Towards and ASIP optimized for multi-standard turbo decoding

Rachid Al Khayat

To cite this version:

Rachid Al Khayat. Towards and ASIP optimized for multi-standard turbo decoding. Electronics. Télécom Bretagne, Université de Bretagne-Sud, 2012. English. NNT : . tel-00821906

HAL Id: tel-00821906
https://theses.hal.science/tel-00821906

Submitted on 13 May 2013

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.

N° d'ordre : 2012telb0247

Sous le sceau de l'Université européenne de Bretagne

Télécom Bretagne

En habilitation conjointe avec l'Université de Bretagne-Sud

Ecole Doctorale – SICMA

Vers une architecture optimisée d'ASIP pour turbo-décodage multi-standard

---

Thèse de Doctorat

Mention : STIC (Sciences et Technologies de l'Information et de la Communication)

Présentée par **Rachid AL KHAYAT**

Département : Electronique

Laboratoire : Lab-STICC / Pôle CACS

Directeur de thèse : Michel Jézéquel

Soutenue le 16 novembre 2012

Jury :

M. Maryline Hélard,	Professeure à l'INSA de Rennes	(Présidente)
M. Fabrice Monteiro,	Professeur à l'Université de Lorraine	(Rapporteur)
M. Smail Niar,	Professeur à l'Université de Valenciennes	(Rapporteur)
M. Guido Masera,	Professeur à Politecnico di Torino	(Examinateur)
M. Philippe Coussy,	Maître de conférences/HDR, à l'Université de Bretagne-Sud	(Examinateur)
M. Amer Baghdadi,	Maître de conférences/HDR, à Télécom Bretagne	(Encadrant)
M. Michel Jézéquel,	Professeur à Télécom Bretagne	(Directeur)

Contents

Introduction 1

1 Turbo Codes and Turbo Decoding 5

1.1 Fundamentals of Channel Coding . . . 6

1.1.1 Coding Theory . . . 6

1.1.2 Channel Model . . . 6

1.1.3 Modulation . . . 7

1.2 Convolutional Codes . . . 7

1.2.1 Recursive Systematic Convolutional Codes . . . 9

1.2.2 Trellis Representation . . . 10

1.3 Turbo Codes . . . 10

1.3.1 Concatenation of Convolutional Codes . . . 11

1.3.2 Turbo Code Interleaver ($\prod$) . . . 14

1.3.2.1 Almost regular permutation (ARP) . . . 15

1.3.2.2 Quadric polynomial permutation (QPP) . . . 16

1.3.3 Multi Standard Channel Convolutional and Turbo Coding Parameters . 16

1.4 Turbo Decoding . . . 16

1.4.1 Soft In Soft Out (SISO) Decoding . . . 17

1.4.1.1 MAP decoding algorithm . . . 18

1.4.1.2 Log-MAP and Max-Log-MAP decoding algorithm . . . 20

1.4.2 Parallelism in Turbo Decoding . . . 22

1.4.2.1 BCJR metric level parallelism . . . 23

1.4.2.2 SISO decoder level parallelism . . . 24

1.4.2.3 Parallelism of Turbo decoder . . . 26

1.4.2.4 Parallelism levels comparison . . . 26

1.4.3 Decoding Large Frames in Turbo Decoding . . . 28

1.4.4 Software Model for Parallel Turbo Decoder . . . 28

1.5 Summary . . . 29

2 ASIP Design Methodologies and Initial Architecture 31

2.1 Customizable Embedded Processors 32

2.2 ASIP Design Methodologies and Tools 33

2.2.1 ASIP Design Approaches 33

2.2.2 Synopsys's ADL-based Design Tool: Processor Designer 34

2.3 Initial ASIP Architecture for Turbo Decoding 36

2.3.1 Overview of the Design Approach and the Architectural Choices 36

2.3.2 Building Blocks of the Initial ASIP 37

2.3.3 Overall Architecture of the Initial ASIP 39

2.3.4 Sample Program of the Initial ASIP 41

2.4 Summary 44

3 Optimized ASIP for Multi-standard Turbo Decoding 45

3.1 State of the Art 46

3.2 Proposed Decoder System Architecture 47

3.2.1 Architecture-Level Optimization 48

3.2.1.1 Pipeline optimization 49

3.2.1.2 Windowing 55

3.2.2 Algorithmic-Level Optimization 58

3.2.3 Memory-Level Optimization 60

3.2.3.1 Normalizing extrinsic information and memory sharing . . . 60

3.2.3.2 Bit-level extrinsic information exchange method 60

3.2.3.3 Interleaver generator 61

3.2.3.4 Restricting the trellis support 64

3.3 Extrinsic Exchange Module 65

3.4 ASIP Dynamic Reconfiguration 65

3.5 System Characteristics and Performance 67

3.6 Power Consumption Analysis for ASIC Implementation 70

3.6.1 Power Consumption Analysis Flow 70

3.6.2 Power Consumption Analysis Results 72

3.6.3 Results Comparison 73

3.7 Summary 75

4 FPGA Prototype for Multi-standard Turbo Decoding 77
4.1 ASIP Design, Validation and Prototyping Flow 78
4.1.1 LISA Abstraction Level 78
4.1.2 HDL Abstraction Level 78
4.1.3 FPGA Implementation Level 80
4.2 Proposed Prototyping Platform for the ASIP-based Turbo Decoder 80
4.2.1 Multi-Standard Transmitter 81
4.2.2 Channel Module 83
4.2.3 Multi-Standard Turbo Decoder 84
4.2.3.1 Memory organization 84
4.2.3.2 Interleaver module 84
4.2.3.3 Prototyped extrinsic exchange module 86
4.2.3.4 Area of the decoder core 87
4.2.4 Error Counter 87
4.2.5 Scheduling and Control 87
4.2.6 Performance Results 88
4.3 Summary 90

5 Towards the Support of LDPC Decoding 91
5.1 LDPC codes 92
5.1.1 Representations of LDPC Codes 92
5.1.2 Quasi-Cyclic Low-Density Parity-Check (QC-LDPC) 93
5.1.3 QC-LDPC Codes Parameters in WiMAX and WiFi Standards 94
5.2 LDPC Decoding Algorithms 94
5.2.1 Min-Sum Algorithm 95
5.2.2 Normalized Min-Sum Algorithm 96
5.3 Multi-ASIP Architecture Overview 97
5.3.1 Multi-ASIP in Turbo Mode 98
5.3.2 Multi-ASIP in LDPC Mode 98
5.4 Computational Scheduling 99
5.4.1 Computational Scheduling in Turbo Mode 99
5.4.2 Computational Scheduling in LDPC Mode 99
5.4.2.1 Simple example with 2-ASIP architecture 99
5.4.2.2 Proposed scheduling with 8-ASIP architecture 102
5.4.2.3 Address generation in LDPC mode 103
5.5 Memory Organization and Sharing 103
5.5.1 Memories in Turbo Mode 103

5.5.2 Memories in LDPC Mode 105
5.5.3 Combined Memory Organization 105
5.6 ASIP Pipeline and Instruction-Set 106
5.6.1 LDPC Decoder Pipeline Stages 107
5.6.2 Pipeline Description in The *CN-update* Phase (RV) 107
5.6.3 Pipeline Description in The *VN-update* Phase (UV) 109
5.6.4 Sample Assembly Program in LDPC Mode 110
5.7 Synthesis Results 111
5.8 Summary 111

Résumé en Français **115**

Glossary **125**

Notations **129**

Bibliography **131**

List of publications **137**

List of Figures

1.1 Transmission scheme using an error correcting code 6
1.2 BPSK constellation 8
1.3 Convolutional encoder (a) 64-state single binary code (b) 64-state single binary RSC (c) 8-state single binary code 8
1.4 RSC encoder (a) 8-state double binary code RSC (b) 8-state single binary RSC 9
1.5 Trellis diagram of the convolutional encoder of Figure 1.3(c) 10
1.6 Turbo encoder examples 11
1.7 Structure of rate 1/3 Turbo encoder in 3GPP LTE 12
1.8 Trellis for 3GPP (Single binary 8 states) 13
1.9 Trellis for WiMAX/DVB-RCS (double binary 8 states) 14
1.10 Structure of rate 1/3 double binary Turbo encoder 15
1.11 Turbo decoder principal 18
1.12 System diagram with Turbo encoder, BPSK modulator, AWGN channel and Turbo decoder 19
1.13 (a) Forward backward scheme (b) Butterfly scheme 24
1.14 One-level Look-ahead Transform for 3GPP-LTE Trellis (Radix-4) 25
1.15 Sub-block parallelism with message passing for metric initialization: (a) Non circular code, (b) Circular Code 27
1.16 Shuffled Turbo decoding 27
1.17 Sub-block parallelism with message passing for metric initialization (a) Non circular code (3GPP), (b) Circular Code (WiMAX) 28
1.18 Organization of the Turbo decoder's software model 29
1.19 BER for 3GPP-LTE frame size 864 bits with different scaling factors 30

2.1 Performance-Flexibility Trade-off of Different Architecture Models 34
2.2 LISA architecture exploration flow 36
2.3 Basic computational units of the initial ASIP (a) Adder node (b) Modulo compare unit 38
2.4 Modulo algorithm extrinsic information processing 39

2.5 Forward Recursion Unit composed of 32 ANF (Adder Node Forward) and 8 4-input Compare Unit (a) Compare Units used for state metric computation (b) Compare Units used for extrinsic information computation 40

2.6 Overall architecture of the initial ASIP 41

3.1 Proposed Turbo decoder system architecture 48

3.2 Pipeline reorganization (a) Initial ASIP stage EX (b) Optimized pipeline stages (EX, MAX1,MAX2) 50

3.3 Modified pipeline stages (EX, MAX) 52

3.4 Architecture of the proposed recursion unit (a) Forward Recursion Unit composed of 32 ANF (b) 8 4-input Compare Units used for state metric computation and 4 8-input Compare Units used for extrinsic computation (c) Adder Node Forward 54

3.5 Critical path comparison: (a) Critical path for TurbASIP$_{v2}$, (b) Critical path for TurbASIP$_{v3}$ 55

3.6 TurbASIP$_{v3}$ pipeline architecture 56

3.7 Windows processing in butterfly scheme (a) Windowing in butterfly computation scheme, (b) Window addressing in butterfly 57

3.8 Unused memory regions in fixed size windowing 58

3.9 Butterfly scheme with Radix-4 59

3.10 QPP Interleaver generators for even and odd addresses with step-size=2 62

3.11 ARP interleaver generator 63

3.12 Interleaving address generator with butterfly scheme 64

3.13 Proposed architecture for the Extrinsic Exchange Module 65

3.14 Config_Reg reserved bits 66

3.15 Parameters can be changed in runtime 67

3.16 BER performance comparison obtained from the 2-ASIP FPGA prototype and the reference software model (fixed point) for (a) 3GPP-LTE block-size=864, (b) WiMAX, block-size=1728 70

3.17 Power consumption analysis flow 71

3.18 Power consumption related to each pipeline stage 73

4.1 Prototyping Flow: (a) LISA abstraction level, (b) HDL abstraction level, (c) FPGA implementation level 79

4.2 Overview of the proposed environment for multi-standard turbo decoding . . . 81

4.3 Turbo encoder with random source generator 82

4.4 Flexible convolutional encoder 83

4.5 Input/output interface of the integrated AWGN channel emulator 83

4.6 TurbASIP and attached memories 85

4.7 Internal data organization in input memories 85

4.8 Architecture of the prototyped Extrinsic Exchange Module 86

4.9 Verification Module 88
4.10 Proposed scheduling of the complete Turbo decoder platform 88
4.11 FER performance obtained from the 2-ASIP Turbo decoder FPGA prototype for WiMAX, block-size=1920 @ 7iteration 89
4.12 FER performance obtained from the 2-ASIP Turbo decoder FPGA prototype for DVB-RCS, block-size=864 @ 7iteration 89

5.1 Tanner Graph of the Hamming code (7,4). 93
5.2 Parity check matrix H for a block length of 1944 bits, code rate 1/2, IEEE 802.11n (1944, 972) LDPC code. H consists of M×N sub-matrices (M=12, N=24 in this example) 93
5.3 Tanner graph example showing the two-phase message passing decoding 95
5.4 Overview of the proposed Multi-ASIP architecture for LDPC/Turbo decoding . 97
5.5 NoC with binary de-Bruijn topology in Turbo mode 98
5.6 NoC with reconfigured binary de-Bruijn topology in LDPC mode 99
5.7 Forward backward schedule 100
5.8 H_{base} example with N=6 and M=2 100
5.9 Proposed computational scheduling in LDPC mode with 2 ASIPs at t=T_0 . . . 101
5.10 Proposed computational scheduling in LDPC mode with 2 ASIPs at t=T_1 . . . 101
5.11 Proposed computational scheduling in LDPC mode with 2 ASIPs at t=T_2 . . . 102
5.12 Proposed computational scheduling in LDPC mode with 2 ASIPs at t=T_3 . . . 102
5.13 Proposed scheduling in LDPC mode with 8 ASIPs 104
5.14 Memory banks organization in Turbo mode: (a) Extrinsic memory, (b) Input memory 105
5.15 LDPC mode : (a) Input memory (b) Extrinsic memory 106
5.16 Pipeline in LDPC Mode 108
5.17 FIFO accesses in *CN-update* and *VN-update* phases 109

List of Tables

1.1 Selection of wireless communication standards and channel codes 17
1.2 Comparison of parallelism levels available in Turbo decoding 28

3.1 State of the art implementations with area normalization 47
3.2 Initial ASIP pipeline usage 49
3.3 Pipeline usage for TurbASIP$_{v1}$ 51
3.4 Pipeline usage for TurbASIP$_{v2}$ 52
3.5 Pipeline usage for TurbASIP$_{v3}$ 53
3.6 Comparison table for added and eliminated logic for the proposed architecture . 53
3.7 Results comparison in terms of PUP, logic area, decoding speed and throughput 53
3.8 Throughput comparison between initial ASIP and TurbASIP$_{v3}$ after applying Radix-4 59
3.9 Symbol-to-bit conversion 60
3.10 Bit-to-symbol conversion 60
3.11 Parameters definition, method of calculation, and number of reserved bits . . . 66
3.12 Typical memories configuration used for one ASIP decoder component (area with @65nm CMOS technology) 68
3.13 TurbASIP$_{v3}$ performance comparison with state of the art implementations . . 69
3.14 Power consumption, throughput and area before and after optimizations 73
3.15 TurbASIP$_{v1}$ energy efficiency results and comparisons 74

4.1 FPGA synthesis results of the Transmitter Module 83
4.2 FPGA synthesis results of the AWGN channel module 84
4.3 FPGA synthesis results of the Receiver module 87
4.4 FPGA synthesis results of the Verification module 87
4.5 FPGA synthesis results of the 2-ASIP Turbo decoder platform prototype 89

5.1 LDPC code parameters in WiFi and WiMAX 94
5.2 Memory requirement in Turbo mode 104
5.3 Memory requirement in LDPC mode 105

5.4 Proposed memory organization to support both Turbo and LDPC modes 106
5.5 Results comparison of the LDPC/Turbo proposed architecture 112

Introduction

SYSTEMS -on-chips in the field of digital communications are becoming extremely diversified and complex with the continuous emerging of new digital communication systems and standards. In this field, channel decoding is one of the most computation, communication, and memory intensive, and thus, power-consuming component.

In fact, the severe requirements imposed in terms of throughput, transmission reliability, and power consumption are the main reasons that make ASIC (Application Specific Integrated Circuit) realizations in the field of digital communications more and more diversified and complex. Cost, energy and high-throughput requirements of channel decoders have been extensively investigated during the last few years and several implementations have been proposed. Some of these implementations succeeded in achieving high throughput for specific standards thanks to the adoption of highly dedicated architectures that work as hardware accelerators. However, these implementations do not take into account flexibility and scalability issues. Particularly this approach implies the allocation of multiple separated hardware accelerators to realize multi-standard systems, which often result in poor hardware efficiency. Furthermore, it implies long design times that are scarcely compatible with the severe time-to-market constraints and the continuous development of new standards and applications.

Besides these severe performance requirements, emerging digital communication systems often require multi-standard interoperability with various qualities of service and diverse compulsory and/or optional techniques inside each standard. This is particularly true for channel coding and the large variety of forward error correction techniques which are proposed today. As an example, IEEE 802.16e (WiMAX) standard specifies the adoption of four error correcting codes (convolutional, Turbo, LDPC, and block Turbo) and each of them is associated to a large multiplicity of code rates and frame lengths. Flexibility requirement of channel decoder becomes highly crucial.

Problems and Objectives

In this context, many recent works have been proposed targeting flexible, yet high throughput, implementations of channel decoders. The flexibility varies from supporting different modes of a single communication standard to the support of multi-standards multi-modes applications. Other implementations have even increased the target flexibility to support different channel coding techniques.

Regarding the architecture model, besides the conventional parametrized hardware models, recent efforts have targeted the use of Application-Specific Instruction-set Processor models (ASIP). Such an architecture model enables the designer to freely tune the flexibility/performance trade-off as required by the considered application requirements. Related contributions are emerging rapidly seeking to improve the resulted architecture efficiency in terms of performance/area and in addition to increase the flexibility support. However, the architecture efficiency of application-specific processors is directly related to the devised instruction set and

pipeline stages usage. Most of recently proposed works do not consider or present this key issue explicitly.

In fact, the need of optimized solutions in terms of performance, power consumption, and area still exist and cannot be neglected against flexibility. In common understanding, a ”blind” approach towards flexibility results in some loss in optimality.

Hence, the main aim of this thesis work is related to unifying flexibility-oriented and optimization-oriented approaches in the design of channel decoders. By considering mainly the challenging turbo decoding application, the objective is to demonstrate how the architecture efficiency of application-specific instruction-set based processors can be considerably improved. Another objective of this work is to investigate the possibility to use the ASIP-based design approach to increase the flexibility of the turbo decoder to support LDPC decoding.

Contributions

Towards these objectives, following are a summary of the major contributions of this thesis work:

Design of Application-Specific Instruction-set Processor (ASIP) for Turbo decoder:

- Proposal and design of a multi-standard ASIP-based Turbo decoder achieving high architecture efficiency in terms of bit/cycle/iteration/mm^2.
- Optimization of the dynamic reconfiguration speed of the proposed ASIP architecture supporting all parameters of 3GPP-LTE/WiMAX/DVB-RCS standards.
- Design of low complexity ARP and QPP interleavers for butterfly scheme and Radix4 technique.
- Proposal and design of a complete FPGA prototype for the proposed multi-standard Turbo decoder.
- Investigate the impact of the ASIP’s pipeline optimization on the energy efficiency.

Design a scalable and flexible high throughput multi-ASIP combined architecture for LDPC and Turbo Decoding:

- Proposal of an efficient memory sharing for flexible LDPC/Turbo decoding.
- Proposal and design of an ASIP combined architecture for LDPC and Turbo decoding.

Thesis Breakdown

The thesis manuscript is divided into five chapters as follows:

Chapter 1 introduces the basic concepts related to convolutional Turbo codes and their decoding algorithms. First an overview of the fundamental concepts of channel coding and the basics for error-correcting codes are introduced. Then, on the transmitter side, we introduce the convolutional codes and the Recursive Systematic Convolutional (RSC) codes. Based on these component codes, Turbo codes principle is presented in the third section with emphasis on the

interleaving function and the various related parameters specified in existing wireless communication standards. Finally, the last section is dedicated to the presentation of the challenging Turbo decoding concept at the receiver side. This includes the presentation of the reference MAP algorithm, the Max-Log-MAP simplification, the various available parallelism techniques, and an example of achievable error rate performance results.

Chapter 2 introduces the ASIP-based design approach and the considered Synopsys (ex. CoWare) design tool: Processor Designer. Based on this design approach, the chapter presents the initial ASIP architecture for Turbo decoding which constitutes the starting point of this thesis work.

Chapter 3 is aimed to illustrate how the application of adequate algorithmic and architecture level optimization techniques on an ASIP for turbo decoding can make it even an attractive and efficient solution in terms of area, throughput, and power consumption. The suggested architecture integrates two ASIP components supporting binary/duo-binary turbo codes and combines several optimization techniques regarding pipeline structure, trellis compression (Radix4), and memory organization. This chapter is organized in the following order. First of all, a brief state of the art section is provided to summarize the available hardware implementations related to this domain. Then, the proposed optimization techniques regarding pipeline structure, trellis compression (Radix4), and memory organization are detailed. Furthermore, in order to improve the speed of reconfigurability of the proposed architecture, a new organization of the instruction program is presented. Then, the achieved Turbo decoder system performance is summarized and the architecture efficiency is compared with state of the art related works. Finally, the impact of the proposed pipeline optimization on energy efficiency is discussed. This last contribution concerns a joint effort with another PhD student at the CEA-LETI: Pallavi Reddy.

Chapter 4 is dedicated to the presentation of the FPGA prototype for the proposed multi-standard Turbo decoder. The first section presents the adopted ASIP-based LISA to FPGA prototyping flow, while the second section details the proposed prototyping environment. This includes: (1) the GUI (Graphical User Interface) in order to configure the platform with desired parameters such as targeted standard, frame size, code rate, and number of iterations from a host computer, (2) the multi-standard flexible transmitter, (3) the hardware channel emulator, (4) the proposed multi-standard Turbo decoder, (5) the error counter, and (6) and the overall scheduling control. The third and last section presents and compares the obtained FPGA synthesis results.

Chapter 5 presents our contribution towards increasing the flexibility of the designed Turbo decoder to support LDPC decoding. It consists of a joint effort with another PhD student at the Electronics department of Telecom Bretagne: Purushotham Murugappa. The main result concerns the proposal and the design of a multi-ASIP architecture for channel decoding supporting binary/duo-binary Turbo codes and LDPC codes. The proposed architecture achieves a fair compromise in area and throughput to support LDPC and turbo codes for an array of standards (LTE, WiMAX, WiFi, DVB-RCS) through efficient sharing of memories and network resources. The chapter starts with a brief introduction on LDPC codes with emphasis on the particular class of structured quasi-cyclic LDPC codes and adopted parameters in the considered wireless standards. Then LDPC iterative decoding is introduced and the low complexity reference decoding algorithms are presented. Subsequent sections present the proposed LDPC/Turbo decoding architecture in a top-down approach. First presenting the functional description of the overall architecture and the main architectural choices and design motivations. Following, the proposed memory organization and sharing is presented. Then, the proposed scheduling, pipeline structure, and instruction set in both Turbo and LDPC decoding mode are presented. The last section presents logic synthesis results along with future perspectives.

CHAPTER 1

Turbo Codes and Turbo Decoding

THIS first chapter introduces the basic concepts related to convolutional Turbo codes and their decoding algorithms. First an overview of the fundamental concepts of channel coding and the basics for error-correcting codes are introduced. Then, on the transmitter side, we introduce the convolutional codes and the Recursive Systematic Convolutional (RSC) codes. Based on these component codes, Turbo codes principle is presented in the third section with emphasis on the interleaving function and the various related parameters specified in existing wireless communication standards. Finally, the last section is dedicated to the presentation of the challenging Turbo decoding concept at the receiver side. This includes the presentation of the reference MAP algorithm, the Max-Log-MAP simplification, the various available parallelism techniques, and an example of achievable error rate performance results.

1.1 Fundamentals of Channel Coding

Channel codes are made up of two main type: convolutional codes and block codes. Convolutional codes are used primarily for real-time error correction and can convert an entire data stream into one single codeword. The encoded bits depend not only on the current informational k input bits but also on past input bits. Block codes tend to be based on the finite field arithmetic and abstract algebra. Block codes accept a block of k information bits and return a block of n coded bits. Block codes are used primarily to correct or detect errors in data transmission. Commonly used block codes are Reed-Solomon codes, BCH codes, Golay codes, Hamming codes and LDPC.

1.1.1 Coding Theory

Coding theory started with the pioneering work of C. Shannon (1948), who established the fundamental problem of reliable transmission of information. Shannon introduced the channel capacity CC (expressed in bits per channel use) as a measure of the maximum information that can be reliably transmitted over a communications channel subject to additive white Gaussian noise. If the information code rate R from the source is less than the channel capacity CC, then it is theoretically possible to achieve reliable (error-free) transmission over the channel by an appropriate coding [1]. Using the set of random codes, this theorem proved the existence of a code enabling information to be transmitted with any given probability of error. However, Shannon's theorem does not give any idea about how to find such a code achieving the channel capacity. The work of Shannon stimulated the coding theory community to find error correcting codes capable of achieving the channel capacity. With such a code C, a message d of k_c information symbols is encoded through an encoder providing a codeword X of C, composed of n_c coded symbols X_i, $i = 1, ..., n_c$ (Figure 1.1). The ratio $R = k_c/n_c$ denotes the code rate. After transmission over a noisy channel, the noisy symbols Y are received by the decoder. This later uses the properties of the code to try to recover the transmission errors by finding the most probable transmitted codeword $\hat{X}$, or equivalently the most probable message $\hat{d}$.

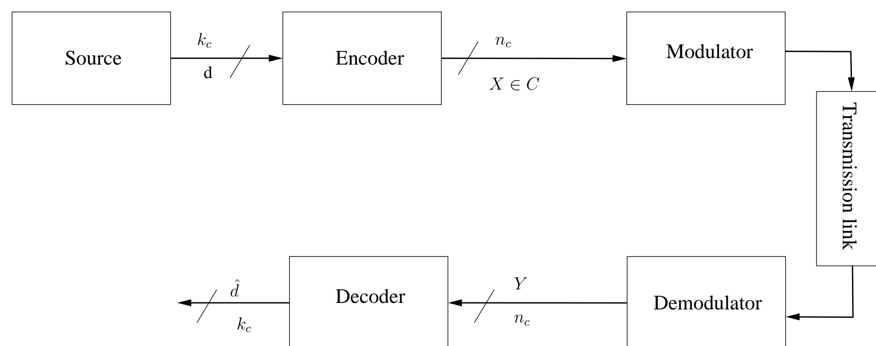

Figure 1.1: Transmission scheme using an error correcting code

1.1.2 Channel Model

The channel depicted in Figure 1.1 corresponds to the succession of three components: the modulator, the transmission link, and the demodulator. The modulator transforms the digital representation of the codeword into an analog signal that is transmitted over the link to the receiver.

At the receiver side, the demodulator converts the analog signal into its digital counterpart that is fed into the decoder.

In perfect modulation, synchronization, and demodulation, we can represent the channel by a discrete-time model. In this thesis, the channel model is represented by a binary-input additive white Gaussian noise (AWGN) of mean 0 and variance σ^2, and adopting binary shift key (BPSK) modulation where $X_i \in 0, 1$ are mapped to modulated symbols $\in -1, 1$. The channel transition probability can be expressed for AWGN channel as follows:

$$p(Y_i|X_i) = \prod_{k=1}^{n_c} \left(\frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(Y_{i,k}-X_{i,k})^2}{\sigma^2}} \right) = K \cdot e^{\frac{\sum_{k=1}^{n_c} Y_{i,k} \cdot X_{i,k}}{\sigma^2}} \quad (1.1)$$

The Bit Error Rate (BER) and Frame Error Rate (FER) performance depends on the signal to noise ratio E_b/N_0, where E_b is the energy per information bit and N_0 is the real power spectrum density of the noise. The Gaussian noise variance σ of the channel is function of signal to noise ratio and of the code rate R:

$$\sigma = \sqrt{\frac{1}{2R \times \frac{E_b}{N_0}}} \quad (1.2)$$

1.1.3 Modulation

The modulation process in a digital communication system maps a sequence of binary data onto a set of corresponding signal waveforms. These waveforms may differ in either amplitude or phase or in frequency, or some combination of two or more signal parameters.

Phase Shift Keying (PSK): In this type of modulation the phase of the carrier signal is changed in accordance with the incoming sequence of the binary data. If the phase of a carrier represent m bits, then $M = 2^m$ different phases are required for all possible combinations of m bits. If we set the value of M = 2, 4, 8 then the corresponding PSK is called BPSK, QPSK and 8-PSK.

BPSK is the adopted modulation for the rest of this thesis . BPSK is the simplest form of phase shift keying (PSK). It uses two phases as illustrated in Figure 1.2 and so can also be named 2-PSK. This modulation is the most robust of all the PSKs since it takes the highest level of noise or distortion to make the demodulator reach an incorrect decision. It is, however, only able to modulate at 1 bit/symbol.

1.2 Convolutional Codes

A convolutional code of rate $R = \frac{k_c}{n_c}$ is a linear function which, at every instant i, transforms an input symbol d_i of k_c bits into an output coded symbol X_i of n_c bits $(k_c > n_c)$. A code is called systematic if a part of the output is made up of systematic bits $s_i = d_i$ and the rest of the bits $(k_c - n_c)$ are made up of parity.

The structure of the convolutional code is constructed with shift registers (made up of ν flip flops) and the modulo two adders (XOR). A code is characterized by a parameter called constraint length $K = \nu + 1$ where value of all ν represents one of the 2^ν states of the encoder. The code can also be called recursive if there is a feedback to the shift registers.

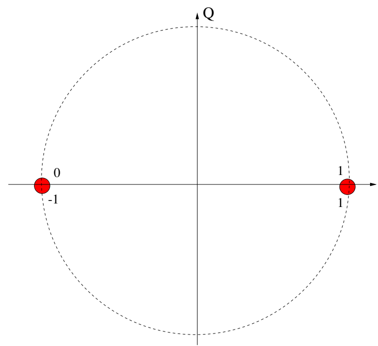

Figure 1.2: BPSK constellation

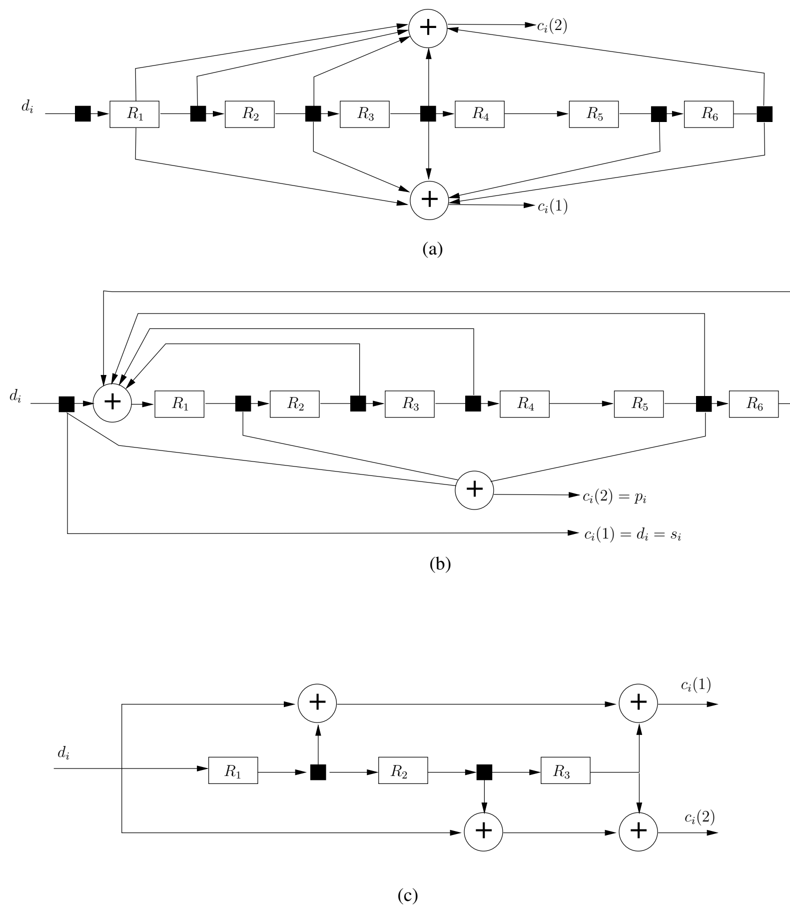

Figure 1.3: Convolutional encoder (a) 64-state single binary code (b) 64-state single binary RSC (c) 8-state single binary code

A convolutional code can be expressed by the generation polynomials which represents the connections between output of the registers and the modulo two adders. Figure 1.3(a) represents a single binary non-systematic, non-recursive, 64-state encoder whose generation polynomials are: $g_1(x) = 1 + x^2 + x^3 + x^5 + x^6$ and $g_2(x) = 1 + x + x^2 + x^3 + x^7$. These polynomials can be represented by their coefficients, (1011011) and (1111001) respectively in binary or $(133; 171)_8$ in octal. The encoder shown in Figure 1.3(b) is a single binary Recursive Systematic Convolutional (RSC) code whereas the one shown in Figure 1.3(c) is 8 states single binary code.

1.2.1 Recursive Systematic Convolutional Codes

The class of Recursive Systematic Convolutional codes (RSC) is of special interest for Turbo codes. RSC codes introduce feedback into the memory of convolutional codes. RSC code is obtained by introducing a feedback loop in the memory (recursive) and by transmitting directly the input of the encoder as an output (systematic). For RSC code, the input of the shift register is a combinatory function of the input block d_k and its delayed versions d_{i-1}, d_{i-2},..., $d_{i-\nu}$. It is worth noting that the set of codewords generated by the RSC encoder is the same as that generated by the corresponding non-systematic encoder. However, the correspondence between the information sequences and the codewords is different. A binary RSC code can be obtained from a binary non-systematic non-recursive convolutional code by replacing the input of the linear shift register by one of the outputs of the non-systematic convolutional code. The corresponding output is replaced by the input of the encoder. This construction leads to an implementation of a Linear Feedback Shift Register (LFSR) as represented in In Figrue (1.4-b). The polynomial generator matrix of the RSC encoder shown in (1.4):

$$g_{1(x)} = \left[1, \frac{1+x^2+x^3}{1+x+x^3}\right] \tag{1.3}$$

RSC codes are used as component codes for Turbo codes, because of their infinite impulse response: if a binary RSC encoder is fed with a single ”1” and then only with ”0”, due to the feedback loop, the weight of the parity sequence produced is infinite. For a non recursive encoder, only a maximum of ν ”1” bits are generated, then the ”1” comes out of the shift register.

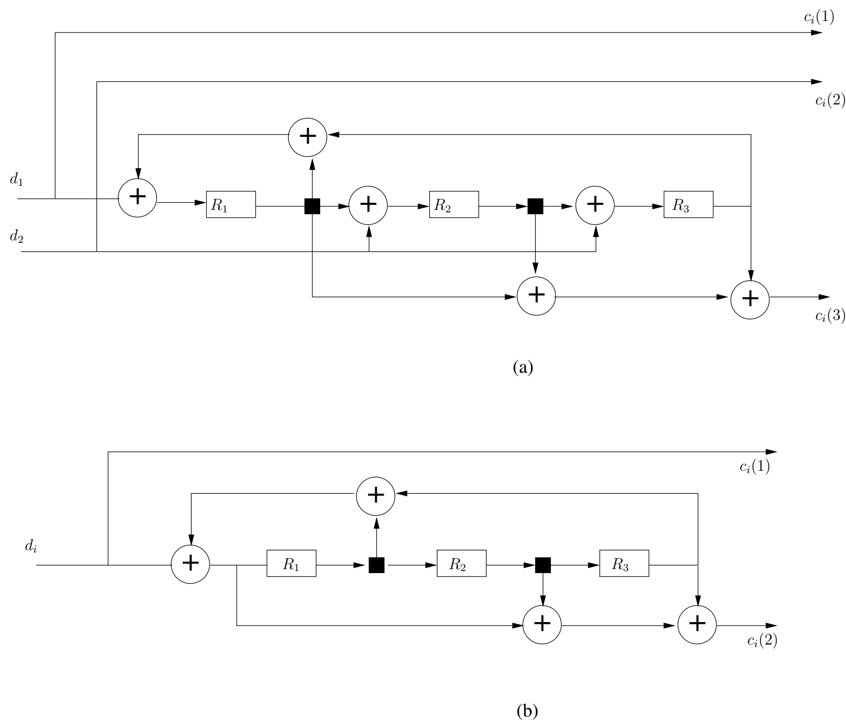

Figure 1.4: RSC encoder (a) 8-state double binary code RSC (b) 8-state single binary RSC

1.2.2 Trellis Representation

Although a convolutional code can be represented graphically in many ways but the trellis representation [2] is the most popular one. It is the most commonly used diagram for illustrating the decoding principle of convolutional codes. The trellis diagram for the convolutional encoder presented in Figure 1.3(c) is represented in Figure 1.5. The x-axis represents the discrete time k and the y-axis the 2^{ν} states of the convolutional encoder. The induced discrete points correspond to the possible successive states of the encoder. The transitions between the states are represented by branches oriented from left to right. The trellis diagram is thus constructed by connecting through the branches all possible evolutions from one state to another. The time evolution of the encoder, for a given information sequence, is viewed on the trellis diagrams by a succession of states connected through the branches. This connected sequence is called a path in the trellis. A sequence of information and coded bits is associated with each path. The decoder uses observations on the coded sequence and tries to recover the associated path in the trellis, thus yielding the decoded information sequence.

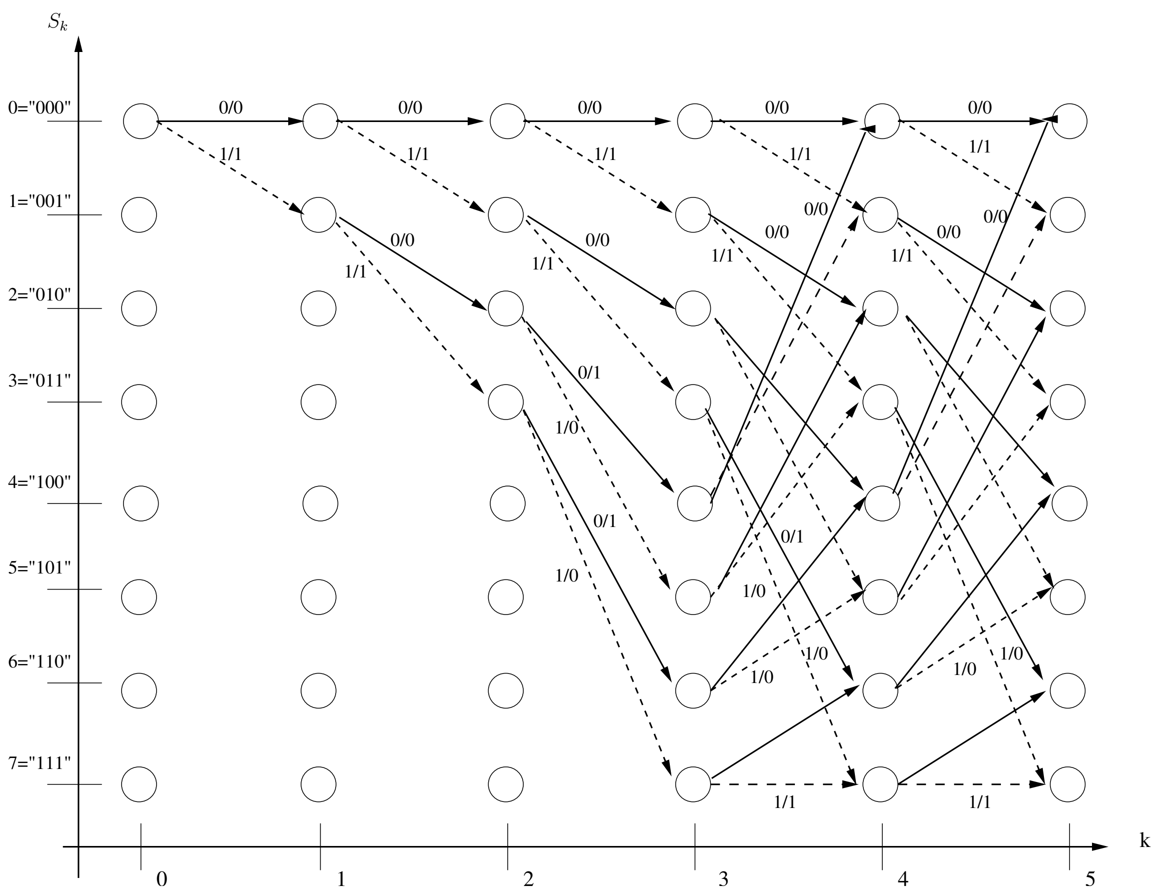

Figure 1.5: Trellis diagram of the convolutional encoder of Figure 1.3(c)

1.3 Turbo Codes

It was known since a long time how to create codes with a correction power ensuring reliable transmission for most applications. However, this knowledge could not be turned into implementation due to the prohibitive complexity of decoders which are associated to such codes.

Now, when a problem is too complex, the approach of ”divide and conquer” can be a good way to simplify it. Based on this approach, the concatenation of codes has been proposed. The idea, introduced by [3], is to build a code with a sufficient correction power from several simple codes. This section briefly introduces the different proposed concatenation techniques of convolutional codes.

1.3.1 Concatenation of Convolutional Codes

In the first architecture Forney concatenated the internal code to an external code as shown in Figure 1.6(a). This is called serial concatenation convolutional codes (SCCC) where output of outer code in input of the inner code. Subsequently, it was observed that the addition of a function of interleaving between the two codes will increase significantly robustness of the concatenated codes. This interleaver plays a crucial role in the construction of the code, because it performs a pseudo random permutation of the latter input sequence, which introduces randomness in the encoding scheme. Therefore what is called nowadays a serial concatenated convolutional codes is more like a representation of Figure 1.6(b). With the advent of Turbo codes [4], a new structure was introduced: the parallel concatenation convolutional codes (PCCC) of two RSC encoders (RCS1, RCS2) presented in Figure 1.6(c). This structure is associated to systematic encoders where the first encoder receives the source data d_i in natural order and at the same time the second encoder receives the interleaved one. The input sequence d_i is encoded twice, once by each encoder. An interleaver Π is performed on the information sequence prior to the RCS2 encoding. In other words, the two constituent RCS encoders code the same information sequence but in a different order. Randomness for error correcting codes is a key attribute enabling us to approach channel capacity. The output is composed of source data and associated parities in natural and interleaved domains. In this way at one instant of time parity of two different symbols are transmitted. They are called ”Turbo codes” in reference to their practical iterative decoding principle, by analogy with the Turbo principle of a Turbo-compressed engine. This latter actually uses a retro-action principle by reusing the exhaust gas to improve its efficiency. The decoding principle is based on an iterative algorithm where two component decoders exchange information improving the error correction efficiency of the decoder during the iterations. At the end of the iterative process, after several iterations, both decoders converge to the decoded codeword, which corresponds to the transmitted codewords when all transmission errors have been corrected.

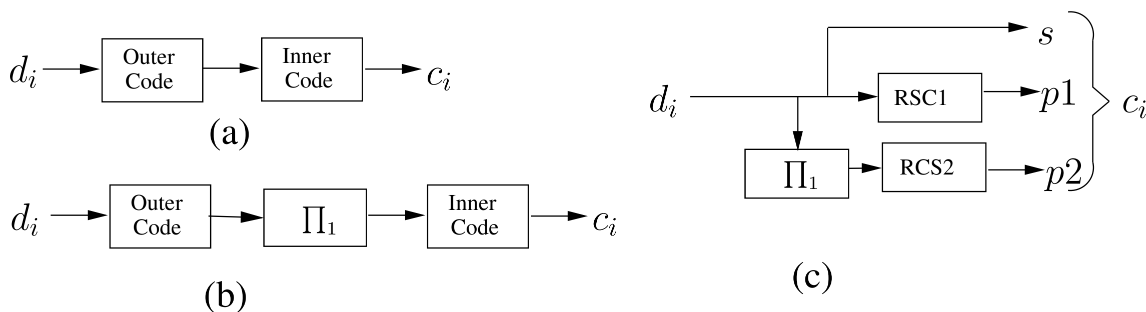

Figure 1.6: Turbo encoder examples

The supported types of channel coding for Turbo codes are usually Single Binary and/ Double Binary Turbo Codes (SBTC and DBTC). We will mainly focus on the Turbo codes defined in 3GPP LTE and WiMAX in the following chapters because the proposed decoder support them.

Binary Turbo Code in 3GPP LTE Standard Turbo coding scheme in 3GPP LTE standard is a parallel concatenated convolutional code (PCCC) with two 8-state constituent encoders and one Quadratic Permutation Polynomial (QPP) interleaver. The coding rate of the Turbo code is 1/3. The structure of the Turbo encoder is shown in Figure 1.7. As seen in the figure, a Turbo

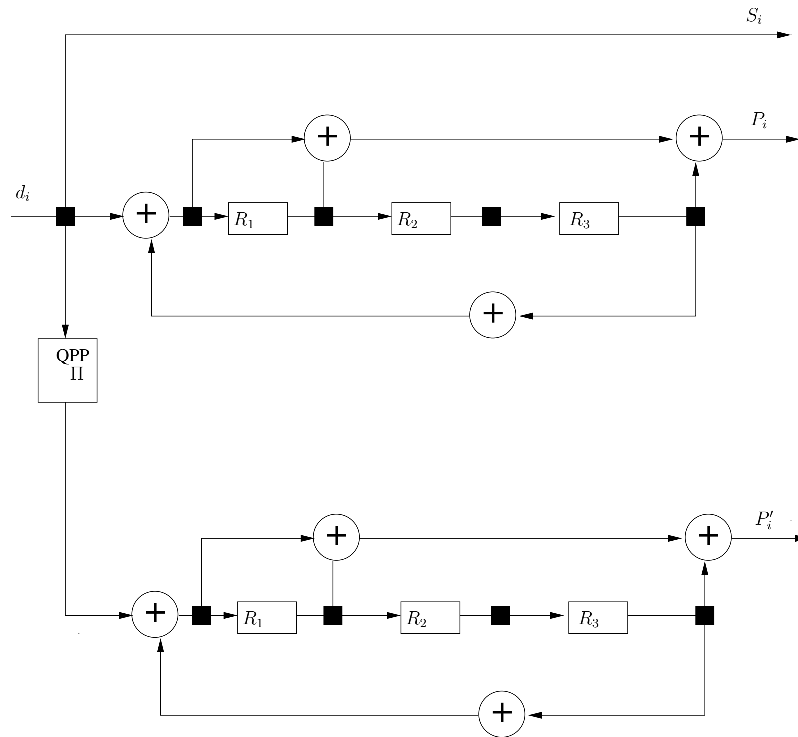

Figure 1.7: Structure of rate 1/3 Turbo encoder in 3GPP LTE

encoder consists of two binary convolutional encoders connected by an interleaver. The basic coding rate is 1/3 which means N data bits will be coded into 3N data bits. The transfer function of the 8-state constituent code for PCCC is:

$$g(x) = \left[1, \frac{1 + x + x^3}{1 + x^2 + x^3}\right] \tag{1.4}$$

The initial value of the shift registers of the 8-state constituent encoders shall be all zeros when starting to encode the input bits. Trellis termination is performed by taking the tail bits from the shift register feedback after all information bits are encoded. Tail bits are padded after the encoding of information bits. The function of the Interleaver is to take each incoming block of N data bits and shuffle them in a pseudo-random manner. One of the new features in the 3GPP LTE Turbo encoder is its quadratic permutation polynomial (QPP) internal interleaver. Figure 1.8 shows trellis representation for WiMAX and DVB-RCS, with nodes corresponding to eight states and the transitions between the states are represented by branches oriented from left to right with 2^1 branches for each state.

Double Binary Turbo Code in IEEE 802.16e WiMAX Standard The convolutional Turbo encoder for the IEEE 802.16e standard is depicted in Figure 1.10. It uses a double binary circular recursive systematic convolutional code. Data couples (d_1, d_2), other than a single bit sequence, are fed to the circular recursive systematic convolutional encoder twice, and four parity bits

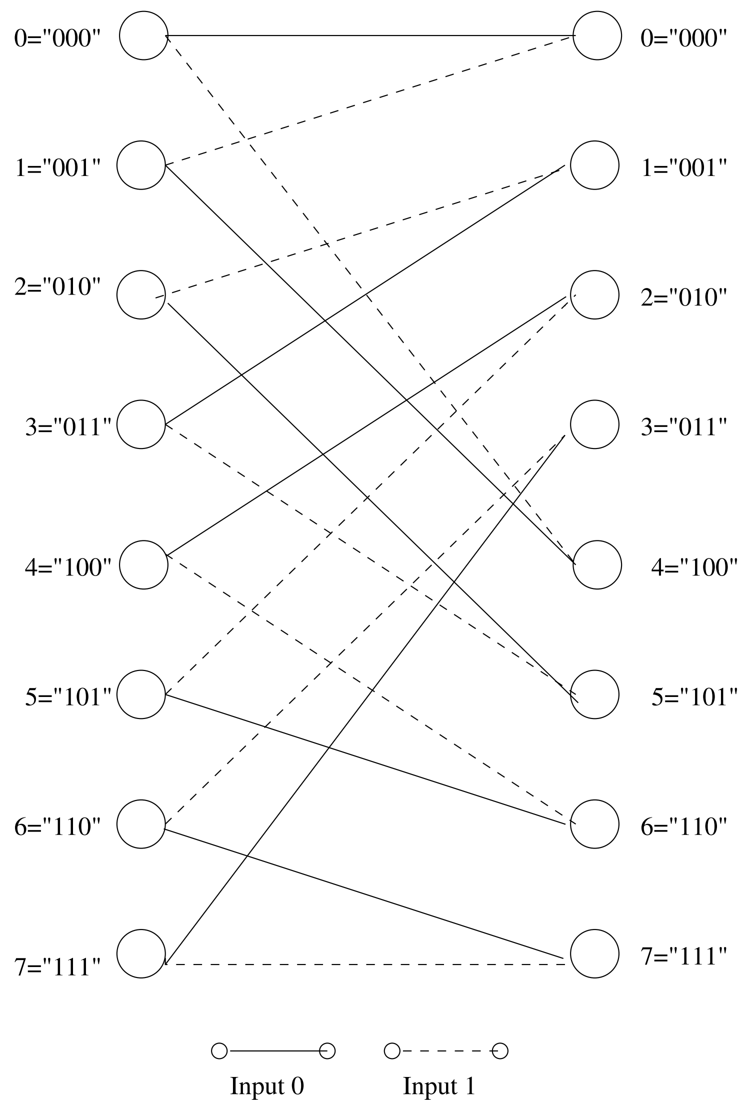

Figure 1.8: Trellis for 3GPP (Single binary 8 states)

(P_1,P_2) and (P_1', P_2') are generated in the natural order and the interleaved order, respectively. The encoder polynomials are described in binary symbol notation as follows:

- For the feedback branch: $1 + x + x^3$

- For the P parity bit: $1 + x^2 + x^3$

- For the P' parity bit: $1 + x^3$

The tail-biting trellis termination scheme is used as opposed to inserting extra tail bits. In this termination scheme, the start state of the trellis equals to the end state of the trellis. Therefore, a pre-encoding operation has to be performed to determine the start state. This is not a complex problem because the encoding process can be performed at a much higher rate. A symbol-wise

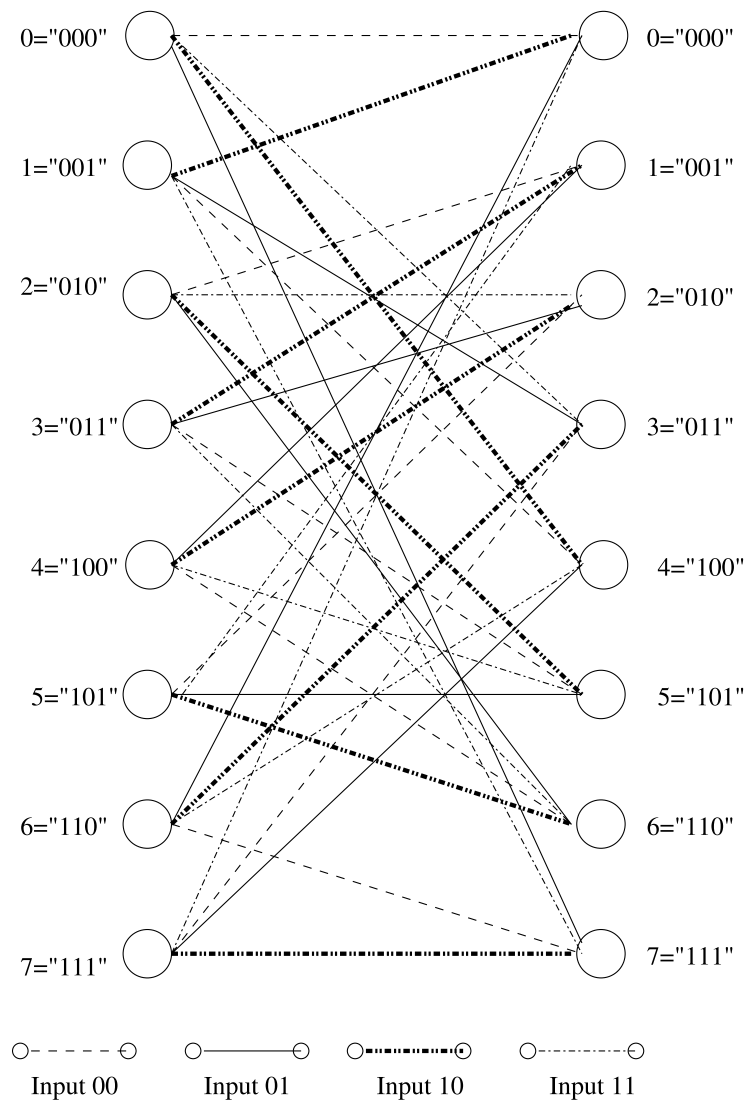

Figure 1.9: Trellis for WiMAX/DVB-RCS (double binary 8 states)

almost regular permutation (ARP) interleaver is used in the WiMAX standard, which can enable parallel decoding of double binary Turbo codes. Figure 1.10 shows trellis representation for WiMAX and DVB-RCS, with nodes corresponding to eight states and the transitions between the states are represented by branches oriented from left to right with 2^2 branch for each state.

