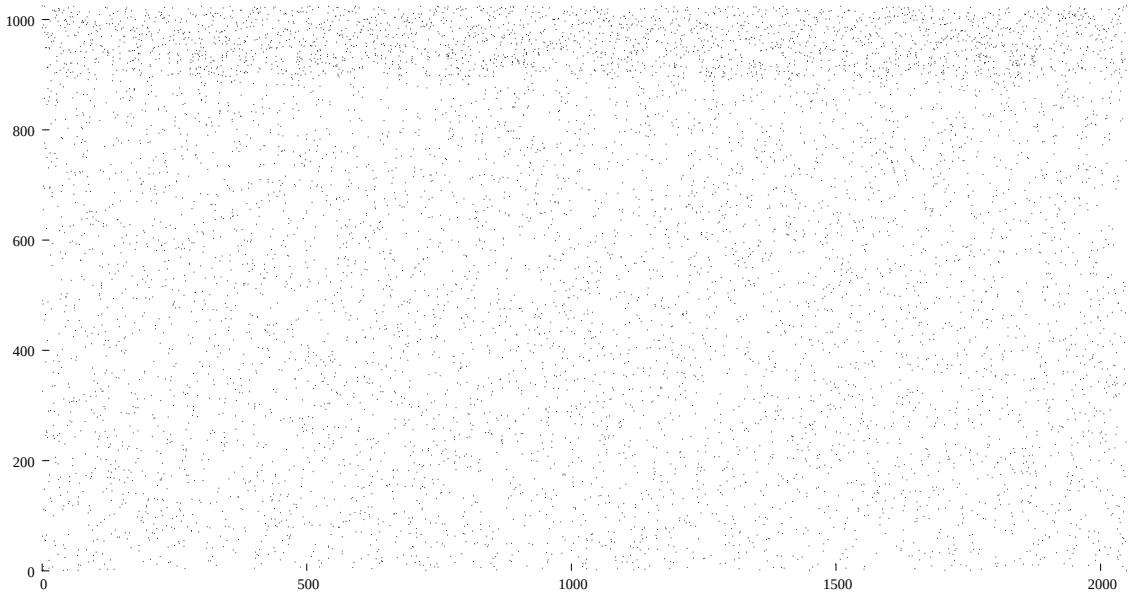



Figure 5.18: An illustration of a random parity-check matrix

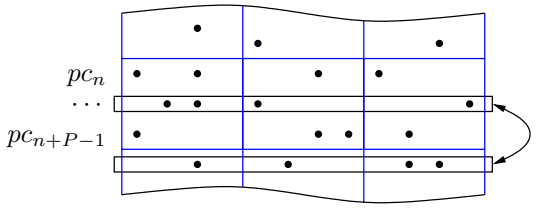



Figure 5.19: Row permutation for spreading the non-zero entries of $H$.

is possible to process rows permutations, as depicted on figure 5.19, provided that they have the same degree.

If no solution can be found by row permutations, column permutations can be performed on the parity-check matrix, as depicted on figure 5.20. If the encoding has been performed with an efficient scheme, such as in (Richardson and Urbanke 2001), the column permutation should be avoided as much as possible , since it requires also a permutation of the received variables before being decoded.

Finally, for the last blocking cases, a "NOP" variable can be introduced (see figure 5.21). This is in fact a No-Operation variable which is added to the $N/P$ variables of the code. This special address will be detected and process separately for doing nothing.

Providing these preprocessing solutions, all the type of parity-check matrices can be decoded by our architecture. This is a important feature for an LDPC performance evaluation tool. Note that the preprocessing software and the special NOP processing implementation are part of our perspectives.

(c) Frédéric Guilloud, Télécom Paris - July 2004

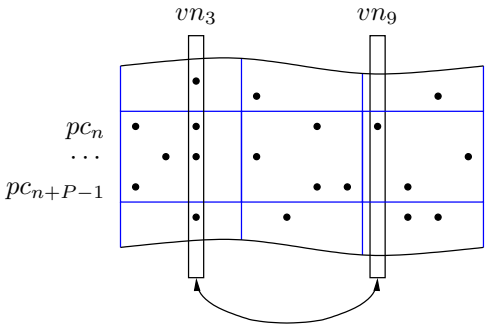



Figure 5.20: Column permutation for spreading the non-zero entries of $H$.

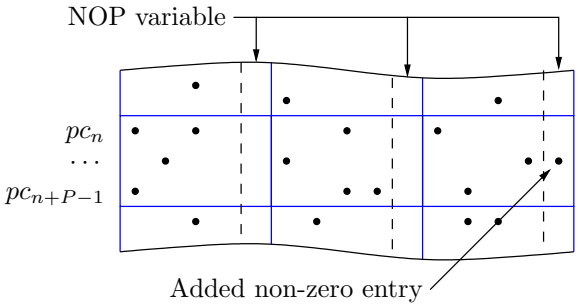



Figure 5.21: Using a "NOP" variable to allow irregular degrees inside regions of the same processing phase.

(c) Frédéric Guilloud, Télécom Paris - July 2004

## 5.6 Conclusion

We presented in this chapter a generic architecture of a decoder which accepts any LDPC codes. The meaning of genericity is twofold:

- first, for a given code size $(N, M)$, and a given parallelism rate $P$, any LDPC code can be decoded by the decoder, without any synthesis before. The new code is uploaded inside the decoder, and the decoding process is ready to run;
- second, the description of the code is versatile: before a synthesis is to be performed, any parameters can be changed. The different components can also be changed easily (shuffle, check node processor, ...).

Even if the architecture has been written for a wide range of LDPC codes, it can be also simplified to fit with some constraint matrices. For example in the work of (Hocevar 2003), only $P$ different spatial permutations are used, instead of the $P!$ possible permutations. The variable addresses are also simplified by the use of circular shifted identity matrices: addressing is a common counter for all the variable processors, as designed also by (Zhang and Parhi 2001).

This generic LDPC decoder implementation will be associated to other LDPC softwares so as to build an LDPC evaluation platform. The synthesis and simulation results of this platform are given in chapter 6.

(c) Frédéric Guilloud, Télécom Paris - July 2004

# Chapter 6

# The platform

**Summary:**

*This last chapter presents in a first part an overview of the platform designed to run intensive simulations with the generic decoder described in the chapter 5. Some synthesis issues are then presented. In a second part, fixed-point simulations are processed to study the influence of quantization on the decoding algorithms. A comparison between software simulations and hardware simulation shows no difference between them. A comparison between the BP and the $\lambda-min$ algorithms is proposed, and the influence of the intrinsic information processing is emphasized. The material of the second part has been partly presented during the GDR ISIS seminar on LDPC (Guilloud, Boutillon, and Danger 2002b).*

## 6.1 Platform description

### 6.1.1 Overview

The simulated communication is depicted on figure 6.1. The encoder is a software running on a PC. The encoded bits are sent to the channel emulator, implemented on a first FPGA, where white gaussian noise samples are added to the BPSK modulated bits. Then the intrinsic information is computed using high precision processing. Finally, the intrinsic information is truncated and rounded, and is sent to the iterative decoding process, on a second FPGA.

This communication platform is implemented on a Xilinx Virtex-E evaluation board from Nallatech systems. The block diagram of the platform is depicted on figure 6.2. It is based on a PCI-card: the Ballyinx card. It is connected to the PCI bus of the PC and it features a Virtex XCV300 in which the PCI interface is implemented. A driver has been written with the Numega SoftIce Driver Suite tools. The driver enables the communication between the Ballyinx and the PC through the PCI-bus, under the

 (c) Frédéric Guilloud, Télécom Paris - July 2004

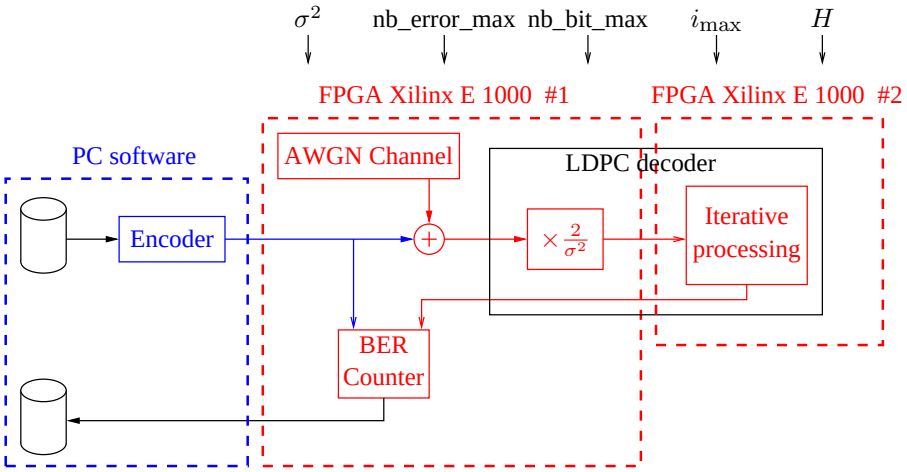



Figure 6.1: The block diagram of the platform for encoding and decoding LDPC codes.

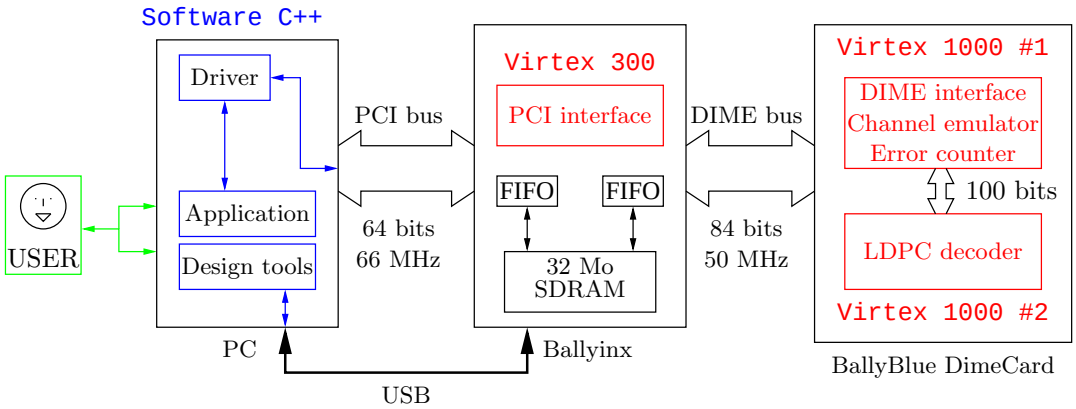



Figure 6.2: The block diagram of the platform using the Nallatech evaluation board.

(c) Frédéric Guilloud, Télécom Paris - July 2004

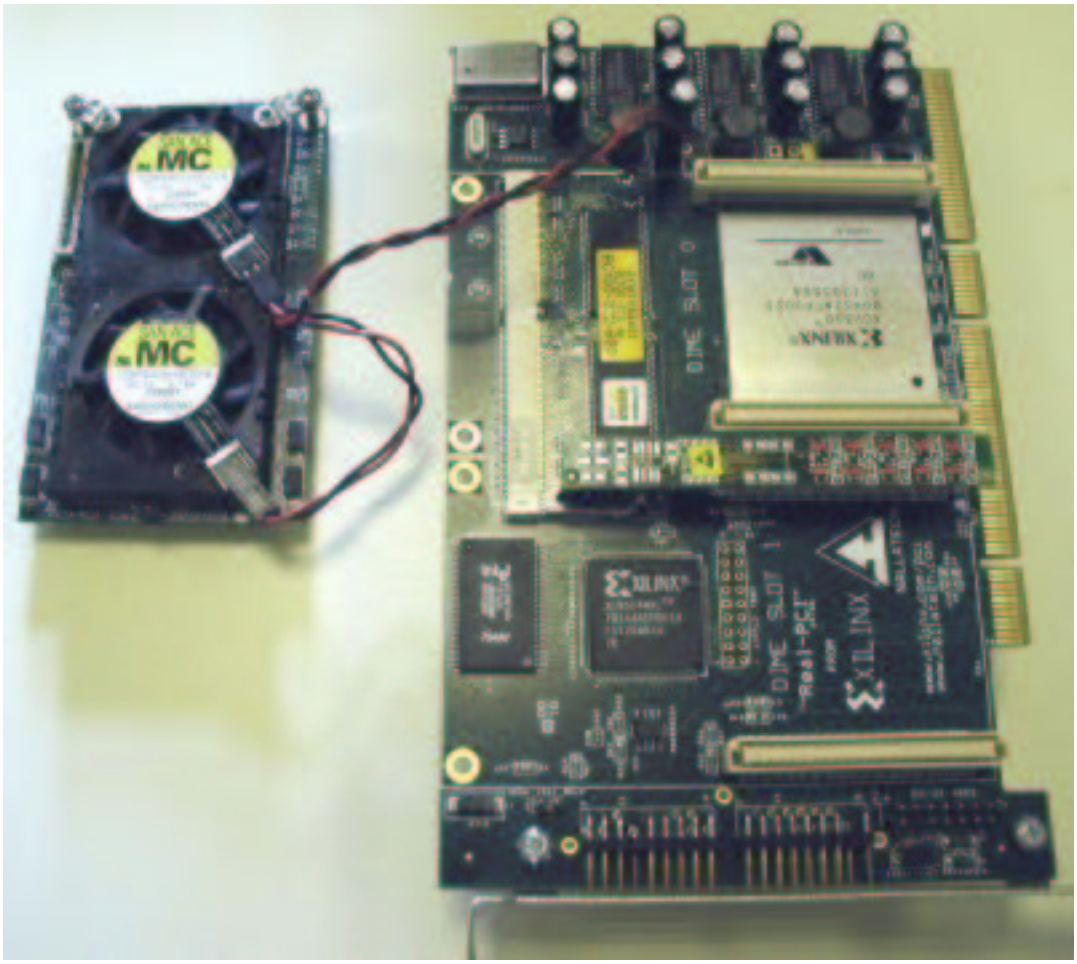



Figure 6.3: The picture of the Ballyinx and the Ballyblue cards (Nallatech evaluation board)

Windows 2000 operating system. A picture of the Ballyinx and the BallyBlue cards is shown on figure 6.3. A C++ software has also been developed to handle the interruptions and to manage the input/output dataflow between the Ballyinx and the PC.

Some DIME-modules can be added to the Ballyinx card and communicate through a DIME bus. One such DIME-module is used which features two Virtex XCV1000E, and is called the BallyBlue card. The second FPGA of the BallyBlue card features the generic implementation of the LDPC decoder which has been described in chapter 5. The first FPGA features all the other blocks of the communication scheme, which are:

- the DIME interface which has been developed as an interface between the BallyBlue and the Ballyinx,
- the channel emulator for the AWGN channel, including the intrinsic information,

(c) Frédéric Guilloud, Télécom Paris - July 2004

- the communication protocol manager between the DIME interface and the LDPC decoder, including an input buffer and an error counter.

### 6.1.2 Intrinsic information processing

The bits coming from the encoder are modulated with a BPSK modulation. Then some gaussian noise with variance $\sigma$ is added to the modulated inputs. The white gaussian noise generator (WGNG) designed by (Danger et al. 2000; Ghazel et al. 2001; Boutillon, Danger, and Gazel 2003) has been used. Finally, the LLR of the channel output is computed and sent to the decoder. We suppose here that the output of the BPSK modulation are the samples $(-1)^{c_n}$, assuming that $\sqrt{E_s} = 1$. From equation (1.33), we have:

$$I_n = (\sigma\mathcal{N}(0,1) + (-1)^{c_n}) \times \frac{2}{\sigma^2} \tag{6.1}$$

where $\mathcal{N}(0,1)$ is the centered probability density function of the gaussian random variable with standard deviation $\sigma = 1$. Equation (6.1) requires one multiplication and one division. So for implementation purpose, (6.1) can be written:

$$I_n = \left(\mathcal{N}(0,1) \pm \frac{1}{\sigma}\right) \times \frac{2}{\sigma} \tag{6.2}$$

where there is one division left. The output of the WGNG is coded with a fixed-point convention. To adjust as much as possible the dynamic range with the number of bits used, the decoder has to take into account dynamics which are not a power of 2. The dynamic range is denoted $\Delta$, and is coded using $N_b$+1 bits. By convention, let $\Delta$ represent the integer value $2^{N_b}$. So the maximum value coded on $N_b$ bits is $\frac{2^{N_b}-1}{2^{N_b}}\Delta$. Hence, (6.2) can be replaced by:

$$I_n = \left(\mathcal{N}(0,1) \pm \frac{1}{\sigma}\right) \times \frac{2}{\sigma} \times \frac{2^{N_b}}{\Delta} \tag{6.3}$$

If $1/\sigma$ is restricted to the multiples of $\Delta$: $1/\sigma = \gamma \times \Delta$, then (6.3) becomes:

$$I_n = \left(\mathcal{N}(0,1) \pm \frac{1}{\sigma}\right) \times \gamma 2^{N_b+1} \tag{6.4}$$

Equation (6.4) is easier to implement than equation (6.2) since the division is replaced by a multiplication, and also right shifts. The architecture of the channel emulator is depicted on figure D.1, in annexe 6. The inverse of sigma is coded on $[3,6]$ bits – 3 bits for the integer part, 6 for the fractional part – and should be a multiple of $\Delta$. It is signed by the incoming coded bits and then added to the output of The results are then multiplied by $\gamma$ which results in a $[8,19]$ bits coded value. The magnitude part is clipped to $\pm\Delta$ and rounded to fit on $N_b$ bits. The output is the concatenation of the coded bit, the sign of the noisy sample and its magnitude. $P$ successive bits are processed, with a latency of $AP + 2$ and then $P$ words of $N_b + 2$ bits are output.

(c) Frédéric Guilloud, Télécom Paris - July 2004

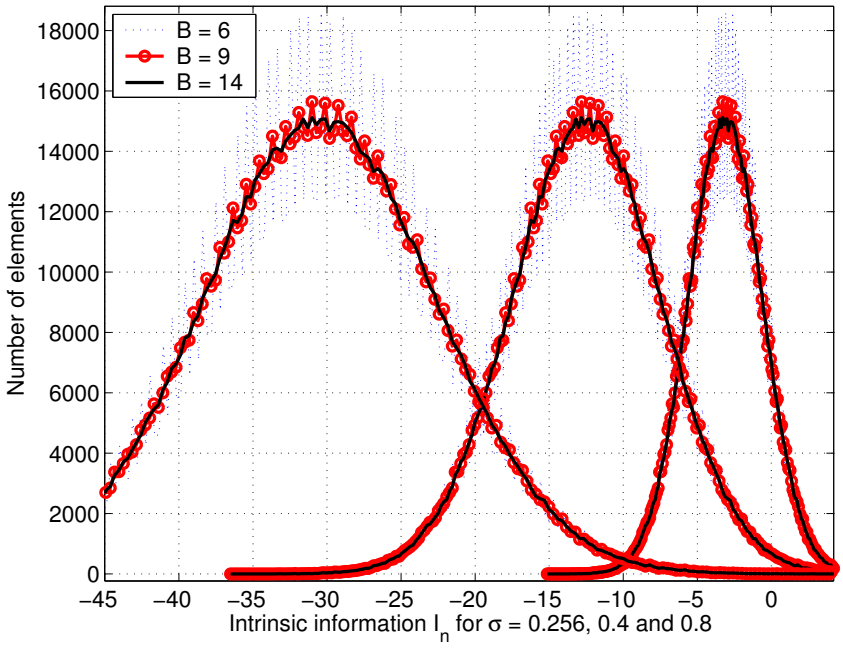



Figure 6.4: Distribution of the intrinsic information $I_n$ as a function of the decimal precision of the gaussian noise generator, and for several value of noise power.

In the WGNG, $A = 4$ successive accumulations are processed on the box-muller variables and the output is not truncated ($B = 14$). A correction block has been added in the WGNG as specified by (Danger et al. 2000) to compensate the non-zero mean by an offset of $A.2^{-B-1}$ and to have a standard deviation set to 1 instead of $\sqrt{A}$. Figure 6.4 depicts the influence of the number of bits $B$ of the fractional part on the distribution of $I_n$, when $\sigma = 0.256, 0.4, 0.8$ and for a coded bit $c_n = 1$. $I_n$ is a gaussian random variable with a mean of $2/\sigma^2$ (respectively $-30.5$, 12.5 and 3.125) and a variance of $2/\sigma$ (respectively 7.8, 5 and 2.5). We can see that if $B$ is not high enough, the less significant bits are lost after the clipping and thus the distribution of $I_n$ contains some peaks: the number of samples before clipping is not the same between each power of 2 value because of the multiplication factor $\gamma$.

## 6.2 Synthesis

### 6.2.1 Illustration of the genericity

The synthesis of the decoder is processed under some fixed constant which can be modified inside a VHDL package as described in the listing E.1, in the annexe E.

The different vector sizes are then calculated using the logarithm to base 2 thanks to a VHDL function which process successive divisions by 2. The latencies of all the different

blocks can also be changed in the package, and the different delays which are implemented will be adjusted, as described in listing E.2 in annexe E. Hence, any different blocks can be changed for comparison purpose.

Once the synthesis is complete, every LDPC code with the same length and the same rate can be decoded, assuming it can bear an edge rate of $P$. So the distribution degree, the non-zero entries of the parity-check matrix and the number of decoding iteration of course can be changed on-the-fly.

The synthesis itself is also generic: numerical examples for the FPGA memory usage are listed in table 6.1 for several combinations of the decoder parameter values. They are based on table 5.3. The information bit rate given in table 5.3 is computed thanks to the equation (5.3), which can be also expressed:

$$D_b = \frac{f_{\text{clk}} NP(1 - M/N)}{kMi_{\max}} \tag{6.5}$$

Note that this information-bit rate is a lower bound since the number of iterations can be lower than $i_{\max}$. The other parameters in table 6.1 are the weight $k$ of the parity-check equations (the LDPC code is supposed to be regular). For each kind of memory, it is specified the number of independent block of memory, the number and the size of the words to store and the number of FPGA RAM blocks required. The last column gives the number of RAM block left in the FPGA: if it is less than zero, the architecture does not fit in the FPGA. Table 6.1 is on the same template as table 5.4, but for the FPGA xilinx Virtex 1000 only. Each line is an example of a particular case.

The first case is an example of a high rate code ($R = 0.875$) of length $N = 8192$. Since the number of parity-check constraint is lower, the bit rate can be very high. The other codes listed below the first case have rates $R = 0.5$. Of course, the bit rate is then much lower (2nd case). If the bit rate is doubled (3rd case), the edge rate is doubled and thus the A-memory is to big to fit in the FPGA (all the permutations have to be possible, so the word size in A increases exponentially). With a lower edge rate (4th case), the density can be increased to $k = 7$ bit per parity-checks. The density can also be increased to $k = 16$ (5th case) if the length of the code is divided by 2. The 6th case is the example of the 2nd case with the $4 - min$ algorithm implemented instead of the $3-$min one. So the number of LLR to be saved for each parity increases from 4 to 5, *i.e.* from 25%. This is why the size of the L-memory (value of the minima) is also increased by 25%. For the 5 first cases, the number of RAM block needed for the A-memory was supposed to be optimized: it means that the number of RAM block to concatenate to get a word of the required size versus the number of RAM block to add to get the number of required words was optimized. For both the 6th and the 7th cases, it is assumed that the A-memory has a fixed dimension (2048 words of 64 bits), since it can not be inferred by the synthesis tool. In the 6th case, the size of the LDPC code is limited by the maximum size of 64 bits in the words of the memory $A$. For the last case, the size of a word in the memory is lower than 64 bits by decreasing the value of $P$, and thus the size of the spatial shuffle

(c) Frédéric Guilloud, Télécom Paris - July 2004

Table 6.1: FPGA RAM Block usage for several examples.

| | FPGA | LDPC decoder | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| case | Clk (Mhz) | N | M | P | i_max | Nb | k | lambda | Rate (Mb/s) |
| 1 | 50 | 8192 | 1024 | 8 | 10 | 8 | 6 | 3 | 46.66667 |
| 2 | 50 | 4096 | 2048 | 8 | 10 | 8 | 6 | 3 | 6.666667 |
| 3 | 50 | 4096 | 2048 | 16 | 10 | 8 | 6 | 3 | 13.33333 |
| 4 | 50 | 4096 | 2048 | 2 | 10 | 8 | 7 | 3 | 1.428571 |
| 5 | 50 | 2048 | 1024 | 8 | 10 | 8 | 16 | 3 | 2.5 |
| 6 | 50 | 4096 | 2048 | 8 | 10 | 8 | 6 | 4 | 6.666667 |
| 7 | 50 | 1536 | 768 | 6 | 10 | 8 | 6 | 3 | 5 |
| 8 | 50 | 3400 | 1700 | 5 | 10 | 8 | 6 | 3 | 4.166667 |

| | Intrinsic and Extrinsic (I,E1,E2) | | | | First accumulation flag (V) | | | | |
|---|---|---|---|---|---|---|---|---|---|
| case | # blocks | Block size (words) | Word size (bits) | FPGA blocks Number | # blocks | Block size (words) | Word size (bits) | FPGA blocks Number | |
| 1 | 24 | 1024 | 8 | 48 | 8 | 1024 | 1 | 8 | |
| 2 | 24 | 512 | 8 | 24 | 8 | 512 | 1 | 8 | |
| 3 | 48 | 256 | 8 | 48 | 16 | 256 | 1 | 16 | |
| 4 | 6 | 2048 | 8 | 24 | 2 | 2048 | 1 | 2 | ... |
| 5 | 24 | 256 | 8 | 24 | 8 | 256 | 1 | 8 | |
| 6 | 24 | 512 | 8 | 24 | 8 | 512 | 1 | 8 | |
| 7 | 18 | 256 | 8 | 18 | 6 | 256 | 1 | 6 | |
| 8 | 15 | 680 | 8 | 30 | 5 | 680 | 1 | 5 | |

| | Sign and LLR addresses (Pt) | | | | LLR Values (L) | | | | |
|---|---|---|---|---|---|---|---|---|---|
| case | # blocks | Block size (words) | Word size (bits) | FPGA blocks Number | # blocks | Block size (words) | Word size (bits) | FPGA blocks Number | |
| 1 | 8 | 768 | 2 | 8 | 8 | 512 | 8 | 8 | |
| 2 | 8 | 1536 | 2 | 8 | 8 | 1024 | 8 | 16 | |
| 3 | 16 | 768 | 2 | 16 | 16 | 512 | 8 | 16 | |
| 4 | 2 | 7168 | 2 | 8 | 2 | 4096 | 8 | 16 | ... |
| 5 | 8 | 2048 | 2 | 8 | 8 | 512 | 8 | 8 | |
| 6 | 8 | 1536 | 2 | 8 | 8 | 1280 | 8 | 24 | |
| 7 | 6 | 768 | 2 | 6 | 6 | 512 | 8 | 6 | |
| 8 | 5 | 2040 | 2 | 5 | 5 | 1360 | 8 | 15 | |

| | Time shuffle and spatial shuffle (A) | | | | Parity Check Size (S) | | | | |
|---|---|---|---|---|---|---|---|---|---|
| case | # blocks | Block size (words) | Word size (bits) | FPGA blocks Number | # blocks | Block size (words) | Word size (bits) | FPGA blocks Number | FGPA RAM blocks left out of 96 |
| 1 | 1 | 768 | 97 | 19 | 1 | 128 | 3 | 1 | 4 |
| 2 | 1 | 1536 | 89 | 34 | 1 | 256 | 3 | 1 | 5 |
| 3 | 1 | 768 | 177 | 34 | 1 | 128 | 3 | 1 | -35 |
| 4 | 1 | 7168 | 23 | 41 | 1 | 1024 | 3 | 1 | 4 |
| 5 | 1 | 2048 | 81 | 41 | 1 | 128 | 4 | 1 | 6 |
| 6 | 1 | 1536 | 89 | 34 | 1 | 256 | 3 | 1 | -3 |
| 7 | 1 | 768 | 59 | 12 | 1 | 128 | 3 | 1 | 47 |
| 8 | 1 | 2040 | 56 | 28 | 1 | 340 | 3 | 1 | 12 |

(c) Frédéric Guilloud, Télécom Paris - July 2004

address. The limiting parameter is then the total number of words in the memory A, which limits the length of the code.

### 6.2.2 Synthesis results

The synthesis has be processed by Leonardo Spectrum 2003-b35. The other tools are from Xilinx (ngdbuild, map, par and bitgen).

A synthesis has been performed with the parameters of the 7th case, described in table 6.1. The summary of the mapper is listed on listing E.3, in annexe E. The number of used block RAMs is equal to 88 as expected in table 6.1. One can also note that only 25% of the slices are used.

## 6.3 Simulations

### 6.3.1 Simulation conditions

The C++ program which has been written for simulating floating point decoding of LDPC codes, described in section 4.3.1, features also a fixed-point representation for the data. For the fixed point coding, two parameters are defined:

- the number of bits $N_b$ used to code the magnitude of the signal;
- the dynamic range $\Delta$ which is the highest magnitude representable.

This representation enables to have a dynamic range which is not necessary a power of 2, and hence the quantization is not a power of 2 as well. The conversion between an absolute real value $r$ and its quantized integer value $r_q$ is:

$$r_q = \left\lceil \frac{r(2^{N_b} - 1)}{\Delta} \right\rceil \tag{6.6}$$

Note that similar fixed-point coding has been also used independently by (Jones et al. 2003).

The same decision rule is applied as described in section 4.3.1: a bit is said to be wrong if the total information $T_n$ is negative, and it is said to be right if it is positive. When the total information is equal to zero, the bit error rate depends on the decision to take, since the all-zero codeword has been sent for all the simulations. Counting half of the bit having a null reliability as errors and the other half as right bits is then considered as a good approximation for the BER which is obtained using random codewords generation.

### 6.3.2 On the BP algorithm

Fixed point simulations have been performed for the code $\mathcal{C}_1$. The figure 6.5 depicts the BER performance for several dynamic range ($\Delta = 2, 5, 10, 40$) and several number of bit

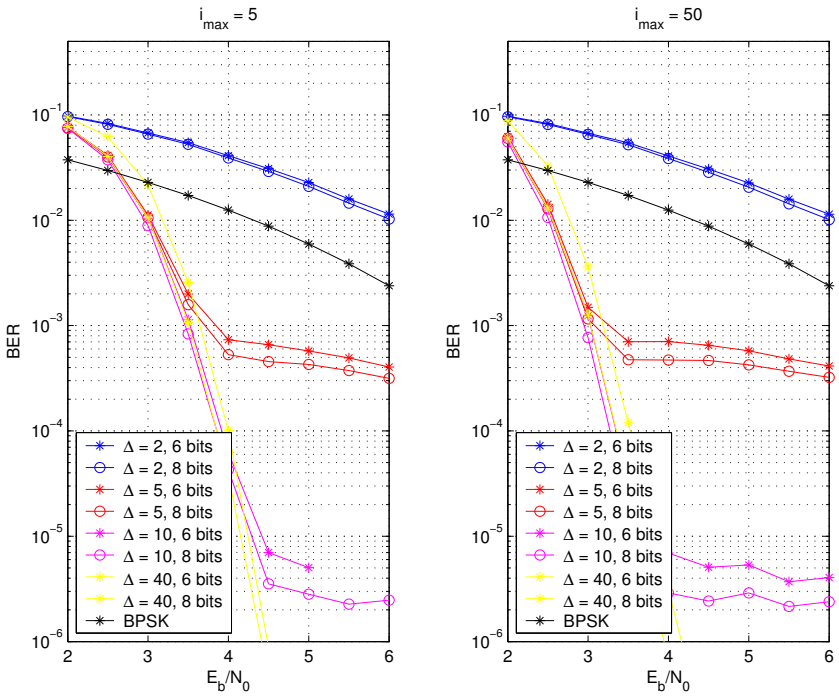



Figure 6.5: Influence of the fixed point coding on the BP algorithm

for the fixed point coding ($N_b = 6, 8$). The first point to notice is the significant error floor, which can be denoted quantization error floor, since it is a consequence of the fixed point coding and not a consequence of the LDPC code itself. This error floor is directly linked to the dynamic range. The wider the dynamic range is, the lower the error floor is. For example, for a dynamic range of $\mp 40$, the error floor is far below $10^{-6}$. If the dynamic is doubled, from $\mp 5$ to $\mp 10$, the error floor is decreased of approximately 2 decades. We can also notice that the number of iterations enables only to let the BER be more vertical before the error floor. But it does not seem to increase the error floor. The number of fixed-point coding bits does not seem to have much impact on the performance. The reason is that in this case the intrinsic information $I_n$ is computed using a floating point coding and floating point operations. The choice of a number $N_b$ of bits and of a dynamic range $\Delta$ yields the least significant bit to weight as much as $q_0 = \Delta 2^{-N_b}$. The influence of $q_0$ is easier to see on figure 6.6 where the intrinsic information is computed inside the decoder (CAN labelled). The error floor is still correlated to the dynamic range, and the performance gap to the floating point BER is linked to $q_0$. We can see for example that for the 2 cases ($\Delta = 40, N_b = 8$) and ($\Delta = 10, N_b = 6$) where $q_0 = 0.15625$, the gap to the floating point curve is the same. Similar results have independently been found by (Declercq and Verdier 2003), where density evolution is performed using classical fixed-point coded messages: the integer part is coded on $a$ bits, and the fractional part is coded on $b$ bits.

(c) Frédéric Guilloud, Télécom Paris - July 2004

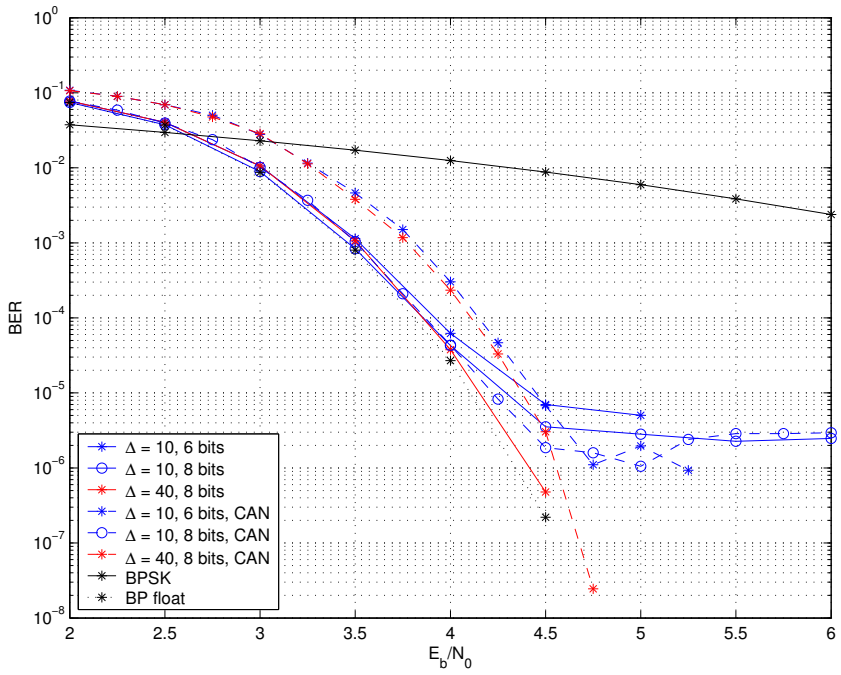



Figure 6.6: Influence of the intrinsic information coding

### 6.3.3 On the $\lambda-$min algorithm

Simulations results for the 3−min algorithm using fixed point coding are depicted on figure 6.7 for code $\mathcal{C}_1$. The fixed point coding is with $N_b = 6$ bits. Two dynamic ranges have been simulated: $\Delta = 10$ and $\Delta = 40$. Each of them are compared with the BP results. The results using floating point coding are also depicted as a reference for both the 3−min and the BP algorithms. The main conclusions that arise from figure 6.7 are that:

1. The performance gap between the BP and the $\lambda-$min algorithms does not seems not increase when the fixed point coding is used instead of the floating point one, in the waterfall region;

2. In fixed point coding, the error floor is higher for the $\lambda-$min algorithm than for the BP algorithm. So this quantization effect depends also on the algorithm, and not only on the quantization parameters.

### 6.3.4 Algorithm comparaison

The genericity of the platform enables to design and compare different suboptimal algorithms such as the 3-min and the A-min* algorithms. Figure 6.8 depicts the different bit error rate obtained for several number of maximum iterations. The code used here is a very small irregular LDPC code of length $N = 400$ and rate $R = 0.5$.

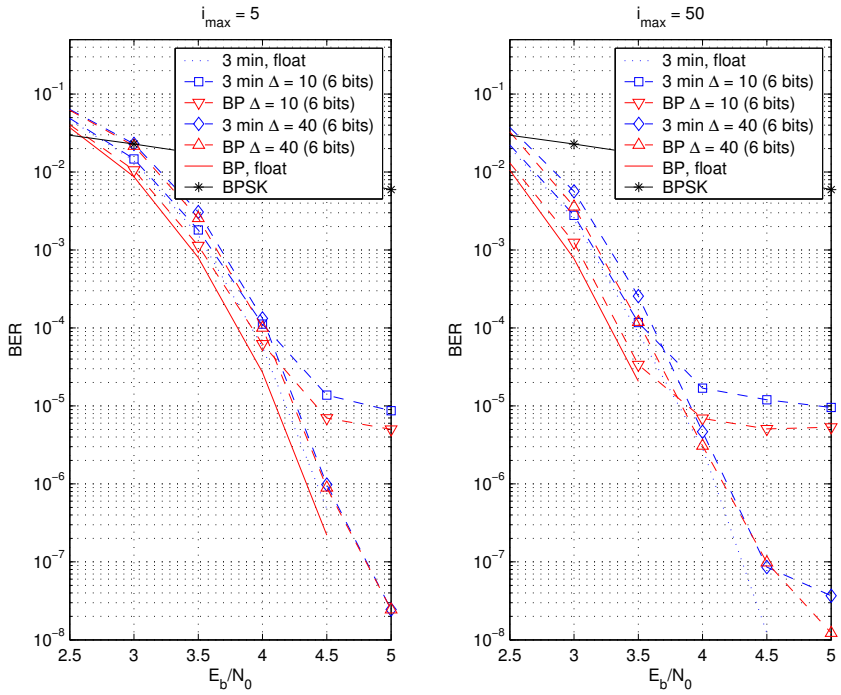



Figure 6.7: Influence of the quantization on the $\lambda-$min algorithm ($\lambda = 3$).