1.3.2 Turbo Code Interleaver ($\prod$)

Interleavers in a digital communication system are used to temporally disperse the data. The primary interest of them in concatenated codes is to put two copies of same symbol (coming to two encoders) at different interval of time. This enables to retrieve at least one copy of a symbol in a situation where the other one has been destroyed by the channel. An interleaver ($\prod$) satisfying this property can be verified by studying the dispersion factor S given by the minimum distance between two symbols in natural order and interleaved order:

$$S = \min_{i,j} \left(|i - j| + |\Pi(i) - \Pi(j)| \right) \tag{1.5}$$

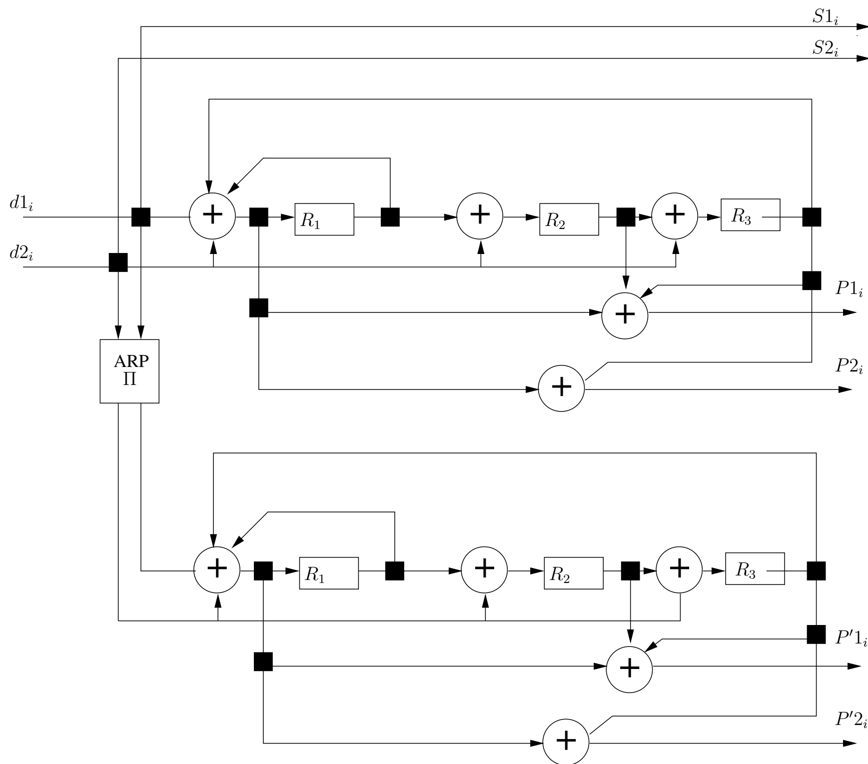

Figure 1.10: Structure of rate 1/3 double binary Turbo encoder

The design of interleavers respecting a dispersion factor can be reasonably achieved through the S-random algorithm proposed in [5]. However, even if this kind of interleaver can be sufficient to validate the performance in the convergence zone of a code, it does not achieve a good asymptotic performance. Therefore to improve the latter, the design of the interleaver must also take into account the nature of component encoders. Complexity of the hardware implementation should, in addition, be taken into account. In fact, the recent wireless standards specify performance and hardware aware interleavling laws for each supported frame length.

In following sections the interleaving functions associated to Turbo codes for different standards are described.

1.3.2.1 Almost regular permutation (ARP)

The ARP interleaver is used in double binary Turbo codes for both standards IEEE 802.16e WiMAX and DVB-RCS. It can be described by the function $\prod(j)$ which provides the interleaved address of each double-binary symbol of index j, where $j = 0, 1, ...N - 1$ and N is the number of symbols in the frame.

$$\prod(j) = (P_0 \times j + P + 1) \; mod \; N \tag{1.6}$$

where

$$
\begin{aligned}
P &= 0 && \text{if } j \ mod\ 4 = 0 \\
P &= \frac{N}{2} + P_1 && \text{if } j \ mod\ 4 = 1 \\
P &= P_2 && \text{if } j \ mod\ 4 = 2 \\
P &= \frac{N}{2} + P_3 && \text{if } j \ mod\ 4 = 3
\end{aligned}
\tag{1.7}
$$

where the parameters P_0, P_1, P_2 and P_3 depend on the frame size and are specified in the corresponding standard [6][7].

Another step of interleaving is specified in these standards which consists of swapping the two bits of alternate couples, i.e $(a_j, b_j) = (b_j, a_j)$ if $j \ mod\ 2 = 0$.

It is worth to note that this interleaver structure is well suited for hardware implementation and presents a collision-free property for certain level of parallelism.

1.3.2.2 Quadric polynomial permutation (QPP)

The interleaver used in single binary Turbo code for standard 3GPP-LTE is called quadric polynomial permutation (QPP) interleaver. It is given by the following expression:

$$\prod(j) = (f_1 j + f_2 j^2) \ mod\ N \tag{1.8}$$

where the parameters f_1 and f_2 are integers, depend on the frame size N ($0 \leq j, f_1, f_2 < N$), and specified in the 3GPP-LTE standard. In this standard, all the frame sizes are even numbers and are divisible by 4 and 8. Moreover, By definition, parameter f_1 is always an odd number whereas f_2 is always an even number. Through further inspection, we can mention one of the several algebraic properties of the QPP interleaver:

$\prod(j)$ has the same even/odd parity as j as shown in 1.9 and 1.10:

$$\prod(2 \times k) \ mod\ 2 = 0 \tag{1.9}$$

$$\prod(2 \times k + 1) \ mod\ 2 = 1 \tag{1.10}$$

This property will be used later in hardware implementation in order to design an extrinsic exchange module to avoid memory collisions. More information on the other properties of QPP interleaver are given in [8][9].

1.3.3 Multi Standard Channel Convolutional and Turbo Coding Parameters

There is a large variety of coding options specified in existing and future digital communication standards, besides the increasing throughput requirement. Table 1.1 presents some Convolutional and Turbo Coding standards and the parameters associated to them.

1.4 Turbo Decoding

On the receiver side the objective is to remove the channel effects to retrieve the original source data. This objective is achieved by exploiting the redundancy and diversity added to source data

Standard	Codes	Rates	States	Block size	Channel Throughput
IEEE-802.11 (WiFi)	CC	1/2 - 3/4	64	1 -4095	6 - 54 Mbps
	CC	2/3	256	.. 1944	.. 450Mbps
IEEE802.16 (WiMAX)	CC	1/2 - 7/8	64	.. 2040	.. 54 Mbps
	DBTC	1/2 - 3/4	8	.. 4800	.. 75 Mbps
DVB-RCS	DBTC	1/3 - 6/7	8	.. 1728	.. 2 Mbps
3GPP-LTE	SBTC	1/3	8	.. 6144	.. 150 Mbps

Table 1.1: Selection of wireless communication standards and channel codes

in the transmitter. In an iterative receiver there are feedback paths in addition to forward paths, through which, constituent units can send the information to previous units iteratively. A Soft In Soft Out (SISO) processing block of an iterative receiver, using channel information and the information received from other units, generates soft outputs at each iteration. As for this thesis the iterative processes for Turbo decoding will be considered.

Iterative/Turbo decoding of Turbo codes involves an exchange of information between constituent component decoders. This exchange is enabled by linking the *a priori* soft probability input port of one component decoder to the extrinsic soft probability output port provided by the other component decoder (Figure 1.11). Only the extrinsic information is transmitted to the other decoder, in order to avoid the information produced by the one decoder being fed back to itself. This constitutes the basic principle of iterative decoding. The SISO decoders are assumed to process soft values of transmitted bits at their inputs. Each SISO decoder computes the extrinsic information related to information symbols, using the observation of the associated systematic (S) and parity symbols (P) coming from the transmission channel and the *a priori* information. Since no *a priori* information is available from the decoding process at the beginning of the iterations they are not used. For the subsequent iterations, the extrinsic information coming from the other decoder are used as *a priori* information for the current SISO decoder. The decisions can be computed from any of the decoders. The SISO processes are executed in a sequential fashion. The decoding process starts arbitrarily with either one decoder, DEC1 for example. After DEC1 processing is completed, DEC2 starts processing and so on.

1.4.1 Soft In Soft Out (SISO) Decoding

Consider the system diagram of Figure 1.12 where source is encoded by convolutional parallel concatenated Turbo encoder which outputs the source (systematic and the parity bits). These encoded bits modulated by passing through a mapper which applies Binary Phase Shift Keying (BPSK) modulation to produce modulated symbol X. Then additive white Gaussian noise is added to these symbols due to passing through Gaussian channel (AWGN). The received symbols Y which are corrupted due to noise form the soft input to the Turbo decoder. The Turbo decoder as explained in previous section is composed of two component decoders DEC1 and DEC2 responsible for the error correction.

In SISO decoding, the term ”Soft-in” refers to the fact that the incoming data may take values other than 0 or 1, in order to indicate reliability. ”Soft-out” refers to the fact that each bit in the decoded output also takes a value indicating reliability. Reviewing the history of decoding algorithms, several techniques have been proposed to decode a convolutional codes such as Viterbi algorithm, proposed by Andrew J. Viterbi [10], and Fano algorithm, proposed by Robert Mario Fano [11]. Both algorithms have considered initially binary inputs and outputs. The

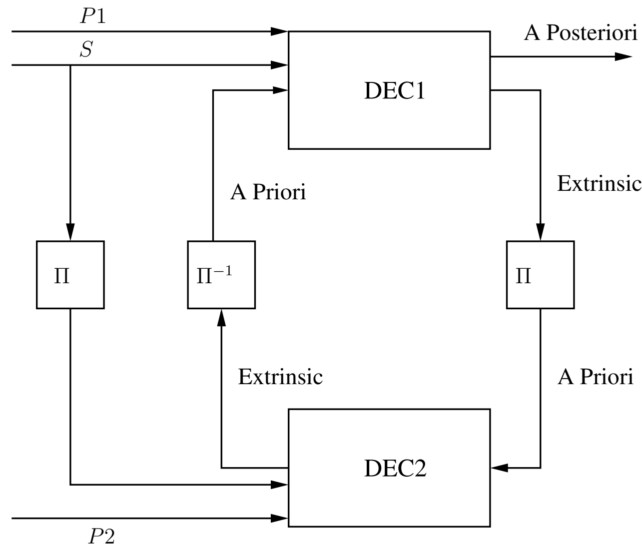

Figure 1.11: Turbo decoder principal

Viterbi algorithm was then modified to accept the soft inputs to improve the decoding [12]. The Soft Output Viterbi Algorithm (SOVA) [13] differs from the classical Viterbi algorithm in that it takes the soft input and provides the soft output by weighting two concurrent paths associated with the hard decision. The CockBahl-Jelinek-Raviv (BCJR) [14] also called MAP (Maximum *A Posteriori*) or forward backward algorithm, is the optimal decoding algorithm which calculates the probability of each symbol from the probability of all possible paths in the trellis between initial and final states. In practice, due to its complexity, implementing the BCJR algorithm is difficult to hardware application, but rather the algorithm is simplified so the logarithmic domain transforms the multiplications into additions. Hence these simplified versions are named as Log-MAP or in sub optimal form as Max-Log-MAP algorithm [15]. The comparison of the SOVA and Max-Log-MAP algorithm in terms of performance and complexity for Turbo codes have been addressed in [15]. The Max-Log-Map algorithm which involves two Viterbi recursions is about twice the complex as SOVA. But, the SOVA has a degradation of 0.2-0.4 dB compared to the Max-Log-MAP algorithm.

1.4.1.1 MAP decoding algorithm

Figure 1.12 shows a source that generate frame of symbols d, where each symbol d_i is composed of n bits. This frame is encoded with an encoder having ν memory element (i.e 3) states, which generates l bits at rate $r = \frac{n}{l}$. On the other hand the MAP decoder generates 2^n *a posteriori* probabilities with respect of the sequence Y received by the decoder, in other word a decision for each possible value of symbol. i.e for double binary symbols ($n = 2$) we have four decisions for each possible value of symbol $d_i \in (00, 01, 10, 11)$. The hard decision is the corresponding value j that maximizes the *a posteriori* probability. These probabilities can be expressed in terms of joint probabilities.

$$Pr(d_i \equiv j|Y) = \frac{p(d_i \equiv j, Y)}{\sum_{k=0}^{2^n-1} p(d_i \equiv k, Y)} \tag{1.11}$$

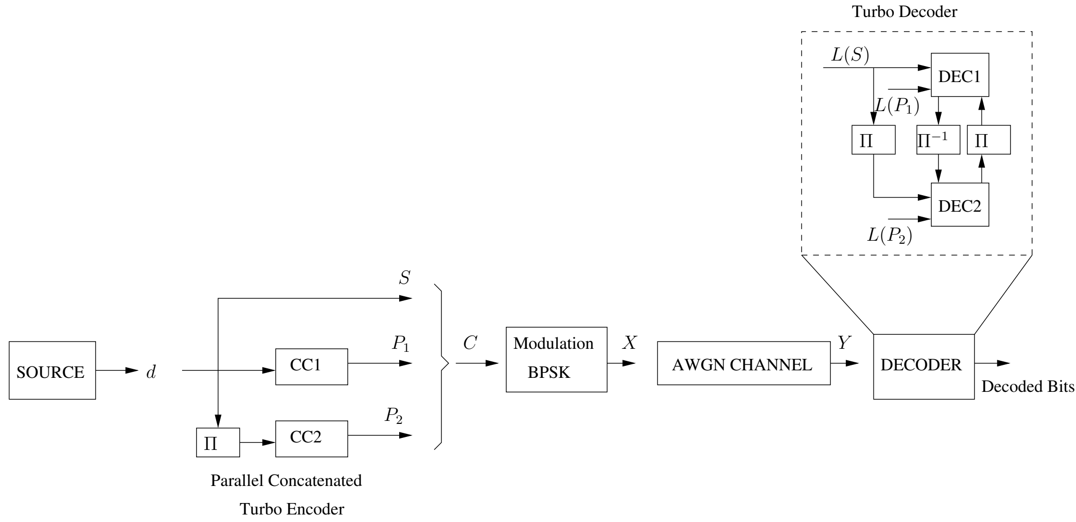

Figure 1.12: System diagram with Turbo encoder, BPSK modulator, AWGN channel and Turbo decoder

The calculation of joint probabilities can be disintegrated between past and future observations due to the Trellis structure of the code. This disintegration utilizes the forward recursion metric $a_i(s)$ (the probability of a state of the trellis at instant i computed from past values), backward recursion metric $b_i(s)$ (the probability of a state of the trellis at instant i computed from future values), and a metric $c_i(s', s)$ (the probability of a transition between two states of the trellis). Using these metrics the expression of 1.11 becomes:

$$p(d_i \equiv j, Y) = \sum_{(s',s)/d_i \equiv j} b_{i+1}(s)a(s')c_i(s', s) \tag{1.12}$$

The forward recursion metrics is calculated in following way:

$$a_{i+1}(s) = \sum_{s'=0}^{2^\nu - 1} a_i(s')c_i(s', s), \text{ for } i = 0 \ldots N - 1 \tag{1.13}$$

The backward recursion metrics is calculated in following way:

$$b_i(s) = \sum_{s'=0}^{2^\nu - 1} b_{i+1}(s')c_i(s', s), \text{ for } i = 0 \ldots N - 1 \tag{1.14}$$

The initialization value of these metrics can be defined by knowing the initial and final state of the trellis, e.g if the encoder starts at state s_0 then $a_0(s_0)$ has the highest probability value 1 while other $a_0(s)$ will be 0. If the initial state is unknown then all states are initialized to same equiprobable value.

Similarly the branch metric can be expressed in the following way:

$$c_i(s', s) = p(Y_i|X_i) \cdot Pr^a(d_i = d_i(s', s)) \tag{1.15}$$

where $p(Y_i|X_i)$ is represents the channel transition probability which can be expressed for a Gaussian channel and formula for Gaussian channel already presented previously in (1.1), where X_i is the ith transmitted symbol after modulation and Y_i is the ith transmitted symbol after adding Gaussian noise.

The *a priori* probability $Pr^a(d_i = d_i(s', s)$, to emit m-ary information corresponding to transition from s' to s is 0 if the transition does not exist in the trellis. Otherwise its value depends upon the statistics of the source. For an equiprobable source $Pr^a(d_i = j) = \frac{1}{2^n}$. In the context of the Turbo decoding, the *a priori* probability takes into account the input extrinsic information.

The decoder generates the extrinsic information which can be calculated the same way the *a posteriori* information is computed in (1.11) but with a modified branch metric:

$$Pr(d_i \equiv j|Y) = \frac{\sum\limits_{(s',s)/d_i \equiv j} b_{i+1}(s)a(s')c_i^{ext}(s',s)}{\sum\limits_{(s',s)} b_{i+1}(s)a(s')c_i^{ext}(s',s)} \quad (1.16)$$

Hence the branch metric does not take into account the already available information of a symbol for which extrinsic information is being generated. For parallel convolutional Turbo codes, systematic part is removed from the branch metric computation and can be expressed as:

$$c_i^{ext}(s',s) = K \cdot e^{\frac{\sum\limits_{k=n+1}^{l} Y_{i,k} \cdot X_{i,k}}{\sigma^2}} \quad (1.17)$$

1.4.1.2 Log-MAP and Max-Log-MAP decoding algorithm

As already mentioned earlier, the MAP algorithm is likely to be considered too complex for implementation in real system. The Log-MAP algorithm which is introduced by [15], is the direct transformation of MAP algorithm in logarithmic domain. Hence, all the metrics M of MAP algorithm will be replaced by metrics $\sigma^2 lnM$. This permits to transform the semi ring sum-product $(R+, +, \times, 0, 1)$ to semi ring $(R^+, max^*, +, 1, 0)$ where the max^* operator is defined as below:

$$max^*(x,y) = \sigma^2 ln(e^{\frac{x}{\sigma^2}} + e^{\frac{y}{\sigma^2}}) = max(x,y) + \sigma^2 ln\left(1 + e^{-\frac{|x-y|}{\sigma^2}}\right) \approx max(x,y) \quad (1.18)$$

This operator can be simplified by an operator max which corresponds to Max-Log-MAP algorithms. Using this approximation the branch metrics of (1.15) and (1.17) can be written as:

$$\gamma_i(s',s) = \sigma^2 ln(c_i(s',s)) = K' + L_i^a(j) + \sum_{k=1}^{l} Y_{i,k} X_{i,k} = \gamma_i^{ex}(s',s) + L_i^a(j) + L_i^{sys}(j) \quad (1.19)$$

and

$$\gamma_i^{ex}(s',s) = \sigma^2 ln(c_i^{ex}(s',s)) = K' + \sum_{k=n+1}^{l} Y_{i,k} \cdot X_{i,k} \quad (1.20)$$

where $L_i^a(j)$ is the metric of *a priori* information and $L_i^{sys}(j)$ is the metric which corresponds to systematic part of the data. It is interesting to note that constant K' is not necessary in practice because it is removed while computing (1.13)and (1.14).

In the same way the forward and backward recursion metrics can be written as:

$$\alpha_{i+1}(s) = \sigma^2 ln(a_{i+1}(s)) = \max_{s'=0}^{2^\nu-1}{}^*(\alpha_i(s') + \gamma_i(s', s)) \text{ for i = 0 .. N-1} \tag{1.21}$$

$$\beta_i(s) = \sigma^2 ln(b_i(s)) = \max_{s'=0}^{2^\nu-1}{}^*(\beta_i(s') + \gamma_i(s', s)) \text{ for i = 0 .. N-1} \tag{1.22}$$

In this case the first and last metrics are initialized in following way :

* in case state S is known, $a(S) = 0$ while the others are $a(s \neq S) = -\infty$.
* in case state is unknown, all the states will have $a(s) = 0$.

The *a posteriori* (2.1) and extrinsic (2.7) informations are transformed into:

$$\begin{aligned} Z_i(j) = \sigma^2 lnPr(d_i \equiv j|Y) \\ = \max_{(s',s)/d_i\equiv j}(\alpha_i(s') + \gamma_i(s', s) + \beta_{i+1}(s)) - \max_{s',s}{}^*(\alpha_i(s') + \gamma_i(s', s) + \beta_{i+1}(s)) \end{aligned} \tag{1.23}$$

$$\begin{aligned} Z_i^{ex}(j) = \max_{(s',s)/d_i\equiv j}(\alpha_i(s') + \gamma_i^{ex}(s', s) \\ +\beta_{i+1}(s)) - \max_{s',s}{}^*(\alpha_i(s') + \gamma_i^{ex}(s', s) + \beta_{i+1}(s)) \end{aligned} \tag{1.24}$$

By simplifying the terms on the right side of the metrics, using the most probable symbol $\hat{j}$ in the following way, one will have:

$$\begin{aligned} \max_{(s',s)}{}^*(\alpha_i(s') + \gamma_i(s', s) + \beta_{i+1}(s)) = \\ \max_{j=0}^{n}{}^* \left(max(\alpha_i(s') + \gamma_i(s', s) + \beta_{i+1}(s))\right) \end{aligned} \tag{1.25}$$

$$\begin{aligned} \max_{(s',s)}{}^*(\alpha_i(s') + \gamma_i(s', s) + \beta_{i+1}(s)) &= \max_{j=0}^{n}{}^* \left(\max_{(s',s)/d_i\equiv j}{}^*(\alpha_i(s') + \gamma_i(s', s) + \beta_{i+1}(s))\right) \\ &\approx \max_{j=0}^{n} \left(\max_{(s',s)/d_i\equiv j}{}^*(\alpha_i(s') + \gamma_i(s', s) + \beta_{i+1}(s))\right) \\ &= \max_{(s',s)/d_i\equiv \hat{j}}{}^*(\alpha_i(s') + \gamma_i(s', s) + \beta_{i+1}(s)) \end{aligned} \tag{1.26}$$

and by distributivity :

$$\begin{aligned} \max_{(s',s)/d_i\equiv j}{}^*(\alpha_i(s') + \gamma_i(s', s) + \beta_{i+1}(s)) = \\ L_i^a(j) + L_i^{sys}(j) + \max_{(s',s)/d_i\equiv j}{}^*(\alpha_i(s') + \gamma_i^{ex}(s', s) + \beta_{i+1}(s)) \end{aligned} \tag{1.27}$$

In the context of this simplification the metrics can be written in the following way:

$$L_i(j) = L_i^a(j) + L_i^{sys}(j) + L_i^{ex}(j) - L_i^{ex}(\hat{j}) - L_i^{sys}(\hat{j}) \tag{1.28}$$

Finally the simplified formulas that are adopted in this work are:

$$Z_i^{ex} = \Gamma \times (Z_i^{apos} - \gamma_i^{int}(s', s)) \tag{1.29}$$

$$Z_i^{apos} = \max_{(s',s)/d_i} (\alpha_{i-1}(s) + \gamma_i^{ex}(s', s) + \beta_i(s)) \tag{1.30}$$

Where Γ is the scaling factor to normalize the extrinsic information to avoid overloading in hardware implementation and its value change regarding the applied standard.

$$\alpha_i(s) = \max_{s',s}(\alpha_{i-1}(s) + \gamma_i(s', s)) \tag{1.31}$$

$$\beta_i(s) = \max_{s',s}(\beta_{i+1}(s) + \gamma_{i+1}(s', s)) \tag{1.32}$$

$$\gamma_i(s', s) = \gamma_i^{int}(s', s) + \gamma_i^{ex}(s', s) \tag{1.33}$$

$$\gamma_i^{int}(s', s) = \gamma_i^{sys}(s', s) + \gamma_i^{par}(s', s) \tag{1.34}$$

$$Z_i^{Hard.dec} = sign(Z_i^{apos}) \tag{1.35}$$

For double binary Turbo codes, the three Log-Likelihood Ratios (LLR) can be normalized as defined by (1.36) where $k \in (01, 10, 11)$ of the ith symbol. s' and s are the previous and current trellis state and $d(s', s)$ is the decision respectively.

$$Z_i^{ex}(d(s, s') = k) = Z_i^{ex}(d(s, s') = k) - Z_i^{ex}(d(s, s') = 00) \tag{1.36}$$

This simplification also known as Log-MAP introduces a very low loss of performance (0.05dB) as compared to MAP algorithm. Although sub-optimal Max-Log-MAP algorithm provides a loss of 0.1dB for double binary code [16] yet it provides many advantages. Firstly it eliminates the logarithmic term from (1.18) which on one hand reduces the implementation complexity (in practice this part is saved in Look Up Tables) and on the other hand reduces the critical path and hence increases the operational frequency as compared to max operator. Moreover, for Max-Log-MAP algorithm the knowledge of σ^2 is not required.

1.4.2 Parallelism in Turbo Decoding

Exploring parallelism in iterative processes with maintaining the error rate performance is very important to satisfy high transmission rate requirement. This section will present the available parallelism in Turbo decoding. A comprehensive study is already presented in [17], so we will briefly summarize it and also explain *Radix-4* technique which is missing in that study.
The Max-Log-MAP algorithm that is executed in Turbo decoder can allow for different techniques of parallelism, these techniques can be categorised in three levels. Each level implies different requirements in terms of memory and logic units when targeting a hardware implementation. These levels are:

- BCJR metric level parallelism.

- SISO decoder level parallelism.
- Turbo decoder level parallelism.

The first parallelism level concerns symbol elementary computations inside a SISO decoder processing the BCJR algorithm. The second parallelism level concerns parallelism between SISO decoders, inside the same Turbo decoder. The third parallelism level duplicates the Turbo decoder hardware itself.

1.4.2.1 BCJR metric level parallelism

This level exploits both the parallelism of inherent structure of the trellis [18] [19], one-level look-ahead trellis transform [20], and the parallel calculations of BCJR [18][21] [19].

Parallelism of trellis transitions: Trellis transition parallelism is the first parallelism available in Max-Log-MAP algorithm. This parallelism is computation of the metrics associated to each transition of a trellis regarding γ, α, β, and the extrinsic information. Trellis-transition parallelism can be obtained from trellis structure as the same operations are repeated for all transition pairs. For Max-Log-MAP algorithm these operations are add-compare-select (ACS) operations. The number of (ACS) operations for BCJR calculation (γ, α, β, and Extrinsic) bounded by the number of transitions in the trellis. i.e. for double binary Turbo codes of 8 states there are 32 transitions (8 states $\times$ 4 branch metrics per state). However in practice for (γ) calculation, the degree of parallelism associated with calculating the branch metric is bounded by the number of possible binary combinations of input and parity bits. Therefore, many transitions could have the same probability in a trellis if they have the same binary combination. i.e. for double binary Turbo codes of 8 states there are 16 different binary combinations of input and parity bits.

On the other hand another advantage of this parallelism, that it implies low area overhead because the only duplicated part is the computational units. In particular, no additional memories are required since all the parallelized operations are executed on the same trellis section, and in consequence on the same data.

Parallelism of BCJR computations: A second metric parallelism can be orthogonally extracted from the BCJR algorithm through a parallel execution of the three BCJR computations (α, β , and extrinsic computation). Parallel execution of backward recursion and extrinsic computations was proposed with the original Forward-Backward scheme, depicted in Figure 1.13(a). So, in this scheme, we can notice that BCJR computation parallelism degree is equal to one in the forward part and two in the backward part. To increase this parallelism degree, several schemes are proposed [21]. Figure 1.13(b) shows the *butterfly scheme* which doubles the parallelism degree of the original scheme through the parallelism between the forward and backward recursion computations. In conclusion, BCJR metric level parallelism achieves optimal area efficiency as it does not affect memory size, which occupies most of the area in a Turbo decoder circuit. Thus, a hardware implementation achieving high throughput should first exploit this parallelism. However the parallelism degree is limited by the decoding algorithm and the code structure. Thus, achieving higher parallelism degree implies exploring higher processing levels.

Parallelism of trellis compression: A third metric parallelism is associated to the parallel execution of BCJR computations related to more than one symbol (or bit). This is achieved by applying one-level look-ahead trellis transformation technique, called commonly as trellis compression technique. Figure 1.14 illustrates this approach when applied to an 8 state single-binary Turbo code. Two single-binary trellis are compressed and transformed in one double-binary trellis enabling to decode each two consecutive bits in parallel as one double-binary

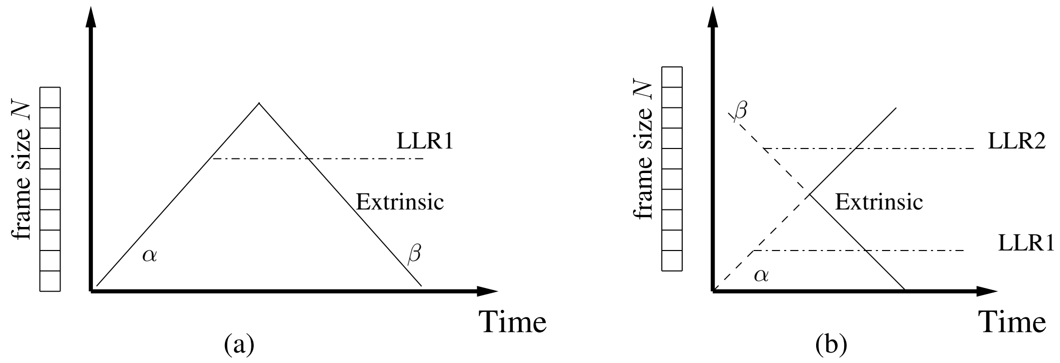

Figure 1.13: (a) Forward backward scheme (b) Butterfly scheme

symbol. This scheme is also denoted as Radix-4 technique. The modified α and β state metrics for this *Radix-4* optimization are given by (1.37) and (1.39). Higher degrees of parallelism can be achieved at this level (Radix-8, Radix-16, etc.), however the complexity increases significantly.

$$\begin{aligned} \alpha_k(s) &= max_{s',s}\{max_{s'',s'}\{\alpha_{k-2}(s^{''}) + \gamma_{k-1}(s^{''}, s')\} + \gamma_k(s^{'}, s)\} \\ &= max_{s'',s}\{\alpha_{k-2}(s^{''}) + \gamma_k(s^{''}, s)\} \end{aligned} \quad (1.37)$$

where $\gamma_k(s^{''}, s)$ is the new branch metric for the combined two-bit symbol (u_{k-1}, u_k) connecting state $s^{''}$ and s:

$$\gamma_k(s^{''}, s) = \gamma_{k-1}(s^{''}, s^{'}) + \gamma_k(s^{'}, s) \quad (1.38)$$

Similarly, the *Radix-4* transform can be applied to the β recursion:

$$\beta_k(s) = max_{s'',s}\{\beta_{k+2}(s^{''}) + \gamma_k(s^{''}, s)\} \quad (1.39)$$

The extrinsic information for u_{k-1} and u_k are computed as:

$$Z_{k-1}^{n.ext} = \Gamma \times (max(Z_{10}^{ext}, Z_{11}^{ext}) - max(Z_{00}^{ext}, Z_{01}^{ext})) \quad (1.40)$$

$$Z_{k}^{n.ext} = \Gamma \times (max(Z_{01}^{ext}, Z_{11}^{ext}) - max(Z_{00}^{ext}, Z_{10}^{ext})) \quad (1.41)$$

Figure 1.14 depicts a single binary 3GPP-LTE trellis, the length of this trellis is reduced and converted to double binary after applying Radix-4. For Turbo decoders that support both single binary (SBTC) and double binary codes (DBTC), applying Radix-4 achieves optimal area efficiency and reuse of resources as it does not affect the memory size nor the computational units.

1.4.2.2 SISO decoder level parallelism

The second level of parallelism concerns the SISO decoder level. It consists of the use of multiple SISO decoders, each executing the BCJR algorithm and processing a sub-block of the same frame in one of the two interleaving orders. At this level, parallelism can be applied either on sub-blocks and/or on component decoders.

Frame sub-blocking: In sub-block parallelism, each frame is divided into M sub-blocks and then each sub-block is processed on a BCJR-SISO decoder using adequate initializations as shown in Figure 1.15 Besides duplication of BCJR-SISO decoders, this parallelism imposes

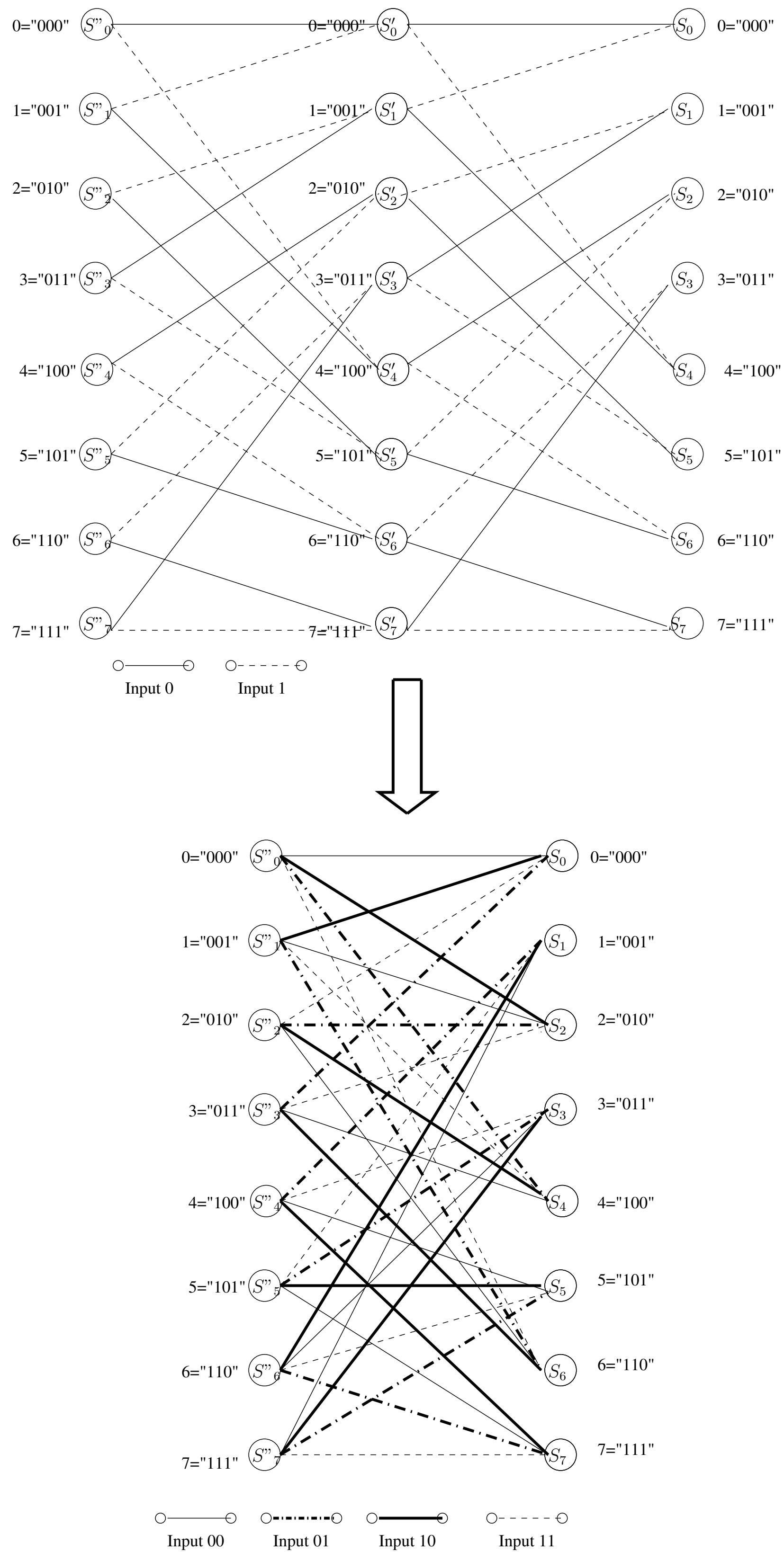

Figure 1.14: One-level Look-ahead Transform for 3GPP-LTE Trellis (Radix-4)

two other constraints. On the one hand, interleaving has to be parallelized in order to scale proportionally the communication bandwidth. Due to the scramble property of interleaving, this parallelism can induce communication conflicts except for interleavers of emerging standards that are conflict-free for certain parallelism degrees. These conflicts force the communication structure to implement conflict management mechanisms and imply a long and variable communication time. This issue is generally addressed by minimizing interleaving delay with specific communication networks [22]. On the other hand, BCJR-SISO decoders have to be initialized adequately either by acquisition or by message passing.
Acquisition method has two implications on implementation. First of all extra memory is required to store the overlapping windows when frame sub-blocking is used and secondly extra time will be required for performing acquisition. Other method, the message passing, which initializes a sub-block with recursion metrics computed during the previous iteration in the neighboring sub-blocks, needs not to store the recursion metric and time overhead is negligible. In [23] a detailed analysis of the parallelism efficiency of these two methods is presented which gives favor to the use of message passing technique. Figure 1.15 shows sub-blocking that apply message passing for initialization, where Figure 1.15(b) is an example of circular Turbo codes such as WiMAX, so in this case, first and last symbols in the frame exchange their metric values (α, β). On the other hand Figure 1.15 (a) presents an example of non circular Turbo codes such as 3GPP, so in this case the encoder generates additional tail bits besides the original frame. These tail bits are used to initialize the last symbol, while the first symbol is always initialized to zero.

Shuffled Turbo decoding: The basic idea of shuffled decoding technique [24] is to execute all component decoders in parallel and to exchange extrinsic information as soon as it is created, so that component decoders use more reliable *a priori* information. Thus the shuffled decoding technique performs decoding (computation time) and interleaving (communication time) fully concurrently while serial decoding implies waiting for the update of all extrinsic information before starting the next half iteration (see Figure 1.16). Thus, by doubling the number of BCJR SISO decoders, component-decoder parallelism halves the iteration period in comparison with originally proposed serial Turbo decoding.

Nevertheless, to preserve error-rate performance with shuffled Turbo decoding, an overhead of iteration between 5 and 50 percent is required depending on the BCJR computation scheme, on the degree of sub-block parallelism, on propagation time, and on interleaving rules [23].

1.4.2.3 Parallelism of Turbo decoder

The highest level of parallelism simply duplicates whole Turbo decoders to process iterations and/or frames in parallel. Iteration parallelism occurs in a pipelined fashion with a maximum pipeline depth equal to the iteration number, whereas frame parallelism presents no limitation in parallelism degree. Nevertheless, Turbo-decoder level parallelism is too area-expensive (all memories and computation resources are duplicated) and presents no gain in frame decoding latency.

1.4.2.4 Parallelism levels comparison

Table 1.2 summarizes the above presented parallelism techniques and presents a comparison in terms of area efficiency and possible parallelism degrees. It is clear that the first and second levels (BCJR metric and SISO decoder) are area efficient since they do not require complete memory duplication which occupies most of the area in a Turbo decoder circuit. However,

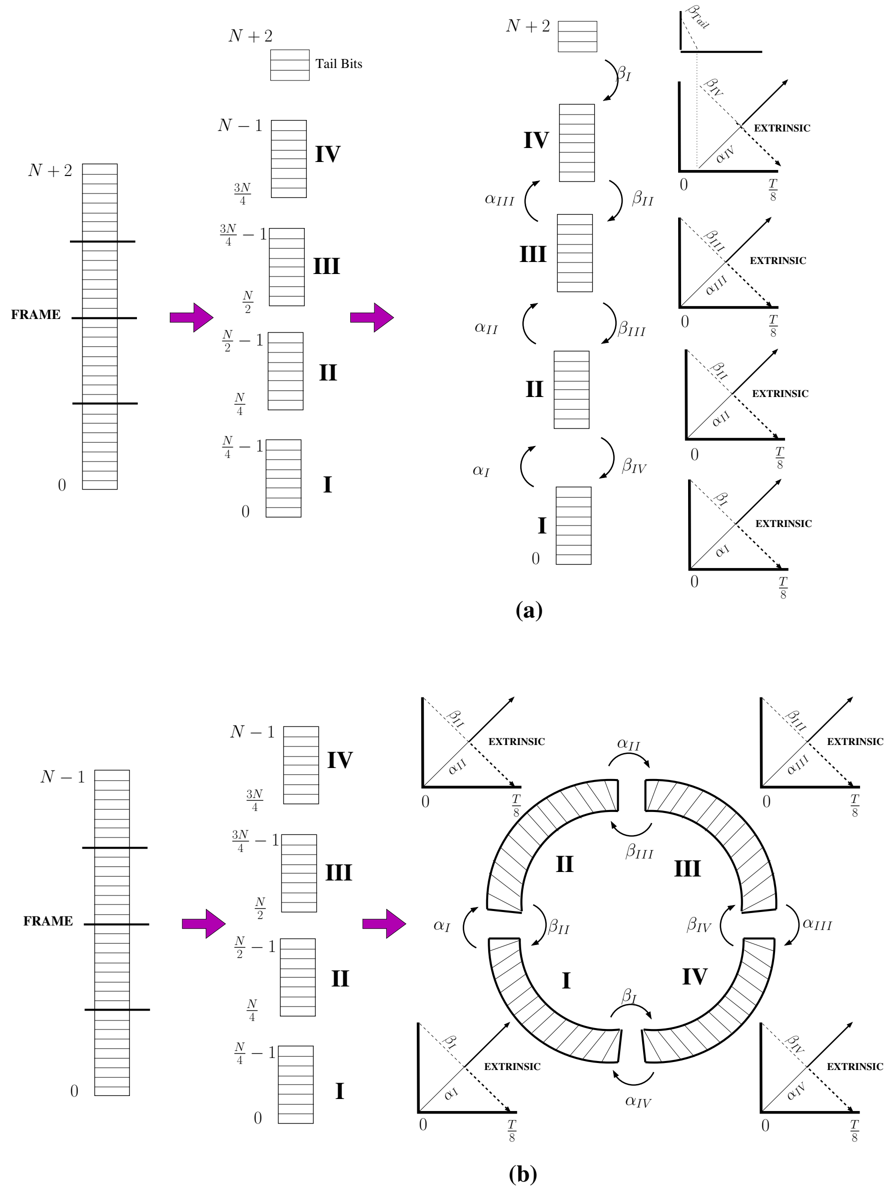

Figure 1.15: Sub-block parallelism with message passing for metric initialization: (a) Non circular code, (b) Circular Code

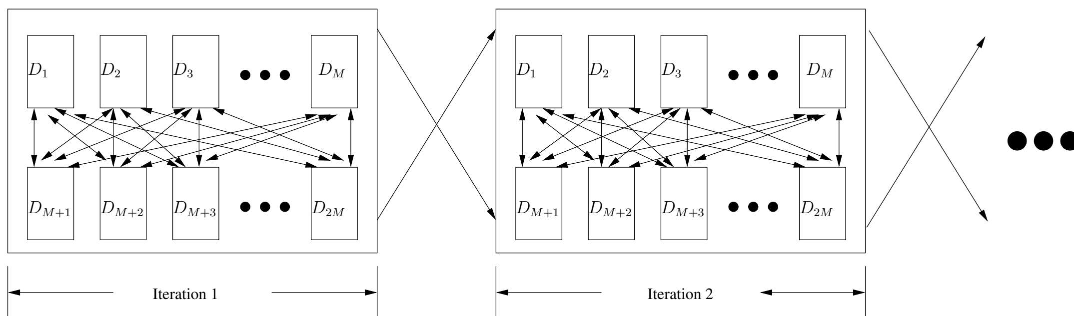

Figure 1.16: Shuffled Turbo decoding

achievable parallelism degrees for both levels are constrained either by the decoding algorithm and component code structure for the first level, or by the interleaving rules and the frame size

for the second level. The third level is the most inefficient in terms of area overhead and should be avoided if possible.

Parallelism Level	Parallelism method	Logic Overhead	Memory Overhead	Area efficiency	Parallelism degree
BCJR metric	Trellis transitions	ACS units are duplicated	No overhead	The most efficient	Limited
	BCJR computations				
	Trellis compression				
SISO decoder	Frame Sub-Blocking	Complete SISO decoders are duplicated	Only internal memories are duplicated (state metrics memories)	Intermediate	High
	Shuffled Turbo decoding				
Turbo decoder	Iterations	Complete Turbo decoders are duplicated	All memories are dublicated (including channel input and extrinsic information memories)	The most inefficient	Unlimited
	Frames				

Table 1.2: Comparison of parallelism levels available in Turbo decoding

1.4.3 Decoding Large Frames in Turbo Decoding

When is needed to decode large frames but we are short with memories. Each frame is divided into M windows, something similar to frame sub-blocking in Section 1.4.2.2. However, instead of processing the sub-blocks in parallels, windows will be processed serially on a BCJR-SISO decoder using adequate initializations as shown in Figure 1.17. Similar to sub-blocking, BCJR-SISO decoder have to be initialized adequately either by acquisition or by message passing.

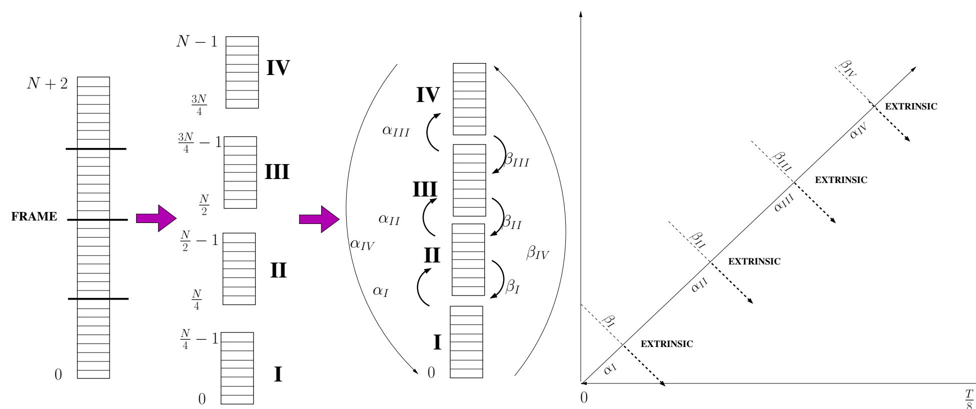

Figure 1.17: Sub-block parallelism with message passing for metric initialization (a) Non circular code (3GPP), (b) Circular Code (WiMAX)

1.4.4 Software Model for Parallel Turbo Decoder

The modeled architecture of the parallel Turbo decoder is shown in Figure 1.18. A software model implementing this parallel Turbo decoder was created in C++ programming language. On the transmitter side, Single/double binary Turbo code of WiMAX/LTE standards with Radix-4

are modeled. The channel is modeled as the channel model is represented by a binary-input additive white Gaussian noise (AWGN). **Scaling Factor Estimation for Radix-4:** As mentioned earlier in Section 1.4.1.2, Max-Log-MAP algorithms provides a loss of 0.1 for double binary code as compared to MAP algorithm when using proper scaling factor. The decoding quality of the Max-Log-MAP decoder is improved by using a scaling factor within the extrinsic calculation. Authors of [25] used a constant scaling factor of $\Gamma = 0.7$. In hardware implementation to reduce complexity, the scaling factor adopted is $\Gamma = 0.75$.