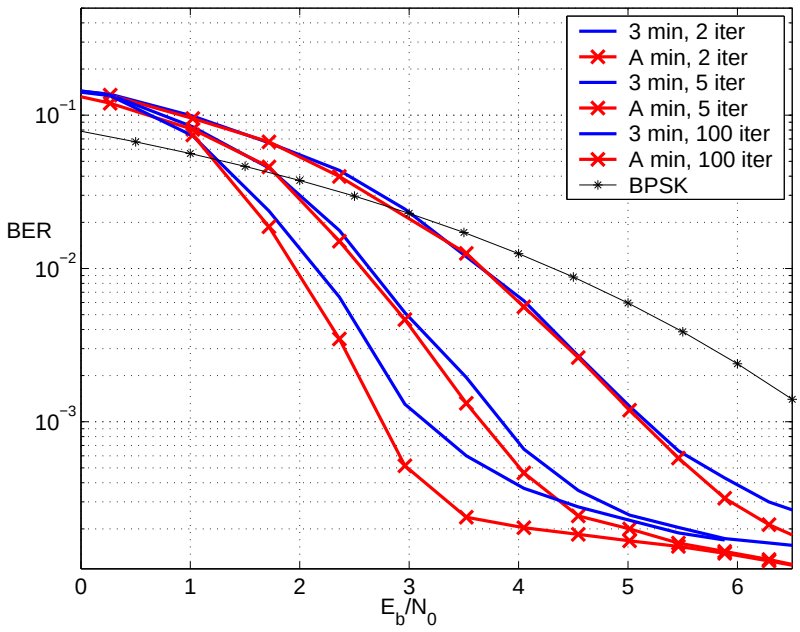



Figure 6.8: Comparaison between the 3-min and the A-min* algorithm.

(c) Frédéric Guilloud, Télécom Paris - July 2004

### 6.3.5 Intrinsic information computing

As described in the previous section, the error floor is due to the dynamic range. And the different ways of calculating the intrinsic information $I_n$ can yield to different performance loss. The intrinsic information (LLR) is defined in (1.33) by:

$$I_n = \frac{2y_n}{\sigma^2} \tag{6.7}$$

where $y_n \sim \mathcal{N}(\pm E_s, \sigma^2)$. So $I_n$ is a random gaussian variable with mean $\mu_I = \pm\frac{2E_s}{\sigma^2}$ and with standard deviation $\sigma_I = \sqrt{\frac{4}{\sigma^2}}$. So as the $E_b/N_0$ ratio increases, $\sigma$ decreases, and thus the mean $\mu_I$ and the variance $\sigma_I^2$ of the intrinsic information increase, yielding the intrinsic information distribution to be clipped by the dynamic range constraints.

To circumvent this issue, 2 cases are distinguished hereafter, whether the channel is assumed to be known ($\sigma$ estimation) or not:

- *The channel parameter $\sigma$ is unknown:* the intrinsic information is then computed using a constant assumed noise-level. In (MacKay and Hesketh 2003), the authors studied the floating point performance of an assumed noise level versus the actual one. Equation (6.1) is then replaced by:

  $$I_n = Ay_n \tag{6.8}$$

  The intrinsic information mean is then a constant: $\mu_I = \pm E_s A$ and the standard deviation is decreasing when the noise level increases: $\sigma_I = A^2\sigma^2$. Figure 6.9 depicts the different performance obtained for several assumed noise level, for 2 dynamic range values. They are compared to the classical intrinsic information computing both in fixed point and floating point simulations. Note that for a fixed number of iterations, an optimization can be performed to get the best performance. This performance is achieved on a wide range of noise level.

- *The channel parameter $\sigma$ is known:* the intrinsic information is then modified so as to lower the increase of both its mean and its variance. The idea is to divide the channel output by $\sigma$ instead of $\sigma^2$ and to add a multiplication factor $A$. Equation (6.1) is then replaced by:

  $$I_n = \frac{Ay}{\sigma} \tag{6.9}$$

  This yields to have a lower mean $\mu_I = \frac{\pm AE_s}{\sigma}$ and a constant standard deviation $\sigma_I = \sqrt{A^2}$ when the noise power decreases. Figure 6.10 depicts the results obtained for several values of $A$ and a comparison with the other classical schemes. The intrinsic information is computed inside the decoder. We can notice that the error floor can be lowered by only changing the intrinsic calculation, but at the expense of a loss in BER performance in the waterfall region.

(c) Frédéric Guilloud, Télécom Paris - July 2004

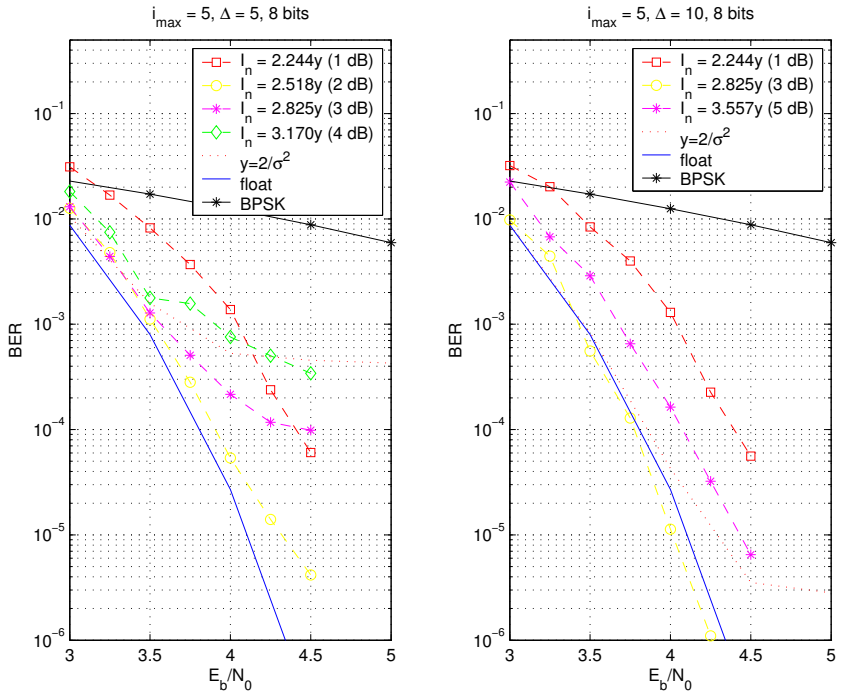



Figure 6.9: The intrinsic information is computed assuming a constant noise power: $I_n = ay$. Comparison between 4 values of $a$ and between the usual intrinsic calculation both in fixed point and floating point coding. All the fixed point are 8 bits coded, with $\Delta = 5$ or 10. The decoding is processed within 5 iterations max.

(c) Frédéric Guilloud, Télécom Paris - July 2004

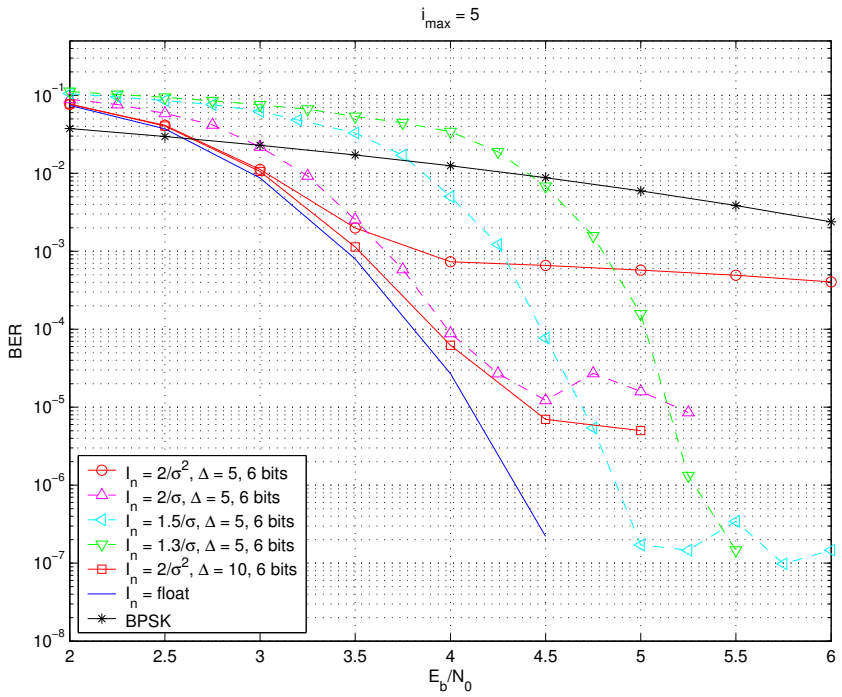



Figure 6.10: The intrinsic information is computed to have a constant standard deviation: $I_n = A^2$. It is simulated for $i_{\max} = 5$, $\Delta = 5$ and 6 bits for coding the data.

(c) Frédéric Guilloud, Télécom Paris - July 2004

## 6.4 conclusion

In this chapter, we have presented an overview of the LDPC platform. It consist of a generic decoder implementation running on an FPGA and of some software for performing fixed-point simulations. The platform has been used to perform LDPC simulations. We have been able to study the influence of the quantization on the decoding algorithms. A comparison between the BP and the $\lambda-$min algorithms has been proposed, as well as a study on the way the intrinsic information processing can modify the BER.

Some additional work should be done to complete the hardware simulation platform of LDPC codes:

- to develop a tool to perform the preprocessing of the matrices to download into the hardware decoder;
- to implement the LDPC encoder;
- to achieve a friendly user interface; the access to the platform using the WEB can also be done.

Moreover, the platform has been designed based on the LDPC state of the art of year 2002. It would be interesting to extend the possibilities of the platform. For example, the node processors could perform the A-min algorithm. In addition, on the system level point of view, the new schedule developed in section 3.4.3 and 3.4.2 could be also implemented.

In the future, it will be possible to run intensive and fast simulations, in order to optimize finite length LDPC codes. A first comparison has shown that hardware simulations are roughly $10^3$ times faster than software simulations performed on a PC Pentium III - 1.7 GHz running under Linux (Debian/1.0.0-0.woody.1).

(c) Frédéric Guilloud, Télécom Paris - July 2004

This page intentionally left blank.

# Conclusion and Perspectives

The low-density parity-check codes (LDPC) were discovered in the early sixties by R. Gallager. They have then been largely forgotten until their rediscovery in the mid-nineties. Nowadays, LDPC codes are, with the turbo-codes, the best performing channel codes, as they are almost reaching the Shannon limit.

LDPC codes encompass a large family of error correcting codes defined by a sparse parity-check matrix. These families are defined by a wide range of parameters. The versatility of LDPC codes enables to perform optimizations on these parameters so as to design a class of codes which can fit different channel specifications. For example, LDPC codes have been designed for optical communications, magnetic storage, multi-input and multi-output channels, and of satellite transmission. LDPC codes have been chosen as a standard for the DVB-S2 protocol, making a starting point to their take-off in the industry. Thus, the architectures of LDPC decoders are now a hot topic in the field of algorithm-architecture adequation. In this thesis, three important milestones have been reached in the design of LDPC decoders.

Our first contribution is the layout of a new original formal framework for the description of LDPC decoders. Researchers have published for about four years now solutions and examples of LDPC decoders. We proposed to classify the architectures of LDPC decoders as the combination of three entities:

1. the generic message passing data flow, which is governed by the parallelism rate,
2. the generic node processor data flow,
3. the control mode for scheduling the message transfers in the decoder.

In addition, we proposed a set of three parameters which quantify the complexity of an LDPC decoder architecture. This framework made also possible the design of new original solutions which are patent pending.

From a chronological point of view, the analysis of the state of the art and of the complexity of LDPC decoders shows that the first decoders solutions were focused on the


(c) Frédéric Guilloud, Télécom Paris - July 2004

simplification of the shuffle network, leading the researchers to construct LDPC codes with such properties. Then the memory requirement issue has been addressed, and solutions such as new schedules have been found.

Our second contribution is therefore a sub-optimal algorithm named the $\lambda-$min algorithm. It circumvents both the complexity processing issue and the memory requirements issue. Without any significant performance loss, this algorithm enables a reduction of memory up to 75% for high rate codes, compared to the optimal algorithm. An original architecture for the check node processors implementing the $\lambda-$min algorithm has also been proposed. These results have been published and are patent pending.

The validation of new algorithms, and the simulations performed to analyse and to optimize LDPC codes can take a lot of time. The emergence of high capacity reconfigurable hardware is igniting a revolution in digital simulations. Exploiting the reconfigurability of FPGAs for example can enhance the simulations now performed on PCs.

Our third contribution is the design and the implementation of a generic architecture LDPC decoder, within a development tool platform based on FPGAs. The word *generic* means first that any LDPC code of a given length $N$ and of a given rate $R$ can be decoded. The second meaning is that the description of the architecture has been designed while keeping in mind that all the components could be changed. It means that other check node processors for example can be implemented, since the parameters and all the latencies are tunable. Moreover, we showed that when using FPGAs from the nowadays technology, this genericity does not bring much overhead compared to other optimized decoders. This decoder has been used first to validate the $\lambda-$min algorithm performance and its associated architecture. It is also intended to speed up the simulations of a wide variety of LDPC codes: hardware simulations are roughly $10^3$ times faster than software simulations performed on a PC Pentium III - 1.7 GHz running under Linux.

The work achieved in this thesis brings a new formal framework to LDPC decoders. A theoretical approach is now available to design LDPC decoders. The generic architecture will be adapted to the other kinds of schedules, and some more components will be added to be able to compare different decoding algorithms. In the future, we will use our LDPC platform as a very efficient tool to study the performance of LDPC codes. We will be able also to perform optimizations in order to find good finite length codes.

Note that only the case of binary LDPC codes have been addressed in this thesis. We plan also to study the non-binary codes having their alphabet in $GF(2^q)$. These codes promise to be more powerful, with even more challenging implementations issues.

(c) Frédéric Guilloud, Télécom Paris - July 2004

# Appendix A

# Minimum BER achievable by coded BPSK systems

According to (Gallager 1968), the BER of a distortion-capacity limit gives the minimum $E_b/N_0$ required for a communication having a given BER $p$, and a code rate $R$. Let $H_2$ denote the entropy of a binary symmetric source:

$$\begin{aligned} \mathcal{R}(p,R) &= R(1-H_2(p)) && \text{(A.1)} \\ &= R(1+p\log_2(p)+(1-p)\log_2(1-p)) && \text{(A.2)} \end{aligned}$$

Then solve:

$$\mathcal{R}(p,R) = C(R\frac{E_b}{N_0}) \qquad \text{(A.3)}$$

where $C$ denotes the capacity of the binary input AWGN channel. The results are plotted on figure A.2 where the blue curves represent the minimum bit error probabiliies for the binary input additive white Gaussian noise output. The red curves are for the AWGN channel. It is to be noted that the AWGN input perform better than the binary input modulation. The difference is more and more obvious as the rate is increasing. The binary-input curve is moving toward the BPSK curve, whereas the AWGN curve is moving toward around 1.7 dB.

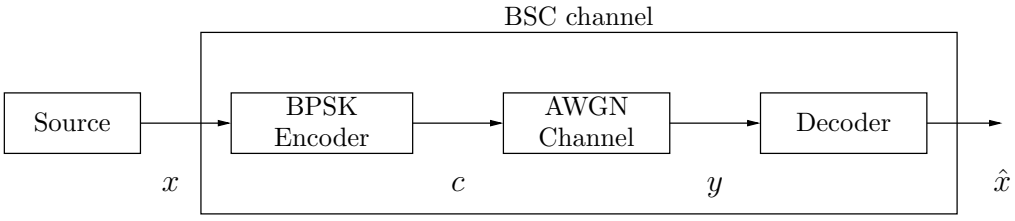



Figure A.1: The binary-input gaussian-output channel (Bi-AWGN channel)


(c) Frédéric Guilloud, Télécom Paris - July 2004

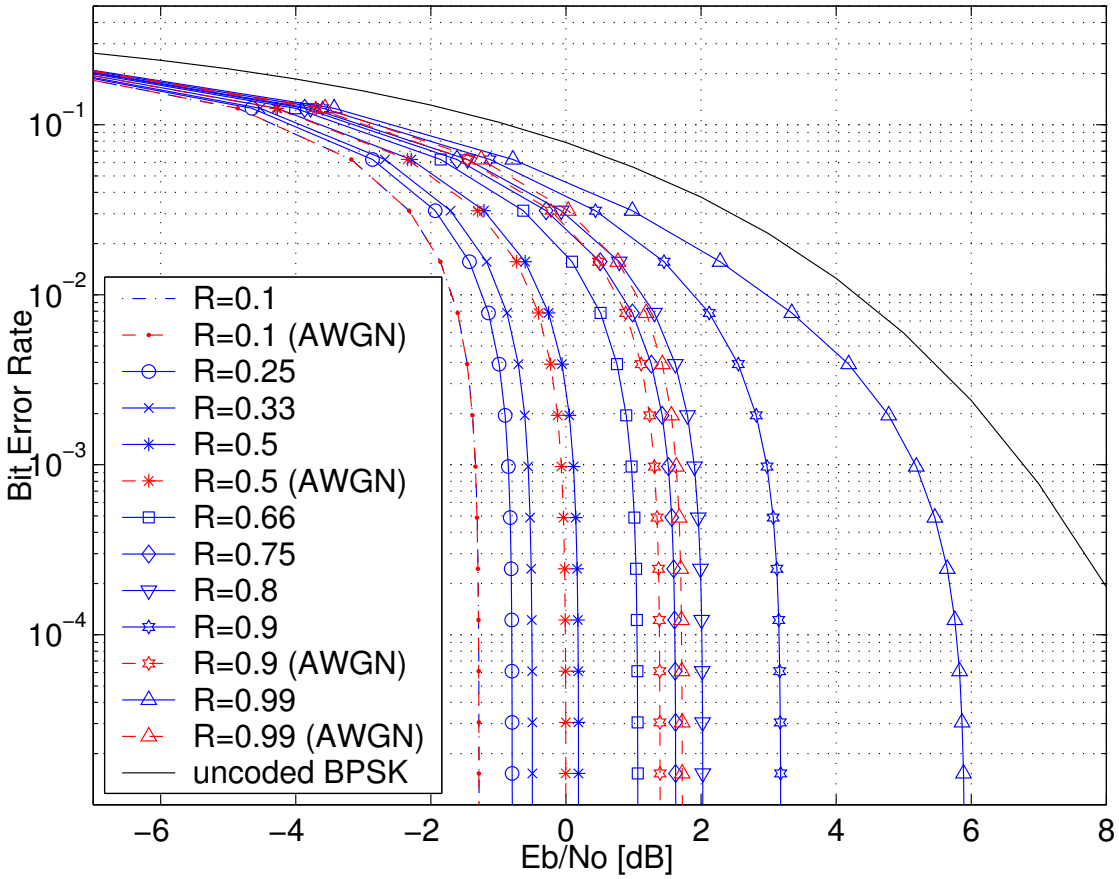



Figure A.2: Minimum bit error probability for coded BPSK modulations, when $E_b/N_0$ is lower than the minimum $E_b/N_0$ dictated by the Shannon bound

(c) Frédéric Guilloud, Télécom Paris - July 2004

# Appendix B

# Log Likelihood Ratios and Parity Checks

**Summary:**

*The aim of this annexe is to derive the expressions of the log likelihood ratio (LLR) of the modulo-2 sum of $n$ variables $c_i$: $\phi = c_1 \oplus c_2 \oplus \cdots \oplus c_n$. It is denoted by $\mathbb{L}(\phi)$ and it is defined by:*

$$\mathbb{L}(\phi) = \ln \frac{Pr(\phi = 0)}{Pr(\phi = 1)} \tag{B.1}$$

*In section B.1 we derive a 2-inputs core expression which can be applied iteratively to compute $\mathbb{L}(L\phi)$. In section B.2 a factorized expression for a direct processing of $\mathbb{L}(\phi)$ is derived.*

## B.1 Iterative expression

### B.1.1 2−variable rule

Let $\phi = c_1 \oplus c_2$ such as depicted in figure B.1. Let $p_1 = \Pr(c_1 = 0)$ and $p_2 = \Pr(c_2 = 0)$. Then we have:

$$\mathbb{L}(c_i) = \frac{\Pr(c_i = 0)}{\Pr(c_i = 1)} = \frac{p_i}{1 - p_i} \tag{B.2}$$

$$\text{and } p_i = \frac{e^{\mathbb{L}(c_i)}}{1 + e^{\mathbb{L}(c_i)}} \tag{B.3}$$


(c) Frédéric Guilloud, Télécom Paris - July 2004

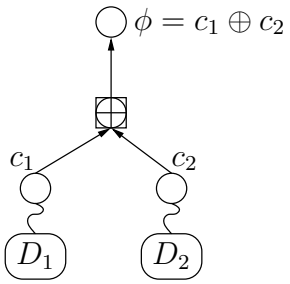



Figure B.1: An XOR operation on 2 bits $c_1$ and $c_2$. They depend on the set $D_1$ and $D_2$

So the LLR of $\phi$ is (Battail and El-Sherbini 1982):

$$\begin{aligned}
\mathbb{L}(\phi) &= \frac{\Pr(c_1=0)\Pr(c_2=0)+\Pr(c_1=1)\Pr(c_2=1)}{\Pr(c_1=1)\Pr(c_2=0)+\Pr(c_1=0)\Pr(c_2=1)} && \text{(B.4)}\\
&= \frac{p_1p_2+(1-p_1)(1-p_2)}{p_2(1-p_1)+p_1(1-p_2)} && \text{(B.5)}\\
&= \frac{1+\exp(\mathbb{L}(c_1)\,\mathbb{L}(c_2))}{\exp(\mathbb{L}(c_1))+\exp(\mathbb{L}(c_2))} \triangleq \mathbb{L}(c_1)\star\mathbb{L}(c_2) && \text{(B.6)}
\end{aligned}$$

Hence, $\mathbb{L}(c_1 \oplus c_2) = \mathbb{L}(c_1)\star\mathbb{L}(c_2)$, where $\star$ defines a binary operator on the LLR's value. Note that also the $\overline{\star}$ operator can be defined as (Hu et al. 2001):

$$\begin{aligned}
\mathbb{L}(c_2) &= \mathbb{L}(\phi \oplus c_1) = \mathbb{L}(\phi)\,\overline{\star}\,\mathbb{L}(c_1) && \text{(B.7)}\\
&= \ln\left|e^{\mathbb{L}(c_1)+\mathbb{L}(\phi)}-1\right| - \left|e^{\mathbb{L}(c_1)-\mathbb{L}(\phi)}-1\right| - \mathbb{L}(\phi) && \text{(B.8)}
\end{aligned}$$

### B.1.2 Hardware efficient implementation

Equation (B.6) can be derived so as to be put into a more efficient form for implementation: In order to simplify the notations, let

$$a = \mathbb{L}(c_1) \text{ and } b = \mathbb{L}(c_2) \tag{B.9}$$

and suppose without loss of generality that $a > b$. Then:

$$\begin{aligned}
\mathbb{L}(a \oplus b) &= \ln\left(\frac{1+e^{a+b}}{e^a+e^b}\right) && \text{(B.10)}\\
&= \ln\left(e^{-a}\frac{1+e^{a+b}}{1+e^{-(a-b)}}\right) && \text{(B.11)}\\
&= -a + \ln\left(\frac{1+e^{a+b}}{1+e^{-|a-b|}}\right) && \text{(B.12)}\\
&= -\text{sign}(a)\,|a| + \ln\left(\frac{1+e^{a+b}}{1+e^{-|a-b|}}\right) && \text{(B.13)}
\end{aligned}$$

There are two cases, depending on the sign of $(a+b)$:

- if $(a+b)<0$ then knowing that $(a>b)$ we have: $|a|<|b|$ and $b<0$; so:

$$\begin{aligned}\mathbb{L}(a\oplus b) &= -\text{sign}(a)|a| + \ln\left(\frac{1+e^{-|a+b|}}{1+e^{-|a-b|}}\right) &\text{(B.14)}\\ &= \text{sign}(b)\,\text{sign}(a)\min(|a|,|b|) \\ &\quad -\ln\left(1+e^{-|a-b|}\right)+\ln\left(1+e^{-|a+b|}\right) &\text{(B.15)}\end{aligned}$$

- if $(a+b)>0$ then knowing that $(a>b)$ we have: $|a|>|b|$ and $a>0$; so:

$$\begin{aligned}\mathbb{L}(a\oplus b) &= -a+\ln\left(e^{(a+b)}\frac{1+e^{-(a+b)}}{1+e^{-|a-b|}}\right) &\text{(B.16)}\\ &= b-\ln\left(1+e^{-|a-b|}\right)+\ln\left(1+e^{-|a+b|}\right) &\text{(B.17)}\\ &= \text{sign}(b)\,\text{sign}(a)\min(|a|,|b|) \\ &\quad -\ln\left(1+e^{-|a-b|}\right)+\ln\left(1+e^{-|a+b|}\right) &\text{(B.18)}\end{aligned}$$

Both of the two cases are then summed up by:

$$\mathbb{L}(a\oplus b)=\text{sign}(b)\,\text{sign}(a)\min(|a|,|b|)-\ln\left(1+e^{-|a-b|}\right)+\ln\left(1+e^{-|a+b|}\right) \quad \text{(B.19)}$$

Such an expression has also been derived in (Hu et al. 2001).

Architectures for operator $\star$ and $\overline{\star}$ are depicted on figure B.2. Note that the sign and the magnitude can also be processed separately.

### B.1.3 $n-$variable rule

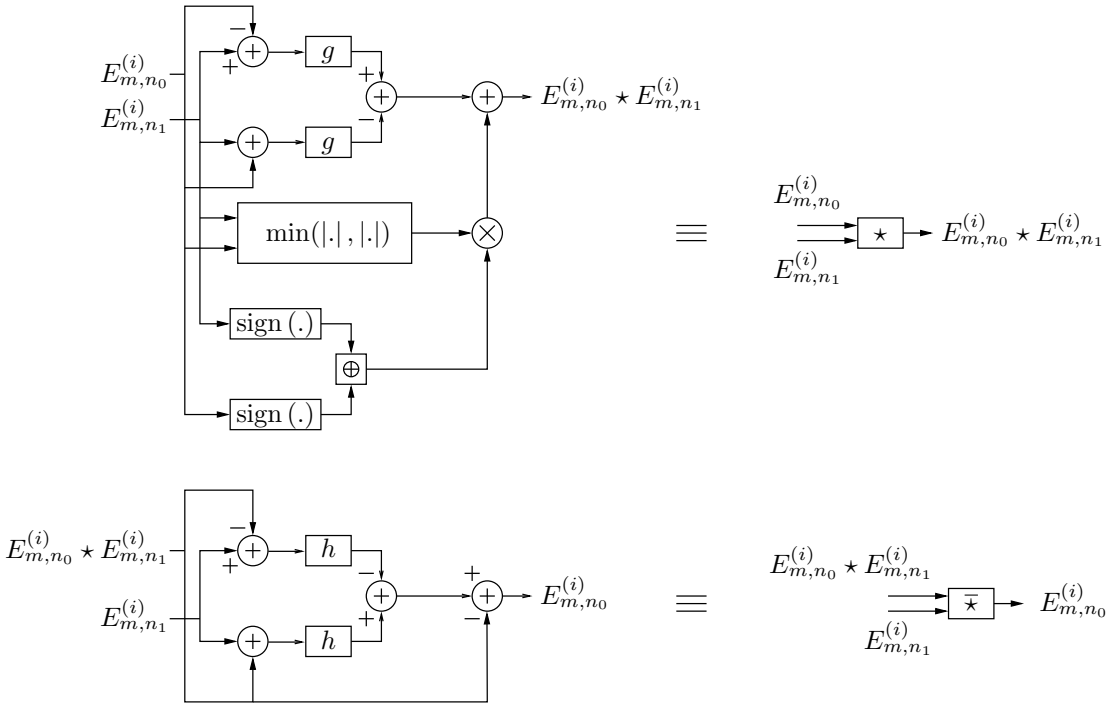
As shown on figure B.3, for parity-check constraints of length $n$, a serial processing can be applied to compute $\mathbb{L}(\phi)$, using the $\star$ operator defined in (B.6):

$$\begin{aligned}\mathbb{L}(\phi) &= \mathbb{L}(c_n)\star\mathbb{L}(\phi_n)\\ \mathbb{L}(\phi_n) &= \mathbb{L}(c_{n-1})\star\mathbb{L}(\phi_{n-1})\\ &\cdots\\ \mathbb{L}(\phi_3) &= \mathbb{L}(c_1)\star\mathbb{L}(c_2)\end{aligned}$$

Hence, the LLR of the parity check $\phi$ is the star operator applied between all the LLR of each variable of the parity:

$$\mathbb{L}(\phi)=\underset{1\le i\le N}{\text{llr}}\left(\mathbb{L}(c_i)\right)=\mathbb{L}(c_1)\star\mathbb{L}(c_2)\star\cdots\star\mathbb{L}(c_n) \quad \text{(B.20)}$$

(c) Frédéric Guilloud, Télécom Paris - July 2004




with $g(x) = \ln\left(1 + e^{-|x|}\right)$ and $h(x) = \ln|e^x - 1|$

Figure B.2: Possible architecture for the 2−input LLR operator defined by (B.19) and its symbol representation.

(c) Frédéric Guilloud, Télécom Paris - July 2004

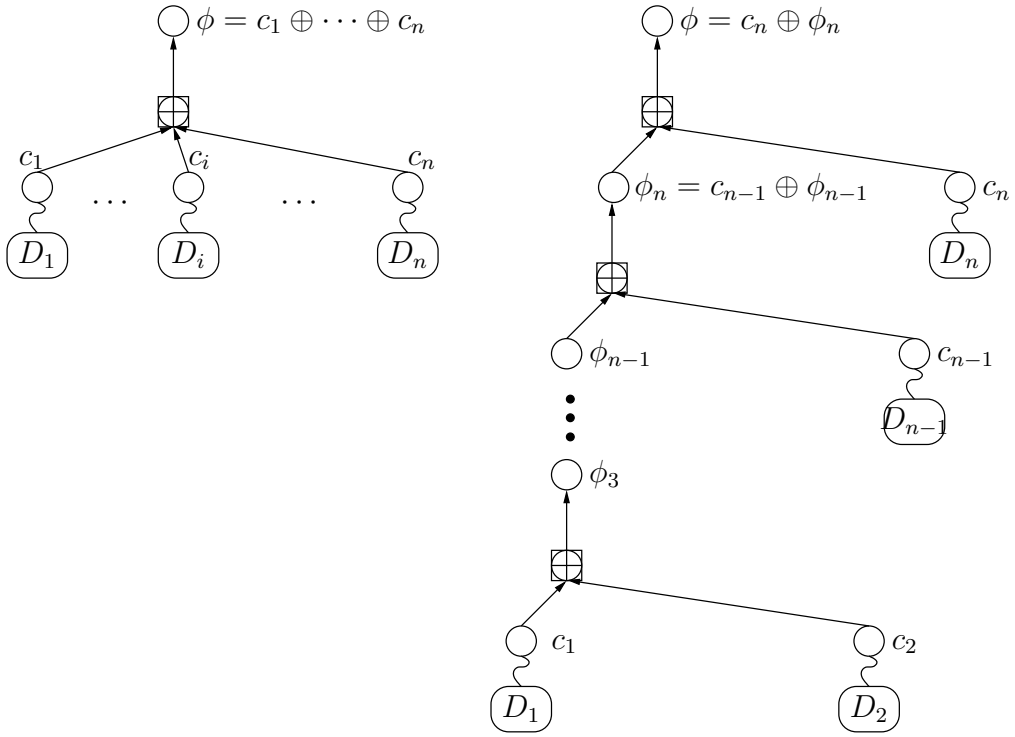



Figure B.3: An XOR operation on $n$ bits $c_i$. They depend on a set $D = \bigcup_{i=1}^{n} D_i$. The $n-1$ XOR operations can be done serially using only 2−inputs XOR (left side).

## B.2 Expression of the LLR using the tanh rule

### B.2.1 2−variable rule

Let $\phi = c_1 \oplus c_2$ be a parity check equation where $c_1$ and $c_2$ are 2 binary variables depending on others variables (resp. $D_1$ and $D_2$), such as illustrated by figure B.1. Let also $p_\phi = \Pr(\phi = 1|D_1, D_2)$. Then:

$$\tanh \frac{1}{2} \log \frac{\Pr(\phi = 0|D_1, D_2)}{\Pr(\phi = 1|D_1, D_2)} = \tanh \frac{1}{2} \log \frac{1 - p_\phi}{p_\phi} \tag{B.21}$$

Using $\tanh(x) = \dfrac{\exp(2x) - 1}{\exp(2x) + 1}$, (B.21) yields to:

$$\tanh \frac{1}{2} \log \frac{\Pr(\phi = 0|D_1, D_2)}{\Pr(\phi = 1|D_1, D_2)} = 1 - 2p_\phi \tag{B.22}$$

Let $p_1 = \Pr(c_1 = 1|D_1, D_2)$ and $p_2 = \Pr(c_2 = 1|D_1, D_2)$. Then:

$$p_\phi = p_1(1 - p_2) + (1 - p_1)p_2 \tag{B.23}$$

because $\phi = 1 \Leftrightarrow (c_1 = 0 \text{ AND } c_2 = 1) \text{ OR } (c_1 = 1 \text{ AND } c_2 = 0)$. Using (B.23) in (B.22) yields:

$$\begin{aligned}
\tanh \frac{1}{2} \log \frac{\Pr(\phi = 0|D_1, D_2)}{\Pr(\phi = 1|D_1, D_2)} &= 1 - 2p_1(1 - p_2) - 2(1 - p_1)p_2 && \text{(B.24)} \\
&= (1 - 2p_1)(1 - 2p_2) && \text{(B.25)} \\
&= \tanh \frac{1}{2} \log \frac{\Pr(c_1 = 0|D_1, D_2)}{\Pr(c_1 = 1|D_1, D_2)} && \text{(B.26)} \\
&\quad \times \tanh \frac{1}{2} \log \frac{\Pr(c_2 = 0|D_1, D_2)}{\Pr(c_2 = 1|D_1, D_2)} && \text{(B.27)}
\end{aligned}$$

### B.2.2 $n$−variable rule

The recursion is easy for proving the general LLR on a modulo−2 sum of $n$ variables $c_i$:

$$\tanh \frac{1}{2} \log \frac{\Pr(\phi = 0|D)}{\Pr(\phi = 1|D)} = \prod_{i=1}^{n} \tanh \frac{1}{2} \log \frac{\Pr(c_i = 0|D_i)}{\Pr(c_i = 1|D_i)} \tag{B.28}$$

which can be written:

$$\log \frac{\Pr(\phi = 0|D)}{\Pr(\phi = 1|D)} = 2 \tanh^{-1} \prod_{i=1}^{n} \tanh \left( \frac{1}{2} \log \frac{\Pr(c_i = 0|D_i)}{\Pr(c_i = 1|D_i)} \right) \tag{B.29}$$

or equivalently:

$$\log \frac{\Pr(\phi = 1|D)}{\Pr(\phi = 0|D)} = -2 \tanh^{-1} \prod_{i=1}^{n} \tanh \left( -\frac{1}{2} \log \frac{\Pr(c_i = 1|D_i)}{\Pr(c_i = 0|D_i)} \right) \tag{B.30}$$

(c) Frédéric Guilloud, Télécom Paris - July 2004

# Appendix C

# Architecture of the node processors

## C.1 Check Node Unit (CNU)

The CNU is a component which features $P$ Parity Check Processor (PCP). They are controlled by the same signals, generated in the CNU. Each PCP is decomposed as described in section 4.4.1.

The PCP components are depicted on figure C.1. The first synthesis block is denoted `SYNTH_R` for read, and the second one is denoted `SYNTH_W` for write. 2 dual port RAM (`dpram`) $L$ and $P$ are implemented in the PCP: the $L$ memory is used to save the $\lambda + 1$ results of the LLR computations, and the $P$ memory is used to save the $k$ pointers to one of this result and the sign of the new value. In fact the pointer to the new value is only coded one bit: the signal `is_default`. If the signal is set to 1, the variable is not in the set $\mathcal{N}_\lambda(m)$. Else, the address of the L-memory is incremented, because the $\lambda + 1$ LLR results are saved following their time of arrival.

The waveforms depicted on figure C.2 enables to define the latencies of the different components. They are denoted by a name beginning with `D_*` There is no problem in pipelining the different parity-check size if they are processed by ascending order of their weight and if the calculation time is lower than the one of the variable reading, which takes $k_i$ clock cycles.