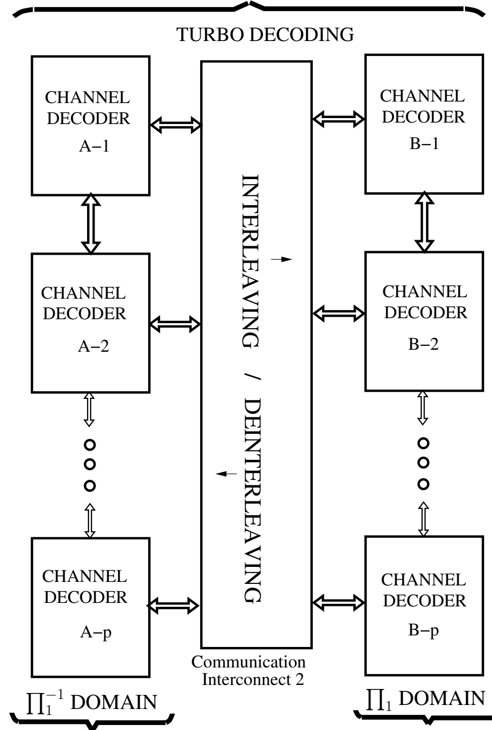

Figure 1.18: Organization of the Turbo decoder's software model

However, extrinsic scaling factor for Max-Log-MAP algorithms when using Radix-4 technique should be re-estimated. Figure 1.19 shows floating point Bits Error Rate (BER) curve for 3GPP-LTE Turbo codes, frames size of 864 bits with code-rate $R = 0.33$, where decoding side uses Radix-4 technique and 8 iteration. The software model is launched to generate BER curves for different scaling factors $\Gamma = 0.3, 0.4, 0.4375, 0.5, 0.7$. The figure shows that for $\Gamma = 0.4$ Turbo decoding has the best BER performance, while for $\Gamma = 0.3$ and $\Gamma = 0.5$ has a degradation of between 0.25db to 0.5db respectively in BER performance compared to $\Gamma = 0.4$ at BER of 10^{-4}. However, for hardware implementation and in order to reduce complexity the scaling factor adopted is $\Gamma = 0.4375$. This value gives very slight degradation comparing with $\Gamma = 0.4$ ($< 0.1db$ at 10^{-4}).

1.5 Summary

In this first chapter, we presented an overview of the fundamental concept of Turbo coding and decoding algorithms. The chapter gave a brief introduction on convolutional Turbo codes along with the different interleaving rules specified in emerging wireless communications standards. Then, the reference MAP algorithm and the hardware-efficient Max-Log-MAP approximation were presented. A classification of available parallelism techniques related to these algorithms

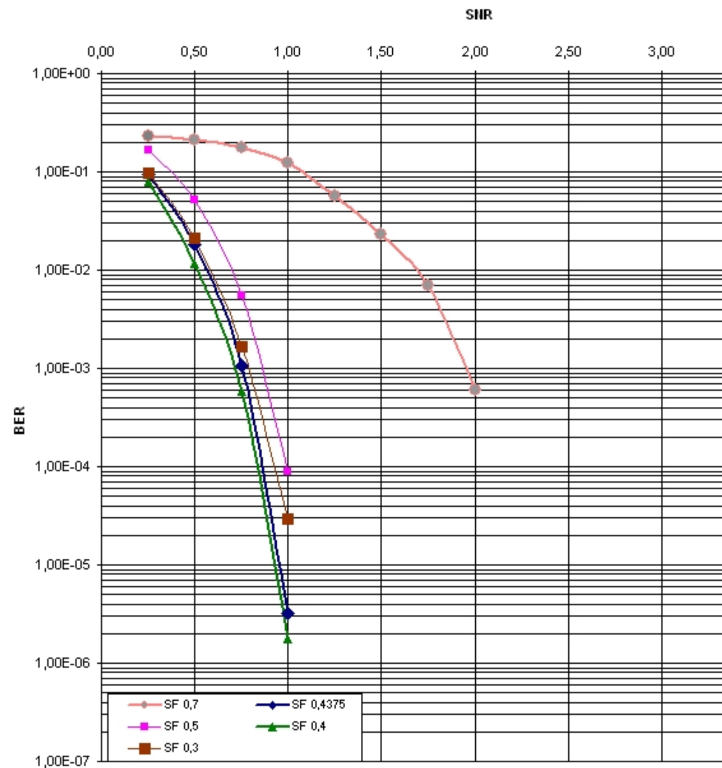

Figure 1.19: BER for 3GPP-LTE frame size 864 bits with different scaling factors

is provided with emphasis on the Radix-4 technique which allows for promising computation commonalities between SBTC and DBTC modes. The windowing technique in Turbo decoding is also highlighted, which is necessary to manage efficiently long frame sizes. Finally, simulation results of error rate performance for SBTC and Radix-4 are presented illustrating the impact of the scaling factor applied on exchanged extrinsic information.

CHAPTER

2 ASIP Design Methodologies and Initial Architecture

APPLICATION Specific Instruction-set Processors (ASIPs) are increasingly used in complex System on Chip (SoC) designs and for many application domains. ASIPs allow the architecture to be tailored to a specific application and requirement in terms of flexibility and performance. This specialization of the core provides a trade-off between the flexibility of a general purpose processor and the performance of a fully dedicated solution.

The first section of this chapter introduces the evolution of embedded processor architectures towards customizable instruction-set ones. This crucial efficiency-driven evolution constitutes our main motivation behind the selection of the ASIP design approach. The second section gives an overview on existing ASIP design flows and presents the considered Synopsys (ex. CoWare) design tool: Processor Designer. Based on this design approach, a first effort has been carried out in the context of a previous thesis study at the Electronic department of Telecom Bretagne to design a high throughput flexible turbo decoder. Thus, in the third and last section of the chapter we will present the initial ASIP architecture which constitutes the starting point of this thesis work.

2.1 Customizable Embedded Processors

The complexity of a large share of the integrated circuits manufactured today is impressive [26][27]: devices with hundreds of millions of transistors are not uncommon. Unsurprisingly, the non-recurrent engineering costs of such high-end application-specific integrated circuits is approaching a hundred million U.S. dollars-a cost hardly bearable by many products individually. It is mainly the need to increase the flexibility and the opportunities for modular reuse that is pushing industry to use more and more software-programmable solutions for practically every class of devices and applications.

On the other hand, processor architecture has evolved dramatically in the last couple of decades [27]: from microprogrammed finite state machines, processors have transformed into single rigid pipelines; then, they became parallel pipelines so that various instructions could be issued at once; next, to exploit the ever-increasing pipelines, instructions started to get reordered dynamically; and, more recently, instructions from multiple threads of executions have been mixed into the pipelines of a single processor, executed at once. However, now something completely different is changing in the lives of these devices: on the whole, the great majority of the high-performance processors produced today address relatively narrow classes of applications. This is related to one of the most fundamental trends that slowly emerged in the last decade: to design tailor-fit processors to the very needs of the application rather than to treat them as rigid fixed entities, which designers include as they are in their products. The emergence of this trend has been made successful thanks to the development of new adequate design methodologies and tools. Such tools enable designers to specify a customizable processor, and in some cases completely design one, in weeks rather than months. Leading companies in providing such methodologies and tools include CoWare (acquired recently by Synopsys),Tensilica, ARC Cores, Hewlett-Packard, and STMicroelectronics. The shape and boundaries of the architectural space covered by the tool chain differentiate the several approaches attempted. Roughly, these approaches can be classified in three categories [27]:

Parameterizable processors are families of processors belonging to a single family and sharing a single architectural skeleton, but in which some of the characteristics can be turned on or off (presence of multipliers, of floating point units, of memory units, and so forth) and others can be scaled (main datapath width, number and type of execution pipelines, number of registers, and so forth).

Extensible processors are processors with some support for application-specific extensions. The support comes both in terms of hardware interfaces and conventions and in terms of adaptability of the tool chain. The extensions possible are often in the form of additional instructions and corresponding functional pipelines but can also include application-specific register files or memory interfaces.

Custom processor development tools are frameworks to support architects in the effort to design from scratch (or, more likely, from simple and/or classic templates) a completely custom processor with its complete tool chain (compiler, simulator, and so forth). Ideally, from a single description in a rich architectural description language (ADL), all tools and the synthesizable description of the desired core can be generated.

In addition to the above mentioned categories which provide hardware flexibility only at design time (and software programmability at run time), it is worth to cite the family of partially reconfigurable ASIPs (rASIP) which targets to add this hardware flexibility at run time. The idea is to combine the programmability of ASIPs with the postfabrication hardware flexibility of reconfigurable structures like FPGAs and CGRAs (Coarse Grained Reconfigurable Architec-

tures). Although several specific rASIP architectures have been proposed in the literature (e.g. [28][29]) and several design methodologies are emerging recently (e.g. [30][31]), there is a lack of commercially available well established tools. Exploring the opportunities offered by this approach in the considered application domain constitutes, however, one of the future research perspectives.

2.2 ASIP Design Methodologies and Tools

Several alternative solutions to ASIPs are available in order to implement the desired task, depending on the requirements [32]: for functions which need lower processing power but should be kept flexible a software implementation running on a General Purpose Processor (GPP) or a micro-controller may be the best solution, if some additional processing power is needed, moving to a domain specific processor, like a Digital Signal Processor (DSP) optimized for signal processing and offering some additional specialized instructions (e.g. Multiply-Accumulate, MAC), may be beneficial. On the contrary, system modules with very high processing requirements are usually implemented as hardware blocks (Application Specific Integrated Circuits, ASICs, or even physically optimized ICs), with no flexibility at all. If some flexibility is required, field programmable devices like Field Programmable Gate Arrays (FPGAs), which allow for reconfiguration after fabrication, may be the right choice if some price in terms of performance, area and power can be paid. ASIPs represent an intermediate solution between DSPs and FPGAs in terms of performance and flexibility, thus becoming in many cases the best choice to play this trade-off, as shown in Figure (2.1).
Several options are available to the ASIP designer for developing his own processor with different degrees of freedom, ranging from the complete specification through an Architecture Description Language (ADL) to a limited post-fabrication reconfigurability achieved via software. ASIPs are often employed as basic components of heterogeneous Multi Processor System on Chips (MPSoCs), due to the ever increasing demand for flexibility, performance and energy efficiency, which forces designers to exploit heterogeneous computational fabrics in order to meet these conflicting requirements. Using ASIPs as Processing Elements (PEs) in complex systems is a viable solution in order to play this trade off.

2.2.1 ASIP Design Approaches

Typically, the development flow of ASIPs starts with the analysis of the application in order to identify its ”hot spots” [32]. Then, an initial architecture is defined, in particular with special custom instructions to improve the efficiency for handling those hot spots. Afterward the applications are run on the processor in order to verify if the target specifications have been met. If that is not the case, the whole flow is iterated to meet the design requirements for given applications.

From this quick overview of the design flow it is clear that some tools are required for implementing it: First, an assembler and a linker are needed in order to run the application code on the processor, together with a compiler if a high-level programming language is used; these tools are required both for design space exploration, when the target application has to be tested in order to improve the architecture, and for software development after the final architecture has been defined. Moreover, an Instruction Set Simulator (ISS) has to be provided so that the application can be run both for profiling and for verification purposes. All these tools directly depend on the instruction set of the processor and hence they have to be re-targeted

each time that the instruction set is modified. Almost all the available ASIP design suites provide these tools and the capability to re-target them when needed, while some of them also include the further ability to automate the process of profiling the application and identifying the instructions which are most suitable for instruction-set extension.
By looking at available commercial solutions for ASIP design, it is possible to identify three main classes based on the degree of freedom which is left to the designer [32]:

- Architecture Description Language (ADL) based solutions (e.g. Synopsys Processor Designer [33], Target IP Designer [34]), which can be also defined as ASIP-from-scratch since every detail of the architecture, including for instance pipeline and memory structures, can be accessed and specified by the designer by means of a proper language. This approach results in the highest flexibility and efficiency, but on the other hand it requires a significant design effort.
- Template architecture based solutions (e.g. Tensilica Xtensa [35], ARC ARChitect [36]), which allow the designer to add custom ISE to a pre-defined and pre-verified core, thus restricting the degree of freedom with respect to the previous approach to the instruction set definition only.
- Software configurable processors and reconfigurable processors (e.g. Stretch [37]), with a fixed hardware including a specific reconfigurable ISE fabric which allows the designer to build custom instructions after the fabrication.

2.2.2 Synopsys's ADL-based Design Tool: Processor Designer

Synopsys Processor Designer is an ASIP design environment entirely based on LISA [38]. LISA was developed at Aachen University of Technology, Germany, with a simulator-centric view [27]. The LISA language is aiming at the formalized description of programmable architecture,

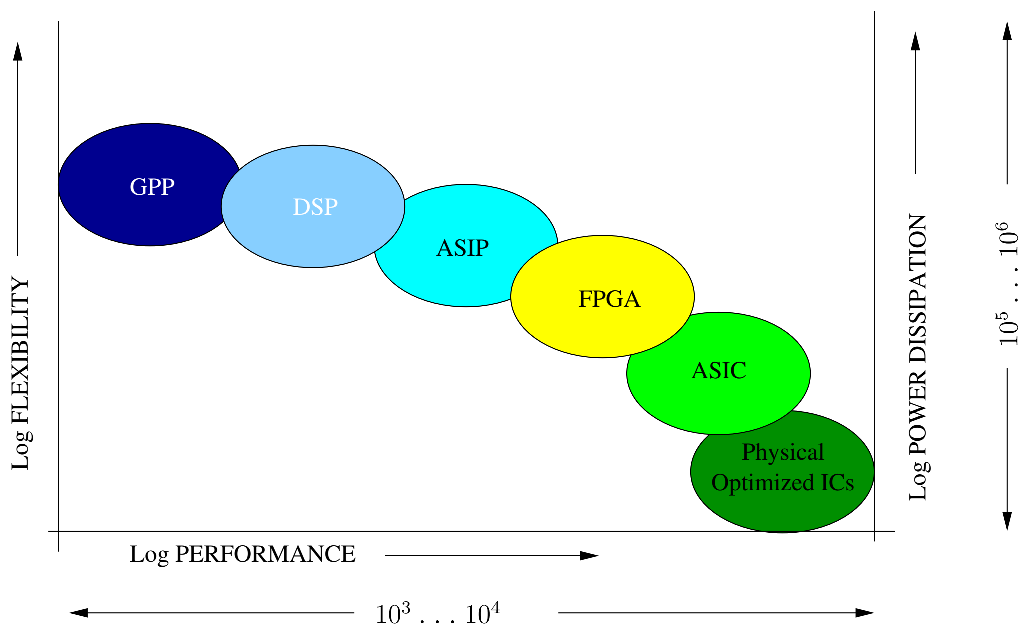

Figure 2.1: Performance-Flexibility Trade-off of Different Architecture Models

their peripherals and interfaces. it was developed to close the gap between purely structural oriented languages (VHDL, Verilog) and instruction set languages for architecture exploration and implementation purposes of a wide range of modern programmable architectures. The language syntax provides a high flexibility to describe the instruction set of various processors such as Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD) and Very Long Instruction Word (VLIW) type architectures. Moreover, processors with complex pipelines can be easily modeled. The language has been used to produce production quality simulators. An important aspect of LISA is its ability to capture control path explicitly. Explicit modeling of both data-path and control is necessary for cycle-accurate simulation.
Processor Designer's high degree of automation greatly reduces the time for developing the software tool suite and hardware implementation of the processor, which enables designers to focus on architecture exploration and development. The usage of a centralized description of the processor architecture ensures the consistency of the Instruction-Set Simulator (ISS), software development tools (compiler, assembler, and linker etc.) and RTL (Register Transfer Level) implementation, minimizing the verification and debug effort.

The LISA machine description provides information consisting of the following model components [32]:

- The memory model lists the registers and memories of the system with their respective bit widths ranges and aliasing.
- The resource model describes the available hardware resources, like registers, and the resource requirements of operations. Resources reproduce properties of hardware structures which can be accessed exclusively by a given number of operations at a time.
- The instruction set model identifies valid combinations of hardware operations and admissible operands. It is expressed by the assembly syntax, instruction word coding, and the specification of legal operands and addressing modes for each instruction.
- The behavioral model abstracts the activities of hardware structures to operations changing the state of the processor for simulation purposes. The abstraction level can range widely between the hardware implementation level and the level of high-level language (HLL) statements.
- The timing model specifies the activation sequence of hardware operations and units.
- The micro-architecture model allows grouping of hardware operations to functional units and contains the exact micro-architecture implementation of structural components such as adders, multipliers, etc.

By using these various model components to describe the architecture, it is then possible to generate a synthesizable HDL representation and the complete software tool suite automatically.

The generation of the software development environment by Processor designer enables to start application software development prior to silicon availability, thus eliminating a common bottleneck in embedded system development. As it is shown in Figure (2.2), the design flow of Processor Designer is a closed-loop of architecture exploration for the input applications. It starts from a LISA 2.0 description, which incorporates all necessary processor-specific components such as register files, pipelines, pins, memory and caches, and instructions, so that the designer can fully specify the processor architecture. Through Processor Designer, the ISS and the complete tool suite (C-compiler, assembler, linker) are automatically generated. Simulation is then run on the architecture simulator and the performance can be analyzed to check whether

the design metrics are fulfilled. If not, architecture specifications are modified in LISA description until design goals are met. At the end, the final version of RTL implementation (Verilog, VHDL and SystemC) together with software tools is automatically generated.
As previously mentioned, ASIPs are often employed as basic components of more complex systems, e.g. MPSoCs. Therefore, it is very important that their design can be embedded into the overall system design. Processor Designer provides possibilities to generate a SystemC model for the processor, so that it can be integrated into a virtual platform. In this way, the interaction of the processor with the other components in the system can be tested. Furthermore, the exploration as well as the software development of the platform at early design stage becomes possible.

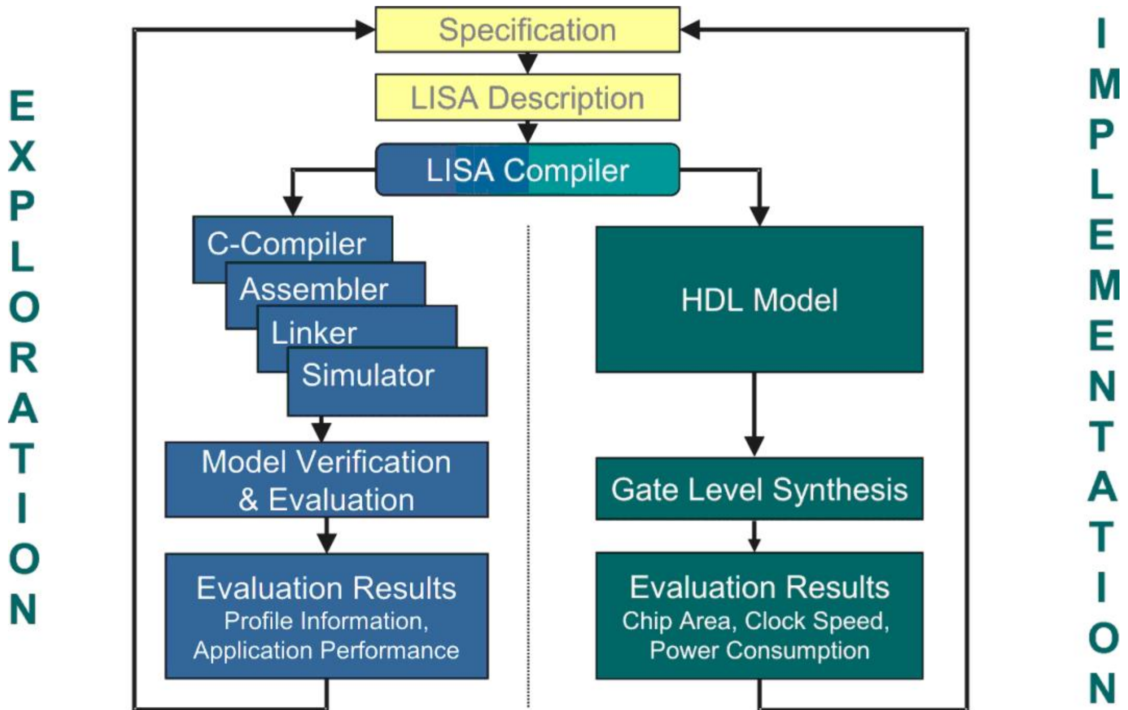

Figure 2.2: LISA architecture exploration flow

2.3 Initial ASIP Architecture for Turbo Decoding

A first effort has been carried out in the context of a previous thesis study at the Electronic department of Télécom Bretagne to design a high throughput flexible turbo decoder [39][40]. In this section we will present the initial ASIP architecture which constitutes the starting point of this new thesis contribution

2.3.1 Overview of the Design Approach and the Architectural Choices

To address high throughput requirement, first of all, a comprehensive study was made in exploring the efficient parallelism at different levels within a turbo decoder. This parallelism study is detailed in [40] and is summarized in Section 1.4.2 of Chapter II. As the first parallelism level (BCJR metric level) occurring inside a BCJR SISO decoder is the most area efficient, a hardware implementation achieving high throughput should first exploit this parallelism. A dedicated

processing architecture with multiple functional units and adequate memory interfaces can be suitable to efficiently perform all computations of a BCJR-SISO decoder. However, as the parallelism degree of this level is limited, further increase of throughput should exploit the second parallelism level (SISO decoder level). This can be done efficiently by instantiating multiple BCJR-SISO processing units with dedicated on-chip communication interconnect.

As the flexibility requirement is also considered, besides high throughput, the processing units and the on-chip communication interconnect should be flexible. A trade-off between performance and flexibility is thus imposed for the processing unit architecture and the ASIP design approach is thus adopted. Regarding the on-chip communication, appropriate NoC architectures are explored and designed.

Extracting the flexibility requirements of target standards as summarized in Table 1.1 (Page 17) was the initial step toward the ASIP design for turbo decoding. The complexity of convolutional turbo codes proposed in most existing and emerging wireless communication standards is limited to eight-state double binary or binary turbo codes. Hence, to fully exploit trellis transition parallelism for all standards, a parallelism degree of 32 is required for worst case eight states double binary turbo codes (8 states $\times$ 4 transitions per state). Regarding BCJR computation parallelism, a parallelism degree of two has been adopted, i.e. two recursion units to implement the Butterfly metric computation scheme presented in Section 1.4.2.1.

In order to present the initial ASIP architecture [26], a bottom-up approach is adopted where the basic building blocks are explained first. Based on these building blocks the architecture of recursion units are then presented and finally the full ASIP architecture is illustrated.

2.3.2 Building Blocks of the Initial ASIP

The flexibility parameters of this ASIP are fundamentally based on supporting single and double binary turbo codes implementing the expressions 1.30, 1.31, 1.32 and 1.33 of max-log-MAP algorithm. Details of the building blocks composing the initial ASIP architecture is briefly discussed below [26]:

Gamma (γ) Metric Computation: As stated in Section 1.4.2.1 (Page 23) on parallelsim of trellis transitions, all possible values of γ related to all transitions of the trellis can be computed in parallel. In hardware this is achieved by the use of 24 simple adders and subtracters per recursion unit, which process input channel LLRs and input extrinsic LLRs to generate γ metrics.

Alpha (α), Beta (β) and Extrinsic Information (Z) Computation: The computations related to state metrics α and β consist of: (1) the recursive addition of γ with α and β to compute them over all sections of the trellis and (2) the selection of maximum α and β related to the transitions associated with each state. Same case with extrinsic information where all three metrics γ, α and β are added on each section of trellis. But for extrinsic information generation, the maximum operation is performed on values related to those transitions which are occurred due to the input for which the extrinsic information is being computed. In literature this operation is often referred as Add Compare Select (ACS) operation.

The basic computational units of the initial ASIP architecture providing ACS are shown in Figure 2.3 An Adder Node (see Figure (2.3 a)) can be used for addition operation required both in state metric and extrinsic information computation. While computing state metrics, the adder node can be configured for one of the input state metric (α of previous symbol or β of next

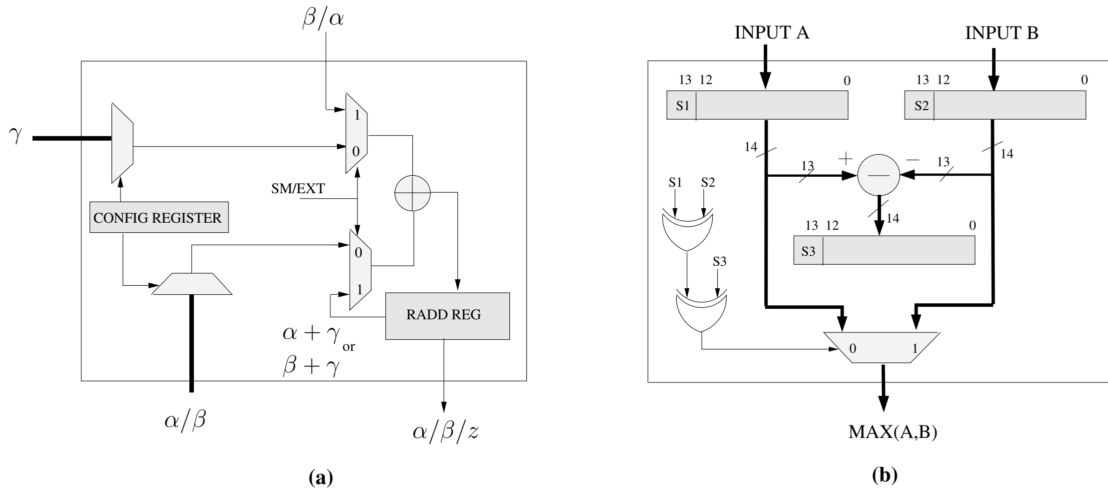

Figure 2.3: Basic computational units of the initial ASIP (a) Adder node (b) Modulo compare unit

symbol) and associated γ using the configuration register, depending upon the trellis selected (WiMAX, 3GPP ...etc.).

While performing the addition operation involved in extrinsic information computation, the already stored sum (in **RADD REG**) of state metric and branch metric (α+γ or β+γ) is added with the other corresponding state metric (β or α respectively). The quantization for the state metrics and extrinsic information is based on modulo algorithm [41] and the signed values are allowed to overflow. When overflow occurs, causing values to move from positive region to negative region, the largest value becomes the smallest. In this situation when maximum finding operation is performed, a simple maximum operation will lead for wrong result. To address this issue, the architecture of specialized *max* operator is proposed in Figure 2.3 b, which detects the largest value even in case of overflow situation and conforms to modulo algorithm. In fact, the first issue is to detect this particular situation which can be done by analyzing the 2 MSBs of the inputs. As shown in Figure 2.4, in which n bits represent the quantization of state metrics and extrinsic information, if a value lies in the region Q-2 its two MSB's will be "01" whereas in Q-3 they will be "10". Hence, if some of the extrinsic informations related to different combinations of a symbol lay in Q-2 and the others in Q-3 this will identify the problematic situation. In this case the second step is to correct the extrinsic information in a way that the largest extrinsic information remains largest. This can be done simply by an unsigned right shifting of all the extrinsic informations.

Forward Backward Recursion Units: As stated above, butterfly scheme of state metric computation and extrinsic information generation is used. Hence, two hardware recursion units, one working in forward and the other in backward direction are used. Since there are 32 transitions in the worst case of 8 state double binary code, each recursion unit is made up of 32 adder nodes. The arrangement of adder nodes in Forward recursion unit is shown in Figure 2.5. To each state of the trellis (row-wise) corresponds to four adder nodes (column-wise) which receive proper γ due to the four possible inputs in double binary code. Same architecture is used for backward recursion unit. As stated above using configuration register each adder node receives its inputs according to the trellis structure.

To perform the *max* operation, 24 2-input compare units are used in each recursion unit. These 24 2-input compare units can be configured to work as 8 4-input compare units. When computation of *max* operation in state metrics generation is required, 8 4-input compare units

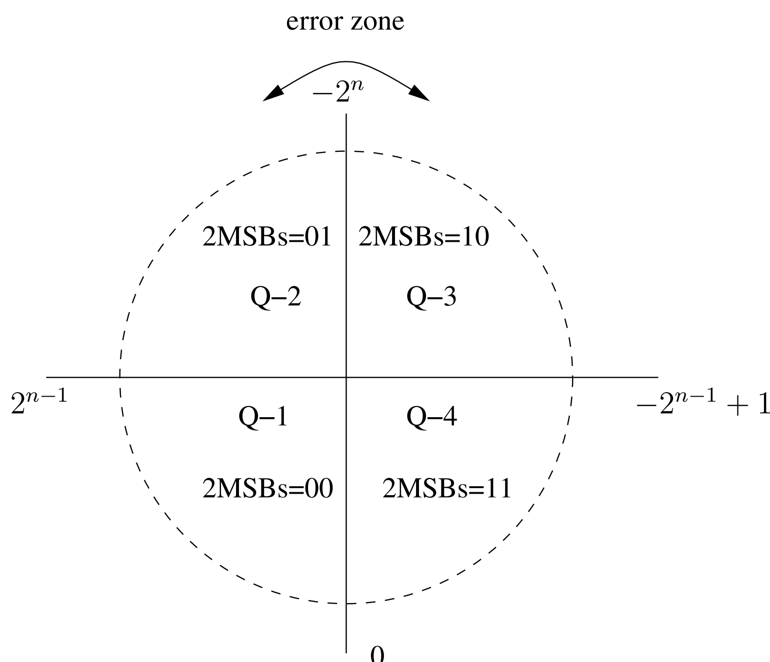

Figure 2.4: Modulo algorithm extrinsic information processing

are connected to the four outputs (output of 4 adder nodes in a column) of each column of the recursion unit as shown in Figure 2.5(a). Hence, at the output of the 8 4-input *max* operators, 8 state metrics of double binary code are obtained. In case of extrinsic information computation, as shown in Figure 2.5(b), the 8 4-input compare units are connected to adder nodes in rows in such a way that 8 elements of a row are divided into 2 sets of 4 adder nodes (first set made up of 4 adder nodes on the left of the row and the second set made up of 4 adder nodes on the right). These 8 sets from 4 rows are connected 8 4-input compare units. The two maximum values obtained in each row are saved back in the **RADD** registers of first two adder nodes of that row. Reusing 4 2-input compare units one can find the maximum per row (between the two candidates) which is the extrinsic information for the corresponding combination of input bits.

2.3.3 Overall Architecture of the Initial ASIP

The overall architecture of the initial ASIP [26] is composed of memory interface, internal registers, basic building blocks and a control unit as shown in Figure 2.6. Regarding memory interface, the application program is saved in the program memory. Config Memory is used to store the configuration of the trellis which can be loaded in the ASIP to switch between different trellis structures. The input data to the ASIP for data decoding is provided from Input and Extrinsic Data memories. To achieve butterfly scheme these two memories are further divided into top and bottom banks to enable two simultaneous memory access. Cross metric memory is used to store the state metrics while left side of butterfly scheme is in progress. These stored metrics are used to compute the extrinsic information during right side of the butterfly scheme. The interleaving/deinterleaving tables are stored in the interleaving memories. Once the ASIP computes the extrinsic information, the interleaving address from these memories is read. This address is placed as header whereas the extrinsic information is placed as payload in the packet which is sent on the network. Finally Read/Write α, β Memories are used to store the last values of state metrics. In the context of sub-blocking parallelism these saved state metrics are used to

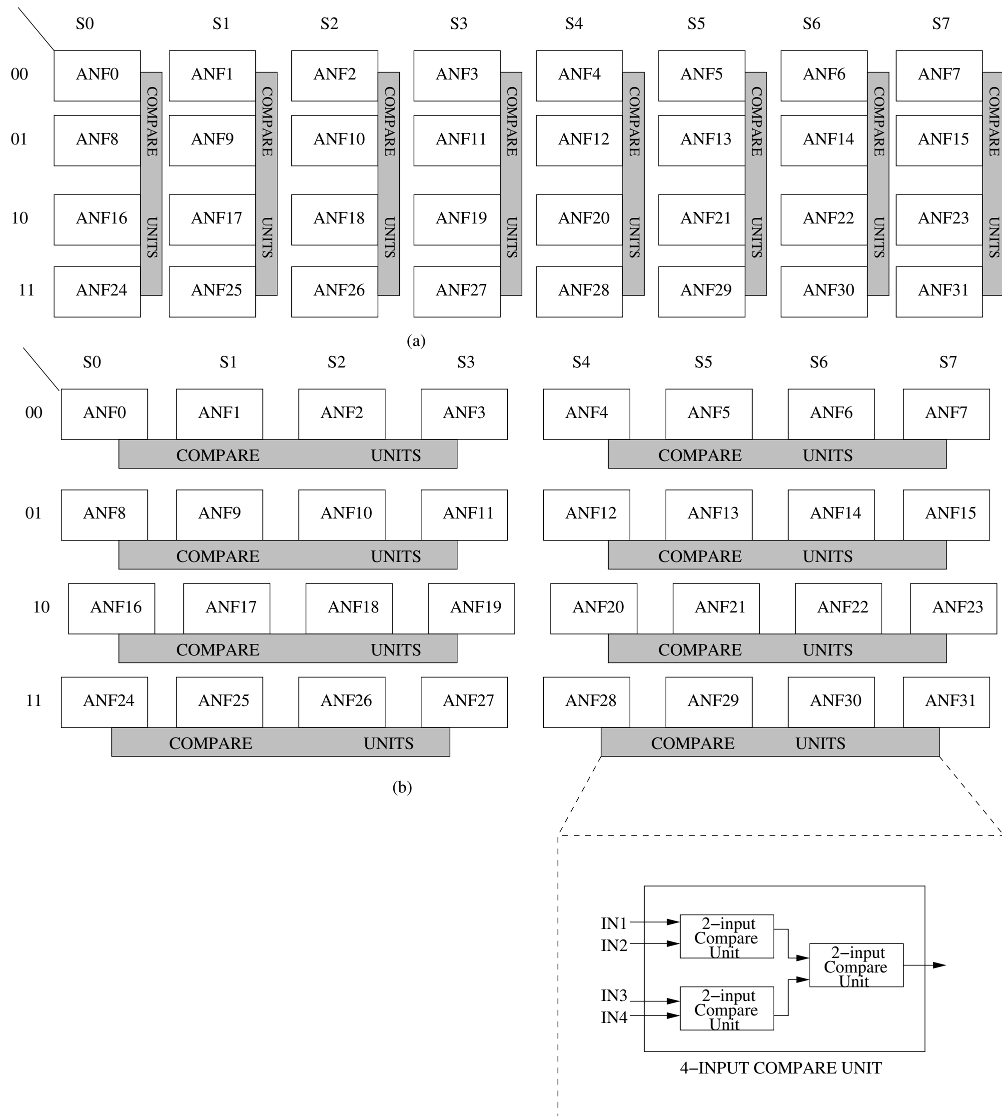

Figure 2.5: Forward Recursion Unit composed of 32 ANF (Adder Node Forward) and 8 4-input Compare Unit (a) Compare Units used for state metric computation (b) Compare Units used for extrinsic information computation

implement message passing method of metrics initialization.

Certain register banks are used for ASIP configuration and storage of different parameters associated to the max-log-MAP algorithm. Two configuration registers are dedicated for the configuration of the two recursion units of the ASIP. The configuration of the ASIP is downloaded from Config Memory into these registers. RMC registers store the state metric after max operations whereas RC registers are used to read state metrics during the computation of the right side of the butterfly scheme. Branch metrics γ are stored in RG registers. RADD registers are part of the ACS units as shown in Figure 2.3.

The role of the control unit is to manage all the resources spread over seven pipelines stages. After instruction fetch and decode pipeline stages, the third pipeline stage is dedicated to fetch the operands from input memories. These two next stages, BM1 and BM2, are for γ computation. In execute (EXE) pipeline stage, resources of Add Compare Select (ACS) operations are placed. The last pipeline stage is to complete the computation and the storage of the extrinsic

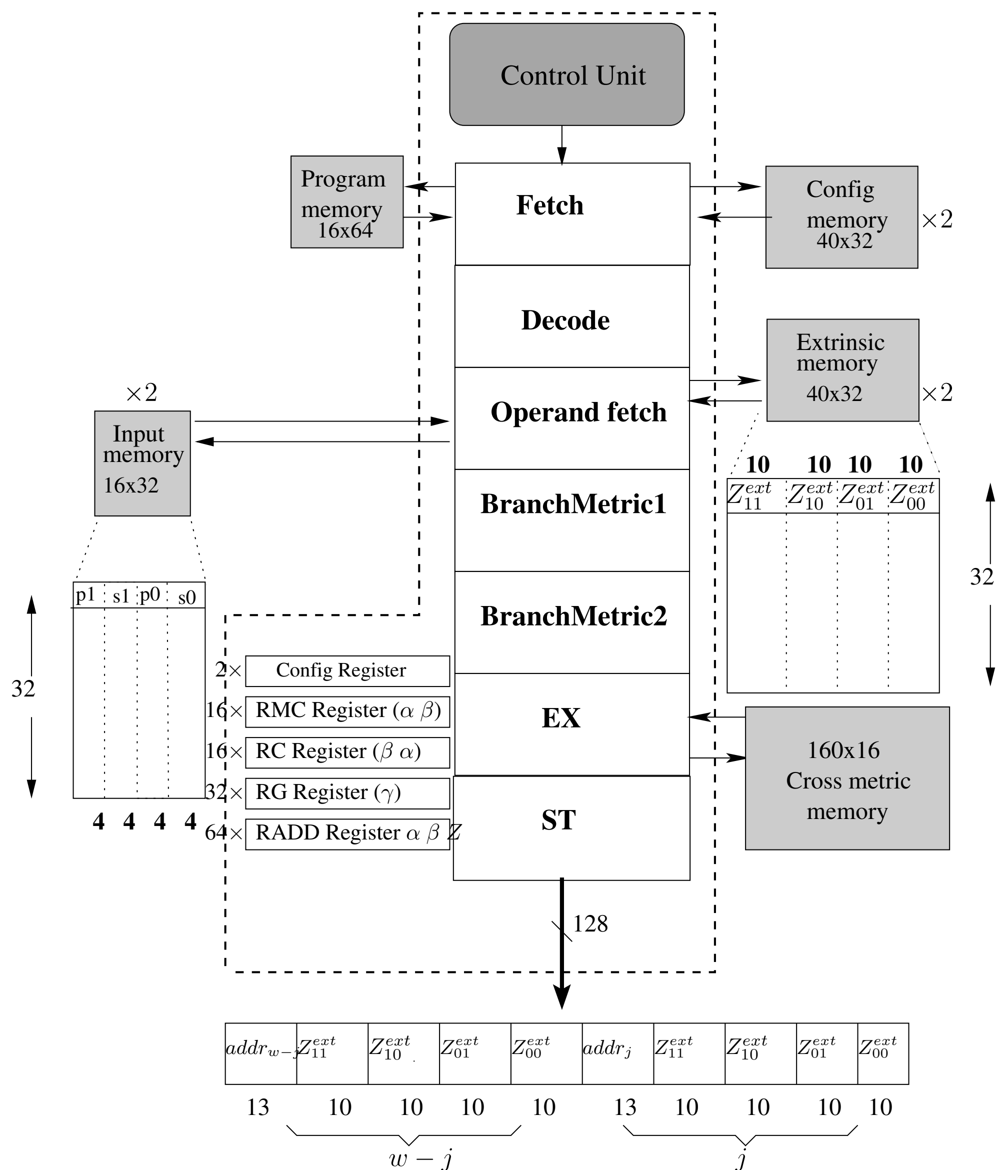

Figure 2.6: Overall architecture of the initial ASIP

information and hard decisions.

2.3.4 Sample Program of the Initial ASIP

To give an explanation on how the initial ASIP can be used for a decoding application, Listing 2.1 presents the piece of code written for dual binary code of WiMAX standard for the first iteration. The text after (;) sign shows the comments.

Listing 2.1: Initial ASIP: assembly code for 8-state double binary turbo code for first iteration

```
1  ; loading configuration in config. registers
2          LD_CONFIG 0
3          LD_CONFIG 1
4          LD_CONFIG 2
```

```
5          LD_CONFIG 3
6  ; setting block length
7          SET_SIZE 48
8  ; scaling of extrinsic information
9          SET_SF 6
10 ; uniform start values for alpha / beta
11         SET_RMC UNIFORM, UNIFORM
12 ; zero over head loop instruction
13         ZOLB _LW1 , _LW1 , _RW1
14 ; left butter fly alpha / beta + gamma
15         DATA LEFT WITHOUT_EXT ADD M
16 ;max for alpha / beta
17 _LW1:   MAX2 STATE METRIC NOTHING
18 ; right butterfly alpha / beta +gamma
19         DATA RIGHT WITHOUT_EXT ADD M
20 ;max for alpha / beta
21         MAX2 STATE METRIC NOTHING
22 ;left butterfly alpha + beta +gamma
23         EXTCALC WITHOUT_EXT ADD I
24 ; first max for extrinsic computation
25         MAX2 SYSTEMATIC NOTHING
26 ; second max for extrinsic computation
27 _RW1:   MAX1 SYSTEMATIC EXT
```

The program starts with LD_CONFIG instruction which configures the ASIP for the WiMAX trellis. Using SET_SIZE instruction the programmer can set the number of double binary symbols in a block. SET_SF is the command to scale the input extrinsics e.g with target double binary application the value 6 corresponds to multiplying the input extrinsic with 0.75 before computing the branch metrics. SET_RMC UNIFORM instruction sets the RMC for zero which means all starting states are equiprobable. ZOLB is the zero overhead loop instruction. With this single instruction, the next two lines of code executes 24 times (half of block size) which is in fact implementing left side of butterfly decoding scheme for 24 symbols without using the extrinsic information. The reason of not using the extrinsic information is due to the fact that during the first shuffled iteration the extrinsic information memories hold the data of the last iteration of the previous frame. After this, the next five instructions (from ”DATA RIGHT..” instruction to ”MAX1 SYSTEMATIC..”) execute 24 times and implement right side of the butterfly decoding scheme. During the execution of left butterfly, the first instruction ”DATA LEFT..” is used to compute state metrics related to all transitions. The next ”MAX2 STATE METRIC” performs compare operation to compute 8 state metrics both in forward and backward directions. In the right part of the butterfly scheme, the first two instructions compute the state metrics and the next three instructions compute the extrinsic information. The first instruction to compute extrinsic is related to performing the summation of α, β and γ while the next two instructions are used to do compare operation on the rows of recursion units in two steps. Finally, at the end of processing of the block of 48 symbols, the ASIP initializes its state metric registers (RMC) using message passing the for next iteration. Hence, during one iteration, 2 clock cycles per 2 symbols are used in left side of butterfly decoding scheme and 5 clock cycles per 2 symbols are used in right side of the butterfly decoding scheme.

After the first iteration the extrinsic information memories hold the right extrinsic information hence one can which can be thus used in the computations of subsequent iterations. The

code for the next five iterations is shown in Listing 2.2 where instructions for DATA LEFT and DATA RIGHT are used with READ EXT option in place of WITHOUT EXT option of Listing 2.1.

Listing 2.2: Initial ASIP: assembly code for 8-state double binary turbo code for iteration numbers 2-6

```
1           REPEAT UNTIL_loop 5 times
2  ;zero over head loop instruction
3           ZOLB _LW1 , _LW1 , _RW1
4  ; left butter fly alpha / beta +gamma
5           DATA LEFT READ_EXT ADD M
6  ;max for alpha / beta
7  _LW1:    MAX2 STATE METRIC NOTHING
8  ; right butterfly alpha / beta +gamma
9           DATA RIGHT READ_EXT ADD M
10 ;max for alpha / beta
11          MAX2 STATE METRIC NOTHING
12 ;left butterfly alpha + beta +gamma
13          EXTCALC READ_EXT ADD I
14 ; first max for extrinsic computation
15          MAX2 SYSTEMATIC NOTHING
16 ; second max for extrinsic computation
17 _RW1:    MAX1 SYSTEMATIC EXT
18 _loop:NOP
```

To compute the hard decision in last iteration, the assembly code is presented in Listing 2.3.

Listing 2.3: Initial ASIP: assembly code for 8-state double binary turbo code for last iteration

```
1  ; zero over head loop instruction
2           ZOLB _LW1 , _LW1 , _RW1
3  ; left butter fly alpha / beta +gamma
4           DATA LEFT READ_EXT ADD M
5  ;max for alpha / beta
6  _LW1:    MAX2 STATE METRIC NOTHING
7  ; right butterfly alpha / beta +gamma
8           DATA RIGHT READ_EXT ADD M
9  ;max for alpha / beta
10          MAX2 STATE METRIC NOTHING
11 ;left butterfly alpha + beta +gamma
12          EXTCALC READ_EXT ADD I
13 ; first max for extrinsic computation
14          MAX2 SYSTEMATIC NOTHING
15 ; second max for extrinsic computation
16 _RW1:    MAX1 SYSTEMATIC HARD
```

The hard decisions are computed by using HARD option in the last instruction of Listing 2.3 which was EXT in Listing 2.2 & 2.1. As far as the throughput is concerned, an average of 3.5 clock cycles are required to generate the extrinsic information of each double binary symbol per iteration (or hard decision during the last iteration).

2.4 Summary

This second chapter has introduced the concept of ASIP-based design and the associated design methodology and tool which are considered in this thesis work. This methodology and the target Turbo decoding application have been illustrated through the presentation of an initial ASIP architecture which has been developed in a previous thesis study at the Electronic department of Telecom Bretagne. In this initial architecture, the main target was to validate the effectiveness of the newly proposed ASIP-design tools in terms of generated HDL code and flexibility limitations. To that end, the target flexibility was set very high to investigate the support of any convolutional code trellis. This target flexibility has led to a reduced architecture efficiency, which was in all cases not the main target of that initial effort on the topic. Furthermore, this initial ASIP architecture was missing several features which will be highlighted, and adequate solutions will be proposed, in the next chapter.

CHAPTER

3 Optimized ASIP for Multi-standard Turbo Decoding

IN Chapter 1, the fundamental requirements of wireless communication radio platform for Turbo codes is laid down and turbo processing is presented as a possible solution to achieve error rate performance reaching theoretical limits. Chapter 2 has presented an initial ASIP architecture for Turbo decoding, targeting mainly flexibility without real consideration of the architecture efficiency. This third chapter is aimed to illustrate how the application of adequate algorithmic and architecture level optimization techniques on an ASIP for turbo decoding can make it even an attractive and efficient solution in terms of area, throughput, and power consumption. The suggested architecture integrates two ASIP components supporting binary/duo-binary turbo codes and combines several optimization techniques regarding pipeline structure, trellis compression (Radix-4), and memory organization. The logic synthesis results yield an overall area of 0.594mm^2 using 65nm CMOS technology. Payload throughput of up to 164Mbps in both double binary Turbo codes (DBTC) and single binary (SBTC) are achievable at 500MHz.

This chapter is organized in the following order. First of all, a brief state of the art section is provided to summarize the available hardware implementations related to this domain. Then, the proposed optimization techniques regarding pipeline structure, trellis compression (Radix-4), and memory organization are detailed. Furthermore, in order to improve the speed of reconfigurability of the proposed architecture, a new organization of the instruction program is presented. Then, the achieved Turbo decoder system performance is summarized and the architecture efficiency is compared with state of the art related works. Finally, the impact of the proposed pipeline optimization on energy efficiency is discussed. This last contribution concerns a joint effort with another PhD student at the CEA-LETI: Pallavi Reddy. To investigate this impact, the power consumption analysis for ASIC target implementation has been conducted on the design before and after optimization. The result shows an interesting gain in normalized energy efficiency of around $\approx$45.6% in comparison with the initial architecture.

3.1 State of the Art

Application-specific processors are being widely investigated these last years in System-on-Chip design. The main reason behind this emerging trend is the increasing requirements of flexibility and high performance in many application domains. Digital communication domain is very representative of this trend where many flexible designs have been recently proposed for the challenging turbo decoding application. For this application, there is a large variety of coding options specified in existing and future digital communication standards, besides the increasing throughput requirement.

Numerous research groups have come up with different architectures providing specific reconfigurability to support multiple standards on a single device. A majority of these works target channel decoding and particularly turbo decoding. The supported types of channel coding for turbo codes are usually Single Binary and/ Double Binary Turbo Codes (SBTC and DBTC).

The available implemented works can be grouped into three categories, multi-standards customizable embedded processor, multi-standards parameterized dedicated architecture (No instruction set), and one-standard dedicated architectures. In order to facilitate the comparison between these works, we will normalize the occupied areas to $@65nm$ technology depending on the conversion formula shown in (3.1).

$$A_1 = A_2 \times (\frac{f_1}{f_2})^2 \tag{3.1}$$

Here,
f_1 - Feature size for technology normalized ($65nm$),
f_2 - Feature size for technology used,
A_1 - Normalized Area,
A_2 - Occupied Area.

In this context, multi-standards customizable embedded processor (ASIP-based) implementation work found in [42], it is a flexible pipeline architecture that supports wide range of 3gpp standards (UMTS, HSPA, EDGE, etc.) and convolutional code (CC). The presented ASIP occupies a small area of 0.5mm^2 in 65nm technology however it achieves a limited throughput of 21Mbps at 300Mhz.

Another customizable embedded processor (Xtensa based) implementation work [43] proposed from Tensilica. The work supports LTE and HSPA+ with maximum throughput of 150Mbps. The occupied area is 1.34mm^2 in 45nm technology (2.8mm^2 in 65 nm)

Besides ASIP-based solutions, other flexible implementations are proposed using a parametrized dedicated architecture (not based on instruction-set), like the work presented in [44]. The proposed architecture supports WiMAX and 3GPP-LTE standards and achieves a high throughput of 100 Mbps. However, the occupied area is large: 10.7mm^2 in 130nm technology (2.675mm^2 in 65nm). Another example is the work proposed in [45] which supports all WiMAX modes, but only 18 out of 188 modes for 3GPP-LTE. However, the lack of flexibility is due to adapting vectorizable and contention-free parallel interleaver. Their occupied area is 3.38mm^2 in 90nm (1.75mm^2 in 65nm), while the maximum throughput achieved is 186.1Mbps when all the 8 MAPs run in parallel at the maximum frequency of 152Mhz. However, the throughput vary from 24Mbps to 93Mbps for frame-sizes less than 360 bits.

On the other hand, several one-standard dedicated architectures exist. In this category we can cite the dedicated architectures presented in [46], and [47] which support only SBTC mode

(3GPP-LTE). In [46] a maximum throughput of 150Mbps is achieved at the cost of a large area of 2.1mm^2 in 65nm. The proposed work in [47] presented a very high throughput of maximum 1.28Gbps at 400Mhz to support LTE-advanced with occupied area of 8.3mm^2 in $65nm$ technology. The high throughput achieved is by maintaining 64 Radix2 MAPs decoder working in parallel, benefiting from adopting a contention-free, (QPP) parallel interleaver. However, to support double binary decoding will force to adopt Radix-4 as well, which will be at higher cost for area and critical path as explained in [47].

The related detailed features for all above mentioned work is tabulated in Table 3.1.