(c) Frédéric Guilloud, Télécom Paris - July 2004

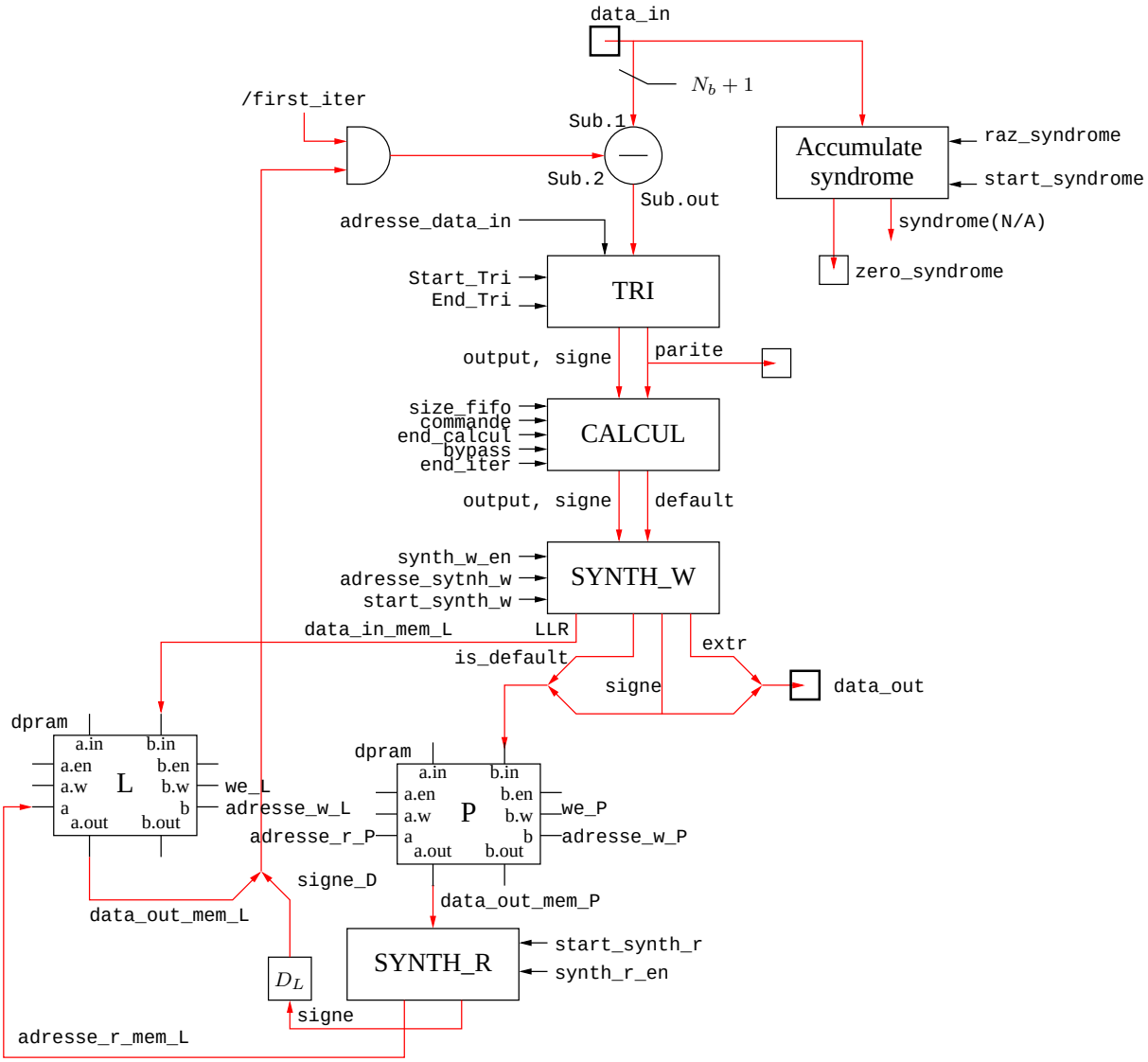



Figure C.1: Parity Check Processor

(c) Frédéric Guilloud, Télécom Paris - July 2004

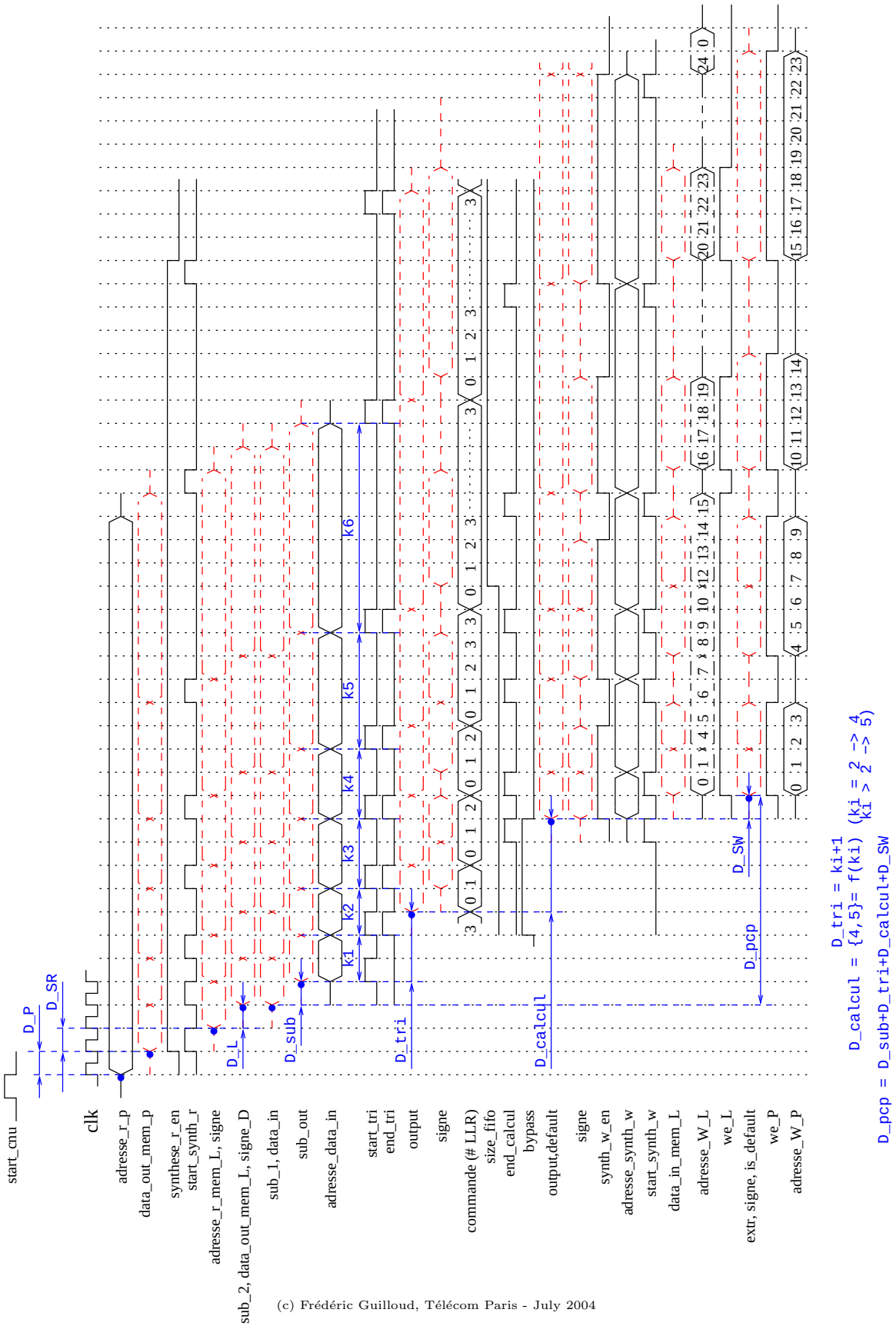



Figure C.2: Waves of the PCP component

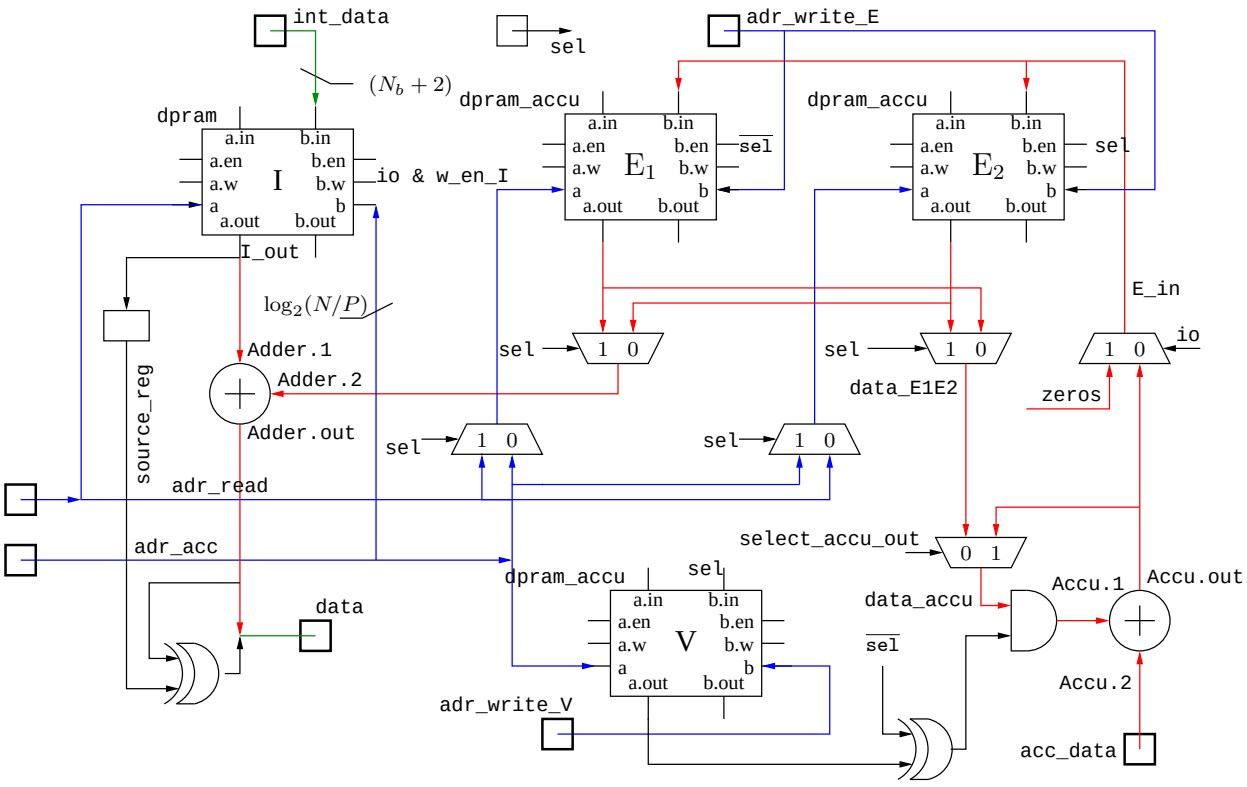



Figure C.3: Variable Processor

## C.2 Variable Node Unit (VNU)

Similarly to the CNU, the VNU features $P$ Variables Processor (VP), which are data paths components, controlled by the same signals generated in the VNU. The VNU also features the A-memory, which contains the parity check matrix. Figure C.3 depicts the architecture of the VP, which is mainly made of memories. Figure C.4 and C.5 depict the waveforms of the signals in the VP and their control respectively during the input/output phase, and during the decoding phase. The architecture of the VP components implements the data path behaviour depicted on figure 5.8. This behaviour may yield 2 kinds of memory conflicts:

1. What if the read and write address of the dual port RAM are the same ? The 3 memories $V$ and alternatively $E_1$ and $E_2$ may be concerned by this event. An accumulation is made of a read and then a write access. If two pipelined accumulations are processed, the read access of the second one will append at the moment at the same time as the write access of the first one. So the second read access will not output the true value, which should have been modified by the first accumulation. Our solution is the `dpram_accu` component, depicted on figure C.6, which features

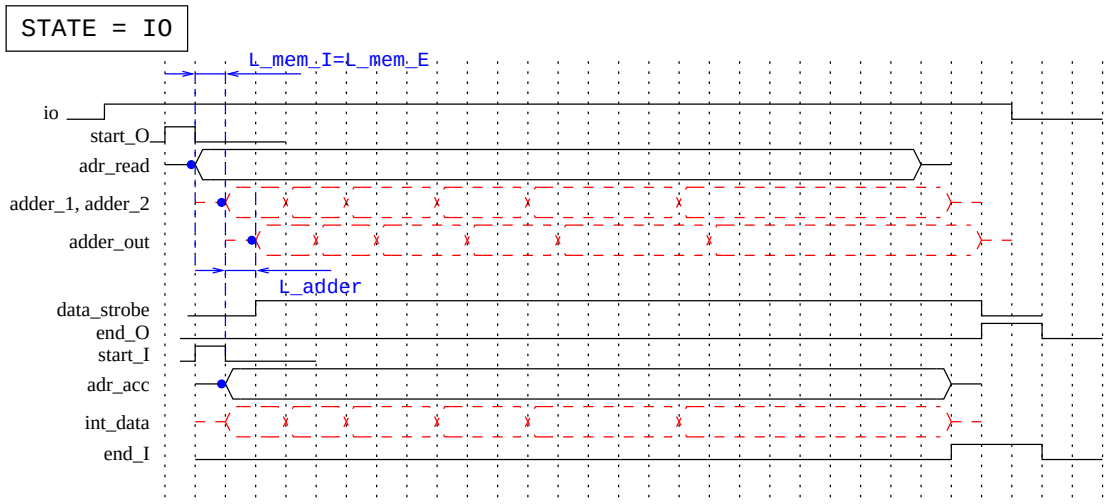



Figure C.4: Waves of the VP component for the `IO` state

a `dpram` component and the detection of such a conflict. The output is then either the `dpram` output (no conflict) or the `dpram` input (conflict). The waveforms corresponding to such a situation are depicted on figure C.7: the same variable is implied in 2 successive parity-check (the $n$-th one and then the $n + P$ one).

2. The second conflict is due to the latency of the accumulation, when a variable is accessed for reading in the $n + P$-th parity-check before the accumulation of this same variable has been processed concerning the $n$-th parity-check. The waveforms corresponding to this situation are depicted on figure C.8. This conflict concerns only the extrinsic memory which is used for the accumulation: alternately $E_1$ and $E_2$. Our solution is the `select_accu_out` signal, which controls a multiplexer. This signal is set to 1 when 2 successive read addresses are identical. The only way this conflict may occurs is when a bit has to be accumulated twice successively. This conflict may occurs since the latency between the read and write part of the accumulation is as much as 2 clock cycles (one for the memory read, and one for the accumulation). *Of course, if this latency has to be changed, this problem will have to be reconsidered.*

(c) Frédéric Guilloud, Télécom Paris - July 2004

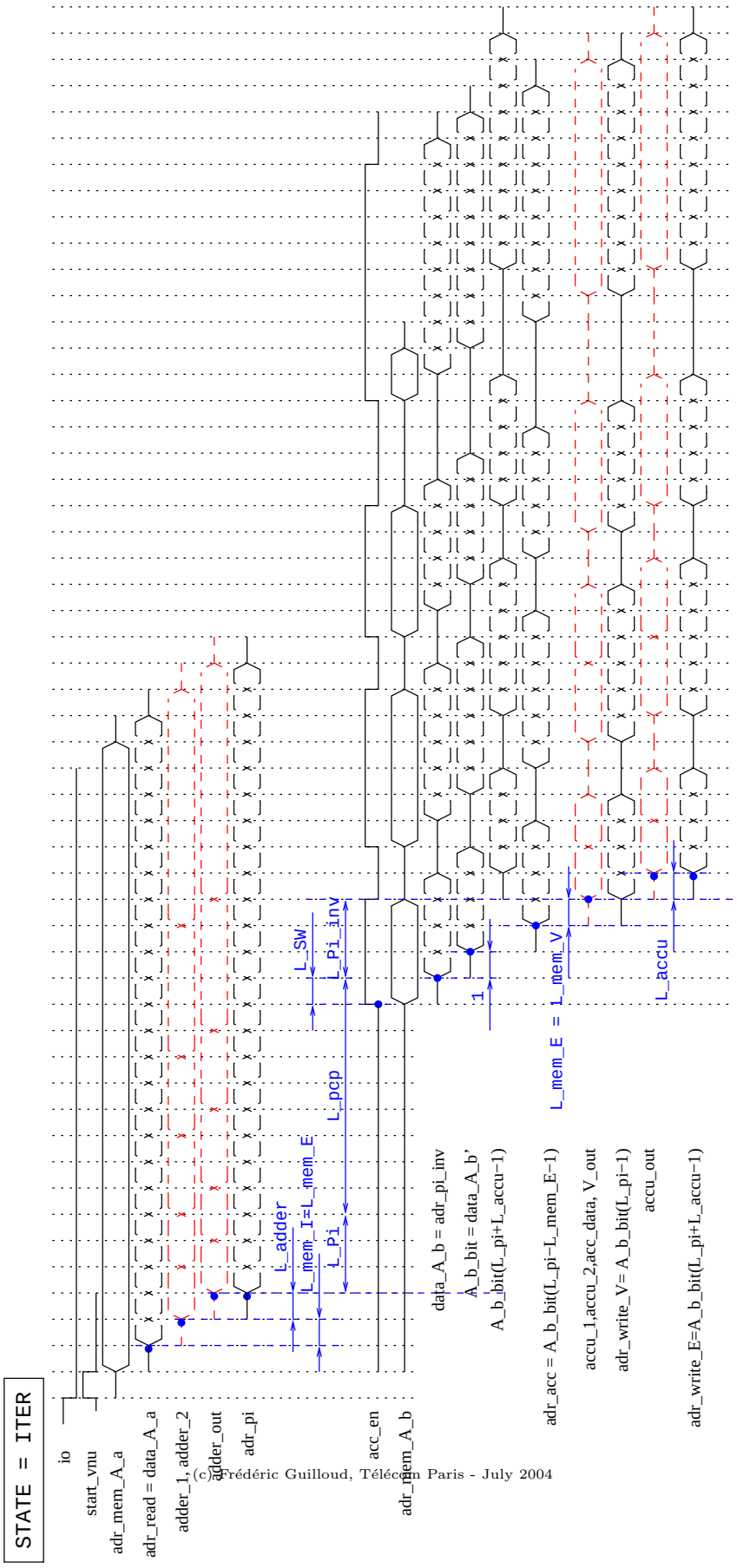



Figure C.5: Waves of the VP component for the `ITER` state

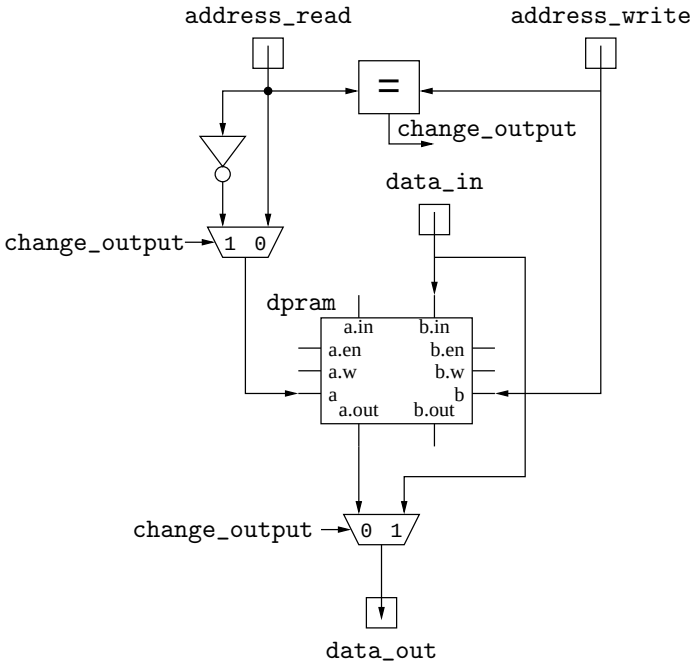



Figure C.6: The `dpram_accu` component

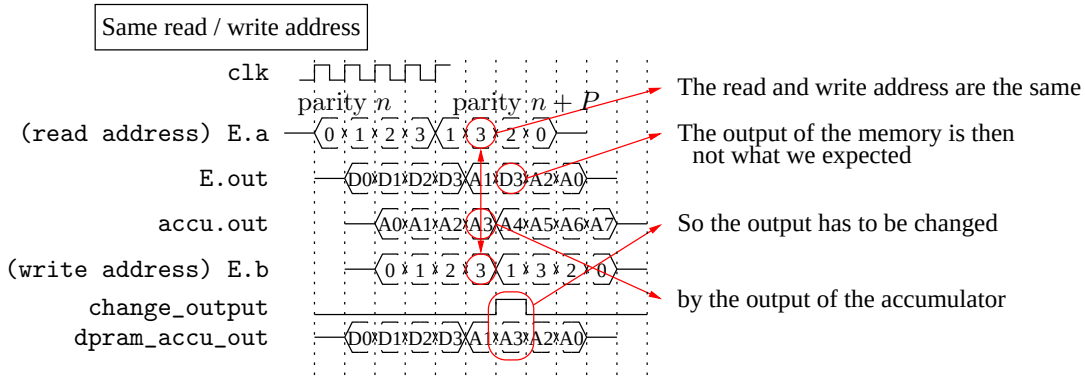



Figure C.7: Example of a memory conflict (1).

(c) Frédéric Guilloud, Télécom Paris - July 2004

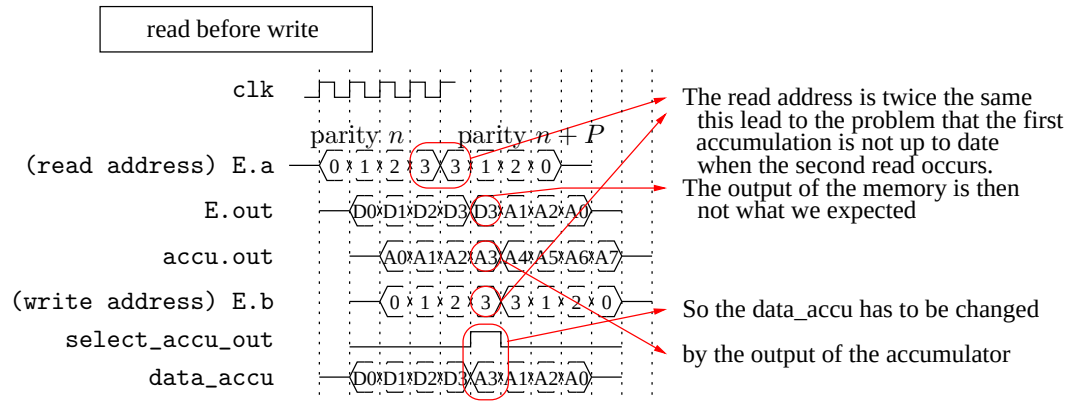



Figure C.8: Example of a memory conflict (2).

(c) Frédéric Guilloud, Télécom Paris - July 2004

# Appendix D

# Platform component architecture


(c) Frédéric Guilloud, Télécom Paris - July 2004

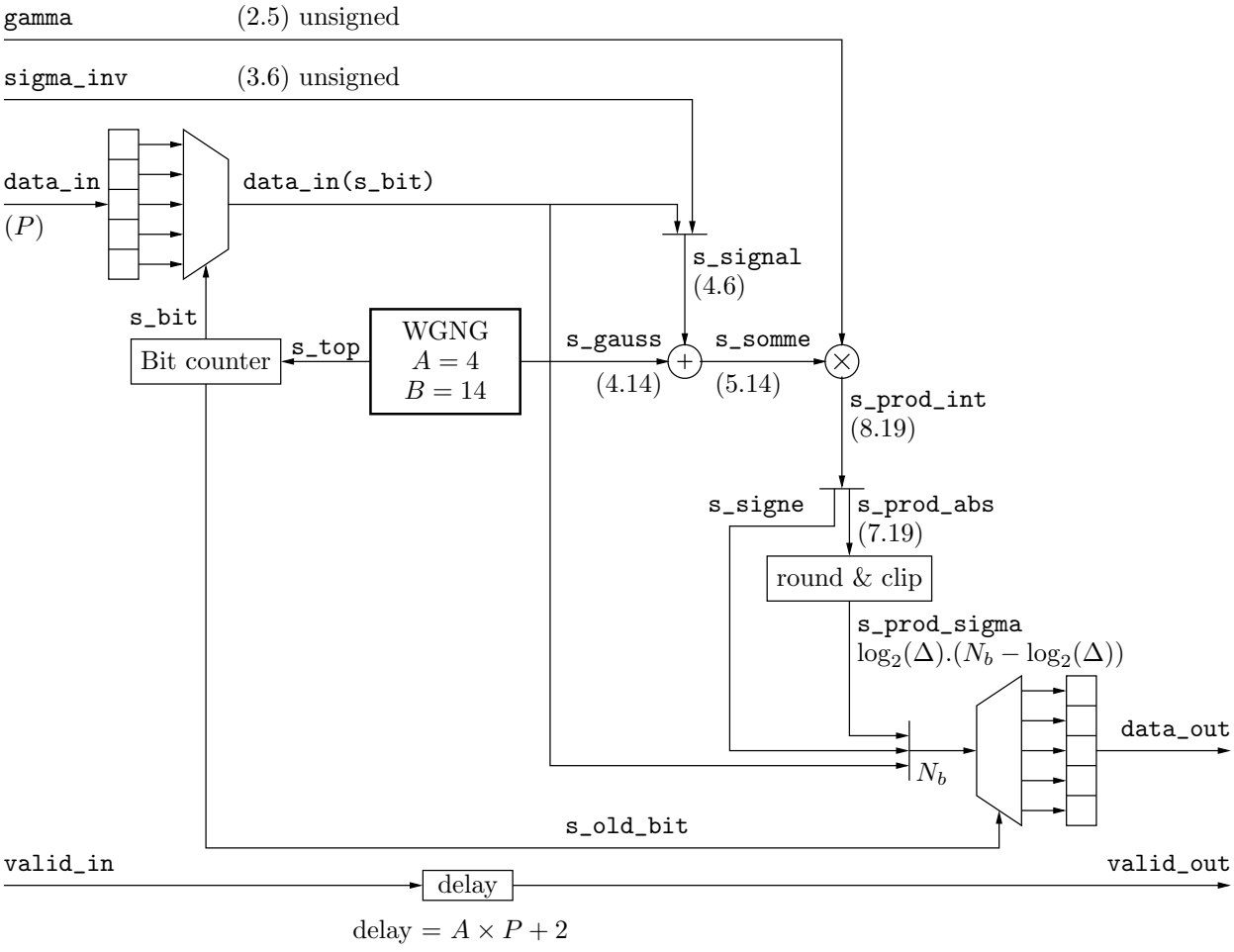



Figure D.1: Architecture of the channel emulator component.

(c) Frédéric Guilloud, Télécom Paris - July 2004

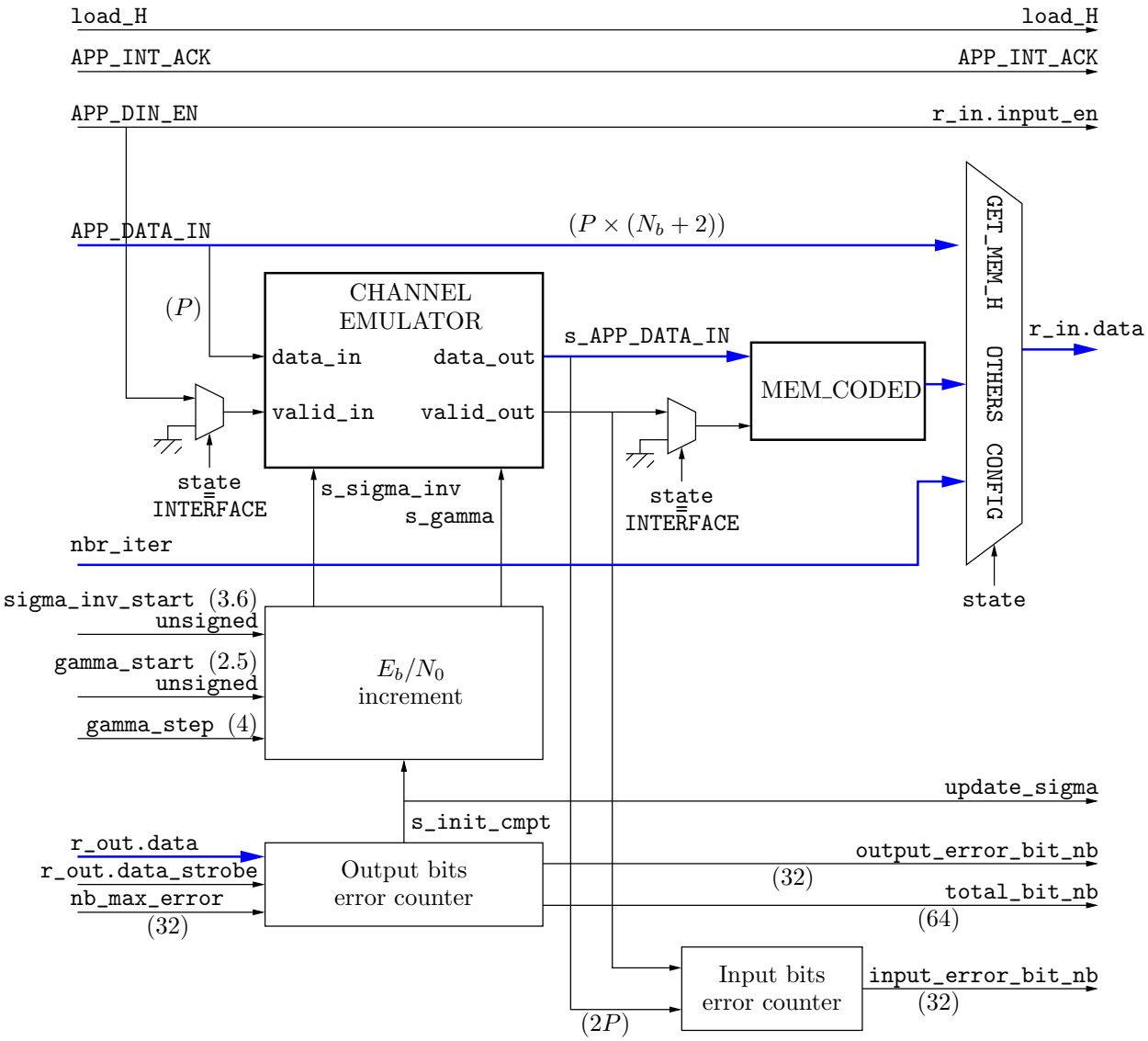



Figure D.2: Architecture of the `protocol` component.

This page intentionally left blank.

# Appendix E

# Listings


(c) Frédéric Guilloud, Télécom Paris - July 2004

Listing E.1: LDPC decoder VHDL package for constant definitions.

```
-- (c) Telecom Paris - 2004 --
-- ldpc_pack.vhd
-- LDPC decoder package

library IEEE;

use IEEE.Std_Logic_1164.ALL;
use work.log_pkg.all; -- enable to compute log_2(x)

package ldpc_pack is

  ----------------------------------------
  -- Constant declaration
  ----------------------------------------

  constant N         : natural := 800;   -- Codeword length
  constant M         : natural := 400;   -- Number of parity
  constant P         : natural := 5;     -- Edge rate
  constant Pmax      : natural := 8;     -- Edge rate max
  constant Nb        : natural := 6;     -- Number of bit for magnitude coding
  constant lambda    : natural := 3;     -- Number of minimum for the lambda-min
  constant kmax      : natural := 32;    -- Highest Parity degree
  constant Dynamique : natural := 6;     -- Dynamic range
```

Listing E.2: LDPC decoder VHDL package for constant definitions.

```
----------------------------------------
-- Latencies
----------------------------------------
constant L_mem_L : natural := 1;        -- Latency for L-memory (read)
constant L_mem_A : natural := 1;        -- Latency for A-memory "
constant L_mem_I : natural := 1;        -- Latency for I-memory
constant L_mem_E : natural := 1;        -- Latency for E-memory
constant L_mem_P : natural := 1;        -- Latency for P-memory
constant L_mem_V : natural := 1;        -- Latency for V-memory
constant L_SR    : natural := 1;       -- Latency for read synthesis
constant L_SW    : natural := 1;       -- Latency for write synthesis
constant L_C     : natural := 1;                -- Latency for LLR processing
constant L_sub   : natural := 1;        -- Latency for subtraction
constant L_adder : natural := 1;        -- Latency for adder
constant L_pi    : natural := 3;        -- Latency for shuffle
constant L_accu  : natural := 1;        -- Latency for accumulation
constant D1      : natural := L_mem_L;              -- delay P.out vs L.out
constant D2      : natural := L_mem_I+L_adder;      -- delay A.a.out vs Pi.a
constant D3      : natural := L_pi+L_accu;                -- delay E.a vs E.b
constant D4      : natural := L_mem_V;                 -- delay V.a vs V.b
constant D5      : natural := L_mem_A+L_mem_V+L_pi; -- delay V.out vs E.out
constant Max_FIFO_Parity_length : natural := 8;       -- FIFO size for pipelining
                                                  --- parity-check sizes in the CNU
```

Listing E.3: map summary for the synthesis 3400x1700x5.

```
Xilinx Mapping Report File for Design 'bloc_fpga_2'
Copyright (c) 1995-2000 Xilinx, Inc.  All rights reserved.

Design Information
------------------
Command Line : map -pr b bloc_fpga_2.ngd -o bloc_fpga_2.ncd bloc_fpga_2.pcf
Target Device  : xv1000e
Target Package : bg560
Target Speed   : -6
Mapper Version : virtexe -- D.27
Mapped Date    : Fri May 07 18:31:50 2004

Design Summary
--------------
   Number of errors:      0
   Number of warnings: 294
   Number of Slices:                3,112 out of 12,288   25%
   Number of Slices containing
      unrelated logic :                 0 out of  3,112    0%
   Number of Slice Flip Flops:      2,257 out of 24,576    9%
   Total Number 4 input LUTs:       4,854 out of 24,576   19%
      Number used as LUTs:                      4,716
      Number used as a route-thru:                138
   Number of bonded IOBs:             104 out of    404   25%
      IOB Flip Flops:                            91
   Number of Block RAMs:               88 out of     96   91%
   Number of GCLKs:                     3 out of      4   75%
   Number of GCLKIOBs:                  2 out of      4   50%
   Number of DLLs:                      3 out of      8   37%
Total equivalent gate count for design:  1,519,335
Additional JTAG gate count for IOBs: 5,088
```

# Appendix F

# Résumé étendu

**Summary:**

*This is a french extended summary of this thesis, as required by the publisher of this thesis.*

## Introduction

Il y a plus de dix ans maintenant, à la conférence internationale ICC 1993 sur les télécommunications, C. Berrou et A. Glavieux ont présenté un nouveau principe de décodage des codes de canal : les Turbocodes, et leur algorithme associé de décodage. Les Turbocodes ont permis d'obtenir des performances inégalées en terme de codes correcteurs d'erreurs, en s'approchant à près de quelques dixièmes de dB de la limite de Shannon, pour des taux d'erreurs binaires de l'ordre de $10^{-5}$.

En plus de l'impact qu'ont eu les Turbocodes sur les systèmes de télécommunication, ils ont également permis de se rendre compte que d'autres codes approchant la capacité existaient. C'est ainsi que les codes de parité à faible densité (LDPC, pour low-density parity-check) inventés au début des années soixante par Robert Gallager, ont été ressuscités au milieu des années quatre-vingt dix par David MacKay. Il se trouve en fait que lorsque les codes LDPC ont été inventés, leur décodage était trop complexe pour la technologie de l'époque, ce qui a conduit à ce que la communauté de chercheurs impliqués dans les codes correcteurs d'erreurs les oublie. Comme les Turbocodes, les codes LDPC peuvent avoir des performances très proches de la limite de Shannon, par un décodage itératif.

Un code LDPC est un code en bloc linéaire défini par une matrice de parité peu dense. Cela signifie qu'un très grand nombre d'entrées de la matrice est nul. L'algorithme de décodage est facile à comprendre grâce à la représentation graphique du code : chaque symbole du mot de code est représenté par un noeud appelé noeud variable, et chaque

 (c) Frédéric Guilloud, Télécom Paris - July 2004

contrainte de parité correspondant à chaque ligne de la matrice de parité est représentée par un noeud parité. Chaque noeud parité est relié aux noeuds variable correspondants aux symboles du mots de code impliqués dans cette parité. Pendant le processus de décodage, des *messages* sont échangés entre les noeuds variable et les noeuds parité le long de chaque connection. Ces messages sont en fait des images des probabilités qu'un symbole ou une contrainte ait une valeur donnée.

Les performances asymptotiques des codes LDPC qui sont maintenant bien comprises, font des LDPC des concurrents sérieux aux Turbocodes. De plus, le choix d'un code LDPC pour la future norme de transmission video par satellite (DVB-S2 pour Digital Video Broadcasting, Satellite) pose la problématique des architectures de décodeurs de codes LDPC. Bien que le décodage des codes LDPC soit relativement simple, la conception d'un décodeur de codes LDPC est elle plus délicate. Cette architecture peut être divisée en trois parties : les processeurs de noeuds (variables et parité), le réseau d'interconnection et la gestion de la mémoire. Les principales contributions de cette thèse se situent dans ces trois parties. Une première contribution est l'établissement d'un cadre global d'analyse et de synthèse des décodeurs LDPC. Notre deuxième contribution est un algorithme sous-optimal de décodage des codes LDPC, associé avec une architecture adaptée qui permet de diminuer la quantité de mémoire nécessaire dans le décodeur, sans apporter de perte significative aux performances de décodage. La troisième contribution de cette thèse est la conception d'une architecture générique de décodeur LDPC, testée et validée au sein d'une plateforme à base de circuits reconfigurables, permettant l'accélération des simulations.

## F.1 Codage de canal

Ce premier chapitre présente la problématique du décodage de canal et plus particulièrement du décodage optimal dans le cas des codes linéaires en bloc. Les notations principales utilisées dans la thèse sont tout d'abord présentées, et notamment celles liées à la représentation graphique des codes linéaires de bloc. On montre ensuite que si l'on fait l'hypothèse que cette représentation graphique ne contient aucun cycle, le décodage optimal peut être effectué en utilisant un algorithme itératif, que l'on nomme algorithme à propagation de croyance (BP, pour Belief Propagtion). Enfin, les performances des codes correcteurs d'erreurs sont rappelées, en précisant la notion de distance par rapport à la limite de Shannon.

## F.2 Les codes LDPC

Dans ce chapitre, nous proposons tout d'abord un bref rappel historique des codes LDPC. Les codes LDPC forment une famille très grande de codes définis par un grand nombre de paramètres. La conception des codes LDPC consiste à trouver un ensemble spécifique de

(c) Frédéric Guilloud, Télécom Paris - July 2004

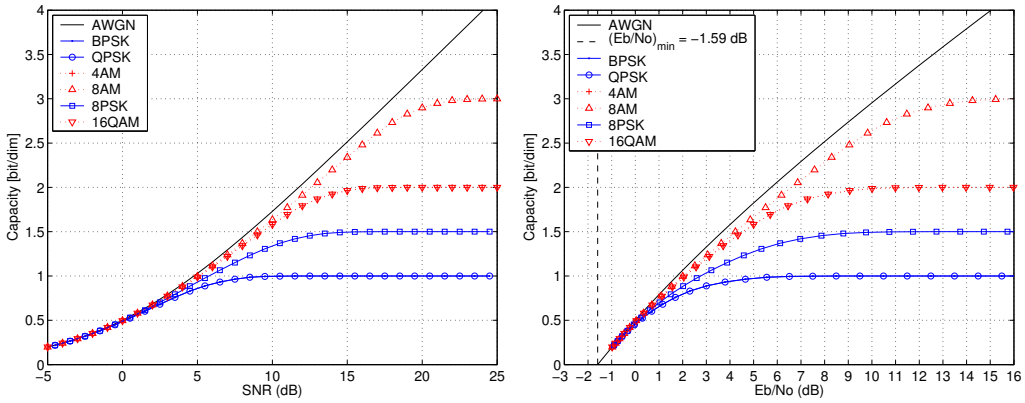



Figure F.1: Capacité du canal à bruit blanc additif gaussien en fonction de $E_b/N_0$ et du rapport signal sur bruit SNR$= 2R\frac{E_b}{N_0}$

paramètres qui conviennent bien à un système de communication donné, et notamment son canal. Lorsque l'ensemble des paramètres est obtenu, un code LDPC particulier de taille finie peut être obtenu à l'aide de constructions aléatoires ou déterministes. Les deux dernières sections de ce chapitre traitent des performances obtenues par un décodage itératif des codes LDPC, ainsi que du problème de séquencement.

## F.3 Analyse unifiée des décodeurs LDPC

Dans ce chapitre, un cadre unifié pour décrire et concevoir des architectures de décodeurs LDPC est proposé. Un ensemble de paramètres est défini pour caractériser le facteur de parallélisme. Les processeurs génériques de noeud (variable et parité) sont classifiés en fonction de leurs flux de données, de leur contrôle et de la position du réseau d'interconnection. La combinaison des différents paramètres associés à ces 3 entités permet de décrire un très grand nombre d'architectures de décodeur LDPC, pouvant utiliser des séquencement très performants. Une analyse de complexité est également proposée à l'aide de trois paramètres décrivant complètement la complexité d'une architecture de décodeur LDPC. Enfin, ce cadre formel d'étude des décodeurs LDPC est appliqué à trois exemples illustrant chacun un séquencement important : le séquencement classique (ou séquencement par inondation), un séquencement récemment proposé (mélange horizontal) et également une architecture nouvelle appliquée à un séquencement de type mélange vertical. Ce chapitre se termine par une description de l'état de l'art des architectures de décodeurs LDPC basée sur le cadre formel de description proposé dans ce chapitre.