	Standard compliant	Architecture	Tech (nm)	Core area (mm^2)	Normalized Core area @90nm (mm^2)	Throughput (Mbps)	F_{clk} (MHz)
[44]	WiMAX, LTE	Eight Radix-4 SISOs	130	10.7	5.35	100 @8iter	250
[45]	WiMAX, LTE(18 modes)	Eight Radix-4 SISOs	90	3.38	3.38	93-186 @6iter	152
[42]	3gpp, CC	ASIP	65	0.5	1	21 @6iter	300
[43]	LTE, HSPA+	Xtensa	45	1.34	2.8	150 @8iter	385
[47]	LTE	64 Radix2 MAPs	65	8.3	16	1280 @6iter	400
[46]	LTE	Four MAPs	65	2.1	4.2	150 @6.5iter	300
[48]	LTE	Eight Radix-4 SISOs	90	2.1	2.1	130 @8iter	275

Table 3.1: State of the art implementations with area normalization

3.2 Proposed Decoder System Architecture

The proposed ASIP architecture considers the base design presented in [40]. In this design, the Turbo decoder system architecture consists of 2 ASIPs interconnected as shown in Figure 3.1. It exploits the various parallelism levels available in turbo decoding including shuffled decoding [40] where the two ASIPs operate in a 1×1 mode. In this mode, one ASIP ($ASIP0$) processes the data in natural order while the other one ($ASIP1$) processes it in interleaved order. The generated extrinsic information messages are exchanged between the two ASIP decoder components via an *Extrinsic Exchange Module*. Furthermore, the system control and configuration are managed by a (*Global Controller & Input Interface Module* and a *Configuration Module*).

Supporting the target standards as summarized in Table 1.1, and competing with other architectures in 3.1, illustrate the need of high throughput turbo decoder architecture with small area. The ASIP payload throughput for double binary mode is $\cong 49.5Mbps@6$ iterations, and $\cong 24.75Mbps@6iter$ for single binary mode. Its logic occupies $0.21mm^2@90nm$, i.e. in order to support 3GPP-LTE standard, throughput should be up-to 150 Mbps, so we need at least a platform 6×6 of the initial ASIPs working in parallel, and a NOC to exchange extrinsic values without conflict. This is a heavy area cost and not competing. Another issue is that the initial ASIP does not support windowing so it cannot process frames bigger than 128 bits unless working in parallel. Also knowing 3GPP code is terminated with tail bits, they are used for

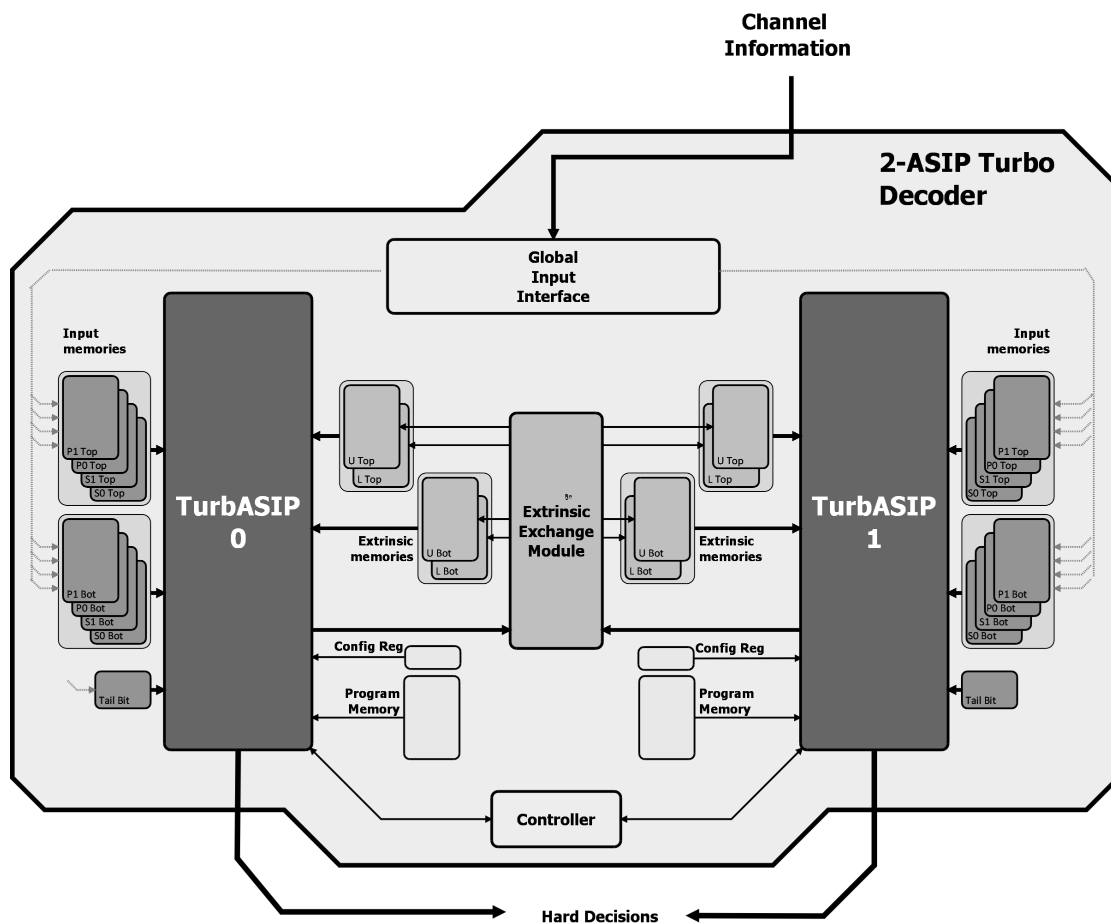

Figure 3.1: Proposed Turbo decoder system architecture

calculating initial β values before starting decoding, which is not supported in the initial ASIP, so it does not support 3GPP.

For all the mentioned issues, the goal of this section is to illustrate how the application of adequate algorithmic and architecture level optimization techniques on an ASIP for Turbo decoding can achieve an attractive and efficient solution in terms of area and throughput. The demonstrated results target channel decoding and particularly turbo decoding. The proposed architecture integrates two ASIP components supporting binary/duo-binary turbo codes and combines several optimization techniques regarding:

- **Architectural optimization:** included optimization of the pipeline structure and applying sliding window technique.
- **Algorithmic optimization:** included trellis compression technique by applying Radix-4 algorithm.
- **Memory reduction:** included techniques to minimize the memory requirements.

3.2.1 Architecture-Level Optimization

In this subsection different architectural optimizations are applied to the initial ASIP to increase its throughput and optimize its performance. As mentioned earlier the optimizations include modifying the pipeline structure and applying sliding window technique to decode large frames.

instruction	pipeline stages							Usage
	FE	DEC	OPF	BM1	BM2	EX	ST	
DATA LEFT	X	X	X	X	X	X	-	6/7
MAX2	X	X	-	-	-	X	-	3/7
DATA RIGHT	X	X	X	X	X	X	-	6/7
EXTCALC	X	X	-	-	-	X	-	3/7
MAX1	X	X	-	-	-	X	X	4/7

Table 3.2: Initial ASIP pipeline usage

3.2.1.1 Pipeline optimization

The initial ASIP pipeline stages as shown in Figure 2.6, consists of 7 stages (**FE**, **DEC**, **OF**, **BM1**, **BM2**, **EX**, **ST**). To calculate the pipeline percentage usage, we should find the usage average of each instruction per cycle. Considering back the assembly code in Listing 2.2 which illustrates the decoding process using the butterfly scheme. The next two lines after ZOLB execute 24 times (half of frame size) which implement left side of butterfly decoding scheme for 24 symbols. After this, the next five instruction (from "DATA RIGHT.." instruction to "MAX1 SYSTEMATIC..") execute 24 times which implement right side of butterfly.

From Table 3.2 we find the initial ASIP pipeline usage percentage $PUP = \frac{\frac{6}{7}+\frac{3}{7}+\frac{6}{7}+\frac{3}{7}+\frac{3}{7}+\frac{3}{7}+\frac{4}{7}}{7} \cong 57\%$. This mean the initial ASIP stay idle for 43%. The architecture optimization is applied by re-arranging the initial ASIP pipeline. The purpose behind this re-arranging is to increase the percentage PUP, this can happen by decreasing the instruction set used in the iterative-loop to generate extrinsic information and hard decision. In this way less clock cycle are needed to generate hard decision so in consequence having higher throughput. Re-arranging the initial ASIP pipeline is done over three versions. For the rest of this thesis the proposed ASIPs in each version will be named as follow: **TurbASIP**$_{v1}$, **TurbASIP**$_{v2}$, and **TurbASIP**$_{v3}$.

Pipeline Optimization Version 1:
As shown in Table 3.2, instructions ("MAX1..", "MAX2..", "EXTCALC..") have low pipeline usage. This is because all these operations are taking place in $(EXstage)$. So 7 clock cycles in total are required to generate extrinsic information for two symbols. The proposed optimization in this version is by merging the instructions [("DATA LEFT..", "MAX2.."),("DATA RIGHT..", "MAX2.."),("EXTCALC..", "MAX2..", "MAX1..")] in one instruction for each group between the parenthesis, so in this case only one instruction is enough to calculate the state metrics and find the maximum values (1.31) & (1.32). In similar way, one instruction will be needed to find maximum A posteriori information (1.30). This can be happened by re-arranging the initial ASIP pipeline and modifying the EX pipeline stage to place the recursion units and max operators in series.

In fact, finding maximum A posteriori information is done in three cascaded levels of 2-input compare units (searching the max out of 8 metric values). Thus, placing them in series with recursion units in one pipeline stage will increase the critical path (i.e. reduce the maximum clock frequency). To break the critical path, two new pipeline stages (MAX1, MAX2) are added after EX pipeline stage to distribute the compare units as shown in Figure 3.2.

To perform the max operation, 28 2-input compare units are used in each recursion unit. Where 24 2-input compare units, located in MAX1 pipeline stage, and are configured to work as 8 4-input compare units. When computation of max operation in state metrics generation is required, 8 4-input compare units are connected to the four outputs of each column of recursion

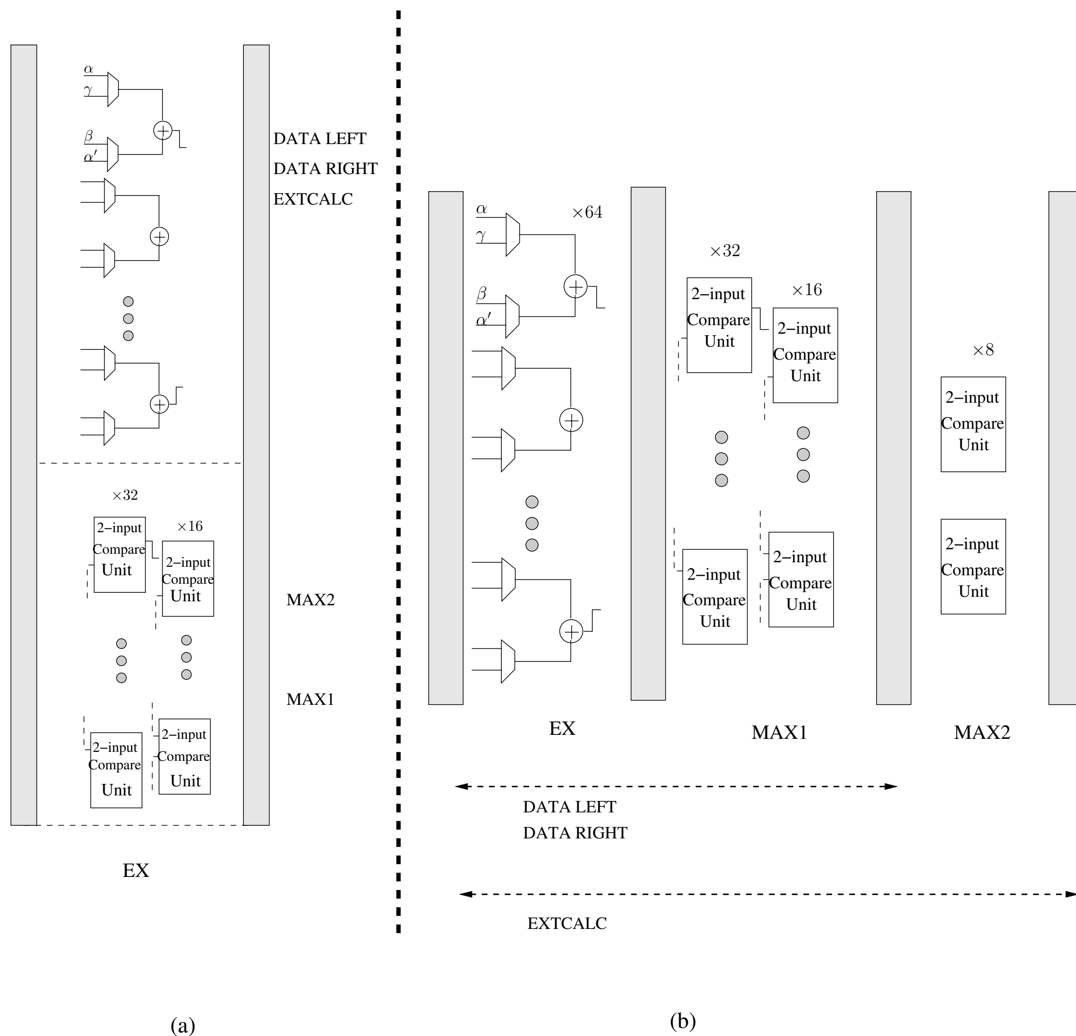

Figure 3.2: Pipeline reorganization (a) Initial ASIP stage EX (b) Optimized pipeline stages (EX, MAX1,MAX2)

unit (selected by "COLUMN" parameter of "DATA.." instruction). In case of extrinsic information computation, the 8 4-input compare units are connected to adder nodes in rows (selected by "LINE" parameter of "EXTCALC.." instruction) . Hence, with 8 4-input compare units user has two biggest candidates per row at the output of 4-input compare units. In contrary to what is explained in 2.3.2, the two largest values in a row are saved in an additional registers EXT, and also additional 4 2-input compare units are added in MAX2 pipeline stage, which can find the maximum between these two candidates which is the extrinsic information for each combination of input bits. In Total 8 2-input compare units and 16 EXT registers are added.
During the decoding process in left butterfly, ACS units (Add, Compare, Select) do the state metric calculations, which take place in EX pipeline stage and then finding the state metrics maximum values in the next clock cycle in MAX1 pipeline stage using instruction "DATA LEFT..". Calculate the state metrics requires two clock cycles so calculating next state metric would be possible after two clock cycles so NOP instruction is put in the left butterfly iteration to compensate one clock cycle as shown in Listing 3.1. During the right butterfly iteration besides finding the state metrics values, ACS units do addition and find maximum A posteriori information (1.30) in (MAX1 & MAX2) pipeline states in two clock cycles using instructions ("DATA LEFT..", "EXTCALC.."). So in total, 4 clock cycles are required to generate extrinsic information for two symbols. Important thing to note that critical path in this optimization remained

instruction	pipeline stages										Usage
	PFE	FE	DEC	OPF	BM1	BM2	EX	MAX1	MAX1	ST	
NOP	X	X	X	-	-	-	-	-	-	-	3/10
DATA LEFT	X	X	X	X	X	X	X	X	-	-	8/10
DATA RIGHT	X	X	X	X	X	X	X	X	-	-	8/10
EXTCALC	X	X	X	-	-	-	X	X	X	X	7/10

Table 3.3: Pipeline usage for TurbASIP$_{v1}$

the same because as shown Figure 3.2 comparators and logic operators in EX stage of the initial ASIP only redistributed to pipeline stages (EX, MAX1, MAX2).

Listing 3.1: TurbASIP$_{v1}$ for Assembly Code

```
1          ZOLB _RW1, _CW1, _LW1
2          NOP
3  _RW1:   DATA LEFT ADD M COLUMN
4  ;save last beta load alpha_init
5          DATA RIGHT  ADD M COLUMN
6  _LW1:   EXTCALC ADD I LINE EXT
```

From Table 3.3 we find, ASIP $PUP = \frac{\frac{8}{10}+\frac{8}{10}+\frac{7}{10}+\frac{3}{10}}{4} = 65\%$. This mean ASIP stay now idle for 35%.

Pipeline Optimization Version 2:
As explained in V.1 optimization instruction NOP is placed in left butterfly iteration because state metric values are calculated in two stages (EX, MAX1) and two clock cycles is required. Giving away NOP instruction would be good approach because it increases the throughput by 25%. This can be done by merging the two stages EX, MAX in one stage. Comparing between the two figures (Figure 3.2 and Figure 3.3). Critical path in V.1 is in EX stage, it should be the sum of critical paths (EX $\oplus$ MAX1) stages of TurbASIP$_{v1}$. We avoided increasing the critical path and in consequence decreasing the throughput by choosing Synopsys design constraints for clock frequency = 500Mhz. However, this choice maintained the critical path as in TurbASIP$_{v1}$. On the other hand, it implies a slight increase in the occupied logic area of $\cong$ 0.02 mm^2. As a result 25% increase for throughput with an increase of 3% of the total area of the whole decoder, which can be considered as a good trade-off.

An assembly code example for the proposed TurbASIP$_{v2}$ is as shown in Listing 3.2. In Left-Butterfly ASIP calculate state metrics and saves them in cross metric memory to use them again in Right-Butterfly, but instruction "DATA LEFT.." writes state metrics value to cross-metric memory in (EX) stage while "DATA RIGHT.." instruction reads the values back in previous stage (BranchMetric2), so when starting right-butterfly loop "DATA RIGHT.." instruction in first loop will read the same value that "DATA RIGHT.." is writing. To avoid this conflict in cross-metric memory, NOP instruction is placed @4 and executed one time for 1 clock cycle delay.

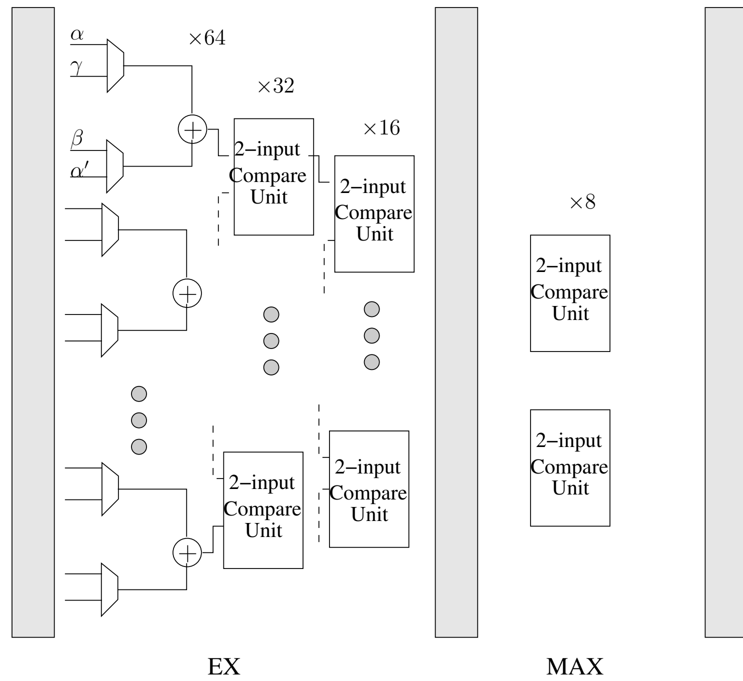

Figure 3.3: Modified pipeline stages (EX, MAX)

instruction	pipeline stages									Usage
	PFE	FE	DEC	OPF	BM1	BM2	EX	MAX	ST	
DATA LEFT	X	X	X	X	X	X	X	-	-	7/9
DATA RIGHT	X	X	X	X	X	X	X	-	-	7/9
EXTCALC	X	X	X	-	-	-	X	X	X	6/9

Table 3.4: Pipeline usage for TurbASIP$_{v2}$

Listing 3.2: TurbASIP$_{v2}$ Assembly Code

```
1           ZOLB _RW1, _CW1, _LW1
2  _RW1:    DATA LEFT ADD M COLUMN
3  ;save last beta load alpha_init
4  _CW1:    NOP
5           DATA RIGHT  ADD M COLUMN
6  _LW1:    EXTCALC ADD I LINE EXT
```

From Table 3.4 we find, ASIP $PUP = \frac{\frac{8}{9}+\frac{8}{9}+\frac{6}{9}}{3} \cong 74\%$. This mean ASIP stay now idle for 26%.

Pipeline Optimization Version 3:
Reaching toward optimal design to achieve efficient architecture and power is related directly to number of decoded bits per cycle as will be explained and formulated in the section of (System Characteristics and Performance). One way to increase the decoded bits per cycle is to increase the pipeline usage percentage PUP. Analyzing the pipeline stages usage for TurbASIP$_{v2}$ optimization in Table 3.4. The table indicates the average Pipeline Usage Percentage: $PUP = \frac{\frac{7}{9}+\frac{7}{9}+\frac{6}{9}}{3} \cong 74\%$. This sub-optimal usage is caused by the idle state of instructions "DATA LEFT" and "DATA RIGHT" in pipeline stages (MAX, ST) and instruction "EXTCALC"

instruction	pipeline stages								Usage
	PFE	FE	DEC	OPF	BM1	BM2	EX	ST	
MetExtCALC LEFT	X	X	X	X	X	X	X	-	7/8
MetExtCALC RIGHT	X	X	X	X	X	X	X	X	8/8

Table 3.5: Pipeline usage for TurbASIP$_{v3}$

in (BM1, BM2, EX). A proposed method to increase the PUP value is by merging the two instructions (DATA ..., EXTCALC ...), so in right-butterfly both (state metric, extrinsic information) calculations are processed in the same cycle. However the challenge in this proposal is to keep the same balance of the functional units (Adders, Registers, Multiplexers) to avoid increasing the complexity, and also to maintain the same critical path. In fact, the proposed merge causes to duplicate part of the functional units. On the other part, it allows to remove several registers and multiplexers which were used for buffering and functional units sharing. Figure 3.4 illustrates the architecture of the proposed recursion unit. In this architecture, each one of the 32 Adder Node Functional units (ANF) integrates 2 adders (instead of one) to compute the two additions ($\alpha+\gamma$ or $\beta+\gamma$) then ($\alpha+\beta+\gamma$) in the same clock cycle. On the other hand, per ANF, the following logic has been removed: 2 10-bit registers used to buffer $\alpha+\gamma$ or $\beta+\gamma$ (RADD Reg. and RC Reg.) and 2 multiplexers used to share the single adder. Furthermore, additional 8 comparators (each of 4 inputs) have been added instead of sharing the existing ones. This allowed removing the corresponding multiplexers besides 4 10-bit registers (EXT Reg.).

The summary of the removed and added logic for the two recursion units of the ASIP is presented in Table 3.6 which shows that the level of complexity is almost remained the same. Synthesis results of the overall ASIP logic present a slight increase of 0.004mm^2 in CMOS 65nm technology.

Logic removed			Logic added		
Logic	Data width (bits)	#	Logic	Data width (bits)	#
RADD Reg	10	64	Adder	10	64
RC Reg	10	16	Adder	10	48
EXT Reg	10	8	MUX	10	48
MUX	10	152	-	-	-

Table 3.6: Comparison table for added and eliminated logic for the proposed architecture

Table 3.7 summaries and compares the results in terms of pipeline usage PUP, logic area, throughput, and decoding speed. It is worth noting that the memory area remains identical with the proposed optimization, and thus not considered in this table.

Figure 3.5 compares the critical path for the proposed optimization architecture TurbASIP$_{v3}$ with previous optimization TurbASIP$_{v2}$, which is in the EX pipeline stage. The compare

	PUP	logic area mm^2 @65nm	Decoding speed Bits/clk/iter	Throughput Mbps
Initial ASIP	57 %		0.083	49.5 @6iter
TurbASIP$_{v1}$	65 %		0.082	86.6 @6iter
TurbASIP$_{v2}$	74 %	0.084	1.33	115.5 @6iter
TurbASIP$_{v3}$	94 %	0.0845	2	164 @6iter

Table 3.7: Results comparison in terms of PUP, logic area, decoding speed and throughput

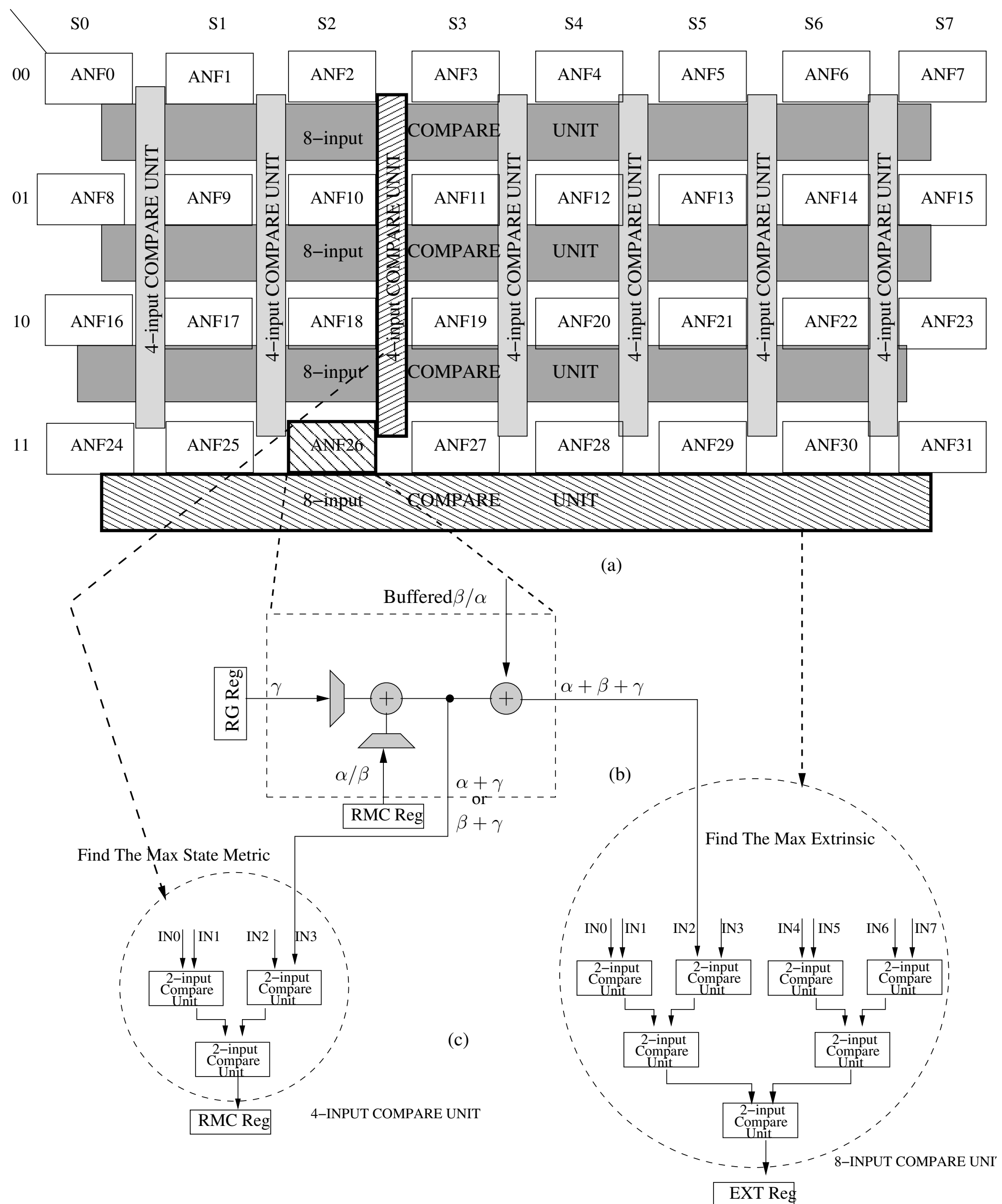

Figure 3.4: Architecture of the proposed recursion unit (a) Forward Recursion Unit composed of 32 ANF (b) 8 4-input Compare Units used for state metric computation and 4 8-input Compare Units used for extrinsic computation (c) Adder Node Forward

units are equivalent to the combination of 1 (adder) and 1 (multiplexer). The critical path for TurbASIP$_{v2}$ integrates 6 multiplexers + 3 adders while that in TurbASIP$_{v3}$ integrates 4 multiplexers + 5 adders. This illustrates how the proposed optimization does not impact the ASIP maximum clock frequency.

Figure 3.6 depicts the pipeline details organization in 8 stages for the proposed TurbASIP$_{v3}$. The numbers in brackets indicate the equations (referred in section 1.4.1.2) mapping on the corresponding pipeline stage. The extrinsic information format, at the output of the ASIP, is also depicted in the same figure for the two modes SBTC and DBTC.

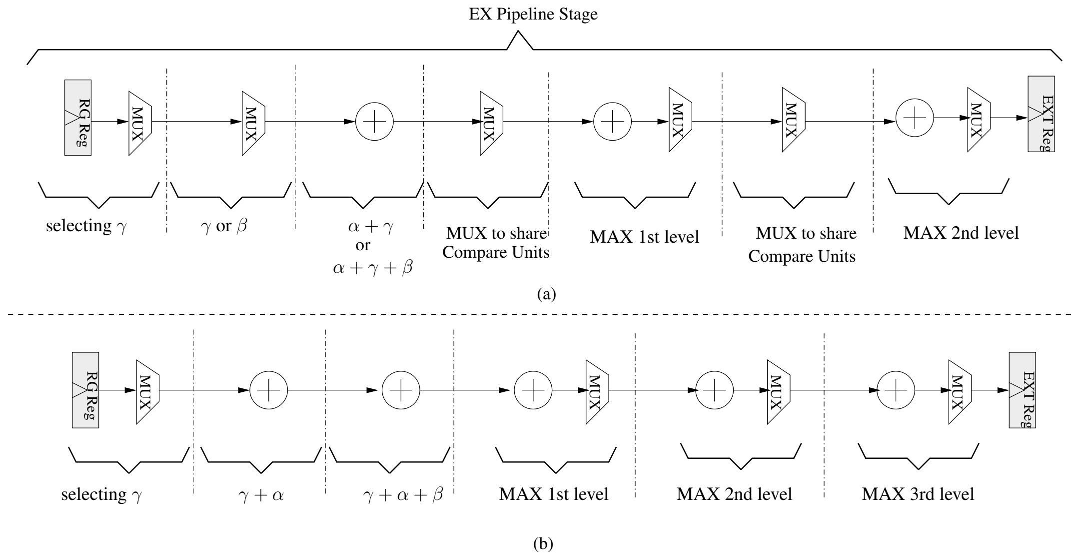

Figure 3.5: Critical path comparison: (a) Critical path for TurbASIP$_{v2}$, (b) Critical path for TurbASIP$_{v3}$

3.2.1.2 Windowing

Another proposed Architectural optimization concerns the implementation of windowing to process large block-size. This is achieved by dividing the frame into N windows, where the window maximum size supported is 128 bits. Figure 3.7 shows the windows processing in butterfly scheme, i.e. ASIP calculates the α values (forward recursion) and β (backward recursion) simultaneously, and when it reaches half of the processed window (left-butterfly) and start the other half (right-butterfly), ASIP can calculate the extrinsic information on the fly along with α and β calculations. State initializations $(\alpha_int(w^i_{(n)})$ across windows are done by message passing where $(\alpha_int(w^i_{(n)})$ is used in next window w_{n+1} of the same iteration (i), while state initializations $\beta_int(w^i_{(n)}))$ are stored to be transfered across windows in next iteration $(i{+}1)$.We firstly proposed to store them in array of registers internally in the ASIP. Each register (R_n) in this array is coordinated with one window (w_n) and holds the beta values for the last symbol of this window to be used in next iteration as initial value in the previous window (w_{i-1}). Since the maximum window size is 128 bits, 48 windows are needed to cover all LTE block-sizes, so 48$\times$80 array of registers is added. But this internal array of register is replaced by single port memory because of the area cost. The obtained area is $\approx 0.07mm^2$ (synthesized using Synopsys tools and $65nm$ CMOS technology), comparing with similar size for single port memory of size $0.0054mm^2$ (see Table 3.12)

Since ASIP apply butterfly scheme and calculate forward and backward state values simultaneously, it should reads window's symbols from two edges at the same time. That is why there are two indexes one points to the beginning of the window (W_{addrA}) and increment to read next symbol. The other index (W_{addrB}) points to the end of the window and decrement to read previous symbol. Supporting addressing for windowing was developed in two steps:

- **Fixed size windowing**:
  This approach proposed two index registers with fixed bit width for (W_{addrA},W_{addrB}), besides a counter (W_i) that increment each time ASIP process new window. The physical address $AddrM$ to read symbol values from Input memory equal the concatenation of both values (index, counter) $AddrM = W_i + W_{addrA}$ or $AddrM = W_i + W_{addrB}$. The drawback for this solution is that when the ASIP processes less than 128 bits (maximum

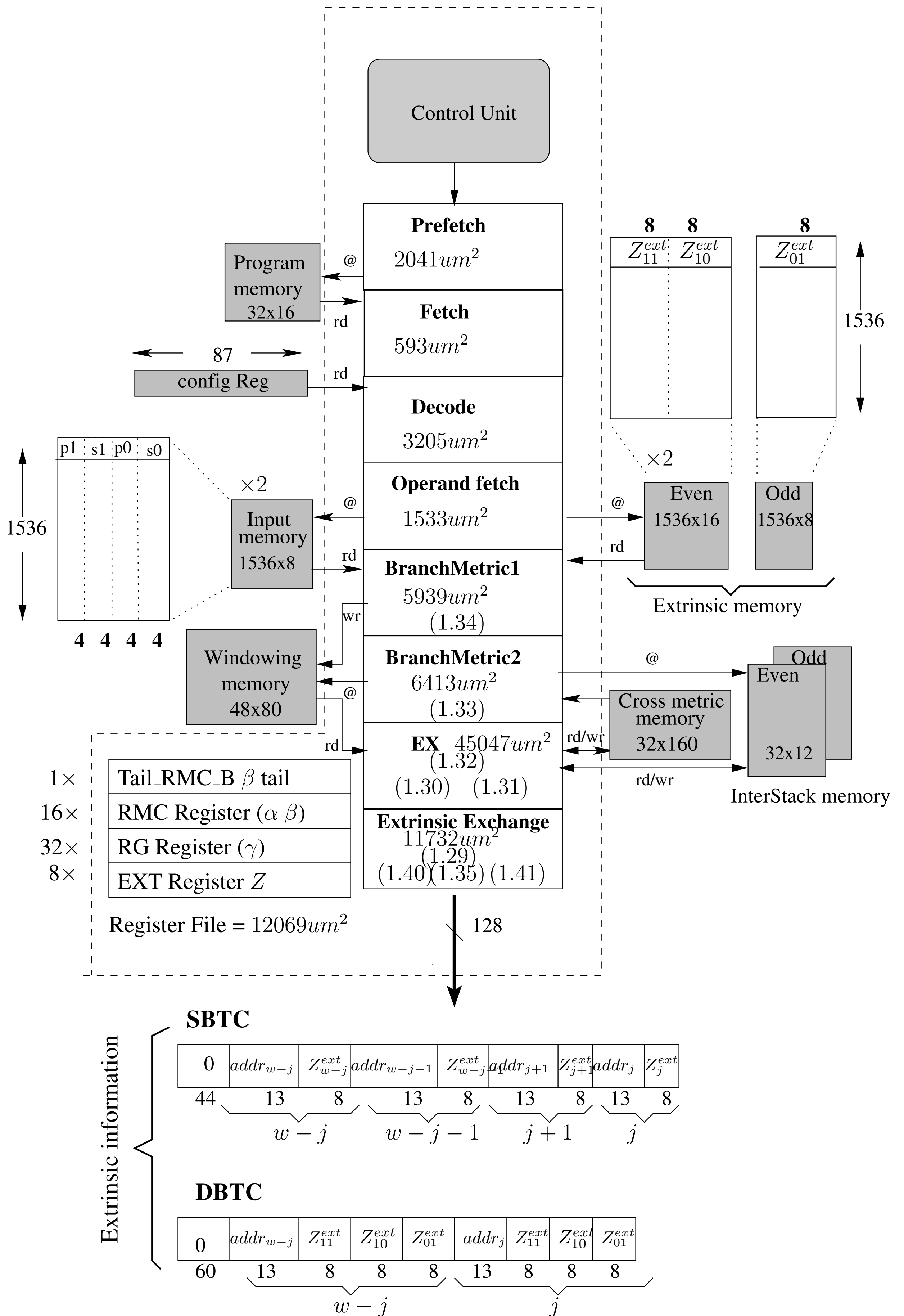

Figure 3.6: TurbASIP$_{v3}$ pipeline architecture

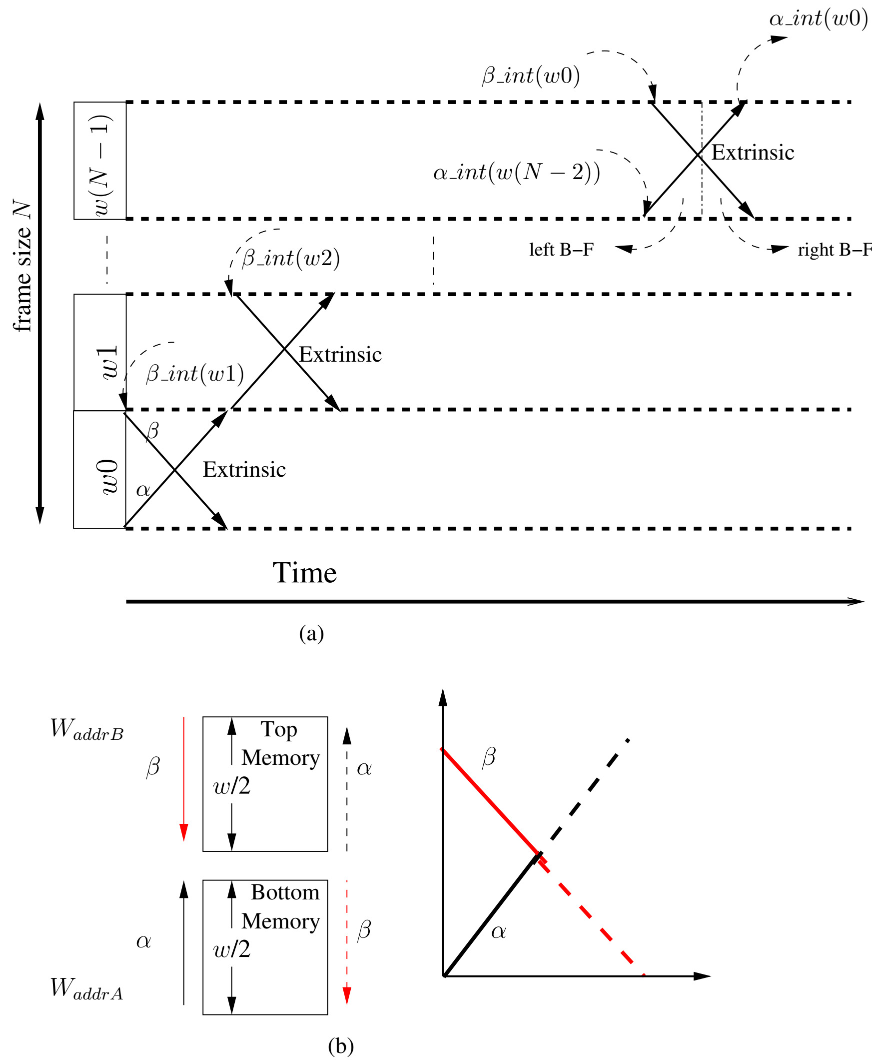

Figure 3.7: Windows processing in butterfly scheme (a) Windowing in butterfly computation scheme, (b) Window addressing in butterfly

window size) in this case parts of the channel and extrinsic memories will be unused as illustrated in Figure 3.8. Each unused part corresponds to the difference between the actual window size and the maximum window size.

- **Flexible Window Size** :
  To overcome the above described issue, additional two 16 bits indexes are found ($addr_A$), ($addr_B$). These two indexes can address the whole channel and extrinsic memories. Index $addr_A$ will increase every time new symbol is decoded, while $addr_B$ decrement each time new symbol is decoded, and when ASIP process new window ($addr_B = addr_A + (w/2) - 1$) where w is the window size. While (W_{addrA},W_{addrB}) are used to address Cross_metric memory.

An assembly code example for decoding WiMAX of block size 1920 symbol is shown in Listing 3.3. To decode such a frame, 30 windows are need ($W_N = \frac{1920}{64}$). SET_WINDOW_ID Instruction sets the window_id register counter to zero, which represents W_i. Two nested loops ”REPEAT UNTIL..” is utilized. The upper loop is for iterations and the inner loop is for windows. Every time new window start decoded, corresponding initial values β from previous iter-

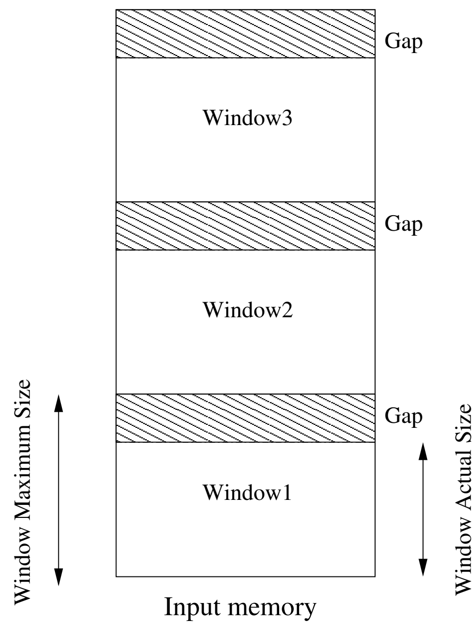

Figure 3.8: Unused memory regions in fixed size windowing

ation are read from Windowing memory with instruction (W_LD_BETA). Every time a window is decoded, β values of last symbol is written to Windowing memory, and window_id register counter is incremented by one with instruction (EXC_WINDOW). When new iteration is started window_id register counter is set to zero again.

Listing 3.3: Processing windowing

```
1           SET_WINDOW_ID 0
2           REPEAT UNTIL _loop 4 TIMES
3           PUSH
4           REPEAT UNTIL _loop1 29 TIMES
5           ZOLB _RW2,_CW2,_LW2
6           W_LD_BETA
7  _RW2:    DATA LEFT READ_EXT ADD M COLUMN
8  _CW2:    NOP
9           DATA RIGHT READ_EXT ADD M COLUMN
10 _LW2:    EXTCALC READ_EXT ADD I LINE EXT
11          EXC_WINDOW
12 _loop1 : NOP
13          POP
14          SET_WINDOW_ID 0
15 _loop:   NOP
```

3.2.2 Algorithmic-Level Optimization

In the initial ASIP, SBTC throughput equals half of DBTC throughput because the decoded symbol is composed of $1bit$ in SBTC while it is 2 $bits$ in the DBTC mode. Trellis compression is applied to overcome this bottleneck as explained in Section 1.4.2.1. This makes the decoding

	Throughput Mbps (SBTC)
Initial ASIP	24.75@6iter
TurbASIP$_{v3}$	164@6iter

Table 3.8: Throughput comparison between initial ASIP and TurbASIP$_{v3}$ after applying Radix-4

calculation for SBTC similar to DBTC as presented in (1.37), (1.39) and (1.38) so no additional ACS units are added. The only extra calculation is to separate the extrinsic information to the corresponding symbol as presented in (1.40) and (1.41) and the cost for its hardware implementation is negligible. Figure 3.9 depicts butterfly scheme with Radix-4, where the numbers indicate the equations (referred in Section 1.4.2.1). In this case four bits (single binary symbols) are decoded each time.

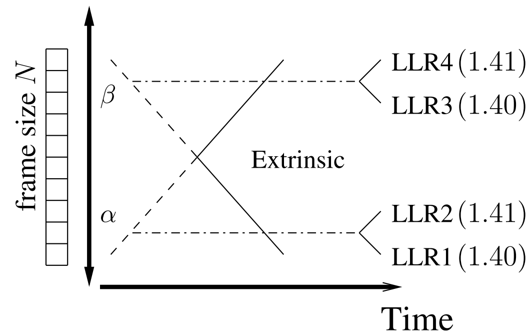

Figure 3.9: Butterfly scheme with Radix-4

As summarized in 1.3.1, 3gpp encoder terminates the trellis by taking the tail bits from the shift register feedback after all information bits are encoded. Tail bits are padded after the encoding of information bits. In the decoder side to calculate β initial value, the tail bits should be decoded before decoding the frame. A special (tail_bit) memory is obtained to save the tail bits, as shown in Figure 3.6.

An assembly code example for processing tail bit is as shown in Listing 3.4. The first two instructions "DATA LEFT.." with parameter "TAILREAD" reads tail bits from (tail_bit) memory and calculate initial β to be saved it in a special register **Tail_RMC_B**, then to be used every iteration.

Listing 3.4: Processing Tail bits

```
1           SET_CONF 3gpp
2  ;Reading Tail bit to calculate initial beta values
3           DATA LEFT WITHOUT_EXT ADD M COLUMN TAILREAD
4           DATA LEFT WITHOUT_EXT ADD M COLUMN TAILREAD
5  ZOLB _RW0,_CW0,_LW0
6  _RW0:    DATA LEFT WITHOUT_EXT ADD M COLUMN
7  _CW0:    NOP
8           DATA RIGHT WITHOUT_EXT ADD M COLUMN
9  _LW0:    EXTCALC WITHOUT_EXT ADD I LINE EXT
```

Table 3.8 shows throughput comparison after applying Radix-4 on TurbASIP$_{v3}$. The throughput is increased more than 6.5 times of the initial ASIP original throughput.

3.2.3 Memory-Level Optimization

Memory reduction is key for an efficient ASIP implementation, since around 65% of TurbASIP area is occupid by memories. The goal is thus to share and optimize as much RAM as possible for both SBTC and DBTC functionalities. Different memory structure optimization levels are studied and some are implemented:

3.2.3.1 Normalizing extrinsic information and memory sharing

Concern the normalization of the extrinsic information as presented in (1.36). In an m-ary turbo code where a symbol is represented in m bits, the number of possible symbol-level extrinsic information values is $2^m - 1$ [49]. Since the value of m is two for double-binary turbo codes, three symbol-level extrinsic information values are defined. This normalization reduces the extrinsic memory by 25% because $(Z_k^{ext}(d(s,s') = 00))$ is not stored. For BPSK channel with quantization of 4 bits, extrinsic information can be quantized in 8 bits.

It is efficient to organize the input and extrinsic memories to be fully shared between SBTC and DBTC. Input memories contain the channel LLRs Λ_n and they are quantized to 4 bits each. The proposed sharing is to have the channel LLRs values systematics (S_0^n, Sn_1) and parties (Pn_0, Pn_1) in DBTC mode for same symbol to be stored in the same memory word. However in SBTC mode, each memory word stores the LLRs values $(S_0^n, S_0^{n+1}, P_0^n, P_0^{n+1})$ for two consecutive bits (single binary symbols). The same approach is proposed for extrinsic memories. As normalized extrinsic values are quantized to 8 bits, in DBTC mode, values $\gamma_{01}^{n.ext}, \gamma_{10}^{n.ext}, \gamma_{11}^{n.ext}$ related to same symbol are stored in the same memory word. While in SBTC mode, each memory word stores the extrinsic values $\gamma_1^{n.ext}, \gamma_1^{n+1.ext}$ for two consecutive bits. In this way the memory resources in two turbo code modes (SBTC/DBTC) are efficiently shared.

3.2.3.2 Bit-level extrinsic information exchange method

Study in [50] proposed two conversions: symbol-to-bit conversion and bit-to-symbol conversion of extrinsic information. With simple conversions, the number of values to be exchanged can be reduced from the number of possible symbols, $2^m - 1$, to the number of bits in a symbol, m, without inducing any modifications to conventional symbol-based double-binary SISO decoders. Therefore, the proposed method can reduce the size of extrinsic information memory in double binary decoding by around 30% regardless of quantization. Table 3.9 and 3.10 demonstate the conversion equations from symbol-to-bit and bit-to-symbol respectivly, where the input symbol u_k consists of pair of two bits, A and B.

$$Z_A = max(Z_{10}^{ext}, Z_{11}^{ext}) - max(0, Z_{01}^{ext})$$
$$Z_B = max(Z_{01}^{ext}, Z_{11}^{ext}) - max(0, Z_{10}^{ext})$$

Table 3.9: Symbol-to-bit conversion

	$(A > 0, B > 0)$	$(A > 0, B < 0)$	$(A < 0, B > 0)$	$(A < 0, B < 0)$
$Z^{ext}(01) =$	$max(Z_A, Z_B) - Z_A$	0	Z^B	Z^B
$Z^{ext}(10) =$	$max(Z_A, Z_B) - Z_B$	Z_A	0	Z^A
$Z^{ext}(11) =$	$max(Z_A, Z_B)$	$Z_A + Z_B$	$Z_A + Z_B$	0

Table 3.10: Bit-to-symbol conversion

To study the Bit-Level Extrinsic conversion technique, it was simulated in fix point soft-model and the BER performance for WiMAX, code rate 1/3, for frame size 1728 bits, was compared with the conventional method (Symbol-level extrinsic information exchange). This lead to a significant degradation of the signal-to-noise ratio, around 0.3 db, which will require additional iteration to enhance the performance but this will decrease the throughput and therefore this method is not implemented.

3.2.3.3 Interleaver generator

Previous design was utilizing interleaver memory table in order to fetch interleaving address for each extrinsic information value. The draw-back of this method is the high area occupation due to the need of storing long address table for big frames such as LTE of block-size 6144 bits. So to eliminate the interleaver memories in the proposed architecture, interleaver address generator is utilized instead. To generate interleaving address for SBTC (LTE) and DBTC (WiMAX, DVB-RCS), two different interleaver generators are required, QPP interleaver generator to support LTE and ARP interleaver generator to support WiMAX and DVB-RCS.