(c) Frédéric Guilloud, Télécom Paris - July 2004

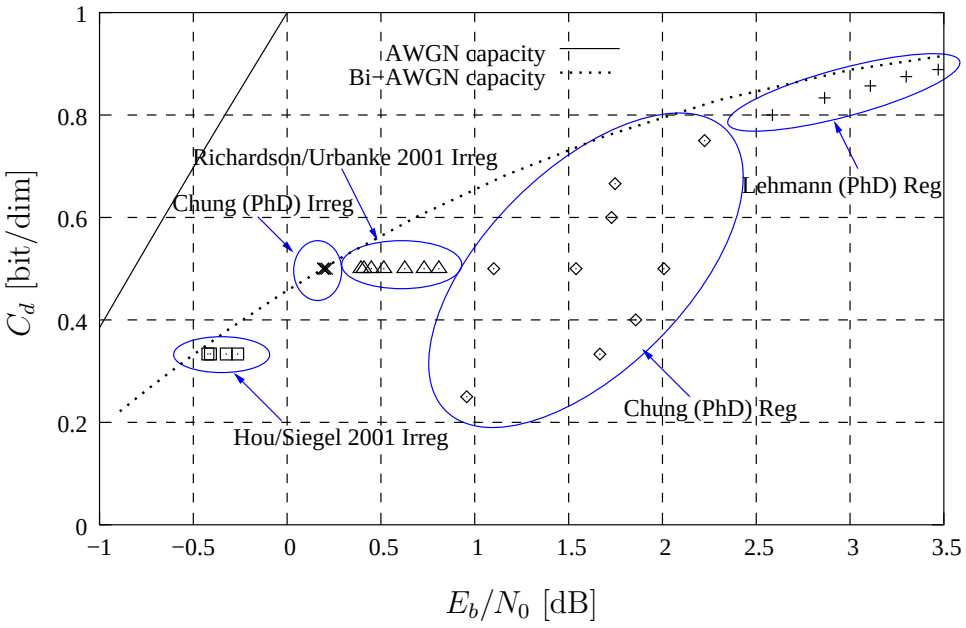



Figure F.2: Les seuils de convergence de différents codes LDPC réguliers ou irréguliers pour différents rendement, dans le cas d'une modulation BPSK

## F.4 L'algorithme $\lambda-$Min

Dans le chapitre précédent, les architectures de décodeur de codes LDPC basées sur les algorithmes à propagation de croyance ont été étudiées. Il est apparu que malgré les différents séquencements possibles, la quantité de mémoire exigée pour de tels algorithmes est très importante parce que toute l'information associée aux connections entre les noeuds (messages variables $\leftrightarrow$ parités) doit être sauvegardée jusqu'à la prochaine itération. Quelques simplifications peuvent être également apportées pour diminuer la complexité de l'algorithme de propagation de croyance, et en particulier la complexité du traitement effectué dans les processeurs de noeud de contrôle. Dans ce cas, un compromis doit être fait entre les simplifications de l'algorithme permettant de diminuer la complexité des processeurs, et la perte de performance résultante. Dans ce chapitre, nous proposons un nouvel algorithme, appelé algorithme $\lambda-$min, comme étant un compromis performance-complexité entre l'algorithme à propagation de croyance et l'algorithme BP-Based. Les résultats de simulation prouvent que de bonnes performances peuvent être obtenues. Une amélioration peut même être apportée par l'ajout de coefficients correctifs. L'architecture des processeurs de noeud de parité mettant en oeuvre l'algorithme de $\lambda-$min est également étudiée. Les résultats de ce chapitre ont été publié en partie dans (Guilloud, Boutillon, and Danger 2003b) et en partie dans (Guilloud, Boutillon, and Danger 2003a) et font l'objet d'une demande de dépôt de brevet aux Etats-Unis (Boutillon,

(c) Frédéric Guilloud, Télécom Paris - July 2004

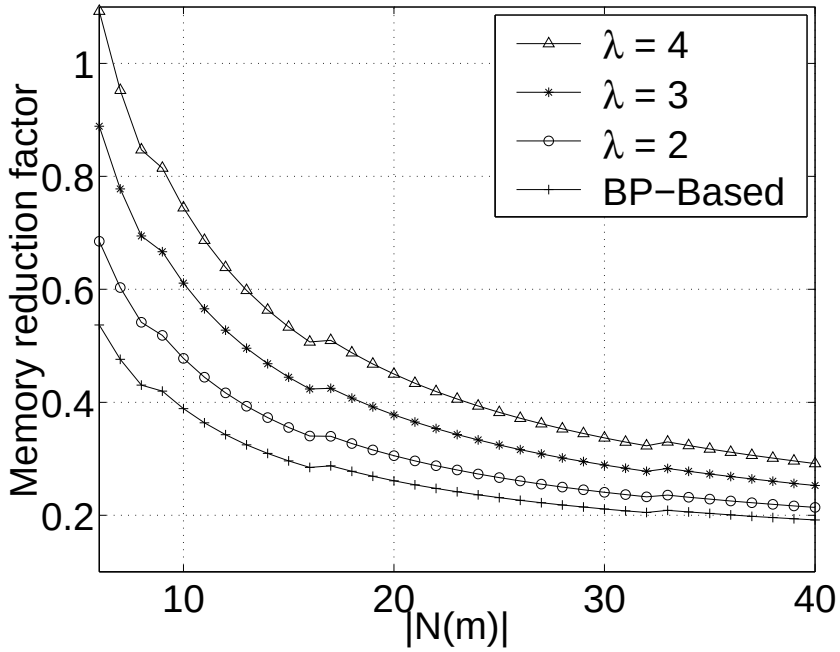



Figure F.3: Gain mémoire entre l'algorithme $\lambda-$min et l'algorithme BP, en fonction du degré de la parité $|N(m)|$ pour plusieurs valeur de $\lambda$, les données étant codées sur $N_b = 5$ bits

Tousch, and Guilloud 2003).

## F.5 Architecture générique de décodeur LDPC

Les simulations permettant d'étudier le comportement des codes LDPC prennent du temps. Le temps nécessaire à la simulation est multiplié par 10 quand le taux d'erreur binaire cible est divisé par 10. Dans ce chapitre, une architecture générique de décodeur LDPC est présentée. Cette architecture est implantée dans un circuit reconfigurable de type FPGA afin de décoder différents codes LDPC et de comparer leurs performances plus rapidement que sur un PC. Ce décodeur générique permet également de comparer différentes réalisations d'architecture de quelques blocs spécifiques entre elles, telles que celle des processeurs de parités par exemple. Enfin une comparaison entre différents algorithmes et/ou séquencements devrait être possible. Ainsi l'aspect générique est une caractéristique importante de cette conception. Le VHDL a été choisie pour décrire l'architecture. Après une vue d'ensemble de l'architecture de décodeur, la gestion des différentes mémoires est détaillée. Ensuite le problème d'interconnection est discuté et l'architecture du bloc de plus haut niveau est décrite. Une description plus détaillée des composants principaux de l'architecture est donnée en annexe C.

(c) Frédéric Guilloud, Télécom Paris - July 2004

## F.6 La plateforme de simulation

Ce dernier chapitre présente dans une première partie une vue d'ensemble de la plateforme conçue pour faire tourner des simulations intensives de décodage LDPC en utilisant le décodeur générique décrit dans le chapitre 5. Quelques commentaires liés à la synthèse sont alors présentés. Dans une deuxième partie, des simulations en virgule fixe sont présentées. Elles permettent d'étudier l'influence de la quantification sur les algorithmes de décodage. Une comparaison entre simulations logicielles et simulation matérielles ne montre aucune différence entre elles. Une comparaison entre l'algorithme à propagation de croyance et les algorithmes $\lambda-$min est présentée, et l'influence du calcul de l'information intrinsèque est soulignée. Les résultats de la deuxième partie ont été en partie présentés lors de la réunion du GDR-ISIS sur les codes LDPC (Guilloud, Boutillon, and Danger 2002b).

## Conclusion

Les codes LDPC forment une vaste famille de codes correcteurs d'erreurs. Ils sont définis à l'aide d'une matrice de parité peu dense. Ces codes possèdent un grand nombre de paramètres, qui peuvent être optimisés pour s'adapter à de nombreux canaux. Cette polyvalence offrent de nombreuses perspectives d'utilisation ; par exemple, des codes LDPC ont été conçus pour des communications optiques, du stockage magnétique, des canaux à entrées multiples et à sorties multiples et de la transmission par satellite. Un code LDPC a d'ailleurs été choisi pour la norme DVB-S2, constituant ainsi un point de départ à l'expression de leur potentiel dans des utilisations industrielles. Ainsi, les architectures des décodeurs LDPC deviennent maintenant un sujet d'actualité dans le domaine de l'adéquation algorithme-architecture. Dans cette thèse, trois étapes importantes ont été atteintes dans la conception des décodeurs de codes LDPC.

Notre première contribution est la mise à disposition d'un nouveau cadre formel original pour la description des décodeurs LDPC. Depuis environ quatre années maintenant, la communauté scientifique publie des solutions et des exemples de décodeurs LDPC. Nous avons proposé un cadre permettant de classifier les architectures des décodeurs LDPC comme résultant de la combinaison de trois entités:

1. une architecture générique de propagation de messages, spécifiée notamment par le taux de parallélisme,

2. une architecture de chemin de données pour les processeurs génériques de noeuds,

3. un mode de contrôle pour le séquencement de la propagation des messages dans le décodeur.

(c) Frédéric Guilloud, Télécom Paris - July 2004

En outre, nous avons proposé un ensemble de trois paramètres qui quantifient la complexité d'une architecture de décodeur LDPC. Ce cadre formel a aussi rendu possible la conception de nouvelles solutions originales pour la conception de décodeur LDPC, et fait actuellement l'objet d'une demande de brevet.

D'un point de vue chronologique, l'analyse de l'état de l'art et de la complexité des décodeurs LDPC montrent que les premières solutions de décodeurs se sont concentrées sur la simplification du réseau d'interconnection, conduisant les chercheurs à construire des codes LDPC avec de telles propriétés. Ensuite le problème de la quantité de mémoire nécessaire à l'algorithme à propagation de croyance a été abordé, et des solutions telles que de nouveaux algorithmes et de nouveaux séquencements ont été proposés.

Notre deuxième contribution va dans ce sens puisque nous avons proposé un algorithme sous-optimal appelé l'algorithme $\lambda-$min. Il diminue la complexité du traitement des parités, ainsi que la quantité de mémoire nécessaire à chaque itération : en effet, sans aucune perte significative de performance, cet algorithme permet une réduction de la mémoire allant jusqu'à 75% pour des codes de taux élevé, comparé à l'algorithme optimal. On a également proposé une architecture originale pour les processeurs de noeud de parité mettant en oeuvre l'algorithme $\lambda-$min. Ces résultats ont été publié et font l'objet d'un dépôt de brevet.

La validation de ces nouveaux algorithmes, et les simulations lancées pour analyser et optimiser des codes LDPC peuvent prendre beaucoup de temps. La capacité énorme qu'offre aujourdhui les matériels reconfigurables peut être mis à profit pour réaliser des simulations numériques rapides et intensives. L'exploitation de la reconfigurabilité des FPGA par exemple peut accélerer efficacement les simulations effectuées jusqu'à présent sur PC.

Notre troisième contribution est la conception et l'utilisation d'une architecture de décodeur LDPC générique, que nous avons intégrée dans une plateforme d'évaluation basée sur des FPGA. Le mot *générique* signifie tout d'abord que n'importe quel code LDPC d'une longueur donnée $N$ et d'un rendement donné $R$ peut être décodé. La deuxième signification est que la description de l'architecture a été conçue tout en gardant à l'esprit que tous les composants devraient pouvoir être changés. Cela signifie par exemple que d'autres processeurs de noeud de parité par exemple peuvent être mis en oeuvre, implantant d'autres algorithmes de décodage. Pour cela, tous les paramètres et toutes latences sont réglables. Nous avons montré qu'en utilisant des FPGA de la technologie actuelle (Xilinx Virtex-1000E), l'aspect générique n'engendrait pas une perte trop importante de perfromance vis à vis d'autres décodeurs optimisés. Ce décodeur a été utilisé d'abord pour valider les performances de l'algorithme $\lambda-$min et son architecture associée. On prévoit également de l'utiliser pour accélérer les simulations d'une grande variété de codes LDPC: les simulations matérielles obtenues offrent un gain de temps d'un facteur

(c) Frédéric Guilloud, Télécom Paris - July 2004

$10^3$ environ par rapport à des simulations logicielles effectuées sur un PC Pentium III - 1, 7 GHz fonctionnant sous Linux.

Le travail réalisé dans cette thèse apporte un nouveau cadre formel à l'étude des décodeurs LDPC. Une approche théorique est maintenant disponible pour concevoir des décodeurs LDPC. L'architecture générique sera adaptée aux autres types de séquencement, et d'autres algorithmes sous optimaux seront conçus pour pouvoir mener des études comparatives entre eux, à des taux d'erreurs binaires relativement faibles. À l'avenir, nous emploierons notre plateforme LDPC comme outil pour étudier les performances de différents codes LDPC, et en particulier effectuer des optimisations afin de trouver de bons codes de longueur finie.

Notons enfin que seul le cas des codes LDPC binaires a été envisagé dans cette thèse. Nous projetons également d'étudier les codes LDPC non-binaires ayant leur alphabet dans $GF(2^q)$. Ces codes promettent d'être plus puissants, et d'offrir des sujets de recherche interessants d'un point de vue architectural et algorithmique.

(c) Frédéric Guilloud, Télécom Paris - July 2004

# Bibliography


Ammar, B., B. Honary, Y. Kou, and S. Lin. July 2002. "Construction of low density parity check codes: a combinatoric design approach." *Proceedings of the IEEE International Symposium on Information Theory.*

Amraoui, A., S. Dusad, and R. Urbanke. July 2002. "Achieving general points in the 2-user Gaussian MAC without time-sharing or rate-splitting by means of iterative coding." *Information Theory, 2002. Proceedings. 2002 IEEE International Symposium on.*

Ardakani, M., and F.R. Kschischang. July 2002. "Designing irregular LPDC codes using EXIT charts based on message error rate." *Proceedings of the IEEE International Symposium on Information Theory.*

Ardakani, Masoud, Terence H. Chan, and Frank R. Kschischang. May 2003. "Properties of the EXIT Chart for One-Dimensional LDPC Decoding Schemes." *Proceedings of CWIT.*

Bahl, L.R., J Cocke, F. Jelinek, and J. Raviv. March 1974. "Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate." *IEEE Transactions on Information Theory* 20:284–287.

Barry, J.R. oct. 2001. Low-Density Parity-Check Codes. Available at `http://www.ece.gatech.edu/~barry/6606/handsout/ldpc.pdf`.

Battail, G., and A. H. M. El-Sherbini. 1982. "Coding for Radio Channels." *Annales des Télécommunications* 37:75–96.

Battail, G., and H. Magalhães De Oliveira. 1993. "Probabilité d'erreur du codage aléatoire avec décodage optimal sur le canal à bruit gaussien additif, affecté ou non de fluctuations d'amplitude (in french)." *Annales des Télécommunications* 48:15–28.

Berrou, C., A. Glavieux, and P. Thitimajshima. May. 23-26, 1993. "Near Shannon limit error-correcting coding and decoding: Turbo-codes." *International Conference on Communication.*

Bhatt, T., K. Narayanan, and N. Kehtarnavaz. Oct. 2000. "Fixed-point DSP Implementation of Low-Density Parity Check Codes." *9th DSP (DSP 2000) Workshop.*



(c) Frédéric Guilloud, Télécom Paris - July 2004

Blanksby, A.J., and C.J. Howland. March 2002. "A 690-mW 1-Gb/s 1024-b, Rate-1/2 Low-Density Parity-Check Code Decoder." *Journal of Solid-State Circuits* 37:404–412.

Bond, J.W., S. Hui, and H. Schmidt. May 2000. "Constructing low-density parity-check codes." *EUROCOMM 2000. Information Systems for Enhanced Public Safety and Security. IEEE/AFCEA*.

Boutillon, E., J. Castura, and F.R. Kschischang. 2000. "Decoder-First Code Design." *Proceedings of the 2nd International Symposium on Turbo Codes and Related Topics*. Brest, France, 459–462.

Boutillon, E., J.-L. Danger, and A. Gazel. Feb 2003. "Design of High Speed AWGN Communication Channel Emulator." *Kluwer Press, Analog Integrated Circuits and Signal Processing International Journal* 34(2):133–142.

Boutillon, Emmanuel, Jacky Tousch, and Frederic Guilloud. 2003, december. *LDPC decoder, corresponding method, system and computer programm*. US Patent pending.

Boutros, J., G. Caire, E. Viterbo, H. Sawaya, and S. Vialle. 2002. "Turbo code at 0.03 dB from capacity limit." *Information Theory, 2002. Proceedings. 2002 IEEE International Symposium on*.

Campello, J., and D.S. Modha. November 2001. "Extended Bit-Filling and LDPC Code Design." *Global Telecommunications Conference*. 985–989.

Chen, J., and M.P.C. Fossorier. May 2002. "Density Evolution for Two Improved BP-Based Decoding Algorithms of LDPC Codes." *IEEE Communication Letters* 6:208–210.

Chen, Jinghu. 2003. "Reduced Complexity Decoding Algorithms For Low-Density Parity Check Codes and Turbo Codes." Ph.D. diss., University of Hawaii.

Chen, Y., and D. Hocevar. 1-5 Dec. 2003. "A FPGA and ASIC Implementation of Rate 1/2, 8088-b Irregular Low Density Parity Check Decoder." *Global Telecommunications Conference, 2003. GLOBECOM '03. IEEE.*

Chung, Richardson, and Urbanke. 2001. "Analysis of Sum-Product Decoding of Low-Density Parity-Check Codes Using a Gaussian Approximation." *IEEETIT: IEEE Transactions on Information Theory*, vol. 47.

Chung, Sae-Young. Sae-Young Chung's Homepage. available at `http://lids.mit.edu/~sychung`.

Chung, S-Y., G.D. Forney, T.J. Richardson, and R.L Urbanke. 2001. "On the Design of Low-Density Parity-Check Codes within 0.0045 dB of the Shannon Limit." *Communications Letters* 5:58–60.

Danger, J.-L., A. Ghazel, E. Boutillon, and H. Laamari. 17-20 Dec. 2000. "Efficient FPGA implementation of Gaussian noise generator for communication channel emu-

lation." *Electronics, Circuits and Systems, 2000. ICECS 2000. The 7th IEEE International Conference on.*

Davey, M. C., and D. J. C. MacKay. 1998. "Low density parity check codes over GF(q)." *IEEE Communications Letters*, vol. 2.

de Baynast, A., and D. Declercq. July 2002. "Gallager codes for multiple user applications." *Information Theory, 2002. Proceedings. 2002 IEEE International Symposium on.*

Declercq, D, and F. Verdier. Sept. 1-5, 2003. "Optimization of LDPC Finite Precision Belief Propagation Decoding with Density Evolution." *3rd International Symposium on Turbo Codes & related topics.*

Djordjevic, I.B., S. Sankaranarayanan, and B.V. Vasic. March 2004. "Projective-Plane Iteratively Decodable Block Codes for WDM High-Speed Long-Haul Transmission Systems." *Lightwave Technology, Journal of*, vol. 22.