QPP Interleaver Generator: The modulo operation in (1.8 is difficult to implement in hardware if the operands are not known in advance. However, a low complexity hardware implementation is proposed in [47]. The QPP interleaving addresses are usually generated in a consecutive order (with step size of d). Since the proposed work adapts Radix-4 technique and two extrinsic information are generated and should be addressed simultaneously, two interleaving addresses for two consecutive orders should be generated. To achieve this task, two QPP generators are proposed, one for odd interleaving addresses and the other for even interleaving addresses. The step size of every generator $d = 2$. The QPP interleaving address can be computed in a recursive manner. The interleaving is computed as follow:

$$\prod(j) = (f_1 \times j + f_2 \times j^2)\%N$$

In the following cycles as j increment by 2, $\prod(j+2)$ is computed recursively as follows:

$$\prod(j+2) = (f_1 \times (j+2) + f_2 \times (j^2+2))\%N \tag{3.2}$$

or we can represent $\prod(j+2)$:

$$\prod(j+2) = \prod(j) + g(j) \tag{3.3}$$

where $g(j) = (8f_2 j + 4f_2 + 2f_1)\ mod\ N$, we note that $g(j)$ can also computed recursively.

$$g(j+2) = (g(j) + 8f_2)\%N \tag{3.4}$$

The even QPP generator starts with initial value $g(0)$ and the odd starts with initial value $g(1)$. The initial values $g(0)$ and $g(1)$ are the seeds and should be pre-computed:

$$g(0) = (4f_2 + 2f_1)\%N \tag{3.5}$$

$$g(1) = (12f_2 + 2f_1)\%N \tag{3.6}$$

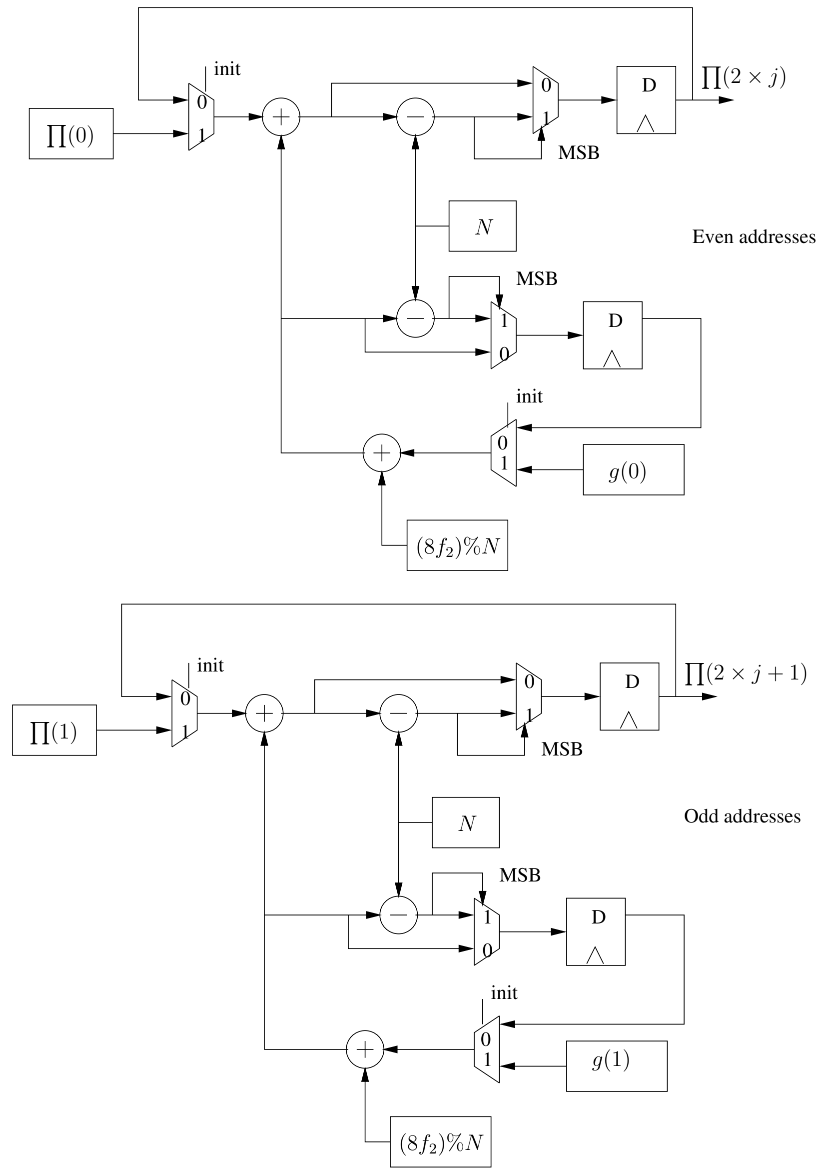

Figure 3.10: QPP Interleaver generators for even and odd addresses with step-size=2

Figure 3.10 depicts the hardware architecture for the two QPP generators, where even generator starts with $j = 0$ and the odd generator starts with $j = 1$, then both of them are incremented by 2 every clock cycle.

ARP Interleaver Generator: The ARP interleaving addresses can also be computed recursively. In this regard, we propose a different formulation of the mathematical expressions in order to optimize the underlined hardware architecture. The basic formula of ARP interleaving is given in (3.7) for index j.

$$\prod(j) = (P_0 \times j + P_j + 1)\%N \tag{3.7}$$

where

$$
\begin{aligned}
P &= 0 && \text{if } j\%4 = 0 \\
P &= \frac{N}{2} + P_1 && \text{if } j\%4 = 1 \\
P &= P_2 && \text{if } j\%4 = 2 \\
P &= \frac{N}{2} + P_3 && \text{if } j\%4 = 3
\end{aligned}
\tag{3.8}
$$

For index $j+1$, the above equation can be written as follows:

$$
\begin{aligned}
\prod(j+1) = (P_0 \times (j+1) + P_{j+1} + 1)\%N = \\
= (P_0 \times (j+1) + P_j - P_j + P_{j+1} + 1)\%N
\end{aligned}
\tag{3.9}
$$

This expression leads to the following recursive equation:

$$
\prod(j+1) = \prod(j) + Seed \tag{3.10}
$$

where $Seed = (P_{j+1} - P_j + P_0)\%N$. Using the formula in (3.8), there are four cases for $Init$ as follows:

$$
\begin{aligned}
Seed0 &= (P_0 - N/2 - P_3)\%N && \text{if } (j+1)\%4 = 0 \\
Seed1 &= (P_0 + N/2 + P_1)\%N && \text{if } (j+1)\%4 = 1 \\
Seed2 &= (P_0 + P_2 - N/2 - P_1)\%N && \text{if } (j+1)\%4 = 2 \\
Seed3 &= (P_0 + P_3 + N/2 - P_2)\%N && \text{if } (j+1)\%4 = 3
\end{aligned}
\tag{3.11}
$$

Using this formulation, the proposed ARP interleaving generator is illustrated in Figure 3.11. This architecture presents lower complexity that the one proposed in [51] (use of two adders rather than four).

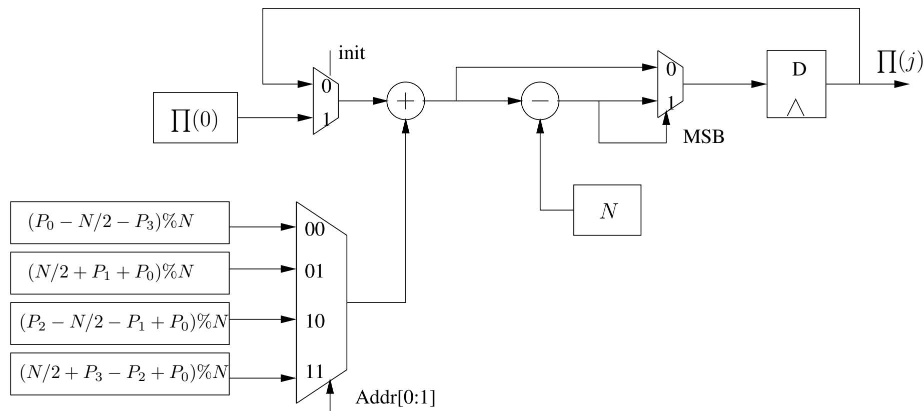

Figure 3.11: ARP interleaver generator

Interleaving address generator with butterfly scheme: Since the proposed turbo decoder uses butterfly scheme, the interleaving addresses are required in the right-butterfly when sending the extrinsic information in both backward and forward directions.

Having address generators in both directions will incurs significant area and control overheads due to the discontinuity of generating addresses (generating addresses only in right butterfly of every sub-block). This discontinuity multiplies the number of required initial values (Seed values). To avoid this issue, we propose to use only interleaving address generators in forward direction, so they generate address continuously in both left and right butterfly. In this case, the interleaving addresses generated in left butterfly can be used later for backward direction of the right butterfly. To that end, those addresses are buffered in small size stack memories (InterStack) of size 12×32. Fig. 3.12 illustrates the proposed interleaving address generation and stack memories in butterfly scheme. It is worth to note that in LTE there is a need for two stack memories (InterStack1, InterStack2) for even and odd addresses respectively because of using RAdix-4 which decode two bits simultaneously. As for WiMAX and DVB-RCS only InterStack1 is used.

As a conclusion a conventional interleaving memory of size $6144\times13\cong 80$Kbits is replaced by $2\times12\times32 = 768$bits and additional very low complexity logic for address generators.

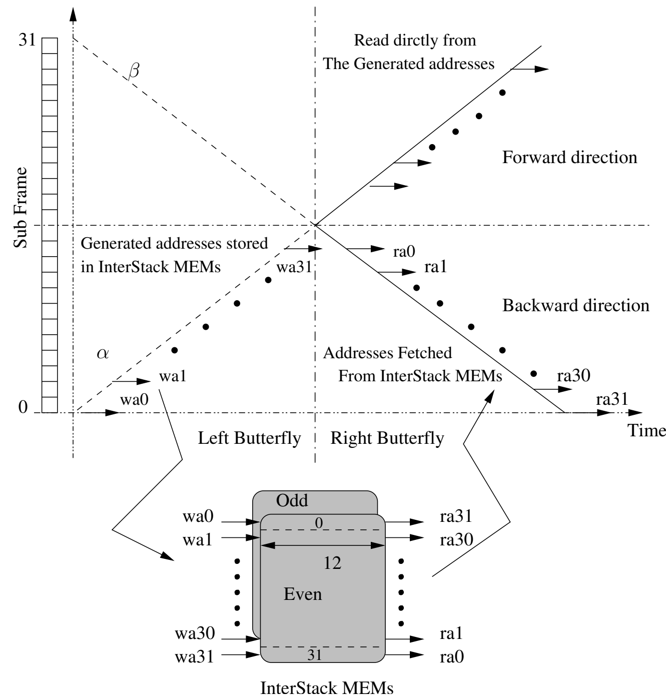

Figure 3.12: Interleaving address generator with butterfly scheme

3.2.3.4 Restricting the trellis support

The main goal from restricting the trellis support to the most used standards (WiMAX, DVB-RCS, 3gpp), rather than leave it open to all different possibilities is to reduce the complexity of multiplexing logic in the recursion units, and also to eliminate the Config memories which store the trellis definitions.

3.3 Extrinsic Exchange Module

In SBTC, Radix-4 with butterfly scheme imply the generation of four extrinsic information every clock cycle in right butterfly. These four extrinsic information should update the extrinsic memories of the other component decoder. One of QPP features is that the generated interleaved address $\prod(j)$ used in LTE has the same even/odd parity as j. For that reason, extrinsic memories are split to four banks (TOP1, TOP2, BOT1, BOT2). Figure 3.13 explains the functionality of the Extrinsic Exchange Module to manage parallel memory access conflicts. If LLRs of odd addresses 1 and 3 are not in conflict then they are sent to memories TOP1 and BOT1 accordingly. Otherwise, LLR with address 3 is stored in the FIFO2. Similarly, LLRs of even addresses are handled. Since no extrinsic information is generated during left butterfly of next processed window, the Extrinsic Exchange Module is free to update the remaining LLRs from the FIFOs.

Similar technique is used in DBTC turbo decoding, but only one FIFO is used since just two extrinsic information are generated every cycle in right butterfly.

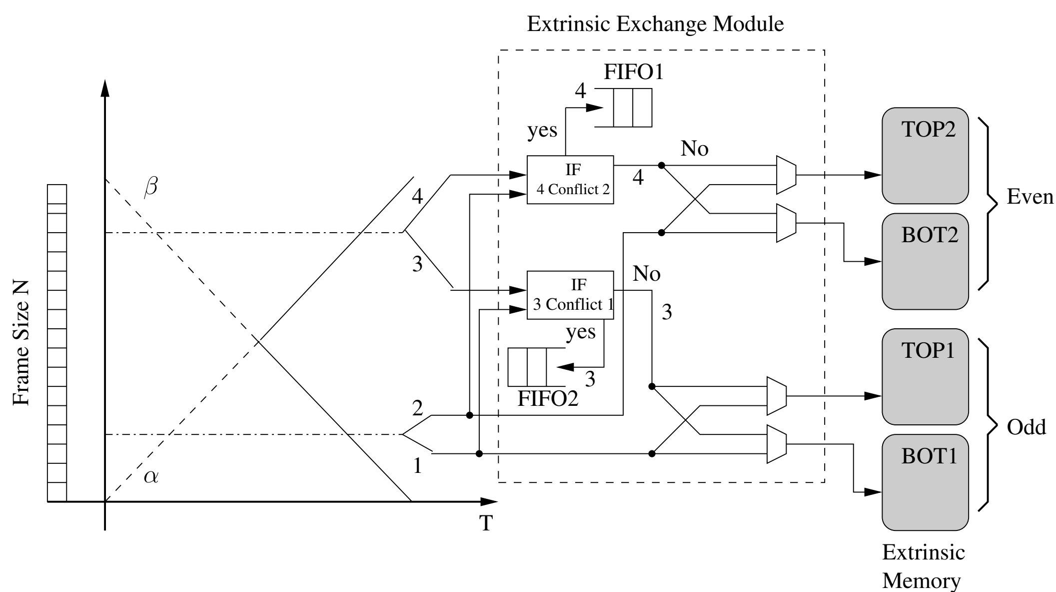

Figure 3.13: Proposed architecture for the Extrinsic Exchange Module

3.4 ASIP Dynamic Reconfiguration

The proposed turbo decoder supports multi-standards (DVB-RCS, LTE, WiMAX) with wide range of block-sizes. In order to speed up the reconfigurability of the proposed architecture, we propose to unify SBTC and DBTC instruction program. To achieve that, all necessary parameters are placed in special register (Config_Reg) of 87 bits. Figure 3.14 details the reserved bits for the needed parameters, and Table 3.11 explains how each parameter can be calculated where FS stands for Frame size in symbols.

The complete unified assembly code is shown in Listing 3.5. LOAD_CONFIG instruction reads from Config_Reg register to define the standard's mode, number of iterations, extrinsic scaling factor, window size, and last window size. The window maximum size supported is 128 bits, so any frame bigger than 128 bits is divided to windows of size 128 bits except for the last window which will be the remaining. ZOLB is the zero overhead loop instruction, instruction at @12 "MetExtCALC LEFT.." executes $W1$ times (half of window size) to compute the state

Parameter	Description	Formula	Size in Bits
NoI	N# of Iterations	$Given\ by\ the\ user$	4
S	Standard (LTE/WiMAX/DVB-RCS)	$S = \begin{cases} 1 & if(LTE) \\ 0 & otherwise \end{cases}$	1
I	intra-symbol permutations	$I = \begin{cases} 1 & if(DVB) \\ 0 & otherwise \end{cases}$	1
SF	Scaling factor	$SF = \begin{cases} 0.4375 & if(LTE) \\ 0.75 & otherwise \end{cases}$	4
$W1$	Window Size−1*	$W1 = \begin{cases} 31 & if(FS \geq 64) \\ (FS/2) - 1 & otherwise \end{cases}$	5
$W2$	Last Window Size−1*	$W2 = \begin{cases} 31 & if(FS\%64 = 0) \\ \frac{(FS - floor(\frac{FS}{64}) \cdot 64)}{2} - 1 & otherwise \end{cases}$	5
NoW	N# of Windows	Ceiling($\frac{FS}{64}$)	6
$NoWL$	N# of Windows−1*	$NoW - 1$	6

* The −1 is required because of the REPEAT & WIN_REPEAT instructions that execute for "the given value +1" times

Table 3.11: Parameters definition, method of calculation, and number of reserved bits

metrics in left butterfly. The instruction at @14 "MetExtCALC RIGHT.." executes $W1$ times to compute the state metrics and extrinsic information in right butterfly. The only exception is when decoding the last window, ZOLB will iterate $W2$ times.

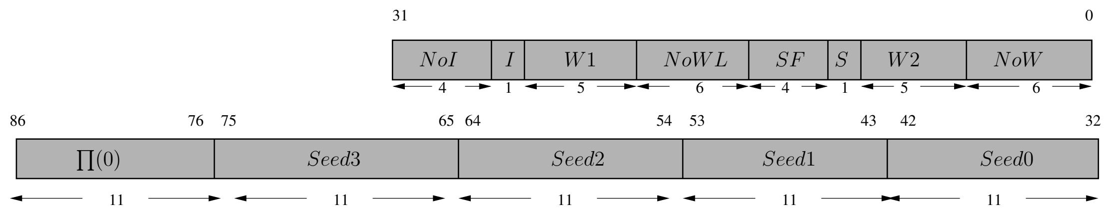

Figure 3.14: Config_Reg reserved bits

Listing 3.5: unified assembly program (3gpp/WiMAX/DVB-RCS)

```
1   ; Read from config Register
2           LOAD_CONFIG
3           set_window_id 0
4   ; Tail bit decoding
5           MetExtCALC LEFT NO_EXT TailRead
6           MetExtCALC LEFT NO_EXT TailRead
7   ; decode first iteration without extrinsic
8           WIN_REPEAT UNTIL _loop0
9           ZOLB _RW0, _CW0, _LW0
10          W_LD_BETA
11  _RW0:   MetExtCALC LEFT NO_EXT
12  _CW0:   NOP
13  _LW0:   MetExtCALC RIGHT NO_EXT EXT
14          EXC_WINDOW
15          NOP
16  _loop0: NOP
17          SET_WINDOW_ID 0
18  ; decode next iterations with extrinsic
19          REPEAT UNTIL _loop1
20          NOP
21          WIN_REPEAT UNTIL _loop2
22          ZOLB _RW2, _CW2, _LW2
23          W_LD_BETA
```

```
24 _RW2:    MetExtCALC LEFT READ_EXT
25 _CW2:    NOP
26 _LW2:    MetExtCALC RIGHT READ_EXT EXT
27          EXC_WINDOW
28          NOP
29 _loop2:  NOP
30 _loop1:  SET_WINDOW_ID 0
31 ; decode hard decision
32          WIN_REPEAT UNTIL _loop3
33          ZOLB _RW4,_CW4,_LW4
34          W_LD_BETA
35 _RW4:    MetExtCALC LEFT READ_EXT
36 _CW4:    NOP
37 _LW4:    MetExtCALC LEFT READ_EXT HARD
38          EXC_WINDOW
39          NOP
40 _loop3:  ST_DEC
```

Figure 3.15, copied image of the configuration software interface, so the user can change the parameters at runtime, which immediately will write to Config_Reg register on FPGA.

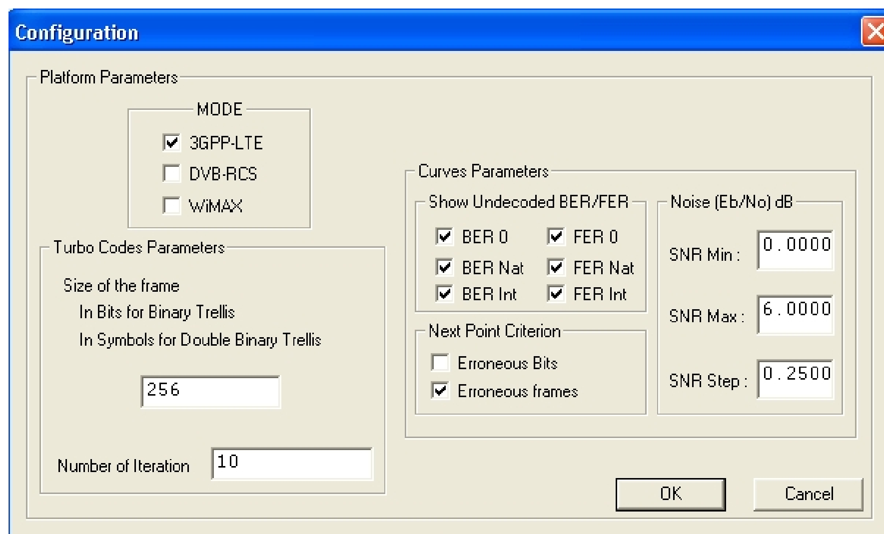

Figure 3.15: Parameters can be changed in runtime

In addition, the rest of the Config_Reg register is divided to 5 parts (11bits each) to store the initial seed values for interleaver generators depending on the decoding standard, besides the initial interleaving address ($\prod(0)$).

3.5 System Characteristics and Performance

The ASIP was modeled in LISA language using Synopsys (ex. CoWare) Processor Designer tool. Generated VHDL code was validated and synthesized using Synopsys tools and 65nm CMOS technology. Obtained results demonstrate a logic area of 0.0845 mm^2 per ASIP with maximum clock frequency of F_{clk}=500MHz. Table 3.12 lists the required memories for each

ASIP. Thus, the proposed turbo decoder architecture with 2 ASIPs occupies a logic area of 0.208 mm^2 with total memory area of 0.436 mm^2. Memories were modeled using low power ST library @65nm CMOS technology. With these results, the turbo decoder throughput can be computed through the (5.12). An average $N_{instr} = 3$ instructions per iteration are needed to generate the extrinsic information for $N_{sym} = 2$ symbols in DBTC mode, where a symbol is composed of $Bits_{sym} = 2$ bits. In SBTC mode, same number of instructions is required for $N_{sym} = 4$ symbols, where symbol is composed of $Bits_{sym} = 1$ bit. Considering $N_{iter} = 6$ iterations, the maximum throughput achieved is 164Mbps in both modes.

$$Throughput = \frac{N_{sym} * Bits_{sym} * F_{clk}}{N_{instr} * N_{iter}} \tag{3.12}$$

Memory sizes are dimensioned to support the maximum block size of the target standards (Table 1.1). This corresponds to the frame size of 6144 bits of 3GPP-LTE standard which results in a memory depth of $\frac{6144}{(N_{sym}=2)\times(N_{mb}=2)} = 1536$ words (for both input and extrinsic memories). Where N_{sym} is number of symbols per memory word and N_{mb} is number of memory blocks ($N_{mb} = 2$ as butterfly scheme is adopted). Table 3.12 presents the utilized memories in TurbASIP$_v$3. It has two single port input memories to store channel values LLR of size 16×1536 and 4 dual port extrinsic memories to save a priori information. Two banks **odd** 7×1536 and two **even** 14×1536. Each ASIP is further equipped with two 80×32 cross-metric memory which implement buffers to store β and α in left butterfly calculation phase and re-utilized in right butterfly phase. It also equipped with 48×80 Windowing memory to store β of last symbol of each window to be used later in next iteration to apply message passing for initialization.

Memory name	#	depth	Width	Type
Program memory	1	32	16	SP
Input memory	2	1536	16	SP
Extrinsic memory odd	2	1536	8	DP
Extrinsic memory even	2	1536	16	DP
Cross-metric memory	2	32	80	SP
Windowing memory	1	48	80	SP
Interleaving Stack memory	2	32	12	SP
Total Area = 0.218 mm^2				

Table 3.12: Typical memories configuration used for one ASIP decoder component (area with @65nm CMOS technology)

Figure 3.16 presents the BER comparison curves between software model (fix point) and FPGA for WiMAX block-size ”1728”, and 3gpp-LTE block-size ”864”.

Table 5.5 compares the obtained results of proposed work architecture with other related works.In order to facilitate the comparison between these works, we will normalize the occupied areas to @65nm technology. The Architecture Efficiency (AE), in bit/cycle/iteration/mm^2, is defined by the following expression:

$$AE = \frac{T \times I}{NA \times F} \tag{3.13}$$

where,

AE - Architecture Efficiency
F - Operational frequency,
NA - Normalized Area,
T - Throughput,
I - Number of turbo decoding iterations.

	TurbASIP$_{v3}$	[43]	[44]	[42]	[45]	[46]	[47]
Standard compliant	WiMAX, DVB-RCS, LTE	LTE, HSPA+	WiMAX, LTE	3GPP, LTE	WiMax, LTE	LTE	LTE
Tech (nm)	65	45	130	65	90	65	65
Core area (mm^2)	0.594	1.34	10.7	0.5	3.38	2.1	8.3
Normalized Core area @65nm (mm^2)	0.594	2.8	2.675	0.5	1.75	2.1	8.3
Throughput (Mbps)	164 @6iter	150 @8itr	187 @8iter	21 @6iter	186 @6iter	150 @6.5iter	1280 @6iter
Parallel MAPs	2	8	8	1	8	1	64
Normalized hardware efficiency (Mbps/mm^2)	264	54	70	42	106	71	154
Decoding speed (bit/cycle/iter.)	2	3.12	6	0.42	7.3	3.25	19.2
F_{clk} (MHz)	500	385	250	300	152	300	400
AE*	3.31	1.11	2.24	0.84	4.2	1.56	2.31

* in bit/cycle/iteration/mm^2

Table 3.13: TurbASIP$_{v3}$ performance comparison with state of the art implementations

Table 5.5 illustrates how the proposed implementation outperforms state of the art in terms of architecture efficiency with considering the flexibility. The presented ASIP in [42] supports wide range of 3gpp standards and convolutional code (CC). Although they reserved 6bits for channel input quantization, still the occupied area 0.5mm^2 is big when it is compared to the achieved throughput of 21Mbps. Thus, a low architecture efficiency of 0.84 is obtained. The other instruction-set pipeline processor (Xtensa) from Tensilica [43] supports single binary turbo code for all block-sizes (LTE, HSPA+). They achieve high throughput of 150Mbps but the occupied area of 1.34mm^2 at 45nm (or 2.8mm^2 at 65nm) is big. Thus, the obtained architecture efficiency, which is equal to 1.11, is low.

The parameterized dedicated architecture in [45] gives a good architecture efficiency result of 4.2. However, the design flexibility efficiency is low. They adapt vectorizable and contention-free parallel interleaver, so they support only 18 modes out of the 188 specified in LTE. Moreover the maximum throughput achieved is only for 9 modes otherwise for frame-sizes less than 360 bits has throughput less than 93Mbps. The parameterized architecture of [44] supports both turbo modes (DBTC and SBTC) and achieves a high throughput of 187Mbps. However, the occupied area is more than 4 times compared to our implementation and it achieves an architecture efficiency of 2.24.

Similarly the LTE-dedicated architecture proposed in [46] achieves high throughput of 150Mbps but at a high cost of almost 3 times the occupied area with architecture efficiency of 1.56.

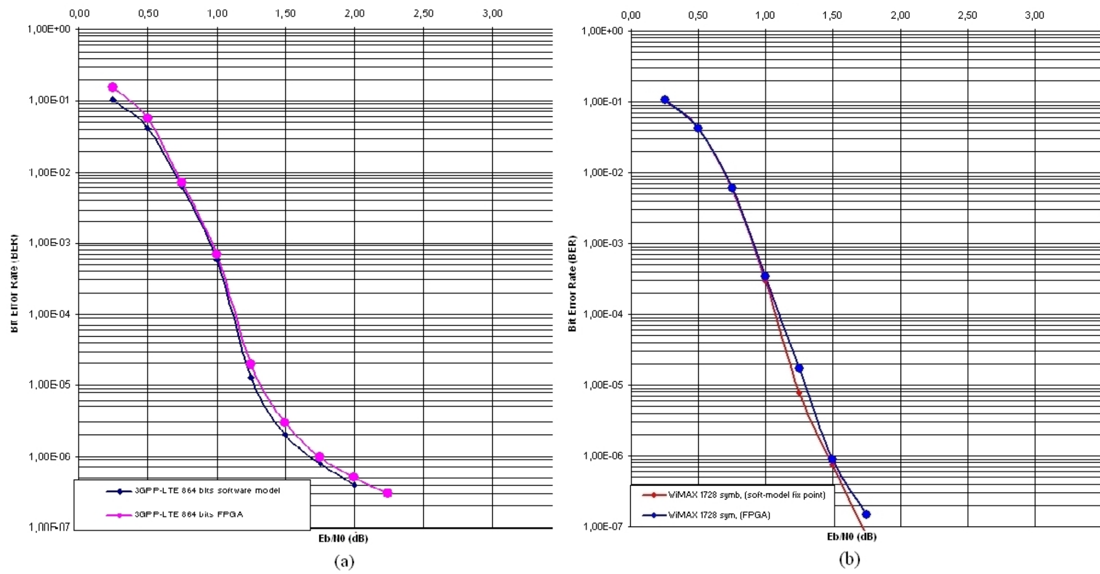

Figure 3.16: BER performance comparison obtained from the 2-ASIP FPGA prototype and the reference software model (fixed point) for (a) 3GPP-LTE block-size=864, (b) WiMAX, block-size=1728

Finally the proposed work in [47] achieves a very high throughput of ~ 7.5 times the proposed work with an area occupation of ~ 13 times the proposed work. The high throughput achieved by maintaining 64 Radix2 MAPs decoder working in parallel, benefiting from adopting a contention-free, (QPP) parallel interleaver. However in order to support WiMAX standard for [47], it should adapt Radix-4 that will be at high cost for area and critical path as described in its work.

3.6 Power Consumption Analysis for ASIC Implementation

In order to discover the impact for the pipeline optimization over energy efficiency, the power consumption analysis in ASIC implementation has been made. It is applied on the design before and after optimization. The power results then compared in order to study the enhancement over the energy efficiency. Intial ASIP and TurbASIP$_{v1}$ are considered for case study.

3.6.1 Power Consumption Analysis Flow

The power analysis is done on the case studies keeping the hierarchy levels as defined in the generated RTL model, in order to observe power consumption in each blocks at each level of hierarchy. As shown in Figure 3.17, the first step is the functional RTL model. This is then synthesized with Synopsys topographic design compiler, using low power technologies (65nm 1V 25Â°). As the interconnect parasitics have a major effect on path delays, the accurate estimates of resistance and capacitance are necessary to calculate path delays. Hence design compiler used, is in topographical mode. In this mode, design compiler leverages the physical implementation solution to derive the ”virtual layout” of the design so that the tool can accurately predict and use real net capacitances instead of wire-load model-based statistical net approximations. In addition, the capacitances are updated as synthesis progresses. That is, it considers the variation of net capacitances in the design by adjusting placement-derived net delays based on an updated

”virtual layout” at multiple points during synthesis. The accurate prediction of net capacitances drives design compiler to generate the gate level netlist. The netlist and sdf (standard delay format) file is then used for the post-synthesis simulations, in Modelsim. The post synthesis simulations generate the .vcd (Value Change Dump) file. The .vcd file is used to capture the signal activities during simulations.

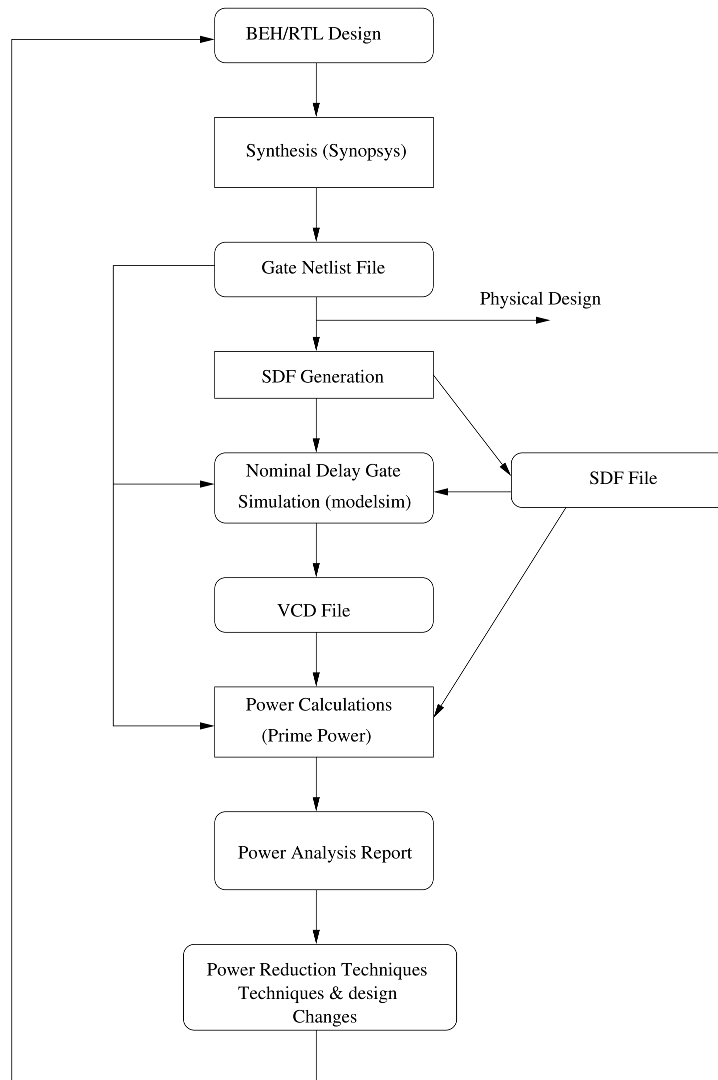

Figure 3.17: Power consumption analysis flow

For the power analysis, Prime-Power (Synopsys) tool is used. This tool uses a statistic and probability estimation algorithm to calculate the average power and construct the average cycle power waveform. For accurate analysis of power with respect to time, statistical activity based power analysis at the gate level is done. The .vcd, sdf files and the netlist are given as input for the power analysis tool. Using these input files this tool builds a detailed power profile of the design based on the circuit connectivity, the switching activity, the net capacitance and the cell-level power behavior data in the library. It then calculates the power behavior for a circuit at the cell level and reports the power consumption at the block level at all hierarchies.

3.6.2 Power Consumption Analysis Results

The summary of power analysis with three levels of hierarchy is shown in Table 3.14. Results confirm the dominant impact of memories on power consumption for turbo decoding, besides detailing the significant pipeline stages part.

Figure 3.18 shows the power consumption of pipeline stages for both ASIPs (Initial ASIP, TurbASIP$_{v1}$), where the consumption for all stages except EXE, MAX1 and MAX2 are almost the same for both versions. While power consumed by EXE is reduced 5% in TurbASIP$_{v1}$. While MAX1 and MAX2 stages are added to the TurbASIP$_{v1}$ architecture and not exist in the intial ASIP, so their consumption to power is kept to zero. MAX1 stage in TurbASIP$_{v1}$ is showing considerable power consumption, while MAX2 is consuming negligible amount of power. The increase of power consumption is due to adding these two stages.

Note: As mentioned in earlier in Section 3.2, Initial ASIP does not support windowing. All memories's sizes are considered to fit 128 bits block-size. To have fair comparison, memories for TurbASIP$_{v1}$ are considered with similar size to the intial ASIP.

In Table 3.14, the second column is showing the power consumed at the different levels of hierarchy by different blocks for both ASIPs. It can be clearly seen that the impact of the optimization on power consumption is only on the pipeline and register file, while the internal memories ConfigA and ConfigB has been removed in TurbASIP$_{v1}$ so their power consumption are zero, and there is no impact on power consumption of external memories. After this optimization, even though the overall power consumption is slightly decreased by 5.5%, at the same time the throughput has been improved by 77%. In order to have fair comparision with respect to throughput and power consumption, energy effeciency can be calculated as in 3.14:

$$EE = \frac{P}{T \times I} \tag{3.14}$$

Here,
EE - Energy Efficiency (nJ/bit/iteration),
P - Power,
T - Throughput,
I - Number of iteration,

The EE indicates how much energy a decoder chip consumes to process a hard bit at iteration. Eventually there is improvement in the energy per decoded bit by **45%**.

Section (hierarchy)			Power (mw)	
			Initial ASIP	TurbASIP$_{v1}$
external memory	input data	Top	2.79	2.79
		Bottom	2.79	2.79
	extrinsic	Top	16.5	16.5
		Bottom	16.5	16.5
	Interleaving	Top	2.78	2.78
		Bottom	2.78	2.78
	Alpha	Read	2.89	2.89
		Write	2.89	2.89
	Beta	Read	2.89	2.89
		Write	2.89	2.89
	Section (Total)		55.7	55.7
ASIP	Internal memory	program	3.43	3.43
		ConfigA	13.0	0
		ConfigB	13.0	0
		Cross Metric	13.0	13.0
		Total	42.43	16.43
	Register File		13.7	25.5
	Pipline		34.23	43.6
	Total		90.13	85.53
Total Power (mW)			145.83	141.23
Throughput (Mbps)@6itr			45	80
Energy per bit (nJ/bit/iteration)			0.49	0.27
Area (mm^2)			0.22	0.23

Table 3.14: Power consumption, throughput and area before and after optimizations

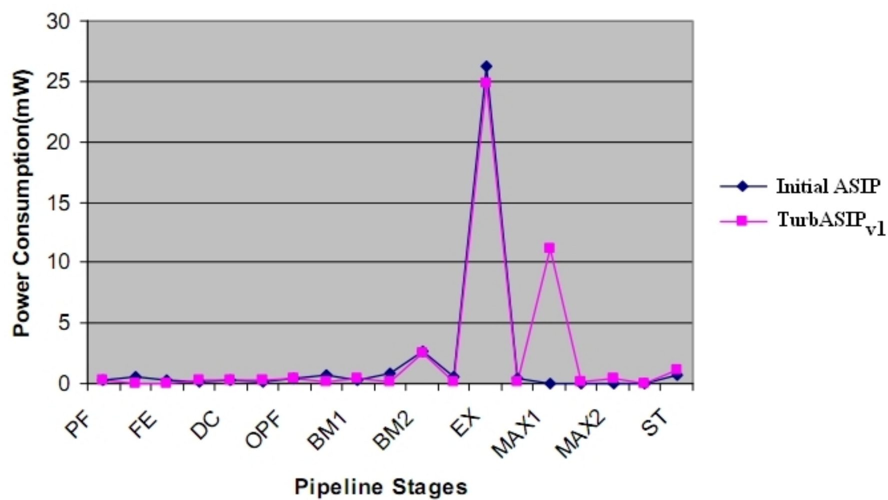

Figure 3.18: Power consumption related to each pipeline stage

3.6.3 Results Comparison

The Turbo decoder system architecture in our case study consists of 2 ASIPs interconnected. It adapts shuffled decoding where the two ASIPs operate in a 1×1 mode. One ASIP processes the

received frames in natural domain and the other in interleaved domain. As the power analysis results presented earlier are for single ASIP, hence for the comparison which will be conducted in this subsection, the power and area are taken into account for two ASIPs.

Table 3.15 is comparing the results of the proposed architecture with relevant existing implementations. We have considered the results of the initial ASIP, TurbASIP$_{v1}$. Comparisons are provided with the reconfigurable ASIP supporting Turbo and Viterbi decoding presented in [52] and with a unified instruction set programmable architecture proposed in [53]. It is necessary to compare energy efficiency results for each work after normalization. Because of each of the implementations is using different technology and clock frequency. If the fabrication technology changes, the voltage applied also varies. Along with it the silicon surface changes causing the variance in capacitance. Power is directly proportional to frequency, capacitance and voltage. With all these reasons there is change in power consumption and area occupied by the same design implemented in different technologies.

Ref	Algorithm	Area (mm^2)	Power (mW)	Tech (nm)	Clock (MHz)	Throughput (Mbps)	Energy Efficiency (nJ/Bit/iter)
The intial ASIP [40]	Turbo (Max-Log MAP)	0.43	292	65	500	45 @6-iter	1.09
[53]	Turbo (LTE)	0.23	230	45	333	100 @6-iter	1.01
[52]	CC, Turbo	0.42	100	65	400	34 @5-iter	0.59
TurbASIP$_{v1}$	Turbo (Max-Log MAP)	0.46	282.5	65	500	80 @6-iter	0.58

Table 3.15: TurbASIP$_{v1}$ energy efficiency results and comparisons

The Normalized Energy Efficiency (NEE) gives energy per decoded bit. Calculating NEE is similar to the formula (3.14) given earlier. However due to comparing implementations in different technologies, we have to multiply it by a factor as shown in formula (3.15):

$$NEE = (P/(T \times I)) \times NEF \tag{3.15}$$

Here,
NEE - Normalized Energy Efficiency,
P - Power,
T - Throughput,
I - Number of iterations,
NEF - Normalized Energy Factor [54]

Normalized Energy Factor (NEF) can be calculated as shown in the formula 3.16:

$$NEF = (\frac{V_{dd1}}{V_{dd2}})^2 \times (\frac{f_1}{f_2})^2 \tag{3.16}$$

Here,
f_1 - Feature size of the target technology for normalization (65nm),
f_2 - Feature size of the used technology,
V_{dd1} - Supply Voltage of the target technology for normalization (0.9 V in 65nm),
V_{dd2} - Supply Voltage of the used technology.

To scale the results from 45nm to 65nm technology, the normalized energy factor (NEF) is 2.64=(0.9 V / 0.8 V)^2×(65 nm / 45nm)^2. Also throughput is of great importance, as the energy is calculated for each bit. The initial ASIP is having highest power consumption, and hence the NEE is highest 1.09 (nJ/Bit/iter). Similar result is achieved in [53] with an NEE = 1.01(nJ/Bit/iter). On the other hand it can be noticed that TurbASIP$_{v1}$ and the work in [52] present the best energy efficiency with NEE = 0.58 and 0.59 (nJ/Bit/iter) respectively.

3.7 Summary

In this chapter we have presented the proposed optimized ASIP-based flexible Turbo decoder supporting all communication modes of 3GPP-LTE, WiMAX and DVB-RCS standards. This main contribution allows further to illustrate how the architecture efficiency of instruction-set based processors can be considerably improved by minimizing the pipeline idle time. Three levels of optimization techniques (Architecture, Algorithmic, and Memory) has been proposed and integrated. Furthermore, the chapter has presented low complexity interleaver design supporting QPP and ARP interleaving in butterfly scheme together with an efficient parallel memory access management. Results show that the ASIP pipeline usage percentage have been maximized to reach 94%, achieving a throughput of 164Mbps with 0.594mm^2 @65nm CMOS technology and an architecture efficiency of 3.31 bit/cycle/iteration/mm^2. Furthermore, the impact of the proposed optimization techniques on power consumption is illustrated. The overall gain in normalized energy efficiency is around ≈45.6% compared to the initial architecture. The achieved architecture efficiency and energy efficiency, considering the supported flexibility, compare favorably with related state of the art implementations

CHAPTER 4

FPGA Prototype for Multi-standard Turbo Decoding

ON board validation is a crucial step to fully validate and demonstrate the effectiveness of any proposed novel hardware architecture. Two additional benefits can be mentioned in this context: valuable feedback to the architecture design particularly regarding system-level interfacing of the Turbo decoder, and the obtained hardware prototype can be used as a rapid simulation environment for digital communication application; e.g. to explore various system parameter associations.

It is, however, a complex task in the context of ASIP-based implementations and flexible channel decoders as a fully flexible environment should be designed. Hence, this chapter is dedicated to the presentation of the FPGA prototype for the proposed multi-standard Turbo decoder. The first section presents the adopted ASIP-based LISA to FPGA prototyping flow, while the second section details the proposed prototyping environment. This includes: (1) the GUI (Graphical User Interface) in order to configure the platform with desired parameters such as targeted standard, frame size, code rate, and number of iterations from a host computer, (2) the multi-standard flexible transmitter, (3) the hardware channel emulator, (4) the proposed multi-standard Turbo decoder, (5) the error counter, and (6) and the overall scheduling control. The third and last section presents and compares the obtained FPGA synthesis results.

4.1 ASIP Design, Validation and Prototyping Flow

While selecting ASIP as the implementation approach [26], an ASIP design flow integrating hardware generation and corresponding software development tools (assembler, linker, debugger, etc.) is mandatory. In our work we have used Processor Designer framework from Synopsys (ex. Coware Inc.) which enables the designer to describe the ASIP at LISA [38] abstraction level and automates the generation of RTL model along with software development tools. ASIP design, validation and prototyping flow has been divided into 3 levels of abstraction as shown in Figure (4.1) and detailed in the following subsections.

4.1.1 LISA Abstraction Level

The first step toward the Application Specific Instruction-set Processor (ASIP) implementation is to employs centralized Architecture Description Language (ADL) processor models, from which software tools, such as C compiler, assembler, linker, and instruction-set simulator, can be automatically generated and the application program writing assembly file (.asm file) to be executed on the ASIP. To simulate the input data memories, their contents are generated from the software reference model and written in a special sections (.section) in the assembly file as defined in the linker command file. Processor Designer framework generates assembler, linker, processor debugger and simulator tools. Assembler and linker process the application program (.asm file) to generate the executable file (.out file) which is used in Processor Debugger to verify both the ASIP model and the application program. Once the ASIP is verified, a special utility ”lvcdgen” is used to generate reference Value Change Dump VCD files from a LISA simulation. The utility can load any shared object of a model, load an application, and run the simulation. During simulation, a VCD file for all global resources of the model is written (global register resources and all pipeline registers, as well as input and output pins). The generated VCD file can be used at lower abstraction levels for verification purpose. In addition, include files for HDL simulators can be generated. These include files can then directly be used to create VCD files in an HDL simulation. The complete flow is shown in Fig (4.1.a).

4.1.2 HDL Abstraction Level

Processor Designer framework provides the Processor Generator tool which manipulate the generated structure of the RTL model by grouping operations into functional units. Each functional unit in the LISA model represents an entity or a module in the HDL (VHDL/Verilog) model. Each operation inside a functional unit is mapped to a single process inside this entity or module. By default, Processor Generator creates a single functional unit in each pipeline stage and assigns all operations in that pipeline stage. The efficiency of the generated HDL depends upon the LISA modeling and the configuration options of Processor Generator. Modeling in LISA which use C compiler and require high-level architecture information, so it is highly recommended that LISA modeling should be as close as possible to HDL e.g if in one pipeline stage we want resource sharing, that resource should be declared once. Otherwise, due to inability to detect sharing, resources will be duplicated in HDL. Other issue is the use of high level operators of LISA which may not be produced by the Processor Generator e.g modulo two operations (”variable % 2” in LISA) should be rather implemented by the LSB manipuation of the considered variable. For memory interface generation, different Memory Interface Definition Files (MIDF) are provided which define the number of ports and latencies. Once memory layout file and executable application program file is available, ”exe2bin” utility inputs them to generate

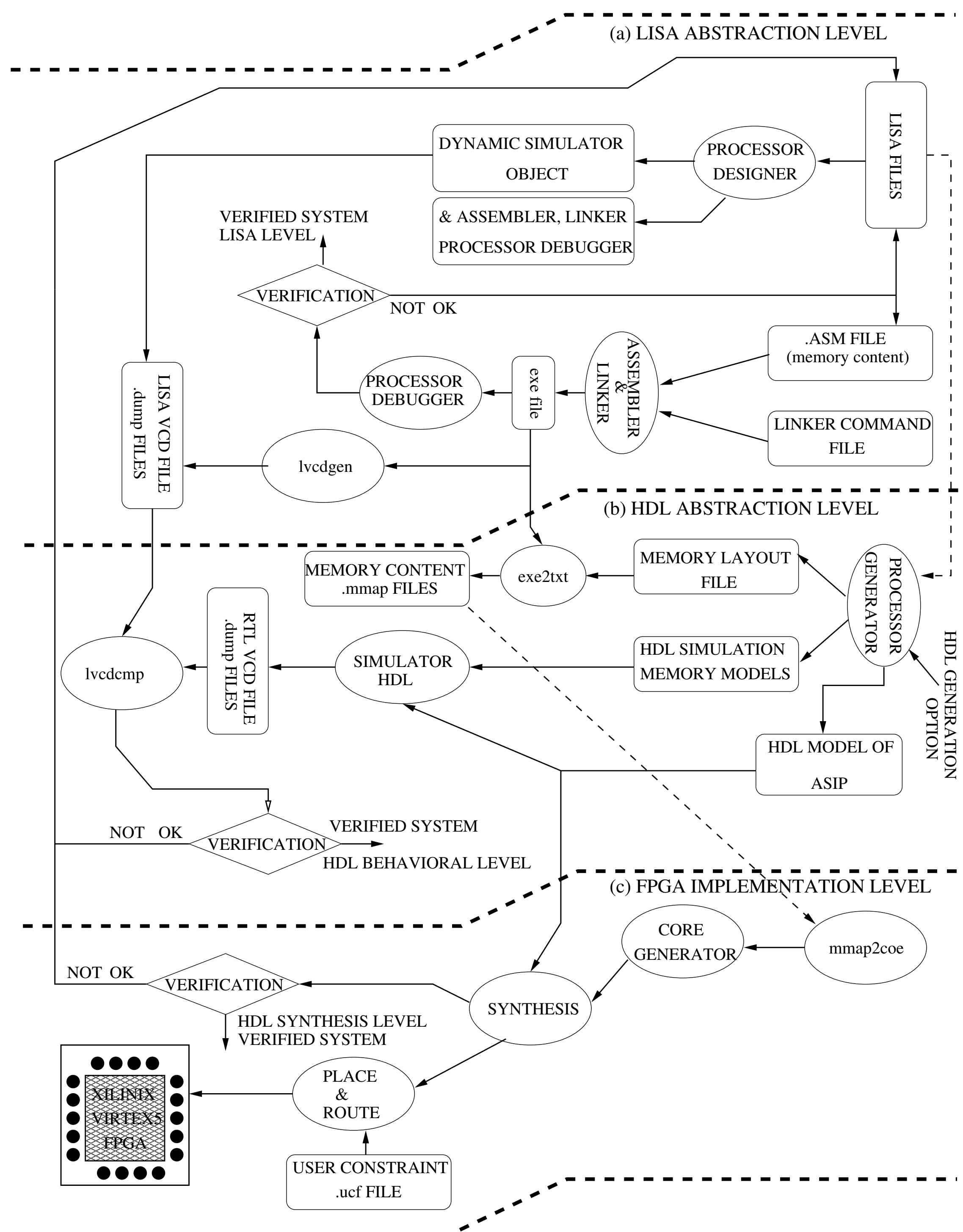

Figure 4.1: Prototyping Flow: (a) LISA abstraction level, (b) HDL abstraction level, (c) FPGA implementation level

the contents of memories in separate .mmap files. With these three inputs (VHDL model, memory model and .mmap files), the VHDL model can be simulated behaviorally using an HDL simulator, e.g ModelSim by Mentor Graphics. To run HDL simulation, LISATEK Processor Generator produces ready-to-use Makefile which can be executed to see either the waveforms or to generate VCD file. To verify the generated ASIP HDL model, the VCD file generated through HDL model and the one generated through LISA model (in previous subsection) can be

compared using ”lvcdcmp” utility, where ”lcvdcmd” is a utility used to compare two VCD files with each other.

4.1.3 FPGA Implementation Level

At this level, the only missing elements are the synthesizable memory models. Depending upon the target FPGA device and the synthesis tool, the declaration of the memory models for simulation can be replaced by equivalent declaration of synthesizable memories. The obtained model containing the ASIP and its memories can be used for synthesis, placement and routing to verify timing and area performances.

However, for a full validation of the ASIP features and the different supported parameters, a complete system environment is required. The following section will present the proposed complete system prototyping platform.

4.2 Proposed Prototyping Platform for the ASIP-based Turbo Decoder

On board validation is a crucial step to validate the ASIP approach feasibility. It is, however, a complex task in the context of flexible channel decoders as a fully flexible environment should be designed.

On the host computer side, the proposed environment integrates a GUI (Graphical User Interface) in order to configure the platform with desired parameters such as targeted standard, frame size, code rate, and number of iterations from a host computer. This GUI also displays results such as BER (Bit Error Rate) and FER (Frame Error Rate) performance curves, besides the achieved throughput on FPGA.

On the FPGA board side, several modules are required: a source of data, a flexible turbo encoder, a channel model, the turbo decoder architecture which includes 2 TurbASIPs, and an error counter. For data source, a pseudo random generator based on a 32 bit LFSR (Linear Feedback Shift Register) is used. The turbo encoder implements both modes of encoding of SBTC and DBTC specified in the supported standards, and can be reconfigured dynamically. The channel model emulates an AWGN (Additive White Gaussian Noise) model. The decoder is made up of two TurbASIPs working in shuffled mode, an interleaving address generator, an input interface (to fill the input LLR memories), and an extrinsic exchange module to manage the exchanges of extrinsic information considering the interleaving rules and all the memories of the ASIPs (input, extrinsic, interleaving, cross metric and program memories).

This choice of having emmitter, channel, decoder, error counter on the FPGA and using the GUI only for configuration and monitoring enbales high throughput.... the platform can then be used as a rapid simulation environement in the research in turbo decoding.... etc....

The FPGA board used for this platform prototype is the DN9000k10pci board from the DiniGroup company which integrates 6 Xilinx Virtex 5 XC5VLX330. Only one Virtex 5 has been used for the proposed prototype.

Figure 4.2 gives an overview of the complete proposed and designed environment for multi-standard turbo decoding. The following subsections describe the different components of the proposed platform.

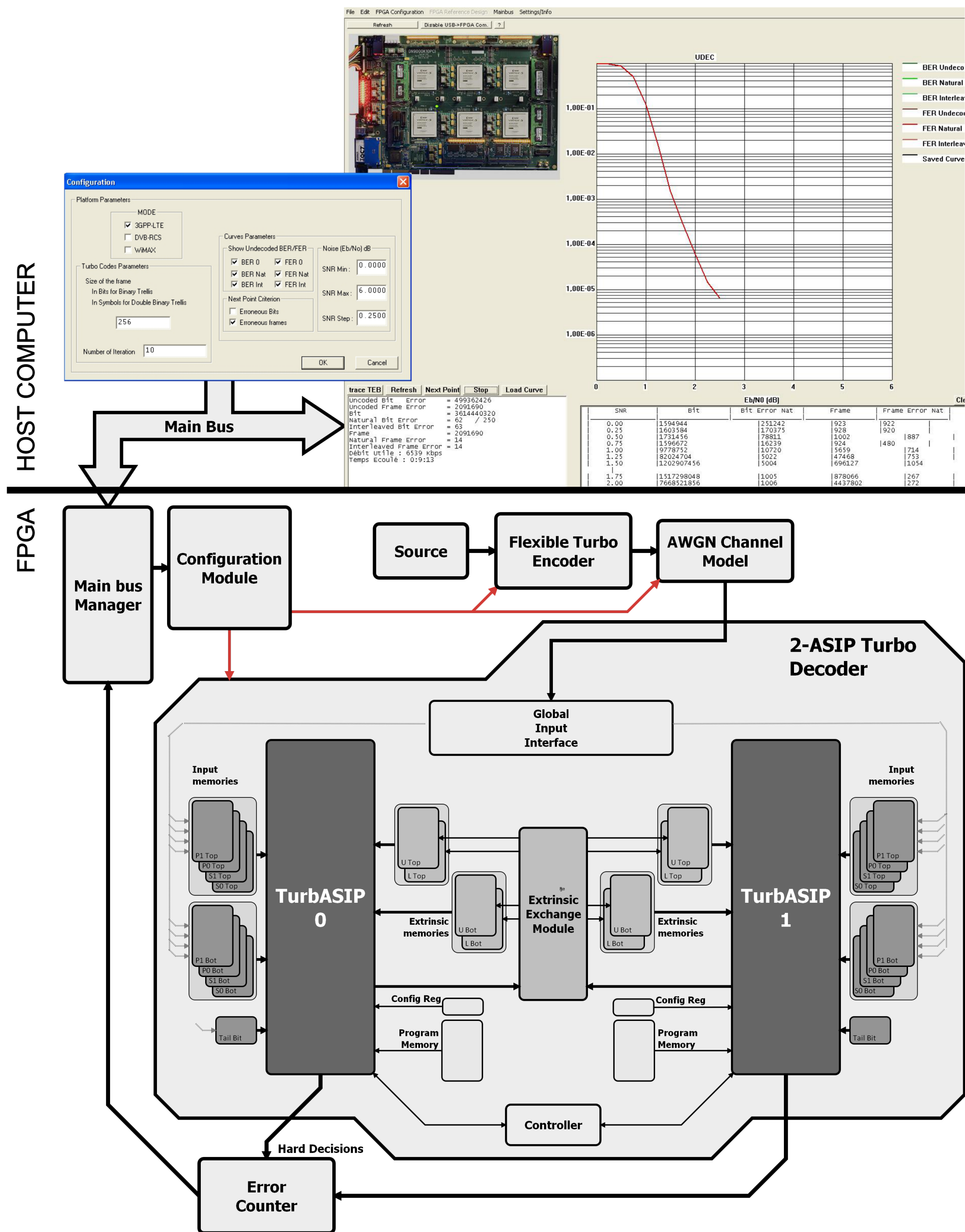

Figure 4.2: Overview of the proposed environment for multi-standard turbo decoding

4.2.1 Multi-Standard Transmitter

The transmitter consists of a data source and a flexible turbo encoder. The data source is emulated by a pseudo random generator based on a 32 bits LFSR architecture. The LFSR can produce 2 bits at each clock cycle, and thus, it is appropriate for binary and double binary modes.

The flexible turbo encoder is presented in Figure 4.3. This encoder supports single and

double binary trellis, without duplicating the hardware resources. It also supports all frame size for 3GPP-LTE, WiMAX and DVB-RCS standards. For that, the convolutional encoder and the interleaver are flexible.

The convolutional encoder is presented in Figure (4.4). The proposed architecture is fully configurable, with dynamically parameterizable connections enabling all possibilities regarding parity and recursion equations. So the encoder can execute in two modes, either as an SBTC encoder (Figure 1.7, Chapter I) or as a DBTC encoder (Figure 1.10, Chapter I).

The data coming from the source are stored in a dual read port RAM memory. One read port for the natural domain convolutional encoder, the other for the interleaved one. When the source completes the generation of the whole frame of data, the memory is read to supply at the same time the natural and interleaved encoders.

The natural read address is generated by a simple counter. On the other hand, the interleaved address is generated on the fly recursively using a flexible interleaving address generator. For SBTC (3GPP-LTE), QPP (Quadratic Polynomial Permutation) interleaving address generator based on the work presented in Chapter 3 (see Subsection 3.2.3.3, Page (61)) has been designed. Similarly, for DBTC (WiMAX and DVB-RCS), an ARP (Almost Regular Permutation) interleaving address generator.

For SBTC mode, only the outputs $S1_i$, $P1_i$, $P'1_i$ and $S'1_i$ are used. $S'1_i$ is used to transmit the interleaved systematic during tail bit termination. $S'1_i$ and $S'2_i$ are not sent to the decoder to be decoded, but are used by the verification module in order to compute BER/FER for the interleaved domain.

The trellis of double binary trellis does not need tail bit termination since the convolutional encoders are circular. A first encoding with the encoder state equals to 0 is done in order to get the final state of the encoder. With this final state, using a Look-Up-Table defined in [7], the circular state to initialize the encoder for a second encoding of the same data is found. For this second encoding phase, the states at the beginning and at the end are identical.

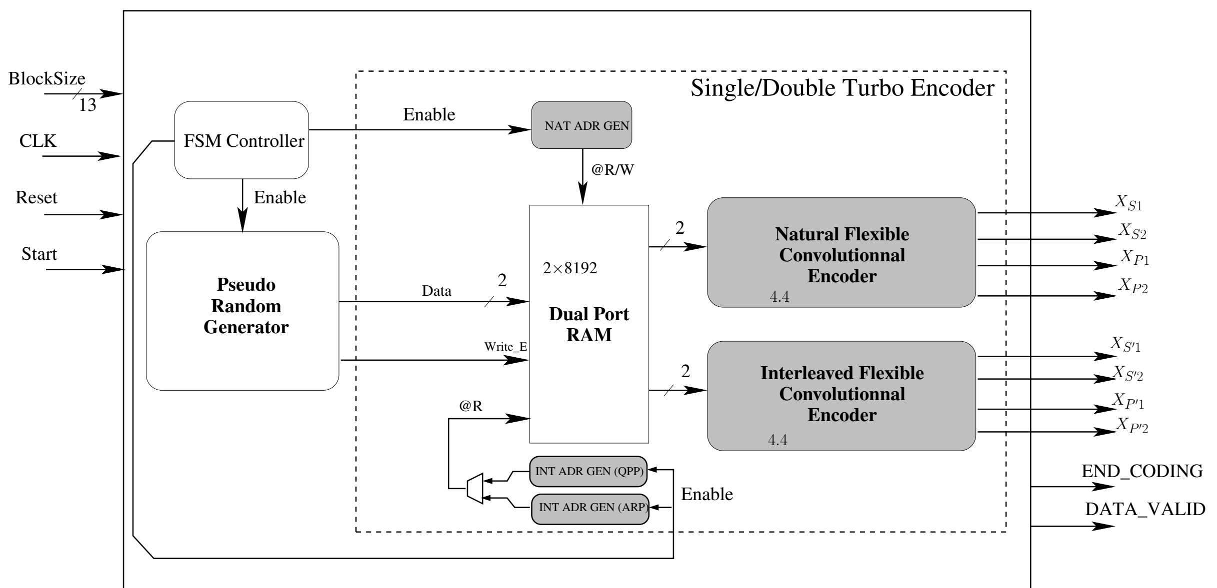

Figure 4.3: Turbo encoder with random source generator

The FPGA synthesis results of this multi-standard transmitter are shown in Table 4.1.

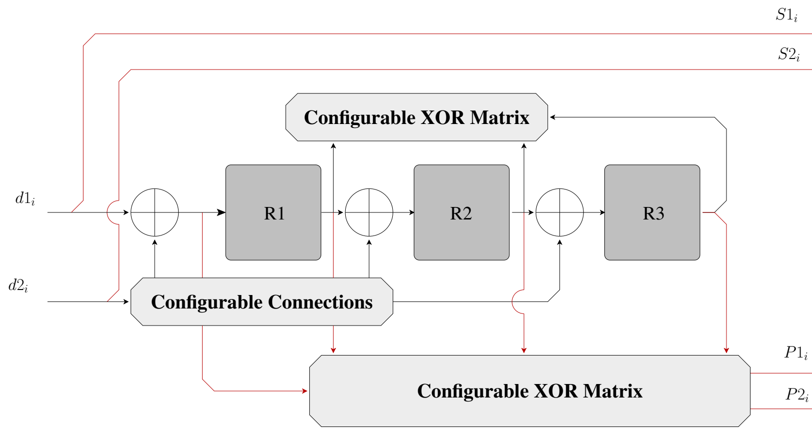

Figure 4.4: Flexible convolutional encoder

FPGA Synthesis Results (Xilinx Virtex5 xc5vlx330)	
Slice Registers	140 out of 207,360 (< 1%)
Slice LUTs	970 out of 207,360 (< 1%)
DSP48Es	0 out of 192
Frequency	119.490MHz

Table 4.1: FPGA synthesis results of the Transmitter Module

4.2.2 Channel Module

The AWGN channel model is considered. To emulate the AWGN channel, a hardware model based on Wallace method was available at the Electronics Department of Telecom Bretagne which is used in this prototype. Figure 4.5 presents the input/output interface of the integrated AWGN channel emulator. The power of noise is represented in 17 bits. The formula to compute

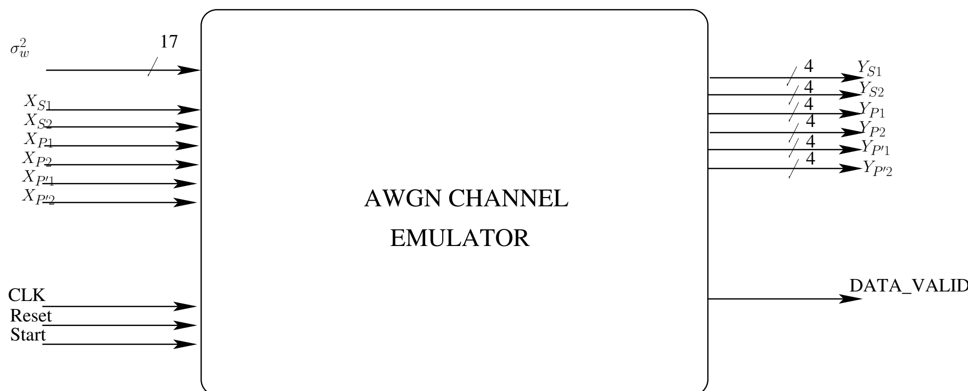

Figure 4.5: Input/output interface of the integrated AWGN channel emulator

the value to apply at the input σ^2 port of the channel module for a target SNR is as follow:

$$\sigma = \sqrt{\frac{10^{\frac{-SNR}{10}}}{2 \times log_2(m) \times R}} \times 2^{16} \tag{4.1}$$

i.e. for code rate R = 1/3, number of symbols in the modulation m=2 (BPSK), and targeting an SNR of 0.25dB, the binary value of σ^2 to apply at the channel module input port is σ^2 =

10011000010100011. The synthesis results of this channel emulator are tabulated in Table 4.2.

FPGA Synthesis Results(Xilinx Virtex5 xc5vlx330)	
Slice Registers	2577 out of 207,360 1%
Slice LUTs	3629 out of 207,360 1%
DSP48Es	30 out of 192 15%
Frequency	188.918MHz

Table 4.2: FPGA synthesis results of the AWGN channel module

4.2.3 Multi-Standard Turbo Decoder

The multi-standard turbo decoder is built around two TurbASIP which can be configured to execute in shuffled or serial modes. One ASIP processes the input frame in the natural order while the other processes it in the interleaved order. In shuffled mode, the two ASIPs execute simultaneously and the extrinsic information are updated in real time (as soon as generated).

In the following are presented the different modules developed in order to build this flexible turbo decoder.

4.2.3.1 Memory organization

Figure (4.6) recalls the memory organization of TurbASIP. TurbASIP has three input memory ports, one for upper halves of processed windows (Top), one for lower halves (Bottom), and one for Tail bit termination.

In order to simplify filling the systematic and parity LLRs, those memories have been split in four distinct memory banks. In fact, the input LLRs which should be stored in one memory word may not arrive at the same time (because of memory sharing between SBTC and DBTC, besides interleaving). However, the output of the four memory banks are concatenated and read by the ASIP as one single word. For the same reason, each Top and Bottom extrinsic memory is split in two banks. However, the cross metric memories do not need to be split.

All the memories in the turbo decoder are synchronous on read and write accesses. They are mapped on Xilinx dedicated block RAMs, allowing better timing and area occupation.

4.2.3.2 Interleaver module

The interleaving module is in charge of generating interleaving addresses. Those interleaving addresses are used by the input interface module to store natural systematic LLRs coming from the channel to the input memories of the interleaved domain.

The interleaving module is made of two main components. The first one generates the interleaving addresses according to the selected standard. However, the generated addresses from the interleaving module or from the TurbASIP interleaving generator are in absolute format, i.e. taking values in the range of the frame size. Hence, they do not correspond directly to locations in the multiple banks of top/bottom memories. Figure 4.7 illustrates this issue due to the adoption of windowing technique in butterfly scheme.

Thus, the second component of the interleaving module transforms the global addresses into local addresses that identify a location in those memories: which memory is targeted (Top

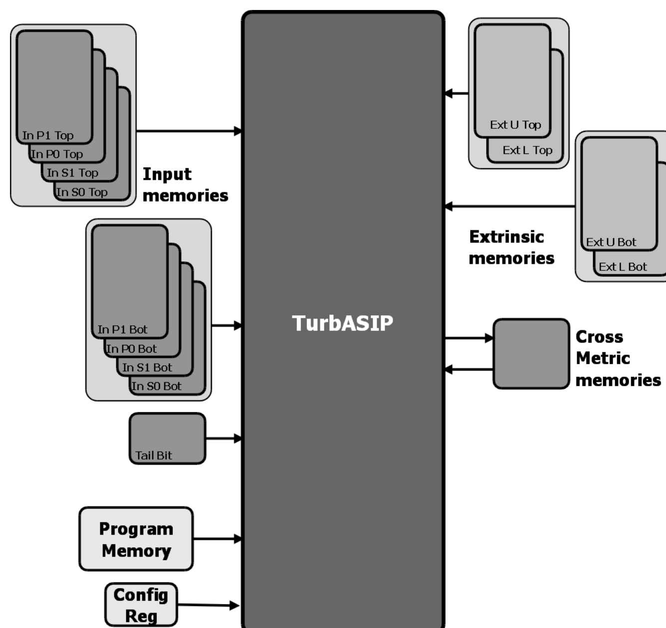

Figure 4.6: TurbASIP and attached memories

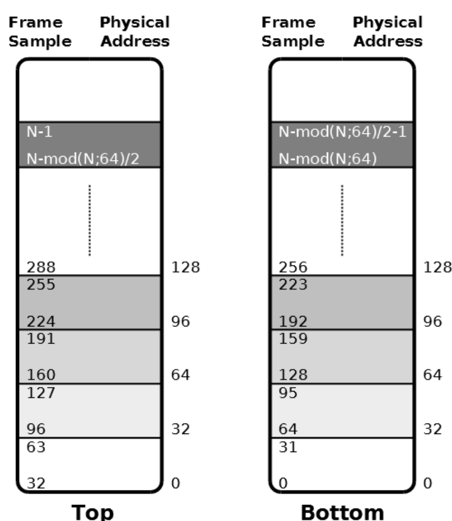

Figure 4.7: Internal data organization in input memories

or Bottom), and at which physical/local address. In fact, the window size of TurbASIP is 64 symbols, so 32 symbols are stored in the Bottom memories and 32 in the Top ones. The frame size may not be dividable by 64, in this case, the last window is less than 64 and starts at the address $N - (N \bmod 64)$, where N refers to the frame size in couples of bits. The LLRs of this last window are equally distributed between top and bottom memories ($(N \bmod 64)/2$ LLRs in each).

To determine in which memory (Top or Bottom) the LLR should go, the bit in the fifth position from LSB of the global address is used ($2^5 = 32$). When this bit is 0, the data should go to the Bottom memory, else to the Top one. There is an exception for the upper part of the last window: when the global address is above the limit $N - (N \bmod 64) + (N \bmod 64)/2$, the data is automatically written in the Top memory. This last window limit is computed before interleaving address generation.

To determine the physical address, this fifth bit is removed, and the MSB part of the address above this fifth bit is right shifted of one position. There is the same exception for the address as for the memory selection. The physical address for the last window is computed a bit differently. Before removing the fifth bit, $(N \bmod 64)/2$ is removed from the global address.

The following example illustrates the above proposed address conversion scheme. Assuming N=240 and a global address GA equals to 182(10) = 10110110. The starting address of the last window is $N - N \bmod 64 = 192$. So the corresponding LLR is not in the last window. GA(5) = 1, so the data goes to the Top memory. If the fifth bit is removed, and the upper part is right shifted, the resulting physical address is 10(2) concatenated with 10110(2) so it is 1010110(2) = 86. So in this example, the LLR corresponding to the global address 182 with a block size equals to 240 will be stored in the Top memory at the address 86.

4.2.3.3 Prototyped extrinsic exchange module

This module is in charge of extrinsic message transfer from a TurbASIP of one domain to the extrinsic memory from the other domain according to the interleaving rules.

TurbASIP builds a 128 bits word containing the extrinsic messages and their respective interleaving addresses. These interleaving addresses are not the global addresses but the local ones: containing one bit to determine Top or Bottom memory destination and the physical address. Because of the Radix-4 compression method applied in SBTC mode, extrinsic Top and Bottom memories are each split in odd and even memories. So the extrinsic exchange module is able to route extrinsic messages toward four distinct memories. Its architecture, illustrated in Figure 4.8, should avoid address conflicts.

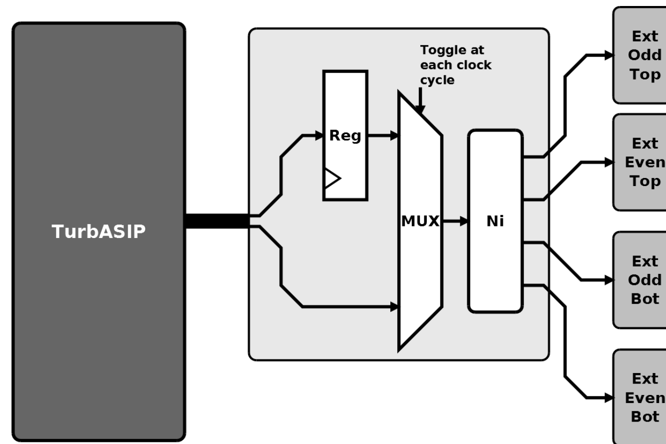

Figure 4.8: Architecture of the prototyped Extrinsic Exchange Module

In DBTC mode, the word contains two extrinsic messages corresponding to two different symbols (Butterfly scheme). Every two clock cycles, a word is generated. Hence, during the first clock cycle one message is routed to the appropriate memory, the other is buffered. During the second clock cycle, the buffered message is routed (time multiplexing).

In SBTC mode, every two clock cycles, the ASIP generates a word containing four extrinsic messages (Butterfly scheme and Radix-4). Similar to DBTC mode, two messages are buffered while the two others are routed. One symbol in Radix-4 compression is composed of two consecutive bits, thus two consecutive addresses. The interleaving rules of the 3GPP-LTE are such that an even address in a domain will correspond to an odd address in the other domain. So

by choosing to route simultaneously two messages of the same Radix-4 symbol, no addressing conflict occurs.

4.2.3.4 Area of the decoder core

The FPGA and ASIC synthesis results of 2-ASIPs and Extrinsic Exchange Module are shown in Table 4.3.

ASIC Synthesis Results (Synopsis Design Compiler)	
Technology	ST 65nm
Conditions	Worst Case (0.90V; 105^oC)
Area	$0.0845mm^2$ (53.3 K Gate)
Frequency	500 MHz
FPGA Synthesis Results(Xilinx Virtex5 xc5vlx330)	
Slice Registers	14,580 out of 207,360 7%
Slice LUTs	17,558 out of 207,360 8%
DSP48Es	0 out of 192
Frequency	96.379MHz

Table 4.3: FPGA synthesis results of the Receiver module

4.2.4 Error Counter

The verification module is used to compute and store the number of transmitted data source bit-s/frames, and the number of erroneous bits/frames at the output of the turbo decoder (in natural and interleaved domains). Those values are used to compute BER and FER which are read by the external host computer. Figure (4.9) presents an overview of the proposed architecture of the verification module, while the synthesis results are presented in Table 4.4.

FPGA Synthesis Results(Xilinx Virtex5 xc5vlx330)	
Slice Registers	34,924 out of 207,360 17%
Slice LUTs	14,468 out of 207,360 7%
DSP48Es	0 out of 192
Frequency	188.918MHz

Table 4.4: FPGA synthesis results of the Verification module

4.2.5 Scheduling and Control

Figure 4.10 illustrates the proposed scheduling of the different platform tasks from clock synchronization to error counting. The clock synchronization, configuration, last window limit computation and interleaving memory filling occur only once at the beginning.

During the clock synchronization, the whole platform is in reset state. This synchronization is done by a DCM of the FPGA (Digital Clock Management). The DCM is in charge of generating the platform clock from two external differential clocks from an oscillator.

The last window limit computation is done following the expression given in Subsection 4.2.3.2.

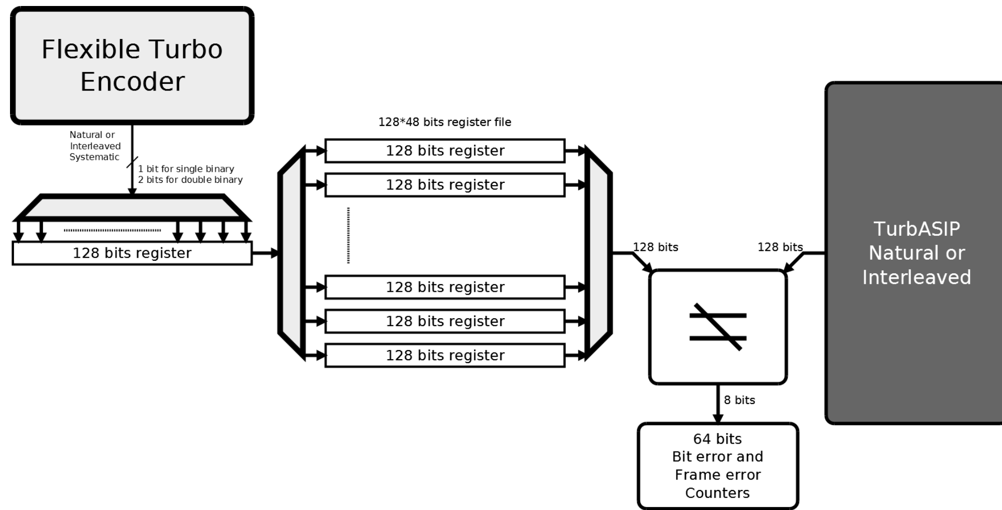

Figure 4.9: Verification Module

Then follow the encoding, the addition of white Gaussian noise, and the decoder input memories filling. Depending on the configuration, the decoding will be in serial mode or shuffled mode. When all the iterations are performed and the hard decisions are produced, the error counter provides the accumulated number of erroneous bits together with the number of processed frames to the GUI. The GUI performs BER/FER computations and displays the error performance curves.

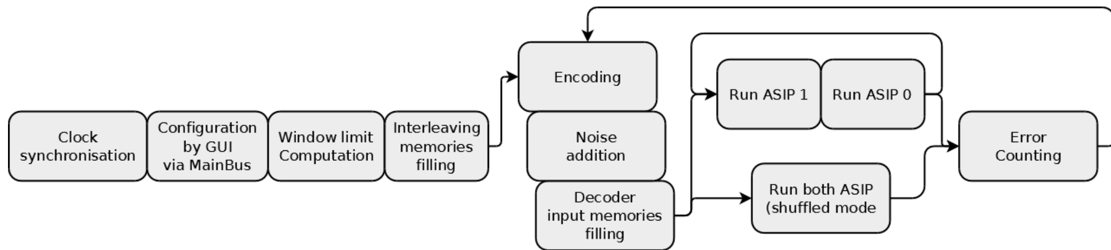

Figure 4.10: Proposed scheduling of the complete Turbo decoder platform

The main controller of the platform is integrated in the turbo decoder module. This controller is implemented by a mealy finite state machine. The different states correspond to the different tasks presented in figure (4.10).

4.2.6 Performance Results

The complete FPGA synthesis results of the turbo decoding system is presented in Table 4.5. The acquired Frame Error Rate performance for WiMAX 240-Byte source data and DVB-RCS 108-Byte source transmitted at $r = 0.33$ and modulated on BPSK are shown in Figure 4.11 and Figure 4.12 respectively. They have been verified that they match the performance of the reference software model.

FPGA Synthesis Results(Xilinx Virtex5 xc5vlx330)	
Slice Registers	53,276 out of 207,360 25%
Slice LUTs	37,477 out of 207,360 21%
DSP48Es	30 out of 192 15%
Frequency	63.041MHz

Table 4.5: FPGA synthesis results of the 2-ASIP Turbo decoder platform prototype

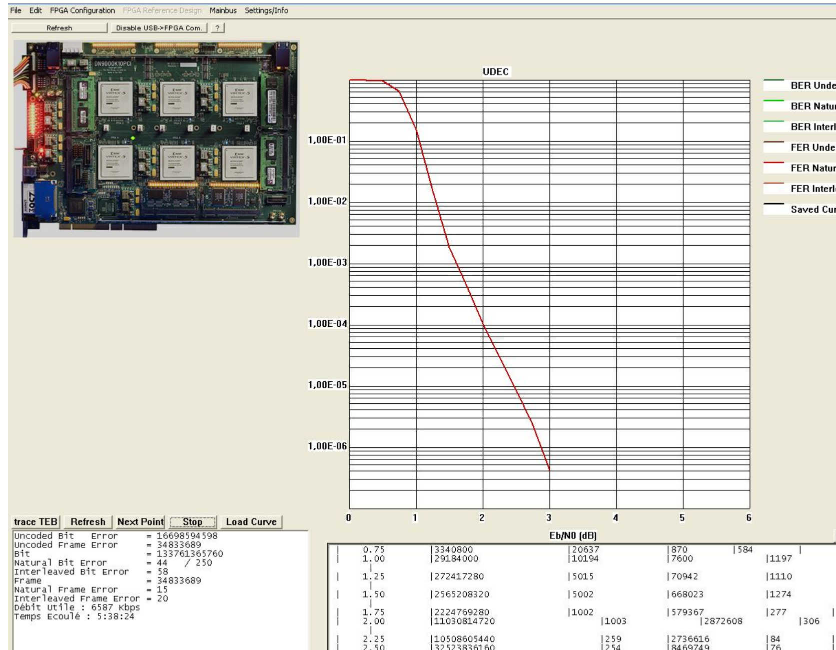

Figure 4.11: FER performance obtained from the 2-ASIP Turbo decoder FPGA prototype for WiMAX, block-size=1920 @ 7iteration

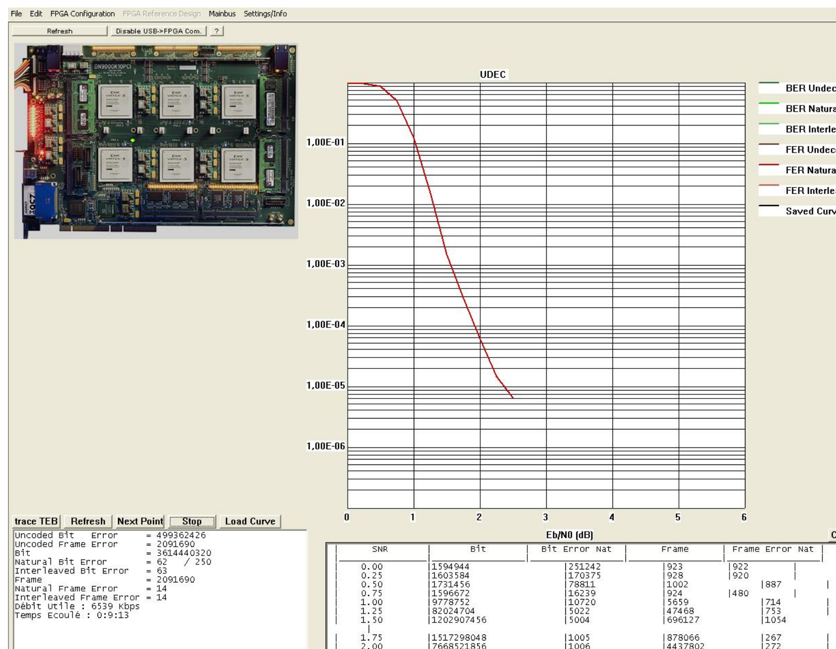

Figure 4.12: FER performance obtained from the 2-ASIP Turbo decoder FPGA prototype for DVB-RCS, block-size=864 @ 7iteration

4.3 Summary

In this chapter, we have presented the proposed and designed FPGA prototyping environment in order to fully validate the 2-ASIP multi-standard Turbo decoder described in the previous chapter. The proposed prototype consists of a complete flexible environment integration a GUI, a multi-standard flexible transmitter, a hardware channel emulator and the proposed multi-standard Turbo decoder. It consists of one of the first available and fully functional FPGA prototypes using the ASIP approach and efficiently exploiting the available parallelism levels in multi-standard Turbo decoding. Furthermore, the proposed environment integrates an efficient and high-speed error rate performance evaluation allowing for on-the-fly BER/FER curves display on the host computer. This flexible and high-throughput FPGA prototype can be further used as a fast simulation environment allowing fast performance evaluation of various combinations of supported parameters.

CHAPTER 5 Towards the Support of LDPC Decoding

BESIDES turbo codes, wireless digital communication standards specify a large variety of channel coding techniques and parameters. Some of them are mandatory, others are optional, where each one is suitable for specific application/communication mode. In this context, LDPC codes are often proposed in these standards (e.g. WiMAX, WiFi, DVB-S2/T2) due to their high error correction performance and simplicity of their iterative decoding. Mainly, the class of structured quasi-cyclic LDPC codes (QC-LDPC) is adopted as it presents very interesting implementation properties in terms of parallelism, interconnection, memory and scalability.

This chapter presents our contribution towards increasing the flexibility of the designed Turbo decoder to support LDPC decoding. It consists of a joint effort with another PhD student at the Electronics department of Telecom Bretagne: Purushotham Murugappa.

The main result concerns the proposal and the design of a multi-ASIP architecture for channel decoding supporting binary/duo-binary turbo codes and LDPC codes. The proposed architecture achieves a fair compromise in area and throughput to support LDPC and turbo codes for an array of standards (LTE, WiMAX, WiFi, DVB-RCS) through efficient sharing of memories and network resources. The Min-Sum algorithm The chapter starts with a brief introduction on LDPC codes with emphasis on the particular class of structured quasi-cyclic LDPC codes and adopted parameters in the considered wireless standards. Then LDPC iterative decoding is introduced and the low complexity reference decoding algorithms are presented. Subsequent sections present the proposed LDPC/Turbo decoding architecture in a top-down approach. First presenting the functional description of the overall architecture and the main architectural choices and design motivations. Following, the proposed memory organization and sharing is presented. Then, the proposed scheduling, pipeline structure, and instruction set in both Turbo and LDPC decoding mode are presented. The last section presents logic synthesis results along with future perspectives.

5.1 LDPC codes

Low-density parity-check (LDPC) codes is a class of linear block codes. Linear block codes have the property of linearity, and they are applied to the source bits in blocks, hence the name linear block codes. The LDPC name comes from the characteristic of their parity-check matrix which contains only a few 1's in comparison to the amount of 0's. Their main advantage is that they provide a performance which is very close to the capacity for a lot of different channels and linear time complex algorithms for decoding. Furthermore are they suited for implementations that make heavy use of parallelism. They were first introduced by Gallager in his PhD thesis in 1960 [55], but they were ignored until 1998. They were rediscovered by MacKay(1999) and Richardson/Urbanke(1998) as an alternative to the capacity achieving codes namely Turbo Codes. Its applied in many applications such as Digital Video Broadcasting Standard (DVB-S2), satellite transmission of digital television WLAN, WiMAX, WiFi ...etc.

5.1.1 Representations of LDPC Codes

Basically there are two different possibilities to represent LDPC codes:

1. Matrix Representation A binary LDPC code is represented by a sparse parity check matrix *H* with dimensions $M \times N$ such that each element h_{mn} is either 0 or 1. N is the length of the codeword, and M is the number of parity bits. Each matrix row *H*(i,$(1 \leq j \leq N)$) introduces one parity check constraint on the input data vector x =$\{x_1, x_2, ... , x_N\}$:

$$H_i.x^T = 0 \tag{5.1}$$

In the following example for Hamming (7,4) code, which is another class of linear block codes and has similar parity check matrix *H* structure to LDPC. It encodes a codeword of 4 data bits into 7 bits by adding three parity bits. A generator matrix **G** is a basis for generating all its possible codewords.

$$C(\text{linear code}) = \begin{vmatrix} 1 & 0 & 1 & 1 \end{vmatrix}$$

$$G = \begin{vmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{vmatrix}$$

$C^T(\text{Codeword}) = C \cdot G$

$$C^T = \begin{vmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{vmatrix}$$

$$H = \begin{vmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{vmatrix}$$

$H.C^T = 0$

2. Graphical Representation The complete *H* matrix can best be described by a Tanner graph [56], a graphical representation of associations between code bits and parity checks. Each row of

H corresponds to a check node (CN), while each column corresponds to a variable node (VN) in the graph. An edge e_{ji} on the Tanner Graph connects a VN_j with CN_i only if the corresponding element h_{ij} is a 1 in H. If the number of edges entering in a node is same for all nodes of the graph, the LDPC code is called regular, being otherwise irregular in case of variable node degree. Generally, irregular LDPC codes yield better decoding performance compared to regular ones. Figure 5.1 illustrates the H matrix for Hamming code (7,4).

Figure 5.1: Tanner Graph of the Hamming code (7,4).

5.1.2 Quasi-Cyclic Low-Density Parity-Check (QC-LDPC)

Quasi-Cyclic LDPC (QC-LDPC) codes are a special class of LDPC codes with a structured H matrix composed of circulant *Permutation matrices* . A circulant permutation matrix is an identity matrix whose rows are cyclically shifted (to the right)s. The row and column weights of a circulant are the same. QC-LDPC can be generated by the expansion of a $Z \times Z$ permutation matrix. Z is also sometimes referred to as expansion factor of the LDPC check matrix. As an example, Figure 5.2 shows the parity check matrix for the (1944, 972) 802.11n LDPC code with sub-matrix size Z=81. In this matrix representation, each square box with a label I_x represents an 81×81 permutation matrices with a shifted value of x, and each empty box represents an 81×81 zero matrix . This structure of the code, implies that the check nodes of a row are connected to

I_{57}				I_{50}		I_{11}		I_{50}		I_{79}		I_0	I_0										
I_3		I_{28}		I_0				I_{55}	I_7				I_0	I_0									
I_{30}				I_{24}	I_{37}			I_{56}	I_{14}					I_0	I_0								
I_{62}	I_{53}			I_{53}			I_3	I_{35}							I_0	I_0							
I_{40}			I_{20}	I_{66}			I_{22}	I_{28}								I_0	I_0						
I_0				I_8		I_{42}		I_{50}			I_8						I_0	I_0					
I_{69}	I_{79}	I_{79}				I_{56}		I_{52}				I_0						I_0	I_0				
I_{65}				I_{38}	I_{57}			I_{72}		I_{27}									I_0	I_0			
I_{64}				I_{14}	I_{52}			I_{30}			I_{32}									I_0	I_0		
	I_{45}		I_{70}	I_0				I_{77}	I_9												I_0	I_0	
I_2	I_{56}		I_{57}	I_{35}						I_{12}												I_0	I_0
I_{24}		I_{61}		I_{60}			I_{27}	I_{51}			I_{16}	I_1											I_0

Figure 5.2: Parity check matrix H for a block length of 1944 bits, code rate 1/2, IEEE 802.11n (1944, 972) LDPC code. H consists of M×N sub-matrices (M=12, N=24 in this example)

unique variable nodes. This simplicity makes QC-LDPC codes attractive for implementation. It reduces the memory requirement based on block matrices with the circulant architecture.

They can be efficiently encoded using simple feedback-shift registers with complexity linearly proportional to the number of parity bits for serial encoding, and to the code length for parallel encoding [57].

5.1.3 QC-LDPC Codes Parameters in WiMAX and WiFi Standards

Low-density parity-check (LDPC) codes have been adapted in new standards such as in the IEEE 802.11n (WiFi) and 802.16e (Mobile WiMAX) standards. These standardized LDPC codes are based on structured irregular QC-LDPC codes which exhibit great levels of scalability, supporting multiple code structures of various code rates and code lengths. As a result, a decoder for these applications must be very flexible and reconfigurable. In WiMAX specification, 19 expansion factors which represent permutation matrix sizes are defined ranging from 24 to 96 with an increment of 4. On the other hand, In WiFi specification, only 3 expansion factors are defined ranging from 27 to 81. Table 5.1 summarizes the LDPC code parameters for these two standards.

Parameter	WiMAX			WiFi		
	#	min	max	#	min	max
Block Lengths	19	576	2304	3	648	1944
Submatrix sizes	19	24	96	3	27	81
Code Rates	4	1/2	5/6	4	1/2	5/6
CN degrees	7	6	20	9	7	22
VN degrees	4	2	6	8	2	12
Edges	90	1824	8448	8	2376	7128

Table 5.1: LDPC code parameters in WiFi and WiMAX

The main differences between these two standards can be summarized as follow:

- Maximum check node degree is 20 in WiMAX and 22 in WiFi.
- Maximum variable node degree is 6 in WiMAX and 12 in WiFi.
- Maximum submatrix size is 96 in WiMAX and 81 in WiFi.
- Maximum number of edges is 8448 in WiMAX and 7128 in WiFi.
- Maximum required channel throughput is up to 100Mbps for the WiMAX, and 450Mbps for WiFi.

5.2 LDPC Decoding Algorithms

Several algorithms are proposed for LDPC decoding and most of them are derived from the well-known belief propagation (BP) algorithm [55]. The principle consists of exchanging iteratively messages (probabilities, or beliefs) along the edges of the Tanner graph. The message (λ^i_{nm}) which is passed from a variable node (VN) n to a check node (CN) m is the probability that VN_n has a certain value (0 or 1). It depends on the channel value (Δ_n) of that variable node n and all the messages from connected check nodes to VN_n except m. Similarly, the message from CN_m to VN_n (Γ^i_{mn}) depends on all messages received from connected VNs except the one being updated (as shown in Figure 5.3). These two phases are often called as *check node*

update and *variable node update*, respectively. This algorithm is also referred to as two-phase message passing (TPMP). When all variable nodes and check nodes have been updated, one iteration is said to be complete. Iterations are executed until all parity-check equations (equation 5.1) or another stopping criterion are satisfied or the maximum number of iterations is reached.

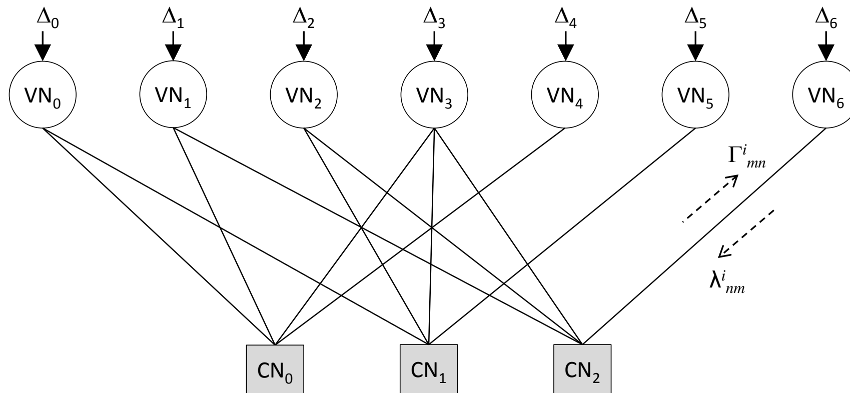

Figure 5.3: Tanner graph example showing the two-phase message passing decoding

5.2.1 Min-Sum Algorithm

The above decoding algorithm is usually described in the log domain using log-likelihood-ratios (LLR) to reduce the computational implementation complexity. The Min-Sum algorithm [58] is the hardware efficient implementation of the TPMP decoding using LLRs. It proposes an approximation to simplify the calculations of updated messages. In this section we present the computations implied by this algorithm with a slightly modified formulation adapted to the hardware architecture which will be presented in subsequent sections.

Every decoding iteration consists of M sub-iterations corresponding to the M-check node groups in the H_{base}. Each sub-iteration i consists of two phases:

1. *CN-update:* all the VN nodes send extrinsic messages given by equation (5.2) to their corresponding CNs in the check node group. These messages contain the sum of the extrinsic messages sent by the other CNs to the VN and the channel value. Here, we denote $M(n)\backslash m$ to be the set of the all check nodes connected to the variable node n except the check node m.

$$\lambda_{nm}^{i} = \Delta_{n}^{i-1} + \sum_{m' \in M(n)\backslash m} \Gamma_{m'n}^{i-1},\ \Gamma_{mn}^{0} = 0 \tag{5.2}$$

2. *VN-update*: when a CN receives all the messages, it sends a message to each connected variable node. We denote $N(m)$ to be the set of the all variable nodes connected to the check node m, and $N(m)\backslash n$ to be the same set except the variable node n. Then the check node to variable node message is given as in equation (5.3). The sign of the message is the

product of the signs of the messages received from all VNs, as given in equation (5.3).

$$\begin{aligned} sgn_m &= \prod_{n \in N(m)} sgn(\lambda^i_{nm}) \\ min_{n'm} &= \min_{n' \in N(m) \backslash n} (|\lambda^i_{n'm}|) \end{aligned} \tag{5.3}$$

It can be observed that the magnitudes of the messages leaving a check node have only two values: either the overall minimum ($min0$) of the received messages $|\lambda^i_{nm}|$ or the second overall minimum ($min1$). In fact, $min1$ is sent to the variable node who sent $min0$. Calculations at the check node thus result in tracking the two running minimums, $min0$ and $min1$, the index of the connected variable node providing $min0$ (ind), and the overall sign (sgn). These four informations are grouped and denoted as *Running Vector* (RV):

$$RV(m) = [min0, min1, ind, sgn]_m \tag{5.4}$$

The extrinsic information (Γ_{mn}) is derived at the variable node as

$$\Gamma^i_{mn} = sgn(\lambda^i_{nm}) \times sgn_m \times (min0 \; or \; min1) \tag{5.5}$$

Thus, the overall estimation (a posteriori LLR) of the decoded bit can be computed as:

$$\lambda^i_n = \Delta_n + \sum_{m \in M(n)} \Gamma^i_{mn} \tag{5.6}$$

It can also be written in the following form:

$$\lambda^i_n = \lambda^i_{nm} + \Gamma^i_{mn} \tag{5.7}$$

The sign of λ^i_n indicates the hard decision on the decoded bit.

Using this last equation, we can rewrite equation(5.2) as:

$$\lambda^i_{nm} = \lambda^i_n - \Gamma^{i-1}_{mn} \tag{5.8}$$

The above steps are repeated for all check node groups to complete one iteration.

5.2.2 Normalized Min-Sum Algorithm

Min-Sum algorithm can induce a loss of about $0.2dB$ compared to the original BP version [59].The error rate performance of the Min-Sum algorithm can be improved by employing a *CN-update* that uses an extrinsic scaling factor α. The normalization factors can be obtained by simulation, typically $0 < \alpha < 1$. Normally, for hardware implementation, the value of $\alpha = 0.875$ is used as it gives a good trade-off between complexity and BER performance. Hence, equation (5.5) becomes:

$$\Gamma^i_{mn} = \alpha \times sgn(\lambda^i_{nm}) \times sgn_m \times (min0 \; or \; min1) \tag{5.9}$$

5.3 Multi-ASIP Architecture Overview

Figure 5.4 shows an overview of the proposed multi-ASIP architecture for LDPC/Turbo decoding. It consists of 8 ASIPs interconnected via a de-Bruijn Network-on-Chip (NoC) [22][60] grouped into two component decoders for Turbo mode. Each ASIP can process three CNs in parallel, a total of 8 ASIPs are required to process one complete sub-matrix in parallel for the minimum sub-matrix size $Z = 24$ (Table 5.1). This number of ASIPs is just enough in turbo mode to work in 4×4 mode to achieve the targeted 150Mbps throughput.

Within each component decoder the ASIPs are also connected by two 8-bit bus (named here to be $\alpha - \beta$ bus) for the exchange of sub-block boundary state metrics. In LDPC decoding mode each ASIP can process three check nodes in parallel.

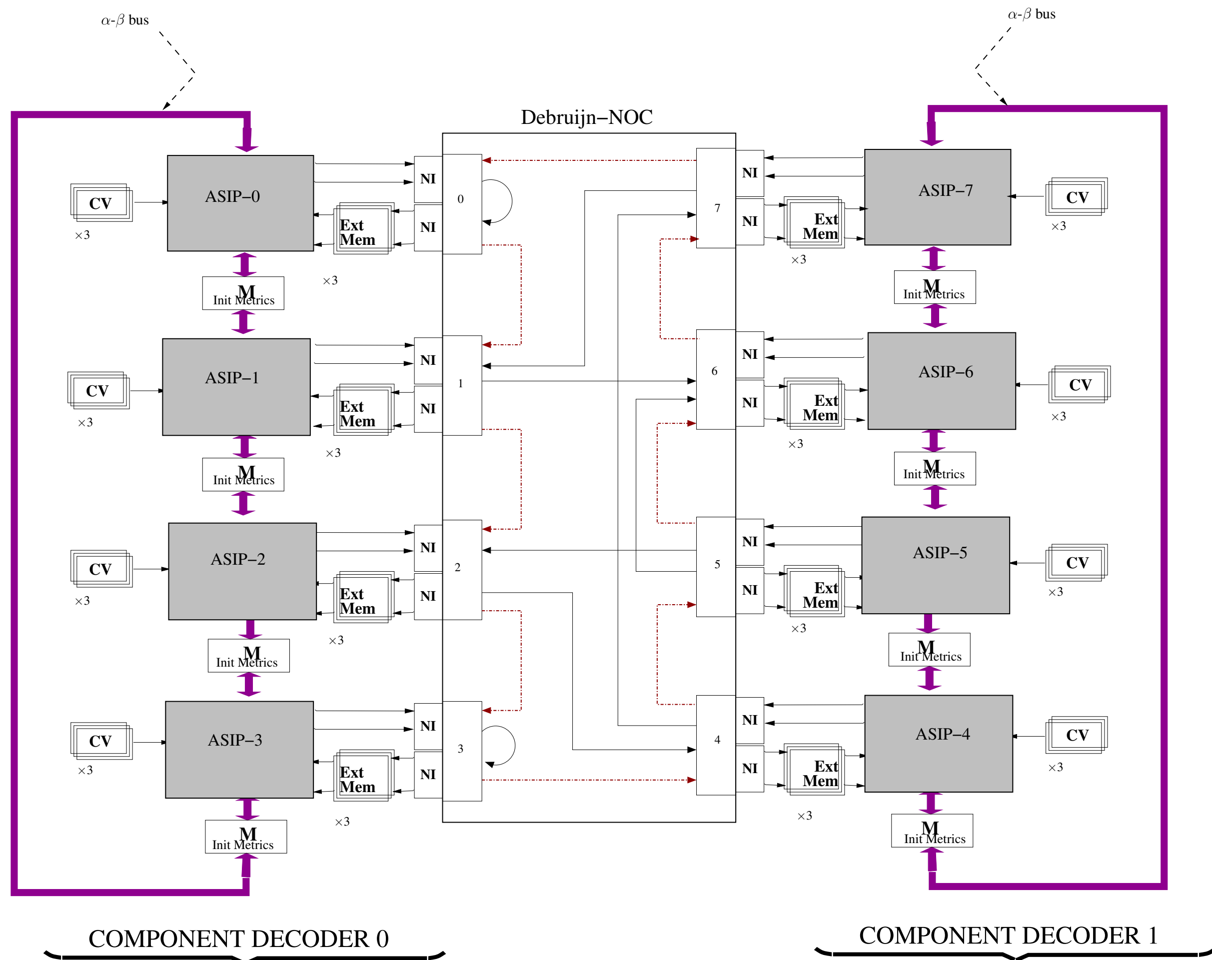

Figure 5.4: Overview of the proposed Multi-ASIP architecture for LDPC/Turbo decoding

Each ASIP has 3 memory banks of size 24×256 used to store the input channel LLR values (CV memories). There are also another 3 banks of size 30×256 used for storing extrinsic information. Each ASIP is further equipped with two 80×32 memories to store state metrics β in Turbo mode and λ^i_{nm} in LDPC mode (not shown in Figure 5.4).