Djurdjevic, I., S. Lin, and K. Abdel-Ghaffar. April 2003. "Graph-theoretic construction of low-density parity-check codes." *Communications Letters, IEEE* 7::171 – 173.

Dolinar, S., D. Divsalar, and F. Pollara. Jan-March 1998. "Code Performance as a Function of Block Size." Technical Report, The Telecommunications and Mission Operations Progress Report 42133.

Etzion, Tuvi, Ari Trachtenberg, and Alexander Vardy. Sept. 1999. "Which Codes Have Cycle-Free Tanner Graphs ?" *IEEE Transactions on Information Theory*, vol. 45.

Forney, G.D. March 1973. "The Viterbi Algorithm." *Proceedings of the IEEE* 61:268–278.

Fossorier, M.P.C., M. Mihaljević, and I. Imai. May 1999. "Reduced Complexity Iterative Decoding of Low-Density Parity-Check Codes Based on Belief Propagation." *Transactions on Communications* 47:673–680.

Gallager, R.G. Jan. 1962. "Low-Density Parity-Check Codes." *IRE Transactions on Information Theory* 8:21–28.

———. 1963. *Low-Density Parity-Check Codes.* Cambridge, MA: MIT Press.

———. 1968. *Information Theory and Reliable Communication.* New York: Wiley.

Garcia-Frias, J., and Wei Zhong. June 2003. "Approaching Shannon performance by iterative decoding of linear codes with low-density generator matrix." *Communications Letters, IEEE* 7:266 – 268.

Ghazel, A., E. Boutillon, J.-L. Danger, G. Gulak, and H. Laamari. 26-28 Aug. 2001. "Design and performance analysis of a high speed AWGN communication channel emulator." *Communications, Computers and signal Processing, 2001. PACRIM. 2001 IEEE Pacific Rim Conference on.*

Guilloud, F., E. Boutillon, and J.-L. Danger. may 2002a. "Bit Error Rate Calculation for a Multiband Non Coherent On-Off Keying Demodulation." *International Conference on Communication, ICC 2002*.

———. 19 dec. 2002b. Implémentation de LDPC sur FPGA : optimisation de l'architecture et étude de précision finie (in french). Journée LDPC du GDR ISIS, available at `http://lester.univ-ubs.fr:8080/~boutillon/Journee_GDR_LDPC/compte_rendu.htm`.

———. septembre 2003a. "Décodage des codes LDPC par l'algorithme $\lambda-$min." *19ème colloque GRETSI sur le traitement du signal et des images*.

———. Sept. 1-5, 2003b. "$\lambda-$min Decoding Algorithm of Regular and Irregular LDPC Codes." *3rd International Symposium on Turbo Codes & related topics*.

Guilloud, F., E. Boutillon, and J.-L Danger. May 2002c. "Etude d'un algorithme itératif d'annulation de repliement spectral lors d une conversion A/N parallèle (in french)." *Proceedings of the 5th "Journées Nationales du Réseau Doctoral de Microélectronique"*.

Hagenauer, J., and P. Hoeher. November 1989. "A Viterbi Algorithm with soft-decision outputs and its applications." *Proceedings of the IEEE Globecom'89*. Dallas, Texas.

Hagenauer, J., E Offer, and L. Papke. March 1996. "Iterative Decoding of Binary Block and Convolutional Codes." *IEEE Transactions on Information Theory* 42:1064–1070.

Haley, D., A. Grant, and J. Buetefuer. Nov. 2002. "Iterative encoding of low-density parity-check codes." *Global Telecommunications Conference, 2002. GLOBECOM '02. IEEE*.

Hocevar, D.E. 2003. "LDPC code construction with flexible hardware implementation." *Communications, 2003. ICC '03. IEEE International Conference on*.

Hou, J., P.H. Siegel, L.B. Milstein, and H.D. Pfister. Sept. 2003. "Capacity-approaching bandwidth-efficient coded modulation schemes based on low-density parity-check codes." *IEEE Transactions on Information Theory* 49:2141–2155.

Hou, Jilei, Paul H. Siegel, and Laurence B. Milstein. May 2001. "Performance Analysis and Code Optimization of Low Density Parity-Check Codes on Rayleigh Fading Channels." *IEEE Journal on Selected Areas in communications*, vol. 19.

Howland, C., and A Blanksby. May 2001a. "A 220mW 1Gb/s 1024-Bit Rate-1/2 Low Density Parity Check Code Decoder." *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on , Volume: 4 , 6-9 May 2001*.

———. May 2001b. "Parallel Decoding Architectures for Low Density Parity Check Codes." *Custom Integrated Circuits, 2001, IEEE Conference on*.

Hu, X.-Y., E. Eleftheriou, D.-M. Arnold, and A. Dholakia. November 2001. "Efficient Implementations of the Sum-Product Algorithm for Decoding LDPC Codes." *Global Telecommunications Conference*. 1036–1036E.

Hu, Xiao-Yu, E. Eleftheriou, and D.-M. Arnold. Nov. 2001. "Progressive edge-growth Tanner graphs." *Proceedings of the IEEE Global Telecommunications Conference GLOBECOM '01.*

IEEE. 2001, February. *Special issue on codes on graphs and iterative algorithm.* IEEE Transactions on Information Theory. IEEE.

Johnson, S.J., and S.R. Weller. Dec. 2003a. "High-rate LDPC codes from unital designs." *Proceedings of the IEEE Global Telecommunications Conference GLOBECOM '03.*

———. Sept. 2003b. "Resolvable 2-designs for regular low-density parity-check codes." *Communications, IEEE Transactions on* 51:1413 – 1419.

Jones, C., E. Vallés, M. Smith, and J. Villasenor. 13-16 Oct. 2003. "Approximate-min* constraint node updating for ldpc code decoding." *Military Communications Conference, 2003. MILCOM 2003. IEEE.*

Kienle, F., M.J. Thul, and N. Wehn. Sept. 1-5, 2003. "Implementation Issues of Scalable LDPC-Decoders." *3rd International Symposium on Turbo Codes & related topics.*

Kim, S., G.E. Sobelman, and J. Moon. 2002. "Parallel VLSI Architectures for a Class of LDPC Codes." *ISCAS 2002.*

Koetter, R, and P Vontobel. Sept. 1-5, 2003. "Graph-covers and iterative decoding of finite length codes." *3rd International Symposium on Turbo Codes & related topics.*

Kou, Y., S. Lin, and M.P.C. Fossorier. November 2001. "Low-Density Parity-Check Codes Based on Finite Geometries : A Rediscovery and New Results." *Transactions on Information Theory* 47:2711–2736.

Kschischang, F.R., and B.J. Frey. 1998. "Iterative Decoding of Compound Codes by Probability Propagation in Graphical Models." *Journal on Selected Areas in Communications* 16:219–230.

Lafferty, J., and D. Rockmore. November 2000. "Codes and Iterative Decoding on Algebraic Expander Graphs." *International Symposium on Information Theory and its Applications.* Honolulu, Hawaiin, U.S.A.

Lehmann, F., and G.M. Maggio. Nov. 2003. "Analysis of the iterative decoding of LDPC and product codes using the Gaussian approximation." *Information Theory, IEEE Transactions on* 49:2993 – 3000.

Levine, B., R.R. Taylor, and H. Schmit. 2000. "Implementation of near Shannon limit error-correcting codes using reconfigurable hardware." *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on , 2000.* 217–226.

(c) Frédéric Guilloud, Télécom Paris - July 2004

Li, Jing, K.R. Narayanan, E. Kurtas, and C.N. Georghiades. May 2002. "On the performance of high-rate TPC/SPC codes and LDPC codes over partial response channels." *Communications, IEEE Transactions on* 50:723 – 734.

Lu, B., G. Yue, and X. Wang. Feb. 2004. "Performance Analysis and Design Optimization of LDPC-Coded MIMO OFDM Systems." *Signal Processing, IEEE Transactions on [see also Acoustics, Speech, and Signal Processing, IEEE Transactions on]* 52:348 – 361.

Luby, M.G., M. Mitzenmacher, M.A. Shokrollahi, and D.A. Spielman. February 2001. "Improved Low-Density Parity-Check Codes Using Irregular Graphs." *Transactions on Information Theory* 47:585–598.

Luby, Michael G., Michael Mitzenmacher, M. Amin Shokrollahi, Daniel A. Spielman, and Volker Stemann. 1997. "Practical loss-resilient codes." *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM Press, 150–159.

Lucas, R., M.P.C. Fossorier, Yu Kou, and Shu Lin. June 2000. "Iterative decoding of one-step majority logic deductible codes based on belief propagation." *Communications, IEEE Transactions on* 48:931–937.

Lucent, Technology. 2004. An Overview of Information Theory. Available at `http://www.lucent.com/minds/infotheory/docs/history.pdf`.

MacKay, D. J. C., and M. C. Davey. 2000. "Evaluation of Gallager Codes for Short Block Length and High Rate Applications." In *Codes, Systems and Graphical Models*, edited by B. Marcus and J. Rosenthal, Volume 123 of *IMA Volumes in Mathematics and its Applications*, 113–130. New York: Springer.

MacKay, David J.C., and Christopher P. Hesketh. 2003. "Performance of low density parity check codes as a function of actual and assumed noise levels." Edited by Sharon Flynn, Ted Hurley, Mícheál Mac an Airchinnigh, Niall Madden, Michael McGettrick, Michel Schellekens, and Anthony Seda, *Electronic Notes in Theoretical Computer Science*, Volume 74. Elsevier.

MacKay, David J.C., and Michael S. Postol. 2003. "Weaknesses of Margulis and Ramanujan-Margulis low-density parity-check cCodes." *Electronic Notes in Theoretical Computer Science*, Volume 74. Elsevier.

Mackay, D.J.C. LDPC database. Available at `http://www.inference.phy.cam.ac.uk/mackay/codes/data.html`.

———. March 1999. "Good Error-Correcting Codes Based on Very Sparse Matrices." *Transactions on Information Theory* 45:399–431.

MacKay, D.J.C, and R.M. Neal. 1995. "Good Codes Based on Very Sparse Matrices." *5th IMA Conference on Cryprography and Coding*. Berlin, Germany: Springer.

———. 1996. "Near Shannon Limit Performance of Low-Density Parity-Check Codes." *Electronic Letter* 32:1645–1646.

MacKay, D.J.C, S.T. Wilson, and M.C. Davey. October 1999. "Comparison of Constructions of Irregular Gallager Codes." *Transaction on Communications* 47:1449–1454.

Mannoni, V., D. Declercq, and G. Gelle. Sept. 2002. "Optimized irregular Gallager codes for OFDM transmission." *Personal, Indoor and Mobile Radio Communications, 2002. The 13th IEEE International Symposium on.*

Mansour, M.M., and N.R. Shanbhag. Aug. 12-14, 2002a. "Low-Power VLSI Decoder Architectures for LDPC Codes." *Int. Symp. Low Power Electronic Design.*

———. Nov 17 - 21, 2002b. "Turbo decoder architectures for low-density parity-check codes." *Global Telecommunications Conference, 2002. GLOBECOM '02. IEEE.*

Mao, Y., and A.H. Banihashemi. October 2001a. "Decoding Low-Density Parity-Check Codes With Probabilistic Scheduling." *Communications Letters* 5:414–416.

Mao, Y, and A. H. Banihashemi. June 2001b. "A heuristic search for good low-density parity-check codes at short block lengths." *IEEE International Conference on Communications. ICC 2001.*

Martinez, A., and M. Rovini. 2003, September. "Iterative decoders based on statistical multiplexing." *Proceedings of the 3rd International Symposium on Turbo Codes and Related Topics.*

McEliece, R.J., D.J.C. MacKay, and J.-F. Cheng. February 1998. "Turbo Decoding as an Instance of Pearl's Belief Propagation Algorithm." *IEEE Journal on Selected Areas in Communications* 16:140–152.

Miller, G., and G. Cohen. November 2003. "The rate of regular LDPC codes." *IEEE Transactions on Information Theory* 49:2989–2992.

Morelos-Zaragoza, R.H. The Art of Error Correcting Coding, Wiley, 2002. program available at `http://the-art-of-ecc.com`.

Narayanan, K.R., I. Altunbas, and R. Narayanaswami. Aug. 2003. "Design of serial concatenated MSK schemes based on density evolution." *IEEE Transactions on Communications* 51:1283 – 1295.

Narayanan, K.R., Xiaodong Wang, and Guosen Yue. Oct. 2002. "LDPC code design for MMSE turbo equalization." *Proceedings of the IEEE Information Theory Workshop.*

Nickl, H., J. Hagenauer, and Burkert. Sept. 1997. "Approaching Shannon's capacity limit by 0.2 dB using simple Hamming codes." *Communications Letters, IEEE* 1:130 – 132.

Oenning, T.R., and Jaekyun Moon. March 2001. "A low-density generator matrix interpretation of parallel concatenated single bit parity codes." *Magnetics, IEEE Transactions on* 37:737 – 741.

Okamura, T. July 2003. "Designing ldpc codes using cyclic shifts." *Proceedings of the IEEE International Symposium on Information Theory.*

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems : Networks of Plausible Inference*. San Mateo: CA : Morgan Kaufmann.

Prabhakar, A., and K. Narayanan. Sept. 2002. "Pseudorandom construction of low-density parity-check codes using linear congruential sequences." *Communications, IEEE Transactions on* 50:1389 – 1396.

Press, William H., Saul A. Teukolsky, and William T. Vetterling. 2nd edition (October 30, 1992). *Numerical recipes in C : the art of scientific computing*. Edited by New York : Cambridge University. Cambridge University Press.

project, SPRING. The European SPRING project, number IST-1999-12342. `http://www.imse.cnm.es/spring`.

Richardson, T.J., M.A. Shokrollahi, and R.L Urbanke. February 2001. "Design of Capacity-Approaching Irregular Low-Density Parity-Check Codes." *Transactions on Information Theory* 47:619–637.

Richardson, T.J., and R.L Urbanke. February 2001. "Efficient Encoding of Low-Density Parity-Check Codes." *Transactions on Information Theory* 47:638–656.

Rosenthal, J., and P. O. Vontobel. June 2001. "Constructions of regular and irregular LDPC codes using Ramanujan graphs and ideas from Margulis." *Proceedings of the IEEE International Symposium on Information Theory*. 4.

Sankaranarayanan, S., B. Vasic, and E.M. Kurtas. Sept. 2003. "Irregular low-density parity-check codes: construction and performance on perpendicular magnetic recording channels." *Magnetics, IEEE Transactions on* 39:2567 – 2569.

Schlegel, Christian. 1997. *Trellis Coding*. IEEE Press.

Shannon, C.E. 1948. *The Mathematical Theory of Communication*. Urbana, IL: University of Illinois Press.

Sipser, M., and D.A. Spielman. November 1996. "Expander Codes." *IEEE Transactions on Information Theory* 42:1710–1722.

Tanner, R. Septembre 1981. "A recursive approach to low complexity codes." *IEEE Transactions on Information Theory* 27:533–547.

ten Brink, S. May 1999. "Convergence of iterative decoding." *Electronics Letters* 35:806 – 808.

Thangaraj, A., and S.W. McLaughlin. Sept. 2002. "Thresholds and scheduling for LDPC-coded partial response channels." *Magnetics, IEEE Transactions on* 38:2307 – 2309.

Thorpe, J. 2002. "Design of LDPC graphs for hardware implementation." *Information Theory, 2002. Proceedings. 2002 IEEE International Symposium on*.

Thul, M.J., F. Gilbert, and N. Wehn. 6-10 April 2003. "Concurrent interleaving architectures for high-throughput channel coding." *Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03). 2003 IEEE International Conference on.*

Tian, Tao, C. Jones, J.D. Villasenor, and R.D. Wesel. 2003. "Construction of irregular LDPC codes with low error floors." *Communications, 2003. ICC '03. IEEE International Conference on.*

Ungerboeck, G. Jan. 1982. "Channel Coding with Multilevel/Phase Signalling." *IEEE Trans. Info. Theory* 28:55–67.

Urbanke, R. LdpcOpt. Available at `http://lthcwww.epfl.ch/research/ldpcopt/`.

Varnica, N., and A. Kavcic. April 2003. "Optimized low-density parity-check codes for partial response channels." *Communications Letters, IEEE* 7:168 – 170.

Vasic, B. July 2002. "Combinatorial constructions of low-density parity check codes for iterative decoding." *Proceedings of the IEEE International Symposium on Information Theory.*

Vasic, B., I.B. Djordjevic, and R.K. Kostuk. Feb. 2003. "Low-density parity check codes and iterative decoding for long-haul optical communication systems." *Lightwave Technology, Journal of* 21:438 – 446.

Verdier, F., and D Declercq. Sept. 1-5, 2003. "A LDPC Parity Check Matrix Construction for Parallel Hardware Decoding." *3rd International Symposium on Turbo Codes & related topics.*

Verdier, F., D. Declercq, and Philippe J.-M. Dec. 2002. "Parallélisation et implantation FPGA d'un décodeur LDPC." *Journées Francophones sur l'Adéquation Algorithme Architecture JFAAA 2003.*

Viterbi, A.J. April 1967. "Error Bound for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm." *IEEE Transactions on Information Theory* 13:260–269.

Waksman, A. January 1968. "A permutation network." *Journal of the Association for Computing Machinery* 15:159–163.

Wiberg, N. 1996. "Codes and Decoding on General Graphs." Ph.D. diss., Linköping University, Sweden.

Wiberg, N., H.-A. Loegliger, and R. Kötter. 1995. "Codes and iterative decoding on general graphs." *Eur. Trans. Telecom.* 6:513–525.

Wolf, J.K. January 1978. "Efficient Maximum Likelihood Decoding of Linear Block Codes Using a Trellis." *IEEE Transactions on Information Theory* 24:76–80.

Wozencraft, J. M., and I. M. Jacobs. 1965. *Principles of Com- munication Engineering.* New York, NY: John Wiley and Sons.

Yeo, E., B. Nikolić, and V. Anantharam. Nov. 25-29, 2001. "High Throughput Low-Density Parity-Check Decoder Architectures." *IEEE Globecom, San Antonio*.

Yeo, E., P. Pakzad, B. Nikolić, and V. Anantharam. March 2001. "VLSI Architectures for Iterative Decoders in Magnetic Recording Channels." *Transactions on Magnetics* 37:748–755.

Zhang, Haotian, and J.M.F. Moura. Dec. 2003. "The design of structured regular LDPC codes with large girth." *Proceedings of the IEEE Global Telecommunications Conference GLOBECOM '03*.

Zhang, Juntan, and M. Fossorier. 3-6 Nov. 2002. "Shuffled belief propagation decoding." *Signals, Systems and Computers, 2002. Conference Record of the Thirty-Sixth Asilomar Conference on*.

Zhang, T., and K. K. Parhi. 2002. "A 56 Mbps (3,6)-regular FPGA LDPC Decoder." *Worshop on Signal Processing Systems 2002, SIPS*.

———. May 2003. "An FPGA Implementation of (3,6)-Regular Low-Density Parity-Check Code Decoder." *EURASIP Journal on Applied Signal Processing, special issue on Rapid Prototyping of DSP Systems* 2003:530–542.

Zhang, T., and K.K. Parhi. Sept. 2001. "VLSI implementation-oriented (3,k)-regular low-density parity-check codes." *Worshop on Signal Processing Systems 2001, SIPS*.

Zhang, T., Z Wang, and K.K. Parhi. May 2001. "On Finite Precision Implementation of Low Density Parity Check Codes Decoder." *Proceedings of ISCAS*. Sydney, Australia.

(c) Frédéric Guilloud, Télécom Paris - July 2004