5.3.1 Multi-ASIP in Turbo Mode

In Turbo decoding mode, the ASIPs are configured to operate in a 4×4 shuffled decoding (see Subsection 1.4.2.2 in Page 26), with four ASIPs (0-3) processing the data in natural order while the other four ASIPs (4-7) are processing it in interleaved order. The generated extrinsic information is exchanged via the NoC (de-Bruijn). The choice of de-Bruijin network was proposed by another Ph.D. student [22]. It was proved to be less complexity than other networks such as Butterfly and Benes networks. Moreover, this network is easier to be reconfigured as a unidirectional interconnect Ring network, which is used in LDPC mode (Presented in the next subsection).

The proposed architecture for each ASIP is based on the TurbASIP$_{v1}$ architecture (see Subsection 3.2.1.1 in Page 49). Since the base system architecture integrates 8 ASIPs, and the maximum target throughput is 150Mbps (LTE), the internal parallelism degree of the ASIP can be reduced. Hence, one recursion unit is used (rather than 2 in the base ASIP architecture) which decreases the logic complexity and still meeting the target throughput. Furthermore, Backward-Forward scheme is used rather than the butterfly scheme for BCJR metrics computation.

The NoC in the Turbo mode is configured in binary de-Bruijn topology as shown in Figure 5.5 and the $\alpha-\beta$ buses are configured to form 2 networks in Ring topology. There are two kinds of messages that are exchanged between the ASIPs.

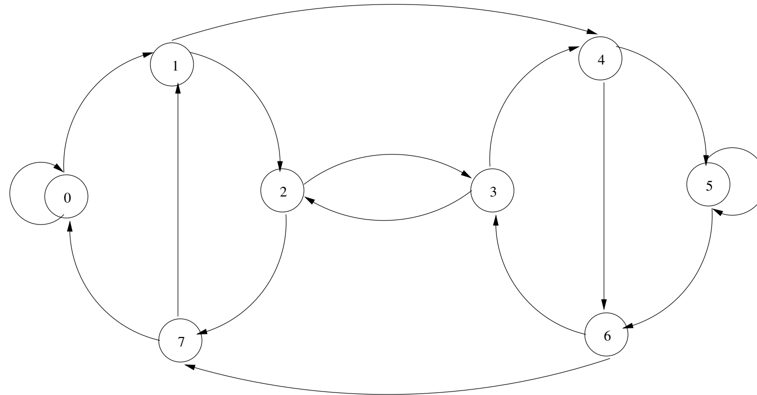

Figure 5.5: NoC with binary de-Bruijn topology in Turbo mode

- Extrinsic LLR information: Extrinsic LLRs of the processed symbols are exchanged between the two component decoders through the binary de-Bruijn NoC. Packet header carries the interleaved address (for ASIPs processing symbols in natural order) or deinterleaved address (for ASIPs processing symbols in interleaved order).
- State metric information: State metrics α and β at sub-block boundaries are exchanged at the end of each iteration between neighboring ASIPs in a circular scheme via the $\alpha - \beta$ Ring network. This implements the message passing initialization method by initializing sub-blocks with the recursion metrics computed during the last iteration in the neighboring sub-blocks (ASIPs).

5.3.2 Multi-ASIP in LDPC Mode

In the LDPC mode, the binary de-Bruijn NoC is reconfigured to form unidirectional interconnect Ring network of 46 bits wide as shown in Figure 5.6. Each variable node message (λ^i_{nm}) and

extrinsic check node message (Γ^i_{mn}) are quantized to 7 and 5 bits respectively. The input frame is partitioned into 8 sub-blocks corresponding to the 8 ASIPs and written into 3 banks. Each banks store channel LLRs of size Z.

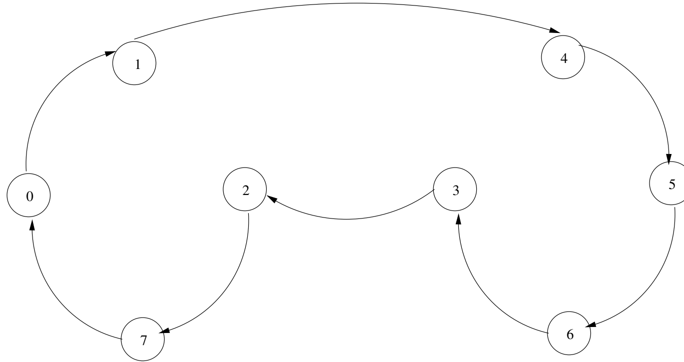

Figure 5.6: NoC with reconfigured binary de-Bruijn topology in LDPC mode

5.4 Computational Scheduling

5.4.1 Computational Scheduling in Turbo Mode

Figure 5.7 illustrates the adopted windowing technique and Backward-Forward BCJR computations scheduling in Turbo mode. The ASIPs calculate first state metrics β in the backward recursion, followed by state metrics α and extrinsic generation in forward recursion. Sub-block boundaries state metrics initializations $(\alpha_int(w^i_{(n-1)}), \beta_int(w^i_{(n-1)}))$ are done by message passing via the two 10-bits $\alpha - \beta buses$ connecting the ASIPs.

5.4.2 Computational Scheduling in LDPC Mode

In LDPC mode, each ASIP operates as a variable node and check node processing engine. An ASIP processes 3 check nodes and its associated edges from the 3 consecutive variable node groups. As an example, ASIP0 processes all the check nodes (3 at a time) that are associated with VNG[0..2], while ASIP1 processes those associated with VNG[3..5], etc. In other words, an ASIP in this mode can process three CNs present in the same group and its corresponding variable nodes present in three different variable node groups. In order to explain the proposed scheduling, we first give a simplified example using 2-ASIP architecture then we present the scheduling for the 8-ASIP configuration.

5.4.2.1 Simple example with 2-ASIP architecture

Let us consider the example of an LDPC check matrix with 36 variable nodes and 12 check nodes represented with the H_{base} permutation matrix of Figure 5.8. Figure 5.9 shows the check nodes and variable nodes mapping for a 2 ASIP decoding architecture (*ASIP0, ASIP1*) and the edges processed at the time *t*=T_0. The example H_{base} matrix has 2 check node groups (CNGs) and 6 variable node groups (VNGs). The proposed scheduling for LDPC decoding is illustrated as follows:

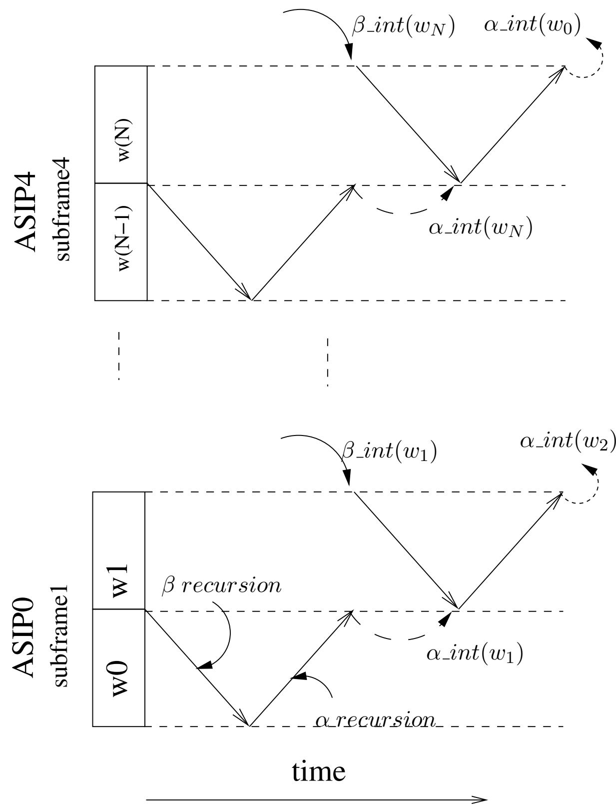

Figure 5.7: Forward backward schedule

0	1	3	4	1	2
4	1	0	1	4	1

Figure 5.8: H_{base} example with N=6 and M=2

$t = T_0$: ASIP0 reads the LLR values associated with the check nodes m=(0,1,2). These LLRs correspond to λ_n and Γ_{mn} values related to the 9 variable nodes n=[(0,1,2),(7,8,9),(15,16,17)]. These variable nodes belong to 3 variable node groups VNG[0,1,2]. Note that memories banks should be organized in a way to enable simultaneous access to all these values.

Similarly, ASIP1 handles the check nodes m=(3,4,5) and the associated variable nodes n=[(19,20,21),(27,28,29), (34,35,30)] of the VNG[3,4,5] (figure 5.9).

Each ASIP calculates variable to check node messages λ_{nm} according to (5.8). Additionally, it processes equation (5.3) and produces 3 sets of messages (corresponding to the three check nodes) that we denote by $RV(m)_{T0}^{ASIP_x}$, where $x = [0, 1]$. The $RV(m)_{T0}^{ASIP_x}$ set of messages carries the following informations:

1. The 2 least minimums (min0, min1). .
2. The ASIP ID and the channel memory bank number to locate the index (ind) corresponding to the least minimum.
3. sgn_m which is the XOR of the sign of the 3 λ_{nm}^i messages calculated for the check node m.

These messages are sent to the next ASIP through the de-Bruijn NOC.

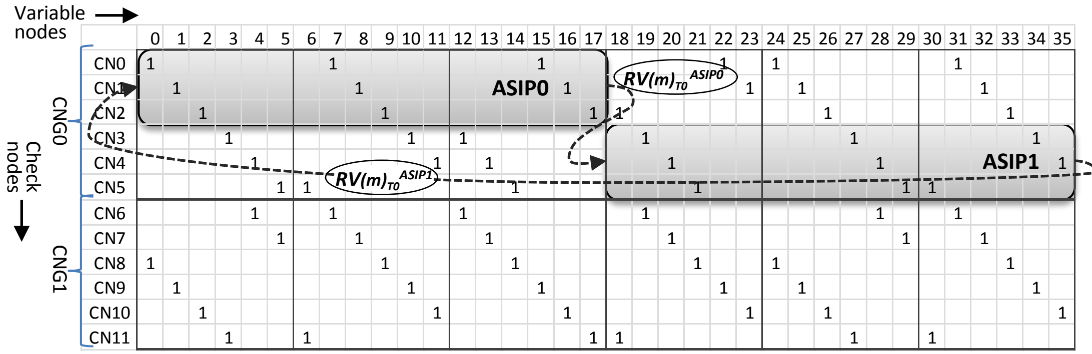

Figure 5.9: Proposed computational scheduling in LDPC mode with 2 ASIPs at t=T_0

$t = T_1$: ASIP0 reads the LLR values associated with the check nodes m=(3,4,5). These LLRs correspond to λ_n and Γ_{mn} values related to the 9 variable nodes n=[(3,4,5),(10,11,6),(12,13,14)].

Similarly, ASIP1 handles the check nodes m=(0,1,2) and the associated variable nodes n=[(18,22,23),(24,25,26), (31,32,33)] of the VNG[0,1,2] (figure 5.10). Both ASIPs generate $RV(m)_{T1}^{ASIP_x}$ messages as in the previous step except that the new messages take into account the $RV(m)_{T0}^{ASIP_x}$ messages received from the other ASIP.

This completes the *CN-update* phase, with $RV(m)_{T1}^{ASIP1}$ containing the final (min0, min1) information associated with check nodes m=(0,1,2).

Similarly, $RV(m)_{T1}^{ASIP0}$ contains the final (min0, min1) information associated to check nodes m=(3,4,5). We represent these final messages to be the *Update Vector* (UV) $UV(m)^{ASIP_x}$ that is circulated again to the next ASIP as shown in Figure 5.10.

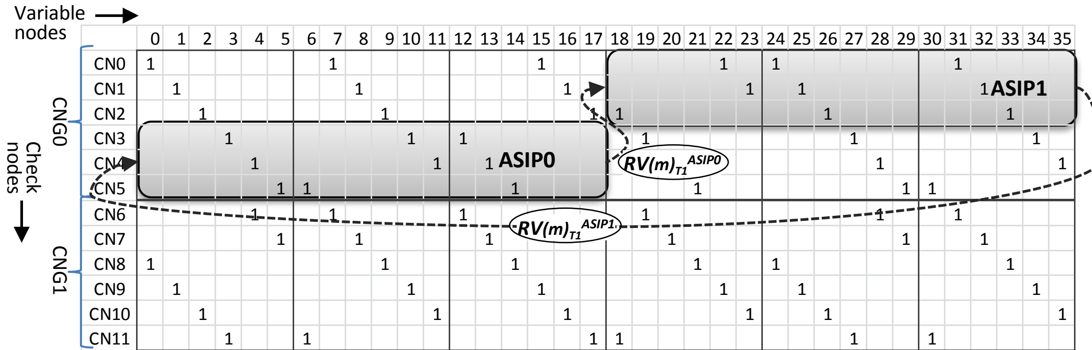

Figure 5.10: Proposed computational scheduling in LDPC mode with 2 ASIPs at t=T_1

$t = T_2$ ASIP0 calculates the final a posteriori LLRs λ_n^i using the $UV(m)^{ASIP1}$ messages and the λ_{nm}^i messages related to n=[(0,1,2),(7,8,9),(15,16,17)] and m=(0,1,2) according to equation 5.7.

Extrinsic messages Γ_{mn}^i are generated according to equation 5.9 and stored in the extrinsic memory banks of the ASIP.

Similarly, ASIP1 calculates LLRs λ_n^i using $UV(m)^{ASIP0}$ message, where n=[(19,20,21),(27,28,29), (34,35,30)] and m=(3,4,5). Both ASIPs forward the $UV(m)^{ASIPx}$ message to the next ASIP (Figure 5.11).

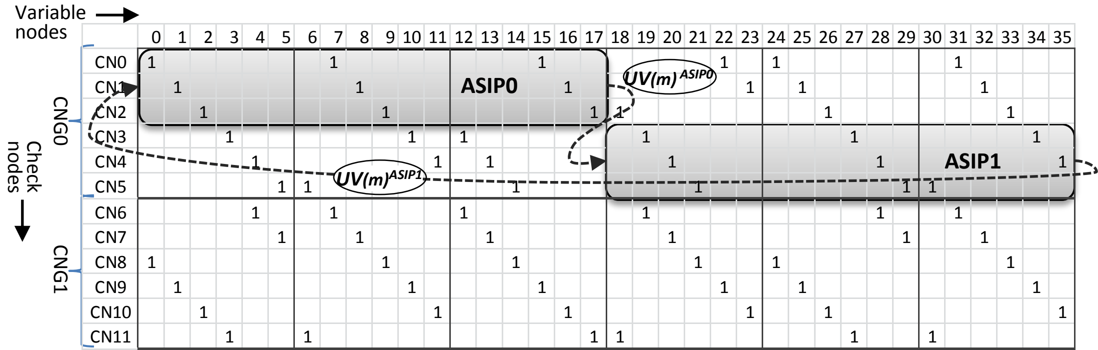

Figure 5.11: Proposed computational scheduling in LDPC mode with 2 ASIPs at t=T_2

$t = T_3$ Similar to the previous step at $t = T_2$, ASIP0 calculates aposteriori LLRs λ_n^i using $UV(m)$ where n=[(19,20,21),(27,28,29),(34,35,30)] and m=(0,1,2) and ASIP1 calculates λ_n^i for n=[(0,1,2),(7,8,9),(15,16,17)] using $UV(m)$ where m=(3,4,5) as shwon in Figure 5.12.

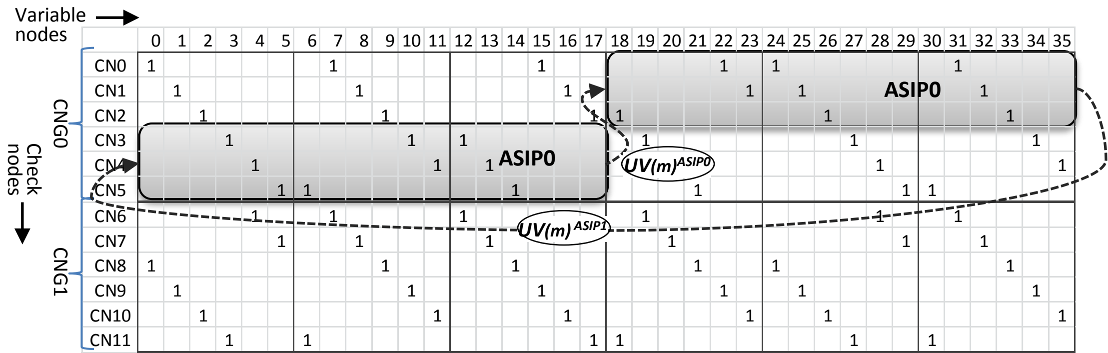

Figure 5.12: Proposed computational scheduling in LDPC mode with 2 ASIPs at t=T_3

The above 4 time steps completes one sub-iteration carried over a Check node group (CNG0) with 2 ASIPs. The first two time steps correspond to the *CN-update* phase (RV), while the last two time steps correspond to the *VN-update* phase (UV). Thus according to the proposed computational scheduling, we complete one sub-iteration in 4 time steps (2 ASIPs x (1RV+1UV)) where each ASIP processes 3 check nodes associated with 3 VNGs simultaneously.

5.4.2.2 Proposed scheduling with 8-ASIP architecture

LDPC check matrices specified in WiFi and WiMAX standards contain 24 VNGs. Processing all the VNGs simultaneously requires 8 ASIPs. Thus, with the proposed scheduling, these 8 ASIPs will be able to process 24 (8x3) check nodes simultaneously. Now, if the CNG contains 24 check nodes (Z=24), a sub-iteration is completed in 16 time steps (2 phases x 8 ASIPs).

However, the sub-matrix size Z can have different values: Z=24, 28, 32, ... 96 for WiMAX and Z=27,54 or 81 for WiFi. So, for the case of Z is greater than 24, the check nodes inside a CNG are divided into sub-groups and the above schedule is applied sub-group by sub-group. Figure 5.13 illustrates the proposed scheduling applied over a WiFi LDPC check matrix with Z=27. Here the check nodes are divided into two sub-groups, one contains 24 check nodes and the second contains 3 check nodes. Thus, one sub-iteration is completed in 32 time steps: 2 sub-groups x 8 ASIPs x (1RV+1UV). However, due to the sub-matrix structure (rotated permutation property), the second sub-group of check nodes does not have any common VNs with the previous sub-group. Therefore, the *VN-update* phase of the first sub-group can take place along with the *CN-update* of the second subgroup, thus completing the sub-iteration in 24 time steps: 8 ASIPs x (1RV+1RVUV+1UV). This scheduling based on sub-groups of check nodes enables to handle efficiently all specified matrix-size Z in the WiFi and WiMAX standards. For any sub-matrix size Z, a sub-iteration is completed in $T_{sub-iteration}$ time steps, given as:

$$T_{sub-iteration} = 8_{ASIPs} \times (1_{RV} + (\lceil Z/(3_{CNs} \times 8_{ASIPs}) \rceil - 1)_{RVUV} + 1_{UV}) \qquad (5.10)$$

5.4.2.3 Address generation in LDPC mode

Each ASIP has access to 3 memory banks holding channel LLR values of the size of one sub-matrix. The check nodes are processed based on the start address given by an internal LDPC address generator. If the number of the ASIPs $N_A = 8$, the address pattern to be generated is given by the following pseudo-code:

$$
\begin{aligned}
&for(subgroup = 0; subgroup < \lceil (Z/(3 * N_A)) \rceil; subgroup++)\{\\
&\quad for(timestep = 0; timestep < 8; timestep++)\\
&\qquad address = mod(I_{x,y} + subgroup * (3 * N_A) + 3 * (timestep), Z)\\
&\}
\end{aligned}
$$

Where $I_{x,y}$ is the shift value for the permutation matrix at location (x, y) and $subgroup$ is the current processed sub-group of check nodes composed of 8×3=24 CNs (e.g. for $Z = 96$, 96 check nodes are processed in 4 sub-groups).

5.5 Memory Organization and Sharing

One of the main features of the proposed architecture concerns the memory sharing between LDPC and Turbo mode. The memory is organized in a way to provide enough bandwidth to support efficiently both Turbo and LDPC modes. Since the memory access and storage patterns are quite different for both modes they are first explained hereafter individually. The interleaver memories are avoided by proposing adequate address generators.

5.5.1 Memories in Turbo Mode

The memory requirement in Turbo mode to support LTE, WiMAX and DVB-RCS standards is summarized in Table 5.2.

The extrinsic memory is organized in 3 banks with the extrinsic LLR information for each symbol being stored as shown in Figure 5.14(a) for double binary mode (DBTC). Similarly, the input memory is organized in 3 banks to support 12 windows per ASIP. The systematic (S_x) and parity (P_x) input soft values are stored as shown in Figure 5.14(b).

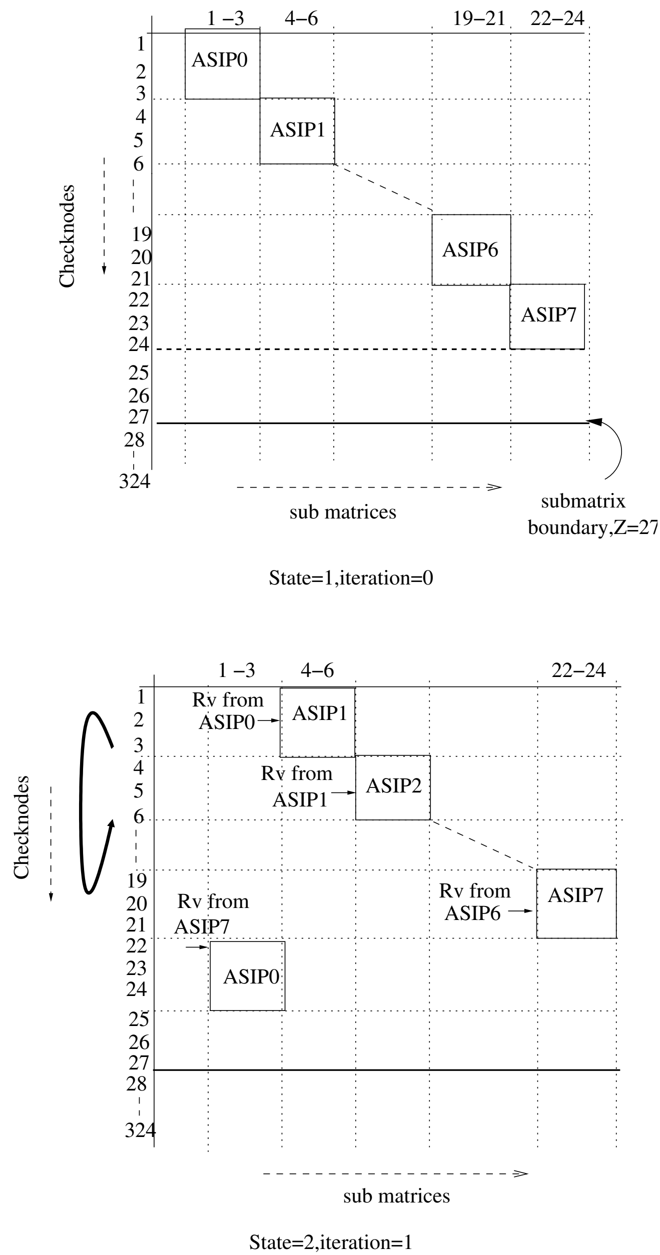

Figure 5.13: Proposed scheduling in LDPC mode with 8 ASIPs

Memory	width (bits)	depth
Input memory	(4×4)=16	(6144)/(4×2)=768
Ext memory (odd)	8	(6144)/(4×2×3)=768
Ext memory (even)	(2×8)=16	(6144)/(4×2×3)=768
Windowing memory	80	12
Cross metric memory	10×8=80	64

Table 5.2: Memory requirement in Turbo mode

The cross metric memory stores the state metrics β values generated during the backward recursion. These values are required to generate the extrinsic information during the forward recursion.

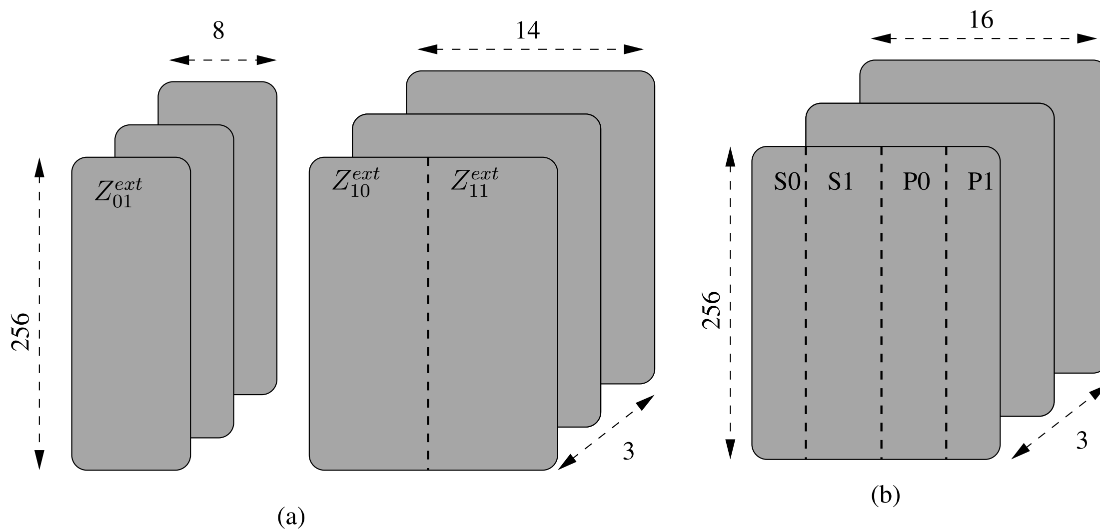

Figure 5.14: Memory banks organization in Turbo mode: (a) Extrinsic memory, (b) Input memory

5.5.2 Memories in LDPC Mode

In LDPC mode, and targeting the support of WiFi and WiMAX, the memories should be dimensioned to allow the maximum frame size of 2304 (Table 5.1). The input frame is partitioned into 8 sub-blocks corresponding to the 8 ASIPs and each sub-block is divided further to 3 banks of size equal to the sub-matrix size Z, with each memory bank is mapped to one variable node group as shown in Figure 5.15(a). Each location of the CV memory holds two λ_n^i messages. As 7 bits are used for the quantization of λ_n^i, hence the CV memory width is 14 bits. Furthermore, as the maximum value of $Z = 96$ (for WiMAX mode), hence the CV memory depth is equal to $\frac{2304}{8(\text{ASIPs})\times 2(\text{Values per memory location})\times 3(\text{Banks})} = 48$.

Similarly, there are 3 Extrinsic memory (Ext) banks per ASIP that store Γ_{mn}^i values. Since the VN degree is at most 12 (for WiFi standard), there can be at most 12 check node messages Γ_{mn}^i for each VN. These messages are stored in extrinsic memory in 4 groups each at an offset of 48 (see Figure 5.15(b)).

Furthermore, two intermediate FIFOs are required to store the update address locations (used during the *VN-update* phase) and the intermediate value (λ_{nm}^i) that is used to add to the update message received respectively. The depth is twice the time required for the *VN-update* phase to start with the*CN-update* phase of the check nodes (i.e. 32).

The memory requirements in LDPC mode are summarized in Table 5.3.

Memory	Number of memories	width(bits)	depth
Input memory	3	7×2=14	48
Ext memory	3	(5×3)×2=30	(12×96)/(3×2)=192
Intermediate FIFO	2	80	32

Table 5.3: Memory requirement in LDPC mode

5.5.3 Combined Memory Organization

The extrinsic and input memories for both modes are shared. The Cross metric memory is partitioned into 2 blocks. In the LDPC mode, the block0 is shared and used to store the intermediate

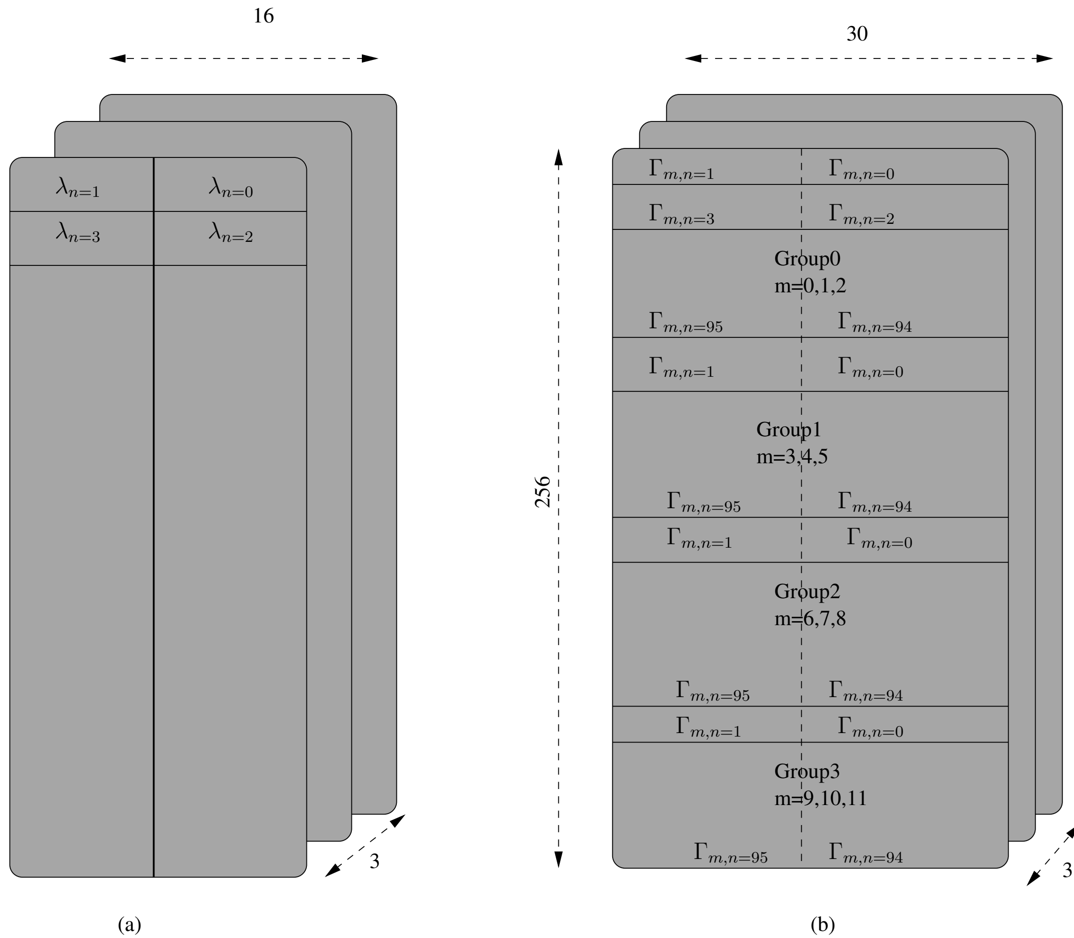

Figure 5.15: LDPC mode : (a) Input memory (b) Extrinsic memory

values λ_{nm}^{i}, whereas block2 is reused to store the read extrinsic values and the update addresses for input memory. The combined memory organization is as shown in Table 5.4. Depth and width of each shared memory are dimensioned to support the maximum requirement in LDPC and Turbo modes.

Memory	Number of memories	width (bits)	depth
Input memory	3	16	256
Ext memory	3	30	256
Intermediate FIFO or Cross metric memory (block0)	1	80	32
Address + Ext mem FIFO or Cross metric memory (blockk2)	1	80	32

Table 5.4: Proposed memory organization to support both Turbo and LDPC modes

5.6 ASIP Pipeline and Instruction-Set

The proposed ASIP architecture integrates two different pipelines stages, one for LDPC decoding and the other for Turbo decoding. This architectural choice has been made based on the following three main reasons: (1) LDPC decoding based on normalized Min-Sum algorithm

requires low complex arithmetic and logic resources, (2) no real/explicit computation commonalities between the two considered algorithms for LDPC decoding and Turbo decoding, and (3) memory and communication requirements account for the major complexity part of the considered two iterative decoders (and more particularly for LDPC). Hence, the proposed ASIP architecture presents and efficient sharing of memory and communication resources while it integrates two different computational resources for both supported algorithms.

Pipeline stages and the instruction set proposed for Turbo mode are derived from the TurbASIP$_{v1}$ presented in Subsection 3.2.1.1. Hence, this section focuses on the presentation of the proposed ASIP pipeline structure and instruction set dedicated for LDPC mode.

5.6.1 LDPC Decoder Pipeline Stages

The proposed ASIP architecture is organized to realize the LDPC decoding in 8 pipeline stages: **Prefetch**, **Fetch**, **Decode**, **Operand Fetch**, **CVnExtRead**, **TwoMinBnk3**, **ReadNoc**, **UpdateNoc**. Figure 5.17 presents the proposed pipeline structure in LDPC mode and how the normalized Min-Sum algorithm computations (referenced by the corresponding equations from Subsection 5.2.1) are mapped on the different pipeline stages.

The hardware resources of these pipeline stages have been devised in a way to support the simultaneous execution of both *VN-update* phase (UV) and *CN-update* phase (RV).

5.6.2 Pipeline Description in The *CN-update* Phase (RV)

The behavior of the proposed pipeline stages in the *CN-update* phase (RV) is described below.

Decode Stage:
Address generation (Subsection 5.4.2.3) for Input and Extrinsic memories is performed in this pipeline stage. These addresses are used in the next stage Operand Fetch to read current channel λ_n and extrinsic Γ_{mn} values.

Operand Fetch Stage:
In order to read the next three consequent λ_n and Γ_{mn} values from Input and Extrinsic memory banks, two clock cycles are required. These two clock cycles are taking place in Operand Fetch and CVnExtRead stages. The read values in this stage are buffered to internal registers.

CVnExtRead Stage:
In this stage, the next remaining consequent values (either one or two values) for both channel Δ and extrinsic information Γ are read from corresponding memories. However, the values read in the previous stage (Operand Fetch) are transferred to different registers to avoid data conflict with other instructions during the pipeline execution. Then, these values (Γ_{mn} and λ_n) are used to compute the intermediate values λ_{nm}. This computation is done by subtracting extrinsic values Γ_{mn} from channel values λ_n (Equation (5.2)).

TwoMinBnk3 Stage:
The intermediate values computed in the previous stage are stored in the intermediate FIFO to be used in the *VN-update* phase to compute the updated channel values. The three intermediate values of corresponding check node are also used to find the two least minimums ($min0$ and $min1$) and to buffer them into registers to be used in the next pipeline stage.

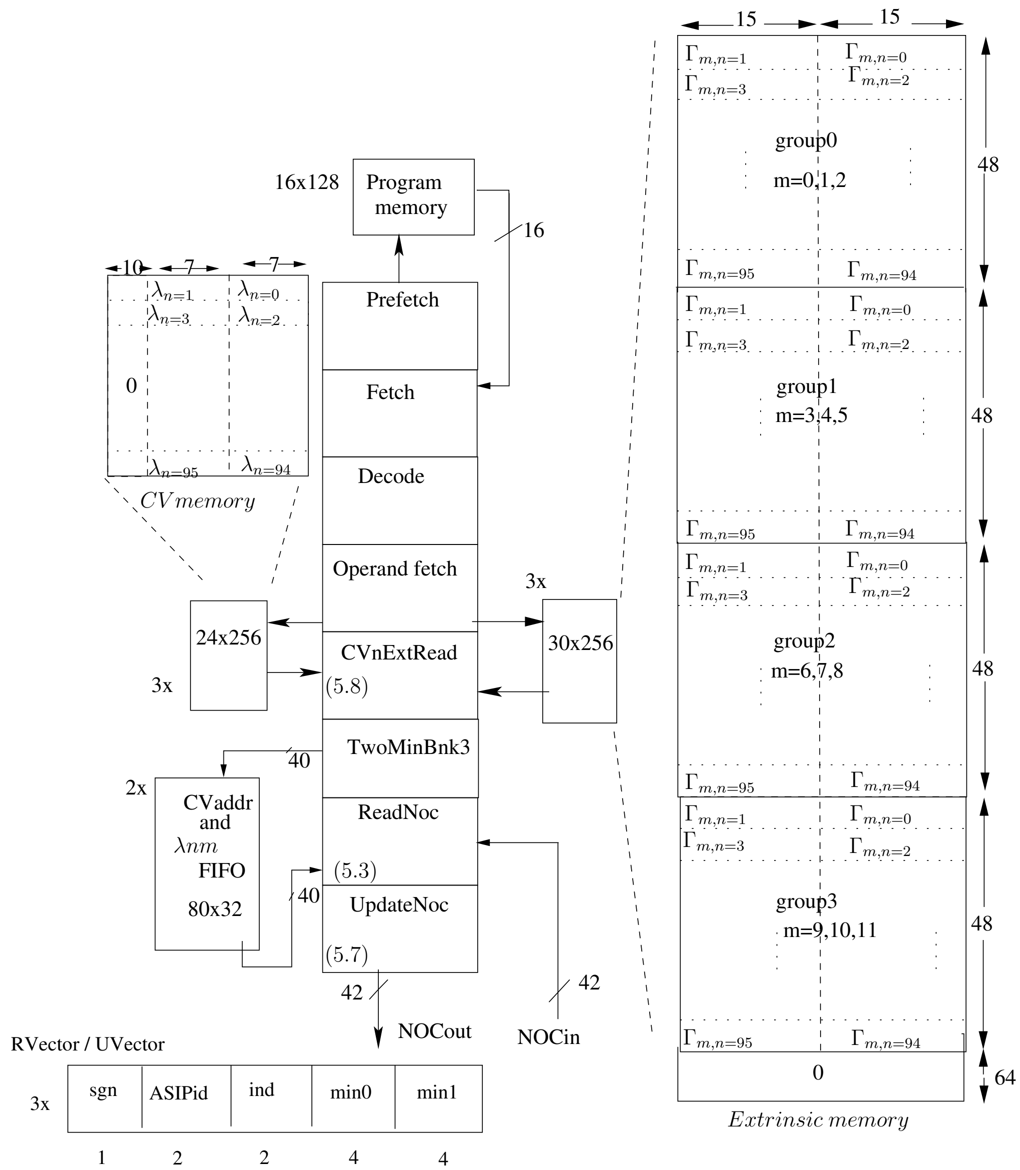

Figure 5.16: Pipeline in LDPC Mode

ReadNoc Stage:
In this stage, the ASIP reads the messages sent by the adjacent ASIP which correspond to the running least minimums for the processed CNs ($RV(m)^{ASIP_x}$). As in the previous stage, the considered ASIP continues finding the two least minimums by comparing the four available values: 2 received via the NoC and 2 computed in the previous pipeline stage.

UpdateNoc Stage:
In this stage, the updated two least minimums found in the previous stage are used to update the $RV(m)^{ASIP_x}$ message which is sent to the other adjacent ASIP. Thus the complete RV includeing the reading and updating of $RV(m)^{ASIP_x}$ message from previous ASIP take 2 clock cycles. (i.e. 1 time step = 2 clock cycles)

5.6.3 Pipeline Description in The *VN-update* Phase (UV)

The behavior of the proposed pipeline stages in the *VN-update* phase (UV) is described below.

CVnExtRead Stage:
Read request to intermediate FIFO is done. This stage executes a read request to the intermediate FIFO to fetch the intermediate values, extrinsic values, and the update addresses for Input memory, that are stored earlier in the *CN-update* phase.

TwoMinBnk3 Stage:
The intermediate values, extrinsic values, and the update addresses for Input memory are read from the intermediate FIFO However, the related intermediate values are stored in two rows in the FIFO, so another read request is done to read the remaining values.

ReadNOC Stage:
The remaining intermediate values extrinsic values, and the update addresses are read from the FIFO for Input memory. The message $RV(m)^{ASIP_x}$ (x= previous ASIPID) which has been received via the NoC from the adjacent ASIP is read.

UpdateNOC Stage:
The new a posteriori channel values λ_n are computed by adding the corresponding intermediate value with the corresponding extrinsic value (Equation (5.7)). Then, Extrinsic and Input memories are updated with the calculated values. Finally, the message $RV(m)^{ASIP_x}$ is forwarded again to the other adjacent ASIP via the NoC.

Figure 5.17 presents the functionality of the intermediate FIFO during the two phases (*CN-update* and *VN-update*).

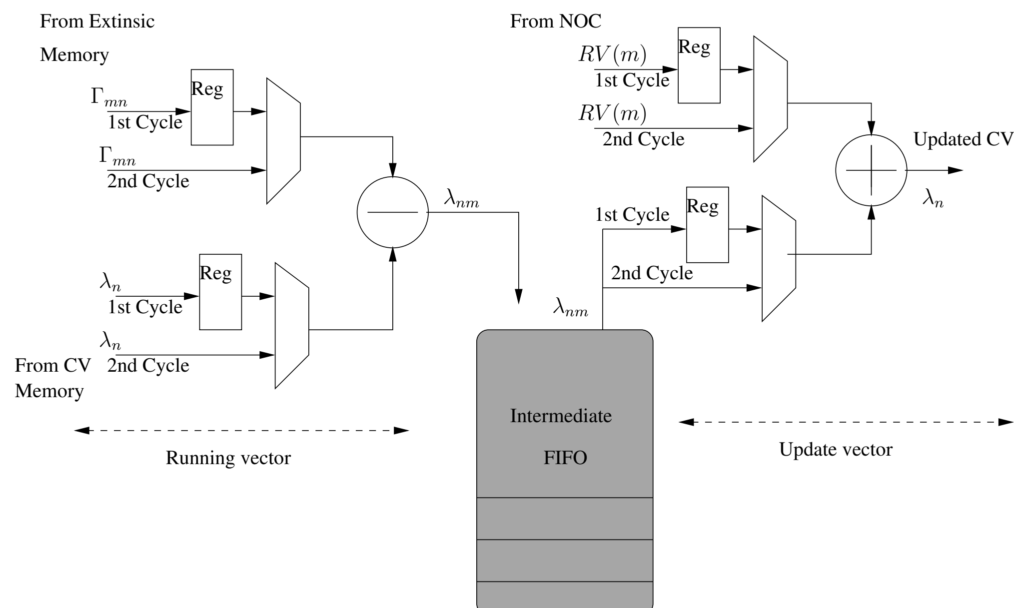

Figure 5.17: FIFO accesses in *CN-update* and *VN-update* phases

5.6.4 Sample Assembly Program in LDPC Mode

Listing 5.1 illustrates the proposed instruction set through an assembly code example for LDPC decoding with sub-matrix size $Z = 27$. The ASIP is first initialized with the number of the processed check nodes, sub-matrix size and three block columns of offset values from H_{base} (Lines 1..22). As the channel data is stored in couples, two clock cycles are needed to Read-/Write the channel data associated with three check nodes. A read operation to Input (CV) and Extrinsic (EXT) memories is accomplished by RunVec and RunVec1 instructions (Lines 29..30). The write operation is accomplished by UpdateVec and UpdateVec1 instructions. RunVec1 also generates the $RV(m)^{ASIP_x}$ message to the next ASIP. Similarly, UpdateVec forwards the $UV(m)^{ASIP_x}$ to the next ASIP. RunVecWithUpt reads the CV and EXT memories for check nodes 24..27, writes locations associated with check nodes 1..3 and then forwards this $UV(m)^{ASIP_x}$ packet to the next ASIP. RunVecWithUpt1 does the second phase read from the CV and EXT memories and also computes $RV(m)^{ASIP_x}$ message before forwarding it to the next ASIP. LDPCAddrGenInit instruction initiates the next address (see Subsection 5.4.2.3) to read or write from CV and EXT memories.

Listing 5.1: LDPC assembly code

```
1  ;8 ASIPs each processing 3 CN =24 at a time
2          LDPCsize PSize,24
3  ;submatrix size
4          LDPCZsize Zsize,27
5  ;rows,NumZerosTriplets
6          LDPCAddrRegInit1 1,7
7  ;SubRows=floor(PSize/Zsize)
8  ;num of ASIPs and RowRem=mod(Zsize,3)
9          LDPCAddrRegInit2 1,8,1
10 ;set ASIP ID
11         LDPCASIPid 0
12 ;writing the H matrix offset column 1
13         LDPCAddrConfig1 0,17
14         LDPCAddrConfig1 1,3
15         :
16 ; column 2
17         LDPCAddrConfig2 0,13
18         LDPCAddrConfig2 1,13
19         :
20 ; column 3
21         LDPCAddrConfig3 0,8
22         LDPCAddrConfig3 1,8
23         :
24 Repeat until _ITER for 20 times
25         PUSH
26         Repeat until _LOOP0 for 8 times
27 ; load and initialize the address generator
28         LDPCAddrGenInit
29         RunVec
30         RunVec1
31 _LOOP0: Repeat until _LOOP1 for 8 times
```

```
32             LDPCAddrGenInit
33             RunVecWithUpt
34             RunVecWithUpt1
35  _LOOP1:    Repeat until _LOOP2 for 8 times
36             LDPCAddrGenInit
37             UpdateVec
38             UpdateVec1
39  _LOOP2:    POP
40  ;ceil(Zsize/2)=14
41  _ITER:     Repeat until _DEC for 14 times
42             NOP
43             HardDecision
44  _DEC:      NOP
```

5.7 Synthesis Results

The proposed ASIP was modeled in LISA language using Synopsys Processor Designer tool. Logic synthesis using $90nm$ CMOS technology results in an area of $0.155mm^2$ per ASIP with maximum clock frequency of $F_{clk} = 520MHz$. The de-Bruijn NoC of size 8 has an area of $0.16mm^2$. Thus, the proposed LDPC/Turbo decoder architecture with 8 ASIPs and interconnecting de-Bruijn network is $1.4mm^2$ with total memory area of $1.2\ mm^2$. The acheived throughput for LDPC mode is given by the expression (5.11). The best throughput achieved is $312Mbps$ for WiMAX code rate $(Crate) = 5/6,\ Z = 96,\ M_b = 4,\ N_b = 24$ and $N_{iter} = 10$ iterations. The architecture has $N_A = 8$ ASIPs each processing $CN_A = 3$ check nodes per $Clk_{CN} = 2$ clocks. Using 5.10 the throughput is given as:

$$\text{Throughput} = \frac{Z * N_b * Crate * F_{clk}}{\text{latency per sub-iteration in time steps} * Clk_{CN} * M_b * N_{iter}} \tag{5.11}$$

Similarly, (5.12) gives the expression of the achieved throughput for Turbo mode. An average $N_{instr} = 4$ instructions are needed to process 1 symbol which is composed of $Bits_{sym} = 2$ bits. Considering $N_{iter} = 6$ iterations, the maximum throughput achieved is $173Mbps$.

$$\frac{Bits_{sym} * F_{clk} * (N_A/2)}{N_{instr} * N_{iter}} \tag{5.12}$$

Table 5.5 compares the obtained results with other related works. The achieved throughput is comparable to [63] in LDPC WiFi mode while the proposed multi-ASIP architecture achieves $75Mbps$ more in LDPC WiMAX mode. In Turbo mode, the acheived throughput is more than 5 times that achieved by [63] at the cost of twice the occupied area (after technology normalization). The architecture presented in [64] occupies 28% more area compared to ours and does not achieve the throughput requirement of LTE. On the other hand, the architecture proposed in [61] achieves higher throughput in LDPC and SBTC modes at the cost of 20% more area and does not support DBTC.

5.8 Summary

In order to meet flexibility and performance constraints of current and future digital communication applications, multiple ASIPs combined with dedicated communication and memory

	Core area mm^2	nm	Throughput in Mbps				F_{clk} in MHz
			LDPC WiMAX	LDPC WiFi	DBTC WiMAX, DVB-RCS	SBTC (LTE)	
Proposed 8-ASIPs	2.6	90	312	263	173@6iter	173 @6iter	520
[61]	3.2	90	600	600	-	450 @6iter	500
[62]	-	90	70	70	54	14	-
[63]	0.62	65	27.7-237.8	34.5-257	18.6-37.2 @5iter	18.6 @5iter	400
[64]	0.9	45	70	100	70	18	150

Table 5.5: Results comparison of the LDPC/Turbo proposed architecture

architectures are required. In this chapter we consider the design of an innovative universal channel decoder architecture model by unifying flexibility-oriented and optimization-oriented approaches. Toward this objective, we have proposed and designed a flexible and scalable multiprocessor platform based on a novel ASIP architecture for high throughput Turbo/LDPC decoding. A new scheduling for LDPC decoding has been proposed to enable scalability in multiprocessing and efficient partial parallelism. Major gains in area were obtained by avoiding the need to have interleaving memories for shuffled Turbo decoding schedule and efficient memory and communication resource sharing between Turbo and LDPC modes. The proposed platform supports Turbo and LDPC codes of most emerging wireless communication standards (WiFi, WiMAX, LTE, and DVB-RCS). This contribution has been developed jointly with another PhD student at the Electronics department of Telecom Bretagne: Purushotham Murugappa. In his thesis, between other contributions, complete FPGA prototyping and ASIC integration of this Turbo/LDPC multi-standard decoder have been developed.

Conclusions and Perspectives

IN this work, we have investigated the possibility to unify flexibility-oriented and optimization-oriented approaches in the design of multi-standard channel decoders. ASIP architecture model has been considered and an optimized ASIP-based Turbo decoder achieving high architecture efficiency in terms of bit/cycle/iteration/mm^2 has been proposed and designed. Obtained results demonstrate how the architecture efficiency of instruction-set based processors can be considerably improved by maximizing the hardware resources utilization and minimizing the pipeline idle time for the different supported applications/modes. The proposed optimizations further improve the reconfiguration speed and the power consumption considerably. A complete FPGA prototype for the proposed multi-standard Turbo decoder was designed. Finally, a scalable and flexible high throughput multi-ASIP combined architecture for LDPC and Turbo decoding was proposed.

In the presented manuscript we firstly presented an overview of the fundamental concept of Turbo coding and decoding algorithms. A brief introduction has been given on convolutional Turbo codes along with the different interleaving rules specified in emerging wireless communications standards. Then, the reference MAP algorithm and the hardware-efficient Max-Log-MAP approximation were presented. A classification of available parallelism techniques related to these algorithms is provided with emphasis on the Radix-4 technique which allows for promising computation commonalities between SBTC and DBTC modes. Finally, simulation results of error rate performance for SBTC and Radix-4 are presented illustrating the impact of the scaling factor applied on exchanged extrinsic information.

The concept of ASIP-based design and the associated design methodology and tool which are considered in this thesis work have been then presented. This methodology and the target Turbo decoding application have been illustrated through the presentation of an initial ASIP architecture which has been developed in a previous thesis study at the Electronic department of Telecom Bretagne. In this initial architecture, the main target was to validate the effectiveness of the newly proposed ASIP-design tools in terms of generated HDL code and flexibility limitations. To that end, the target flexibility was set very high to investigate the support of any convolutional code trellis. This target flexibility has led to a reduced architecture efficiency, which was in all cases not the main target of that initial effort on the topic.

The first proposed architecture consists of an optimized ASIP-based flexible Turbo decoder supporting all communication modes of 3GPP-LTE, WiMAX and DVB-RCS standards. This main contribution allows further to illustrate how the architecture efficiency of instruction-set based processors can be considerably improved by minimizing the pipeline idle time. Three levels of optimization techniques (Architecture, Algorithmic, and Memory) has been proposed and integrated. The proposed ASIP integrates a low complexity interleaver design supporting QPP and ARP interleaving in butterfly scheme together with an efficient parallel memory access management. Results show that the ASIP pipeline usage percentage have been maximized to reach 94%, achieving a throughput of 164Mbps with 0.594mm^2 @65nm CMOS technology and an architecture efficiency of 3.31 bit/cycle/iteration/mm^2. Furthermore, the impact of the proposed optimization techniques on power consumption is illustrated. The overall gain in normalized en-

ergy efficiency is around ≈45.6% compared to the initial architecture. The achieved architecture efficiency and energy efficiency, considering the supported flexibility, compare favorably with related state of the art implementations

The proposed optimized multi-standard 2-ASIP Turbo decoder has been in addition validated by means of a complete flexible FPGA prototype using a logic emulation board (DN9000K10PCI) integrating Xilinx Virtex 5 devices. The proposed prototype consists of a complete flexible environment integration a GUI, a multi-standard flexible transmitter, a hardware channel emulator and the proposed multi-standard Turbo decoder. It consists of one of the first available and fully functional FPGA prototypes using the ASIP approach and efficiently exploiting the available parallelism levels in multi-standard Turbo decoding. Furthermore, the proposed environment integrates an efficient and high-speed error rate performance evaluation allowing for on-the-fly BER/FER curves display on the host computer. This flexible and high-throughput FPGA prototype can be further used as a fast simulation environment allowing fast performance evaluation of various combinations of supported parameters.

Beyond the support of multi-standard Turbo decoding, a flexible Turbo/LDPC channel decoder presents a real added value in the seek to support the large variety of channel coding techniques specified in emerging wireless communication standards. Toward this objective, we have proposed and designed a flexible and scalable multiprocessor platform based on a novel ASIP architecture for high throughput Turbo/LDPC decoding. A new scheduling for LDPC decoding has been proposed to enable scalability in multiprocessing and efficient partial parallelism. Major gains in area were obtained by avoiding the need to have interleaving memories for shuffled Turbo decoding schedule and efficient memory and communication resource sharing between Turbo and LDPC modes. The proposed platform supports Turbo and LDPC codes of most emerging wireless communication standards (WiFi, WiMAX, LTE, and DVB-RCS). This contribution has been developed jointly with another PhD student at the Electronics department of Telecom Bretagne: Purushotham Murugappa. In his thesis, between other contributions, complete FPGA prototyping and ASIC integration of this Turbo/LDPC multi-standard decoder have been developed.

Perspectives

Regarding work perspectives, several ideas can be investigated:

- Increasing the flexibility to support other interleaving rules (such as that of HSPA+ for SBTC), other standards (such as DVB-S2/T2 for LDPC), and other coding techniques (such as non-binary LDPC codes).
- Exploring other architecture-level technique such as the use of dynamic reconfiguration concept associated with the ASIP design approach. Having a dynamically reconfigurable fabric attached to the ASIP pipeline can enable better resource sharing and usage over different algorithm variants and parameters.
- Exploring other emerging target technologies such as the 3D integration which can enable further improvements in energy, reconfiguration speed, and architecture efficiency.

Résumé en Français

Les systèmes sur puces dans le domaine des communications numériques deviennent extrêmement diversifiés et complexes avec la constante émergence de nouveaux standards et de nouvelles applications. Dans ce domaine, le turbo-décodeur est l'un des composants les plus exigeants en termes de calcul, de communication et de mémoire, donc de consommation d'énergie.

En effet, les exigences croissantes imposées en termes de débit, de fiabilité des transmissions et de consommation d'énergie sont les principales raisons qui font que les réalisations ASIC (*Application-Specific Integrated Circuit*) dans le domaine des communications numériques deviennent de plus en plus diversifiées et complexes. Le décodage de canal a été largement considéré au cours des dernières années et plusieurs implémentations ont été proposées. Certaines de ces implémentations ont réussi à atteindre un débit élevé pour des standards ou paramètres spécifiques grâce à des architectures entièrement dédiées. Cependant, ces implémentations ne considèrent pas les problèmes liés aux besoins de flexibilité et d'extensibilité. En effet, une telle approche implique l'intégration de plusieurs accélérateurs matériels dédiés pour réaliser des systèmes multi-standards, qui correspond souvent à une mauvaise efficacité matérielle. De plus, cette approche implique des temps de conception longs qui ne sont guère compatibles avec les contraintes de temps de mise sur le marché et le développement continu de nouvelles normes et applications.

Outre les exigences de performances croissantes, les nouveaux systèmes de communications numériques imposent une interopérabilité multi-standard avec le support de différentes qualités de service et diverses techniques obligatoires et/ou optionnelles à l'intérieur de chaque standard. Cela est particulièrement vrai pour le codage de canal où une grande variété de techniques de correction d'erreurs sont proposées aujourd'hui. À titre d'exemple, la norme IEEE 802.16e (WiMAX) spécifie l'adoption de quatre codes correcteurs d'erreurs (codes convolutifs, turbocodes, codes LDPC, et codes en blocs) et chaque code est associé à de multiple paramètres en termes de rendements de codage et de longueurs de trame. Ainsi, la nouvelle exigence de flexibilité de l'implémentation du décodeur de canal devient particulièrement cruciale.

Problèmes et objectifs de la thèse

Dans ce contexte, de nombreux travaux ont été proposés récemment visant des implémentations de décodeurs de canal flexibles. La flexibilité varie du support de plusieurs modes d'un même standard de communication au support d'applications multi-standards. D'autres implémentations ont même visé une flexibilité plus large pour supporter différentes techniques de codage de canal.

En ce qui concerne le modèle d'architecture, outre le modèle classique d'architecture paramétrée, des efforts récents ont exploré l'utilisation de modèles de processeurs à jeu d'instructions dédié à l'application (*ASIP : Application-Specific Instruction-set Processor*). Un tel modèle d'architecture permet au concepteur d'affiner librement le compromis de flexibilité/performance tel que requis par les exigences des applications envisagées. De nombreuses contributions émergent actuellement dans ce contexte, cherchant à améliorer l'efficacité architecturale en termes de performance/surface en plus d'accroître la flexibilité supportée. Cependant, l'efficacité architecturale des processeurs dédiés à l'application est directement liée au jeu d'instructions défini ainsi qu'au taux d'utilisation des étages de pipeline. La plupart des travaux proposés récemment ne considèrent pas ces aspects explicitement.

En effet, des solutions optimales en termes de performance, de consommation d'énergie et de surface sont encore à inventer et ne doivent pas être négligées au profit de la flexibilité. Communément, une approche "aveugle" de la flexibilité conduit à des pertes en optimalité.

Par conséquent, ce travail de thèse s'inscrit dans l'objectif principal d'unifier l'approche orientée sur la flexibilité et celle orientée sur l'optimalité dans la conception de décodeurs de canal. En considérant principalement l'application de turbo-décodage, l'objectif est de montrer comment l'efficacité architecturale des processeurs à jeu d'instructions dédié à l'application peut être considérablement améliorée. Un autre objectif de ce travail est d'étudier la possibilité d'utiliser l'approche de conception basée sur le concept ASIP pour accroître la flexibilité du turbo-décodeur pour supporter le décodage des codes LDPC.

Contributions

Pour atteindre les objectifs cités ci-dessus, plusieurs contributions ont été proposées dans le cadre de ce travail de thèse :

Conception d'un turbo-décodeur multi-standard basé sur le concept ASIP :

- Proposition et conception d'un turbo-décodeur multi-standard basé sur le concept ASIP assurant une efficacité architecturale élevée en bit/cycle/iteration/mm^2.

- Optimisation de la vitesse de reconfiguration dynamique de l'ASIP proposé supportant tous les paramètres spécifiés dans les normes 3GPP-LTE/WiMAX/DVB-RCS.

- Conception d'entrelaceurs ARP (*Almost Regular Permutation*) et QPP (*Quadric Polynomial Permutation*) de faible complexité pour le schéma de décodage de type papillon avec la technique de compression de treillis de type Radix-4.
- Proposition et mise en œuvre d'un prototype FPGA de système de communication complet intégrant le turbo-décodeur multi-standard proposé.
- Analyse de l'impact des optimisations des étages de pipeline de l'ASIP proposé sur la consommation énergétique.

Conception d'une architecture multi-ASIP flexible et extensible supportant le décodage des turbocodes et des codes LDPC :

- Proposition d'un partage efficace des bancs de mémoires pour le décodage flexible LDPC/-Turbocodes.
- Proposition et conception d'une architecture d'ASIP supportant le décodage des turbocodes et des codes LDPC.

Structure du manuscrit

Le manuscrit de thèse est organisé en cinq chapitres, il détaille les majeures contributions citées ci-dessus.

Chapitre 1

Le **premier chapitre** présente les concepts de base liés aux turbocodes convolutifs et à leurs algorithmes de décodage. Les notions de base sont présentées concernant la capacité d'un canal de transmission bruité, les opérations de codage mises en œuvre pour assurer une réception sans erreurs de données émises, ainsi que les nombreux paramètres spécifiés dans les standards de communication sans fil (Table 1). Le codage convolutif et les codes convolutifs récursifs systématiques sont introduits, ces derniers étant éléments constitutifs des classes de turbocodes considérés dans le reste du manuscrit. Les différentes règles d'entrelacement spécifiées dans les normes de communication sans fil émergents sont aussi introduites. Ensuite, l'algorithme de référence pour le turbo-décodage, MAP (*Maximum A Posteriori*), et la version simplifiée pour une implémentation matérielle efficace Max-Log-MAP ont été présentés. Une classification des techniques de parallélisme relatives à cet algorithme est fourni en mettant l'accent sur la technique de compression de treillis (Radix-4) qui permet d'unifier les calculs relatifs au décodage des turbocodes simple-binaires (*SBTC : Single Binary Turbo Codes*) et des turbocodes double-binaires (*DBTC : Double Binary Turbo Codes*). La technique de fenêtrage (*windowing*) est également mise en évidence pour l'efficacité qu'elle apporte pour gérer le décodage de trames de longue taille. Enfin, un exemple de résultats de simulation de performance de taux d'erreurs

binaires est présenté (mode SBTC avec Radix-4) pour illustrer l'effet du facteur d'échelle appliqué sur l'information extrinsèque échangée dans le processus itératif de turbo-décodage.

Standard	Codes	Rendement	Nb. d'états	Taille de trame	Débit
IEEE-802.11	CC	1/2 - 3/4	64	1 -4095	6 - 54 Mbps
(WiFi)	CC	2/3	256	.. 1944	.. 450Mbps
IEEE802.16	CC	1/2 - 7/8	64	.. 2040	.. 54 Mbps
(WiMAX)	DBTC	1/2 - 3/4	8	.. 4800	.. 75 Mbps
DVB-RCS	DBTC	1/3 - 6/7	8	.. 1728	.. 2 Mbps
3GPP-LTE	SBTC	1/3	8	.. 6144	.. 150 Mbps

Table 1: Exemples de standards de communication sans fil : différents paramètres liés aux codes convolutifs et aux turbocodes sont spécifiés

Chapitre 2

Le **deuxième chapitre** introduit la conception de systèmes basés sur le modèle ASIP, en la positionnant comme approche architecturale intermédiaire entre celles à disposition des concepteurs. Les méthodologies et outils de développement sont également brièvement présentés. Le chapitre se termine par la description d'une architecture ASIP développée précédemment au sein du département Électronique de Télécom Bretagne et servant, dans les chapitres suivants, de point de départ aux travaux développés dans le cadre de cette thèse. En effet, dans cette architecture initiale l'objectif principal était de valider l'efficacité des outils récemment proposés pour la conception d'ASIP en termes de code HDL généré et de contraintes sur la flexibilité atteignable par cette approche. À cette fin, un degré très élevé de flexibilité a été considéré pour évaluer le support de tout type de treillis de code convolutif. Ce degré de flexibilité a conduit à une efficacité architecturale réduite, qui n'était dans tous les cas pas l'objectif principal de cet effort initial sur le sujet. En outre, cette architecture initiale d'ASIP pour le turbo-décodage manquait plusieurs fonctionnalités qui sont mises en évidence, et des solutions adéquates sont proposées dans le troisième chapitre de ce manuscrit de thèse.

Chapitre 3

Le **troisième chapitre** présente la majeure contribution de cette thèse concernant la proposition, sur la base d'une approche ASIP, d'une architecture optimisée pour le turbo-décodage capable de concilier les objectifs évoqués précédemment. Le chapitre illustre comment l'application de techniques d'optimisation adéquates à trois niveaux (algorithme, architecture et mémoire) sur un ASIP pour un turbo-décodage multi-standard peut le rendre une solution encore attrayante et efficace en termes de surface, de débit et de consommation énergétique. L'architecture proposée intègre 2 TurbASIP supportant le décodage des turbocodes simple- et double-binaires et combinant plusieurs techniques d'optimisation concernant la structure de pipeline, la compression de treillis (Radix-4) et l'organisation de la mémoire. La Figure 1 donne un aperçu de l'architecture proposée.

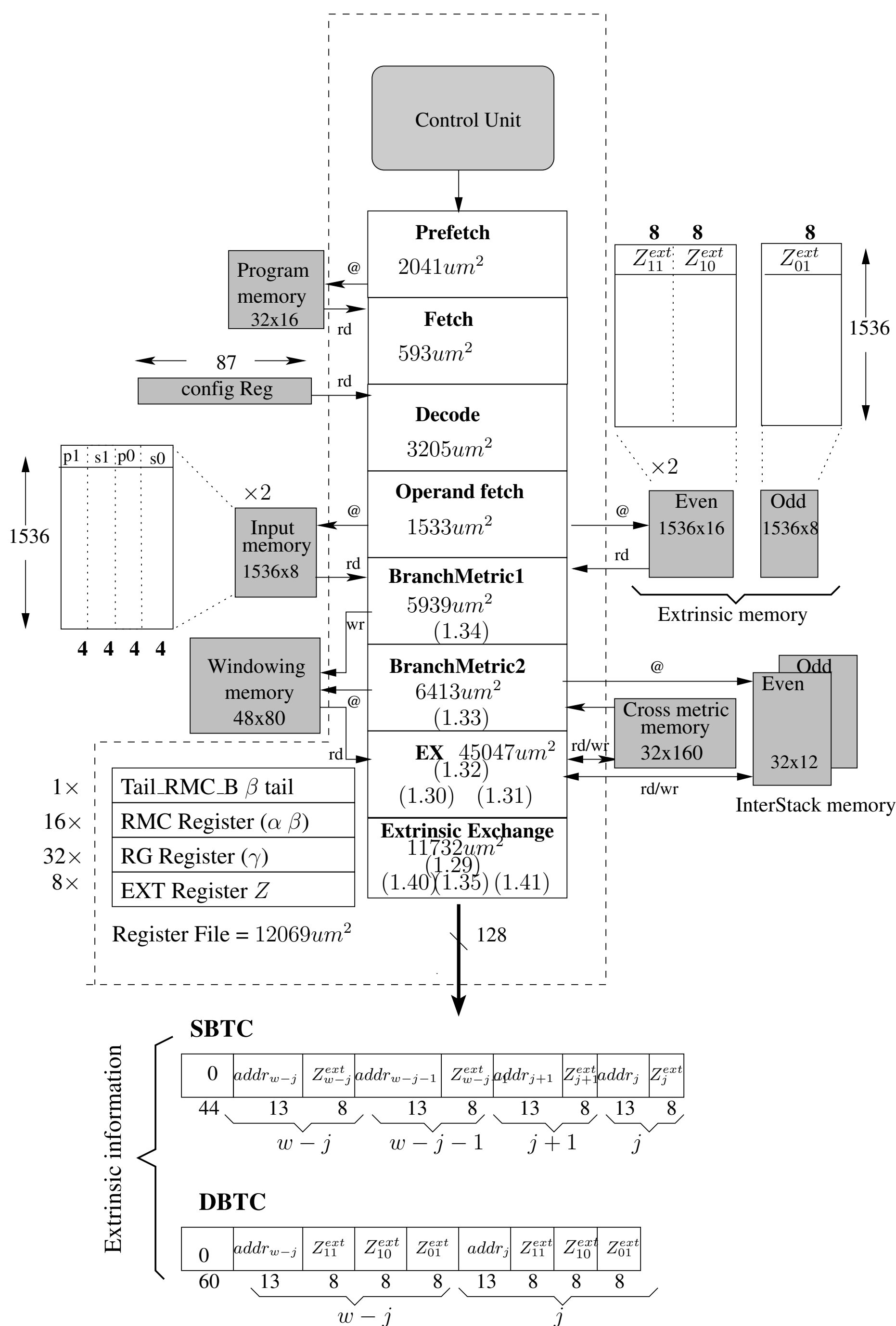

Figure 1: Aperçu de l'architecture proposée de l'ASIP de turbo-décodage multi-standard

Le chapitre commence par un bref état de l'art pour résumer les réalisations matérielles disponibles relatives à ce domaine. Ensuite, les techniques d'optimisation proposées concernant la structure de pipeline, la compression du treillis (Radix-4) et l'organisation de la mémoire sont détaillées. Le chapitre présente aussi la conception d'entrelaceurs ARP et QPP de faible complexité pour le schéma de décodage de type papillon avec une gestion efficace des accès parallèles aux bancs mémoires. En outre, afin d'améliorer la vitesse de reconfiguration de l'architecture proposée, une nouvelle organisation du programme d'instruction est présentée.

Ensuite, la performance obtenue du système de turbo-décodeur multi-standard est évaluée et l'efficacité architecturale est comparée avec les travaux correspondants dans l'état de l'art. Enfin, l'impact de l'optimisation proposée du pipeline sur l'efficacité énergétique est analysé et discuté. Cette dernière contribution constitue un effort conjoint avec une doctorante du CEA-LETI : Pallavi Reddy. Pour étudier cet impact, une analyse détaillée de la consommation d'énergie de l'ASIP pour une cible CMOS 65nm a été effectuée avant et après l'application des techniques d'optimisations proposées. Les résultats montrent un gain important en efficacité énergétique normalisée de l'ordre de $\approx$45.6% par rapport à l'architecture initiale.

Les résultats de la synthèse logique sur une cible technologique CMOS 65nm donnent une surface totale de 0.594 mm^2. Le débit de décodage, pour une fréquence de fonctionnement de 500 MHz, atteint 164 Mbps dans les deux modes SBTC (3GPP-LTE) et DBTC (WiMAX et DVB-RCS). Les résultats montrent que le taux d'utilisation des étages de pipeline de l'ASIP proposé a été maximisé pour atteindre 94% avec une efficacité architecturale de 3.31 bit/cycle/iteration/mm^2. Considérant les performances obtenues en termes d'efficacité architecturale, d'efficacité énergétique, et de degré de flexibilité, l'architecture proposée se compare favorablement par rapport aux implémentations existantes dans ce domaine (Table 2).

	TurbASIP$_{v3}$	[43]	[44]	[42]	[45]	[46]	[47]
Standards supportés	WiMAX, DVB-RCS, LTE	LTE, HSPA+	WiMAX, LTE	3GPP, LTE	WiMax, LTE	LTE	LTE
Tech. (nm)	65	45	130	65	90	65	65
Surface (mm^2)	0.594	1.34	10.7	0.5	3.38	2.1	8.3
Surface normalisée @65nm (mm^2)	0.594	2.8	2.675	0.5	1.75	2.1	8.3
Débit (Mbps)	164 @6iter	150 @8itr	187 @8iter	21 @6iter	186 @6iter	150 @6.5iter	1280 @6iter
Nb. de MAP	2	8	8	1	8	1	64
F_{clk} (MHz)	500	385	250	300	152	300	400
Efficacité architecturale*	3.31	1.11	2.24	0.84	4.2	1.56	2.31

* en bit/cycle/iteration/mm^2

Table 2: Résultats de synthèse de la dernière version du TurbASIP proposée et comparaison avec les implémentations existantes

Chapitre 4

Le **quatrième chapitre** décrit l'environnement de prototypage basé sur une plateforme FPGA, qui a été développé afin de tester et de valider les concepts et l'architecture proposés au chapitre précédent. En effet, la validation sur plateforme matérielle est une étape cruciale pour démontrer la faisabilité et l'efficacité de toute nouvelle architecture matérielle proposée. Deux avantages supplémentaires peuvent être mentionnés dans ce contexte: (a) retours précieux pour l'étape de conception d'architecture, en particulier en ce qui concerne les aspects liés à l'interface au niveau système de turbo-décodeur et (b) le prototype de matériel obtenu peut être utilisé comme un environnement de simulation rapide pour des applications de communications numériques,

par exemple pour explorer diverses associations de paramètres du système.

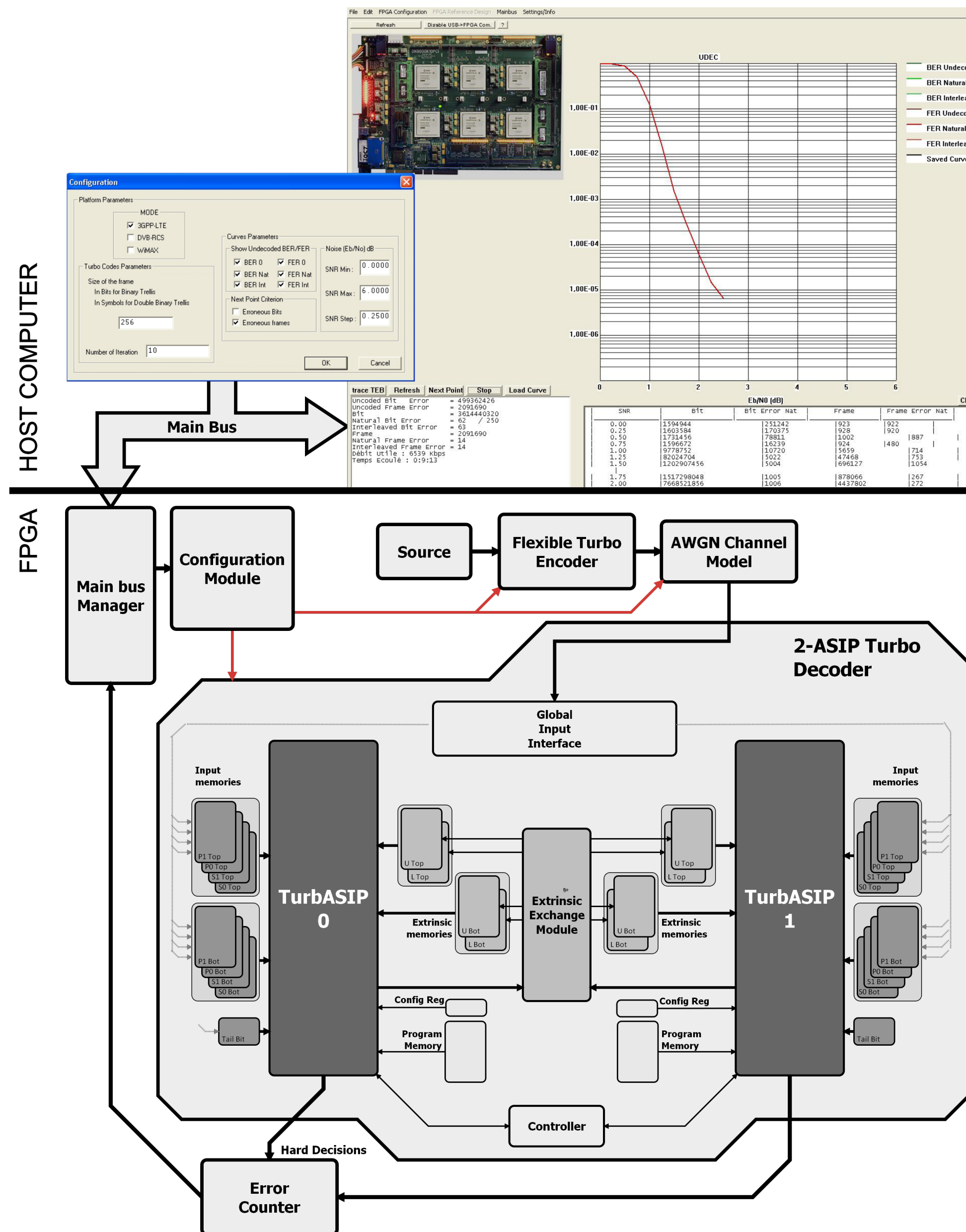

Figure 2: Aperçu de l'environnement de prototypage FPGA du turbo-décodeur multi-standard proposé

Cependant, le prototypage matériel constitue une tâche complexe surtout dans le contexte d'implémentations flexibles de décodeurs de canal puisqu'un environnement totalement flexible devrait être conçu. Ainsi, ce chapitre est consacré à la présentation du prototype FPGA du turbo-décodeur multi-standard proposé. La première partie du chapitre présente le flot de conception adopté de processeurs dédiés à l'application de type ASIP. Le flot complet, à partir de la description en langage LISA (*Language for Instruction Set Architecture*) jusqu'au prototypage FPGA, est illustré. La deuxième partie du chapitre détaille l'environnement de prototypage proposé, basé la plateforme DN9000K10PCI de chez DiniGroup et qui intègre 6 circuits FPGA de type Xilinx Virtex 5 LX330. Cet environnement, illustré dans la Figure 2, inclus les éléments

suivants :

- Une interface graphique (*GUI : Graphical User Interface*) qui s'exécute sur un ordinateur hôte pour configurer la plateforme avec les paramètres désirés en termes de standard cible, de taille de trame, de rendement de codage et de nombre d'itérations. Cette interface permet aussi de tracer, en temps réel, les courbes de taux d'erreurs binaires et d'afficher l'évaluation du débit de décodage atteint.
- Une implémentation matérielle d'une chaine de transmission flexible multi-standard.
- Un émulateur matériel de canal à bruit Gaussien additif (*AWGN : Additif White Gaussian Noise*).
- Le turbo-décodeur multi-standard proposé (décrit dans le troisième chapitre).
- Un module matériel pour évaluer les taux d'erreurs binaires.
- Un contrôleur global de l'ordonnancement des traitements sur la plateforme de communication.

La troisième et dernière partie du chapitre présente et compare les résultats de synthèse obtenus pour cible Xilinx FPGA Virtex 5 LX330.

Chapitre 5

Le **cinquième chapitre** est consacré à la présentation de la dernière contribution de cette thèse concernant l'augmentation de la flexibilité du turbo-décodeur afin de pouvoir également décoder une autre famille de codes correcteurs d'erreurs, celles des codes LDPC. Il s'agit d'une contribution conjointe avec un autre doctorant au département Electronique de Télécom Bretagne: M. Purushotham Murugappa.

En effet, outre les turbocodes, les normes de communication sans fil numériques spécifient une grande variété de techniques de codage de canal associées à de nombreux paramètres. Certains d'entre eux sont obligatoires, d'autres optionnels, où chacun est adapté à un mode de communication spécifique. Dans ce contexte, les codes LDPC sont souvent proposés dans ces normes (par exemple dans les standards WiMAX, WiFi, DVB-S2/T2) en raison de leur haute performance de correction d'erreur et de la simplicité de leur principe de décodage itératif. Principalement, la classe de codes LDPC structurés quasi-cycliques (QC-LDPC) est adoptée comme elle présente des propriétés de mise en œuvre très intéressantes en termes de parallélisme, d'interconnexion, de mémoire et d'extensibilité.

Le résultat principal obtenu dans cette thèse dans le cadre de cette activité concerne la proposition et la conception d'une architecture multi-ASIP flexible et extensible, supportant le décodage des turbocodes (SBTC et DBTC) ainsi que le décodage des codes QC-LDPC. L'architecture proposée, qui supporte les paramètres spécifiés dans les standards LTE, WiMAX,

WiFi et DVB-RCS, réalise un bon compromis entre surface et débit de décodage grâce à un partage efficace des mémoires et des réseaux de communications utilisés. Le chapitre commence par une brève introduction sur les codes LDPC en mettant l'accent sur la catégorie de codes QC-LDPC et les paramètres associés dans les normes sans fil considérées. L'algorithme de référence pour le décodage des codes LDPC et la version simplifiée pour une implémentation matérielle efficace (*Normalized Min-Sum algorithm*) sont ensuite présentés. Puis, la démarche conceptuelle est introduite par une approche descendante, en partant des objectifs fonctionnels (support du décodage QC-LDPC et turbo-codes), avec une argumentation des différents choix architecturaux, d'organisation de la mémoire, d'ordonnancement des opérations et de la structuration du pipeline. Le chapitre se termine par la présentation et discussion des résultats de synthèse logique.

Conclusions et perspectives

Dans ce travail de thèse nous avons exploré la possibilité d'unifier l'approche orientée sur la flexibilité et celle orientée sur l'optimalité dans la conception de décodeurs de canal multi-standards. Un modèle d'architecture basé sur le concept ASIP a été considéré pour la conception d'un turbo-décodeur multi-standard assurant une efficacité architecturale élevée en bit/cycle/iteration/mm^2. Les résultats obtenus montrent comment l'efficacité architecturale des processeurs à jeu d'instructions dédié à l'application peut être considérablement améliorée en optimisant l'utilisation des ressources matérielles et en minimisant le temps d'inactivité des étages du pipeline pour les différents modes de décodage supportés. De plus, les optimisations proposées améliorent considérablement la vitesse de reconfiguration et la consommation d'énergie. Un prototype FPGA de système de communication complet intégrant le turbo-décodeur multi-standard proposé a été mise en œuvre. Enfin, une première contribution a été proposée vers la conception d'une architecture multi-ASIP flexible et extensible supportant le décodage des turbocodes et des codes LDPC.

En ce qui concerne les perspectives de travail, plusieurs idées peuvent être étudiées :

- Augmentation de la flexibilité pour supporter d'autres règles d'entrelacement (comme celle du standard HSPA+), d'autres normes (comme DVB-S2/T2) et d'autres techniques de codage correcteur d'erreurs (tels que les codes LDPC non-binaires).
- Exploration d'autres possibilités d'optimisations au niveau du modèle d'architecture comme l'utilisation du concept de reconfiguration dynamique associée à l'approche ASIP. Avoir une structure reconfigurable dynamiquement attaché au pipeline de l'ASIP peut permettre un meilleur partage des ressources pour les différentes variantes algorithmiques et les divers paramètres à supporter.
- Exploration d'autres technologies cibles émergentes telles que l'intégration 3D qui peut permettre de nouvelles améliorations en termes de consommation d'énergie, de vitesse de reconfiguration et d'efficacité architecturale.

Glossary

3GPP	3rd Generation Partnership Project
ACS	Addition Comparaison Selection
ADL	Architectural Description Language
AE	Area Efficiency
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction-set Processor
ARP	Almost Regular Permutation
AWGN	Additive White Gaussian Noise
BCJR	Bahl-Cock-Jelinek-Raviv
BER	Bit Error Rate
BP	Belief propagation
BPSK	Binary Phase Shift Keying
CAD	Computer Aided Design
CC	Convolutional Codes
CN	Check Node
CNG	Check Node Group
CMOS	Complementary Metal Oxide Semi-Conductor
CRSC	Circular Recursive Systematic Convolutional
DBTC	Double Binary Turbo Codes
DVB-RCS	Digital Video Broadcasting Return Channel Satellite
DVB-T	Digital Video Broadcasting Terrestrial
DSP	Digital Signal Processor
ECC	Error Control Coding
EE	Energy Efficiency
EU	Euclidean Unit
FER	Frame Error Rate
FEC	Forward Error Correction
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FS	Flooding Schedule
FSK	Frequency Shift Keying

FSM	Finite State Machine
GSM	Global System for Mobile Communications
GPS	Global Positioning System
GUI	Graphical User Interface
HDL	Hardware Description Language
HSS	Horizontal Shuffle Scheduling
ISS	Instruction Set Simulator
LDPC	Low-Density Parity-Check
LLR	Log Likelihood Ratio
LTE	Log Term Evolution
LUT	Look Up Table
MAP	Maximum A Posteriori
MIDF	Memory Interface Definition Files
NI	Network Interface
NEE	Normalized Energy Efficiency
NEF	Normalized Energy Factor
NMS	Normalized Min-Sum algorithm
NoC	Network on Chip
PCCC	Parallel Concatenated Convolutional Codes
PSK	Phase Shift Keying
QC	Quasi-Cycle
QPP	Quadratic Permutation Polynomial
RAM	Random Access Memory
RTL	Register Transfer Level
SBTC	Single Binary Turbo Codes
SCCC	Serial Concatenated Convolutional Codes
SIMD	Single Instruction Multiple Data
SISO	Soft In Soft Out
SNR	Signal to Noise Ratio
SoC	System on Chip
SOVA	Soft Output Viterbi Algorithm
TPMP	Two-Phase Message Passing
UMTS	Universal Mobile Telecommunications System
USB	Universal Serial Bus
VCD	Value Change Dump

VHDL	VHSIC hardware description language
VLIW	Very Long Instruction Word
VN	Variable Node
VNG	Variable Node Group
VSS	Vertical Shuffle Scheduling
WiMAX	Worldwide Interoperability for Microwave Access
WLAN	Wireless Local Area Network
ZOL	Zero Overhead Loop

Notations

Channel Coding:

X_i	Coded symbol of index i
Y_i	Modulated symbol of index i
E_b	Energy per information bit
N_0	Real power spectrum density of the noise
R	Code rate
σ	Gaussian noise noise variance
$p(Y_i\|X_i)$	The channel transition probability

Turbo Decoding:

$\Pi(i)$	Interleaved order of index i
N	Frame size in symbols (1 symbol = 1 bit in SBTC and 1 symbol = 2 bits in DBTC)
d_i	Information symbol
a	Decoder forward recursion metrics in log-MAP algorithm
b	Decoder backward recursion metrics in log-MAP algorithm
c	Decoder branch metrics in log-MAP algorithm
α	Decoder forward recursion metrics in Max-log-MAP algorithm
β	Decoder backward recursion metrics in Max-log-MAP algorithm
γ	Decoder branch metrics in Max-log-MAP algorithm
Z^{ext}	Turbo decoder extrinsic information
Z^{apos}	Turbo decoder *a posteriori* information
$Z^{Hard.dec}$	Turbo decoder hard decision
Γ	Extrinsic scaling factor

LDPC Decoding:

M	Number of check nodes
N	Number of variable nodes
m	Check node m
n	Variable node n
Z	Permutation matrix size
H	Parity check matrix
λ_{nm}	Variable node message
Γ_{mn}	Check node message
λ_n	a posteriori LLR
Δ_n	Channel value for variable node n
$min0_m$	Overall minimum of the received $\|\lambda_{nm}\|$
$min1_m$	Second minimum of the received $\|\lambda_{nm}\|$

ind_m	The index of the connected VN$_m$ providing $min0$
sgn_m	Product of the signs of the received λ_{nm}
$RV(m)$	Running Vector group $[min0, min1, ind, sgn]_m$

Others:

F	Operational frequency
P	Power
T	Turbo decoder throughput
I	Number of iteration
f	technology Feature size
V_{dd}	Supply Voltage
NA	Normalized Area
NEF	Normalized Energy Factor
NEE	Normalized Energy Efficiency
EE	Energy Efficiency
AE	Architecture Efficiency

Bibliography

[1] "3GPP Technical Specification Group,Multiplexing and channel coding (FDD)," Tech. Rep., Avril 1999.

[2] S.Lin and D. Costello, *Error Control Coding*. Englewood Cliffs, NJ:Prentice Hall, 1982.

[3] G. J. Forney, *"Performance of concatenated codes", Key papers in the development of coding theory*, E. Berlekamp, Ed. IEEE Press, 1974.

[4] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes. 1," *In Proc. IEEE International Conference on Communications, ICC'93.*, vol. 2, pp. 1064–1070, 1993.

[5] S. Dolinar and D. Divsalar, "Weight distributions for turbo codes using random and non-random permutations," The Telecommunications and Data Acquisition Report, Tech. Rep. pp. 56-65., 1995.

[6] C. Douillard, M. Jezequel, C. Berrou, J. Tousch, N. Pham, and N. Brengarth, "The Turbo Code Standard for DVB-RCS," in *Proc. of the International Symposium on Turbo Codes and Related Topics, Brest, France*. ELEC - Dépt. Electronique (Institut Télécom-Télécom Bretagne), 2000, pp. 535 –538.

[7] *802.16 IEEE Standard for Local and metropolitan area networks, Part 16: Air Interface for Fixed Broadband Wireless Access Systems*, Std., 2004.

[8] O. Takeshita and J. Daniel, "New deterministic interleaver designs for turbo codes," vol. 46, no. 6, pp. 1988–2006, 2000.

[9] O. Takeshita, "On maximum contention-free interleavers and permutation polynomials over integer rings," *IEEE Transactions on Information Theory*, vol. 52, no. 3, pp. 1249 –1253, Mar. 2006.

[10] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, April 1967.

[11] R. Fano, "A heuristic discussion of probabilistic decoding," *IEEE Transactions on Information Theory*, vol. 9, no. 2, pp. 64–74, 1963.

[12] J. Forney, G.D., "The viterbi algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.

[13] J. Hagenauer and P. Hoeher, “A Viterbi algorithm with soft-decision outputs and its applications,” in *Proc. of the IEEE Global Telecommunications Conf., and Exhibition. Communications Technology for the 1990s and Beyond. (GLOBECOM)*, 1989, pp. 1680–1686.

[14] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, “Optimal decoding of linear codes for minimizing symbol error rate (Corresp.),” *IEEE Transactions on Information Theory*, vol. 20, no. 2, pp. 284–287, 1974.

[15] P. Robertson, E. Villebrun, and P. Hoeher, “A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain,” in *Proc. of the International Conference on Communications, (ICC). Seattle, Gateway to Globalization*, vol. 2, 1995, pp. 1009–1013 vol.2.

[16] C. Douillard and C. Berrou, “Turbo codes with rate-m/(m+1) constituent convolutional codes,” *IEEE Transactions on Communications,*, vol. 53, no. 10, pp. 1630 – 1638, oct. 2005.

[17] O. Muller, A. Baghdadi, and M. Jézéquel, “Exploring Parallel Processing Levels for Convolutional Turbo Decoding, booktitle = ICTTA’06, International Conference on Information and Communication Technologies, from Theory to Applications, Damascus,” vol. 2, 2006, pp. 2353–2358.

[18] G. Masera, G. Piccinini, M. R. Roch, and M. Zamboni, “VLSI architectures for turbo codes,” vol. 7, no. 3, pp. 369–379, 1999.

[19] E. Boutillon, W. J. Gross, and P. G. Gulak, “VLSI architectures for the MAP algorithm,” vol. 51, no. 2, pp. 175–185, 2003.

[20] Y. Zhang and K. Parhi, “High-Throughput Radix-4 logMAP Turbo Decoder Architecture,” in *Proc. of the Fortieth Asilomar Conf. Signals, Systems and Computers (ACSSC)*, 2006, pp. 1711–1715.

[21] ——, “Parallel Turbo decoding,” in *Proc. of the Int. Symp. Circuits and Systems (ISCAS)*, vol. 2, 2004.

[22] H. Moussa, “Architecture de Réseaux sur Puce Pour Décodeur Canal Multiprocesseurs,” Ph.D. dissertation, ELEC - Dept. Electronique, TELECOM Bretagne, 2009.

[23] O. Muller, “Architectures multiprocesseurs monopuces génériques pour turbo-communications haut-débit,” Ph.D. dissertation, ELEC - Dept. Electronique, TELECOM Bretagne, 2007.

[24] J. Zhang and M. Fossorier, “Shuffled iterative decoding,” *IEEE Transactions on Communications*, vol. 53, no. 2, pp. 209 – 213, feb. 2005.

[25] J. Vogt and A. Finger, “Improving the max-log-MAP turbo decoder,” *Electronics Letters*, vol. 36, no. 23, pp. 1937 –1939, nov 2000.

[26] A. R. Jafri, “Architectures multi-ASIP pour turbo récepteur flexible,” Ph.D. dissertation, Dépt. Electronique (Institut Mines-Télécom), Université de Bretagne Sud (UBS), Lab-STICC, Université Européenne de Bretagne (UEB), 2011.

[27] P. Ienne and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications*, ser. Series in Systems on Silicon. Massachusetts: Morgan Kaufmann, jul 2006.

[28] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Panainte, "The molen polymorphic processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363 – 1375, nov. 2004.

[29] X. Chen, A. Minwegen, Y. Hassan, D. Kammler, S. Li, T. Kempf, A. Chattopadhyay, and G. Ascheid, "FLEXDET: Flexible, Efficient Multi-Mode MIMO Detection Using Reconfigurable ASIP," in *Proc. of the IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, may 2012, pp. 69 –76.

[30] K. Karuri, A. Chattopadhyay, X. Chen, D. Kammler, L. Hao, R. Leupers, H. Meyr, and G. Ascheid, "A Design Flow for Architecture Exploration and Implementation of Partially Reconfigurable Processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 10, pp. 1281 –1294, oct. 2008.

[31] A. La Rosa, L. Lavagno, and C. Passerone, "Software development for high-performance, reconfigurable, embedded multimedia systems," *IEEE Design Test of Computers*, vol. 22, no. 1, pp. 28 – 38, jan.-feb. 2005.

[32] D. NOGUET, A. Baghdadi, C. Moy, and G. Masera, "Report on the state for the art on hardware architectures for flexible radio and intensive signal processing, Tech. Rep. 216715 NEWCOM++ DC.1. [Online]. Available: http://www.newcom-project.eu:8080/Plone/public-deliverables/research/DR.C.1_final.pdf

[33] "Coware processor designer homepage." [Online]. Available: http://www.synopsys.com/Systems/BlockDesign/ProcessorDev/Pages/default.aspx

[34] "Target ip designer homepage." [Online]. Available: http://www.retarget.com/products/ipdesigner.php

[35] "Tensilica xtensa 7 homepage." [Online]. Available: http://www.tensilica.com/products/x7_processor_generator.htm

[36] "Arc configurable cores homepage." [Online]. Available: http://www.synopsys.com/IP/ProcessorIP/ARCProcessors/

[37] "Stretch software-configurable processors homepage." [Online]. Available: http://www.stretchinc.com/technology/

[38] A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, O. Wahlen, and H. Meyr, "A methodology for the design of application specific instruction set processors (ASIP) using the machine description language LISA," in *Proc. of the IEEE/ACM Int. Conf. Computer Aided Design (ICCAD)*, 2001, pp. 625–630.

[39] O. Muller, A. Baghdadi, and M. Jezequel, "ASIP-Based Multiprocessor SoC Design for Simple and Double Binary Turbo Decoding," in *Proc. of the ACM/IEEE Design, Automation and Test in Europe* ***DATE'06***, vol. 1, 2006, pp. 1–6.

[40] ——, "From Parallelism Levels to a Multi-ASIP Architecture for Turbo Decoding," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 1, pp. 92–102, 2009.

[41] A. P. Hekstra, "An alternative to metric rescaling in Viterbi decoders," vol. 37, no. 11, pp. 1220–1222, 1989.

[42] C. Brehm, T. Ilnseher, and N. Wehn, "A scalable multi-ASIP architecture for standard compliant trellis decoding," in *Proc. of the International SoC Design Conference (ISOCC)*, nov. 2011, pp. 349–352.

[43] Tensilica, "Multi standard turbo decoder - hardware performance, software flexibility," August 2011.

[44] J.-H. Kim and I.-C. Park, "A Unified Parallel Radix-4 Turbo Decoder for Mobile WiMAX and 3GPP-LTE," *In Proc. IEEE Custom Integrated Circuits Conference, CICC'09.*, pp. 487–490, 2009.

[45] C.-H. Lin, C.-Y. Chen, E.-J. Chang, and A.-Y. Wu, "A 0.16nJ/bit/iteration 3.38mm2 turbo decoder chip for WiMAX/LTE standards," in *Proc. of the International Symposium on Integrated Circuits (ISIC)*, dec. 2011, pp. 168–171.

[46] M. May, T. Ilnseher, N. Wehn, and W. Raab, "A 150 Mbit/s 3GPP LTE Turbo Code Decoder," *In Proc. Design, Automation and Test in Europe Conference & Exhibition, DATE'10*, pp. 1420–1425, 2010.

[47] Y. Sun and J. R. Cavallaro, "Efficient hardware implementation of a highly-parallel 3GPP LTE/LTE-advance turbo decoder," *Integration, the VLSI journal*, vol. 44, no. 4, pp. 305–315, 2011.

[48] C. Cheng-Chi, Wong. Hsie-Chia, "Reconfigurable Turbo Decoder With Parallel Architecture for 3GPP LTE System," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 57, no. 7, pp. 566–570, 2010.

[49] Y. Gao and M. Soleymani, "Triple-binary circular recursive systematic convolutional turbo codes," in *Proc. of the International Symposium on Wireless Personal Multimedia Communications.*, vol. 3, oct. 2002, pp. 951–955 vol.3.

[50] J.-H. Kim and I.-C. Park, "Bit-Level Extrinsic Information Exchange Method for Double-Binary Turbo Codes," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 56, no. 1, pp. 81 –85, jan. 2009.

[51] ——, "Double-Binary Circular Turbo Decoding Based on Border Metric Encoding," *IEEE Transactions on Circuits and Systems II, Express Briefs*, vol. 55, no. 1, pp. 79 –83, jan. 2008.

[52] T. Vogt and N. Wehn, "A Reconfigurable ASIP for Convolutional and Turbo Decoding in an SDR Environment," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 10, pp. 1309 –1320, oct. 2008.

[53] F. Naessens, B. Bougard, S. Bressinck, L. Hollevoet, P. Raghavan, L. Van der Perre, and F. Catthoor, "A unified instruction set programmable architecture for multi-standard advanced forward error correction," in *Proc. of the IEEE Workshop on Signal Processing Systems (SiPS)*, oct. 2008, pp. 31–36.

[54] C.-H. Lin, C.-Y. Chen, T.-H. Tsai, and A.-Y. Wu, "Low-Power Memory-Reduced Traceback MAP Decoding for Double-Binary Convolutional Turbo Decoder," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 56, no. 5, pp. 1005 –1016, may 2009.

[55] R. Gallager, *Low-Density Parity-Check Codes*. Cambridge, MIT Press, 1963.

[56] R. Tanner, "A recursive approach to low complexity codes," *Information Theory, IEEE Transactions on*, vol. 27, no. 5, pp. 533 – 547, sep 1981.

[57] S. Lin, L. Chen, J. Xu, and I. Djurdjevic, "Near shannon limit quasi-cyclic low-density parity-check codes," in *Proc. of the IEEE Global Telecommunications Conference, GLOBECOM '03.*, vol. 4, dec. 2003, pp. 2030 –2035 vol.4.

[58] J. Chen, A. Dholakia, E. Eleftheriou, M. Fossorier, and X.-Y. Hu, "Reduced-Complexity Decoding of LDPC Codes," *IEEE Transactions on Communications*, vol. 53, no. 8, pp. 1288 – 1299, Aug. 2005.

[59] J. Chen, A. Dholakia, E. Eleftheriou, M. Fossorier, and X. Hu, "Reduced-Complexity Decoding of LDPC Codes," vol. 53, no. 7, 2005.

[60] H. Moussa, A. Baghdadi, and M. Jezequel, "Binary de Bruijn interconnection network for a flexible LDPC/turbo decoder," in *Proc. of the IEEE Int. Symp. on Circuits and Systems, ISCAS'08*, 2008, pp. 97–100.

[61] Y. Sun and J. Cavallaro, "A Flexible LDPC/Turbo Decoder Architecture," *Journal of Signal Processing Systems*, pp. 1–16, 2010. [Online]. Available: http://dx.doi.org/10.1007/s11265-010-0477-6

[62] A. Niktash, H. Parizi, A. Kamalizad, and N. Bagherzadeh, "RECFEC: A Reconfigurable FEC Processor for Viterbi, Turbo, Reed-Solomon and LDPC Coding," in *Proc. of the IEEE Wireless Communications and Networking Conference, (WCNC).*, 31 2008-april 3 2008, pp. 605 –610.

[63] M. Alles, T. Vogt, and N. Wehn, "FlexiChaP: A reconfigurable ASIP for convolutional, turbo, and LDPC code decoding," in *Proc. of the International Symposium on Turbo Codes and Related Topics*, sep. 2008, pp. 84–89.

[64] G. Gentile, M. Rovini, and L. Fanucci, "A multi-standard flexible turbo/LDPC decoder via ASIC design," in *Proc. of the International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*, sept. 2010, pp. 294–298.

List of publications

Journals

[1] R. Al-Khayat, A. Baghdadi, M. Jézéquel. ”ASIP Design and Architecture Efficiency for Multi-standard Turbo Decoding”, *under preparation*.

International Conferences

[2] R. Al-Khayat, A. Baghdadi, M. Jézéquel. ”Architecture Efficiency of Application Specific Processor: a 170Mbit/s 0.644mm^2 Multi-standard Turbo Decoder ”. *SoC 2012 : International Symposium on System on Chip*, Tampere, Finland, 11-12 Oct. 2012.

[3] R. Al-Khayat, R. Murugappa, A. Baghdadi, M. Jézéquel. “Area and throughput optimized ASIP for multi-standard turbo decoding”. *RSP 2011 : 22nd International Symposium on Rapid System Prototyping*, Karlsruhe, Germany, 24-27 May 2011.

[4] R. Murugappa, R. Al-Khayat, A. Baghdadi, M. Jézéquel. “A flexible high throughput multi-ASIP architecture for LDPC and turbo decoding”. *DATE 2011 : Conference on Design, Automation and Test in Europe*, Grenoble, France, 14-18 March 2011.

[5] P. Reddy, F. Clermidy, R. Al-Khayat, A. Baghdadi, M. Jézéquel. “Power consumption analysis and energy efficient optimization for turbo decoder implementation”. *SoC 2010 : International Symposium on System on Chip*, Tampere, Finland, 29-30 Sept. 2010.

National Conferences

[6] P. Murugappa, P. Reddy, R. Alkhayat, J-N. Bazin, A. Baghdadi, F. Clermidy, M. Jézéquel “A Flexible Multi-ASIP SoC for Turbo/LDPC Decoder ”. *System In Package (SOC-SIP) 2012 : Colloque national du groupe de recherches System On Chip*, Paris, France, 13-15 june 2012.

[7] P. Reddy, F. Clermidy, R. Al-Khayat, A. Baghdadi, M. Jézéquel. “TurbASIP power consumption analysis and optimization”. *System In Package (SOC-SIP) 2010 : Colloque national du groupe de recherches System On Chip*, Paris, France, 09-11 june 2010.